Parallelizing the IDA* Heuristic Search Algorithm

Experiences in Solving the 15-Puzzle with a Parallelized Java IDA* Implementation

Erik van Zijst and Sander van Loo
January 19th, 2003

Vrije Universiteit,
Faculty of Sciences, Department of Mathematics and Computer Science,
Amsterdam, The Netherlands

{erik,sander}@marketxs.com

1. Abstract

This text describes our experiences in solving the well-known 15-puzzle optimally, using our parallel implementation of the heuristic, depth-first search algorithm IDA*[1] on the large, distributed multicomputer DAS[2] at the Vrije Universiteit in Amsterdam and determining the speedups that can be achieved using multiple processors. In our tests we will use both a standard Java runtime environment from IBM that comes with a JIT (Just In Time) compiler and Manta[3], a Java compiler developed at the Vrije Universiteit that compiles Java source code to native x86 code and features a very fast RMI implementation that can make use of the Myrinet network offered by DAS to reduce communication overhead. We will run the program on different processor configurations to see how well the implementation scales and how Manta can influence performance.

2. Introduction

The 15-puzzle was invented by Sam Loyd, a recreational mathematician in the 1870s[4] and appeared in the scientific literature shortly thereafter[5]. The puzzle consists of a rectangular frame that is filled with tiles, leaving one position empty or blank. Any tile horizontally or vertically adjacent to the blank position can be slid into that position. The task of our program is to rearrange the tiles from some random initial configuration into a particular goal configuration, ideally or optimally in a minimum number of moves.

3. Iterative-Deepening A*

To find the best paths for a given initial board configuration, we use IDA*. IDA* has the advantage over algorithms like A*[6], that it does not require all previously evaluated paths to be kept in memory during the execution. With puzzles like the 15-puzzle, that often require hundreds of million positions to be evaluated and has a total number of 10^{13} states, A* will quickly exhaust the all available memory on most problems. Just like A*, IDA* is a heuristic algorithm that gives every node a cost metric based on its ply (its depth in the game tree) and its shortest estimated path to a goal state, derived from the applied heuristic and defined as:

$$f(n) = g(n) + h(n)$$

where *g(n)* is the node's ply and *h(n)* is a heuristic estimate of the length of a shortest path from node *n* to a goal state. It is important that *h(n)* is admissible, meaning that it never overestimates the distance to a goal. During evaluation of the game tree, nodes that have a cost that is higher than the game's estimated total cost, are discarded and their paths pruned. The puzzle's total cost is estimated upfront by the same heuristic that is used to compute *h(n)*. If no solution is found after the whole game tree has been searched up to its maximum ply or *bound*, the bound is incremented and the algorithm restarted, leading to an iterative deepening approach.

4. Implementation

For our tests, we decided to create three programs from scratch that are not derived in any way from the sequential sample code offered by the VU, to avoid the situation of parallelizing a *black box*. We have first implemented a simple sequential version of the program that uses IDA* to solve any random board configuration. The program does not generate or shuffle its own problems, but instead reads both a shuffled and a goal layout text file and computes all optimal solutions. When the program finds a solution, it recursively backtracks the game tree and prints all moves

that lead to the goal configuration. This feature was later used to verify the correctness of the program. All programs use the Manhattan distances heuristic without any optimizations such as *linear-conflicts*[7].

The second program we wrote is a multithreaded version of our sequential program. It has a main thread that starts the application, reads the shuffled board, estimates the distance of the shortest path to the goal configuration and starts expanding the game tree. When a the tree reaches a configured ply, defined as the pre-branch factor on the command line, the node is placed in a central queue and not expanded any further by the main thread. Aside from the main thread, the application has a number of worker threads that can be specified on the command line. These workers read jobs from the central job queue and expand them further. When a solution is found it is printed to stdout. When the main thread has placed its last job on the job queue, it closes the queue and initiates a barrier to wait for all worker threads to finish their work. A closed job queue that is empty, immediately throws an exception when its get-method is invoked, rather than blocking until the main thread inserts a new job. When a worker thread catches this exception, it enters the barrier to synchronize with the other threads.

