A Parallel Implementation of an Interior-Point Algorithm for Multicommodity Network Flows*

Jordi Castro[1]** and Antonio Frangioni[2]

[1] Statistics and Operations Research Dept.,
Universitat Politècnica de Catalunya
Pau Gargallo 5, 08028 Barcelona (Spain)
jcastro@eio.upc.es
[2] Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa (Italy)
frangio@di.unipi.it

Abstract. A parallel implementation of the specialized interior-point algorithm for multicommodity network flows introduced in [5] is presented. In this algorithm, the positive definite systems of each iteration are solved through a scheme that combines direct factorization and a preconditioned conjugate gradient (PCG) method. Since the solution of at least k independent linear systems is required at each iteration of the PCG, k being the number of commodities, a coarse-grained parallellization of the algorithm naturally arises. Also, several other minor steps of the algorithm are easily parallelized by commodity. An extensive set of computational results on a shared memory machine is presented, using problems of up to 2.5 million variables and 260,000 constraints. The results show that the approach is especially competitive on large, difficult multicommodity flow problems.

1 Introduction

Multicommodity flows are among the most challenging linear problems, due to the large size of these models in real world applications (e.g., routing in telecommunications networks). Indeed, these problems have been used to test the efficiency of early interior-point solvers for linear programming [1]. The need to solve very large instances has led to the development of both specialized algorithms and parallel implementations.

In this paper, we present a parallel implementation of a specialized interior-point algorithm for multicommodity flows [5]. In this approach, the block-angular

* This work has been supported by the European Center for Parallelism of Barcelona (CEPBA).
** Author supported by CICYT Project TAP99-1075-C02-02.

J.M.L.M. Palma et al. (Eds.): VECPAR 2000, LNCS 1981, pp. 301–315, 2001.

structure of the coefficient matrix is exploited for performing in parallel the solution of small linear systems related to the different commodities, unlike general-purpose parallel interior-point codes [2,8,18] where the parallelization effort is focused on the Cholesky factorization of one large system. This has already been proposed [17,10,14]; however, all the previous approaches require to compute and factorize the Schur complement. This can become a significant serial bottleneck, since this matrix is usually prohibitively dense. Although this bottleneck can be partly eluded by using parallel linear algebra routines, our approach takes a more radical route by avoiding to form the Schur complement, and using an iterative method instead. There have been other proposals along these lines [23,15], but limited to the sequential case; also, so far no results have been shown for these algorithms. The implementation presented in this paper significantly improves on the preliminary one described in [6]. There, only some of the major routines were parallelized, and less attention was paid to communication and data distribution. Working on these details allowed us to obtain new and better computational results.

From the multicommodity point of view, this approach is different from most other parallel solvers [7,16,20,26,22,13] in that is not based on a decomposition approach. The structure of the multicommodity flow problem has led to a number of specialized algorithms, most of which share the idea of decomposing in some way the problem into a set of smaller independent problems. These are all iterative methods, where at each step the subproblems are solved, and their results are used in some way to modify the subproblems to be solved at the next iteration. Hence, these approaches are naturally suited for coarse-grained parallelization. Parallel price-directive decomposition approaches have been proposed based on bundle methods [7,20], analytic center methods [13] or linear-quadratic penalty functions [22]. Parallel resource-directive approaches are described in [16]. Finally, experiences with a parallel interior-point decomposition method are presented in [26]. A discussion of these and other parallel decomposition approaches can be found in [7]. A general description of the parallelization of mathematical programming algorithms can be found in [3,24].

The paper is organized as follows. Section 2 presents the formulation of the problem to be solved. Section 3 outlines the specialized interior-point algorithm for multicommodity flows proposed in [5], including a brief description of the general path-following method. Section 4 deals with the parallelization issues of the algorithm. Finally, Section 5 presents and discuss the computational results.

2 Problem Formulation

The multicommodity flow problem requires to find the least-cost routing of a set of k commodities through a network of m nodes and n arcs, where the arcs have an individual capacity for each commodity, and a mutual capacity for all the commodities. The node-arc formulation of the problem is

$$\begin{aligned} &\min \textstyle\sum_{i=1}^{k} c^i x^i \\ &\text{s.t.} \quad \left[\begin{array}{cccccc} E & 0 & \dots & 0 & 0 \\ 0 & E & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & E & 0 \\ \hline I & I & \dots & I & I \end{array}\right] \left[\begin{array}{c} x^1 \\ x^2 \\ \vdots \\ x^k \\ \hline x^0 \end{array}\right] = \left[\begin{array}{c} b^1 \\ b^2 \\ \vdots \\ b^k \\ \hline u \end{array}\right] \\ &\qquad 0 \le x^0 \le u \ , \qquad 0 \le x^i \le u^i \quad i = 1 \dots k. \end{aligned} \tag{1}$$

Vectors $x^i \in \mathbb{R}^n$ are the flow arrays for each commodity, while $x^0 \in \mathbb{R}^n$ are the slacks of the mutual capacity constraints. $E \in \mathbb{R}^{m \times n}$ is the node-arc incidence matrix of the underlying directed graph, while I denotes the $n \times n$ identity matrix. We shall assume that E is a full row-rank matrix: this can always be guaranteed by removing any of the redundant node balance constraints. $c^i \in \mathbb{R}^n$ and $u^i \in \mathbb{R}^n$ are respectively the flow cost vector and the individual capacity vector for commodity i, while $u \in \mathbb{R}^n$ is the vector of the mutual capacities. Finally, $b^i \in \mathbb{R}^m$ is the vector of supplies/demands for commodity i at the nodes of the network.

