COMP4510 **Assignment 1 Sample Solution**

Assignment Objectives:

- *To reinforce your understanding of some key concepts/techniques introduced in class.*
- *To introduce you to doing independent study in parallel computing.*

Assignment Questions:

[10] 1. We used a number of terms/concepts informally in class relying on intuitive explanations to understand them. Provide concrete definitions and examples of the following terms/concepts:
   a) Parallel speedup
   b) Parallel efficiency
   c) Amdahl's Law

   Do some online research and provide a concrete definition of Gustafson-Barsis's law. What do this law tell us. How does it relate to Amdahl's law?

   Parallel speedup is defined as the ratio of the time required to compute some function using a single processor (T_1) divided by the time required to compute it using P processors (T_P). That is: $speedup = T_1/T_P$. For example if it takes 10 seconds to run a program sequentially and 2 seconds to run it in parallel on some number of processors, P, then the speedup is 10/2=5 times.

   Parallel efficiency measures how much use of the parallel processors we are making. For P processors, it is defined as: $efficiency = 1/P \times speedup = 1/P \times T_1/T_P$. For example, continuing with the same example, if P is 10 processors and the speedup is 5 times, then the parallel efficiency is 5/10=.5. In other words, on average, only half of the processors were used to gain the speedup and the other half were idle.

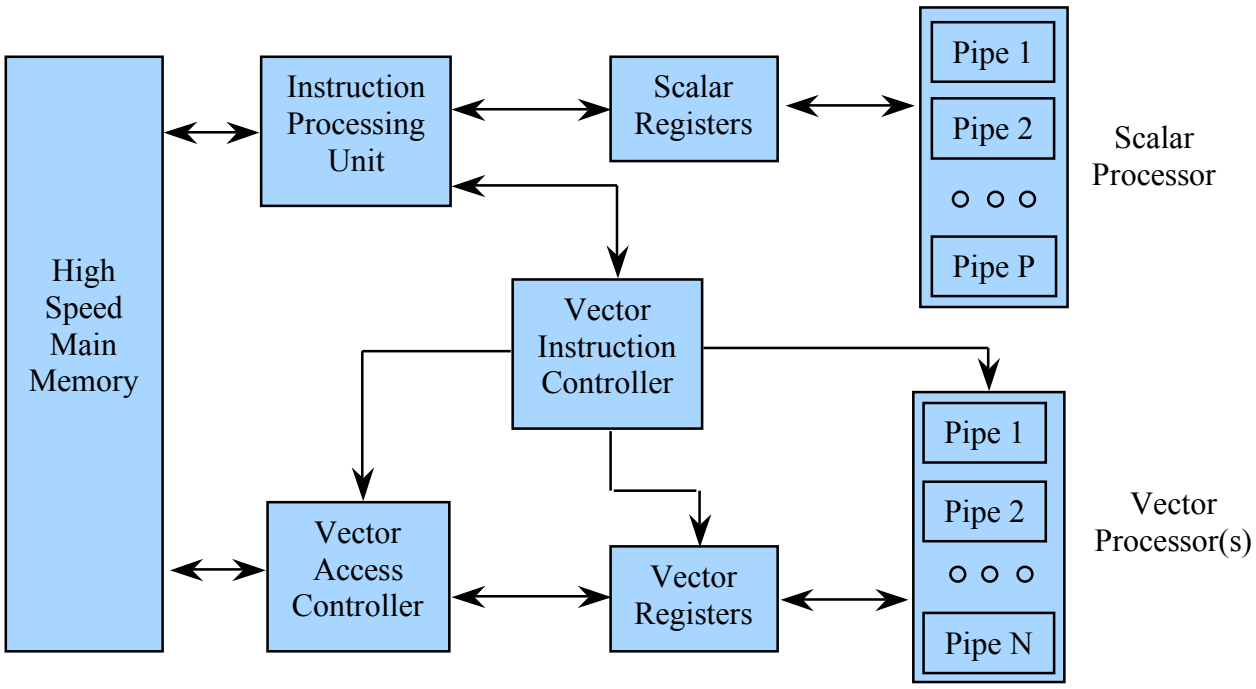
   Amdahl's law states that the maximum speedup possible in parallelizing an algorithm is limited by the sequential portion of the code. Given an algorithm which is P% parallel, Amdahl's law states that: $MaximumSpeedup=1/(1- (P/100))$. For example if 80% of a program is parallel, then the maximum speedup is 1/(1-0.8)=1/.2=5 times. If the program in question took 10 seconds to run serially, the best we could hope for in a parallel execution would be for it to take 2 seconds (10/5=2). This is because the serial 20% of the program cannot be sped up and it takes .2 x 10 seconds = 2 seconds even if the rest of the code is run perfectly in parallel on an infinite number of processors so it takes 0 seconds to execute.

