Scientific Computing on Bulk Synchronous Parallel Architectures

R. H. Bisseling*

Department of Mathematics, Utrecht University

and

W. F. McColl†

Programming Research Group, Oxford University

April 27, 1994

*Author's address: Department of Mathematics, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, The Netherlands (bisseling@math.ruu.nl). This work was initiated while this author was a Research Mathematician at Koninklijke/Shell-Laboratorium, Amsterdam.

†Author's address: Programming Research Group, Oxford University, Wolfson Building, Parks Road, Oxford OX1 3QD, UK (mccoll@prg.oxford.ac.uk). Part of this work was done while this author was a Visiting Scientist at Koninklijke/Shell-Laboratorium, Amsterdam.

Abstract

Bulk synchronous parallel architectures offer the prospect of achieving both scalable parallel performance and architecture independent parallel software. They provide a robust model on which to base the future development of general purpose parallel computing systems. In this paper we theoretically and experimentally analyse the efficiency with which a wide range of important scientific computations can be performed on bulk synchronous architectures. The computations considered include the iterative solution of sparse linear systems, molecular dynamics, linear programming, and the solution of partial differential equations on a multidimensional discrete grid. These computations are analysed in a uniform manner by formulating their basic procedures as sparse matrix-vector multiplications.

1 Introduction

Bulk synchronous parallel (BSP) architectures [30] offer the prospect of achieving both scalable parallel performance and architecture independent parallel software. They provide a robust model on which to base the future development of general purpose parallel computing systems. In this paper, we theoretically and experimentally analyse the efficiency with which a wide range of important scientific computations can be performed on BSP architectures. The computations considered include the iterative solution of sparse linear systems, molecular dynamics, linear programming, and the solution of partial differential equations on a discrete grid. We analyse these computations in a uniform manner by formulating their basic procedures as a sparse matrix-vector multiplication. In our analysis, we introduce the *normalised BSP cost* of an algorithm as an expression of the form $a + bg + cl$, where a, b, and c are scalar values which depend on the algorithm, on the number of processors, and on the chosen data distribution. The scalars g and l are parameters that characterise the hardware: $g \geq 1$ is the communication throughput ratio and $l \geq 1$ is the network periodicity. An ideal parallel algorithm has the values $a = 1$, $b = 0$, and $c = 0$; an algorithm with load imbalance has a value $a > 1$; an algorithm with communication overhead has a value $b > 0$; and an algorithm with synchronisation overhead has a value $c > 0$.

As an example, consider the execution of a five-point Laplacian finite difference operator on a two-dimensional toroidal grid. This operator computes new values at a grid point using the old values at the grid point and its direct neighbours to the north, east, south, and west. Our BSP algorithm for this computation has a normalised cost on 100 processors of $1.0 + 0.022g + 0.00056l$ for a grid of size 200×200. This low cost is achieved by distributing the grid by orthogonal domain partitioning over the processors, assigning a square block of 20×20 grid points to each processor. The resulting cost value implies that this computation can be performed efficiently on BSP computers with $g \leq b^{-1} \approx 45$ and $l \leq c^{-1} \approx 1800$.

In the design of efficient BSP algorithms, it is important to find a good data distribution. In fact, the choice of a data distribution is one of the main means of influencing the performance of the algorithm. In the BSP model, the partitioning of the data is a crucial issue, as opposed to the mapping of the resulting partitions to particular processors, which is irrelevant. This leads to an emphasis on problem dependent techniques of data partitioning, instead of on hardware dependent techniques that take network topologies into account. The algorithm designer who is liberated from such hardware considerations may concentrate on exploiting the essential features of the problem. In our case, this leads, surprisingly, to the application of sphere packing techniques to reduce communication in molecular dynamics simulations and to the application of tiling techniques to reduce communication in discrete grid calculations.

We present experimental results for the multiplication $u := Av$ of a sparse matrix A and a vector v. The experiments are performed on the sparse matrix test library MLIB, which we developed with the aim of capturing the essence of a range of important scientific computations in the uniform format of a sparse matrix. The library contains matrices with a regular structure, such as the adjacency matrix of a multidimensional toroidal grid, and also matrices with an irregular structure, such as random sparse matrices. Furthermore, the library contains matrices with an underlying, but hidden structure (given as supplementary information), such as the matrices that describe the short-range interaction between particles in a molecular dynamics simulation.

Our BSP algorithm for sparse matrix-vector multiplication imposes the constraint that the vectors u and v and the diagonal of A are distributed in the same way and that the matrix A is distributed in a so-called *Cartesian* manner. This means that the p processors are numbered by two-dimensional Cartesian coordinates (s, t), and that each matrix row is assigned to a set of processors with the same first coordinate s, and each matrix column to a set of processors with the same second coordinate t. This distribution leads to a simple sparse matrix-vector multiplication algorithm. Within this scheme, various choices are possible. For general sparse matrices, with no known structure, a good choice is to distribute the matrix diagonal randomly over the processors, taking care that each processor receives an equal number of diagonal elements, and using a *square* Cartesian processor numbering, i.e. with $0 \leq s, t < \sqrt{p}$. For matrices with a known structure, this method can be greatly improved upon by using techniques such as spatial decomposition of the corresponding physical domain. We present several new techniques based on spatial decomposition and demonstrate their practical utility by numerical experiments.

2 The BSP model

For a detailed account of the BSP model, and of the various routing and hashing results which can be obtained for it, the reader is referred to [30] (see also [31]). We concentrate here on presenting a view of how a bulk synchronous parallel architecture would be described, and how it would be used. A *bulk synchronous parallel (BSP) computer* consists of: a set of processor-memory pairs; a communications network that delivers messages in a point-to-point manner; and a mechanism for the efficient barrier synchronisation of all, or a subset, of the processors. There are no specialised broadcasting or combining facilities. If we define a time step to be the time required for a single local operation, i.e. a basic operation such as addition or multiplication on locally held data values, then the performance of any BSP computer can be characterised by the following four parameters:

p = number of processors
s = processor speed, i.e. number of time steps per second
l = synchronisation periodicity, i.e. minimal number of time steps between successive synchronisation operations
g = (total number of local operations performed by all processors in one second) / (total number of words delivered by the communications network in one second)

The parameter l is related to the network latency, i.e. to the time required for a non-local memory access in a situation of continuous message traffic. The parameter g corresponds to the frequency with which non-local memory accesses can be made; in a machine with a higher value of g one must make non-local memory accesses less frequently. Let the term "realising an h-*relation*" denote the general packet routing problem where each processor has at most h packets to send to various processors in the network, and where each processor is also due to receive at most h packets from other processors. Here, a packet is one word of information, such as a real number or an integer. The g parameter of a BSP computer is based on the time required to realise h-relations in a situation of continuous message traffic; g is the value such that an h-relation can be performed in gh time steps.

A BSP computer operates in the following way. A computation consists of a sequence of parallel *supersteps*, where each superstep is a sequence of steps, followed by a barrier synchronisation at which point any memory accesses take effect. During a superstep, each processor has to carry out a set of programs or threads, and it can do the following: (i) perform a number of computation steps, from its set of threads, on values held locally at the start of the superstep; (ii) send and receive a number of messages corresponding to non-local read and write requests. For simplicity, we assume in this paper that a superstep either performs steps of (i), or steps of (ii), but not of both.

