Symmetric and Asymmetric Parallelization of a Cost-Decomposition Algorithm for Multicommodity Flow Problems

Paola Cappanera • Antonio Frangioni
Dipartimento di Informatica, Università di Pisa, Corso Italia, 40, 56125 Pisa, Italy
cappaner@di.unipi.it • frangio@di.unipi.it

We study the coarse-grained parallelization of an efficient bundle-based cost-decomposition algorithm for the solution of multicommodity min-cost flow (MMCF) problems. We show that a code exploiting only the natural parallelism inherent in the cost-decomposition approach, i.e., solving the min-cost flow subproblems in parallel, obtains satisfactory efficiencies even with many processors on large, difficult MMCF problems with many commodities. This is exactly the class of instances where the decomposition approach attains its best results in sequential. The parallel code we developed is highly portable and flexible, and it can be used on different machines. We also show how to exploit a common characteristic of current supercomputer facilities, i.e., the side-to-side availability of massively parallel and vector supercomputers, to implement an asymmetric decomposition algorithm where each architecture is used for the tasks for which it is best suited.
(*Networks-Graphs, Multicommodity; Programming, Large-Scale Systems; Programming, Non-differentiable*)

1. Introduction

The multicommodity min-cost flow (MMCF) problem is a generalization of the ordinary single-commodity min-cost flow (MCF) problem, in which flows of a different nature (*commodities*) must be routed at minimal cost on a network, competing for the resources represented by the arc capacities. On the theoretical side, MMCF is intimately related with approximation algorithms for several relevant graph problems (Klein et al. 1990). From a practical point of view, MMCF and its variants (Crainic et al. 2001) are (sub)models of a wide variety of transportation and scheduling problems, where often many large "easy" MMCFs have to be solved in order to solve one hard problem. Although MMCF is a structured linear program (LP), standard LP techniques often fail to be efficient enough in practice, and several specialized algorithms have been proposed for its solution during the last four decades.

Among the approaches to MMCF, *decomposition methods* are perhaps the most successful, as demonstrated by the constant stream of work dedicated to this class of algorithms. In Frangioni and Gallo (1999), extensive computational experience showed that a bundle-type cost-decomposition approach to MMCF is effective in practice, especially on problems with "many" commodities. In this paper, the results obtained with a parallel version of that code are presented and discussed.

In the last few years, many parallel approaches to MMCF have been developed (Zenios and Mulvey 1988; Medhi 1990; Zenios 1991; Ferris and Mangasarian 1992, 1994; Lustig and Li 1992; Pinar and Zenios 1992; Gnanendran and Ho 1993; De Leone et al. 1993, 1994; Zenios 1993, Kontogiorgis et al. 1996; De Silva

0899-1499/03/1504/0369
1526-5528 electronic ISSN

and Abramson 1998; Ferris and Horn 1998; Castro and Frangioni 2001), demonstrating the broad interest in the solution of large-scale MMCF problems. The present approach can be classified among those with *complex coordinator*, that seem to be best suited for implementation on coarse-grained massively parallel machines. Unlike most other attempts, the parallel code directly derives from a sequential implementation whose actual effectiveness has been convincingly shown. The parallel code is an extension of the sequential one—that had not been developed for parallelization—rather than an entirely new code, and it is able to benefit immediately from any improvement in the sequential algorithm due to the inheritance mechanism of the programming language used. It obtains quite satisfactory results despite having been developed to be portable, and therefore having not been substantially modified to best suit the parallel machines where it has been tested.

The structure of the work is the following: In §2 the MMCF problem is introduced, and the parallel approaches proposed in the literature are briefly reviewed. Section 3 describes the sequential bundle approach, while in §4 the issues related to its coarse-grained parallelization are discussed. In the remaining paragraphs, the computational experiences are described: In §5 the details of the available (hardware and software) environment are given, in §6 the test problems are introduced, and, finally, in §7 the computational results are presented and conclusions are drawn.

2. MMCF: Formulation and Parallel Approaches

In the (linear) MMCF problem, a directed graph $G(N, A)$, where $|N| = n$ and $|A| = m$, is given. A set of k commodities (types of flow) has to be routed on G at minimal total cost while satisfying the usual flow-conservation constraints at the nodes. Flow x_{ij}^h of the h-th commodity on arc (i, j) has individual lower and upper bounds and a (linear) cost $c_{ij}^h x_{ij}^h$, while *mutual capacity constraints* bound the total quantity of flow, irrespective of the commodity, on arc (i, j). A formulation of the problem is

$$\text{(MMCF)} \quad \begin{cases} \min \Sigma_h \Sigma_{ij} c_{ij}^h x_{ij}^h & \\ \Sigma_j x_{ij}^h - \Sigma_j x_{ji}^h = b_i^h & \forall i, h \quad \text{(a)} \\ 0 \le x_{ij}^h \le u_{ij}^h & \forall i, j, h \quad \text{(b)} \\ \Sigma_h x_{ij}^h \le u_{ij} & \forall i, j \quad \text{(c)} \end{cases}$$

where (a) are the flow-conservation constraints and (b), (c) are respectively the individual and mutual capacity constraints. We can restate the problem in matrix notation, using the node-arc incidence matrix $\boldsymbol{E}$ of G, as follows:

$$\text{(MMCF)} \quad \begin{cases} \min \Sigma_h c^h x^h & \\ \begin{bmatrix} \boldsymbol{E} & \cdots & \boldsymbol{0} \\ \vdots & \ddots & \vdots \\ \boldsymbol{0} & \cdots & \boldsymbol{E} \\ \boldsymbol{I} & \cdots & \boldsymbol{I} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{x}^1 \\ \vdots \\ \boldsymbol{x}^k \end{bmatrix} \le \begin{bmatrix} \boldsymbol{b}^1 \\ \vdots \\ \boldsymbol{b}^k \\ \boldsymbol{u} \end{bmatrix} & \\ 0 \le x^h \le u^h & \forall h. \end{cases}$$

This (node-arc) formulation highlights the block-angular structure of MMCF.

Let us remark that the number of commodities in real-life MMCFs ranges from just a few, as in most distribution problems, to very many, such as the number of all the possible O/D pairs in some telecommunication models. This is of course crucial in the choice of the most suitable solution algorithm (Frangioni and Gallo 1999).

MMCF is only one example of the many models in planning and scheduling that exhibit a block-angular structure, representing spatial or temporal partial decomposability. In these problems, decision variables can be broken down to largely independent blocks that correspond to first-level decisions (which may represent a time period, a geographical region, or, such as in our case, a commodity) satisfying a subset of the constraints. The blocks interact via coupling constraints related to second-level coordination, such as shared resource allocation. The block-angular structure naturally leads to a number of different *decomposition algorithms*, whereby a globally convergent master problem is iteratively updated by solving a set of subproblems (one for each block), until some

prescribed convergence criterion is met. The execution of such an algorithm alternates a local and a global computation phase. In the local phase, optimization subproblems corresponding to blocks, substantially independent of each other, are solved. In the global phase, the solution information from the subproblems is used in a coordination (master) problem, the results of which are used to modify the objective function and the constraints set of the subproblems in the next step.

Decomposition algorithms are traditionally classified as either *cost-decomposition* or *resource-decomposition* approaches.

Cost-decomposition approaches remove the coupling constraints by putting them in the objective function via Lagrangean relaxation, in the form of a barrier function or using penalty terms. The resulting nonlinear (possibly nondifferentiable) problem is solved—somehow exploiting the block-angular structure of the problem—and some parameters are updated in the attempt to achieve feasibility w.r.t. the relaxed constraints. Lagrangean-based approaches such as the *Dantzig-Wolfe* method (Ho et al. 1988, Gnanendran and Ho 1993) or the *bundle* method (Medhi 1990, Ferris and Horn 1998, Frangioni and Gallo 1999) directly approach the maximization of the nondifferentiable Lagrangean function, whose calculation breaks down in the solution of an independent subproblem for each block. Other methods are based on differentiable but non-separable functions, such as the *augmented Lagrangean* (De Leone et al. 1994, Kontogiorgis et al. 1996), *linear-quadratic* (Pinar and Zenios 1992) or *exponential* (Grigoriadis and Khachiyan 1995) *penalty functions*, or *logarithmic barrier functions* (Meyer and Schultz 1992, De Leone et al. 1994). Since these functions are non-separable, iterative (mostly inner linearization) methods are used to compute them approximately by solving only block-separable subproblems. However, even bundle methods can be seen as approximate augmented Lagrangean approaches; indeed, several of the above cost-decomposition approaches can be cast into a unified framework (Frangioni 2002).

