A Hybrid Process Farm/Work Pool Implementation in a Distributed Environment Using MPI

Nathan Sachs
Mathematics and Computer Science Department
South Dakota School of Mines & Technology
Nathan.Sachs@gold.sdsmt.edu

Jeffrey McGough
Mathematics and Computer Science Department
South Dakota School of Mines & Technology
Jeff.McGough@sdsmt.edu

Abstract

This paper is aimed at those seeking a simple way to implement distributed tasks, those tasks intended to be run on a cluster of computers. The focus is on a tool which allows a programmer to create these types of programs simply by defining sub-tasks and what data will be accessed during each operation. This allows the programmer to write programs in a more algorithmic manner, leaving communication details to the underlying system.

The tool mentioned employs characteristics of both the process farm and work pool models and achieves communication through the use of MPI. Mechanics of the tool are discussed and timing tests comparing the tool with handwritten MPI programs are presented.

Introduction

Although desktop computers today are far superior to the largest of machines from just a few years ago in both speed and storage capacity, they are still too slow and too small to handle some problems. Engineering simulations, market predictions, weather simulations and many other applications often require machines with processing power or storage capacity that far exceeds that which can be found in inexpensive, stand-alone machines. In the past, people spent exorbitant amounts of money to rent time on large specialized supercomputers to do their processing. Smaller multiprocessor machines of all sorts have also proven themselves capable of solving large problems in a shortened amount of time. Yet consumers have still shied away from the costs of these systems. Of late, there has been a trend toward clusters of common, desktop machines connected by a generic network. These clusters, or distributed systems, yield a high return on investment which, combined with the relative ease and low costs of maintaining and upgrading these systems, attracts many companies today.

Programming distributed systems poses a lot of challenging problems for programmers. The main problem in these environments that is generally absent in multiprocessor systems is the lack of a shared memory between the processing units. Clusters are generally made up of a collection of stand-alone machines. Each of these machines has its own memory space which cannot be directly accessed by other computers. If a variable is stored on one computers and needs to be accessed by another it has to be communicated across the network. No default mechanism will handle this communication for the programmer. Research projects from several universities have come up with various ways of implementing a distributed shared memory. An interesting list of these and other tools for parallel programming may be found at [1]. These tools are very generic and are able to maintain copies of data on several machines and automatically update all copies when one is modified. This approach is very, yet the programmer is often familiar enough with the application that she can greatly reduce the amount of network traffic by using some other means such as passing information explicitly using the standard message passing interface (MPI).

Another challenge that immediately plagues programmers when writing distributed software is the problem of correctly and efficiently dividing the problem into segments which can be executed on different machines simultaneously. One of the major goals when breaking problems apart is to minimize the amount of time that the processing units are idle. There is a high correlation between the amount of idle time and the total amount of time needed to solve the problem. Generally speaking, if the idle time among the various machines increases then so will the length of time used in solving the problem at hand. Breaking the problem into pieces that will keep all machines busy can be quite difficult.

An additional aspect of this complexity is that any given network is different from the next. One network may have fast computers and slow communication. Another may have slower machines and faster communication. Still others may have a variety of different types of machines, some with many processors, others with only one, and all of

these running at different speeds.

We have developed a tool that handles many of these issues for the programmer allowing him or her to focus on algorithmic concerns. This tool, called the MPI/Workpool, mixes the process farm and work pool paradigms to organize and distribute work to be done and makes use of MPI for inter-node communication. MPI was chosen because it is highly portable and widely available [2]. The MPI/Workpool system handles all communication for the programmer and seeks to minimize the amount of idle time on the various machines by matching available work with available machines. The programmer will be expected to partition the work into its constituent parts and indicate to the system how those parts interact. Additionally, she will need to tell the system what portions of memory are to be accessed for reading and writing during operation of each subtask. The MPI/Workpool is implemented as a set of user level libraries. All a programmer needs to do to make use of it is to link with these libraries. In the body of this paper we will seek to describe this tool and how it provides this service. Specifically, we will discuss how the various machines are told what to do, how the data gets passed around, and how the results are collected when work is complete.

