М.Э.Абрамян

Задачи по технологии MPI (английские и русские формулировки)

Оглавление

If the number of processes is not defined in a task then this number is assumed to be not greater than 16. A zero-rank process in the MPI_COMM_WORLD communicator is called a *master process* throughout all task groups. All other processes are called *slave processes*.

If a task does not specify the maximal size of an input sequence of number then this size should be considered as 20.

1. Processes and their ranks

MPI1Proc1. Input a real number X in each process of the MPI_COMM_WORLD communicator and output its opposite value $-X$. Also output the total number of processes in the *master process* (that is, a rank-zero process). For data input and output use the input-output stream pt. Also output the value $-X$ in the debug section using the Show function, which is also defined in the taskbook.

MPI1Proc2. Input an integer A in each process of the MPI_COMM_WORLD communicator and output doubled value of A. Also output the total number of processes in the *master process* (that is, a rank-zero process). For data input and output use the input-output stream pt. In the main process, duplicate the data output in the debug section by displaying on separate lines the doubled value of A and the total number of processes (use two calls of the ShowLine function, which is defined in the taskbook along with the Show function).

MPI1Proc3. Input a real number X and output its opposite value in the master process. Also output the rank of *slave processes* (which are processes whose rank is greater than 0); the rank of each process should be output in the process of this rank. In addition, duplicate the data output in the debug section by displaying the value of $-X$ with the "-X=" comment and the rank values with the "rank=" comments (use the Show function calls with two parameters).

MPI1Proc4. Input one integer in processes with even rank (inclusive of the main process) and output the doubled value of input number. Do not perform any action in processes with odd rank.

MPI1Proc5. Input one integer in processes with even rank (inclusive of the main process), input one real number in processes with odd rank. Output doubled value of input number in each process.

MPI1Proc6. Input one integer in slave processes with even rank, input one real number in processes with odd rank. Output doubled value of input number in each slave process. Do not perform any action in the master process.

MPI1Proc7. An integer N (> 0) and a sequence of N real numbers are given in each process with even rank (inclusive of the master process). Output the

sum of given numbers in each process. Do not perform any action in processes with odd rank.

MPI1Proc8. An integer N (> 0) and a sequence of N real numbers are given in each process. Output the sum of given numbers in each process with even rank (inclusive of the main process), output the average of given numbers in each process with odd rank.

MPI1Proc9. An integer N (> 0) and a sequence of N real numbers are given in each process. Output the sum of given numbers in each slave process with even rank, output the average of given numbers in each process with odd rank, output the product of given numbers in the master process.

MPI1Proc10. An integer N (> 0) and a sequence of N numbers are given in each process. The sequence contains real numbers in the slave processes with odd rank (1, 3, …) and integers in the slave processes with even rank (2, 4 …). The type of elements in the master process depends on the number of processes K: if K is an odd number then the sequence contains integers otherwise the sequence contains real numbers. Output the minimal element of the given sequence in each even-rank process (inclusive of the master process), output the maximal element of the given sequence in each odd-rank process.

2. Point-to-point communication

2.1. Blocking communications

MPI2Send1. An integer is given in each process. Send all given integers to the master process using the MPI_Send and MPI_Recv functions (the blocking functions for standard communication mode) and output received integers in the master process in ascending order of ranks of sending processes.

MPI2Send2. A real number is given in each slave process. Send all given numbers to the master process using the MPI_Bsend (the blocking function for buffered send) and MPI_Recv functions and output received numbers in the master process in descending order of ranks of sending processes. Use the MPI_Buffer_attach function for attaching a buffer to a process.

MPI2Send3. Four integers are given in each slave process. Send all given integers to the master process using one call of the MPI_Send function for each sending process. Output received integers in the master process in ascending order of ranks of sending processes.

MPI2Send4. An integer N ($0 < N < 5$) and a sequence of N integers are given in each slave process. Send all given sequences to the master process using one call of the MPI_Bsend function for each sending process. Output received integers in the master process in ascending order of ranks of sending

processes. Use the MPI_Get_count to determine the size of received sequences.

MPI2Send5. A sequence of real numbers is given in the master process; the size of sequence is equal to the number of slave processes. Send each element of given sequence to corresponding slave process using the MPI_Send function: the first number should be sent to the process 1, the second number should be sent to the process 2, and so on. Output received numbers in the slave processes.

MPI2Send6. A sequence of real numbers is given in the master process; the size of sequence is equal to the number of slave processes. Send each element of given sequence to corresponding slave process (in inverse order) using the MPI_Bsend function: the first number should be sent to the last process, the second number should be sent to the last but one process, and so on. Output received numbers in the slave processes.

MPI2Send7. An integer N and a sequence of N real numbers is given in the master process; $K - 1 \le N < 10$, K is the number of processes. Send elements of given sequence with order number 1, 2, …, $K - 2$ to slave process of rank 1, 2, …, $K - 2$ respectively, and send remaining elements of the sequence to the process $K - 1$. Output received numbers in the slave processes. Use the MPI_Send function to send data, use the MPI_Get_count function to determine the size of received sequences in the process $K - 1$.

MPI2Send8. An integer is given in each slave process; only one of given integers is nonzero-valued. Send nonzero integer to the master process. Output the received number and the rank of sending process in the master process. Use the MPI_Recv function with the MPI_ANY_SOURCE parameter to receive data in the master process.

MPI2Send9. An integer N is given in each slave process; one of given integers is positive, others are zero-valued. Also a sequence of N real numbers is given in the slave process with nonzero integer N. Send the given sequence to the master process. Output the received numbers and the rank of sending process in the master process. Use the MPI_Recv function with the MPI_ANY_SOURCE parameter to receive data in the master process.

MPI2Send10. An integer N is given in each slave process, an integer K (> 0) is given in the master process; the number K is equal to number of slave processes whose given integers N are positive. Send all positive integers N to the master process. Output sum of received numbers in the master process. Use the MPI_Recv function with the MPI_ANY_SOURCE parameter to receive data in the master process.

MPI2Send11. A real number is given in each process. Send the given number from the master process to all slave processes and send the given numbers

from the slave processes to the master process. Output the received numbers in each process. The numbers received by the master process should be output in ascending order of ranks of sending processes. Use the MPI_Ssend function to send data.

Note. The MPI_Ssend function provides a *synchronous data transfer mode*, in which the operation of sending a message will be completed only after the receiving process starts to receive this message. In the case of data transfer in synchronous mode, there is a danger of *deadlocks* because of the incorrect order of the function calls for sending and receiving data.

MPI2Send12. An integer is given in each process. Using the MPI_Ssend and MPI_Recv functions, perform the right cyclic shift of given data by step 1 (that is, the given integer should be sent from the process 0 to the process 1, from the process 1 to the process 2, …, from the last process to the process 0). Output the received number in each process.

Note. See note to MPI2Send11.

MPI2Send13. An integer is given in each process. Using the MPI_Ssend and MPI_Recv functions, perform the left cyclic shift of given data by step −1 (that is, the given integer should be sent from the process 1 to the process 0, from the process 2 to the process 1, …, from the process 0 to the last process). Output the received number in each process.

Note. See note to MPI2Send11.

MPI2Send14. Two integers are given in each process. Send the first integer to the previous process and the second integer to the next process (use the MPI_Ssend and MPI_Recv functions). The last process is assumed to be the previous one for the master process, the master process is assumed to be the next one for the last process. Output the received numbers in each process in the following order: the number received from the previous process, then the number received from the next process.

Note. See note to MPI2Send11.

MPI2Send15. A real number A and an integer N are given in each process; the set of given integers N contains all values in the range 0 to $K-1$, K is the number of processes. Send the number A to the process N in each process (use the MPI_Send and MPI_Recv functions and the MPI_ANY_SOURCE parameter). Output the received number and the rank of sending process in each process.

MPI2Send16. An integer N is given in each process; the value of N is equal to 1 for one process and is equal to 0 for others. Also a sequence of $K-1$ real numbers is given in the process with nonzero integer N; K is the number of processes. Send each number from the given sequence to one of other processes in ascending order of ranks of receiving processes. Output the received number in each process.

MPI2Send17. A sequence of $K-1$ integers is given in each process; K is the number of processes. Send one of the integers from the given sequence in each process to the corresponding process in ascending order of ranks of receiving processes. Output the received numbers in each process in ascending order of ranks of sending processes.

MPI2Send18. The number of processes is an even number. An integer N ($0 < N < 5$) and a sequence of N real numbers are given in each process. Exchange given sequences of processes 0 and 1, 2 and 3, and so on, using the MPI_Sendrecv function. Output the received sequence of real numbers in each process.

MPI2Send19. A real number is given in each process. Change the order of given numbers to inverse one by sending the given numbers from the process 0 to the last process, from the process 1 to the last but one process, …, from the last process to the process 0. Use the MPI_Sendrecv_replace function. Output the received number in each process.

MPI2Send20. A real number A and its order number N (as an integer) are given in each slave process; the set of integers N contains all values in the range 0 to $K-1$, K is the number of processes. Send all numbers A to the master process and output the received numbers in ascending order of their order numbers N. Do not use arrays. Use the order number N as a msgtag parameter of the MPI_Send function.

MPI2Send21. An integer L (≥ 0) and a sequence of L pairs (A, N) are given in each slave process; A is a real number and N is the order number of A. The sum of all integers L is equal to $2K$, where K is the number of processes; the set of integers N contains all values in the range 1 to $2K$. Send all numbers A to the master process and output the received numbers in ascending order of their order numbers N. Do not use arrays. Use the order number N as a msgtag parameter of the MPI_Send function.

MPI2Send22. A sequence of pairs (T, A) is given in the master process; the size of sequence is equal to the number of slave processes. An integer T is equal to 0 or 1; if $T = 0$ then A is an integer, otherwise A is a real number. Send one of the numbers A to the corresponding slave process (the first number to the process 1, the second number to the process 2, and so on) and output received numbers in the slave processes. Use the value of T as a msgtag parameter of the MPI_Send function to send information about the type of number A; use the MPI_Probe function with the parameter MPI_ANY_TAG to receive this information.

Note. To avoid the code duplication, use the auxiliary *template functions* `template <typename T> void send (int t, int dest, MPI_Datatype d)` to send data and `template <typename T> void recv (MPI_Datatype d)` to re-

ceive data. Use a number equal to 0 or 1 for the t parameter and the rank of receiving process for the dest parameter.

MPI2Send23. Two integers *T*, *N* and a sequence of *N* numbers are given in each slave process. An integer *T* is equal to 0 or 1; if *T* = 0 then the given sequence contains integers, otherwise it contains real numbers. Send all given sequences to the master process and output received numbers in the ascending order of ranks of sending processes. Use the value of *T* as a msgtag parameter of the MPI_Send function to send information about the sequence type; use the MPI_Probe function with the parameter MPI_ANY_TAG to receive this information.

Note. To avoid the code duplication, use the auxiliary *template functions* `template <typename T> void send(int t, MPI_Datatype d)` to send data and `template <typename T> void recv(MPI_Datatype d, MPI_Status s)` to receive data. Use a number equal to 0 or 1 for the t parameter and the result returned by the MPI_Probe function for the s parameter.

MPI2Send24. The number of processes *K* is an even number: *K* = 2*N*. A sequence of *N* real numbers is given in each even-rank process (0, 2, …, *K* − 2), a sequence of *N* integers is given in each odd-rank process (1, 3, …, *K* − 1). Using the MPI_Sendrecv_replace function, perform the cyclic shift of all real-valued sequences in the direction of increasing the ranks of processes and the cyclic shift of all integer sequences in the direction of decreasing the ranks of processes (that is, the real-valued sequences should be sent from the process 0 to the process 2, from the process 2 to the process 4, …, from the process *K* − 2 to the process 0 and the integer sequences should be sent from the process *K* − 1 to the process *K* − 3, from the process *K* − 3 to the process *K* − 5, …, from the process 1 to the process *K* − 1). Output received data in each process. To determine the rank of the receiving process, use the expression containing the % operator of taking the remainder after integer division. Use the MPI_ANY_SOURCE parameter as the rank of sending process.

Note. To avoid the code duplication, use the auxiliary *template function* `template <typename T> void sendrecv(int rank, int size, MPI_Datatype d, int step)`. The step parameter specifies a shift value, which should be equal to 2 for real-valued sequences and equal to −2 for integer ones.

MPI2Send25. The number of processes *K* is an even number: *K* = 2*N*. A sequence of *R* + 1 real numbers is given in the first half of the processes, where *R* is the process rank (*R* = 0, 1, …, *N* − 1). A sequence of 2*N* − *R* integers is given in the second half of the processes, where *R* is the process rank (*R* = *N*, *N* + 1, …, 2*N* − 1). Using the MPI_Sendrecv function, send the given seguences from each half of the processes to the corresponding process of the other half (that is, the sequence from the process 0 should be sent to the process *N*, from the process 1 — to the process *N* + 1, from the

process N — to the process 0, from the process $2N-1$ — to the process $N-1$, and so on). Output received data in each process.

Note. To avoid the code duplication, use the auxiliary *template function* `template <typename T1, typename T2> void sendrecv(MPI_Datatype d1, int cnt1, int rank2, MPI_Datatype d2, int cnt2)`, where the d1 and cnt1 parameters define the properties of the process that calls the function (the type and the number of elements of sending sequence) and the parameters rank2, d2, cnt2 are the rank and the similar properties of the process involved in data exchange.

2.2. Nonblocking communications

MPI2Send26. An integer N is given in each process. The value of N is equal to 0 in all processes, except for one, and it is equal to 1 in some selected process. Also an integer sequence A of size $K-1$ is given in the selected process, where K is the number of processes. Do not save the sequence A in array. Send one element of the sequence A at a time to other processes in ascending order of their ranks and output the received number in each process. Use the required number of the MPI_Issend and MPI_Wait function calls (sending a message in synchronous nonblocking mode) in the selected process and the MPI_Recv function call in the other processes. Additionally, display the duration of each MPI_Wait function call (in milliseconds) in the debug section. To do this, call the MPI_Wtime function before and after the MPI_Wait call and use the Show function to display the difference between returned values of the MPI_Wtime function multiplied by 1000. Check how the debugging information changes if the MPI_Isend function (sending a message in standard nonblocking mode) will be used instead of the MPI_Issend function.

MPI2Send27. An integer N is given in each process. The value of N is equal to -1 in some selected process of the rank R and it is equal to R in the other processes. A real number A is also given in all processes, except for the selected one. Send the numbers A to the selected process and output received numbers in ascending order of ranks of sending processes. Use the required number of the MPI_Recv function calls in the selected process and the MPI_Issend and MPI_Test function call in the other processes. Repeat the MPI_Test function call until it returns a nonzero flag, and display the required number of iterations of the loop in the debug section using the Show function. Check how the debugging information changes if the MPI_Isend function will be used instead of the MPI_Issend function.

MPI2Send28. An integer N is given in each process. The value of N is equal to -1 in some selected process of the rank R and it is equal to R in the other processes. A real number A is also given in all processes, except for the selected one. Send the numbers A to the selected process and output received

numbers in descending order of ranks of sending processes. Use the required number of the MPI_Irecv and MPI_Test function calls (receiving a message in nonblocking mode) in the selected process and the MPI_Ssend function call in the other processes. Repeat the MPI_Test function call after each MPI_Irecv function call until MPI_Test returns a nonzero flag, and display the required number of iterations of the loop in the debug section using the Show function. Check how the debugging information changes if the MPI_Send function will be used instead of the MPI_Ssend function.

MPI2Send29. An integer *N* is given in each process. The value of *N* is equal to −1 in some selected process of the rank *R* and it is equal to *R* in the other processes. A real number *A* is also given in all processes, except for the selected one. Send the numbers *A* to the selected process and output the sum *S* of received numbers. Use the required number of the MPI_Irecv and MPI_Waitany function calls in the selected process and the MPI_Ssend function call in the other processes. Declare an array *Q* of the MPI_Request type in the selected process and call the MPI_Irecv functions in a loop with the a separate element of *Q* for each function call. Then call the MPI_Waitany function in a second loop to accumulate the sum *S*. Additionally, display the following data in the debug section in each iteration of the second loop (using the Show and ShowLine function call): the value of *A* added to the sum at this iteration, and the rank of the process that sent this value.

MPI2Send30. An integer *N* is given in each process. The value of *N* is equal to 0 in all processes, except for two, and it is equal to 1 in the first selected process (*the sender*) and it is equal to 2 in the second selected process (*the receiver*). Also an integer *R* and a sequence of *K* integers are given in the sender, where *R* is the rank of the receiver and *K* is the number of processes. Do not save the sequence *A* in array. Send all elements of the sequence *A* to the receiver and output the received numbers in the same order. Use the single call of the MPI_Ssend_init function and the required number of the MPI_Start and MPI_Wait function calls in the sender, and the single call of the MPI_Recv_init function and the required number of the MPI_Start and MPI_Wait function calls in the receiver. Additionally, display the duration of each MPI_Wait function call (in milliseconds) in the debug section (for both the sender and the receiver). To do this, call the MPI_Wtime function before and after the MPI_Wait call and use the Show function to display the difference between returned values of the MPI_Wtime function multiplied by 1000. Check how the debugging information changes if the MPI_Send_init function will be used instead of the MPI_Ssend_init function.

MPI2Send31. An integer *N* is given in each process. The value of *N* is equal to 2 in the first selected process (*the receiver*), it is equal to 1 in some other selected processes (*the senders*), it is equal to 0 in all other processes. Also an integer *R* and a sequence *A* of *K* integers are given in each sender, where *R* is the rank of the receiver and *K* is the number of processes, and the number of senders *C* is given in the receiver. Send all sequences *A* to the receiver and output the sums *S* of elements of all sequences *A* with the same indices (in ascending order of indices). Use the single call of the MPI_Ssend function in each sender. Declare an array *Q* of the MPI_Request type in the receiver and call the MPI_Recv_init functions in a loop with a separate element of *Q* for each function call. Then call the MPI_Startall function in the receiver and, after that, call the MPI_Waitany function in a second loop to accumulate the sums *S*. Additionally, display the following data in the debug section in each iteration of the second loop (using two Show function and one ShowLine function calls): the duration of each MPI_Waitany function call (in milliseconds), the returned value of the third parameter (named index) of the MPI_Waitany function, and the rank of current sender that corresponds to the index parameter. To find the duration, call the MPI_Wtime function before and after the MPI_Waitany call and calculate the difference between returned values of the MPI_Wtime function multiplied by 1000. To find the rank of the current sender, use the value of the last parameter (of the MPI_Status type) returned by the MPI_Waitany function.

MPI2Send32. An integer *N* is given in each process. The value of *N* is equal to 1 in the first selected process (*the sender*), it is equal to 2 in some other selected processes (*the receivers*), it is equal to 0 in all other processes. Also a real number *A*, an integer *C*, and a sequence *R* of *C* integers are given in the sender, where *C* is the number of receivers and *R* contains ranks of all receivers. Send the number *A* to all receivers and output it in each receiver. Use the single call of the MPI_Recv function in each receiver. Declare an array *Q* of the MPI_Request type in the sender and call the MPI_Ssend_init functions in a loop with a separate element of *Q* for each function call. Then call the MPI_Startall function in the sender and, after that, call the MPI_Testany function in a second loop (the MPI_Testany function should be called in a nested loop until it returns a nonzero flag). Additionally, display the following data in the debug section in each iteration of the second loop (using the Show and ShowLine function call): the returned value of the third parameter (named index) of the MPI_Testany function (when it returns a nonzero flag), and the number of MPI_Testany function calls (that is, the number of iterations of the nested loop). Check how the debugging information changes if the MPI_Send_init function will be used instead of the MPI_Ssend_init function.

3. Collective communications

3.1. Collective data transfer

MPI3Coll1. An integer is given in the master process. Send the given integer to all slave processes using the MPI_Bcast function. Output the received integer in all slave processes.

MPI3Coll2. A sequence of 5 real numbers is given in the master process. Send the given sequence to all slave processes using the MPI_Bcast function. Output received data in all slave processes.

MPI3Coll3. A real number is given in each process. Send the given numbers to master process using the MPI_Gather function. Output received numbers in the master process in ascending order of ranks of sending processes (starting with the number that is given in the master process).

MPI3Coll4. A sequence of 5 integers is given in each process. Send the given sequences to master process using the MPI_Gather function. Output received data in the master process in ascending order of ranks of sending processes (starting with the sequence that is given in the master process).

MPI3Coll5. A sequence of $R + 2$ integers is given in each process; the integer R is equal to rank of the process (there are given 2 integers in the process 0, 3 integers in the process 1, and so on). Send the given sequences to master process using the MPI_Gatherv function. Output received data in the master process in ascending order of ranks of sending processes (starting with the sequence that is given in the master process).

MPI3Coll6. A sequence of K real numbers is given in the master process; K is the number of processes. Send one element of given sequence to each process (inclusive of the master process) using the MPI_Scatter function. Output the received number in each process.

MPI3Coll7. A sequence of $3K$ real numbers is given in the master process, K is the number of processes. Send three elements of given sequence to each process (inclusive of the master process) using the MPI_Scatter function. Output received numbers in each process.

MPI3Coll8. A sequence of K real numbers is given in the master process; K is the number of processes. Using the MPI_Scatterv function, send elements of given sequence to all processes as follows: the first element should be sent to the process $K - 1$, the second element should be sent to the process $K - 2$, …, the last element should be sent to the process 0). Output the received number in each process.

MPI3Coll9. A sequence of $K(K + 3)/2$ integers is given in the master process; K is the number of processes. Using the MPI_Scatterv function, send $R + 2$ elements of given sequence to the process of rank R, where $R = 0, \ldots, K - 1$: the first two elements should be sent to the process 0, the next three elements should be sent to the process 1, and so on. Output received numbers in each process.

MPI3Coll10. A sequence of $K + 2$ real numbers is given in the master process; K is the number of processes. Using the MPI_Scatterv function, send three elements of given sequence to each process as follows: elements with order numbers in the range $R + 1$ to $R + 3$ should be sent to the process of rank R, where $R = 0, \ldots, K - 1$ (the initial three elements should be sent to the process 0; the second, the third, and the fourth element should be sent to the process 1, and so on). Output received numbers in each process.

MPI3Coll11. A real number is given in each process. Send given numbers to all process using the MPI_Allgather function. Output received data in each process in ascending order of ranks of sending processes (inclusive of the number received from itself).

MPI3Coll12. Four integers are given in each process. Send given integers to all processes using the MPI_Allgather function. Output received data in each process in ascending order of ranks of sending processes (inclusive of the numbers received from itself).

MPI3Coll13. A sequence of $R + 2$ integers is given in each process; R is the rank of process (that is, two integers are given in the process 0, three integers are given in the process 1, and so on). Send given integers to all processes using the MPI_Allgatherv function. Output received data in each process in ascending order of ranks of sending processes (inclusive of the numbers received from itself).

MPI3Coll14. A sequence of K real numbers is given in each process; K is the number of processes. Using the MPI_Alltoall function, send one element of each given sequence to each process as follows: first element of each sequence should be sent to the process 0, second element of each sequence should be sent to the process 1, and so on. Output received numbers in each process in ascending order of ranks of sending processes (inclusive of the number received from itself).

MPI3Coll15. A sequence of $3K$ integers is given in each process; K is the number of processes. Using the MPI_Alltoall function, send three elements of each given sequence to each process as follows: the initial three elements of each sequence should be sent to the process 0, the next three elements of each sequence should be sent to the process 1, and so on. Output received numbers in each process in ascending order of ranks of sending processes (inclusive of the numbers received from itself).

MPI3Coll16. A sequence of $K(K + 1)/2$ integers is given in each process; K is the number of processes. Using the MPI_Alltoallv function, send some elements of each given sequence to each process as follows: the first element of each sequence should be sent to the process 0, the next two elements of each sequence should be sent to the process 1, the next three elements of each sequence should be sent to the process 2, and so on. Output received numbers in each process in ascending order of ranks of sending processes (inclusive of the numbers received from itself).

MPI3Coll17. A sequence of $K + 1$ real numbers is given in each process; K is the number of processes. Using the MPI_Alltoallv function, send two elements of each given sequence to each process as follows: the initial two elements of each sequence should be sent to the process 0, the second and the third element of each sequence should be sent to the process 1, …, the last two elements of each sequence should be sent to the last process. Output received numbers in each process in ascending order of ranks of sending processes (inclusive of the numbers received from itself).

MPI3Coll18. A sequence of $K + 1$ real numbers is given in each process; K is the number of processes. Using the MPI_Alltoallv function, send two elements of each given sequence to each process as follows: the last two elements of each sequence (with the order numbers $K + 1$ and K) should be sent to the process 0, the elements of each sequence with the order numbers $K - 1$ and K should be sent to the process 1, …, the initial two elements of each sequence should be sent to the last process. Output received numbers in each process in ascending order of ranks of sending processes (inclusive of the numbers received from itself).

3.2. Global reduction operations

MPI3Coll19. A sequence of $K + 5$ integers is given in each process; K is the number of processes. Find sums of elements of all given sequences with the same order number using the MPI_Reduce function with the MPI_SUM operation. Output received sums in the master process.

MPI3Coll20. A sequence of $K + 5$ real numbers is given in each process; K is the number of processes. Find the minimal value among the elements of all given sequences with the same order number using the MPI_Reduce function with the MPI_MIN operation. Output received minimal values in the master process.

MPI3Coll21. A sequence of $K + 5$ integers is given in each process; K is the number of processes. Using the MPI_Reduce function with the MPI_MAXLOC operation, find the maximal value among the elements of all given sequences with the same order number and also the rank of process that contains this maximal value. Output received maximal values

and ranks in the master process (first, output all maximal values, then output all corresponding ranks).

MPI3Coll22. A sequence of $K + 5$ real numbers is given in each process; K is the number of processes. Find products of elements of all given sequences with the same order number using the MPI_Allreduce function with the MPI_PROD operation. Output received products in each process.

MPI3Coll23. A sequence of $K + 5$ real numbers is given in each process; K is the number of processes. Using the MPI_Allreduce function with the MPI_MINLOC operation, find the minimal value among the elements of all given sequences with the same order number and also the rank of process that contains this minimal value. Output received minimal values in the master process and output corresponding ranks in each slave process.

MPI3Coll24. A sequence of K integers is given in each process; K is the number of processes. Using the MPI_Reduce_scatter function, find sums of elements of all given sequences with the same order number and send one sum to each process as follows: the first sum should be sent to the process 0, the second sum should be sent to the process 1, and so on. Output the received sum in each process.

MPI3Coll25. A sequence of $2K$ real numbers is given in each process; K is the number of processes. Using the MPI_Reduce_scatter function, find maximal values among elements of all given sequences with the same order number and send two maximal values to each process as follows: the initial two maximums should be sent to the process 0, the next two maximums should be sent to the process 1, and so on. Output received data in each process.

MPI3Coll26. A sequence of $K(K + 3)/2$ integers is given in each process; K is the number of processes. Using the MPI_Reduce_scatter function, find minimal values among elements of all given sequences with the same order number and send some minimal values to each process as follows: the initial two minimums should be sent to the process 0, the next three minimums should be sent to the process 1, …, the last $K + 1$ minimums should be sent to the process $K - 1$. Output received data in each process.

MPI3Coll27. A sequence of $K + 5$ real numbers is given in each process; K is the number of processes. Using the MPI_Scan function, find products of elements of given sequences with the same order number as follows: the products of elements of sequences given in the processes of rank 0, …, R should be found in the process R ($R = 0, 1, \ldots, K - 1$). Output received data in each process; in particular, products of elements of all given sequences should be output in the process $K - 1$.

MPI3Coll28. A sequence of $K + 5$ integers is given in each process; K is the number of processes. Using the MPI_Scan function, find maximal values

among elements of given sequences with the same order number as follows: the maximal values of elements of sequences given in the processes of rank 0, …, R should be found in the process R ($R = 0, 1, \ldots, K - 1$). Output received data in each process.

4. Derived datatypes and data packing

4.1. The simplest derived types

MPI4Type1. A sequence of $K - 1$ triples of integers is given in the master process; K is the amount of processes. Send all given data to each slave process using derived datatype with three integer elements and one collective operation with the derived datatype. Output received data in each slave process in the same order.

MPI4Type2. A sequence of $K - 1$ triples of integers is given in the master process; K is the amount of processes. Send one given triple at a time to each slave process using derived datatype with three integer elements and one collective operation with the derived datatype. Output received integers in each slave process in the same order.

MPI4Type3. A triple of integers is given in each slave process. Send all given triples to the master process using derived datatype with three integer elements and one collective operation with the derived datatype. Output received data in the master process in ascending order of ranks of sending processes.

MPI4Type4. A sequence of $K - 1$ triples of numbers is given in the master process; K is the amount of processes. Two initial items of each triple are integers, the last item is a real number. Send all given triples to each slave process using derived datatype with three elements (two integers and a real number) and one collective operation with the derived datatype. Output received data in each slave process in the same order.

MPI4Type5. A sequence of $K - 1$ triples of numbers is given in the master process; K is the amount of processes. The first item and the last item of each triple are integers, the middle item is a real number. Send one given triple at a time to each slave process using derived datatype with three elements (an integer, a real number, an integer) and one collective operation with the derived datatype. Output received data in each slave process in the same order.

MPI4Type6. A triple of numbers is given in each slave process. The first item of each triple is a real number, the other items are integers. Send all given triples to the master process using derived datatype with three elements (a real number and two integers) and one collective operation with the derived

datatype. Output received data in the master process in ascending order of ranks of sending processes.

MPI4Type7. A triple of numbers is given in each process. The first item and the last item of each triple are integers, the middle item is a real number. Send the given triples from each process to all processes using derived datatype with three elements (an integer, a real number, an integer) and one collective operation with the derived datatype. Output received data in each process in ascending order of ranks of sending processes (inclusive of data received from itself).

MPI4Type8. A sequence of R triples of numbers is given in each slave process; R is the rank of process. Two initial items of each triple are integers, the last item is a real number. Send all given triples to the master process using derived datatype with three elements (two integers and a real number) and one collective operation with the derived datatype. Output received data in the master process in ascending order of ranks of sending processes.

4.2. Data packing

MPI4Type9. Two sequences of K numbers are given in the master process; K is the amount of processes. The first given sequence contains integers, the second given sequence contains real numbers. Send all data to each slave process using the MPI_Pack and MPI_Unpack functions and one collective operation. Output received data in each slave process in the same order.

MPI4Type10. A sequence of $K-1$ triples of numbers is given in the master process; K is the amount of processes. The first item and the last item of each triple are integers, the middle item is a real number. Send one given triple at a time to each slave process using the pack/unpack functions and one collective operation. Output received numbers in each slave process in the same order.

MPI4Type11. A sequence of $K-1$ triples of numbers is given in the master process; K is the amount of processes. Two initial items of each triple are integers, the last item is a real number. Send all given triples to each slave process using the pack/unpack functions and one collective operation. Output received data in each slave process in the same order.

MPI4Type12. A triple of numbers is given in each slave process. Two initial items of each triple are integers, the last item is a real number. Send the given triples from each slave process to the master process using the pack/unpack functions and one collective operation. Output received data in the master process in ascending order of ranks of sending processes.

MPI4Type13. A real number and a sequence of R integers are given in each slave process; R is the rank of process (one integer is given in the process 1, two integers are given in the process 2, and so on). Send all giv-

en data from each slave process to the master process using the pack/unpack functions and one collective operation. Output received data in the master process in ascending order of ranks of sending processes.

4.3. Additional ways of derived types creation

MPI4Type14. Two sequences of integers are given in the main process: the sequence A of the size $3K$ and the sequence B of the size K, where K is the number of slave processes. The elements of sequences are numbered from 1. Send NR elements of the sequence A to each slave process R ($R = 1, 2, \ldots, K$) starting with the A_R and increasing the ordinal number by 2 (R, $R + 2$, $R + 4$, …). For example, if N_2 is equal to 3 then the process 2 should receive the elements A_2, A_4, A_6. Output all received data in each slave process. Use one call of the MPI_Send, MPI_Probe, and MPI_Recv functions for sending numbers to each slave process; the MPI_Recv function should return an array that contains only elements that should be output. To do this, define a new datatype that contains a single integer and an additional empty space (*a hole*) of a size that is equal to the size of integer datatype. Use the following data as parameters for the MPI_Send function: the given array A with the appropriate displacement, the amount N_R of sending elements, a new datatype. Use an integer array of the size N_R and the MPI_INT datatype in the MPI_Recv function. To determine the number N_R of received elements, use the MPI_Get_count function in the slave processes.

Note. Use the MPI_Type_create_resized function to define the hole size for a new datatype (this function should be applied to the MPI_INT datatype). In the MPI-1, the zero-size *upper-bound marker* MPI_UB should be used jointly with the the MPI_Type_struct for this purpose (in MPI-2, the MPI_UB pseudo-datatype is deprecated).

MPI4Type15. An real-valued square matrix of order K is given in the master process; K is the number of slave processes. Elements of the matrix should be stored in a one-dimensional array A in a row-major order. The columns of matrix are numbered from 1. Send R-th column of matrix to the process of rank R ($R = 1, 2, \ldots, K$) and output all received elements in each slave process. Use one call of the MPI_Send and MPI_Recv functions for sending elements to each slave process; the MPI_Recv function should return an array that contains only elements that should be output. To do this, define a new datatype that contains a single real number and an additional empty space (*a hole*) of the appropriate size. Use the following data as parameters for the MPI_Send function: the given array A with the appropriate displacement, the amount K of sending elements (i. e., the size of column), a new datatype. Use a real-valued array of the size K and the MPI_DOUBLE datatype in the MPI_Recv function.

Note. See the note to MPI4Type14.

MPI4Type16. *R*-th column of a real-valued square matrix of order *K* is given in the slave process of rank *R* (*R* = 1, 2, …, *K*); *K* is the number of slave processes, the columns of matrix are numbered from 1. Send all columns to the master process and store them in a one-dimensional array *A* in a row-major order. Output all elements of *A* in the master process. Use one call of the MPI_Send and MPI_Recv functions for sending elements of each column; the resulting array *A* with the appropriate displacement should be the first parameter for the MPI_Recv function, and a number 1 should be its second parameter. To do this, define a new datatype (in the master process) that contains *K* real numbers and an empty space (*a hole*) of the appropriate size after each number. Define a new datatype in two steps. In the first step, define auxiliary datatype that contains one real number and additional hole (see the note to MPI4Type14). In the second step, define the final datatype using the MPI_Type_contiguous function (this datatype should be the third parameter for the MPI_Recv function). The MPI_Type_commit function is sufficient to call only for the final datatype. Use a real-valued array of size *K* and the MPI_DOUBLE datatype in the MPI_Send function.

MPI4Type17. The number of slave processes *K* is a multiple of 3 and does not exceed 9. An integer *N* is given in each process, all the numbers *N* are the same and are in the range from 3 to 5. Also an integer square matrix of order *N* (*a block*) is given in each slave process; the block should be stored in a one-dimensional array *B* in a row-major order. Send all arrays *B* to the master process and compose a block matrix of the size (*K*/3) × 3 (the size is indicated in blocks) using a row-major order for blocks (i. e., the first row of blocks should include blocks being received from the processes 1, 2, 3, the second row of blocks should include blocks from the processes 4, 5, 6, and so on). Store the block matrix in the one-dimensional array *A* in a row-major order. Output all elements of *A* in the master process. Use one call of the MPI_Send and MPI_Recv functions for sending each block *B*; the resulting array *A* with the appropriate displacement should be the first parameter for the MPI_Recv function, and a number 1 should be its second parameter. To do this, define a new datatype (in the master process) that contains *N* sequences, each sequence contains *N* integers, and an empty space (*a hole*) of the appropriate size should be placed between the sequences. Define the required datatype using the MPI_Type_vector function (this datatype should be the third parameter for the MPI_Recv function). Use the array *B* of size *N*·*N* and the MPI_INT datatype in the MPI_Send function.

MPI4Type18. The number of slave processes *K* is a multiple of 3 and does not exceed 9. An integer *N* in the range from 3 to 5 and an integer *block matrix* of the size (*K*/3) × 3 (the size is indicated in blocks) are given in the master process. Each block is *a lower triangular matrix* of order *N*, the block contains all matrix elements, inclusive of zero-valued ones. The block matrix

should be stored in the one-dimensional array *A* in a row-major order. Send a non-zero part of each block to the corresponding slave process in a row-major order of blocks (i. e., the blocks of the first row should be sent to the processes 1, 2, 3, the blocks of the second row should be sent to the processes 4, 5, 6, and so on). Output all received elements in each slave process (in a row-major order). Use one call of the MPI_Send, MPI_Probe, and MPI_Recv functions for sending each block; the resulting array *A* with the appropriate displacement should be the first parameter for the MPI_Send function, and a number 1 should be its second parameter. To do this, define a new datatype (in the master process) that contains *N* sequences, each sequence contains non-zero part of the next row of a lower triangular block (the first sequence consists of 1 element, the second sequence consists of 2 elements, and so on), and an empty space (*a hole*) of the appropriate size should be placed between the sequences. Define the required datatype using the MPI_Type_indexed function (this datatype should be the third parameter for the MPI_Send function). Use an integer array *B*, which contains a non-zero part of received block, and the MPI_INT datatype in the MPI_Recv function. To determine the number of received elements, use the MPI_Get_count function in the slave processes.

MPI4Type19. The number of slave processes *K* is a multiple of 3 and does not exceed 9. An integer *N* is given in each process, all the numbers *N* are the same and are in the range from 3 to 5. Also an integer *P* and a non-zero part of an integer square matrix of order *N* (*a Z-block*) are given in each slave process. The given elements of Z-block should be stored in a one-dimensional array *B* in a row-major order. These elements are located in the Z-block in the form of the symbol "Z", i. e. they occupy the first and last row, and also the antidiagonal. Define a zero-valued integer matrix of the size $N\cdot(K/3) \times 3N$ in the master process (all elements of this matrix are equal to 0 and should be stored in a one-dimensional array *A* in a row-major order). Send a non-zero part of the given Z-block from each slave process to the master process in ascending order of ranks of sending processes and write each received Z-block in the array *A* starting from the element of array *A* with index *P* (the positions of Z-blocks can overlap, in this case the elements of blocks received from processes of higher rank will replace some of the elements of previously written blocks). Output all elements of *A* in the master process. Use one call of the MPI_Send and MPI_Recv functions for sending each Z-block; the array *A* with the appropriate displacement should be the first parameter for the MPI_Recv function, and a number 1 should be its second parameter. To do this, define a new datatype (in the master process) that contains *N* sequences, the first and the last sequences contain *N* integers, the other sequences contain 1 integer, and an empty space (*a hole*) of the appropriate size should be placed

between the sequences. Define the required datatype using the MPI_Type_indexed function (this datatype should be the third parameter for the MPI_Recv function). Use the array *B*, which contains a non-zero part of a Z-block, and the MPI_INT datatype in the MPI_Send function.

Note. Use the msgtag parameter to send the Z-block insertion position *P* to the main process. To do this, set the value of *P* as the msgtag parameter for the MPI_Send function in slave processes, call the MPI_Probe function with the MPI_ANY_TAG parameter in the master process (before calling the MPI_Recv function), and analyze its returned parameter of the MPI_Status type.

MPI4Type20. The number of slave processes *K* is a multiple of 3 and does not exceed 9. An integer *N* is given in each process, all the numbers *N* are the same and are in the range from 3 to 5. Also an integer *P* and a non-zero part of an integer square matrix of order *N* (*an U-block*) are given in each slave process. The given elements of U-block should be stored in a one-dimensional array *B* in a row-major order. These elements are located in the U-block in the form of the symbol "U", i. e. they occupy the first and last column, and also the last row. Define a zero-valued integer matrix of the size $N \cdot (K/3) \times 3N$ in the master process (all elements of this matrix are equal to 0 and should be stored in a one-dimensional array *A* in a row-major order). Send a non-zero part of the given U-block from each slave process to the master process in ascending order of ranks of sending processes and write each received U-block in the array *A* starting from the element of array *A* with index *P* (the positions of U-blocks can overlap, in this case the elements of blocks received from processes of higher rank will replace some of the elements of previously written blocks). Output all elements of *A* in the master process. Use one call of the MPI_Send and MPI_Recv functions for sending each U-block; the array *A* with the appropriate displacement should be the first parameter for the MPI_Recv function, and a number 1 should be its second parameter. To do this, define a new datatype (in the master process) that contains appropriate number of sequences with empty spaces (*holes*) between them. Define the required datatype using the MPI_Type_indexed function (this datatype should be the third parameter for the MPI_Recv function). Use the array *B*, which contains a non-zero part of an U-block, and the MPI_INT datatype in the MPI_Send function.

Note. See the note to MPI4Type19.

4.4. The MPI_Alltoallw function (MPI-2)

MPI4Type21. Solve the MPI4Type15 task by using one collective operation instead of the MPI_Send and MPI_Recv functions to pass data.

Note. You cannot use the functions of the Scatter group, since the displacements for the passing data items (columns of the matrix) should be specified in *bytes* rather than in elements. Therefore, you should use the function MPI_Alltoallw introduced in MPI-2, which allows you to configure the collective communications in the most flexible way. In this case, the MPI_Alltoallw function should be used to implement a data passing of the Scatter type (and most of the array parameters used in this function need to be defined differently in the master and slave processes).

MPI4Type22. Solve the MPI4Type16 task by using one collective operation instead of the MPI_Send and MPI_Recv functions to pass data.
Note. See the note to MPI4Type21. In this case, the MPI_Alltoallw function should be used to implement a data passing of the Gather type.

---

5. Process groups and communicators

5.1. Creation of new communicators

MPI5Comm1. A sequence of *K* integers is given in then master process; *K* is the number of processes whose rank is an even number (0, 2, …). Create a new communicator that contains all even-rank processes using the MPI_Comm_group, MPI_Group_incl, and MPI_Comm_create functions. Send one given integer to each even-rank process (including the master process) using one collective operation with the created communicator. Output received integer in each even-rank process.

MPI5Comm2. Two real numbers are given in each process whose rank is an odd number (1, 3, …). Create a new communicator that contains all odd-rank processes using the MPI_Comm_group, MPI_Group_excl, and MPI_Comm_create functions. Send all given numbers to each odd-rank process using one collective operation with the created communicator. Output received numbers in each odd-rank process in ascending order of ranks of sending processes (including numbers received from itself).

MPI5Comm3. Three integers are given in each process whose rank is a multiple of 3 (including the master process). Using the MPI_Comm_split function, create a new communicator that contains all processes with ranks that are a multiple of 3. Send all given numbers to master process using one collective operation with the created communicator. Output received integers in the master process in ascending order of ranks of sending processes (including integers received from the master process).
Note. When calling the MPI_Comm_split function in processes that are not required to include in the new communicator, one should specify the constant MPI_UNDEFINED as the color parameter.

MPI5Comm4. A sequence of 3 real numbers is given in each process whose rank is an even number (including the master process). Find the minimal value among the elements of the given sequences with the same order number using a new communicator and one global reduction operation. Output received minimums in the master process.

Note. See the note to MPI5Comm3.

MPI5Comm5. A real number is given in each process. Using the MPI_Comm_split function and one global reduction operation, find the maximal value among the numbers given in the even-rank processes (including the master process) and the minimal value among the numbers given in the odd-rank processes. Output the maximal value in the process 0 and the minimal value in the process 1.

Note. The program should contain a single MPI_Comm_split call, which creates the both required communicators (each for the corresponding group of processes).

MPI5Comm6. An integer K and a sequence of K real numbers are given in the master process, an integer N is given in each slave process. The value of N is equal to 1 for some processes and is equal to 0 for others; the number of processes with $N = 1$ is equal to K. Send one real number from the master process to each slave process with $N = 1$ using the MPI_Comm_split function and one collective operation. Output the received numbers in these slave processes.

Note. See the note to MPI5Comm3.

MPI5Comm7. An integer N is given in each process; the value of N is equal to 1 for at least one process and is equal to 0 for others. Also a real number A is given in each process with $N = 1$. Send all numbers A to the first process with $N = 1$ using the MPI_Comm_split function and one collective operation. Output received numbers in this process in ascending order of ranks of sending processes (including the number received from this process).

Note. See the note to MPI5Comm3.

MPI5Comm8. An integer N is given in each process; the value of N is equal to 1 for at least one process and is equal to 0 for others. Also a real number A is given in each process with $N = 1$. Send all numbers A to the last process with $N = 1$ using the MPI_Comm_split function and one collective operation. Output received numbers in this process in ascending order of ranks of sending processes (including the number received from this process).

Note. See the note to MPI5Comm3.

MPI5Comm9. An integer N is given in each process; the value of N is equal to 1 for at least one process and is equal to 0 for others. Also a real num-

ber A is given in each process with $N = 1$. Send all numbers A to each process with $N = 1$ using the MPI_Comm_split function and one collective operation. Output received numbers in these processes in ascending order of ranks of sending processes (including the number received from itself).
Note. See the note to MPI5Comm3.

MPI5Comm10. An integer N is given in each process; the value of N is equal to 1 for some processes and is equal to 2 for others, there are at least one process with $N = 1$ and one process with $N = 2$. Also an integer A is given in each process. Using the MPI_Comm_split function and one collective operation, send integers A from all processes with $N = 1$ to each process with $N = 1$ and from all processes with $N = 2$ to each process with $N = 2$. Output received integers in each process in ascending order of ranks of sending processes (including the integer received from itself).
Note. See the note to MPI5Comm5.

MPI5Comm11. An integer N is given in each process; the value of N is equal to 1 for at least one process and is equal to 0 for others. Also a real number A is given in each process with $N = 1$. Find the sum of all real numbers A using the MPI_Comm_split function and one global reduction operation. Output the received sum in each process with $N = 1$.
Note. See the note to MPI5Comm3.

MPI5Comm12. An integer N is given in each process; the value of N is equal to 1 for some processes and is equal to 2 for others, there are at least one process with $N = 1$ and one process with $N = 2$. Also a real number A is given in each process. Using the MPI_Comm_split function and one global reduction operation, find the minimal value among the numbers A given in the processes with $N = 1$ and the maximal value among the numbers A given in the processes with $N = 2$. Output the minimal value in each process with $N = 1$ and the maximal value in each process with $N = 2$.
Note. See the note to MPI5Comm5.

5.2. Virtual topologies

MPI5Comm13. An integer N (> 1) is given in the master process; the number of processes K is assumed to be a multiple of N. Send the integer N to all processes and define a Cartesian topology for all processes as a $(N \times K/N)$ grid using the MPI_Cart_create function (ranks of processes should not be reordered). Find the process coordinates in the created topology using the MPI_Cart_coords function and output the process coordinates in each process.

MPI5Comm14. An integer N (> 1) is given in the master process; the number N is not greater than the number of processes K. Send the integer N to all processes and define a Cartesian topology for the initial part of processes as

a ($N \times K/N$) grid using the MPI_Cart_create function (the symbol "/" denotes the operator of integer division, ranks of processes should not be reordered). Output the process coordinates in each process included in the Cartesian topology.

MPI5Comm15. The number of processes K is an even number: $K = 2N$, $N > 1$. A real number A is given in the processes 0 and N. Define a Cartesian topology for all processes as a ($2 \times N$) grid. Using the MPI_Cart_sub function, split this grid into two one-dimensional subgrids (namely, *rows*) such that the processes 0 and N were the master processes in these rows. Send the given number A from the master process of each row to each process of the same row using one collective operation. Output the received number in each process (including the processes 0 and N).

MPI5Comm16. The number of processes K is an even number: $K = 2N$, $N > 1$. A real number A is given in the processes 0 and 1. Define a Cartesian topology for all processes as a ($N \times 2$) grid. Using the MPI_Cart_sub function, split this grid into two one-dimensional subgrids (namely, *columns*) such that the processes 0 and 1 were the master processes in these columns. Send the given number A from the master process of each column to each process of the same column using one collective operation. Output the received number in each process (including the processes 0 and 1).

MPI5Comm17. The number of processes K is a multiple of 3: $K = 3N$, $N > 1$. A sequence of N integers is given in the processes 0, N, and $2N$. Define a Cartesian topology for all processes as a ($3 \times N$) grid. Using the MPI_Cart_sub function, split this grid into three one-dimensional subgrids (namely, *rows*) such that the processes 0, N, and $2N$ were the master processes in these rows. Send one given integer from the master process of each row to each process of the same row using one collective operation. Output the received integer in each process (including the processes 0, N, and $2N$).

MPI5Comm18. The number of processes K is a multiple of 3: $K = 3N$, $N > 1$. A sequence of N integers is given in the processes 0, 1, and 2. Define a Cartesian topology for all processes as a ($N \times 3$) grid. Using the MPI_Cart_sub function, split this grid into three one-dimensional subgrids (namely, *columns*) such that the processes 0, 1, and 2 were the master processes in these columns. Send one given integer from the master process of each column to each process of the same column using one collective operation. Output the received integer in each process (including the processes 0, 1, and 2).

MPI5Comm19. The number of processes K is equal to 8 or 12. An integer is given in each process. Define a Cartesian topology for all processes as a three-dimensional ($2 \times 2 \times K/4$) grid (ranks of processes should not be reordered), which should be considered as 2 two-dimensional ($2 \times K/4$) subgrids (namely, *matrices*) that contain processes with the identical first

coordinate in the Cartesian topology. Split each matrix into two one-dimensional rows of processes. Send given integers from all processes of each row to the master process of the same row using one collective operation. Output received integers in the master process of each row (including integer received from itself).

MPI5Comm20. The number of processes K is equal to 8 or 12. An integer is given in each process. Define a Cartesian topology for all processes as a three-dimensional $(2 \times 2 \times K/4)$ grid (ranks of processes should not be reordered), which should be considered as $K/4$ two-dimensional (2×2) subgrids (namely, *matrices*) that contain processes with the identical third coordinate in the Cartesian topology. Split this grid into $K/4$ matrices of processes. Send given integers from all processes of each matrix to the master process of the same matrix using one collective operation. Output received integers in the master process of each matrix (including integer received from itself).

MPI5Comm21. The number of processes K is equal to 8 or 12. A real number is given in each process. Define a Cartesian topology for all processes as a three-dimensional $(2 \times 2 \times K/4)$ grid (ranks of processes should not be reordered), which should be considered as 2 two-dimensional $(2 \times K/4)$ subgrids (namely, *matrices*) that contain processes with the identical first coordinate in the Cartesian topology. Split each matrix into $K/4$ one-dimensional columns of processes. Using one global reduction operation, find the product of all numbers given in the processes of each column. Output the product in the master process of the corresponding column.