The multicommodity flow problem is a linear program with $\bar{m} = km + n$ constraints and $\bar{n} = (k+1)n$ variables. In some real-world models, k can be very large: for instance, in many telecommunication problems a commodity represents the flow of data/voice between two given nodes of the network, and therefore $k \approx m^2$. Thus, the resulting linear program can be huge even for graphs of moderate size. However, the coefficient matrix of the problem is highly structured: it has a block-staircase form, each block being a node-arc incidence matrix. Several methods have been proposed which exploit this structure; one is the specialized interior-point algorithm described in the next section.

3 A Specialized Interior-Point Algorithm

In [5], a specialized interior-point algorithm for multicommodity flows has been presented and tested. This algorithm, and the code that implements it, will be referred to as IPM.

IPM is a specialization of the path-following algorithm for linear programming [27]. Let us consider the following linear programming problem in primal form

$$\min \{\ cx \ :\ Ax = b,\ x + s = u,\ x, s \ge 0\ \}\ , \tag{2}$$

where $x \in \mathbb{R}^{\bar{n}}$ and $s \in \mathbb{R}^{\bar{n}}$ are respectively the primal variables and the slacks of the box constraints, $u \in \mathbb{R}^{\bar{n}}$, $c \in \mathbb{R}^{\bar{n}}$ and $b \in \mathbb{R}^{\bar{m}}$ are respectively the upper bounds, the cost vector and the right hand side vector, and $A \in \mathbb{R}^{\bar{m} \times \bar{n}}$ is a full row-rank matrix. The dual of (2) is

$$\max \{\ yb - wu \ :\ yA + z - w = c,\ z, w \ge 0\ \}\ , \tag{3}$$

where $y \in \mathbb{R}^{\bar{m}}$, $z \in \mathbb{R}^{\bar{n}}$ and $w \in \mathbb{R}^{\bar{n}}$ are respectively the dual variables of the structural constraints $Ax = b$, the dual slacks and the dual variables of the box constraints $x \le u$.

Replacing the inequalities in (2) by a logarithmic barrier in the objective function, with parameter μ, the KKT optimality conditions of the resulting problem are

$$\begin{array}{ll} r_{xz} \equiv & \mu e - XZe = 0 \\ r_{sw} \equiv & \mu e - SWe = 0 \\ r_b \equiv & b - Ax = 0 \\ r_c \equiv & c - (yA + z - w) = 0 \\ r_u \equiv & u - x - s = 0 \\ & (x, s, z, w) \geq 0 \ , \end{array} \tag{4}$$

where e is the vector of 1's of proper dimension, and each uppercase letter corresponds to the diagonal matrix having as diagonal elements the entries of the corresponding lowercase vector. In the algorithm we impose $r_u = 0$, i.e. $s = u - x$, thus eliminating $\bar{n}$ variables.

The (unique) solutions of (4) for each possible $\mu > 0$ describe a continuous trajectory, known as the *central path*, which, as μ tends to 0, converges to the optimal solutions of (2) and (3). A path-following algorithm attempts to reach close to these optimal solutions by following the central path. This is done by performing a damped version of Newton's iteration applied to the nonlinear system (4), as shown in (5). The steplengths α_P and α_D are the maximum allowable values in the range $(0, 1]$ such that the new iterate will keep on being strictly positive (note that when $\alpha_P = \alpha_D = 1$ the algorithm performs a pure Newton iteration). A more detailed description of the algorithm can be found in many linear programming textbooks, e.g. [27].

Procedure *PathFollowing*(A, b, c, u):
  Initialize $x > 0, s > 0, y, z > 0, w > 0$;
  while (x, s, y, z, w) is not optimum **do**
    $\Theta = (X^{-1}Z + S^{-1}W)^{-1}$;
    $r = S^{-1}r_{sw} + r_c - X^{-1}r_{xz}$;
    $(A\Theta A^T)\Delta y = r_b + A\Theta r$;
    $\Delta x = \Theta(A^T \Delta y - r)$;
    $\Delta w = S^{-1}(r_{sw} + W\Delta x)$;
    $\Delta z = r_c + \Delta w - A^T \Delta y$;
    Compute $\alpha_P \in (0, 1], \alpha_D \in (0, 1]$;
    $x \leftarrow x + \alpha_P \Delta x$;
    $(y, z, w) \leftarrow (y, z, w) + \alpha_D(\Delta y, \Delta z, \Delta w)$;

(5)

The main computational burden of the algorithm is the solution of the system

$$(A\Theta A^T)\Delta y = r_b + A\Theta r \equiv \bar{b} \ . \tag{6}$$

Note that $A\Theta A^T$ is symmetric and positive definite, as Θ is clearly a positive definite diagonal matrix. Usually, interior-point codes solve (6) through a Cholesky factorization, preceeded by a permutation of the columns of A aimed at

minimizing the fill-in effect. Several effective heuristics have been developed for computing such a permutation. Unfortunately, when A is the constraints matrix of (1), the Cholesky factors of $A\Theta A^T$ turn out to be rather dense anyway [5].

