SANDIA REPORT

SAND2004-6440
Unlimited Release
Printed February 2005

LDRD Final Report on Massively-Parallel Linear Programming: the parPCx System

Erik G. Boman, Sandia National Laboratories
Ojas Parekh, Emory University
Cynthia A. Phillips, Sandia National Laboratories

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/ordering.htm

SAND2004-6440
Unlimited Release
Printed February 2005

LDRD Final Report on Massively-Parallel Linear Programming: the parPCx System

Erik G. Boman and Cynthia A. Phillips
Algorithms and Discrete Math Department
Sandia National Laboratories
Mail Stop 1110
P.O. Box 5800
Albuquerque, NM 87185-1110

Ojas Parekh
Emory University
Math/Computer Science Department
400 Downman Drive
Atlanta, GA 30322

Abstract

This report summarizes the research and development performed from October 2002 to September 2004 at Sandia National Laboratories under the Laboratory-Directed Research and Development (LDRD) project "Massively-Parallel Linear Programming". We developed a linear programming (LP) solver designed to use a large number of processors. LP is the optimization of a linear objective function subject to linear constraints. Companies and universities have expended huge efforts over decades to produce fast, stable serial LP solvers. Previous parallel codes run on shared-memory systems and have little or no distribution of the constraint matrix. We have seen no reports of general LP solver runs on large numbers of processors.

Our parallel LP code is based on an efficient serial implementation of Mehrotra's interior-point predictor-corrector algorithm (PCx). The computational core of this algorithm is the assembly and solution of a sparse linear system. We have substantially rewritten the PCx code and based it on Trilinos, the parallel linear algebra library developed at Sandia. Our interior-point method can use either direct or iterative solvers

for the linear system. To achieve a good parallel data distribution of the constraint matrix, we use a (pre-release) version of a hypergraph partitioner from the Zoltan partitioning library.

We describe the design and implementation of our new LP solver called parPCx and give preliminary computational results. We summarize a number of issues related to efficient parallel solution of LPs with interior-point methods including data distribution, numerical stability, and solving the core linear system using both direct and iterative methods. We describe a number of applications of LP specific to US Department of Energy mission areas and we summarize our efforts to integrate parPCx (and parallel LP solvers in general) into Sandia's massively-parallel integer programming solver PICO (Parallel Interger and Combinatorial Optimizer). We conclude with directions for long-term future algorithmic research and for near-term development that could improve the performance of parPCx.

Acknowledgment

We thank Professor Steven Wright (University of Wisconsin) for his collaboration on this LDRD project. He provided useful insight on interior-point methods and the PCx code. We thank Vicki Howle (Sandia) for assistance with preconditioners. Jonathan Eckstein (Rutgers) designed and implemented the core portion of the PICO ramp up. We thank Mike Heroux and Robert Hoekstra for help with Trilinos and Epetra, and Ken Stanley and Padma Raghavan for the DSCPACK/Amesos integration. We also acknowledge Rob Bisseling, David Day, Bruce Hendrickson, Marzio Sala, and Michael Saunders for helpful discussions.

Contents

1 Introduction 9
2 Linear Programming Applications 12
3 parPCx Design Issues 15
4 Iterative Methods and Preconditioning 17
4.1 Numerical results for preconditioning 18
4.2 Support preconditioners 21
4.3 Preconditioning the augmented system 21
4.4 Parallel preconditioning 22
5 Data distribution 23
6 Code Overview 24
6.1 PCx 24
6.2 Trilinos 24
6.3 parPCx 25
6.4 How to use the code 27
6.5 PICO modifi cations 27
7 Parallel Results 28
8 Future Work 29
References 34

LDRD Final Report on Massively-Parallel Linear Programming: the parPCx System

1 Introduction

A linear program (LP) problem is the minimization or maximization of a linear function of rational variables subject to linear equalities and inequalities. More specifically, the standard form of a linear program is

$$\min c^T x \text{ subject to } Ax = b, x \geq 0,$$

where A is an $m \times n$ constraint matrix, c and x are vectors of length n and b is a vector of length m. The problem is theoretically tractable, and large problems (n up to tens of thousands) can be solved quite efficiently with current technology. When special structure is present, problems with millions or even billions of variables have been solved.

Linear programming has been a workhorse optimization technology since the 1940s. Largely it is used in approximations for more complex nonlinear systems and for systems with integrality constraints (integer programs). Sandia National Laboratories needs a massively-parallel linear programming solver for two main applications: finding approximate solutions to huge graph problems arising in homeland defense and infrastructure surety, and fast computation of lower bounds in the PICO (Parallel Integer and Combinatorial Optimizer) massively-parallel integer programming code. There are currently no general parallel linear-programming codes that can scale to the level we require. Current general parallel linear-programming solvers scale only to tens of processors. We have seen no results reported for runs on more than sixteen processors.

There are two practical serial strategies for solving linear programs: simplex methods and interior-point methods. The simplex method begins at a vertex of the polytope of feasible solutions and progressively moves to better vertices till it finds an optimal solution (there will always be a vertex with the optimal solution). There are no known polynomial-time simplex algorithms, but in practice, simplex solves many linear-programming problems in time approximately linear in the size of the input. This matches the smoothed complexity result of Spielman and Teng[36]. The computational core of a sparse simplex method (designed to keep the working set of data comparable in size to the original number of nonzeros in constraint matrix A) is difficult to parallelize.

Practical interior-point methods generate iterates that solve both the linear program and its dual (the problem $\max b^T y$ subject to $A^T y + s = c$, $s \geq 0$) simultaneously. Each iterate of the method satisfies the inequality constraints strictly; that is, $x > 0$ and $s > 0$. (This is the meaning of the term "interior".) The iterates approach a point at which the other conditions

necessary for optimality—the equality constraints $Ax = b$ and $A^T y + s = c$, and the complementarity condition $x's = 0$—are satisfied. Steps generated by these interior-point methods typically have two components: an "affine scaling" component which aims to satisfy the optimality conditions directly, and a "centering" component that stabilizes the algorithm and ensures steady progress toward a solution. Mehrotra's algorithm includes systematic and well tested heuristics for combining the affine scaling and centering directions and for choosing the step length at each iteration.

The computational core of this algorithm is the solution of a sparse system of linear equations with the matrix ADA^T, where A is the original constraint matrix and D is a positive diagonal matrix which varies with each iteration. The main technical challenge in this LDRD project was to determine and implement methods that use thousands of processors effectively in the assembly and solution of sparse symmetric matrices of this type. There are three main research thrusts: finding scalable direct methods based on Cholesky decomposition, finding effective iterative methods, and implementing methods where the constraint matrix is so large it must be distributed. We are not aware of any previous parallel linear-programming code where the constraint matrix is distributed.

There are a few parallel commercial interior-point (barrier) solvers available today, most notably CPLEX (from ILOG) and parallel OSL (from IBM). They both parallelize the linear algebra (Cholesky factorization). CPLEX's parallel barrier solver only runs on shared-memory computers and a modest number of processors. Parallel OSL is, as far as we know, only available for IBM SP computers.

The first parallel interior-point LP solver we are aware of, was developed by Bisseling et al. [4] at Shell in the early 1990's. The code was specially written for a network of transputers (distributed memory). They achieved a remarkable speedup of up to a factor 88 on 400 processors for large sparse LPs from the oil industry.

Since it is difficult to achieve good parallel speed up on sparse Cholesky Factorization on general systems, much research has focused on LPs with special structure, especially staircase and block-angular structure. Grigoriadis and Khachiyan [22] and Schultz and Meyer [33] have developed parallel research codes specialized for such systems. OOPS by Gondzio and Sarkassian [21] is a recent LP and QP solver, not currently available to the public as source code. OOPS shows good parallel performance on up to 16 processors for large, sparse problems with block structure, like multicommodity network problems. It has solved systems with up to tens of millions of variables and constraints.