MPI5Comm22. The number of processes K is equal to 8 or 12. A real number is given in each process. Define a Cartesian topology for all processes as a three-dimensional $(2 \times 2 \times K/4)$ grid (ranks of processes should not be reordered), which should be considered as $K/4$ two-dimensional (2×2) subgrids (namely, *matrices*) that contain processes with the identical third coordinate in the Cartesian topology. Split this grid into $K/4$ matrices of processes. Using one global reduction operation, find the sum of all numbers given in the processes of each matrix. Output the sum in the master process of the corresponding matrix.

MPI5Comm23. Positive integers M and N are given in the master process; the product of the numbers M and N is less than or equal to the number of processes. Also integers X and Y are given in each process whose rank is in the range 0 to $M \cdot N - 1$. Send the numbers M and N to all processes and define a Cartesian topology for initial $M \cdot N$ processes as a two-dimensional $(M \times N)$ grid, which is periodic in the first dimension (ranks of processes should not be reordered). Using the MPI_Cart_rank function, output the rank of process with the coordinates X, Y (taking into account periodicity)

in each process included in the Cartesian topology. Output −1 in the case of erroneous coordinates.

Note. If invalid coordinates are specified when calling the MPI_Cart_rank function (for instance, in the case of negative coordinates for non-periodic dimensions) then the function itself returns an error code (instead of the successful return code MPI_SUCCESS) whereas the return value of the rank parameter is undefined. So, in this task, the number −1 should be output when the MPI_Cart_rank function return a value that differs from MPI_SUCCESS. To suppress the output of error messages in the debug section of the Programming Taskbook window, it is enough to set the special error handler named MPI_ERROR_RETURN before calling a function that may be erroneous (use the MPI_Comm_set_errhandler function or, in MPI-1, the MPI_Errhandler_set function). When an error occurs in some function, this error handler performs no action except setting an error return value for this function. In MPICH version 1.2.5, the MPI_Cart_rank function returns the rank parameter equal to −1 when the process coordinates are invalid. This feature may sumplify the solution; however, in this case, one also should suppress the output of error messages by means of special error handler setting.

MPI5Comm24. Positive integers M and N are given in the master process; the product of the numbers M and N is less than or equal to the number of processes. Also integers X and Y are given in each process whose rank is in the range 0 to $M \cdot N - 1$. Send the numbers M and N to all processes and define a Cartesian topology for initial $M \cdot N$ processes as a two-dimensional ($M \times N$) grid, which is periodic in the second dimension (ranks of processes should not be reordered). Using the MPI_Cart_rank function, output the rank of process with the coordinates X, Y (taking into account periodicity) in each process included in the Cartesian topology. Output −1 in the case of erroneous coordinates.

Note. See the note to MPI5Comm23.

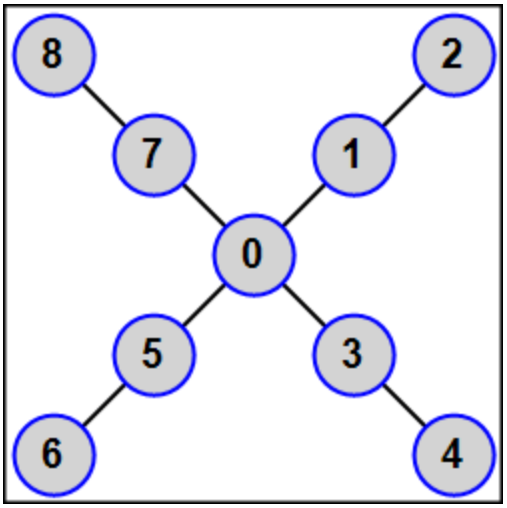
MPI5Comm25. A real number is given in each slave process. Define a Cartesian topology for all processes as a one-dimensional grid. Using the MPI_Send and MPI_Recv functions, perform a shift of given data by step −1 (that is, the real number given in each process should be sent to the process of the previous rank). Ranks of source and destination process should be determined by means of the MPI_Cart_shift function. Output received data in each destination process.

MPI5Comm26. The number of processes K is an even number: $K = 2N$, $N > 1$. A real number A is given in each process. Define a Cartesian topology for all processes as a two-dimensional ($2 \times N$) grid (namely, *matrix*); ranks of processes should not be reordered. Using the MPI_Sendrecv function, per-

form a cyclic shift of data given in all processes of each row of the matrix by step 1 (that is, the number *A* should be sent from each process in the row, except the last process, to the next process in the same row and from the last process in the row to the first process in the same row). Ranks of source and destination process should be determined by means of the MPI_Cart_shift function. Output received data in each process.

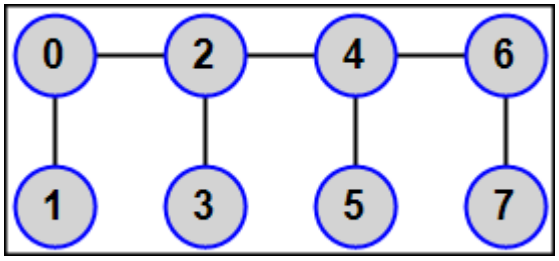
MPI5Comm27. The number of processes *K* is equal to 8 or 12. A real number *A* is given in each process. Define a Cartesian topology for all processes as a three-dimensional ($2 \times 2 \times K/4$) grid (ranks of processes should not be reordered), which should be considered as $K/4$ two-dimensional (2×2) subgrids (namely, *matrices*) that contain processes with the identical third coordinate in the Cartesian topology and are ordered by the value of this third coordinate. Using the MPI_Sendrecv_replace function, perform a cyclic shift of data given in all processes of each matrix by step 1 (that is, the numbers *A* should be sent from all processes of each matrix, except the last matrix, to the corresponding processes of the next matrix and from all processes of the last matrix to the corresponding processes of the first matrix). Ranks of source and destination process should be determined by means of the MPI_Cart_shift function. Output received data in each process.

MPI5Comm28. The number of processes *K* is an odd number: $K = 2N + 1$ ($1 < N < 5$). An integer *A* is given in each process. Using the MPI_Graph_create function, define a graph topology for all processes as follows: the master process must be connected by edges to all odd-rank processes (that is, to the processes 1, 3, …, $2N - 1$); each process of the rank *R*, where *R* is a positive even number (2, 4, …, $2N$), must be connected by edge to the process of the rank $R - 1$. Thus, the graph represents *N*-beam star whose center is the master process, each star beam consists of two slave processes of ranks *R* and $R + 1$, and each odd-rank process *R* is adjacent to star center (namely, the master process).



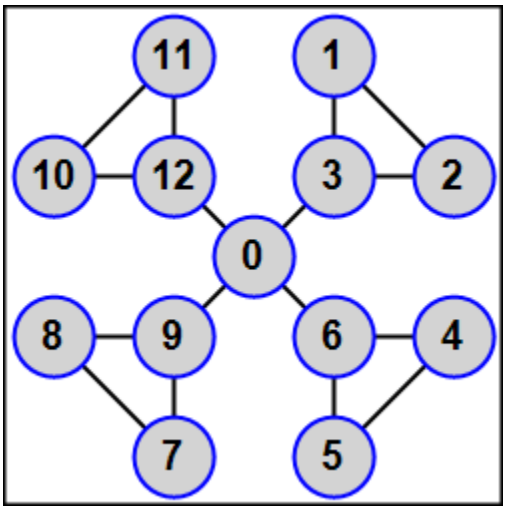
Using the MPI_Send and MPI_Recv functions, send the given integer A from each process to all its adjacent processes (its *neighbors*). The amount and ranks of neighbors should be determined by means of the MPI_Graph_neighbors_count and MPI_Graph_neighbors functions respectively. Output received data in each process in ascending order of ranks of sending processes.

MPI5Comm29. The number of processes K is an even number: $K = 2N$ ($1 < N < 6$). An integer A is given in each process. Using the MPI_Graph_create function, define a graph topology for all processes as follows: all even-rank processes (including the master process) are linked in a chain 0 — 2 — 4 — 6 — … — ($2N - 2$); each process with odd rank R (1, 3, …, $2N - 1$) is connected by edge to the process with the rank $R - 1$. Thus, each odd-rank process has a single neighbor, the first and the last even-rank processes have two neighbors, and other even-rank processes (the "inner" ones) have three neighbors.



Using the MPI_Sendrecv function, send the given integer A from each process to all its neighbors. The amount and ranks of neighbors should be determined by means of the MPI_Graph_neighbors_count and MPI_Graph_neighbors functions respectively. Output received data in each process in ascending order of ranks of sending processes.

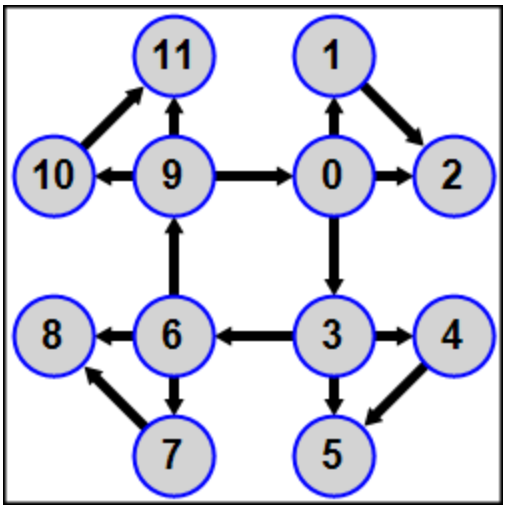
MPI5Comm30. The number of processes K is equal to $3N + 1$ ($1 < N < 5$). An integer A is given in each process. Using the MPI_Graph_create function, define a graph topology for all processes as follows: processes of ranks R, $R + 1$, $R + 2$, where $R = 1, 4, 7, \ldots$, are linked by edges and, moreover, each process whose rank is positive and is a multiple of 3 (that is, the process 3, 6, …) is connected by edge to the master process. Thus, the graph represents N-beam star whose center is the master process, each star beam consists of three linked slave processes, and each slave process with rank that is a multiple of 3 is adjacent to star center (namely, the master process).



Using the MPI_Sendrecv function, send the given integer A from each process to all its neighbors. The amount and ranks of neighbors should be determined by means of the MPI_Graph_neighbors_count and MPI_Graph_neighbors functions respectively. Output received data in each process in ascending order of ranks of sending processes.

5.3. The distributed graph topology (MPI-2)

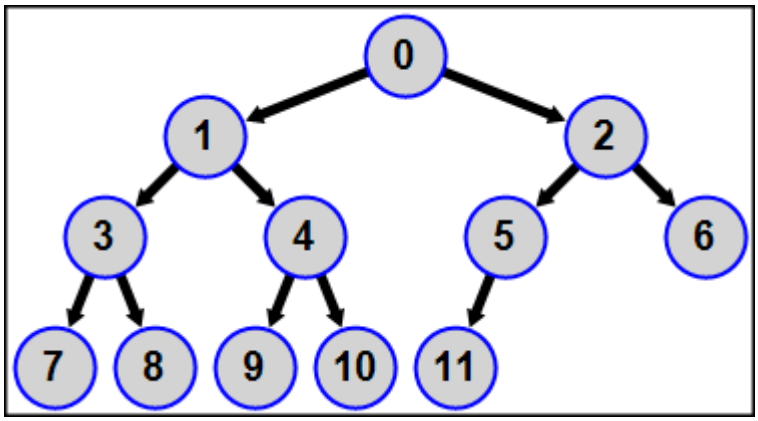
MPI5Comm31. The number of processes K is a multiple of 3; an integer A is given in each process. Using the MPI_Dist_graph_create function, define a distributed graph topology for all processes as follows: all processes whose rank is a multiple of 3 (0, 3, …, $K-3$) are linked in a ring, each process in this ring is the source for the next process of the ring (that is, the process 0 is the source for the process 3, the process 3 is the source for the process 6, …, the process $K-3$ is the source for the process 0); besides, the process $3N$ ($N = 0, 1, \ldots, K/3-1$) is the source for the processes $3N+1$ and $3N+2$, and the process $3N+1$ is the source for the process $3N+2$.



The complete definition of the graph topology should be given in the master process (whereas the second parameter of the MPI_Dist_graph_create

function should be equal to 0 in all slave processes). The weights parameter should be equal to MPI_UNWEIGHTED, the info parameter should be equal to MPI_INFO_NULL, ranks of processes should not be reordered. Using the MPI_Send and MPI_Recv functions, send the given numbers from the source processes to the destination processes and output the sum of the given number *A* and all received numbers in each process. The amount and ranks of source and destination processes should be determined by means of the MPI_Dist_graph_neighbors_count and MPI_Dist_graph_neighbors functions.

MPI5Comm32. The number of processes is in the range 4 to 15; an integer *A* is given in each process. Using the MPI_Dist_graph_create function, define a distributed graph topology for all processes as follows: all processes form a binary tree with the process 0 as a tree root, the processes 1 and 2 as a tree nodes of level 1, the processes from 3 to 6 as a tree nodes of level 2 (the processes 3 and 4 are the child nodes of the process 1 and the processes 5 and 6 are the child nodes of the process 2), and so on. Each process is the source for all its child nodes, so each process has 0 to 2 destination processes.



The complete definition of the graph topology should be given in the master process (whereas the second parameter of the MPI_Dist_graph_create function should be equal to 0 in all slave processes). The weights parameter should be equal to MPI_UNWEIGHTED, the info parameter should be equal to MPI_INFO_NULL, ranks of processes should not be reordered. Using the MPI_Send and MPI_Recv functions in each process, find and output the sum of the given number *A* and all the numbers that are given in ancestors of all levels—from the tree root (the master process) to the nearest ancestor (the parent process). The amount and ranks of source and destination processes should be determined by means of the MPI_Dist_graph_neighbors_count and MPI_Dist_graph_neighbors functions.

6. Parallel file input-output (MPI-2)

Use the char[12] array to store the filename, use the MPI_Bcast function with the MPI_CHAR datatype parameter to send the filename from the master process to the slave processes.

You do not need to set the view of the data in the file by means the MPI_File_set_view function in the initial two subgroups (MPI6File1–MPI6File16); it is enough to use the default view, in which both the elementary datatype and the filetype are of the MPI_BYTE type, the initial displacement is equal to 0 for all processes, and the "native" data representation is used. The same data representation should be specified for the file view in the tasks of the third subgroup (MPI6File17–MPI6File30).

Use the MPI_Type_get_extent function to determine the extent of the MPI_INT and MPI_DOUBLE types.

Use the function MPI_Type_create_resized to specify the additional empty space in tasks devoted to the file view setting. One can use also *the zero-size upper-bound marker* MPI_UB but this pseudo-datatype is deprecated in the MPI-2 standard.

6.1. Local functions for file input-output

MPI6File1. The name of existing file of integers is given in the master process; the amount of file items, which should be read, and their ordinal numbers are given in each slave process (the file items are numbered from 1). File items with some numbers may be missing in the source file. Using the required number of the MPI_File_read_at local function calls in each process, read existing file items with the specified ordinal numbers from the source file and output them in the same order. To check existence of file item with the specified ordinal number, you can either use the MPI_File_get_size function, or analyze the MPI_Status parameter of the MPI_File_read_at function.

MPI6File2. The name of file is given in the master process; the amount of pairs of integers and the pairs themselves are given in each slave process; the first term of the pair is the ordinal number of the file item, the second term of the pair is the value of this file item (the file items are numbered from 1, all ordinal numbers are different and cover range from 1 to some integer). Create a new file of integers with the given name and write the given data to this file using the required number of the MPI_File_write_at local function calls in each slave process.

MPI6File3. The name of existing file of real numbers is given in the master process. The file contains elements of the $K \times N$ matrix, where K is the number of slave processes. Using one call of the MPI_File_read_at local function in each slave process, read and output elements of Rth matrix row

in the process of rank *R* (rows are numbered from 1). Use the MPI_File_get_size function to determine the file size.

MPI6File4. The name of file is given in the master process; a sequence of *R* real numbers is given in each slave process, where *R* is the process rank. Create a new file of real numbers with the given name and write the given data to this file in ascending order of ranks of processes containing these data. Use one call of the MPI_File_write_at local function in each slave process.

MPI6File5. The name of existing file of integers is given in the master process. The file contains all integers in the range from 1 to *K*, where *K* is the maximal rank of process. Read and output two sequences of the file items in each slave process. The first sequence contains the *initial* part of the file items until the first item whose value is equal to the process rank (including this item); the second sequence contains the *final* part of the file items and has the same size as the first one. Elements of each sequence should be output in the order they are stored in the file. Use the required number of the MPI_File_read local function calls in each process (without using arrays) and also the MPI_File_get_position function call to determine the size of the first sequence and the MPI_File_seek function call with the MPI_SEEK_END parameter to move the file pointer to the beginning of the second sequence.

MPI6File6. The name of file is given in the master process; an integer is given in each slave process. Create a new file of integers with the given name and write *K* successive copies of each given integer to this file, where *K* is the number of slave processes. The order of integers in the file should be the inverse of the order of the slave processes (i. e., *K* copies of the integer from the process 1 should be written at the end of file, *K* copies of the integer from the process 2 should be written before them, and so on). Use one call of the MPI_File_write local function in each slave process and also the MPI_File_seek function with the MPI_SEEK_SET parameter.

MPI6File7. The name of existing file of integers is given in the master process. The sum of all file item values is greater than *K*, where *K* is the number of slave processes. Read initial file items in each slave process until the sum of their values exceeds the rank of process and output this sum and the amount *N* of read numbers. After that, in addition, read and output the values of the *last N* file items in the order they are stored in the file. Use the required number of the MPI_File_read local function calls in each slave process (without using arrays) and also the MPI_File_get_position function call to determine the amount *N* and the MPI_File_seek function call with the MPI_SEEK_END parameter to move the file pointer to beginning of the group of the last *N* items.

MPI6File8. The name of file is given in the master process; a real number is given in each slave process. Create a new file of real numbers with the given name and write *R* successive copies of each given real number to this file, where *R* is equal to the rank of the process in which this number is given. The order of numbers in the file should be the *inverse* of the order of the slave processes (i. e., single copy of the real number from the process 1 should be written at the end of file, two copies of the real number from the process 2 should be written before it, and so on). Use one call of the MPI_File_write local function in each slave process and also the MPI_File_seek function with the MPI_SEEK_SET parameter.

6.2. Collective functions for file input-output

MPI6File9. The name of existing file of integers is given in the master process. Read and output the $R + 1$ file items in each process starting with the item with the ordinal number $R + 1$, where *R* is the process rank (0, 1, …). File items are numbered from 1; thus, you should read only the initial file item in the process 0, the two following items in the process 1, the three items starting with the third one in the process 2, and so on. If the file does not contain enough items then, in some processes, the number of output items may be less than the required one. Use one call of the MPI_File_read_all collective function call and also the MPI_File_get_size function to determine the file size.

Note. In MPICH2 1.3, the function MPI_File_read_all does not allow to determine the number of actually read file items based on the information contained in the MPI_Status parameter: this parameter always contain the number of items to be read, and if there are not enough items in the file then zero values are appended to the output array.

MPI6File10. The name of file is given in the master process; a sequence of *R* integers is given in each slave process, where *R* is the process rank (1, 2, …). Create a new file of integers with the given name and write the given sequences to this file in ascending order of ranks of processes containing these sequences. Use one call of the MPI_File_write_all collective function (for all processes including the process 0) and also the MPI_File_seek function with the MPI_SEEK_SET parameter.

MPI6File11. The name of existing file of real numbers is given in the master process. In addition, an integer is given in each process. This integer is equal to 0 or it is equal to the ordinal number of one of the existing items of the file (the file items are numbered from 1). Using the MPI_Comm_split function, create a new communicator containing only those processes in which a non-zero integer is given. Using the MPI_File_read_at_all collec-

tive function for all processes of this communicator, read and output a file item located at a position with the given ordinal number.

MPI6File12. The name of existing file of real numbers is given in the master process. In addition, an integer is given in each process. This integer is equal to 0 or it is equal to the ordinal number of one of the existing items of the file (the file items are numbered from 1). Using the MPI_Comm_split function, create a new communicator containing only those processes in which a non-zero integer is given. Using the MPI_File_write_at_all collective function for all processes of this communicator, replace the value of the file item, that has the given ordinal number, by the value of the process rank in the new communicator (the rank should be converted to a real number).

MPI6File13. The name of existing file of integers is given in the master process. In addition, an integer is given in each process. This integer is equal to 0 or 1. Using the MPI_Comm_split function, create a new communicator containing only those processes in which the number 1 is given. Using the MPI_File_read_ordered collective function for all processes of this communicator, read and output the $R + 1$ file item, where R is the process rank in the new communicator (items should be read in ascending order: the first item in the process 0, two next items in the process 1, three next items in the process 2, and so on). If the file does not contain enough items, then, in some processes, the number N of output items may be less than required one or even may be equal to zero. In addition, output the number N of actually read items and the new value P of the shared file pointer in each process of the new communicator. Use the MPI_Status parameter of the MPI_File_read_ordered function to determine the number N of actually read items. Use the MPI_File_get_position_shared function to determine the current value of P (this value should be the same in all processes).

MPI6File14. The name of file is given in the master process. In addition, an integer N is given in each process. Create a new file of integers with the given name. Using the MPI_Comm_split function, create a new communicator containing only those processes in which a non-zero integer is given. Using the MPI_File_write_ordered collective function for all processes of the new communicator, write K successive copies of each given integer N to this file, where K is the number of processes in the *new* communicator. The integers N should be written to the file in the ascending order of ranks of processes containing these integers.

MPI6File15. The name of existing file of integers is given in the master process. The file contains at least K items, where K is the number of processes. Using the MPI_Comm_split function, create a new communica-

tor containing only processes with odd rank (1, 3, …). Using one call of the MPI_File_seek_shared and MPI_File_read_ordered collective functions, read and output two file items at a time in each process of the new communicator: the second and the first item from the end (in this order) should be read and output in the process of the rank 1 in the MPI_COMM_WORLD communicator, the fourth and third item from the end should be read and output in the process of the rank 3, and so on.

Note. To ensure the required order of data reading in the MPI_File_read_ordered function, you should inverse the order of the processes in the created communicator (in comparison with the processes in the MPI_COMM_WORLD communicator).

MPI6File16. The name of file is given in the master process. In addition, an integer N (≥ 0) and N real numbers are given in each process. Create a new file of real numbers with the given name. Using the MPI_Comm_split function, create a new communicator containing only those processes in which a non-zero integer N is given. Using one call of the MPI_File_write_ordered collective function for all processes of the new communicator, write the given real numbers to the file in the *inverse* order: at first, all the real numbers from the process with the maximal rank in the communicator MPI_COMM_WORLD should be written (in inverse order), after that, all the numbers from the process with the previous rank, and so on.

Note. To ensure the required order of data writing in the MPI_File_write_ordered function, you should inverse the order of the processes in the created communicator (in comparison with the processes in the MPI_COMM_WORLD communicator).

6.3. File view setting for file input-output

MPI6File17. The name of existing file of integers is given in the master process. The file contains $2K$ items, where K is the number of processes. Using one call of the MPI_File_read_all collective function (and without using the MPI_File_seek function), read and output two file items at a time in each process. The file items should be read and output in the order in which they are stored in a file. To do this, use the MPI_File_set_view function to define a new file view with the MPI_INT elementary datatype, the same filetype, and the appropriate displacement (the displacement will be different for different processes).

MPI6File18. The name of existing file of integers is given in the master process. The file contains elements of the $K \times 5$ matrix, where K is the number of processes. In addition, an integer N ($1 \leq N \leq 5$) is given in each process; this integer determines the ordinal number of a *selected* element in

some matrix row, namely, in the first row for the process 0, in the second row for the process 1, and so on (the row elements are numbered from 1). Using one call of the MPI_File_write_at_all collective function with the second parameter equal to $N - 1$, replace the value of the selected element in each matrix row by the rank of the corresponding process (the selected element in the first row should be replaced by 0, the selected element in the second row should be replaced by 1, and so on). To do this, use the MPI_File_set_view function to define a new file view with the MPI_INT elementary datatype, the same filetype, and the appropriate displacement (the displacement will be different for different processes).

MPI6File19. The name of existing file of real numbers is given in the master process. The file contains elements of the $K \times 6$ matrix, where K is the number of processes. In addition, an integer N ($1 \le N \le 6$) is given in each process; this integer determines the ordinal number of a *selected* element in some matrix row, namely, in the last row for the process 0, in the last but one row for the process 1, and so on (the row elements are numbered from 1). Using one call of the MPI_File_read_at_all collective function with the second parameter equal to $N - 1$, read and output the value of the selected row element in the corresponding process (the selected element in the first row should be output in the last process, the selected element in the second row should be output in the last but one process, and so on). To do this, use the MPI_File_set_view function to define a new file view with the MPI_DOUBLE elementary datatype, the same filetype, and the appropriate displacement (the displacement will be different for different processes).

MPI6File20. The name of file is given in the master process. In addition, a sequence of $R + 1$ real numbers is given in each process, where R is the process rank (0, 1, …). Create a new file of real numbers with the given name. Using one call of the MPI_File_write_all collective function (and without using the MPI_File_seek function), write all given numbers to the file in the order that is inverse to the order in which they are given in processes: at first, the numbers from the last process (in the inverse order) should be written, after that, the numbers from the last but one process (in the inverse order), and so on. To do this, use the MPI_File_set_view function to define a new file view with the MPI_DOUBLE elementary datatype, the same filetype, and the appropriate displacement (the displacement will be different for different processes).

MPI6File21. The name of existing file of integers is given in the master process. The file contains $3K$ items, where K is the number of processes. Read and output three file items, namely, A, B, C, in each process. These items are located in the given file as follows: A_0, A_1, …, A_{K-1}, B_0, B_1, …, B_{K-1}, C_0, C_1, …, C_{K-1} (an index indicates the process rank). To do this, use one call

of the MPI_File_read_all collective function and a new file view with the MPI_INT elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of an integer and a terminal empty space of a size equal to the extent of $K - 1$ integers.

MPI6File22. The name of file is given in the master process. In addition, three integers, namely, *A*, *B*, *C*, are given in each process. The number of processes is equal to *K*. Create a new file of integers with the given name and write the given integers to this file as follows: A_{K-1}, A_{K-2}, …, A_0, B_{K-1}, B_{K-2}, …, B_0, C_{K-1}, C_{K-2}, …, C_0 (an index indicates the process rank). To do this, use one call of the MPI_File_write_all collective function and a new file view with the MPI_INT elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of an integer and a terminal empty space of a size equal to the extent of $K - 1$ integers.

MPI6File23. The name of existing file of real numbers is given in the master process. The file contains $6K$ items, where *K* is the number of processes. Read and output six file items, namely, *A*, *B*, *C*, *D*, *E*, *F*, in each process. These items are located in the given file as follows: A_0, B_0, C_0, A_1, B_1, C_1, …, A_{K-1}, B_{K-1}, C_{K-1}, D_0, E_0, F_0, D_1, E_1, F_1, …, D_{K-1}, E_{K-1}, F_{K-1} (an index indicates the process rank). To do this, use one call of the MPI_File_read_all collective function and a new file view with the MPI_DOUBLE elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of three real numbers and a terminal empty space of the appropriate size.

MPI6File24. The name of file is given in the master process. In addition, four real numbers, namely, *A*, *B*, *C*, *D*, are given in each process. The number of processes is equal to *K*. Create a new file of real numbers with the given name and write the given real numbers to this file as follows: A_{K-1}, B_{K-1}, A_{K-2}, B_{K-2}, …, A_0, B_0, C_{K-1}, D_{K-1}, C_{K-2}, D_{K-2}, …, C_0, D_0 (an index indicates the process rank). To do this, use one call of the MPI_File_write_all collective function and a new file view with the MPI_DOUBLE elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of two real numbers and a terminal empty space of the appropriate size.

MPI6File25. The name of file is given in the master process. In addition, $3 \cdot (R + 1)$ integers are given in the process of rank *R* ($R = 0, 1, \ldots, K - 1$, where *K* is the number of processes): 3 integers, namely, *A*, *B*, *C*, are given in the process 0, 6 integers, namely, *A*, *A*', *B*, *B*', *C*, *C*', are given in the process 1, 9 integers, namely, *A*, *A*', *A*'', *B*, *B*', *B*'', *C*, *C*', *C*'', are given in the process 2, and so on. Create a new file of real numbers with the given

name and write the given integers to this file as follows: A_0, A_1, A'_1, A_2, A'_2, A''_2, …, B_0, B_1, B'_1, B_2, B'_2, B''_2, … (an index indicates the process rank). To do this, use one call of the MPI_File_write_all collective function and a new file view with the MPI_INT elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of $R + 1$ integers and a terminal empty space of the appropriate size.

MPI6File26. The name of file is given in the master process. In addition, four real numbers, namely, *A*, *B*, *C*, *D*, are given in each process. The number of processes is equal to *K*. Create a new file of real numbers with the given name and write the given real numbers to this file as follows: A_0, A_1, …, A_{K-1}, B_{K-1}, …, B_1, B_0, C_0, C_1, …, C_{K-1}, D_{K-1}, …, D_0 (an index indicates the process rank). To do this, use one call of the MPI_File_write_all collective function and a new file view with the MPI_DOUBLE elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of two real numbers (with an additional empty space between these numbers) and a terminal empty space of the appropriate size.

MPI6File27. The name of existing file of real numbers is given in the master process. The file contains elements of the $(K/2) \times K$ matrix, where *K* is the number of processes (*K* is an even number). Read and output elements of $(R + 1)$th matrix column in the process of rank *R* ($R = 0, \ldots, K - 1$, columns are numbered from 1). To do this, use one call of the MPI_File_read_all collective function and a new file view with the MPI_DOUBLE elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of one real number and a terminal empty space of the appropriate size.

MPI6File28. The name of file is given in the master process. In addition, an integer *N* and a sequence of *K*/2 real numbers are given in each process, where *K* is the number of processes (*K* is an even number). The numbers *N* are different for all processes and are in the range from 1 to *K*. Create a new file of real numbers with the given name. Write a $(K/2) \times K$ matrix to this file; each process should write its sequence of real numbers into a column of the matrix with the ordinal number *N* (the columns are numbered from 1). To do this, use one call of the MPI_File_write_all collective function and a new file view with the MPI_DOUBLE elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of one real number and a terminal empty space of the appropriate size.

MPI6File29. The name of existing file of integers is given in the master process. The file contains elements of a block matrix of the size $(K/3) \times 3$

(the size is indicated in blocks), where *K* is the number of processes (*K* is a multiple of 3). Each block is a square matrix of order *N* (all the numbers *N* are the same and are in the range from 2 to 5). Read and output one block of the given matrix in each process in a row-major order of blocks. To do this, use one call of the MPI_File_read_all collective function and a new file view with the MPI_INT elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of *N* integers and a terminal empty space of the appropriate size. Use the MPI_File_get_size function to determine the number *N*.

MPI6File30. The name of file and an integer *N* are given in the master process (*N* is in the range 2 to 5). In addition, two integers *I* and *J* are given in each process. The integers *I* and *J* determine a position (that is, row and column numbers) of some square block of a block matrix of the size (*K*/3) × 3 (the size is indicated in blocks), where *K* is the number of processes (*K* is a multiple of 3). The integers *I* are in the range of 1 to *K*/3, the integers *J* are in the range 1 to 3; all processes contain different positions of blocks. Each block is a square matrix of order *N*. Create a new file of integers with the given name. Write a (*K*/3) × 3 block matrix to this file; each process should write a matrix block to the block position (*I*, *J*). All the elements of the block written by the process of rank *R* (*R* = 0, 1, …, *K* − 1) should be equal to the number *R*. To do this, use one call of the MPI_File_write_all collective function and a new file view with the MPI_INT elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of *N* integers and a terminal empty space of the appropriate size. Use the MPI_Bcast collective function to send the value of *N* to all processes.

7. One-sided communications (MPI-2)

When defining an access window object using the MPI_Win_create function, it is recommended to specify the disp_unit displacement (the third parameter) equal to the size of the data item of the corresponding type (it is always either MPI_INT or MPI_DOUBLE in all tasks, so the size can be obtained using the MPI_Type_get_extent or MPI_Type_size function). In this case one can indicate the target_disp displacement (the fifth parameter of the MPI_Get, MPI_Put, MPI_Accumulate functions) equal to the initial index of the used part of the array (rather than the number of bytes from the beginning of the array to the required part, as in the case of the disp_unit parameter equal to 1).

It is suffice to specify the MPI_INFO_NULL constant as the info parameter (the fourth parameter of the MPI_Win_create function).

If you do not need to create a window in some processes then the value 0 should be specified as the size parameter (the second parameter of the MPI_Win_create function) in these processes.

It is suffice to specify a constant 0 as the assert parameter in all synchronizing functions (MPI_Win_fence, MPI_Win_start, MPI_Win_post, MPI_Win_lock); this parameter is the last but one parameter in all these functions.

In the first subgroup (MPI7Win1–MPI7Win17), you should use the MPI_Win_fence *collective* function as a synchronizing function that should be called both before the actions related to the one-way data transfer and after these actions but before the actions related to access to the transferred data.

The tasks of the second subgroup (MPI7Win18–MPI7Win30) require the use of *local* synchronization: the MPI_Win_start, MPI_Win_complete, MPI_Win_post, MPI_Win_wait functions or a pair of the MPI_Win_lock, MPI_Win_unlock functions. In the tasks of this subgroup, there is always specified which kind of the local synchronization you should use.

7.1. One-sized communications with the simplest synchronization

MPI7Win1. An integer is given in each slave process. Create an access window of the size K of integers in the master process (K is the number of slave processes). Using the MPI_Put function call in the slave processes, send all the given integers to the master process and output received integers in the ascending order of ranks of sending processes.

MPI7Win2. A sequence of R real numbers is given in each slave process, where R is the process rank (1, 2, …). Create an access window of the appropriate size in the master process. Using the MPI_Put function call in the slave processes, send all the given real numbers to the master process and output received numbers in the ascending order of ranks of sending processes.

MPI7Win3. An array A of K integers is given in the master process, where K is the number of slave processes. Create an access window containing the array A in the master process. Using the MPI_Get function call in the slave processes, receive and output one element of the array A in each slave process. Elements of the array A should be received in the slave processes in descending order of their indices (that is, the element A_0 should be received in the last process, the element A_1 should be received in the last but one process, and so on).

MPI7Win4. An array A of $K + 4$ real numbers is given in the master process, where K is the number of slave processes. Create an access window containing the array A in the master process. Using the MPI_Get function call in the slave processes, receive and output five elements of the array A in

each slave process starting with the element of index $R - 1$, where R is the slave process rank ($R = 1, 2, \ldots, K - 1$).

MPI7Win5. An array A of K integers is given in the master process, where K is the number of slave processes. In addition, an index N (an integer in the range 0 to $K - 1$) and an integer B are given in each slave process. Create an access window containing the array A in the master process. Using the MPI_Accumulate function call in the slave processes, multiply the element A_N by the number B and then output the modified array A in the master process.

Note. Some slave processes can contain the same value of N; in this case the element A_N will be multiplied several times. This circumstance does not require additional synchronization due to the features of the MPI_Accumulate function implementation.

MPI7Win6. An array A of $2K - 1$ real numbers is given in the master process (K is the number of slave processes), and array B of R real numbers is given in each slave process (R is the process rank, $R = 1, 2, \ldots, K - 1$). Create an access window containing the array A in the master process. Using the MPI_Accumulate function call in the slave processes, add the values of all the elements of array B from the process of rank R to the elements of array A starting with the index $R - 1$ (the single element B_0 from the process 1 should be added to the element A_0, the elements B_0 and B_1 from the process 2 should be added to the elements A_1 and A_2 respectively, the elements B_0, B_1, and B_2 from the process 3 should be added to elements A_2, A_3, and A_4 respectively, and so on). Output the modified array A in the master process.

Note. Elements of array A, starting from the index 2, will be modified several times by adding values from the different slave processes. This circumstance does not require additional synchronization due to the features of the MPI_Accumulate function implementation.

MPI7Win7. An array A of $2K$ integers is given in the master process, where K is the number of slave processes. Create an access window containing two integers in each slave process. Using the MPI_Put function call in the master process, send and output two elements of the array A in each slave process. Elements of the array A should be sent to slave processes in ascending order of their indices (that is, the elements A_0 and A_1 should be sent to the process 1, the elements A_2 and A_3 should be sent to the process 2, and so on).

MPI7Win8. An integer R and a real number B are given in each process. All the integers R are different and are in the range from 0 to $K - 1$, where K is the number of processes. Create an access window containing one real number in each process. Using the MPI_Put function call in each process, send the

number B from this process to the process R and output received numbers in all processes.

MPI7Win9. An array A of K integers is given in each process, where K is the number of processes. Create an access window containing the array A in each process. Using several calls of the MPI_Get function in each process R ($R = 0, \ldots, K - 1$), receive and output elements of all arrays A with the index R. Received elements should be output in descending order of ranks of sending processes (that is, the element received from the process $K - 1$ should be output first, the element received from the process $K - 2$ should be output second, and so on).

Note. The function MPI_Get, as well as other one-way communication functions, can also be used to access the window created in the calling process.

MPI7Win10. An array A of 5 real numbers and integers N_1 and N_2 are given in each process. Each of the numbers N_1 and N_2 is in the range 0 to 4. Create an access window containing the array A in each process. Using two calls of the MPI_Get function in each process, receive and output the element of index N_1 from the array A of the previous process and then receive and output the element of index N_2 from the array A of the next process (the numbers N_1 and N_2 are taken from the calling process, processes are taken in a cyclic order).

MPI7Win11. The number of processes K is an even number. An array A of $K/2$ integers is given in each process. Create an access window containing the array A in all the odd-rank processes ($1, 3, \ldots, K - 1$). Using the required number of calls of the MPI_Accumulate function in each even-rank process, add the element $A[I]$ of the process $2J$ to the element $A[J]$ of the process $2I + 1$ and output the changed arrays A in all the odd-rank processes.

Note. The required changing of the given arrays can be described in the another way: if B denotes a matrix of order $K/2$ whose rows coincide with the arrays A given in the even-rank processes and C denotes a matrix of the same order whose rows coincide with the arrays A given in the odd-rank processes then the matrix C should be transformed as follows: elements of the row I of the matrix B should be added to the corresponding elements of the column I of the matrix C.

MPI7Win12. Solve the MPI7Win11 task by creating access windows in even-rank processes and using the MPI_Get function calls instead of the MPI_Accumulate function calls in odd-rank processes.

Note. Since the numbers received from the even-rank processes must be added to the elements of array A after the second MPI_Win_fence synchro-

nization function call, it is convenient to use an auxiliary array to store the received numbers.

MPI7Win13. Three integers N_1, N_2, N_3 are given in each process; each given integer is in the range 0 to $K - 1$, where K is the number of processes (the values of some of these integers in each process may coincide). In addition, an array A of $R + 1$ real numbers is given in each process, where R is the process rank (0, …, $K - 1$). Create an access window containing the array A in all the processes. Using three calls of the MPI_Accumulate function in each process, add the integer $R + 1$ to all elements of the arrays A given in the processes N_1, N_2, N_3, where R is the rank of the process that calls the MPI_Accumulate function (for instance, if the number N_1 in the process 3 is equal to 2 then a real number 4.0 should be added to all the elements of array A in the process 2). If some of the integers N_1, N_2, N_3 coincide in the process R then the number $R + 1$ should be added to the elements of the corresponding arrays several times. Output the changed arrays A in each process.

MPI7Win14. An array of K real numbers is given in each process, where K is the number of processes. The given array contains a row of an upper triangular matrix A, inclusive of its zero-valued part (the process of rank R contains the Rth row of the matrix, the rows are numbered from 0). Create an access window containing the given array in all the processes. Using the required number of calls of the MPI_Get function in each process, write the rows of the matrix transposed to the given matrix A (inclusive of its zero-valued part) in the given arrays. Then output the changed arrays in each process. Do not use auxiliary arrays.

Notes. (1) The rows of the transposed matrix coincide with the columns of the original matrix, so the resulting matrix will be the lower triangular one. (2) You should write zero values to the required array elements only after the second call of the MPI_Win_fence function. (3) You can not create an access window for the last process.

MPI7Win15. Solve the MPI7Win14 task by using the MPI_Put function calls instead of the MPI_Get function calls.

Note. In this case, you can not create an access window for the master process.

MPI7Win16. One row of the square real-valued matrix A of order K is given in each process, where K is the number of processes (the process of rank R contains the Rth row of the matrix, the rows are numbered from 0). In addition, a real number B is given in each process. Create an access window containing the given row of the matrix A in all the processes. Using the required number of calls of the MPI_Accumulate function in each process R (R = 0, …, $K - 1$), change the matrix row given in the next process as fol-

lows: all row elements that are less than the number B from the process R should be replaced by this number B (processes are taken in a cyclic order). Then, using K calls of the MPI_Get function in each process, receive and output the Rth column of the transformed matrix A in the process R ($R = 0, \ldots, K - 1$, the columns are numbered from 0).

Note. You should call the MPI_Win_fence synchronization function *three times* in each process.

MPI7Win17. One row of the square real-valued matrix A of order K is given in each process, where K is the number of processes (the process of rank R contains the Rth row of the matrix, the rows are numbered from 0). In addition, a real number B is given in each process. Create an access window containing the given row of the matrix A in all the processes. Using the required number of calls of the MPI_Accumulate function in each process R ($R = 0, \ldots, K - 1$), change the matrix row given in the previous process as follows: all row elements that are greater than the number B from the process R should be replaced by this number B (processes are taken in a cyclic order). Then, using K calls of the MPI_Accumulate function in each slave process, add the first element of the row from each slave process R ($1, \ldots, K - 1$) to all the elements of the Rth column of the transformed matrix A (the columns are numbered from 0). Output the new contents of the given row of the matrix A in each process after all transformations.

Note. You should call the MPI_Win_fence synchronization function *three times* in each process.

7.2. Additional types of synchronization

MPI7Win18. The number of processes K is an even number. An integer A is given in each even-rank process ($0, 2, \ldots, K - 2$). Create an access window containing one integer in all the odd-rank processes ($1, 3, \ldots, K - 1$). Using the MPI_Put function call in each even-rank process $2N$, send the integer A to the process $2N + 1$ and output the received integers. Use the MPI_Win_start and MPI_Win_complete synchronization functions in the even-rank processes and the MPI_Win_post and MPI_Win_wait synchronization functions in the odd-rank processes. Use the MPI_Group_incl function to create a group of processes specified as the first parameter of the MPI_Win_start and MPI_Win_post functions. The MPI_Group_incl function should be applied to the group of the MPI_COMM_WORLD communicator (use the MPI_Comm_group function to obtain the group of the MPI_COMM_WORLD communicator).

Note. Unlike the MPI_Win_fence *collective* synchronization function, used in previous tasks, the synchronization functions used in this and the subsequent tasks are *local* ones and, in addition, allow to specify the groups of origin and target processes for one-way communications.

MPI7Win19. An array A of K real numbers is given in the master process, where K is the number of slave processes. Create an access window containing the array A in the master process. Using the MPI_Get function call in each slave process, receive and output one of elements of the array A. The elements should be received in descending order of their indices (that is, the element with the index $K - 1$ should be received in the process 1, the element with the index $K - 2$ should be received in the process 2, and so on). Use the MPI_Win_start and MPI_Win_complete synchronization functions in the slave processes and the MPI_Win_post and MPI_Win_wait synchronization functions in the master process. Use the MPI_Group_incl function to create a group of processes specified as the first parameter of the MPI_Win_start function, use the MPI_Group_excl function to create a group of processes specified as the first parameter of the MPI_Win_post function. The MPI_Group_incl and MPI_Group_excl functions should be applied to the group of the MPI_COMM_WORLD communicator.

MPI7Win20. The number of processes K is a multiple of 3. An array A of 3 real numbers is given in the processes of rank $3N$ ($N = 0, \ldots, K/3 - 1$). Create an access window containing the array A in all processes in which this array is given. Using one call of the MPI_Get function in the processes of rank $3N + 1$ and $3N + 2$ ($N = 0, \ldots, K/3 - 1$), receive and output one element A_0 and two elements A_1, A_2 respectively from the process $3N$ (namely, the process 1 should output the element A_0 received from the process 0, the process 2 should output the elements A_1 and A_2 received from the process 0, the process 4 should output the element A_0 received from the process 3, and so on). Use the MPI_Win_post and MPI_Win_wait synchronization functions in the processes of rank $3N$ and the MPI_Win_start and MPI_Win_complete synchronization functions in the other processes.

MPI7Win21. The number of processes K is an even number. An array A of $K/2$ real numbers and an array N of $K/2$ integers are given in the master process. All the elements of the array N are distinct and are in the range 1 to $K - 1$. Create an access window containing one real number in each slave process. Using the required number of calls of the MPI_Put function in the master process, send the real number A_I to the slave process of rank N_I ($I = 0, \ldots, K/2 - 1$). Output the received number (or 0.0 if the process did not receive data) in each slave process. Use the MPI_Win_post and MPI_Win_wait synchronization functions in the slave processes and the MPI_Win_start and MPI_Win_complete synchronization functions in the master process.

MPI7Win22. An array A of K real numbers (where K is the number of slave processes) and an array N of 8 integers are given in the master process. All the elements of the array N are in the range 1 to K; some elements of this array may have the same value. In addition, an array B of R real numbers is

given in the slave process of rank R ($R = 1, \ldots, K$). Create an access window containing the array B in each slave process. Using the required number of calls of the MPI_Accumulate function in the master process, add all the elements of the array A to the corresponding elements of the array B from the process of rank N_I, $I = 0, \ldots, 7$ (that is, the element A_0 should be added to the element B_0, the element A_1 should be added to the element B_1, and so on). Elements of the array A can be added several times to some arrays B. Output the array B (which may be changed or not) in each slave process. Use the MPI_Win_post and MPI_Win_wait synchronization functions in the slave processes and the MPI_Win_start and MPI_Win_complete synchronization functions in the master process.

MPI7Win23. An array A of 5 real numbers is given in each process. In addition, two arrays N and M of 5 integers are given in the master process. All the elements of the array N are in the range 1 to K, where K is the number of slave processes, all the elements of the array M are in the range 0 to 4. Some elements of both the array N and the array M may have the same value. Create an access window containing the array A in each slave process. Using the required number of calls of the MPI_Get function in the master process, receive the element of A with the index M_I from the process N_I ($I = 0, \ldots, 4$) and add the received element to the element A_I in the master process. After changing the array A in the master process, change all the arrays A in the slave processes as follows: if some element of the array A from the slave process is greater than the element, with the same index, of the array A from the main process then replace this element in the slave process by the corresponding element from the master process (to do this, use the required number of calls of the MPI_Accumulate function in the master process). Output the changed arrays A in each process. Use two calls of the MPI_Win_post and MPI_Win_wait synchronization functions in the slave processes and two calls of the MPI_Win_start and MPI_Win_complete synchronization functions in the master process.

MPI7Win24. An integer N is given in each slave process, all the integers N are distinct and are in the range 0 to $K - 1$, where K is the number of slave processes. Create an access window containing an array A of K integers in each slave process. Without performing any synchronization function calls in the master process (except calling the MPI_Barrier function) and using a sequence of calls of the MPI_Win_lock, MPI_Win_unlock, MPI_Barrier, MPI_Win_lock, MPI_Win_unlock synchronization functions in the slave processes, change element of the array A with index N by assigning the rank of the slave process, which contains the integer N, to this element (to do this, use the MPI_Put function) and then receive and output all the elements of the changed array A in each slave process (to do this, use the

MPI_Get function). Use the MPI_LOCK_SHARED constant as the first parameter of the MPI_Win_lock function.

Note. The MPI_Win_lock and MPI_Win_unlock synchronization functions are used mainly for one-way communications with *passive targets*. In such a kind of one-way communications, the target process does not process the data transferred to it but acts as their storage, which is accessible to other processes.

MPI7Win25. The number of processes K is a multiple of 3. An array A of 5 real numbers is given in the processes of rank $3N$ ($N = 0, \ldots, K/3 - 1$), an integer M and a real number B are given in the processes of rank $3N + 1$. The given integers M are in the range 0 to 4. Create an access window containing the array A in all processes in which this array is given. Using the MPI_Accumulate function call in the processes of rank $3N + 1$ ($N = 0, \ldots, K/3 - 1$), change the array A from the process $3N$ as follows: if the array element with the index M is greater than the number B then this element should be replaced by the number B (the numbers M and B are taken from the process $3N + 1$). Then send the changed array A from the process $3N$ to the process $3N + 2$ and output the received array in the process $3N + 2$; to do this, use the MPI_Get function call in the process of rank $3N + 2$. Use the MPI_Win_lock, MPI_Win_unlock, MPI_Barrier synchronization functions in the processes of rank $3N + 1$, the MPI_Barrier, MPI_Win_lock, MPI_Win_unlock synchronization functions in the processes of rank $3N + 2$, and the MPI_Barrier function in the processes of rank $3N$. Use the MPI_LOCK_EXCLUSIVE constant as the first parameter of the MPI_Win_lock function.

MPI7Win26. An array A of 5 positive real numbers is given in each slave process. Create an access window containing an array B of 5 zero-valued real numbers in the master process. Without performing any synchronization function calls in the master process (except calling the MPI_Barrier function) and using a sequence of calls of the MPI_Win_lock, MPI_Win_unlock, MPI_Barrier, MPI_Win_lock, MPI_Win_unlock synchronization functions in the slave processes, change elements of the array B by assigning the maximal value of the array A elements with the index I ($I = 0, \ldots, 4$) to the array B element with the same index (to do this, use the MPI_Accumulate function) and then receive and output all the elements of the changed array B in each slave process (to do this, use the MPI_Get function). Use the MPI_LOCK_SHARED constant as the first parameter of the MPI_Win_lock function.

MPI7Win27. Two real numbers X, Y (the coordinates of a some point on a plane) are given in each slave process. Using the MPI_Get function in the master process, receive real numbers X_0, Y_0 in this process that are equal to

the coordinates of the point that is the most remote from the origin among all the points given in the slave processes. Then send the numbers X_0, Y_0 from the master process to all the slave processes and output these numbers in the slave processes; to do this, use the MPI_Get function call in the slave processes. Use the MPI_Win_lock, MPI_Win_unlock, MPI_Barrier synchronization functions in the master process and the MPI_Barrier, MPI_Win_lock, MPI_Win_unlock synchronization functions in the slave processes.

Note. This task can not be solved by using one-way communications only on the side of the slave processes by means of the lock/unlock synchronizations.