   The Gustafson-Barsis law states that speedup tends to increase with problem size (since the fraction of time spent executing serial code goes down). Gustafason-Barsis' law is thus a measure of what is known as "scaled speedup" (scaled by the number of processors used on a problem) and it can be stated as: $MaximumScaledSpeedup=p+(1-p)s$, where p is the number of processors and s is the fraction of total execution time spent in serial code. This law tells us that attainable speedup is often related to problem size not just the number of processors used. In essence Amdahl's law assumes that the percentage of serial code is independent of problem size. This is not necessarily true. (E.g. consider overhead for

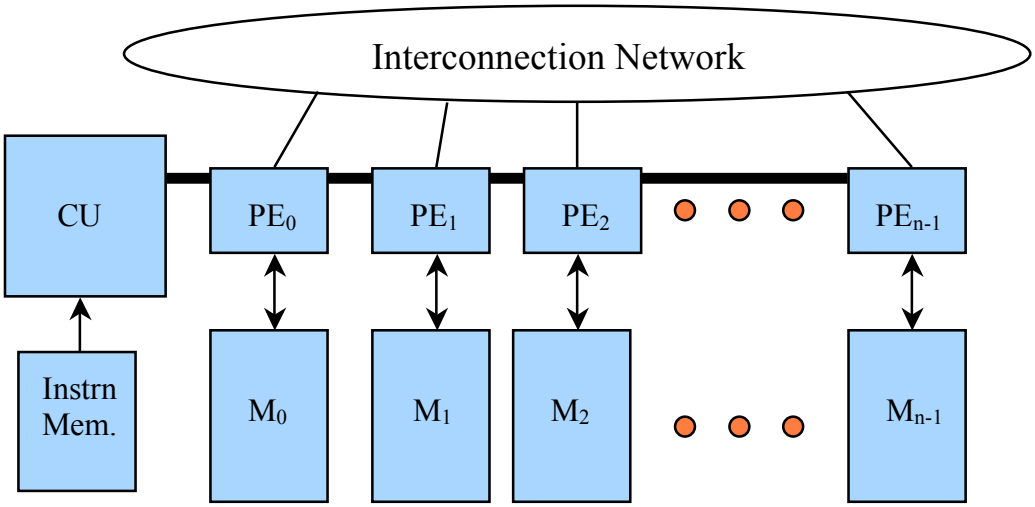
managing the parallelism: synchronization, etc.). Thus, in some sense, Gustafon-Barsis' law generalizes Amdahl's law.

[15] 2. Another class of parallel architecture is the pipelined vector processor (PVP). PVP machines consist of one or more processors each of which is tailored to perform vector operations very efficiently. An example of such a machine is the NEC SX-5. Do a little online research and then describe what a PVP is, and how, specifically, it supports fast vector operations. Another type of closely related parallel architecture is the vector processor (AP). AP machines support fast array operations. An early example of such a machine was the ILLIAC-IV. Do a little more research then describe what an AP is and how it supports fast array operations. Which architecture do you think offers more potential parallelism? Why?

A PVP is a parallel architecture where each machine consists of one or more processors designed explicitly to support vector operations. To make vector operations efficient, each processor has multiple, deep D-unit pipelines and supports a vector instruction set. The availability of the vector instructions allows a continuous flow of vector elements into the D-Unit pipelines thereby making them efficient. The high level architecture of a PVP looks like:



An array processor is a parallel architecture designed to make operations on arrays efficient. It is structurally very different from a PVP in that the processing elements (PEs), which are similar to CPU cores, are themselves structured as an array. By mapping elements of the arrays we wish to operate on across the PEs (and their corresponding local memories) we can enable parallel operation. The high level architecture of an AP looks like:



In general, APs are likely to be more scalable and therefore offer more potential parallelism. Of course, this will depend on the problem being solved being able to make use of the parallelism offered. The parallelism provided in one D-Unit pipeline in a PVP is limited by the number of stages in the pipeline. Further, there is unlikely to be sufficient ILP to keep very many D-Unit pipelines busy.

