

Parallel Programming Interfaces

Background

Different hardware architectures have led to fundamentally different ways parallel computers are programmed today. There are two basic architectures that general parallel computers fall under.

One class of parallel architectures presents a single unified address space to each processor. In these shared memory computers, usually the memory is physically distributed across the system but each processor is able to access any part of it through a single address space. The system hardware is responsible for presenting this abstraction to each processor. Communication between processors is done implicitly through normal memory load and store operations.

The second class of parallel architectures uses message passing as the primary means of communication. These computers usually have separate memory spaces and, in the case of clusters, are often made up of many individual single processor computers. Messages are sent from processor to processor through the network using software primitives.

These two architectures have led to two very different programming interfaces. A programming interface, in the context of this document, is the environment in which the programmer must work in. Often the programming interface reflects the hardware interface of the system. In shared memory computers, each processor has direct access to all data in memory. Such architectures have led to programming interfaces such as OpenMP where the programmer is able to use global data structures accessible from all processes. Message passing architectures have led to programming interfaces such as MPI where the programmer is able to explicitly send messages directly between processes.

Different programming interfaces offer the programmer a different set of tools. Often certain interfaces are very well suited for certain applications but impose limitations in others applications. Often these can be very fundamental limitations that restrict the programmer's ability to express the true algorithm that he wants to implement. In other cases, the limitations are less fundamental but are nevertheless very relevant. The programming interface can often force the programmer to deal with problems that are not related to the true algorithm and thus make it more difficult to solve the problem at hand.

In addition to programming ease issues, performance limitations can be imposed by the programming interface. Message passing interfaces require the software to explicitly send messages between the processors. Such software intervention can lead to very poor performance. For this reason, shared memory architectures generally have much lower processor-to-processor latencies than message passing architectures.

Synchronization

Synchronization between processors is one of the most difficult aspects of parallel programming. In message passing interfaces such as MPI, the programmer must ensure that communication is done correctly and at the appropriate time.

In shared memory interfaces such as OpenMP, such coordination for sends and receives is not necessary. However, there is the problem of atomicity. Often the ability to perform several memory operations atomically is required for correct program execution. Unlike message passing systems, achieving such atomicity in shared memory is a nontrivial task. Traditionally, a technique called locking has been used to solve this problem. A lock is a memory location that, by convention, protects a block of code that needs to be run atomically. Once a processor obtains the lock, that processor can execute the atomic block. All other processors wanting to execute that code must wait until the lock is released. The processor holding the lock must release it once it is done executing the atomic block. Consider the following bank account example:

```
1 Transfer (fromAccount, toAccount, amount) {  
2     if (fromAccount>amount) {  
3         fromAccount=fromAccount-amount  
4         toAccount=toAccount+amount  
5     }  
6 }
```

Figure 1 – This subroutine is designed to transfer money from one account to another if the source account has sufficient funds. This subroutine would clearly work as designed when running in serial. However, in shared memory parallel execution, another processor may also be running the same code on the same accounts at the same time. For example, it is easy to see how `fromAccount` may end up with a negative balance if another processor checks and changes the value of `fromAccount` after the current processor executes line 2 but before line 3.

```
1 Transfer (fromAccount, toAccount, amount) {  
2     LOCK (fromAccount, toAccount)  
3     if (fromAccount>amount) {  
4         fromAccount=fromAccount-amount  
5         toAccount=toAccount+amount  
6     }  
7     UNLOCK (fromAccount, toAccount)  
7 }
```

Figure 2 – With the locking instructions placed in the correct places, the account check and update is forced to be atomic.

Dealing with locks is a very difficult aspect of parallel programming in shared memory. Locking is not natural for programmers and thus can often lead to undesirable results such as deadlock when used incorrectly. Even the simple task of locking two independent objects can result in deadlock if done incorrectly. There are standard methods of avoiding such undesirable situations but they are often complex and difficult to reason about. Therefore, having to think about locks simply distracts the programmer from the actual algorithm being implemented. In addition, locks can potentially pose a significant performance penalty on the parallel code since lock are, after all, simply memory locations.

Transactional Memory

Conventional locking is not the most efficient means of achieving atomicity. Ideally, the programmer should have the ability to perform several memory operations atomically without having to worry about all the issues associated with locking. Transactional memory achieves this goal. Transactional memory is intended to replace conventional locking techniques to achieve higher performance and a more intuitive programming environment.

Transactional memory is a hardware mechanism that allows the programmer to define atomic regions (called transactions) containing memory accesses to multiple independent addresses. Instructions are added to the processor ISA to provide the programmer a method of defining these transactional regions. The hardware is responsible for ensuring that every transaction is executed atomically when viewed from the global memory system.