However, the structure of A can be used to solve (6) without computing the factorization of $A\Theta A^T$. Note that Θ is partitioned into the k blocks Θ^i, $i = 1 \dots k$, one for each commodity, plus the block Θ^0 corresponding to the slack variables x^0 of the mutual capacity constraints. Hence,

$$A\Theta A^T = \left[\begin{array}{c|c} B & C \\ \hline C^T & D \end{array}\right] = \left[\begin{array}{ccc|c} E\Theta^1 E^T & \dots & 0 & E\Theta^1 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & E\Theta^k E^T & E\Theta^k \\ \hline \Theta^1 E^T & \dots & \Theta^k E^T & \Theta^0 + \sum_{i=1}^k \Theta^i \end{array}\right], \tag{7}$$

i.e., B is the block diagonal matrix having the $m \times m$ matrices $B_i = E\Theta^i E^T$, $i = 1 \dots k$, as diagonal elements, and

$$C^T = \left[C_1^T \dots C_k^T\right] = \left[\Theta^1 E^T \dots \Theta^k E^T\right] .$$

Exploiting (7), and partitioning the vectors Δy and $\bar{b}$ accordingly, the solution of (6) is reduced to

$$\left(D - \sum_{i=1}^k C_i^T B_i^{-1} C_i\right) \Delta y^0 = \bar{b}^0 - \sum_{i=1}^k C_i^T B_i^{-1} \bar{b}^i \equiv \beta^0 \tag{8}$$

$$B_i \Delta y^i = (\bar{b}^i - C_i \Delta y^0) \equiv \beta^i, \qquad i = 1 \dots k . \tag{9}$$

The matrix

$$H = D - C^T B^{-1} C = D - \sum_{i=1}^k C_i^T B_i^{-1} C_i \tag{10}$$

is known as the Schur complement.

Thus, (6) can be solved by means of (8), involving the Schur complement H, followed by the k subsystems (9) involving the matrices B_i. The latter step can be easily parallelized. However, solving (8) with a direct method, as advocated in [17,10], requires forming and factorizing H. As shown in [5], this matrix typically becomes rather dense, hence such a direct approach may become computationally too expensive. Furthermore, it represents a formidable serial bottleneck for a parallel implementation of the code. As suggested in [17], this bottleneck can be reduced by using parallel linear algebra routines [2,8,18]. However, it is also possible to avoid forming H at all, solving (8) by means of an iterative algorithm.

Since H is symmetric and positive definite, a preconditioned conjugate gradient (PCG) method can be used. In [5], a family of preconditioners is proposed, based on the following characterization of the inverse of H:

$$H^{-1} = \left(\sum_{i=0}^{\infty} (D^{-1}Q)^i\right) D^{-1} \qquad \text{where} \qquad Q = \sum_{i=1}^k C_i^T B_i^{-1} C_i \tag{11}$$

A preconditioner for (8) can be obtained by truncating the above power series at the h-th term. Clearly, the higher h, the better the preconditioning will be, and the fewer PCG iterations will be required. However, preconditioning one vector requires solving $k \times h$ linear systems involving the matrices B_i, thereby increasing the cost of each PCG iteration. The best trade-off between the reduction of the iterations count and the cost of each iteration is $h = 0$, corresponding to the diagonal preconditioner D^{-1} [5].

The IPM code, implementing this algorithm, has shown to be competitive with a number of other sequential approaches [5]. It is written mainly in C, with only the Cholesky factorization routines (devised by E. Ng and B. Peyton [21]) coded in Fortran. Both the sequential and parallel versions can be freely obtained for academic purposes from

`http://www-eio.upc.es/~jcastro/software.html`.

4 Parallelization of the Algorithm

The solution of (6) is by far the most expensive procedure in the interior-point algorithm, consuming up to 97% of the total execution time for large problems. With the above approach, this can be accomplished by means of the following steps:

- Factorization of the k matrices B_i; note that the current implementation uses sequential Cholesky solvers, but parallel Cholesky solvers could be used for increasing the degree of parallelism of the approach.
- Computation of $\beta^0 = \bar{b}^0 - \sum_{i=1}^{k} C_i^T B_i^{-1} \bar{b}^i$, which requires k backsolves on the factorizations of B_i and matrix-vector products of the form $C_i^T v^i$.
- For each iteration of the PCG, computation of $(D - \sum_{i=1}^{k} C_i^T B_i^{-1} C_i)v$, which requires backsolves on the factorizations of B_i and matrix-vector products of the form $C_i v^i$ and $C_i^T v^i$.
- Computation of $\beta^i = \bar{b}^i - C_i \Delta y^0$, which requires matrix-vector products of the form $C_i v^i$.
- Solution of the systems $B_i \Delta y^i = \beta^i$.

Hence, most of the parallelization effort boils down to performing in parallel the factorization of the B_is, backward and forward substitution with these factorizations and matrix-vector products involving C_i or C_i^T. Thus, there is no need for sophisticated implementations of parallel linear algebra routines. Note that higher-order preconditioners ($h > 0$) would complicate somehow the above scheme, but the basic blocks would remain the same.

