PDC Semester 1 2022 Examination Paper

This paper has been edited for more compact formatting, and to remove questions not covered in the 2023 course.

Question 1 [25 marks]

Part A

(In the text below ^ denotes exponentiation i.e. 10^3 is 1000 and 10^(-3) is 0.001)

Suppose a program must execute 2*10^12 instructions in order to solve a particular problem. Suppose further that a single processor system can solve the problem in 10^6 seconds. So, on average, the single processor system executes 10^6 or a million instructions per second.

Now suppose that the program has been parallelized for execution on a distributed-memory system. Suppose also that if the parallel program uses p processors, each processor will execute 10^12/p instructions and each processor must send (10^9) (p-1) messages.

Finally, suppose that there is no additional overhead in executing the parallel program. That is, the program will complete after each processor has executed all of its instructions and sent all of its messages, and there won't be any delays due to things such as waiting for messages. Note that the execution of the program and the sending of messages can't be done in parallel.

Answer the two questions below. Explain your reasoning clearly and concisely.
An approximate answer is acceptable.

Q1A1: Suppose it takes 10^(−9) seconds to send a message.

How long will it take the program to run with 2000 processors, if each processor is as fast as the single processor on which the serial program was run?
[9 marks]

Q1A2: Suppose it takes 10^(−3) seconds to send a message.

How long will it take the program to run with 2000 processors?
[6 marks]

Part B

Is a program that obtains linear speedup strongly scalable?
Recall: A program is said to be strongly scalable if when we increase the number of processes/threads, we can keep the efficiency fixed without increasing the problem size.

Explain your answer clearly and concisely.
[5 marks]

Part C

Your facility uses a parallel computer, named Tiny.

You run a test program with a problem of size 100,000. For 1 to 5 processors respectively, the times that you obtain are: 120, 60, 42, 36 and 34 minutes.

Answer the following question, using the above measurements.

Q1C1: Calculate speedups for 1,2,3,4,5 processors from the measurements above. Write the answers in the format (number of processors, speedup).
[2 marks]

Q1C2: Explain briefly what the first 5 results (from 1 to 5 processors) tell you about the performance of the program – comment on any performance issues and whether the program is suitable for running on a high-performance computer with more than 5 processors.
[3 marks]

TOTAL (Question 1) 25 marks

Question 2 [20 marks]

Consider each of the statement below.

State whether the statement is true or false, then write no more than one line of explanation for your answer. Your explanation must be concise. If you believe that the statement is false, explain why. If you think it is true, provide some useful additional information about the statement. Each statement can attract a maximum of 2 marks, 1 for true or false and 1 for explanation.

a. Busy waiting is usually implemented using a lock.
b. My parallel program has an efficiency of only 40% with 8 processors. But it still may make sense to run it on a parallel system.
c. OpenMP uses higher level programming, so deadlock is impossible when OpenMP is used.
d. Increasing problem size always increases speedup.
e. SPMD is a modern term for the concept of SIMD.
f. Numerical weather prediction is embarrassingly parallel.
g. Pipelining is an example of task parallelism.
h. The concept of barrier is important in both shared and distributed memory programming.
i. A hypercube interconnection network can be used to emulate a ring topology.
j. When using Foster's Methodology (PCAM), we try to identify as many tasks as possible, for better load balancing.

TOTAL (Question 2) 20 marks

Question 3 [25 marks]

Part A
Consider a system of three MPI tasks.

Q3A1: What is a deadlock in this context?
[2 marks]

Q3A2: Write concise pseudocode (as executed by each of the three tasks, in the usual way) illustrating a situation in which deadlock occurs.
[3 marks]

Q3A3: Explain clearly and concisely how the deadlock occurs in your pseudocode.
[3 marks]

Part B

This question is about parallel summation of a number of integers (see pages 4 to 6 of Chapter 1 of the textbook). These pages are attached as an Appendix at the end of the exam for reference.

Q3B1: The first part of the global sum example (when each core adds its assigned computed values) is usually considered to be an example of data-parallelism.

Explain concisely how this part exhibits data parallelism.
[2 marks]

Q3B2: The second part of the first global sum (when the cores send their partial sums to the master core, which adds them) could be considered to be an example of task-parallelism.

Explain concisely how this part exhibits task parallelism.
[3 marks]

Q3B3: Now consider the second part of the second global sum (when the cores use a tree structure to add their partial sums).

Is this an example of data- and/or task-parallelism?

Explain your answer, clearly and concisely.
[5 marks]

Part C

This question concerns interconnection networks in parallel computers.

Q3C1: Explain briefly and concisely how routing is performed in a multistage interconnection network (MIN).
[4 marks]

Q3C2: Explain why a MIN generally performs better than a bus.
[3 marks]

TOTAL (Question 3) 25 marks

Question 4 GPU / OpenCL

Q4A: Explain the difference between array processors and vector processors. State which type of processor a modern GPU best represents, and explain why.
[4 marks]

End of exam questions

Appendix follows.

An efficient parallel implementation of a serial program may not be obtained by finding efficient parallelizations of each of its steps. Rather, the best parallelization may be obtained by stepping back and devising an entirely new algorithm.