We developed two parallel interior-point LP codes based on the prexisting serial PCx interior-point code. These two codes comprise the current *parPCx* system. The first version parallelizes only the linear solver within PCx. This is the most compute intensive part of the code, and thus benefits the most from parallel solution. We developed a generic solver class that can use either direct or iterative solvers. We access direct solvers through the Amesos package and iterative solvers through the AztecOO package. Both of these packages are in Trilinos [23], Sandia's collection of compatible software packages to support parallel linear algebra. Koka et. al. developed the IPS code during the timeframe of the research we

report. IPS also substitutes a parallel core within PCx, but its parallel linear-algebra core is a fast small-scale (e.g. 4-processor) parallel Cholesky customized for a shared-memory Intel platform [26].

The second version of our parallel LP code uses the same (parallel) solver class, but in addition we have made the entire interior-point method fully data parallel using the Epetra parallel matrix package. In this way, the constraint matrix is stored distributed across processors, enabling us to solve very large problems that will not fit on a single processor. This version of the code is still not fully debugged as of this writing.

A crucial issue in the data distribution is how to distribute the sparse constraint matrix. We use a hypergraph model of the matrix that accurately represents the communication cost in matrix-vector multiplication. We use a hypergraph partitioner to find a good distribution. Note that we cannot directly use graph partitioners in his case because the constraint matrix is rectangular. One can approximate the functionality by replacing each hyperedge with a clique and using a graph partitioner, but it doesn't work as well. Our code allows for both row and column decomposition of the matrix. Our partitioning technique is also beneficial for certain types of preconditioning.

An advantage of using the Amesos library for direct solvers is that it is easy to switch among several external sparse linear solver packages. One drawback is that we cannot modify the external solvers, which prevents us from using specialized techniques for small pivots as is customary for interior-point methods. This leads to numerical difficulties for some types of problems. We have found that we can partially avoid these problem using regularization techniques, which are non-intrusive.

As anticipated, finding effective preconditioners for iterative solvers has proved challenging. The linear systems arising in the predictor-corrector method are extremely ill-conditioned and standard preconditioning techniques (Jacobi, incomplete Cholesky) did not work well. We generalized a class of support preconditioners. However, the linear systems from the LP iterations are also not normally symmetric diagonally dominant, where support preconditioners work well theoretically and in practice. Therefore, this type of support tree method also did not work well in general. We have the infrastructure to experiment with new preconditioners and have begun research on new methods, but finding effective new general preconditioning methods will require far more time and effort.

The final major goal of the LDRD project was to incorporate parallel LP methods into the PICO parallel integer programming solver. As mentioned above, one primary use of linear programming technology is as a bounding subroutine within integer-programming solvers. An integer program (IP) is a linear program with additional integrality constraints on (a subset of) the variables. Addition of integrality constraints makes the problem formally intractable, more challenging in practice, and far more useful. PICO is a parallel branch-and-bound (branch-and-cut) system for intelligently searching the space of feasible integer solutions. A key to practical performance of such a system is efficient identification and elimination of subregions that provably contain no interesting solution (none that are better than a feasible solution already in hand). Because a linear program is an efficiently-

solvable relaxation of an integer program, it provides a lower bound for a minimization problem or an upper bound for a maximization problem. If this bound is worse than a current feasible solution, we can eliminate the entire subregion from further consideration. We have made several modifications to PICO to allow the use of parallel LP solvers. However, PICO can not use parPCx directly until parPCx (or another system) provides a cross-over operation. This finds an optimal vertex solution to the LP from an optimal interior-point solution. PICO uses vertex solutions to efficiently resolve closely-related LPs later in the search.

The remainder of this report is organized as follows: Section 2 describes some applications of linear programming to the mission of Sandia National Laboratories and describes how a parallel linear programming solver can improve the performance of parallel integer programming solvers. Section 3 describes design issues for parallel interior-point LP solvers and describes major design decisions for parPCx. Section 4 describes iterative methods for solving the core linear system for an interior point method. In particular it discusses options for preconditioning the system. Section 5 discusses methods for solving LPs when the constraint matrix is too large to fit on a single processor. Section 6 gives details of the implementation of parPCx and describes high-level modifications to PICO to allow parallel LP bounding. Section 7 lists and discusses some empirical results for parPCx. Finally Section 8 concludes by listing directions for future research and development.

2 Linear Programming Applications

In this section, we describe some of the linear-programming applications that motivated this research. These are primarily problems in homeland security. We also discuss how a parallel linear-programming solver can improve the performance of parallel integer programming codes.

The first LP application is the *k*-hurdle problem, which models a simple defense-placement problem in networks. Given a network with a start point (or more generally a "super" start point that simultaneously represents all possible network entry points) and a target point, we wish to place a minimum number of "hurdles" so that an intruder going from the start to the target must pass at least *k* hurdles. The hurdles can represent guards, cameras, sensors, etc. We proved (prior to this work) that due to special structure, the variables for this linear program will always be integral at optimality. Therefore, we can solve this problem with an LP. More complex versions involving costs for the sensors, failure probabilities, etc, become integer programs. In particular, researchers at Sandia National Laboratories have formulated a number of integer programs related to sensor placement in municipal water networks [2, 3, 42]. Large instances can have difficult root problems. Thus an IP code could benefit from parallelized root solves. Furthermore, we've seen instances for a real water network where the (difficult) root linear program returned an integral, and therefore optimal, solution. Unfortunately, these problems tend to lose the network structure from the *k*-hurdle problem that would make preconditioning easier (see Section 4).

Another potential application of LP is mitigation planning for biowarfare. Los Alamos has a unique population microsimulation capability, originally demonstrated in transportation applications, and more recently used in EPISIM for simulation of disease transmission. Simulations of population interactions in a large city, such as those proposed for the National Infrastructure Simulation and Analysis Center (NISAC), can produce graphs with 10 billion edges. We developed and justified an IP model for allocating limited vaccinations. Given the size of the graph and the uncertainty of data and modeling, one will likely need only an approximate solution to this problem. One might, for example, find a problem-specific decomposition of the LP solution to find a feasible approximately-optimal solution to the IP. The quality of the approximation depends upon the quality of the decomposition which in turn depends critically upon the quality of the IP formulation (how closely the LP feasible region approximates the convex hull of the integer points). See [9] for the general theory of decomposition-based approximation algorithms.

Another more direct application of LP is the computation of strategies to attack or harden infrastructure networks. The analysis system envisioned by Carlson, Turnquist, and Nozick [8] uses an LP to compute expected losses from particular types of attacks/attackers in a network. They model attacks (e.g. success/failure of individual steps) with a hidden Markov process and model the attacker with a Markov decision process. Standard post-solution LP sensitivity analysis of the attack-loss computation can indicate good directions for hardening the network. SCADA (supervisory control and data acquisition) networks was a primary motivation for that work. SCADA networks control critical US infrastructure such as the power grid and natural gas distribution. These LPs will become huge for the analysis of large systems, and they directly retain network structure, making them good candidates for support preconditioners (see Section 4).

A final application of LP is as a bounding procedure in the solution of integer programs, and therefore they can contribute to all the varied applications of integer programming. The optimal solution to a linear program generally has rational (fractional) values. However, in settings where the variables represent decisions, a fractional value is infeasible. For example, in general one cannot build half an airplane (and gain half its utility). As described in the introduction, an integer program is a linear program augmented with (nonlinear) integrality constraints on some or all of the variables. Integer programs are far more powerful than linear programs. We can represent decisions with binary variables (zero for no and one for yes, for example). Thus integer programs can represent many resource allocation problems or problems related to the study of natural systems. For some problems where the values of the variables are very large (e.g. deciding how many dollars the US government will spend in a year), rounding fractional variables to nearest integers is a reasonable approximation, provided this rounded value satisfies the other linear constraints. Generally integer variables take on a tiny range of values and numerical rounding does not generally even lead to a feasible solution, never mind a good solution. However, even in this case, linear programs provide valuable bounds for searching as described in the introduction.

