Exam — Parallel Programming

1st session — January 8th, 2025 — Duration : 2 hours.

All documents are allowed. **Like in all exams, your answers must come with explanations.** There are 3 independent parts on 3 pages. Don't hesitate to give examples or draw pictures.

Exercice 1 – Implementation of `MPI_Alltoall`

We consider the `MPI_Alltoall` function :

```c
int MPI_Alltoall(const void *sendbuf, int sendcount,
        MPI_Datatype sendtype, void *recvbuf, int recvcount,
        MPI_Datatype recvtype, MPI_Comm comm);
```

Recall that this collective operation has the following effect : each process sends the i-th slice of `sencdount` elements taken from `sendbuf` to the process of rank i. In other terms, each process sends `sendbuf[i*sendcount:(i+1)*sendcount]` to the process of rank i. At the same time, each process receives the data of the process of rank i in `recvbuf[i*recvcount:(i+1)*recvcount]`.

1. Provide an absolute lower-bound on the time taken to complete this operation, assuming that there are p ranks in the communicator and that the size of `sendbuf` is $n =$ `sendcount` $\times p$.

Solution :

Because each process needs to transmit $\frac{p-1}{p}n$ items on the network, if the network has bandwidth β, then this must take at least $\beta\frac{p-1}{p}n$ seconds.

In addition, data from process 0 must reach all other processes. This requires at least $\log_2(p-1)$ messages to be sent in sequence. All-in-all, the lower-bound is $\alpha \log_2 p + \beta\frac{p-1}{p}n$, if α denotes the latency of the network.

2. Assume that the number of processes is odd ($p = 2k + 1$). There is a simple algorithm in p *communication rounds*, where in step $0 \leq t < p$, process j exchange data with process $t - j \bmod p$. (each process may send a single message during a "communication round", regardless of its size). What is the running time ?

Solution :

In each time step, all processes send message of size n/p. So the total running time is $p\alpha + n\beta$. The bandwidth term is nearly optimal, but the latency term is bad, especially for small messages.

3. With $n = 5$, draw a series of 5 pictures to illustrate the communication pattern.
4. Write the code of a C function that implement this algorithm *in-place* (the same buffer contain the input data on function entry and the output data on function exit). Do not allocate extra memory. Use :

```c
// send count items to dest from sendbuf and receive from source into recvbuf
int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
     int dest, int sendtag, void *recvbuf, int recvcount,
     MPI_Datatype recvtype, int source, int recvtag,
     MPI_Comm comm, MPI_Status *status);

// send count items to dest from buf, then receive from source into buf
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype,
     int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
     MPI_Status *status);
```

Solution :

```
void homemade_Alltoall(int *buf, int n, MPI_Comm comm)
{
        int rank, size;
        MPI_Comm_size(comm, &size);
        MPI_Comm_rank(comm, &rank);

        for (int t = 0; t < size; t++) {
                int peer = (t - rank + size) % size;
                MPI_Sendrecv_replace(buf + peer*n, n, MPI_INT,
                       peer, 0, peer, 0, comm, MPI_STATUS_IGNORE);
        }
}
```

5. One of the hardest case is that of small messages. Assume that $n = 2^k$ and that each rank sends a single data item to each other rank. There is a protocol that performs the all-to-all shuffle in k communications rounds, using the "recursive doubling" technique.

   Draw a picture (or a series of pictures) to illustrate the protocol with $n = 4$ or $n = 8$. They must explain who sends what to whom, and when.
6. What is the total running time ?

Solution :

Each node sends half of its data in each phase, so the total running time is $\log_2 p\alpha + \frac{n}{2}\log_2 p\beta$. The latency term is optimal, but the bandwidth term is larger than the lower-bound (and the previous algorithm).

7. A network may be bisected into two equal-sized partitions. The *bisection bandwidth* of a network topology is the minimum bandwidth available between any two such partitions. In other term, the network is cut into two partitions of the same size by removing the smallest possible number of links ; what is the aggregated bandwidth of the removed links ?

   Provide a lower-bound on the time taken to complete `MPI_Alltoall` in terms of the bisection bandwidth of the network. (hint : How much data has to cross the cut ?)

   WARNING : the bandwidth can be measured in items/s or in s/items.

Solution :

Cut the network in two equal-sized partitions. Each of the $p/2$ nodes on the left must send a slice of size n/p to each one of the $p/2$ node of the other partition. So the total volume of data that must be sent from one partition to the other is $np/4$.

If the bisection bandwidth is γ s/item, then the time needed to complete the operation is at least $\gamma np/4$.

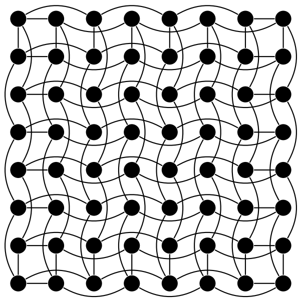
8. What is the bisection bandwidth of a 1D torus (a ring) ?

$$\begin{tikzcd}
P_0 \arrow[r, leftrightarrow] \arrow[rrr, leftrightarrow, bend left] & P_1 \arrow[r, leftrightarrow] & P_2 \arrow[r, leftrightarrow, dashed] & P_{n-1}
\end{tikzcd}$$

Solution :

It is sufficient (and necessary) to cut two network links to bisect the network, so the bisection bandwidth is $\gamma = \beta/2$. It follows that `MPI_Alltoall` requires time at least $\beta np/8$ on a 1D-torus.