The BSP computer is a two-level memory model [24], i.e. each processor has its own physically local memory module; all other memory is non-local, and is accessible in a uniformly efficient way. By uniformly efficient, we mean that the time taken for a processor to read from, or write to, a non-local memory element in another processor-memory pair should be independent of which physical memory module the value is held in. The algorithm designer and the programmer should not be aware of any hierarchical memory organisation based on network locality in the particular physical interconnect structure currently used in the communications network, as in special purpose parallel computing [23]. Instead, performance of the communications network should be described only in terms of its global properties, e.g. the maximum time required to perform a non-local memory operation, and the maximum number of such operations which can simultaneously exist in the network at any time.

The complexity of a superstep S in a BSP algorithm is determined as follows. Let the work w be the maximum number of local computation steps executed by any processor during S. Let h_{s} be the maximum number of messages sent by any processor during S, and h_{r} be the maximum number of messages received by any processor during S. In the original BSP model, the cost of S is $\max\{l, w, gh_{\mathrm{s}}, gh_{\mathrm{r}}\}$ time steps. (An alternative [14] is to charge $\max\{l, w + gh_{\mathrm{s}}, w + gh_{\mathrm{r}}\}$ time steps for superstep S.) In this paper, we will charge $l + w + g \cdot \max\{h_{\mathrm{s}}, h_{\mathrm{r}}\}$ time steps for S. The cost of a BSP algorithm is simply the sum of the costs of its supersteps. Different cost definitions reflect different assumptions about the implementation of supersteps, and in particular about which operations can be done in parallel and which ones must be done in sequence. The difference is not crucial; for instance, our cost is between one and three times the cost in the original model. Our choice for charging the cost of a superstep is motivated by its convenience in obtaining a simple numeric expression for the cost of an algorithm for a particular problem on a BSP computer with unknown characteristic parameters l and g. Using our definition, one obtains a simple expression of the form $a + bg + cl$ for the cost of an algorithm, where $a, b,$ and c are numeric constants. Note that in this case the original definition does not necessarily lead to a simple expression.

In designing algorithms for a BSP computer with a high g value, we need to achieve a measure of *communication slackness* by exploiting thread locality in the two-level memory, i.e. we must ensure that for every non-local memory access we request, we are able to perform approximately g operations on local data. To achieve architecture independence in the BSP model, it is therefore appropriate to design parallel algorithms and programs which are parameterised not only by n, the size of the problem, and p, the number of processors, but also by l and g. This can indeed be done, because the network performance of a BSP computer is captured in global terms using the values l and g. (A language that supports this style of programming is GL [25].) The resulting algorithms can therefore be efficiently implemented on a range of BSP architectures with widely differing l and g values.

A systematic study of direct bulk synchronous algorithms remains to be done. Some first steps in this direction are described in [14, 30]. This paper significantly extends that work by theoretically and experimentally analysing the efficiency with which a wide range of important scientific computations can be performed on bulk synchronous architectures.

3 Linear algebra in scientific computing

Linear algebra is of crucial importance to scientific computing. The main reason for this is the large amount of computing time consumed by linear algebra computations in a wide range of application areas. Often, applications require the solution of large linear systems or large eigensystems. This has lead to the extensive use of linear algebra libraries such as LINPACK, EISPACK, and their common successor LAPACK [2]. To achieve portability, many scientific computer programs rely on using the common Basic Linear Algebra Subprograms (BLAS) for their vector, matrix-vector, and matrix-matrix operations. Today, efficient BLAS implementations are available for most computer architectures. Another reason for the importance of linear algebra is that the language of linear algebra provides a powerful formalism for expressing scientific computations, including many computations that are not commonly thought of as linear algebra computations. A prime example of the benefit of this approach is the use of matrix-vector notation to formulate Fast Fourier Transform algorithms [32].

One important application of linear algebra occurs in the solution of partial differential equations (PDEs) by finite difference, finite element, or finite volume methods. These require the repeated solution of systems of linear equations $Ax = b$, where A is an $n \times n$ nonsingular matrix, and x and b are vectors of length n. Usually, the matrix

A is *sparse*, i.e., only $\mathcal{O}(n)$ of its n^2 elements are nonzero. The system can be solved by a direct algorithm, using Cholesky factorisation in the case of a symmetric positive definite matrix A, or LU decomposition in the general case, see [15]. An alternative approach is to use an iterative algorithm, based on successive improvements of approximate solution vectors $x^{(k)}$. Two important iterative algorithms are the conjugate gradient algorithm [19] for symmetric positive definite matrices A and the generalised minimal residual algorithm [29] for general matrices. Iterative methods use the matrix A mainly in a multiplicative manner, by computing matrix-vector products of the form $u := Av$. Iterative methods are increasingly becoming popular, because they enable the solution of very large linear systems such as those originating in PDE solving on large three-dimensional grids. Discretising a PDE on a grid of $100 \times 100 \times 100$ points with one variable per grid point already leads to linear systems of one million equations in one million variables. Such systems may arise for instance in the simulation of oil reservoirs, in the modelling of semiconductor devices, and in aerodynamics computations. Iterative methods may often solve these systems within reasonable time and with acceptable memory use, because the systems are sparse and remain so during their solution, whereas direct methods will break down, because they create too many new nonzero elements in the matrix.

Another application of linear algebra in scientific computing is in the solution of linear programming (LP) problems, such as the problem of minimising the cost $c^T x$ under the constraint $Ax \leq b$ (to be interpreted component-wise), where A is an $m \times n$ matrix, c and x are vectors of length n, and b is a vector of length m. This optimisation problem can be solved by the simplex method [8], which involves a sequence of rank-one updates of the form $A := A + uv^T$, or by an interior-point method [21], which involves multiplying the matrix A by its transpose and solving a symmetric positive definite linear system of the form $AA^T u = v$. This system is usually solved by Cholesky factorisation (see [4] for a parallel implementation), although currently much research is being done on the applicability of iterative methods.

Linear algebra is also important in the field of molecular quantum chemistry, where various properties of molecules are determined from first principles by solving the time-independent Schrödinger equation, for instance by the direct SCF method [1]. Although the dominant part of this computation is the calculation of 2-electron integrals and their incorporation into a Hamiltonian matrix, other important parts are the computation of the eigenvalues and eigenvectors of this matrix, and the multiplication of matrices. Since the latter parts are more difficult to parallelise than the trivially parallel integral calculations, they may well dominate the computing time on a parallel computer [16].

The examples above suggest that a first approach to achieving general purpose parallel computing for scientific applications may be based on developing BSP algorithms for linear algebra computations and implementing these algorithms on a computer which resembles the BSP model as much as possible. For scientific applications that are not based on linear algebra, we may still be able to capture the essence of the computation in linear algebra language, so that we can use BSP techniques developed for linear algebra to gain further insight into these applications as well.

This paper focuses on one particular linear algebra operation, the sparse matrix-vector multiplication, for the following reasons:

1. The sparse matrix-vector multiplication is the basis of iterative methods for the solution of sparse linear systems $Ax = b$. At every iteration, the matrix A (and in certain cases its transpose) is multiplied by a vector, and the resulting vector is used to update the best current approximate solution. Similarly, this multiplication is also the basis for the Lanczos method [22], an iterative method which can be used to find the extremal eigenvalues of a sparse symmetric matrix (see e.g. [15]).

