Nathaniel A. Wendt, Daniel J. Rahm, and Christopher King

Dr. Yanqing Ji

CPSC/CPEN 435: Parallel Computing

Final Project

04/30/2013

Gonzaga University

Department of Electrical and Computer Engineering

**Introduction**

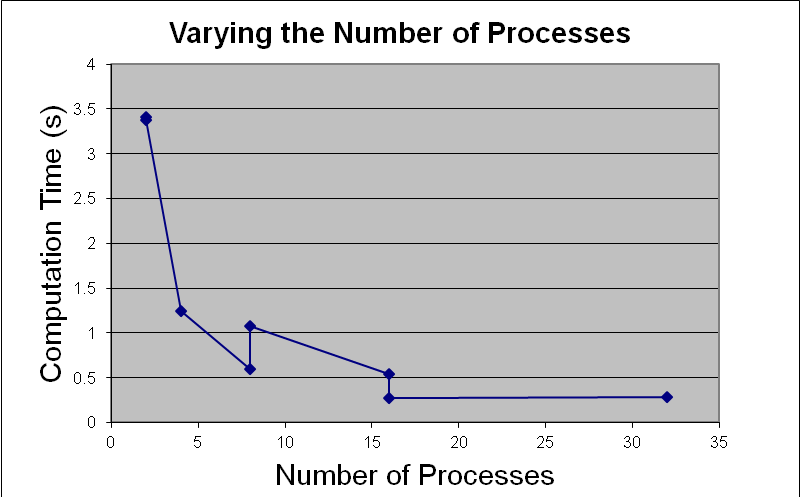
The purpose of this project is to explore the capabilities of hybrid MPI and Pthread program integration and their effects on matrix multiplication operations. The model used in this report will be matrix multiplication as A x B = C, where A, B, and C are matrices of n x n elements.

**Implementation**

The program uses MPI because the computations occur on a cluster of nodes, and threads are used to make the most of the shared space programming opportunities on each node. The MPI will use a master/slave model to distribute the tasks and the threads on each node will perform the computation. MPI non-blocking send and receive is used, and the threads make use of disjoint buffers to overlap computation and communication. Finally, the user will enter the number of messages the master will send out to each process and the program will record and output the computation time.

**Performance Analysis**

Scalability for Processes

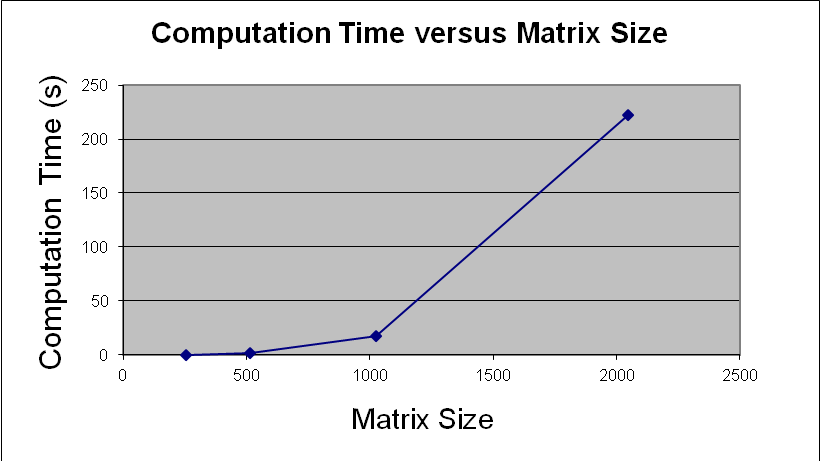


**Figure 1**

Figure 1 shows the how varying the number of processes affects the scalability of the program. Both the matrix and blocksize were fixed at 512, and the number of threads was set to four. So, only the nodes and the processors per node varied.

As more processes are used the performance increases until the processes equal sixteen and that the performance plateaus. Figure 1 has multiple data points for 8 and 16 processes because there are two combinations in which to assign nodes and processes for each of these numbers. For 8 processes, there can be either 4 nodes with 2 processes per node, or 2 nodes with 8 processes per node. In these cases, the more efficient data point is the one where the number of nodes is increased instead of number of processes per node. If 4 or more processes are assigned to each node, in addition to running concurrent threads, the resources of the node are fought over and performance decreases.