Transactional memory is an active research area that I am involved in independent of this project. However, from the programmer's perspective, transactional memory is simply an additional feature added to the shared memory programming interface.

Project Goals

The goal of this project is to compare the different parallel programming interfaces available to the programmer. MPI and OpenMP are the two main programming interfaces being studied. OpenMP with transactional memory will be evaluated as well. Both programming ease as well as performance will be considered in the evaluation. The goal is to understand the effect that the different interfaces have on the programmer and their ability to solve problems in parallel.

Methodology

The first step in the comparison is to choose an appropriate test algorithm as the benchmark. Since there are no standard benchmarks for this type of evaluation, an independent test algorithm must be chosen. This algorithm should be representative of large class of programs that are used in the "real world".

Once a test algorithm is chosen, it must be implemented in serial and then parallelized. Parallelizing the algorithm for both MPI and OpenMP should result in as few changes as possible. During the parallelizing processes, the programming ease aspect of the comparison can be made. Once the algorithm is parallelized, it can be run and tested for the performance comparisons. A 32 processor SGI Origin 2000 is available for the evaluation purposes.

Transactional memory comparisons can be made using a software simulator. A complete transactional memory system has been implemented the UVSIM software simulator in the past month. UVSIM is an execution-driven simulator that accurately models the MIPS R10k processor in a shared memory system similar to the SGI Origin 3000. It was developed by the University of Utah based on the RSIM simulator originally developed at Rice University and the University of Illinois at Urbana-Champaign. UVSIM can accurately run normal IRIX binaries. Therefore, it is possible to run the test code in the simulator for evaluation. This was unfortunately not done for this project. Please see details in the *Parallelizing for Transactional Memory* section.

The Push-Relabel Maximum Flow Algorithm

Finding the maximum flow of a graph has some very useful applications especially in the field of operations research. The push-relabel algorithm is currently the best solution to the maximum flow problem. It was originally presented as a serial algorithm by Goldberg and Tarjan in 1986. Recently, there has been some research into more complicated parallel implementations of the algorithm. The parallelization of the algorithm that will be used for this study will not use any of the recent complicated mechanisms. Rather, since programming ease is something that will be evaluated, the parallel implementation will only use some straightforward techniques discovered as part of this project.

The serial implementation of the algorithm is illustrated in figure 3.

```
while (active queue no empty) do {  
    Select node i from head of active queue  
    Push-relabel(i)  
}
```

Figure 3 – The push-relabel algorithm simply keeps track of the current “active” nodes in the graph. Active nodes are simply nodes that can supply flow. Each active node is used to pushed flow to adjacent nodes based on their labels. Then the active node and all adjacent nodes are relabeled. This is illustrated in figure 4.

```
Push-relabel(i) {  
    ...  
    if (excess[ i ] >= residual[ i, j ]) do {  
        excess[ i ] -= residual[ i, j ]  
        excess[ j ] += residual[ i, j ]  
    }  
    ...  
    If (node j is active) do {  
        relabel(j) and add it to end of active queue  
    }  
    ...  
}
```

Figure 4 – The push-relabel action on a particular node is simply a push of excess flow to adjacent nodes and a relabel of the current node.

More details of the algorithm can be obtained from the original Goldberg and Tarjan paper or most network optimization textbooks. Figure 3 and 4 are presented here to illustrate the main flow of the algorithm that are be relevant when parallelizing it.

Results

The following is a discussion of the parallelization process for the push-relabel algorithm. First, a serial version of the algorithm was implemented. In parallelization, the goal was to make as few changes as possible to the serial version so that sensible comparisons could be made. However, in some cases, many changes to the serial version were necessary just to get speedup.

Parallelizing for OpenMP

Shared memory parallel interfaces such as OpenMP have the very attractive advantage that parallelization of serial code is very simple. Since all of the data is visible by all processors, each processor can act on a piece of it in the same way as in the serial code. However, atomicity must be insured for correct program execution. Therefore, only two main changes were necessary on the serial code to ensure correct execution in parallel:

- 1) Atomic Queue Operations: Since a global queue is kept to store all the active nodes, it is necessary to ensure that two processors are not operating on the queue at the same time. Therefore, each addition and deletion operation from the queue was protected by a lock.
- 2) Atomic Flow Push: From figure 4, it is apparent that the pushing of excess flow from one node to another must be done atomically for the same reasons as the bank transfer case in figure 1. Therefore, each node needs to be locked before any flow can be pushed.

These two simple changes were sufficient to ensure correct program execution in parallel. However, as illustrated in figure 5, some more work needed to be done to obtain speedup.