By contrast, the base idea underlying resource-decomposition approaches (De Leone et al. 1993) consists of transforming the "global" constraints into "local" constraints. In the MMCF case, for instance, this is done by choosing a tentative a-priori distribution of the mutual capacity of the arcs among the commodities and solving the subproblems with "strengthened" individual capacities. The reduced costs of the corresponding optimal flows can be used to update the distribution. A problem with this approach is the difficulty of effectively coping with infeasible distributions of the capacities, i.e., empty subproblems.

Modifications of the individual capacities are sometimes performed even in cost-decomposition methods (Grigoriadis and Khachiyan 1995) to speed up convergence. Also, decomposition methods with both flow and capacity variables (De Leone et al. 1994, Kontogiorgis et al. 1996) have been proposed.

Another—possibly more important—distinction is drawn between decomposition methods based on *complex coordinators* and those based or *simple coordinators*. All decomposition methods can be parallelized by partitioning the subproblems among different processors (processing elements, or PEs). The resulting algorithms all have the common "master-slave" structure: at each iteration, a master problem is solved in the—in principle, serial—master phase, then information is sent to the slaves and the—concurrent—slave phase begins where each slave modifies and solves its subproblem. Clearly, the performance of distributed decomposition, as measured by *speedup* or *efficiency*, is adversely affected by the magnitude of the serial master phase. Thus, deciding which tasks the master has to accomplish is a critical point, since there is a trade-off between the complexity of the coordinating mechanism of the decomposition process and the overall parallel performance (De Leone et al. 1994). Simple coordinators that only require summing and averaging of subproblem information are, on the surface, better suited to parallel environments. On the other hand, complex coordinators that require the solution of an optimization problem may lead to a much smaller number of iterations, and hence to higher overall effectiveness. This trade-off clearly depends on the relative computational burden of the master problem and of the subproblems, which may also depend on the particular instance—e.g., from the number or "tightness" of the coupling constraints.

A prominent decomposition method based on a simple coordinator is the alternating directions method (De Leone et al. 1994, Kontogiorgis et al. 1996), which is basically an augmented Lagrangean method where the nonseparable augmented Lagrangean is only approximately solved with one iteration of a block-Gauss-Seidel approach. The master phase is very simple, as PEs only combine their solution in order to form, as aggregates, new values for the subproblem proximal terms and Lagrangean multipliers. Another method based on a simple coordinator is the parallel constraint distribution method (Ferris and Mangasarian 1992), which distributes the constraints among the PEs and modifies each subproblem objective function with Augmented Lagrangean terms from other PEs.

Approaches based on complex coordinators are the Dantzig-Wolfe method (Ho et al. 1988, Gnanendran and Ho 1993) and bundle methods (Medhi 1990, De Leone et al. 1993, Ferris and Horn 1998, Frangioni and Gallo 1999). At each step, the master uses (potentially all) the information collected from all the previous slave phases—columns or subgradients, depending on the viewpoint—to compute the new vector of prices to be broadcasted to the slaves. Thus, the master phase requires the solution of possibly large-scale linear (in the DW case) or quadratic (in the bundle case) programs, whose size also tends to increase with the iterations.

The distinction between simple and complex coordinators is unclear: the "complexity" of the coordinator is not actually inherent to the approach, but rather a function of several design decisions, so that a whole range of possibilities exists between very simple and very complex coordinators. In fact, many decomposition algorithms employ coordinators that require the solution of a potentially complex optimization problem, whose cost is kept low only by forcing the size of the optimization problem to be very low. This is the case of the linear-quadratic penalty algorithm of Pinar and Zenios (1992), of the exponential-penalty approximation algorithm of Grigoriadis and Khachiyan (1995), of the interior-point method of Meyer and Schultz (1992), and of the parallel variable distribution method of Ferris and Mangasarian (1994). Yet, all these methods are naturally extended to complex coordinator variants. Conversely, bundle methods can be made to work with limited-size master problems (Frangioni 2002). Furthermore, even complex coordinators can be implemented in different ways, which can have a profound impact on their complexity. For instance, both the DW and bundle methods can be implemented in either the "aggregate" or the "disaggregate" variant, the distinction being if each solution from a block becomes a separate column, or they are all aggregated into an unique column; the first solution typically has a (much) faster rate of convergence, at the cost of a (much) larger master problem. Finally, the potential serial bottleneck of solving a complex master problem can be faced by decomposing it. In De Leone et al. (1994), for instance, the decomposition algorithm of Meyer and Schultz (1992) is extended by employing p simple coordinators instead of a single complex one; the p coordinating problems are solved in parallel, each producing a proposed next iterate, and the proposal with the best (least) objective value is then chosen as the overall solution. Deciding whether this two-level coordinator is to be considered simple or complex is thus a critical task.

Finally, let us remark that decomposition-type algorithms are not the only kind of parallel approaches developed for MMCF. Primal-dual (*row-action*) algorithms have been proposed (Zenios and Mulvey 1988; Zenios 1991, 1993) that are capable of exploiting both fine-grained and coarse-grained parallelism (indeed, these algorithms can be seen as extremely fine-grained cost-decomposition methods). Although in principle interesting, row-action algorithms tend to suffer in practice from a convergence rate that is not really high. More recently, specialized parallel implementations of interior-point algorithms for LP have been proposed and tested (Lustig and Li 1992, De Silva and Abramson 1998, Castro and Frangioni 2001). These algorithms exploit the block-staircase structure of the coefficient matrix of MMCF to avoid direct factorization of a very large matrix, which also usually exhibits very dense "lower-right" corners. Instead, smaller and much sparser systems can be solved in parallel (one for each block), confining the fill-in in the Shur complement matrix that is dealt

with in the sequential part. It is even possible to avoid forming the (dense) Shur complement at all, employing an iterative method; this has shown to be quite effective in practice, even in sequential (Castro 2000), thanks to the very good convergence rate of interior-point methods for LP.

3. The (Sequential) Bundle Approach

To address the solution of MMCF, the cost-decomposition code of Frangioni and Gallo (1999) (MMCFB) considers its Lagrangean relaxation w.r.t. the "complicating" constraints (c), i.e.,

$$(\mathrm{RMG}_\lambda) \qquad \min\{\Sigma_h\Sigma_{ij} c_{ij}^h x_{ij}^h + \Sigma_{ij}\lambda_{ij}(\Sigma_h x_{ij}^h - u_{ij}) : \boldsymbol{E}\boldsymbol{x}^h = \boldsymbol{b}^h, \mathbf{0} \le \boldsymbol{x}^h \le \boldsymbol{u}^h \ \forall h\},$$

and solves the corresponding Lagrangean dual

$$(\mathrm{DMMCF}) \quad \max_{\boldsymbol{\lambda}\ge 0}\{\varphi(\boldsymbol{\lambda}) = \min\{\Sigma_h(\boldsymbol{c}^h + \boldsymbol{\lambda})\boldsymbol{x}^h - \boldsymbol{\lambda}\boldsymbol{u} : \boldsymbol{E}\boldsymbol{x}^h = \boldsymbol{b}^h, \mathbf{0} \le \boldsymbol{x}^h \le \boldsymbol{u}^h \ \forall h\}\}.$$

For any fixed $\boldsymbol{\lambda}$, $\varphi(\boldsymbol{\lambda})$ can be quickly computed by solving k independent (single-commodity) MCFs. This is especially true if each commodity has a single origin (or destination) node and no single-commodity upper bounds exist (i.e., $u_{ij}^h = +\infty \ \forall i, j, h$), since under these assumptions the subproblems are shortest path tree problems (SPTs) w.r.t. the modified costs $(\boldsymbol{c}^h + \boldsymbol{\lambda})$.

