CPE 412/512

Fall Semester 2010

Solution

Homework Assignment Number 3

1)Write a multiprocess message passing implementation of a matrix/matrix multiplication program where the first matrix is of size l x m and the second matrix is of size m x n. Assume that the quantities being multiplied are of type float. Write the program in a general manner to allow the number of message passing processes and threads to be independently varied between the numbers of 1 to 8. You can assume that the workload is evenly divisible between the processing entities (i.e. n/ [the number of message passing processes] has a remainder of zero). Illustrate the correctness of this program for all values of processes from 2 to 8. Expand upon the reference program given at
http://www.ece.uah.edu/~wells/cpe412_512_fl_10/material/hw/hw3/mm_mult_serial.cpp.

Please refer to the instructor's solution at
***http://www.ece.uah.edu/~wells/cpe412_512_fl_10/material/hw/hw3/mm_mult_MPI.cpp*.**
This solution was more general than was required -- data size did not have to be a multiple of the number of processes.

2)Using the altix batch queuing system (see separate handout), measure the serial runtime characteristics of the original program that was given to you for square matrices of with dimension of 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, and 2000. Graph the runtime characteristics. How does the base algorithm behave? What order is this algorithm? Does the runtime data support this?
Background note: It is interesting to note that each time you request a runtime queue on the altix system using the run_script you have to potential to get a different set of processing cores. This will affect runtime because there are several generations of processing cores (called nodes) within the Altix system. These nodes are

altix2, altix3, altix5 - single core 1.4 GHz Itanium2
altix6 - single core 1.5 GHz Itanium2
altix7, altix8 - dual core 1.6 GHz Itanium2
altix9 - dual core 1.67 GHz Itanium2

All processors within any given node are always identical.
The altix2,3,5,6 are Altix 350 series, the altix7,8,9 are Altix 450 series

The 350 series is a collection of 2 processor "bricks" in a ring topology. Thus there can be varying length lags in memory access and message passing as data has to hop more steps around the ring. Thus processors 0,1 are on brick0, processors 2,3 are on brick1, etc.

The 450 series has 2 dual-core chips per blade, and is routed between blades. Thus cores 0-3 are on blade 0, etc.
If you look at the error log file that is produced when you run the log file you will see the actual set of node listed near the top of the file.
For example

```
# Your job used:
#      8 CPUs on altix9
```

Meaning your job ran on the dual core 1.67 GHz Itanium2 slot (dual core processor).
The specific list of cores being used by a job is also printed by run_script in the error log file using two lines such as

using taskset with following CPU list
1,3,5,7,8,9,10,11

which means the job used processing cores 1,3,5,7,8,9,10, and 11 within the previously specified subsystem (altix9 in this example).

With that said, one reason you may be getting inconsistent results in timing is due to the fact that every time you request a system you are getting one with a different raw processing capabilities. Another reason is that even though, your job is the only one running on a particular processing core. Other users may be causing data contention in the interprocess communication network that exists between the processing cores. There are also caching effects and small data size affects to consider.

To overcome the queue selecting different sets of processors during my timing runs, I placed all my runs (one processor, two processor, four processor and eight processor runs) in a single run time script.(I ran it with the command: "run_script runit.scr" where runit.scr was my run time script)