Speedup vs. # of Processors

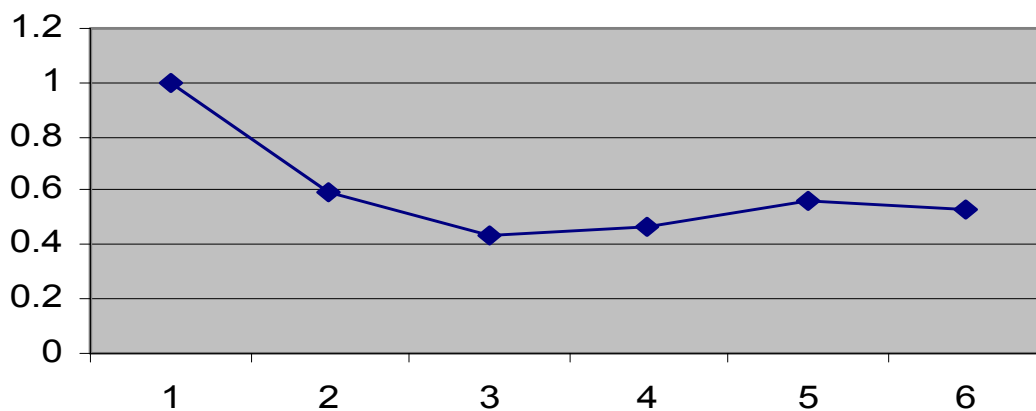


Figure 5 – Relative speedup of the first OpenMP version on a 400 node dense graph.

As shown in figure 5, simply translating the serial code directly as described above does not result in any speedup when run in parallel. After some investigation, it was discovered that

this “negative” speedup was caused by contention for the active queue. As in all parallel programs, if there is one central data structure that provides work to all the processors, there will be significant contention for that data structure. In the push-relabel algorithm, that data structure is the active node queue.

The following techniques were used to try to solve this problem:

- 1) Independent Graph Walk: It was noticed that the order in which the nodes are traversed while pushing does not affect the correctness of the algorithm. This was pointed out by Goldberg and Tarjan however a queue is generally used in practice since many recent studies have shown that a breadth-first-search type approach leads to better practical serial runtime. The theoretical time bound is the same in both cases. Therefore, much of the contention for the central queue can be alleviated by allowing each processor the ability to walk the graph in a depth-first-search manner when pushing. The central queue is used only when a relabel is required.
- 2) Simultaneous Queue Operations: I was noticed that real conflict between processors only occurs when two or more processors are working on the same node. Since operations are only on the head and the tail of the queue, generally conflicts between head and tail operations rarely occur. Therefore, with some clever locking, the entire queue does not need to be locked on every queue operation; only the head and tail do.
- 3) Independent Active Queues: Even though head and tail queue operations can be done in parallel, this still limits the number of processors operating on the queue to 2 (one on the head, one on the tail). When the number of processors is higher than 2, that limitation still creates a significant amount of contention for the active queue. Therefore, the active queue can be broken into independent small queues that are maintained by individual processors.
- 4) Random Arc Choice: In addition to contention for the central queue, results showed that there was also significant contention for the nodes. This was unexpected since the number of processors is generally much smaller than the size of the graph. However, some investigation showed that this was because the current implementation picks arcs in numerical order when traversing the graph. Therefore, all processors will be working on the low numbered nodes first and the higher numbered nodes later. This causes unnecessary contention. This problem is solved by simply choosing the arcs at random in the graph walk.

The above techniques proved to increase the speedup significantly but, as shown in figure 6, not sufficiently. No actual “positive” speedup was obtained even though significant improvements were made.