Although the above procedures are by far the most important, a number of other minor steps can be easily parallelized, such as the computation of the other primal and dual directions (Δx^i, Δz^i, Δw^i), the computation of the primal and dual steplenghts α_P and α_D, the updating of the current primal and dual solution, the computation of the primal and dual objective function values and so on. It is easy to see that all the data concerning one given commodity i (x^i, c^i, u^i, y^i, w^i ...) can be stored in the local memory of the one processor that is in charge

of that commodity, and it is never required by other processors. This ensures a good "locality" of data, and a low need for inter-processor communication. It should also be noted that the number of operations required for each commodity is the same, which guarantees the load balancing between processors, at least as long as the number of commodities assigned to each processor is the same.

4.1 Parallel Programming Environment

The parallel version of the IPM code, pIPM, has been developed on the Silicon Graphics Origin2000 (SGI O2000) server located at the European Center for Parallelism of Barcelona (CEPBA), running an IRIX64 6.5 Unix operating system. The SGI O2000 offers both message-passing and shared-memory programming paradigms, although the main memory is physically distributed among the processors. The server has 64 MIPS R10000 processors running at 250Mhz, each of them with 32+32Kb L1 cache and 4Mb L2 cache and credited of 14.7 SPECint95 and 24.5 SPECfp95. A total of 8Gb of memory is distributed among these processing elements. This computer appeared at position 275 of the TOP500 November 1998 supercomputer sites list [11].

The default programming style supported by the SGI O2000 is a custom shared-memory version of C [25], with parallel constructs specified by means of compiler directives (`#pragma`s). Both OpenMP and SGI-specific pragmas are supported by the SGI O2000, but we mainly used the SGI-specific ones for the current version of pIPM. Placement of the memory on the processors and communication is hidden to the programmer and automatically performed by the system. The main advantage of this choice is ease of portability: existing codes can be parallelized with a limited effort. It is even possible to avoid maintaining two different versions (sequential and parallel) of the same code, which is important to optimize the development efforts.

However, this programming style also has a number of drawbacks, mainly a limited control over memory ownership and limited support for vector-broadcast and vector-reduce operations. Placement of the data structures in the local memory of the processors can be only partly (and indirectly) influenced by the programmer. Also, the granularity of memory placement is that of the virtual memory pages (16K) rather than that of the individual data structures. All this can result in cache misses and page faults from the local memory of each processor, decreasing the performance of the parallel codes. Although advanced directives allow a more detailed control over these features, the use of those directives requires a more extensive rewriting of the code, thus loosing part of the benefits in terms of portability and ease of maintenance. Because of that, the computational results presented in Section 5 were obtained with the default data distribution provided by the system (the same used in [2]). However, the assignment of commodities to processors was optimized for this distribution, hopefully limiting the possible negative effects. The limited support for broadcast/reduce operations is understandable in a shared-memory oriented language; however, it may result in poorer performances for codes, like pIPM, where these operations amount at almost the totality of the communication time.

5 Computational Results

5.1 The Instances

Three sets of multicommodity instances were used for the computational experiments. The first is made up of 18 problems obtained with an improved version of Ali and Kennington's Mnetgen generator [12]. These instances are very large (up to about 2.5 millions of variables and 260,000 constraints), with a number of commodities which varies from very few (8) to quite many (512). This is useful for characterizing the trends in the performances of the code as the number of commodities varies [7,12].

The second set consists of ten of the PDS (Patient Distribution System) problems. These problems arise from a logistic model for evacuating patients from a place of military conflict. The different instances arise from the same basic scenario by varying the time horizon, i.e., the number of days covered by the model. The PDS problems have been considered, until recently, essentially impossible to solve with a high degree of accuracy. Although this has changed, they are still quite challenging multicommodity instances.

The third set of problems is made of the four Tripart problems and of the Gridgen1 problem. These instances were obtained respectively with the Tripartite generator and with a multicommodity version of the well-known Gridgen single-commodity flow generator [4]. These are very difficult multicommodity flow instances, as shown in Section 5.3.

The dimensions of each problem are reported in Tables 1, 2 and 3. Columns "m", "n", and "k" show the number of nodes, arcs, and commodities. Columns "$\bar{n}$" and "$\bar{m}$" give the number of variables and constraints of the linear problem. All the instances can be downloaded from

`http://www.di.unipi.it/di/groups/optimize/Data.`

5.2 Performance Measures

The following well-known performance measures [3] will be considered for assessing the performances of pIPM. Denoting by T_p the execution time obtained with p processors, the *speedup* S_p with p processors can be defined as $S_p = T_1/T_p$. The fraction of the sequential execution time consumed in the parallel region of the code will be denoted by f; values of f close to 1 are necessary in order to obtain good speedups, as demonstrated by *Amdahl's law*

$$S_p \leq \overline{S_p} = \frac{1}{f/p + (1-f)} \leq \frac{1}{(1-f)} \; .$$

The *efficiency* with p processors is

$$E_p = \frac{S_p}{p} \leq \overline{E_p} = \frac{\overline{S_p}}{p} \; .$$

E_p represents the fraction of the time that a particular processor (of the p available) is usefully employed during the execution of the algorithm. $\overline{S_p}$ and $\overline{E_p}$

are respectively the *ideal* speedup and efficiency, the maximum ones that can be obtained due to the inherent serial bottlenecks in the algorithm.

Another interesting performance measure is the *absolute* speedup, obtained by replacing T_1 with the execution time of the *best* serial algorithm known. This is usually difficult to obtain, and it will be discussed separately.