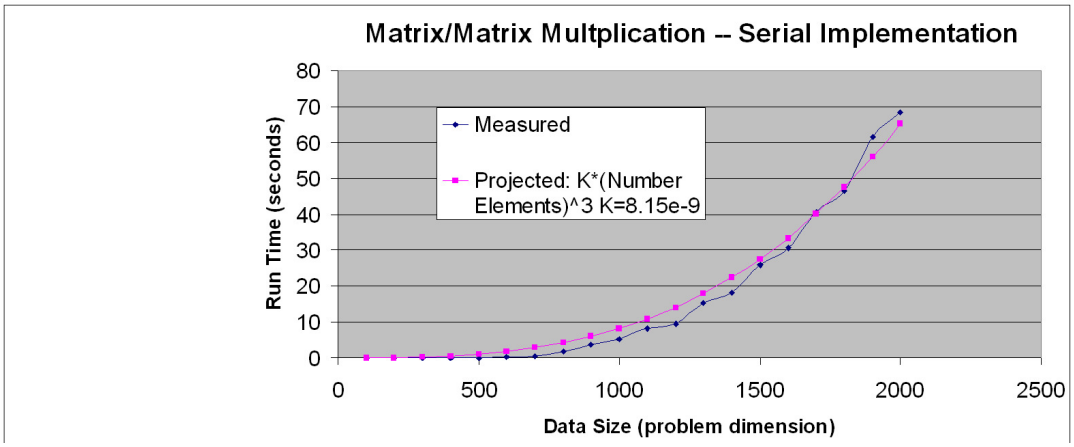
```
mm_mult_serial 100
mm_mult_serial 200
mm_mult_serial 300
mm_mult_serial 400
mm_mult_serial 500
mm_mult_serial 600
mm_mult_serial 700
mm_mult_serial 800
mm_mult_serial 900
mm_mult_serial 1000
mm_mult_serial 1100
mm_mult_serial 1200
mm_mult_serial 1300
mm_mult_serial 1400
mm_mult_serial 1500
mm_mult_serial 1600
mm_mult_serial 1700
mm_mult_serial 1800
mm_mult_serial 1900
mm_mult_serial 2000
mpirun -np 2 mm_mult_MPI 100
mpirun -np 2 mm_mult_MPI 200
mpirun -np 2 mm_mult_MPI 300
mpirun -np 2 mm_mult_MPI 400
mpirun -np 2 mm_mult_MPI 500
mpirun -np 2 mm_mult_MPI 600
mpirun -np 2 mm_mult_MPI 700
mpirun -np 2 mm_mult_MPI 800
mpirun -np 2 mm_mult_MPI 900
mpirun -np 2 mm_mult_MPI 1000
mpirun -np 2 mm_mult_MPI 1100
mpirun -np 2 mm_mult_MPI 1200
mpirun -np 2 mm_mult_MPI 1300
mpirun -np 2 mm_mult_MPI 1400
mpirun -np 2 mm_mult_MPI 1500
mpirun -np 2 mm_mult_MPI 1600
mpirun -np 2 mm_mult_MPI 1700
mpirun -np 2 mm_mult_MPI 1800
mpirun -np 2 mm_mult_MPI 1900
mpirun -np 2 mm_mult_MPI 2000
mpirun -np 4 mm_mult_MPI 100
mpirun -np 4 mm_mult_MPI 200
mpirun -np 4 mm_mult_MPI 300
```

```
mpirun -np 8 mm_mult_MPI 900
mpirun -np 8 mm_mult_MPI 1400
mpirun -np 8 mm_mult_MPI 1500
mpirun -np 8 mm_mult_MPI 1600
mpirun -np 8 mm_mult_MPI 1700
mpirun -np 8 mm_mult_MPI 1800
mpirun -np 8 mm_mult_MPI 1900
mpirun -np 8 mm_mult_MPI 2000
```

My runit.scr file

Then I chose 8 CPUs when I was selecting the number of processors during the run_script operation. This meant that a good portion of the time I had more CPUs than I needed but I would be make all the measurements using the same set of CPUs.

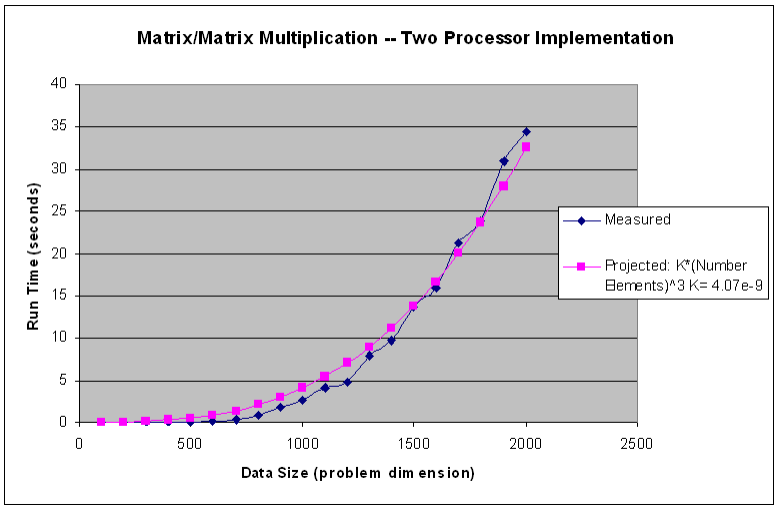
The run time characteristics of the matrix/matrix multiplication are shown below for the case where 8 processors on the altix9 subsystem were used. For the serial (only one of these processors being used) case where both matrices are square and of the same dimension then the runtime versus the data size can be plotted. If the dimension of the matrices are nxn then we will perform n^2 dot products to compute each element, which each dot product having an order n number of operations (n multiplications and n-1 additions). The overall process is therefore order n^3. This is confirmed by the graph shown below where the measured run time is plotted along with a projected along with the Kn3 curve, where K was found empirically to be approximately 8.15×10^{-9}. There is a good match between theoretical and actual data.