9. What is the bisection bandwidth of an $\sqrt{n} \times \sqrt{n}$ 2D torus ?



Solution :

It is necessary to cut a full column, that has $2\sqrt{n}$ links. The bisection bandwidth is therefore $\gamma = \beta/2\sqrt{n}$.

10. Describe (in plain language) a *really* simple algorithm to perform `MPI_Alltoall` on the 1D Torus. Focus on describing who sends what to whom, and when. Draw a picture with $n = 4$ to illustrate it. What is the performance of the algorithm ?

Solution :

All processes send data to their successor and receive from their predecessor.

There is a *really* simple algorithm that does $p - 1$ steps where : in step $t = 0$, all processes send *all* their data to their successor (and receive the data from their predecessor). Then, in steps $1 \leq t < p - 1$, each process stores the part that corresponds to its rank and sends the data it has received in the previous step.

Each step takes time $\alpha + n\beta$ (where n is the total size of the array), so the total time needed is $(p - 1)\alpha + (p - 1)n\beta$.

It is possible to improve the bandwidth term a little, as follows :
- At time $t = 0$, the processes do nothing (!).
- At time $1 \leq t \leq p - 1$, the process of rank i sends the $t - 1$ slices received at time $t - 1$, followed by its own input slice for the process of rank $i - t$ (modulo p). So it sends a message of size tn/p. Note that at the end of the step, t slices of the input array have been sent and may be recycled to receive stuff.

At time $t = p - 1$, all the processes receive the data they need (all the input slices from all the other processes).

At time t, all the processes send tn/p items. So the total transmission time is

$$\sum_{t=1}^{p-1} \left(\alpha + t\frac{n}{p}\beta\right) = (p-1)\alpha + \frac{p-1}{2}n\beta$$

[recall that $1 + \cdots + p - 1 = p(p - 1)/2$]. The “bandwidth term” is twice better than beffore. It is 4 times worse than the lower-bound (but the lower-bound may not be tight). However, each process uses only half of the available outgoing bandwidth (they could also send to their predecessor...).

The latency term is worse than the logarithmic lower-bound, but this is somewhat unavoidable on such a restricted network topology.

11. Write a C function that implements the function. To simplify memory management, assume that the data items that are transferred are integers. You may allocate extra memory if necessary.

Solution :

This is the really simple algorithm.

```c
void Alltoall_ring(const int *sendbuf, int *recvbuf, int n, MPI_Comm comm)
{
        int rank, size;
        MPI_Comm_size(comm, &size);
        MPI_Comm_rank(comm, &rank);
        int succ = (rank + 1) % size;           // successor
        int prec = (rank - 1 + size) % size;    // predecessor

        for (int j = 0; j < n; j++)             // copy my own data
                recvbuf[rank * n + j] = sendbuf[rank * n + j];

        int *scratch = malloc(size * n * sizeof(int));
        for (int i = 1; i < size; i++) {
                if (i == 1)
                MPI_Sendrecv(sendbuf, n*size, MPI_INT, succ, 0, scratch,
                                n*size, MPI_INT, prec, 0, comm, MPI_STATUS_IGNORE);
                else
                        MPI_Sendrecv_replace(scratch, n*size, MPI_INT, succ, 0,
                                                prec, 0, comm, MPI_STATUS_IGNORE);
                // we have received the data from process incoming
                int incoming = (rank - i + size) % size;
                for (int j = 0; j < n; j++)       // copy it
                        recvbuf[incoming * n + j] = scratch[rank * n + j];
        }
        free(scratch);
}
```

12. Describe (in plain language) a simple algorithm to perform `MPI_Alltoall` on the 2D Torus. Focus on describing who sends what to whom, and when. What is the performance of the algorithm ?

Solution :

Suppose that processes are laid out on a $p \times q$ grid, and numbered in row-major order.

In the sendbuffer of each process, the slices $[ip : (i+1)p]$ must end up in the i-th row. Therefore, all the process perform a 1D All-to-all shuffle on their own column. Once this is done, each process owns all the slices of its own column that must be dispatched on its own row. Then all processes do an 1D all-to-all shuffle on their own row to finish the job.

This takes time $(p + q - 2)\alpha + n(p + q - 2)/2\beta$. If $p \approx q$, then this is $\approx 2\sqrt{pq}\alpha + n\sqrt{pq}\beta$.

The latency term is bad, but the bandwidth term is optimal up to a constant factor.

