BALANCING PRIORITIES AND LOAD FOR STATE SPACE SEARCH ON LARGE PARALLEL MACHINES

BY

JONATHAN ALLEN BOOTH

B.S., University of Illinois at Urbana-Champaign, 2000

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

To my parents

Acknowledgments

I'd like to extend a huge thanks to the following people, listed in no particular order: Kathleen, for providing a reality check while I was writing my thesis, Orion for some messy memory-leak debugging help in the charm core, Prof. Kale for his help and direction, my parents for encouraging me and all the help they've given me during my entire college career. I'd also like to thank my friends and lab members who've encouraged and put up with my babbling about my program and problem.

Table of Contents

Chapter 1

Introduction

Parallel applications require load balance to achieve good performance. Many regular, iterative applications can be statically or dynamically load balanced by moving a small bit of the domain from one processor to another. Other applications, such as searches, don't operate over a regular domain. These applications must still have their work balanced to achieve good performance.

Getting a high megaops rating out of a parallel search is easy. Getting a fast and optimal solution out is not so easy. Without load balancing, combinatorial computational nodes tend to waste time on less promising work or sit idle.

This paper presents and compares the effectiveness of a variety of load balancing strategies for search applications, using fifteen puzzle[1] as the example case.

1.1 State Space Search

State space search describes a class of problems with four main features; a start state, intermediate states, a goal state, and a set of rules for transitioning between states. The search proceeds from the start state, using the rules to transition to and through intermediate states, finally arriving at the goal state.

In order to speed up searches, heuristics[2] have been developed for many state space search problems that can be used to guide the search towards more promising branches of the search tree. Using these heuristics in a serial program is an easy way to achieve a faster search; sort all possible states that can be reached from the current state by how promising their heuristic results are and

[1] Described in Chapter 2

[2] Fast but simplistic methods of determining the quality of a state

search the best first. Even with fairly nieve heuristics, this often allows much of the search space to be skipped, leading to a far faster time to solution.

Parallelizing and achieving good speedup for simplistic state space search algorithms, such as a full tree search without a heuristic is easy. Work only need be balanced either statically or dynamically so no processor falls idle.

Good speedup becomes a more elusive goal in a parallel search using a heuristic. It is easy enough to sort each processor's work, but that will not balance work across different processors. One processor may have a large amount of work while another processor sits idle.

Balancing work between processors so that none sit idle will alleviate this problem but can introduce a worse one; one processor may have all the promising work, while the rest of the processors in the system waste time on less promising work. These wasted processors may cause an even worse problem by injecting false-positives[3] into the system.

A new type of load balancing, one that can balance not only the quantity of work, but also the quality of work across all the processors is needed.

1.2 Seed Balancing

Seeds are simply states in a state space search that are waiting in work queues. They were generated by the application of the problem's rules on a parent state, but they have not yet been tested to see if they are the goal state, nor have they had the rule set applied to them to generate new states.

Seed balancing refers to load balancing these seeds across processors so that every processor has work to do. In addition, seeds have priorities assigned to them via the searches HERE function, representing the seed's predicted importance. Seed balancing also strives to balance fairly the priority of seeds over all processors.

Random seed balancing is the simplest method. Each seed created is assigned to a random processor in the system. Unfortunately random seed balancing has no consideration for the priority of a seed. Nor does it provide any guarantee every processor has work to do.

This is not sufficient, especially for large search problems or large parallel machines. Large machines with small problems tend to have processors sit idle as the random distribution of seeds

[3] Work that seems promising but doesn't lead to a solution

fails to evenly spread out work. Large search problems can easily focus all the important work on one processor.

New balancers that concern themselves with balancing the amount and priority of seeds across processors must be used instead. Three such possibilities, explained in detail in Chapter 3, are an all-to-all balancer, a dense graph balancer, and lastly a distributed master-slave balancer. These balancers will take some computation time away from the search in order to do the seed balancing, but by keeping all processors working on high priority work, they will result in a solution being found faster as in the serial case.

1.3 Related Work

The Parallel Programming Lab at the University of Illinois has previously researched into methods similar to those described in this paper for seed balancing for smaller numbers of processors.

A object creation balancing strategy[7] was investigated in 1992 by L. Kale and A. Sinha. This work considered only up to five hundred processors however. Some of the balancers in this paper have been adapted from the idea of the multiple masters with tokens in that paper. The tokens idea has however been dropped. Given the costs to initiate communication are much larger than the per byte cost, there is little need to send a token that will be only slightly smaller than the full problem instance itself. In particular, in the fifteen puzzle problem explored in this paper, the problem only occupies on the order of 32 bytes.

T. Marsland and A. Reinefeld surveyed[5] the field with an extensive paper on both serial state space search algorithms and parallel implementations of A*, IDA*. The methods for balancing load they used were centralized sorted data structures, appropriate to architectures of the time with shared memories and tens of processors.

Effective use of prioritized parallel task execution was demonstrated[2] by L. Kale et al in 1995. However the method used there was method invocations rather than new object creations which runs through a different load balancing mechanism in the Charm++ system.