5.3 The Results

Tables 1, 2 and 3 show the computational results obtained. Columns "IP" and "PCG" report the total number of interior-point and PCG iterations, respectively. Column "f" gives the fraction of the total sequential time consumed in the parallel region of the code. Column "p" gives the number of processors used in the execution. "T_p" denotes the execution (wall-clock) time, excluding initializations. Columns "S_p" and "E_p" give respectively the observed speedups and efficiencies, while columns "$\overline{S_p}$" and "$\overline{E_p}$" report their ideal values.

Analyzing the results, the following trends emerge:

- f is always fairly large, and increases with the problem size; the largest problems attain very high ideal efficiencies. This indicates that the approach has a good potential for scalability, at least in theory, for very large scale problems.
- For fixed p and k, E_p almost always increases with the size of the underlying network, in all three groups of instances. This is reasonable: the computational burden of the PCG iteration grows quadratically with the number of nodes, while the communication cost grows only linearly. This seems to indicate that the approach is especially suited for problems where the size of the network is large w.r.t. the number of commodities. Remarkably, IPM has been shown to be particularly efficient, at least w.r.t. decomposition approaches, exactly for this kind of instances [12].
- Keeping p and the size of the network fixed, E_p initially increases with k; however for "large" values of k E_p stalls, and may even decrease. This phenomenon, clearly visible in the Mnetgen results, is difficult to explain. For fixed p, increasing k can, in theory, only increase the fraction of time that is spent in the parallel part of the algorithm, while the sequential bottleneck and the communication requirements should remain the same. Indeed, $\overline{E_p}$ is monotonically nondecreasing with k. This decrease in efficiency is most likely an effect of the page-based memory placement, which may cause data logically pertaining to one processor to be physically located on another.
- For any fixed instance, E_p obviously decreases as p increase; unfortunately, the decrease is much faster than that predicted by $\overline{E_p}$, so that the gap between E_p and $\overline{E_p}$ increases with p. However, for fixed p the gap decreases when the size of the network increase, and a similar—although less clear—trend seems to exist w.r.t. k. Thus, whatever mechanism be responsible for this discrepancy between E_p and $\overline{E_p}$, its effects seem to lessen as the instances grow larger.

Table 1. Dimensions and results for the Mnetgen problems.

	m	n	k	$\bar{n}$	$\bar{m}$	f	IP	PCG	p	T_p	S_p	$\overline{S_p}$	E_p	$\overline{E_p}$
128-8	128	1089	8	9801	2113	92.2	42	831	1	3.2	1.0	1.0	1.0	1.0
									8	2.1	1.5	5.2	0.2	0.6
128-16	128	1114	16	18938	3162	95.1	48	2530	1	14.3	1.0	1.0	1.0	1.0
									8	7.7	1.9	6.0	0.2	0.7
									16	8.0	1.8	9.2	0.1	0.6
128-32	128	1141	32	37653	5237	95.4	56	2355	1	32.1	1.0	1.0	1.0	1.0
									8	12.9	2.5	6.1	0.3	0.8
									16	13.8	2.3	9.5	0.1	0.6
									32	19.6	1.6	13.2	0.1	0.4
128-64	128	1171	64	76115	9363	97.1	72	5480	1	139.2	1.0	1.0	1.0	1.0
									8	39.7	3.5	6.7	0.4	0.8
									16	34.7	4.0	11.1	0.3	0.7
									32	28.6	4.9	16.9	0.2	0.5
									64	40.3	3.5	22.6	0.1	0.4
128-128	128	1204	128	155316	17588	96.6	85	5033	1	409.2	1.0	1.0	1.0	1.0
									8	74.4	5.5	6.5	0.7	0.8
									16	122.8	3.3	10.6	0.2	0.7
									32	122.7	3.3	15.6	0.1	0.5
									64	73.3	5.6	20.4	0.1	0.3
256-8	256	2165	8	19485	4213	95.6	57	2713	1	20.7	1.0	1.0	1.0	1.0
									8	8.3	2.5	6.1	0.3	0.8
256-16	256	2308	16	39236	6404	96.5	59	3465	1	58.0	1.0	1.0	1.0	1.0
									8	21.0	2.8	6.4	0.3	0.8
									16	21.3	2.7	10.5	0.2	0.7
256-32	256	2314	32	76362	10506	97.3	67	5438	1	252.2	1.0	1.0	1.0	1.0
									8	52.6	4.8	6.7	0.6	0.8
									16	44.2	5.7	11.4	0.4	0.7
									32	54.6	4.6	17.4	0.1	0.5
256-64	256	2320	64	150800	18704	98.0	80	7644	1	757.3	1.0	1.0	1.0	1.0
									8	128.5	5.9	7.0	0.7	0.9
									16	93.7	8.1	12.3	0.5	0.8
									32	99.1	7.6	19.8	0.2	0.6
									64	169.3	4.5	28.3	0.1	0.4
256-128	256	2358	128	304182	35126	98.8	98	12535	1	2672.1	1.0	1.0	1.0	1.0
									8	351.3	7.6	7.4	1.0	0.9
									16	298.7	8.9	13.6	0.6	0.8
									32	257.0	10.4	23.3	0.3	0.7
									64	263.5	10.1	36.4	0.2	0.6
256-256	256	2204	256	566428	67740	98.9	107	16901	1	6725.1	1.0	1.0	1.0	1.0
									8	1219.7	5.5	7.4	0.7	0.9
									16	763.4	8.8	13.7	0.6	0.9
									32	502.0	13.4	23.9	0.4	0.7
									64	477.9	14.1	37.8	0.2	0.6
512-8	512	4373	8	39357	8469	96.4	66	3870	1	90.5	1.0	1.0	1.0	1.0
									8	22.9	4.0	6.4	0.5	0.8
512-16	512	4620	16	78540	12812	97.6	73	5364	1	322.3	1.0	1.0	1.0	1.0
									8	72.0	4.5	6.8	0.6	0.9
									16	63.1	5.1	11.8	0.3	0.7
512-32	512	4646	32	153318	21030	98.8	103	22460	1	2721.4	1.0	1.0	1.0	1.0
									8	454.7	6.0	7.4	0.7	0.9
									16	299.3	9.1	13.6	0.6	0.8
									32	289.3	9.4	23.3	0.3	0.7
512-64	512	4768	64	309920	37536	99.2	95	27004	1	9244.5	1.0	1.0	1.0	1.0
									8	1271.5	7.3	7.6	0.9	0.9
									16	702.8	13.2	14.3	0.8	0.9
									32	507.9	18.2	25.6	0.6	0.8
									64	563.8	16.4	42.6	0.3	0.7
512-128	512	4786	128	617394	70322	99.3	112	28631	1	19385.9	1.0	1.0	1.0	1.0
									8	3237.0	6.0	7.6	0.7	1.0
									16	1780.6	10.9	14.5	0.7	0.9
									32	1271.5	15.2	26.3	0.5	0.8
									64	848.5	22.8	44.4	0.4	0.7
512-256	512	4810	256	1236170	135882	99.5	130	32676	1	43251.2	1.0	1.0	1.0	1.0
									8	7401.6	5.8	7.7	0.7	1.0
									16	5306.7	8.2	14.9	0.5	0.9
									32	2783.7	15.5	27.7	0.5	0.9
									64	2205.9	19.6	48.7	0.3	0.8
512-512	512	4786	512	2455218	266930	99.6	194	48229	1	135753.7	1.0	1.0	1.0	1.0
									8	25257.7	5.4	7.8	0.7	1.0
									16	14198.4	9.6	15.1	0.6	0.9
									32	8325.3	16.3	28.5	0.5	0.9
									64	5226.0	26.0	51.1	0.4	0.8