A Brief Mathematical Background

Many of the problems for which distributed systems may be used are heavily based in linear algebra computations. This is due to the fact that when creating large simulations such as those mentioned above-weather simulations, engineering simulations, and so forth-large systems of equations often arise. The solutions to these systems are often found using linear algebraic techniques. For example, a mechanical engineer may wish to find the eigenvalues of a system in order to minimize vibration on a structure in motion [3]. An electrical engineer may wish to solve a large system of equations in order to calculate 3D capacitance in a microelectronic circuit [4]. An atmospheric scientist may also wish to find the solution to a system of equations for the purpose of modeling the interactions of various chemicals in the atmosphere-ocean system [5] or for simply predicting the weather [6]. Each of these applications involves computationally intensive linear algebra.

These types of problems are well suited to distributed systems. The mathematical operations involved may often be reduced to a series of operations that yield the same results. Properly performed, these reductions may yield algorithms that can be implemented in parallel. Here we take a look at a very simple parallel implementation of matrix multiplication, one of the clearest parallel algorithms there is. We also take a glimpse at a parallel method for solving a triangular system of equations. This will allow us to demonstrate the necessity of holding back some operations until others are completed. Throughout the paper we will refer to these two examples in describing aspects of the MPI/Workpool system.

Matrix Multiplication

Suppose we have two matrices A and B, both n-by-n. To form the matrix $C = C + AB$, we can use the equation $c_{i,j} = c_{i,j} + \sum_{k=1}^{n} a_{i,k} * b_{k,j}$ for all $c_{i,j}$ in C. In code we may write:

```
for ( i = 0; i < n; i++ )
    for ( j = 0; j < n; j++ )
        for ( k = 0; k < n; k++ )
            C[i][j] += A[i][k] * B[k][j];
```

If we want to do the same operation in parallel, we may divide the matrices A and C into block rows with height nb. In other words, we can treat the first nb rows of A (or C) as a matrix in and of itself. Then, the calculation of the i^{th} block row of C, call it C_i, can be calculated using $C_i = A_i * B$ for all i. Now, we can let each operation take place on a separate processor. In the end, we'll simply gather the results into the matrix C. This gives us a straight-forward parallel algorithm for solving the problem of matrix multiplication.

Solution of a Triangular System of Equations

A slightly more interesting algorithm is the parallel solution to a triangular system of linear equations, discussed in [7]. This is a system of n equations of the form:

$$a_1 x_1 + a_2 x_2 + \cdots + a_k x_k = b_k, \quad k = 1,2,\ldots,n$$

This can be represented by $Ax = b$ where A is an n-by-n, lower-triangular matrix and x and b are vectors of length n. We'll divide each of A, x, and b into blocks of order nb and number the blocks as shown in figure 1.

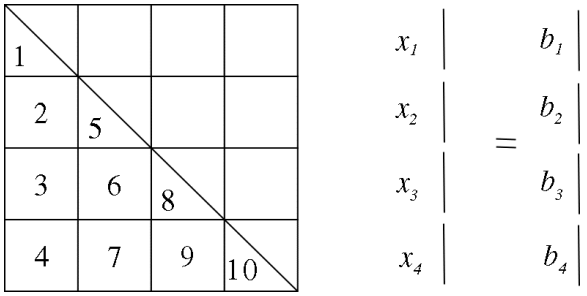

Figure 1: Blocking of A, x, and b for Parallel Triangular Solve

In the diagram we have assumed that nb is such that $3nb < n \leq 4nb$. In other words, the blocking factor nb yields four rows and columns in A. The algorithm described will work for any positive value nb, but for purpose of illustration we will assume A has four block rows and columns.

Note that x_1 can be found by solving the smaller triangular system $A_1 x_1 = b_1$. Then, one may calculate x_2 by solving the equation $A_5 x_2 = b_2 - A_2 x_1$. Another approach, however, would be to modify b_2 by saying $b_2 \leftarrow b_2 - A_2 x_1$. We can do this same computation for b_3 and b_4. In other words, $b_3 \leftarrow b_3 - A_3 x_1$ and $b_4 \leftarrow b_4 - A_4 x_1$. Now we have solved for x_1 and have a smaller triangular system to solve, depicted in figure 2.