1.4 Thesis Outline

Chapter 2 gives an overview of the general state space search problem, and its implementation in Charm++, as well as a short overview of Charm++. Chapter 3 describes the Charm++ seed balancer system and the seed balancing strategies tested. In Chapter 4 the results of the seed balancers run against the sample search applications are compared. Chapter 5 concludes the paper and gives suggestions for future work.

Chapter 2

State Space Search

Many famous problems in computer science are state space search problems. The traveling salesman problem, chess, scheduling, and Othello are all members of this class. The tree-search structure of these computations lends itself to an easy implementation in Charm++.

2.1 Charm++ System

The software used to implement and test these methods of seed balancing is Charm++ [3]. Charm++ is a portable parallel runtime system for C++. Charm++ was developed to run on distributed or shared memory machines. It targets any machine, from a laptop to a cluster of idle desktop workstations to a supercomputer.

A Charm++ program consists of multiple C++ class objects (called chares) that exist and run in parallel with each other on all the processors of the system. Each object is assigned to a single processor and communicates with the other objects in the system via message passing.

Message passing in Charm++ is as simple as regular C++ method invocation. A special proxy to a remote object is defined by Charm, which the calling object invokes methods on as usual. The key difference from normal C++ method invocation is that methods invoked in this manner are asynchronous: they can execute along side the caller, and do not return a result.

In a typical Charm++ application, there will be many more chares than processors in the system. This independence between the number of processors and the number of processes in a system is called processor virtualization. Virtualization [1] allows us to overlap computation and communication.

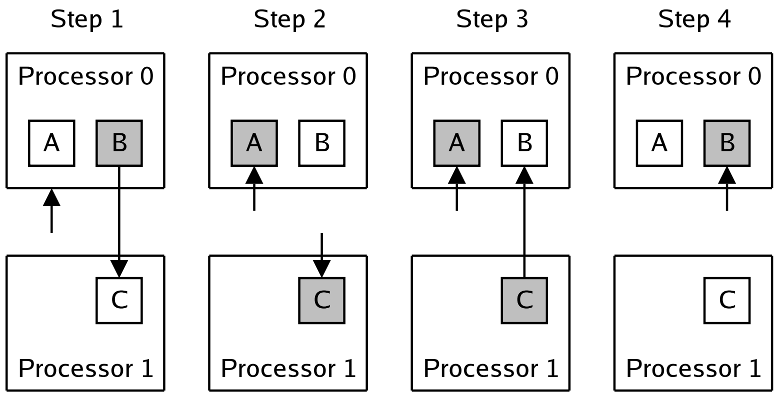

Figure 2.1: Message passing keeps processors busy while waiting on network delays.

For example, consider figure 2.1. Let there be 3 objects in the system, A and B on processor 0; C on processor 1. Let A be waiting to run and B be running. B sends a message to C requesting data (Step 1). B then falls idle, waiting for C to reply. Since A has a message to process, it runs (Step 2). In the meantime C receives B's message and replies with the data (Step 3). When A finishes executing, B then processes C's reply (Step 4).

This overlap allows Charm++ programs to tolerate network latency. In the case of state space search, where there is not a lot of communication anyway, this overlap will allow seed balancing's large amount of communication to overlap with the search progressing. In particular, the proposed all-to-all seed balancer presented in Chapter 4 will take heavy advantage of this overlap as it requires the most communication. Once the seed balancer finishes setting up the sends of work to other processors, it can pick up doing the work it still has. The seed balancer does not block on the sends, so the time to setup the communication is all that is lost.

2.2 Search Library

The test application used in this paper, fifteen puzzle, could have been written as a single monolithic application. However, many of the components in it are reusable by other search applications. These components have been collected into a search library.

The search library is distributed as part of the Charm++ system. It is built on top of charm, and hides the complexity of the algorithms implemented in it from the user. All the user needs to know to use it is how to create a class that describes their problem, and how the search library will pass results back to them. The library currently supports A*[1] and iterative-deepening A* searches.

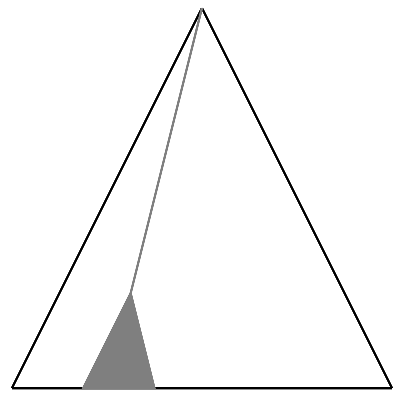

Figure 2.2: A* search sweep, high priority nodes on the left

An A* search tree involves a broom sweep of the tree, from the left to the right of the tree. Left nodes are the highest priority nodes, right nodes are the lowest priority nodes. A sample sweep [4] is shown in 2.2. Active nodes – those that are being computed right now – is the broom in the middle. Nodes to the left of that are inactive; they have been searched and did not contain the solution. Nodes to the right of it are yet to be searched.