MPI7Win28. Solve the MPI7Win27 task using the single access window containing the numbers X_0, Y_0 in the master process. Use the MPI_Get and MPI_Put functions in the slave processes to find the numbers X_0, Y_0 (for some processes, the MPI_Put function is not required), use the MPI_Get function to send the numbers X0, Y0 to all the slave processes (as in the MPI7Win27 task). To synchronize exchanges when find the numbers X_0, Y_0, use two calls of each of the MPI_Win_start and MPI_Win_complete functions in the slave processes and calls of the MPI_Win_post and MPI_Win_wait functions *in a loop* in the master process (it is necessary to define *a new group of processes* at each iteration of the loop; this group should be used in the MPI_Win_post function call). To synchronize sending numbers X_0, Y_0 to slave processes, use the MPI_Barrier function in the master process and the MPI_Barrier, MPI_Win_Lock, MPI_Win_unlock functions in the slave processes (as in the MPI7Win27 task).

Note. The solution method described in this task allows one-way communications to be used only on the side of the slave processes (in contrast to the method described in the MPI7Win27 task) but it requires to apply a synchronizations that different from the lock/unlock ones.

MPI7Win29. One row of the square integer-valued matrix of order K is given in each process, where K is the number of processes (the process of rank R contains the Rth row of the matrix, the rows are numbered from 0). Using the MPI_Get function calls in the master process, receive a matrix row with the minimal sum S of elements in this process and also find the number N of matrix rows with this minimal sum (if $N > 1$ then the last of such rows, that is, the row with the maximal ordinal number, should be saved in the master process). Then send this matrix row, the sum S, and the number N to each slave process using the MPI_Get function in these processes. Output all received data in each process. To do this, create an access window containing $K + 2$ integers in each process; the first K elements of the window should contain the elements of the matrix row, the next element should contain the sum S of its elements, and the last element should contain the

number *N*. Use the MPI_Win_lock, MPI_Win_unlock, MPI_Barrier synchronization functions in the master process and the MPI_Barrier, MPI_Win_lock, MPI_Win_unlock synchronization functions in the slave processes.

Note. This task can not be solved by using one-way communications only on the side of the slave processes by means of the lock/unlock synchronizations.

MPI7Win30. Solve the MPI7Win29 task using the single access window containing the matrix row and the numbers *S* and *N* in the master process. Use the MPI_Get and MPI_Put functions in the slave processes to find the matrix row with the minimal sum and the related numbers *S* and *N* (for some processes, the MPI_Put function is not required), use the MPI_Get function to send the row with the minimal sum and the numbers *S* and *N* to all the slave processes (as in the MPI7Win29 task). To synchronize exchanges when find the matrix row, use two calls of each of the MPI_Win_start and MPI_Win_complete functions in the slave processes and calls of the MPI_Win_post and MPI_Win_wait functions *in a loop* in the master process (it is necessary to define *a new group of processes* at each iteration of the loop; this group should be used in the MPI_Win_post function call). To synchronize sending the row with the minimal sum and the numbers *S* and *N* to slave processes, use the MPI_Barrier function in the master process and the MPI_Barrier, MPI_Win_Lock, MPI_Win_unlock functions in the slave processes (as in the MPI7Win29 task).

Note. The solution method described in this task allows one-way communications to be used only on the side of the slave processes (in contrast to the method described in the MPI7Win29 task) but it requires to apply a synchronizations that different from the lock/unlock ones.

8. Inter-communicators and process creation

The basic tools for creation of inter-communicators and their use for point-to-point communication are defined in the MPI-1 standard. Therefore, 5 tasks of this group (MPI8Inter1–MPI8Inter4 and MPI8Inter9) can be solved using the MPICH 1.2.5 system. Other tasks are devoted to the new functions for inter-communicator creation (MPI8Inter5–MPI8Inter8), to the collective communications via inter-communicators (MPI8Inter10–MPI8Inter14) and to use inter-communicators for process creation (MPI8Inter15–MPI8Inter22). All these features have appeared in the MPI-2 standard, so you should use the MPICH2 1.3 system to solve these tasks.

You should use a *copy* of the communicator MPI_COMM_WORLD as the *peer* communicator (the third parameter of the MPI_Intercomm_create function). Use the MPI_Comm_dup function to create this copy.

The parameters of the MPI_Comm_spawn function, which is used for process creation in then MPI8Inter15–MPI8Inter22 tasks, should be as follows: the first parameter should be the name of the executable file ptprj.exe, the second parameter argv is enough to specify the NULL constant, the fourth parameter info should be the MPI_NULL_INFO constant, the last parameter array_of_errcodes should be the MPI_ERRCODES_IGNORE constant. If the task does not specify the source communicator for the process creation then it is assumed that this communicator should be the MPI_COMM_WORLD one.

Instead of the string "ptprj.exe", you can use the function char* GetExename() that is implemented in the Programming Taskbook and returns the full name of the executable file.

8.1. Inter-communicator creation

MPI8Inter1. The number of processes K is an even number. An integer X is given in each process. Using the MPI_Comm_group, MPI_Group_range_incl, and MPI_Comm_create functions, create two communicators: the first one contains the even-rank processes in the same order (0, 2, …, $K/2 - 2$), the second one contains the odd-rank processes in the same order (1, 3, …, $K/2 - 1$). Output the ranks R of the processes included in these communicators. Then combine these communicators into an inter-communicator using the MPI_Intercomm_create function. Using the MPI_Send and MPI_Recv functions for this inter-communicator, send the integer X from each process to the process with the same rank from the other group of the inter-communicator and output the received integers.

MPI8Inter2. The number of processes K is an even number. An integer C and a real number X are given in each process. The numbers C are equal to 0 or 1, the amount of integers 1 is equal to the amount of integers 0. The integer C is equal to 0 in the process of rank 0 and is equal to 1 in the process of rank $K - 1$. Using one call of the MPI_Comm_split function, create two communicators: the first one contains processes with $C = 0$ (in the same order) and the second one contains processes with $C = 1$ (in the inverse order). Output the ranks R of the processes included in these communicators (note that the first and the last processes of the MPI_COMM_WORLD communicator will receive the value $R = 0$). Then combine these communicators into an inter-communicator using the MPI_Intercomm_create function. Using the MPI_Send and MPI_Recv functions for this inter-communicator, send the real number X from each process to the process with the same rank from the other group of the inter-communicator and output the received numbers.

MPI8Inter3. The number of processes K is a multiple of 3. A real number X is given in the processes of rank $3N$ ($N = 0, \ldots, 3K - 3$), real numbers X and Y

are given in the processes of rank $3N + 1$, a real number Y is given in the processes of rank $3N + 2$. Using the MPI_Comm_group, MPI_Group_range_incl, and MPI_Comm_create functions, create three communicators: the first one contains processes of rank $3N$ in the same order (0, 3, …, $K - 3$), the second one contains processes of rank $3N + 1$ in the inverse order ($K - 2$, $K - 5$, …, 1), the third one contains processes of rank $3N + 2$ in the same order (2, 5, …, $K - 1$). Output the ranks R of the processes included in these communicators. Then combine these communicators into two inter-communicators using the MPI_Intercomm_create function. The first inter-communicator contains the first and second group of processes, the second one contains the second and third group of processes. Using the MPI_Send and MPI_Recv functions for these inter-communicators, exchange the numbers X in the processes with the same rank in the first and second group and the numbers Y in the processes with the same rank in the second and third group. Output the received number in each process.

Note. The MPI_Intercomm_create function should be called once for processes of the first and third groups, and twice for processes of the second group, and this number of calls should be performed for the MPI_Send and MPI_Recv functions.

MPI8Inter4. The number of processes K is a multiple of 3. Three integers are given in each process. The first integer (named C) is in the range 0 to 2, the amount of each value 0, 1, 2 is equal to $K/3$, processes 0, 1, 2 contain C with the value 0, 1, 2 respectively. Using one call of the MPI_Comm_split function, create three communicators: the first one contains processes with $C = 0$ (in the same order), the second one contains processes with $C = 1$ (in the same order), the third one contains processes with $C = 2$ (in the same order). Output the ranks R of the processes included in these communicators (note that the processes 0, 1, 2 of the MPI_COMM_WORLD communicator will receive the value $R = 0$). Then combine these communicators into three inter-communicators using two calls of the MPI_Intercomm_create function in each process. The first inter-communicator contains groups of processes with C equal to 0 and 1, the second one contains groups of processes with C equal to 1 and 2, the third one contains groups of processes with C equal to 0 and 2 (thus, the created inter-communicators will form a *ring* connecting all three previously created groups). Denoting two next given integers in the first group as X and Y, in the second group as Y and Z, and in the third group as Z and X (in this order) and using two calls of the MPI_Send and MPI_Recv functions for these inter-communicators, exchange the numbers X in the processes with the same rank in the first and second group, the numbers Y in the processes with the same rank in the second and third group, and the num-

bers Z in the processes with the same rank in the first and third group. Output the received numbers in each process.

MPI8Inter5. The number of processes K is a multiple of 4. An integer X is given in each process. Using the MPI_Comm_group, MPI_Group_range_incl and MPI_Comm_create function, create two communicators: the first one contains the first half of the processes (of rank 0, 1, …, $K/2 - 1$ in this order), the second one contains the second half of the processes (of rank $K/2$, $K/2 + 1$, …, $K - 1$ in this order). Output the ranks R_1 of the processes included in these communicators. Then combine these communicators into an inter-communicator using the MPI_Intercomm_create function. Using the MPI_Comm_create function for this inter-communicator, create a new inter-communicator whose the first group contains the even-rank processes of the first group of the initial inter-communicator (in the same order) and the second group contains the odd-rank processes of the second group of the initial inter-communicator (in the inverse order). Thus, the first and second groups of the new inter-communicator will include the processes of the MPI_COMM_WORLD communicator with ranks 0, 2, …, $K/2 - 2$ and $K - 1$, $K - 3$, …, $K/2 + 1$ respectively. Output the ranks R_2 of the processes included in the new inter-communicator. Using the MPI_Send and MPI_Recv functions for the new inter-communicator, send the integer X from each process to the process with the same rank from the other group of the inter-communicator and output the received numbers.

MPI8Inter6. The number of processes K is a multiple of 4. A real number X is given in each process. Using the MPI_Comm_group, MPI_Group_range_incl and MPI_Comm_create function, create two communicators: the first one contains the first half of the processes (of rank 0, 1, …, $K/2 - 1$ in this order), the second one contains the second half of the processes (of rank $K/2$, $K/2 + 1$, …, $K - 1$ in this order). Output the ranks R_1 of the processes included in these communicators. Then combine these communicators into an inter-communicator using the MPI_Intercomm_create function. Using one call of the MPI_Comm_split function for this inter-communicator, create two new inter-communicators: the first one contains the even-rank processes of the initial inter-communicator, the second one contains the odd-rank processes of the initial intercommunicator; the processes of the second group of each new inter-communicator should be in the inverse order. Thus, the first new communicator will include groups of the processes of the MPI_COMM_WORLD communicator with ranks 0, 2, …, $K/2 - 2$ and $K - 2$, $K - 4$, …, $K/2$, the first new communicator will include groups of the processes of the MPI_COMM_WORLD communicator with ranks 1, 3, …, $K/2 - 1$ and $K - 1$, $K - 3$, …, $K/2 + 1$. Output the ranks R_2 of the processes included in the new inter-communicators. Using the MPI_Send and MPI_Recv func-

tions for the new inter-communicators, send the integer X from each process to the process with the same rank from the other group of this inter-communicator and output the received numbers.

MPI8Inter7. The number of processes K is an even number. An integer C is given in each process. The numbers C are equal to 0 or 1. A single value of $C = 1$ is given in the first half of the processes, the number of values of $C = 1$ is greater than one in the second half of the processes and, in addition, there is at least one value $C = 0$ in the second half of the processes. Using the MPI_Comm_split function, create two communicators: the first one contains the first half of the processes (of rank 0, 1, …, $K/2 - 1$ in this order), the second one contains the second half of the processes (of rank $K/2$, $K/2 + 1$, …, $K - 1$ in this order). Output the ranks R_1 of the processes included in these communicators. Then combine these communicators into an inter-communicator using the MPI_Intercomm_create function. Using the MPI_Comm_split function for this inter-communicator, create a new inter-communicator with groups which contain processes from the corresponding groups of the initial inter-communicator with the values $C = 1$ (in the inverse order). Thus, the first group of the new inter-communicator will include a single process, and the number of processes in the second group will be in the range 2 to $K/2 - 1$. Output the ranks R_2 of the processes that are included in the second group of the new inter-communicator (this group contains more than one process). Input an array Y of K_2 integers in the single process of the first group of the new inter-communicator, where K_2 is the number of the processes in the second group. Input an integer X in each process of the second group of the new inter-communicator. Using the required number of calls of the MPI_Send and MPI_Recv functions for all the processes of the new inter-communicator, send all the integers X to the single process of the first group and send the element of the array Y with the index R_2 to the process R_2 of the second group ($R_2 = 0, 1, \ldots, K_2 - 1$). Output all received numbers (the integers X should be output in ascending order of ranks of sending processes).

Note. In the MPICH 2 version 1.3, the MPI_Comm_split function call for some inter-communicator is erroneous if some of the values of its color parameter are equal to MPI_UNDEFINED. Thus, you should use only *non-negative* values of color in this situation. In addition, the program can behave incorrectly if the MPI_Comm_split function create *empty groups* for some inter-communicators (this is possible if the same color values are specified for all processes of one of the groups of the initial inter-communicator and these color values are different from color values for some processes of the other group).

MPI8Inter8. An integer C is given in each process. The integer C is in the range 0 to 2, all the values of C (0, 1, 2) are given for the even-rank processes and

for the odd-rank processes. Using one call of the MPI_Comm_split function, create two communicators: the first one contains the even-rank processes (in ascending order of ranks), the second one contains the odd-rank processes (in ascending order of ranks). Output the ranks R_1 of the processes included in these communicators. Then combine these communicators into an inter-communicator using the MPI_Intercomm_create function. Using one call of the MPI_Comm_split function for this inter-communicator, create three new inter-communicators with groups which contain processes from the corresponding groups of the initial inter-communicator with the same values of C (in the same order). Thus, for instance, the first group of the first new inter-communicator will include the even-rank processes with $C = 0$ and the second group of the third new inter-communicator will include the odd-rank processes with $C = 2$. Output the ranks R_2 of the processes included in the new inter-communicators. Input an integer X in the processes of the first group of each new inter-communicator, input an integer Y in the processes of the second group of each new inter-communicator. Using the required number of calls of the MPI_Send and MPI_Recv functions for all the processes of all the new inter-communicators, send all the integers X to each process of the second group of the same inter-communicator and send all the integers Y to each process of the first group of the same inter-communicator. Output all received numbers in ascending order of ranks of sending processes.

MPI8Inter9. The number of processes K is an even number. An integer C is given in each process. The integer C is in the range 0 to 2, the first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K/2$. Using the MPI_Comm_split function, create two communicators: the first one contains processes with $C = 1$ (in the same order), the second one contains processes with $C = 2$ (in the same order). Output the ranks R of the processes included in these communicators (output the integer -1 if the process is not included into the created communicators). Then combine these communicators into an inter-communicator using the MPI_Intercomm_create function. A group containing processes with $C = 1$ is considered to be the first group of the created inter-communicator and the group of processes with $C = 2$ is considered to be its second group. Input an integer X in the processes of the first group, input an integer Y in the processes of the second group. Using the required number of calls of the MPI_Send and MPI_Recv functions for all the processes of the inter-communicator, send all the integers X to each process of the second group and send all the integers Y to each process of the first group. Output all received numbers in ascending order of ranks of sending processes.

8.2. Collective communications for inter-communicators

MPI8Inter10. The number of processes K is an even number. An integer C is given in each process. The integer C is in the range 0 to 2, the first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K/2$. Using the MPI_Comm_split function, create two communicators: the first one contains processes with $C = 1$ (in the same order), the second one contains processes with $C = 2$ (in the same order). Output the ranks R of the processes included in these communicators (output the integer −1 if the process is not included into the created communicators). Then combine these communicators into an inter-communicator using the MPI_Intercomm_create function. A group containing processes with $C = 1$ is considered to be the first group of the created inter-communicator and the group of processes with $C = 2$ is considered to be its second group. Input integers R_1 and R_2 in each process of the inter-communicator. The values of the numbers R_1 coincide in all processes and indicate the rank of the selected process of the first group; the values of the numbers R_2 also coincide in all processes and indicate the rank of the selected process of the second group. A sequence of three integers X is given in the selected process of the first group, a sequence of three integers Y is given in the selected process of the second group. Using two calls of the MPI_Bcast collective function in each process of the inter-communicator, send the numbers X to all the processes of the second group, send the numbers Y to all the processes of the first group, and output the received numbers.

MPI8Inter11. The number of processes K is an even number. An integer C is given in each process. The integer C is in the range 0 to 2, the first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K/2$. Using the MPI_Comm_split function, create two communicators: the first one contains processes with $C = 1$ (in the same order), the second one contains processes with $C = 2$ (in the same order). Output the ranks R of the processes included in these communicators (output the integer −1 if the process is not included into the created communicators). Then combine these communicators into an inter-communicator using the MPI_Intercomm_create function. A group containing processes with $C = 1$ is considered to be the first group of the created inter-communicator and the group of processes with $C = 2$ is considered to be its second group. Input an integer R_1 in each process of the inter-communicator. The values of the number R_1 coincide in all processes and indicate the rank of the selected process of the first group. An array X of K_2 integers is given in the selected process of the first group, where K_2 is the number of processes in the second group. Using one call of the MPI_Scatter collective function in each

process of the inter-communicator, send the element $X[R_2]$ to the process R_2 of the second group ($R_2 = 0, \ldots, K_2 - 1$) and output the received numbers.

MPI8Inter12. The number of processes K is an even number. An integer C is given in each process. The integer C is in the range 0 to 2, the first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K/2$. Using the MPI_Comm_split function, create two communicators: the first one contains processes with $C = 1$ (in the same order), the second one contains processes with $C = 2$ (in the same order). Output the ranks R of the processes included in these communicators (output the integer −1 if the process is not included into the created communicators). Then combine these communicators into an inter-communicator using the MPI_Intercomm_create function. A group containing processes with $C = 1$ is considered to be the first group of the created inter-communicator and the group of processes with $C = 2$ is considered to be its second group. Input an integer R_2 in each process of the inter-communicator. The values of the number R_2 coincide in all processes and indicate the rank of the selected process of the second group. An integer X is given in all the processes of the first group. Using one call of the MPI_Gather collective function in each process of the inter-communicator, send all the integers X to the selected process of the second group. Output the received numbers in this process in ascending order of ranks of sending processes.

MPI8Inter13. The number of processes K is an even number. An integer C is given in each process. The integer C is in the range 0 to 2, the first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K/2$. Using the MPI_Comm_split function, create two communicators: the first one contains processes with $C = 1$ (in the same order), the second one contains processes with $C = 2$ (in the same order). Output the ranks R of the processes included in these communicators (output the integer −1 if the process is not included into the created communicators). Then combine these communicators into an inter-communicator using the MPI_Intercomm_create function. A group containing processes with $C = 1$ is considered to be the first group of the created inter-communicator and the group of processes with $C = 2$ is considered to be its second group. An integer X is given in each process of the first group, an integer Y is given in each process of the second group. Using one call of the MPI_Allreduce collective function in each process of the inter-communicator, receive the number Y_{min} in each process of the first group and the number X_{max} in each process of the second group, where the number Y_{min} is the minimal value of the given integers Y and the number X_{max} is the maximal value of the given integers X. Output the received numbers.

MPI8Inter14. The number of processes K is an even number. An integer C is given in each process. The integer C is in the range 0 to 2, the first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K - 1$. Using the MPI_Comm_split function, create two communicators: the first one contains processes with $C = 1$ (in the same order), the second one contains processes with $C = 2$ (in the inverse order). Output the ranks R of the processes included in these communicators (output the integer −1 if the process is not included into the created communicators). Then combine these communicators into an inter-communicator using the MPI_Intercomm_create function. A group containing processes with $C = 1$ is considered to be the first group of the created inter-communicator and the group of processes with $C = 2$ is considered to be its second group. An array X of K_2 integers is given in each process of the first group, where K_2 is the number of processes in the second group; an array Y of K_1 integers is given in each process of the second group, where K_1 is the number of processes in the first group. Using one call of the MPI_Alltoall collective function in each process of the inter-communicator, send the element $Y[R_1]$ of each array Y to the process R_1 of the first group ($R_1 = 0, \ldots, K_1 - 1$) and send the element $X[R_2]$ of each array X to the process R_2 of the second group ($R_2 = 0, \ldots, K_2 - 1$). Output the received numbers in ascending order of ranks of sending processes.

8.3. Process creation

MPI8Inter15. A real number is given in each process. Using the MPI_Comm_spawn function with the first parameter "ptprj.exe", create one new process. Using the MPI_Reduce collective function, send the sum of the given numbers to the new process. Output the received sum in the debug section using the Show function in the new process. Then, using the MPI_Bcast collective function, send this sum to the initial processes and output it in each process.

MPI8Inter16. An array A of K real numbers is given in each process, where K is the number of processes. Using one call of the MPI_Comm_spawn function with the first parameter "ptprj.exe", create K new processes. Using the MPI_Reduce_scatter_block collective function, send the maximal value of the elements $A[R]$ of the given arrays to the new process of rank R ($R = 0, \ldots, K - 1$). Output the received maximal value in the debug section using the Show function in each new process. Then, using the MPI_Send and MPI_Recv functions, send the maximal value from the new process of rank R ($R = 0, \ldots, K - 1$) to the initial process of the same rank and output the received numbers in the initial processes.

MPI8Inter17. The number of processes K is an even number. Arrays of $K/2$ real numbers are given in the processes of rank 0 and 1. Using one call of the MPI_Comm_spawn function with the first parameter "ptprj.exe", create two new processes. Using one call of the MPI_Comm_split function for the inter-communicator connected with the new processes, create two new inter-communicators: the first one contains the group of even-rank initial processes (0, …, $K - 2$) and the new process of rank 0 as the second group, the second one contains the group of odd-rank initial processes (1, …, $K - 1$) and the new process of rank 1 as the second group. Using the MPI_Send function in the initial processes and the MPI_Recv function in the new processes, send all the given numbers from the first process of the first group of each inter-communicator to the single process of its second group. Output the received numbers in the debug section using the Show function in the new processes. Then, using the MPI_Scatter collective function for inter-communicators, send one number from the new process to each process of the first group of the corresponding inter-communicator (in ascending order of ranks of receiving processes) and output the received numbers.

MPI8Inter18. The number of processes K is an even number. Arrays A of $K/2$ real numbers are given in each process. Using one call of the MPI_Comm_spawn function with the first parameter "ptprj.exe", create K new processes. Using one call of the MPI_Comm_split function for the inter-communicator connected with the new processes, create two new inter-communicators: the first one contains the group of even-rank initial processes (0, …, $K - 2$) and the even-rank new processes as the second group, the second one contains the group of odd-rank initial processes (1, …, $K - 1$) and the odd-rank new processes as the second group. Perform the following actions for each created inter-communicator: (1) find the minimal value (for the first inter-communicator) or the maximal value (for the second one) of the elements $A[R]$ ($R = 0, \ldots, K/2 - 1$) of all the arrays A given in the first group of this inter-communicator; (2) send the found value to the new process of rank R in the second group of the corresponding inter-communicator. For instance, the minimal of the first elements of the arrays given in the even-rank initial processes should be sent to the first of the new processes, the maximal of the first elements of the arrays given in the odd-rank initial processes should be sent to the second of the new processes (since this process has rank 0 in the corresponding inter-communicator). To do this, use the MPI_Reduce_scatter_block collective function. Output the received values in the debug section using the Show function in each new process. Then, using the MPI_Reduce collective function, find the minimum of the values received in the second group of the first inter-communicator, send the found minimum to the first process of

the first group of this inter-communicator (that is, to the process 0 in the MPI_COMM_WORLD communicator), and output the received minimum. Also, find the maximum of the values received in the second group of the second inter-communicator, send the found maximum to the first process of the first group of this inter-communicator (that is, to the process 1 in the MPI_COMM_WORLD communicator), and output the received maximum.

MPI8Inter19. An array A of $2K$ integers is given in the master process, where K is the number of processes. Using one call of the MPI_Comm_spawn function with the first parameter "ptprj.exe", create K new processes. Using the MPI_Intercomm_merge function for the inter-communicator connected with the new processes, create a new intra-communicator which include both the initial and the new processes. The order of the processes in the new intra-communicator should be as follows: the initial processes, then the new ones (to specify this order, use the appropriate value of the parameter high of the MPI_Intercomm_merge function). Using the MPI_Scatter collective function for the new intra-communicator, send the element $A[R]$ of the array A to the process of rank R in this intra-communicator ($R = 0, \ldots, 2K - 1$). Output the numbers received in the initial processes in the section of results, output the numbers received in the new processes in the debug section using the Show function. Then, using the MPI_Reduce collective function in this intra-communicator, find and output the sum of all numbers in the process of rank 1 in this intra-communicator.

MPI8Inter20. The number of processes K is *not* a multiple of 4. An integer A is given in each process. Using one call of the MPI_Comm_spawn function with the first parameter "ptprj.exe", create such a number of new processes (1, 2 or 3) that the total number of processes K_0 in the application would be a multiple of 4. Define an integer A equal to $-R - 1$ in each new process, where R is the process rank. Using the MPI_Intercomm_merge function for the inter-communicator connected with the new processes, create a new intra-communicator which include both initial and new processes. The order of the processes in the new intra-communicator should be as follows: the initial processes, then the new ones (to specify this order, use the appropriate value of the parameter high of the MPI_Intercomm_merge function). Using the MPI_Cart_create function for the new intra-communicator, define a Cartesian topology for all processes as a two-dimensional ($K_0/4 \times 4$) grid, which is periodic in the second dimension (ranks of processes should not be reordered). Find the process coordinates in the created topology using the MPI_Cart_coords function. Output the coordinates found in the initial processes in the section of results, output the coordinates found in the new processes in the debug section with the "X = " and "Y = " comments using the Show function. Using the MPI_Comm_shift and

MPI_Sendrecv_replace functions, perform a cyclic shift of the integers *A* given in all processes of each column of the grid by step −1 (that is, the number *A* should be sent from each process in the column, with the exception of the first process, to the previous process in the same column and from the first process in the column to the last process in the same column). Output the integers *A* received in the initial processes in the section of results, output the integers *A* received in the new processes in the debug section with the "A = " comment using the Show function.

MPI8Inter21. A real number is given in each process; this number is denoted by the letter *A* in the master process and by the letter *B* in the slave processes. Using two calls of the MPI_Comm_spawn function with the first parameter "ptprj.exe", create two groups of new processes as follows: the first group (named the *server* group) should include one process, the second group (named the *client* group) should include $K - 1$ processes, where K is the number of initial processes. Send the number *A* from the master process to the single new process of the server group, send the number *B* from each slave process to the corresponding new process of the client group (in ascending order of the process ranks). Output the number received in each new process in the debug section using the Show function. Using the MPI_Open_port, MPI_Publish_name, and MPI_Comm_accept functions on the server side and the MPI_Lookup_name and MPI_Comm_connect functions in the client side, establish a connection between two new groups of processes by means of a new inter-communicator. Using the MPI_Send and MPI_Recv functions for this inter-communicator, receive the number *A* in each process of the client group from the process of the server group. Found the sum of the received number *A* and the number *B*, which is received earlier from the initial slave process, and output the sum $A + B$ in the debug section using the Show function in each process of the client group. Send this sum to the corresponding initial slave process and output the received sum in this process (the sum found in the process of rank R of the client group should be sent to the initial process of rank $R + 1$).

Note. The MPI_Lookup_name function call in the client processes should be performed *after* the function MPI_Publish_name call in the server process. You can, for example, use the MPI_Barrier function for the initial processes and the server process: in the server process, the MPI_Barrier function should be called after the call of the MPI_Publish_name function, whereas in the initial processes, the MPI_Barrier function should be called before the call of the MPI_Comm_spawn function which create the client group.

MPI8Inter22. An integer N is given in each process. The integer N can take three values: 0, 1 and K ($K > 1$). There is exactly one process with the value $N = 1$ and exactly K processes with the value $N = K$. In addition, an integer

A is given in the processes with the non-zero integer *N*. Using the MPI_Comm_split finction, split the initial communicator MPI_COMM_WORLD into two ones: the first new communicator should include the process with $N = 1$, the second one should include the processes with $N = K$. Using one call of the MPI_Comm_spawn function with the first parameter "ptprj.exe" for each new communicator, create two groups of new processes. The number of processes in each new group must coincide with the number of processes in the corresponding communicator (that is, the first group, named the *server group*, should include one process and the second one, named the *client group*, should include *K* processes). Send the integer *A* from each initial process to the new process; the rank of the receiving process should coincide with the rank of the sending process in the new communicator. Output the received integers in the debug section using the Show function. Using the MPI_Open_port, MPI_Publish_name, and MPI_Comm_accept functions on the server side and the MPI_Lookup_name and MPI_Comm_connect functions in the client side, establish a connection between two new groups of processes by means of a new inter-communicator. Using the MPI_Gather collective function for this inter-communicator, send all the integers *A* from the processes of the client group to the single process of the server group and output the received numbers in the debug section using several calls of the Show function in the process of the server group. Then, using the MPI_Send and MPI_Recv functions, send all these numbers from the process of the server group to the initial process that has created the server group. Output the received numbers in this initial process.

Note. The MPI_Lookup_name function call in the client processes should be performed *after* the MPI_Publish_name function call in the server process. You can, for example, send the number *A* to the process of the server group using the MPI_Ssend function and call the MPI_Barrier function for the MPI_COMM_WORLD communicator after the call of the MPI_Ssend function (on the side of the receiving process, you should receive the number *A* only after the call of the MPI_Publish_name function). In the other processes of the MPI_COMM_WORLD communicator, you should call the MPI_Barrier function and then send the numbers *A* to the processes of the client group. Thus, any of the processes of the client group will receive the number *A* only when the process of the server group has already called the MPI_Publish_name function.

9. Parallel matrix algorithms

All numeric data in tasks are integers. Matrices should be input and output by rows. Files with the matrix elements also contain them in a row-major order.

The number of processes in tasks related to the band algorithms (MPI9Matr2–MPI9Matr20) does not exceed 5. The number of processes in tasks related to the block algorithms (MPI9Matr21–MPI9Matr44) does not exceed 16.

Use the char[12] array to store the file name, use the MPI_Bcast function with the MPI_CHAR datatype parameter to send the file name from the master process to the slave processes.

The program templates for each task already contain descriptions of integer variables for storing the numeric data mentioned in tasks (in particular, the matrix sizes), pointers to arrays for storing the matrices themselves, as well as variables of the MPI_Datatype and MPI_Comm type. These variables should be used in all the functions that you need to implement when solving tasks. All names of variables correspond to the notations used in the task formulations. For arrays associated with bands or blocks of matrices, the names **`a`**, **`b`**, **`c`**, **`t`** are used; for arrays associated with the initial matrices *A*, *B*, and their resulting product *C*, the names with the underline are used (namely, **`a_`**, **`b_`**, **`c_`**).

Tasks with the file input-output (MPI9Matr8–MPI9Matr10, MPI9Matr18–MPI9Matr20, MPI9Matr29–MPI9Matr31, MPI9Matr42–MPI9Matr44) require using of the MPI-2 library (the MPICH2 1.3 system). To solve the other tasks in this group, you can use any version of MPI (MPICH 1.3.5 or MPICH2 1.3).

9.1. Non-parallel matrix multiplication algorithm

MPI9Matr1. Integers *M*, *P*, *Q*, a matrix *A* of the size $M \times P$, and a matrix *B* of the size $P \times Q$ are given in the master process. Find and output a $M \times Q$ matrix *C* that is the product of the matrices *A* and *B*.

The formula for calculating the elements of the matrix *C* under the assumption that the rows and columns of all matrices are numbered from 0 is as follows: $C_{I,J} = A_{I,0} \cdot B_{0,J} + A_{I,1} \cdot B_{1,J} + \ldots + A_{I,P-1} \cdot B_{P-1,J}$, where $I = 0, \ldots, M - 1$, $J = 0, \ldots, Q - 1$.

To store the matrices *A*, *B*, *C*, use one-dimensional arrays of size $M \cdot P$, $P \cdot Q$, and $M \cdot Q$ placing elements of matrices in a row-major order (that is, the matrix element with indices *I* and *J* will be stored in the element of the corresponding array with the index $I \cdot N + J$, where *N* is the number of columns of the matrix). The slave processes are not used in this task.

9.2. Band algorithm 1 (horizontal bands)

MPI9Matr2. Integers *M*, *P*, *Q*, a matrix *A* of the size $M \times P$, and a matrix *B* of the size $P \times Q$ are given in the master process. In the first variant of the band algorithm of matrix multiplication, each matrix multiplier is divided into *K* horizontal bands, where *K* is the number of processes (hereinafter bands are distributed by processes and used to calculate a part of the total matrix product in each process).

The band of the matrix A contains N_A rows, the band of the matrix B contains N_B rows. The numbers N_A and N_B are calculated as follows: $N_A = \text{ceil}(M/K)$, $N_B = \text{ceil}(P/K)$, where the operation "/" means the division of real numbers and the function ceil performs rounding up. If the matrix contains insufficient number of rows to fill the last band then the zero-valued rows should be added to this band.

Add, if necessary, the zero-valued rows to the initial matrices, save them in one-dimensional arrays in the master process, and then send the matrix bands from these arrays to all processes as follows: a band with the index R is sent to the process of rank R ($R = 0, 1, \ldots, K - 1$), all the bands A_R are of the size $N_A \times P$, all the bands B_R are of the size $N_B \times Q$. In addition, create a band C_R in each process to store the part of the matrix product $C = AB$ which will be calculated in this process. Each band C_R is of the size $N_A \times Q$ and is filled with zero-valued elements.

The bands, like the initial matrices, should be stored in one-dimensional arrays in a row-major order. To send the matrix sizes, use the MPI_Bcast collective function, to send the bands of the matrices A and B, use the MPI_Scatter collective function.

Include all the above mentioned actions in a Matr1ScatterData function (without parameters). As a result of the call of this function, each process will receive the values N_A, P, N_B, Q, as well as one-dimensional arrays filled with the corresponding bands of the matrices A, B, C. Output all obtained data (that is, the numbers N_A, P, N_B, Q and the bands of the matrices A, B, C) in each process after calling the Matr1ScatterData function. Perform the input of initial data in the Matr1ScatterData function, perform the output of the results in the Solve function.

Note. To reduce the number of the MPI_Bcast function calls, all matrix sizes may be sent as a single array.

MPI9Matr3. Integers N_A, P, N_B, Q and one-dimensional arrays filled with the corresponding bands of matrices A, B, C are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr2 task). Implement the first step of the band algorithm of matrix multiplication as follows: multiply elements in the bands A_R and B_R of each process and perform the cyclic sending each band B_R to the process of the previous rank (that is, from the process 1 to the process 0, from the process 2 to the process 1, …, from the process 0 to the process $K - 1$, where K is the number of processes).

Use the MPI_Sendrecv_replace function to send the bands. To determine the ranks of the sending and receiving processes, use the expression containing the % operator that gives the remainder of a division.

Include all the above mentioned actions in a Matr1Calc function (without parameters). Output the new contents of the bands C_R and B_R in each process; perform data input and output in the Solve function.

Note. As a result of multiplying the bands A_R and B_R, each element of the band C_R will contain a *part* of the terms included in the elements of the product AB; all elements of the band B_R and some of the elements of the band A_R will be used (in particular, the first N_B elements of the band A_0 will be used in the process 0 in the first step and the last N_B elements of the band A_{K-1} will be used in the process $K - 1$ in the first step).

MPI9Matr4. Integers N_A, P, N_B, Q and one-dimensional arrays filled with the corresponding bands of matrices A, B, C are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr2 task). Modify the Matr1Calc function, which was implemented in the previous task; the modified function should provide execution of any step of the band algorithm of matrix multiplication.

To do this, add the parameter named step to the function (this parameter specifies the step number and may be in the range 0 to $K - 1$, where K is the number of processes) and use the value of this parameter in the part of the algorithm that deals with the recalculation of the elements of the band C_R (the cyclic sending of the bands B_R does not depend on the value of the parameter step).

Using two calls of the modified Matr1Calc function with the parameters 0 and 1, execute two initial steps of the band algorithm and output the new contents of the bands C_R and B_R in each process. Perform data input and output in the Solve function.

Note. The parameter step determines which part of the band A_R will be used for the next recalculation of the elements of the band C_R (note that these parts should be selected cyclically).

MPI9Matr5. Integers N_A, P, N_B, Q and one-dimensional arrays filled with the corresponding bands of matrices A, B, C are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr2 task). In addition, a number L with the same value is given in each process. The value of L is in the range 3 to K, where K is the number of processes, and determines the number of steps of the band algorithm.

Using the function Matr1Calc(I) (see the previous task) in a loop with the parameter I ($I = 0, \ldots, L - 1$), execute the initial L steps of the band algorithm and output the new contents of the bands C_R and B_R in each process. Perform data input and output in the Solve function.

Remark. If the value of L is equal to K then the bands C_R will contain the corresponding parts of the final matrix product AB.

MPI9Matr6. An integer M (the number of rows of the matrix product) is given in the master process. In addition, integers N_A, Q and one-dimensional arrays filled with the $N_A \times Q$ bands of matrix C are given in each process (the given bands of C are obtained as a result of K steps of the band algorithm — see the MPI9Matr5 task). Send all the bands C_R to the master process and output the received matrix C of the size $M \times Q$ in this process.

To store the resulting matrix C in the master process, use a one-dimensional array sufficient to store the matrix of the size $(N_A \cdot K) \times Q$. To send data to this array, use the MPI_Gather collective function.

Include all the above mentioned actions in a Matr1GatherData function (without parameters). Perform the input of initial data in the Solve function, perform the output of the resulting matrix in the Matr1GatherData function.

MPI9Matr7. Integers M, P, Q, a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process (thus, the given data coincide with the given data in the MPI9Matr2 task).

Using successively the Matr1ScatterData, Matr1Calc (in a loop), and Matr1GatherData functions, that are developed in the MPI9Matr2−MPI9Matr6 tasks, find a matrix C, which is equal to the product of the initial matrices A and B, and output this matrix in the master process.

In addition, output the current contents of the band C_R in each process after each call of the Matr1Calc function. Modify the Matr1Calc function (see the MPI9Matr4 task), before using in this task, as follows: the bands B_R should not be sent when the parameter step is equal to $K - 1$.

MPI9Matr8. Integers M, P, Q and two file names are given in the master process. The given files contain elements of a matrix A of the size $M \times P$ and a matrix B of the size $P \times Q$. Modify the initial stage of the band algorithm of matrix multiplication (see the MPI9Matr2 task) as follows: each process should read the corresponding bands of the matrices A and B directly from the given files using the MPI_File_seek and MPI_File_read_all collective functions (a new file view is not required).

To send the sizes of matrices and file names, use the MPI_Bcast collective function.

Include all these actions in a Matr1ScatterFile function (without parameters). As a result of the call of this function, each process will receive the values N_A, P, N_B, Q, as well as one-dimensional arrays filled with the corresponding bands of the matrices A, B, C. Output all obtained data (that is, the numbers N_A, P, N_B, Q and the bands of the matrices A, B, C) in each process after calling the Matr1ScatterFile function. Perform the input of initial data in the Matr1ScatterFile function, perform the output of the results in the Solve function.

Remark. For some bands, some of their elements (namely, the last rows) or even the entire bands should not be read from the source files and will remain zero-valued ones. However, this situation does not require special processing, since the MPI_File_read_all function automatically stops reading the data (without generating any error message) when the end of the file is reached.

MPI9Matr9. Integers N_A, Q and one-dimensional arrays filled with the $N_A \times Q$ bands C_R are given in each process (the given bands C_R are obtained as a result of K steps of the band algorithm of matrix multiplication — see the MPI9Matr5 task). In addition, an integer M (the number of rows of the matrix product) and the name of file (to store this product) are given in the master process.

Send the number M and the file name to all processes using the MPI_Bcast function. Write all the parts of the matrix product contained in the bands C_R to the resulting file, which will eventually contain a matrix C of the size $M \times Q$. To write the bands to the file, use the MPI_File_seek and MPI_File_write_all collective functions.

Include all these actions (namely, the input of file name, sending number M and the file name, and writing all bands to the file) in a Matr1GatherFile function. Perform the input of all initial data, except the file name, in the Solve function.

Note. When writing data to the resulting file, it is necessary to take into account that some of the bands C_R may contain trailing zero-valued rows that are not related to the resulting matrix product (the number M should be sent to all processes in order to control this situation).

MPI9Matr10. Integers M, P, Q and three file names are given in the master process. The first two file names are related to the existing files containing the elements of matrices A and B of the size $M \times P$ and $P \times Q$, respectively, the third file should be created to store the resulting matrix product $C = AB$. Using successively the Matr1ScatterFile, Matr1Calc (in a loop), and Matr1GatherFile functions (see the MPI9Matr8, MPI9Matr5, MPI9Matr9 tasks), find a matrix C and write its elements to the resulting file.

In addition, output the current value of the c[step] in each process after each call of the Matr1Calc function, where c is a one-dimensional array containing the band C_R, and step is the algorithm step number (0, 1, …, $K - 1$). Thus, the element c[0] should be output on the first step of the algorithm, the element c[1] should be output on the second step of the algorithm, and so on.

9.3. Band algorithm 2 (horizontal and vertical bands)

MPI9Matr11. Integers P and Q are given in each process; in addition, a matrix B of the size $P \times Q$ is given in the master process. The number Q is a multiple of the number of processes K. Input the matrix B into a one-dimensional array of the size $P \cdot Q$ in the master process and define a new datatype named MPI_BAND_B that contains a vertical band of the matrix B. The width of the vertical band should be equal to $N_B = Q/K$ columns. When defining the MPI_BAND_B datatype, use the MPI_Type_vector and MPI_Type_commit functions.

Include this definition in a Matr2CreateTypeBand(p, n, q, t) function with the input integer parameters p, n, q and the output parameter t of the MPI_Datatype type; the parameters p and n determine the size of the vertical band (the number of its rows and columns), and the parameter q determines the number of columns of the matrix from which this band is extracted.

Using the MPI_BAND_B datatype, send to each process (inclusive of the master process) the corresponding band of the matrix B in the ascending order of ranks of receiving processes. Sending should be performed using the MPI_Send and MPI_Recv functions; the bands should be stored in one-dimensional arrays of the size $P \cdot N_B$. Output the received band in each process.

Remark. In the MPICH2 version 1.3, the MPI_Send function call is erroneous if the sending and receiving processes are the same. You may use the MPI_Sendrecv function to send a band to the master process. You may also fill a band in the master process without using tools of the MPI library.

MPI9Matr12. Integers M, P, Q, a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process. In the second variant of the band algorithm of matrix multiplication, the first multiplier (the matrix A) is divided into K horizontal bands and the second multiplier (the matrix B) is divided into K vertical bands, where K is the number of processes (hereinafter bands are distributed by processes and used to calculate a part of the total matrix product in each process).

The band of the matrix A contains N_A rows, the band of the matrix B contains N_B columns. The numbers N_A and N_B are calculated as follows: $N_A = \text{ceil}(M/K)$, $N_B = \text{ceil}(Q/K)$, where the operation "/" means the division of real numbers and the function ceil performs rounding up. If the matrix contains insufficient number of rows (or columns) to fill the last band then the zero-valued rows (or columns) should be added to this band.

Add, if necessary, the zero-valued rows or columns to the initial matrices, save them in one-dimensional arrays in the master process, and then send the matrix bands from these arrays to all processes as follows: a band with

the index R is sent to the process of rank R ($R = 0, 1, \ldots, K - 1$), all the bands A_R are of the size $N_A \times P$, all the bands B_R are of the size $P \times N_B$. In addition, create a band C_R in each process to store the part of the matrix product $C = AB$ which will be calculated in this process. Each band C_R is of the size $(N_A \cdot K) \times N_B$ and is filled with zero-valued elements.

The bands, like the initial matrices, should be stored in one-dimensional arrays in a row-major order. To send the matrix sizes, use the MPI_Bcast collective function, to send the bands of the matrix A, use the MPI_Scatter collective function, to send the bands of the matrix B, use the MPI_Send and MPI_Recv functions and also the MPI_BAND_B datatype created by the Matr2CreateTypeBand function (see the previous task and a note to it).

Include all the above mentioned actions in a Matr2ScatterData function (without parameters). As a result of the call of this function, each process will receive the values N_A, P, N_B, as well as one-dimensional arrays filled with the corresponding bands of the matrices A, B, C. Output all obtained data (that is, the numbers N_A, P, N_B and the bands of the matrices A, B, C) in each process after calling the Matr2ScatterData function. Perform the input of initial data in the Matr2ScatterData function, perform the output of the results in the Solve function.

Notes. (1) When input the matrix B into an array in the master process, it should be taken into account that this array may contain elements corresponding to additional zero-valued columns.

(2) To reduce the number of the MPI_Bcast function calls, all matrix sizes may be sent as a single array.

MPI9Matr13. Integers N_A, P, N_B and one-dimensional arrays filled with the corresponding bands of matrices A, B, C are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr12 task). Implement the first step of the band algorithm of matrix multiplication as follows: multiply elements in the bands A_R and B_R of each process and perform the cyclic sending each band A_R to the process of the previous rank (that is, from the process 1 to the process 0, from the process 2 to the process 1, …, from the process 0 to the process $K - 1$, where K is the number of processes).

Use the MPI_Sendrecv_replace function to send the bands. To determine the ranks of the sending and receiving processes, use the expression containing the % operator that gives the remainder of a division.

Include all the above mentioned actions in a Matr2Calc function (without parameters). Output the new contents of the bands C_R and A_R in each process; perform data input and output in the Solve function.

Note. In this variant of the band algorithm, the bands A_R contain the full rows of the matrix A and the bands B_R contain the full columns of the matrix B, so, as a result of their multiplication, the band C_R will contain part of

the elements of the final matrix product already at the first step of the algorithm (the other elements of the band C_R will remain zero-valued). The location of the found elements in the band C_R depends on the rank of the process (in particular, the first N_A rows of the band C_0 in the process 0 will be filled in the first step and the last N_A rows of the band C_{K-1} in the process $K - 1$ will be filled in the first step).

MPI9Matr14. Integers N_A, P, N_B and one-dimensional arrays filled with the corresponding bands of matrices A, B, C are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr12 task). Modify the Matr2Calc function, which was implemented in the previous task; the modified function should provide execution of any step of the band algorithm of matrix multiplication.

To do this, add the parameter named step to the function (this parameter specifies the step number and may be in the range 0 to $K - 1$, where K is the number of processes) and use the value of this parameter in the part of the algorithm that deals with the recalculation of the elements of the band C_R (the cyclic sending of the bands A_R does not depend on the value of the parameter step).

Using two calls of the modified Matr2Calc function with the parameters 0 and 1, execute two initial steps of the band algorithm and output the new contents of the bands C_R and A_R in each process. Perform data input and output in the Solve function.

Note. The parameter step determines which rows of the band C_R will be calculated in this step of the algorithm (note that these rows are selected cyclically).

MPI9Matr15. Integers N_A, P, N_B and one-dimensional arrays filled with the corresponding bands of matrices A, B, C are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr12 task). In addition, a number L with the same value is given in each process. The value of L is in the range 3 to K, where K is the number of processes, and determines the number of steps of the band algorithm.

Using the function Matr2Calc(I) (see the previous task) in a loop with the parameter I ($I = 0, \ldots, L - 1$), execute the initial L steps of the band algorithm and output the new contents of the bands C_R and A_R in each process. Perform data input and output in the Solve function.

Remark. If the value of L is equal to K then the bands C_R will contain the corresponding parts of the final matrix product AB.

MPI9Matr16. Integers M and Q (the numbers of rows and columns of the matrix product) are given in the master process. In addition, integers N_A, N_B and one-dimensional arrays filled with the $(N_A \cdot K) \times N_B$ bands of the matrix C are given in each process (the given bands of C are obtained as a result of

K steps of the band algorithm — see the MPI9Matr15 task). Send all the bands C_R to the master process and output the received matrix C of the size $M \times Q$ in this process.

To store the resulting matrix C in the master process, use a one-dimensional array sufficient to store the matrix of the size $(N_A \cdot K) \times (N_B \cdot K)$. To send data to this array, use the MPI_Send and MPI_Recv functions and the MPI_BAND_C datatype created by the Matr2CreateTypeBand function (see the MPI9Matr11 task and a note to it).

Include all the above mentioned actions in a Matr2GatherData function (without parameters). Perform the input of initial data in the Solve function, perform the output of the resulting matrix in the Matr2GatherData function.

Note. When output the matrix C in the master process, it should be taken into account that an array, which is intended for matrix storage, may contain elements corresponding to additional zero-valued columns.

MPI9Matr17. Integers M, P, Q, a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process (thus, the given data coincide with the given data in the MPI9Matr12 task).

Using successively the Matr2ScatterData, Matr2Calc (in a loop), and Matr2GatherData functions, that are developed in the MPI9Matr12−MPI9Matr16 tasks, find a matrix C, which is equal to the product of the initial matrices A and B, and output this matrix in the master process.

In addition, output the current contents of the band C_R in each process after each call of the Matr2Calc function.

Modify the Matr2Calc function (see the MPI9Matr14 task), before using in this task, as follows: the bands A_R should not be sent when the parameter step is equal to $K - 1$.

MPI9Matr18. Integers M, P, Q and two file names are given in the master process. The given files contain elements of a matrix A of the size $M \times P$ and a matrix B of the size $P \times Q$. The number Q is a multiple of the number of processes K. Modify the initial stage of the band algorithm of matrix multiplication (see the MPI9Matr12 task) as follows: each process should read the corresponding bands of the matrices A and B directly from the given files.

To send the sizes of matrices and file names, use the MPI_Bcast collective function. Use the MPI_File_seek and MPI_File_read_all collective functions to read the horizontal bands of the matrix A. To read the vertical bands of the matrix B, set the appropriate file view using the MPI_File_set_view function and the MPI_BAND_B file type defined with the Matr2CreateTypeBand function (see the MPI9Matr11 task), and then use the MPI_File_read_all function.

Include all these actions in a Matr2ScatterFile function (without parameters). As a result of the call of this function, each process will receive the values N_A, P, N_B, as well as one-dimensional arrays filled with the corresponding bands of the matrices A, B, C. Output all obtained data (that is, the numbers N_A, P, N_B and the bands of the matrices A, B, C) in each process after calling the Matr2ScatterFile function. Perform the input of initial data in the Matr2ScatterFile function, perform the output of the results in the Solve function.

Note. A condition that the number Q is a multiple of K allows us to perform reading of the bands B_R using the same file type in all processes.

If this condition is not fulfilled then it would be necessary to use special types that ensure the correct reading from the file and write to the array of "truncated" bands of the matrix B in the last processes (in addition, in this case it would be necessary to send to each process the value of Q which is necessary for the correct type definition for "truncated" bands).