2. The sparse matrix-vector multiplication represents the execution of the finite difference operator in certain PDE solvers. This even holds in the common case of *matrix-free* solvers which do not form the finite difference matrix explicitly, but instead apply the finite difference operator directly on the current approximate solution vector. An example is the five-point Laplacian finite difference operator used to solve a second-order elliptic PDE on a two-dimensional grid of size $r \times r$. This operator can be formulated in matrix terms (see e.g. [28]) by defining an $n \times n$ matrix A, with $n = r^2$, by

$$a_{ij} = \begin{cases} -4 & \text{if } i = j \\ 1 & \text{if } i = j \pm 1, j \pm r \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

The solution value of the PDE at a grid point $(k, l), 0 \le k, l < r$ corresponds to a component $x_i, 0 \le i < n$, of the solution vector x of a linear system $Ax = b$, by the relation $i = kr + l$. In a matrix-free PDE solver based on an iterative linear system solver, the equivalent of the sparse matrix-vector multiplication $u := Av$ will simply be executed by summing the values of v in the neighbouring grid points $(k + 1, l)$, $(k - 1, l)$, $(k, l + 1)$, and $(k, l - 1)$, and subtracting from the result four times the value of v in the grid point (k, l), to produce the value of u in that grid point.

3. The sparse matrix-vector multiplication may be used to model two-particle interactions in molecular dynamics simulations. As an example, consider an orthogonal three-dimensional molecular dynamics universe of size $1 \times 1 \times 1$ with periodic boundaries. The universe is filled with n particles, numbered $0 \le i < n$. Each particle moves under the influence of the forces caused by the other particles. Each force is determined by a potential, such as the Lennard-Jones potential for nonbonded particles. Let F_{ij} denote the force on particle i due to particle j, so that the total force on particle i is $F_i = \sum_{j=0}^{n-1} F_{ij}$. Note that $F_{ii} = 0$. The force F_{ij} is a function of the position $\mathbf{r}_i$ of particle i and the position $\mathbf{r}_j$ of particle j, $F_{ij} = F(\mathbf{r}_i, \mathbf{r}_j)$. Therefore, to compute the force on a particle i, one needs, besides the position of the particle i itself, the positions of all the other particles with which it interacts. The need for information about particle positions can be expressed in an $n \times n$ matrix A, defined by

$$a_{ij} = \begin{cases} 1 & \text{if } i = j \text{ or particles } i \text{ and } j \text{ interact} \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

An analogy to the force computation from the positions of the particles is the sparse matrix-vector multiplication $u := Av$, where u is a vector that models the force components and v is a vector that models the particle positions. For short-range potentials, there exists a cut-off radius $r_c > 0$, such that $a_{ij} = 0$ if the distance between particle i and particle j is larger than or equal to r_c. For $r_c \ll 1$ this leads to A being very sparse. The movement of the particles will cause the sparsity pattern of the matrix to change during the course of the simulation. All efficient simulation methods exploit the sparsity to limit the total number of force computations. Furthermore, distributed memory parallel algorithms based on geometric parallelism also exploit the sparsity to reduce the number of communications of current particle positions [12].

Simplifying molecular dynamics simulations by modelling their essence in matrix terms may give remarkable new insights, and may even lead to new ways of performing these simulations. A recent example of this approach is the work of Hendrickson and Plimpton [18] on parallel many-body simulations (such as molecular dynamics). They achieve a reduction in communication volume by an order of $\sqrt{p}$, compared to all-to-all communication, by using techniques from dense linear algebra and carefully translating them to the many-body context. The main idea in their method is to cluster the force computations in a particular way, and to replace the all-to-all communication of particle positions by partial broadcasts of these positions and partial combines of accumulated forces. No sparsity is used to reduce the communication. (It is used, of course, to reduce the total number of force computations.) Because of this, the method is most suited for long-range or medium-range potentials, with a break-even point that is much more favourable than that of conventional all-to-all methods.

4 The MLIB test set of sparse matrices

Our motivation for developing a new library of sparse matrices, MLIB, came from the desire to mimic various areas of scientific computing in one common format and to use this format to investigate parallel scientific computing. The essential properties of problems in a wide range of application areas can often be captured in one sparse matrix or one family of matrices with the same structure. An example is the solution of a PDE on a regular two-dimensional $r \times r$ grid using the five-point Laplacian finite-difference operator, see Section 3. Taking the grid points as vertices and their neighbour relations as directed edges, while assuming periodic boundary conditions, we obtain a directed r-ary, two-dimensional hypercube graph. In general, PDE-solvers on regular grids give rise to hypercube graphs with a large radix and a low dimension. The adjacency matrix A of such a graph is sparse and its size grows rapidly with increasing dimension or increasing radix. On a distributed-memory parallel computer it would be efficient to distribute the matrix and the related vectors by using the knowledge of the underlying neighbour structure of the graph. This may eliminate unnecessary communication of grid variables.

At present, there exists a library of sparse matrices, the Harwell-Boeing (HB) library [10, 11], which is widely used to test sparse matrix algorithms. It contains many examples of matrices that occur in practical applications. We have included a small subset of eleven matrices of the HB library in MLIB, mainly to facilitate our experiments on such practical matrices. For our specific purpose of mimicking scientific computation, the HB matrices are not well suited, and therefore we decided to design our own library. We do not claim in any way that the matrix library MLIB is complete or representative. We present it as a first attempt to capture some features of scientific computing in the common format of sparse matrices.

The matrix library MLIB consists of 34 sparse matrices and their generating programs. Each matrix is represented by a file which contains the nonzero elements of the matrix stored by the coordinate scheme (see [9]). The element $a_{ij} \neq 0$ is stored as a triple (i, j, x) where i is the row index, j the column index, and $x = a_{ij}$ the numerical value. The numerical values of MLIB are dummies, except in the case of the HB subset, which retains the original numerical values. At this stage, our interest is in sparsity patterns and their implications for parallel computing, and not in numerical issues. Nevertheless, we decided to include numerical values in the format, to enable possible future use of such values. The format of a matrix file is: first, a line containing the matrix size $m \times n$; then the nonzeros, one per line; after that, a terminator line "-1"; and, optionally, additional information on the matrix, such as particle positions in the case of molecular dynamics matrices. The MLIB library is available upon request from the authors. More details can be found in the documentation of the generating programs.

The MLIB library contains the following classes of matrices:

- `hyp`.$r.d.D$, the hypercube matrix with radix r, dimension d, and distance D, $r, d, D \geq 1$. For $D = 1$, this is the adjacency matrix of the directed r-ary, d-dimensional hypercube graph. The vertices of this graph form a d-dimensional grid of $n = r^d$ points; they are numbered lexicographically. Each vertex has directed edges to itself and to its immediate neighbours in each direction. The size of the hypercube matrix is $n \times n$. For $D > 1$, the hypercube graph is obtained by connecting each vertex to those vertices that can by reached by a path of length $\leq D$ in the original $D = 1$ graph. This models certain higher-order finite difference operators.

- `dense`.n, the dense matrix of size $n \times n$. All elements of this matrix are nonzero.

- `random`.$n.\rho^{-1}$, an $n \times n$ matrix with a random sparsity structure and a nonzero density ρ. This matrix is generated by using the pseudo-random number generator `ran2` from [28]. (All random numbers used in this paper were generated by this generator.)

- `hb.x`, the matrix `x` from the HB collection [10]. For a description of the matrix, see [11]. The subset of the HB collection that is included in MLIB consists of eleven matrices from various application fields. It is the same subset as the one used in [3] to test a parallel iterative linear system solver.

- `md`.n.$r_c{}^{-1}$, an $n \times n$ matrix which corresponds to n particles in a three-dimensional molecular dynamics simulation with short-range potentials, see Section 3. The particles i and j interact, i.e. $a_{ij} \neq 0$, if $||\mathbf{r}_i - \mathbf{r}_j|| \leq r_{\mathrm{c}}$, where $\mathbf{r}_i$ is the position of particle i and r_c the cut-off radius. The positions $\mathbf{r}_i = (x_i, y_i, z_i)$, with $0 \leq x_i, y_i, z_i \leq 1$, are given at the end of the file. The interactions assume periodic boundaries.