The third program is a distributed version of the multithreaded program. Instead of using in-process worker threads, it uses remote worker processes that communicate over RMI. The main program starts the application and creates a central job queue that can be accessed by remote worker processes over RMI. When a worker process is started on a remote host, it registers itself at the job queue and starts dequeuing jobs from the job queue to expand them locally. Termination of the algorithm is analogous to the multithreaded version. When the main thread is done, it closes the queue and synchronizes with all workers in a central barrier inside the job queue object. The distributed program features two enhancements over the multithreaded version. First of all, the minimum estimated size of the jobs put in the queue can be configured. This is called the minimum grain size of the leaf jobs and has a default value of 5. Without pruning, a job with a cost of five can expand into 3^5 = 243 child nodes. Jobs with a cost less than the grain size are not put in the queue, but computed directly by the main thread to avoid excessive communication overhead. The second enhancement is that of workers pre-fetching new jobs into a local, private job queue while computing their current job to avoid the worker's execution thread ever having to wait for the network. To this end, every worker thread has an associated thread that tries to keep the local queue full at all times. The pre-fetch thread runs at a higher priority, ensuring a timely refill after a job has been dequeued. The size of the local queues is configurable. A large value minimizes IO wait and keeps the worker threads busy, but also introduces additional synchronization delay at the end of an iteration. Because jobs at the same ply can differ greatly in actual cost, one worker could empty its local queue much faster than another, resulting in a lot of idle time waiting for the last worker to empty its local queue, decreasing scalability. A similar problem exists during the start of an iteration, when one worker process keeps the central queue empty and the other workers idle, because its pre-fetch buffer has a high capacity.

5. Correctness

Since the sequential sample implementation downloaded from the course's website does not give any statistics, nor the actual paths of the solutions it claims to discover, it could not be used to verify the correctness of our programs. To be sure our programs are correct, we ran the same problem multiple times on different machines and compared their results. We also verified whether the paths printed by the program were indeed valid moves leading to the goal configuration and whether the optimal solution was as short as the example program said. To detect possible race conditions, we included a counter that keeps track of the total number of expanded nodes. In all situations, the programs managed to solve the puzzle optimally in the same amount of computations.

6. Tests

We used the distributed program to run a number of tests on the DAS and measured the performance of the program solving problems of different size on up to 32 processors. The initial board configurations used for the tests were generated by running the example program with a length of 68, 82, 84 and 86.

The first board was generated after 68 shuffle moves and is saved in dist68.txt. It has 35 optimal solutions of 64 moves and will be computed by evaluating 275890257 nodes, using a pre-branch factor of 10, a minimum leaf job grain-size of 25 and local worker queues with a capacity of 7. It has the following layout:

14	10	13	12
6	5	9	
15	1	8	4
11	7	3	2

The second puzzle was generated after 82 shuffle moves and is saved in dist82.txt. It has 21 optimal solutions of 62 moves and will be computed by evaluating 775922910 nodes, using a pre-branch factor of 10, a minimum leaf job grain-size of 25 and local worker queues with a capacity of 7. It has the following layout:

15	6	14	13
1	5	10	12
	8	9	4
11	7	3	2

The third puzzle was generated after 84 shuffle moves and is saved in dist84.txt. It has 21 optimal solutions of 64 moves and will be computed by evaluating 1649425810 nodes, using a pre-branch factor of 10, a minimum leaf job grain-size of 25 and local worker queues with a capacity of 7. It has the following layout:

15	6	14	13
1	5	10	12
11	8	9	4
7		3	2

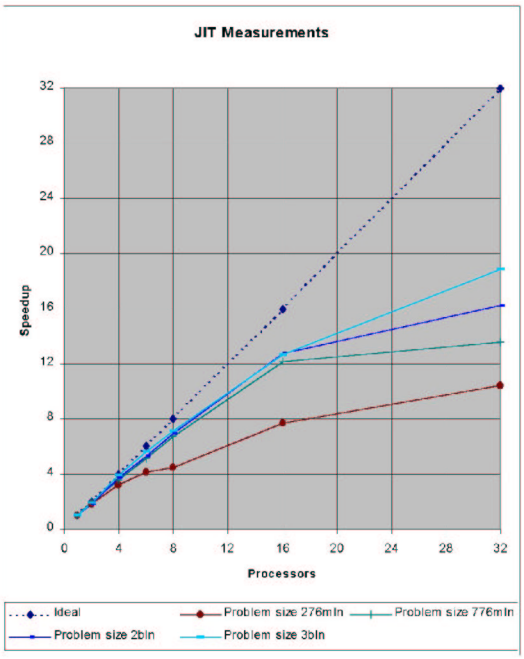
The fourth and last puzzle was generated after 86 shuffle moves and is saved in dist86.txt. It has 25 optimal solutions of 66 moves and will be computed by evaluating 3096066968 nodes, using a pre-branch factor of 10, a minimum leaf job grain-size of 25 and local worker queues with a capacity of 7. It has the following layout:

