Parallel Computing: Exercise session 1

Emil Løvbak & Sahar Chehrazad

19 October 2020

This exercise session consists of pen-and-paper exercises and covers the following topics:

- Ahmdal's law and limitations to parallizing software
- Communication overhead
- Common pitfalls in parallel software
- Data partitioning

For questions you can contact Emil Løvbak (`emil.loevbak@kuleuven.be`) or Sahar Chehrazad (`sahar.chehrazad@kuleuven.be`).

Question 1: Ahmdal's law

Consider the following segment of code:

```
1  int n = 10000;
2  int A[n];
3  A[0] = 1;                    // 1e-5 ms
4  A[1] = 1;                    // 1e-5 ms
5  for (int i=2; i<n; i++)
6    A[i] = A[i-1] + A[i-2];    // 1e-5 ms
7  for (int i=0; i<n; i++) {
8    A[i] += 4;                 // 1e-5 ms
9    printf("%d ", A[i]);       // 0.01 ms
10 }
11 sumA = 0;                    // 1e-6 ms
12 for (int i=0; i<n; i++)
13   sumA += A[i];              // 1e-5 ms
14 printf("%d", sumA);          // 0.01 ms
```

1. Which operations can easily be parallelized? Which cannot?
   - *3,4: Can happen concurrently, but not worth the overhead.*
   - *5,6: Non-parallelizable loop as there is a sequential data dependency.*
   - *7,8,9: If the loop is parallelized then the prints will occur out of order.*
   - *11: Sequential*
   - *12,13: This loop is parallelizable. There will be some overhead due to a reduction, but we will ignore this here.*
   - *14: Sequential*
2. What is the serial runtime of this code?

   $10^{-5}ms + 10^{-5}ms + (10000-2)10^{-5}ms + 10000(10^{-5}ms + 10^{-2}ms) + 10^{-6}ms + 10000 \times 10^{-5}ms + 0.01ms$
   $= 100.3ms$

3. What would the runtime be on 20 cores? What speedup is achieved? (Ignore parallel overhead and reduction operations)

   $10^{-5}ms + 10^{-5}ms + (10000-2)10^{-5}ms + 10000(10^{-5}ms + 10^{-2}ms) + 10^{-6}ms + \frac{10000}{20} \times 10^{-5}ms + 0.01ms$
   $= 100.2ms$

   Speedup practically 1

4. What would the runtime be on an infinite number of cores? What speedup is achieved?

   $10^{-5}ms + 10^{-5}ms + (10000-2)10^{-5}ms + 10000(10^{-5}ms + 10^{-2}ms) + 10^{-6}ms + \frac{10000}{\infty} \times 10^{-5}ms + 0.01ms$
   $= 100.2ms$

   Speedup practically 1

5. Can we modify the code to make this better?

   The print statement on line 9 is slowing everything down. Let's instead append our elements to a string using `char *strcat(char *s1, const char *s2);` *and print the whole string after the loop. Let us assume adding elements to the string is as expensive as the other computations on array elements, i.e.,* $10^{-5}ms$ *and that the print itself is equally expensive:*

   $10^{-5}ms + 10^{-5}ms + (10000-2)10^{-5}ms + 10000 \times 2 \times 10^{-5}ms + 10^{-6}ms + 10000 \times 10^{-5}ms + 2 \times 0.01ms$
   $= 0.420ms$

   The parallel runtime is:

   $10^{-5}ms + 10^{-5}ms + (10000-2)10^{-5}ms + \frac{10000}{20} \times 2 \times 10^{-5}ms + 10^{-6}ms + \frac{10000}{20} \times 10^{-5}ms + 2 \times 0.01ms$
   $= 0.135ms$

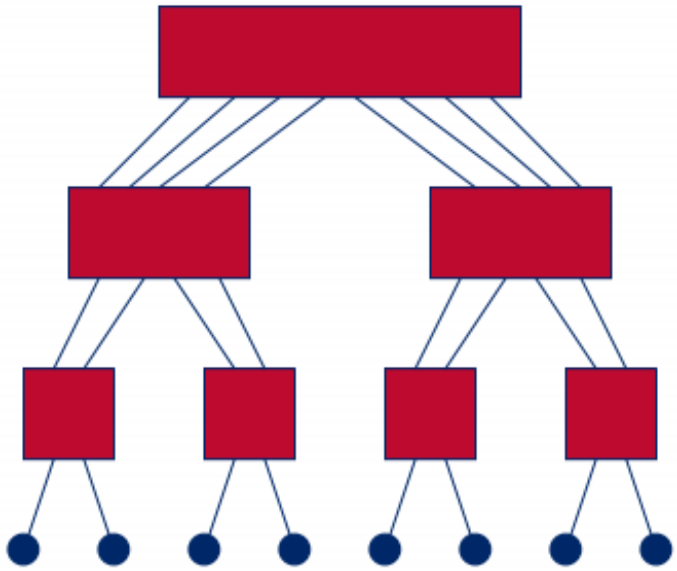
   The speedup is $\frac{420}{135} = 3.11$ *for 20 cores. With infinitely many core this becomes 3.5.*

What can we conclude?