Speedup vs. # of Processors

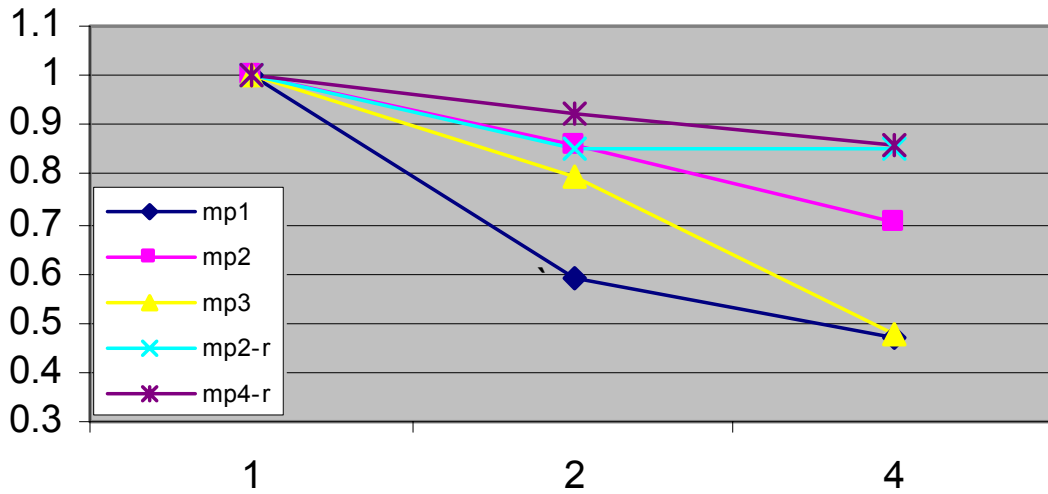


Figure 6 – Relative speedup obtained from various versions of the parallel implementation using the techniques described above. The input is a 400 node dense graph. mp1 – the original directly translated parallel version, mp2 – the mp1 version with independent graph walk, mp3 – the mp2 version with simultaneous queue operations, mp2-r – the mp2 version with random arc choice, mp4-r – the mp1 version with random arc choice and independent active queues.

To understand the reason that positive speedup was still not attainable, some investigation of the underlying hardware must be done. The SGI Origin 2000 (machine used for testing) uses a cache-coherency protocol that allows only one processor to “own” a specific cache line. This implies that even if two processors are working on different data, they may be causing a lot of cache-coherency communication if the data is on the same cache line. For node contention, this can easily be solved by simply make the problem size larger. Figure 7 illustrates results for larger problem sizes.

Larger problem sizes lowers to probability that two processors will be accessing independent data from the same cache line. However, run-time restrictions did not permit dense graphs as inputs. Therefore, another class of graphs, random level graphs, were used for all large input tests. Random level graphs are graphs that have many levels where each level is connected only to the level below it. Each node in is connected to only a few nodes in the level below it.

Speedup vs. # of Processors

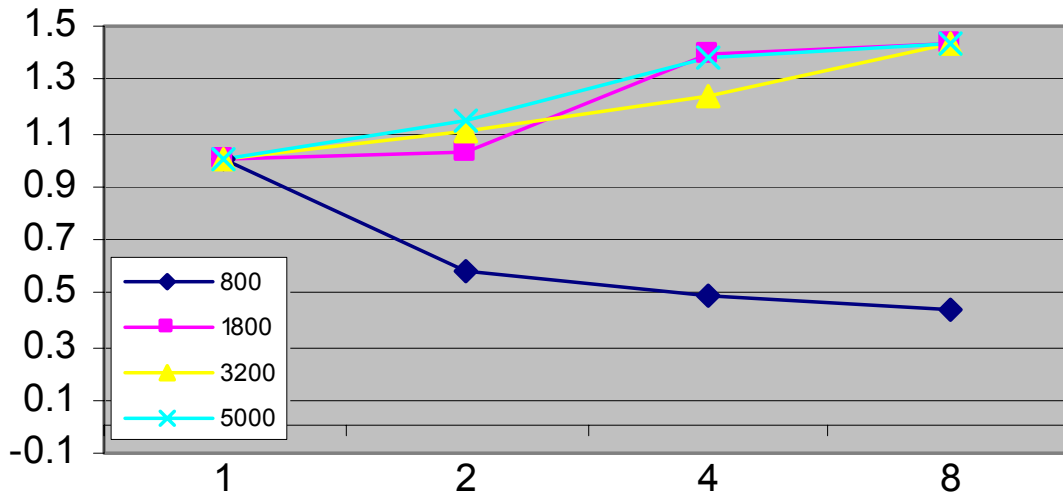


Figure 7 – Relative speedup of the best parallel implementation (mp4-rf) for various input sizes. The input is a random level graph with number of nodes varying. The random level graph has 2x1 dimensions (the width of each level is 2 times the number of levels) and has 3 connections from each node to the next level.

As illustrated in figure 7, a speedup of around 1.5x was obtained by simply increasing the problem size so that the cache is used more effectively. The same technique can be used to distribute the lock variables in memory as well. Lock variables also cause contention when being sent from processor to processor because of cache line coherency conflicts. However, this lock location optimization was not done since even the most optimized parallel implementations from recent research can only obtain a speedup of around 3-4x on 8 processors. Therefore, it was decided that a speedup of 1.5x was sufficient and close to optimal for the current parallel implementation.