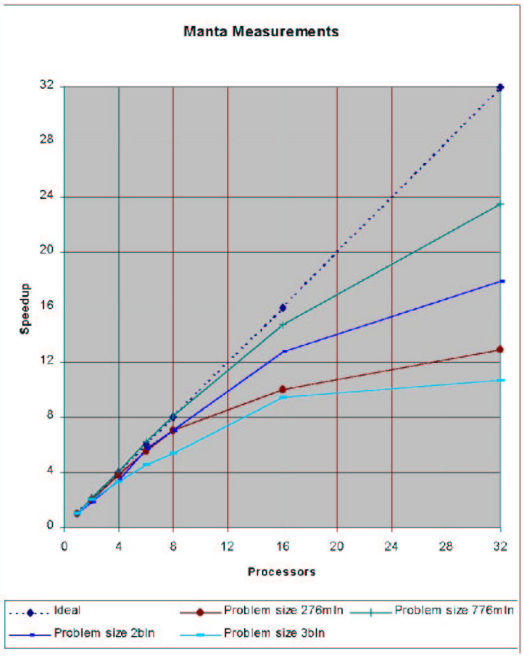
15	6	14	13
1	5	10	12
11	8	9	4
7	3	2	

Each puzzle will be solved using both a pure Java version (compiled and run by IBM's JVM with JIT) and a native x86 version (compiled with Manta with the compiler switches `-myrinet -no_bounds -no_cast_checks -fast_math`) on 1, 2, 4, 6, 8, 16 and 32 processors. The exact commands used to run the tests can be found in appendix I.

7. Results

The charts below show the speedup of the program when deployed on more than one processor for both the JIT and the Manta version.





The charts clearly show the program's rather poor speedups when run on more than 16 processors. With the exception of the largest

problem size in the Manta tests, the program appears to scale better as the problem sizes increase. In our situation there is a synchronization issue when both an iteration of the algorithm is started and when it terminates. In both cases, the worker's pre-fetch buffers may cause other worker processors to remain idle as explained in paragraph 4. The smaller the total computing time, the larger the effect of this issue becomes, this may also explain the poor results we had with very small problems. However, disabling job pre-fetching on smaller problems did not yield any better speedups, due to increased communication overhead of the many, relatively small jobs. Increasing the job size in this case would decrease part of the communication overhead, but at the same time lead to a larger synchronization period in the barrier when the last job to be expanded has a high cost. The scalability could probably be improved by applying *work-stealing* after the last job has been removed from the central job queue. With such an approach, the local pre-fetch buffers could be kept relatively large. Another useful enhancement would be the ability to give a new job in the queue to the longest waiting worker. This would circumvent the startup problems introduced by larger pre-fetch buffers.

It is interesting to see that in almost all cases, the JIT version of the program outperforms the native Manta version using Myrinet. Especially in deployments with only one DAS node, and larger problem sizes, JIT is nearly twice as fast as Manta. The local pre-fetch buffers certainly allow the JIT version to scale better due to decreased communication overhead. Every job fetched from the central queue requires an RMI invocation that is up to 30 times faster on Manta's Myrinet RMI implementation. Without pre-fetching, the JIT application would have suffered from a much larger communication overhead, leading to much more processor idle time. In our tests however, most communication is asynchronous and execution threads almost never have to wait for the network, rendering most of Myrinet's advantages obsolete. What remains unclear however, is why Manta is not at least as fast as JIT and why Manta performs relatively better than JIT with increased problem sizes.

Appendix II contains the results of all tests, including the total execution time of every test.