Parallelization is a very useful tool to speed up software, but it has it's limitations. It is also important to consider serial speed up techniques. Throwing more hardware at an inherently serial algorithm will give little speedup. If sufficient hardware is available, it can make sense to use a more expensive algorithm, i.e., that does more computation, but is easier to parallelize.

Question 2: The cost of parallelization

Consider that we are running some parallel program on the following architecture, where each link has a communication latency of 1 ms and a bandwidth $B = 10\ 000$ MB/s:



The program consists of two stages of computation, separated by an all-to-all communication step, where each core communicates directly with each other core. The first computation takes each core $T_1 = 10$ ms and the second computation takes each core $T_2 = 2$ ms. The total amount of data in the computation N is 100 MB. The data is distributed over the $p = 8$ cores. Assume that a switch can send data while receiving data. Also assume that a link is unidirectional, i.e., data cannot pass from A to B while it is also passing from B to A, but once a link is open A and B can alternately send each other data without incurring extra latencies.

1. How long does the program take to run?

   With this setup, the most efficient approach is to first exchange data with each direct neighbor, then to pass data to a core connected one level up in the tree and then to pass data through the top of the tree.

   - *Computation time:* $T_{comp} = T_1 + T_2 = 12ms$
   - *Communication latency:* $T_{lat} = 6ms + 4ms + 2ms = 12ms$ *(Time needed to set up the three communication channels that each core needs to have.)*
   - *Transmission time:* $T_{trans} = 2 \times (4+2+1) \times \frac{N}{pB} = 17.5ms$ *(Bi-directional communication with 3 different partner cores)*
   - *Total runtime* $= T_{comp} + T_{lat} + T_{trans} = 12ms + 12ms + 17.5ms = 41.5ms.$

2. How long would a serial program take to run on the same hardware? (Ignore caching effects.)

   $pT_{comp} = 96ms$

3. What percentage of the parallel runtime is due to communication?

$$\frac{T_{lat} + T_{trans}}{T_{comp} + T_{lat} + T_{trans}} = \frac{12ms + 17.5ms}{12ms + 12ms + 17.5ms} = 71\%$$

4. What happens if the total amount of data in the computation is 1000 MB, assuming both computations are $\mathcal{O}(N)$?

   - *Serial runtime:* $960ms$
   - *Parallel runtime with the same method as above:* $120ms + 12ms + 175ms = 307ms$
   - *Parallel runtime with 8 one to all broadcasts:* $120ms + 8 \times \left(6ms + \frac{1000/8MB}{10000MB/s}\right) = 268ms$

5. What percentage of the parallel runtime is then due to communication?

$$\frac{12ms + 175ms}{120ms + 12ms + 175ms} = 61\%, \quad \frac{148ms}{120ms + 148ms} = 55\%$$

6. What changes in the 100 MB case if there is only one link between each pair of switches?

   The bottleneck lies in the information that is passed through the top of the tree, as this has the largest latency and the lowest bandwith. The most efficient form of communication is for each core to sequentially do a broadcast, as it takes just as long to transmit data from one core to its furthest partner as it takes to transmit data from one core to all of its partners and only one core can send through the top of the tree at a given moment. The latency for setting up a broadcast is determined by the longest path through the tree. The total communication time becomes

$$p \times \left(6ms + \frac{N}{pB}\right) = 58ms.$$

7. **What can we conclude?**

   It is difficult a-priori to determine which network is best for the application at hand. If we have more data, then bandwidth becomes more important. If we have less data then latency becomes more important. If the computational cost of the algorithm is at least linear, the overhead due to communication decreases as the problem size grows, if we ignore caching effects and other effects not present in the theoretical model.

Question 3: Problems with loops

Consider the following code segments. What issues can you see in each case?

```
1 for (int i=0; i<n; i++)
2   for (int j=0; j<n; j++)
3     C[i][j] = 0.0;
4     #pragma omp parallel for
5     for (int k=0; k<n; k++)
6       C[i][j] += A[i][k]*B[k][j];
```

There is shared data in the matrix C. It is important to remember to do a reduction on the elements of C at the end of the parallel loop.

```
1 for (int i=0; i<n; i++)
2   #pragma omp parallel for
3   for (int j=0; j<n; j++)
4     B[i][j] = A[i][j];
```

Parallel environment set up inside nested loops causes unnecessary overhead. We can move the parallel directive out of the loop.

```
1 #pragma omp parallel for
2 for (int i=0; i<n; i++)
3   for (int j=0; j<i; j++)
4     B[i][j] = [i][j];
```