As an example, suppose that we need to compute n values and add them together. We know that this can be done with the following serial code:

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

Now suppose we also have p cores and p is much smaller than n. Then each core can form a partial sum of approximately n/p values:

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
   my_x = Compute_next_value(. . .);
   my_sum += my_x;
}
```

Here the prefix `my_` indicates that each core is using its own, private variables, and each core can execute this block of code independently of the other cores.

After each core completes execution of this code, its variable `my_sum` will store the sum of the values computed by its calls to `Compute_next_value`. For example, if there are eight cores, $n = 24$, and the 24 calls to `Compute_next_value` return the values

$$1,4,3,\ 9,2,8,\ 5,1,1,\ 6,2,7,\ 2,5,0,\ 4,1,8,\ 6,5,1,\ 2,3,9,$$

then the values stored in `my_sum` might be

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Here we're assuming the cores are identified by nonnegative integers in the range $0, 1, \ldots, p-1$, where p is the number of cores.

When the cores are done computing their values of `my_sum`, they can form a global sum by sending their results to a designated "master" core, which can add their results:

```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
```

```
} else {
   send my_x to the master;
}
```

In our example, if the master core is core 0, it would add the values $8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95$.

But you can probably see a better way to do this—especially if the number of cores is large. Instead of making the master core do all the work of computing the final sum, we can pair the cores so that while core 0 adds in the result of core 1, core 2 can add in the result of core 3, core 4 can add in the result of core 5 and so on. Then we can repeat the process with only the even-ranked cores: 0 adds in the result of 2, 4 adds in the result of 6, and so on. Now cores divisible by 4 repeat the process, and so on. See Figure 1.1. The circles contain the current value of each core's sum, and the lines with arrows indicate that one core is sending its sum to another core. The plus signs indicate that a core is receiving a sum from another core and adding the received sum into its own sum.

For both "global" sums, the master core (core 0) does more work than any other core, and the length of time it takes the program to complete the final sum should be the length of time it takes for the master to complete. However, with eight cores, the master will carry out seven receives and adds using the first method, while with the second method it will only carry out three. So the second method results in an improvement of more than a factor of two. The difference becomes much more

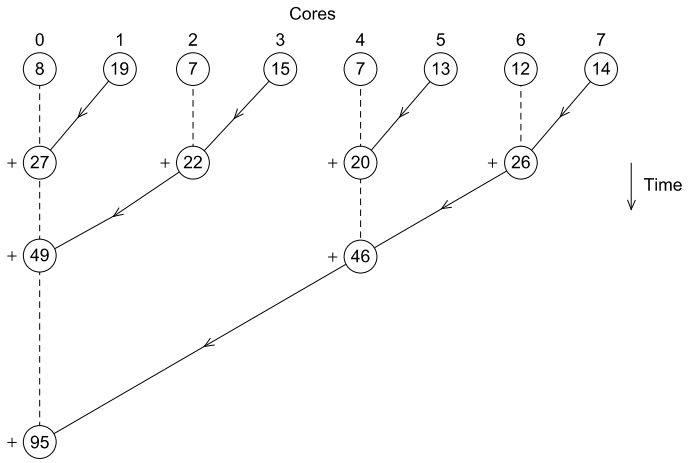

FIGURE 1.1

Multiple cores forming a global sum

dramatic with large numbers of cores. With 1000 cores, the first method will require 999 receives and adds, while the second will only require 10, an improvement of almost a factor of 100!

The first global sum is a fairly obvious generalization of the serial global sum: divide the work of adding among the cores, and after each core has computed its part of the sum, the master core simply repeats the basic serial addition—if there are p cores, then it needs to add p values. The second global sum, on the other hand, bears little relation to the original serial addition.

The point here is that it's unlikely that a translation program would "discover" the second global sum. Rather there would more likely be a predefined efficient global sum that the translation program would have access to. It could "recognize" the original serial loop and replace it with a precoded, efficient, parallel global sum.

We might expect that software could be written so that a large number of common serial constructs could be recognized and efficiently **parallelized**, that is, modified so that they can use multiple cores. However, as we apply this principle to ever more complex serial programs, it becomes more and more difficult to recognize the construct, and it becomes less and less likely that we'll have a precoded, efficient parallelization.

Thus, we cannot simply continue to write serial programs, we must write parallel programs, programs that exploit the power of multiple processors.

1.4 HOW DO WE WRITE PARALLEL PROGRAMS?

There are a number of possible answers to this question, but most of them depend on the basic idea of *partitioning* the work to be done among the cores. There are two widely used approaches: **task-parallelism** and **data-parallelism**. In task-parallelism, we partition the various tasks carried out in solving the problem among the cores. In data-parallelism, we partition the data used in solving the problem among the cores, and each core carries out more or less similar operations on its part of the data.

As an example, suppose that Prof P has to teach a section of "Survey of English Literature." Also suppose that Prof P has one hundred students in her section, so she's been assigned four teaching assistants (TAs): Mr A, Ms B, Mr C, and Ms D. At last the semester is over, and Prof P makes up a final exam that consists of five questions. In order to grade the exam, she and her TAs might consider the following two options: each of them can grade all one hundred responses to one of the questions; say P grades question 1, A grades question 2, and so on. Alternatively, they can divide the one hundred exams into five piles of twenty exams each, and each of them can grade all the papers in one of the piles; P grades the papers in the first pile, A grades the papers in the second pile, and so on.

In both approaches the "cores" are the professor and her TAs. The first approach might be considered an example of task-parallelism. There are five tasks to be carried out: grading the first question, grading the second question, and so on. Presumably, the graders will be looking for different information in question 1, which is about