8. Conclusions

In this text we presented a number of programs to solve the 15-puzzle optimally using the iterative-deepening A* heuristic search algorithm. We have tested the distributed parallel program on the DAS cluster and analyzed the speedups achieved with multiple processors for both a Java JIT version using standard 100Mbps partly switched ethernet and a native Manta version using 1.2Gbps Myrinet. To minimize processor idle time, we implemented job pre-fetching. Achieving satisfactory speedups however proved to be a matter of finding the very delicate balance between number of processors, minimum grain-size of the leaf jobs and the size of the pre-fetch buffers. More robust scalability could probably be obtained by adding a work-stealing mechanism as well as a priority aware central job queue. An even better approach would probably be to add Transposition Table Driven Work Scheduling[8] to eliminate the idle time during startup and termination of the algorithm's iterations, while keeping all communication asynchronous.

[1] Korf, R.E., Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, Vol. 27, No. 1, 1985, pp. 97-109

[2] Distributed ASCI Supercomputer (DAS), *http://www.cs.vu.nl/das*

[3] Manta; Fast Parallel Java, *http: //www.cs.vu.nl/manta*

[4] Loyd, S., *Mathematical Puzzles of Sam Loyd*, selected and edited by Martin Gardner, Dover, New York, 1959

[5] Johnson, W.W. And W.E. Storey, Notes on the 15 puzzle, *American Journal of Mathematics*, Vol. 2, 1879, pp. 397-404.

[6] Hart, P.E., N.J. Nilsson, and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics*, Vol. 4, No. 2, 1968, pp. 100-107

[7] Korf, R.E. and Felner, A., Disjoint Pattern Database Heuristics, *Artificial Intelligence Journal*, Vol. 134, No. 1-2, Jan. 2002, pp. 9-22

[8] John W. Romein, Henri E. Bal and Jonathan Schaeffer, Transposition Table Driven Work Scheduling in Distributed Search, *American Association for Artificial Intelligence*, July 1999, pp. 725-731

