Università di Pisa

Dipartimento di Informatica

Technical Report : TR-96-36

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

Paola Cappanera, Antonio Frangioni

November 07, 1996

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

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

Paola Cappanera[‡]

Antonio Frangioni[‡]

Abstract: *We study the coarse-grained parallelization of a Bundle-based cost decomposition algorithm for the solution of linear Multicommodity Min Cost Flow problems, that has already been shown to be effective in sequential. The aim of the study was to investigate if exploitation of the natural parallelism inherent in the cost decomposition method, i.e. the simultaneous solution of the Min Cost Flow subproblems, was enough to obtain acceptable efficiencies without modifying the basic approach: as a result, we obtained an highly portable and flexible code which can be used on different machines. The computational results show that satisfactory efficiencies can be obtained even with many processors on large, difficult MMCF problems with many commodities, that are exactly the class of instances where the sequential code attains the best results. We also show how to exploit a common characteristic of nowadays supercomputer facilities, i.e. the side-to-side availability of massively parallel and powerful vector supercomputers, by implementing an asymmetric decomposition algorithm where each architecture is used for the tasks it is suited most.*

Keywords: Parallel Algorithms, Multicommodity Flows, Bundle methods

This research has been supported by the Italian National Research Council (CNR) within the national program on transportation “Progetto Finalizzato Trasporti 2”.

‡ Dipartimento di Informatica, Università di Pisa
Corso Italia, 40 - 56100 Pisa - Italy
e-mail: cappaner@di.unipi.it, gallo@di.unipi.it

0. Introduction

The Multicommodity Min Cost Flow problem (MMCF) is a generalization of the ordinary single-commodity Min Cost Flow problem, in which flows of different nature (*commodities*) must be shipped 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 hard graph problems of large interest [BL84] [KAR90], while from a practical point of wiev, MMCF and its variants are (sub)models of a wide variety of hard transportation and scheduling problems [AKL93] [Ba93] [BS93], where often very large instances of the "simple" linear problem have to be repeatedly solved in order to approach the solution of the hard integer version. Although MMCF is a structured Linear Program, the usual LP techniques fail to be efficient enough on practical problems, and several specialized algorithms have been proposed for its solution during the last two decades.
Among the approaches to MMCF, *decomposition methods* are perhaps the most intriguing and successful, as demonstrated by the constant stream of work dedicated to this class of algorithms: in a previous work [FG96], a large computational experience with a Bundle-type cost decomposition approach to MMCF has shown this approach to be efficient in practice, especially on problems where the number of commodities is "high" w.r.t. the dimension of the graph. During the last few years, a large theoretical and practical effort [ZM88] [Me90] [CCZ91] [Fe91] [FM91] [Ze91] [PZ92] [GH93] [LGM93] [FH94] [FM94] [Ko94] [LKM94] [Ze93] [ZaA95] [ZaG95] has been done to develop parallel approaches to MMCF, demonstrating the interest that the solution of large-scale Multicommodity flow problems has in the OR community: in this work, the results obtained by the parallelization of the efficient Bundle-type code are presented and discussed. The presented approach can be classified among those with *complex coordinator*, that seem to be best suited for implementation on coarse-grained massively parallel machines: however, differently from all the previous attempts, the parallel code directly derives from an original sequential implementation, that had not initially been thought for parallelization, and it has not been substantially modified to best suit the parallel machine where it has been ported; furthermore, while convincing proofs of the actual effectiveness of the sequential code are available, this is not true for almost all the other known approaches.

The structure of the work is the following: in §1 the MMCF problem is introduced, and the parallel approaches to the problem proposed in the literature are briefly reviewed; §2 describes the sequential Bundle approach, while in §3 we discuss the issues related to its coarse-grained parallelization. In the remaining paragraphs, the computational experiences we performed are described: in §4 the details of the available (hardware and software) environment are given, in §5 the test problems are introduced, and finally in §6 the computational results are presented and conclusions are drawn.

1. MMCF: formulation and parallel approaches

We consider the (Linear) Multicommodity Min Cost Flow Problem (MMCF): given a directed graph $G(N, E)$, where $|N| = n$ and $|E| = m$, a set of k commodities (types of flow) have to be routed on G at minimal total cost. Flow x_{ij}^h of the h-th commodity on arc (i, j) has individual lower and upper bounds

(respectively 0 and u_{ij}^h) and a linear cost function $c_{ij}^h x_{ij}^h$, while *mutual* (or *aggregate) capacity* constraints $\sum_h x_{ij}^h \le u_{ij}$ limit the total quantity of flow on arc (i, j), regardless of the commodity.
A formulation of the problem is

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

where (a) are the (usual) independent flow conservation constraints for each commodity and (b), (c) are respectively the "individual" and mutual capacity constraints: when restated in matrix notation, using the node-arc incidence matrix $\boldsymbol{E}$ of G, this (node-arc) formulation highlights the block-structured nature of MMCF

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

A special case of MMCF is the one in which no single-commodity upper bounds exist (i.e. $u_{ij}^h = +\infty\ \forall i, j, h$): in this case, the k independent MCF subproblems that are obtained relaxing the mutual capacity constraints (c) become Shortest Path Tree problems. From the point of view of the number of different commodities, real life MMCFs range from the case in which the commodities are just a few (as in many distribution problems) to the case in which k approaches the number of all the possible O/D pairs (as in traffic circulation models): of course, the capability of exploiting the special structure of the problem to be solved is crucial to the design of efficient algorithms.

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: decision variables can be broken down to largely independent blocks, that correspond to first-level decisions satisfying a subset of the constraints, which may represent a time period, a geographical region or, such as in our case, a commodity. The blocks interact via coupling constraints related to second-level coordination of block decisions, such as shared resource allocation restrictions: therefore, 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 to 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, decoupled optimization subproblems corresponding to blocks are solved, the subproblems being autonomous processes, i.e. substantially independent of each other; in the global phase the solution information is combined and a coordination problem is solved, the results of which are used to modify the objective function or the constraints set of the subproblems in the next step.

Cost (otherwise called *Price-directed*) *Decomposition* methods determine a separable function φ() and a convex (usually polyhedral) set Λ such that solving the original problem is the same as solving max{ φ(λ) : λ ∈ Λ }: typical pricing mechanisms are putting the coupling constraints in the objective via Lagrangean relaxation, in the form of a barrier function or using other penalty terms. The separability of the objective function yields a coarse-grained parallelism that can be exploited, within a distributed-memory environment, by assigning dedicated processors to the Master problem and to each of the subproblems: the resulting algorithm has the common "Master-Slave" structure, where at each iteration the "Master" calculates the new costs λ associated with the mutual constraints and broadcast them to the "Slaves", that concurrently uses them to modify their objective function and solve the resulting subproblems, then a new Master phase takes place, during which the Master aggregates solution information coming from Slaves to calculate φ(λ). The execution of a such algorithm alternates between a serial Master phase, during which the subproblem processors are idle, and a concurrent Slave phase during which the Master problem is idle: clearly, the performance of distributed decomposition, as measured by *speedup* or *efficiency*, is adversely affected by the magnitude of the serial work load in the implementation. Thus, efficiency can be expected to be low when the amount of computational effort expended on the Master problem is significant - as might be expected, for instance, when the coupling constraints are numerous, or when they are "tight" - so that efficiency is limited by a factor that depends on the ratio of the duration of the Master phase to the duration of the bottleneck subproblem (the Amdhal's Law), although speedup can often be improved by scaling the problem size with the number of processors. Attempts at improving efficiency should, therefore, focus directly on achieving better load balance between the Master and subproblem processors: deciding which tasks the Master has to accomplish is a possibly critical point of any such parallel algorithm, since there is a trade-off between the complexity of the coordinating mechanism used in the decomposition process and the overall performance in a parallel environment. According to [GH93] [LKM94], Cost Decomposition methods can be divided into methods based on *complex coordinators* (that require the solution of an optimization problem) or *simple coordinators* (that require only summing and averaging of subproblem information): methods based on simple coordinators are, on the surface, better suited to parallel environments since they avoid the potential serial bottleneck of solving the possibly large single coordination problem, but, on the other hand, this coordination approach may lead to a large number of iterations and hence slow convergence relative to methods with complex coordinators.