One of the goals of A* search is to keep the memory consumed by the search small. For example, consider a problem where each state has two unique child states, where the solution state occurs after 51 steps. A breadth-first-search of this problem would have to hold at least $2^5$0 states in memory before the solution is found. If each state somehow only occupied a byte of memory, that would require a petabyte of memory to run. A* in contrast searches depth-first, requiring closer to $2 * 50$ states in memory at once.

A parallel implementation of A* that does not evenly balance the high-priority work fails the left to right sweep property of the search. Without balancing, processors that initially received low-priority work will spawn more and more low priority work. This produces a search that looks more like a breadth-first-search like as the load imbalance increases.

Unfortunately, A* searches do not always perform well. Finding a solution using heuristic-guided searching is fast, but finding the optimal solution may not be so fast. Figure 2.3 shows

[1] A depth bounded depth-first search

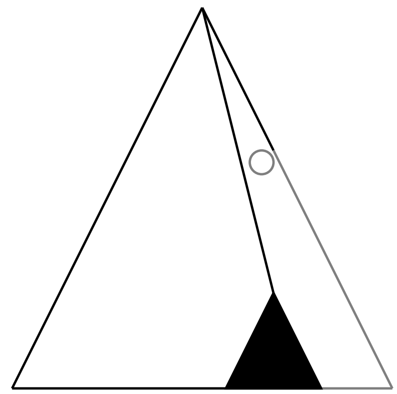

Figure 2.3: A* search wastes time

how a solution can hide in the low-heuristic side of the search tree, forcing the search to do far more searching than it needs to. In the worst case, if the problem being searched is searching to an unbounded depth, A* may never find the solution.

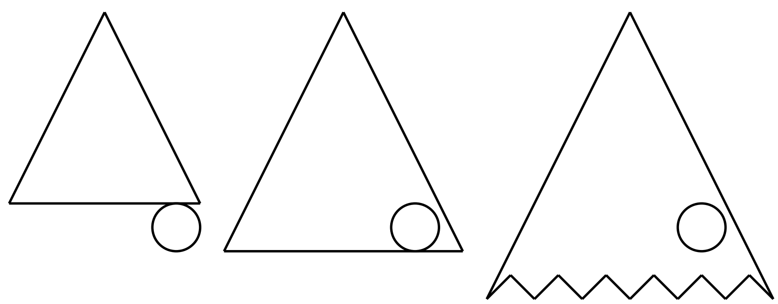

Figure 2.4: IDA* search

One way to fix this is with an IDA* search. An IDA* search is based on the following variation of the A* search. Rather than start searching at some large maximal depth, search smaller depths first. This approach is shown in figure 2.4. IDA* ends up doing some duplicate work because the higher levels of the tree are not saved between iterations[2]. The overhead of this extra work is offset by the savings of not searching too far past the solution. Once a solution is found, any node of greater depth is abandoned. To guarantee the optimality of the solution found, nodes of lesser depth continue to be searched until they are exhausted.

IDA* also avoids the ramp-up and trail-off periods an A* search computation experiences. To do this, different depth A* searches are run in parallel, with the lowest depth search having the

[2]They cannot be saved, in deep enough searches as the number of nodes could outgrow the memory of the machine

highest priority. Because there is now more work available in the system, when one A* search is winding down, another can start its computation and keep processors busy.

Internally the library is built on the Charm++ concept of chares and groups. A chare is a single object. Multiple chares run in parallel with each other on the multiple processors in a system. A group is a collection of chares where there is exactly one chare of that type on every processor in the system. The library also defines an abstract 'problem' class that the user will derive their particular problem class from.

A group is used for communication (solution updates for pruning in the tree) and keeping track of nodes and chares expanded. The group also performs a per-search quiescence detection to track search completion.

Chares hold the nodes of the search tree. Each chare contains at least one instance of the problem class. At creation, a chare makes a decision; should it spawn children (parallel-phase), or should it search the rest of its subtree by itself (serial-phase). A chare also checks to see if a solution has already been found that would invalidate it's existence, using the group overseeing the computation.

When a node in the tree is evaluated it is counted and reported to the group. If a chare finds a solution it also reports this to the group. Any children to be spawned are passed to the group as well, before the chare exits and deletes itself.

Once the communication group detects the search has finished, it reports the best solution (if any) back to the users and deletes itself.

2.3 Fifteen Puzzle

Fifteen puzzle is a classic state space search problem. A grid of tiles numbered from zero to fourteen is arranged in a four by four square. The last spot is left blank.

The rules of fifteen puzzle are simple. Take a given configuration and turn it into one that reads from zero to fourteen, in order. Valid moves are to slide any tile next to the hole into the hole slot. Shorter path solutions are considered of higher quality.