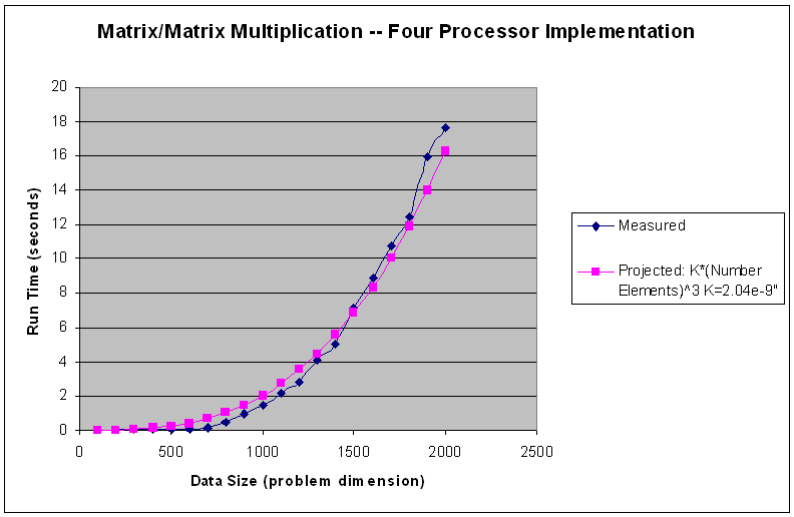
3 Repeat these measurements for the 2, 4 and 8 process MPI implementations. Create three separate graphs of the runtime characteristics. How do these implementations behave as the data size is increased? What is the order of these implementations?

Note: my parallel implementation took into account the uneven distribution of data that occurs when the data size is not a multiple of the number of processes. This is not required, though. In the case, where your implementation did not make this adjustment, then it will not correctly handle all of the scenarios that I have asked you to evaluate (i.e. 8 processor 100 element case -- 100 elements is not a multiple of 8). In that case you can just make measurements on the code as written which should approximate the performance of a working implementation.

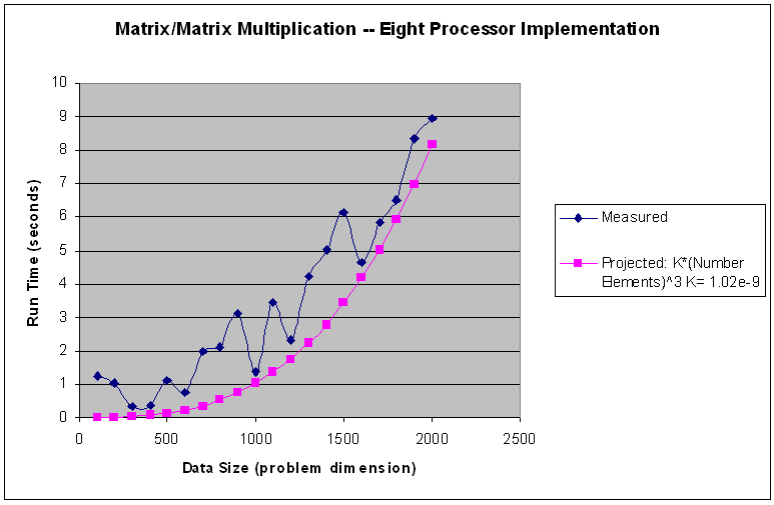
In the two processor case, there is generally an improvement in execution time. This improvement in execution time is approximately two fold (1/2 the execution time of the serial) as the data size gets larger. The graph, though has the same general shape as the serial one. This is confirmed by the relatively close match with the projected Kn^3 curve where K is equal to 4.07×10^{-9}, which is one half the serial value of K. This means that the order of the algorithm was not changed by employing parallel processing. It is still an order n^3 algorithm, but the execution time is decreased by almost a factor of 2 as the data size gets larger.



In the four processor case, there is once again a general improvement in execution time. This improvement in execution time is approximately four fold (1/4 the execution time of the serial) as the data size gets larger. The graph, though has the same general shape as the serial one. This is confirmed by the relatively close match with the projected Kn^3 curve where K is equal to 2.04×10^{-9}, which is about one fourth the serial value of K. This means that the order of the algorithm was not changed by employing parallel processing. It is still an order n^3 algorithm, but the execution time is decreased by almost a factor of 4 as the data size gets larger.