Simple coordinators

The Alternating Directions Methods (ADI) [GM75] [DR56] [PR55] [Ko94] solve a general optimization problem involving the minimization of the sum of two convex functions subject to linear constraints: in the Slave phase each processor (hereafter called PE) solves a minimization subproblem involving constraints and variables pertaining to its assigned block, and a modified objective that consists of a block of the original separable objective plus a Lagrangean linear term and a quadratic proximal term; the Master phase is very simple, as processors only combine their solution in order to form, as aggregates, new values for the subproblem proximal terms and Lagrangean multipliers.
Other methods based on a simple coordinator are the Parallel Constraint Distribution (PCD) method [FM91], that distributes the constraints among the PEs and modifies each subproblem objective function with augmented Lagrangean terms from other PEs, and the Parallel Variable Distribution

(PVD) method [FM94], that distributes the variables among the PEs, so that each PE has primary responsibility for updating its own block of variables while allowing the remaining ones to change in a restricted fashion - the process is followed by a Master problem of controlled size, in which the affine hull of (some of) the points computed in the Slave phase is searched for an optimal point.

Complex coordinators

The classic Dantzig-Wolfe decomposition method and the Bundle method are based on complex coordinators: at each step, the Master gathers informations arising from the previous Slave phase, partially redefines the Master problem by adding one or more columns, in the DW method, or a subgradient, in Bundle methods, and solves it; the solution is then broadcasted to the Slaves to create a new set of prices and the cycle repeats. We remark here the Master problem of DW method is a Linear Program, whereas in the Bundle approach it is a Quadratic Program, which is usually more expensive than linear ones: however, in a Bundle method the result of the Slave phase is usually stored in an aggregated mode, and this drastically reduces the size of the Master problem w.r.t. usual DW implementations where at each iteration the Master has to store k columns, one for each Slave - not surprisingly, is possible to see [CFN96] that DW and Bundle methods are in fact closely related to each other. Also, analogies exist with the Linear Quadratic Penalty algorithm [DPZ91] [PZ92], that is based on penalizing the violation of the coupling constraints via a smoothed convex exact penalty function, and solving the resulting nonseparable problem via a Simplicial Decomposition approach: the Master subproblem, that is very similar to the "stabilized Master problem" of Bundle methods, involves heavy dense linear algebra computations and tend to dominate the total solution time, therefore this method greatly benefits of implementations on vector supercomputers.

In [ZaA95], relative merits and disadvantages of simple and complex coordination for a same method are discussed: there, the coupling constraints are moved to the objective function via augmented Lagrangean terms and a Nonlinear Jacobi Algorithm (NJA) is then used to solve the resulting problem. The Master problem may be very simple (solution informations of the Slave phase are summed) or rather complex (based on Multidimensional Search or on Multiple LineSearch): the results seem to indicate that a complex coordination problem makes much better use of the direction information computed by the subproblems, thus improving the convergence of the NJA in practice. Also, the potential serial bottleneck of solving a complex coordinator problem could be attacked by attempting to solve the coordination problem itself in parallel, even though this may not be always possible since such problems may be non linear and, even when linear, may not have structural properties that allow a natural decomposition. In [ZaG95], for instance, the three phase algorithm of [Sc91] [MS91], based on a barrier decomposition method, has been 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 point produced by the coordinator that has the best (least) objective value is then chosen as the next iterate. This process can be viewed as "two-level" coordination, where the first level consists of producing possible iterates and the second is a simple pass through these candidates to pick the best as the next iterate.

2. The (sequential) Bundle approach

To address the solution of MMCF, the Cost Decomposition method of [FG96] (hereafter named 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 corresponding Lagrangean Dual

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

by maximizing the polyhedral (concave, nondifferentiable) function

$$\varphi(\ \lambda\ ) = \Sigma_h\ \min\{\ (\ \boldsymbol{c}^h + \lambda\ )\boldsymbol{x}^h : \boldsymbol{E}\boldsymbol{x}^h = \boldsymbol{b}^h\ ,\ \mathbf{0} \le \boldsymbol{x}^h \le \boldsymbol{u}^h\ \} - \lambda \boldsymbol{u}$$

For any fixed λ, $\varphi(\ \lambda\ )$ can be quickly computed by solving k independent (single-commodity) MCFs: this is especially true in the case of no single-commodity upper bounds for the flow on the arcs, i.e. $\boldsymbol{u}^h = +\infty\ \forall h$, since then all the subproblems are Shortest Path problems w.r.t. the modified costs $(\ \boldsymbol{c}^h + \lambda\ )$.

The solution of (DMMCF) is attained via a Bundle-type algorithm [CFN96], a class of NonDifferentiable Optimization approaches mainly characterized by storing the first-order information about $\varphi()$, obtained in the previous iterations, in a disaggregated form: having visited a (finite) sequence of points $\{\ \lambda_i\ \}$, the "Bundle" $\beta = \{\ \langle\ \lambda_i\ ,\ \varphi(\ \lambda_i\ )\ ,\ \boldsymbol{g}(\ \lambda_i\ )\ \rangle\ \}$ is used to compute a *tentative ascent direction* $\boldsymbol{d}_i$, along which λ_{i+1} is choosen, where $\boldsymbol{g}(\ \lambda\ )$ is a *subgradient* of $\varphi()$ in λ that is readily obtained as $\Sigma_h\, \boldsymbol{x}^h(\lambda) - \boldsymbol{u}$, being $(\ \boldsymbol{x}^1(\ \lambda\ )\ \ldots\ \boldsymbol{x}^k(\ \lambda\ )\ )$ any optimal solution of the Lagrangean Relaxation (RMG_λ). At each step, calculation of $\boldsymbol{d}_i$ requires the solution of the structured Quadratic Programming subproblem

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

where the $\boldsymbol{g}_i = \boldsymbol{g}(\ \lambda_i\ )$, $\alpha_i = \alpha_i(\ \bar{\lambda}\ ) = \varphi(\ \lambda_i\ ) + \boldsymbol{g}_i(\ \bar{\lambda} - \lambda_i\ ) - \varphi(\ \bar{\lambda}\ )$ are the *linearization errors* w.r.t. the *current point* $\bar{\lambda}$ and $t > 0$ is the so-called *trust region parameter*: $(\Delta_{\beta t})$ can be viewed either as (a Lagrangean relaxation of) a "least-square version" of the classical Master Problem of Dantzig-Wolfe Decomposition, or, in the usual terminology of NonDifferentiable Optimization, as (a Lagrangean relaxation of) the problem of finding the minimum norm vector in some inner approximation of the $\max_{i\in\beta}\{\ \alpha_i\ \}$-subdifferential of $\varphi()$ in $\bar{\lambda}$. Furthermore, its Quadratic Dual

$$(\Pi_{\beta t}) \qquad \max_{\boldsymbol{d},v}\ \{\ v - 1/2t \left\| \boldsymbol{d} \right\|^2 : v \le \boldsymbol{d}\boldsymbol{g}_i + \alpha_i\ \ i \in \beta\ \}$$

can be rewritten as

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

i.e. the maximization of the *Cutting Plane model* $\varphi_\beta(\ \lambda\ ) = \min_{i\in\beta}\ \{\ \varphi(\ \lambda_i\ ) + \boldsymbol{g}_i(\ \lambda - \lambda_i\ )\ \} - \varphi(\ \bar{\lambda}\ )$, the polyhedral upper approximation of $\varphi()$ built up with the first order information found so far, minus a simple quadratic penalty term (sometimes called *stabilising term*), weighted with t: the stabilizing term discourage 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 the current point, playing here the role played by the *trust radius* in the Trust Region methods for

smooth optimization (Figure 1).

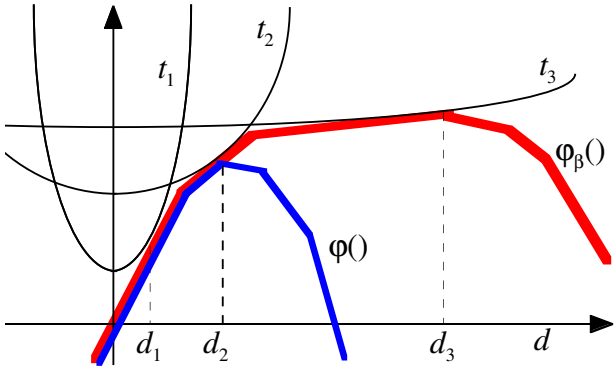

Figure 1: effect of the stabilizing term for $t_1 < t_2 < t_3$ - d_i are the optimal solutions

This gives the "dual" interpretation of $(\Delta_{\beta t})$, in which the Bundle is a representation of the Cutting Plane model $\varphi_\beta()$ - since $\varphi()$ itself is polyhedral, $\varphi_\beta()$ is just the approximation of $\varphi()$ containing only the given (small) subset of the faces in β; for sufficiently large values of t, $(\Pi_{\beta t})$ is equivalent to the "naive" maximization of $\varphi_\beta()$ alone