MPI9Matr19. Integers N_A, N_B and one-dimensional arrays filled with the $(N_A \cdot K) \times N_B$ bands C_R are given in each process (the given bands C_R are obtained as a result of K steps of the band algorithm of matrix multiplication — see the MPI9Matr15 task). In addition, an integer M (the number of rows of the matrix product) and the name of file (to store this product) are given in the master process. The number of columns Q of the matrix product is a multiple of the number of processes K (and, therefore, is equal to $N_B \cdot K$).

Send the number M and the file name to all processes using the MPI_Bcast function. Write all the parts of the matrix product contained in the bands C_R to the resulting file, which will eventually contain a matrix C of the size $M \times Q$.

To write the bands to the file, set the appropriate file view using the MPI_File_set_view function and the MPI_BAND_C file type defined with the Matr2CreateTypeBand function (see the MPI9Matr11 task), and then use the MPI_File_write_all function.

Include all these actions (namely, the input of file name, sending number M and the file name, and writing all bands to the file) in a Matr2GatherFile function. Perform the input of all initial data, except the file name, in the Solve function.

Note. When writing data to the resulting file, it is necessary to take into account that the bands C_R may contain trailing zero-valued rows that are not related to the resulting matrix product (the number M should be sent to all processes in order to control this situation).

MPI9Matr20. Integers M, P, Q and three file names are given in the master process. The first two file names are related to the existing files containing the elements of matrices A and B of the size $M \times P$ and $P \times Q$, respectively,

the third file should be created to store the resulting matrix product $C = AB$. The number Q is a multiple of the number of processes K. Using successively the Matr2ScatterFile, Matr2Calc (in a loop), and Matr2GatherFile functions (see the MPI9Matr18, MPI9Matr15, MPI9Matr19 tasks), find a matrix C and write its elements to the resulting file.

In addition, output the current value of the c[step] in each process after each call of the Matr2Calc function, where c is a one-dimensional array containing the band C_R, and step is the algorithm step number (0, 1, …, $K - 1$). Thus, the element c[0] should be output on the first step of the algorithm, the element c[1] should be output on the second step of the algorithm, and so on.

9.4. Cannon's block algorithm

MPI9Matr21. Integers M and P are given in each process; in addition, a matrix A of the size $M \times P$ is given in the master process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$, the numbers M and P are multiples of K_0. Input the matrix A into a one-dimensional array of the size $M \cdot P$ in the master process and define a new datatype named MPI_BLOCK_A that contains a $M_0 \times P_0$ block of the matrix A, where $M_0 = M/K_0$, $P_0 = P/K_0$.

When defining the MPI_BLOCK_A type, use the MPI_Type_vector and MPI_Type_commit functions. Include this definition in a Matr3CreateTypeBlock(m0, p0, p, t) function with the input integer parameters m0, p0, p and the output parameter t of the MPI_Datatype type; the parameters m0 and p0 determine the size of the block, and the parameter p determines the number of columns of the matrix from which this block is extracted.

Using the MPI_BLOCK_A datatype, send to each process (in ascending order of ranks of processes, inclusive of the master process) the corresponding block of the matrix A in a row-major order of blocks (that is, the first block should be sent to the process 0, the next block in the same row of blocks should be sent to the process 1, and so on). Sending should be performed using the MPI_Send and MPI_Recv functions; the blocks should be stored in one-dimensional arrays of the size $M_0 \cdot P_0$. Output the received block in each process.

Remark. In the MPICH2 version 1.3, the MPI_Send function call is erroneous if the sending and receiving processes are the same. You may use the MPI_Sendrecv function to send a block to the master process. You may also fill a block in the master process without using tools of the MPI library.

MPI9Matr22. Integers M_0, P_0 and a matrix A of the size $M_0 \times P_0$ are given in each process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$. Input the matrix A into a one-dimensional array of the size $M_0 \cdot P_0$ in each process and create a new communicator named MPI_COMM_GRID using the

MPI_Cart_create function. The MPI_COMM_GRID communicator defines a Cartesian topology for all processes as a two-dimensional periodic $K_0 \times K_0$ grid (ranks of processes should not be reordered).

Include the creation of the MPI_COMM_GRID communicator in a Matr3CreateCommGrid(comm) function with the output parameter comm of the MPI_Comm type. Using the MPI_Cart_coords function for this communicator, output the process coordinates (I_0, J_0) in each process.

Perform a cyclic shift of the matrices A given in all processes of each grid row I_0 by I_0 positions left (that is, in descending order of ranks of processes) using the MPI_Cart_shift and MPI_Sendrecv_replace functions. Output the received matrix in each process.

MPI9Matr23. Integers M, P, Q, a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$. In the block algorithms of matrix multiplication, the initial matrices are divided into K blocks and are interpreted as square block matrices of the order K_0 (hereinafter blocks are distributed by processes and used to calculate a part of the total matrix product in each process).

The block of the matrix A is of the size $M_0 \times P_0$, the block of the matrix B is of the size $P_0 \times Q_0$, the numbers M_0, P_0, Q_0 are calculated as follows: $M_0 = \text{ceil}(M/K_0)$, $P_0 = \text{ceil}(P/K_0)$, $Q_0 = \text{ceil}(Q/K_0)$, where the operation "/" means the division of real numbers and the function ceil performs rounding up. If the matrix contains insufficient number of rows (or columns) to fill the last blocks then the zero-valued rows (or columns) should be added to these blocks.

Add, if necessary, the zero-valued rows or columns to the initial matrices (as a result, the matrices A and B will have the size $(M_0 \cdot K_0) \times (P_0 \cdot K_0)$ and $(P_0 \cdot K_0) \times (Q_0 \cdot K_0)$ respectively), save them in one-dimensional arrays in the master process, and then send the matrix blocks (in a row-major order) from these arrays to all processes (in ascending order of its ranks): the process R will receive the blocks A_R and B_R, $R = 0, \ldots, K - 1$. In addition, create a block C_R in each process to store the part of the matrix product $C = AB$ which will be calculated in this process. Each block C_R is of the size $M_0 \times Q_0$ and is filled with zero-valued elements.

The blocks, like the initial matrices, should be stored in one-dimensional arrays in a row-major order. To send the matrix sizes, use the MPI_Bcast collective function, to send the blocks of the matrices A and B, use the MPI_Send and MPI_Recv functions and also the MPI_BLOCK_A and MPI_BLOCK_B datatypes created by the Matr3CreateTypeBlock function (see the MPI9Matr21 task and a note to it).

Include all the above mentioned actions in a Matr3ScatterData function (without parameters). As a result of the call of this function, each process

will receive the values M_0, P_0, Q_0, as well as one-dimensional arrays filled with the corresponding blocks of the matrices *A*, *B*, *C*. Output all obtained data (that is, the numbers M_0, P_0, Q_0 and the blocks of the matrices *A*, *B*, *C*) in each process after calling the Matr3ScatterData function. Perform the input of initial data in the Matr3ScatterData function, perform the output of the results in the Solve function.

Notes. (1) When input the matrices *A* and *B* into arrays in the master process, it should be taken into account that these arrays may contain elements corresponding to additional zero-valued columns.

(2) To reduce the number of the MPI_Bcast function calls, all matrix sizes may be sent as a single array.

MPI9Matr24. Integers M_0, P_0, Q_0 and one-dimensional arrays filled with the corresponding blocks of matrices *A*, *B*, *C* are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr23 task). Implement the initial block redistribution used in the Cannon's algorithm for block matrix multiplication.

To do this, define a Cartesian topology for all processes as a two-dimensional periodic $K_0 \times K_0$ grid, where $K_0 \cdot K_0$ is equal to the number of processes (ranks of processes should not be reordered), and perform a cyclic shift of the blocks A_R given in all processes of each grid row I_0 by I_0 positions left (that is, in descending order of ranks of processes), $I_0 = 0, \ldots, K_0 - 1$, and perform a cyclic shift of the blocks B_R given in all processes of each grid column J_0 by J_0 positions up (that is, in descending order of ranks of processes), $J_0 = 0, \ldots, K_0 - 1$.

To create the MPI_COMM_GRID communicator associated with the Cartesian topology, use the Matr3CreateCommGrid function implemented in the MPI9Matr22 task. Use the MPI_Cart_coords, MPI_Cart_shift, MPI_Sendrecv_replace functions to perform the cyclic shifts (compare with MPI9Matr22).

Include all the above mentioned actions in a Matr3Init function (without parameters). Output the received blocks A_R and B_R in each process; perform data input and output in the Solve function.

MPI9Matr25. Integers M_0, P_0, Q_0 and one-dimensional arrays filled with the corresponding blocks of matrices *A*, *B*, *C* are given in each process. The blocks C_R are zero-valued, the initial redistribution for the blocks A_R and B_R has already been performed in accordance with the Cannon's algorithm (see the previous task). Implement one step of the Cannon's algorithm of matrix multiplication as follows: multiply elements in the blocks A_R and B_R of each process and perform a cyclic shift of the blocks A_0 given in all processes of each row of the Cartesian periodic grid by 1 position left (that is, in descending order of ranks of processes) and perform a cyclic shift of the

blocks B_0 given in all processes of each grid column by 1 position up (that is, in descending order of ranks of processes).

To create the MPI_COMM_GRID communicator associated with the Cartesian topology, use the Matr3CreateCommGrid function implemented in the MPI9Matr22 task. Use the MPI_Cart_shift and MPI_Sendrecv_replace functions to perform the cyclic shifts (compare with MPI9Matr22).

Include all the above mentioned actions in a Matr3Calc function (without parameters). Output the new contents of the blocks C_R, A_R, and B_R in each process; perform data input and output in the Solve function.

Remark. A special feature of the Cannon's algorithm is that the actions at each step are not depend on the step number.

MPI9Matr26. Integers M_0, P_0, Q_0 and one-dimensional arrays filled with the corresponding blocks of matrices *A*, *B*, *C* are given in each process. The blocks C_R are zero-valued, the initial redistribution for the blocks A_R and B_R has already been performed in accordance with the Cannon's algorithm (see the MPI9Matr24 task). In addition, a number *L* with the same value is given in each process. The value of *L* is in the range 2 to K_0, where $K_0 \cdot K_0$ is the number of processes, and determines the number of steps of the Cannon's algorithm.

Using the function Matr3Calc (see the previous task) in a loop, execute the initial *L* steps of the Cannon's algorithm and output the new contents of the blocks C_R, A_R, and B_R in each process. Perform data input and output in the Solve function.

Note. If the value of *L* is equal to K_0 then the blocks C_R will contain the corresponding parts of the final matrix product *AB*.

MPI9Matr27. Integers *M* and *Q* (the numbers of rows and columns of the matrix product) are given in the master process. In addition, integers M_0, Q_0 and one-dimensional arrays filled with the $M_0 \times Q_0$ blocks of the matrix *C* are given in each process (the given blocks of *C* are obtained as a result of K_0 steps of the Cannon's algorithm — see the MPI9Matr26 task). Send all the blocks C_R to the master process and output the received matrix *C* of the size $M \times Q$ in this process.

To store the resulting matrix *C* in the master process, use a one-dimensional array sufficient to store the matrix of the size $(M_0 \cdot K_0) \times (Q_0 \cdot K_0)$. To send the blocks C_R to this array, use the MPI_Send and MPI_Recv functions and the MPI_BLOCK_C datatype created by the Matr3CreateTypeBlock function (see the MPI9Matr21 task and a note to it).

Include all the above mentioned actions in a Matr3GatherData function (without parameters). Perform the input of initial data in the Solve function, perform the output of the resulting matrix in the Matr3GatherData function.

Note. When output the matrix C in the master process, it should be taken into account that an array, which is intended for matrix storage, may contain elements corresponding to additional zero-valued columns.

MPI9Matr28. Integers M, P, Q, a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process (thus, the given data coincide with the given data in the MPI9Matr23 task).

Using successively the Matr3ScatterData, Matr3Calc (in a loop), and Matr3GatherData functions, that are developed in the MPI9Matr23−MPI9Matr27 tasks, find a matrix C, which is equal to the product of the initial matrices A and B, and output this matrix in the master process.

In addition, output the current contents of the block C_R in each process after each call of the Matr3Calc function.

The MPI_COMM_GRID communicator, which is used in the Matr3Init and Matr3Calc functions, should not be created several times. To do this, modify the Matr3CreateCommGrid function; the modified function should not perform any actions when it is called with the parameter comm that is not equal to MPI_COMM_NULL.

In addition, modify the Matr3Calc function (see the MPI9Matr25 task), before using in this task, as follows: add the parameter named step to this function (step = 0, …, $K_0 - 1$); the blocks A_R and B_R should not be sent when the parameter step is equal to $K_0 - 1$.

MPI9Matr29. Integers M, P, Q and two file names are given in the master process. The given files contain elements of a matrix A of the size $M \times P$ and a matrix B of the size $P \times Q$. Modify the stage of receiving blocks for the Cannon's algorithm of matrix multiplication (see the MPI9Matr23 task) as follows: each process should read the corresponding blocks of the matrices A and B directly from the given files. In this case, all processes should receive not only the sizes M_0, P_0, Q_0 of the blocks, but also the sizes M, P, Q of the initial matrices, which are needed to determine correctly the positions of blocks in the source files.

To send the sizes of matrices and file names, use the MPI_Bcast collective function. Use the MPI_File_read_at local function to read each row of the block (a new file view is not required).

Include all these actions in a Matr3ScatterFile function (without parameters). As a result of the call of this function, each process will receive the values M, P, Q, M_0, P_0, Q_0, as well as one-dimensional arrays filled with the corresponding blocks of the matrices A, B, C. Output all obtained data (that is, the numbers M, P, Q, M_0, P_0, Q_0 and the blocks of the matrices A, B, C) in each process after calling the Matr3ScatterFile function. Perform the input of initial data in the Matr3ScatterFile function, perform the output of the results in the Solve function.

Note. For some blocks, some of their elements (namely, the last rows and/or columns) should not be read from the source files and will remain zero-valued ones. To determine the actual size of the block being read (the number of rows and columns), it is required to use the sizes of the initial matrices and the coordinates (I_0, J_0) of the block in a square Cartesian grid of order K_0 (note that $I_0 = R/K_0$, $J_0 = R\%K_0$, where R is the process rank).

Remark. Whereas the values of P and Q are necessary to ensure the correct reading of the file blocks, the value of M is not required for this purpose, since the attempt to read data beyond the end of file is ignored (without generating any error message). However, the value of M is required at the final stage of the algorithm (see the next task), so it must also be sent to all processes.

MPI9Matr30. Integers M, Q, M_0, Q_0 and one-dimensional arrays filled with the $M_0 \times Q_0$ blocks C_R are given in each process (the given blocks C_R are obtained as a result of K_0 steps of the Cannon's block algorithm of matrix multiplication — see the MPI9Matr25 task). In addition, the name of file to store the matrix product is given in the master process.

Send the file name to all processes using the MPI_Bcast function. Write all the parts of the matrix product contained in the blocks C_R to the resulting file, which will eventually contain a matrix C of the size $M \times Q$.

Use the MPI_File_write_at local function to write each row of the block to the file (a new file view is not required).

Include all these actions (namely, the input of file name, sending the file name, and writing all blocks to the file) in a Matr3GatherFile function. Perform the input of all initial data, except the file name, in the Solve function.

Note. When writing data to the resulting file, it is necessary to take into account that some of the blocks C_R may contain trailing zero-valued rows and/or columns that are not related to the resulting matrix product (see also the note and the remark for the previous task).

MPI9Matr31. Integers M, P, Q and three file names are given in the master process. The first two file names are related to the existing files containing the elements of matrices A and B of the size $M \times P$ and $P \times Q$, respectively, the third file should be created to store the resulting matrix product $C = AB$.

Using successively the Matr3ScatterFile, Matr3Init, Matr3Calc (in a loop), and Matr3GatherFile functions (see the MPI9Matr29, MPI9Matr24, MPI9Matr25, and MPI9Matr30 tasks), find a matrix C and write its elements to the resulting file.

In addition, output the current value of the c[step] in each process after each call of the Matr3Calc function, where c is a one-dimensional array containing the block C_R, and step is the algorithm step number (0, 1, …, $K_0 - 1$). Thus, the element c[0] should be output on the first step of the al-

gorithm, the element c[1] should be output on the second step of the algorithm, and so on.

9.5. Fox's block algorithm

MPI9Matr32. Integers M and P are given in each process; in addition, a matrix A of the size $M \times P$ is given in the master process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$, the numbers M and P are multiples of K_0. Input the matrix A into a one-dimensional array of the size $M \cdot P$ in the master process and define a new datatype named MPI_BLOCK_A that contains a $M_0 \times P_0$ block of the matrix A, where $M_0 = M/K_0$, $P_0 = P/K_0$.

When defining the MPI_BLOCK_A type, use the MPI_Type_vector and MPI_Type_commit functions. Include this definition in a Matr4CreateTypeBlock(m0, p0, p, t) function with the input integer parameters m0, p0, p and the output parameter t of the MPI_Datatype type; the parameters m0 and p0 determine the size of the block, and the parameter p determines the number of columns of the matrix from which this block is extracted.

Using the MPI_BLOCK_A datatype, send to each process (in ascending order of ranks of processes, inclusive of the master process) the corresponding block of the matrix A in a row-major order of blocks (that is, the first block should be sent to the process 0, the next block in the same row of blocks should be sent to the process 1, and so on). Sending should be performed using the MPI_Alltoallw function; the blocks should be stored in one-dimensional arrays of the size $M_0 \cdot P_0$. Output the received block in each process.

Notes. (1) Use the MPI_Send and MPI_Recv functions instead of the MPI_Alltoallw function when solving this task using the MPI-1 library.

(2) The MPI_Alltoallw function introduced in MPI-2 is the only collective function that allows you to specify the displacements for the sent data in *bytes* (not in elements). This gives opportunity to use it in conjunction with complex data types to implement any variants of collective communications (in our case, we need to implement a communication of the scatter type).

It should be note that all array parameters of the MPI_Alltoallw function associated with the sent data must be differently defined in the master and slave processes. In particular, the array scounts (which determines the number of sent elements) must contain the values 0 in all the slave processes and the value 1 in the master process (the sent elements are of the MPI_BLOCK_A datatype).

At the same time, arrays associated with the received data will be defined in the same way in all processes; in particular, the zero-indexed element of the array rcounts (which determines the number of received elements) must

be equal to $M_0 \cdot P_0$, and all other elements of this array must be equal to 0 (the received elements are of the MPI_INT datatype).

It is necessary to pay special attention to the correct definition of elements in the array sdispls of displacements for the sent data in the master process (in the slave processes, it is enough to use the zero-valued array sdispls).

MPI9Matr33. Integers M_0, P_0 and a matrix A of the size $M_0 \times P_0$ are given in each process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$. Input the matrix A into a one-dimensional array of the size $M_0 \cdot P_0$ in each process and create a new communicator named MPI_COMM_GRID using the MPI_Cart_create function. The MPI_COMM_GRID communicator defines a Cartesian topology for all processes as a two-dimensional periodic $K_0 \times K_0$ grid (ranks of processes should not be reordered).

Include the creation of the MPI_COMM_GRID communicator in a Matr4CreateCommGrid(comm) function with the output parameter comm of the MPI_Comm type. Using the MPI_Cart_coords function for this communicator, output the process coordinates (I_0, J_0) in each process.

On the base of the MPI_COMM_GRID communicator, create a set of communicators named MPI_COMM_ROW, which are associated with the rows of the initial two-dimensional grid. Use the MPI_Cart_sub function to create the MPI_COMM_ROW communicators.

Include the creation of the MPI_COMM_ROW communicators in a Matr4CreateCommRow(grid, row) function with the input parameter grid (the communicator associated with the initial two-dimensional grid) and the output parameter row (both parameters are of the MPI_Comm type). Output the process rank R_0 for the MPI_COMM_ROW communicator in each process (this rank must be equal to J_0).

In addition, send the matrix A from the grid element (I_0, I_0) to all processes of the same grid row I_0 ($I_0 = 0, \ldots, K_0 - 1$) using the MPI_Bcast collective function for the MPI_COMM_ROW communicator. Save the received matrix in the auxiliary matrix T of the same size as the matrix A (it is necessary to copy the matrix A to the matrix T in the sending process before the call of the MPI_Bcast function). Output the received matrix T in each process.

MPI9Matr34. Integers P_0, Q_0 and a matrix B of the size $P_0 \times Q_0$ are given in each process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$. Input the matrix B into a one-dimensional array of the size $P_0 \cdot Q_0$ in each process and create a new communicator named MPI_COMM_GRID, which defines a Cartesian topology for all processes as a two-dimensional periodic $K_0 \times K_0$ grid.

Use the Matr4CreateCommGrid function (see the MPI9Matr33 task) to create the MPI_COMM_GRID communicator. Using the MPI_Cart_coords

function for this communicator, output the process coordinates (I_0, J_0) in each process.

On the base of the MPI_COMM_GRID communicator, create a set of communicators named MPI_COMM_COL, which are associated with the columns of the initial two-dimensional grid. Use the MPI_Cart_sub function to create the MPI_COMM_COL communicators.

Include the creation of the MPI_COMM_COL communicators in a Matr4CreateCommCol(grid, col) function with the input parameter grid (the communicator associated with the initial two-dimensional grid) and the output parameter col (both parameters are of the MPI_Comm type). Output the process rank R_0 for the MPI_COMM_COL communicator in each process (this rank must be equal to I_0).

In addition, perform a cyclic shift of the matrices B given in all processes of each grid column J_0 by 1 position up (that is, in descending order of ranks of processes) using the MPI_Sendrecv_replace function for the MPI_COMM_COL communicator (to determine the ranks of the sending and receiving processes, use the expression containing the % operator that gives the remainder of a division). Output the received matrix in each process.

MPI9Matr35. Integers M, P, Q, a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$. In the block algorithms of matrix multiplication, the initial matrices are divided into K blocks and are interpreted as square block matrices of the order K_0 (hereinafter blocks are distributed by processes and used to calculate a part of the total matrix product in each process).

The block of the matrix A is of the size $M_0 \times P_0$, the block of the matrix B is of the size $P_0 \times Q_0$, the numbers M_0, P_0, Q_0 are calculated as follows: $M_0 = \text{ceil}(M/K_0)$, $P_0 = \text{ceil}(P/K_0)$, $Q_0 = \text{ceil}(Q/K_0)$, where the operation "/" means the division of real numbers and the function ceil performs rounding up. If the matrix contains insufficient number of rows (or columns) to fill the last blocks then the zero-valued rows (or columns) should be added to these blocks.

Add, if necessary, the zero-valued rows or columns to the initial matrices (as a result, the matrices A and B will have the size $(M_0 \cdot K_0) \times (P_0 \cdot K_0)$ and $(P_0 \cdot K_0) \times (Q_0 \cdot K_0)$ respectively), save them in one-dimensional arrays in the master process, and then send the matrix blocks (in a row-major order) from these arrays to all processes (in ascending order of its ranks): the process R will receive the blocks A_R and B_R, $R = 0, \ldots, K - 1$. In addition, create two blocks C_R and T_R filled with zero-valued elements in each process: the block C_R is intended to store the part of the matrix product

$C = AB$, which will be calculated in this process, the block T_R is an auxiliary one. Each block C_R and T_R is of the size $M_0 \times Q_0$.

The blocks, like the initial matrices, should be stored in one-dimensional arrays in a row-major order. To send the matrix sizes, use the MPI_Bcast collective function, to send the blocks of the matrices A and B, use the MPI_Alltoallw collective function and also the MPI_BLOCK_A and MPI_BLOCK_B datatypes created by the Matr4CreateTypeBlock function (see the MPI9Matr32 task and notes to it).

Include all the above mentioned actions in a Matr4ScatterData function (without parameters). As a result of the call of this function, each process will receive the values M_0, P_0, Q_0, as well as one-dimensional arrays filled with the blocks A_R, B_R, C_R, T_R. Output all obtained data (that is, the numbers M_0, P_0, Q_0 and the blocks A_R, B_R, C_R, T_R) in each process after calling the Matr4ScatterData function. Perform the input of initial data in the Matr4ScatterData function, perform the output of the results in the Solve function.

Notes. (1) When input the matrices A and B into arrays in the master process, it should be taken into account that these arrays may contain elements corresponding to additional zero-valued columns.

(2) To reduce the number of the MPI_Bcast function calls, all matrix sizes may be sent as a single array.

MPI9Matr36. Integers M_0, P_0, Q_0 and one-dimensional arrays filled with the blocks A_R, B_R, C_R, T_R are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr35 task). A virtual Cartesian topology in the form of a square grid of order K_0 is used for all processes, the value of $K_0 \cdot K_0$ is equal to the number of the processes. Each step of the Fox's block algorithm of matrix multiplication consists of two stages.

In the first stage of the first step, the block A_R is sent from the process with the grid coordinates (I_0, I_0) to all processes of the same grid row I_0 ($I_0 = 0, \ldots, K_0 - 1$). The received block is saved in the block T_R in the receiving processes. Then the block T_R is multiplied by the block B_R from the same process and the result is added to the block C_R.

Implement the first stage of the first step of the Fox's algorithm. To do this, create the MPI_COMM_GRID and MPI_COMM_ROW communicators using the Matr4CreateCommGrid and Matr4CreateCommRow functions implemented in the MPI9Matr33 task. Use the MPI_Bcast function for the MPI_COMM_ROW communicator to send the blocks A_R (compare with MPI9Matr33).

Include all the above mentioned actions in a Matr4Calc1 function (without parameters). Output the new contents of the blocks T_R and C_R in each process; perform data input and output in the Solve function.

MPI9Matr37. Integers M_0, P_0, Q_0 and one-dimensional arrays filled with the blocks A_R, B_R, C_R, T_R are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr35 task).

Implement the second stage of the first step of the Fox's algorithm of matrix multiplication as follows: perform a cyclic shift of the blocks B_R given in all processes of each column of the Cartesian periodic grid by 1 position up (that is, in descending order of ranks of processes).

To do this, create the MPI_COMM_GRID and MPI_COMM_COL communicators using the Matr4CreateCommGrid and Matr4CreateCommCol functions implemented in the MPI9Matr34 task, then use the MPI_Bcast function for the MPI_COMM_COL communicator to perform the cyclic shift of the blocks B_R (compare with MPI9Matr34).

Include all the above mentioned actions in a Matr4Calc2 function (without parameters). Output the received blocks B_R in each process; perform data input and output in the Solve function.

MPI9Matr38. Integers M_0, P_0, Q_0 and one-dimensional arrays filled with the blocks A_R, B_R, C_R, T_R are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr35 task).

Modify the Matr4Calc1 function, which was implemented in the MPI9Matr36 task; the modified function should provide execution of the first stage of any step of the Fox's algorithm. To do this, add the parameter named step to the function (this parameter specifies the step number and may be in the range 0 to $K_0 - 1$, where K_0 is the order of the Cartesian grid of processes) and use the value of this parameter in the part of the algorithm that deals with the sending the blocks A_R: the block A_R should be sent from the process with the grid coordinates (I_0, $(I_0 + \text{step})\%K_0$) to all processes of the same grid row I_0, $I_0 = 0, \ldots, K_0 - 1$ (the recalculation of the elements of the block C_R does not depend on the value of the parameter step).

Using successively the calls of Matr4Calc1(0), Matr4Calc2(), Matr4Calc1(1) (the Matr4Calc2 function provides the second stage of each step of the algorithm — see the MPI9Matr37 task), execute two initial steps of the Fox's algorithm and output the new contents of the blocks T_R, B_R, and C_R in each process. Perform data input and output in the Solve function.

MPI9Matr39. Integers M_0, P_0, Q_0 and one-dimensional arrays filled with the blocks A_R, B_R, C_R, T_R are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr35 task). In addition, a number L with the same value is given in each process. The value of L is in the range 3 to K_0 and determines the number of steps of the Fox's algorithm.

Using successively the calls of Matr4Calc1(0), Matr4Calc2(), Matr4Calc1(1), Matr4Calc2(), …, Matr4Calc1($L - 1$) (see the MPI9Matr38 and MPI9Matr37 tasks), execute the initial L steps of the Fox's algorithm

and output the new contents of the blocks T_R, B_R, and C_R in each process. Perform data input and output in the Solve function.

Remark. If the value of L is equal to K_0 then the blocks C_R will contain the corresponding parts of the final matrix product AB. Note that the second stage (associated with the call of the Matr4Calc2 function) is not necessary at the last step of the algorithm.

MPI9Matr40. Integers M and Q (the numbers of rows and columns of the matrix product) are given in the master process. In addition, integers M_0, Q_0 and one-dimensional arrays filled with the $M_0 \times Q_0$ blocks of the matrix C are given in each process (the given blocks of C are obtained as a result of K_0 steps of the Fox's algorithm — see the MPI9Matr39 task).

Send all the blocks C_R to the master process and output the received matrix C of the size $M \times Q$ in this process. To store the resulting matrix C in the master process, use a one-dimensional array sufficient to store the matrix of the size $(M_0 \cdot K_0) \times (Q_0 \cdot K_0)$. To send the blocks C_R to this array, use the MPI_Alltoallw collective function and the MPI_BLOCK_C datatype created by the Matr4CreateTypeBlock function (see the MPI9Matr32 task and notes to it).

Include all the above mentioned actions in a Matr4GatherData function (without parameters). Perform the input of initial data in the Solve function, perform the output of the resulting matrix in the Matr4GatherData function.

Note. When output the matrix C in the master process, it should be taken into account that an array, which is intended for matrix storage, may contain elements corresponding to additional zero-valued columns.

MPI9Matr41. Integers M, P, Q, a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process (thus, the given data coincide with the given data in the MPI9Matr35 task).

Using successively the Matr4ScatterData, Matr4Calc1, Matr4Calc2, and Matr4GatherData functions, that are developed in the MPI9Matr35−MPI9Matr40 tasks, find a matrix C, which is equal to the product of the initial matrices A and B, and output this matrix in the master process. The Matr4Calc1 and Matr4Calc2 functions should be called in a loop, the number of Matr4Calc2 function calls must be one less than the number of Matr4Calc1 function calls.

In addition, output the current contents of the block C_R in each process after each call of the Matr4Calc1 function.

The MPI_COMM_GRID, MPI_COMM_ROW, and MPI_COMM_COL communicators that are used in the Matr4Calc1 and Matr4Calc2 functions, should not be created several times. To do this, modify the Matr4CreateCommGrid, Matr4CreateCommRow, and Matr4CreateCommCol functions (see the MPI9Matr33 and MPI9Matr34

tasks); the modified functions should not perform any actions when it is called with the parameter comm that is not equal to MPI_COMM_NULL.

MPI9Matr42. Integers M, P, Q and two file names are given in the master process. The given files contain elements of a matrix A of the size $M \times P$ and a matrix B of the size $P \times Q$. The numbers M, P, and Q are multiples of the order K_0 of square grid of processes.

Modify the stage of receiving blocks for the Fox's algorithm of matrix multiplication (see the MPI9Matr35 task) as follows: each process should read the corresponding blocks of the matrices A and B directly from the given files.

To send the sizes of matrices and file names, use the MPI_Bcast collective function. To read the blocks from the files, set the appropriate file views using the MPI_File_set_view function and the MPI_BLOCK_A and MPI_BLOCK_B file types defined with the Matr4CreateTypeBlock function (see the MPI9Matr32 task), and then use the MPI_File_read_all function.

Include all these actions in a Matr4ScatterFile function (without parameters). As a result of the call of this function, each process will receive the values M_0, P_0, Q_0, as well as one-dimensional arrays filled with the blocks A_R, B_R, C_R, T_R (the blocks C_R and T_R should contain zero-valued elements). Output all obtained data (that is, the numbers M_0, P_0, Q_0 and the blocks A_R, B_R, C_R, T_R) in each process after calling the Matr4ScatterFile function. Perform the input of initial data in the Matr4ScatterFile function, perform the output of the results in the Solve function.

Remark. A condition that the numbers M, P, Q are multiples of K_0 means that you do not need to add zero-valued rows and/or zero-valued columns to the blocks obtained from the matrices A and B, and therefore you may perform reading of the blocks A_R and B_R using the same file types (namely, MPI_BLOCK_A and MPI_BLOCK_B) in all processes.

If this condition is not fulfilled then it would be necessary to use special types that ensure the correct reading from the file and write to the array of "truncated" blocks of the matrices A and B in some processes (in addition, in this case it would be necessary to send to each process the values of P and Q that are necessary for the correct type definition for "truncated" blocks).

MPI9Matr43. Integers M_0, Q_0 and one-dimensional arrays filled with the $M_0 \times Q_0$ blocks C_R are given in each process (the given blocks C_R are obtained as a result of K_0 steps of the Fox's block algorithm of matrix multiplication — see the MPI9Matr39 task). In addition, the name of file to store the matrix product is given in the master process. The numbers M and Q (the numbers of rows and columns of the matrix product) are multiples of the order K_0 of square grid of processes (thus, $M = M_0 \cdot K_0$, $Q = Q_0 \cdot K_0$).

Send the file name to all processes using the MPI_Bcast function. Write all the parts of the matrix product contained in the blocks C_R to the resulting file, which will eventually contain a matrix C of the size $M \times Q$.

To write the blocks to the files, set the appropriate file view using the MPI_File_set_view function and the MPI_BLOCK_C file type defined with the Matr4CreateTypeBlock function (see the MPI9Matr32 task), and then use the MPI_File_write_all collective function.

Include all these actions (namely, the input of file name, sending the file name, and writing all blocks to the file) in a Matr4GatherFile function. Perform the input of all initial data, except the file name, in the Solve function.

Remark. A condition that the numbers M and Q are multiples of K_0 means that the blocks C_R do not contain "extra" zero-valued rows and/or columns, and therefore you may perform writing of the blocks C_R to the file using the same file type (namely, MPI_BLOCK_C) in all processes.

MPI9Matr44. Integers M, P, Q and three file names are given in the master process. The first two file names are related to the existing files containing the elements of matrices A and B of the size $M \times P$ and $P \times Q$, respectively, the third file should be created to store the resulting matrix product $C = AB$. The numbers M, P, and Q are multiples of the order K_0 of square grid of processes.

Using successively the Matr4ScatterFile, Matr4Calc1, Matr3Calc2, and Matr4GatherFile functions (see the MPI9Matr42, MPI9Matr38, MPI9Matr37, and MPI9Matr43 tasks), find a matrix C and write its elements to the resulting file. The Matr4Calc1 and Matr4Calc2 functions should be called in a loop, the number of Matr4Calc2 function calls must be one less than the number of Matr4Calc1 function calls.

In addition, output the current value of the c[step] in each process after each call of the Matr4Calc1 function, where c is a one-dimensional array containing the block C_R, and step is the algorithm step number (0, 1, …, $K_0 - 1$). Thus, the element c[0] should be output on the first step of the algorithm, the element c[1] should be output on the second step of the algorithm, and so on.

10. Процессы и их ранги

MPI1Proc1. В каждом из процессов, входящих в коммуникатор MPI_COMM_WORLD, прочесть одно вещественное число и вывести его противоположное значение. Для ввода и вывода данных использовать поток ввода-вывода pt, определенный в задачнике. Кроме того, отобразить найденное значение в разделе отладки, используя функцию Show, также определенную в задачнике.

MPI1Proc2. В каждом из процессов, входящих в коммуникатор MPI_COMM_WORLD, прочесть одно целое число *A* и вывести его удвоенное значение. Кроме того, для *главного процесса* (процесса ранга 0) вывести количество процессов, входящих в коммуникатор MPI_COMM_WORLD. Для ввода и вывода данных использовать поток ввода-вывода pt. В главном процессе продублировать вывод данных в разделе отладки, отобразив на отдельных строках удвоенное значение *A* и количество процессов (использовать два вызова функции ShowLine, определенной в задачнике наряду с функцией Show).

MPI1Proc3. В главном процессе прочесть вещественное число *X* и вывести его противоположное значение, в каждом из остальных процессов (*подчиненных процессов*, ранг которых больше 0) вывести его ранг. Кроме того, продублировать вывод данных в разделе отладки, отобразив значение $-X$ с комментарием «-X=», а значения рангов с комментариями «rank=» (использовать функцию Show с двумя параметрами).

MPI1Proc4. В процессах четного ранга (включая главный) ввести целое число и вывести его удвоенное значение. В процессах нечетного ранга не выполнять никаких действий.

MPI1Proc5. В процессах четного ранга (включая главный) ввести целое число, в процессах нечетного ранга ввести вещественное число. В каждом процессе вывести удвоенное значение введенного числа.

MPI1Proc6. В подчиненных процессах четного ранга ввести целое число, в процессах нечетного ранга ввести вещественное число. В каждом подчиненном процессе вывести удвоенное значение введенного числа. В главном процессе не выполнять никаких действий.

MPI1Proc7. В каждом процессе четного ранга (включая главный) дано целое число N (> 0) и набор из N вещественных чисел. Вывести в каждом из этих процессов сумму чисел из данного набора. В процессах нечетного ранга не выполнять никаких действий.

MPI1Proc8. В каждом процессе дано целое число N (> 0) и набор из N вещественных чисел. В процессах четного ранга (включая главный) вывести сумму чисел из данного набора, в процессах нечетного ранга вывести среднее арифметическое чисел из данного набора.

MPI1Proc9. В каждом процессе дано целое число N (> 0) и набор из N вещественных чисел. В подчиненных процессах четного ранга вывести сумму чисел из данного набора, в процессах нечетного ранга вывести среднее арифметическое чисел из данного набора, в главном процессе вывести произведение чисел из данного набора.

MPI1Proc10. В каждом процессе дано целое число N (> 0) и набор из N чисел, причем в подчиненных процессах нечетного ранга (1, 3, …) набор содержит вещественные числа, в подчиненных процессах четного ранга (2, 4, …) — целые числа, а тип элементов в главном процессе зависит от общего количества процессов: если количество процессов нечетное, то набор содержит целые числа, а если четное, то вещественные. В процессах четного ранга (включая главный) вывести минимальный элемент из данного набора, в процессах нечетного ранга вывести максимальный элемент.

11. Обмен сообщениями между отдельными процессами

11.1. Блокирующая пересылка данных

MPI2Send1. В каждом подчиненном процессе дано целое число. Переслать эти числа в главный процесс, используя функции MPI_Send и MPI_Recv (стандартные блокирующие функции для передачи и приема сообщения), и вывести их в главном процессе. Полученные числа выводить в порядке возрастания рангов переславших их процессов.

MPI2Send2. В каждом подчиненном процессе дано вещественное число. Переслать эти числа в главный процесс, используя функции MPI_Bsend (посылка сообщения с буферизацией) и MPI_Recv, и вывести их в главном процессе. Полученные числа выводить в порядке убывания рангов переславших их процессов. Для задания буфера использовать функцию MPI_Buffer_attach.

MPI2Send3. В каждом подчиненном процессе даны четыре целых числа. Переслать эти числа в главный процесс, используя по одному вызову функции MPI_Send для каждого передающего процесса, и вывести их в главном процессе. Полученные числа выводить в порядке возрастания рангов переславших их процессов.

MPI2Send4. В каждом подчиненном процессе дано целое число N ($0 < N < 5$) и набор из N целых чисел. Переслать данные наборы в главный процесс, используя по одному вызову функции MPI_Bsend для каждого передающего процесса, и вывести наборы в главном процессе в порядке возрастания рангов переславших их процессов. Для

определения размера пересланного набора использовать функцию MPI_Get_count.

MPI2Send5. В главном процессе дан набор вещественных чисел; количество чисел равно количеству подчиненных процессов. С помощью функции MPI_Send переслать по одному числу в каждый из подчиненных процессов (первое число в процесс 1, второе — в процесс 2, и т. д.) и вывести в подчиненных процессах полученные числа.

MPI2Send6. В главном процессе дан набор вещественных чисел; количество чисел равно количеству подчиненных процессов. С помощью функции MPI_Bsend переслать по одному числу в каждый из подчиненных процессов, перебирая процессы в обратном порядке (первое число в последний процесс, второе — в предпоследний процесс, и т. д.), и вывести в подчиненных процессах полученные числа.

MPI2Send7. В главном процессе дано целое число N и набор из N чисел; $K - 1 \le N < 10$, где K — количество процессов. С помощью функции MPI_Send переслать по одному числу их данного набора в процессы 1, 2, …, $K - 2$, а оставшиеся числа — в процесс $K - 1$, и вывести полученные числа. В процессе $K - 1$ для определения количества полученных чисел использовать функцию MPI_Get_count.

MPI2Send8. В каждом подчиненном процессе дано целое число, причем только для одного процесса это число отлично от нуля. Переслать ненулевое число в главный процесс и вывести в главном процессе полученное число и ранг процесса, переславшего это число. Для приема сообщения в главном процессе использовать функцию MPI_Recv с параметром MPI_ANY_SOURCE.

MPI2Send9. В каждом подчиненном процессе дано целое число N, причем для одного процесса это число больше нуля, а для остальных равно нулю. В процессе с ненулевым N дан также набор из N чисел. Переслать данный набор чисел в главный процесс и вывести в главном процессе полученные числа и ранг процесса, переславшего этот набор. При приеме сообщения использовать параметр MPI_ANY_SOURCE.

MPI2Send10. В каждом подчиненном процессе дано целое число N, в главном процессе дано целое число K (> 0), равное количеству тех подчиненных процессов, в которых даны положительные числа N. Переслать все положительные числа N в главный процесс и вывести в нем сумму полученных чисел. Для приема сообщений в главном процессе использовать функцию MPI_Recv с параметром MPI_ANY_SOURCE.

MPI2Send11. В каждом процессе дано вещественное число. Переслать число из главного процесса во все подчиненные процессы, а все числа

из подчиненных процессов — в главный, и вывести в каждом процессе полученные числа (в главном процессе числа выводить в порядке возрастания рангов переславших их процессов). Для отправки сообщений использовать функцию MPI_Ssend.

Указание. Функция MPI_Ssend обеспечивает *синхронный режим* пересылки данных, при котором операция отправки сообщения будет завершена только после начала приема этого сообщения процессом-получателем. В случае пересылки данных в синхронном режиме возникает опасность *взаимных блокировок* (deadlocks) из-за неправильного порядка вызова функций отправки и получения сообщений.

MPI2Send12. В каждом процессе дано целое число. С помощью функций MPI_Ssend и MPI_Recv осуществить для всех процессов циклический сдвиг данных с шагом 1, переслав число из процесса 0 в процесс 1, из процесса 1 в процесс 2, …, из последнего процесса в процесс 0. В каждом процессе вывести полученное число.

Указание. См. указание к задаче MPI2Send11.

MPI2Send13. В каждом процессе дано целое число. С помощью функций MPI_Ssend и MPI_Recv осуществить для всех процессов циклический сдвиг данных с шагом −1, переслав число из процесса 1 в процесс 0, из процесса 2 в процесс 1, …, из процесса 0 в последний процесс. В каждом процессе вывести полученное число.

Указание. См. указание к задаче MPI2Send11.

MPI2Send14. В каждом процессе даны два целых числа. С помощью функций MPI_Ssend и MPI_Recv переслать первое число в предыдущий процесс, а второе — в последующий процесс (для процесса 0 считать предыдущим последний процесс, а для последнего процесса считать последующим процесс 0). В каждом процессе вывести числа, полученные от предыдущего и последующего процесса (в указанном порядке).

Указание. См. указание к задаче MPI2Send11.

MPI2Send15. В каждом процессе даны два числа: вещественное A и целое N, причем набор чисел N содержит все значения от 0 до $K-1$, где K — количество процессов. Используя функции MPI_Send и MPI_Recv (с параметром MPI_ANY_SOURCE), выполнить в каждом процессе пересылку числа A в процесс N и вывести полученное число, а также ранг процесса, из которого число было получено.

MPI2Send16. В каждом процессе дано целое число N, причем для одного процесса значение N равно 1, а для остальных равно 0. В процессе с $N = 1$ дан также набор из $K-1$ числа, где K — количество процессов. Переслать из этого процесса по одному из чисел данного набора в ос-

тальные процессы, перебирая ранги получателей в возрастающем порядке, и вывести в каждом из них полученное число.

MPI2Send17. В каждом процессе дан набор из $K - 1$ целого числа, где K — количество процессов. Для каждого процесса переслать по одному из данных в нем чисел в остальные процессы, перебирая ранги процессов-получателей в возрастающем порядке, и вывести полученные числа в порядке возрастания рангов переславших их процессов.

MPI2Send18. Количество процессов — четное число. В каждом процессе дано целое число N ($0 < N < 5$) и набор из N чисел. С помощью функции MPI_Sendrecv выполнить обмен исходными наборами между парами процессов 0 и 1, 2 и 3, и т. д. В каждом процессе вывести полученный набор чисел.

MPI2Send19. В каждом процессе дано вещественное число. С помощью функции MPI_Sendrecv_replace поменять порядок исходных чисел на обратный (число из процесса 0 должно быть передано в последний процесс, число из процесса 1 — в предпоследний процесс, …, число из последнего процесса — в процесс 0). В каждом процессе вывести полученное число.

MPI2Send20. В каждом подчиненном процессе дано вещественное число A и его порядковый номер N (целое число); набор всех номеров N содержит все целые числа от 1 до $K - 1$, где K — количество процессов. Переслать числа A в главный процесс и вывести их в порядке, соответствующем возрастанию их номеров N. Массивы не использовать; для передачи номера N указывать его в качестве параметра msgtag функции MPI_Send.

MPI2Send21. В каждом подчиненном процессе дано целое число L (≥ 0) и набор из L пар чисел (A, N), где A — вещественное число, а N — его порядковый номер. Все числа L в сумме равны $2K$, где K — количество процессов; набор номеров N, данных во всех процессах, содержит все целые числа от 1 до $2K$. Переслать числа A в главный процесс и вывести их в порядке, соответствующем возрастанию их номеров N. Для передачи номера N указывать его в качестве параметра msgtag функции MPI_Send.

MPI2Send22. В главном процессе дан набор пар чисел (T, A); количество пар равно числу подчиненных процессов. Число T — целое, равное 0 или 1. Число A — целое, если $T = 0$, и вещественное, если $T = 1$. Переслать по одному числу A в каждый из подчиненных процессов (первое число в процесс 1, второе — в процесс 2, и т. д.) и вывести полученные числа. Для передачи информации о типе пересланного числа указывать число T в качестве параметра msgtag функции MPI_Send, для

получения этой информации использовать функцию MPI_Probe с параметром MPI_ANY_TAG.

Указание. Чтобы избежать дублирования кода, используйте вспомогательные *шаблонные функции* `template<typename T> void send(int t, int dest, MPI_Datatype d)` (для отправки данных) и `template<typename T> void recv(MPI_Datatype d)` (для получения данных). В качестве параметра t указывайте число, равное 0 или 1, в качестве параметра dest — ранг процесса-получателя.

MPI2Send23. В каждом подчиненном процессе даны два целых числа *T*, *N* и набор из *N* чисел. Число *T* равно 0 или 1. Набор содержит целые числа, если *T* = 0, и вещественные числа, если *T* = 1. Переслать исходные наборы в главный процесс и вывести полученные числа в порядке возрастания рангов переславших их процессов. Для передачи информации о типе пересланных чисел указывать число *T* в качестве параметра msgtag функции MPI_Send, для получения этой информации использовать функцию MPI_Probe с параметром MPI_ANY_TAG.

Указание. Чтобы избежать дублирования кода, используйте вспомогательные *шаблонные функции* `template<typename T> void send(int t, MPI_Datatype d)` (для отправки данных) и `template<typename T> void recv(MPI_Datatype d, MPI_Status s)` (для получения данных). В качестве параметра t указывайте число, равное 0 или 1, в качестве параметра s — результат, возвращенный функцией MPI_Probe.

MPI2Send24. Количество процессов *K* является четным: $K = 2N$. В процессах четного ранга (0, 2, …, $K - 2$) дан набор из *N* вещественных чисел, в процессах нечетного ранга (1, 3, …, $K - 1$) — набор из *N* целых чисел. Используя функцию MPI_Sendrecv_replace, выполнить циклический сдвиг всех наборов вещественных чисел в направлении увеличения рангов процессов и циклический сдвиг всех наборов целых чисел в направлении уменьшения рангов (таким образом, вещественные наборы надо переслать из процесса 0 в процесс 2, из процесса 2 — в процесс 4, …, из процесса $K - 2$ — в процесс 0; целочисленные наборы надо переслать из процесса $K - 1$ в процесс $K - 3$, из процесса $K - 3$ в процесс $K - 5$, …, из процесса 1 в процесс $K - 1$). Вывести в каждом процессе полученные числа. Для определения рангов процессов-получателей использовать выражение, содержащее операцию % нахождения остатка от деления, в качестве ранга процесса-отправителя достаточно указывать константу MPI_ANY_SOURCE.

Указание. Чтобы избежать дублирования кода, используйте вспомогательную *шаблонную функцию* `template<typename T> void`

`sendrecv(int rank, int size, MPI_Datatype d, int step)`, где параметр step указывает величину сдвига, равную 2 для наборов вещественных чисел и −2 для целочисленных наборов.

MPI2Send25. Количество процессов K является четным: $K = 2N$. В первой половине процессов дан набор целых чисел размера $R + 1$, где R — ранг процесса ($R = 0, 1, \ldots, N - 1$), во второй половине процессов дан набор вещественных чисел размера $2N - R$, где R — ранг процесса ($R = N, N + 1, \ldots, 2N - 1$). Используя функцию MPI_Sendrecv, переслать исходные наборы из каждой половины процессов в соответствующий процесс другой половины (в частности, набор из процесса 0 надо переслать в процесс N, из процесса 1 — в процесс $N + 1$, из процесса N — в процесс 0, из процесса $2N - 1$ — в процесс $N - 1$). Вывести в каждом процессе полученные числа.

Указание. Чтобы избежать дублирования кода, используйте вспомогательную *шаблонную функцию* `template<typename T1, typename T2> void sendrecv(MPI_Datatype d1, int cnt1, int rank2, MPI_Datatype d2, int cnt2)`, где параметры d1 и cnt1 определяют характеристики процесса, вызвавшего функцию (тип элементов в наборе и их количество), а параметры rank2, d2, cnt2 — ранг и аналогичные характеристики процесса, с которым выполняется обмен данными.