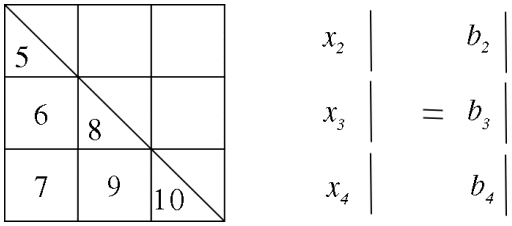

Figure 2: Reduced Block Triangular Solve

We may now solve the reduced system using the same method. Note, however, that the system need not be completely reduced before computing x_2 using $A_5 x_2 = b_2$. Specifically, b_3 and b_4 need not have been updated before this computation. However, b_3 must be updated twice before the solution of $A_8 x_3 = b_3$. Since the operations $b_3 \leftarrow b_3 - A_3 x_1$ and $b_3 \leftarrow b_3 - A_6 x_2$ cannot occur simultaneously, we will somewhat arbitrarily say that the latter depends on the completion of the former. Those following references may notice a slight discrepancy between figure 3 and the graph in [7]. This is due to the fact that we are making this assertion. Now, we have the following ten steps:

Table 1: Steps Involved in Performing Blocked Triangular Solve

Step 1: Solve $A_1 x_1 = b_1$	Step 6: Update $b_3 \leftarrow b_3 - A_6 x_2$
Step 2: Update $b_2 \leftarrow b_2 - A_2 x_1$	Step 7: Update $b_4 \leftarrow b_4 - A_7 x_2$
Step 3: Update $b_3 \leftarrow b_3 - A_3 x_1$	Step 8: Solve $A_8 x_3 = b_3$
Step 4: Update $b_4 \leftarrow b_4 - A_4 x_1$	Step 9: Update $b_4 \leftarrow b_4 - A_9 x_3$
Step 5: Solve $A_5 x_2 = b_2$	Step 10: Solve $A_{10} x_4 = b_4$

As mentioned some of these steps require prior completion of others. We may draw a graph of these dependencies as follows with the arrows indicating the order in which the steps must occur, proceeding in the direction of the arrows.

As can be seen, each of steps 2, 3, and 4 may begin as soon as step 1 is complete. Generally it is up to the programmer to ensure that steps do not begin prematurely by using locks of some sort. MPI/Workpool will take care of this blocking for the programmer, releasing the available work when dependencies have been fulfilled. This will be discussed in greater detail below.

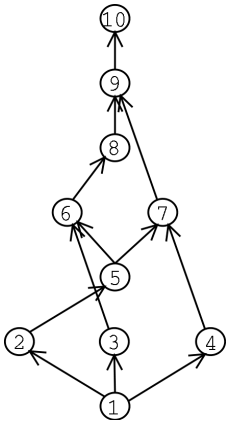

Figure 3: Dependency Graph for Triangular Solve with 10 Nodes

Algorithmic Paradigms for Parallel Programming

As mentioned above, the MPI/Workpool approaches distributed computing using a mix of two common parallel programming paradigms. Parallel processing, as used in this document, refers to programs that can have multiple threads or processes on CPUs simultaneously, and are run on shared variable machines. Distributed computing, on the other hand, refers to programs that run simultaneously on separate machines connected via a network which lack the ability to directly access one another's memory spaces. Process farming is an approach used on both parallel and distributed systems, while work pools require at least some shared memory. The MPI/Workpool approach takes from both of these to provide a work pool like approach in a distributed setting.