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

provided that the latter problem has finite maximum, which is very often *not* the case, expecially in earlier stages of the algorithm's operations: Bundle methods can therefore be also seen as a variant of the Cutting Plane algorithm, that uses $\bar{\lambda} + \boldsymbol{d}^*$ (being $\boldsymbol{d}^*$ the solution of (Π_β)) as the next trial point.
Once the tentative ascent direction has been found, there are several ways of exploiting it: basically, either a line search is started along $\boldsymbol{d}_i$, in the style of Line Search methods for smooth optimization, or the value of $\varphi(\bar{\lambda} + \boldsymbol{d}_i)$ and the relative subgradient are used to adjust the parameter t, in the style of Trust Region methods for smooth optimization. Common to both approaches is the possibility not to "accept" the step, i.e. to remain in $\bar{\lambda}$, if an insufficient increase in the value of the objective function is attained: in this case, the newly obtained first-order information is used to update the Cutting Plane model $\varphi_\beta()$, and evenctually t is reduced to be "more conservative" in the following steps. 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 letting the QP subproblem "aware" of the constraints, i.e. by solving the extended problem

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

that guarantees feasibility of the tentative point $\lambda = \bar{\lambda} + \boldsymbol{d}_i$ w.r.t. the $\lambda \geq \mathbf{0}$ constraints.
It is out of the scope of this paper to describe the details of the sequential MMCFB code (the interested reader is referred to [FG96]), but some of its main characteristics are:

- the use of a fast, specialized QP solver [Fr96] that uses a *two-level active set* strategy explicitly designed to be efficient for the kind of reoptimization arising in Bundle algorithms: the C++ code efficiently supports all the operations performed by a Bundle code on the data of subproblem $(\Pi'_{\beta t})$, such as insertion ad deletion of subgradients from β and, in particular, the online *creation and destruction of variables*
- a *Lagrangean variables generation strategy*, where, rather than working with the whole vector of

Lagrangean multipliers, only an "active" subset of the variables (those that at some iteration have the corresponding entry of the direction ***d*** strictly positive) is modified at each iteration we: this is useful because, especially for "easy" problems, few Lagrangean variables ever become nonzero at any point of the algorithm, and the "active set" is very stable after the first few iterations, so that the (relatively costly) "pricing" operation of determining whether new variables have to be added can be performed with a 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 an user-provided "distance estimator" t^*

The MMCFB code has shown to be efficient on several different classes of instances: the key components of its performance are the state-of-the-art (single-commodity) MCF solvers, that keep the cost of the $\varphi()$ calculation low, the specialized Bundle method that reaches convergence in a limited number of iterations and the efficient QP solver that makes the cost of solving $(\Pi_{\beta t}')$ at each step acceptable.

3. Parallelization of the Bundle approach

The initial aim of the project [Ca96] was to directly exploit the existing MMCFB code, instead of developing an "ad hoc" parallel approach: this is important in view of the fact that almost nothing is known about the actual effectiveness, on a serial computer, of most of the parallel approaches proposed in the literature, so that only "relative" measures of parallel efficiency can be obtained - on the contrary, the behaviour of MMCFB on several classes of problems was well-known. Using the inheritance mechanism of C++, the parallel code (hereafter named PMMCFB) has been designed in such a way updates on the sequential code would automatically re-echo on the parallel one; for instance, early versions of MMCFB used a Line Search approach to find the new point λ_{i+1} along the current direction $\boldsymbol{d}_i$, that was subsequently abandoned in favour of a more Trust Region oriented approach, but these changes have not required any modification to PMMCFB.

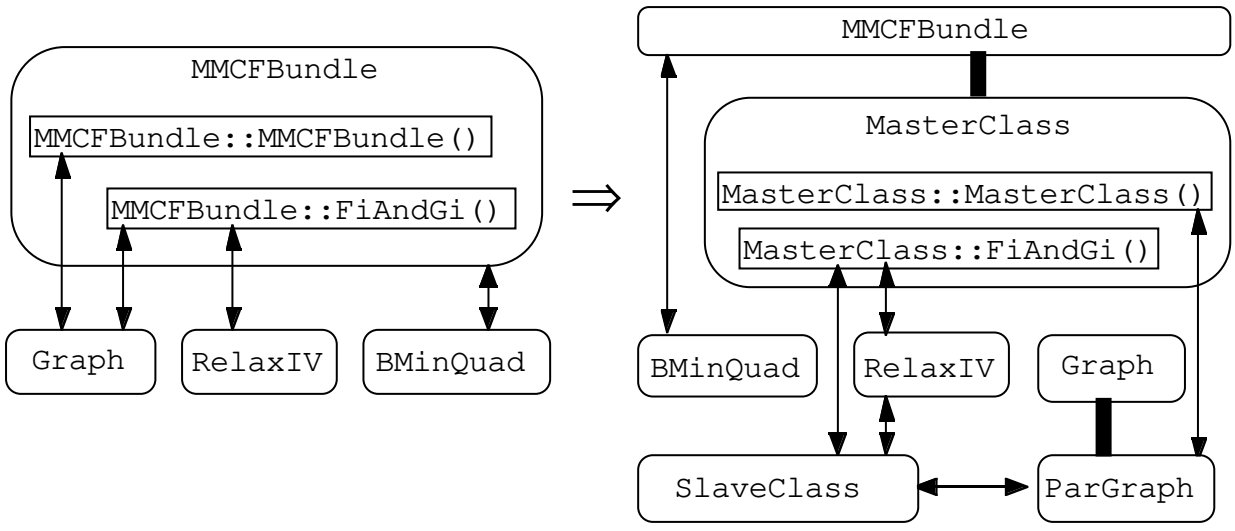

Figure 2: basic structure of the serial and parallel codes

By using a C++ terminology, PMMCFB *derives* from MMCFB, as the latter consists of some C++ *classes* interacting as shown in Figure 2, and the former differs only for the introduction of the three *derived* classes `MasterClass`, `SlaveClass` and `ParGraph` (in the figure, arrows represents functional interactions and thick lines inheritance relationships):

- the `Graph` class provides a mean for reading descriptions of MMCF problems 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 provides *methods* that support both matrix-based and decomposition-based MMCF solvers by hiding the details of the mutual capacity constraints structure, making the code very easy to adapt to different forms of network-structured problems with side constraints. Pre-processing features such as identification of redundant mutual capacity constraints are also provided, together with memory management methods that allow releasing of data that has already been loaded into solver-specific data structures, making the code capable of dealing with very large problems
- 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 `BMinQuad` class implements the specialized BTT algorithm [Fr96] for the solution of the box-constrained QP ($\Pi'_{\beta t}$) - actually, BTT uses an hierarchic approach, and it is in turn *derived* from the `MinQuad` class solving the version of the problem without the box constraints
- the `RelaxIV` class implements a MCF solver based on the RelaxIV algorithm [Be91], that is known to be fast and to reoptimize efficiently: since at each iteration only (a possibly small subset of) the prices of the MCF subproblems change, efficient exploitation of the latest optimal solution is very important in order to reduce the amortized computational cost of the (RMG_λ) solution
- the `ParGraph` class derives from `Graph`, and provides support for splitting the data of the problem among the different PEs: a *constructor* of the class that reads the data from a PVM *channel* can be used to set up the data structures of the instance relative to each slave PE, and methods to properly send them the data are provided for being used at the Master PE, where one of the original `Graph` constructors is used instead
- the `MasterClass` class derives from `MMCFBundle`, and differs from its parent class in the constructor - that splits the data among the Slaves by using the `ParGraph` methods - and in the `FiAndGi()` method only: at each iteration, the sequential code is executed until `FiAndGi()` is invoked, then (for to the *virtual* mechanism) control is passed to `MasterClass::FiAndGi()` where the new prices λ are broadcasted to the Slaves, the results are gathered and the corresponding objective function and subgradient are calculated prior to returning the control to the `MMCFBundle` methods
- the `SlaveClass` class simply implements the MCF 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 constraint would not require changing a single line of both `MasterClass` and `SlaveClass`. Although the Slaves could send their solutions to the Master as soon as possible, the use of a

reduce operation (that involves some form of synchronization) has been preferred, since efficient implementations are given on almost any parallel machine: besides, in a Bundle approach the Master cannot proceed until all the subproblems are terminated, differently from a DW approach where the Master could begin processing proposals received from the Slaves piecemeal, but even there [GH93] such a strategy increases the number of cycles required for overall convergence, thereby cancelling out any benefits due to improved load balance.