Appendix I

Below is the script that was used to run all tests and to gather the results in text files.

```
# JIT

for j in 1 2; do for i in 1 2 4 8 16 32; do ( prun -v -t 10:00 \
./run_ibm_java $i nl.vu.pp.distributed.FifteenPuzzle -board ../dist58.txt \
-prebranch 7 -grainsize 5 -prefetch 5 >> ../doc/tests/jit-58-$i.txt ) ; done ; done

for j in 1 2; do for i in 1 2 4 6 8 16 32; do ( prun -v -t 10:00 \
./run_ibm_java $i nl.vu.pp.distributed.FifteenPuzzle -board ../dist66.txt \
-prebranch 10 -grainsize 25 -prefetch 7 >> ../doc/tests/jit-66-$i.txt ) ; done ; done

for j in 1 2; do for i in 1 2 4 6 8 16 32; do ( prun -v -t 1:00:00
./run_ibm_java $i nl.vu.pp.distributed.FifteenPuzzle -board ../dist68.txt
-prebranch 10 -grainsize 25 -prefetch 7 >> ../doc/tests/jit-68-$i.txt ) ; done ; done

for j in 1 2; do for i in 1 2 4 6 8 16 32; do ( prun -v -t 2:10:00 \
./run_ibm_java $i nl.vu.pp.distributed.FifteenPuzzle -board ../dist70.txt \
-prebranch 8 -grainsize 25 -prefetch 7 >> ../doc/tests/jit-70-$i.txt ) ; done ; done

for j in 1 2; do for i in 1 2 4 6 8 16 32; do ( prun -v -t 2:10:00 \
./run_ibm_java $i nl.vu.pp.distributed.FifteenPuzzle -board ../dist74.txt \
-prebranch 8 -grainsize 25 -prefetch 7 >> ../doc/tests/jit-74-$i.txt ) ; done ; done

for j in 1 2; do for i in 1 2 4 6 8 16 32; do ( prun -v -t 4:00:00 \
./run_ibm_java $i nl.vu.pp.distributed.FifteenPuzzle -board ../dist82.txt \
-prebranch 10 -grainsize 25 -prefetch 7 >> ../doc/tests/jit-82-$i.txt ) ; done ; done

for j in 1 2; do for i in 1 2 4 6 8 16 32; do ( prun -v -t 8:00:00 \
./run_ibm_java $i nl.vu.pp.distributed.FifteenPuzzle -board ../dist84.txt \
-prebranch 10 -grainsize 25 -prefetch 7 >> ../doc/tests/jit-84-$i.txt ) ; done ; done

for j in 1 2; do for i in 32 16 8 6 4 2 1; do ( prun -v -t 16:00:00 \
./run_ibm_java $i nl.vu.pp.distributed.FifteenPuzzle -board ../dist86.txt \
-prebranch 10 -grainsize 25 -prefetch 7 >> ../doc/tests/jit-86-$i.txt ) ; done ; done


# MANTA

for j in 1 2; do for i in 1 2 4 6 8 16 32; do ( prun -v -t 10:00 \
./15puzzle-distributed $i -board dist58.txt -prebranch 7 -grainsize 5 \
-prefetch 5 >> doc/tests/manta-58-$i.txt ) ; done ; done

for j in 1 2; do for i in 1 2 4 6 8 16 32; do ( prun -v -t 10:00 \
./15puzzle-distributed $i -board dist66.txt -prebranch 10 -grainsize 25 \
-prefetch 7 >> doc/tests/manta-66-$i.txt ) ; done ; done

for j in 1 2; do for i in 1 2 4 6 8 16 32; do ( prun -v -t 1:00:00 \
./15puzzle-distributed $i -board dist68.txt -prebranch 10 -grainsize 25 \
-prefetch 7 >> doc/tests/manta-68-$i.txt ) ; done ; done

for j in 1 2; do for i in 1 2 4 6 8 16 32; do ( prun -v -t 2:10:00 \
./15puzzle-distributed $i -board dist70.txt -prebranch 8 -grainsize 25 \
-prefetch 7 >> doc/tests/manta-70-$i.txt ) ; done ; done

for j in 1 2; do for i in 1 2 4 6 8 16 32; do ( prun -v -t 2:10:00 \
./15puzzle-distributed $i -board dist74.txt -prebranch 8 -grainsize 25 \
-prefetch 7 >> doc/tests/manta-74-$i.txt ) ; done ; done

for j in 1 2; do for i in 1 2 4 6 8 16 32; do ( prun -v -t 4:00:00 \
./15puzzle-distributed $i -board dist82.txt -prebranch 10 \
-grainsize 25 -prefetch 7 >> doc/tests/manta-82-$i.txt ) ; done ; done

for j in 1 2; do for i in 1 2 4 6 8 16 32; do ( prun -v -t 8:00:00 \
./15puzzle-distributed $i -board dist84.txt -prebranch 10 \
-grainsize 25 -prefetch 7 >> doc/tests/manta-84-$i.txt ) ; done ; done

for j in 1 2; do for i in 32 16 8 6 4 2 1; do ( prun -v -t 16:00:00 \
./15puzzle-distributed $i -board dist86.txt -prebranch 10 \
-grainsize 25 -prefetch 7 >> doc/tests/manta-86-$i.txt ) ; done ; done
```

Appendix II

Below are the results of all tests.

lit	-	Problem size	8mln	7962313 jobs	(dist58.txt)
	1 cpu	46.36	44.73 average:	45.54 speedup:	1
	2 cpu	29.27	30.014) average:	29.64 speedup:	1.54
	4 cpu	21.62	22.5 average:	22.06 speedup:	2.06
	6 cpu		average:	speedup:	
	8 cpu	16.22	15.32 average:	15.77 speedup:	2.89
	16 cpu	13.35	14.04 average:	13.7 speedup:	3.33
	32 cpu	13.46	13.18 average:	13.32 speedup:	3.42

manta	-	Problem size	7mln	7962313 jobs	(dist58.txt)
	1 cpu	65.95	65.86 average:	65.9 speedup:	1
	2 cpu	34.71	35.05 average:	34.88 speedup:	1.89
	4 cpu	19.07	18.59 average:	18.83 speedup:	3.5
	6 cpu	13.01	13.55 average:	13.28 speedup:	4.96
	8 cpu	12.5	16.24 average:	14.37 speedup:	4.59
	16 cpu	9.96	9.93 average:	9.94 speedup:	6.63
	32 cpu	14.5	11.8 average:	13.15 speedup:	5.01