Parallelizing for MPI

Unlike OpenMP, writing correct parallel code in MPI is not simple. Since the graph is distributed among the processors, the entire algorithm needed to be changed to accommodate the new location of the nodes. In addition, since walking a graph can result in a very irregular communication between the processors, thinking about a message passing coordination was very difficult. The pseudo-code in figure 8 illustrates the first attempt at a parallel MPI version of the push-relabel algorithm.

```
Repeat {  
    While (local active queue not empty) do {  
        Select node i from head of local active queue  
        Push-relabel(i)  
    }  
    Update node owners of changes  
} until there are no more active nodes
```

Figure 8 – The first implementation of the parallel push-relabel algorithm in MPI. All graph nodes and arcs are distributed among the processors.

Unfortunately, testing showed that the algorithm given in figure 8 does not produce the correct solution at all times. In all the test cases, the solution produced was only 1-2 units (of flow) off of the known correct solution. After much investigation, it was discovered that the algorithm in figure 8 is incorrect.

The problem is that the property of atomicity is not conserved. More than one processor may be working on one node at the same time. The atomicity problem is not the same as that in shared memory. Rather, the problem occurs when more than one processor performs relabels on the same node. The same node is changed in two different ways by the two different processors. The update step makes both of the changes visible which results in an inconsistent graph (a graph labeling that could not have occurred in the serial algorithm).

The atomicity problem can be solved once we notice that pushing flow can be done in parallel but relabeling cannot. Pushing can be done in parallel because the push operation can be applied in any order and the final excess flow is always the same. Relabels, on the other hand, determine which nodes the flow is pushed to and thus cannot be done in any order. Therefore, the algorithm can be made correct by simply doing all the relabels in serial.

However, a serial operation is very expensive in terms of performance. In addition, the use of MPI incurs a very large communication latency overhead that also adds to the “negative” speedup shown in figure 9.

Speedup vs. # of Processors

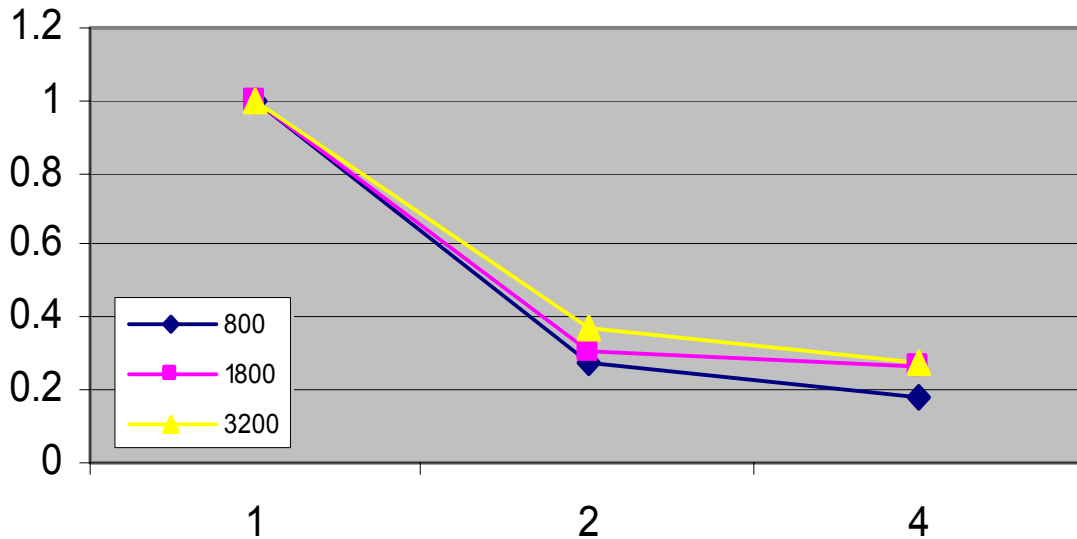


Figure 9 – Relative speedup of the correct MPI implementation on a random level graph with number of nodes as indicated.

Unfortunately, unlike the OpenMP version, techniques to increase the speedup of the MPI version are not apparent. The serial relabel step is necessary for correctness and the MPI library latency cannot be avoided. There are probably optimizations that can be made but none have occurred to me even after many days of thinking about the problem.

Parallelizing for Transactional Memory

Parallelizing for transactional memory is very similar to parallelizing for OpenMP since the serial code will work correctly as long as the correct atomic regions are defined. As in the OpenMP case, only the push/relabel of a node and active queue operations need to be done atomically. With transactional memory, these operations simply need to be placed within a transaction and atomicity is obtained through hardware.

Transactional memory implementations were completed but, unfortunately, not run. A complete working transactional memory system has been implemented in the simulator however additional technical difficulties arose. The simulator requires that all binaries be linked statically with any libraries. For other testing, these libraries have been obtained directly from SGI. Static libraries are no longer distributed by SGI to the general public. SGI staff had to build these libraries specifically for use with the simulator. Unfortunately, a few C++ libraries were not built when the original libraries were obtained a few months ago. All code for this project was written in C++ and required a few of the libraries that are currently missing. Over a week ago, these libraries were requested from SGI but they have not built them for us yet. Therefore, transactional memory speedup numbers are not available at this time. However, once the

libraries arrive, they can be simply obtained by running the transactional memory test code through the simulator.

