Exam — Parallel Programming

1st session — 10 January 2024 — 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 – Network Topology of the Jean-Zay Computer

The Jean-Zay computer (the flagship machine of the National Center for Scientific Research) uses a bizarre 100Gbit/s OmniPath network. There are 256 non-blocking 48-port switches. This makes 12288 network links in total. The switches are arranged in a 8-dimensional hypercube :

- Each switch is connected to 8 other switches.
- Switch number i is connected to number j if the representations of i and j in binary differ in a single location. For instance, switch 42 = `0b00101010` is connected to :

42	00101010
43	0010101**1**
40	0010100**0**
46	00101**1**10
34	0010**0**010
58	001**1**1010
10	00**0**01010
106	0**1**101010
170	**1**0101010

When switch i has to forward a message to switch j, it follows a simple algorithm :

- If $i = j$, then routing is complete
- Otherwise, write i in binary as $a_7a_6a_5a_4a_3a_2a_1a_0$ and j as $b_7b_6b_5b_4b_3b_2b_1b_0$
- Let k be the smallest integer such that $a_k \neq b_k$ ($i \neq j$ therefore k is well-defined).
- Forward the message to switch $a_7a_6 \dots (1 - a_k) \dots a_1a_0$ (only the k-th bit is changed)
- Each step reduces the distance by one bit

Each switch-to-switch connection uses 4 parallel links for an aggregated bandwidth of 400Gbit/s. Therefore, each switch has 32 links connected to other switches plus 16 links connected to 16 compute nodes. Up to 4096 compute nodes can be connected in this way. We assume that sending a message of n bytes accross a network link takes time $\alpha + n\beta$. Routing through a switch incurs an extra latency of α seconds. The switches are *fair* : if there is some contention accross a network link, then the available bandwidth is shared equitably between the competing data flows. The switches are capable of *broadcasting* : in addition to routing an an incomming data packet to the correct link, they can also forward it to any subset of their network links at the same time. All questions below are relative to this network topology.

1. Is the network blocking or non-blocking ? (assume that all 4096 nodes, except two, are busy sending and receiving data. Can the two remaining ones communicate at full speed in all cases ?)

> **Solution :**
> Blocking. If 4 nodes connected to switch #0 communicate at full speed with 4 other nodes connected to switch #1, then they saturate the 4 links between #0 and #1. If a fifth node on #0 try to send a message to a fifth node connected to #1, then it will observe the contention.

2. Consider the time taken by an `MPI_Send(...)` operation with a message of size n bytes. Does it depend on the ranks of the sender and the receiver ? (assume that all the other nodes are idle)

> **Solution :**
> If i and j are connected to the same switch, then the time needed is $3\alpha + n\beta$ (the message traverses 2 links + 1 switch). In the worst case, the message will be routed through 8 switches (k advances by one in each iteration of the routing protocol). So, *if there is no contention in the network*, then the worst-case message transmission time is $10\alpha + n\beta$.

3. Provide a worst-case upper-bound on the time taken by `MPI_Send(...)`.

Solution :
In the worst case, 8 switches have to be crossed. So the latency term is upper-bounded by 10α.
The problem is that each traversed switch may experience some contention on the four parallel network links that have to be taken to move to the next one. In the worst case, when a message is routed from switch i to switch j, then the 7 other switches connected to i send traffic directed toward j, and 16 other nodes connected to i try to do the same. So, there are, in the worst case, $16 + 7 * 4 = 44$ parallel 100Gbit/s data flows that have to fit into only 4 links. If this is done fairly, then each data flow gets only 1/11 of the full bandwidth of a single link.
So the absolute worst case upper-bound is $10\alpha + 11\beta n$.

4. Is there a natural *lower-bound* for the cost of collective operations ?

Solution :
In most collective operations, a single node must send or receive a full message of size n, or eventually $n(p-1)/p$ items. The time required to exchange this much data with the switch it is connected to is at least $\alpha + \beta n(p-1)/p$. Then this node must communicate with the farthest node from its point of view, which is 10 links away. So a natural lower-bound is :

$$10\alpha + \beta n(p-1)/p$$

(or $10\alpha + n$ for broadcast).