The heuristic used for this implementation of fifteen puzzle is the minimum (Manhattan) distance from solution. This is a guaranteed lower bound on the actual distance to solution. It can

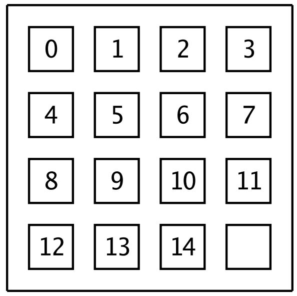

Figure 2.5: Fifteen Puzzle (Solution State)

be computed by summing over all tiles how far that tile must move to be in its solution position.

```
// column(tile) gives the current column that tile is in
// row(tile) gives the current row that tile is in.
int distToSoln = 0;
for ( int tile = 0 ; tile < 15 ; tile++ ) {
  distToSoln += abs(column(tile) - tile%4);
  distToSoln += abs(row(tile) - tile/4);
}
```

This heuristic of course ignores some situations, such as where two tiles direct route to their solution position requires moving through each other. It also has no concept of impossible puzzles that can occur when two tiles are flipped, such as the one shown in figure 2.6.

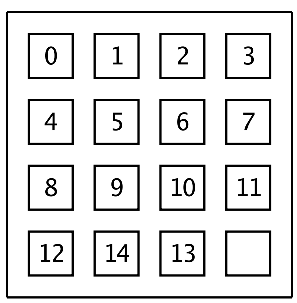

Figure 2.6: An Impossible Fifteen Puzzle

This particular implementation of fifteen puzzle has some modifications added for efficiency. The distance to solution is not computed every time it is needed, rather it is computed only once and then updated when a tile is moved (by +1 or -1 as appropriate). The old position of the hole is remembered so a child will not have its parent state as one of its children.

Chapter 3

Load Balancing

There are two essential components to good performance for a parallel search tree application. The first is to make sure every processor has work it can do so that no processors sit idle, known as the starvation problem. The second is to make sure every processor is doing useful work, known as the low-priority problem.

Accomplishing this via a central queue is possible for problems with large enough grain-size. For smaller grain-size problems, the central queue becomes a bottleneck and slows down the computation.

3.1 Charm++ Seed Balancing

Seed balancing in Charm++ is accomplished through a Converse load balancer module. Converse is the underlying communication layer for Charm++. Any Charm++ program can make use of such a seed balancer, but not all seed balancers are appropriate to all programs. The seed balancer for a program is chosen at compile-time, as a link option.

3.2 Strategies

Four strategies are explored in depth in this paper.

- Random
- All-to-all
- Graph

- Master-Slave

Random is the simplest strategy, for each piece of work assign it to a random processor. All-to-all balancing is conceptually simple; it essentially joins all the seeds into one list which is then divided among processors. "Essentially," because the described method implies a serial bottleneck. Instead a distributed method of splitting each processor's list and sending those smaller splits out to the other processors is used.

There is master-slave balancing, which is the most complex strategy relying on a set of masters to balance load among themselves and a set of slaves attached to each master. Graph based balancing is a special, simpler case of master-slave where there are no slaves, only masters.

All of the following strategies have the ability to overlap their communication with computation. All of the strategies reserve some work for themselves when balancing. Once the processor has initialized communication, it is then free to process the work that was not transferred.

3.2.1 Random

Random has been included as a baseline strategy. It should solve the starvation problem given the size of puzzles investigated in this paper. It does not address the low-priority problem as it does not consider the priority of a chare at all.

3.2.2 All-to-All

All-to-all balancing represents the most communication intensive seed balancer of the set. In all-to-all balancing, every processor splits its seeds into P sets[1], where P is the number of processors in the system. Each processor then broadcasts the i'th set to the i'th processor, transferring those seeds to that processor. This balancing strategy is guaranteed to solve the issue of starvation, as well as the low-priority problem. Any processor that happened to have high priority work will send each processor at least one high priority work instance to work on.

The drawback of this method is the all-to-all communication takes a long time to finish running. With modern, DMA enabled network cards, the processor time taken is very short. Due to this and Charm++'s ability to overlap communication and computation, the time unused by the processor

[1] With a cap on how many seeds it will send to each processor so it doesn't flood the network

is not lost. Instead, after a small time spent setting up communication, the processor will then resume handling messages waiting in its queue while the network card finishes sending and receiving load balancing messages.

A motivation behind all-to-all balancing is Charm++'s ability to overlap computation and communication. The delay incurred by sending many nodes is hidden because the processors still have work left to do while the communication completes.

3.2.3 Graph

Graph based balancing works by generating a dense graph over all the nodes. In particular, I chose to use a graph of degree $log_2(processors)$. Such a graph is dense enough to have a small diameter[2] without being communication intensive. The graph is a directed graph, and does not guarantee communication both ways between any pair of nodes. Edges in the graph denote the flow of seed balancing from one processor to another.

This balancing strategy is guaranteed to solve the issue of starvation, as well as the low-priority problem. The drawback of this method is that if the diameter of the graph is large it can take a long time for a pool of high-priority work to propagate out from an overloaded processor.

A processor in a graph based balancer chooses the nodes to send its work to by using the sequence of powers of two, modulo the number of processors. This will give it a degree of $log_2(processors)$ and help to keep the diameter of the graph small. For example, processor 3 out of 10 would choose neighbors of 4, 5, 7, and 1 (11).

3.2.4 Master-Slave

Master-slave [7] balancing is a blend of centralized and distributed strategies. It bases off graph balancing; a small set of nodes is setup in a graph balancer manner. These nodes are designated as local master nodes and each local master receives a set of slave nodes that query the master for work when they need it.