Actually, this is a simplified description, since the code has several other useful characteristics: for instance, `MMCFBundle` is in turn derived from `MMCFClass`, an *abstract base class* implementing a standard interface for MMCF solvers that makes application programs highly independent from the particular kind of solver that is used - as a consequence, all the projects where the original sequential code has been used could benefit of a parallel environment without ever knowing that parallel techniques are being 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 MMCF solver, that uses the public methods of such objects for changing the costs, solving the problems and collecting the results, Therefore, making BMMCF to efficiently deal with special classes of instances just amounts at developing the appropriate subproblem solver, and in fact the derived class `SPTree` has already been developed for dealing with SPT subproblems: furthermore, `MMCFBundle` is completely independent from upgrades in the MCF solver, or even changes of algorithm. Moreover, all the modules of the code use a set of "abstract" basic data types, making it very easy to port the code under different architectures and tailoring it to the characteristics of both the computer that is being used and the class of instances that are being solved - for instance, if all the data in the MCF suproblems is integral, the corresponding data structures can be implented using integer rather than floating point numbers. Finally, many different compile-time options are given to the end user for customizing the code without having to bother with 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 paragraph after that the environments where the code has been tested have been introduced in some detail.

4. The parallel environment

PMMCFB has been mainly tested on a 64-processor Cray T3D from Cray Research Inc., incorporating up to 2048 state-of-the-art DEC Alpha microprocessors, each capable of 150 MFLOPS peak performance: a processing element (PE) in the system comprises the Alpha processor, local DRAM memory with a capacity of either 16 or 64 MBytes and custom support logic for the fast bi-directional 3-D torus interconnect network ensuring short connection paths and high bisection bandwidth, with peak interprocessor communication rates of 300 MBytes per second in every direction. 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: in fact, memory (although physically distributed among processors) is globally addressable, but an highly-optimized implementation of PVM [Cr94a] is also provided for portability of the codes. Cray T3D is running UNICOS MAX, a distributed operating system, whose functions are divided between the PEs of the CRAY T3D system and the UNICOS operating system on the Cray C90 host, that allows the user to edit, compile and link programs, and to initiate, control and terminate all T3D

processes. The C90 system installed in CINECA is a C94/2148 and consists of 2 autonomous CPUs with 14 independent functional units and 8 vector registers, capable of almost 240 MFLOPS peak performance (clock cycle 4.2 ns); the Cray T3D system installed in CINECA incorporates 64 PE each with 64 MByte of RAM, 8 KByte of Instruction Cache and 8 KByte of Data Cache.

PMMCFB has been basically designed to work under any distributed environment where a suitable version of PVM was available, but, via compile-time switches, it is also possible to activate architecture-specific portions of code for all the basic operations that may differ between the architectures, i.e. communication and/or process control primitives, up to the extent that it would be quite easy to port the code under any other message-passing paradigm, such as MPI. As an example of the above point, the first thing a PVM task does is to find out if it is the Master process - by convention, the Master is the only task that has not a *parent*, that is the task not started-up by a `pvm_spawn()` - but when operating on T3D in Stand-Alone mode the concept of parent does not apply and therefore PE_0 is chosen to be the Master (`pvm_spawn()` call is not allowed). Analogously, before the Master begins to send messages to the Slaves, they have to send an *ack* to inform him that they have successfully started-up, and the necessary synchronization point can be implemented either in an explicit way through a `pvm_barrier()`, or in a implicit way through message-passing; furthermore, the tasks have to join into a global group to accomplish broadcast and reduce operations, so that the Master have to wait until the number of members of such group don't reach the number of the Slaves plus one (himself): on T3D, where all processes of the user partition are launched at the same time and a global group is pre-defined for all applications, the synchronization point is avoided. As far as communication is concerned, PMMCFB uses `pvm_bcast()` to send a common data to all PEs, `pvm_send()` for point-to-point communications and `pvm_reduce()` to calculate $\boldsymbol{g}(\lambda)$ and $\varphi(\lambda)$, but these calls can be substituted with other architecture-specific primitives, such as the alternative shared memory routines of T3D [Cr94b]; furthermore, to face the case of `pvm_reduce()` not being available, a home-made implementation of the reduce operation, based on a binary tree, is also provided in alternative to (p - 1) `pvm_recv()` in the Master.

MCF		QP	
C90	T3D	C90	T3D
26.94	25.51	0.03	0.05
21.98	20.59	0.03	0.04
11.64	10.70	0.02	0.02
45.71	43.77	0.07	0.15
22.85	21.62	0.04	0.08
16.07	14.83	0.04	0.03
243.88	232.47	0.78	1.78
99.12	91.22	0.16	0.73
133.75	126.35	0.23	0.70
526.24	486.24	1.44	4.63
355.84	325.70	0.75	3.62
233.00	217.45	0.45	1.57

Table 1: times for computing $\varphi(\lambda)$ and $(\Pi'_{\beta r})$, 128-128

MCF		QP	
C90	T3D	C90	T3D
298.03	298.60	0.25	0.45
129.05	121.07	0.09	0.12
80.51	74.58	0.11	0.12
510.35	499.00	0.45	1.28
217.54	211.27	0.17	0.55
80.74	76.40	0.13	0.25
553.34	554.80	0.92	2.55
529.95	526.59	0.79	6.27
379.24	373.49	0.42	1.82
2518.62	2546.11	6.04	26.39
1203.13	1167.85	2.05	11.54
777.06	786.76	1.23	6.63

Table 2: times for computing $\varphi(\lambda)$ and $(\Pi'_{\beta r})$, 256-128

A remarkable characteristic of the available parallel environment was the side-to-side presence of two

radically different architectures, the vector supercomputer Cray C90 and the massively parallel MIMD machine Cray T3D: while the latter is clearly the machine where the $\varphi(\lambda)$ 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) BTT algorithm for the solution of ($\Pi'_{\beta t}$). This self-evident fact was indeed confirmed by a preliminary study, a sample of which is illustrated in Table 1 and Table 2 below: there, the time spent by a MMCFB for calculating $\varphi(\lambda)$ (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 §4). As the table clearly shows, the highly combinatorial RelaxIV code used as a MCF solver in MMCFB is not faster on the powerful C90 than on a single PE of the T3D, while the BTT code is up to six times faster, and the difference tend to increase with the size of the problem (larger in Table 2 than in Table 1, and in the second six problems of each table than in the first six): this is not surprising, since RelaxIV uses virtually no vector operations and is therefore almost no vectorizable, while the costly operations in BTT are all vector ones. Note that, also, all the results have been obtained by only invoking the vectorization options provided by the C90 compiler, but further performance improvements could probably be obtained by hand-made vectorization of the BTT code, that has been originally developed for Unix workstations.

These results clearly suggest the possibility of an *asymmetric version* of PMMCFB, where each of the two basic operations are executed on the machine that is suited most for the task: this called for a number of modifications of the code, that however were always resolved with the above compile-time technique, allowing not to resort to the development of separate codes. Still, some details of the asymmetric implementation are worth noticing: for instance, with PVM operating in Distributed Mode, C90-T3D communications should go via the *PVM-daemon* (running on the C90) and the slow Unix *sockets* system, and since the number of socket connections that the daemon can handle is limited, by default only the T3D PE_0 is given a connection to the daemon - although additional connections can be made, performance rapidly degrades. Therefore, in the specialized implementation the T3D PE_0 acts as a SubMaster between the Master running on C90 and the Slaves running on the other PEs of T3D: otherwise being a "normal" Slave, i.e. solving its share of MCFs, only the SubMaster directly communicates with the C90, so that for most of the time data exchange is performed via the fast T3D interconnection network. For instance, the reduce operation is implemented in two phases, where in the first one the Slaves use a `pvm_reduce()` to accumulate solutions in the SubMaster and in the second one PE_0 sends the results to the Master; 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 cannot be formed. As far as Master-SubMaster communications are concerned, it is possible to choose either PVM or combining I/O and PVM [Cr94a]: the idea is to separate the two parts of a PVM message, control information and actual data, using the slower PVM (Unix sockets) for the first and faster UNICOS I/O features (read/write operation on a memory-mapped file) for the second. Some other (compile-time) choices are left to the user, such as if the Master has to solve a group of MCFs or not: however, let us now proceed with the description of the test set that we used for the actual experiments.