Table 2. Dimensions and results for the PDS problems.

	m	n	k	$\bar{n}$	$\bar{m}$	f	IP	PCG	p	T_p	S_p	$\overline{S_p}$	E_p	$\overline{E_p}$
PDS1	126	372	11	4464	1758	83.3	30	169	1	0.7	1.0	1.0	1.0	1.0
									6	0.5	1.3	3.3	0.2	0.5
									11	0.7	0.9	4.1	0.1	0.4
PDS10	1399	4792	11	57504	20181	94.7	66	1107	1	44.8	1.0	1.0	1.0	1.0
									6	25.3	1.8	4.7	0.3	0.8
									11	24.6	1.8	7.2	0.2	0.7
PDS20	2857	10858	11	130296	42285	96.6	69	1911	1	254.1	1.0	1.0	1.0	1.0
									6	70.9	3.6	5.1	0.6	0.9
									11	62.6	4.1	8.2	0.4	0.7
PDS30	4223	16148	11	193776	62601	97.9	92	3835	1	777.1	1.0	1.0	1.0	1.0
									6	206.4	3.8	5.4	0.6	0.9
									11	189.2	4.1	9.1	0.4	0.8
PDS40	5652	22059	11	264708	84231	97.9	73	1872	1	1288.1	1.0	1.0	1.0	1.0
									6	258.4	5.0	5.4	0.8	0.9
									11	194.1	6.6	9.1	0.6	0.8
PDS50	7031	27668	11	332016	105009	98.8	100	4711	1	3486.4	1.0	1.0	1.0	1.0
									6	727.3	4.8	5.7	0.8	0.9
									11	530.1	6.6	9.8	0.6	0.9
PDS60	8423	33388	11	400656	126041	99.0	106	5215	1	6262.0	1.0	1.0	1.0	1.0
									6	1252.4	5.0	5.7	0.8	1.0
									11	745.4	8.4	10.0	0.8	0.9
PDS70	9750	38396	11	460752	145646	99.2	116	7015	1	10873.8	1.0	1.0	1.0	1.0
									6	2112.2	5.1	5.8	0.9	1.0
									11	1268.5	8.6	10.2	0.8	0.9
PDS80	10989	42472	11	509664	163351	99.2	107	3768	1	8855.0	1.0	1.0	1.0	1.0
									6	1726.3	5.1	5.8	0.9	1.0
									11	1093.8	8.1	10.2	0.7	0.9
PDS90	12186	46161	11	553932	180207	99.4	135	9357	1	20784.3	1.0	1.0	1.0	1.0
									6	3950.5	5.3	5.8	0.9	1.0
									11	2447.8	8.5	10.4	0.8	0.9

Table 3. Dimensions and results for the Tripart and Gridgen problems.