5. Propose a simple algorithm to do the `MPI_Bcast(...)` operation with a short message. (assume that the switches cooperate to this operation). What is its running time ? Hint : start with only 4 or 8 switches. Then you will find an algorithm that works for 2^k switches for any k.

Solution :
There may be several possibilities. Here is a simple idea : the root (process 0, say) sends the message to its switch. Then, each time the message reaches a switch, it is forwarded 1) to all the adjacent nodes, and 2) to adjacent switches with agreater index. This is "fun" and terminates in 8 steps (switch 255 is reached last).
Here is a more standard idea. Suppose that the broadcast originates from switch #0. At time step i, all switches that have received the message forward it to their neighbor in the i-th dimension. At $t = 0$, #0 sends the #1. At $t = 1$, #0 sends to #2 and #1 sends to #3, etc. After step i, 2^i nodes have received the message, so 8 steps are necessary in total.

6. How to to do `MPI_Bcast(...)` for long messages efficiently ?

Solution :
Split the long message in small chunks and run the broadcast algorithm for small messages repeatedly, in a pipleline.

7. Write the simplest possible piece of C code, using only `MPI_Sendrecv(...)`, that implements the `MPI_Alltoall(...)` operation. Hint : send to `rank` $+ i$ and receive from `rank` $- i$.

Solution :
Here is an equivalent of the function :

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

For simplicity, it is instanciated with arrays of double-precision floating point numbers. Each process sends n items to each other process.

```c
void handmade_Alltoall(const double *sendbuf, double *recvbuf, int n, MPI_Comm comm)
{
      int rank, size;
      MPI_Comm_rank(comm, &rank);
      MPI_Comm_size(comm, &size);
      for (int i = 1; i < size; i++) {
            int next = (rank + i) % size;
            int prev = (rank + size - i) % size;
            MPI_Sendrecv(&sendbuf[next*n], n, MPI_DOUBLE, next, 0,
                         &recvbuf[prev*n], n, MPI_DOUBLE, prev, 0,
                         comm, MPI_STATUS_IGNORE);
          }
}
```

8. What can you say about its running time ?

Solution :

Let N denote the total length of the array owned by each process and p the number of processes, so that each process sends N/p items to every other one.

It is difficult to analyze the running time precisely because of the potential contention. In the worst case, we know that $p-1$ successive messages have to be sent by each process, and that each of them require $10\alpha + 11\beta(N/p)$ seconds in the worst case. This leads to $\approx 10p\alpha + 11\beta N$. The latency term is really bad, but the bandwidth term is decent (this technique is OK for long messages).

9. ($\star\star\star$) Design an algorithm that works in a logarithmic number of "phases", where in each phase all nodes exchange some data with some other node (all these exchanges should happen in parallel of course). What is the running time ?

Solution :

Main idea : in phase k, exchange 50% with your neighbor in dimension k ("à la recursive doubling"). The problem is to find out what to exchange. After tring with small example, one finds that the process of coordinate zero sends/receive chunks of size 2^k with a stride of 2^{k+1} starting at offset 2^k.

Exercice 2 – Vectorization and Low-Level Stuff

We consider the problem of implementing the following simple operation :

$$d \leftarrow d \oplus e$$
$$c \leftarrow c \oplus d$$
$$b \leftarrow b \oplus c$$
$$a \leftarrow a \oplus b$$

where $\oplus$ denotes the XOR operation and a, b, c, d and e are long bit strings (of size 512Kbit for instance). The author was recently facing this problem. Concretely, a, b, c, d and e are stored in memory in arrays of n 32-bit words, so that $n \approx 16\ 000$. We can assume that the memory addresses of these arrays are correctly aligned. We can also assume that n is a multiple of 8.