We now describe in more detail how a parallel LP solver can accelerate parallel integer programming codes. In particular, in this project, we targeted the PICO (Parallel Integer

and Combinatorial Optimization) integer programming solver developed by researchers at Sandia National Laboratories and Rutgers University [16]. The search for an optimal solution to an IP starts with a single root "subproblem", namely the original full IP. Whenever we cannot eliminate a problem by bounding or other methods such as adding constraints and inferring variable restrictions, we partition the feasible region represented by that subproblem into a small number of subproblems, usually two. Though one could burst the feasible region into a large number of independent pieces from the start, in practice this leads to performance so poor there can even be slowdown anomalies [15]. Therefore, a practical search requires careful early evaluation of gradients and other information associated with the branching (partitioning). In particular, we must build the search tree by carefully bounding and partitioning each unresolved subproblem. Parallel LP can be particularly useful early in the search when there are fewer independent subregions to evaluate than there are processors. In this case, the excess processors might search for feasible solutions to improve the pruning. However, the search cannot proceed until the bounding is complete, so allowing more processors to actively speed the bounding could significantly improve early search performance. Even in settings with modest parallelism, where fewer processors are "wasted" with a serial bounding process, a parallel LP solver can significantly improve bounding at the root. Solving the LP represented by the root problem typically requires one to two orders of magnitude more time than solving subsequent (related) LPs.

After the root LP solve, most subsequent solves are closely related to some previous LP solve. Partitioning usually narrows the bounds on a single variable or related set of variables. When bounding a single node, we frequently resolve the LP after adding additional constraints (called cuts). Even in a serial solve, these related LPs cannot all be processed consecutively (for example, only one of the two regions created in a branch can immediately follow the processing of the parent region). However, if we save some structural information about that previous solution, it significantly speeds solution of the new related problem. Specifically, the basis of the previous optimal solution specifies a particular vertex of the feasible polytope. This point is infeasible for the original (primal) problem by design, but it is feasible for the dual problem. If we start from that point in a dual simplex LP algorithm, we generally only have to travel through a few vertices to arrive at a new optimal solution. Unfortunately, interior-point methods do not generally return a basis, since they do not generally give a vertex solution (this is also an advantage for a single LP solve, since it gives information on all optimal solutions). There is still no accepted efficient method for finding an optimal basis from an interior-point solution. Current methods could take as long as a primal simplex solve in the worst case, though in practice one can expect significantly better performance. parPCx does not have this *crossover* operation implemented at this time.

3 parPCx Design Issues

In this section, we discuss the parallel interior-point computations in a little more detail, discuss design parallelization issues, and explain our design decisions.

Although simplex methods are classic LP solvers and still widely used, interior-point methods (IPM) have become increasingly popular the last 20 years. They have some nice theoetical properties. For example, they simultaneously compute both the primal and the dual solution. This gives a concise, easily-checkable proof of optimality since the objective values of the primal and dual solution are equal precisely when they are both optimal. Also, when there are multiple optimal solutions, one can derive structural information about the full set of optimal solutions from the single solution returned from an IPM. They (or variants thereof) provably run in time polynomial in the input size. Futhermore, with the introduction of primal-dual methods [44] they became effi cient and useful in practice. In fact, in practice, IPM converge in a constant number of iterations (independent of problem size). Sparse simplex methods are diffi cult to parallelize, while IPM involve a lot of linear algebra that can be parallelized. Therefore, most parallel LP solvers are based on IPM and we follow this strategy too.

Recall that an LP in standard form is

$$\min c^T x \tag{1}$$
$$\text{s.t. } Ax = b \tag{2}$$
$$x \geq 0 \tag{3}$$

where typically the constraint matrix A is large but sparse. IPMs generate a sequence of points that are strictly interior to the polytope defi ned by the constraints $Ax = b$ and $x \geq 0$. Only in the limit are all constraints satisfi ed (with equality) at an optimal solution. We did not attempt to develop new IPM algorithms. See [43, 44] for a full description of interior point algorithms for linear programming.

The most time consuming part (typically 80-90%) of an IPM, is to solve the linear system for computing the search directions. This involves solving linear systems with matrices of the form

$$\begin{pmatrix} -D^{-1} & A^T \\ A & 0 \end{pmatrix}$$

where $D^{-1} > 0$ is diagonal. This matrix is symmetric but indefi nite, and is known as the *augmented* system. Most people prefer solving the Schur complement system, which gives the matrix ADA^T. This system is known as the *normal equations*. The matrix for the normal equations is symmetric positive defi nite and has lower dimension than the matrix for the augmented system. However the matrix is often more dense, and more ill-conditioned.

Good parallel performance depends critically on effi cient parallel solution of these large, sparse, symmetric, linear systems of equations (either the normal equations or the augmented system). Writing a complete LP solver from scratch is a major undertaking, and

not necessary. To allow us to concentrate on parallelizing this most critical computation, we based our parallel solver on the existing serial code PCx [13]. This is a open-source LP solver that implements an IPM and Mehrotra's predictor-corrector method in about 5000 lines of C code. PCx is robust and has successfully solved many challenging test problems.

PCx isolates the linear solver in a separate module, so with a modest amount of effort, one can use any linear solver package. Therefore, we built our parallel version of PCx, called parPCx, on top of Sandia's linear algebra framework Trilinos. This provides maximum flexibility in testing current and future linear systems solvers. Trilinos contains many parts, called packages. The packages most relevant to this project are:

Epetra: Basic linear algebra classes for matrices and vectors.

EpetraExt: Extensions to Epetra.

Amesos: Common interface to third party sparse direct solvers.

AztecOO: Contains iterative solvers like CG and GMRES and some basic preconditioners.

Ifpack: Incomplete factorization preconditioners.

Although our goal is a single unified parallel code, parPCx currently has two versions. In the first version we replaced the linear solver in PCx with a parallel solver based on Trilinos objects, but the rest of the PCx code remains serial. For purposes of discussion in this report only, we will call this version `qPCx`[1]. The second version is designed for problems too large for a serial solver. Memory is a limiting resource. Therefore, we implemented a fully distributed version, where all the data is distributed among processors. This required a major rewrite of the PCx code to convert all vectors and matrices to Epetra objects (classes). We also converted the code from C to C++. We call the second version `PCy`[1]. Section 6 contains a more detailed description of the code.

With this framework, We can use either direct or iterative methods to solve the linear systems. All previous IPM codes we are aware of use direct solvers, with good reason. As the IPM converges to a solution, the diagonal matrix D becomes increasingly ill-conditioned and so does the matrix ADA^T. Iterative methods converge slowly or not at all. In theory, the convergence can be accelerated using preconditioners, but IPM systems are notoriously difficult to precondition.

Direct solvers are quite robust even on ill-conditioned problems. For the normal equations, the system is symmetric positive definite so a Cholesky factorization exists. We need

[1]These seemingly obscure names come from the history of the PCx name itself. PCx stands for Predictor-Corrector version x. The authors intended to replace the x with a version number. Therefore PCy is the successor of PCx. Our qPCx code has the same behavior as the initial pPCx code described in Section 6, but instead of using a single solver, it has a flexible solver interface with access to all solvers that match the Trilinos framework interface. Therefore, in some sense, qPCx is a successor to pPCx. Since these names are internal to this document only, we hope the reader will forgive us.

to explicitly form $B = ADA^T$, which is usually smaller in dimension but denser than A. The Cholesky factors are even denser due to fill. We remark that PCx handles dense columns in the matrix A separately, because otherwise B would become completely dense.