This strategy retains some of the advantages of a serial run; namely that work is processed in priority order within a local master. If the number of slaves per local master is adjusted to a

[2]So work is distributed quickly

reasonable number the master won't become a bottleneck such as would occur in a fully centralized strategy.

Master-slave balancing is guaranteed to solve the issue of starvation among masters, though slaves sometimes can starve for data if they are waiting for a master to reply with more work. It also solves the low-priority problem. This strategy has less of an issue with the diameter of the graph being large due to the smaller number of total nodes in the graph.

The drawback to master-slave balancing is that it is the most complex strategy, requiring distributed and centralized balancing at the same time. Any slowdown that occurs on master nodes will also effect all their slave nodes as it impedes their ability to acquire new work.

Chapter 4

Experimental Results

Experimental results were gathered for two machines, one small and the other large.

A linux cluster called Cool is the small test case. It is a 32 node Intel Xeon 550 Mhz cluster, with four processors and half a gigabyte of memory per node. Cool's interconnect is switched 100 Mbit ethernet.

The large machine is the Lemieux [6] machine at Pittsburgh Supercomputing Center. It is a 750 node Compaq Alphaserver ES45, with four processors and four gigabytes of memory per node. It uses Quadrics' elan interconnect network to link the nodes together.

4.1 Method

The fifteen puzzle test program, as well as the search library and Charm++ were all compiled with gcc on Cool, and cxx on Lemieux. All non-data collection debugging output was turned off in order to minimize its impact on network performance during the tests.

The balancers were implemented with buffered-sends. Buffers were flushed at 64 seeds or at the end of the load balancing step, whichever comes first.

Tests on Cool were monitored to insure no other user's jobs interfered with the test. Processors were allocated evenly across the eight nodes of the Cool cluster.

As Lemieux only allocates one job to a node, there was no need to monitor for other users interfering with the computation. Processors were allocated to fill nodes, rather than evenly like Cool.

Times reported exclude program startup time. The fifteen puzzle test program records wall clock time at program start and at the time the search library returns the solution.

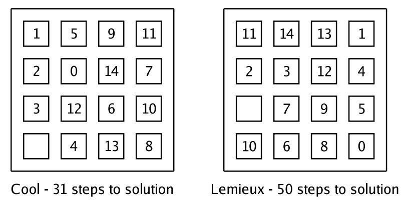

Figure 4.1: The two test fifteen puzzle configurations

The test problems used are shown in figure 4.1. Cool's test puzzle is much smaller than Lemieux's due to the massive difference in the computational power of the two machines.

4.2 Results

The results on Cool showed using anything other than the random seed balancer is a good idea. Lemieux's results further refined this to using the simplest and least network intensive balancer, the graph balancer.

Speedups shown are versus the smallest number of processors the test was run on. For Cool, this means speedups are relative to 4 processors. For Lemieux, this means speedups are relative to 128 processors. This was done due to the excessive run times larger problems would have had on a single processor on Lemieux. For internal consistency Cool was handled the same way.

4.2.1 Cool

Cool had all the non-random balancers perform similarly as can been seen in figure 4.2, with graph balancing being slightly faster than the other non-random balancers. Random performed poorly, which is not surprising but is useful to have shown experimentally. The exact run times for the balancers can be seen in table 4.1.

The only unexpected entry in figure 4.2 is the master-slave balancer running on 8 processors. This is due to a special case occurring on 4 processors. There are 3 slaves assigned to each master processor, so when master-slave runs on only 4 processors, it has only one master and balances load more evenly.

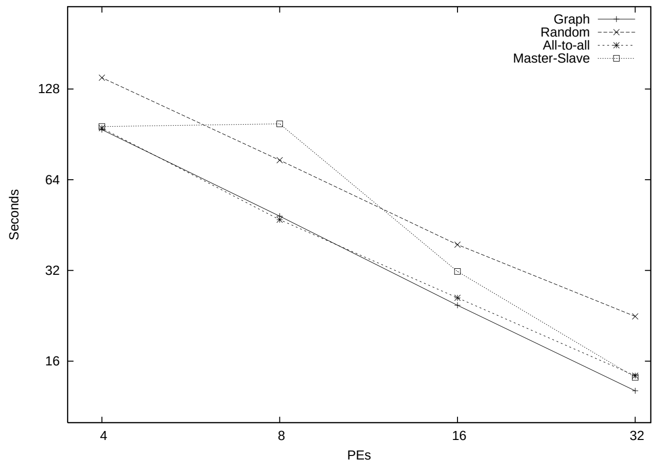

Figure 4.2: Run Times on Cool Cluster, problem D=31; log-log scale

	Processors			
	4	8	16	32
All-to-all	94.86	47.13	25.95	14.32
Graph	93.81	48.45	24.51	12.76
Master-Slave	96.02	98.07	31.75	14.14
Random	139.55	74.28	38.97	22.51

Table 4.1: Run times for balancers on the Cool cluster

The speedup shown in figure 4.3 is not perfect, things go well until past 16 processors. Part of this is due to computation ramp-up and ramp-down effects. When the computation starts, there is only one seed on the first processor. No special logic was implemented to help the balancers quickly get a lot of seeds out there so all processors can get working. This problem gets worse with more processors, as they sit idle waiting for the system as a whole to contain enough work to feed all the processors.

