A Massively Parallel Transportation Solution[†]

David J. Haglin[‡]
Computer & Info Sciences Dept.
Mankato State University
Mankato, MN, USA 56002-8400
Phone: (507) 389-5306
Fax: (507) 389-6376
haglin@mankato.msus.edu

John A. Kaliski
Department of Management
Mankato State University
Mankato, MN, USA 56002-8400
Phone: (507) 389-5337
Fax: (507) 389-5497
johnk@mankato.msus.edu

Abstract

An effective metaheuristic for solving non-square linear programming problems is demonstrated to be well-suited to a distributed memory environment. In serial computing environments such a metaheuristic was shown in practice to greatly speed the convergence properties of several existing interior point methods. By simultaneously controlling communication overhead and (nearly) uniformly distributing work across multiple processors, it is demonstrated that this metaheuristic experimentally exhibits optimal scaled speedup. The analysis technique used for this problem is well-suited to any problem which is memory intensive and has a fairly well-understood serial experimental running time.

Keywords: Transportation, Interior Point Method, Massively Parallel, Scaled-Speedup, SIMD

1 Introduction

In [1, 2, 3, 4, 5] a metaheuristic — an improvement technique that applies to a family of algorithms — was developed which greatly improves the serial convergence properties

---

[†] We gladly thank the Joint Institute for Computational Science (JICS) of the University of Tennessee for providing us with access to a MasPar MP-2.

[‡] Supported in part by NSF under grant DUE-9552312.

of several existing interior point methods (IPMs) for disproportionate rectangular linear programming (LP) problems, and potentially general convex programming problems. The metaheuristic is based on the simple idea that for disproportionate rectangular LPs, the vast majority of computational time is wasted working with variables (commonly termed as *nonbasic*) which will eventually be set to zero in the final, optimal solution.

This metaheuristic can be effectively parallelized. By simultaneously controlling communication overhead, using inexpensive communication primitives, and (nearly) uniformly distributing work across multiple processors, we conclude that this metaheuristic experimentally exhibits optimal scaled speedup. Our problem is not well-suited to traditional speedup analysis, so we looked at the *scaled speedup* technique described in [6, 7]. We employed a scaled speedup analysis technique where the *ideal* growth was derived from serial experimental results reported in the literature.

To our knowledge, such investigations of the inherent parallelism of this metaheuristic have not been reported in the literature.

1.1 Problem description

Consider the following disproportionate LP in standard form:

$$\begin{array}{ll} \text{Minimize} & c^T x \\ \text{Subject to} & Ax = b \\ & x \geq 0 \end{array}$$

where $A \in \Re^{m \times n}$, and m and n are significantly different. Any solution x is said to be feasible if $Ax = b$ and $x \geq 0$. Also x is said to be an interior point if $Ax = b$ and $x > 0$.

One such problem is the *transportation* problem, which is an example of a network flow problem where s *suppliers* need to find the least costly way of shipping b_i units of an item to d *demanders* who require b_j units of the same item. Without loss of generality, we may assume that $s \geq$d. This problem is one of the oldest and most fundamental to Operations Research. The A matrix for the transportation problem (hereafter referred to as the *transportation constraint matrix*) has dimension $m \times n$, where $m = (s + d)$, and $n = (s \cdot d)$, and has the following form:

$$A = \begin{pmatrix} e^T & 0 & \ldots & 0 \\ 0 & e^T & \ldots & 0 \\ \ldots & & & \\ 0 & 0 & \ldots & e^T \\ I & I & \ldots & I \end{pmatrix}$$

Table 1: General IPM Scheme

1. Build the inscribed ellipsoid centered around current iterate
2. Calculate a least squares (orthogonal) projection of a merit function onto the surface the ellipsoid.
3. Using the projection as a direction of movement, calculate the positive step size to optimize the merit function along direction while remaining in the interior of the feasible region.
4. Update the current iterate.

where e is a d-dimensional vector of 1's and I is a $(d \times d)$-dimensional identity matrix. Note that there are s copies of e^T in A. Intuitively, this transportation constraint matrix has a column for each possible supplier–demander pairing. There are exactly two 1's in each column; the top 1 entry indicates the supplier and the bottom 1 entry indicates the demander.

