

Suggested solutions for the IN3200/IN4200 exam of spring 2026

Question 1 (3 points)

What does CPU stand for in the context of high-performance computing?

Select one alternative:

- Computing power unit
- Central programming unit
- Central processing unit
- Central performance unit
- Central profiling unit

Suggested solution: “Central processing unit” is correct.

Question 2 (3 points)

Consider a 16-way set-associative cache with 2 MB as the capacity and 64 bytes as the cacheline size.

How many possible locations can a specific cacheline from memory be placed in this cache?

Select one alternative:

- 1
- 16
- 64
- 32768
- 1048576

Suggested solution: “16” is correct.

Question 3 (3 points)

Amdahl's law can be used to estimate speedup by calculating

$$\frac{1}{s + \frac{p}{N}}$$

What do p and N mean?

Select one alternative:

- ' p ' is the problem size, ' N ' is the number of workers
- ' p ' is the number of workers, ' N ' is the problem size
- ' p ' is the percentage of parallelizable work, ' N ' is the problem size
- ' p ' is the percentage of floating-point calculation, ' N ' is the number of workers
- ' p ' is the percentage of parallelizable work, ' N ' is the number of workers

Suggested solution: ' p ' is the percentage of parallelizable work, ' N ' is the number of workers.

Question 4 (9 points)

Which of the following statements are incorrect?

Select one or more alternatives:

- Memory bandwidth alone determines how much time it takes to load a cacheline from memory.
- It is the programmer's responsibility to carry out branch prediction.
- On a computer, loading data from cache is always faster than loading data from memory.
- Deadlock can happen in OpenMP programs.
- The possibility of having 'false sharing' is due to private caches on a shared-memory system.
- A parallel program that has no communication overhead should achieve perfect speedup.

Suggested solution:

- Memory bandwidth alone determines how much time it takes to load a cacheline from memory. (Incorrect: because latency also matters.)
- It is the programmer's responsibility to carry out branch prediction. (Incorrect: because it's a hardware task.)
- On a computer, loading data from cache is always faster than loading data from memory. (Correct.)
- Deadlock can happen in OpenMP programs. (Correct.)

- The possibility of having 'false sharing' is due to private caches on a shared-memory system. (Correct.)
- A parallel program that has no communication overhead should achieve perfect speedup. (Incorrect: because other factors can also lead to imperfect speedup, such as load imbalance, non-parallelizable computation, competition for non-communication resources, and parallelization-induced extra computation.)

Question 5 (10 points)

Suppose a processor uses a 5-stage pipeline to execute instructions. Stage 1 of the pipeline takes 5 clock cycles to complete, stage 2 takes 4 cycles, stage 3 takes 3 cycles, stage 4 takes 2 cycles, and stage 5 takes 1 cycle. How many cycles are necessary to execute 1000 instructions?

Suggested solution: The first instruction will require $5+4+3+2+1=15$ clock cycles to complete. Afterwards, a new instruction will complete every 5 cycles. Thus, 1000 instructions will require in total $15+999 \times 5=5010$ clock cycles.

Question 6 (12 points)

```
for (int i=0; i<100; i++)
  for (int j=0; j<100; j++)
    for (int k=0; k<100; k++)
      if (i>0 && i<99 && j>0 && j<99 && k>0 && k<99)
        values[k][j][i] = sin(Pi*(i+j+k)/100.0);
      else
        values[k][j][i] = 0.0;
```

The above serial code is written in the C language. Please rewrite it with the purpose of optimising the performance when executed on a single processor core, while achieving the same computational result. Please add comments in your rewritten code to provide necessary explanations.

Note: In the above code, 'sin' is the standard mathematical sine function, 'Pi' is a constant numerical value equal to π , and 'values' is a 3D array of floating-point values of dimension $100 \times 100 \times 100$.

Suggested solution: There are three problems of inefficiency in the original code:

1. Memory-inefficient traversal of the 3D array, which can be fixed by swapping the loop levels.
2. Branching inside the for-loops, which can be avoided by loop peeling.
3. Unnecessarily repeated evaluations of the very expensive sine function. Note: the sine function is periodic by 2π , so there can only be 200 different results when evaluating $\sin(\pi(i+j+k)/100)$ for any integer values of i , j and k . A lookup table of pre-calculated values should be used.