	m	n	k	$\bar{n}$	$\bar{m}$	f	IP	PCG	p	T_p	S_p	$\overline{S_p}$	E_p	$\overline{E_p}$
Tripart1	192	2096	16	35632	5168	93.6	65	3733	1	34.9	1.0	1.0	1.0	1.0
									4	21.3	1.6	3.4	0.4	0.8
									8	17.9	1.9	5.5	0.2	0.7
									16	19.6	1.8	8.2	0.1	0.5
Tripart2	768	8432	16	143344	20720	91.8	63	2652	1	156.6	1.0	1.0	1.0	1.0
									4	71.6	2.2	3.2	0.5	0.8
									8	55.4	2.8	5.1	0.4	0.6
									16	60.3	2.6	7.2	0.2	0.4
Tripart3	1200	16380	20	343980	40380	94.9	84	9343	1	1140.7	1.0	1.0	1.0	1.0
									4	408.4	2.8	3.5	0.7	0.9
									10	300.5	3.8	6.9	0.4	0.7
									20	304.8	3.7	10.2	0.2	0.5
Tripart4	1050	24815	35	893340	61565	95.6	96	8498	1	3273.2	1.0	1.0	1.0	1.0
									5	893.7	3.7	4.3	0.7	0.9
									7	721.5	4.5	5.5	0.6	0.8
									35	601.1	5.4	14.0	0.2	0.4
Gridgen1	1025	3072	320	986112	331072	99.5	173	49981	1	37234.9	1.0	1.0	1.0	1.0
									8	10533.2	3.5	7.7	0.4	1.0
									16	7678.7	4.8	14.9	0.3	0.9
									32	4426.5	8.4	27.7	0.3	0.9
									64	3248.6	11.5	48.7	0.2	0.8

Since, except for PDS problems with $p = 6$, each processor is assigned the same number of commodities, there can be no load imbalance between the processors. Thus, the gap between E_p and $\overline{E_p}$ can only be explained as being due to communication time. Indeed, pIPM requires more communication than most other parallel codes for multicommodity flows. Most of communication occurs during the computation of $\left(D - \sum_{i=1}^{k} C_i^T B_i^{-1} C_i\right) v$, where v is the current estimate of the solution of (8), at each PCG iteration. This requires first the broadcast of v from the "master" processor (the one executing the serial-only part of the code) to all the other processors, followed by a vector-reduce operation to accumulate all the partial results $C_i^T B_i^{-1} v$ back to the "master" processor. The amount of communication is essentially the same as in the decomposition approaches [7,13,22], and substantially lower than that of the other specialized parallel interior-point codes [17,10], which need to share the (dense) matrices $C_i^T B_i^{-1} C_i$ in order to form the Schur complement H. However, in pIPM communication occurs at *every PCG iteration*, i.e., much more often than in decomposition codes. The other specialized parallel interior-point codes have a much smaller number of communication "rounds", one for each interior-point iteration, although each round is more expensive.

Thus, pIPM may be inherently more vulnerable to slowdowns induced by communication costs. Indeed, the efficiency of pIPM seems to be, on average, somehow worse than that of the approach in [17], even though direct comparison is difficult due to the different sets of test problems. The instances used in [17] are much smaller, and the cost of forming and factorizing H grows rapidly with the size of the problem.

Furthermore, the current implementation of pIPM, using the parallel constructs available in the SGI O2000 C compiler [25], is not aggressively optimized particularly in the two critical operations, i.e., broadcasts and vector-reduces. Both are currently obtained by means of read/write operations to shared vectors, which are presumably less efficient than the typical system-provided implementation which exploits information about the topology of the interconnection network and the available communication hardware. Also, a part of the communication overhead could be due to a non-optimal placement of the data structures in the local memory of the processors, especially at the boundaries of the virtual memory pages. Thus, we believe that there is still room for (potentially large) reductions of the gap between the observed and the theoretical speedup/efficiency of the code. However, pIPM already attains quite satisfactory efficiencies in some instances, most notably the largest PDS problems.

As far as the absolute speedup is concerned, IPM is known not to be the fastest sequential code for some of the test instances. In [12], a bundle-based decomposition approach has been shown to outperform IPM on the Mnetgen instances, while IPM was competitive on the PDS problems. Furthermore, recent developments in the field of simplex methods [19] have lead to impressive performance improvements for these algorithms on multicommodity flow problems. Nowadays, even the largest PDS problems can be solved in less than an

Table 4. Comparing Cplex 6.5, IPM, and pIPM on the Tripart and Gridgen problems.

Problem	IPM	Cplex 6.5	Cplex6.5 / pIPM*
Tripart1	40	74	3.3
Tripart2	249	627	6.5
Tripart3	1584	2851	6.7
Tripart4	4983	33235	36.3
Gridgen1	126008	$\geq$ 2.8e+6	253.1

* Considering the maximum number of processors for pIPM

hour of CPU with the state-of-the-art simplex code Cplex 6.5 [9]. However, the simplex method is not easily parallelized. Furthermore, other multicommodity problems, like the Tripart and the Gridgen, are much more difficult to solve; ϵ-approximation algorithms can approximatively solve them in a relatively short time [4], but only if the required accuracy is not high. On these instances, the interior-point algorithm in Cplex 6.5 is far more efficient than the dual simplex, but it is in turn largely outperformed by IPM, as shown in Table 4. Columns "IPM" and "Cplex 6.5" show the running time required for the solution of the problem by IPM and Cplex 6.5, respectively, on a Sun Ultra2 2200/200 workstation (credited of 7.8 SPECint95 and 14.7 SPECfp95) with 1Gb of main memory. The last column shows the estimated ratio between the running time of Cplex 6.5 and that of pIPM run on the largest possible number of processors, proving that, at least for the largest and more difficult instances of the set, pIPM provides a competitive approach.