1.2 Previous work

The metaheuristic mentioned above was designed in [1, 2, 3, 4, 5] to be applied to general IPMs. IPMs are the first known class of algorithms which solve LP (and other convex programming problems) in polynomial time. At a high level most IPMs proceed in a similar manner. Starting from an initial interior solution, the IPMs iterate the steps in Table 1 until an optimal solution is found.

By far the most theoretically and computationally expensive part of the IPM-class algorithm is the formulation and computation of the least squares projection. The primary goal of the metaheuristic developed in [1, 2, 3, 4, 5] was to reduce the computational burden that this projection imposes when solving disproportionate rectangular LPs. This is accomplished is through the realization that for a given iteration many of the constraints are far enough from the current iterate x so as to have very little impact on the inscribed ellipsoid and thus very little impact on the generated direction. Therefore if these distant constraints are temporarily ignored in the current iteration's least squares projection, a close approximation to the original direction may be found with a significant computational savings. In [3, 4] this metaheuristic was shown to reduce the (serial) run times of a common IPM by between 62% and 96%.

The reader should note that because the current iterate x changes with each iteration, the subset of close constraints which are used in the projection calculation vary significantly and unpredictably during the execution of the algorithm. This variability eliminates the benefits of using many of the standard a priori symbolic ordering techniques. Also this variability may cause (in serial implementations) difficulties with paging due to the inherent lack of locality of reference in the memory access patterns.

1.3 Model of distributed computing

Our algorithm should work well on any *shared-nothing* model of parallel computation where each processing entity (PE) has local memory and is capable of communicating with all other PEs over some interconnection network. The communication primitives we require are: (i) broadcast, (ii) reduce, (iii) random-read, and (iv) random-write. The *broadcast* primitive involves one PE sending a value to every other PE. The *reduce* primitive combines a distributed set of values using any associative binary operator to produce a single result. With the *random-read* (fetch) primitive a *destination* subset of the PEs receive data from a *source* subset (not necessarily disjoint) of the PEs. Each destination receives exactly one piece of data, but multiple destinations may receive data from the same source PE. The *random-write* (send) primitive is the most complex and the most expensive we require. It involves source PEs sending data to destination PEs. Unlike the random-read, a destination PE may receive more than one piece of data. So some "collision" handling must be done, such as selecting the minimum value or summing the values.

2 Distribution of data

Our A matrix has a very well-defined structure which allows us to avoid creating the entire matrix in memory. Instead, we logically maintain block-columns of the A matrix on each PE. The first block-column is the first d columns of the A matrix, and subsequent block-columns follow similarly. This logical distribution allows each PE to maintain its portion of the A matrix without actually storing it in its memory. Specifically, given an offset into its d columns, it can trivially compute the two row positions holding a value of one.

The cost vector c has two different views within our problem. In the LP, the c vector is an n-tuple. We can also logically view this data as an $s \times d$ matrix, which can be thought of as an adjacency matrix for a complete bipartite graph with s (suppliers) as one partition and d (demanders) as the other partition. In keeping with the logical distribution of the A matrix, the s suppliers are distributed across s PEs. As such, all c values associated with supplier i are housed at PE i. Note that this distribution implies utilizing s PEs. We distribute x and other n-tuple data in a similar manner.

For convenience, label the right-hand side vector b as $(b_1, \ldots, b_s, b_{s+1}, \ldots, b_{s+d})$. Let $B_s = (b_1, \ldots, b_s)$ and $B_d = (b_{s+1}, \ldots, b_{s+d})$. Recall that B_s indicates the suppliers capacity and B_d indicates the demanders requirements. We distribute B_s across the s PEs and B_d across the first d PEs. Other m-tuple data is distributed in a similar manner.

Recall that the subset of close constraints vary significantly during the execution of the algorithm, which causes sporadic memory references. Therefore, any static distribution of the data based on a preprocessing step cannot ensure minimum communication overhead. However, with our distribution of data, the bipartiteness of the adjacency matrix view of c guarantees that each PE will have some work do to at every step.