5. 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: for this work, a suite of 78 problems, randomly generated by (an improved version of) the well-known Mnetgen [AK77], have been employed. But for the very large problems, the same data set had been previously used for testing the efficiency of MMCFB [FG96]: there already, the choice of Mnetgen had proven fruitful for analyzing the influence of some characteristics of the MMCF instance to be solved on the relative effectiveness of the several 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, ... , n } (Mnetgen cannot generate problems with $k > n$), 12 different problems have been generated, for a total of 6 *classes* of problems characterised 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, and both these subclasses are in turn subdivided of into 3 "easy" (A and C) and 3 "hard" (B and D) problems: an easy problem has mutual capacity constraints on the 40% of the arcs and 10% of the arcs with an "high" cost, while in a hard problem the arcs with mutual capacity constraints and an "high" cost are the 80% and 30% respectively. 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: 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.

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

Table 3: characteristics of the problems

To test the code on larger problems, the "sparse" problems groups (A and B) with 512 nodes and 512 commodities, whose LP formulation has more than 800.000 variables, were also generated: curiously enough, it was not possible to test the (512, 512) "dense" groups, or problems with more nodes, because of limitations of the *disk space* available on the server machine (each of these problems file takes several megabytes), but the computational results seem to show that even much larger problems could be solved in a very reasonable time.
Table 3 resumes the (averaged) characteristics of the test problems: each problem group is designated by *n-k-g*, where $g \in \{ A, B, C, D \}$ identifies the typology, while k, n, m and b are respectively the number of commodities, the number of nodes, the total number of arcs and the number of arcs having *mutual capacity constraints* of the underlying topological network; the remaining columns describe the characteristics of the LP formulation of the problems, *Constr* being the total number of constraints ($m \times k + b$, flow conservation + coupling) and *Var* being the total number of variables ($m \times k$) of the problem.
Actually, preliminary experiments were conducted on a larger class of problems, typically having less than 64 commodities: lately, those problems were discarded for two reasons, the first being that, with the current approach, it was impossible to exploit all the parallelism provided by the available parallel computers, and the second being that they were essentially too "easy", i.e. solvable within a few seconds on a workstation, so that their parallelization could only be of a limited interest in practice. However, it must be clearly stated that, a posteriori, we found this choices having lead us to selecting essentially the class of problems that are in some sense "best suited" for our approach, as the following computational results will show, i.e. those with a relatively large number of commodities w.r.t. the size of the graph: for instance, we could as well had chosen 1024-256 problems as "the big ones". Still, problems with such a structure are quite common in practice, for instance in many real-world problems related to telecommunication networks, so that the obtained results should be considered of practical interest: moreover, the limited amount of CPU time available forced us to select among the many possibilities, and the current set of problems seems to be able to indicate both potentials and limitations of the proposed approach.

6. Computational results and conclusions

Let us start with a brief discussion on the *performance measures* [BT89] [Pa89] [GW89] that have been used: it is nowadays widely acknowledged [HP90] that a reliable performance metric for any computer system is just *program execution time*, since it incorporates several other measures of goodness, that, when taken independently, do not portray accurately the overall performance; in a sequential computer, these measures are the quality of the instruction set, the capability of the compiler to produce code that keeps the ALU busy and minimizes stalling, the latency and bandwidth of the memory subsystem, the clock rate of the chip, and the rate of execution of floating-point operations. When the object of benchmarking is a parallel system, program execution time can also mirror the communication and synchronization costs, the latency and bandwidth of the interconnection network, and the quality of the routing scheme for inter-processor message passing: hence, we have chosen execution time rather than some abstract complexity measure as our basic unit of measure. Let now T_p stand for the execution time of a program on a system with p exactly alike 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* $\mu_p = s_p / p$. The value of μ_p is usually less than one (i.e. $s_p \quad p$), indicating efficiency loss due to a number of factors such as overhead introduced by communication (message packing and transfer), PE startup (getting identification parameters, setting the communication topology and opening the channels), duplication of operations, and PE suspension (forced idleness) due to load imbalance or message dependencies: in fact, it has to be true in general, since any algorithm running on p processors can be simulated as p serial copies running on a single processor in roughly p-fold time. Therefore, *linear speedup* ($s_p \quad p$) is the best one can hope for, although in the literature (and also in our experiments) *superlinear* speedups ($s_p > p$) have sometimes been observed: this phenomenon is usually due to the effect of *memory hierarchies*, i.e. to the fact that the data of the problem is subdivided between the different PEs, possibly reducing cache misses and/or page faults and the corresponding disk I/O. However, a linear speedup is usually difficult enough to achieve, because of the limit on the amount of parallelism that can be extracted from an algorithm: this is quantitatively described by Amdahl's Law [Am67], which states that *parallel speedup cannot exceed the inverse of the fraction of the residual sequential code*, in formulas

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

where $s \in [0, 1]$ is the fraction of inherently sequential (non-parallelizable) code. Thus, if s 0 and the communications overhead is negligible the speedup will be almost linear, but if, say, 10% of the operations must be performed sequentially (i.e. s = 0.1), then the maximum speedup is 10, even if an infinite number of processors were to be utilized: for PMMCFB, s will represent 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 definitions can be problematic to apply: for problems such as Linear Programming, for instance, there is no uncontested "best" serial algorithm, and although bounds in operations count are available, these are usually worst-case estimates and the actual performance may depend strongly on the particular data set and on the quality of the implementation - however, for the instances that have been used MMCFB can be considered one of the most efficient approaches [FG96], and consequently $s_p^* \quad s_p$. Moreover, generally speaking, the *absolute* speedup does not reflect the *degree of parallelization* of the algorithm, as measured against its serial implementation, while the *relative* speedup and efficiency does, and it seems to be highly indicated as the right performance measure in our case. However, a problem 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 several questions on how to define an appropriate efficiency measure for this case should need to be answered. Although a number of different alternatives can be proposed, we essentially decided not to address the question at all, presenting the data "as is" 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 allow us to sensibly reduce the overall solution time.

Results on the Cray T3D

In the following section, the computational results obtained by the *symmetric* PMMCFB on the Cray T3D are reported: in each table, T_p will denote the running time in seconds, taken by the solution of

the problem (excluding loading and preprocessing time) with p processors, μ_p^I the *ideal efficiency* according to the Amdahl's Law, and μ_p the actual *relative efficiency* achieved; near to the *sequential time* T_1, s will indicate a very close estimate of the fraction (expressed as percentage) of inherently sequential code. In Table 4, the aggregated computational results for all the problem classes are reported: the results are visualized in the corresponding Chart 1, where the obtained relative efficiency is depicted as a function of both p an the problem class, and in Chart 2, Chart 3 and Chart 4 where the ideal and actual efficiencies are compared for p equals to 4, 16 and 64 respectively.