This effect favors balancers that can quickly get work out to a large number of nodes. This is obviously true for all-to-all and random, slightly less so for graph and master-slave. This becomes even more pronounced on larger machines.

4.2.2 Lemieux

The performance of four of the balancers on Lemieux is shown in figure 4.4. The scalability of the graph balancer is good up to 1024 processors, then it shows a slight drop-off in scalability. Other

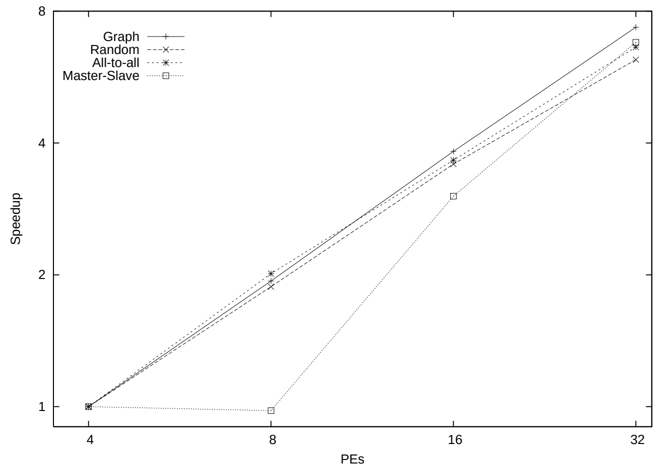

Figure 4.3: Speedup on Cool Cluster, problem D=31; log-log scale

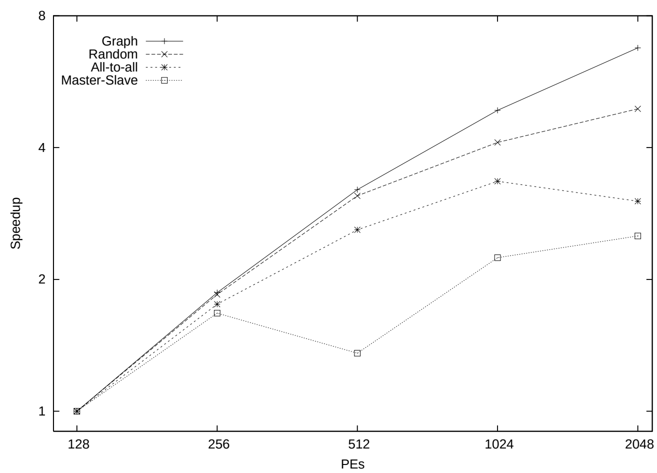

Figure 4.4: Speedup on Lemieux, problem D=50; log-log scale

balancers that don't achieve a good load-balance of work and scale poorly past 512 processors.

Speedups cannot tell the whole story however, unless the run times for processors are also known. As figure 4.5 shows, not only does the graph based load balancer have the best scalability, but it also runs in less than half the time of other balancers[1]. This is a result of the good load balance of the graph based balancer, and also the simplicity of the graph based balancer. It uses little processor time and network bandwidth and so runs faster than other more complex or resources-intensive balancers.

[1] See table 4.2 for exact run times

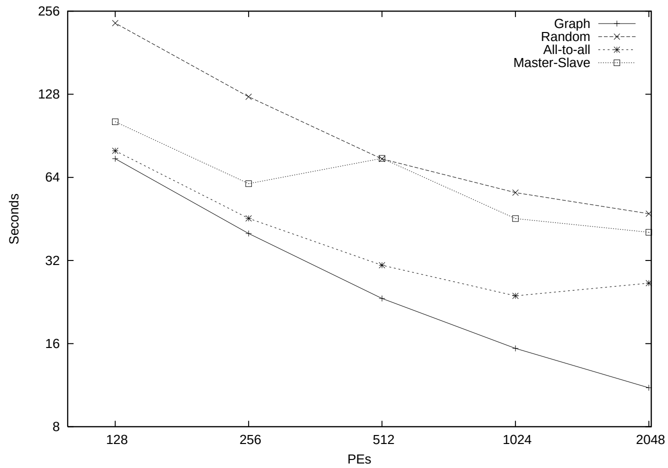

Figure 4.5: Run Times on Lemieux, problem D=50; log-log scale

	Processors				
	128	256	512	1024	2048
All-to-all	79.88	45.48	30.77	23.83	26.48
Graph	74.79	40.07	23.32	15.37	11.07
Master-Slave	101.77	60.74	75.00	15.37	11.07
Random	231.79	125.41	74.67	56.38	47.26

Table 4.2: Run times for balancers on Lemieux

4.2.3 Speculative Overhead

A known problem of the IDA* method[4] causes it to scale poorly when it runs many iterations. Running multiple iterations in parallel can alleviate this, but still will not totally eliminate the problem on a machine as large as Lemieux. Another method of assessing the quality of a load balancer is necessary.