3 Pseudo code of algorithm

```
Begin(Main Loop)
1.      Partition
1.1.        Select a subset of the close constraints
1.2.        Find minimum-cost spanning tree (MST) for preconditioning
2.      Compute right-hand side (RHS) as $b$ or as gradient potential function
3.      Compute Preconditioned conjugate gradient (CG) using MST
3.1.        Presolve (Symbolic Order)
3.2.        While (Residual $>$ some Threshold)
3.2.1.          Compute Preconditioned Residual
3.2.2.          CG Iteration
4.      Compute partial or complete Primal Value
5.      Compute Alpha (step size)
6.      Update iterate
End(Main Loop)
```

In our implementation the outer loop is done in serial. For the largest problems we solved (consisting of 8,000,000 variables), the outer loop typically ran fewer than 60 iterations. We perform each of the steps of the main loop in parallel where an obvious parallel method exists for many of these steps. We describe our parallelization of steps 1., 3.1, and 3.2.1 of the main loop.

3.1 Partition Step

The serial implementation of Step 1 consists of a partial quick-sort algorithm to feed the lightest edges into Dijkstra's MST algorithm. The idea is essentially to select a pivot, perform a partition step of moving all elements (edge weights) smaller than the pivot to its left and larger edge weights to its right, and then to ascertain whether the ordinal position of the pivot is with certain tolerances. If it is, sort the left partition, provide these edge weights to Dijkstra's algorithm to find the MST, and use these light edges as the subset of the close constraints. If the pivot does not satisfy certain tolerances, then recursively seek such a pivot following the traditional methods of the quick-sort.

Since our parallel implementation has the edge weights spread out among the PEs, we must use certain tricks to emulate the serial implementation's behavior. First, our pivot selection is done by having each PE find its lightest edge weight in its local memory (i.e. find the lightest edge adjacent to its associated supplier vertex). Since the lightest edge incident to a vertex will be part of the MST, we can include all of these lightest edges in the MST without further computation. Note that prior to invoking Dijkstra's algorithm, we already have at least half of the MST edges needed.

The largest of these lightest edges is chosen to be the pivot, and each PE finds all local edge weights lighter than the pivot and moves them to the bottom of the array. Our experiments showed that the number of edge weights lighter than the pivot on any PE was seldom more than 30, and surprisingly insensitive to the problem size. Since we generally have such a small number, we have each PE perform an insert sort on its collection of edge weights lighter than the pivot. We are now ready to feed these edge weights (minus the lightest edges already known to be in the MST) to Dijkstra's algorithm to find an MST. We consider edges in ascending order by weight exactly like the serial version of Dijkstra's algorithm. But the tree merges needed to perform efficient cycle detection is done in parallel.

3.2 Presolve and Preconditioner Computation

The serial implementation of Step 3 consists of solving the linear system based on $A\,D\,A^T$ using a CG method, where D is a diagonal scaling matrix. Let M be the columns of the transportation constraint matrix which correspond to the edges of the MST. It is known that M is an effective cg preconditioner for this system since it is easily invertible and is a good approximation for A in $A\,D\,A^T$ given the weighting structure of D. Inverting M involves successively pruning the leaves of the tree until no vertices remain. Inverting M^T is essentially a breadth-first search (BFS) on a graph formed by interpreting M as an adjacency matrix. [1]

In parallel, we prune all leaves concurrently which results in a running time based on the width of the MST. It is known how to perform a parallel BFS efficiently [8].

---

[1] The BFS interpretation of inverting M^T is the author's observation and does not appear in the literature.

Table 2: Communication Costs per Main Loop Iteration

	Primitive			
Step Number	Broadcast	Reduce	Random Read	Random Write
1.1	2	4		
1.2	$d-1$	$5d$		
2	1	$d/2$		
3.1		3		$2d+s+tw+hsc$
3.2.1		$1.5cgi \cdot tw$	$cgi \cdot tw$	
3.2.2	$2cgi$	$5cgi$	$cgi \cdot hsc$	$cgi \cdot hsc$
4	1	2	hsc	
5	$d+1$	1		
6	1			