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

Table 4: aggregated computational results for the Cray T3D

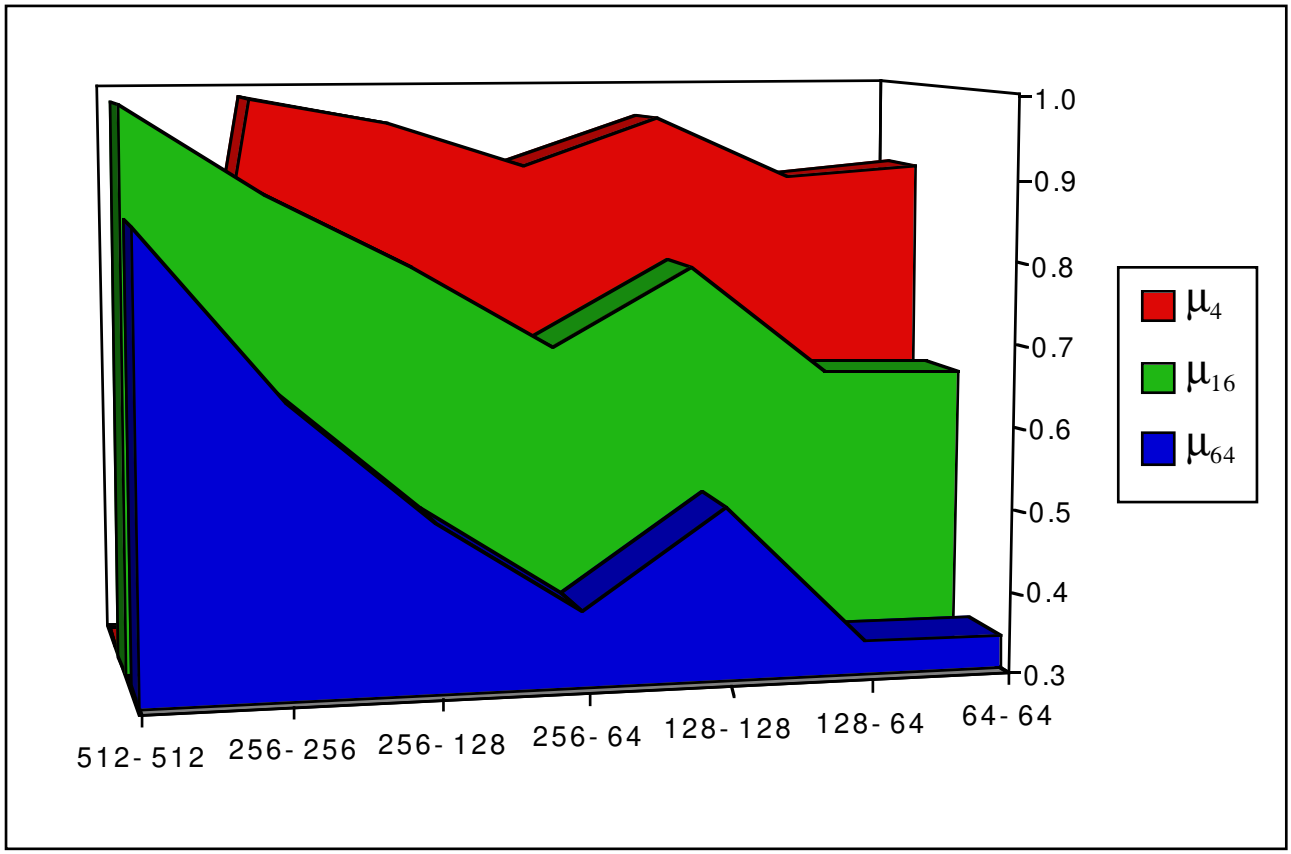

Chart 1: efficiency as a function of p and the problem group from Table 4

These computational results already show that satisfactory efficiencies (always over 63%) are consistently obtained when using at most 16 PEs, that is the typical number of processors for most of the experiences reported in the literature; on the contrary, as p further increases μ_p soon degrades down to 34% on some classes of instances.

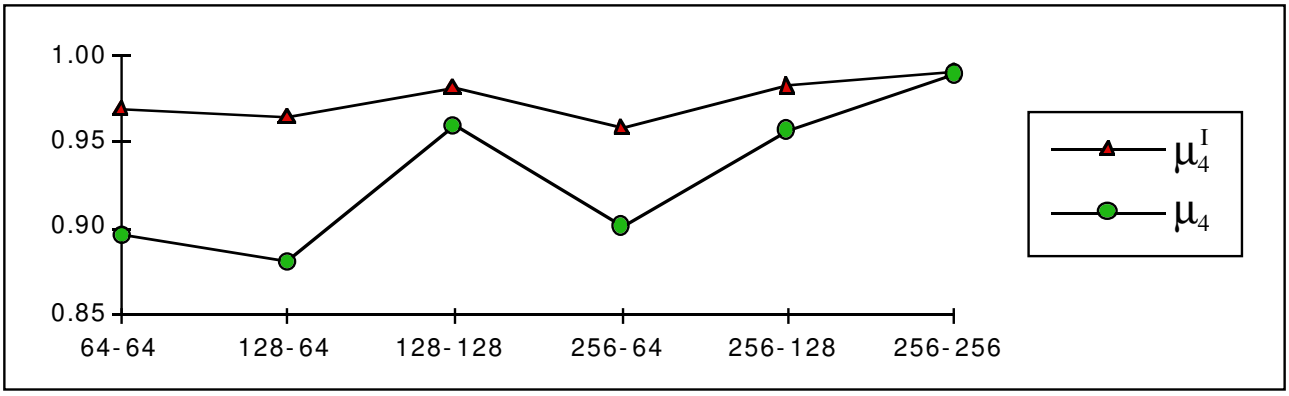

Chart 2: ideal and actual efficiency for $p = 4$

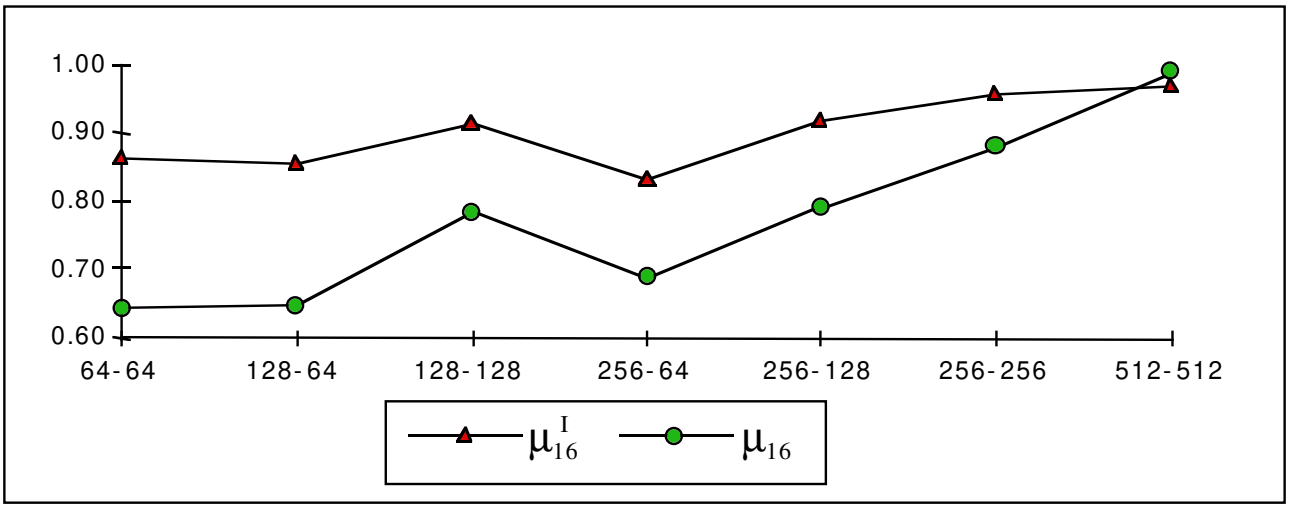

Chart 3: ideal and actual efficiency for $p = 16$

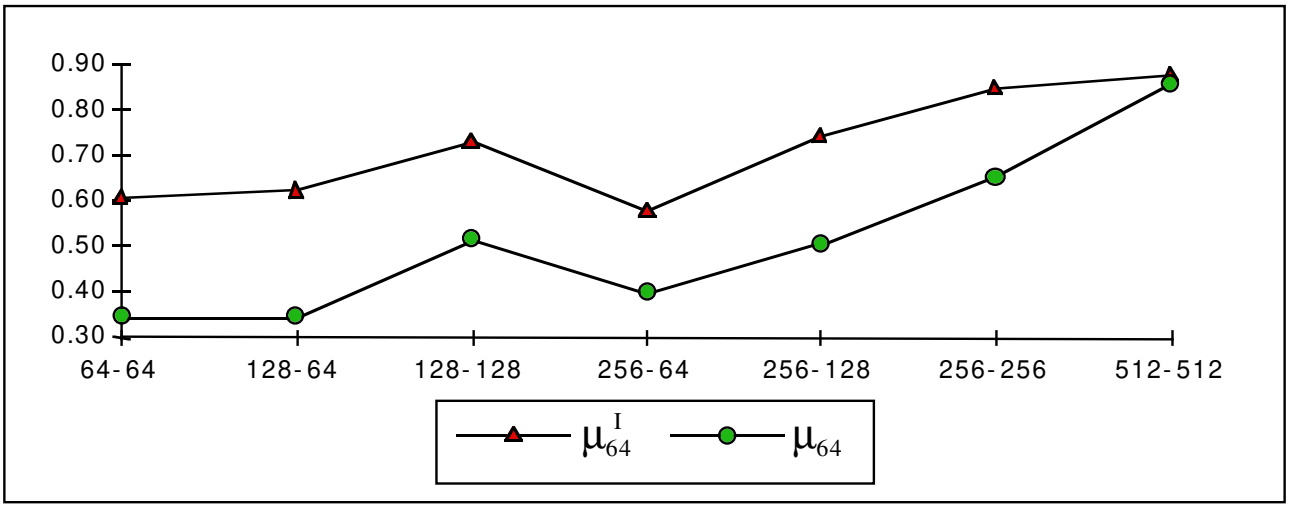

Chart 4: ideal and actual efficiency for $p = 64$

However, Chart 1 clearly shows that μ_p also has clear dependencies on the "size" of each subproblem and the number of commodities, that can be summarized as follows:

– for a fixed graph size, μ_p increases as k does (cfr. the 256-64, 256-128 and 256-256 problems):

this has a clear explanation, since for increasing k the relative weight of the non-parallelizable part of the algorithm (measured by s) decreases

- for a fixed m / k, μ_p increases when m does (cfr. 64-64, 128-128 and 256-256 problems): this means that, although both the communication and the QP solution costs increase with m, this increase is largely compensated by a corresponding decrease of s
- for a fixed k, μ_p seems to be "stable" enough (cfr. 64-64, 128-64 and 256-64 problems), meaning that the QP solution costs at least does not increase too faster than the subproblem solution cost as m increases.