Consider the following two ways to implement the operation :

```
/* option A */
for (int i = 0; i < n; i++) {
  d[i] ^= e[i];
  c[i] ^= d[i];
  b[i] ^= c[i];
  a[i] ^= b[i];
}
```

```
/* option B */
for (int i = 0; i < n; i++)
  d[i] ^= e[i];
for (int i = 0; i < n; i++)
  c[i] ^= d[i];
for (int i = 0; i < n; i++)
  b[i] ^= c[i];
for (int i = 0; i < n; i++)
  a[i] ^= b[i];
```

1. What is the operational intensity of both versions ?

   Solution :
   Option A : each iteration loads five 32-bit words and does four arithmetic operation. So $OI = 4/(5 \times 4) = 1/5$.
   Option B : each iteration loads two 32-bit words and does one arithmetic operation. So $OI = 1/(2 \times 4) = 1/8$.

2. Among these two ways of implementing the operation, which one is preferable (and why) ? Give several arguments if possible.

   Solution :
   Option A is preferable because of its higher operational intensity (it is "less memory bound"). Also, the loop counter is incremented/compared only n times in total, VS. $4n$ times in option B.

3. Is the operation most likely memory-bound or CPU-bound ?

   Solution :
   The operational intensity is low (constant as n increases), so the operation is most likely memory-bound.

4. Write a vectorized loop using "intrinsics". Hint : looking in the "intrisics guide" reveals :

```
__m512i _mm512_load_si512 (void const* mem_addr);
__m512i _mm512_xor_si512(__m512i a, __m512i b)
void _mm512_store_si512 (void* mem_addr, __m512i a);
```

   Solution :

```
for (int i = 0; i < n; i += 16) {
__m512i ve = _mm512_load_si512(&e[i]);
__m512i vd = _mm512_load_si512(&d[i]);
__m512i vc = _mm512_load_si512(&c[i]);
__m512i vb = _mm512_load_si512(&b[i]);
__m512i va = _mm512_load_si512(&a[i]);
vd = _mm512_xor_si512(vd, ve);
vc = _mm512_xor_si512(vc, vd);
vb = _mm512_xor_si512(vb, vc);
va = _mm512_xor_si512(va, vb);
_mm512_store_si512(&d[i], vd);
_mm512_store_si512(&c[i], vc);
_mm512_store_si512(&b[i], vb);
_mm512_store_si512(&a[i], va);
}
```

From now on, suppose that this is executed on a single core of an Intel Xeon Gold 6248 CPU (as found in the Jean-Zay computer). It has AVX512. A single core can execute several SIMD instructions in parallel during each clock cycle. The number of SIMD instructions that can be executed depends on their nature and on the size of their operands. Here is a brief summary (source `https://uops.info`) :

operation	AVX2 (256-bit vectors)	AVX-512 (512-bit vectors)
register $\leftarrow$ register $\oplus$ register	3	2
register $\leftarrow$ memory	2	2
memory $\leftarrow$ register	1	1

In theory, a core can do all of these in a single clock cycle.

5. Assuming that the maximum possible amount of instruction-level parallelism takes place, give a lower-bound on the number of CPU cycles required to execute the operation, using AVX2 and using AVX512. This is the *CPU lower-bound*.

Solution :

All the results are given using the "option A" loop.

Using AVX2, we can manipulate eight 32-bit words simultaneously. There are $5n/8$ (256-bit) loads, $4n/8$ (256-bit) XORs, and $4n/8$ (256-bit) stores. In a single CPU cycle, 3 XORs + 2 loads + 1 store can take place. The loads need at least $5n/16 \approx 0.31n$ cycles, the XORs need at least $4n/24 \approx 0.17n$ cycles, and the stores need $0.5n$ cycles. The stores impose the highest lower-bound.

Using AVX512, we can process twice more items per instruction, but we can only do *two* XORs per cycle. The loads require $5n/32 \approx 0.16n$ cycles, the XORs $n/8 = 0.125n$ cycles and the stores $n/4$ cycles. The stores impose again the highest lower-bound.