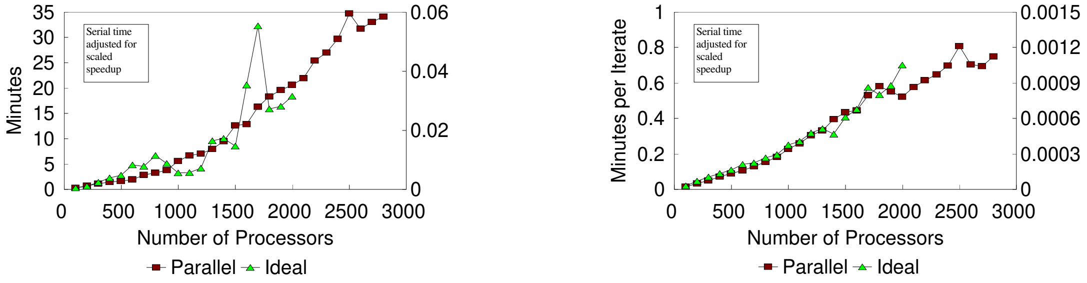
4 Communication analysis

Our implementation of the algorithm was done on a massively parallel SIMD machine. The number of communication steps is dependent upon the input data. In Table 2, we count the number of communications based on the following internal parameters: the number of CG iterations (cgi), the highest number of selected columns on a single PE (hsc), and the width (diameter) of the minimum spanning tree (tw).

Experimentally, we observed that hsc was surprisingly insensitive to the problem size (and hence the number of PEs) and typically remained below 30. Although theoretically $cgi = s + d - 1$, we found similar conjugate gradient performance to that reported in [4]; cgi was always less than $(s+d)/4$ and frequently $cgi < (s+d)/10$.

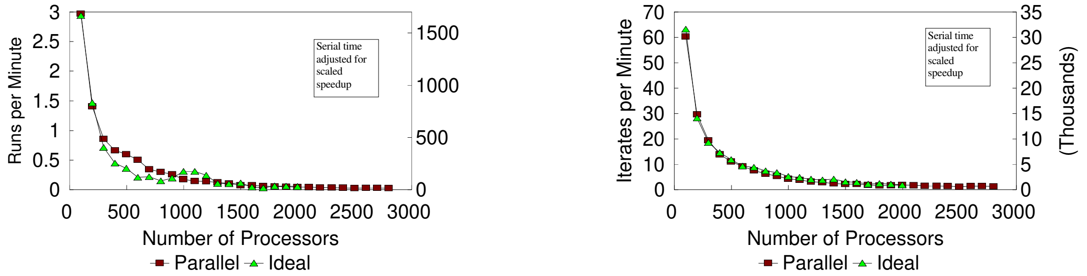
5 Experiments and results

Due to the size of the LPs studied, we analyzed our work using the *scaled-speedup* technique presented in [6]. We ran problems were $s = d$ because such problems were reported in the literature for serial implementations. We ran these experiments for problem sizes of $s = 100, 200, ..., 2800$, running ten randomly generated input instances for each problem size. We considered the average times and number of main loop iterations from the ten trials. We found that $s = 2800$ was the largest problem size that could be stored in the local memory of the PEs on our machine.



a. Total Time

b. Time per Iterate



c. Runs per Minute

d. Iterates per Minute

Figure 1: Ideal vs Scaled Speedup

Our intention was to ascertain the algorithm's inherent parallelizability. Due to the memory requirements for these large problems, fixed-size speedup analysis was not an option [7]. Instead we looked to scaled-speedup analysis as the evaluation technique. Our analysis method involved projecting an *ideal* growth pattern from the serial run-time data assuming perfect parallel speedup, using the following formula:

experimental serial running time/number of PEs

This was the ideal used in Figure 1. One difficulty with this approach is that the serial run-time data was gathered some time ago on a different machine. So we are faced with comparing two sets of data gathered in different units. To offset irregularities in the units, we considered a large range of data and scaled the graphs in such a way to most closely match the growth functions. In addition, we removed the scaling on the timing data by dividing each time value by the maximum within that paradigm (serial vs parallel), resulting in a scaleless measure of running time differences as shown in Figure 2.