- `mdr`.n.$r_c{}^{-1}$.ρ^{-1}, an $n \times n$ matrix which corresponds to n particles in a three-dimensional molecular dynamics simulation with short-range potentials and, additionally, an artificial long-range potential for certain randomly selected particle pairs. The sparsity pattern of this matrix is the union of the sparsity patterns of a short-range molecular dynamics matrix with cut-off radius r_c and a random sparse matrix with density ρ. Here, long-range interactions between selected particles represent interactions between distant clusters of particles. The aim of this procedure is to mimic e.g. multipole expansions.

- `lp`.n, an $n \times n$ matrix which resembles certain symmetric matrices that occur in the solution of LP problems by interior point methods [21] (for a parallel implementation, see [4]). The matrix is constructed by placing dense square submatrices of random size at random places in the matrix, with bias towards small sizes. This captures a structural feature that we observed in certain LP problems. We would like to add a disclaimer about this particular matrix class: one may argue about whether this represents the typical structure of LP matrices. Therefore, we present this type of matrix as just a first and modest attempt to capture some of the common characteristics of LP matrices.

Table 1 presents the size and the number of nonzeros of the 34 matrices from MLIB.

5 Sparse matrix-vector multiplication

In this section, we present a parallel algorithm for the multiplication of a sparse matrix A and a dense vector $\mathbf{v}$,

$$\mathbf{u} := A\mathbf{v}, \tag{3}$$

which produces a dense vector $\mathbf{u}$. The matrix $A = (a_{ij},\ 0 \leq i, j < n)$ has size $n \times n$ and the vectors $\mathbf{u} = (u_i,\ 0 \leq i < n)$ and $\mathbf{v} = (v_i,\ 0 \leq i < n)$ have length n. We assume that the matrix is distributed by a *Cartesian distribution* [6]. This means that the processors are numbered by two-dimensional identifiers (s, t), with $0 \leq s < q_0$ and $0 \leq t < q_1$, where $p = q_0 q_1$ is the number of processors, and that there are mappings $\phi_0 : \{0, 1, \ldots, n-1\} \rightarrow \{0, 1, \ldots, q_0 - 1\}$ and $\phi_1 : \{0, 1, \ldots, n-1\} \rightarrow \{0, 1, \ldots, q_1 - 1\}$ such that matrix elements are distributed according to

$$a_{ij} \longmapsto \text{processor}(\phi_0(i), \phi_1(j)). \tag{4}$$

Matrix A	Order n	Nonzeros $nz(A)$
hyp.2.10.1	1024	11264
hyp.2.10.2	1024	57344
hyp.2.10.3	1024	180224
hyp.3.10.1	59049	1240029
hyp.3.8.1	6561	111537
hyp.20.4.1	160000	1440000
hyp.30.3.1	27000	189000
hyp.50.3.1	125000	875000
hyp.50.2.1	2500	12500
hyp.100.2.1	10000	50000
hyp.200.2.1	40000	200000
dense.100	100	10000
dense.500	500	250000
random.1000.1000	1000	1002
random.1000.100	1000	10013
random.1000.10	1000	100000
hb.impcolb	59	312
hb.west0067	67	294
hb.fs5411	541	4285
hb.steam2	600	13760
hb.shl400	663	1712
hb.bp1600	822	4841
hb.jpwh991	991	6027
hb.sherman1	1000	3750
hb.sherman2	1080	23094
hb.lns3937	3937	25407
hb.gemat11	4929	33185
lp.1000	1000	66512
lp.6000	6000	321256
md.6000.20	6000	25054
md.6000.10	6000	155592
md.6000.8	6000	300928
mdr.6000.10.2000	6000	175176
mdr.6000.8.1000	6000	337380

Table 1: Matrix library MLIB

Note that the elements of a matrix row are mapped to processors with the same first identifier coordinate and that the elements of a matrix column are mapped to processors with the same second coordinate. This reflects the row-wise and column-wise nature of many linear algebra algorithms and this often leads to reduced communication requirements in linear algebra computations on distributed-memory parallel computers. This two-dimensional numbering of processors originates in special purpose algorithms for mesh networks of processors [3, 6]. In the present work, however, the two-dimensional numbering reflects a property of the problem to be solved and not of any particular network topology: the BSP-model is topology-independent. We assume that vectors are distributed in the same manner as the diagonal of the matrix, i.e. according to

$$u_i \longmapsto \text{processor}(\phi_0(i), \phi_1(i)). \tag{5}$$

This particular distribution scheme is flexible enough to accommodate many commonly used distribution methods while it is also sufficiently restrictive to impose efficient communication patterns. The flexibility is illustrated by the following two examples. The first example concerns the two-dimensional Laplacian operator. One often uses domain partitioning to split the corresponding discrete grid into blocks of grid points with the aim of allocating blocks to processors. Since each grid point corresponds to one vector component, this amounts to distributing the vector over the processors in a locality-preserving manner. The complete row i of the Laplacian matrix is usually allocated to the same processor as vector component i. In our scheme, this can simply be achieved by taking $q_0 = p$ and $q_1 = 1$. Another example is the *square grid distribution*, which is the matrix distribution defined by

$$\phi_0(i) = \phi_1(i) = i \textbf{ mod } q_0, \tag{6}$$

where $q_0 = q_1 = \sqrt{p}$. This distribution is optimal for linear algebra computations such as dense LU decomposition [6]. This distribution is known under various names, such as *scattered square decomposition* [13] and *cyclic storage* [20]. (The grid distribution of a matrix should not be confused with discrete grids used to model e.g. PDE's.) Our general distribution scheme leaves much freedom in choosing particular mappings, and this can be exploited to achieve a good load balance and to reduce communication, see the next section. A detailed discussion and motivation of this distribution scheme in the context of sparse matrix-vector multiplication is given in [3].

Figure 1 presents a parallel sparse matrix-vector multiplication algorithm for a BSP computer. The algorithm consists of four supersteps: a fan-out of vector components to the processors that need them; a multiplication of the local part of the sparse matrix by the corresponding part of the input vector; a fan-in of partial sums; and, finally, the computation of the local part of the output vector. The fan-out and the fan-in are h-relations; the other supersteps are local computations. The communication requirements are derived from the computations on the basis of the "need to know". Matrix elements are not communicated. The only communication needed is that of vector components and of partial sums used to compute new vector components. The input and output vectors are required to be distributed in the same manner. This facilitates repeated application of the algorithm, e.g. in an iterative linear system solver. The sparsity of the matrix is exploited in two ways: first, computations are performed only for nonzero elements; second, communications are performed only if the matrix element that makes them necessary is nonzero.