11.2. Неблокирующая пересылка данных

MPI2Send26. В каждом процессе дано целое число N; во всех процессах, кроме одного, значение N равно 0, в некотором выделенном процессе значение N равно 1. В выделенном процессе также дан набор целых чисел A размера $K - 1$, где K — количество процессов. Не сохраняя набор A в массиве, переслать по одному элементу этого набора из выделенного процесса во все остальные процессы, перебирая их в порядке возрастания рангов, и вывести в каждом процессе полученное число. Для пересылки данных использовать требуемое количество вызовов функций MPI_Issend (посылка сообщения в синхронном неблокирующем режиме) и MPI_Wait в выделенном процессе и функцию MPI_Recv в остальных процессах. Дополнительно отобразить в разделе отладки продолжительность каждого вызова функции MPI_Wait в миллисекундах; для этого вызвать функцию MPI_Wtime до и после MPI_Wait и с помощью функции Show вывести разность возвращенных функцией MPI_Wtime значений, умноженную на 1000. Проверить, как изменится отладочная информация, если вместо функции MPI_Issend использовать функцию MPI_Isend (посылка сообщения в стандартном неблокирующем режиме).

MPI2Send27. В каждом процессе дано целое число N; в некотором выделенном процессе значение N равно −1, а в остальных процессах N является одинаковым и равно рангу R выделенного процесса. Во всех процессах, кроме выделенного, также дано вещественное число A. Переслать данные числа A в выделенный процесс и вывести их в порядке возрастания рангов процессов-отправителей. Для пересылки данных использовать требуемое количество вызовов функции MPI_Recv в выделенном процессе и функции MPI_Issend и MPI_Test в остальных процессах. Вызов функции MPI_Test повторять в цикле до тех пор, пока она не вернет ненулевой флаг, и отобразить в разделе отладки потребовавшееся количество итераций этого цикла, используя функцию Show. Проверить, как изменится отладочная информация, если вместо функции MPI_Issend использовать функцию MPI_Isend.

MPI2Send28. В каждом процессе дано целое число N; в некотором выделенном процессе значение N равно −1, а в остальных процессах N является одинаковым и равно рангу R выделенного процесса. Во всех процессах, кроме выделенного, также дано вещественное число A. Переслать данные числа A в выделенный процесс и вывести их в порядке убывания рангов процессов-отправителей. Для пересылки данных использовать требуемое количество вызовов функций MPI_Irecv (прием сообщения в неблокирующем режиме) и MPI_Test в выделенном процессе и функцию MPI_Ssend в остальных процессах. После каждого вызова функции MPI_Irecv организовать цикл, в котором вызывать функцию MPI_Test, пока она не вернет ненулевой флаг, и отображать в разделе отладки потребовавшееся количество итераций этого цикла, используя функцию Show. Проверить, как изменится отладочная информация, если вместо функции MPI_Ssend использовать функцию MPI_Send.

MPI2Send29. В каждом процессе дано целое число N; в некотором выделенном процессе значение N равно −1, а в остальных процессах N является одинаковым и равно рангу R выделенного процесса. Во всех процессах, кроме выделенного, также дано вещественное число A. Переслать данные числа A в выделенный процесс и вывести их сумму S. Для пересылки данных использовать требуемое количество вызовов функций MPI_Irecv и MPI_Waitany в выделенном процессе и функцию MPI_Ssend в остальных процессах. В выделенном процессе описать массив Q типа MPI_Request и организовать вызов функций MPI_Irecv в отдельном цикле, указывая для каждого вызова свой элемент массива Q. После выхода из этого цикла организовать второй цикл, в котором вызывать функцию MPI_Waitany и накапливать сумму S. Дополнительно на каждой итерации второго цикла отображать в разделе отладки следующие данные (используя по одному вызову

функций Show и ShowLine): значение *A*, добавленное к сумме на данной итерации цикла, и ранг процесса, переславшего это значение.

MPI2Send30. В каждом процессе дано целое число *N*; во всех процессах, за исключением двух, значение *N* равно 0; в одном из оставшихся процессов (*процессе-отправителе*) значение *N* равно 1, в другом (*процессе-получателе*) значение *N* равно 2. В процессе-отправителе также дано целое число *R* — ранг процесса получателя и набор *A* целых чисел размера *K*, где *K* — количество процессов. Не сохраняя набор *A* в массиве, переслать все его элементы процессу-получателю и вывести их в том же порядке. Для пересылки данных использовать единственный вызов функции MPI_Ssend_init и требуемое количество вызовов функций MPI_Start и MPI_Wait в процессе-отправителе и единственный вызов функции MPI_Recv_init и требуемое количество вызовов функций MPI_Start и MPI_Wait в процессе-получателе. Дополнительно отобразить в разделе отладки продолжительность каждого вызова функции MPI_Wait в миллисекундах (как для процесса-отправителя, так и для процесса-получателя); для этого вызвать функцию MPI_Wtime до и после MPI_Wait и с помощью функции Show вывести разность возвращенных функцией MPI_Wtime значений, умноженную на 1000. Проверить, как изменится отладочная информация, если вместо функции MPI_Ssend_init использовать функцию MPI_Send_init.

MPI2Send31. В каждом процессе дано целое число *N*; в одном из процессов (*процессе-получателе*) значение *N* равно 2, в некоторых процессах (*процессах-отправителях*) значение *N* равно 1, в остальных процессах значение *N* равно 0. В каждом из процессов-отправителей также дано целое число *R* — ранг процесса-получателя и набор *A* целых чисел размера *K*, где *K* — количество процессов. В процессе-получателе дано целое число *C* — количество процессов-отправителей. Переслать все наборы *A* процессу-получателю и вывести набор *S* суммарных значений элементов всех наборов *A* с одинаковыми индексами (в порядке возрастания индексов). Для пересылки данных использовать единственный вызов функции MPI_Ssend в процессах-отправителях. В процессе-получателе описать массив *Q* типа MPI_Request и организовать вызов функций MPI_Recv_init в отдельном цикле, указывая для каждого вызова свой элемент массива *Q*. После выхода из этого цикла выполнить единственный вызов функции MPI_Startall и организовать второй цикл, в котором вызывать функцию MPI_Waitany и накапливать суммарные значения в массиве *S*. Дополнительно на каждой итерации второго цикла отображать в разделе отладки следующие данные (используя два вызова функции Show и один вызов ShowLine): продолжительность каждого вызова функции MPI_Waitany в милли-

секундах, значение третьего параметра index, возвращенное функцией MPI_Waitany, и ранг процесса-отправителя, соответствующий параметру index. Для нахождения продолжительности вызвать функцию MPI_Wtime до и после MPI_Waitany и вычислить разность возвращенных функцией MPI_Wtime значений, умножив ее на 1000. Для нахождения ранга процесса-отправителя проанализировать значение последнего параметра (типа MPI_Status), возвращенного функцией MPI_Waitany.

MPI2Send32. В каждом процессе дано целое число *N*; в одном из процессов (*процессе-отправителе*) значение *N* равно 1, в некоторых процессах (*процессах-получателях*) значение *N* равно 2, в остальных процессах значение *N* равно 0. В процессе-отправителе также дано вещественное число *A*, целое число *C* — количество процессов-получателей и набор целых чисел *R* размера *C*, содержащий ранги процессов-получателей. Переслать число *A* всем процессам-получателям и вывести его в каждом из этих процессов. Для пересылки данных использовать функцию MPI_Recv в процессах-получателях. В процессе-отправителе описать массив *Q* типа MPI_Request и организовать вызов функций MPI_Ssend_init в отдельном цикле, указывая для каждого вызова свой элемент массива *Q*. После выхода из этого цикла выполнить единственный вызов функции MPI_Startall и организовать второй цикл, в котором вызывать функцию MPI_Testany во вложенном цикле, пока она не вернет ненулевой флаг. Дополнительно на каждой итерации второго цикла отображать в разделе отладки следующие данные (используя по одному вызову функций Show и ShowLine): значение третьего параметра index, возвращенное функцией MPI_Testany (в ситуации, когда она вернула ненулевой флаг), и количество произведенных вызовов функции MPI_Testany (т. е. количество итераций вложенного цикла). Проверить, как изменится отладочная информация, если вместо функции MPI_Ssend_init использовать функцию MPI_Send_init.

12. Коллективные взаимодействия

12.1. Коллективная пересылка данных

MPI3Coll1. В главном процессе дано целое число. Используя функцию MPI_Bcast, переслать это число во все подчиненные процессы и вывести в них полученное число.

MPI3Coll2. В главном процессе дан набор из 5 чисел. Используя функцию MPI_Bcast, переслать этот набор во все подчиненные процессы и вывести в них полученные числа в том же порядке.

MPI3Coll3. В каждом процессе дано вещественное число. Используя функцию MPI_Gather, переслать эти числа в главный процесс и вывести их в порядке возрастания рангов переславших их процессов (первым вывести число, данное в главном процессе).

MPI3Coll4. В каждом процессе дан набор из 5 целых чисел. Используя функцию MPI_Gather, переслать эти наборы в главный процесс и вывести их в порядке возрастания рангов переславших их процессов (первым вывести набор чисел, данный в главном процессе).

MPI3Coll5. В каждом процессе дан набор из $R + 2$ целых чисел, где число R равно рангу процесса (в процессе 0 даны 2 числа, в процессе 1 даны 3 числа, и т. д.). Используя функцию MPI_Gatherv, переслать эти наборы в главный процесс и вывести полученные наборы в порядке возрастания рангов переславших их процессов (первым вывести набор, данный в главном процессе).

MPI3Coll6. В главном процессе дан набор из K чисел, где K — количество процессов. Используя функцию MPI_Scatter, переслать по одному числу в каждый процесс (включая главный) и вывести в каждом процессе полученное число.

MPI3Coll7. В главном процессе дан набор из $3K$ чисел, где K — количество процессов. Используя функцию MPI_Scatter, переслать по 3 числа в каждый процесс (включая главный) и вывести в каждом процессе полученные числа.

MPI3Coll8. В главном процессе дан набор из K чисел, где K — количество процессов. Не меняя порядок расположения чисел в исходном наборе и используя функцию MPI_Scatterv, переслать по одному числу в каждый процесс; при этом первое число надо переслать в процесс $K - 1$, второе число — в процесс $K - 2$, …, последнее число — в процесс 0. Вывести в каждом процессе полученное число.

MPI3Coll9. В главном процессе дан набор из $K(K + 3)/2$ целых чисел, где K — количество процессов. Используя функцию MPI_Scatterv, переслать в каждый процесс часть чисел из данного набора; при этом в процесс ранга R надо переслать $R + 2$ очередных числа (в процесс 0 — первые два числа, в процесс 1 — следующие три числа, и т. д.). В каждом процессе вывести полученные числа.

MPI3Coll10. В главном процессе дан набор из $K + 2$ чисел, где K — количество процессов. Используя функцию MPI_Scatterv, переслать в каждый процесс три числа из данного набора; при этом в процесс ранга R должны быть пересланы числа с номерами от $R + 1$ до $R + 3$ (в процесс 0 — первые три числа, в процесс 1 — числа со второго по четвертое, и т. д.). В каждом процессе вывести полученные числа.

MPI3Coll11. В каждом процессе дано вещественное число. Используя функцию MPI_Allgather, переслать эти числа во все процессы и вывести их в каждом процессе в порядке возрастания рангов переславших их процессов (включая число, полученное из этого же процесса).

MPI3Coll12. В каждом процессе даны четыре целых числа. Используя функцию MPI_Allgather, переслать эти числа во все процессы и вывести их в каждом процессе в порядке возрастания рангов переславших их процессов (включая числа, полученные из этого же процесса).

MPI3Coll13. В каждом процессе дан набор из $R + 2$ целых чисел, где число R равно рангу процесса (в процессе 0 даны 2 числа, в процессе 1 даны 3 числа, и т. д.). Используя функцию MPI_Allgatherv, переслать эти наборы во все процессы и вывести их в порядке возрастания рангов переславших их процессов (включая числа, полученные из этого же процесса).

MPI3Coll14. В каждом процессе дан набор из K чисел, где K — количество процессов. Используя функцию MPI_Alltoall, переслать в каждый процесс по одному числу из всех наборов: в процесс 0 — первые числа из наборов, в процесс 1 — вторые числа, и т. д. В каждом процессе вывести числа в порядке возрастания рангов переславших их процессов (включая число, полученное из этого же процесса).

MPI3Coll15. В каждом процессе дан набор из $3K$ целых чисел, где K — количество процессов. Используя функцию MPI_Alltoall, переслать в каждый процесс три очередных числа из каждого набора (в процесс 0 — первые три числа, в процесс 1 — следующие три числа, и т. д.). В каждом процессе вывести числа в порядке возрастания рангов переславших их процессов (включая числа, полученные из этого же процесса).

MPI3Coll16. В каждом процессе дан набор из $K(K + 1)/2$ целых чисел, где K — количество процессов. Используя функцию MPI_Alltoallv, переслать в каждый процесс часть чисел из каждого набора; при этом в процесс R должно быть переслано $R + 1$ очередное число (в процесс 0 — первое число каждого набора, в процесс 1 — следующие два числа, и т. д.). В каждом процессе вывести полученные числа.

MPI3Coll17. В каждом процессе дан набор из $K + 1$ числа, где K — количество процессов. Используя функцию MPI_Alltoallv, переслать в каждый процесс два числа из каждого набора; при этом в процесс 0 надо переслать первые два числа, в процесс 1 — второе и третье число, …, в последний процесс — последние два числа каждого набора. В каждом процессе вывести полученные числа.

MPI3Coll18. В каждом процессе дан набор из $K + 1$ числа, где K — количество процессов. Используя функцию MPI_Alltoallv, переслать в ка-

ждый процесс два числа из каждого набора; при этом в процесс 0 надо переслать последние два числа, в процесс 1 — два числа, предшествующих последнему, …, в последний процесс — первые два числа каждого набора. В каждом процессе вывести полученные числа.

12.2. Коллективные операции редукции

MPI3Coll19. В каждом процессе дан набор из $K + 5$ целых чисел, где K — количество процессов. Используя функцию MPI_Reduce для операции MPI_SUM, просуммировать элементы данных наборов с одним и тем же порядковым номером и вывести полученные суммы в главном процессе.

MPI3Coll20. В каждом процессе дан набор из $K + 5$ чисел, где K — количество процессов. Используя функцию MPI_Reduce для операции MPI_MIN, найти минимальное значение среди элементов данных наборов с одним и тем же порядковым номером и вывести полученные минимумы в главном процессе.

MPI3Coll21. В каждом процессе дан набор из $K + 5$ целых чисел, где K — количество процессов. Используя функцию MPI_Reduce для операции MPI_MAXLOC, найти максимальное значение среди элементов данных наборов с одним и тем же порядковым номером и ранг процесса, содержащего это максимальное значение. Вывести в главном процессе вначале все максимумы, а затем — ранги содержащих их процессов.

MPI3Coll22. В каждом процессе дан набор из $K + 5$ чисел, где K — количество процессов. Используя функцию MPI_Allreduce для операции MPI_PROD, перемножить элементы данных наборов с одним и тем же порядковым номером и вывести полученные произведения во всех процессах.

MPI3Coll23. В каждом процессе дан набор из $K + 5$ чисел, где K — количество процессов. Используя функцию MPI_Allreduce для операции MPI_MINLOC, найти минимальное значение среди элементов данных наборов с одним и тем же порядковым номером и ранг процесса, содержащего минимальное значение. Вывести в главном процессе минимумы, а в остальных процессах — ранги процессов, содержащих эти минимумы.

MPI3Coll24. В каждом процессе дан набор из K целых чисел, где K — количество процессов. Используя функцию MPI_Reduce_scatter, просуммировать элементы данных наборов с одним и тем же порядковым номером, переслать по одной из полученных сумм в каждый процесс (первую сумму — в процесс 0, вторую — в процесс 1, и т. д.) и вывести в каждом процессе полученную сумму.

MPI3Coll25. В каждом процессе дан набор из $2K$ чисел, где K — количество процессов. Используя функцию MPI_Reduce_scatter, найти максимумы среди элементов этих наборов с одним и тем же порядковым номером, переслать по два найденных максимума в каждый процесс (первые два максимума — в процесс 0, следующие два — в процесс 1, и т. д.) и вывести в каждом процессе полученные данные.

MPI3Coll26. В каждом процессе дан набор из $K(K + 3)/2$ целых чисел, где K — количество процессов. Используя функцию MPI_Reduce_scatter, найти минимальные значения среди элементов этих наборов с одним и тем же порядковым номером и переслать первые два минимума в процесс 0, следующие три — в процесс 1, …, последние $K + 1$ минимумов — в процесс $K - 1$. Вывести в каждом процессе полученные данные.

MPI3Coll27. В каждом процессе дан набор из $K + 5$ чисел, где K — количество процессов. Используя функцию MPI_Scan, найти в процессе ранга R ($R = 0, 1, \ldots, K - 1$) произведения элементов с одним и тем же порядковым номером для наборов, данных в процессах с рангами от 0 до R, и вывести найденные произведения (при этом в процессе $K - 1$ будут выведены произведения элементов из всех наборов).

MPI3Coll28. В каждом процессе дан набор из $K + 5$ целых чисел, где K — количество процессов. Используя функцию MPI_Scan, найти в процессе ранга R ($R = 0, \ldots, K - 1$) максимальные значения среди элементов с одним и тем же порядковым номером для наборов, данных в процессах с рангами от 0 до R, и вывести в каждом процессе найденные максимумы.

13. Производные типы и упаковка данных

13.1. Использование простейших производных типов

MPI4Type1. В главном процессе дана $K - 1$ тройка целых чисел, где K — количество процессов. Используя производный тип, содержащий три целых числа, и одну коллективную операцию пересылки данных, переслать все данные из главного процесса в подчиненные и вывести их в подчиненных процессах в том же порядке.

MPI4Type2. В главном процессе дана $K - 1$ тройка целых чисел, где K — количество процессов. Используя производный тип, содержащий три целых числа, и одну коллективную операцию пересылки данных, переслать по одной тройке чисел в каждый из подчиненных процессов и вывести их в подчиненных процессах в том же порядке.

MPI4Type3. В каждом подчиненном процессе дана тройка целых чисел. Используя производный тип, содержащий три целых числа, и одну

коллективную операцию пересылки данных, переслать числа из подчиненных процессов в главный и вывести полученные числа в порядке возрастания рангов переславших их процессов.

MPI4Type4. В главном процессе дана $K - 1$ тройка чисел, где K — количество процессов, причем первые два числа каждой тройки являются целыми, а третье число — вещественным. Используя производный тип, содержащий три числа (два целых и одно вещественное), переслать числа из главного процесса в подчиненные и вывести их в подчиненных процессах в том же порядке.

MPI4Type5. В главном процессе дана $K - 1$ тройка чисел, где K — количество процессов, причем первое и третье число каждой тройки являются целыми, а второе число — вещественным. Используя производный тип, содержащий три числа (целое, вещественное, целое), переслать по одной тройке чисел в каждый из подчиненных процессов и вывести их в подчиненных процессах в том же порядке.

MPI4Type6. В каждом подчиненном процессе даны три числа: одно вещественное и два целых. Используя производный тип, содержащий три числа (одно вещественное и два целых), переслать числа из подчиненных процессов в главный и вывести полученные числа в порядке возрастания рангов переславших их процессов.

MPI4Type7. В каждом процессе даны три числа: первое и третье являются целыми, а второе — вещественным. Используя производный тип, содержащий три числа (целое, вещественное, целое), переслать данные из каждого процесса во все процессы и вывести в каждом процессе полученные числа в порядке возрастания рангов переславших их процессов (включая числа, полученные из этого же процесса).

MPI4Type8. В каждом подчиненном процессе даны R троек чисел, где R — ранг процесса. Два первых числа в каждой тройке являются целыми, а последнее — вещественным. Используя производный тип, содержащий три числа (два целых и одно вещественное), переслать числа из подчиненных процессов в главный и вывести полученные числа в порядке возрастания рангов переславших их процессов.

13.2. Пересылка упакованных данных

MPI4Type9. В главном процессе даны два набора: первый содержит K целых, а второй K вещественных чисел, где K — количество процессов. Используя функции упаковки MPI_Pack и MPI_Unpack и одну коллективную операцию пересылки данных, переслать все данные из главного процесса в подчиненные и вывести их в подчиненных процессах в том же порядке.

MPI4Type10. В главном процессе дана $K - 1$ тройка чисел, где K — количество процессов, причем первое и третье число каждой тройки является целым, а второе число — вещественным. Используя функции упаковки и одну коллективную операцию пересылки данных, переслать по одной тройке чисел в подчиненные процессы и вывести их в подчиненных процессах в том же порядке.

MPI4Type11. В главном процессе дана $K - 1$ тройка чисел, где K — количество процессов, причем первые два числа каждой тройки являются целыми, а третье число — вещественным. Используя функции упаковки и одну коллективную операцию пересылки данных, переслать все данные из главного процесса в подчиненные и вывести их в подчиненных процессах в том же порядке.

MPI4Type12. В каждом подчиненном процессе даны три числа: два целых и одно вещественное. Используя функции упаковки и одну коллективную операцию пересылки данных, переслать числа из подчиненных процессов в главный и вывести полученные числа в порядке возрастания рангов переславших их процессов.

MPI4Type13. В каждом подчиненном процессе дан набор из одного вещественного и R целых чисел, где значение R равно рангу процесса (в процессе 1 дано одно целое число, в процессе 2 — два целых числа, и т. д.). Используя функции упаковки и одну функцию передачи и приема, переслать все данные в главный процесс и вывести эти данные в порядке возрастания рангов переславших их процессов.

13.3. Более сложные виды производных типов

MPI4Type14. В главном процессе даны два набора целых чисел: A размера $3K$ и N размера K, где K — количество подчиненных процессов. Считая, что элементы наборов нумеруются от 1, переслать и вывести в каждом подчиненном процессе ранга R ($R = 1, 2, \ldots, K$) N_R элементов из набора A, начиная с элемента A_R и перебирая их через один (например, если N_2 равно 3, то в процесс ранга 2 надо переслать элементы A_2, A_4, A_6). Использовать для пересылки каждого набора элементов по одному вызову функций MPI_Send, MPI_Probe и MPI_Recv, причем функция MPI_Recv должна возвращать массив, содержащий только те элементы, которые требуется вывести. Для этого в главном процессе определить новый тип, содержащий единственный целочисленный элемент и дополнительный конечный *пустой промежуток*, равный размеру элемента целого типа. Использовать в функции MPI_Send исходный массив A с требуемым смещением, указав в качестве второго параметра количество элементов, равное N_R, а в качестве третьего параметра — новый тип. В функции MPI_Recv использовать целочис-

ленный массив размера N_R и тип MPI_INT. Для определения количества N_R переданных элементов использовать в подчиненных процессах функцию MPI_Get_count.

Указание. Для задания завершающего пустого промежутка при определении нового типа в MPI-2 следует использовать функцию MPI_Type_create_resized (в данном случае эту функцию надо применить к типу MPI_INT). В MPI-1 надо использовать *метку нулевого размера* типа MPI_UB совместно с функцией MPI_Type_struct (в стандарте MPI-2 тип MPI_UB объявлен устаревшим).

MPI4Type15. В главном процессе дана вещественная квадратная матрица порядка K, где K — количество подчиненных процессов (матрица должна храниться в одномерном массиве A, в котором элементы матрицы располагаются по строкам). Считая, что столбцы матрицы нумеруются от 1, переслать и вывести в каждом подчиненном процессе ранга R (R = 1, 2, …, K) элементы столбца матрицы с номером R. Использовать по одному вызову функций MPI_Send и MPI_Recv, причем функция MPI_Recv должна возвращать массив, содержащий только те элементы, которые требуется вывести. Для этого в главном процессе определить новый тип, содержащий единственный вещественный элемент и дополнительный конечный пустой промежуток требуемого размера. Использовать в функции MPI_Send массив A с требуемым смещением, указав в качестве второго параметра число K (т. е. количество элементов в каждом столбце), а в качестве третьего параметра — новый тип. В функции MPI_Recv использовать вещественный массив размера K и тип MPI_DOUBLE.

Указание. См. указание к задаче MPI4Type14.

MPI4Type16. В каждом из подчиненных процессов дан столбец вещественной квадратной матрицы порядка K, где K — количество подчиненных процессов, причем в процессе ранга R (R = 1, …, K) содержится столбец с номером R, если считать, что столбцы нумеруются от 1. Переслать все столбцы матрицы в главный процесс, разместив их в одномерном массиве A, в котором элементы матрицы хранятся по строкам, и вывести полученный массив A. Для пересылки каждого столбца использовать по одному вызову функций MPI_Send и MPI_Recv, причем функция MPI_Recv должна в качестве первого параметра содержать массив A (с требуемым смещением), а в качестве второго параметра — число 1. Для этого в главном процессе определить новый тип, содержащий K вещественных элементов, после каждого из которых следует пустой промежуток требуемого размера. Определение нового типа провести в два этапа: на первом создать вспомогательный тип, состоящий из одного вещественного числа и требуемого конечного пустого промежутка (см. указание к MPI4Type14),

на втором — создать на его основе итоговый тип с помощью функции MPI_Type_contiguous, который использовать в качестве третьего параметра функции MPI_Recv (функцию MPI_Type_commit достаточно вызвать только для итогового типа). В функции MPI_Send использовать вещественный массив размера K и тип MPI_DOUBLE.

MPI4Type17. Количество подчиненных процессов K кратно 3 и не превосходит 9. В каждом процессе дано целое число N; все числа N одинаковы и лежат в диапазоне от 3 до 5. Кроме того, в каждом подчиненном процессе дана целочисленная квадратная матрица порядка N (*блок*), которую следует прочесть в одномерный массив B по строкам. Переслать все массивы B в главный процесс и составить из них *блочную матрицу* размера (K/3) × 3 (размер указан в блоках), располагая блоки по строкам (первая строка блоков должна содержать блоки, полученные из процессов ранга 1, 2, 3, вторая — из процессов ранга 4, 5, 6, и т. д.). Блочная матрица должна храниться по строкам в одномерном массиве A. Использовать для пересылки каждого блока B по одному вызову функций MPI_Send и MPI_Recv, причем функция MPI_Recv должна в качестве первого параметра содержать массив A (с требуемым смещением), а в качестве второго параметра — число 1. Для этого в главном процессе определить новый тип, содержащий N наборов элементов, причем каждый набор содержит N целых чисел, а между наборами располагается пустой промежуток требуемого размера (при определении нового типа использовать функцию MPI_Type_vector). Указать созданный тип в качестве третьего параметра функции MPI_Recv. В функции MPI_Send использовать целочисленный массив B (размера $N \cdot N$) и тип MPI_INT.

MPI4Type18. Количество подчиненных процессов K кратно 3 и не превосходит 9. В главном процессе дано целое число N, лежащее в диапазоне от 3 до 5, и целочисленная блочная матрица порядка (K/3) × 3 (размер указан в блоках). Каждый блок представляет собой *нижнетреугольную* квадратную матрицу порядка N. Блочную матрицу следует прочесть в одномерный массив A по строкам. Переслать в каждый подчиненный процесс ненулевую часть соответствующего блока исходной матрицы, перебирая блоки по строкам (блоки из первой строки должны пересылаться в процессы ранга 1, 2, 3, блоки из второй строки — в процессы ранга 4, 5, 6, и т. д.), и вывести в каждом подчиненном процессе полученные данные (по строкам). Использовать для пересылки каждого блока по одному вызову функций MPI_Send, MPI_Probe и MPI_Recv, причем функция MPI_Send должна в качестве первого параметра содержать массив A (с требуемым смещением), а в качестве второго параметра — число 1. Для этого в главном процессе определить новый тип, содержащий N наборов элементов, причем каждый

набор содержит ненулевую часть очередной строки блока (первый набор состоит из одного элемента, второй из двух, и т. д.), а между наборами располагается пустой промежуток требуемого размера (при определении нового типа использовать функцию MPI_Type_indexed). Указать созданный тип в качестве третьего параметра функции MPI_Send. В функции MPI_Recv использовать целочисленный массив *B* (содержащий ненулевую часть полученного блока) и тип MPI_INT. Для определения количества переданных элементов использовать в подчиненных процессах функцию MPI_Get_count.

MPI4Type19. Количество подчиненных процессов *K* кратно 3 и не превосходит 9. В каждом процессе дано целое число *N*; все числа *N* одинаковы и лежат в диапазоне от 3 до 5. Кроме того, в каждом подчиненном процессе дано целое число *P* и ненулевые элементы целочисленной квадратной матрицы порядка *N* (*Z-блока*), которую следует прочесть в одномерный массив *B* по строкам. Ненулевые элементы располагаются в *Z*-блоке в форме символа «Z», т. е. занимают первую и последнюю строку, а также побочную диагональ. Определить в главном процессе нулевую целочисленную прямоугольную матрицу размера $N\cdot(K/3) \times 3N$, создав для этого одномерный массив *A* и обнулив его элементы (которые должны храниться в массиве *A* по строкам). Перебирая подчиненные процессы в порядке возрастания их рангов, переслать из каждого подчиненного процесса данную в нем ненулевую часть *Z*-блока и записать полученный блок в массив *A*, начиная с элемента массива *A* с индексом *P* (позиции *Z*-блоков могут перекрываться; в этом случае элементы блоков, полученных из процессов большего ранга, будут заменять некоторые из элементов ранее записанных блоков). Использовать для пересылки ненулевой части каждого *Z*-блока по одному вызову функций MPI_Send и MPI_Recv, причем функция MPI_Recv должна в качестве первого параметра содержать массив *A* (с требуемым смещением), а в качестве второго параметра — число 1. Для этого в главном процессе определить новый тип, содержащий *N* наборов элементов, причем первый и последний набор содержат по *N* целых чисел, остальные наборы — по одному числу, а между наборами располагается пустой промежуток требуемого размера (при определении нового типа использовать функцию MPI_Type_indexed). Указать созданный тип в качестве третьего параметра функции MPI_Recv. В функции MPI_Send использовать целочисленный массив *B*, содержащий ненулевую часть *Z*-блока (по строкам), и тип MPI_INT.

Указание. Для передачи в главный процесс позиции *P* вставки *Z*-блока используйте параметр msgtag; для этого в подчиненных процессах задавайте значение *P* в качестве параметра msgtag функции

MPI_Send, а в главном процессе до вызова функции MPI_Recv вызывайте функцию MPI_Probe с параметром MPI_ANY_TAG и анализируйте возвращенный ею параметр типа MPI_Status.

MPI4Type20. Количество подчиненных процессов *K* кратно 3 и не превосходит 9. В каждом процессе дано целое число *N*; все числа *N* одинаковы и лежат в диапазоне от 3 до 5. Кроме того, в каждом подчиненном процессе дано целое число *P* и ненулевые элементы целочисленной квадратной матрицы порядка *N* (*U-блока*), которую следует прочесть в одномерный массив *B* по строкам. Ненулевые элементы располагаются в *U*-блоке в форме символа «U», т. е. занимают первый и последний столбец, а также последнюю строку. Определить в главном процессе нулевую целочисленную прямоугольную матрицу размера $N \cdot (K/3) \times 3N$, создав для этого одномерный массив *A* и обнулив его элементы (которые должны храниться в массиве *A* по строкам). Перебирая подчиненные процессы в порядке возрастания их рангов, переслать из каждого подчиненного процесса данную в нем ненулевую часть *U*-блока и записать полученный блок в массив *A*, начиная с элемента массива *A* с индексом *P* (позиции *U*-блоков могут перекрываться; в этом случае элементы блоков, полученных из процессов большего ранга, будут заменять некоторые из элементов ранее записанных блоков). Использовать для пересылки ненулевой части каждого *U*-блока по одному вызову функций MPI_Send и MPI_Recv, причем функция MPI_Recv должна в качестве первого параметра содержать массив *A* (с требуемым смещением), а в качестве второго параметра — число 1. Для этого в главном процессе определить новый тип, содержащий требуемое количество наборов элементов с пустыми промежутками между ними (при определении нового типа использовать функцию MPI_Type_indexed). Указать созданный тип в качестве третьего параметра функции MPI_Recv. В функции MPI_Send использовать целочисленный массив *B*, содержащий ненулевую часть *U*-блока (по строкам), и тип MPI_INT.

Указание. См. указание к задаче MPI4Type19.

13.4. Коллективная функция MPI_Alltoallw (MPI-2)

MPI4Type21. Решить задачу MPI4Type15, используя для пересылки данных вместо функций MPI_Send и MPI_Recv одну коллективную операцию.

Указание. Использовать функции группы Scatter не удастся, так как значения смещений для рассылаемых элементов (столбцов матрицы) требуется указывать не в элементах, а в *байтах*. Поэтому подходящим вариантом будет введенная в MPI-2 функция MPI_Alltoallw, позволяющая наиболее гибким образом настраивать вариант коллективной

пересылки. В данном случае с ее помощью следует реализовать вариант рассылки вида Scatter (при этом большинство параметров-массивов, используемых в этой функции, необходимо по-разному определять в главном и подчиненных процессах).

MPI4Type22. Решить задачу MPI4Type16, используя для пересылки данных вместо функций MPI_Send и MPI_Recv одну коллективную операцию.

Указание. См. указание к задаче MPI4Type21. В данном случае с помощью функции MPI_Alltoallw следует реализовать вариант рассылки вида Gather.

14. Группы процессов и коммуникаторы

14.1. Создание новых коммуникаторов

MPI5Comm1. В главном процессе дан набор из *K* целых чисел, где *K* — количество процессов четного ранга (0, 2, …). С помощью функций MPI_Comm_group, MPI_Group_incl и MPI_Comm_create создать новый коммуникатор, включающий процессы четного ранга. Используя одну коллективную операцию пересылки данных для созданного коммуникатора, переслать по одному исходному числу в каждый процесс четного ранга (включая главный) и вывести полученные числа.

MPI5Comm2. В каждом процессе нечетного ранга (1, 3, …) даны два вещественных числа. С помощью функций MPI_Comm_group, MPI_Group_excl и MPI_Comm_create создать новый коммуникатор, включающий процессы нечетного ранга. Используя одну коллективную операцию пересылки данных для созданного коммуникатора, переслать исходные числа во все процессы нечетного ранга и вывести эти числа в порядке возрастания рангов переславших их процессов (включая числа, полученные из этого же процесса).

MPI5Comm3. В каждом процессе, ранг которого делится на 3 (включая главный процесс), даны три целых числа. С помощью функции MPI_Comm_split создать новый коммуникатор, включающий процессы, ранг которых делится на 3. Используя одну коллективную операцию пересылки данных для созданного коммуникатора, переслать исходные числа в главный процесс и вывести эти числа в порядке возрастания рангов переславших их процессов (включая числа, полученные из главного процесса).

Указание. При вызове функции MPI_Comm_split в процессах, которые не требуется включать в новый коммуникатор, в качестве параметра color следует указывать константу MPI_UNDEFINED.

MPI5Comm4. В каждом процессе четного ранга (включая главный процесс) дан набор из трех элементов — вещественных чисел. Используя новый коммуникатор и одну коллективную операцию редукции, найти минимальные значения среди элементов исходных наборов с одним и тем же порядковым номером и вывести найденные минимумы в главном процессе. Новый коммуникатор создать с помощью функции MPI_Comm_split.

Указание. См. указание к задаче MPI5Comm3.

MPI5Comm5. В каждом процессе дано вещественное число. Используя функцию MPI_Comm_split и одну коллективную операцию редукции, найти максимальное из чисел, данных в процессах с четным рангом (включая главный процесс), и минимальное из чисел, данных в процессах с нечетным рангом. Найденный максимум вывести в процессе 0, а найденный минимум — в процессе 1.

Указание. Программа должна содержать единственный вызов функции MPI_Comm_split, создающий оба требуемых коммуникатора (каждый в соответствующей группе процессов).

MPI5Comm6. В главном процессе дано целое число K и набор из K вещественных чисел, в каждом подчиненном процессе дано целое число N, которое может принимать два значения: 0 и 1 (количество подчиненных процессов с $N = 1$ равно K). Используя функцию MPI_Comm_split и одну коллективную операцию пересылки данных, переслать по одному вещественному числу из главного процесса в каждый подчиненный процесс с $N = 1$ и вывести в этих подчиненных процессах полученные числа.

Указание. См. указание к задаче MPI5Comm3.

MPI5Comm7. В каждом процессе дано целое число N, которое может принимать два значения: 0 и 1 (имеется хотя бы один процесс с $N = 1$). Кроме того, в каждом процессе с $N = 1$ дано вещественное число A. Используя функцию MPI_Comm_split и одну коллективную операцию пересылки данных, переслать числа A в первый из процессов с $N = 1$ и вывести их в порядке возрастания рангов переславших их процессов (включая число, полученное из этого же процесса).

Указание. См. указание к задаче MPI5Comm3.

MPI5Comm8. В каждом процессе дано целое число N, которое может принимать два значения: 0 и 1 (имеется хотя бы один процесс с $N = 1$). Кроме того, в каждом процессе с $N = 1$ дано вещественное число A. Используя функцию MPI_Comm_split и одну коллективную операцию пересылки данных, переслать числа A в последний из процессов с $N = 1$ и вывести их в порядке возрастания рангов переславших их процессов (включая число, полученное из этого же процесса).

Указание. См. указание к задаче MPI5Comm3.

MPI5Comm9. В каждом процессе дано целое число N, которое может принимать два значения: 0 и 1 (имеется хотя бы один процесс с $N = 1$). Кроме того, в каждом процессе с $N = 1$ дано вещественное число A. Используя функцию MPI_Comm_split и одну коллективную операцию пересылки данных, переслать числа A во все процессы с $N = 1$ и вывести их в порядке возрастания рангов переславших их процессов (включая число, полученное из этого же процесса).

Указание. См. указание к задаче MPI5Comm3.

MPI5Comm10. В каждом процессе дано целое число N, которое может принимать два значения: 1 и 2 (имеется хотя бы один процесс с каждым из возможных значений). Кроме того, в каждом процессе дано целое число A. Используя функцию MPI_Comm_split и одну коллективную операцию пересылки данных, переслать числа A, данные в процессах с $N = 1$, во все процессы с $N = 1$, а числа A, данные в процессах с $N = 2$, во все процессы с $N = 2$. Во всех процессах вывести полученные числа в порядке возрастания рангов переславших их процессов (включая число, полученное из этого же процесса).

Указание. См. указание к задаче MPI5Comm5.

MPI5Comm11. В каждом процессе дано целое число N, которое может принимать два значения: 0 и 1 (имеется хотя бы один процесс с $N = 1$). Кроме того, в каждом процессе с $N = 1$ дано вещественное число A. Используя функцию MPI_Comm_split и одну коллективную операцию редукции, найти сумму всех исходных чисел A и вывести ее во всех процессах с $N = 1$.

Указание. См. указание к задаче MPI5Comm3.

MPI5Comm12. В каждом процессе дано целое число N, которое может принимать два значения: 1 и 2 (имеется хотя бы один процесс с каждым из возможных значений). Кроме того, в каждом процессе дано вещественное число A. Используя функцию MPI_Comm_split и одну коллективную операцию редукции, найти минимальное значение среди чисел A, которые даны в процессах с $N = 1$, и максимальное значение среди чисел A, которые даны в процессах с $N = 2$. Найденный минимум вывести в процессах с $N = 1$, а найденный максимум — в процессах с $N = 2$.

Указание. См. указание к задаче MPI5Comm5.

14.2. Виртуальные топологии

MPI5Comm13. В главном процессе дано целое число N (> 1), причем известно, что количество процессов K делится на N. Переслать число N во все процессы, после чего, используя функцию MPI_Cart_create, оп-

ределить для всех процессов декартову топологию в виде двумерной решетки — матрицы размера $N \times K/N$ (порядок нумерации процессов оставить прежним). Используя функцию MPI_Cart_coords, вывести для каждого процесса его координаты в созданной топологии.

MPI5Comm14. В главном процессе дано целое число N (> 1), не превосходящее количества процессов K. Переслать число N во все процессы, после чего определить декартову топологию для начальной части процессов в виде двумерной решетки — матрицы размера $N \times K/N$ (символ «/» обозначает операцию деления нацело, порядок нумерации процессов следует оставить прежним). Для каждого процесса, включенного в декартову топологию, вывести его координаты в этой топологии.

MPI5Comm15. Число процессов K является четным: $K = 2N$, $N > 1$. В процессах 0 и N дано по одному вещественному числу A. Определить для всех процессов декартову топологию в виде матрицы размера $2 \times N$, после чего, используя функцию MPI_Cart_sub, расщепить матрицу процессов на две одномерные строки (при этом процессы 0 и N будут главными процессами в полученных строках). Используя одну коллективную операцию пересылки данных, переслать число A из главного процесса каждой строки во все процессы этой же строки и вывести полученное число в каждом процессе (включая процессы 0 и N).

MPI5Comm16. Число процессов K является четным: $K = 2N$, $N > 1$. В процессах 0 и 1 дано по одному вещественному числу A. Определить для всех процессов декартову топологию в виде матрицы размера $N \times 2$, после чего, используя функцию MPI_Cart_sub, расщепить матрицу процессов на два одномерных столбца (при этом процессы 0 и 1 будут главными процессами в полученных столбцах). Используя одну коллективную операцию пересылки данных, переслать число A из главного процесса каждого столбца во все процессы этого же столбца и вывести полученное число в каждом процессе (включая процессы 0 и 1).

MPI5Comm17. Число процессов K кратно трем: $K = 3N$, $N > 1$. В процессах 0, N и $2N$ дано по N целых чисел. Определить для всех процессов декартову топологию в виде матрицы размера $3 \times N$, после чего, используя функцию MPI_Cart_sub, расщепить матрицу процессов на три одномерные строки (при этом процессы 0, N и $2N$ будут главными процессами в полученных строках). Используя одну коллективную операцию пересылки данных, переслать по одному исходному числу из главного процесса каждой строки во все процессы этой же строки и вывести полученное число в каждом процессе (включая процессы 0, N и $2N$).

MPI5Comm18. Число процессов K кратно трем: $K = 3N$, $N > 1$. В процессах 0, 1 и 2 дано по N целых чисел. Определить для всех процессов декартову топологию в виде матрицы размера $N \times 3$, после чего, используя функцию MPI_Cart_sub, расщепить матрицу процессов на три одномерных столбца (при этом процессы 0, 1 и 2 будут главными процессами в полученных столбцах). Используя одну коллективную операцию пересылки данных, переслать по одному исходному числу из главного процесса каждого столбца во все процессы этого же столбца и вывести полученное число в каждом процессе (включая процессы 0, 1 и 2).

MPI5Comm19. Количество процессов K равно 8 или 12, в каждом процессе дано целое число. Определить для всех процессов декартову топологию в виде трехмерной решетки размера $2 \times 2 \times K/4$ (порядок нумерации процессов оставить прежним). Интерпретируя эту решетку как две матрицы размера $2 \times K/4$ (в одну матрицу входят процессы с одинаковой первой координатой), расщепить каждую матрицу процессов на две одномерные строки. Используя одну коллективную операцию пересылки данных, переслать в главный процесс каждой строки исходные числа из процессов этой же строки и вывести полученные наборы чисел (включая числа, полученные из главных процессов строк).

MPI5Comm20. Количество процессов K равно 8 или 12, в каждом процессе дано целое число. Определить для всех процессов декартову топологию в виде трехмерной решетки размера $2 \times 2 \times K/4$ (порядок нумерации процессов оставить прежним). Интерпретируя полученную решетку как $K/4$ матриц размера 2×2 (в одну матрицу входят процессы с одинаковой третьей координатой), расщепить эту решетку на $K/4$ указанных матриц. Используя одну коллективную операцию пересылки данных, переслать в главный процесс каждой из полученных матриц исходные числа из процессов этой же матрицы и вывести полученные наборы чисел (включая числа, полученные из главных процессов матриц).

MPI5Comm21. Количество процессов K равно 8 или 12, в каждом процессе дано вещественное число. Определить для всех процессов декартову топологию в виде трехмерной решетки размера $2 \times 2 \times K/4$ (порядок нумерации процессов оставить прежним). Интерпретируя эту решетку как две матрицы размера $2 \times K/4$ (в одну матрицу входят процессы с одинаковой первой координатой), расщепить каждую матрицу процессов на $K/4$ одномерных столбцов. Используя одну коллективную операцию редукции, для каждого столбца процессов найти произведение исходных чисел и вывести найденные произведения в главных процессах каждого столбца.

MPI5Comm22. Количество процессов K равно 8 или 12, в каждом процессе дано вещественное число. Определить для всех процессов декартову топологию в виде трехмерной решетки размера $2 \times 2 \times K/4$ (порядок нумерации процессов оставить прежним). Интерпретируя полученную решетку как $K/4$ матриц размера 2×2 (в одну матрицу входят процессы с одинаковой третьей координатой), расщепить эту решетку на $K/4$ указанных матриц. Используя одну коллективную операцию редукции, для каждой из полученных матриц найти сумму исходных чисел и вывести найденные суммы в каждом процессе соответствующей матрицы.

MPI5Comm23. В главном процессе даны положительные целые числа M и N, произведение которых не превосходит количества процессов; кроме того, в процессах с рангами от 0 до $M \cdot N - 1$ даны целые числа X и Y. Переслать числа M и N во все процессы, после чего определить для начальных $M \cdot N$ процессов декартову топологию в виде двумерной решетки размера $M \times N$, являющейся периодической по первому измерению (порядок нумерации процессов оставить прежним). В каждом процессе, входящем в созданную топологию, вывести ранг процесса с координатами X, Y (с учетом периодичности), используя для этого функцию MPI_Cart_rank. В случае недопустимых координат вывести -1.

Примечание. Если при вызове функции MPI_Cart_rank указаны недопустимые координаты (например, отрицательные координаты для измерений, по которым декартова решетка не является периодической), то сама функция возвращает ненулевое значение (являющееся признаком ошибки), а возвращаемое значение параметра rank является неопределенным. Таким образом, в данном задании число -1 следует выводить, если функция MPI_Cart_rank вернула ненулевое значение. Чтобы подавить возникающие при этом сообщения об ошибках, отображаемые в разделе отладки окна задачника, достаточно перед вызовом функции, который может привести к ошибке, установить с помощью функции MPI_Comm_set_errhandler (MPI_Errhandler_set в MPI-1) специальный вариант *обработчика ошибок* MPI_ERROR_RETURN, который при возникновении ошибки не выполняет никаких действий, кроме установки ненулевого возвращаемого значения для этой функции. Следует заметить, что в версии MPICH 1.2.5 в случае указания недопустимых координат функция MPI_Cart_rank возвращает значение параметра rank, равное -1, что позволяет упростить решение, избавившись от проверки возвращаемого значения функции. Однако и в этом случае желательно подавить вывод сообщений об ошибках описанным выше способом.

MPI5Comm24. В главном процессе даны положительные целые числа M и N, произведение которых не превосходит количества процессов; кроме того, в процессах с рангами от 0 до $M \cdot N - 1$ даны целые числа X и Y. Переслать числа M и N во все процессы, после чего определить для начальных $M \cdot N$ процессов декартову топологию в виде двумерной решетки размера $M \times N$, являющейся периодической по второму измерению (порядок нумерации процессов оставить прежним). В каждом процессе, входящем в созданную топологию, вывести ранг процесса с координатами X, Y (с учетом периодичности), используя для этого функцию MPI_Cart_rank. В случае недопустимых координат вывести −1.

Примечание. См. примечание к заданию MPI5Comm23.

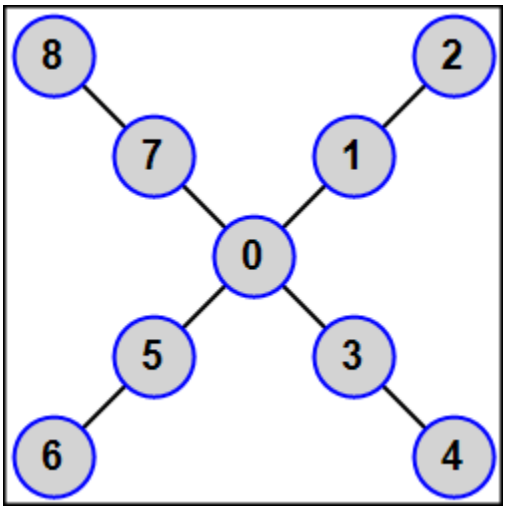
MPI5Comm25. В каждом подчиненном процессе дано вещественное число. Определить для всех процессов декартову топологию в виде одномерной решетки и осуществить простой сдвиг исходных данных с шагом −1 (число из каждого *подчиненного* процесса пересылается в предыдущий процесс). Для определения рангов посылающих и принимающих процессов использовать функцию MPI_Cart_shift, пересылку выполнять с помощью функций MPI_Send и MPI_Recv. Во всех процессах, получивших данные, вывести эти данные.

MPI5Comm26. Число процессов K является четным: $K = 2N$, $N > 1$; в каждом процессе дано вещественное число A. Определить для всех процессов декартову топологию в виде матрицы размера $2 \times N$ (порядок нумерации процессов оставить прежним) и для каждой строки матрицы осуществить циклический сдвиг исходных данных с шагом 1 (число A из каждого процесса, кроме последнего в строке, пересылается в следующий процесс этой же строки, а из последнего процесса — в главный процесс этой строки). Для определения рангов посылающих и принимающих процессов использовать функцию MPI_Cart_shift, пересылку выполнять с помощью функции MPI_Sendrecv. Во всех процессах вывести полученные данные.

MPI5Comm27. Количество процессов K равно 8 или 12, в каждом процессе дано вещественное число. Определить для всех процессов декартову топологию в виде трехмерной решетки размера $2 \times 2 \times K/4$ (порядок нумерации процессов оставить прежним). Интерпретируя полученную решетку как $K/4$ матриц размера 2×2 (в одну матрицу входят процессы с одинаковой третьей координатой, матрицы упорядочены по возрастанию третьей координаты), осуществить циклический сдвиг исходных данных из процессов каждой матрицы в соответствующие процессы следующей матрицы (из процессов последней матрицы данные перемещаются в первую матрицу). Для определения рангов посы-

лающих и принимающих процессов использовать функцию MPI_Cart_shift, пересылку выполнять с помощью функции MPI_Sendrecv_replace. Во всех процессах вывести полученные данные.

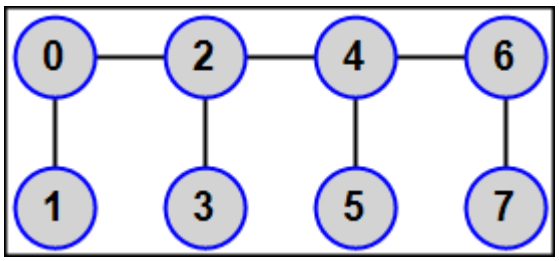
MPI5Comm28. Число процессов K является нечетным: $K = 2N + 1$ ($1 < N < 5$); в каждом процессе дано целое число A. Используя функцию MPI_Graph_create, определить для всех процессов топологию графа, в которой главный процесс связан ребрами со всеми процессами нечетного ранга (1, 3, …, $2N - 1$), а каждый процесс четного положительного ранга R (2, 4, …, $2N$) связан ребром с процессом ранга $R - 1$ (в результате получается N-лучевая звезда, центром которой является главный процесс, а каждый луч состоит из двух подчиненных процессов R и $R + 1$, причем ближайшим к центру является процесс нечетного ранга R).