The quality of a load balancer can be seen by examining the number of extra "useless" nodes that are run. This number should be proportional to the number of processors in the system, and should not be widely variant [2].

Instrumentation was added to the code to record the number of nodes past the solution. This can easily be tracked: solve the problem to figure out the priority of the solution node P, then compare each node's priority with that priority. Any node with a priority higher than P is "wasted" work.

As you can see in figure 4.6, overhead does increase with the number of processors for the graph

[2] Widely variant overhead indicates a balancer will not produce consistent speedups.

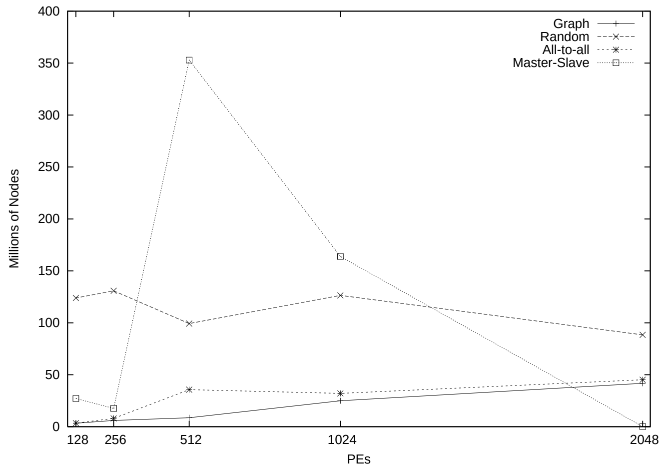

Figure 4.6: Overhead of seed balancers on Lemieux, problem D=50, solution requires approx 760 million nodes to be expanded

and all-to-all load balancers. This is as expected; when the work load is too small to effectively load balance then all nodes available will be evaluated, be they low or high priority.

Figure 4.6 also shows overhead is bounded versus the number of processors for the graph and all-to-all based balancers. Speculative overhead is not growing exponentially and causing the scaling problems. This shows the graph and all-to-all load balancers are generating good balance across all processors.

The random balancer actually has a smaller overhead on larger numbers of processors. With more processors available, it's less likely that one processors will get overloaded with high-priority work (including the solution). Less chance of overloaded processors means the under-loaded processors have less time to waste expanding useless nodes. The master-slave balancer also is inconsistent, due to the way managers and slaves interact in it.

4.2.4 Network Bandwidth

Analysis was also done to insure that it was not network overload causing the poor scalability. Figure 4.7 shows that the random load balancer is having a problem with network congestion while the rest are not.

Obviously part of the problem with the random balancer is that it is sending orders of magnitude more nodes around than the other balancers. This is to be expected; if all the high priority work

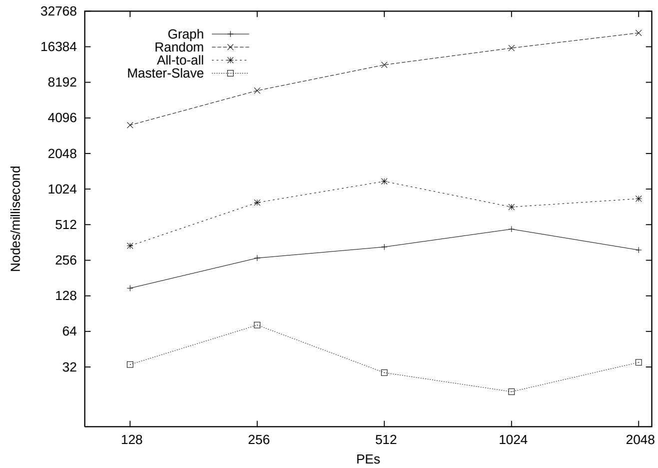

Figure 4.7: Nodes sent per second on Lemieux, problem D=50

happens to get stuck on one node, a lot of extra useless nodes will end up expanded and sent across the network. In addition, nodes have a 1/processors chance of staying on their home processor, which is much much lower than the chance of staying put in other balancers.

All-to-all achieves reasonable performance considering the large number of nodes sent around, especially compared to master-slave. Graph balancing sends very few nodes around as well, which is not unexpected given its low overhead and fast runtime. The trend of less nodes sent across the network as the number of processors increase is also a very good trend – it won't cease scaling due to network congestion like random is already at 2048 processors.

4.2.5 Further Tests

Further runs were done on Lemieux, to confirm that the D=50 test problem above was not simply a "lucky problem". Another problem was chosen, shown below in figure 4.8.

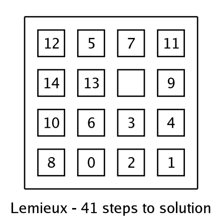

Figure 4.8: A smaller test problem for Lemieux

This problem is far smaller than the previous problem for Lemieux, and as such the scalability in figure 4.9 of it is far worse than the larger problem.