6. In such a CPU, each core has a 32KB L1 cache and a 1MB L2 cache. In each clock cycle, at most a single L1 $\leftrightarrow$ L2 transfer can take place. It can be either L1 $\rightarrow$ L2 or L1 $\leftarrow$ L2 but not both. A tranfer is at most 512 bits. Knowing this, give a lower-bound on the number of CPU cycles needed to perform the operation. This is the *L1 $\leftrightarrow$ L2 lower-bound*.

Solution :

Each array is larger than the L1 cache. Therefore, whatever happens, everything that is read or written has to be transferred between the L1 and the L2.

$5n$ (32-bit) words are read in total and $4n$ are written, which makes a total of $288n$ bits. This results in a total of (at least) $288n/512 = 0.5625n$ (512-bit) transfers in total, hence this many CPU cycles. Note that this is *more that twice as much* as the CPU lower-bound (this confirms that the operation is memory-bound).

7. The CPU has a 27.5MB L3 cache shared between 20 cores. In each clock cycle, at most a single L2 $\leftrightarrow$ L3 transfer can take place, and a transfer is at most 256 bits. Assume that a and b are already

in L2 cache, but that c, d and e are not. Also assume that c and d have to be written back to the L3 cache at the end. Knowing this, give a lower-bound on the number of CPU cycles needed to perform the operation. This is the *L2 ↔ L3 lower-bound*.

> **Solution :**
> Same reasoning as before : $5n$ (32-bit) words must be transferred to/from the L3 cache, so this makes $160n$ bits. This requires at least $0.625n$ transfers, hence at least this many cycles. This is even worse than the L1 ↔ L2 lower-bound.

8. In the end, is there any advantage to using AVX512 instead of AVX2 ?

> **Solution :**
> There is absolutely no advantage (this was verified experimentally !). The reason is that the time needed to transfer the data from the caches to the CPU dominate the time needed by the CPU to process them using AVX2. Using AVX512, the CPU can process faster... and this is completely useless because the data cannot arrive fast enough.

Exercice 3 – Searching a Needle in a Haystack with OpenMP

We consider a sequence of items $x_0, x_1, x_2, \ldots, x_n$ that all belong to some set A and we are given a predicate $P : A \to \{0, 1\}$. The problem consists in finding the smallest i such that $P(x_i) = 1$, if it exists. The length of the sequence is not known in advance. It could be zero ($n = -1$), or it could be infinite ($n = +\infty$) ! This is also a variant of a problem that the author was facing recently.

The sequence $(x_i)_{i \geq 0}$ is represented by an *iterator* composed of the four functions described below. These function use an opaque object `iterator *it` to represent their internal state, and in particular the index of the current element of the sequence. Elements of the sequence are represented by values of type `element *`.

```
/* Return a fresh iterator at the beginning of the sequence */
iterator * sequence_prepare();

/* Return the current element of the sequence, or NULL if the iterator is beyond the end */
element * sequence_fetch(const iterator *it);

/* Return true if the iterator is beyond the end of the sequence */
bool sequence_finished(const iterator *it);

/* Advance the iterator to the next element of the sequence */
void sequence_advance(iterator *it);
```

The procedure that searches the “good” element of the sequence is the following :

```
/* Return the smallest i such that P(x[i]), or -1 if no such i exists */
long search()
{
        iterator *it = sequence_prepare();
        long i = 0;
        while (!sequence_finished(it)) {
                element *x = sequence_fetch(it);
                if (P(x))
                        return i;
                i += 1;
                sequence_advance(it);
        }
        return -1;
}
```

The problem is to do this in parallel on a multicore machine.

1. Write the C code of an efficient parallel version of `search()` using OpenMP.
   Remember : the compiler will reject a program in which a `return` statement leaves a parallel section. It is also forbidden to use `break` inside loops parallelized using `#pragma omp for`.

Solution :

There are several possible solutions. The main mistake to avoid was calling `sequence_advance(it);` in parallel, because this causes a potentially conflictual access to `it` (what is the result if two threads do it simultaneously ? There is no guarantee that the operation is thread-safe).