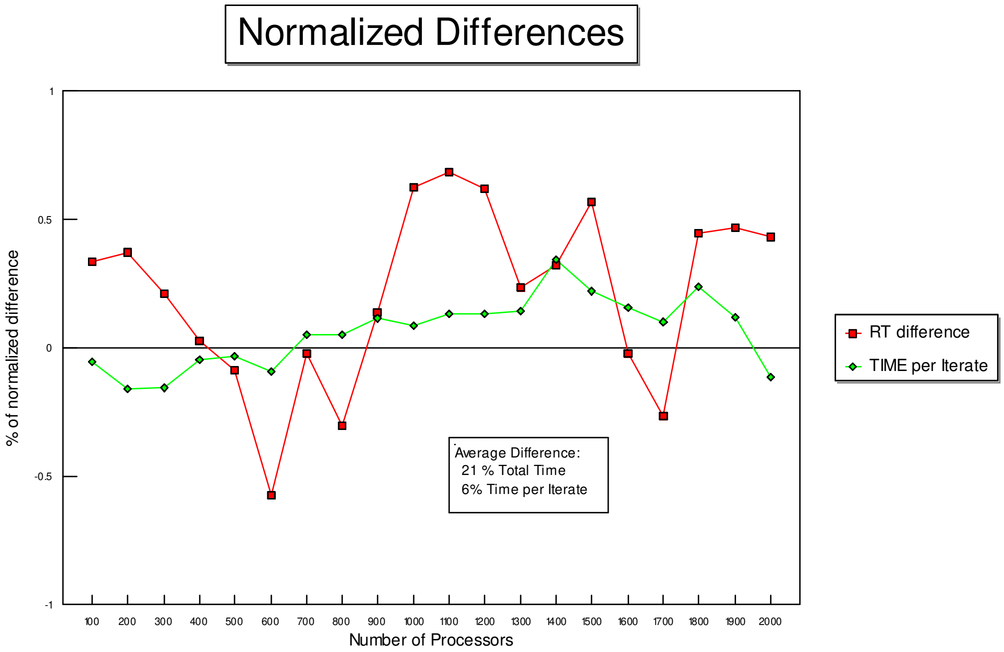


Figure 2: Scaleless Time Differences

For completeness, we have included charts in Figures 3 and 4 depicting where our parallel implementation spends time for various problem sizes. The critical algorithm steps are selecting the constraints subset (Step 1), solving the linear system using CG (Step 3), and determining the step size (Step 5). Other computations are lumped into the *Other* category.

Note that the serial charts are for a slightly different problem — the crew scheduling problem. Unfortunately no such time breakdown is available in the literature for the serial implementation of the transportation problem; however in [3] the same (serial) metaheuristic was used to solve both the transportation and crew scheduling problems, where it was shown that the metaheuristic performed very similarly for both of these problems. Comparing the trends depicted in Figures 3 and 4, we find that over the problem sizes considered, the metaheuristic effectively reduces the effort needed to generate the search direction in both the serial and parallel paradigms. However in the serial system we see that other steps of the algorithm (e.g. Step 6) take on an increasing relative importance as the problem size grew, eventually limiting the maximum problem size which could be solved in reasonable amounts of computing time. In our parallel implementation this was not the case. By effectively parallelizing each step, we were able to solve problems across the size range we considered with relatively little change in these relative proportions. As such we found that the primary limiting factor with the parallel implementation was simply the amount of memory available to each PE.