(DMMCF) is solved with a *proximal bundle* algorithm, a class of nondifferentiable optimization approaches mainly characterized by storing the first-order information about $\varphi(\ )$, obtained at the previous iterations, in a disaggregated form. Having visited a (finite) sequence of points $\{\boldsymbol{\lambda}_i\}$, the "bundle" $\beta = \{\langle \boldsymbol{\lambda}_i, \varphi(\boldsymbol{\lambda}_i), \boldsymbol{g}(\boldsymbol{\lambda}_i)\rangle\}$ is used to compute a tentative ascent direction $\boldsymbol{d}_i$, along which $\boldsymbol{\lambda}_{i+1}$ is chosen, where $\boldsymbol{g}(\boldsymbol{\lambda}) = \Sigma_h \boldsymbol{x}^h(\boldsymbol{\lambda}) - \boldsymbol{u}$ is the subgradient of $\varphi(\ )$ in $\boldsymbol{\lambda}$ identified by an(y) optimal solution $[\boldsymbol{x}^1(\boldsymbol{\lambda}), \ldots, \boldsymbol{x}^k(\boldsymbol{\lambda})]$ of (RMG_λ). At each step, calculation of $\boldsymbol{d}_i$ requires the solution of the quadratic-programming (QP) subproblem

$$(\Delta_{\beta t}) \qquad \min_\theta\{1/2t\|\Sigma_{i\in\beta}\boldsymbol{g}_i\theta_i\|^2 + \Sigma_{i\in\beta}\alpha_i\theta_i : \Sigma_{i\in\beta}\theta_i = 1, \boldsymbol{\theta} \ge \mathbf{0}\}$$

where $\boldsymbol{g}_i = \boldsymbol{g}(\boldsymbol{\lambda}_i)$ and $\alpha_i = \alpha_i(\bar{\boldsymbol{\lambda}}) = \varphi(\boldsymbol{\lambda}_i) + \boldsymbol{g}_i(\bar{\boldsymbol{\lambda}} - \boldsymbol{\lambda}_i) - \varphi(\bar{\boldsymbol{\lambda}})$ is the linearization errors w.r.t. the current point $\bar{\boldsymbol{\lambda}}$. Here, $t > 0$ is the so-called *proximal parameter*; slightly different forms of $(\Delta_{\beta t})$ are needed for different variants of Bundle methods, such as *Proximal Level* algorithms (Ferris and Horn 1998). Once $\boldsymbol{d}_i$ has been found, the value of $\varphi(\bar{\boldsymbol{\lambda}} + \boldsymbol{d}_i)$ and the relative subgradient are used to adjust the current point $\bar{\boldsymbol{\lambda}}$, if a sufficient increase in the value of the $\varphi(\ )$ is attained, and the parameter t. Anyway, the newly obtained first-order is added to β.

$(\Delta_{\beta t})$ can be viewed either as a (Lagrangean relaxation of a) "least-square version" of the master problem of Dantzig-Wolfe decomposition, or, in the nondifferentiable optimization terminology, as (a Lagrangean relaxation of) the problem of finding the minimum norm vector in an inner approximation of the $\max_{i\in\beta}\{\alpha_i\}$- subdifferential of $\varphi(\ )$ in $\bar{\boldsymbol{\lambda}}$. Furthermore, its (quadratic) dual is

$$(\Pi_{\beta t}) \qquad \max_d\{\varphi_\beta(\bar{\boldsymbol{\lambda}} + \boldsymbol{d}) - 1/2t\|\boldsymbol{d}\|^2\},$$

i.e., the maximization of the *cutting-plane model* $\varphi_\beta(\boldsymbol{\lambda}) = \min_{i\in\beta}\{\varphi(\boldsymbol{\lambda}_i) + \boldsymbol{g}_i(\boldsymbol{\lambda} - \boldsymbol{\lambda}_i)\} - \varphi(\bar{\boldsymbol{\lambda}})$, the polyhedral upper approximation of $\varphi(\ )$ built up with the first-order information found so far, plus a quadratic *proximal term*, weighted with t. The proximal term discourages directions leading to "far-away" points, where $\varphi_\beta(\ )$ is probably a "bad" approximation of $\varphi(\ )$, and the parameter t somehow measures our "trust" in $\varphi_\beta(\ )$ as we move further from $\bar{\boldsymbol{\lambda}}$ (Figure 1).

For sufficiently large values of t, $(\Pi_{\beta t})$ is equivalent to the "naive" maximization of $\varphi_\beta(\ )$ alone,

$$(\Pi_\beta) \qquad \max_d\{\varphi_\beta(\bar{\boldsymbol{\lambda}} + \boldsymbol{d})\},$$

provided that the latter problem has a finite maximum, which is very often *not* the case, especially in earlier stages of the algorithm's operations. Bundle methods are therefore a variant of the DW/cutting-plane algorithm, which uses the optimal solution of (Π_β) as the next trial point.

Actually, for solving MMCF a *constrained* bundle algorithm is required, since nonnegativity of the Lagrangean variables has to be ensured. This can be tackled by just making the QP subproblem "aware" of the constraints, i.e., by solving the extended problem

$$(\Pi'_{\beta t}) \qquad \max_d\{\varphi_\beta(\bar{\boldsymbol{\lambda}} + \boldsymbol{d}) - 1/2t\|\boldsymbol{d}\|^2 : \boldsymbol{d} \ge -\bar{\boldsymbol{\lambda}}\},$$

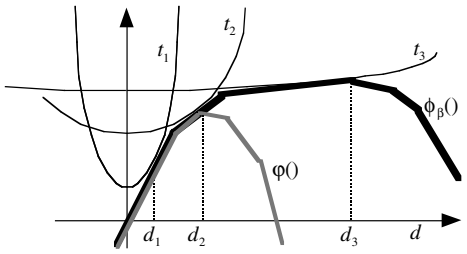

Figure 1 **Effect of the Proximal Term for $t_1 < t_2 < t_3 : d_i$ Are the Optimal Solutions**

which guarantees feasibility of the tentative point w.r.t. the $\boldsymbol{\lambda} \geq \mathbf{0}$ constraints.

The above method is used in the sequential MMCFB code; it is beyond the scope of this paper to describe it in full detail; the interested reader is referred to Frangioni and Gallo (1999). MMCFB has been shown to be efficient on several different classes of instances, the key components of its performance are:

— the use of a fast, specialized QP solver (Frangioni 1996) that uses a *two-level-active-set* strategy, explicitly designed to be efficient for the kind of reoptimization arising in bundle algorithms, and which supports online *creation and destruction of variables*;

— the use of an efficient MCF solver to keep the cost of the solution of (RMG_λ) low;

— the ability of exploiting the structure of the flow subproblems by identifying those that are in fact SPTs and using a specialized shortest-path code instead of the more general MCF solver;

— a *Lagrangean variables generation strategy*, where only an "active" subset of the variables is modified at each iteration: this greatly helps in reducing the computational cost of solving ($\Pi'_{\beta t}$), as few Lagrangean variables ever become nonzero and the "active set" is very stable after the first few iterations, so that the (relatively costly) check of whether new variables have to be added can be performed with low frequency;

— a new t handling scheme based on two different heuristics for the increase and decrease respectively, proper settings of some critical optimization parameters, and a new stopping condition based on a user-provided "distance estimator" t^*.

4. Parallelization of the Bundle Approach

The parallel code, pMMCFB, is developed as an extension of the existing sequential MMCFB, rather than as an independent code, exploiting the inheritance mechanisms provided by C++. This allows updates on the sequential code automatically to reecho on the parallel one. For instance, early versions of MMCFB used a line search to find the new point $\boldsymbol{\lambda}_{i+1}$ along direction $\boldsymbol{d}_i$, that was subsequently abandoned in favor of a trust region approach; these changes have not required any modification to pMMCFB.

pMMCFB shares most of the base code of MMCFB, which consists of several C++ *classes*. In the following the most important ones are briefly described (see Frangioni (1997) for more detail). The `Graph` class provides a mean for reading MMCF problems, preprocessing (e.g., identifying redundant mutual capacity constraints) and storing them in memory, along with an interface that can be used by any MMCF solver to access and change the data. It also supports decomposition-based MMCF solvers by providing *methods* that hide the details of the mutual capacity constraints structure. The `MMCFBundle` class implements the main bundle algorithm; the value of the objective and the subgradient are computed inside the (*virtual*) method `FiAndGi( )`, that is the only other function interacting with `Graph` apart from the constructor, where the data structures are set up. The `RelaxIV` class implements a MCF solver based on the RelaxIV algorithm (Bertsekas 1991), that is known to be fast and to reoptimize efficiently.

pMMCFB extends the sequential code by introducing the three *derived* classes `MasterClass`, `SlaveClass`, and `ParGraph`. In the following we will give a brief description of such classes in order to point out how the portability of the code has been obtained (the interested reader is referred to Cappanera (1996) for further detail). `ParGraph` derives from `Graph`, and provides support for splitting the data of the problem among the different PEs. A constructor of the class, which is used in each slave, reads the data from a PVM channel. Methods to send the data along the channel are provided for being used at the master PE, where the original `Graph` constructors are

used instead. `MasterClass` derives from `MMCFBundle`, and only re-implements the constructor—where the data are spliced among the Slaves by using the `ParGraph` methods—and the method `FiAndGi( )`. At each iteration, the sequential code is executed until `FiAndGi( )` is invoked, then control is passed to `MasterClass::FiAndGi( )` where the new prices $\boldsymbol{\lambda}$ are broadcasted to the slaves, the results are gathered, and the corresponding objective function and subgradient are calculated, prior to returning control to the `MMCFBundle` methods. Note that the slaves synchronize by sending their solutions to the master through a *reduce* operation; this may introduce active waiting, but it exploits the very efficient reduce implementations available on most parallel machines. Besides, in an "aggregated" bundle approach as MMCFB the master cannot proceed until all the subproblems are terminated, differently from a "disaggregated" DW approach where the master can begin processing columns received from the slaves piecemeal (Gnanendran and Ho 1993). Finally, `SlaveClass` implements the (RMG_λ) solution phase, nearly identical to the one in `MMCFBundle::FiAndGi( )`, for the subset of the commodities assigned to the `ParGraph` object of the corresponding PE. Thanks to the inheritance mechanism and the use of the derived `ParGraph` class, extension of the code to any block-structured network problem with side constraints would not require changing `MasterClass` or `SlaveClass`.