When B is nearly singular, the Cholesky factorization process may break down because diagonal pivots become zero or near-zero [44]. One strategy to address this problem is to replace very small pivot elements by a very large number, see [44, Ch. 11]. Unfortunately, this requires a small but intrusive modification to the Cholesky solver code. While PCx comes with a customized version of the Ng-Peyton Cholesky solver, in parPCx we treat Cholesky solvers as "black boxes" that cannot be modified. This means that parPCx is less robust than PCx. An alternative strategy to handle severe ill-conditioning is to apply regularization. Regularization involves adding positive terms to the diagonal. This shifts the smallest eigenvalue away from zero, but also changes the solution to the linear system and thus also the search directions.

A main attraction of iterative solvers is that we don't need to form $B = ADA^T$. Instead we only need to multiply vectors by A and A^T. The reduced memory requirement potentially allows us to solve larger problems. A subtle point is that in order to construct preconditioners, we may need to explicitly form all or parts of B anyway. For example, (block) diagonal preconditioners require the (block) diagonal entries. For incomplete factorizations, the obvious approach is to explicitly form $B = ADA^T$, and then compute the incomplete factor of B. In theory, one could save some memory by only computing a row (or column) of B at a time (on demand). We did not pursue this option since this reduces the memory requirement by at most a factor two, and Trilinos (AztecOO, IFPACK) does not easily allow this strategy. Thus, we explicitly form B when using incomplete factorizations.

We could also apply a hybrid direct-iterative approach. For example, once we have computed a Cholesky factor for $B_k = AD_kA^T$, we can use that factor as a preconditioner at the next step for $B_{k+1} = AD_{k+1}A^T$. This strategy works best when the change in the diagonal is small. Several researchers have suggested improvements on this scheme [41, 1], which all involve updating the Cholesky factors. Since our software is based on treating the linear solver as a "black box" module, we don't have direct access to the Cholesky factors and cannot update them.

4 Iterative Methods and Preconditioning

Direct solvers do not scale well for large number of processors, so iterative methods are essential for large-scale parallelism. Since parallelism is a main motivation, we do not necessarily need to develop iterative solvers that are faster than direct solvers in serial. We shall therefore revisit basic iterative methods even if they have been found lacking by previous authors.

The first decision is what system to solve and which iterative method to use. Since

PCx uses the normal equations, we began with this system. The matrix $B = ADA^T$ is symmetric positive definite, so conjugate gradients (PCG) is the iterative method of choice. One disadvantage of this approach is that the condition number of B can be very large. Solving the corresponding least-squares system using LSQR or CGLS is more numerically stable [5]. We have not yet pursued this option because currently there is no such iterative method in Trilinos, but Saunders [31] has recently reported encouraging results with the primal-dual code PDCO, which employs LSQR.

For preconditioning, there are many options. In preconditioning there is usually a trade-off: Simple preconditioners are quick and easy to construct, but may not reduce the number of iterations by much. More elaborate preconditioners take longer (more work) to construct and to apply, but can significantly reduce the number of iterations. In the first category we find Jacobi and block Jacobi. As expected, these methods lead to poor convergence or no convergence, see Section 4.1. The next method to try is incomplete Cholesky (IC). In its simplest form, the (level-0 or no-fill) incomplete Cholesky factor L of B has the sparsity pattern of the lower triangular part of B and satisfies $LL^T \approx B$. When we allow some fill in L, either based on the graph structure or by a numerical threshold, the resulting preconditioners are of higher quality but require more storage.

4.1 Numerical results for preconditioning

We first tried Jacobi and incomplete Cholesky in AztecOO. At every step in the IPM we recompute ADA^T and form the corresponding preconditioner. The results were quite discouraging. As the LP solver moves towards the solution, the linear systems become more ill-conditioned so we expect the number of iterations to increase. The preconditioner should remedy this growth to some extent. Figure 1 lists the sizes of three LPs from the Netlib LP data set [17]. Figures 2–4 present the results for these problems. For each test problem we show the major iterations (IPM step) and the value of the barrier parameter μ, which is the duality measure $\sum_{i=1}^{n}(x_i s_i)/n$. As we converge towards a solution, μ goes to zero, and the linear systems become more ill-conditioned. We solve two linear systems (predictor and corrector) in each major (IPM) iteration, and we give the CG iteration count for both. We show results for three different preconditioners, and for completeness we show the value of μ separately for each method, even though they should be the same. (In fact, the μ values are almost always identical for the same problem, a clear indication that we generated identical sequences of iterates.)

The only problem we could successfully solve using PCG was afiro (2). For this particular problem, the number of iterations stays almost constant. For all other problems, the number of CG iterations increases as we move towards the solution. We observe that as predicted, Jacobi preconditioning requires the most iterations while IC(0) requires more iterations than IC(1). For the modszk1 problem (Figure 3), the number of iterations increases dramatically as the outer iteration progresses. Eventually the CG iteration fails to converge, indicated by a dash. (We had a limit of 1000 iterations.) Increasing the fill (thereby also the quality) in incomplete factorizations only helps temporarily; eventually the linear sys-

Problem	Rows	Columns
afi ro	27	51
modszk1	687	1620
maros-r7	3136	9408

Figure 1. Selected Netlib LP problems

IPM step	Jacobi		IC(0)		IC(1)	
	log(μ)	iter.	log(μ)	iter.	log(μ)	iter.
0	2.98	20+21	2.98	8+9	2.98	5+5
1	2.39	22+22	2.39	9+9	2.39	6+5
2	1.43	20+22	1.43	7+8	1.43	4+4
3	0.79	22+23	0.79	6+6	0.79	3+4
4	-0.04	24+24	-0.04	7+7	-0.04	4+4
5	-0.87	25+25	-0.87	7+7	-0.87	4+4
6	-3.40	26+26	-3.40	8+8	-3.40	4+5
7	-8.86	24+25	-9.21	7+7	-9.16	4+4
8	-10.4	21+24				

Figure 2. CG iterations for Netlib problem afiro with Jacobi and incomplete Cholesky preconditioners. For each interior point method step we show the value of the barrier parameter μ (a duality measure) and the number of CG iterations for calculating the predictor and the corrector.

tem becomes almost singular and all the methods we tried will fail. For the maros-r7 test problem (Figure 4), there is an extremely sharp transition between good convergence and no convergence.

We used a fi xed residual tolerance of 10^{-6} in these experiments. One can argue that less precision is needed initially, far from the solution. We did not experiment with adaptive tolerances because we require high accuracy close to the solution and the standard preconditioners would likely fail under these conditions.

We conclude that diagonal and IC preconditioning are not suitable techniques for IPM. This is consistent with conventional wisdom. Several variations may improve the preconditioners. Adding diagonal pertubations is often a helpful approach, and has given rise to *modified* and *relaxed* incomplete factorizations. These variations are numerically more stable because adding a positive term to the diagonal shifts the eigenvalues away from zero. We did limited experiments with such variations using the IFPACK package, but any improvement was small. This is a direction that should be investigated further.

IPM step	Jacobi		IC(0)		IC(1)	
	log(μ)	iter.	log(μ)	iter.	log(μ)	iter.
0	5.85	87+89	5.85	20+19	5.85	7+8
1	5.73	145+154	5.73	34+36	5.73	11+11
2	5.28	136+138	5.28	29+29	5.28	13+13
3	4.91	234+242	4.91	37+38	4.91	25+26
4	4.49	341+364	4.49	57+59	4.49	28+33
5	3.52	413+488	3.52	79+80	3.52	58+58
6	3.19	800+802	3.19	135+141	3.19	89+90
7	–	–	3.05	209+210	3.05	131+122
8	–	–	2.70	257+266	2.70	162+157
9	–	–	2.07	396+395	2.07	246+216
10	–	–	1.42	577+634	1.42	555+559
11	–	–	–	–	–	–

Figure 3. CG iterations for Netlib problem modszk1. See Figure 2 for further description.