lit	-	Problem size	14mln	14032634 jobs	dist74.txt
	1 cpu	79.61	78.97 average:	79.29 speedup:	1
	2 cpu	54.11	50.8 average:	52.46 speedup:	1.15
	4 cpu	34.98	35.96 average:	35.47 speedup:	2.24
	6 cpu	30.97	37.96 average:	34.46 speedup:	2.3
	8 cpu	33.6	29.59 average:	31.59 speedup:	2.51
	16 cpu	24.56	23.18 average:	23.87 speedup:	3.32
	32 cpu	24.71	26.95 average:	25.83 speedup:	3.07

manta	-	Problem size	14mln	14032634 jobs	dist74.txt
	1 cpu	123.85	115.59 average:	119.72 speedup:	1
	2 cpu	64.97	62.23 average:	63.6 speedup:	1.88
	4 cpu	39.28	44.47 average:	41.87 speedup:	2.86
	6 cpu	44.2	37.34 average:	40.77 speedup:	2.94
	8 cpu	27.62	30.36 average:	28.99 speedup:	4.13
	16 cpu	27.53	27.45 average:	27.49 speedup:	4.36
	32 cpu	20.34	28.96 average:	24.65 speedup:	4.86

lit	-	Problem size	59mln	58832778 jobs	dist70.txt
	1 cpu	320.92	331.61 average:	326.27 speedup:	1
	2 cpu	187.25	188.85 average:	188.05 speedup:	1.74
	4 cpu	115.38	111.84 average:	113.61 speedup:	2.87
	6 cpu	84.45	87.31 average:	85.88 speedup:	3.8
	8 cpu	67.29	73.33 average:	70.31 speedup:	4.64
	16 cpu	61.65	64.64 average:	63.15 speedup:	5.17
	32 cpu	56.89	52.2 average:	54.55 speedup:	5.98

manta	-	Problem size	59mln	58832778 jobs	dist70.txt
	1 cpu	509.46	484.52 average:	496.99 speedup:	1
	2 cpu	248.69	242.44 average:	245.57 speedup:	2.02
	4 cpu	132.65	134.63 average:	133.64 speedup:	3.72
	6 cpu	98.67	93.52 average:	96.09 speedup:	5.17
	8 cpu	81.67	87.05 average:	84.36 speedup:	5.89
	16 cpu	69.98	74.88 average:	72.43 speedup:	6.86
	32 cpu	60.26	50.75 average:	55.51 speedup:	8.95

lit	-	Problem size	92mln	91545972 jobs	dist66.txt
	1 cpu	519.22	501.64 average:	510.43 speedup:	1
	2 cpu	299.25	300.99 average:	300.12 speedup:	1.7
	4 cpu	169.28	168.32 average:	168.8 speedup:	3.02
	6 cpu	132.29	121.86 average:	127.07 speedup:	4.02
	8 cpu	96.22	98.16 average:	97.19 speedup:	5.25
	16 cpu	92.76	74.06 average:	83.41 speedup:	6.12
	32 cpu	64.4	65.99 average:	65.19 speedup:	7.83

manta	-	Problem size	92mln	91545972 jobs	dist66.txt
	1 cpu	774.07	830.3 average:	802.18 speedup:	1
	2 cpu	432.56	393.15 average:	412.86 speedup:	1.94
	4 cpu	229.51	212.64 average:	221.08 speedup:	3.63
	6 cpu	140.58	156.22 average:	148.4 speedup:	5.41
	8 cpu	121.13	103.96 average:	112.54 speedup:	7.13
	16 cpu	87.33	59.83 average:	73.58 speedup:	10.9
	32 cpu	78.35	72.66 average:	75.5 speedup:	10.62

lit	-	Problem size	276mln	275890257 jobs	dist68.txt
	1 cpu	1461.5	1533.34 average:	1497.42 speedup:	1
	2 cpu	785.73	825.23 average:	805.48 speedup:	1.86
	4 cpu	468.54	460.04 average:	464.29 speedup:	3.23
	6 cpu	370.53	356.52 average:	363.53 speedup:	4.12
	8 cpu	336.33	335.54 average:	335.93 speedup:	4.46
	16 cpu	198.1	192.7 average:	195.4 speedup:	7.66
	32 cpu	131.38	156.87 average:	144.13 speedup:	10.39