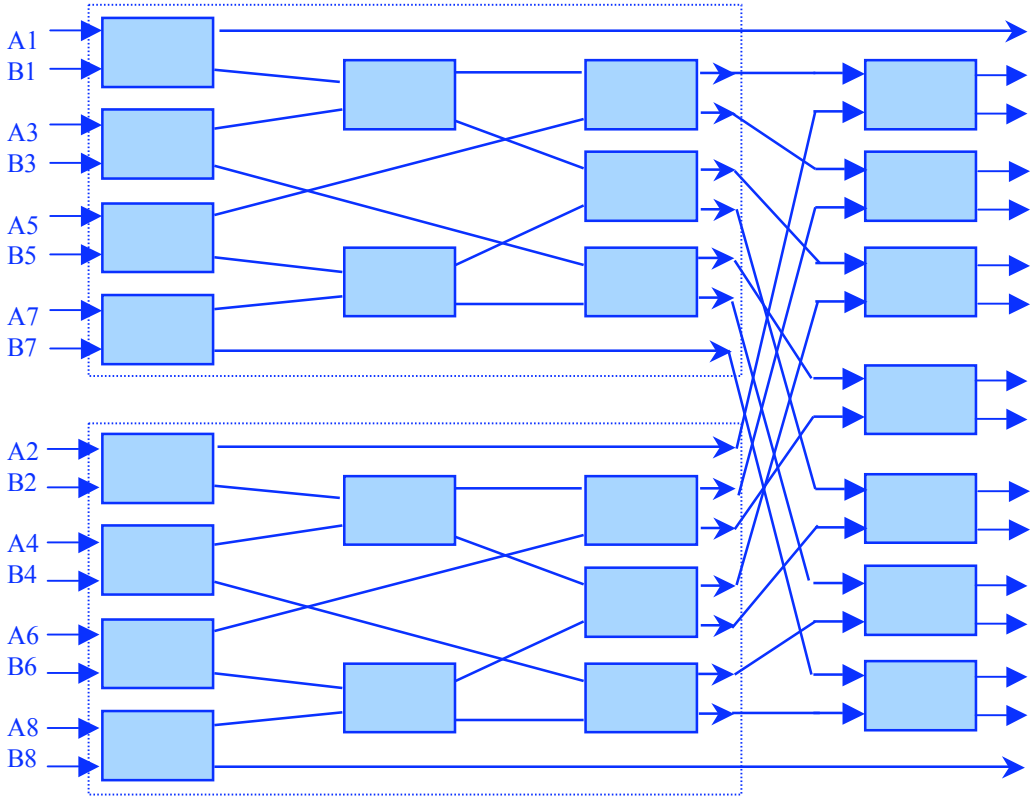
[10] 3. Design an EREW-PRAM algorithm for computing the inner product of two vectors, X and Y, of length N, (where N is a power of two). Your algorithm should run in O(logN) time with N processors. You should express your algorithm in a fashion similar to the "fan-in" algorithm done in class using PARDO and explicitly indicating which variables are shared between processes and which are local.

There are a variety of possible O(logN) EREW PRAM parallel algorithms for inner product. Here is one possible algorithm.