IPM step	Jacobi		IC(0)		IC(1)	
	log(μ)	iter.	log(μ)	iter.	log(μ)	iter.
0	4.68	43+41	4.68	3+3	4.68	2+2
1	4.11	41+46	4.11	3+3	4.11	2+2
2	4.05	63+67	4.05	3+3	4.05	2+2
3	3.84	56+56	3.84	3+3	3.84	2+2
4	3.59	58+57	3.59	4+4	3.59	2+2
5	3.02	61+64	3.02	4+4	3.02	2+2
6	2.39	63+72	2.39	4+5	2.39	2+2
7	1.87	109+125	1.87	7+8	1.87	2+3
8	1.45	222+248	1.45	120+126	1.45	3+3
9	0.92	437+497	–	–	0.92	4+4
10	0.58	879+ –	–	–	0.58	10+12
11	–	–	–	–	0.10	19+18
12	–	–	–	–	–	–

Figure 4. CG iterations for Netlib problem maros-r7. See Figure 2 for further description.

The results above suggest that iterative methods work well in the beginning, even if they show poor convergence later when the systems become ill-conditioned. One strategy may therefore be to use iterative solvers as long as the iteration count is below a certain threshold. When iteration count exceeds the threshold, we switch to a direct method. We have not tested this strategy, but it should be easy to implement within the parPCx framework.

4.2 Support preconditioners

There are some preconditioners that have provably good quality, in the sense that the effective condition number (after preconditioning) is bounded. However, such preconditioners rely on certain properties of the matrix and do not work in general. *Support preconditioners* [7, 35] is a promising class. They work for systems that are symmetric and diagonally dominant. In particular they work for $B = ADA^T$ where the constraint matrix A has a network structure. That is, each column has exactly two nonzeros of equal magnitude, typically $+1$ and -1. As implied by their name, network LPs can represent graph or network problems and they do arise in practice. The simplest support preconditioners are spanning trees. Vaidya [38] first proposed spanning tree preconditioners in unpublished work [38]. Resende and Veiga [30] first used them in network optimization. Several variations of spanning tree preconditioners have been proposed for network optimization, see [25, 29, 27]. CG plus spanning tree preconditioners provably converges with total work that is a low polynomial in the input size, independent of the condition number [38, 6]. Monteiro et al. [28] recently rediscovered this in the context of interior-point methods. In recent work, Spielman and Teng [34] show that by using spanning subgraphs (denser than trees) one can theoretically achieve optimal asymptotic performance, but this has not yet been tested in practice.

Our strategy for support preconditioners is to build an interface between Trilinos and TAUCS [37], a solver library developed at Tel-Aviv University that includes several types of support-graph preconditioners. Vicki Howle (SNL-CA) has started this integration, but it is not yet available for use in Trilinos or parPCx. A major difficulty is what to do for systems that do not have network structure. In these cases, support preconditioners do not apply directly. Finding extensions to the general symmetric case is an area of active research.

4.3 Preconditioning the augmented system

Alternatively, we can solve the augmented system. This is

$$K = \begin{pmatrix} -D & A^T \\ A & 0 \end{pmatrix},$$

where D is diagonal but may be very ill-conditioned. As mentioned earlier, the augmented system is usually better conditioned than the normal equations (reduced system). Diagonal

scaling can eliminate the ill-conditioning in D. Suppose we want to preserve symmetry. If we choose $D_1 = D^{-1/2}$, then $D_1 D D_1 = I$. A suitable scaling for K is then $diag(D_1, I)$, and the scaled K is

$$\bar{K} = \begin{pmatrix} -I & D_1 A^T \\ A D_1 & 0 \end{pmatrix}.$$

Unfortunately, this only "moves" the ill-conditioning from the diagonal to the off-diagonal blocks, so $\bar{K}$ is often still ill-conditioned. In particular, the reduced system for $\bar{K}$ is the same as for K so nothing has been gained! (The Schur complements are $AD_1D_1A^T = AD^{-1}A^T$.) A possible remedy is to use a scaling of the type $diag(D_1, D_2)$ and then we obtain

$$\hat{K} = \begin{pmatrix} -I & D_1 A^T D_2 \\ D_2 A D_1 & 0 \end{pmatrix},$$

which for a suitable D_2 may be better conditioned. We have not yet investigated this approach.

We remark that regularization can be effective on the augmented system. Several authors (Vanderbei [39], Saunders and Tomlin [32]) have proposed solving the regularized system

$$\tilde{K} = \begin{pmatrix} -D - \gamma^2 I & A^T \\ A & \delta^2 I \end{pmatrix},$$

for some scalars δ, γ. Such regularization may also allow $\tilde{K}$ to be factored into (sparse) $L\tilde{D}L^T$ form with a Cholesky solver, even if $\tilde{D}$ is indefinite. (We say $\tilde{K}$ is quasi-definite.)

Some advanced preconditioning techniques try to identify the active constraints at the solution, and use this information to construct good preconditioners, see e.g. [19]. Note that the preconditioners in [19] were primarily developed for nonlinear programming, and they require a (sparse) factorization, which may be problematic in parallel.

4.4 Parallel preconditioning

The discussions and results in the previous sections were concerned with preconditioning in serial, not in parallel. Iterative solvers are generally well-suited for parallel processing, since matrix-vector multiplication is easy to parallelize. However, many preconditioners have no known parallel versions. Specifically, incomplete factorizations are difficult to parallelize, though some progress has been made in recent years [24]. Therefore, such preconditioners are usually applied within a domain decomposition method. The global matrix B is partitioned among p processors, and each processor performs a local solve on its submatrix B_p. This step involves a local preconditioner on each processor. This functionality is already built into AztecOO and Ifpack. Clearly, the matrix partitioning will affect the quality of the overall preconditioner. We propose a novel approach based on hypergraph partitioning. This is the same approach we use for direct solvers, so the technique is described in detail in Section 5.

5 Data distribution

Data distribution can significantly impact parallel performance on distributed memory machines. In IPMs, the major parallel issue is how to distribute the constraint matrix A, and also ADA^T if it is formed explicitly. We restrict our attention to 1-dimensional data decompositions, that is, matrices are partitioned either by rows or columns. There are some indications that 2-dimensional decompositions are better [4, 40] and this option should be considered for future versions. We did not attempt 2-dimensional decompositions for the current code because this is more difficult to implement and has not been well tested in Epetra, our underlying parallel matrix library.

The explicit form of $B = ADA^T$ is a square symmetric matrix. One usually partitions matrices by representing the matrix as a graph where each node is a row and two rows r_1 and r_2 are connected if some column has a nonzero in both r_1 and r_2. One can then use standard graph partitioning on this graph. This produces a row partitioning of matrix B which is also a row partition of matrix A. Although graph partitioning is a good way to reduce communication cost, it is not optimal. If we cannot explicitly represent B, but instead must partition A directly, graph partitioning is suboptimal because A is rectangular. Instead, we propose using *hypergraph* partitioning.

Hypergraph models [10] address many of the drawbacks of graph models. As in the graph model, hypergraph vertices represent a row (or column) in the matrix. However, hypergraph edges (hyperedges) are sets of two *or more* related vertices. For sparse matrices, if a vertex is a matrix row, then the columns represent the hyperedges. The number of hyperedges cut by partition boundaries is an exact representation of communication volume, not merely an approximation [10].