Some other implementation details may be noteworthy. For instance, `MMCFBundle` is in turn derived from `MMCFClass`, an *abstract base class* implementing a standard interface for MMCF solvers that makes application programs independent from the particular kind of solver that is used. An analogous structure has been used for the MCF solver: the abstract base class `MCFClass` defines a standard interface for MCF solvers, and k objects of proper derived classes are constructed within the MMCFB, which uses the public methods of such objects for changing the costs, solving the problems, and collecting the results. This makes `MMCFBundle` completely independent from upgrades in the MCF solver, or even changes of algorithm, as well as easily adaptable to specially-structured classes of instances. Indeed, this mechanism has been used to deal efficiently with SPT subproblems, by essentially only developing the derived class `SPTree`.

All the modules use a set of "abstract" basic data types, allowing one easily to port the code under different architectures, and to tailor it to the characteristics of both the computer that is being used and the class of instances that are being solved. Finally, compile-time options are given to the end user for customizing the code without knowing the details of the implementation; some of these switches can be used to overcome the limitations—or exploit the peculiarities—of a given parallel machine, as discussed in the next section.

5. The Parallel Environment

pMMCFB has been mainly tested on a Cray T3D, incorporating 64 Alpha microprocessors, each capable of 150 MFLOPS peak performance. Each PE in the system is composed of the Alpha processor, with 8 + 8 (instruction + data) Kb of cache, 64 Mb of local memory, and custom support logic for the bi-directional 3-D torus interconnect network, with peak interprocessor communication rates of 300 Mb/s. The Cray T3D system design supports some major latency-hiding mechanisms and offers a wide range of synchronization mechanisms to accommodate both SIMD and MIMD programming styles. Memory, although physically distributed among PEs, is globally addressable, but a highly-optimized implementation of PVM (Cray 1994a) is also provided. The Cray T3D is running UNICOS MAX, a distributed operating system whose functions are divided between the PEs of the T3D and the Cray C90 host, which allows the user to edit, compile, and link programs, and to initiate, control, and terminate all T3D processes. The available C90 system was a C94/2148 with 2 CPUs, each with 14 independent functional units and 8 vector registers, capable of almost 240 MFLOPS peak performance (clock cycle = 4.2 Ns).

pMMCFB has been designed to work under any distributed environment where PVM is available; however, it is also possible to activate—via compile-time switches—architecture-specific portions of code for communication or process control primitives. This allows some architecture-specific optimization of the

code. For instance, when running PVM on T3D in standalone mode there is no need to set up a Master (PE_0 is always chosen), to create the slave processes by `pvm_spawn( )` and to wait for them to join into a proper global group and signal their presence through a `pvm_barrier( )`. Also, the standard `pvm_bcast( )`, `pvm_send( )`, and `pvm_reduce( )`, which are used in pMMCFB respectively to send common data to all PEs, for point-to-point communication and to calculate $g(\boldsymbol{\lambda})$ and $\varphi(\boldsymbol{\lambda})$, can be substituted with architecture-specific primitives, such as the `SHMEM` routines of the T3D (Cray 1994b). Finally, two self-implemented reduce procedures, one based on a binary tree and the other one on $p-1$ `pvm_recv( )`, are provided for those cases in which `pvm_reduce( )` is not available.

A remarkable characteristic of the available parallel environment was the side-to-side presence of two very different architectures: the vector supercomputer Cray C90 and the massively parallel MIMD Cray T3D. While the latter is clearly the machine where the $\varphi(\boldsymbol{\lambda})$ calculation is to be performed, the former is particularly efficient on the kind of vector operations that forms the computational core of the (otherwise non-parallelizable) algorithm for the solution of $(\Pi'_{\beta t})$. This is confirmed by Table 1, where the time spent by MMCFB for calculating $(\varphi\boldsymbol{\lambda})$ (MCF) and solving $(\Pi'_{\beta t})$ (QP) on the two architectures is compared for two particular groups of instances (the 128-128 and 256-128 problems, see §5). As the table clearly show, the MCF solver is not faster on the C90 than on a single PE of the T3D, while the QP code is up to six times faster, and the difference increases with the size of the problem (larger for 256-128 problems than for 128-128, and in the latter six problems of each group than in the former six). This is not surprising, since the MCF solver contains few vector operations, and is therefore almost not vectorizable, while the converse is true for the QP code. Note that all the results have been obtained by only invoking the vectorization options provided by the C90 compiler; further performance improvements might have been obtained by hand-made vectorization.

Table 1 $\varphi(\boldsymbol{\lambda})$ **and** $(\Pi'_{\beta t})$ **Times on the Two Different Architectures**

128-128				256-128			
MCF		QP		MCF		QP	
C90	T3D	C90	T3D	C90	T3D	C90	T3D
26.94	25.51	0.03	0.05	298.03	298.60	0.25	0.45
21.98	20.59	0.03	0.04	129.05	121.07	0.09	0.12
11.64	10.70	0.02	0.02	80.51	74.58	0.11	0.12
45.71	43.77	0.07	0.15	510.35	499.00	0.45	1.28
22.85	21.62	0.04	0.08	217.54	211.27	0.17	0.55
16.07	14.83	0.04	0.03	80.74	76.40	0.13	0.25
243.88	232.47	0.78	1.78	553.34	554.80	0.92	2.55
99.12	91.22	0.16	0.73	529.95	526.59	0.79	6.27
133.75	126.35	0.23	0.70	379.24	373.49	0.42	1.82
526.24	486.24	1.44	4.63	2518.62	2546.11	6.04	26.39
355.84	325.70	0.75	3.62	1203.13	1167.85	2.05	11.54
233.00	217.45	0.45	1.57	777.06	786.76	1.23	6.63

These results prompted the development of an *asymmetric version* of pMMCFB, where the master and the slaves are executed on different machines. Only limited modifications were necessary, some of which are, however, worth describing. Basically, in the specialized implementation the PE_0 of T3D acts as a submaster, being the only slave that directly communicates with the C90. This allows for most of the data exchange to be performed via the fast T3D interconnection network, avoiding C90-T3D communication through the *PVM-daemon* (running on the C90) and the slow Unix sockets system, which would severely degrade performance. This also serves to overcome a difficulty arising with `pvm_reduce( )`, i.e., that dynamic groups consisting of tasks running on both the T3D *and* the C90 are not allowed. Master-submaster communication speed can also be improved by combining I/O and PVM (Cray 1994a), i.e., by separating the two parts of a PVM message, control information and actual data, using the slower PVM (Unix sockets) for the former and faster UNICOS I/O (read/write operation on a memory-mapped file) for the latter. Finally, it is possible to decide, via compile-time switches, whether or not the master solves a group of MCFs.

6. The Test Problems

Several data sets and random generators of MMCF problems are available, and have been used in the literature to test one or the other of the proposed approaches; the ones employed here, along with

many others, can be retrieved at `http://www.di.unipi.it/di/groups/optimize/Data/MMCF.html`.

In order to test PMMCFB, a suite of 78 *Mnetgen* problems has been selected. But for the very large problems, the same data set had been previously used for testing MMCFB (Frangioni and Gallo 1999); there already, the choice of Mnetgen had proven fruitful for analyzing the influence of some characteristics of the MMCF instances on the relative effectiveness of alternative sequential algorithms.