Process Farms

A pair of common shared variable paradigms have been employed in creation of the MPI/Workpool. The first of these is known as a process farm. The idea of a process farm is that a master process can compute the sequential portions of the code independently, and then spawn slave processes when it is time to do the parallel sections of code. This has the benefit that the parallel and sequential portions of code can be completely separated. However, it does not have any facilities for load balancing. [8]

Work Pools

Another approach is the use of a work pool. A work pool is a global structure that maintains a list of work to be done. This list can be an unordered list, a standard queue, or a priority queue. The global nature of the work pool structure along with the benefits described below make this approach a common one in the shared variable setting. Idle processes can query the work pool for new work, complete the assigned work, and may sometimes create new work to be placed into the work pool. Each item in the work pool defines a task of work that needs to be done and any information needed for performing

the task. If no work is available when the work pool is queried the requesting process may begin waiting until either work becomes available or the task for which the work pool was created is complete. Load balancing is a natural side affect in work pools; while one processor is busy with a lengthy task another can busy itself with many shorter tasks. [8, 9]

One attractive feature of work pools is that they allow the parallel portion of code to be sub-divided into tasks, thus giving the code more of a procedural feel. For example, the algorithm for performing a parallel triangular solve discussed above consists of two sub-tasks: solve a small triangular system, and do an update on the *b* vector. Although these sub-tasks happen many times and in an iterative fashion, the entire process can be viewed as the repeated use of these two tasks on various parts of the data. As seen above, certain of these steps will need to be completed before others begin. This fits well into the work pool model. As step n is completed those steps that depended on step n may be added to the work pool. Thus, by defining each of these sub-tasks we can create a work pool that will perform the full triangular solve. Each node in the work pool will contain an indicator which task to perform and a manifest describing which portions of the data to operate upon. Since many of these tasks can be executed simultaneously we have a parallel implementation of the process.

MPI/Workpool's Paradigm

The MPI/Workpool is neither a true process farm nor a true work pool. Much of the work pool structure is there. Processes can query the work pool for work. Each work node contains an indicator of which task to perform and has associated with it a list of objects in memory needed to perform the task. However, since this work pool is intended for use in a distributed setting, only one of the many participating machines has access to the work pool structure itself. This node, then, becomes the master node keeping track of what work is available and what processes are seeking work. This master node has the responsibility of pairing nodes seeking work along with available work in an efficient manner. The master must also transmit the needed portions of data to the worker nodes when they are assigned a task and receive results from those nodes when the task is complete. The worker nodes must simply be ready to request work and know how to do that work when assigned. Thus, the benefits of the work pool model have been brought to the realm of distributed computing.

Internal Organization of the MPI/Workpool

The work pool has three main structures for keeping track of those processes seeking work and that work which is available. There is an array of work that needs eventually to be completed, a priority queue of work that is available immediately, and a priority queue of processors waiting for work. Clearly, at any given time either of the priority queues will be empty; if a process becomes available and work has been waiting then they will be paired up immediately, and vice-versa.

The following sections discuss some of the conceptual ideas of the MPI/Workpool and how they are used to pair work with available processes as quickly as possible while ensuring correctly running code.

Work Nodes

A work node is the representation used by MPI/Workpool to describe a piece of work to be done and the associated memory objects accessed during the processing of the task. The work pool contains an array of these nodes. However, not all of the work nodes in this array are considered to be in the work pool. Only those nodes currently available to be worked, in other words, those nodes that have been moved into the priority queue, are considered to be in the work pool. How, then are work nodes moved from this array of all work to be done to the priority queue of available work? The answer to this question relies on a discussion of the treatment of dependencies within the MPI/Workpool.

Dependencies

When a work node is created it simply contains a task id and a priority. The task id indicates to the receiving worker process the type of operation that is represented by the work node. The priority indicates to the master process how urgent the processing of the node may be. Work nodes will be assigned from the work pool in order of their priorities, starting with the highest. After the node is created, objects may be added which indicate what data to send to the worker process for the completion of its task. Still, the node is not considered to be in the work pool. Rather, it is simply in the list of work that will eventually need to be done. A work node is moved from this list into the actual work pool by having all of its dependencies filled.