Although hypergraph partitioning is NP-hard, good heuristic algorithms have been developed. The dominant algorithms are extensions of the multilevel partitioning algorithms for graph partitioning. Hypergraph partitioning's effectiveness has been demonstrated in many areas, including VLSI layout, sparse matrix decompositions, and database storage and data mining. Several (serial) hypergraph partitioners are available (e.g., hMETIS, PaToH, Mondriaan). Devine *et al* [14] are developing a parallel hypergraph partitioner as part of the Zoltan project for parallel load-balancing for scientific computing. With Rob Hoekstra (SNL), we have developed an interface between EpetraExt and Zoltan for general sparse matrices (hypergraphs). This code takes a sparse Epetra_CrsMatrix as input and returns a row map.

By reversing the role of nodes and hyperedges, a hypergraph partitioner can partition the columns of a matrix rather than the rows. The cut sizes (communication volume) for the column partition may be very different from that of the row partition (see [40, 10]), and it is difficult to predict which is better *a priori*. Therefore, we allow both distributions. Since Epetra is row-based, it is simplest to store matrices by rows. Our parPCx implementation contains a matrix class `AAT` that stores either A or A^T by rows. Note that A^T partitioned by rows corresponds to A partitioned by columns.

In the case where we explicitly form $B = ADA^T$, the row map (distribution) of B should equal the row map of A. Computing a column map (distribution) for A is not helpful, because it does not provide a way to distribute B.

We used parPCx for all the numerical experiments in this report. That version of the code uses a simple linear row map (that is, no hypergraph partitioning). The upcoming PCy version employs the EpetraExt/Zoltan hypergraph partitioning and will thus be much more communication effi cient.

6 Code Overview

6.1 PCx

Joe Czyzyk (Argonne), Sanjay Mehrotra (Northwestern), Michael Wagner (Cornell), and Stephen Wright (Argonne, later Wisconsin) developed PCx [13] in 1996. It implements an interior-point method with predictor-corrector algorithm for linear programming. The code consists of about 5000 lines of C code. In addition to the actual LP solver, there is a driver program, an MPS reader, and a pre-solver. PCx uses direct factorization and therefore needs a sparse Cholesky solver. PCx provides an abstract Cholesky solver interface. It can use any external Cholesky solver with such an interface. In particular, PCx provides interfaces for the Ng-Peyton and WSSMP (Gupta, IBM) direct solvers.

Michael Wagner at Cornell developed a partially parallel PCx version called pPCx [12]. It is based on the Psspd parallel Cholesky solver by Chunguang Sun [2].

Although pPCx has demonstrated good parallel performance on modest number of processors, it does not meet Sandia's needs. First, it does not have a clear path to large-scale parallelism, because Cholesky solvers are not scalable on distributed-memory machines. Second, it requires the storage of the entire constraint matrix on every processor. Third, neither pPCX nor PSSPD is actively maintained.

6.2 Trilinos

Our parallel version of PCx is based on Trilinos [23], a state-of-the-art parallel library for linear algebra. Trilinos is a collection of linear algebra packages developed at Sandia. It's written in C++ and provides both basic matrix and vector classes (Epetra), direct linear solvers (Amesos), and iterative solvers (AztecOO). ParPCx may use several other Trilinos packages in the future, including IFPACK for preconditioners, ML for multigrid solvers, and Belos for block Krylov solvers.

[2] There is no obvious published description of Psspd, but as of this writing, one can download a user's guide from www.cs.cornell.edu/Info/People/csun/psspd/psspd_guide.ps.

An important feature of Trilinos and, in particular, Epetra, is that the application can control the parallel data layout via so-called *maps*. Maps may also replicate data over some or all processors; a feature we used heavily.

6.3 parPCx

6.3.1 FY03 code development

During FY03 we developed an initial parallel code by interfacing pPCx with a collection of modern parallel linear solvers. This included IBM's PWSMP, using pPCx's native interface, as well an array of both direct and iterative solvers provided by the Trilinos suite, including Padma Ragavan's DSCPACK. The latter proved more challenging than initially expected, primarly because the Epetra component of Trilinos was undergoing heavy development and documentation and certain features useful to our application were sparse. We developed a generic Solver interface for Trilinos. Only the linear solves are parallelized. This code is referred to as qPCx.

6.3.2 FY04 code development

We improved the FY03 version, which evolved into parPCx. In FY04 we also began working on a fully-distributed-memory version, codenamed PCy. The PCy code is not based on pPCx. Rather, we started all over again with a clean and current version of PCx. (Many bug fixes had been made in PCx after pPCx split off.) The PCy code development required significant amounts of resources spent on software engineering to rewrite the PCx code. First, we had to convert the code from C to C++. Second, we had to replace all arrays representing vectors and matrices with the appropriate Epetra parallel vector and matrix classes. This was an intrusive, nontrivial process. We quickly realized that we were among the first applications to stress or even use certain Epetra features (e.g. rectangular matrics and local maps).

The current status (October 2004) is that PCy code is mostly finished, except a few features like dense column handling. Unfortunately, the code has not been debugged enough to produce reliable results yet, but it is close to being completed.

6.3.3 Solver interface

Both the fully distributed version of our code (PCy) and the version which only parallelizes the linear solves (qPCx) leverage the same linear solver interface. This dynamic object-oriented interface revolves around a generic Solver class, which encapsulates both direct and iterative solvers. Each suite of solvers requires a subclass derived from the generic solver. This subclass specifies solver-specific implementations of the generic methods.

The interface was designed to provide rapid integration of new linear solvers and preconditioners as they become available. This mechanism also allows parPCx to dynamically choose an appropriate linear solver. That feature is not available in PCx or Coleman et al.'s pPCx.

Interfacing a new solver package only requires the concrete implementation of a few methods in a Solver-derived class: SetupSolve, which given the D matrix corresponding to the current iteration of the PCx algorithm prepares the solver for subsequent solves, and Solve, which is used to perform repeated linear solves for a fixed D matrix. This generic interface works for both direct and iterative solvers; for example, during a call to SetupSolve, a direct solver might perform a Cholesky factorization, while an iterative solver might construct a preconditioner. The Solver class provides this level of flexibility via an abstract representation of the ADA^T matrix over which the solves occur. This representation is the other major component of our solver interface.

The AATtype class builds upon the abstract matrix representation facilities provided by Epetra. Matrices can be stored in a variety of ways. Direct solvers need an explicit ADA^T matrix to factor, while iterative solver only requires the means to multiply ADA^T by a vector. The latter can easily be achieved by three separate matrix-vector multiplications using A^T, A, and D. This can be further optimized since multiplications using both A and A^T can use the same set of data. AATtype addresses exactly this issue, by providing a multiplication interface to both A and A^T while relying on a single stored Epetra matrix object. We optimize the storage of this single matrix, which represents both A and A^T, during the partitioning phase of our code.

6.3.4 Aztec_ADAT and Aztec_Op

As mentioned earlier, iterative solvers do not necessarily need to explicitly construct ADA^T; however in this case, the construction of an effective preconditioner relies on an approximation of ADA^T. Such an approximation may lie anywhere in the spectrum between what is considered a trivial approximation, such as the diagonal of ADA^T to ADA^T in its entirety. Selecting a suitable approximation to ADA^T and designing a preconditioner based on this approximation are intimately related tasks, which are often problem-specific and seem to draw upon elements of art as well as science.

To facilitate experimentation in this vein, we have developed two interface classes to AztecOO, Trilinos's primary iterative linear solver. One version of the AztecOO interface class explicitly constructs ADA^T, much like its counterpart for direct solvers. This version, Solver_Aztec_ADAT, is designed mainly to serve as a platform for rapid experimentation, as once an appropriate approximation and preconditioner are chosen, ADA^T need not be computed explicitly. On the other hand of the spectrum, we provide another version, Solver_Aztec_Op, that performs multiplications over ADA^T by storing just the data for A and D; this version approximates ADA^T by using only its diagonal, which is relatively easy to compute and is space efficient. Both classes are designed to support user-defined precon-

ditioners and endow one with the flexibility of choosing between a top-down or bottom-up approach to preconditioner design.