The test set has been generated with the following rules: for each value of n in $\{64, 128, 256\}$ and k in $\{64, \ldots, n\}$ (Mnetgen cannot generate problems with $k > n$), 12 different problems were generated, for a total of 6 *classes* of problems characterized by a pair (n, k). Within each class, 6 problems (*groups* A and B) are "sparse" and 6 (groups C and D) are "dense", with m/n respectively equal to about 3 and 8. Both these subclasses are in turn subdivided of into 3 "easy" (A and C) and 3 "hard" (B and D) problems; easy problems have mutual capacity constraints on 40% of the arcs and 10% of the arcs with a "high" cost, while these figures are 80% and 30%, respectively, for hard problems. Finally, the 3 subproblems within each group differs in the number of arcs that have individual capacity constraints, fixed to 30%, 60%, and 90%, respectively; note that, because of the individual capacities, the corresponding MCF subproblems are *not* SPTs. Usually, within the same group the "difficulty" of the problem decreases as the percentage of arcs with individual arc bounds increases, since the number of "tight" mutual constraints tend to diminish.

To test the code on larger problems, the "sparse" problems groups (A and B) with $n = k = 512$ were also generated, whose LP formulation has more than $8 \cdot 10^5$ variables. Curiously enough, it was not possible to test the (512, 512) "dense" groups, or problems with larger n, because of limitations of the *disk space* available on the server machine, but the computational results seem to show that even much larger problems may be solved in a very reasonable time.

Table 2 summarizes the (averaged) characteristics of the test problems: in the table, each problem group is designated by $n - k - g$, with $g \in \{A, B, C, D\}$, b is the number of arcs having mutual capacity constraints, Constr $= m \times k + b$ is the total number of constraints, and Var $= m \times k$ is the total number of variables in the LP formulation of the problem.

Table 2 Characteristics of the Test Problems

Group	k	n	m	b	Var	Constr
64-64-*A*	64	64	209	86	13376	4182
64-64-*B*	64	64	197	160	12608	4256
64-64-*C*	64	64	516	193	33003	4289
64-64-*D*	64	64	524	415	33536	4511
128-64-*A*	64	128	388	155	24811	8347
128-64-*B*	64	128	405	328	25920	8520
128-64-*C*	64	128	1159	458	74155	8650
128-64-*D*	64	128	1185	932	75819	9124
128-128-*A*	128	128	394	156	50475	16540
128-128-*B*	128	128	407	332	52053	16716
128-128-*C*	128	128	1219	486	156032	16870
128-128-*D*	128	128	1213	966	155307	17350
256-64-*A*	64	256	825	327	52779	16711
256-64-*B*	64	256	801	647	51264	17031
256-64-*C*	64	256	2337	922	149547	17306
256-64-*D*	64	256	2330	1828	149120	18212
256-128-*A*	128	256	814	304	104149	33072
256-128-*B*	128	256	812	655	103893	33423
256-128-*C*	128	256	2344	909	300032	33677
256-128-*D*	128	256	2353	1861	301184	34629
256-256-*A*	256	256	824	314	210859	65850
256-256-*B*	256	256	830	679	212480	66215
256-256-*C*	256	256	2168	841	554923	66377
256-256-*D*	256	256	2190	1772	560640	67308
512-512-*A*	512	512	1633	644	836096	262788
512-512-*B*	512	512	1635	1291	837120	263435

Preliminary experiments were also conducted on other sets of problems, typically having $k < 64$. Those problems were later discarded for two reasons: the first was that it was impossible to exploit all the parallelism provided by the available machines, and the second was that they were "too easy," i.e., solvable within a few seconds on a workstation, so that their parallelization was hardly interesting in practice. A posteriori, problems with a large number of commodities turned out to be especially "well-suited" for our approach. Still, problems with this structure are quite common in practice, for instance in telecommunications, so that the results obtained can be of practical interest. Moreover, the limited amount of

CPU time available forced us to select among many possibilities, and the current set of problems seems to be able to indicate both potentials and limitations of the parallel code.

7. Computational Results and Conclusions

In this section, the computational results obtained by both symmetric and asymmetric versions of pMMCFB are presented and discussed. We remark that these results correspond to a solution of the MMCF instances with a relative precision of about 10^{-6} (average relative gap $9.94 \cdot 10^{-7}$, maximum $2.35 \cdot 10^{-6}$) w.r.t. the optimal solution obtained with a commercial LP solver. The relative aggregate unfeasibility (norm of the unfeasibility/$\|\boldsymbol{u}\|$) is about 10^{-5} (average $4.65 \cdot 10^{-6}$, maximum $1.92 \cdot 10^{-5}$). However, any precision, both for optimality and feasibility, is in principle attainable by properly setting the stopping parameters of the code. Also, the convergence of the method is usually pretty fast when near the optimal solution.

7.1. Performance Measures

Let us start with a brief discussion on the *performance measures* (Patton 1989) that have been used. It is now widely acknowledged (Hennessy and Patterson 1990) that a reliable performance metric for any computer system is just *program execution time*, since it measures several characteristics of the system that influence the overall performance. In a sequential computer, these characteristics are the quality of the instruction set, the ALU and FPU performances, the optimizing capability of the compiler, the latency and bandwidth of the memory subsystem, and the clock rate of the chip. When the object of benchmarking is a parallel system, execution time also mirrors communication and synchronization costs, the latency and bandwidth of the interconnection network, and the quality of the routing scheme for inter-processor communication. Thus, we chose execution time as the basic unit of measure.

Let T_p stand for the execution time of a program on a system with p identical processors, and T^* be the time it takes to execute the "best" known serial algorithm for the same problem, on the same data set, on a single processor; widely accepted efficiency measures are the *absolute speedup* $s_p^* = T^*/T_p$, the *relative speedup* $s_p = T_1/T_p$, and the *relative efficiency* $\mu_p = s_p/p$. μ_p is usually less than one (actually, this has to be true under fairly general assumption), hence $s_p \leq p$, indicating efficiency loss due to communication overhead, duplication of operations, and active wait due to load imbalance or message dependencies. Therefore, *linear speedup* ($s_p = p$) is the best one can hope for, although *superlinear* speedups ($s_p > p$) can sometimes be observed; this phenomenon is usually due to *memory hierarchies*, i.e., to reduction of cache misses and page faults that arises from the splitting of the problem data among the different PEs. However, linear speedup is usually quite difficult to achieve, because of the limit on the amount of parallelism that can be extracted from an algorithm as described by Amdahl's law

$$\mu_p \leq \mu_p^I = \frac{1}{(p-1)s+1},$$

where $s \in [0, 1]$ is the fraction of inherently sequential (non-parallelizable) code. For pMMCFB, s is all the time that is *not* spent in $\varphi(\ )$ calculation, i.e., mainly the time for the solution of $(\Pi'_{\beta t})$.

In certain cases, however, the above measures can be difficult to compute. For problems such as linear programming, for instance, there is no uncontested "best" serial algorithm; however, for the instances at hand MMCFB is one of the most efficient approaches (Frangioni and Gallo 1999), and therefore $s_p^* = s_p$. Moreover, the absolute speedup does not reflect the *degree of parallelization* of the algorithm, as measured against its serial implementation, while the relative speedup does; thus, s_p and μ_p seem to be highly indicated as the right performance measures in our case. However, problems arise with the asymmetric pMMCFB, where different parts of the code are run on very different machines; clearly, the above definitions cease to be sensible, and the definition of appropriate efficiency measures unclear. The best solution is probably that of presenting the data "as are" and leaving the conclusion to the reader; since the purpose of this research was essentially practical, we will be satisfied by showing that the asymmetric implementation could allow us to reduce the overall solution time.

Table 3 Aggregated Computational Results for the Cray T3D

Group	T_1	s%	T_4	T_{16}	T_{64}	μ_4^I	μ_{16}^I	μ_{64}^I	μ_4	μ_{16}	μ_{64}
64-64	21.31	1.10	5.98	2.08	1.00	0.97	0.86	0.61	0.90	0.64	0.34
128-64	123.66	1.25	35.70	13.16	7.01	0.96	0.86	0.62	0.88	0.65	0.34
128-128	159.78	0.66	42.04	12.65	4.95	0.98	0.91	0.73	0.96	0.78	0.51
256-64	466.35	1.51	129.75	44.69	21.89	0.96	0.83	0.58	0.90	0.69	0.39
256-128	718.35	0.62	188.96	57.23	22.99	0.98	0.92	0.74	0.96	0.79	0.50
256-256	1404.48	0.30	348.46	98.30	33.85	0.99	0.96	0.85	0.99	0.88	0.65
512-512	15898.89	0.22	*	1025.26	291.40	*	0.97	0.88	*	0.99	0.86

7.2. Results of the Symmetric pMMCFB (Cray T3D)

In this section, the computational results obtained by the symmetric pMMCFB on the Cray T3D are reported. In each table, T_p denotes the running time, in seconds, taken by the solution of the problem (excluding loading and preprocessing) with p processors, μ_p^I the *ideal efficiency* according to Amdahl's law, μ_p the actual relative efficiency achieved, and s% an estimate of the percentage of inherently sequential code.