Переслать число A из каждого процесса всем процессам, связанным с ним ребрами (*процессам-соседям*). Для определения количества процессов-соседей и их рангов использовать функции MPI_Graph_neighbors_count и MPI_Graph_neighbors, пересылку выполнять с помощью функций MPI_Send и MPI_Recv. Во всех процессах вывести полученные числа в порядке возрастания рангов переславших их процессов.

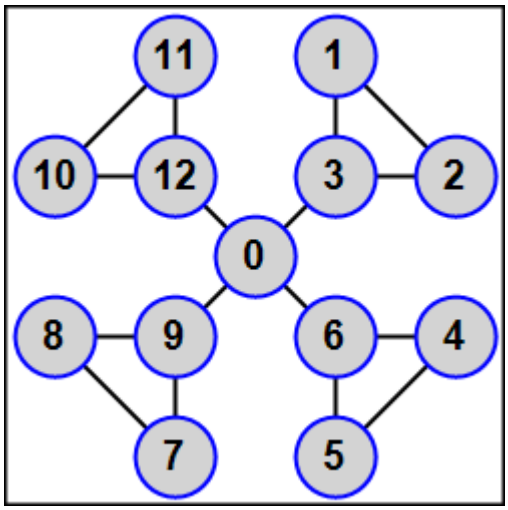
MPI5Comm29. Число процессов K является четным: $K = 2N$ ($1 < N < 6$); в каждом процессе дано целое число A. Используя функцию MPI_Graph_create, определить для всех процессов топологию графа, в которой все процессы четного ранга (включая главный процесс) связаны в цепочку: 0 — 2 — 4 — 6 — … — ($2N - 2$), а каждый процесс нечетного ранга R (1, 3, …, $2N - 1$) связан с процессом ранга $R - 1$ (в результате каждый процесс нечетного ранга будет иметь единственного соседа, первый и последний процессы четного ранга будут иметь

по два соседа, а остальные — «внутренние» — процессы четного ранга будут иметь по три соседа).



Переслать число *A* из каждого процесса всем процессам-соседям. Для определения количества процессов-соседей и их рангов использовать функции MPI_Graph_neighbors_count и MPI_Graph_neighbors, пересылку выполнять с помощью функции MPI_Sendrecv. Во всех процессах вывести полученные числа в порядке возрастания рангов переславших их процессов.

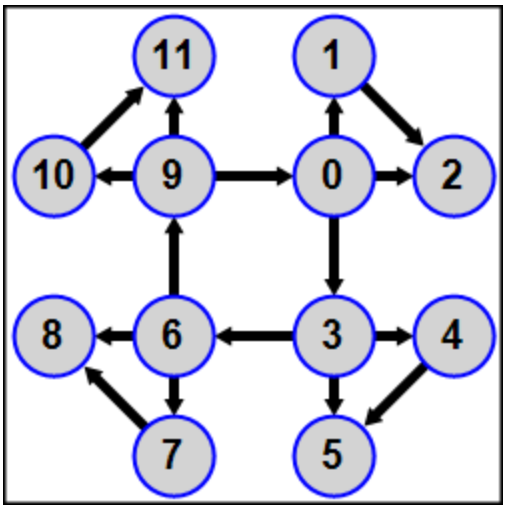
MPI5Comm30. Количество процессов *K* равно $3N + 1$ ($1 < N < 5$); в каждом процессе дано целое число *A*. Используя функцию MPI_Graph_create, определить для всех процессов топологию графа, в которой процессы R, $R + 1$, $R + 2$, где $R = 1, 4, 7, \ldots$, связаны между собой ребрами, и, кроме того, каждый процесс положительного ранга, кратного трем (3, 6, …), связан ребром с главным процессом (в результате получается *N*-лучевая звезда, центром которой является главный процесс, а каждый луч состоит из трех связанных между собой процессов, причем с центром связан процесс ранга, кратного трем).



Переслать число *A* из каждого процесса всем процессам-соседям. Для определения количества процессов-соседей и их рангов использовать функции MPI_Graph_neighbors_count и MPI_Graph_neighbors, пересылку выполнять с помощью функции MPI_Sendrecv. Во всех процессах вывести полученные числа в порядке возрастания рангов переславших их процессов.

14.3. Топология распределенного графа (MPI-2)

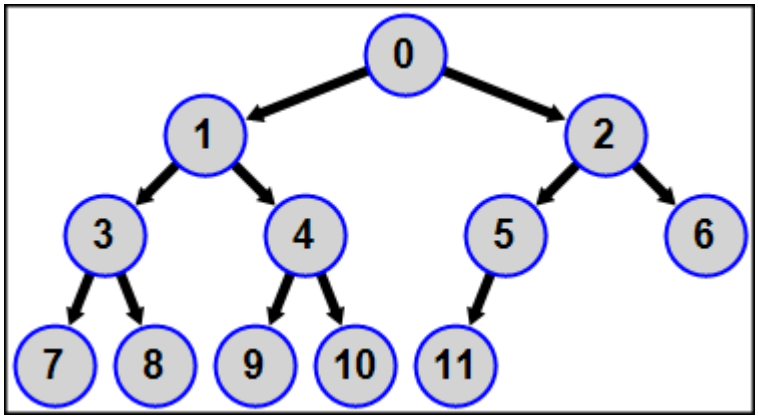
MPI5Comm31. Количество процессов K кратно 3; в каждом процессе дано целое число A. Используя функцию MPI_Dist_graph_create, определить для всех процессов топологию распределенного графа, в которой все процессы рангов, кратных 3 (0, 3, …, $K - 3$) образуют кольцо, причем каждый из процессов, входящих в кольцо, является источником для последующего процесса (процесс 0 является источником для процесса 3, процесс 3 — для процесса 6, …, процесс $K - 3$ — для процесса 0) и, кроме того, процесс ранга $3N$ ($N = 0, 1, …, K/3 - 1$) является источником для процессов ранга $3N + 1$ и $3N + 2$, а процесс ранга $3N + 1$ является источником для процесса ранга $3N + 2$.



Полное определение топологии дать в главном процессе (в подчиненных процессах второй параметр функции MPI_Dist_graph_create должен быть равен 0); параметр weights положить равным MPI_UNWEIGHTED, параметр info — равным MPI_INFO_NULL, порядок нумерации процессов не изменять. Переслать числа A от процессов-источников процессам-приемникам и вывести в каждом процессе сумму исходного числа A и всех чисел, полученных от процессов-источников. Для определения количества источников и приемников, а также их рангов использовать функции MPI_Dist_graph_neighbors_count и MPI_Dist_graph_neighbors, пересылку выполнять с помощью функций MPI_Send и MPI_Recv.

MPI5Comm32. Количество процессов лежит в диапазоне от 4 до 15, в каждом процессе дано целое число A. Используя функцию MPI_Dist_graph_create, определить для всех процессов топологию распределенного графа, которая представляет собой бинарное дерево с корнем в главном процессе 0, вершинами первого уровня в процессах рангов 1 и 2, вершинами второго уровня в процессах рангов 3-6 (причем процессы ранга 3 и 4 являются дочерними вершинами про-

цесса 1, а процессы ранга 5 и 6 — дочерними вершинами процесса 2), и т. д. Каждый процесс является источником для своих дочерних вершин; таким образом, каждый процесс имеет от 0 до 2 процессов-приемников.



Полное определение топологии дать в главном процессе (в подчиненных процессах второй параметр функции MPI_Dist_graph_create должен быть равен 0); параметр weights положить равным MPI_UNWEIGHTED, параметр info — равным MPI_INFO_NULL, порядок нумерации процессов не изменять. Найти и вывести в каждом процессе сумму исходного числа *A* и чисел, данных в процессах-предках *всех уровней* — от корня (главного процесса) до ближайшего предка (родительского процесса-источника). Для определения количества источников и приемников, а также их рангов использовать функции MPI_Dist_graph_neighbors_count и MPI_Dist_graph_neighbors, пересылку данных выполнять с помощью функций MPI_Send и MPI_Recv.

15. Параллельный ввод-вывод файловых данных (MPI-2)

Для хранения имени файла достаточно использовать массив char[12], а для его пересылки из главного процесса в подчиненные — функцию MPI_Bcast с параметром типа MPI_CHAR.

В первых двух подгруппах (задания MPI6File1–MPI6File16) не требуется настраивать вид файловых данных с помощью функции MPI_File_set_view; достаточно использовать вид по умолчанию, при котором базовые и файловые элементы имеют тип MPI_BYTE, начальное смещение для всех процессов равно 0 и используется представление «native». Это же представление надо указывать и при явной настройке вида файловых данных в заданиях третьей подгруппы (MPI6File17–MPI6File30).

Для определения размера типов MPI_INT и MPI_DOUBLE следует применять функцию MPI_Type_size.

Для указания завершающего пустого промежутка (в заданиях, связанных с настройкой вида файловых данных) следует использовать функцию MPI_Type_create_resized. Альтернативным вариантом является использование специального типа MPI_UB (*метки нулевого размера*), однако в стандарте MPI-2 этот тип объявлен устаревшим.

15.1. Локальные функции для файлового ввода-вывода

MPI6File1. В главном процессе дано имя существующего файла целых чисел. В каждом из подчиненных процессов дано количество файловых элементов, предназначенных для считывания в этом процессе, и порядковые номера этих элементов (элементы нумеруются от 1). Элементы с некоторыми номерами в исходном файле могут отсутствовать. Используя в каждом процессе требуемое количество вызовов локальной функции MPI_File_read_at, прочесть из файла существующие элементы с указанными номерами и вывести их в том же порядке. Для проверки наличия элемента с указанным порядковым номером можно либо использовать функцию MPI_File_get_size, либо анализировать информацию, возвращенную в параметре типа MPI_Status функции MPI_File_read_at.

MPI6File2. В главном процессе дано имя файла. В каждом из подчиненных процессов дано количество пар целых чисел и сами пары, в которых первый член равен порядковому номеру файлового элемента, а второй — значению этого файлового элемента (элементы нумеруются от 1, все номера различны и принимают все значения от 1 до некоторого целого числа). Создать новый файл целых чисел с указанным именем и записать в него данные элементы, используя в каждом подчиненном процессе требуемое количество вызовов локальной функции MPI_File_write_at.

MPI6File3. В главном процессе дано имя существующего файла вещественных чисел, содержащего элементы прямоугольной матрицы размера $K \times N$, где K — число подчиненных процессов. В каждом подчиненном процессе прочесть и вывести элементы R-й строки матрицы, где R — ранг подчиненного процесса (строки нумеруются от 1), используя один вызов локальной функции MPI_File_read_at. Для определения размера файла использовать функцию MPI_File_get_size.

MPI6File4. В главном процессе дано имя файла, в каждом из подчиненных процессов дано R вещественных чисел, где R — ранг процесса. Создать новый файл вещественных чисел с указанным именем и записать в него данные элементы в порядке возрастания рангов содержащих их процессов. Использовать в каждом подчиненном процессе один вызов локальной функции MPI_File_write_at.

MPI6File5. В главном процессе дано имя существующего файла целых чисел. Файл обязательно содержит все целые числа в диапазоне от 1 до *K*, где *K* — максимальный ранг процесса. В каждом подчиненном процессе прочесть и вывести два фрагмента файловых данных: первый содержит *начальную* часть элементов до первого элемента, равного рангу процесса (включая этот элемент), второй содержит *завершающую* часть файловых элементов и имеет тот же размер, что и первый фрагмент. Элементы в каждом фрагменте выводятся в порядке их следования в файле. Использовать в каждом подчиненном процессе необходимое количество вызовов локальной функции MPI_File_read (без применения массивов), а также вызовы функций MPI_File_get_position для определения размера первого фрагмента и MPI_File_seek с параметром MPI_SEEK_END для перемещения к началу второго фрагмента.

MPI6File6. В главном процессе дано имя файла, в каждом из подчиненных процессов дано целое число. Создать новый файл целых чисел с указанным именем и записать в него *K* подряд идущих копий каждого числа, где *K* равно количеству подчиненных процессов. Порядок следования чисел в файле должен быть обратным порядку подчиненных процессов (*K* копий числа из процесса 1 должны находиться в конце файла, перед ними должны располагаться *K* копий числа из процесса 2, и т. д.). Использовать в каждом подчиненном процессе один вызов локальной функции MPI_File_write, а также функцию MPI_File_seek с параметром MPI_SEEK_SET.

MPI6File7. В главном процессе дано имя существующего файла вещественных чисел. Известно, что сумма значений всех файловых элементов превосходит *K*, где *K* — количество подчиненных процессов. В каждом подчиненном процессе считывать начальные элементы файла, пока сумма их значений не превысит ранг процесса, после чего вывести найденную сумму *S* и количество *N* прочитанных чисел. После этого дополнительно прочесть и вывести значения *N последних* файловых элементов (в порядке их следования в файле). Использовать в каждом подчиненном процессе необходимое количество вызовов локальной функции MPI_File_read (без применения массивов), а также вызовы функций MPI_File_get_position для определения *N* и MPI_File_seek с параметром MPI_SEEK_END для перемещения к началу фрагмента из *N* последних файловых элементов.

MPI6File8. В главном процессе дано имя файла, в каждом из подчиненных процессов дано вещественное число. Создать новый файл вещественных чисел с указанным именем и записать в него *R* подряд идущих копий каждого числа, где *R* равно рангу процесса, в котором дано это

число. Порядок следования чисел в файле должен быть *обратным* порядку подчиненных процессов (в конце файла должно находиться единственное число из процесса 1, перед ним должны располагаться две копии числа из процесса 2, и т. д.). Использовать в каждом подчиненном процессе один вызов локальной функции MPI_File_write, а также функцию MPI_File_seek с параметром MPI_SEEK_SET.

15.2. Коллективные функции для файлового ввода-вывода

MPI6File9. В главном процессе дано имя существующего файла целых чисел. В каждом процессе прочесть и вывести $R + 1$ элемент файла, начиная с элемента с номером $R + 1$, где число R равно рангу процесса (0, 1, …). Файловые элементы нумеруются от 1; таким образом, в процессе ранга 0 требуется прочесть и вывести только начальный файловый элемент, в процессе ранга 1 — два следующих файловых элемента, в процессе ранга 2 — три элемента, начиная с третьего, и т. д. Если файл содержит недостаточное количество элементов, то в некоторых процессах число выведенных элементов может быть меньше требуемого. Использовать один вызов коллективной функции MPI_File_read_all, а также функцию MPI_File_seek с параметром MPI_SEEK_SET и функцию MPI_File_get_size для определения размера исходного файла.

Примечание. В реализации MPICH2 версии 1.3 функция MPI_File_read_all не позволяет определить количество фактически считанных файловых элементов на основе информации, содержащейся в параметре типа MPI_Status: в этом параметре количество прочитанных элементов всегда равно требуемому, а при недостаточном количестве файловых элементов в завершающую часть выходного массива записываются нулевые значения.

MPI6File10. В главном процессе дано имя файла, в каждом из подчиненных процессов дан набор из R целых чисел, где R — ранг процесса (1, 2, …). Создать новый файл целых чисел с указанным именем и записать в него все данные числа в порядке их следования (процессы перебираются в порядке возрастания их рангов). Использовать один вызов коллективной функции MPI_File_write_all (для всех процессов, в том числе и для процесса ранга 0), а также функцию MPI_File_seek с параметром MPI_SEEK_SET.

MPI6File11. В главном процессе дано имя существующего файла вещественных чисел. Кроме того, в каждом процессе дано целое число, равное либо 0, либо порядковому номеру одного из существующих файловых элементов (элементы нумеруются от 1). Используя функцию MPI_Comm_split, создать новый коммуникатор, содержащий только

те процессы, в которых дано число, отличное от 0, и с помощью коллективной функции MPI_File_read_at_all прочесть и вывести в каждом процессе из этого коммуникатора элемент, расположенный в позиции с указанным порядковым номером.

MPI6File12. В главном процессе дано имя существующего файла вещественных чисел. Кроме того, в каждом процессе дано целое число, равное либо 0, либо порядковому номеру одного из существующих файловых элементов (элементы нумеруются от 1). Используя функцию MPI_Comm_split, создать новый коммуникатор, содержащий только те процессы, в которых дано число, отличное от 0, и с помощью коллективной функции MPI_File_write_at_all заменить исходное значение файлового элемента в позиции с указанным порядковым номером на значение ранга процесса в новом коммуникаторе (преобразовав ранг в вещественное число).

MPI6File13. В главном процессе дано имя существующего файла целых чисел. Кроме того, в каждом процессе дано целое число, равное либо 0, либо 1. Используя функцию MPI_Comm_split, создать новый коммуникатор, содержащий только те процессы, в которых дано число 1, и с помощью коллективной функции MPI_File_read_ordered прочесть и вывести в каждом процессе из этого коммуникатора $R + 1$ элемент исходного файла, где R — ранг процесса в новом коммуникаторе (элементы считываются последовательно: один элемент в процессе 0, два следующих элемента — в процессе 1, три следующих — в процессе 2, и т. д.). Если файл содержит недостаточное количество элементов, то в некоторых процессах число выведенных элементов N может быть меньше требуемого или даже равно нулю. В каждом процессе, входящем в новый коммуникатор, дополнительно вывести количество N фактически прочитанных элементов и новое значение P коллективного файлового указателя. Для определения количества прочитанных элементов N использовать информацию, возвращенную в параметре типа MPI_Status функции MPI_File_read_ordered, для нахождения значения P использовать функцию MPI_File_get_position_shared (во всех процессах значение P должно быть одинаковым).

MPI6File14. В главном процессе дано имя файла. Кроме того, в каждом процессе дано целое число N. Создать новый файл целых чисел с указанным именем. Используя функцию MPI_Comm_split, создать новый коммуникатор, содержащий только те процессы, в которых число N не равно 0, и с помощью коллективной функции MPI_File_write_ordered записать в файл K подряд идущих копий каждого из чисел N, где K равно количеству процессов в *новом* коммуникаторе, а числа располагаются в порядке возрастания рангов содержащих их процессов.

MPI6File15. В главном процессе дано имя существующего файла целых чисел, содержащего не менее K элементов, где K — количество процессов. С помощью функции MPI_Comm_split создать новый коммуникатор, содержащий только процессы с нечетным рангом (1, 3, …). Используя по одному вызову коллективных функций MPI_File_seek_shared и MPI_File_read_ordered, прочесть и вывести в каждом процессе из нового коммуникатора по 2 элемента исходного файла, причем предпоследний и последний файловые элементы должны считываться и выводиться (в указанном порядке) в процессе с рангом 1 в исходном коммуникаторе MPI_COMM_WORLD, четвертый и третий с конца элементы — в процессе с рангом 3, и т. д.

Указание. Для того чтобы обеспечить требуемый порядок считывания данных в функции MPI_File_read_ordered, необходимо изменить в созданном коммуникаторе порядок следования процессов на противоположный (по сравнению с исходным порядком в коммуникаторе MPI_COMM_WORLD).

MPI6File16. В главном процессе дано имя файла. Кроме того, в каждом процессе дано целое число N (≥ 0) и N вещественных чисел. Создать новый файл целых чисел с указанным именем. Используя функцию MPI_Comm_split, создать новый коммуникатор, содержащий только те процессы, в которых число N не равно 0, и с помощью единственного вызова коллективной функции MPI_File_write_ordered записать в файл все вещественные числа в порядке, *обратном* их следованию в наборе исходных данных: вначале записываются (в обратном порядке) все числа из процесса с наибольшим рангом в коммуникаторе MPI_COMM_WORLD, затем все числа из процесса с предыдущим рангом, и т. д.

Указание. Для того чтобы обеспечить требуемый порядок записи данных в функции MPI_File_write_ordered, необходимо изменить в созданном коммуникаторе порядок следования процессов на противоположный (по сравнению с исходным порядком в коммуникаторе MPI_COMM_WORLD).

15.3. Настройка вида данных для файлового ввода-вывода

MPI6File17. В главном процессе дано имя существующего файла целых чисел, содержащего $2K$ элементов, где K — количество процессов. Используя единственный вызов коллективной функции MPI_File_read_all (и не применяя функцию MPI_File_seek), прочесть и вывести по 2 элемента в каждом процессе, перебирая элементы в порядке их следования в файле. Для этого предварительно с помощью функции MPI_File_set_view определить новый вид данных с базовым

типом MPI_INT, таким же файловым типом и подходящим смещением (своим для каждого процесса).

MPI6File18. В главном процессе дано имя существующего файла целых чисел, содержащего элементы прямоугольной матрицы размера $K \times 5$, где K — количество процессов. Кроме того, в каждом процессе дано целое число N ($1 \le N \le 5$) — порядковый номер *выделенного* элемента в некоторой строке матрицы (элементы в строке нумеруются от 1), причем с процессом ранга 0 связывается первая строка матрицы, с процессом ранга 1 — вторая строка, и т. д. Используя единственный вызов коллективной функции MPI_File_write_at_all со вторым параметром, равным $N - 1$, изменить в каждой строке исходной матрицы выделенный элемент, заменив его значение на ранг связанного с ним процесса (выделенный элемент в первой строке следует заменить на 0, во второй строке — на 1, и т. д.). Для этого предварительно с помощью функции MPI_File_set_view определить новый вид данных с базовым типом MPI_INT, таким же файловым типом и подходящим смещением (своим для каждого процесса).

MPI6File19. В главном процессе дано имя существующего файла вещественных чисел, содержащего элементы прямоугольной матрицы размера $K \times 6$, где K — количество процессов. Кроме того, в каждом процессе дано целое число N ($1 \le N \le 6$) — порядковый номер *выделенного* элемента в некоторой строке матрицы (элементы в строке нумеруются от 1), причем с процессом ранга 0 связывается последняя строка матрицы, с процессом ранга 1 — предпоследняя строка, и т. д. Используя единственный вызов коллективной функции MPI_File_read_at_all со вторым параметром, равным $N - 1$, прочесть в каждой строке исходной матрицы выделенный элемент и вывести его значение в соответствующем процессе (выделенный элемент в первой строке следует вывести в последнем процессе, во второй строке — в предпоследнем процессе, и т. д.). Для этого предварительно с помощью функции MPI_File_set_view определить новый вид данных с базовым типом MPI_DOUBLE, таким же файловым типом и подходящим смещением (своим для каждого процесса).

MPI6File20. В главном процессе дано имя файла. Кроме того, в каждом процессе дан набор из $R + 1$ вещественного числа, где R — ранг процесса (0, 1, …). Создать новый файл целых чисел с указанным именем. Используя единственный вызов коллективной функции MPI_File_write_all (и не применяя функцию MPI_File_seek), записать в файл все данные числа в порядке, обратном их следованию в исходном наборе: вначале записываются (в обратном порядке) все числа из процесса с наибольшим рангом, затем все числа из процесса с преды-

дущим рангом, и т. д. Для этого предварительно с помощью функции MPI_File_set_view определить новый вид данных с базовым типом MPI_DOUBLE, таким же файловым типом и подходящим смещением (своим для каждого процесса).

MPI6File21. В главном процессе дано имя существующего файла целых чисел, содержащего $3K$ элементов, где K — количество процессов. В каждом процессе прочесть и вывести три элемента A, B, C, расположенных в исходном файле в следующем порядке (индекс указывает ранг процесса): A_0, A_1, …, A_{K-1}, B_0, B_1, …, B_{K-1}, C_0, C_1, …, C_{K-1}. Для этого использовать единственный вызов коллективной функции MPI_File_read_all, предварительно определив новый файловый вид данных с базовым типом MPI_INT, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из одного целочисленного элемента и завершающего пустого промежутка, размер которого равен протяженности набора из $K - 1$ целого числа.

MPI6File22. В главном процессе дано имя файла. Кроме того, в каждом процессе дано по 3 целых числа: A, B, C. Количество процессов равно K. Создать новый файл целых чисел с указанным именем и записать в него исходные числа в следующем порядке (индекс указывает ранг процесса): A_{K-1}, A_{K-2}, …, A_0, B_{K-1}, B_{K-2}, …, B_0, C_{K-1}, C_{K-2}, …, C_0. Для этого использовать единственный вызов коллективной функции MPI_File_write_all, предварительно определив новый файловый вид данных с базовым типом MPI_INT, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из одного целочисленного элемента и завершающего пустого промежутка, размер которого равен протяженности набора из $K - 1$ целого числа.

MPI6File23. В главном процессе дано имя существующего файла вещественных чисел, содержащего $6K$ элементов, где K — количество процессов. В каждом процессе прочесть и вывести шесть элементов A, B, C, D, E, F, расположенных в исходном файле в следующем порядке (индекс указывает ранг процесса): A_0, B_0, C_0, A_1, B_1, C_1, …, A_{K-1}, B_{K-1}, C_{K-1}, D_0, E_0, F_0, D_1, E_1, F_1, …, D_{K-1}, E_{K-1}, F_{K-1}. Для этого использовать единственный вызов коллективной функции MPI_File_read_all, предварительно определив новый файловый вид данных с базовым типом MPI_DOUBLE, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из трех вещественных элементов и завершающего пустого промежутка подходящего размера.

MPI6File24. В главном процессе дано имя файла. Кроме того, в каждом процессе дано по 4 вещественных числа: A, B, C, D. Количество процессов равно K. Создать новый файл вещественных чисел с указанным именем и записать в него исходные числа в следующем порядке

(индекс указывает ранг процесса): A_{K-1}, B_{K-1}, A_{K-2}, B_{K-2}, …, A_0, B_0, C_{K-1}, D_{K-1}, C_{K-2}, D_{K-2}, …, C_0, D_0. Для этого использовать единственный вызов коллективной функции MPI_File_write_all, предварительно определив новый файловый вид данных с базовым типом MPI_DOUBLE, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из двух вещественных элементов и завершающего пустого промежутка подходящего размера.

MPI6File25. В главном процессе дано имя файла. Кроме того, в процессе ранга R ($R = 0, 1, \ldots, K - 1$, где K — количество процессов) дано $3 \cdot (R + 1)$ целых чисел: в процессе 0 даны 3 числа A, B, C, в процессе 1 — 6 чисел A, A', B, B', C, C', в процессе 2 — 9 чисел A, A', A'', B, B', B'', C, C', C'', и т. д. Создать новый файл целых чисел с указанным именем и записать в него исходные числа в следующем порядке (индекс указывает ранг процесса): A_0, A_1, A'_1, A_2, A'_2, A''_2, …, B_0, B_1, B'_1, B_2, B'_2, B''_2, … . Для этого использовать единственный вызов коллективной функции MPI_File_write_all, предварительно определив новый файловый вид данных с базовым типом MPI_INT, подходящим смещением (своим для каждого процесса) и новым файловым типом (своим для каждого процесса), состоящим из $R + 1$ целого элемента и завершающего пустого промежутка подходящего размера.

MPI6File26. В главном процессе дано имя файла. Кроме того, в каждом процессе дано по 4 вещественных числа: A, B, C, D. Количество процессов равно K. Создать новый файл вещественных чисел с указанным именем и записать в него исходные числа в следующем порядке (индекс указывает ранг процесса): A_0, A_1, …, A_{K-1}, B_{K-1}, …, B_1, B_0, C_0, C_1, …, C_{K-1}, D_{K-1}, …, D_0. Для этого использовать единственный вызов коллективной функции MPI_File_write_all, предварительно определив новый файловый вид данных с базовым типом MPI_DOUBLE, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из двух вещественных элементов (с дополнительным пустым промежутком между этими элементами) и завершающего пустого промежутка подходящего размера.

MPI6File27. В главном процессе дано имя существующего файла вещественных чисел, содержащего элементы прямоугольной матрицы размера $(K/2) \times K$, где K — количество процессов (четное число). В процессе ранга R, $R = 0, \ldots, K - 1$, прочесть и вывести элементы столбца исходной матрицы с номером $R + 1$ (столбцы нумеруются от 1). Для этого использовать единственный вызов функции MPI_File_read_all, предварительно определив новый файловый вид данных с базовым типом MPI_DOUBLE, подходящим смещением (своим для каждого

процесса) и новым файловым типом, состоящим из вещественного элемента и завершающего пустого промежутка подходящего размера.

MPI6File28. В главном процессе дано имя файла. Кроме того, в каждом процессе дано целое число N и $K/2$ вещественных чисел, где K — количество процессов (четное число). Числа N для всех процессов различны и лежат в диапазоне от 1 до K. Создать новый файл вещественных чисел с указанным именем и записать в него матрицу размера $(K/2) \times K$, причем каждый процесс должен записать свой набор вещественных чисел в столбец матрицы с порядковым номером N (столбцы нумеруются от 1). Для этого использовать единственный вызов коллективной функции MPI_File_write_all, предварительно определив новый файловый вид данных с базовым типом MPI_DOUBLE, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из вещественного элемента и завершающего пустого промежутка подходящего размера.

MPI6File29. В главном процессе дано имя существующего файла целых чисел, содержащего элементы блочной прямоугольной матрицы. Размер матрицы в блоках равен $(K/3) \times 3$, где K — количество процессов (число K кратно 3). Каждый блок представляет собой квадратную матрицу порядка N (значение N может лежать в диапазоне от 2 до 5). В каждом процессе требуется прочесть и вывести один блок исходной матрицы, перебирая блоки по строкам. Для этого использовать единственный вызов функции MPI_File_read_all, предварительно определив новый файловый вид данных с базовым типом MPI_INT, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из N целочисленных элементов и завершающего пустого промежутка подходящего размера. Для определения числа N использовать размер исходного файла, получив его с помощью функции MPI_File_get_size.

MPI6File30. В главном процессе дано имя файла и число N, которое может лежать в диапазоне от 2 до 5. Кроме того, в каждом процессе даны два целых числа I и J, определяющие позицию (номер строки и столбца) некоторого квадратного блока блочной прямоугольной матрицы размера $(K/3) \times 3$ блоков, где K — количество процессов (число K кратно 3). Значения I лежат в диапазоне от 1 до $K/3$, значения J лежат в диапазоне от 1 до 3; все процессы содержат различные позиции блоков. Каждый блок представляет собой квадратную матрицу целых чисел порядка N. Создать новый файл целых чисел с указанным именем и записать в него блочную матрицу размера $(K/3) \times 3$, причем каждый процесс должен записать в матрицу блок в позиции (I, J), а все элементы блока, записанного процессом ранга R ($R = 0, 1, \ldots, K - 1$),

должны быть равны числу *R*. Для этого использовать единственный вызов функции MPI_File_write_all, предварительно определив новый файловый вид данных с базовым типом MPI_INT, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из *K* целочисленных элементов и завершающего пустого промежутка подходящего размера. Для передачи значения *N* во все процессы использовать коллективную функцию MPI_Bcast.

16. Односторонние коммуникации (MPI-2)

При определении окна доступа с помощью функции MPI_Win_create рекомендуется всегда указывать смещение disp_unit (третий параметр), равным протяженности элемента данных соответствующего типа (в заданиях это всегда либо тип MPI_INT, либо тип MPI_DOUBLE; протяженность следует определять с помощью функции MPI_Type_get_extent). В этом случае при последующих вызовах функций одностороннего доступа MPI_Get, MPI_Put, MPI_Accumulate будет достаточно указывать смещение target_disp (пятый параметр любой из этих функций) равным начальному индексу используемого фрагмента массива (а не числу байтов от начала массива до требуемого фрагмента).

В качестве параметра info (четвертый параметр функции MPI_Win_create) достаточно указывать константу MPI_INFO_NULL.

Если в некоторых процессах окно доступа определять не требуется, то в качестве параметра size (второго параметра функции MPI_Win_create) в этих процессах следует указывать значение 0.

В качестве параметра assert во всех синхронизирующих функциях (MPI_Win_fence, MPI_Win_start, MPI_Win_post, MPI_Win_lock) достаточно указывать константу 0 (данный параметр является предпоследним параметром во всех перечисленных функциях).

В первой подгруппе (задания MPI7Win1–MPI7Win17) в качестве синхронизирующих функций следует использовать *коллективную* функцию MPI_Win_fence, которая должна вызываться как перед действиями, связанными с односторонней пересылкой данных, так и после этих действий (но перед действиями, связанными с доступом к пересланным данным).

Задания второй подгруппы (MPI7Win18–MPI7Win30) требуют применения *локальных* вариантов синхронизации: функций MPI_Win_start, MPI_Win_complete, MPI_Win_post, MPI_Win_wait или пары функций MPI_Win_lock, MPI_Win_unlock. В заданиях этой подгруппы всегда указывается, какой вариант синхронизации требуется использовать.

16.1. Односторонние коммуникации с простейшей синхронизацией

MPI7Win1. В каждом из подчиненных процессов дано одно целое число. В главном процессе определить окно доступа размера K целых чисел (K — количество подчиненных процессов) и, используя функцию MPI_Put в подчиненных процессах, записать в главный процесс все исходные числа, после чего вывести эти числа в порядке возрастания рангов переславших их процессов.

MPI7Win2. В каждом из подчиненных процессов дано R вещественных чисел, где R — ранг процесса (1, 2, …). В главном процессе определить окно доступа подходящего размера и, используя функцию MPI_Put в подчиненных процессах, записать в главный процесс все исходные числа, после чего вывести эти числа в порядке возрастания рангов переславших их процессов.

MPI7Win3. В главном процессе дан массив A из K целых чисел, где K — количество подчиненных процессов. Определить в главном процессе окно доступа, содержащее массив A, и, используя функцию MPI_Get в подчиненных процессах, получить и вывести в них по одному элементу массива A, перебирая элементы с конца (элемент A_0 получить в последнем процессе, A_1 в предпоследнем, и т. д.).

MPI7Win4. В главном процессе дан массив A из $K + 4$ вещественных чисел, где K — количество подчиненных процессов. Определить в главном процессе окно доступа, содержащее массив A, и, используя функцию MPI_Get в подчиненных процессах, получить и вывести в них по пять элементов массива A, начиная с элемента с индексом $R - 1$, где R — ранг подчиненного процесса ($R = 1, 2, \ldots, K - 1$).

MPI7Win5. В главном процессе дан массив A из K целых чисел, где K — количество подчиненных процессов. В каждом подчиненном процессе дан индекс N (число от 0 до $K - 1$) и целое число B. В главном процессе определить окно доступа, содержащее массив A, и, используя функцию MPI_Accumulate в каждом подчиненном процессе, умножить элемент A_N на число B, после чего вывести в главном процессе измененный массив A.

Примечание. Некоторые подчиненные процессы могут содержать совпадающие значения N; в этом случае элемент A_N должен умножаться на несколько чисел. Данное обстоятельство не требует дополнительных действий по синхронизации в силу особенностей реализации функции MPI_Accumulate.

MPI7Win6. В главном процессе дан вещественный массив A размера $2K - 1$ (K — количество подчиненных процессов), в каждом подчиненном процессе дан вещественный массив B размера R, где R — ранг

процесса (1, 2, …, $K - 1$). В главном процессе определить окно доступа, содержащее массив A, и, используя функцию MPI_Accumulate в каждом подчиненном процессе, прибавить к элементам массива A, начиная с индекса $R - 1$, значения всех элементов массива B из процесса ранга R, после чего вывести в главном процессе измененный массив A (единственный элемент B_0 из процесса 1 прибавляется к элементу A_0, элементы B_0 и B_1 из процесса 2 прибавляются соответственно к элементам A_1 и A_2, элементы B_0, B_1 и B_2 из процесса 3 прибавляются соответственно к элементам A_2, A_3 и A_4, и т. д.).

Примечание. К элементам массива A, начиная с индекса 2, потребуется прибавлять несколько значений из разных подчиненных процессов. Данное обстоятельство не требует дополнительных действий по синхронизации в силу особенностей реализации функции MPI_Accumulate.

MPI7Win7. В главном процессе дан массив A из $2K$ целых чисел, где K — количество подчиненных процессов. Во всех подчиненных процессах определить окно доступа из двух целых чисел и, используя несколько вызовов функции MPI_Put в главном процессе, записать и вывести в каждом подчиненном процессе по два элемента из массива A, перебирая их в исходном порядке (элементы A_0 и A_1 надо вывести в процессе 1, элементы A_2 и A_3 — в процессе 2, и т. д.).

MPI7Win8. В каждом процессе дано целое число R и вещественное число B. Все числа R различны и лежат в диапазоне от 0 до $K - 1$, где K — количество процессов. Во всех процессах определить окно доступа из одного вещественного числа и, используя функцию MPI_Put в каждом процессе, переслать число B из этого процесса в процесс R, после чего вывести полученные числа во всех процессах.

MPI7Win9. В каждом процессе дан массив A из K целых чисел, где K — количество процессов. Во всех процессах определить окно доступа, содержащее массив A, и, используя несколько вызовов функции MPI_Get в каждом процессе R ($R = 0, \ldots, K - 1$), получить и вывести элементы всех массивов A с индексом R, перебирая эти элементы в порядке убывания рангов содержащих их процессов (вначале выводится элемент, полученный из процесса ранга $K - 1$, затем элемент из процесса ранга $K - 2$, и т. д.).

Примечание. Функцию MPI_Get, как и другие функции, обеспечивающие односторонние коммуникации, можно использовать и для доступа к окну, определенному в вызывающем процессе.

MPI7Win10. В каждом процессе дан массив A из 5 вещественных чисел и целые числа N_1 и N_2, каждое из которых лежит в диапазоне от 0 до 4. Во всех процессах определить окно доступа, содержащее массив A, и,

используя по два вызова функции MPI_Get в каждом процессе, получить из предыдущего процесса элемент его массива A с индексом N_1, а из последующего процесса — элемент с индексом N_2 (числа N_1 и N_2 берутся из вызывающего процесса, процессы перебираются циклически). Вывести в каждом процессе полученные числа в указанном порядке.

MPI7Win11. Количество процессов K — четное число. В каждом процессе дан массив A из $K/2$ целых чисел. Во всех процессах нечетного ранга (1, 3, …, $K-1$) определить окно доступа, содержащее массив A, и, используя требуемое число вызовов функции MPI_Accumulate в каждом процессе четного ранга, добавить элемент $A[I]$ из процесса ранга $2J$ к элементу $A[J]$ из процесса ранга $2I+1$ ($I, J = 0, \ldots, K/2-1$) и вывести измененные массивы A во всех процессах нечетного ранга.

Указание. Требуемое преобразование можно описать по-другому: если через B обозначить матрицу порядка $K/2$, строки которой совпадают с массивами A из процессов четного ранга, а через C — матрицу того же порядка, строки которой совпадают с массивами A из процессов нечетного ранга, то преобразование состоит в прибавлении к I-му столбцу матрицы C соответствующих элементов I-й строки матрицы B.

MPI7Win12. Решить задачу MPI7Win11, определив окна доступа в процессах четного ранга и используя вместо функций MPI_Accumulate функции MPI_Get в процессах нечетного ранга.

Указание. Поскольку числа, полученные из процессов четного ранга, необходимо добавлять к элементам массива A после вызова функции синхронизации MPI_Win_fence, удобно использовать вспомогательный массив для хранения полученных чисел.

MPI7Win13. В каждом процессе даны три целых числа N_1, N_2, N_3, каждое из которых лежит в диапазоне от 0 до $K-1$, где K — количество процессов (значения некоторых из этих чисел в каждом процессе могут совпадать). Кроме того, в каждом процессе дан массив A вещественных чисел размера $R+1$, где R — ранг процесса (0, …, $K-1$). Во всех процессах определить окно доступа, содержащее массив A, и, используя по три вызова функции MPI_Accumulate в каждом процессе, добавить ко всем элементам массива A в процессах рангов N_1, N_2 и N_3 вещественное число, равное $R+1$, где R — ранг процесса, вызвавшего функции MPI_Accumulate (например, если число N_1 в процессе ранга 3 равно 2, то ко всем элементам массива A из процесса 2 надо добавить вещественное число 4.0). Если некоторые из чисел N_1, N_2, N_3 в процессе R совпадают, то числа $R+1$ надо добавлять к элементам со-

ответствующих массивов несколько раз. Вывести измененные массивы *A* в каждом процессе.

MPI7Win14. В каждом процессе дан массив вещественных чисел размера *K* (*K* — количество процессов), содержащий строку верхнетреугольной матрицы *A*, включая ее начальную нулевую часть (процесс ранга *R* содержит *R*-ю строку матрицы в предположении, что строки нумеруются от 0). Во всех процессах определить окно доступа, содержащее исходный массив, и, используя требуемое количество вызовов функции MPI_Get в каждом процессе, записать в эти массивы (и затем вывести) строки матрицы, транспонированной к исходной матрице *A*, включая ее завершающую нулевую часть. Вспомогательные массивы не использовать.

Указания. (1) Строки транспонированной матрицы совпадают со столбцами исходной матрицы, таким образом, полученная матрица будет нижнетреугольной. (2) Обнулять требуемые элементы массивов перед их выводом необходимо после второго вызова функции MPI_Win_fence. (3) Окно доступа для последнего процесса можно не создавать.

MPI7Win15. Решить задачу MPI7Win14, используя вместо вызовов функций MPI_Get вызовы функций MPI_Put.

Указание. В данном случае можно не создавать окно доступа для главного процесса.

MPI7Win16. В каждом процессе дана одна строка вещественной квадратной матрицы *A* порядка *K*, где *K* — количество процессов (процесс ранга *R* содержит *R*-ю строку матрицы в предположении, что строки нумеруются от 0). Кроме того, в каждом процессе дано вещественное число *B*. Во всех процессах определить окно доступа, содержащее строку матрицы *A*, и, используя требуемое число вызовов функции MPI_Accumulate в каждом процессе ранга *R* ($R = 0, \ldots, K - 1$), заменить в строке матрицы из следующего процесса все элементы, меньшие числа *B* из процесса *R*, на это число (процессы перебираются циклически). Затем, используя *K* вызовов функции MPI_Get в каждом процессе, получить и вывести столбец преобразованной матрицы *A* с номером, совпадающим с рангом процесса (столбцы также нумеруются от 0).

Указание. При выполнении этого задания в каждом процессе необходимо *трижды* вызывать функцию синхронизации MPI_Win_fence.

MPI7Win17. В каждом процессе дана одна строка вещественной квадратной матрицы *A* порядка *K*, где *K* — количество процессов (процесс ранга *R* содержит *R*-ю строку матрицы в предположении, что строки нумеруются от 0). Кроме того, в каждом процессе дано вещественное

число *B*. Во всех процессах определить окно доступа, содержащее строку матрицы *A*, и, используя требуемое число вызовов функции MPI_Accumulate в каждом процессе ранга *R* (*R* = 0, …, *K* − 1), заменить в строке матрицы из предыдущего процесса все элементы, большие числа *B* из процесса *R*, на это число (процессы перебираются циклически). Затем, используя *K* вызовов функции MPI_Accumulate в каждом подчиненном процессе, добавить начальный элемент строки из каждого подчиненного процесса ранга *R* (1, …, *K* − 1) ко всем элементам столбца преобразованной матрицы *A* с номером *R* (столбцы также нумеруются от 0). После всех описанных преобразований вывести в каждом процессе новое содержимое соответствующей строки матрицы *A*.

Указание. При выполнении этого задания в каждом процессе необходимо *трижды* вызывать функцию синхронизации MPI_Win_fence.

16.2. Дополнительные виды синхронизации

MPI7Win18. Количество процессов *K* — четное число. В каждом процессе четного ранга (0, 2, …, *K* − 2) дано целое число *A*. Во всех процессах нечетного ранга (1, 3, …, *K* − 1) определить окно доступа из одного целого числа и, используя функцию MPI_Put в каждом процессе четного ранга 2*N*, переслать число *A* в процесс ранга 2*N* + 1 и вывести его в этом процессе. Для синхронизации использовать функции MPI_Win_start и MPI_Win_complete в процессах четного ранга и функции MPI_Win_post и MPI_Win_wait в процессах нечетного ранга; для создания группы процессов, указываемой в качестве первого параметра функций MPI_Win_start и MPI_Win_post, использовать функцию MPI_Group_incl, применив ее к группе, полученной из коммуникатора MPI_COMM_WORLD (с помощью функции MPI_Comm_group).

Примечание. В отличие от *коллективной* функции MPI_Win_fence, использовавшейся в предыдущих заданиях данной группы, синхронизирующие функции, используемые в этом и последующих заданиях, являются *локальными* и, кроме того, позволяют явным образом определить группы инициирующих (origin) и целевых (target) процессов при односторонних коммуникациях.

MPI7Win19. В главном процессе дан массив *A* из *K* вещественных чисел, где *K* — количество подчиненных процессов. В главном процессе определить окно доступа, содержащее массив *A*, и, используя функцию MPI_Get в каждом подчиненном процессе, получить и вывести один из элементов массива *A*, перебирая элементы в обратном порядке (в процессе ранга 1 надо получить элемент массива с индексом *K* − 1, в процессе ранга 2 — элемент с индексом *K* − 2, и т. д.). Для синхрони-

зации использовать функции MPI_Win_start и MPI_Win_complete в подчиненных процессах и функции MPI_Win_post и MPI_Win_wait в главном процессе; для создания группы процессов, указываемой в качестве первого параметра функции MPI_Win_start, использовать функцию MPI_Group_incl, для создания группы, указываемой в функции MPI_Win_post, использовать функцию MPI_Group_excl (применив обе эти функции к группе, полученной из коммуникатора MPI_COMM_WORLD).

MPI7Win20. Число процессов K кратно 3. В процессах ранга $3N$ ($N = 0, \ldots, K/3 - 1$) дан массив A из трех вещественных чисел. Во всех процессах, в которых дан массив, определить окно доступа, содержащее этот массив, и, используя по одному вызову функции MPI_Get в процессах ранга $3N + 1$ и $3N + 2$ ($N = 0, \ldots, K/3 - 1$), прочесть и вывести соответственно один элемент A_0 и два элемента A_1 и A_2 из процесса ранга $3N$ (процесс 1 должен вывести элемент A_0, полученный из процесса 0, процесс 2 — элементы A_1 и A_2, полученные из процесса 0, процесс 4 — элемент A_0, полученный из процесса 3, и т. д.). Для синхронизации использовать функции MPI_Win_post и MPI_Win_wait в процессах ранга $3N$ и функции MPI_Win_start и MPI_Win_complete в остальных процессах.

MPI7Win21. Количество процессов K — четное число. В главном процессе дан массив A из $K/2$ вещественных чисел и массив N целых чисел того же размера. Все элементы массива N различны и лежат в диапазоне от 1 до $K - 1$. В каждом подчиненном процессе определить окно доступа из одного вещественного числа и, используя требуемое количество вызовов функции MPI_Put в главном процессе, переслать в каждый из подчиненных процессов ранга N_I ($I = 0, \ldots, K/2 - 1$) число A_I и вывести полученное число (в остальных подчиненных процессах вывести вещественное число 0.0). Для синхронизации использовать функции MPI_Win_post и MPI_Win_wait в подчиненных процессах и функции MPI_Win_start и MPI_Win_complete в главном процессе.

MPI7Win22. В главном процессе дан массив A вещественных чисел размера K (K — количество подчиненных процессов) и массив N целых чисел размера 8. Все элементы массива N лежат в диапазоне от 1 до K; некоторые элементы данного массива могут совпадать. В каждом подчиненном процессе дан вещественный массив B размера R, где R — ранг процесса ($R = 1, \ldots, K$). Определить в каждом подчиненном процессе окно доступа, содержащее массив B, и, используя требуемое количество вызовов функции MPI_Accumulate в главном процессе, добавить ко всем элементам массивов B из процессов ранга N_I ($I = 0, \ldots, 7$) элементы массива A с теми же индексами (к элементу B_0 добав-

ляется элемент A_0, к элементу B_1 — элемент A_1, и т. д.); для некоторых массивов B элементы из массива A могут добавляться несколько раз. В каждом подчиненном процессе вывести массив B (который либо изменится, либо сохранит исходные значения элементов). Для синхронизации использовать функции MPI_Win_post и MPI_Win_wait в подчиненных процессах и функции MPI_Win_start и MPI_Win_complete в главном процессе.

MPI7Win23. Во всех процессах даны вещественные массивы A размера 5. Кроме того, в главном процессе даны целочисленные массивы N и M размера 5 каждый. Все элементы массива N лежат в диапазоне от 1 до K, где K — количество подчиненных процессов, все элементы массива M лежат в диапазоне от 0 до 4; некоторые элементы как в массиве N, так и в массиве M могут совпадать. В каждом подчиненном процессе определить окно доступа, содержащее массив A, и, используя требуемое количество вызовов функции MPI_Get в главном процессе, получить из процесса ранга N_I ($I = 0, \ldots, 4$) элемент массива A с индексом M_I и добавить его значение к элементу массива A главного процесса с индексом I. После изменения массива A в главном процессе выполнить следующую корректировку массивов A всех подчиненных процессов: заменить в них те элементы, которые больше элемента массива A с тем же индексом из главного процесса, на этот элемент, используя требуемое количество вызовов функции MPI_Accumulate в главном процессе. В каждом процессе вывести преобразованные массивы A. Для синхронизации использовать два вызова пары функции MPI_Win_post и MPI_Win_wait в подчиненных процессах и два вызова пары функции MPI_Win_start и MPI_Win_complete в главном процессе.

MPI7Win24. В каждом подчиненном процессе дано целое число N; все числа N различны и лежат в диапазоне от 0 до $K - 1$, где K — количество подчиненных процессов. В главном процессе определить окно доступа, содержащее целочисленный массив A размера K. Не выполняя никаких вызовов синхронизирующих функций в главном процессе (кроме вызова функции MPI_Barrier) и используя в подчиненных процессах последовательность вызовов синхронизирующих функции MPI_Win_lock, MPI_Win_unlock, MPI_Barrier, MPI_Win_lock, MPI_Win_unlock, записать в элементы массива A с индексом N ранг подчиненного процесса, содержащего данное значение N (с помощью функции MPI_Put), после чего получить и вывести в каждом подчиненном процессе все элементы преобразованного массива A (с помощью функции MPI_Get). В качестве первого параметра функции MPI_Win_lock указывать MPI_LOCK_SHARED.

Примечание. Синхронизирующие функции MPI_Win_lock, MPI_Win_unlock используются преимущественно при односторонних коммуникациях с *пассивными целевыми процессами* (passive targets), при которых целевой процесс не обрабатывает переданные ему данные, а выступает в роли их хранилища, доступного для других процессов.