Dependencies are inherent in many parallel programming algorithms. Figure 3 depicts the dependencies involved in performing a parallel solution to a triangular system of linear equations. Each dependency in the graph represents a data dependency between two steps. For example, step 1 computes x_1, which is needed by each of steps 2, 3, and 4. This implies data dependencies from step 1 to each of these steps. Generally, any case in which a piece of data is to be used by one step and written by another there is a data dependency between the two steps. This means the two steps must be performed in the same order as they would have been if executed sequentially.

Any case in which locks are used to ensure dependencies are met gives rise to the risk that the system might go into a state of deadlock. This occurs when a ring of dependencies is formed. For example, if step 5 depends on step 6, which depends on step 7, which, in turn, depends on step 5 then deadlock will occur allowing none of these steps ever to begin. In terms of the MPI/Workpool, none of these steps will ever enter the work pool. Thus, the list of work to be performed will never be complete and the worker nodes will wait indefinitely because there will always be work left to perform, yet the work pool will be empty. No dependencies will be able to be fulfilled, so nothing new will move into the work pool. This can be avoided by creating a dependency graph that

contains no cycles. In the absence of cycles the dependency graph will have a finite length and can always be completed, assuming each task has finite duration. Currently, the MPI/Workpool leaves to the programmer the tasks of determining dependencies and ensuring that they do not create a cycle.

Priorities

As is implied in Figure 3 steps 2, 3, and 4 in the triangular solve may occur simultaneously. Notice that due to the dependencies an unlimited number of nodes would still need seven time steps to complete the operation, assuming each step takes one time step, and the time needed to transition from one step to the next is ignored. Suppose, however, that we only have two worker processes. It is still possible to finish the entire operation in only seven time steps. However, to ensure this optimization requires the use of priority.

If the priorities for each step are set to be the reverse order of their respective step numbers, in other words step 1 has the highest priority, then step 2, and so on to step 10, then the operation can happen in only seven steps on two worker processes. Step 1 would occur first followed by 2 and 3, simultaneously. Next, steps 4 and 5 can happen followed by 6 and 7. Finally, steps 8, 9, and 10 would have to occur sequentially in order.

If, however, the priorities were ordered in the opposite way, with step 10 having the highest priority and step 1 having the lowest, things are not quite as efficient. In this case step 1 will still occur first. Although step 1 has the lowest priority, it is the only step that can run in the beginning of the program, being the only node in the dependency graph having no dependencies. Once step 1 completes steps 3 and 4 will run, followed by step 2. Step 5 is next, followed by 6 and 7. Finally, steps 8, 9, and 10 occur sequentially to finish off the process. This scenario will run in eight time steps.

So, we can see that the proper use of priority in this situation can reduce the amount of time from eight time steps to seven, approximately a 12.5% decrease in the run-time. Other situations can lead to a much higher improvement in run-time due to good use of priority. This is particularly the case in those algorithms that have wide dependency graphs and dependencies that skip many levels in the graph.

Priority in the MPI/Workpool is assigned to a node of work when it is first created. This value is then used as the priority when the node is added to the work queue. So, when a worker process requests a work node it will be assigned one that has the highest priority out of those that have their dependencies filled.

Objects: The Medium by Which Data Are Communicated

An object is the term used by the MPI/Workpool to refer to a portion of memory that must be packaged and sent along with a work node in order for the receiving process to

be able to complete its assigned task. Objects are often associated with variables, although a given variable may have multiple objects within its boundaries. For example, if the programmer wishes to multiply two blocks within a single matrix, then each of those blocks can be treated as objects. Objects need not be laid out in memory in contiguous blocks. For example, the column of a matrix (in C or C++) may be viewed as a vector object. This will not be a contiguous block of memory unless the matrix is only one column wide since, in C, only the rows of matrices are stored contiguously in memory. Nevertheless, it may be that just that column is needed for processing some task. It would be silly to pass the entire matrix. Rather, a vector object can be defined which allows for this sort of discontinuity.

Many Types of Objects - Matrix, Vector, Cube, etc.

In addition to vector objects there are many other possible types of objects. Matrix objects would be especially needed in distributed scientific computing applications since most well known parallel algorithms for linear algebra treat matrices as collections of sub-matrices. Cubic objects may also be useful, and linked list objects as well. The application will guide what object type to use.