Even though no specific performance numbers have been obtained, some qualitative speedup observations can be made. Since transactional memory only replaces locks in OpenMP, the speedup trend should look similar to that of OpenMP. In addition, since locks are not needed, all the overhead associated with locks is alleviated. Therefore, the rate of speedup should be somewhat higher than that of OpenMP with locks. However, the effects of transaction conflicts cannot be predicted from the simple OpenMP case.

Performance Comparisons

To understand some of the tradeoffs between the two programming interfaces a test was done to measure the observed memory bandwidth and latency of both interfaces. Figure 10 shows the results of this test.

Memory Bandwidth	[Mbits/s]
MPI Single	3.7
MPI Blocked	377
OpenMP	430
Memory Latency	[10⁻⁶ s]
MPI	232
OpenMP	5

Figure 10 – Observed bandwidth and latency between 2 processors on a 32 processor SGI Origin 2000.

In figure 10, *MPI Single* refers to sending data individually, word by word. There is a lot of overhead since `MPI_Send` needs to be called over and over. All MPI calls are simply library calls so there is a lot of software overhead required just to get the message out onto the network. *MPI Blocked*, on the other hand, is when `MPI_Send` is called only once with a large block of data. The bandwidth obtained with *MPI Blocked* is two orders of magnitude larger than *MPI Single* since the overhead of calling `MPI_Send` multiple times is not present. However, since MPI communication is still handled in software, there is some overhead associated with blocked sending that OpenMP does not suffer from. However, OpenMP truly dominates over MPI in communication latency. OpenMP communication is handled directly in hardware whereas MPI calls require quite a bit of software overhead.

The speed of communication is very important for both OpenMP as well as MPI codes. For OpenMP, it plays a large role because, in addition to normal communication, locks are simply memory locations. Access to memory is very costly in terms of performance even when the memory is local to the processor. This phenomenon can be seen in figure 11.

Code	[s]
Without locks	1.4
With locks	7.8

Figure 11 – Runtime for OpenMP code running on 1 processor with and without locks.

Obtaining a lock requires an atomic operation by the processor on a memory location. These atomic memory operations are built into the processor ISA for this specific purpose. However, most processor designs discourage their excessive use since they are very slow. In the case of maximum flow, locks are being obtained and released very frequently since each push requires two locks. This is the reason for the drastic performance decrease observed in figure 11.

As a note, transactional memory is designed to solve this exact problem. Since locks are most often not required (more than one processor is not working on the same node at the same time), speculative execution of the atomic sections can be done without violating atomicity. Transactional memory uses such a technique.

In addition to lock overhead, communication latency is a large source of overhead especially for non-embarrassingly parallel applications such as the push-relabel algorithm. Since there are many small sends between the processors (node updates), the latency of the communication becomes the bottleneck. The SGI Origin 2000 was designed to have low latency, even for MPI routines, when compared to conventional networks such as Ethernet. The importance of low latency really becomes apparent when we compare results from the SGI Origin 2000 and a machine using a conventional network. Figure 12 shows the speedup (negative of course) of the same MPI code from figure 9 on a cluster using Ethernet.