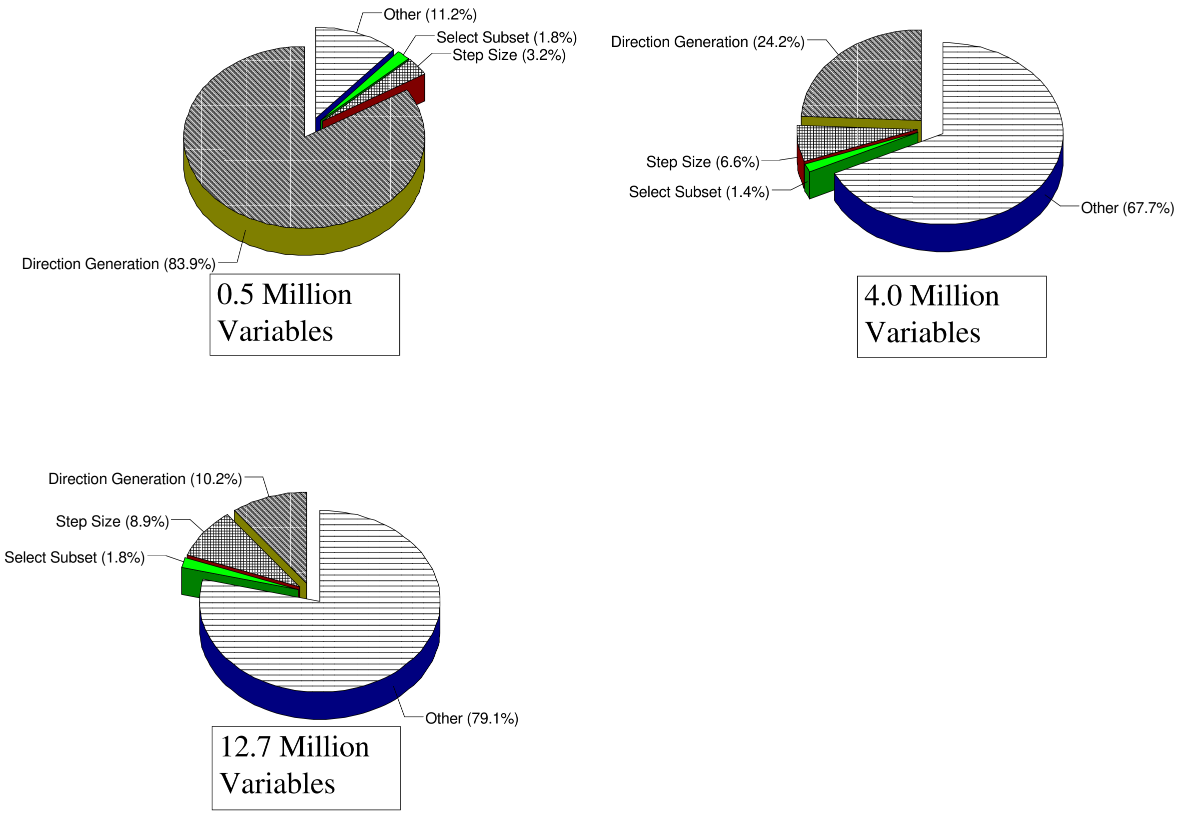
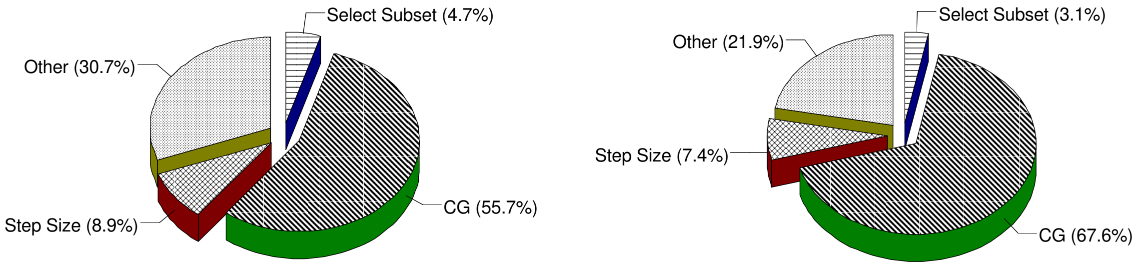


Figure 3: Run Time Breakdown (Serial)