manta	-	Problem size	276mln	275890257 jobs	dist68.txt
	1 cpu	2293.8	2289.6 average:	2291.7 speedup:	1
	2 cpu	1180.12	1169.2 average:	1174.66 speedup:	1.95
	4 cpu	600.01	596.83 average:	598.42 speedup:	3.83
	6 cpu	396	437.33 average:	416.67 speedup:	5.5
	8 cpu	333.9	321.64 average:	327.77 speedup:	6.99
	16 cpu	238	218.23 average:	228.12 speedup:	10.05
	32 cpu	181.39	175.44 average:	178 speedup:	12.88

lit	-	Problem size	776mln	775922910 jobs	dist82.txt
	1 cpu	4375.97	4403.83 average:	4389.9 speedup:	1
	2 cpu	2309.62	2188.64 average:	2249.13 speedup:	1.95
	4 cpu	1237.17	1220.52 average:	1228.84 speedup:	3.57
	6 cpu	844.96	860.13 average:	852.54 speedup:	5.15
	8 cpu	657.56	659.34 average:	658.45 speedup:	6.67
	16 cpu	351.72	371.68 average:	361.7 speedup:	12.14
	32 cpu	365.83	283.56 average:	324.7 speedup:	13.52

manta	-	Problem size	776mln	775922910 jobs	dist82.txt
	1 cpu	7120.42	7062.63 average:	7091.53 speedup:	1
	2 cpu	3313.07	3233.88 average:	3273.47 speedup:	2.17
	4 cpu	1718.19	1802.25 average:	1760.22 speedup:	4.03
	6 cpu	1155.45	1146.08 average:	1150.76 speedup:	6.16
	8 cpu	853.69	904.5 average:	879.1 speedup:	8.07
	16 cpu	460.51	503.33 average:	481.92 speedup:	14.72
	32 cpu	317.66	285.64 average:	301.65 speedup:	23.51

lit	-	Problem size	2bln	1649425810 jobs	dist84.txt
	1 cpu	8878.24	9013.4 average:	8945.82 speedup:	1
	2 cpu	4960.77	4870.25 average:	4915.51 speedup:	1.82
	4 cpu	2423.67	2376.18 average:	2399.92 speedup:	3.73
	6 cpu	1651.75	1728.05 average:	1689.9 speedup:	5.29
	8 cpu	1293.49	1277.35 average:	1285.42 speedup:	6.96
	16 cpu	705.71	698.48 average:	702.1 speedup:	12.74
	32 cpu	469.64	633.72 average:	551.68 speedup:	16.22

manta	-	Problem size	2bln	1649425810 jobs	dist84.txt
	1 cpu	13779.04	13764.15 average:	13771.6 speedup:	1
	2 cpu	7178.46	7694.52 average:	7436.49 speedup:	1.85
	4 cpu	4382.92	3693.97 average:	4038.44 speedup:	3.41
	6 cpu	2430.56	2427.71 average:	2429.14 speedup:	5.67
	8 cpu	1931.25	1979.67 average:	1955.46 speedup:	7.04
	16 cpu	1178.46	984.22 average:	1081.34 speedup:	12.74
	32 cpu	676.18	747.41 average:	771.79 speedup:	17.84

lit	-	Problem size	3bln	3096066968 jobs	dist86.txt
	1 cpu	17406.59	17555.85 average:	17481.22 speedup:	1
	2 cpu	8842	9359.63 average:	9100.81 speedup:	1.92
	4 cpu	4492.05	4453.47 average:	4472.76 speedup:	3.91
	6 cpu	3067.57	3120.51 average:	3094.04 speedup:	5.65
	8 cpu	2514.93	2412.46 average:	2463.7 speedup:	7.1
	16 cpu	1312.26	1456.35 average:	1384.31 speedup:	12.63
	32 cpu	994.01	861.25 average:	927.63 speedup:	18.85

manta	-	Problem size	3bln	3096066968 jobs	dist86.txt
	1 cpu	27655.55	25813.41 average:	26734.48 speedup:	1
	2 cpu	14251.62	13117.98 average:	13684.8 speedup:	1.95
	4 cpu	8011.7	8172.51 average:	8092.11 speedup:	3.3
	6 cpu	5901.63	5934.07 average:	5917.85 speedup:	4.52
	8 cpu	5273.79	4703.42 average:	4988.6 speedup:	5.36
	16 cpu	2211.15	3483.08 average:	2847.12 speedup:	9.39
	32 cpu	3212.36	1796.35 average:	2504.36 speedup:	10.68