6.4 How to use the code

The parPCx codes (qPCx and PCy) are fully backwards compatible with PCx, and can be used as a direct replacement for PCx. One can either use the driver program, which reads a problem from an MPS file, or one can use the subroutine interface. In the latter case, the application must construct the specific data structures PCx uses. The PCy version also contains a parallel interface such that if the application already has data in distributed form, these can be passed directly. This option requires the application to create the LP data using Epetra data types.

The code is built on Trilinos 4.0 (or later). The following Trilinos libraries are required: Epetra, EpetraExt, AztecOO, and Amesos. Parallel direct solution requires at least one of the solvers DSCPACK, MUMPS, or SuperLU/SuperLUdist. The user must correctly configure Trilinos and Amesos. The parPCx code comes in a single source directory with a makefile, which has to be slightly adapted to local conditions. Typing 'make' will build both the library (libparPCx.a) and the driver (parPCx). The driver accepts two option: `-s solver` specifies the desired solver, and `-f file` specifies an MPS input file. Valid solver options are currently Dscpack, Mumps, and Klu (direct), and Aztec_adat and Aztec_op (iterative). Preconditioning options must currently be changed in the source code (Solver_AztecADAT) but this will change in the future. It is also fairly easy to add new preconditioners that conform to the Epetra operator interface.

6.5 PICO modifications

We have modified the core and integer programming modules of PICO to include a new ramp-up procedure to exploit procedural (as opposed to subproblem) parallelism early in the search. This is a natural way to integrate parallel (root) LP solves into PICO. We use ramp up early in a branch-and-bound computation when there isn't sufficient subproblem parallelism to keep a large number of processors busy. For example, at the root of the search tree, there is only one problem. In this setting, it is best to evaluate subproblems in parallel, parallelizing individual steps including bounding, cut generation, gradient calculations, searching for feasible solutions, etc. A parallel LP solver naturally fits into this new ramp up, certainly for the root solve, and possibly even for early subproblems if there are a significant number of new cuts (additional constraints). When there are enough independent subproblems relative to the number of processors (or there is insufficient parallelism at the subproblem level), the computation switches to tree-level parallelism where processors evaluate independent subproblems in parallel. Because processors stay in lock step during ramp up, PICO makes this transition without communication and with an initially perfect load balance.

Problem	rows	cols	KLU (1)	DSC (1)	DSC (2)	DSC (4)	DSC (8)	DSC (16)
25fv47	821	1571	3.26	3.36	4.03	3.99	4.21	4.46
maros-r7	3136	9408	48.5	20.2	16.7	13.8	11.6	11.7

Figure 5. Time (sec.) for parallel solution for different linear solvers. Number of processors in parentheses.

To integrate parPCx into PICO, we also built an (IBM Open-Source Initiative) interface for the solver and the extended PICO LP interface (of course, throwing exceptions for calls to methods that are only applicable to simplex methods). The solver is not fully debugged as of this writing, but likely will be before this report is published. Because parPCx uses the PCx interface, the OSI solver interface now enables access to the PCx serial solver, pPCx and parPCx for any code that uses this standard interface.

7 Parallel Results

The parPCx system is still under development so it is still too early to publish meaningful parallel results. However, for the purpose of this status report we have done some parallel runs. All results in this section were obtained with the qPCx code, where only the solver is parallel. The fully distributed memory version (PCy) should outperform qPCx, but we are still debugging this version.

All the results below use test problems from the Netlib LP collection [17], which despite its age is still a dominant LP benchmark suite. We used a Compaq Alpha cluster with 32 processors, of which we could use up to 16.

We examine the results with direct solvers here, as we already discussed the performance of iterative solvers in Section 4.1. Even though the only changes from PCx are the formation of the ADA^T matrix and the Cholesky solver, the solutions from parPCx sometimes differed slightly from PCx solutions. That is, parPCx did approach the same solution as PCx but due to small numerical differences the status of the final iterate was in a few cases classified as "unknown" or "infeasible" instead of "optimal". Also, a few problems did not converge or failed in other ways. There may be several causes, but the lack of special treatment of very small pivots is a likely factor. We therefore focus on cases where parPCx (qPCx) and PCx did produce the same answers. In Figure 5 we show the execution time for two test problems with varying number of processors using the DSCPACK linear solver.

For the 25fv47 problem there was no parallel speedup. This is a small problem, and it appears that the parallel overhead and data redistribution cost cancelled the small savings from computing the Cholesky factorization in parallel. For the larger maros-r7 problem,

there is some parallel speedup but it is poor. Clearly, we cannot expect linear speedup since only the Cholesky solver is parallel, but we had hoped for better speedup. We are currently investigating the cause. One contributing factor is that vectors are replicated across processors, and therefore require a lot of communication to update. There may also be inefficiencies in the Amesos interface layer, which sits between parPCx and the third-party linear solver (e.g., DSCPACK). We have also identified the sparse matrix-matrix multiplication as a slow part on some problems; apparently due to an inefficient implementation in EpetraExt. (For some problems, computing ADA^T took substantially longer than factoring ADA^T.)

We also attempted to run qPCx on some much larger problems (up to 100,000 variables), like CRE-B, but qPCx crashed with floating-point exception using parallel Cholesky (DSCPACK) so we were only able to perform serial solves (using KLU). We did obtain the optimal solution with the KLU solver and need to investigate why the code failed with DSCPACK. This shows that robustness is a serious issue in interior-point methods, even with direct solvers.

8 Future Work

Although many goals of this LDRD project have been accomplished, some work still remains before the parPCx/PCy codes become truly useful. A continuation of the project would need to tackle:

- Debug and test PCy. The fully distributed code still has bugs, and must be debugged and verified against the original PCx.
- Make all vectors distributed (parallel). Currently, most vectors are replicated across processors but this is easy to change using Epetra. Important to get good parallel speedup.
- Dense column handling using either the Sherman-Morrison-Woodbury formula or column splitting. This feature is missing in parPCx and only partially implemented in PCy. Important for performance.
- Profiling and performance optimization. The current code is too slow.
- Debug and test PICO/OSI interface.
- Implement cross-over operation. Essential for use in the MIP solver PICO.

A wish list for future expansion might include:

- Expand solver class to optionally solve augmented system via direct or iterative methods. This may be a better approach than the normal equations in many cases.

- Provide simpler methods to add new preconditioners.
- Implement LSQR for solving least-squares version of the augmented system. This will almost surely work better (more robust) than the current method (CG on the normal equations).
- Investigate better scalings to reduce condition number.
- Implement direct–iterative hybrid solver.
- Investigate new preconditioners. Good scalable parallel performance can only be achieved with iterative methods.
- Parallel presolve. Currently, the presolver is serial.

Some of our parallel code (in particular, the Solver class) may be useful in other, more general, interior-point method codes. For example, we could integrate our parallel solver with the OOQP convex QP package. This would require some modifi cations since the linear systems for QPs require the Hessian.

References

[1] V. Baryamureeba and T. Steihaug. Properties and computational issues of a preconditioner for interior point methods. Technical Report 180, Dept. of Informatics, Univ. of Bergen, Norway, 1999.

[2] J. Berry, L. Fleischer, W. E. Hart, and C. A. Phillips. Sensor placement in municipal water networks. In *Proceedings of the World Water and Environmental Resources Congress (electronic proceedings)*. ASCE, 2003.

[3] J. Berry, W. E. Hart, C. A. Phillips, and J. Uber. A general integer-programming-based framework for sensor placement in municipal water networks. In *Proceedings of the 6th Annual Symposium on Water Distribution System Analysis (electronic proceedings)*. ASCE, 2004.