An optimized code can be as follows:

```

int i,j,k;
double *lookup_table = malloc(200*sizeof(double));

lookup_table[0] = 0.0;    // sin(0) = 0.0
for (i=1; i<=49; i++)
    lookup_table[i] = sin(Pi*i/100.0);
lookup_table[50] = 1.0;    // sin(Pi/2) = 1.0
for (i=51; i<=100; i++)
    lookup_table[i] = lookup_table[100-i];    // sine function is symmetric about Pi/2
for (i=101; i<200; i++)
    lookup_table[i] = -lookup_table[i-100];    // sin(x) = -sin(x-Pi)

for (j=0; j<100; j++)
    for (i=0; i<100; i++)
        data[0][j][i] = 0.0;

for (k=1; k<99; k++) {
    for (i=0; i<100; i++)
        data[k][0][i] = 0.0;
    for (j=1; j<99; j++) {
        data[k][j][0] = 0.0;
        for (i=1; i<99; i++)
            data[k][j][i] = lookup_table[(k+j+i)%200];
        data[k][j][99] = 0.0;
    }
    for (i=0; i<100; i++)
        data[k][99][i] = 0.0;
}

for (j=0; j<100; j++)
    for (i=0; i<100; i++)
        data[99][j][i] = 0.0;

```

Observation: the number of sine function evaluations is reduced from 98^3 to 49. If the action of giving zero values to the boundary layers is programmed as separate for-loops, this may unnecessarily cause additional memory traffic, bad for performance.

Question 7 (5 points)

Sampled dense matrix multiplication is mathematically denoted by $C = (A \times B) \odot S$, where A , B , C and S are all square matrices with n rows and n columns. Here, A and B are dense matrices, whereas C and S are sparse matrices and have the same sparsity pattern. Each nonzero value in sparse matrix C , on row i and column j , is calculated as $C_{i,j} = S_{i,j} (A_{i,1}B_{1,j} + A_{i,2}B_{2,j} + \dots + A_{i,n}B_{n,j}) = S_{i,j} (\sum_{k=1}^n A_{i,k}B_{k,j})$, where $S_{i,j}$ is the corresponding nonzero value in sparse matrix S on row i and column j .

Please write a function in the C language, with the following syntax, to implement a sampled dense matrix multiplication:

```
void sampled_matrix_multiplication_crs (struct sparse_mat_crs *C, double **A, double **B,
```

```
struct sparse_mat_crs *S)
```

You can assume that the definition of `struct sparse_mat_crs` is as follows:

```
struct sparse_mat_crs {
    int n, nnz;
    double *val;
    int *row_ptr, *col_idx;
};
```

Note: the sparse matrices C and S are both stored in the compressed row storage format. The 'row_ptr' and 'col_idx' arrays have the same contents for the 'C' and 'S' structs, where array 'val' in struct 'S' is of length 'nnz' and contains all the nonzeros in sparse matrix S . The 'val' array in struct 'C' is also of length 'nnz', and its content is to be calculated inside function `sampled_matrix_multiplication_crs`. The values of dense matrices A and B are stored in two separate 2D arrays, each with n rows and n columns.

Suggested solution:

```
void sampled_matrix_multiplication_crs (struct sparse_mat_crs *C, double **A, double **B,
                                      struct sparse_mat_crs *S)
{
    int i, j, k, m;
    for (i=0; i<C->n; i++)
        for (m=C->row_ptr[i]; m<C->row_ptr[i+1]; m++) {
            double v = 0.0;
            j = C->col_idx[m];
            for (k=0; k<C->n; k++)
                v += A[i][k]*B[k][j];
            C->val[m] = S->val[m]*v;
        }
}
```

Question 8 (15 points)

Following the previous question of sampled dense matrix multiplication, how many floating-point operations are needed (when all the matrices are of dimension $n \times n$ and sparse matrices C and S have nnz nonzero values)?

On a computer with a single-level cache, how much memory traffic will occur in connection with the sampled dense matrix multiplication? Please discuss two scenarios: (1) the cache capacity is very small with respect to the problem size; (2) the cache capacity is very large with respect to the problem size. (You can assume that the cache is empty before the calculation starts, and that sparse matrices C and S have at least one nonzero value per row, and at least one nonzero value per column.)

Only for IN4200 students: How can the memory traffic be reduced even if the the cache capacity is very small with respect to the problem size? Which changes are needed in the data structure and the implementation? (You don't need to code, it is sufficient to briefly explain the idea and strategy.)