Exercice 2 – Local search in a large graph

This is a ("real-life") combinatorial optimization problem. Consider a very large undirected graph $G = (V, E)$. Nodes have a *score*, and the goal is to find a node with a "high" score. Provably finding the nodle with the highest score is impossible, because it would require exploring all the graph, and this is impossible because it is too large. Instead, heuristic techniques (greedy algorithms...) yield an "okay" node used as the starting point of the local search. The underlying idea is that good nodes are usually near other good nodes. Then the *neighborhood* of the starting point is searched exhaustively by a breadth-first-search. The procedure returns a *local maximum*, i.e. a node that is strictly better than all its neighbors.

The C code of a sequential function that does this is in the auxiliary document. Here are a few quick comments. Nodes are represented as 64-bit integers (`u64`). The graph is represented implicitly : there is a function that returns the neighbors of a given node. This function is fast. There is also a function that returns the score of a node. This function is slooooow.

The code implements a breadth-first search ("parcours en largeur"). Nodes are popped from the bottom of a queue, and their neighbors are added to the top of the queue (this way, nodes are explored by increasing distance from the starting point).

Discovered nodes are added to the queue if they are "new" (not already added to the queue) and if their score is good enough. A simple hash table is used to detect nodes already added to the queue. It is very fast but it has false negatives (it answers "no" when a node has, in fact, already been added to the queue). When it says "yes", the result is certain. This may result in some nodes being added to the queue several times (it is not good for performance but it does not affect the result).

The goal is to parallelize this code using OpenMP.

1. Where is most likely the bottleneck ?

   Solution :

   The inner loop is the for loop. In there, each individual statement is (almost) an "elementary operation", except... the call to `node_score(node)`, which is said to be slow.

2. While it may not be completely impossible, parallelizing the outer `while` loop of line 41 leads to serious complications. Explain what could potentially go wrong if several iterations of the outer while loop were executed in parallel ? (several answers are possible)

   Solution :

   Potential problems :
   - Iteration $i+1$ pops the node that was pushed by iteration i (this happen when the queue contains a single element at the begining of iteration i). So there is an unavoidable data dependency.
   - The `realloc(queue, capacity)` of line 54 conflicts with all accesses to the queue, because it may modify the memory location of the queue (accesses to the previous adress will likely segfault).
   - Nodes may no longer be processed by increasing distance from the starting point. This is not really a problem of correctness, but it may alter the behavior of the function.

3. A less ambitious, but more practical strategy consists in parallelizing the inner `for` loop of line 58. Is there a problem of load balancing ? How to do this efficiently ?

   Solution :

   Iterations of the loop may stop early because of the `continue` statement on line 63, even before calling the expensive `node_score` function. So, load balancing issues are to be expected. So either use :

   ```
   #pragma omp parallel for schedule(dynamic)
   for (int i = 0; i < k; i++) {
   ```

   or use tasks (more complicated, unclear gain).

4. What are the potentially conflicting memory accesses inside the body of the `for` loop of line 58 ? Indicate a potential consequence if a conflict occur.

   Solution :

   Here are the potential conflicts.

   - Reading `queue[top]` on line 78 VS incrementing `top` on line 79 or zeroing it on line 75 (the writes may conflict with themselves).
   - Reading `T[h]` on line 62 and writing it on line 80 (idem).
   - Reading `best_score` one lines 66 and 69, modifying it on line 70 (idem).

   Potential problem with `best_score` : two neighbors u_1 and u_2 have a better score than the current best, with u_2 the best one of the two. Both threads enter the body of the "if" statement on line 70. The score of u_2 is written first, and the score of u_1 is written last. Conclusion : the node with the best score is lost.

   Potential problem with `top` : two threads write `queue[top]` on line 78 (one overwrites the other), then both increment `top`. Conclusion : one node is lost, and the queue contains one slot of garbage.

Other potential problem with `top` : one thread sets `top` to zero on line 75, then another increment `top` on line 79. Conclusion : `queue[0]` is garbage.

The potential problem with `T[h]` is interesting, because the consequence is clearly acceptable : the worst that can happen is that a node is added to the queue when it shouldn't have. But this is not going to make the program crash or return a wrong result.

5. Propose a way to avoid *all* conflicts while evaluating `node_score` in parallel ? Write the corresponding C code. Is there any inconvenient ?

   Hint : proceed in two phases.

Solution :

```c
// phase 1: compute all the scores (of unseen nodes) in parallel

u64 scores[max_neighbors];

#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < n_neighbors; i++) {
    int node = neighbors[i];

    u64 h = murmur64(node) % hsize;
    if (T[h] == node)
        scores[i] = 0;     // skip node already in the queue
    else
        scores[i] = node_score(node);
}

// phase 2: push unseen neighbors to the top of the queue
for (int i = 0; i < n_neighbors; i++) {
     int node = neighbors[i];
     int score = scores[i];
     if (score < best_score)
         continue;         // skip the node (suboptimal)

     if (score > best_score) {
                best_score = score;
                best_node = node;
                printf("New best score: %d (node=%ld)\n", score, node);
                // flush the queue
                bot = 0;
                top = 0;
     }

     queue[top] = node;            // add node to the queue
     top += 1;
     T[h] = node;             // add node to the hash table
}
```

The inconvenient is that there is a sequential portion (the second for loop).

6. Write a parallelized version of the `for` loop of lines 58–81.