6 Conclusions

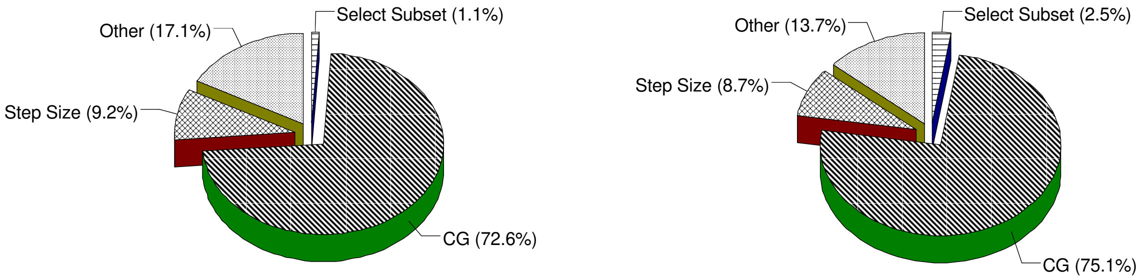
Our work on this problem shows that the metaheuristic developed in [1, 2, 3, 4, 5] can be effectively parallelized. Serial implementations of the metaheuristic have a very random memory access pattern which tends to inhibit good performance in a distributed memory model of parallel computation. However, our distribution of data has exhibited controlled communication overhead levels allowing (nearly) optimal scaled speedup. An ancillary benefit to our investigation was the capability to solve (in a reasonable amount of time) a problem with 8,000,000 variables. The largest problem previously solved with this metaheuristic had 4,000,000 variables.

Initially, we struggled with how best to analyze the running times we were seeing from our code. The literature suggests using a scaled speedup technique when memory is a



a) 500 PEs

b) 1000 PEs



c) 2000 PEs

d) 2800 PEs

Figure 4: Run Time Breakdown (Parallel)

limitation (as it is in our case) [7]. So, we tried a scaled speedup comparison of our parallel running times to the linear growth pattern of the problem size relative to the number of PEs. We realized that this analysis was irrelevant because it did not consider the actual growth pattern of the serial implementation's running time. Once we projected an ideal by dividing the actual serial running time by the number of PEs we used to solve that problem size, our analysis became both meaningful and compelling. This was true even though the serial and parallel trials are measured in different units since they were performed on separate machines at different times. We then removed the scale on each of the two timing trials by dividing the serial (resp. parallel) timing data by the maximum measurement taken among the serial (parallel) trials. Our conclusion is that we lost approximately 6 percent efficiency per major iteration, in practice, when going from a serial to a massively parallel compute engine. We were surprised to realize how meaningful this analysis technique is, and yet it is not a commonly used method.

There are several areas of future work. We would like to adapt our code to utilize the number of available PEs, independent of the problem size. Our parallel implementation did not explicitly rely on the interconnection network. We should therefore try our algorithm in a loosely-coupled distributed computing environment. Perhaps a particular interconnection network may be leveraged to realize further gains.

There are many problems that are effectively solved in serial using an interior point method. We have demonstrated that one such problem is effectively solvable in a concurrent processing environment. There seems to be a dearth of parallel interior point method studies. Our experience indicates some nice synergies between IPMs and parallel and distributed computing.

Acknowledgment

The authors wish to thank Mark Gilbert and Rod Waltermann for their help in some of the coding.

References

[1] G. B. Dantzig and Yinyu Ye. A build–up interior method for linear programming : Affine scaling form. Technical Report SOL 90–4, Systems Optimization Laboratory, Dept. of Operations Research, Stanford University, Stanford, CA 94305, USA, February 1990.

[2] J. A. Kaliski. Further results of decomposition variants for large scale linear programming. Working Paper No 91–21, Dept. of Management Science, College of Business Administration, University of Iowa, Iowa City, IA 52240, USA, December 1991.

[3] J. A. Kaliski. *A decomposition variant for large scale linear programming.* PhD thesis, Dept. of Management Science, College of Business Administration, University of Iowa, Iowa City, IA 52240, USA, May 1992.

[4] John A. Kaliski and Yinyu Ye. A short-cut potential reduction algorithm. *Management Science*, 39(6):757–776, June 1993.

[5] K. Tone. An active–set strategy in interior point method for linear programming. *Mathematical Programming*, 59:345–360, 1993.

[6] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Mathematics*, 9(4):609–638, July 1994.

[7] Lawrence A. Crowl. How to measure, present, and compare parallel performance. *IEEE Parallel & Distributed Technology*, pages 9–25, Spring 1994.

[8] Hillel Gazit and Gary L. Miller. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Information Processing Letters*, 28(2):61–65, June 1988.