Suggested solution:

- For computing each nonzero value in C , we need n additions and $n + 1$ multiplications. So the total number of floating-point operations is $(2n + 1)nnz$. (It is also possible to only use $n - 1$ additions per nonzero value, leading to totally $2n \cdot nnz$ floating-point operations.)
- For a very large cache ('everything is cached perfectly'), the amount of memory traffic associated with loading values from A and B will be $16n^2$ bytes in total. (Note: we have assumed that each row of A is used at least once, and each column of B is used at least once.) The amount of memory traffic associated with sparse matrix C is $4(n + 1)$ bytes related to loading its array 'row_ptr', $4nnz$ bytes related to loading its array 'col_idx', and $8nnz$ bytes related to storing its 'val' array. (If stores require loading cachelines beforehand, $16nnz$ bytes are needed.) For sparse matrix S , the memory traffic related to loading its array 'val' is $8nnz$. (The arrays 'row_ptr' and 'col_idx' of S are not used at all.)
- For a very small cache, however, we run the risk of having to (re)load an entire row of A for each nonzero in C (even for computing multiple nonzeros on the row of C). So the amount of memory loaded from A can be $nnz \cdot 8n$ bytes. For matrix B , it is even worse (due to loading its columns). Specifically, the amount of memory loaded from B can be as much as $nnz \cdot n \cdot (\text{size of cacheline})$. Regarding C , although the code goes through its arrays 'row_ptr', 'col_idx' and 'val' in an ideal sequence, reloading the same cachelines for these arrays can still be the result due to too small cache capability (for example, an entire row of A and an entire column of B need to be loaded between storing $C \rightarrow \text{val}[m]$ and $C \rightarrow \text{val}[m+1]$). Therefore, the loading traffic related to 'row_ptr' will be approximately $n \cdot (\text{size of cacheline})$, the loading traffic related to 'col_idx' will be $nnz \cdot (\text{size of cacheline})$, the storing traffic related to 'val' will also be $nnz \cdot (\text{size of cacheline})$, plus the same amount for the loading traffic (if load is needed before store). The loading traffic related to loading the 'val' array of S will also be $nnz \cdot (\text{size of cacheline})$.
- **Only for IN4200 students:** One problem for a "small" cache is: A row from matrix A has to be reloaded from memory when later computing another nonzero value on the same row of C . Another problem for a "small" cache is that reloading the same columns from matrix B is inevitable (and extra expensive). To alleviate these problems, one strategy is to **not** calculate the nonzeros in sparse matrix C row by row. Instead, we can calculate the nonzeros in a "tiled" manner, while the calculation is done in a "step-wise" procedure where each step uses one block from A and one block from B (similar to Cannon's algorithm in the context of parallelizing dense matrix-matrix multiplications). This will allow the calculation of several nonzeros on the same row of C to reuse values loaded from A . Similarly, such a "tiled" ordering of the nonzeros may also allow some reuse of values loaded from B . To enable such a memory-traffic-reducing strategy, however, we need to have a new data structure for C and S , such that each of them is stored as a "coarse" matrix where each entry (a "tile") is itself a sparse sub-matrix.

Question 9 (10 points)

```
// start of code piece number 1
int i,j;
double sum=0.0;
for (i=0; i<9999; i++)
```

```

    for (j=0; j<10; j++) {
        data[i][j] = data[i+1][j];
        sum = sum + data[i][j];
    }
//end of code piece number 1

//start of code piece number 2
#define max_num_threads 20000
int i,j, num_threads, thread_id;
double sum=0.0, p_sum[max_num_threads];
for (i=0; i<max_num_threads; i++)
    p_sum[i] = 0;

#pragma omp parallel
{
    thread_id = omp_get_num_threads();
    num_threads = omp_get_thread_num();
#pragma omp for
    for (i=0; i<9999; i++)
        for (j=0; j<10; j++) {
            data[i][j] = data[i+1][j];
            p_sum[thread_id] = p_sum[thread_id] + data[i][j];
        }
} // end of omp parallel

for (i=0; i<num_threads; i++)
    sum = sum + p_sum[i];

```

We have two pieces of code. The first one (containing just the nested for-loop) is a serial C code. The second piece of code is an attempt at parallelizing the serial code using OpenMP. But it has programming errors and inefficiencies. What are these errors and inefficiencies in the OpenMP code? (Please provide simple explanations if needed.)