{ $A: \ n \times n, \ \text{distr}(A) = \phi,$
$\mathbf{v}: \ n, \ \text{distr}(\mathbf{v}) = \text{distr}(\text{diag}(A))$ }

{ fan-out }
for all $j : 0 \leq j < n \ \wedge \ \phi_0(j) = s \ \wedge \ \phi_1(j) = t$ **do**
  send v_j to processors $\{(\phi_0(i), t) : 0 \leq i < n \ \wedge \ a_{ij} \neq 0\}$;

{ local sparse matrix-vector multiplication }
for all $i : 0 \leq i < n \ \wedge \ \phi_0(i) = s \ \wedge \ (\exists r : 0 \leq r < n \ \wedge \ \phi_1(r) = t \ \wedge \ a_{ir} \neq 0)$ **do**
begin
  $u_{it} := 0$;
  for all $j : 0 \leq j < n \ \wedge \ \wedge \phi_1(j) = t \ \wedge \ a_{ij} \neq 0$ **do** $u_{it} := u_{it} + a_{ij}v_j$
end;

{ fan-in }
for all $i : 0 \leq i < n \ \wedge \ \phi_0(i) = s \ \wedge \ u_{it} \neq 0$ **do**
  send u_{it} to processor $(s, \phi_1(i))$;

{ summation of partial sums }
for all $i : 0 \leq i < n \ \wedge \ \phi_0(i) = s \ \wedge \ \phi_1(i) = t$ **do**
begin
  $u_i := 0$;
  for all $k : 0 \leq k < q_1 \ \wedge \ u_{ik} \neq 0$ **do** $u_i := u_i + u_{ik}$
end

{$\mathbf{u}: \ n, \ \mathbf{u} = A\mathbf{v}, \ \text{distr}(\mathbf{u}) = \text{distr}(\mathbf{v})$}

Figure 1: Sparse matrix-vector multiplication algorithm for processor (s, t)

The notation of the algorithm should be interpreted as follows. The text given is the program text for a processor (s, t), with $0 \leq s < q_0$ and $0 \leq t < q_1$. The execution of the program depends on the parameters s and t. The **for all**-statements are implemented using an efficient data structure, so that unnecessary tests (such as $a_{ij} \neq 0$ or $u_{it} \neq 0$) are avoided. This implies that local vector components are easily accessible, and that local matrix nonzeros are stored in a sparse data structure that provides row-wise access. This data structure does not store rows that are locally empty. (A suitable data structure is the collection of sparse row vectors [9], with pointers only to the rows that are locally non-empty.) In our exposition, we assume that there are no accidental zeros caused by numerical cancellations. The h-relations are described by including for each data element to be communicated a “send"-statement in the program text of the source processor, together with the address of the destination processor. It is assumed that processors are willing to receive all the data that are sent to them in an h-relation. Because of this, there is no need to include explicit “receive"-statements. All data are described using global indices (in an implementation, it may be convenient to convert these to local indices). In particular, communicated data are described in global terms, which is convenient for making assertions in the program text about these data. The global description enables us to make such assertions, irrespective of whether they belong to the text of sending or receiving processors. The destination address of a message is determined by the sending processor. For the fan-in, this is done on the basis of pre-computed information, based on the sparsity pattern of A. In an implementation of the h-relations, the messages are packed into a send-buffer by the sending processor, then communicated, and after that stored in a receive-buffer and unpacked by the receiving processor.

The BSP cost of the sparse matrix-vector multiplication algorithm is determined as follows. The first superstep is the fan-out, which is a communication superstep. Let $h_{\mathrm{r}}(s,t)$ be the number of components v_j received by processor (s,t) and $h_{\mathrm{s}}(s,t)$ the number of components sent. Then define

$$h_{\mathrm{r}} = \max\{h_{\mathrm{r}}(s,t) : \ 0 \le s < q_0 \ \wedge \ 0 \le t < q_1\}, \tag{7}$$

$$h_{\mathrm{s}} = \max\{h_{\mathrm{s}}(s,t) : \ 0 \le s < q_0 \ \wedge \ 0 \le t < q_1\}, \tag{8}$$

$$h = \max\{h_{\mathrm{r}}, \ h_{\mathrm{s}}\}. \tag{9}$$

The BSP cost of the first superstep is $l + gh$, see Section 2.

The second superstep is the local sparse matrix-vector multiplication, which is a computation superstep. Let

$$r_i(t) = |\{j : \ 0 \le j < n \ \wedge \ a_{ij} \ne 0 \ \wedge \ \phi_1(j) = t\}|, \tag{10}$$

be the number of nonzeros in processor part t of matrix row i, $0 \le i < n$. Then the number of floating point operations of processor (s,t) is

$$w(s,t) = \sum_{\substack{i=0 \\ \phi_0(i)=s, \ r_i(t)>0}}^{n-1} (2r_i(t) - 1). \tag{11}$$

In this operation count, we include only non-trivial floating point operations; we exclude trivial operations involving zero operands. The maximum amount of work of a processor is

$$w = \max\{w(s,t) : \ 0 \le s < q_0 \ \wedge \ 0 \le t < q_1\}. \tag{12}$$

The BSP cost of the second superstep is $l + w$, see Section 2.

The third superstep is similar to the first, except that partial sums u_{ik} are communicated, instead of vector components v_j. The fourth superstep is similar to the second; its cost is determined as follows. Let

$$s_i = |\{k : 0 \le k < q_1 \wedge u_{ik} \ne 0\}|, \tag{13}$$

be the number of nonzero partial sums produced by matrix row i, $0 \le i < n$. Then the number of floating point operations of processor (s, t) is

$$w(s,t) = \sum_{\substack{i = 0 \\ \phi_0(i) = s,\ s_i > 0}}^{n-1} (s_i - 1). \tag{14}$$

The total BSP cost of the algorithm is obtained by adding the costs of the four supersteps. We denote the BSP cost for p processors by $T(p)$.

The BSP cost as defined above can be used to compare the efficiency of different distributions of the same matrix. To obtain a meaningful measure for comparison of different matrices it is necessary to normalise the cost. We define the *normalised BSP cost* $C(p)$ by

$$C(p) = \frac{pT(p)}{T_{\text{seq}}}, \tag{15}$$

where T_{seq} is the cost of the sequential algorithm. This sequential cost is defined by

$$T_{\text{seq}} = \sum_{\substack{i = 0 \\ r_i > 0}}^{n-1} (2r_i - 1), \tag{16}$$

where

$$r_i = |\{j : 0 \le j < n \wedge a_{ij} \ne 0\}|, \tag{17}$$

for $0 \le i < n$. In other words, the normalised BSP cost $C(p)$ of an algorithm is the ratio between the time $T(p)$ of that algorithm on a BSP computer and the time T_{seq} of a perfectly parallelised sequential algorithm. The normalised BSP cost of an algorithm is an expression of the form $a + bg + cl$, where a, b, and c are scalar values which depend on the algorithm, on the number of processors, and on the chosen data distribution. The scalars g and l are parameters that characterise the hardware, see Section 2. The normalised BSP cost of an ideal parallel algorithm is $1 + 0g + 0l$.

In summary, we have presented a simple methodology that leads to a useful measure of the efficiency of BSP algorithms and distributions. This measure, the normalised BSP cost $C(p)$, can, of course, be used to distinguish good algorithms and distributions from bad ones, but also to identify easy and hard problems for BSP computers.

6 Results for structure independent distributions

We have implemented a program that computes the normalised BSP cost $a + bg + cl$ of the sparse matrix-vector multiplication algorithm of Fig. 1 for a given sparse matrix and a given data distribution. In this section, we use this program to obtain experimental results on the performance of different data distribution schemes in a wide range of problem areas. Our cost statistics can be used to predict the computing time on an actual BSP computer, provided that the g and l parameters of the machine are available. For our experiments, we fix the number of processors at $p = 100$. The problem size, however, may vary, so that we are still able to investigate scalability.

Table 2 presents the normalised computing cost a for seven different data distributions and for all sparse matrices from MLIB, cf. Table 1. Table 3 presents the normalised communication cost b for the different data distributions and the normalised synchronisation cost c for a distribution that requires all the four supersteps of the algorithm to be present. (For a row distribution, with $q_1 = 1$, there is no need for a fan-in and a summation of partial sums, so that the number of supersteps becomes two and c is halved.) The value of c depends only on the number of supersteps, the number of processors, and the amount of work of the sequential algorithm, but in general not on the chosen distribution. For all distributions, the vectors **u** and **v** are distributed in the same manner as the diagonal of the matrix. All distributions, except “PRAM", are Cartesian, cf. eqn. 4.