These observations are particularly relevant due to the fact that, as shown in [FG96], 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: hence, PMMCFB seems to be particularly well-suited for large instances with "many" commodities, since for such problems the code is effective in sequential, and furthermore the obtained efficiencies are high even with a large number of processors. Furthermore, the gap between actual and ideal efficiency seems to follow and 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 tends to reduce (up to, eventually, closing) as the size of the problem increase.

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-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: disaggregated computational results for the Cray T3D

A number of other observations can be made by analyzing the results a little more in details: to this aim, Table 5 reports the results for each single problem group - note that it contains violations of the

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. As in some cases the *observed* efficiency is exactly the same as *ideal* efficiency, we claim that communication time is not a significant factor, at least for these problems classes: this is confirmed by the estimates that can be done on the declared performances of the machine, and especially by the fact that the gap between real and ideal behaviour of the algorithm diminishes as m (and therefore the amount of exchanged data per iteration) increases, clearly indicating that the loss of efficiency is due to Master-Slaves synchronization rather than to communication overhead. Remarkably, the largest problems in this suite, the 512-512 with over 800.000 variables, always attained very high efficiencies, as large as 85% even with 64 PEs, and a very little (or no) gap between ideal and actual efficiency: for the B group, the parallel code reduces the "wall-clock time" needed for solution of these very large problems of over a factor of 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 Table 6, this is done for the 64-64 (where the code performs significantly worse than in all the others), where the instances from 1 to 12 are ordered by increasing *loss of efficiency* $\Delta\mu = \mu_{64}^{I} - \mu_{64}$, and the percentage of arcs having single-commodity constraints is shown as *sc*.

	$\Delta\mu$	*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%	80.2	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%	60.8	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%	35.7	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%	19.6	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%	16.7	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%	18.7	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

Table 6: even more disaggregated computational results for the 64-64 problems

For the analysis, the problems can be essentially grouped in 2 suites, the first 6 having $\Delta\mu$ 0.27 and the others having $\Delta\mu$ 0.3; belonging to C and D classes, the first ones are larger and hence more "difficult" to solve, the exception being the swap 4(B)-9(C) that 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 just a glance to the column of sequential times, and by the fact that, on average, D problems perform better than C ones and B problems perform better than A ones. From the simple observation that, using 64 PEs, each PE is assigned exactly one MCF, we can easily conclude that, for the small or easy problems, time spent on busy waiting for synchronization is relatively large, resulting in poor performances: all the results found so far could perhaps be summarized by the fact that the larger and more difficult an instance is, the better its relative efficiency is likely to be, possibly indicating 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.

Results for the asymmetric version (Cray T3D + Cray C90)

As remarked at the begin of this paragraph, it is a fairly difficult task even to define the speedup of the asymmetric PMMCFB: at least two speed-up definitions are possible, as the "serial execution time" could be that of MMCFB on the C90 *or* on a single PE of the T3D - and both these choices disregard the fact that the two architectures behave differently on different parts of the code. Some technical difficulties further complicated the matter: one relevant problem was that, in the available environment, the C90 machine was *not* run in a dedicated mode, so that the Master was repeatedly de-scheduled during the execution of the algorithm, increasing the idle time of the (dedicated) T3D; even worse, the C90-T3D communications using PVM pass through a separate C90 process (the PVM-daemon), so that receiving one message possibly involved waiting for *two* processes being scheduled for execution. Although the latter problem could have been solved by abolishing PVM and directly using the sockets, eliminating the extra idle time appeared to be hopeless: in fact, the wall-clock time of the asymmetric PMMCFB actually *increased* of up to an order of magnitude, and was also fluctuating with the workload of the C90 (that was, however, always very high). The only way to get rid of the problem was to approximate the total execution time by the sum of the CPU time in the C90 and the wall-clock time of the T3D, excluding from both the time spent in the C90-T3D communications: such an estimate is very close to the time that could be obtained if a dedicated C90 would be available, since it only disregards the very low cost of the actual data transfer. Actually, the last statement has to be qualified, since the obtained estimate is close only if the C90 does *not* solve MCFs itself (so that it is immediately available when the T3D finishes), i.e. when the two machines are *never* working at the same time: although this may definitely be a bad choice for a small p, with 16 or 64 processors it is unlikely to diminish the maximum number of MCFs that have to be solved by a PE of the T3D, and hence to improve the performances. However, for $p = 1$ (i.e. the C90 solving the ($\Pi_{\beta t}'$) and coordinating the Bundle algorithm and a single PE of the T3D solving all the MCFs) such an estimate could be taken as another reasonable "serial execution time", giving a third way for defining relative speedup and efficiency: since the T3D is almost always faster than the C90 in solving the MCFs, this can be interpreted as the time of the code ran on an "abstract" sequential machine combining the advantages of both, since all the communication times have been disregarded. Finally, if *efficiencies* had to be compared, the question of how one CPU of C90 has to contribute to the *number* of processors p in the definition should have to be answered: although this further degree of freedom could be avoided by confining ourselves to the use of *speedups* only, all these problems call for the only solution of avoiding any efficiency measure but the plain execution times.

Hence, the results reported in Table 7 are limited to the direct comparison of the times for the symmetric and asymmetric versions of PMMCFB: in both cases, T_p has to be intended as the time obtained with p PEs of the T3D - i.e. excluding the C90 for the asymmetric version from the number of available processors - and the difference Δt_p between the corresponding values, expressed as a percentage of the maximum between the two, is used as the only "goodness" indicator; the reader should remind, however, that due to the above mentioned synchronization problems between the two machines such difference is only indicative of the *potential* improvement obtainable on a system with a dedicated C90.

		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%

Table 7: time comparison for the C90, the symmetric and asymmetric version

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 could have been obtained with some of the larger instances.

Conclusions

The aim of this work was to test whether 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, would lead to an efficient parallel code even by starting with a complex-coordinator code not originally designed for a parallel implementation: the main advantage of such an approach, if carried along in conjunction with suitable software development techniques, is to produce an highly-portable parallel code starting from a sequential one whose effectiveness had already been proved. The computational experience obtained with the Bundle-based cost decomposition algorithm MMCFB seem to show that the resulting parallel code can be highly efficient even with a large number of processors: perhaps, the most interesting finding is that the classes of problems on which it attains its better results are exactly those where the sequential code is known to outperform several other approaches, i.e. “difficult” problems with relatively “many” commodities w.r.t. the size of the graph. Although the results of the basic version being already quite satisfactory, with e.g. the running time of the very large instances taken down from more than 6 hours to about 7 minutes, it has also been possible to further improve on them by exploiting (apparently for the first time) a quite common characteristic of nowadays supercomputing facilities, i.e. the side-to-side availability of vector and massively parallel supercomputers: a proper combination of algorithmic design, software development techniques and parallel architectures seem to have a great potential for the solution of very large-scale Multicommodity Flow Problems.

Acknowledgements

This work has been supported by grants provided by CINECA (Casalecchio di Reno, BO): in the 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. We wish to acknowledge Giovanni Erbacci and Francesca Durì for their valuable help and suggestions during one Author’s stay at CINECA and after.