Due to the limitless types of objects a programmer might want to use we have opted to make a general interface for defining objects rather than creating a concrete list. Currently the only object type that is fully implemented is a matrix object, yet others may be readily added. Below is a section on how objects may be defined.

Mobile Objects

As mentioned above, the two tasks in the triangular solve are each used many times during processing and each use differs from the others only by the data that are accessed. So it is in the matrix multiplication example. We have that $C_i = A_i * B$ for all i. Rather than creating an object for each C_i and another for each A_i we will only create one object that represents a block row in C, call it Object 1. We create another in A, call it Object 2, and a third for the entire matrix B, Object 3. Objects 1 and 2 we will cause to start in the first block row and continue on down to the bottom of the matrices. Figure 4 shows how this will occur.

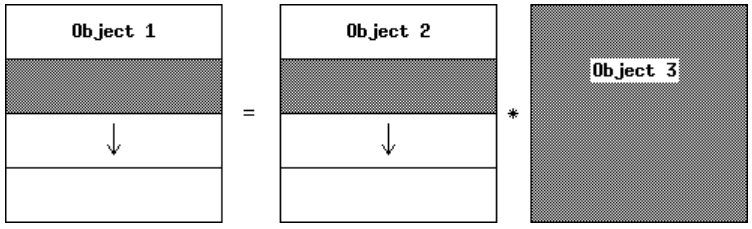

Figure 4: Objects Used in Block Matrix Multiplication

Now, for each instance, k, of Object 1 we will add a node of work to the work pool which

will calculate $C_k = A_k * B$. The MPI/Workpool refers to these instances as iterations of the object.

The Object Handler

Due to the extensible nature of the object handling software it is necessary to have a uniform interface to a general object. Otherwise, the addition of a new object type would give rise to a need for modifying either the work pool code itself or the object handling software to be able to handle the new object type. The removal of this new object type would be an even greater task since the code defining the object would have necessarily been intermingled with the object handling code itself. To avoid such confusion the object handler acts as both an interface to the objects themselves and as a tool for creating, destroying, and otherwise managing objects.

The object handler maintains a list of all objects that have been created. It is through this list that an object may be referenced. Each object, upon creation, is given a unique index into the object list. So, to access a specific instance of a specific object both its index and iteration are needed. It is these two numbers that the programmer uses when wanting to associate an object with a work node.

Defining a New Type of Object

Object types can best be described in object-oriented terms. It is useful to consider an abstract base class that defines a generic object. An object type would be a class derived from this base class. Unfortunately, since the MPI/Workpool is written entirely in C, classes could not be used. What is done instead is to define each object type in terms of a set of functions that can act upon it. A structure called ObjectFunctions maintains a set of function pointers for a single object type. Elements of this structure include, but are not limited to:

- getObjectAttribute: get some attribute about an object such as the height or width of a matrix object,
- packObjectForMPI: copy an object into a buffer suitable for sending using MPI,
- unpackObjectForMaster: copy an object out of a receive buffer into its containing structure,
- unpackObjectForWorker: copy an object out of a receive buffer into a buffer which was created for holding exactly one instance of the object.

For any new object type each of these functions must be created. Once written, however, they simply need to be linked into the object handler by adding a few items to objectHandlerStructs.h and one line to objectHandler.c.

Finally, a structure defining the data needed to describe specific instances of the object type must also be created. For example, the structure for a matrix object may have fields for defining the height and width of an object instance.

Thus, the object handler may be easily extended to handle any sort of object a programmer may wish to throw at it. Having done this the routines needed for handling the new object type will be stored in the library libobjects.a and will be available in any subsequent compilations. Removing an object type is as easy as removing the few items that were added to objectHandlerStructs.h and objectHandler.c.