Note: the 2D array 'data' is of dimension 10000×10 .

Suggested solution: In the OpenMP code there are the following programming errors:

1. The variable `j` is shared, thus leading to race conditions when used in the `j`-index for loop.
2. The variables `thread_id` and `num_threads` are also shared, thus leading to race conditions when they are assigned with values (simultaneously by all the threads) inside the parallel region.
3. The use of functions `omp_get_num_threads` and `omp_get_thread_num` should be swapped.
4. The `i`-indexed for loop cannot be directly parallelized using OpenMP's `for` directive. This is because the last row assigned to each thread depends on the first row assigned to the next thread.

In addition to the above programming errors, there are two inefficiencies:

1. The use of array `p_sum` is unnecessary. (It can be completely avoided by using OpenMP's reduction clause. See the solution suggestion below.)

2. Frequent updates of `p_sum[thread_id]` by each thread will **not** cause race conditions. However, due to the closeness of the different `p_sum[thread_id]` values in memory, **false sharing** will cause a major inefficiency.

Question 10 (10 points)

Continue with the previous question, please write a correct and efficient OpenMP implementation.

Suggested solution: The work of updating the 9999 rows of data (except its last row) are to be evenly divided among the threads, where each thread gets a consecutive block of rows. It is important to note that the new content of each thread's last row is the old content of the next thread's top row. To avoid that the old content of this particular row is overwritten "too early", it needs to be stored temporarily somewhere beforehand. (**Note: OpenMP's `for` directive cannot be used directly for the `i`-indexed `for` loop. The `j`-indexed `for` loop has too few iterations to be worthy of parallelization.**) Last but not least, the calculation of `sum` can be easily achieved by using `reduction` in OpenMP.

```
int i, j;
double sum=0.0;

#pragma omp parallel private(j) reduction(+: sum)
{
    int thread_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    int my_start, my_end;
    double my_last_row[10];

    my_start = (thread_id*9999)/num_threads;
    my_end = ((thread_id+1)*9999)/num_threads;
    for (j=0; j<10; j++)
        my_last_row[j] = data[my_end][j];

#pragma omp barrier          // important to synchronize all the threads here

    for (i=my_start; i<my_end-1; i++)
        for (j=0; j<10; j++) {
            data[i][j] = data[i+1][j];
            sum += data[i][j];
        }

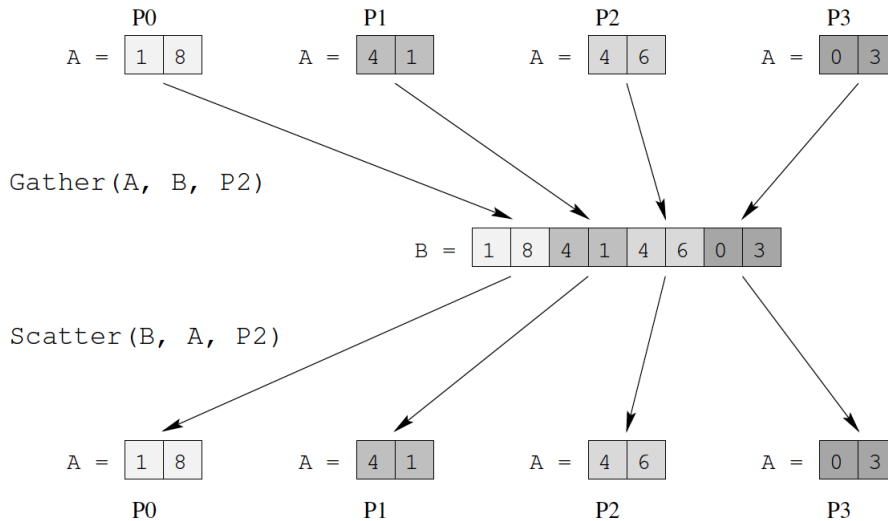
    i = my_end - 1;
    for (j=0; j<10; j++) {
        data[i][j] = my_last_row[j];
        sum += data[i][j];
    }
} // end of parallel region
```

Question 11 (10 points)

```
MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
          void *recvbuf, int recvcount, MPI_Datatype recvtype,
          int root, MPI_Comm comm)
```

The MPI collective command `MPI_Gather` lets a 'root process' gather an equal amount data from all MPI processes in a communicator, see the illustrating figure.