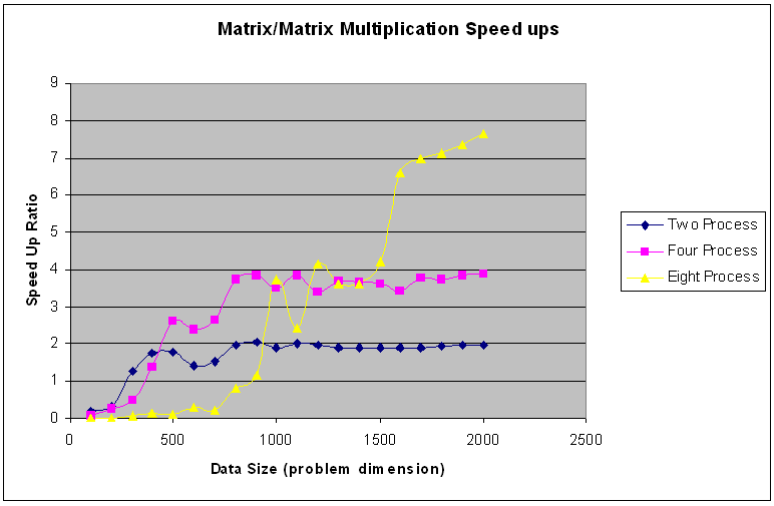


In the eight processor case, the graph is less clear. This is probably due to network contention and the differences in the latency and bandwidth between the four processor "bricks" in the system. But there is still a general improvement in execution time. This improvement in execution time is approaching eight fold (1/8 the execution time of the serial) for the largest data sizes. The graph, is lower bounded by the Kn^3 curve where K is equal to 1.02×10^{-9}, which is about one eighth the serial value of K. We could change this value of K a bit and move the graph above our measured results. This means that the order of the algorithm is n^3 which is not changed by employing parallel processing. The execution time is decreased by almost a factor of 8 as the data size gets larger.



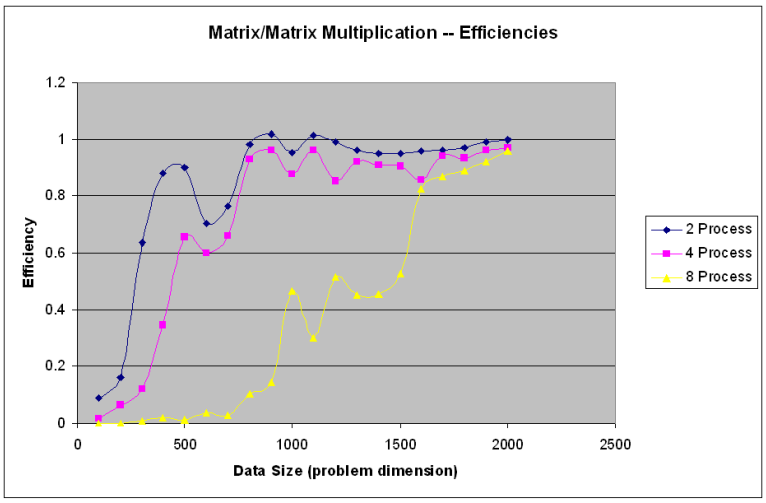
4 Using the reference serial program as a base, on a single graph, show the speedup versus the data size for each of the 2, 4, and 8 process MPI implementation. Also create a graph the shows the Efficiency versus the data size for each of the 2, 4 and 8 process MPI implementations.

Graph for speedups are shown below. There is variability within these graphs due to the factors discussed previously (non-deterministic interprocess communication, caused by other user applications, caching effects, etc.). There are a couple of cases in the two processor situation where we measured super-linear speedup. But this speedup is marginal. Reasons for it include nondeterministic effects on the serial run time caused by I/O contention and other OS scheduling effects. General trends are as the data size increases, the performance approaches that of the theoretical maximum which is the number of processing cores that are being used. Also, the smaller the number of processing cores employed, the smaller the data size has to be to reach this value. Also the larger the number of processing cores that are present the greater the degree on nondeterminism in terms of the execution time.



Graph for efficiencies are shown below. The smaller the number of processors that are employed the quicker the efficiencies approach the ideal of 1 as the data size is increased. There are two data points that exceed an efficiency of 1. These are where my application had super-linear speedup. As discussed previously, the reasons for this was nondeterminis-

tic effects in the system that caused my serial program to slow down a bit from what it would normally execute. To filter out such effects, one could run the application many time over each data point and take the average of these runs.



Follow the program turn-in procedure outline below. *Due date for In-class students is Tuesday September 21, 2010. Due date for DL students is Thursday September 23, 2010*

Homework Assignment Turn in Procedure

You are to turn in an electronic copy of the homework assignment on or before its due date. This copy should contain a printout of all source code and the resulting output of the program or some form of a log of interactive activity captured in a text file. Acceptable file formats include pdf, doc, and ascii text. All files should be sent as standard E-mail attachments and should include all source level files and brief instructions as how to compile and execute your code. No object code files should be included. The subject line of the E-mail should contain the following; CPE x12, HW #num, where x is 4 or 5, representing which course you are taking (412 or 512) and num is the homework assignment number (Example CPE 412 student would put CPE 412, HW #1 for the first homework assignment. The E-mail is to be set to wellsbe@uah.edu.