The “PRAM" distribution is obtained by assigning nonzero elements randomly to the processors. This distribution is non-Cartesian, since in general there do not exist mappings ϕ_0 and ϕ_1 that satisfy eqn. 4. The “PRAM" distribution is included in the table, because it simulates the use of a BSP machine in PRAM mode, with randomised allocation of data by hashing. This mode of operation may be advantageous on machines with a low value of g [30]. In Tables 2 and 3, following the column of the “PRAM" distribution, there are three columns with results for random distributions. The random distribution with $q_0 = 100$ and $q_1 = 1$ assigns matrix rows i randomly to processors $(\phi_0(i), 0)$, with $0 \leq \phi_0(i) < 100$. The random distribution with $q_0 = q_1 = 10$ assigns an identifier $\phi_0(i)$, with $0 \leq \phi_0(i) < 10$, randomly to each matrix row i, and, independently, an identifier $\phi_1(j)$, with $0 \leq \phi_1(j) < 10$, to each matrix column j. An equalised random distribution of rows is similar to a random distribution, but it assigns the same number of rows to each identifier, if n **mod** $q_0 = 0$. Otherwise, the number of rows will differ by at most one. This procedure is equivalent to randomly permuting the rows and then distributing them according to the block distribution $\phi_0(i) = i$ **div** ℓ, where $\ell = n/q_0$ and it is assumed that n **mod** $q_0 = 0$. This random permutation procedure was proposed by Ogielski and Aiello [26] for use in a parallel algorithm for sparse matrix-vector multiplication. Ogielski and Aiello also present a probabilistic analysis that shows the advantages of this matrix distribution. (Our matrix-vector multiplication algorithm differs from theirs in that we reduce communication by exploiting sparsity and by choosing a vector distribution that matches the distribution of the matrix diagonal. Their algorithm has the same communication requirements as

row distr.	PRAM	random	random	eq. random	grid	block	diag.
q_0		100	10	10	10	10	10
column distr.			random	eq. random	grid	grid	
q_1		1	10	10	10	10	10
hyp.2.10.1	1.44	1.74	1.58	1.41	4.26	1.07	1.26
hyp.2.10.2	1.34	1.72	1.39	1.16	2.43	1.03	1.15
hyp.2.10.3	1.20	1.74	1.33	1.07	1.74	1.03	1.12
hyp.3.10.1	1.04	1.08	1.05	1.04	3.21	1.01	1.02
hyp.3.8.1	1.14	1.24	1.16	1.13	3.52	1.02	1.08
hyp.20.4.1	1.02	1.03	1.04	1.03	8.82	1.00	1.02
hyp.30.3.1	1.08	1.13	1.17	1.09	8.46	1.00	1.05
hyp.50.3.1	1.03	1.05	1.05	1.04	8.46	1.00	1.02
hyp.50.2.1	1.29	1.43	1.39	1.33	7.78	1.00	1.19
hyp.100.2.1	1.14	1.20	1.17	1.16	7.78	1.00	1.10
hyp.200.2.1	1.06	1.10	1.09	1.08	7.78	1.00	1.05
dense.100	2.14	4.03	2.41	1.12	1.41	1.00	1.00
dense.500	1.12	2.12	1.48	1.01	1.08	1.00	1.00
random.1000.1000	2.10	2.32	2.30	2.18	4.21	1.88	2.13
random.1000.100	1.48	1.77	1.61	1.46	4.00	1.29	1.28
random.1000.10	1.28	1.73	1.36	1.11	1.49	1.09	1.08
hb.impcolb	4.14	5.88	5.11	4.06	5.66	4.43	3.84
hb.west0067	3.90	5.33	4.70	3.74	7.29	3.84	3.42
hb.fs5411	1.69	2.12	2.71	2.50	6.41	2.42	2.24
hb.steam2	1.55	2.02	1.68	1.40	3.11	1.11	1.22
hb.shl400	4.98	31.52	5.05	4.74	6.99	4.38	4.74
hb.bp1600	2.24	7.73	2.32	2.15	4.64	2.34	2.11
hb.jpwh991	1.53	1.86	1.69	1.58	5.52	1.48	1.39
hb.sherman1	1.61	1.93	1.80	1.68	11.29	1.85	1.52
hb.sherman2	1.47	1.84	1.51	1.34	3.79	1.27	1.27
hb.lns3937	1.25	1.25	1.31	1.27	4.61	1.62	1.21
hb.gemat11	1.23	1.33	1.26	1.24	4.30	1.28	1.18
lp.1000	1.37	2.08	1.49	1.34	1.72	1.93	1.31
lp.6000	1.14	1.43	1.18	1.15	1.81	1.42	1.15
md.6000.20	1.21	1.31	1.26	1.25	5.78	1.19	1.18
md.6000.10	1.14	1.26	1.15	1.11	2.83	1.07	1.07
md.6000.8	1.11	1.26	1.11	1.08	2.04	1.05	1.05
mdr.6000.10.2000	1.13	1.25	1.14	1.11	2.71	1.06	1.06
mdr.6000.8.1000	1.11	1.26	1.11	1.07	1.91	1.05	1.04

Table 2: Computation cost for data distributions with $p = 100$

	Communication (in g)							Synch. (in l)
row distr.	PRAM	random	random	eq. random	grid	block	diag.	
q_0		100	10	10	10	10	10	> 1
column distr.			random	eq. random	grid	grid		
q_1		1	10	10	10	10	10	> 1
hyp.2.10.1	1.55	0.79	1.04	0.99	4.61	0.46	0.68	0.0186
hyp.2.10.2	1.31	0.66	0.30	0.29	1.59	0.16	0.17	0.0035
hyp.2.10.3	0.81	0.41	0.10	0.09	0.52	0.06	0.06	0.0011
hyp.3.10.1	0.95	0.48	0.42	0.42	3.65	0.31	0.39	0.0002
hyp.3.8.1	1.10	0.55	0.58	0.58	4.39	0.39	0.47	0.0018
hyp.20.4.1	0.93	0.47	0.64	0.64	2.35	0.18	0.61	0.0001
hyp.30.3.1	1.01	0.51	0.85	0.74	3.08	0.21	0.68	0.0011
hyp.50.3.1	0.94	0.47	0.69	0.69	3.08	0.19	0.67	0.0002
hyp.50.2.1	1.24	0.62	1.06	1.02	4.44	0.27	0.84	0.0178
hyp.100.2.1	1.04	0.52	0.86	0.85	4.44	0.24	0.77	0.0044
hyp.200.2.1	0.96	0.48	0.77	0.77	4.44	0.23	0.73	0.0011
dense.100	2.57	1.26	0.38	0.33	0.91	0.09	0.09	0.0201
dense.500	0.42	0.21	0.04	0.04	0.18	0.02	0.02	0.0008
random.1000.1000	3.27	1.68	3.02	2.89	14.60	2.26	2.54	0.3010
random.1000.100	1.77	0.89	1.14	1.10	6.37	0.74	0.73	0.0210
random.1000.10	1.10	0.54	0.17	0.16	0.91	0.09	0.09	0.0020
hb.impcolb	7.06	3.83	4.70	4.40	10.80	1.77	2.30	0.7080
hb.west0067	6.28	3.20	4.66	4.23	11.71	1.92	2.46	0.7678
hb.fs5411	2.55	1.58	1.40	1.29	6.96	1.17	1.03	0.0498
hb.steam2	1.75	0.88	0.78	0.74	3.98	0.25	0.40	0.0149
hb.shl400	5.24	15.37	2.79	2.64	10.07	2.43	2.49	0.1449
hb.bp1600	2.64	3.68	1.46	1.37	7.52	0.95	1.10	0.0451
hb.jpwh991	1.62	0.82	1.25	1.22	6.79	0.71	0.88	0.0361
hb.sherman1	1.64	0.82	1.37	1.33	3.39	0.57	1.04	0.0615
hb.sherman2	1.55	0.78	0.64	0.62	3.91	0.27	0.42	0.0089
hb.lns3937	1.26	0.64	0.92	0.91	7.37	0.84	0.76	0.0085
hb.gemat11	1.38	0.70	0.96	0.95	7.64	0.58	0.80	0.0065
lp.1000	1.22	0.60	0.21	0.20	1.17	0.11	0.14	0.0030
lp.6000	0.88	0.44	0.17	0.17	1.36	0.11	0.14	0.0006
md.6000.20	1.11	0.56	0.92	0.91	6.70	0.80	0.80	0.0091
md.6000.10	1.09	0.54	0.43	0.43	3.27	0.34	0.34	0.0013
md.6000.8	0.97	0.49	0.24	0.24	1.80	0.18	0.18	0.0007
mdr.6000.10.2000	1.07	0.53	0.39	0.39	2.96	0.30	0.30	0.0012
mdr.6000.8.1000	0.95	0.47	0.21	0.21	1.61	0.16	0.16	0.0006