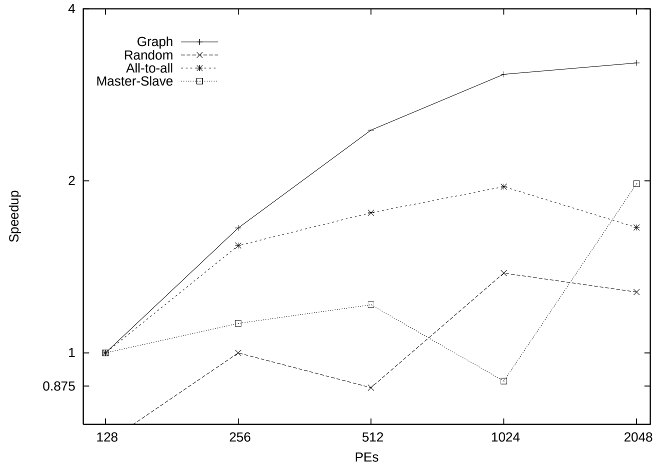

Figure 4.9: Speedup on Lemieux, problem D=41; log-log scale

Figure 4.10 shows that the overhead is still reasonable for the graph based balancer. This is good; even though the scalability starts to fall off due to the tiny problem size, the amount of overhead is growing at a linear rate.

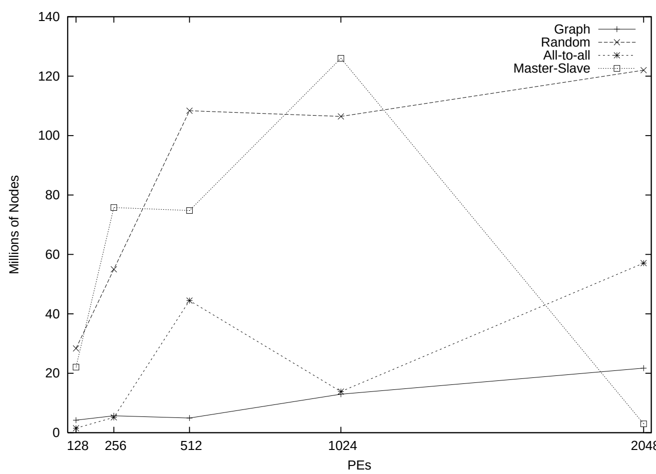

Figure 4.10: Overhead of seed balancers on Lemieux, problem D=41; solution requires approx 200 million nodes to be expanded

Thus the previous results are confirmed for a different problem; the graph based load balancer is the best among the strategies explored in this paper.

Chapter 5

Conclusion and Future Work

The results in this paper show that graph based seed balancing is a good choice to implement. It works well on both small and large scale machines, has a low computation and network overhead, is easy to implement, and does not require machine-specific performance tuning.

However, this should not be taken to mean that application performance issues such as the ramp-up and ramp-down time in each iteration of IDA* can be ignored – these issues must be fixed to achieve good performance on thousands of processors. A good load balancer like the graph balancer can help hide these problems, but cannot fix them.

It may be possible to further fix the ramp-up and ramp-down slowdown via a dynamic load balancing strategy. Rather than a single fixed strategy throughout the life of the computation, a different strategy could be used for early, middle, and late stages. All-to-all to quickly get work distributed to all processors, switching to graph based balancing once all processors have work to do in order to avoid network congestion, and even switching off load balancing during the late stages, as the time to send a node over the network may far outweigh the time to expand it locally.

Another solution to the ramp-up and ramp-down problem would be to write the application in a smarter way. Given the high cost of communication, have all processors expanding the tree of nodes in a breadth-first manner. Once a sufficient amount of work has been created, split up the work and begin normal, load balanced, parallel execution. A fast load balancer that handles already mostly balanced loads, like graph balancing, would then be ideal to keep the machine balanced.

A load balancer that took more information about the problem into account may also be worth researching. Consider if the amount of work below a given node could be estimated accurately. If this information was passed to the load balancer, then high priority work could be more evenly balanced. No processor could end up with its high priority work taking a long time to complete versus the high priority work on other processors.

References

[1] L. V. Kale and Attila Gursoy. Performance benefits of message driven execution. In *Intel Supercomputer User's Group*, St. Louis, October 1993.

[2] L. V. Kale, B. H. Richards, and T. D. Allen. Efficient parallel graph coloring with prioritization. In *Lecture Notes in Computer Science*, volume 1068, pages 190–208. Springer-Verlag, August 1995.

[3] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[4] L.V. Kalé and Vikram Saletore. Efficient parallel execution of IDA* on shared and distributed memory multiprocessors. In *Proceedings of the Sixth Distributed Memory Computing Conference (DMCC6)*, Portland, Oregon, April 1991.

[5] T. Anthony Marsland and Alexander Reinefeld. Heuristic search in one and two player games. Technical Report TR 93-02, Edmonton, Canada, 1993.

[6] Pittsburgh Supercomputing Center (PSC). Lemieux. `http://www.psc.edu/machines/tcs/lemieux.html`, April 2003. Description of the Lemieux supercomputer.

[7] Amitabh Sinha and L.V. Kale. A load balancing strategy for prioritized execution of tasks. In *Workshop on Dynamic Object Placement and Load Balancing, in co-operation with ECOOP's 92*, Utrecht, The Netherlands, April 1992.