MPI7Win25. Количество процессов K кратно 3. В процессах ранга $3N$ ($N = 0, \ldots, K/3 - 1$) дан вещественный массив A размера 5, в процессах ранга $3N + 1$ дано целое число M, лежащее в диапазоне от 0 до 4, и вещественное число B. В каждом процессе, содержащем исходный массив A, определить окно доступа с этим массивом. Используя функцию MPI_Accumulate в каждом процессе ранга $3N + 1$ ($N = 0, \ldots, K/3 - 1$), преобразовать массив A из процесса ранга $3N$ следующим образом: если элемент массива с индексом M больше числа B, то он заменяется на число B (числа M и B берутся из процесса ранга $3N + 1$). После этого, используя функцию MPI_Get в каждом процессе ранга $3N + 2$, получить и вывести в нем все элементы преобразованного массива A из процесса ранга $3N$. Использовать синхронизирующие функции MPI_Win_Lock, MPI_Win_unlock, MPI_Barrier в процессах ранга $3N + 1$, функции MPI_Barrier, MPI_Win_Lock, MPI_Win_unlock в процессах ранга $3N + 2$ и функцию MPI_Barrier в процессах ранга $3N$. В качестве первого параметра функции MPI_Win_lock указывать MPI_LOCK_EXCLUSIVE.

MPI7Win26. В каждом подчиненном процессе дан вещественный массив A размера 5 с положительными элементами. В главном процессе определить окно доступа, содержащее вещественный массив B размера 5 с нулевыми элементами. Не выполняя никаких вызовов синхронизирующих функций в главном процессе (кроме вызова функции MPI_Barrier) и используя в подчиненных процессах последовательность вызовов синхронизирующих функции MPI_Win_lock, MPI_Win_unlock, MPI_Barrier, MPI_Win_lock, MPI_Win_unlock, записать в каждый из элементов массива B максимальный из элементов массивов A с тем же индексом (с помощью функции MPI_Accumulate), после чего получить и вывести в каждом подчиненном процессе все элементы преобразованного массива B (с помощью функции MPI_Get). В качестве первого параметра функции MPI_Win_lock указывать MPI_LOCK_SHARED.

MPI7Win27. В каждом подчиненном процессе даны два вещественных числа X, Y — координаты точки на плоскости. Используя вызовы функции MPI_Get в главном процессе, получить в нем числа X_0, Y_0, равные координатам той точки (X, Y) среди точек, данных в подчи-

ненных процессах, которая является наиболее удаленной от начала координат. После этого, используя вызовы функции MPI_Get в подчиненных процессах, получить и вывести в каждом из них числа X_0, Y_0, найденные в главном процессе. В каждом процессе определить окно доступа, содержащее два вещественных числа (X, Y для подчиненных процессов, X_0, Y_0 для главного процесса). Использовать синхронизирующие функции MPI_Win_lock, MPI_Win_unlock, MPI_Barrier в главном процессе и функции MPI_Barrier, MPI_Win_lock, MPI_Win_unlock в подчиненных процессах.

Примечание. Данную задачу невозможно решить, используя односторонние коммуникации только на стороне подчиненных процессов и при этом выполняя синхронизацию с помощью блокировок lock/unlock.

MPI7Win28. Решить задачу MPI7Win27, используя единственное окно доступа в главном процессе, содержащее числа X_0, Y_0. Для нахождения чисел X_0, Y_0 использовать функцию MPI_Get и функцию MPI_Put в подчиненных процессах (для некоторых процессов вызывать функцию MPI_Put не потребуется), для пересылки найденных чисел X_0, Y_0 во все подчиненные процессы использовать в них функцию MPI_Get (как и в задаче MPI7Win27). Для синхронизации обменов на этапе нахождения чисел X_0, Y_0 использовать два вызова пары функций MPI_Win_start и MPI_Win_complete в подчиненных процессах и вызовы *в цикле* двух пар функций MPI_Win_post и MPI_Win_wait в главном процессе (при этом на каждой итерации цикла необходимо определять *новую группу процессов*, используемую при вызове функций MPI_Win_post). Для синхронизации действий при пересылке чисел X_0, Y_0 в подчиненные процессы использовать, как и в задаче MPI7Win27, функцию MPI_Barrier в главном процессе и функции MPI_Barrier, MPI_Win_Lock, MPI_Win_unlock в подчиненных процессах.

Примечание. Описанный вариант решения (в отличие от варианта, приведенного в задаче MPI7Win27) позволяет использовать односторонние коммуникации только на стороне подчиненных процессов, однако при этом на первом этапе решения требуется применять вариант синхронизации, отличный от блокировок lock/unlock.

MPI7Win29. В каждом процессе дана одна строка целочисленной квадратной матрицы порядка K, где K — количество процессов (в процессе ранга R дана строка матрицы с номером R; строки нумеруются от 0). Используя вызовы функции MPI_Get в главном процессе, получить в нем строку матрицы с минимальной суммой элементов S и, кроме того, найти количество N строк с минимальной суммой (если $N > 1$, то в главном процессе надо сохранить последнюю из таких строк, т. е. строку с наибольшим номером). После этого, используя вызовы

функции MPI_Get в подчиненных процессах, получить и вывести в каждом из них строку с минимальной суммой S, найденную в главном процессе, значение суммы S и количество N строк с минимальной суммой. В каждом процессе определить окно доступа, содержащее $K + 2$ целых числа; в первых K элементах окна содержатся элементы строки матрицы, в следующем элементе — сумма их значений S, а последний элемент предназначен для хранения числа N. Использовать синхронизирующие функции MPI_Win_lock, MPI_Win_unlock, MPI_Barrier в главном процессе и функции MPI_Barrier, MPI_Win_lock, MPI_Win_unlock в подчиненных процессах.

Примечание. Данную задачу невозможно решить, используя односторонние коммуникации только на стороне подчиненных процессов и выполняя синхронизацию с помощью блокировок lock/unlock.

MPI7Win30. Решить задачу MPI7Win29, используя единственное окно доступа в главном процессе, содержащее строку матрицы и числа S и N. Для нахождения строки с минимальной суммой элементов и связанных с ней характеристик использовать функцию MPI_Get и функцию MPI_Put в подчиненных процессах (для некоторых процессов вызывать функцию MPI_Put не потребуется), для пересылки найденных данных во все подчиненные процессы использовать в них функцию MPI_Get (как и в задаче MPI7Win29). На этапе нахождения строки с минимальной суммой функцию MPI_Get следует использовать только для получения значений S и N. Для синхронизации обменов на этапе нахождения строки с минимальной суммой использовать два вызова пары функций MPI_Win_start и MPI_Win_complete в подчиненных процессах и вызовы *в цикле* двух пар функций MPI_Win_post и MPI_Win_wait в главном процессе (при этом на каждой итерации цикла необходимо определять *новую группу процессов*, используемую при вызове функций MPI_Win_post). Для синхронизации действий при пересылке найденных данных в подчиненные процессы использовать, как и в задаче MPI7Win29, функцию MPI_Barrier в главном процессе и функции MPI_Barrier, MPI_Win_Lock, MPI_Win_unlock в подчиненных процессах.

Примечание. Описанный вариант решения (в отличие от варианта, приведенного в задаче MPI7Win29) позволяет использовать односторонние коммуникации только на стороне подчиненных процессов, однако при этом на первом этапе решения требуется применять вариант синхронизации, отличный от блокировок lock/unlock.

17. Интеркоммуникаторы и динамическое создание процессов (MPI-2)

Базовые возможности, связанные с созданием интеркоммуникаторов и их использованием для обмена сообщениями между отдельными процессами, определены в стандарте MPI-1. Поэтому 5 заданий данной группы можно выполнять с применением системы MPICH 1.2.5 (это задания MPI8Inter1–MPI8Inter4 и MPI8Inter9). Прочие задания посвящены новым возможностям по созданию интеркоммуникаторов (MPI8Inter5–MPI8Inter8), средствам коллективного обмена для интеркоммуникаторов (MPI8Inter10–MPI8Inter14) и использованию интеркоммуникаторов для динамического создания процессов (MPI8Inter15–MPI8Inter22). Все эти возможности появились в стандарте MPI-2, поэтому для выполнения заданий необходимо подключать к программе систему MPICH2 1.3.

В качестве коммуникатора-*посредника* (третьего параметра peer функции MPI_Intercomm_create) следует указывать *копию* коммуникатора MPI_COMM_WORLD, созданную с помощью функции MPI_Comm_dup.

При динамическом создании процессов с помощью функции MPI_Comm_spawn в заданиях MPI8Inter15–MPI8Inter22 в качестве первого параметра command следует указывать имя исполняемого файла ptprj.exe, в качестве второго параметра argv достаточно указать константу NULL, в качестве четвертого параметра info — константу MPI_NULL_INFO, в качестве последнего параметра array_of_errcodes — константу MPI_ERRCODES_IGNORE. Если в задании не указано, какой коммуникатор должен использоваться в качестве исходного при создании новых процессов, то предполагается, что этим коммуникатором является MPI_COMM_WORLD.

Вместо строки «ptprj.exe» можно использовать реализованную в задачнике функцию char* GetExename(), которая возвращает полное имя исполняемого файла.

17.1. Создание интеркоммуникаторов

MPI8Inter1. Количество процессов K — четное число. В каждом процессе дано целое число X. Используя функции MPI_Comm_group, MPI_Group_range_incl и MPI_Comm_create, создать коммуникаторы, первый из которых содержит процессы четного ранга в том же порядке (0, 2, …, $K/2 - 2$), а второй — процессы нечетного ранга в том же порядке (1, 3, …, $K/2 - 1$). Вывести ранги процессов R в созданных коммуникаторах. Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. Используя функции MPI_Send и MPI_Recv для созданного интеркоммуникатора, получить для каждого процесса число X из процесса того же

ранга, входящего в другую группу этого же интеркоммуникатора, и вывести полученные числа.

MPI8Inter2. Количество процессов K — четное число. В каждом процессе дано целое число C и вещественное число X. Числа C равны либо 0, либо 1, количество значений 1 равно количеству значений 0, причем в процессе ранга 0 дано C, равное 0, а в процессе ранга $K - 1$ дано C, равное 1. Используя один вызов функции MPI_Comm_split, создать коммуникаторы, первый из которых содержит процессы со значениями $C = 0$ в том же порядке, а второй — процессы со значениями $C = 1$ в обратном порядке. Вывести ранги процессов R в созданных коммуникаторах (при этом значение $R = 0$ получат первый и последний процессы исходного коммуникатора MPI_COMM_WORLD). Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. Используя функции MPI_Send и MPI_Recv для созданного интеркоммуникатора, получить для каждого процесса число X из процесса того же ранга, входящего в другую группу этого интеркоммуникатора, и вывести полученные числа.

MPI8Inter3. Количество процессов K кратно 3. В процессах ранга $3N$ ($N = 0, \ldots, 3K - 3$) дано вещественное число X, в процессах ранга $3N + 1$ даны вещественные числа X и Y, в процессах ранга $3N + 2$ дано вещественное число Y. Используя функции MPI_Comm_group, MPI_Group_range_incl и MPI_Comm_create, создать коммуникаторы, первый из которых содержит группу процессов ранга $3N$ в том же порядке ($0, 3, \ldots, K - 3$), второй — группу процессов ранга $3N + 1$ в обратном порядке ($K - 2, K - 5, \ldots, 1$), а третий — группу процессов ранга $3N + 2$ в том же порядке ($2, 5, \ldots, K - 1$). Вывести ранги процессов R в созданных коммуникаторах. Затем, используя функцию MPI_Intercomm_create, объединить созданные коммуникаторы в два интеркоммуникатора: первый должен содержать первую и вторую группы процессов, а второй — вторую и третью группы. Используя функции MPI_Send и MPI_Recv для созданных интеркоммуникаторов, поменять числа X для процессов одинакового ранга, входящих в первую и вторую группу, и числа Y для процессов одинакового ранга, входящих во вторую и третью группу. Вывести в каждом процессе полученные числа.

Указание. Следует обратить внимание на то, что функция MPI_Intercomm_create должна вызываться один раз для процессов, входящих в первую и третью группы, и два раза для процессов, входящих во вторую группу, и такое же количество вызовов должно быть для функций MPI_Send и MPI_Recv.

MPI8Inter4. Количество процессов K кратно 3. В каждом процессе даны три целых числа. Первое число (обозначаемое буквой C) лежит в диапазоне от 0 до 2, причем каждое из значений 0, 1, 2 встречается одинаковое число раз (равное $K/3$) и при этом в процессах ранга 0, 1, 2 значения C совпадают с рангами этих процессов. Используя один вызов функции MPI_Comm_split, создать коммуникаторы, первый из которых содержит процессы со значениями $C = 0$ в том же порядке, второй — процессы со значениями $C = 1$ в том же порядке, а третий — процессы со значениями $C = 2$ в том же порядке. Вывести ранги процессов R в созданных коммуникаторах (при этом значение $R = 0$ получат первые три процесса исходного коммуникатора MPI_COMM_WORLD). Затем, используя по два вызова функции MPI_Intercomm_create в каждом процессе, объединить созданные коммуникаторы в три интеркоммуникатора: первый должен содержать группы процессов со значениями C, равными 0 и 1, второй — группы со значениями 1 и 2, третий — группы со значениями 0 и 2 (таким образом, созданные интеркоммуникаторы будут образовывать кольцо, связывающее все три ранее созданные группы). Считая, что в первой группе процессов два следующих исходных числа обозначаются буквами X и Y, во второй группе — Y и Z, а в третьей — Z и X (в указанном порядке) и используя в каждом процессе по два вызова функций MPI_Send и MPI_Recv для созданных интеркоммуникаторов, поменять числа X для процессов одинакового ранга, входящих в первую и вторую группу, числа Y — для процессов одинакового ранга, входящих во вторую и третью группу, и числа Z — для процессов одинакового ранга, входящих в первую и третью группу. Вывести в каждом процессе полученные числа.

MPI8Inter5. Количество процессов K кратно 4. В каждом процессе дано целое число X. Используя функции MPI_Comm_group, MPI_Group_range_incl и MPI_Comm_create, создать коммуникаторы, первый из которых содержит первую половину процессов (с рангами 0, 1, …, $K/2 - 1$ в указанном порядке), а второй — вторую половину (с рангами $K/2$, $K/2 + 1$, …, $K - 1$ в указанном порядке). Вывести ранги процессов R_1 в созданных коммуникаторах. Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. После этого, используя функцию MPI_Comm_create для созданного интеркоммуникатора, получить новый интеркоммуникатор, первая группа которого содержит процессы из первой группы исходного интеркоммуникатора с четными рангами (в том же порядке), а вторая группа — процессы из второй группы исходного интеркоммуникатора с нечетными рангами в обратном порядке (таким образом, в первую группу нового интеркоммуникатора

будут входить процессы исходного коммуникатора MPI_COMM_WORLD с рангами 0, 2, …, $K/2 - 2$, а во вторую — с рангами $K - 1$, $K - 3$, …, $K/2 + 1$). Вывести ранги процессов R_2, входящих в новый интеркоммуникатор. Используя функции MPI_Send и MPI_Recv для нового интеркоммуникатора, получить для каждого процесса число X из процесса того же ранга, входящего в другую группу этого интеркоммуникатора, и вывести полученные числа.

MPI8Inter6. Количество процессов K кратно 4. В каждом процессе дано вещественное число X. Используя функции MPI_Comm_group, MPI_Group_range_incl и MPI_Comm_create, создать коммуникаторы, первый из которых содержит первую половину процессов (с рангами 0, 1, …, $K/2 - 1$ в указанном порядке), а второй — вторую половину (с рангами $K/2$, $K/2 + 1$, …, $K - 1$ в указанном порядке). Вывести ранги процессов R_1 в созданных коммуникаторах. Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. После этого, используя один вызов функции MPI_Comm_split для созданного интеркоммуникатора, получить два новых интеркоммуникатора. Первый из новых интеркоммуникаторов содержит процессы исходного интеркоммуникатора с четными рангами, а второй — с нечетными, причем процессы во второй группе каждого из новых интеркоммуникаторов должны располагаться в обратном порядке (таким образом, если использовать ранги процессов для исходного коммуникатора MPI_COMM_WORLD, то первый интеркоммуникатор содержит группы процессов ранга 0, 2, …, $K/2 - 2$ и $K - 2$, $K - 4$, …, $K/2$, а второй — группы процессов ранга 1, 3, …, $K/2 - 1$ и $K - 1$, $K - 3$, …, $K/2 + 1$). Вывести ранги процессов R_2, входящих в новые интеркоммуникаторы. Используя функции MPI_Send и MPI_Recv для новых интеркоммуникаторов, получить для каждого процесса число X из процесса того же ранга, входящего в другую группу этого интеркоммуникатора, и вывести полученные числа.

MPI8Inter7. Количество процессов K — четное число. В каждом процессе дано целое число C, равное либо 0, либо 1, причем известно, что в первой половине процессов дано единственное значение $C = 1$, а во второй половине количество значений $C = 1$ больше одного и, кроме того, имеется хотя бы одно значение $C = 0$. Используя функцию MPI_Comm_split, создать коммуникаторы, первый из которых содержит первую половину процессов (с рангами 0, 1, …, $K/2 - 1$ в указанном порядке), а второй — вторую половину (с рангами $K/2$, $K/2 + 1$, …, $K - 1$ в указанном порядке). Вывести ранги процессов R_1 в созданных коммуникаторах. Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. После этого, используя функцию MPI_Comm_split для созданного ин-

теркоммуникатора, получить новый интеркоммуникатор, группы которого содержат процессы из соответствующих групп исходного интеркоммуникатора со значениями $C = 1$, взятые в обратном порядке (таким образом, в первую группу нового интеркоммуникатора будет входить единственный процесс, а количество процессов во второй группе будет лежать в диапазоне от 2 до $K/2 - 1$). Вывести ранги процессов R_2, входящих в ту группу нового интеркоммуникатора, которая содержит более одного процесса. В единственном процессе первой группы нового интеркоммуникатора ввести массив Y из K_2 целых чисел, где K_2 — количество процессов во второй группе; в каждом процессе второй группы этого же интеркоммуникатора ввести по одному целому числу X. Используя требуемое количество вызовов функций MPI_Send и MPI_Recv для всех процессов нового интеркоммуникатора, получить в процессе первой группы все числа X из процессов второй группы (в порядке возрастания рангов этих процессов), а в каждом процессе второй группы ранга R_2 (0, 1, …, $K_2 - 1$) получить элемент с индексом R_2 из массива Y, данного в процессе первой группы. Вывести полученные числа.

Примечание. В реализации MPICH 2 версии 1.3, используемой в задачнике, функция MPI_Comm_split приводит к ошибочной программе, если она применяется к интеркоммуникатору и при этом какие-либо из значений ее параметра color равны MPI_UNDEFINED. Таким образом, для корректной работы программ необходимо использовать только *неотрицательные* значения color. Кроме того, программа в дальнейшем может вести себя некорректно, если в результате применения функции MPI_Comm_split к интеркоммуникатору создаются *пустые группы* (это возможно, если для всех процессов одной из групп исходного интеркоммуникатора указываются одинаковые значения color, отличные от некоторых значений color для процессов другой группы).

MPI8Inter8. В каждом процессе дано целое число C, лежащее в диапазоне от 0 до 2, причем известно, что и у процессов четного ранга, и у процессов нечетного ранга имеется хотя бы по одному из значений 0, 1, 2. Используя один вызов функции MPI_Comm_split, создать коммуникаторы, первый из которых содержит процессы с четными рангами (в порядке возрастания рангов), а второй — процессы с нечетными рангами (также в порядке возрастания рангов). Вывести ранги процессов R_1 в созданных коммуникаторах. Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. После этого, используя один вызов функции MPI_Comm_split для созданного интеркоммуникатора, получить три новых интеркоммуникатора, группы которых содержат процессы из соответствующих групп исходного интеркоммуникатора с одинако-

выми значениями C, взятые в том же порядке (таким образом, в первую группу первого интеркоммуникатора будут входить процессы четного ранга со значениями $C = 0$, а, например, во вторую группу третьего интеркоммуникатора будут входить процессы нечетного ранга со значениями $C = 2$). Вывести ранги процессов R_2 в новых интеркоммуникаторах. В процессах из первых групп созданных интеркоммуникаторов ввести по одному целому числу X, а в процессах из вторых групп — по одному целому числу Y. Используя требуемое количество вызовов функций MPI_Send и MPI_Recv для всех процессов новых интеркоммуникаторов, переслать все числа X каждому из процессов второй группы этого же коммуникатора, а все числа Y — каждому из процессов первой группы и вывести полученные числа в порядке возрастания рангов переславших их процессов.

MPI8Inter9. Количество процессов K — четное число. В каждом процессе дано целое число C, лежащее в диапазоне от 0 до 2, причем известно, что в процессе ранга 0 дано число $C = 1$, а первое из значений $C = 2$ имеется у процесса ранга $K/2$. Используя функцию MPI_Comm_split, создать коммуникаторы, первый из которых содержит процессы со значениями $C = 1$ в том же порядке, а второй — процессы со значениями $C = 2$ в том же порядке. Вывести ранги процессов R в созданных коммуникаторах (если процесс не входит ни в один из созданных коммуникаторов, то вывести для него число −1). Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. После этого ввести в процессах первой группы созданного интеркоммуникатора (соответствующей значениям $C = 1$) по одному целому числу X, а в процессах второй группы — по одному целому числу Y. Используя требуемое количество вызовов функций MPI_Send и MPI_Recv для всех процессов созданного интеркоммуникатора, переслать все числа X каждому из процессов второй группы этого коммуникатора, а все числа Y — каждому из процессов первой группы и вывести полученные числа в порядке возрастания рангов переславших их процессов.

17.2. Коллективные операции для интеркоммуникаторов

MPI8Inter10. Количество процессов K — четное число. В каждом процессе дано целое число C, лежащее в диапазоне от 0 до 2, причем известно, что в процессе ранга 0 дано число $C = 1$, а первое из значений $C = 2$ имеется у процесса ранга $K/2$. Используя функцию MPI_Comm_split, создать коммуникаторы, первый из которых содержит процессы со значениями $C = 1$ в том же порядке, а второй — процессы со значениями $C = 2$ в том же порядке. Вывести ранги процессов R в созданных коммуникаторах (если процесс не входит ни в один из созданных

коммуникаторов, то вывести для него число −1). Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create (группа, содержащая процессы со значениями $C = 1$, считается первой группой созданного интеркоммуникатора, а группа процессов со значениями $C = 2$ — второй). После этого ввести в процессах обеих групп созданного интеркоммуникатора целые числа R_1 и R_2. Значения чисел R_1 во всех процессах совпадают и указывают ранг выделенного процесса первой группы; значения чисел R_2 во всех процессах также совпадают и указывают ранг выделенного процесса второй группы. В выделенном процессе первой группы дан набор из трех целых чисел X, в выделенном процессе второй группы — набор из трех целых чисел Y. Используя по два вызова коллективной функции MPI_Bcast в каждом процессе созданного интеркоммуникатора, переслать набор чисел X во все процессы второй группы, а набор чисел Y во все процессы первой группы, после чего вывести полученные числа.

MPI8Inter11. Количество процессов K — четное число. В каждом процессе дано целое число C, лежащее в диапазоне от 0 до 2, причем известно, что в процессе ранга 0 дано число $C = 1$, а первое из значений $C = 2$ имеется у процесса ранга $K/2$. Используя функцию MPI_Comm_split, создать коммуникаторы, первый из которых содержит процессы со значениями $C = 1$ в том же порядке, а второй — процессы со значениями $C = 2$ в том же порядке. Вывести ранги процессов R в созданных коммуникаторах (если процесс не входит ни в один из созданных коммуникаторов, то вывести для него число −1). Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create (группа, содержащая процессы со значениями $C = 1$, считается первой группой созданного интеркоммуникатора, а группа процессов со значениями $C = 2$ — второй). После этого ввести в процессах обеих групп созданного интеркоммуникатора целое число R_1, которое совпадает во всех процессах и указывает ранг выделенного процесса первой группы. В выделенном процессе первой группы дан набор из K_2 целых чисел X, где K_2 — количество процессов во второй группе. Используя один вызов коллективной функции MPI_Scatter в каждом процессе созданного интеркоммуникатора, переслать по одному числу из набора X во все процессы второй группы (в порядке возрастания рангов процессов), после чего вывести полученные числа.

MPI8Inter12. Количество процессов K — четное число. В каждом процессе дано целое число C, лежащее в диапазоне от 0 до 2, причем известно, что в процессе ранга 0 дано число $C = 1$, а первое из значений $C = 2$ имеется у процесса ранга $K/2$. Используя функцию MPI_Comm_split,

создать коммуникаторы, первый из которых содержит процессы со значениями $C = 1$ в том же порядке, а второй — процессы со значениями $C = 2$ в том же порядке. Вывести ранги процессов R в созданных коммуникаторах (если процесс не входит ни в один из созданных коммуникаторов, то вывести для него число −1). Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create (группа, содержащая процессы со значениями $C = 1$, считается первой группой созданного интеркоммуникатора, а группа процессов со значениями $C = 2$ — второй). После этого ввести в процессах обеих групп созданного интеркоммуникатора целое число R_2, которое совпадает во всех процессах и указывает ранг выделенного процесса второй группы. В каждом процессе первой группы дано по одному целому числу X. Используя один вызов коллективной функции MPI_Gather в каждом процессе созданного интеркоммуникатора, переслать исходные числа X (в порядке возрастания рангов содержащих их процессов) в выделенный процесс второй группы, после чего вывести в этом процессе полученные числа.

MPI8Inter13. Количество процессов K — четное число. В каждом процессе дано целое число C, лежащее в диапазоне от 0 до 2, причем известно, что в процессе ранга 0 дано число $C = 1$, а первое из значений $C = 2$ имеется у процесса ранга $K/2$. Используя функцию MPI_Comm_split, создать коммуникаторы, первый из которых содержит процессы со значениями $C = 1$ в том же порядке, а второй — процессы со значениями $C = 2$ в том же порядке. Вывести ранги процессов R в созданных коммуникаторах (если процесс не входит ни в один из созданных коммуникаторов, то вывести для него число −1). Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create (группа, содержащая процессы со значениями $C = 1$, считается первой группой созданного интеркоммуникатора, а группа процессов со значениями $C = 2$ — второй). В каждом процессе первой группы дано по одному целому числу X, в каждом процессе второй группы дано по одному целому числу Y. Используя один вызов коллективной функции MPI_Allreduce в каждом процессе созданного интеркоммуникатора, получить в каждом процессе первой группы число Y_{min} — минимальное из исходных чисел Y, а в каждом процессе второй группы число X_{max} — максимальное из исходных чисел X, после чего вывести полученные числа.

MPI8Inter14. Количество процессов K — четное число. В каждом процессе дано целое число C, лежащее в диапазоне от 0 до 2, причем известно, что в процессе ранга 0 дано число $C = 1$, а первое из значений $C = 2$ имеется у процесса ранга $K - 1$. Используя функцию MPI_Comm_split, создать коммуникаторы, первый из которых содержит процессы со

значениями $C = 1$ в том же порядке, а второй — процессы со значениями $C = 2$ в обратном порядке. Вывести ранги процессов R в созданных коммуникаторах (если процесс не входит ни в один из созданных коммуникаторов, то вывести для него число −1). Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create (группа, содержащая процессы со значениями $C = 1$, считается первой группой созданного интеркоммуникатора, а группа процессов со значениями $C = 2$ — второй). В каждом процессе первой группы дан массив целых чисел X размера K_2, где K_2 — количество процессов во второй группе, в каждом процессе второй группы дан массив целых чисел Y размера K_1, где K_1 — количество процессов в первой группе. Используя один вызов коллективной функции MPI_Alltoall в каждом процессе созданного интеркоммуникатора, переслать в процесс ранга R_1 первой группы ($R_1 = 0, \ldots, K_1 - 1$) элементы с индексом R_1 из всех массивов Y, а в процесс ранга R_2 второй группы ($R_2 = 0, \ldots, K_2 - 1$) элементы с индексом R_2 из всех массивов X. Вывести полученные числа в порядке возрастания рангов переславших их процессов.

17.3. Динамическое создание процессов

MPI8Inter15. В каждом процессе дано вещественное число. Используя функцию MPI_Comm_spawn с первым параметром «ptprj.exe», создать один новый процесс. С помощью коллективной функции MPI_Reduce переслать в созданный процесс сумму исходных чисел и отобразить ее в разделе отладки, используя в этом процессе функцию Show. Затем с помощью коллективной функции MPI_Bcast переслать найденную сумму в исходные процессы и вывести эту сумму в каждом процессе.

MPI8Inter16. В каждом процессе дан массив из K вещественных чисел, где K — количество процессов. Используя один вызов функции MPI_Comm_spawn с первым параметром «ptprj.exe», создать K новых процессов. С помощью коллективной функции MPI_Reduce_scatter_block переслать в созданный процесс ранга R ($R = 0, \ldots, K - 1$) максимальный из элементов исходных массивов с индексом R и отобразить полученные максимальные элементы в разделе отладки, используя в каждом новом процессе функцию Show. Затем с помощью функций MPI_Send и MPI_Recv переслать найденный максимальный элемент из нового процесса ранга R ($R = 0, \ldots, K - 1$) в исходный процесс того же ранга и вывести полученные элементы в исходных процессах.

MPI8Inter17. Количество процессов K — четное число. В процессах ранга 0 и 1 даны массивы вещественных чисел размера $K/2$. Используя один

вызов функции MPI_Comm_spawn с первым параметром «ptprj.exe», создать два новых процесса. Используя один вызов функции MPI_Comm_split для интеркоммуникатора, связанного с созданными процессами, создать два новых интеркоммуникатора: первый содержит группу исходных процессов четного ранга (0, …, $K - 2$), а также, в качестве второй группы, первый из новых процессов (ранга 0), второй содержит группу исходных процессов нечетного ранга (1, …. $K - 1$), а также, в качестве второй группы, второй из новых процессов (ранга 1). Используя функции MPI_Send в исходных процессах и функции MPI_Recv в новых процессах, переслать все исходные числа из первого процесса первой группы каждого интеркоммуникатора в единственный процесс, входящий в его вторую группу. Отобразить полученные числа в разделе отладки, используя в новых процессах функцию Show. Затем с помощью коллективной функции MPI_Scatter для интеркоммуникаторов переслать по одному числу из новых процессов во все процессы первой группы соответствующего интеркоммуникатора (в порядке возрастания рангов процессов) и вывести полученные числа.

MPI8Inter18. Количество процессов K — четное число. В каждом процессе дан массив вещественных чисел размера $K/2$. Используя один вызов функции MPI_Comm_spawn с первым параметром «ptprj.exe», создать K новых процессов. Используя один вызов функции MPI_Comm_split для интеркоммуникатора, связанного с созданными процессами, создать два новых интеркоммуникатора: первый содержит группу исходных процессов четного ранга (0, …. $K - 2$), а также, в качестве второй группы, созданные процессы четного ранга, второй содержит группу исходных процессов нечетного ранга (1, …. $K - 1$), а также, в качестве второй группы, созданные процессы нечетного ранга. Для каждого из созданных интеркоммуникаторов выполнить следующие действия: среди элементов исходных массивов с индексом R ($R = 0, \ldots, K/2 - 1$), входящих в первую группу каждого интеркоммуникатора, найти минимальный (для первого интеркоммуникатора) или максимальный (для второго) и переслать этот экстремальный элемент в тот из новых процессов, который имеет ранг R во второй группе соответствующего интеркоммуникатора (например, минимальный из первых элементов массивов, которые даны в исходных процессах четного ранга, следует переслать в первый из созданных процессов, а максимальный из первых элементов массивов, которые даны в исходных процессах нечетного ранга, следует переслать во второй из созданных процессов, поскольку этот процесс имеет ранг 0 в соответствующем интеркоммуникаторе). Для выполнения этих действий использовать коллективную функцию MPI_Reduce_scatter_block. Полученные экстремальные эле-

менты отобразить в разделе отладки, используя в каждом новом процессе функцию Show. Затем с помощью коллективной функции MPI_Reduce найти минимальный элемент из тех минимумов, которые были получены во второй группе первого интеркоммуникатора, переслать и вывести его в первом процессе первой группы этого интеркоммуникатора (т. е. в процессе ранга 0 в исходном коммуникаторе MPI_COMM_WORLD), а также найти максимальный элемент из тех максимумов, которые были получены во второй группе второго интеркоммуникатора, переслать и вывести его в первом процессе первой группы этого интеркоммуникатора (т. е. в процессе ранга 1 в исходном коммуникаторе MPI_COMM_WORLD).

MPI8Inter19. В главном процессе дан массив A из $2K$ целых чисел, где K — количество процессов. Используя один вызов функции MPI_Comm_spawn с первым параметром «ptprj.exe», создать K новых процессов. С помощью коллективной функции MPI_Intercomm_merge для интеркоммуникатора, связанного с созданными процессами, создать новый интракоммуникатор, содержащий как исходные, так и новые процессы (параметр high функции MPI_Intercomm_merge необходимо задать таким образом, чтобы в созданном интракоммуникаторе вначале располагались исходные, а затем созданные процессы). Используя коллективную функцию MPI_Scatter для нового интракоммуникатора, переслать по одному элементу массива A из главного процесса во все процессы приложения (как исходные, так и созданные) в порядке возрастания их рангов в новом интракоммуникаторе. В исходных процессах вывести полученные числа, в новых процессах отобразить полученные числа в разделе отладки, используя в этих процессах функцию Show. Затем, используя коллективную функцию MPI_Reduce в этом же интракоммуникаторе, переслать в исходный процесс ранга 1 сумму всех чисел и вывести ее в этом процессе.

MPI8Inter20. Количество процессов K не делится на 4. В каждом процессе дано целое число A. Используя один вызов функции MPI_Comm_spawn с первым параметром «ptprj.exe», создать такое количество новых процессов (1, 2 или 3), чтобы общее число процессов K_0 в приложении стало кратным 4. В каждом из созданных процессов задать значение числа A, равное $-R-1$, где R — ранг созданного процесса. Используя функцию MPI_Intercomm_merge для интеркоммуникатора, связанного с созданными процессами, создать новый интракоммуникатор, содержащий как исходные, так и новые процессы (параметр high функции MPI_Intercomm_merge необходимо задать таким образом, чтобы в созданном интракоммуникаторе вначале располагались исходные, а затем новые процессы). Используя функцию MPI_Cart_create для созданного интракоммуникатора, определить для

всех процессов декартову топологию в виде двумерной решетки размера $(K_0/4) \times 4$, являющейся периодической по второму измерению (порядок нумерации процессов оставить прежним). Используя функцию MPI_Cart_coords, определить координаты *X*, *Y* каждого процесса в данной решетке; в исходных процессах вывести эти координаты, в новых процессах отобразить координаты в разделе отладки с помощью функций Show, снабдив их комментариями «X = » и «Y = ». Выполнить циклический сдвиг значений *A* в каждом столбце созданной решетки с шагом −1 (т. е. в направлении убывания рангов процессов), используя для этого функции MPI_Comm_shift и MPI_Sendrecv_replace. В исходных процессах вывести полученные значения *A*, в новых процессах отобразить их в разделе отладки с помощью функции Show, снабдив комментарием «A = ».

MPI8Inter21. В каждом процессе дано вещественное число: в главном процессе это число обозначается буквой *A*, в подчиненных процессах — буквой *B*. Используя два вызова функции MPI_Comm_spawn с первым параметром «ptprj.exe», создать две группы новых процессов: первая группа (группа-*сервер*) должна содержать один процесс, вторая группа (группа-*клиент*) должна содержать $K - 1$ процесс, где *K* — количество исходных процессов. Переслать в процесс группы-сервера число *A* из главного процесса, а в процессы группы-клиента — числа *B* из подчиненных процессов (в порядке возрастания рангов процессов) и отобразить полученные числа в разделе отладки, используя в каждом новом процессе функцию Show. С помощью функций MPI_Open_port, MPI_Publish_name и MPI_Comm_accept на стороне сервера и функций MPI_Lookup_name и MPI_Comm_connect на стороне клиента установить связь между двумя новыми группами процессов, создав для них новый интеркоммуникатор. Используя функции MPI_Send и MPI_Recv для этого интеркоммуникатора, получить в каждом процессе группы-клиента число *A* из процесса группы-сервера и найти сумму этого числа и числа *B*, полученного ранее из исходных подчиненных процессов. Отобразить найденную сумму $A + B$ в разделе отладки, используя в каждом процессе группы-клиента функцию Show, а также переслать эту сумму в соответствующий исходный подчиненный процесс и вывести полученную сумму в этом процессе (сумма, найденная в процессе ранга *R* группы-клиента, должна быть переслана в исходный процесс ранга $R + 1$).

Примечание. В программе необходимо обеспечить такую последовательность действий, чтобы вызов функции MPI_Lookup_name в процессах клиента гарантированно выполнялся *после* вызова функции MPI_Publish_name в процессе сервера. Можно, например, использовать функцию MPI_Barrier для исходных процессов и процесса серве-

ра, вызвав ее в процессе сервера только после завершения вызова функции MPI_Publish_name, и вызвать функцию MPI_Comm_spawn для создания группы-клиента уже после выхода из функции MPI_Barrier в исходных процессах.

MPI8Inter22. В каждом процессе дано целое число *N*, которое может принимать три значения: 0, 1 и *K* (*K* > 1). Известно, что имеется ровно один процесс со значением *N* = 1 и ровно *K* процессов со значением *N* = *K*. В тех процессах, в которых число *N* не равно 0, также дано целое число *A*. Используя функцию MPI_Comm_split, разбить исходный коммуникатор MPI_COMM_WORLD на два, в каждый из которых входят процессы с одинаковым ненулевым значением *N*. Используя один вызов функции MPI_Comm_spawn для каждого из созданных коммуникаторов, создать два набора новых процессов; число новых процессов должно совпадать с количеством процессов в соответствующем коммуникаторе (таким образом, в одной новой группе будет содержаться 1 процесс, в другой — *K* процессов). Переслать в каждый из новых процессов число *A* из того исходного процесса, ранг которого в созданном коммуникаторе совпадает с рангом нового процесса. Отобразить полученные числа в разделе отладки, используя в каждом новом процессе функцию Show. Считая новую группу, состоящую из одного процесса, *группой-сервером*, а новую группу из *K* процессов — *группой-клиентом*, установить между ними связь с помощью функций MPI_Open_port, MPI_Publish_name и MPI_Comm_accept на стороне сервера и MPI_Lookup_name и MPI_Comm_connect на стороне клиента. Используя коллективную функцию MPI_Gather для интеркоммуникатора, связывающего группу-клиент и группу-сервер, переслать все числа *A* из процессов группы-клиента в процесс группы-сервера и отобразить полученные числа в разделе отладки, используя несколько вызовов функции Show в процессе группы-сервера. Кроме того, переслать эти числа с помощью функций MPI_Send и MPI_Recv из процесса группы-сервера в тот исходный процесс, который создал группу-сервер, и вывести полученные числа в этом исходном процессе.

Примечание. В программе необходимо обеспечить такую последовательность действий, чтобы вызов функции MPI_Lookup_name в процессах клиента гарантированно выполнялся *после* вызова функции MPI_Publish_name в процессе сервера. Можно, например, поступить следующим образом. Для пересылки числа *A* в процесс группы-сервера применить функцию MPI_Ssend, вызвав после нее функцию MPI_Barrier для исходного коммуникатора MPI_COMM_WORLD и организовав прием этого числа в процессе группы-сервера только после выполнения функции MPI_Publish_name. В остальных процессах исходного коммуникатора MPI_COMM_WORLD следует вначале вы-

звать функцию MPI_Barrier, а затем организовать пересылку чисел *A* в созданные процессы группы-клиента. Таким образом, любой из процессов группы-клиента получит число *A* только тогда, когда процесс группы-сервера уже выполнит вызов функции MPI_Publish_name.

18. Параллельные матричные алгоритмы

Все числа в заданиях являются целыми. Матрицы должны вводиться и выводиться по строкам. Файлы с элементами матриц также содержат их по строкам.

Количество процессов в заданиях, связанных с ленточными алгоритмами (MPI9Matr2–MPI9Matr20), не превосходит 5, а в заданиях, связанных с блочными алгоритмами (MPI9Matr21–MPI9Matr44), не превосходит 16.

Для хранения имени файла достаточно использовать массив char[12], а для его пересылки из главного процесса в подчиненные — функцию MPI_Bcast с параметром типа MPI_CHAR.

В заготовках для каждого задания уже содержатся описания целочисленных переменных для хранения числовых данных, упоминаемых в заданиях (в частности, размеров матриц), указателей на массивы для хранения самих матриц, а также переменных типа MPI_Datatype и MPI_Comm. Эти переменные следует использовать во всех функциях, которые требуется реализовать при выполнении заданий. Все имена переменных соответствуют обозначениям, применяемым в формулировках заданий. Для массивов, связанных с полосами или блоками матриц, используются имена `a`, `b`, `c`, `t`; для массивов, связанных с исходными матрицами *A*, *B* и их результирующим произведением *C*, используются имена с подчеркиванием: `a_`, `b_`, `c_`.

Задания, в которых используется файловый ввод-вывод (MPI9Matr8–MPI9Matr10, MPI9Matr18–MPI9Matr20, MPI9Matr29–MPI9Matr31, MPI9Matr42–MPI9Matr44), требуют подключения библиотеки MPI-2 (система MPICH2 1.3). Для выполнения остальных заданий данной группы можно использовать любую версию MPI (MPICH 1.3.5 или MPICH2 1.3).

18.1. Непараллельный алгоритм умножения матриц

MPI9Matr1. В главном процессе даны числа *M*, *P*, *Q* и матрицы *A* и *B* размера $M \times P$ и $P \times Q$ соответственно. Найти и вывести в главном процессе матрицу *C* размера $M \times Q$, являющуюся произведением матриц *A* и *B*.

Формула для вычисления элементов матрицы *C* в предположении, что строки и столбцы всех матриц нумеруются от 0, имеет вид:

$C_{I,J} = A_{I,0} \cdot B_{0,J} + A_{I,1} \cdot B_{1,J} + \ldots + A_{I,P\text{-}1} \cdot B_{P\text{-}1,J}$, где $I = 0, \ldots, M - 1$, $J = 0, \ldots, Q - 1$.

Для хранения матриц *A*, *B*, *C* использовать одномерные массивы размера $M \cdot P$, $P \cdot Q$ и $M \cdot Q$, размещая в них элементы матриц по строкам (при этом элемент матрицы с индексами *I* и *J* будет храниться в элементе соответствующего массива с индексом $I \cdot N + J$, где *N* — количество столбцов матрицы). При выполнении данного задания подчиненные процессы не используются.

18.2. Ленточный алгоритм 1 (горизонтальные полосы)

MPI9Matr2. В главном процессе даны числа *M*, *P*, *Q* и матрицы *A* и *B* размера $M \times P$ и $P \times Q$ соответственно. В первом варианте ленточного алгоритма перемножения матриц каждая матрица-сомножитель разбивается на *K* горизонтальных полос, где *K* — количество процессов (в дальнейшем полосы распределяются по процессам и используются для вычисления в каждом процессе части итогового матричного произведения).

Полоса для матрицы *A* содержит N_A строк, полоса для матрицы *B* содержит N_B строк; числа N_A и N_B вычисляются по формулам $N_A = \text{ceil}(M/K)$, $N_B = \text{ceil}(P/K)$, где операция «/» означает вещественное деление, а функция ceil выполняет округление с избытком. Если матрица содержит недостаточно строк для заполнения последней полосы, то полоса дополняется нулевыми строками.

Сохранить исходные матрицы, дополненные при необходимости нулевыми строками, в одномерных массивах в главном процессе, после чего организовать пересылку полос из этих массивов во все процессы: в процесс ранга *R* ($R = 0, 1, \ldots, K - 1$) пересылается полоса с индексом *R*, все полосы A_R имеют размер $N_A \times P$, все полосы B_R имеют размер $N_B \times Q$. Кроме того, создать в каждом процессе полосу C_R для хранения фрагмента матричного произведения $C = AB$, которое будет вычисляться в этом процессе; каждая полоса C_R имеет размер $N_A \times Q$ и заполняется нулевыми элементами.

Полосы, как и исходные матрицы, должны храниться по строкам в одномерных массивах соответствующего размера. Для пересылки размеров матриц использовать коллективную функцию MPI_Bcast, для пересылки полос матриц *A* и *B* использовать коллективную функцию MPI_Scatter.

Оформить все описанные действия в виде функции Matr1ScatterData (без параметров), в результате вызова которой каждый процесс получает значения N_A, *P*, N_B, *Q*, а также одномерные массивы, заполненные соответствующими полосами матриц *A*, *B*, *C*. После вызова функции Matr1ScatterData вывести в каждом процессе полученные данные

(числа N_A, P, N_B, Q и полосы матриц A, B, C). Ввод исходных данных осуществлять в функции Matr1ScatterData, вывод результатов выполнять во внешней функции Solve.

Указание. Для уменьшения числа вызовов функции MPI_Bcast все пересылаемые размеры матриц можно поместить во вспомогательный массив.

MPI9Matr3. В каждом процессе даны числа N_A, P, N_B, Q, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr2). Реализовать первый шаг ленточного алгоритма перемножения матриц, выполнив перемножение элементов, содержащихся в полосах A_R и B_R каждого процесса, после чего организовать циклическую пересылку каждой полосы B_R в процесс предыдущего ранга (из процесса 1 в процесс 0, из процесса 2 в процесс 1, …, из процесса 0 в процесс $K - 1$, где K — количество процессов).

Циклическую пересылку полос B_R выполнять с помощью функции MPI_Sendrecv_replace, используя для определения рангов процесса-отправителя и процесса-получателя операцию % взятия остатка от деления.

Оформить эти действия в виде функции Matr1Calc (без параметров). Вывести новое содержимое полос C_R и B_R в каждом процессе (ввод и вывод данных выполнять во внешней функции Solve).

Указание. В результате перемножения полос A_R и B_R каждый элемент полосы C_R будет содержать *часть* слагаемых, входящих в итоговое произведение AB; при этом будут использованы все элементы полосы B_R и часть элементов полосы A_R (в частности, на первом шаге в процессе 0 будут использованы N_B первых столбцов полосы A_0, а в процессе $K - 1$ — N_B последних столбцов полосы A_{K-1}).

MPI9Matr4. В каждом процессе даны числа N_A, P, N_B, Q, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr2). Модифицировать функцию Matr1Calc, реализованную в предыдущем задании, таким образом, чтобы она обеспечивала выполнение любого шага алгоритма ленточного умножения.

Для этого добавить к ней параметр step, определяющий номер шага (изменяется от 0 до $K - 1$, где K — количество процессов), и использовать значение этого параметра в той части алгоритма, которая связана с пересчетом элементов полосы C_R (действия, связанные с циклической пересылкой полос B_R, от значения параметра step не зависят).

Используя два вызова модифицированной функции Matr1Calc с параметрами 0 и 1, выполнить два начальных шага ленточного алгоритма

и вывести в каждом процессе новое содержимое полос C_R и B_R (ввод и вывод данных выполнять во внешней функции Solve).

Указание. Номер шага step определяет, какая часть элементов полосы A_R будет использована при очередном пересчете элементов полосы C_R (следует обратить внимание на то, что эти части перебираются циклически).

MPI9Matr5. В каждом процессе даны числа N_A, P, N_B, Q, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr2). Кроме того, в каждом процессе дано одно и то же число L, лежащее в диапазоне от 3 до K (K — количество процессов) и определяющее требуемое число шагов ленточного алгоритма.

Вызывая в цикле по параметру I ($I = 0, \ldots, L - 1$) функцию Matr1Calc(I), разработанную в предыдущем задании, выполнить L начальных шагов ленточного алгоритма и вывести в каждом процессе новое содержимое полос C_R и B_R (ввод и вывод данных выполнять во внешней функции Solve).

Примечание. Если значение L равно K, то полосы C_R будут содержать соответствующие фрагменты итогового матричного произведения AB.

MPI9Matr6. В главном процессе дано число M — количество строк результирующего матричного произведения. Кроме того, в каждом процессе даны числа N_A, Q, а также одномерные массивы, заполненные полосами матрицы C (размера $N_A \times Q$), которые были получены в результате выполнения K шагов ленточного алгоритма перемножения матриц (см. MPI9Matr5). Переслать все полосы C_R в главный процесс и вывести в нем полученную матрицу C (размера $M \times Q$).

Для хранения результирующей матрицы C в главном процессе использовать одномерный массив, достаточный для хранения матрицы размера $(N_A \cdot K) \times Q$; для пересылки данных в этот массив использовать коллективную функцию MPI_Gather.

Оформить эти действия в виде функции Matr1GatherData (без параметров). Ввод данных выполнять во внешней функции Solve, вывод полученной матрицы включить в функцию Matr1GatherData.

MPI9Matr7. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно (таким образом, исходные данные совпадают с исходными данными для задания MPI9Matr2).

Последовательно вызывая разработанные в заданиях MPI9Matr2−MPI9Matr6 функции Matr1ScatterData, Matr1Calc (в цикле)

и Matr1GatherData, получить и вывести в главном процессе матрицу C, равную произведению исходных матриц A и B.

После каждого вызова функции Matr1Calc дополнительно выводить в каждом процессе текущее содержимое полосы C_R. Перед использованием в данном задании следует модифицировать разработанную в MPI9Matr4 функцию Matr1Calc таким образом, чтобы при значении параметра step, равном $K - 1$, не выполнялась пересылка полос B_R.

MPI9Matr8. В главном процессе даны числа M, P, Q, а также имена двух файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно. Модифицировать начальный этап ленточного алгоритма перемножения матриц (см. MPI9Matr2) таким образом, чтобы каждый процесс считывал соответствующие полосы матриц A и B непосредственно из исходных файлов, используя коллективные функции MPI_File_seek и MPI_File_read_all (новый вид файловых данных создавать не требуется).

Для пересылки размеров матриц и имен файлов использовать коллективную функцию MPI_Bcast.

Оформить все действия в виде функции Matr1ScatterFile (без параметров), в результате вызова которой каждый процесс получает значения N_A, P, N_B, Q, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C. После вызова функции Matr1ScatterFile вывести в каждом процессе полученные данные (числа N_A, P, N_B, Q и полосы матриц A, B, C). Ввод исходных данных осуществлять в функции Matr1ScatterFile, вывод результатов выполнять во внешней функции Solve.

Примечание. Для некоторых полос часть элементов (последние строки) или даже вся полоса не должна считываться из исходных файлов и будет оставаться нулевой. Однако эта ситуация не требует специальной обработки, так как при достижении конца файла функция MPI_File_read_all автоматически прекращает считывание данных, не генерируя никакого сообщения об ошибке.

MPI9Matr9. В каждом процессе даны числа N_A, Q, а также одномерные массивы, заполненные полосами C_R (размера $N_A \times Q$), полученными в результате выполнения K шагов ленточного алгоритма перемножения матриц (см. MPI9Matr5). Кроме того, в главном процессе дано число M — количество строк результирующего матричного произведения и имя файла для хранения этого произведения.