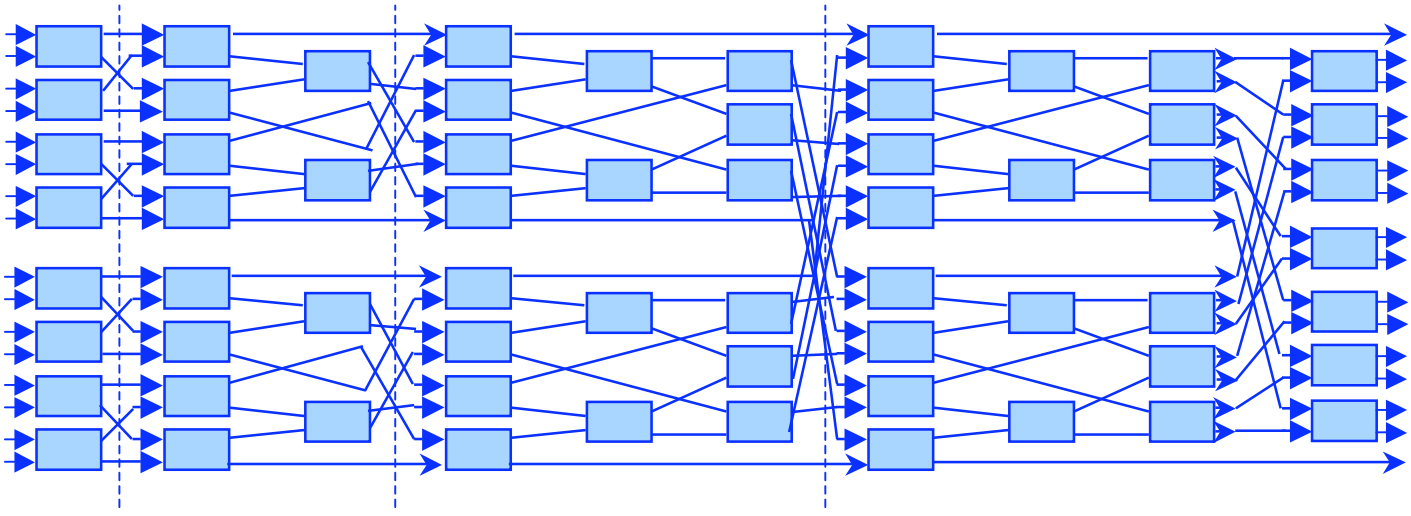
```
// Start by computing the element-wise product of X and Y
FOR i:=1 TO N PARDO
   Prod[i]:=X[i]*Y[i];

// Now use fan-in to accumulate the sum of the products
//    Note: I am doing fan-in not parallel prefix
Dist=1;
REPEAT log2N TIMES
    FOR i:=1 TO N PARDO
        IF ((myRank%(Dist*2))==0) THEN
            Prod[myRank]+=Prod[myRank+Dist]
    Dist:=Dist*2;
innerPrd=Prod[0];
```

[10] 4. Extend Batcher's Odd-Even merge network to **sort** two 8 element sequences of numbers. Draw the resulting network machine and provide an illustrative example to show that your extension works.

We begin by extending the merge network alone, from the one given in class that can merge two 4 element sequences to one that can merge two sorted 8 element sequences. We use two of the smaller merge networks (with odd and even inputs grouped) and feed the outputs of the two small networks to another layer of comparators as shown. In all cases, corresponding pairs are routed to the same node in the new layer to compare adjacent result values



Now we combine the necessary stages of the merge network to implement the sort in the same way as was done in class for the 4 element networks.