In Table 3, the aggregated computational results for all the problem classes are reported. The results are visualized in the corresponding Figure 2, where μ_p is plotted as a function of both p and the problem class, and in Figures 3, 4, and 5, where μ_p^I and μ_p are compared for $p = 4, 16$, and 64 respectively.

These results show that satisfactory efficiencies (always over 64%) are consistently obtained when using at most 16 PEs, which is the typical number of processors for most of the experiences reported in the literature. However, as p further increases, μ_p soon degrades down to 34% on some classes of instances.

Yet, Figure 2 shows that μ_p also depends on the "size" of the problem in the following ways:

— for a fixed graph size, μ_p increases as k does (cf. the 256-64, 256-128, and 256-256 problems): increasing k, the relative weight of the non-parallelizable code (measured by s) decreases;

— for a fixed m/k, μ_p increases when m does (cf. 64-64, 128-128, and 256-256 problems): thus, although both the communication and the QP solution costs increase with m, this increase is largely compensated by the corresponding decrease of s;

— for a fixed k, μ_p is "stable" (cf. 64-64, 128-64, and 256-64 problems): thus, the QP solution cost at least does not increase much faster than the subproblem solution cost as m increases.

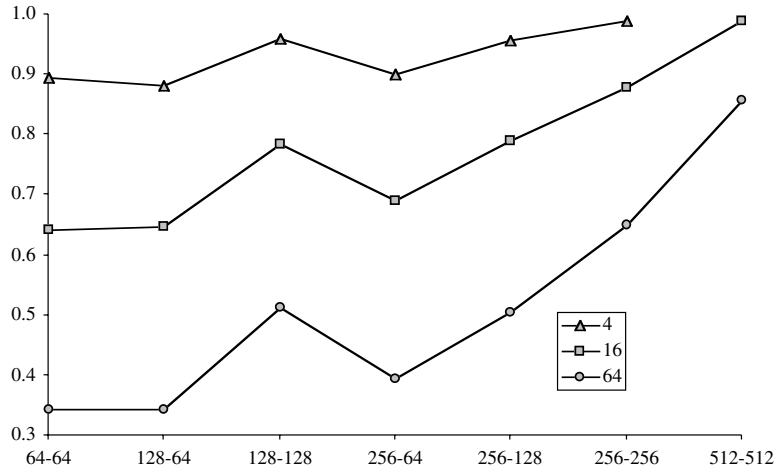

Figure 2 Efficiency as a Function of *p* and the Problem Group from Table 3

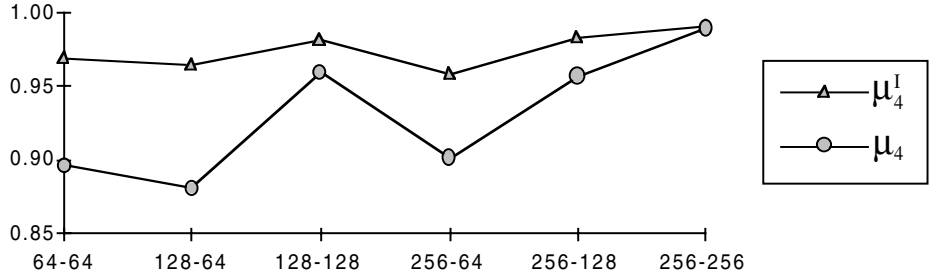

Figure 3 Ideal and Actual Efficiency for $p = 4$

Furthermore, the gap between ideal and actual efficiency seems to follow an analogous law: although large for "small" problems, and increasing with the number of processors p, it is smaller for instances with relatively many commodities, and it reduces (up to, eventually, closing) as the size of the problem increases.

These observations are particularly relevant because the *sequential* MMCFB becomes more and more competitive w.r.t. a number of alternative approaches as the size of the problem increases, and for "small" values of m/k (Frangioni and Gallo 1999). Hence, pMMCFB seems to be particularly well-suited for large instances with "many" commodities, since for these problems the code is effective in sequential mode, and the efficiency is high even with a large number of processors.

Other observations can be made by analyzing the results in a little more detail. Table 4 reports the results for each single problem group; note that it contains violations of Amdahl's law (128-128-*C* with 4 PEs) and superlinear speedups (256-256-*C* and *D* with 4 PEs, 512-512-*A* with 16 PEs), presumably due to the above-mentioned effects of memory hierarchies. Table 4 shows that communication time does not appear to be a significant factor. This is predicted by estimates based on the declared performances of the machine, and is confirmed by the fact that the gap between μ_p^I and μ_p diminishes as m (and therefore the amount of exchanged data per iteration) increases, up to closing in several cases, clearly indicating that the loss of efficiency is due to master-slaves synchronization rather than to communication overhead. The largest problems in the suite always attain very high efficiencies, even with 64 PEs, and little or no gap between μ_p^I and μ_p; for the *B* group, the parallel code reduces the "wall-clock time" of a factor over 50, from more than 6 hours to about 7 minutes.

Further insight on the nature of the results can be obtained by examining the results for each of the 12 instances of any particular class of problems; in

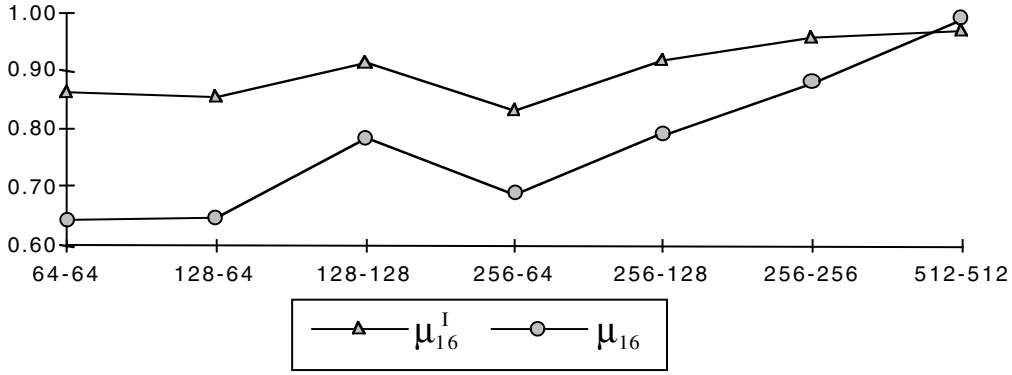

Figure 4 Ideal and Actual Efficiency for $p = 16$

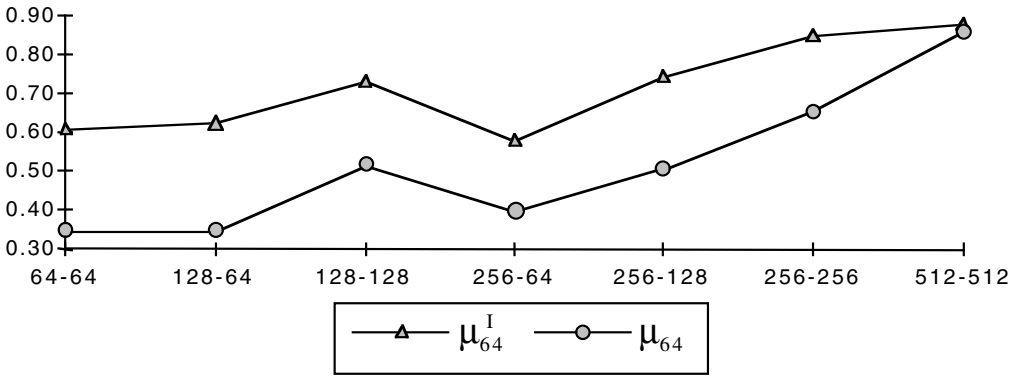

Figure 5 Ideal and Actual Efficiency for $p = 64$

Table 5, this is done for the 64-64 problems, where pMMCFB performs significantly *worse* than in all other cases. The instances are ordered by increasing *loss of efficiency* $\Delta\mu = \mu^I_{64} - \mu_{64}$, and the percentage of arcs having single-commodity constraint is shown as *sc*. The results show that problems can be essentially divided into two suites: the first six, having $\Delta\mu \leq 0.27$, mostly belong to the *C* and *D* classes, while the others, having $\Delta\mu \geq 0.3$, are mostly of groups *A* and *B*. Thus, the problems in the first group are larger,