Table 3: Communication and synchronisation cost for data distributions with $p = 100$

in the case of a dense matrix. Their vector distribution is based on a lexicographic ordering, which has no relation to the matrix distribution.) The cost results for the random distributions are the averages over 100 runs of the random distribution program. The standard deviations are small, so that we consider the results to be reliable. The random distributions were generated by using the pseudo-random number generator `ran2` from [28].

Tables 2 and 3 also present results for two deterministic distributions: the grid/grid distribution, which is the square grid distribution of eqn. 6, and the block/grid distribution, which is defined for the general case by

$$\ell_0 = \left\lfloor \frac{n}{q_0} \right\rfloor, \quad \ell_1 = \left\lceil \frac{n}{q_0} \right\rceil, \quad r = n \,\mathbf{mod}\, q_0, \tag{18}$$

$$\phi_0(i) = \begin{cases} i \,\mathbf{div}\, \ell_1 & \text{if } i < r\ell_1, \\ r + (i - r\ell_1) \,\mathbf{div}\, \ell_0 & \text{if } i \geq r\ell_1, \end{cases} \tag{19}$$

$$\phi_1(i) = i \,\mathbf{mod}\, q_1, \text{ for } 0 \leq i < n. \tag{20}$$

This distribution allocates rows in consecutive blocks to processors, and columns in a cyclic fashion. It was proposed as a suitable distribution for iterative linear system solvers [3], because it distributes the matrix diagonal over all the processors so that it can easily be matched with a vector distribution. (The square grid distribution does not have this advantage, because it distributes the diagonal over only $\sqrt{p}$ processors.) Finally, Tables 2 and 3 present a column with the results for the “diagonal" distribution. This distribution is determined by taking an equalised random distribution of the matrix diagonal over the processors. Note that in our distribution scheme, for a given choice of q_0 and q_1, the distribution of the matrix diagonal fully determines the distribution of the complete matrix and that of the vectors, see eqns 4 and 5.

The results of Table 2 show that it is relatively easy to obtain a good load balance, i.e. $a \approx 1$, and hence a minimal computation cost, except for very small matrices such as `dense.100`, `hb.impcolb`, and `hb.west0067`, and for extremely sparse ones such as `hb.shl400`. Most distributions lead to a normalised computation cost of between one and two. The exception is the square grid distribution, which leads to excessive workloads on diagonal processors (s, s) in the summation of the partial sums, because these are the only processors that participate in this superstep. (Note that this is directly related to the heavy communication obligations of the diagonal processors in the fan-in, since these processors are the only receivers of data.) A breakdown of the total BSP cost into the contributions of the separate supersteps confirms this analysis. Furthermore, it shows that the load balance of the grid distribution in the local sparse matrix-vector multiplication is about the same as that of the other distributions, except in the case of matrices with an unfavourable nonzero structure. This may occur if there is a correlation between the row and the column nonzero structures, resulting e.g. in diagonals of nonzeros. This may lead to a bad load balance for certain numbers of processors. This phenomenon can be observed for some of the hypercube matrices and the `hb.sherman` matrices. Furthermore, Table 2 shows that equalised random distributions lead to a better load balance than standard random distributions. In general, distributions that impose constraints balance the workload better. For example, the "PRAM" distribution does not impose any constraints except for an identical distribution of matrix diagonal and vectors. It does not perform very well on small problems and even for larger problems there are superior distributions, such as the "diagonal" distribution, which imposes an equal division of the matrix diagonal over the processors and hence causes a good load balance in the summation of partial sums.

The results of Table 3 show that it is quite hard to achieve a low communication cost for general sparse matrices, i.e. if one cannot exploit any structural knowledge about the matrix. Even for the best structure independent distributions, block/grid and "diagonal", one needs a BSP computer with $g \leq 10$ to solve most problems efficiently. The best performance is obtained by square distributions, i.e. distributions with $q_0 = q_1 = \sqrt{p}$. This leads to a factor of $\sqrt{p}/2$ communication reduction for dense [3] and general sparse matrices, compared to a row distribution. This is due to a $\sqrt{p}$-fold increase in the reuse of communicated data, at the cost of an extra communication phase, the fan-in. (A similar analysis can be performed for the BSP model.) This effect can most clearly be seen by comparing the random distribution for $q_0 = 100$ and $q_1 = 1$ with the random/random distribution for $q_0 = q_1 = 10$, in particular for relatively dense matrices such as `hyp.2.10.3`, `dense.500`, `random.1000.10`, `md.6000.8`, and `md.6000.8.1000`. On the other hand, for very sparse matrices such as `random.1000.1000` and `md.6000.2`, the introduction of the fan-in for $q_1 > 1$ doubles the communication, without much compensation by reuse of data. The "PRAM" distribution performs poorly, because nearly all the vector data must be fetched from non-local memories. This distribution is viable only if g is very close to one. Again, the square grid distribution is the worst distribution: the diagonal processors are the only ones that send data in the fan-out, and they are also the only ones that receive data in the fan-in; this may degrade performance by a factor of $\sqrt{p}$. The best distributions are the block/grid distribution and the "diagonal" distribution. They perform equally well for problems that have a random nature, such as the `random`, `md`, and `mdr` matrices. For problems that have some local structure that is reflected in the matrix, the block/grid distribution is able to discover part of this structure and to exploit it, to some extent. This can be observed for the `hyp` matrices, `hb.steam2`, and the `hb.sherman` matrices, which are all derived from multidimensional grids. Obviously, the random construction of the "diagonal" distribution prevents discovery of any structure. In a few cases, `hb.fs5411` and `hb.lns3937`, the block/grid distribution is outperformed by the "diagonal" one; this may be caused by an unfavourable structure that does not suit the block/grid distribution.

The synchronisation cost of the sparse matrix-vector multiplication is low, because it has at most four supersteps. The normalised synchronisation cost is

$$c \approx \frac{4}{2nz(A)/p} = \frac{2p}{nz(A)}. \tag{21}$$

This implies that problems with more than 200,000 nonzeros can be solved efficiently on a 100-processor BSP computer with $l \leq 1000$.

7 Results for structure dependent distributions

Table 4 shows the normalised communication cost for hypercube matrices of distance one and dimension $d = 2, 3, 4$, distributed by domain partitioning of the corresponding hypercube graph. The radix r is the number of points in each dimension, and $P_k,\ 0 \leq k < d$, is the number of subdomains into which dimension k is split. For example, the first line of the table gives the cost for a 50 $\times$ 50 grid that is split into 50 $\times$ 2 blocks, each of size 1 $\times$ 25. In all cases, we choose $q_0 = p$ and $q_1 = 1$, because we found no advantage in other choices of q_0 and q_1 for domain distribution of hypercube matrices of distance one. The distribution of the grid points and hence of the vector components uniquely determines the distribution of the matrix.