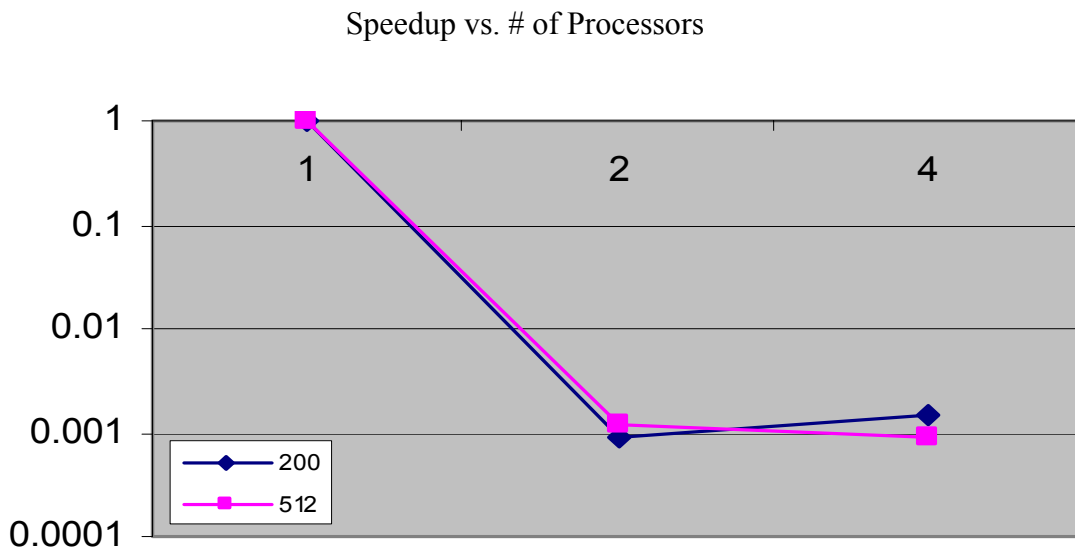


Figure 12 – Relative speedup of MPI code running on a Beowulf cluster with Gigabit Ethernet interconnect (cagfarm). The input was a random level graph with number of nodes as indicated.

Compared with figure 9, the cluster gives about a 200x relative slowdown even when running the same code. Note that the cluster uses a Gigabit Ethernet interconnect which has higher theoretical bandwidth than the SGI Origin interconnect. This implies that the latency is the limiting factor in the MPI runtime. Latency is a factor in the OpenMP codes as well but not a limiting factor since OpenMP latency is orders of magnitude smaller than MPI (as shown in figure 10). Figure 13 shows the runtime of both OpenMP and MPI.

Runtime (s) vs. # of processors

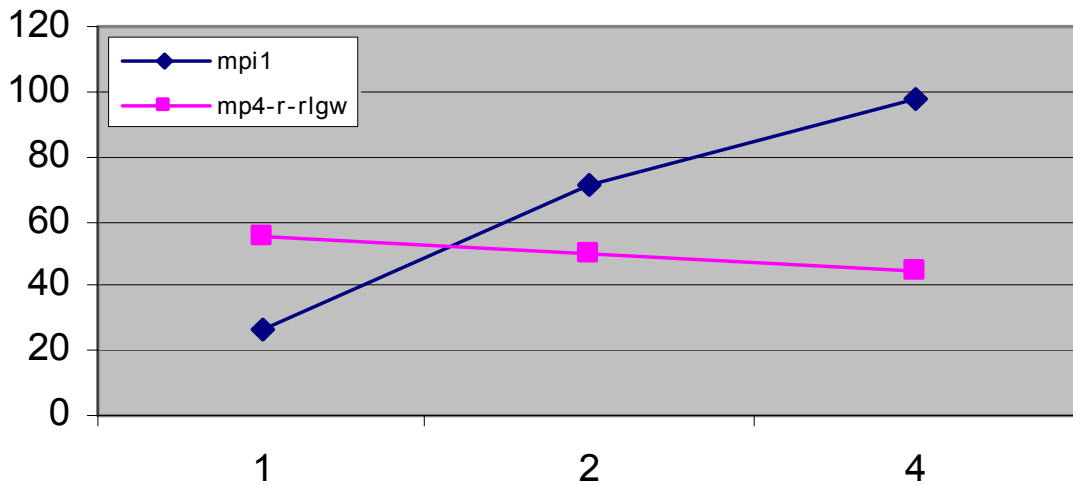


Figure 13 – Runtime of MPI and OpenMP versions of the push-relabel algorithm. Input is a random level graph with 3200 nodes.

As illustrated in figure 13, on one processor the lock overhead of OpenMP dominates the runtime and causes it to be slower than the MPI implementation. However, when the number of processors is increased, the network latency of MPI starts to dominate and, for that reason, results in an increase in runtime.

Unfortunately, no comparison with transactional memory was completed in time for this document. It is expected that the transactional memory implementation will give a curve that will be slightly lower than the OpenMP curve in figure 13 since lock overhead will be alleviated.

Programming Ease Comparisons

Since programming ease is not something that can be measured like performance, much of the following will be very qualitative and be based on my experiences in this project.

OpenMP

As mentioned briefly earlier, OpenMP offers a very simple way of parallelizing serial code. Though such a method results in correct program execution, it does not necessarily lead to speedup when run on more than one processor. Therefore, some work needs to be done to obtain a practical parallel version. However, as shown earlier, these modifications are often straight forward and involve very little change to the original algorithm.

The largest problem with OpenMP is that locks need to be used to achieve atomicity. Even in the push-relabel algorithm, lock management is a large issue. For example, when obtaining locks for multiple nodes during a push, the order in which the locks are obtained is very important. Incorrect ordering of locks can very quickly lead to deadlock (from experience).

In addition, often the lock ordering needs to be done in software since it is not predetermined. Thus, additional overhead beyond just obtaining and releasing the lock is often required to get it right.