Communication: How it happens

The absence of a need to worry about communication issues is one of the nicest features of the MPI/Workpool. Nevertheless, it is always good to understand what is going on under the hood. Perhaps the process of communication could be best illustrated by use of an example. For purposes of this example we will let a three-tuple represent an object accessed by a particular work node. We will say that (a, b, y) indicates that object a, iteration b is referenced by the work node and it is written. Similarly, (a, b, n) indicates the task will access, but not modify, the object. We will also refer to an object simply by the notation (a, b). Table 2 shows a scenario in which there are 3 nodes of work to be done, 2 of which are in the work available queue.

Table 2: A Possible Scenario in the Work Pool

	Node 1	Node 2	Node 3
Dependants	3	3	4,5
Dependencies	0	0	2
Objects Accessed	(1, 1, n), (2, 2, y)	(1, 2, y), (3, 1, n)	(1, 1, y), (1, 2, n)
Available	yes	yes	no
Finished	no	no	no

The dependency graph for this situation is given in figure 5.

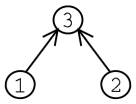

Figure 5: Dependency Graph for Scenario in Table 2

Now, suppose we have two worker processes digesting the work in the pool. For the purpose of this example we will assume all work nodes have the same priority. This is how things might proceed.

Processor 1 requests work from the MPI/Workpool via an MPI message. Work is available, so node 1 is removed from the queue. The manager node sends a message to processor 1 containing the task id of node 1 along with a note that objects (1, 1) and (2, 2) will follow. The manager sends object (1, 1), followed by (2, 2). Processor 1 receives the messages in turn and begins processing of work node 1. Subsequently, processor 2 requests work and has a similar interaction before starting to process work node 2.

Processor 2 finishes with node 2 (processor 1 is still working on node 1). Having completed its task, processor 2 sends a message to the manager stating that work on node 1 is complete. At this point, processor 2 will send all written objects back to the managing process in the same order in which they were received. In processing node 2 there was only one written object, namely (1, 2), so only that one is sent back to the managing process. Before listening for subsequent messages the manager will decrement the number of dependencies left for each of those nodes that depend on node 2. Node 3 depends on node 2 and has a dependency count of 2, which is set to 1. Having successfully notified the manager of completion, processor 2, once again, requests work. Now, there is no work available so the manager places process 2's request into a priority queue of waiting processes.

Later, process 1 posts a message that it has completed node 1. Similar to before, object (2, 2) is communicated to the manager by processor 1. After receiving object (2, 2) the manager decrements the number of dependencies on node 3 by 1. Node 3 now will have zero dependencies and so the manager immediately sends the needed messages to processor 2 to get it working on node 3. Note that up until this time processor 2 has simply been waiting on a response. The request for new work is a blocking one and processor 2 will wait indefinitely until work supplied. Another point to mention here is that processor 2 already has object (1, 2). Knowing this, the manager will skip sending this object to processor 2.

Now, when processor 1 requests work the table of work to be done is empty. Rather than putting processor 1 into the queue of processes waiting for work, the manager will send a message to processor 1 stating that work is complete. Processor 1 may now begin the process of cleaning up and exiting. Similarly, the next time processor 2 requests work it will be relieved of its duties.

Timing Tests

Since the purpose of the MPI/Workpool is to help programmers write distributed software more easily and, assumedly, distributed systems are meant to yield faster results than their uniprocessor counterparts, it only makes sense that we would show the results of a few timing tests. Figure 6 compares the time used in running the parallel matrix multiply algorithm using the MPI/Workpool versus a hand coded MPI version of the same. These tests were run on a Beowulf cluster with 16 nodes. One node was used to distribute the work, while the remaining fifteen acted as worker nodes. The matrices A and C were divided using a blocking factor of 50. Timings were taken by running each algorithm at each specified matrix size at least 6 times and continuing until the second best time was no more that 0.5% greater than the best time. After this, the best time was taken to be the time needed to do the computation. The reason for the bump at matrix sizes of 750 and 800 is the fact that this is the point at which there were more pieces of work to be done that processors for doing the work.