Переслать число M и имя файла во все процессы (используя функцию MPI_Bcast) и записать все фрагменты матричного произведения, содержащиеся в полосах C_R, в результирующий файл, который в итоге будет содержать матрицу C размера $M \times Q$.

Для записи полос в файл использовать коллективные функции MPI_File_seek и MPI_File_write_all.

Оформить считывание имени файла, пересылку значения M и имени файла, а также все действия по записи полос в файл в виде функции Matr1GatherFile (считывание всех исходных данных, кроме имени файла, должно осуществляться во внешней функции Solve).

Указание. При записи данных в результирующий файл необходимо учитывать, что некоторые полосы C_R могут содержать завершающие нулевые строки, не связанные с полученным произведением (именно для проверки этой ситуации требуется пересылать во все процессы значение M).

MPI9Matr10. В главном процессе даны числа M, P, Q, а также имена трех файлов: вначале даются имена двух существующих файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно, а затем имя файла для хранения результирующего матричного произведения $C = AB$.

Последовательно вызывая разработанные в заданиях MPI9Matr8, MPI9Matr5 и MPI9Matr9 функции Matr1ScatterFile, Matr1Calc (в цикле) и Matr1GatherFile, получить в результирующем файле произведение исходных матриц A и B, найденное с помощью первого варианта ленточного алгоритма.

После каждого вызова функции Matr1Calc дополнительно выводить в каждом процессе текущее значение элемента c[step], где c — одномерный массив, содержащий полосу C_R, а step — номер шага алгоритма (0, 1, …, $K - 1$); таким образом, на первом шаге алгоритма следует вывести элемент c[0], на втором шаге — элемент c[1], и т. д.

18.3. Ленточный алгоритм 2 (горизонтальные и вертикальные полосы)

MPI9Matr11. В каждом процессе даны числа P и Q; кроме того, в главном процессе дана матрица B размера $P \times Q$. Известно, что число Q кратно количеству процессов K. Прочесть в главном процессе матрицу B в одномерный массив размера $P \cdot Q$ и определить новый тип MPI_BAND_B, содержащий вертикальную полосу матрицы B шириной $N_B = Q/K$ столбцов.

При определении типа MPI_BAND_B использовать функции MPI_Type_vector и MPI_Type_commit, оформив это определение в виде функции Matr2CreateTypeBand(p, n, q, t), где целочисленные параметры p, n, q являются входными, а параметр t типа MPI_Datatype является выходным; при этом параметры p и n определяют размер вертикальной полосы (число ее строк и столбцов), а параметр q — число столбцов матрицы, из которой извлекается эта полоса.

Используя тип MPI_BAND_B, переслать в каждый процесс (включая главный) соответствующую полосу матрицы B, перебирая полосы в порядке возрастания рангов процессов-получателей. Пересылку выполнять с помощью функций MPI_Send и MPI_Recv; полосы хранить в одномерных массивах размера $P \cdot N_B$. Вывести в каждом процессе полученную полосу.

Примечание. В реализации MPICH2 версии 1.3 с помощью функции MPI_Send нельзя выполнить пересылку данных в тот же процесс, из которого данные посылаются (происходит зависание программы). Для пересылки полосы в главный процесс можно использовать функцию MPI_Sendrecv; можно также заполнить соответствующую полосу в главном процессе, не прибегая к средствам библиотеки MPI.

MPI9Matr12. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно. Во втором варианте ленточного алгоритма перемножения матриц первая матрица (A) разбивается на K горизонтальных полос, а вторая (B) — на K вертикальных полос, где K — количество процессов (в дальнейшем полосы распределяются по процессам и используются для вычисления в каждом процессе части итогового матричного произведения).

Полоса для матрицы A содержит N_A строк, полоса для матрицы B содержит N_B столбцов; числа N_A и N_B вычисляются по формулам $N_A = \text{ceil}(M/K)$, $N_B = \text{ceil}(Q/K)$, где операция «/» означает вещественное деление, а функция ceil выполняет округление с избытком. Если матрица содержит недостаточно строк (или столбцов) для заполнения последней полосы, то полоса дополняется нулевыми строками (столбцами).

Сохранить исходные матрицы, дополненные при необходимости нулевыми строками или столбцами, в одномерных массивах в главном процессе, после чего организовать пересылку полос из этих массивов во все процессы: в процесс ранга R ($R = 0, 1, \ldots, K - 1$) пересылается полоса с индексом R, все полосы A_R имеют размер $N_A \times P$, все полосы B_R имеют размер $P \times N_B$. Кроме того, создать в каждом процессе полосу C_R для хранения фрагмента матричного произведения $C = AB$, который будет вычисляться в этом процессе; каждая полоса C_R имеет размер $(N_A \cdot K) \times N_B$ и заполняется нулевыми элементами.

Полосы, как и исходные матрицы, должны храниться по строкам в одномерных массивах соответствующего размера. Для пересылки размеров матриц использовать коллективную функцию MPI_Bcast, для пересылки полос матрицы A использовать коллективную функцию MPI_Scatter, для пересылки полос матрицы B использовать функции MPI_Send и MPI_Recv, а также вспомогательный тип

MPI_BAND_B, созданный с помощью функции Matr2CreateTypeBand (см. предыдущее задание и примечание к нему).

Оформить все описанные действия в виде функции Matr2ScatterData (без параметров), в результате вызова которой каждый процесс получает значения N_A, P, N_B, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C. После вызова функции Matr2ScatterData вывести в каждом процессе полученные данные (числа N_A, P, N_B и полосы матриц A, B, C). Ввод исходных данных осуществлять в функции Matr2ScatterData, вывод результатов выполнять во внешней функции Solve.

Указания. (1) При считывании матрицы B в главном процессе следует учитывать, что предназначенный для ее хранения массив может содержать элементы, соответствующие дополнительным нулевым столбцам.

(2) Для уменьшения числа вызовов функции MPI_Bcast все пересылаемые размеры матриц можно поместить во вспомогательный массив.

MPI9Matr13. В каждом процессе даны числа N_A, P, N_B, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr12). Реализовать первый шаг ленточного алгоритма перемножения матриц, выполнив перемножение элементов, содержащихся в полосах A_R и B_R каждого процесса, после чего организовать циклическую пересылку каждой полосы A_R в процесс предыдущего ранга (из процесса 1 в процесс 0, из процесса 2 в процесс 1, …, из процесса 0 в процесс $K - 1$, где K — количество процессов).

Циклическую пересылку полос A_R выполнять с помощью функции MPI_Sendrecv_replace, используя для определения рангов процесса-отправителя и процесса-получателя операцию % взятия остатка от деления.

Оформить эти действия в виде функции Matr2Calc (без параметров). Вывести новое содержимое полос C_R и A_R в каждом процессе (ввод и вывод данных выполнять во внешней функции Solve).

Указание. Поскольку в данном варианте ленточного алгоритма полосы A_R содержат полные строки матрицы A, а полосы B_R — полные столбцы матрицы B, в результате их перемножения уже на первом шаге алгоритма полоса C_R будет содержать часть элементов итогового матричного произведения (прочие элементы полосы останутся нулевыми). Расположение найденных элементов в полосе C_R зависит от ранга процесса (в частности, на первом шаге в процессе 0 будут заполнены N_A первых строк полосы C_0, а в процессе $K - 1$ — N_A последних строк полосы C_{K-1}).

MPI9Matr14. В каждом процессе даны числа N_A, P, N_B, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr12). Модифицировать функцию Matr2Calc, реализованную в предыдущем задании, таким образом, чтобы она обеспечивала выполнение любого шага алгоритма ленточного умножения.

Для этого добавить к ней параметр step, определяющий номер шага (изменяется от 0 до $K - 1$, где K — количество процессов) и использовать значение этого параметра в той части алгоритма, которая связана с пересчетом элементов полосы C_R (действия, связанные с циклической пересылкой полос A_R, от значения параметра step не зависят).

Используя два вызова модифицированной функции Matr2Calc с параметрами 0 и 1, выполнить два начальных шага ленточного алгоритма и вывести в каждом процессе новое содержимое полос C_R и A_R (ввод и вывод данных выполнять во внешней функции Solve).

Указание. Номер шага step определяет, какие наборы строк полосы C_R будут вычислены на данном этапе алгоритма (следует обратить внимание на то, что эти наборы строк перебираются циклически).

MPI9Matr15. В каждом процессе даны числа N_A, P, N_B, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr12). Кроме того, в каждом процессе дано одно и то же число L, лежащее в диапазоне от 3 до K (K — количество процессов) и определяющее требуемое число шагов ленточного алгоритма.

Вызывая в цикле по параметру I ($I = 0, \ldots, L - 1$) функцию Matr2Calc(I), разработанную в предыдущем задании, выполнить L начальных шагов ленточного алгоритма и вывести в каждом процессе новое содержимое полос C_R и A_R (ввод и вывод данных выполнять во внешней функции Solve).

Примечание. Если значение L равно K, то полосы C_R будут содержать соответствующие фрагменты итогового матричного произведения AB.

MPI9Matr16. В главном процессе даны числа M и Q — количество строк и столбцов результирующего матричного произведения. Кроме того, в каждом процессе даны числа N_A, N_B, а также одномерные массивы, заполненные полосами матрицы C (размера $(N_A \cdot K) \times N_B$), которые были получены в результате выполнения K шагов ленточного алгоритма перемножения матриц (см. MPI9Matr15). Переслать все полосы C_R в главный процесс и вывести в нем полученную матрицу C (размера $M \times Q$).

Для хранения результирующей матрицы C в главном процессе использовать одномерный массив, достаточный для хранения матрицы размера $(N_A \cdot K) \times (N_B \cdot K)$. Для пересылки полос C_R использовать функции MPI_Send и MPI_Recv, а также вспомогательный тип MPI_BAND_C, созданный с помощью функции Matr2CreateTypeBand (см. задание MPI9Matr11 и примечание к нему).

Оформить эти действия в виде функции Matr2GatherData (без параметров). Ввод данных выполнять во внешней функции Solve, вывод полученной матрицы включить в функцию Matr2GatherData.

Указание. При выводе матрицы C в главном процессе следует учитывать, что предназначенный для ее хранения массив может содержать элементы, соответствующие дополнительным нулевым столбцам.

MPI9Matr17. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно (таким образом, исходные данные совпадают с исходными данными для задания MPI9Matr12).

Последовательно вызывая разработанные в заданиях MPI9Matr12−MPI9Matr16 функции Matr2ScatterData, Matr2Calc (в цикле) и Matr2GatherData, получить и вывести в главном процессе матрицу C, равную произведению исходных матриц A и B.

После каждого вызова функции Matr2Calc дополнительно выводить в каждом процессе текущее содержимое полосы C_R.

Перед использованием в данном задании следует модифицировать разработанную в MPI9Matr14 функцию Matr2Calc таким образом, чтобы при значении параметра step, равном $K - 1$, не выполнялась пересылка полос A_R.

MPI9Matr18. В главном процессе даны числа M, P, Q, а также имена двух файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно. Известно, что число Q кратно количеству процессов K. Модифицировать начальный этап ленточного алгоритма перемножения матриц (см. MPI9Matr12) таким образом, чтобы каждый процесс считывал соответствующие полосы матриц A и B непосредственно из исходных файлов.

Для пересылки размеров матриц и имен файлов использовать коллективную функцию MPI_Bcast. Для считывания горизонтальных полос матрицы A использовать коллективные функции MPI_File_seek и MPI_File_read_all; для считывания вертикальных полос матрицы B задать соответствующий вид данных, используя функцию MPI_File_set_view и новый файловый тип MPI_BAND_B, определенный с помощью функции Matr2CreateTypeBand (см. MPI9Matr11), после чего также использовать функцию MPI_File_read_all.

Оформить все действия в виде функции Matr2ScatterFile (без параметров), в результате вызова которой каждый процесс получает значения

N_A, P, N_B, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C. После вызова функции Matr2ScatterFile вывести в каждом процессе полученные данные (числа N_A, P, N_B и полосы матриц A, B, C). Ввод исходных данных осуществлять в функции Matr2ScatterFile, вывод результатов выполнять во внешней функции Solve.

Примечание. Дополнительное условие о кратности значения Q числу K позволяет организовать считывание полос B_R с использованием одинакового файлового типа во всех процессах.

При отсутствии этого условия потребовалось бы применять специальные типы, обеспечивающие корректное считывание из файла и запись в массив «укороченных» полос матрицы B в последних процессах (кроме того, в этом случае потребовалось бы переслать каждому процессу значение Q, необходимое для правильного определения типов для «укороченных» полос).

MPI9Matr19. В каждом процессе даны числа N_A, N_B, а также одномерные массивы, заполненные полосами C_R (размера $(N_A \cdot K) \times N_B$), полученными в результате выполнения K шагов ленточного алгоритма перемножения матриц (см. MPI9Matr15). Кроме того, в главном процессе дано число M — количество строк результирующего матричного произведения и имя файла для хранения этого произведения. Дополнительно известно, что количество столбцов Q результирующего матричного произведения кратно числу процессов (и, следовательно, равно $N_B \cdot K$).

Переслать число M и имя файла во все процессы (используя функцию MPI_Bcast) и записать все фрагменты матричного произведения, содержащиеся в полосах C_R, в результирующий файл, который в итоге будет содержать матрицу C размера $M \times Q$.

Для записи полос в файл задать соответствующий вид данных, используя функцию MPI_File_set_view и новый файловый тип MPI_BAND_C, определенный с помощью функции Matr2CreateTypeBand (см. MPI9Matr11), после чего использовать коллективную функцию MPI_File_write_all.

Оформить считывание имени файла, пересылку значения M и имени файла, а также все действия по записи полос в файл в виде функции Matr2GatherFile (считывание всех исходных данных, кроме имени файла, должно осуществляться во внешней функции Solve).

Указание. При записи данных в результирующий файл необходимо учитывать, что полосы C_R могут содержать завершающие нулевые строки, не связанные с полученным произведением (именно для проверки этой ситуации требуется пересылать во все процессы значение M).

MPI9Matr20. В главном процессе даны числа M, P, Q, а также имена трех файлов: вначале даются имена двух существующих файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно, а затем имя файла для хранения результирующего матричного произведения $C = AB$. Дополнительно известно, что число Q кратно количеству процессов K.

Последовательно вызывая разработанные в заданиях MPI9Matr18, MPI9Matr15 и MPI9Matr19 функции Matr2ScatterFile, Matr2Calc (в цикле) и Matr2GatherFile, получить в результирующем файле произведение исходных матриц A и B, найденное с помощью второго варианта ленточного алгоритма.

После каждого вызова функции Matr2Calc дополнительно выводить в каждом процессе текущее значение элемента c[step], где c — одномерный массив, содержащий полосу C_R, а step — номер шага алгоритма (0, 1, …, $K - 1$); таким образом, на первом шаге алгоритма следует вывести элемент c[0], на втором шаге — элемент c[1], и т. д.

18.4. Блочный алгоритм Кэннона

MPI9Matr21. В каждом процессе даны числа M и P; кроме того, в главном процессе дана матрица A размера $M \times P$. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$, а числа M и P кратны числу K_0. Прочесть в главном процессе матрицу A в одномерный массив размера $M \cdot P$ и определить новый тип MPI_BLOCK_A, содержащий блок матрицы A размера $M_0 \times P_0$, где $M_0 = M/K_0$, $P_0 = P/K_0$.

При определении типа MPI_BLOCK_A использовать функции MPI_Type_vector и MPI_Type_commit, оформив это определение в виде функции Matr3CreateTypeBlock(m0, p0, p, t), где целочисленные параметры m0, p0, p являются входными, а параметр t типа MPI_Datatype является выходным; при этом параметры m0 и p0 определяют размеры блока, а параметр p — число столбцов матрицы, из которой извлекается этот блок.

Используя тип MPI_BLOCK_A, переслать в каждый процесс (включая главный) соответствующий блок матрицы A, перебирая блоки по строкам и пересылая их в процессы в порядке возрастания их рангов (первый блок пересылается в процесс 0, следующий за ним блок в этой же строке пересылается в процесс 1, и т. д.). Пересылку выполнять с помощью функций MPI_Send и MPI_Recv; блоки хранить в одномерных массивах размера $M_0 \cdot P_0$. Вывести в каждом процессе полученный блок.

Примечание. В реализации MPICH2 версии 1.3 с помощью функции MPI_Send нельзя выполнить пересылку данных в тот же процесс, из которого данные посылаются (происходит зависание программы). Для

пересылки блока в главный процесс можно использовать функцию MPI_Sendrecv; можно также заполнить соответствующий блок в главном процессе, не прибегая к средствам библиотеки MPI.

MPI9Matr22. В каждом процессе даны числа M_0 и P_0, а также матрица A размера $M_0 \times P_0$. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$. Прочесть в каждом процессе матрицу A в одномерный массив размера $M_0 \cdot P_0$ и определить на множестве исходных процессов коммуникатор MPI_COMM_GRID, задающий декартову топологию двумерной квадратной циклической решетки порядка K_0 (исходный порядок нумерации процессов сохраняется).

При определении коммуникатора MPI_COMM_GRID использовать функцию MPI_Cart_create, оформив это определение в виде функции Matr3CreateCommGrid(comm) с выходным параметром comm типа MPI_Comm. Вывести в каждом процессе координаты (I_0, J_0) этого процесса в созданной топологии, используя функцию MPI_Cart_coords.

Для каждой строки I_0 полученной решетки осуществить циклический сдвиг матриц A на I_0 позиций влево (т. е. в направлении убывания рангов процессов), используя функции MPI_Cart_shift и MPI_Sendrecv_replace. Вывести в каждом процессе матрицу, полученную в результате сдвига.

MPI9Matr23. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$. В блочных алгоритмах перемножения матриц исходные матрицы разбиваются на K блоков, образуя квадратные блочные матрицы порядка K_0 (в дальнейшем блоки распределяются по процессам и используются для вычисления в каждом процессе части итогового матричного произведения).

Блок для матрицы A имеет размер $M_0 \times P_0$, блок для матрицы B имеет размер $P_0 \times Q_0$; числа M_0, P_0, Q_0 вычисляются по формулам $M_0 = \text{ceil}(M/K_0)$, $P_0 = \text{ceil}(P/K_0)$, $Q_0 = \text{ceil}(Q/K_0)$, где операция «/» означает вещественное деление, а функция ceil выполняет округление с избытком. Если матрица содержит недостаточно строк (или столбцов) для заполнения последних блоков, то блоки дополняются нулевыми строками (столбцами).

Сохранить исходные матрицы A и B, дополненные при необходимости нулевыми строками и столбцами до размеров $(M_0 \cdot K_0) \times (P_0 \cdot K_0)$ и $(P_0 \cdot K_0) \times (Q_0 \cdot K_0)$, в одномерных массивах в главном процессе, после чего организовать пересылку блоков из этих массивов во все процессы, перебирая блоки по строкам и пересылая их в процессы в порядке возрастания их рангов (процесс ранга R получит блоки A_R и B_R, $R = 0, \ldots, K - 1$). Кроме того, создать в каждом процессе блок C_R для

хранения фрагмента матричного произведения $C = AB$, которое будет вычисляться в этом процессе; каждый блок C_R имеет размер $M_0 \times Q_0$ и заполняется нулевыми элементами.

Блоки, как и исходные матрицы, должны храниться по строкам в одномерных массивах соответствующего размера. Для пересылки размеров матриц использовать коллективную функцию MPI_Bcast, для пересылки блоков матриц A и B использовать функции MPI_Send и MPI_Recv, а также вспомогательные типы MPI_BLOCK_A и MPI_BLOCK_B, созданные с помощью функции Matr3CreateTypeBlock (см. задание MPI9Matr21 и примечание к нему).

Оформить все описанные действия в виде функции Matr3ScatterData (без параметров), в результате вызова которой каждый процесс получает значения M_0, P_0, Q_0, а также одномерные массивы, заполненные соответствующими блоками матриц A, B, C. После вызова функции Matr3ScatterData вывести в каждом процессе полученные данные (числа M_0, P_0, Q_0 и блоки матриц A, B, C). Ввод исходных данных осуществлять в функции Matr3ScatterData, вывод результатов выполнять во внешней функции Solve.

Указания. (1) При считывании матриц A и B в главном процессе следует учитывать, что предназначенные для ее хранения массивы могут содержать элементы, соответствующие дополнительным нулевым столбцам.

(2) Для уменьшения числа вызовов функции MPI_Bcast все пересылаемые размеры матриц можно поместить во вспомогательный массив.

MPI9Matr24. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, заполненные соответствующими блоками матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr23). Реализовать начальное перераспределение блоков, используемое в алгоритме Кэннона блочного перемножения матриц.

Для этого задать на множестве исходных процессов декартову топологию двумерной квадратной циклической решетки порядка K_0 (где $K_0 \cdot K_0$ равно количеству процессов), сохранив исходный порядок нумерации процессов, и выполнить для каждой строки I_0 полученной решетки ($I_0 = 0, \ldots, K_0 - 1$) циклический сдвиг блоков A_R на I_0 позиций влево (т. е. в направлении убывания рангов процессов), а для каждого столбца J_0 решетки ($J_0 = 0, \ldots, K_0 - 1$) циклический сдвиг блоков B_R на J_0 позиций вверх (т. е. также в направлении убывания рангов процессов).

Для определения коммуникатора MPI_COMM_GRID, связанного с декартовой топологией, использовать функцию

Matr3CreateCommGrid, реализованную в задании MPI9Matr22. При выполнении циклического сдвига использовать функции MPI_Cart_coords, MPI_Cart_shift, MPI_Sendrecv_replace (ср. с MPI9Matr22).

Оформить описанные действия в виде функции Matr3Init (без параметров). Вывести в каждом процессе блоки A_R и B_R, полученные в результате сдвига (ввод и вывод данных выполнять во внешней функции Solve).

MPI9Matr25. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, заполненные соответствующими блоками матриц A, B и C, причем известно, что блоки C_R являются нулевыми, а для блоков A_R и B_R уже выполнено их начальное перераспределение в соответствии с алгоритмом Кэннона (см. предыдущее задание). Реализовать один шаг алгоритма Кэннона перемножения матриц, выполнив перемножение элементов, содержащихся в блоках A_R и B_R каждого процесса, после чего организовать для каждой строки декартовой решетки циклический сдвиг блоков A_R на 1 позицию влево (т. е. в направлении убывания рангов процессов), а для каждого столбца решетки циклический сдвиг блоков B_R на 1 позицию вверх (т. е. также в направлении убывания рангов процессов).

Для циклической пересылки данных использовать коммуникатор MPI_COMM_GRID, создав его с помощью функции Matr3CreateCommGrid (см. задание MPI9Matr22), и функции MPI_Cart_shift и MPI_Sendrecv_replace (ср. с MPI9Matr22).

Оформить эти действия в виде функции Matr3Calc (без параметров). Вывести новое содержимое блоков C_R, A_R и B_R в каждом процессе (ввод и вывод данных выполнять во внешней функции Solve).

Примечание. Особенностью алгоритма Кэннона является то, что действия на каждом его шаге не зависят от номера шага.

MPI9Matr26. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, заполненные соответствующими блоками матриц A, B и C, причем известно, что блоки C_R являются нулевыми, а для блоков A_R и B_R уже выполнено их начальное перераспределение в соответствии с алгоритмом Кэннона (см. задание MPI9Matr24). Кроме того, в каждом процессе дано одно и то же число L, лежащее в диапазоне от 2 до K_0 (где $K_0 \cdot K_0$ равно количеству процессов) и определяющее требуемое число шагов алгоритма Кэннона.

Вызывая в цикле функцию Matr3Calc, разработанную в предыдущем задании, выполнить L начальных шагов алгоритма Кэннона и вывести в каждом процессе новое содержимое блоков C_R, A_R и B_R (ввод и вывод данных выполнять во внешней функции Solve).

Примечание. Если значение L равно K_0, то блоки C_R будут содержать соответствующие фрагменты итогового матричного произведения AB.

MPI9Matr27. В главном процессе даны числа M и Q — число строк и столбцов результирующего матричного произведения. Кроме того, в каждом процессе даны числа M_0, Q_0, а также одномерные массивы, заполненные блоками матрицы C (размера $M_0 \times Q_0$), которые были получены в результате выполнения K_0 шагов блочного алгоритма Кэннона перемножения матриц (см. MPI9Matr26). Переслать все блоки C_R в главный процесс и вывести в нем полученную матрицу C (размера $M \times Q$).

Для хранения результирующей матрицы C в главном процессе использовать одномерный массив, достаточный для хранения матрицы размера $(M_0 \cdot K_0) \times (Q_0 \cdot K_0)$. Для пересылки блоков C_R использовать функции MPI_Send и MPI_Recv, а также вспомогательный тип MPI_BLOCK_C, созданный с помощью функции Matr3CreateTypeBlock (см. задание MPI9Matr21 и примечание к нему). Оформить эти действия в виде функции Matr3GatherData (без параметров). Ввод данных выполнять во внешней функции Solve, вывод полученной матрицы включить в функцию Matr3GatherData.

Указание. При выводе матрицы C в главном процессе следует учитывать, что предназначенный для ее хранения массив может содержать элементы, соответствующие дополнительным нулевым столбцам.

MPI9Matr28. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно (таким образом, исходные данные совпадают с исходными данными для задания MPI9Matr23).

Последовательно вызывая разработанные в заданиях MPI9Matr23−MPI9Matr27 функции Matr3ScatterData, Matr3Init, Matr3Calc (в цикле) и Matr3GatherData, получить и вывести в главном процессе матрицу C, равную произведению исходных матриц A и B.

После каждого вызова функции Matr3Calc дополнительно выводить в каждом процессе текущее содержимое блока C_R.

Для того чтобы коммуникатор MPI_COMM_GRID, используемый в функциях Matr3Init и Matr3Calc, не создавался несколько раз, модифицировать функцию Matr3CreateCommGrid таким образом, чтобы при ее вызове для уже определенного коммуникатора (не равного MPI_COMM_NULL) она не выполняла никаких действий.

Перед использованием в данном задании следует модифицировать разработанную в MPI9Matr25 функцию Matr3Calc, добавив в нее параметр step, равный номеру шага (step = 0, …, $K_0 - 1$), и изменив ее таким образом, чтобы при значении параметра step, равном $K_0 - 1$, не выполнялась пересылка блоков A_R и B_R.

MPI9Matr29. В главном процессе даны числа M, P, Q, а также имена двух файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно. Модифицировать этап получения блоков для блочного алгоритма Кэннона перемножения матриц (см. MPI9Matr23) таким образом, чтобы каждый процесс считывал соответствующие блоки матриц A и B непосредственно из исходных файлов. В данном случае всем процессам требуется переслать не только размеры блоков M_0, P_0, Q_0, но и размеры исходных матриц M, P, Q, которые требуются для правильного определения позиций блоков в исходных файлах.

Для пересылки размеров матриц и имен файлов использовать коллективную функцию MPI_Bcast. Для считывания блоков использовать локальные функции MPI_File_read_at, вызывая отдельную функцию для считывания каждой строки блока (новый вид файловых данных создавать не требуется).

Оформить все действия в виде функции Matr3ScatterFile (без параметров), в результате вызова которой каждый процесс получает значения M, P, Q, M_0, P_0, Q_0, а также одномерные массивы, заполненные соответствующими блоками матриц A, B, C. После вызова функции Matr3ScatterFile вывести в каждом процессе полученные данные (числа M, P, Q, M_0, P_0, Q_0 и блоки матриц A, B, C). Ввод исходных данных осуществлять в функции Matr1ScatterFile, вывод результатов выполнять во внешней функции Solve.

Указание. При чтении файловых данных следует учитывать, что для некоторых блоков часть элементов (последние строки и/или столбцы) не должна считываться из исходных файлов и будет оставаться нулевой. Для определения фактического размера считываемого блока (числа строк и числа столбцов) потребуется использовать размеры исходных матриц и координаты блока (I_0, J_0) в квадратной декартовой решетке порядка K_0, которые легко определить по рангу процесса R: $I_0 = R/K_0$, $J_0 = R\%K_0$.

Примечание. В то время как значения P и Q необходимы для обеспечения правильного считывания файловых блоков, значение M можно не использовать, поскольку попытка чтения данных за концом файла просто игнорируется, не приводя к ошибке. Однако значение M потребуется на завершающем этапе алгоритма (см. следующее задание), поэтому его тоже необходимо переслать всем процессам.

MPI9Matr30. В каждом процессе даны числа M, Q, M_0, Q_0, а также одномерные массивы, заполненные блоками C_R (размера $M_0 \times Q_0$), полученными в результате выполнения K_0 шагов блочного алгоритма Кэннона перемножения матриц (см. MPI9Matr25). Кроме того, в главном процессе дано имя файла для хранения результирующего матричного произведения.

Переслать имя файла во все процессы (используя функцию MPI_Bcast) и записать все фрагменты матричного произведения, содержащиеся в блоках C_R, в результирующий файл, который в итоге будет содержать матрицу C размера $M \times Q$.

Для записи данных в файл использовать локальные функции MPI_File_write_at, вызывая отдельную функцию для записи каждой строки блока (новый вид файловых данных создавать не требуется).

Оформить считывание имени файла, его пересылку, а также все действия по записи данных в файл в виде функции Matr3GatherFile (считывание всех исходных данных, кроме имени файла, должно осуществляться во внешней функции Solve).

Указание. При записи файловых данных следует учитывать, что для некоторых блоков C_R часть элементов (последние строки и/или столбцы, заполненные нулями) не должна записываться в результирующий файл. См. также указание и примечание к предыдущему заданию.

MPI9Matr31. В главном процессе даны числа M, P, Q, а также имена трех файлов: вначале даются имена двух существующих файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно, а затем имя файла для хранения результирующего матричного произведения $C = AB$.

Последовательно вызывая разработанные в заданиях MPI9Matr29, MPI9Matr24, MPI9Matr25 и MPI9Matr30 функции Matr3ScatterFile, Matr3Init, Matr3Calc (в цикле) и Matr3GatherFile, получить в результирующем файле произведение исходных матриц A и B, найденное с помощью блочного алгоритма Кэннона.

После каждого вызова функции Matr3Calc дополнительно выводить в каждом процессе текущее значение элемента c[step], где c — одномерный массив, содержащий блок C_R, а step — номер шага алгоритма (0, 1, …, $K_0 - 1$); таким образом, на первом шаге алгоритма следует вывести элемент c[0], на втором шаге — элемент c[1], и т. д.

18.5. Блочный алгоритм Фокса

MPI9Matr32. В каждом процессе даны числа M и P; кроме того, в главном процессе дана матрица A размера $M \times P$. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$, а числа M и P кратны числу K_0. Прочесть в главном процессе матрицу A в одномерный массив размера $M \cdot P$ и определить новый тип MPI_BLOCK_A, содержащий блок матрицы A размера $M_0 \times P_0$, где $M_0 = M/K_0$, $P_0 = P/K_0$.

При определении типа MPI_BLOCK_A использовать функции MPI_Type_vector и MPI_Type_commit, оформив это определение в виде функции Matr4CreateTypeBlock(m0, p0, p, t), где целочисленные параметры m0, p0, p являются входными, а параметр t типа

MPI_Datatype является выходным; при этом параметры m0 и p0 определяют размеры блока, а параметр p — число столбцов матрицы, из которой извлекается этот блок.

Используя тип MPI_BLOCK_A, переслать в каждый процесс (включая главный) соответствующий блок матрицы A, перебирая блоки по строкам и пересылая их в процессы в порядке возрастания их рангов (первый блок пересылается в процесс 0, следующий за ним блок в этой же строке пересылается в процесс 1, и т. д.). Пересылку выполнять с помощью коллективной функции MPI_Alltoallw; блоки хранить в одномерных массивах размера $M_0 \cdot P_0$. Вывести в каждом процессе полученный блок.

Указания. (1) При выполнении задания с применением библиотеки MPI-1 вместо функции MPI_Alltoallw следует использовать функции MPI_Send и MPI_Recv.

(2) Функция MPI_Alltoallw, введенная в MPI-2, является единственной коллективной функцией, которая позволяет указывать смещения для пересылаемых данных в *байтах*, а не в элементах. Это дает возможность использовать ее совместно со сложными типами данных для реализации любых вариантов коллективных обменов (в данном случае требуется реализовать вариант вида Scatter).

Следует учитывать, что при подобном варианте рассылки все параметры-массивы функции MPI_Alltoallw, связанные с посылаемыми данными, необходимо по-разному определять в главном и подчиненных процессах. В частности, массив scounts (определяющий количество посылаемых данных) должен содержать значения 0 во всех подчиненных процессах и значения 1 в главном процессе (посылаемые элементы имеют тип MPI_BLOCK_A).

В то же время, массивы, связанные с принимаемыми данными, будут определяться одинаковым образом во всех процессах; в частности, в массиве rcounts (определяющем количество принимаемых данных) элемент с индексом 0 должен быть равен $M_0 \cdot P_0$, а все остальные элементы должны быть равны 0 (принимаемые элементы имеют тип MPI_INT).

Необходимо обратить особое внимание на правильное определение элементов в массиве sdispls смещений для посылаемых данных в главном процессе (в подчиненных процессах этот массив достаточно обнулить).

MPI9Matr33. В каждом процессе даны числа M_0 и P_0, а также матрица A размера $M_0 \times P_0$. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$. Прочесть в каждом процессе матрицу A в одномерный массив размера $M_0 \cdot P_0$ и определить на множестве исходных процессов коммуникатор MPI_COMM_GRID, задающий декарто-

ву топологию двумерной квадратной циклической решетки порядка K_0 (исходный порядок нумерации процессов сохраняется).

При определении коммуникатора MPI_COMM_GRID использовать функцию MPI_Cart_create, оформив это определение в виде функции Matr4CreateCommGrid(comm) с выходным параметром comm типа MPI_Comm. Вывести в каждом процессе координаты (I_0, J_0) этого процесса в созданной топологии, используя функцию MPI_Cart_coords.

На основе коммуникатора MPI_COMM_GRID создать набор коммуникаторов MPI_COMM_ROW, связанных со строками исходной двумерной решетки. Для определения коммуникаторов MPI_COMM_ROW использовать функцию MPI_Cart_sub, оформив это определение в виде вспомогательной функции Matr4CreateCommRow(grid, row) с входным параметром grid (коммуникатором, связанным с исходной двумерной решеткой) и выходным параметром row (оба параметра типа MPI_Comm). Вывести в каждом процессе его ранг R_0 для коммуникатора MPI_COMM_ROW (этот ранг должен совпадать со значением J_0).

Кроме того, для каждой строки I_0 полученной решетки осуществить пересылку матрицы A из столбца I_0 во все процессы этой же строки, используя коллективную функцию MPI_Bcast для коммуникатора MPI_COMM_ROW и сохранив результат во вспомогательной матрице T того же размера, что и матрица A (перед пересылкой необходимо скопировать в матрицу T рассылающего процесса элементы пересылаемой матрицы A). Вывести в каждом процессе полученную матрицу T.

MPI9Matr34. В каждом процессе даны числа P_0 и Q_0, а также матрица B размера $P_0 \times Q_0$. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$. Прочесть в каждом процессе матрицу B в одномерный массив размера $P_0 \cdot Q_0$ и определить на множестве исходных процессов коммуникатор MPI_COMM_GRID, задающий декартову топологию двумерной квадратной циклической решетки порядка K_0.

Для определения коммуникатора MPI_COMM_GRID использовать функцию Matr4CreateCommGrid (см. задание MPI9Matr33). Вывести в каждом процессе его координаты (I_0, J_0) в созданной топологии, используя функцию MPI_Cart_coords.

На основе коммуникатора MPI_COMM_GRID создать набор коммуникаторов MPI_COMM_COL, связанных со столбцами исходной двумерной решетки. Для определения коммуникаторов MPI_COMM_COL использовать функцию MPI_Cart_sub, оформив это определение в виде функции Matr4CreateCommCol(grid, col) с вход-

ным параметром grid (коммуникатором, связанным с исходной двумерной решеткой) и выходным параметром col (оба параметра типа MPI_Comm). Вывести в каждом процессе его ранг R_0 для коммуникатора MPI_COMM_COL (этот ранг должен совпадать со значением I_0). Кроме того, для каждого столбца J_0 полученной решетки осуществить циклический сдвиг матриц B на 1 позицию вверх (т. е. в направлении убывания рангов процессов), используя функции MPI_Sendrecv_replace для коммуникатора MPI_COMM_COL (при определении рангов процесса-отправителя и процесса-получателя использовать операцию % взятия остатка от деления). Вывести в каждом процессе матрицу, полученную в результате сдвига.

MPI9Matr35. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$. В блочных алгоритмах перемножения матриц исходные матрицы разбиваются на K блоков, образуя квадратные блочные матрицы порядка K_0 (в дальнейшем блоки распределяются по процессам и используются для вычисления в каждом процессе части итогового матричного произведения).

Блок для матрицы A имеет размер $M_0 \times P_0$, блок для матрицы B имеет размер $P_0 \times Q_0$; числа M_0, P_0, Q_0 вычисляются по формулам $M_0 = \text{ceil}(M/K_0)$, $P_0 = \text{ceil}(P/K_0)$, $Q_0 = \text{ceil}(Q/K_0)$, где операция «/» означает вещественное деление, а функция ceil выполняет округление с избытком. Если матрица содержит недостаточно строк (или столбцов) для заполнения последних блоков, то блоки дополняются нулевыми строками (столбцами).

Сохранить исходные матрицы A и B, дополненные при необходимости нулевыми строками и столбцами до размеров $(M_0 \cdot K_0) \times (P_0 \cdot K_0)$ и $(P_0 \cdot K_0) \times (Q_0 \cdot K_0)$, в одномерных массивах в главном процессе, после чего организовать пересылку блоков из этих массивов во все процессы, перебирая блоки по строкам и пересылая их в процессы в порядке возрастания их рангов (процесс ранга R получит блоки A_R и B_R, $R = 0, \ldots, K - 1$). Кроме того, создать в каждом процессе два блока, заполненных нулевыми элементами: блок C_R размера $M_0 \times Q_0$ для хранения фрагмента матричного произведения $C = AB$, которое будет вычисляться в этом процессе, и вспомогательный блок T_R того же размера $M_0 \times P_0$, что и блок A_R.

Блоки, как и исходные матрицы, должны храниться по строкам в одномерных массивах соответствующего размера. Для пересылки размеров матриц использовать коллективную функцию MPI_Bcast, для пересылки блоков матриц A и B использовать коллективную функцию MPI_Alltoallw, а также вспомогательные типы MPI_BLOCK_A и MPI_BLOCK_B, созданные с помощью функции

Matr4CreateTypeBlock (см. задание MPI9Matr32, а также указания к нему).

Оформить все описанные действия в виде функции Matr4ScatterData (без параметров), в результате вызова которой каждый процесс получает значения M_0, P_0, Q_0, а также одномерные массивы, содержащие элементы блоков A_R, B_R, C_R и T_R. После вызова функции Matr4ScatterData вывести в каждом процессе полученные данные (числа M_0, P_0, Q_0 и блоки A_R, B_R, C_R, T_R). Ввод исходных данных осуществлять в функции Matr4ScatterData, вывод данных выполнять во внешней функции Solve.

Указания. (1) При считывании матриц A и B в главном процессе следует учитывать, что предназначенные для ее хранения массивы могут содержать элементы, соответствующие дополнительным нулевым столбцам.

(2) Для уменьшения числа вызовов функции MPI_Bcast все пересылаемые размеры матриц можно поместить во вспомогательный массив.

MPI9Matr36. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, содержащие элементы блоков A_R, B_R, C_R и T_R (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr35). Каждый шаг блочного алгоритма Фокса перемножения матриц состоит из двух этапов.

На первом этапе первого шага для каждой строки I_0 квадратной решетки процессов порядка K_0 ($I_0 = 0, \ldots, K_0 - 1$, где $K_0 \cdot K_0$ равно количеству процессов) выполняется пересылка блока A_R из процесса, расположенного в строке I_0 и столбце с тем же номером I_0, во все процессы этой же строки (с сохранением пересланного блока в блоке T_R), после чего полученный в результате этой пересылки блок T_R умножается на блок B_R из этого процесса, и результат добавляется к блоку C_R.

Реализовать первый этап первого шага алгоритма Фокса. Для пересылки блоков A_R использовать функцию MPI_Bcast для коммуникатора MPI_COMM_ROW, создав этот коммуникатор с помощью функции Matr4CreateCommRow (см. задание MPI9Matr33, в котором описывается аналогичная пересылка данных).

Оформить все описанные действия в виде функции Matr4Calc1 (без параметров). Вывести новое содержимое блоков T_R и C_R в каждом процессе (ввод и вывод данных выполнять во внешней функции Solve).

MPI9Matr37. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, содержащие элементы блоков A_R, B_R, C_R и T_R (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr35).

Реализовать второй этап первого шага алгоритма Фокса, который состоит в циклическом сдвиге блоков B_R для каждого столбца декартовой решетки на 1 позицию вверх (т. е. в направлении убывания рангов процессов).

Для циклической пересылки блоков B_R использовать функцию MPI_Sendrecv_replace для коммуникатора MPI_COMM_COL, создав этот коммуникатор с помощью функции Matr4CreateCommCol (см. задание MPI9Matr34, в котором описывается аналогичная пересылка данных).

Оформить эти действия в виде функции Matr4Calc2 (без параметров). Вывести новое содержимое блока B_R в каждом процессе (ввод и вывод данных выполнять во внешней функции Solve).

MPI9Matr38. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, содержащие элементы блоков A_R, B_R, C_R и T_R (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr35).

Модифицировать функцию Matr4Calc1, реализованную в задании MPI9Matr36, таким образом, чтобы она обеспечивала выполнение первого этапа на любом шаге алгоритма Фокса. Для этого добавить к ней параметр step, определяющий номер шага (изменяется от 0 до $K_0 - 1$, где K_0 — порядок декартовой решетки процессов), и учесть значение этого шага при рассылке блоков A_R: на шаге step для каждой строки I_0 декартовой решетки должна выполняться рассылка блока A_R из процесса, расположенного в столбце с номером $(I_0 + \text{step})\%K_0$ (действия, связанные с пересчетом блоков C_R, от номера шага не зависят).

Выполнить два начальных шага алгоритма Фокса, последовательно вызвав функции Matr4Calc1(0), Matr4Calc2() (обеспечивающую второй этап шага алгоритма — см. MPI9Matr37) и Matr4Calc1(1), и вывести в каждом процессе новое содержимое блоков T_R, B_R и C_R (ввод и вывод данных выполнять во внешней функции Solve).

MPI9Matr39. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, содержащие элементы блоков A_R, B_R, C_R и T_R (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr35). Кроме того, в каждом процессе дано одно и то же число L, лежащее в диапазоне от 3 до K_0 и определяющее требуемое число шагов алгоритма Фокса.

Выполнить L начальных шагов алгоритма Фокса, вызвав функции, разработанные в заданиях MPI9Matr38 и MPI9Matr37, в следующей последовательности: Matr4Calc1(0), Matr4Calc2(), Matr4Calc1(1), Matr4Calc2(), …, Matr4Calc1($L - 1$). Вывести в каждом процессе новое содержимое блоков T_R, B_R и C_R (ввод и вывод данных выполнять во внешней функции Solve).

Примечание. Если значение L равно K_0, то блоки C_R будут содержать соответствующие фрагменты матричного произведения AB. Обратите внимание на то, что второй этап (связанный с вызовом функции Matr4Calc2) на последнем шаге алгоритма выполнять не требуется.

MPI9Matr40. В главном процессе даны числа M и Q — число строк и столбцов результирующего матричного произведения. Кроме того, в каждом процессе даны числа M_0, Q_0, а также одномерные массивы, заполненные блоками матрицы C (размера $M_0 \times Q_0$), которые были получены в результате выполнения K_0 шагов блочного алгоритма Фокса перемножения матриц (см. MPI9Matr39).

Переслать все блоки C_R в главный процесс и вывести в нем полученную матрицу C (размера $M \times Q$). Для хранения результирующей матрицы C в главном процессе использовать одномерный массив, достаточный для хранения матрицы размера $(M_0 \cdot K_0) \times (Q_0 \cdot K_0)$. Для пересылки блоков C_R использовать коллективную функцию MPI_Alltoallw, а также вспомогательный тип MPI_BLOCK_C, созданный с помощью функции Matr4CreateTypeBlock (см. задание MPI9Matr32, а также указания к нему).

Оформить эти действия в виде функции Matr4GatherData (без параметров). Ввод данных выполнять во внешней функции Solve, вывод полученной матрицы включить в функцию Matr4GatherData.

Указание. При выводе матрицы C в главном процессе следует учитывать, что предназначенный для ее хранения массив может содержать элементы, соответствующие дополнительным нулевым столбцам.

MPI9Matr41. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно (таким образом, исходные данные совпадают с исходными данными для задания MPI9Matr35).

Последовательно вызывая разработанные в заданиях MPI9Matr35−MPI9Matr40 функции Matr4ScatterData, Matr4Calc1, Matr4Calc2 и Matr4GatherData, получить и вывести в главном процессе матрицу C, равную произведению исходных матриц A и B. Функции Matr4Calc1 и Matr4Calc2 должны вызываться в цикле, причем количество вызовов функции Matr4Calc2 должно быть на 1 меньше количества вызовов функции Matr4Calc1.

После каждого вызова функции Matr4Calc1 дополнительно выводить в каждом процессе текущее содержимое блока C_R.

Для того чтобы вспомогательные коммуникаторы MPI_COMM_GRID, MPI_COMM_ROW и MPI_COMM_COL, используемые в функциях Matr4Calc1 и Matr4Calc2, не создавались несколько раз, модифицировать функции Matr4CreateCommGrid, Matr4CreateCommRow, Matr4CreateCommCol (см. MPI9Matr33 и MPI9Matr34) таким образом,

чтобы при их вызове для уже определенного коммуникатора (не равного MPI_COMM_NULL) они не выполняли никаких действий.

MPI9Matr42. В главном процессе даны числа M, P, Q, а также имена двух файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно. Дополнительно известно, что числа M, P и Q кратны порядку K_0 квадратной решетки процессов.

Модифицировать этап формирования блоков для блочного алгоритма Фокса перемножения матриц (см. MPI9Matr35) таким образом, чтобы каждый процесс считывал соответствующие блоки матриц A и B непосредственно из исходных файлов.

Для пересылки размеров матриц и имен файлов использовать коллективную функцию MPI_Bcast. Для считывания блоков задать соответствующий вид данных, используя функцию MPI_File_set_view и типы MPI_BLOCK_A и MPI_BLOCK_B, определенные с помощью функции Matr4CreateTypeBlock (см. MPI9Matr32) после чего использовать коллективную функцию MPI_File_read_all.

Оформить все действия в виде функции Matr4ScatterFile (без параметров), в результате вызова которой каждый процесс получает значения M_0, P_0, Q_0, одномерные массивы, содержащие элементы блоков A_R, B_R, C_R и T_R (блоки C_R и T_R должны быть нулевыми). После вызова функции Matr4ScatterFile вывести в каждом процессе полученные данные (числа M_0, P_0, Q_0 и блоки A_R, B_R, C_R, T_R). Ввод исходных данных осуществлять в функции Matr4ScatterFile, вывод результатов выполнять во внешней функции Solve.

Примечание. Дополнительное условие о кратности чисел M, P, Q числу K_0 означает, что блоки, полученные из матриц A и B, не требуется дополнять нулевыми строками и/или столбцами, и поэтому для чтения из файлов блоков матриц A и B в любые процессы можно использовать одинаковые файловые типы MPI_BLOCK_A и MPI_BLOCK_B.

При отсутствии этого условия потребовалось бы применять специальные типы, обеспечивающие корректное считывание из файла и запись в массив «укороченных» блоков матриц A и B (кроме того, в этом случае потребовалось бы переслать каждому процессу значения P и Q, необходимые для правильного определения типов для «укороченных» блоков).

MPI9Matr43. В каждом процессе даны числа M_0, Q_0, а также одномерные массивы, заполненные блоками C_R (размера $M_0 \times Q_0$), полученными в результате выполнения K_0 шагов блочного алгоритма Фокса перемножения матриц (см. MPI9Matr39). Кроме того, в главном процессе дано имя файла для хранения результирующего матричного произведения. Дополнительно известно, что число строк M и число столбцов

Q матричного произведения кратны порядку K_0 квадратной решетки процессов (таким образом, $M = M_0 \cdot K_0$, $Q = Q_0 \cdot K_0$).

Переслать имя файла во все процессы (используя функцию MPI_Bcast) и записать все фрагменты матричного произведения, содержащиеся в блоках C_R, в результирующий файл, который в итоге будет содержать матрицу C размера $M \times Q$.

Для записи блоков задать соответствующий вид данных, используя функцию MPI_File_set_view и файловый тип MPI_BLOCK_C, определенный с помощью функции Matr4CreateTypeBlock (см. MPI9Matr32), после чего использовать коллективную функцию MPI_File_write_all.

Оформить считывание имени файла, его пересылку, а также все действия по записи данных в файл в виде функции Matr4GatherFile (считывание всех исходных данных, кроме имени файла, должно осуществляться во внешней функции Solve).

Примечание. Дополнительное условие о кратности чисел M и Q числу K_0 означает, что блоки C_R не содержат «лишних» нулевых строк и/или столбцов, и поэтому для их записи в файл из любых процессов можно использовать одинаковые файловые типы MPI_BLOCK_C.

MPI9Matr44. В главном процессе даны числа M, P, Q, а также имена трех файлов: вначале даются имена двух существующих файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно, а затем имя файла для хранения результирующего матричного произведения $C = AB$. Дополнительно известно, что числа M, P и Q кратны порядку K_0 квадратной решетки процессов.

Последовательно вызывая разработанные в заданиях MPI9Matr42, MPI9Matr38, MPI9Matr37 и MPI9Matr43 функции Matr4ScatterFile, Matr4Calc1, Matr4Calc2 и Matr4GatherFile, получить в результирующем файле произведение исходных матриц A и B, найденное с помощью блочного алгоритма Фокса. Функции Matr4Calc1 и Matr4Calc2 должны вызываться в цикле, причем количество вызовов функции Matr4Calc2 должно быть на 1 меньше количества вызовов функции Matr4Calc1.

После каждого вызова функции Matr4Calc1 дополнительно выводить в каждом процессе текущее значение элемента c[step], где c — одномерный массив, содержащий блок C_R, а step — номер шага алгоритма ($0, 1, \ldots, K_0 - 1$); таким образом, на первом шаге алгоритма следует вывести элемент c[0], на втором шаге — элемент c[1], и т. д.