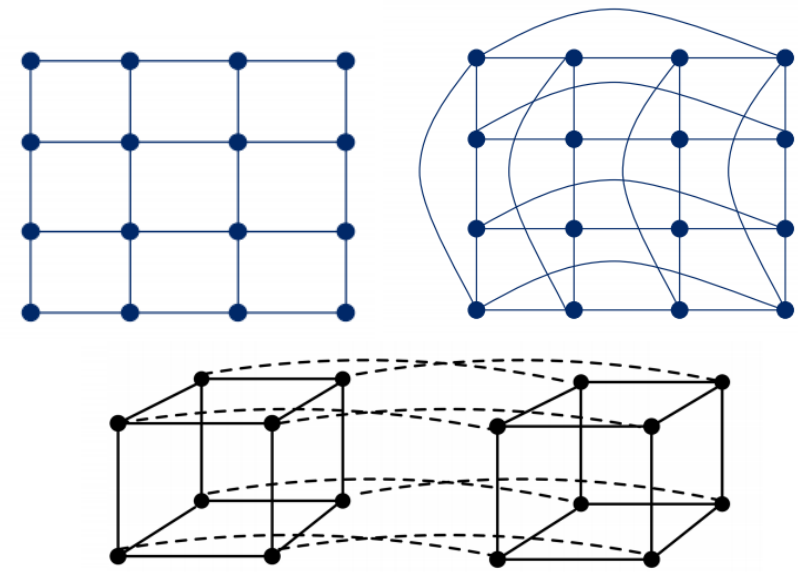
Each iteration of the outer loop contains a different amount of work. This will cause imbalanced workloads. A flexible scheduling approach can avoid this.

```
1 #pragma omp parallel for
2 for (int i=0; i<n; i++)
3   A[i] += 2.3;
```

Risk of false sharing if distribution of indexes over cores is poorly chosen.

Question 4: Bandwidth

Consider the following graphs where each link has a communication latency ℓ and a bandwidth B:



Answer the following questions for both a general number of cores and the cases given. (You can assume that the number of cores p is a "nice" number.)

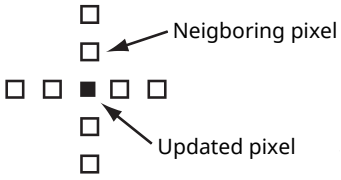
1. What are the maximal and minimal communication latencies?
   - *2D-mesh: Corner to corner:* $2(\sqrt{p}-1)\ell \to 6\ell$
   - *Circular-mesh: Maximal horizontal latency + maximal vertical latency:* $2\left(\frac{\sqrt{p}}{2}\right)\ell \to 4\ell$
   - *Hypercube: One link in each direction:* $\log_2(p)\ell \to 4\ell$
2. What are the bisection bandwidths?
   - *2D-mesh:* $\sqrt{p}B \to 4B$
   - *Circular-mesh:* $2\sqrt{p}B \to 8B$
   - *Hypercube:* $\frac{p}{2}B \to 8B$
3. Which graph in general has the highest connectivity?

   Hypercube
4. Which graph in general is the least prone to failure?

   Hypercube

Question 5: Image processing

A digital image is given as a matrix of $n \times n$ pixels with 3 numbers per pixel (R-G-B values). An iterative algorithm performs in each iteration step per pixel 50 operations ($+$, $\times$, ...) which involves the pixel values of 8 neighbours (see figure). The algorithm runs on a parallel distributed memory computer with 16 processors.



$Mflop$ rate	$166 Mflops$
startup time (t_s)	$50.00\mu s$
time per word (t_w)	$0.10\mu s$

The image size is 624×624 (i.e. n=624). How will you partition the data? Compute the speed-up S and the parallel efficiency E) if communication can be neglected, and ii) if communication cannot be neglected, taking into account the given $Mflop$ rate, startup time and communication time per word. How would you partition the data if the startup time is very large? **What can you conclude?**

- *Runtime:*
  - *Serial:* 624×624 *pixels* $\times \frac{50 ops}{166 \times 10^6 ops/s} = 117ms$
  - *Parallel:* $\frac{117ms}{16} = 7.33ms$
- *Communication time:*
  - *Columns/rows: Communication with two neighbors 3 words per pixel:*

$$2\left(3 \times 2 \times 624 \times 10\mu s + 50\mu s\right) = 75ms$$

  - *Grid: Communication with four neighbors 3 words per pixel:*

$$4\left(3 \times 2 \times \frac{624}{4} \times 10\mu s + 50\mu s\right) = 38ms$$

- *Speed-up:*
  - *Without communication:* $\frac{117ms}{7.33ms} = 16$
  - *With communication:* $\frac{117ms}{45ms} = 2.6$
- *Efficiency*
  - *Without communication:* $\frac{16}{16} = 100\%$
  - *With communication:* $\frac{2.6}{16} = 16\%$
- *As the startup time grows, the grid structure becomes less optimal until it becomes better to partition the data in columns/rows. It is important to consider how to partition data when designing parallel software.*