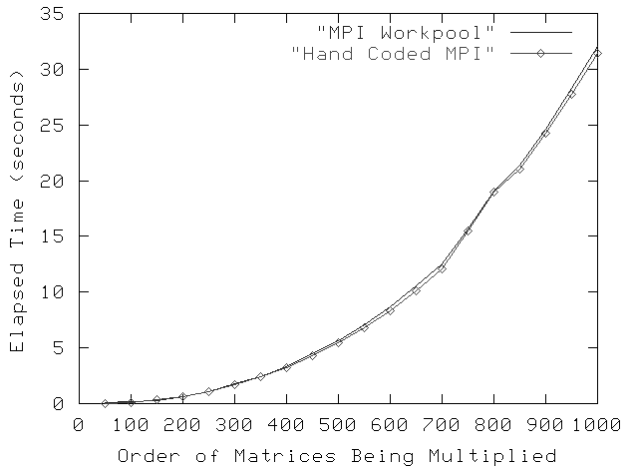

Figure 6: Comparison of Run Times for Distributed Matrix Multiply

Naturally, the MPI/Workpool introduces a little bit of bookkeeping overhead. In fact, with a matrix size of 50 (1 block row), the MPI/Workpool took nearly twice as long to solve the problem as the hand coded version; and, with a matrix size of 100, it took 10% longer. However, for matrix sizes of 150 and greater, the MPI/Workpool never took more than 5% longer than the hand coded MPI program.

System Limitations

As is the case with work pools in the shared variable environment, communication between processes is not easy or advised. This is largely due to the fact that there is no way to tell what the other processes are doing. They may be working on some other task, the same task, or nothing at all.

Conclusions

The MPI/Workpool provides a simple interface for creating programs targeted toward the cluster environment. This tool allows a programmer to create a distributed program at an accelerated rate by defining sub-tasks to perform each unique operation and associating objects with those sub-tasks. The system provides a flexible interface for defining objects and handles all communication for the programmer, eliminating many of the obstacles found in programming with explicit message passing. The MPI/Workpool brings a work pool-like system to the distributed environment, with low overhead and high flexibility.

The MPI/Workpool implementation is freely available online along with further documentation at [10].

References

1. *SAL - parallel computing - communication libraries*. (n.d.). Retrieved March 5, 2003, from Scientific Applications on Linux Web site: http://sal.kachinatech.com/C/2/

2. Snir, M., Otto, S., Huss-Ledernam, S., Walker, D., & Dongarra, J. (1998). *MPI – The complete reference: Volume 1, the MPI core, second edition*. Cambridge, MA: The MIT Press.

3. Stein, J. L. (2002, October 19). *Modeling, analysis and control of dynamic systems*. Retrieved March 6, 2003, from http://www-personal.engin.umich.edu/~stein /ME360/ps10.pdf

4. Khapaev, M. M. (n.d.). *Variable accuracy 3D inductance and capacitance extraction*. Retrieved March 6, 2003 from Moscow State University, Faculty of Computational Mathematics and Cybernetics Web site: http://bmik.ru/vm /sotr/vmhap/ind_cap.html

5. Tarko, A. M. (1999). *Modeling the global carbon and nitrogen cycle in atmosphere - ocean system*. Retrieved March 6, 2003, from http://www.ccas.ru/~tarko /ocean_e.htm

6. Ding, H. Q., Chan, C., Gennery, D. B., & Ferraro, R. D. (1999, March 1). *Parallel climate data assimilation PSAS package achieves 18 gigaflops on the 512-node Intel Paragon*. Retrieved March 6, 2003, from http://www.cacr.caltech.edu /Publications/annreps/annrep95/earth2.html

7. Dongarra, J. D., Duff, I. S., Sorensen, D. C., & van der Vorst, H. A. (1998). *Numerical linear algebra for high-performance computers*. Philadelphia: SIAM Publications.

8. Hwang, K. & Zhiwei, Xu. (1998). *Scalable parallel computing*. Boston: McGraw-Hill.

9. Carriero, N., & Gelernter, D. (1990). *How to write parallel programs: A first course*. Cambridge, MA: The MIT Press.

10. Sachs, N. (2003). *MPI/Workpool online documentation*. Retrieved March 16, 2003, from http://www.mcs.sdsmt.edu/~nsachs/mpiwp.html