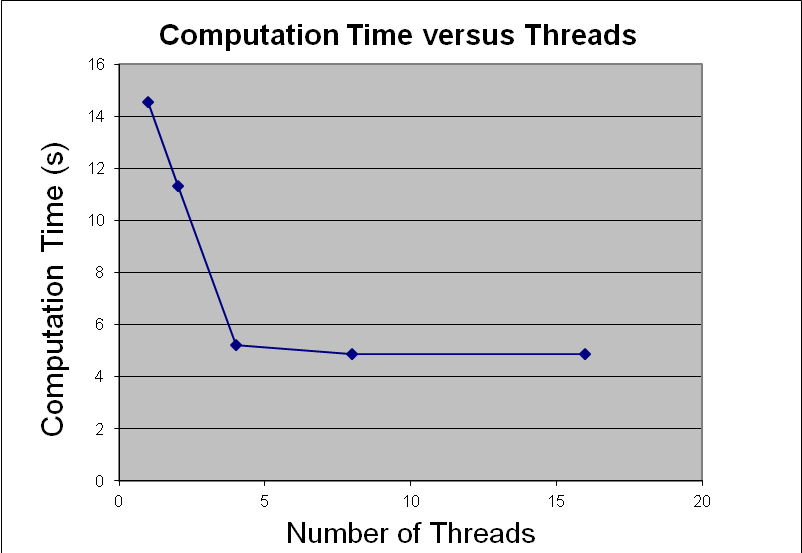
Scalability of the Matrix Size



**Figure 2**

Figure 2 shows how scalable the program is for increasing sizes of the matrix. As the matrix gets larger the computation time takes longer. So, this program is not scalable for large matrices.

How Threads Impact the performance



**Figure 3**

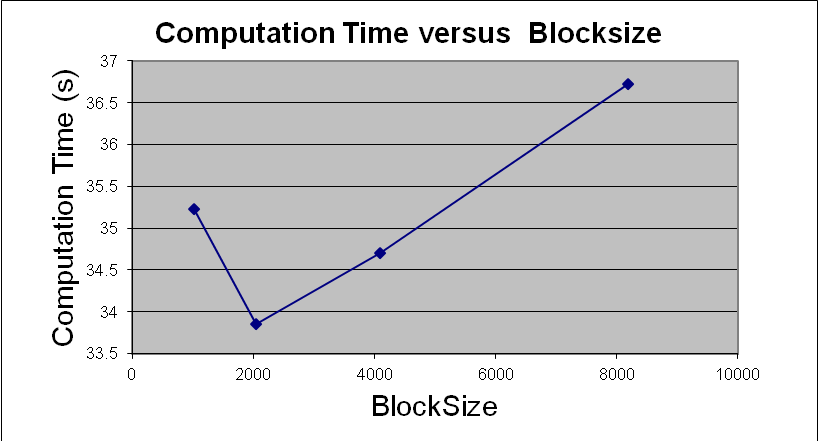
Figure 3 shows how the number of threads changes the performance of the matrix. In this case, the number of nodes and processes per node are set at two. Both the matrix and blocksize are set at 512.

As evidenced by Figure 3, adding more threads improves the computation time. However, the performance plateaus after 8 threads due to the competition for resources.

Effect of Message Size on Performance

Figure 4 shows how the computation time changes for a given blocksize. The parameters for this test were two nodes, two processors per nodes, two threads, and the size of the matrix set to 1024.

The optimal performance occurred with blocksize set to 2048, and as it decreased or increased the computation time increased due to the communication overhead. The general case for this is when the block size is set to be two times the size of the matrix (1024 in this case).



**Figure 4**

Comparison with Previous Labs

Table 1 below shows the performance of this algorithm with previous labs. The final project operates more efficiently than Lab 3 and Lab 6, but less efficiently than Lab 5. This is probably due to the overhead necessary to set up the thread libraries and initialize the extra memory for computations. However, overall, the project can more efficiently scale to higher values of n with the extra help of threads in addition to MPI library techniques.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n = 512 | 4 processes | 8 processes | 16 processes |  |
|  | 1.24393 | 0.598316 | 0.53539 | Project |
|  | N/A | 1.074522 | 0.2747 | Lab 3 (Less processes and more nodes) |
|  | 2.093 | 1.391 | 0.785 | Lab 3 (More processes and less nodes) |
|  | 1.0483 | 0.424 | 0.399 | Lab 5 (Non-blocking commands introduced) |
|  | 1.473 | .739614 | .376 | Lab 6 (Only threads) |

**Table 1**

Discussion and Conclusion

Some difficulties in implementation include allowing the program to scale to have various block sizes. These block sizes change the sizes of the send and receive buffers and also how the threads break up how much work to do. Rather than having the threads be statically calculated based on the number of rows assigned to a process, it had to vary based on the block size. Furthermore, allowing the program to work with different numbers of threads was difficult because the bounds of the computation buffers changed based on the block sizes. Going from 1 to 8 threads was radically different and needed its own case to implement. Calculating which portion of the array to access was the most difficult portion of the lab.

The hybrid MPI and Pthread program performed very well, but one needs to keep in mind how to implement it properly. First, it is not scalable for large sizes of matrices and for large amounts of processes. However, for the matrices used, it operates very efficiently if one uses the right amount of nodes, processes, and threads. It is best to have more nodes and fewer processors per nodes than the converse. Threads per process must not compete with one another for access to memory otherwise the performance of the program will plateau or deteriorate. Finally, one must send the proper amount of message data; communication overhead will occur for too little or too large amounts of sent data.

In conclusion, this algorithm computes large amounts of data in a very efficient manner if one keeps in mind the correct parameters to use to optimize its efficiency.

Contributions

Nathaniel A. Wendt: Algorithm Design and Implementation

Daniel J. Rahm: Algorithm Testing

Christopher J. R. King: Data interpretation and Report Writing