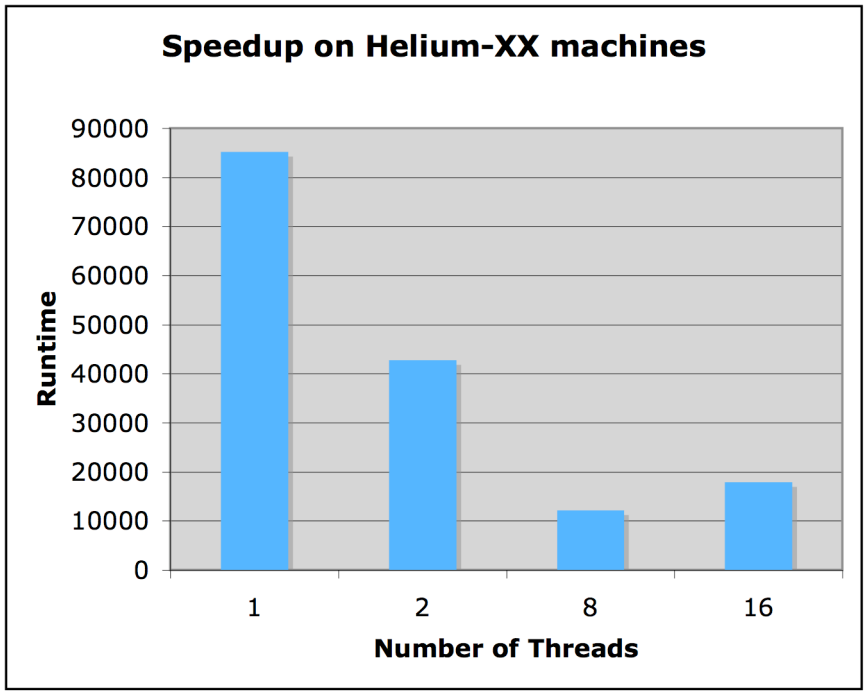


[20] 5. You are to write a simple p-threads program that will implement a parallel search in a large vector. Your main thread will declare a global shared vector of 6,400,000 integers where each element is initialized to the value of the corresponding vector index (i.e. element i in the vector will always contain the value i). The main thread should then prompt for and read two values (one between 0 and 6,399,999 which is the value to be searched for in the vector and a second (having the value of 1, 2, 8 or 16) which is the number of threads to use in the search. In C, you might use the following code sequence to do the necessary I/O:

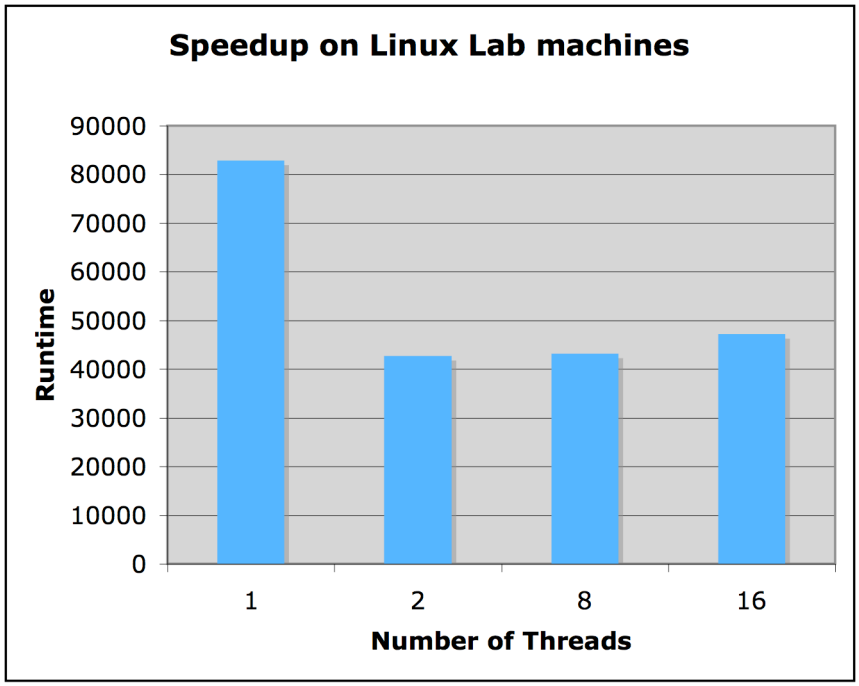
```
int  SearchVal, NumThreads;

printf("Please enter the value to search for: ");
scanf("%d",&SearchVal);
printf("Please enter the number of threads to use: ");
scanf("%d",&NumThreads);
```

Your main thread should then spawn `NumThreads` threads each running the function `search` which is passed its thread number, `MyThreadNum` (in the range 0 to `NumThreads`-1), as a parameter. The searching in the vector must be partitioned into `NumThreads` pieces with each piece being done by a separate thread. After partitioning, each piece of the vector will contain `NumElts = 6,400,000/NumThreads` elements.) Your search routine should thus search for the value `SearchVal` in elements `MyThreadNum * NumElts` through `((MyThreadNum+1) * NumElts) - 1` (inclusive) of the vector. For convenience, you may assume that `SearchVal` is declared globally like the vector. Whichever thread finds the value in the vector should print a message announcing that the value has been found and the position in the vector at which it was found. It is not necessary to stop other threads once a value is found. You are to include calls to measure the runtime of your program (C code will be provided on the homepage for this purpose if you need it). Run your program multiple times with 1, 2, 8 and 16 threads on a machine in the Linux lab and then on one of the machines helium-XX.cs.umanitoba.ca (where XX=01 through 05) and collect the results. Compute the average times for each case and then generate a simple bar graph of your two sets of runs comparing the relative average execution times. Given the results in each case, how many execution units (i.e. processors or cores) do you think each machine has? Some links to p-threads tutorials are provided in case anyone is unfamiliar with programming with p-threads.