[4] R. H. Bisseling, T. M. Doup, and L.D.J.C. Loyens. A parallel interior point algorithm for linear programming on a network of transputers. *Annals of Operations Research*, 43:51–86, 1993.

[5] Åke Björck. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, PA, 1996.

[6] E. G. Boman, D. Chen, B. Hendrickson, and S. Toledo. Maximum-weight-basis preconditioners. *Numerical Linear Algebra with Appl.*, 11(8–9):695–721, 2004.

[7] E. G. Boman and B. Hendrickson. Support theory for preconditioning. *SIAM J. on Matrix Anal. and Appl.*, 25(3):694–717, 2004. (Published electronically on 17 Dec 2003.).

[8] R. E. Carlson, M. A. Turnquist, and L. K. Nozick. Expected losses, insurability, and benefi ts from reducing vulnerability to attack. Technical Report SAND2004-0742, Sandia National Laboratories, 2004.

[9] R.D. Carr and G. Konjevod. Polyhedral combinatorics. In H.J. Greenberg, editor, *Tutorials on Emerging Methodologies and Applications in Operations Research*. Kluwer Academic Press, 2004.

[10] Ü. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Dist. Systems*, 10(7):673–693, 1999.

[11] D. Chen and S. Toledo. Vaidya's preconditioners: Implementation and experimental study. *ETNA*, 16, 2003. Available from http://etna.mcs.kent.edu.

[12] T. Coleman, J. Czyzyk, C. Sun, M. Wagner, and S.J. Wright. pPCx: parallel software for linear programming. In *Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1997.

[13] J. Czyzyk, S. Mehrotra, and S.J. Wright. PCx user's guide. Technical Report Tech. report OTC 96/01, Argonne National Lab, 1996.

[14] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002. http://www.cs.sandia.gov/Zoltan.

[15] J. Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. *SIAM Journal on Optimization*, 4(4):794–814, 1994.

[16] Jonathan Eckstein, Cynthia A. Phillips, and William E. Hart. PICO: An object-oriented framework for parallel branch-and-bound. In *Proc Inherently Parallel Algorithms in Feasibility and Optimization and Their Applications*, Elsevier Scientifi c Series on Studies in Computational Mathematics, pages 219–265, 2001.

[17] D.M. Gay. Electronic mail distribution of linear programming test problems. *COAL Newsletter, Math. Prog. Soc.*, 13:10–12, 2003.

[18] P. Gill, W. Murray, and M. Wright. *Practical Optimization*. Academic Press, 1981.

[19] P.E. Gill, W. Murray, D.B. Ponceleon, and M.A. Saunders. Preconditioners for indefi nite systems arising in optimization. *SIAM J. on Matrix Analysis*, 13(1):292–311, 1992.

[20] P.E. Gill, W. Murray, D.B. Ponceleon, and M.A. Saunders. Primal-dual methods for linear programming. *Math. Prog.*, 70(3):251–277, 1995.

[21] J. Gondzio and R. Sarkissian. Parallel interior-point solver for structured linear programs. *Mathematical Programming*, 96(3):561–584, 2003.

[22] M.D. Grigoriadis and L.G. Khachiyan. An interior-point method for bordered block-diagonal linear programs. *SIAM J. Optim.*, 6(4):913–932, 1996.

[23] Michael Heroux, Roscoe Bartlett, Vicki Howle, Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, Albuquerque, NM, 2003. http://software.sandia.gov/trilinos.

[24] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Sci. Comp.*, 22(6):2194–, 2001.

[25] J.J. Judice, J. Patricio, L.F. Portugal, M.G.C. Recende, and G. Veiga. A study of preconditioners for network interior point methods. *Computational Optimization and Applications*, 24(1):5–35, 2003.

[26] P. Koka, T. Suh, M Smelyanskiy, R. Grzeszczuk, and C. Dulong. Construction and performance characterization of parallel interior point solver on 4-way intel itanium 2 multiprocessor system. In *IEEE Seventh Annual Workshop on Workload Characterization*, 2004.

[27] S. Mehrotra and J. Wang. Conjugate gradient based implementation of interior point methods for network flow problems. In L. Adams and J. Nazareth, editors, *Linear and Nonlinear Conjugate Gradient Related Methods*, pages 124–142. SIAM, 1995.

[28] R.D.C. Monteiro, J.W. O'Neill, and T. Tsuchiya. Uniform boundedness of a preconditioned normal matrix used in interior-point methods. *SIAM J. Optim.*, 15(1):96–100, 2004.

[29] L.F. Portugal, M.G.C. Recende, G. Veiga, and J.J. Judice. A truncated primal-infeasible dual-feasible interior point network flow method. *Networks*, 35:91–108, 2000.

[30] M. Resende and G. Veiga. An effi cient implementation of a network interior point method. In D. Johnson and C. McGeoch, editors, *Network Flow and Matching: First DIMACS Implementation Challenge*, volume 12, pages 299–384, Providence, RI, 1993. AMS.

[31] M.A. Saunders. PDCO: primal-dual interior method for convex objectives. Software available at http://www.stanford.edu/group/SOL/software/pdco.html.

[32] M.A. Saunders and J.A. Tomlin. Solving regularized linear programs using barrier methods and KKT systems. Technical Report Report SOL 96-4, Dept. of EESOR, Stanford University, 1996.

[33] G. Schultz and R.R. Meyer. An interior-point method for block angular optimization. *SIAM J. Optim.*, 1(1):583–602, 1991.

[34] D. Spielman and S-H. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsifi cation, and solving linear systems. Manuscript available from www.arxiv.org/abs/cs.DS/0310051, 2003.

[35] D. Spielman and S-H. Teng. Solving sparse, symmetric, diagonally-dominant linear systems in time $0(m^{1.31})$. In *Proc. of FOCS'03*, pages 416–427, 2003.

[36] D.A. Spielman and S.H. Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. of the ACM*, 51(3):385–463, May 2004.

[37] Sivan Toledo. TAUCS - a library of sparse linear solvers. Software available at http://www.math.tau.ac/∼stoledo/taucs/.

[38] P. M. Vaidya. Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. Unpublished manuscript. A talk based on this manuscript was presented at the IMA Workshop on Graph Theory and Sparse Matrix Computations, Minneapolis, October 1991.

[39] R. Vanderbei. Symmetrical quasidefi nite matrices. *SIAM J. Optimization*, 5(1):100–113, Feb. 1995.

[40] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. Preprint 1238, Dept. Mathematics, Utrecht University, May 2002. To appear in *SIAM Review*, 2005.

[41] W. Wang and D.P. O'Leary. Adaptive use of iterative methods in interior point methods for linear programming. Tech. report CS-3650, Univ. of Maryland, 1995.

[42] J.-P. Watson, H. J. Greenberg, and W. E. Hart. A multiple-objective analysis of sensor placement optimization in water networks. In *Proceedings of the 6th Annual Symposium on Water Distribution System Analysis (electronic proceedings)*. ASCE, 2004.

[43] M. H. Wright. Interior methods for constrained optimization. In A. Iserles, editor, *Acta Numerica 1992*, pages 341–407. Cambridge University Press, 1992.

[44] S. Wright. *Primal-Dual Interior-Point Methods*. SIAM, Philadelphia, 1997.

DISTRIBUTION:

3 Prof. Ojas Parekh
Math/CS Department
400 Downman Dr.
Atlanta, GA 30322

3 Prof. Stephen J. Wright
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706

1 MS 9018
Central Technical Files, 8945-1

3 MS 1110
Erik Boman, 09215

3 MS 1110
Cynthia Phillips, 09215

1 MS 1110
Michael Heroux, 09214

3 MS 9159
Victoria Howle, 08962

1 MS 1110
Suzanne L. K. Rountree, 09215

2 MS 0899
Technical Library, 9616

1 MS 0123
D. Chavez, LDRD Offi ce, 1011