Lock ordering, however, is a relatively simple problem compared with the other issues involved with locks. When adding the ability to perform simultaneous queue operations, for example, the locking mechanism was very difficult to reason about. Allowing two processors to operate on the queue at the same is a simple task. However, implementing separate locking for the head and the tail of the queue took many hours of thinking and debugging.

MPI

MPI is very difficult to work with since the parallel version of the code often bares no resemblance to the serial version. This is because MPI imposes a very awkward restriction on the programmer. The programmer must think about which pieces of the data each processor is responsible for and how that data will be needed by other processors. In addition, explicit coordination of messages between processors is required.

Since MPI has such a high latency overhead, obtaining any speedup is very difficult for codes, such as push-relabel, that do not have regular communication patterns. Unlike OpenMP, it is not always apparent how to obtain speedup since the message passing environment imposes such a constraint on how the algorithm is organized. In this project, the most difficult aspect of parallelizing the push-relabel code for MPI was simply getting it to run correctly. Once that was done, there were no apparent ways of making it run faster.

As figure 10 showed, MPI does not suffer a large performance penalty in terms of memory bandwidth when data is sent in large blocks. Though it may be possible to exploit this fact when writing code that uses memory in regular patterns and communicates at regular intervals (such as matrix codes), it is not general enough to support more pointer-based codes such as graph algorithms.

In addition, matrix codes are much easier to parallelize in MPI since the memory access patterns are very regular. In some sense, matrix codes are simply many embarrassingly parallel sections run at regular intervals. This property makes it easy to parallelize them using MPI. However, in the general case as illustrated by push-relabel, MPI is very difficult to work with.

Transactional Memory

Though no performance data was obtained for transactional memory, all the necessary code was written using transactional memory. Therefore, observations can be made about the programming experience in comparison to OpenMP and MPI.

Transactional memory is essentially OpenMP without locks. Therefore, it was very simple to parallelize serial code using transactional memory. Transactional memory made it very simple to define atomic regions since locks were no longer necessary. The most notable case was the simultaneous queue operation. As mentioned earlier, this addition was very difficult with locks but became trivial with transactions. Unfortunately, currently there is no performance data to show that transactions do not impose a performance penalty for this added convenience. However, such a tradeoff is very unlikely since transactions act exactly like normal memory operations when there are no conflicts and conflicts occur very infrequently.

Figure 14 shows the code length of the different versions of the algorithm. Code length gives an indication of the relative complexity of each version. In all cases, as much code re-use as possible was done. This was done to ensure that a sensible comparison could be made. The general algorithm never changed. All changes and additions were made to accommodate the programming interface and to achieve performance as indicated in previous sections.

Version	Length (c++ lines)
Serial	447
OpenMP	588
MPI	840
Transactional Memory	571

Conclusions

OpenMP is easier to program with than MPI. Though it is possible to parallelize very regular code in MPI, it is not general enough to allow for easy parallelization of pointer-based algorithms. In addition, OpenMP offers a significant performance advantage over MPI since MPI relies on software for communication. In cases where communication is very irregular and not in large blocks, the MPI performance penalty can be significant. The only drawback of OpenMP is locks. Locks are not desired since both programming ease and performance suffer. Transactional memory promises to solve that problem. The programming ease aspect was shown to be true in even the simple case of accessing a queue in parallel. Unfortunately, no performance results were obtained as part of this project. However, it is very likely that the performance of transactions will exceed OpenMP.

Future Work

Transactional memory is an active research area that I am involved with outside of this project. I am working primarily on the hardware architecture of transactional memory. The performance results that I was not able to obtain for this project are very important in the design of the hardware architecture. I plan to have some performance figures in the next week or two (whenever SGI gets back to me). Once it is possible to run codes such as the parallel push-relabel through the simulator, the design tradeoffs of a transactional memory system will become clearer. Then, work on optimizing the hardware architecture can be done. Work on transactional memory is related to my Masters thesis in the department of Electrical Engineering and Computer Science.

References

Anderson, R. and Setubal, J. C. [1995]. *A Parallel Implementation of the Push-Relabel Algorithm for the Maximum Flow Problem*, Journal of Parallel and Distributed Computing, Volume 29(1), 17-26.

Goldberg, A. V. and Tarjan, R. E. [1988]. *A New Approach to the Maximum Flow Problem*, Journal of the ACM, Volume 35(4), 921-940.

Kuszmaul, B. C. [2003]. Conversations, MIT LCS, Cambridge, MA.

Lie, S. [2003]. *Transactional Memory*, MEng Thesis Proposal, MIT EECS, Cambridge, MA.