(a) Equal-Size Gather and Scatter Operations



You are now asked to develop a special implementation, named `MPI_Gather_p2p`, that can only use point-to-point MPI communication to achieve the same result as `MPI_Gather`, while also avoiding that the 'root process' becomes a clear performance bottleneck.

For IN3200 students: Explain your algorithm (without actual MPI coding) by using an example of having 16 MPI processes in the communicator, with process 0 being the root process. Please state clearly how many point-to-point messages will be communicated. For each point-to-point message, explain who the sender is, who the receiver is, the content of message, and when it will take place.

The time usage of communicating a single point-to-point message of size M bytes can be estimated as $c + \frac{M}{B}$, where latency and communication bandwidth are known constant values. How much time does it take to complete your `MPI_Gather_p2p` in a communicator with 16 MPI processes, each having M bytes of data, to be gathered by process 0?

For IN4200 students: Explain your algorithm (without actual MPI coding) by using an example of having 13 MPI processes in the communicator, where process with rank 5 is the root process. Please state clearly how many point-to-point messages will be communicated. For each point-to-point message, explain who the sender is, who the receiver is, the content of message, and when it will take place.

The time usage of communicating a single point-to-point message of size M bytes can be estimated as $c + \frac{M}{B}$, where latency c and communication bandwidth B are known constant values. How much time does it take to complete your `MPI_Gather_p2p` in a communicator with 13 MPI processes, each having M bytes of data, to be gathered by process 5?

Suggested solution: Let us denote the processes by $P_0, P_1, \dots$, etc. The data to be collected from P_i is denoted by D_i .

For IN3200 students

There are multiple equally good solutions which all have in total 4 stages. The following is one example.

During stage 1, **8 messages are sent simultaneously**: P1 sends D1 to P0; P3 sends D3 to P2; P5 sends D5 to P4; P7 sends D7 to P6; P9 sends D9 to P8; P11 sends D11 to P10; P13 sends D13 to P12; P15 sends D15 to P14.

During stage 2, **4 messages are sent simultaneously**: P2 sends D2D3 to P0; P6 sends D6D7 to P4; P10 sends D10D11 to P8; P14 sends D14D15 to P12.

During stage 3, **2 messages are sent simultaneously**: P4 sends D4D5D6D7 to P0; P12 sends D12D13D14D15 to P8.

During stage 4, one message containing D8D9D10D11D12D13D14D15 is sent from P8 to P0.

The total time needed will be

$$c + \frac{M}{B} + c + \frac{2M}{B} + c + \frac{4M}{B} + c + \frac{8M}{B} = 4c + \frac{15M}{B}$$

For IN4200 students

There are multiple equally good solutions which all have in total 4 stages. The following is one example.

During stage 1, **6 messages are sent simultaneously**: P2 sends D2 to P1; P4 sends D4 to P3; P6 sends D6 to P5; P8 sends D8 to P7; P10 sends D10 to P9; P12 sends D12 to P11.

During stage 2, **3 messages are sent simultaneously**: P3 sends D3D4 to P1; P7 sends D7D8 to P5; P11 sends D11D12 to P9.

During stage 3, **2 messages are sent simultaneously**: P1 sends D1D2D3D4 to P0; P9 sends D9D10D11D12 to P5.

During stage 4, one message containing D0D1D2D3D4 is sent from P0 to P5.

The total time needed will be

$$c + \frac{M}{B} + c + \frac{2M}{B} + c + \frac{4M}{B} + c + \frac{5M}{B} = 4c + \frac{12M}{B}$$

Question 12 (10 points) for IN3200 students

```
void MPI_Gather_p2p (const int *sendbuf, int sendcount,
                    int *recvbuf, MPI_Comm comm)
```

Following the previous question, please implement your own `MPI_Gather_p2p`, which is meant to be called by all MPI processes inside a communicator. (Remember: only point-to-point messages can be used inside `MPI_Gather_p2p`.) **Note:** This function is simplified than the standard `MPI_Gather` such that only integers will be the content of the messages. It also assumes the following: the `sendcount` value is the same on all processes, meaning the number of integers to be sent from each process; process 0 is the root process, thus not specified as input to the function; `recvbuf` points to a pre-allocated large enough buffer in the memory of the root process; the total number of MPI processes in the communicator is power of 2 (not necessarily 16).