Table 4 Disaggregated Computational Results for the Cray T3D

Group	T_1	*s*%	T_4	T_{16}	T_{64}	μ^I_4	μ^I_{16}	μ^I_{64}	μ_4	μ_{16}	μ_{64}
64-64-*A*	3.27	0.73	0.90	0.31	0.15	0.95	0.79	0.48	0.89	0.64	0.34
64-64-*B*	8.56	0.85	2.38	0.82	0.37	0.97	0.85	0.58	0.90	0.65	0.33
64-64-*C*	14.52	1.06	4.04	1.42	0.70	0.98	0.89	0.65	0.90	0.63	0.34
64-64-*D*	58.89	1.78	16.59	5.78	2.77	0.98	0.91	0.71	0.89	0.66	0.35
128-64-*A*	6.97	0.55	1.97	0.65	0.30	0.98	0.92	0.74	0.89	0.67	0.36
128-64-*B*	26.47	0.71	7.56	2.52	1.11	0.98	0.90	0.69	0.88	0.66	0.37
128-64-*C*	82.37	0.77	23.03	7.73	3.67	0.98	0.90	0.68	0.89	0.67	0.36
128-64-*D*	378.85	2.98	110.25	41.73	22.98	0.92	0.70	0.38	0.87	0.60	0.29
128-128-*A*	23.91	0.21	6.06	1.94	0.68	0.99	0.97	0.88	0.99	0.77	0.54
128-128-*B*	30.90	0.41	8.60	2.48	0.96	0.99	0.94	0.80	0.92	0.78	0.50
128-128-*C*	177.67	0.86	44.80	14.01	5.63	0.97	0.89	0.65	0.99	0.79	0.50
128-128-*D*	406.63	1.16	108.70	32.18	12.54	0.97	0.85	0.58	0.94	0.79	0.51
256-64-*A*	46.28	0.80	13.32	4.23	1.77	0.98	0.89	0.67	0.88	0.70	0.41
256-64-*B*	100.52	0.63	28.89	9.09	3.56	0.98	0.91	0.72	0.90	0.70	0.43
256-64-*C*	598.23	1.60	160.19	54.88	25.92	0.95	0.82	0.55	0.94	0.71	0.40
256-64-*D*	1120.37	3.02	316.60	110.56	56.33	0.92	0.70	0.37	0.89	0.65	0.33
256-128-*A*	155.91	0.21	41.85	12.44	4.92	0.99	0.97	0.88	0.95	0.79	0.48
256-128-*B*	282.99	0.38	76.40	21.74	7.87	0.99	0.95	0.81	0.94	0.81	0.54
256-128-*C*	702.94	0.70	183.43	57.79	21.63	0.98	0.91	0.70	0.96	0.76	0.51
256-128-*D*	1731.54	1.18	454.18	136.93	57.55	0.97	0.85	0.58	0.97	0.80	0.48
256-256-*A*	431.58	0.12	115.39	31.60	10.16	1.00	0.98	0.93	0.94	0.85	0.65
256-256-*B*	744.50	0.22	204.28	54.91	17.16	0.99	0.97	0.88	0.92	0.85	0.68
256-256-*C*	1381.11	0.41	322.95	95.71	34.11	0.99	0.94	0.80	1.07	0.90	0.64
256-256-*D*	3060.72	0.46	751.23	210.98	73.98	0.99	0.93	0.77	1.03	0.91	0.64
512-512-*A*	7662.02	0.16	*	467.91	138.20	*	0.98	0.91	*	1.02	0.87
512-512-*B*	24135.76	0.28	*	1582.61	444.61	*	0.96	0.85	*	0.95	0.85

Table 5 **Even More Disaggregated Computational Results for the 64-64 Problems**

	$\Delta\mu$	sc	T_1	s%	T_4	T_{16}	T_{64}	μ_4^I	μ_{16}^I	μ_{64}^I	μ_4	μ_{16}	μ_{64}
11(*D*)	0.10	60%	80.20	2.24	22.96	8.13	3.92	0.94	0.75	0.42	0.87	0.62	0.32
10(*D*)	0.14	30%	60.77	1.79	16.90	5.77	2.85	0.95	0.79	0.47	0.90	0.66	0.33
12(*D*)	0.19	90%	35.71	1.31	9.91	3.43	1.56	0.96	0.84	0.55	0.90	0.65	0.36
8(*C*)	0.23	60%	19.57	1.35	5.38	1.92	0.97	0.96	0.83	0.54	0.91	0.64	0.31
7(*C*)	0.25	30%	16.69	1.22	4.61	1.65	0.82	0.96	0.85	0.57	0.90	0.63	0.32
4(*B*)	0.27	30%	18.65	0.90	5.17	1.73	0.79	0.97	0.88	0.64	0.90	0.67	0.37
1(*A*)	0.30	30%	5.57	0.87	1.54	0.53	0.25	0.97	0.88	0.65	0.91	0.66	0.34
5(*B*)	0.31	60%	4.89	0.83	1.34	0.52	0.22	0.98	0.89	0.66	0.91	0.59	0.34
6(*B*)	0.32	90%	2.16	0.84	0.61	0.21	0.10	0.98	0.89	0.65	0.88	0.63	0.34
3(*A*)	0.32	90%	2.26	0.72	0.63	0.22	0.10	0.98	0.90	0.69	0.90	0.65	0.37
9(*C*)	0.35	90%	7.31	0.60	2.11	0.68	0.30	0.98	0.92	0.73	0.87	0.67	0.38
2(*A*)	0.41	60%	1.96	0.59	0.55	0.19	0.10	0.98	0.92	0.73	0.89	0.65	0.32

and therefore more "difficult" to solve. The exception, i.e., the swap 4(*B*)-9(*C*), is still meaningful since, as remarked in §5, 4(*B*) is the most difficult among the small problems whereas 9(*C*) is the easiest among the big ones.

This relation between $\Delta\mu$ and the "difficulty" of the problem is also confirmed by the sequential times, and by the fact that, on average, *D* problems perform better than *C* ones and *B* problems perform better than *A* ones. This is easily justified by observing that, using 64 PEs, each PE is assigned exactly one MCF; for the small or easy problems, time spent waiting for synchronization is relatively large, resulting in poor performance.

All the results so far may perhaps be summarized as follows: the larger and more difficult an instance is, the better its relative efficiency is likely to be. This possibly indicates that the present approach is well-suited for large-scale, difficult MMCFs with many commodities, i.e., the class of problems where a parallel approach appears to be really instrumental.

7.3. Results of the Asymmetric pMMCFB (Cray T3D + Cray C90)

As noted in §7.1, it is difficult even to define the speedup of the asymmetric pMMCFB; at least two definitions are possible, as T_1 can be that of MMCFB on the C90 *or* on the T3D, and both these choices disregard the fact that the two architectures behave differently on different parts of the code (cf. Table 1). Some technical difficulties further complicate the matter. The main problem was that, in the available environment, the C90 was *not* run in dedicated mode, so that the process running the master was repeatedly de-scheduled during the execution of the algorithm, increasing the idle time of the (dedicated) T3D. Actually, since PVM messages pass through the PVM-daemon, receiving one message requires *two* processes being scheduled for execution. The effect of this extra idle time was dramatic: the wall-clock time of the asymmetric pMMCFB actually *increased* up to an order of magnitude, and was also fluctuating with the workload of the C90 (that was, however, always very high). In order to estimate the results that might be obtained with a dedicated C90, we therefore approximated the total execution time by the sum of the CPU time of the C90 and the wall-clock time of the T3D, excluding from both the time spent in the C90-T3D communications; the latter was a very large fraction of the wall-clock time, since it includes the active waiting due to the de-scheduling of the master process on the C90. Such an estimate can be assumed to be tight, since it only disregards the very low cost of the actual data transfer, at least if the C90 does *not* solve MCFs (so that it is immediately available when the T3D finishes), i.e., if the two machines are *never* working at the same time.