6 Conclusions and Future Research

The parallel code pIPM presented in this work can be an efficient tool for the solution of certain types of large and difficult multicommodity problems. Quite good speedups are achieved in some instances, such as the large PDS problems. In other cases, a gap between the ideal efficiency and the observed one exists. However, we are confident that a more efficient implementation of reduce/broadcast operations and a better placement of data structures—which could mean using MPI or PVM as parallel environments—can make pIPM even more competitive on a widest range of multicommodity instances.

References

1. Adler, I., Resende, M.G.C., Veiga, G.: An implementation of Karmarkar's algorithm for linear programming. Math. Prog. **44** (1989) 297–335
2. Andersen, E.D., Andersen, K.D.: A parallel interior-point algorithm for linear programming on a shared memory machine. CORE Discussion Paper **9808** (1998), CORE, Louvain-La-Neuve, Belgium

3. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and Distributed Computation. Prentice-Hall, Englewood Cliffs (1995)
4. Bienstock D.: Approximately solving large-scale linear programs. I: Strengthening lower bounds and accelerating convergence. CORC Report **1999–1** (1999), Columbia University, NY
5. Castro, J.: A specialized interior-point algorithm for multicommodity network flows. SIAM J. on Opt. **10(3)** (2000) 852–877
6. Castro, J.: Computational experience with a parallel implementation of an interior-point algorithm for multicommodity network flows. In: M. Powell, S. Scholtes (eds.): System Modelling and Optimization. Methods, Theory and Applications. Kluwer, New York (2000) 75–95
7. Cappanera, P., Frangioni, A.: Symmetric and asymmetric parallelization of a cost-decomposition algorithm for multi-commodity flow problems. INFORMS J. on Comp., to appear (2000)
8. Coleman, T.F., Czyzyk, J., Sun, C., Wagner, M., Wright, S.J.: pPCx: parallel software for linear programming. Proceedings of the Eight SIAM Conference on Parallel Processing in Scientific Computing, SIAM, March 1997
9. ILOG CPLEX: ILOG Cplex 6.5 Reference Manual Library. ILOG (1999).
10. De Silva, A., Abramson, D.A.: A parallel interior-point method and its application to facility location problems. Computational Optimization and Applications **9**(3) (1998) 249–273
11. Dongarra, J.J., Meuer, H.W., Strohmaier, E.: TOP500 supercomputer sites. Technical Report **UT-CS-98-404** (1998), Computer Science Dept. University of Tennessee
12. Frangioni, A., Gallo, G.: A bundle type dual-ascent approach to linear multicommodity min cost flow problems. INFORMS J. on Comp. **11**(4) (1999) 370–393
13. Gondzio, J., Sarkissian, R., Vial, J.-P.: Parallel implementation of a central decomposition method for solving large scale planning problems. HEC Technical Report **98.1** (1998)
14. Jessup, E.R., Yang, D., Zenios, S.A.: Parallel factorization of structured matrices arising in stochastic programming. SIAM J. on Opt. **4**(4) (1994) 833–846
15. Kamath, A.P., Karmarkar, N.K., Ramakrishnan, K.G.: Computational and complexity results for an interior-point algorithm on multicommodity flow problems. Technical Report **TR-21-93** (1993), Dip. di Informatica, Università di Pisa, Italy
16. Kontogiorgis, S., De Leone, R., Meyer, R.R.: Alternating directions splitting for block angular parallel optimization. JOTA **90**(1) (1996) 1–29
17. Lustig, I.J., Li, G.: An implementation of a parallel primal-dual interior-point method for block-structured linear programs. Computational Optimization and Applications **1** (1992) 141–161
18. Lustig, I.J., Rothberg, E.: Gigaflops in linear programming. O.R. Letters **18**(4) (1996) 157–165
19. McBride, R.D.: Advances in Solving the Multicommodity Flow Problem. SIAM J. on Opt. **8**(4) (1998) 947–955
20. Medhi, D.: Parallel bundle-based decomposition for large-scale structured mathematical programming problems. Annals of O.R. **22** (1990) 101–127
21. Ng, E., Peyton, B.W.: Block sparse Cholesky algorithms on advanced uniprocessor computers. SIAM J. Sci. Comput. **14** (1993) 1034–1056
22. Pinar, M.C., Zenios, S.A.: Parallel decomposition of multicommodity network flows using a linear-quadratic penalty algorithm. ORSA J. on Comp. **4** (1992) 235–249

23. Portugal, L.. Resende, M.G.C.. Veiga, G.. Júdice, J.: A truncated interior-point method for the solution of minimum cost flow problems on an undirected multicommodity flow network. Proceedings of First Portuguese National Telecommunications Conference, Aveiro, Portugal (1997) 381–384 (in Portuguese)
24. Rosen, J.B. (ed.): Supercomputers and large-scale optimization: algorithms, software, applications. Annals of O.R. **22** (1990)
25. Silicon Graphics Inc.: C Language Reference Manual (1998)
26. Schultz, G., Meyer, R.: An interior-point method for block-angular optimization. SIAM J. on Opt. **1** (1991) 583–682
27. Wright, S.J.: Primal-Dual Interior-Point Methods. SIAM, Philadelphia, PA (1997)