Suggested solution:

```
void MPI_Gather_p2p (const int *sendbuf, int sendcount, int *recvbuf, MPI_Comm comm)
{
    int my_rank, total_procs, i, done;
    int *tempbuf, tempbuf_size=sendcount;
    MPI_Status status;

    MPI_Comm_rank (comm, &my_rank);
    MPI_Comm_size (comm, &total_procs);

    for (i=2; i<total_procs; i*=2) // assuming total_procs is power of 2
        if ((my_rank%i)==0)
            tempbuf_size *=2;

    if ((my_rank%2)==0) {
        if (my_rank==0)
            tempbuf = recvbuf; // rank 0 is the root, reuse recvbuf as tempbuf
        else
            tempbuf = malloc(tempbuf_size*sizeof(int)); // allocate a temporary buffer
        for (i=0; i<sendcount; i++)
            tempbuf[i] = sendbuf[i];
    }
    else
        tempbuf = sendbuf; // on odd-numbered rank, let tempbuf point to sendbuf

    for (done=0, i=1; i<total_procs; i*=2)
        if (!done) {
            if ((my_rank%(2*i))==0)
                MPI_Recv (&(tempbuf[i*sendcount]), i*sendcount, MPI_INT, my_rank+i, i, comm, &status);
            else {
                MPI_Send (tempbuf, i*sendcount, MPI_INT, my_rank-i, i, comm);
                done = 1;
            }
        }

    if ((my_rank%2)==0 && my_rank!=0)
        free(tempbuf);
}
```

Question 12 (10 points) for IN4200 students

```
void MPI_Gather_p2p (const int *sendbuf, int sendcount,
    int *recvbuf, int root, MPI_Comm comm)
```

Following the previous question, please implement your own `MPI_Gather_p2p`, which is meant to be called by all MPI processes inside a communicator. (Remember: only point-to-point messages can be used inside `MPI_Gather_p2p`.) **Note:** This function is simplified than the standard `MPI_Gather` such that only integers will be the content of the messages. It also assumes the following: the `sendcount` value is the same on all processes, meaning the number of integers to be sent from each process; `recvbuf` points to a pre-allocated large enough buffer in the memory of the root process. The implementation should handle any total number of MPI processes in the communicator (not necessarily 13).

Suggested solution:

```
void MPI_Gather_p2p (const int *sendbuf, int sendcount, int *recvbuf, int root, MPI_Comm comm)
{
    int my_rank, total_procs, i, j, done, my_new_rank;
    int *tempbuf, *msg_size, offset;
    MPI_Status status;
```

```

MPI_Comm_size (comm, &total_procs);
MPI_Comm_rank (comm, &my_rank);
my_new_rank = (my_rank-root+total_procs)%total_procs; // shift all ranks so 'root' has 0 as new rank

msg_size = malloc(total_procs*sizeof(int)); // total amount of data each process temporarily contains
for (j=0; j<total_procs; j++)
    msg_size[j] = sendcount;

for (i=2; i<2*total_procs; i*=2)
    for (j=0; j<total_procs; j+=i)
        if ((j%i)==0 && (j+i/2)<total_procs)
            msg_size[j] += msg_size[j+i/2];

if ((my_new_rank%2)==0) {
    tempbuf = malloc(msg_size[my_new_rank]*sizeof(int)); // allocate a temporary buffer
    for (i=0; i<sendcount; i++)
        tempbuf[i] = sendbuf[i];
}
else
    tempbuf = sendbuf; // on odd-numbered new rank, let tempbuf point to sendbuf

for (done=0, offset=sendcount, i=1; i<total_procs; i*=2)
    if (!done) {
        if ((my_new_rank%(2*i))==0 && (my_new_rank+i)<total_procs) {
            MPI_Recv (&(tempbuf[offset]), msg_size[my_new_rank+i], MPI_INT, (my_rank+i)%total_procs, i, comm, &status);
            offset += msg_size[my_new_rank+i];
        }
        else if (my_new_rank%(2*i)) {
            MPI_Send (tempbuf, msg_size[my_new_rank], MPI_INT, (my_rank-i+total_procs)%total_procs, i, comm);
            done = 1;
        }
    }

if (my_rank==root) {
    for (i=0; i<root*sendcount; i++)
        recvbuf[i] = tempbuf[i+(total_procs-root)*sendcount];
    for (i=root*sendcount; i<total_procs*sendcount; i++)
        recvbuf[i] = tempbuf[i-root*sendcount];
}

if ((my_new_rank%2)==0)
    free(tempbuf);
free(msg_size);
}

```