Here is a first solution :

```
/* Option A */
long search()
{
        iterator *local_it = sequence_prepare();
        long i = tid;
        long winner = -1;
        #pragma omp parallel
        {
                while (!sequence_finished(it)) {
                        element *local_x;
                        long local_i;
                        #pragma omp critical   // avoid conflicting access to ``it''
                        {
                                local_i = i;   // save (i, x[i]) locally
                                local_x = sequence_fetch(it);
                                sequence_advance(it);
                                i += 1;
                        }
                        // test x[i]
                        if (P(local_x)) {
                                #pragma omp critical
                                {
                                        if (winner < 0 || local_i < winner)
                                                winner = local_i;
                                }
                        }
                }
        }
        return winner;
}
```

Here is another more complicated way to do the same thing, without critical section around "advance" :

```
/* Option B */
long search()
{
        iterator *it = sequence_prepare();
        long i = 0;
        long winner = -1;
        int T = omp_get_max_threads();
        element *A[T];
        #pragma omp parallel
        {
          int T = omp_get_num_threads();
          int tid = omt_get_thread_num();
          while (!sequence_finished(it)) {
                // phase I: a single thread computes the next T elements of the sequence
                #pragma omp single
                for (int j = 0; j < T; j++) {
                        if (sequence_finished(it)) {
                                A[j] = NULL;
                        } else {
                                A[j] = sequence_fetch(it);
                                sequence_advance(it);
                        }
                }
                // phase II: test the T elements in parallel
                if ((A[tid] != NULL) && P(A[tid])) {
                        #pragma omp critical
                        {
                                if (winner < 0 || i + tid < winner)
                                        winner = i + tid;
                        }
                }
                if (winner >= 0)
                        break;
                #pragma omp single
                i += T;
        }
        return winner;
}
```

Here is another different solution :

```
/* Option C */
long search()
{
        long winner = -1;
        #pragma omp parallel
        {
                int T = omp_get_num_threads();
                int tid = omt_get_thread_num();
                // thread tid looks at items of index tid + k*T, i.e. at x[tid::T]
                iterator *local_it = sequence_prepare();
                long i = tid;
                // advance local iterator by tid steps
                for (int j = 0; j < tid; j++)
                        if (!sequence_finished(local_it))
                                sequence_advance(local_it);

                int local_winner = -1;
                while (local_winner < 0 && !sequence_finished(local_it)) {
                        element *x = sequence_fetch(local_it);
                        if (P(x)) {
                                #pragma omp critical
                                {
                                        if (winner < 0 || i < winner)
                                                winner = i;
                                }
                        }
                        // advance local iterator by T steps
                        for (int j = 0; j < T; j++)
                                if (!sequence_finished(local_it))
                                        sequence_advance(local_it);
                        // refresh local winner -- make sure the thread local
                        // private view of the memory (of ``winner'') is updated
                        // omp flush would be sufficient, but this is simpler
                        #pragma omp atomic read
                        local_winner = winner
                }
        }
        return winner;
}
```

Option A/B only calls P in parallel, while option C does more stuff in parallel... and also more operations in total.

Let n denote the number of iterations of the sequential program. Let α denote the time taken by "finished + advance", β the time needed by "fetch" and γ the time needed by "P". The sequential program takes time $T_{seq} = n(\alpha + \beta + \gamma)$.

Using p threads, in both options, each iteration of the "while" loop processes a batch of p iterations, so there are n/p iterations in total.

In option A/B, a single iteration of the "while" loop takes time $p(\alpha + \beta) + \gamma$ (all the P's are evaluated in parallel).

In option C, a single iteration of the "while" loop takes time $p\alpha + \beta + \gamma$

Thus, option A/B requires time $T_A \approx n\alpha + n\beta + \frac{n}{p}\gamma$ (if n is large so that doing $\pm p$ iterations does not change anything).

Option C requires time $T_B \approx n\alpha + \frac{n}{p}\beta + \frac{n}{p}\gamma$, so it could be a bit faster, especially if "fetch" is slow. But it does more operations in total (and this might be bad from an energy consumption point of view). In option A/B, all threads but one are passive a fraction of the time.