See the course homepage for sample code (`pthreadSearch.c`). The graph of the results on the helium-XX machines is:



The graph of the results on the Linux lab machines is:

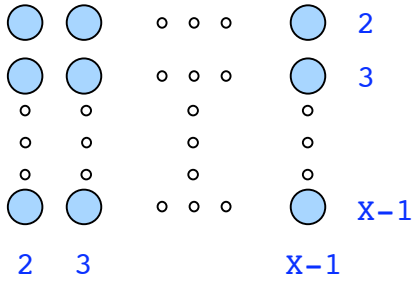


Based on the results, it would appear that the Linux lab machines have two cores and the helium-XX machines have eight cores. This is determined by the point at which adding more processors ceases to yield speedup. Interestingly, the helium-XX machines actually have 16 cores so there is some other factor affecting the performance of the embarrassingly parallel search process.

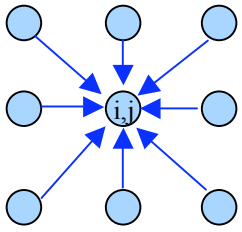
[10] 6. A simple approach to blur boundaries in a grayscale bitmap image would be to replace each pixel with the average of its immediate neighboring pixels. A given pixel, $P_{x,y}$ at location (x,y) in an image of size (X,Y) will have neighbors at (x-1,y-1), (x-1,y), (x-1,y+1), (x, y-1), (x,y+1), (x+1,y-1), (x+1,y), and (x+1, y+1) unless it is on a boundary of the image (i.e. x=0 or x=X-1 and/or y=0 or y=Y). Illustrate the application of the PCAM method to this problem. Assume that X=Y=N, that N mod 16=0 and that you have 16 processors. Justify your choices.

The PCAM parallel algorithm design methodology consists of four steps: Partitioning, Communication, Agglomeration, and Mapping.

In partitioning, we identify the fundamental individual units of computation performed by the algorithm. In this problem, each internal pixel (i.e. ones not on a border) will be updated since the blurring operation is well defined for these pixels. Thus, we could partition the problem into (X-2)*(Y-2) partitions as follows:



In the communication step, we determine the communication patterns between the partitions. In this problem, each partition communicates with its neighboring partitions as defined in the problem description since the computation is $NewP_{x,y}=(P_{x-1,y-1}+P_{x-1,y}+P_{x-1,y+1}+P_{x,y-1}+P_{x,y+1}+P_{x+1,y-1}+P_{x+1,y}+P_{x+1,y+1}/8)$. In essence, each node must send its old pixel value to each of its neighbors and every partition can do this in parallel. Hence, for a particular partition/pixel, (i,j), the communication patterns are as follows:



This communication pattern repeats throughout the partitions. We assume that pixels on the boundaries are available through this communication process.

In agglomeration, we agglomerate partitions related by communication into larger "agglomerates". Since the communication pattern is regular, we can choose to divide out collection of partitions anywhere we like and we will only have inter-agglomerate communication at the agglomerate boundaries. We also know that X=Y=N and that N is

divisible by 16, which is the number of processors. Thus, the number of partitions (including those corresponding to boundary pixels) is NxN which is also divisable by 16. We typically want a number of agglomerates which is a small multiple of the number of processors. We could easily divide up the partitions into 16 groups in each of the x and y dimensions giving 256 agglomerates. If we then chose one dimension to distribute over the available processors during mapping, this would result in 16 agglomerates per processor which is likely a larger number than would be ideal to tolerate possible scheduling issues (i.e. 16 is not as small a multiple as we would like). We could, of course, instead divide the partitions into 16 groups in one dimension and 4 in the other, resulting in 64 agglomerates and, eventually, 4 per processor. To minimize communication/sharing across agglomerates, we should always agglomerate partitions corresponding to adjacent pixels in the image.

Finally, in mapping, we assign the agglomerates onto the available processors. Assuming that communication/data sharing is equally efficient among all processors then the assignment of agglomerates to processors is arbitrary.

Total: 75 marks