References

[AK77] A.I. ALI, J.L. KENNINGTON "**MNETGEN program documentation**" *Technical Report IEOR* **77003**, Department of Industrial Engeneering and Operations Research, Southern Methodist University, 1977

[AKL93] A.I. ALI, J.L. KENNINGTON AND T.T. LIANG "**Assignment with En-Route Training of Navy Personnel**" *Nav. Res. Log.* **40**, p. 581-592, 1993

[Am67] G. AMDAHL "**The validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities**" In *Proceedings of the AFIPS Spring Joint Computer Conference, Atlantic City, NJ*, 30, p. 483-485, AFIPS Press, Reston, VA, 1967

[AS95] D. ABRAMSON, A. DE SILVA "**A Parallel Interior Point Method and its Application to Facility Location Problems**" *Research Report* **CIT:95-12**, School of Computing and Information Technology, Griffith University, Nathan, 1995

[Ba93] C. BARNHART "**Dual Ascent Methods for Large-Scale Multicommodity Flow Problems**" *Nav. Res. Log.* **40**(3), p. 305-324, 1993

[BBS94] S. BOOTH, C. BROUGH AND A. SIMPSON "**Notes on Porting PVM Codes to the T3D**", University of Edinburgh EPCC, 1994

[BDG94] A. BEGUELIN, J. DONGARRA, A. GEIST ET AL. "**PVM 3 User's Guide and Reference Manual**" *Oak Ridge National Laboratory*, Oak Ridge, Tennessee, 1994

[Be91] D.P. BERTSEKAS "**Linear Network Optimization: Algorithms and Codes**" *MIT Press,* Cambridge, MA, 1991

[BL84] S.N. BHATT, T. LEIGHTON "**A Framework for Solving VLSI Layout Problems**" *J. Comp. & Syst. Sci.* **28**(2), p. 300-343, 1984

[BS93] C. BARNHART, Y. SHEFFY "**A Network-Based Primal-Dual Heuristic for Multicommodity Network Flow Problems**" *Trans. Sci.* **27**(2), p. 102-117, 1993

[BT89] D.P. BERTSEKAS, J.N. TSITSIKLIS "**Parallel and Distributed Computation: Numerical Methods**" *Prentice-Hall Inc.*, 1989

[Ca96] P. CAPPANERA "**Parallelizzazione di un Algoritmo di Decomposizione per Problemi di Flusso di Costo Minimo di Tipo Multicommodity**" *Master Degree Thesis*, Department of Computer Science, University of Pisa, 1996

[CCZ91] Y. CENSOR, E.D. CHAJAKIS AND S.A. ZENIOS "**A Primal-Dual Algorithm for Large-Scale Multicommodity Optimization Problems and its Parallel Implementation**" *TR **91-09-03***, The Wharton School, Dec. Sci. Dep., 1991

[CFG95] P. CAPPANERA, A. FRANGIONI AND G. GALLO "**Coarse-Grained Parallelization of Dual Approaches to Multicommodity Min Cost Flow Problems**" *Science and SuperComputing at CINECA*, 1995

[CFN96] P. CARRARESI, A. FRANGIONI AND M. NONATO "**Applying Bundle Methods to Optimization of Polyhedral Functions: An Applications-Oriented Development**" *Ricerca Operativa* Vol. **25** n. 74, p. 5-49, 1996

[Cr94a] CRAY RESEARCH INC. "**PVM and HeNCE Programmer's Manual**", 1994

[Cr94b] CRAY RESEARCH INC. "**SHMEM Technical Note for C**", 1994

[DPZ91] R.S. DEMBO, M.C PINAR AND S.A. ZENIOS "**A Smooth Penalty Function Algorithm for Network-Structured Problems**" *Technical Report* **TR 90-12-05**, Decision Sciences Department, The Wharton School, University of Pennsylvania, 1991

[DR56] J. DOUGLAS and H.H. RACHFORD Jr. "**On the Numerical Solution of Heat Conduction Problems in Two and Three-space Variables**" *Transactions of the American Mathematical Society* **82**, p. 421-439, 1956

[Fe91] M.C. FERRIS "**Parallel Constraint Distribution for Convex Quadratic Programs**" *Technical Report* **TR 1009**, Computer Science Department, University of Wisconsin-Madison, 1991 (to appear on Mathematics of O.R.)

[FG96] A. FRANGIONI, G. GALLO "**A Bundle type Dual-ascent Approach to Linear MultiCommodity Min Cost Flow Problems**" *Technical Report* **TR 01/96**, Dipartimento di Informatica, Università di

Pisa, 1996 (submitted for publication to *ORSA J. on Computing*)

[FH94] M.C. FERRIS, J.D. HORN "**Partitioning Mathematical Problems for Parallel Solution**", *Techical Report* **TR 1232**, Computer Science Department, University of Wisconsin-Madison, 1994

[FM91] M.C. FERRIS, O.L. MANGASARIAN "**Parallel Constraint Distribution**" *SIAM J. on Optimixation* **1**(4) , p. 487-500, 1992

[FM94] M.C. FERRIS, O.L. MANGASARIAN "**Parallel Variable Distribution**" *SIAM J. on Optimization* **4**(4), p. 815-832, 1994

[Fr96] A. FRANGIONI "**Solving Semidefinite Quadratic Problems Within Nonsmooth Optimization Algorithms**" *Computers & O.R.* Vol. **23**(11), p. 1099-1118, 1996

[GH93] S.K. GNANENDRAN , J.K. HO "**Load Balancing in the Parallel Optimization of Block-Angular Linear Programs**" *Math. Prog.* **62**, p. 41-67, 1993

[GM75] R. GLOWINSKI, A. MARROCCO "**Sur l'Approximation par l ments Finis d'Ordre Un, et la R solution par P nalisation-Dualit d'une Classe de Probl mes de Dirichlet Nonlin aires**" *Revue Fran aise d'Automatique, Informatique et Recherche Op rationelle* **2**, p, 41-76, 1975

[GW89] M.Y. WU, D.D. GAJSKI "**Computer-Aided Programming for Message-Passing Systems: Problems and a Solution**" In *Proceedings of the IEEE, special issue on supercomputer technology*, p. 1983-1991, IEEE Press, 1989

[HLS88] J.K. HO, T.C. LEE AND R.P. SUNDARRAJ "**Decomposition of Linear Programs Using Parallel Computation**" *Math. Prog.* **42**, p. 391-405, 1988

[HP90] J.L. HENNESSY, D.A PATTERSON "**Computer Architecture: a Quantitative Approach**" *Morgan Kaufmann*, 1990

[KAR90] P. KLEIN, A. AGRAWAL, R. RAVI AND S. RAO "**Approximation Through Multicommodity Flow**" *Proc. 31th FOCS*, p. 726-727, 1990

[Ko94] S.A. KONTOGIORGIS "**Alternating Directions Methods for the Parallel Solution of Large-Scale Block-Structured Optimization Problems**" *Technical Report* **MP-TR-94-13** (PhD thesis), University of Wisconsin-Madison, 1994

[LGM93] R. DE LEONE, M. GAUDIOSO AND M.F. MONACO "**Nonsmooth Optimization Methods for Parallel Decomposition of Multicommodity Flow problems**" *Annals of O.R.* **44**, p. 299-311, 1993

[LKM94] R. DE LEONE, S. KONTOGIORGIS, R.R. MEYER, A. ZAKARIAN AND G. ZAKERI "**Coordination in Coarse-Grained Decomposition**" *SIAM J. on Optimization* **4**(4), p. 777-793, 1994

[Me90] D. MEDHI "**Parallel Bundle-Based Decomposition for Large-Scale Structured Mathematical Programming Problems**" *Annals of O.R.* **22**, p. 101-127, 1990

[MS91] R.R. MEYER, G.L. SCHULTZ "**An Interior Point Method for Block-Angular Optimization**" *SIAM J. on Optimization* **1**(4), p. 121-152, 1992

[Pa89] P.C. PATTON "**Performance Limits for Parallel Processors**" In G.F. Carey editor, *Parallel Supercomputing: Methods, Algorithms and Applications*, p. 1-16 John Wiley & Sons, 1989

[PR55] D.W. PEACEMAN, H.H. RACHFORD Jr. "**The Numerical Solution of Parabolic and Elliptic Differential Equations**" *SIAM Journal* **3**, p. 28-42, 1955

[PZ92] M.C. PINAR, S.A. ZENIOS "**Parallel Decomposition of Multicommodity Network Flows Using a Linear-Quadratic Penalty Algorithm**" *ORSA J. on Computing* **4**(3), p. 235-249, 1992

[Sc91] G.L. SCHULTZ "**Barrier Decomposition for the Parallel Optimization of Block-Angular Programs**" *PhD Thesis*, University of Wisconsin, Madison, 1991

[ZaA95] A.A. ZAKARIAN "**Nonlinear Jacobi and ε-Relaxation Methods for Parallel Network Optimization**" *Technical Report* **MP-TR-95-17**, (PhD thesis), University of Wisconsin-Madison, 1995

[ZaG95] G. ZAKERI "**Multi-Coordination Methods for Parallel Solution of Block-Angular Programs**" *Technical Report* **MP-TR-95-08** (PhD thesis), University of Wisconsin-Madison, 1995

[Ze91] S.A. ZENIOS "**On the Fine-Grain Decomposition of Multicommodity Transportation Problems**" *SIAM J. on Optimization* **1**(4), p. 643-669, 1991

[Ze93] S.A. ZENIOS "**Data-Parallel Computing for Network-structured Optimization Problems**"

Comput. Optim. and Appl. **3**, p. 199-242, 1993

[ZM88] S.A. ZENIOS, J.M. MULVEY "**Vectorization and Multitasking of Nonlinear Network Programming Algorithms**" *Math. Prog.* **42**, p. 449-470, 1988