This is how the execution times of the asymmetric pMMCFB, reported in Table 6, have been computed. In the table, T_p has to be interpreted as the time obtained with p PEs of the T3D, i.e., excluding the

Table 6 **Time Comparison for the C90: The Symmetric and Asymmetric Versions**

		Symmetric		Asymmetric				
	C90	T_{16}	T_{64}	T_1	T_{16}	T_{64}	Δt_{16}	Δt_{64}
128-128-*A*	21.41	1.72	0.66	20.81	1.72	0.66	0%	0%
128-128-*B*	30.46	2.42	0.94	30.22	2.38	0.96	1%	−1%
128-128-*C*	174.99	14.35	5.91	171.80	13.90	5.82	3%	1%
128-128-*D*	405.03	30.97	12.32	384.71	29.45	10.77	5%	13%
256-128-*A*	177.12	13.76	5.35	175.87	13.92	5.28	−1%	1%
256-128-*B*	279.83	20.94	7.17	277.20	20.33	7.29	3%	−2%
256-128-*C*	522.13	42.77	16.98	530.81	42.32	15.28	1%	10%
256-128-*D*	1610.55	129.47	54.94	1639.63	117.80	44.32	9%	19%

C90 (for the asymmetric version) from the processors count; this is fair, as the C90 can be seen as a "specialized coprocessor" of PE_0 of T3D, which is basically only used to solve more efficiently ($\Pi'_{\beta t}$). Δt_p is the relative difference between the corresponding values, which is reported as a very rough measure of the *potential* wall-clock time improvement obtainable on a system with a dedicated C90.

Although clearly still preliminary, these results seem to show that the asymmetric version can be significantly faster, saving up to 20% of total time especially on the larger and more difficult groups. Note that, due to limitations of the CPU time available, we only ran this set of experiments on the medium-size 128-128 and 256-128 classes, although presumably larger benefits might have been obtained with some of the larger instances.

7.4. Conclusions

The aim of this work was to test whether the exploitation of the natural parallelism inherent in any decomposition algorithm for multicommodity min-cost flow problems, i.e., the parallel solution of k MCFs at each step, can lead to an efficient parallel code even by starting with a complex-coordinator code not originally designed for a parallel implementation. The parallel code, extending the effective sequential bundle-based cost-decomposition solver MMCFB, turned out to be highly efficient even with a large number of processors, with, e.g., the running time of the largest instances reduced from more than 6 hours to about 7 minutes. Perhaps the most interesting finding was that the classes of problems on which pMMCFB attained its best results were exactly those where MMCFB is known to outperform several other approaches, i.e., "difficult" problems with relatively "many" commodities w.r.t. the size of the graph. The results of the basic version can be further improved, at least ideally, by exploiting—apparently for the first time—a quite-common characteristic of current supercomputing facilities, i.e., the side-to-side availability of vector and massively parallel supercomputers. We believe that these results show a potential for parallel bundle-based decomposition approaches as tools for the effective solution of large-scale, difficult MMCFs.

Acknowledgments

This research has been supported by the Italian National Research Council (CNR) within the national program on transportation "Progetto Finalizzato Trasporti 2." Access to the Cray T3D has been made possible by grants provided by CINECA (Casalecchio di Reno, BO); in an early phase of the project, we used a temporary account of the Fourth Summer School on Parallel and Vector Supercomputing attended by one of the authors. The authors acknowledge Giovanni Erbacci and Francesca Durì for their valuable help and suggestions during one author's stay at CINECA and thereafter.

References

Bertsekas, D. P. 1991. *Linear Network Optimization: Algorithms and Codes*. MIT Press, Cambridge, MA.

Cappanera, P. 1996. Parallelizzazione di un algoritmo di decomposizione per problemi di flusso di costo minimo di tipo multicommodity. *Master's thesis*, Department of Computer Science, University of Pisa, Pisa, Italy.

Castro, J. 2000. A specialized interior-point algorithm for multicommodity network flows. *SIAM J. Optim.* **10** 852–877.

Castro, J., A. Frangioni. 2001. A parallel implementation of an interior-point algorithm for multicommodity network flows. *Lecture Notes in Computer Science*, **No. 1981**, Vector and Parallel Processing—VECPAR 2000, J. M. Palma. J. Dongarra and V. Hernandez eds. Springer-Verlag, 301–315.

Crainic, T. G., A. Frangioni, B. Gendron. 2001. Bundle-based relaxation methods for multicommodity capacitated fixed charge network design problems. *Discrete Appl. Math.* **112** 73–99.

Cray Research Inc. 1994a. PVM and HeNCE Programmer's Manual, Eagan, MN.

Cray Research Inc. 1994b. SHMEM Technical Note for C, Eagan, MN.

De Leone, R., M. Gaudioso, M. F. Monaco. 1993. Nonsmooth optimization methods for parallel decomposition of multicommodity flow problems. *Ann. Oper. Res.* **44** 299–311.

De Leone, R., R. R. Meyer, S. Kontogiorgis, A. Zakarian, G. Zakeri. 1994. Coordination in coarse-grained decomposition. *SIAM J. Optim.* **4** 777–793.

De Silva, A., D. Abramson. 1998. A parallel interior point method and its application to facility location problems. *Comput. Optim. Appl.* **9** 249–273.

Ferris, M. C., J. D. Horn. 1998. Partitioning mathematical problems for parallel solution. *Math. Programming* **80** 35–61.

Ferris, M. C., O. L. Mangasarian. 1992. Parallel constraint distribution. *SIAM J. Optim.* **1** 487–500.

Ferris, M. C., O. L. Mangasarian. 1994. Parallel variable distribution. *SIAM J. Optim.* **4** 815–832.

Frangioni, A. 1996. Solving semidefinite quadratic problems within nonsmooth optimization algorithms. *Comput. Oper. Res.* **23** 1099–1118.

Frangioni, A. 1997. Dual-ascent methods and multicommodity flow problems. *Ph.D. thesis TD* **5/97**, Dipartimento di Informatica, Università di Pisa, Pisa, Italy.

Frangioni, A. 2002. Generalized bundle methods. *SIAM J. Optim.* Forthcoming.

Frangioni, A., G. Gallo. 1999. A bundle type dual-ascent approach to linear multicommodity min cost flow problems. *INFORMS J. Comput.* **11** 370–393.

Gnanendran, S. K., J. K. Ho. 1993. Load balancing in the parallel optimization of block-angular linear programs. *Math. Programming* **62** 41–67.

Grigoriadis, M. D., L. G. Khachiyan. 1995. An exponential-function reduction method for block-angular convex programs. *Networks* **26** 59–68.

Hennessy, J. L., D. A Patterson. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA.

Ho, J. K., T. C. Lee, R. P. Sundarraj. 1988. Decomposition of linear programs using parallel computation. *Math. Programming* **42** 391–405.

Klein, P., A. Agrawal, R. Ravi, S. Rao. 1990. Approximation through multicommodity flow. *Proc. 31th IEEE Ann. Sympos. Foundations of Comput. Sci.*, 726–727.

Kontogiorgis, S. A., R. De Leone, R. R. Meyer. 1996. Alternating directions splittings for block angular parallel optimization. *J. Optim. Theory Appl.* **90** 1–29.

Lustig, I. J., G. Li. 1992. An implementation of a parallel primal-dual interior-point method for block-structured linear programs. *Comput. Optim. Appl.* **1** 141–161.

Medhi, D. 1990. Parallel bundle-based decomposition for large-scale structured mathematical programming problems. *Annals of Operations Research* **22** 101–127.

Meyer, R. R., G. L. Schultz. 1992. An interior point method for block-angular optimization. *SIAM J. Optim.* **1** 121–152.

Patton, P. C. 1989. Performance limits for parallel processors. G. F. Carey, ed. *Parallel Supercomputing: Methods, Algorithms and Applications*, John Wiley & Sons, New York, 1–16.

Pinar, M. C., S. A. Zenios. 1992. Parallel decomposition of multicommodity network flows using a linear-quadratic penalty algorithm. *ORSA J. Comput.* **4** 235–249.

Zenios, S. A. 1991. On the fine-grain decomposition of multicommodity transportation problems. *SIAM J. Optim.* **1** 643–669.

Zenios, S. A. 1993. Data-parallel computing for network-structured optimization problems. *Comput. Optim. Appl.* **3** 199–242.

Zenios, S. A., J. M. Mulvey. 1988. Vectorization and multitasking of nonlinear network programming algorithms. *Math. Programming* **42** 449–470.

Accepted by Richard S. Barr; received March 1996; revised August 2000, December 2000; accepted May 2002.