The results of Table 4 show that the lowest communication cost for separate dimension splitting is achieved if the resulting blocks are cubic. This is an immediate consequence of the surface-to-volume effect, where the communication across the block boundaries grows as the number of points near the surface, and the computation as the number of points within the volume of the block. In two dimensions, partitioning the grid into square blocks of size $r/\sqrt{p} \times r/\sqrt{p}$ reduces the communication by a factor of about $\sqrt{p}/2$, compared to splitting it into strips of size $r/p \times r$. This can be seen for example in the reduction by a factor of five for the 200 $\times$ 200 hypercube grid, comparing the cost for $P_0 = 100, P_1 = 1$ with that for $P_0 = 10, P_1 = 10$. The surface-to-volume ratio for cubes in dimension d is $2dp^{1/d}/r$. For each grid point, $4d + 1$ floating point operations must be performed. The value h of the h-relation to be realised equals the number of exterior boundary points, because all the values of these points must be received. By symmetry, the same argument holds for sending. Therefore, the normalised communication cost for cubic partitioning is

$$b = \frac{2dp^{1/d}}{(4d+1)r} \approx \frac{p^{1/d}}{2r}. \tag{22}$$

This formula explains the results for $d = 2$ and $P_0 = P_1 = 10$ in Table 4. It implies for instance that two-dimensional grid problems with more than 45 grid points per direction can be solved efficiently on 100-processor BSP computers with $g \leq 10$. This indicates that PDE solving on such a BSP computer is feasible, already for relatively small problem sizes.

It is possible to improve the distribution further, by partitioning the domain along specific hyperplanes, not necessarily parallel to the coordinate hyperplanes. (Note that

radix r	dim. d	P_0	P_1	P_2	P_3	comm. (in g)
50	2	50	2			0.231
50	2	10	10			0.089
100	2	100	1			0.222
100	2	50	2			0.116
100	2	10	10			0.044
200	2	100	1			0.111
200	2	50	2			0.058
200	2	10	10			0.022
40	3	20	5	1		0.096
40	3	10	10	1		0.077
40	3	10	5	2		0.065
40	3	5	5	4		0.054
20	4	20	5	1	1	0.147
20	4	10	10	1	1	0.118
20	4	10	5	2	1	0.100
20	4	5	5	4	1	0.082
20	4	5	5	2	2	0.082

Table 4: Communication cost for low-dimensional hypercube matrices with domain partitioning for $p = 100$

this implies that the dimensions are not split up separately.) An example is the case of the two-dimensional hypercube grid, which can be split into digital spheres of the form

$$B_R(\mathbf{a}) = \{\mathbf{x} \in \mathbf{Z}^2 : ||\mathbf{x} - \mathbf{a}||_1 \leq R\}, \tag{23}$$

where the norm in dimension d is defined by $||\mathbf{x}||_1 = \sum_{i=0}^{d-1} |x_i|$. In other words, all grid points with a Manhattan distance less than or equal to R to the centre **a** of such a sphere are allocated to the same processor. The spheres wrap around the boundaries of the grid. Figure 2 illustrates this distribution.

For an infinite grid, the centers of the spheres form a lattice, consisting of all integer linear combinations of the vectors $\mathbf{v}_0 = (R+1, R)$ and $\mathbf{v}_1 = (-R, R+1)$. Together the spheres form a tiling of the plane $\mathbf{Z}^2$. The advantage of tile partitioning over block partitioning is that there are a factor of $\sqrt{2}$ less points in the boundary layer, for sufficiently large partition sizes. Therefore, the normalised communication cost b is reduced by a factor of $\sqrt{2}$. For the example of Fig. 2, the cost is $b \approx 0.071$, compared to $b \approx 0.088$ for the corresponding block partitioning. Fig. 3 shows the normalised communication cost for the two distributions as a function of the number of grid points

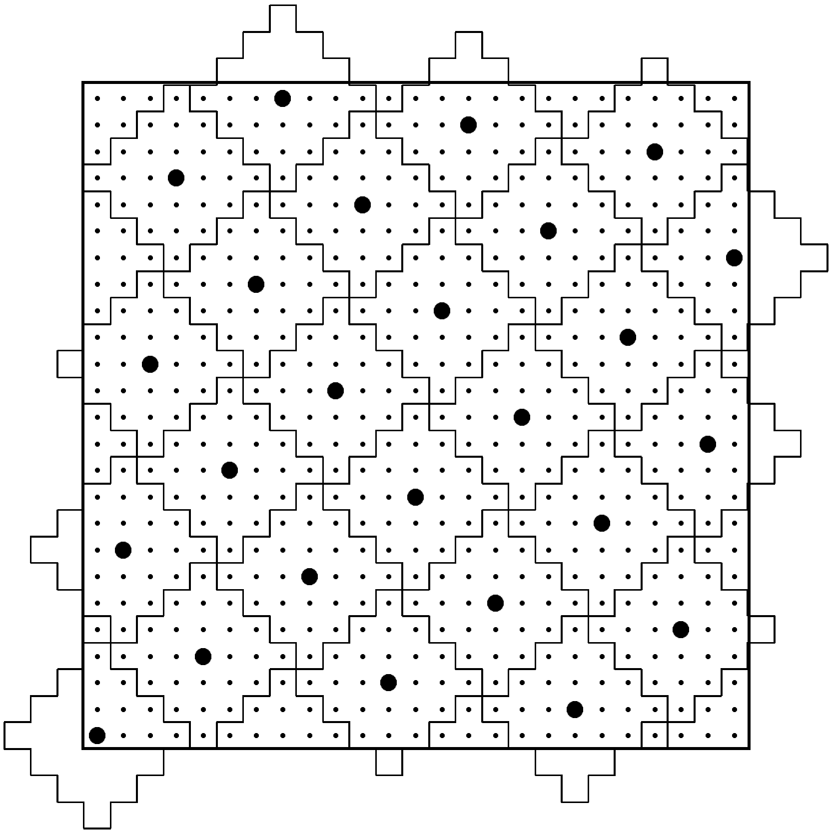

Figure 2: Partitioning of a 25×25 hypercube grid into 25 digital spheres of radius $R = 3$

per processor. The tile distribution is clearly superior, showing for instance a reduction by a factor of 1.34 for 221 grid points per processor. Note that problems of this size can be solved efficiently on BSP computers with $g \leq 50$, provided the tile distribution is used.

A complication that should be mentioned is that there may be a mismatch between the number of processors and the size of the grid. A perfect block distribution is possible only for very specific (square) numbers of grid points per processor, and similarly a perfect tile distribution is possible only for $2R^2 + 2R + 1$ grid points per processor, with R a non-negative integer. In the non-ideal case, a good distribution can still be obtained by splitting the plane along diagonal lines at suitable distances and assigning grid points accordingly.

Table 5 shows the BSP cost for various distributions of the molecular dynamics matrix `md.6000.10`. This matrix represents a three-dimensional universe of 6000 particles, contained in a box of size $1 \times 1 \times 1$ with periodic boundary conditions. Particles interact if their distance is less than $r_c = 0.1$. For convenience, the upper part of the table repeats the cost results for a few structure independent distributions from Tables 2 and 3. The lower part of the table presents the cost of structure dependent distributions; these exploit additional knowledge about the particle positions to assign

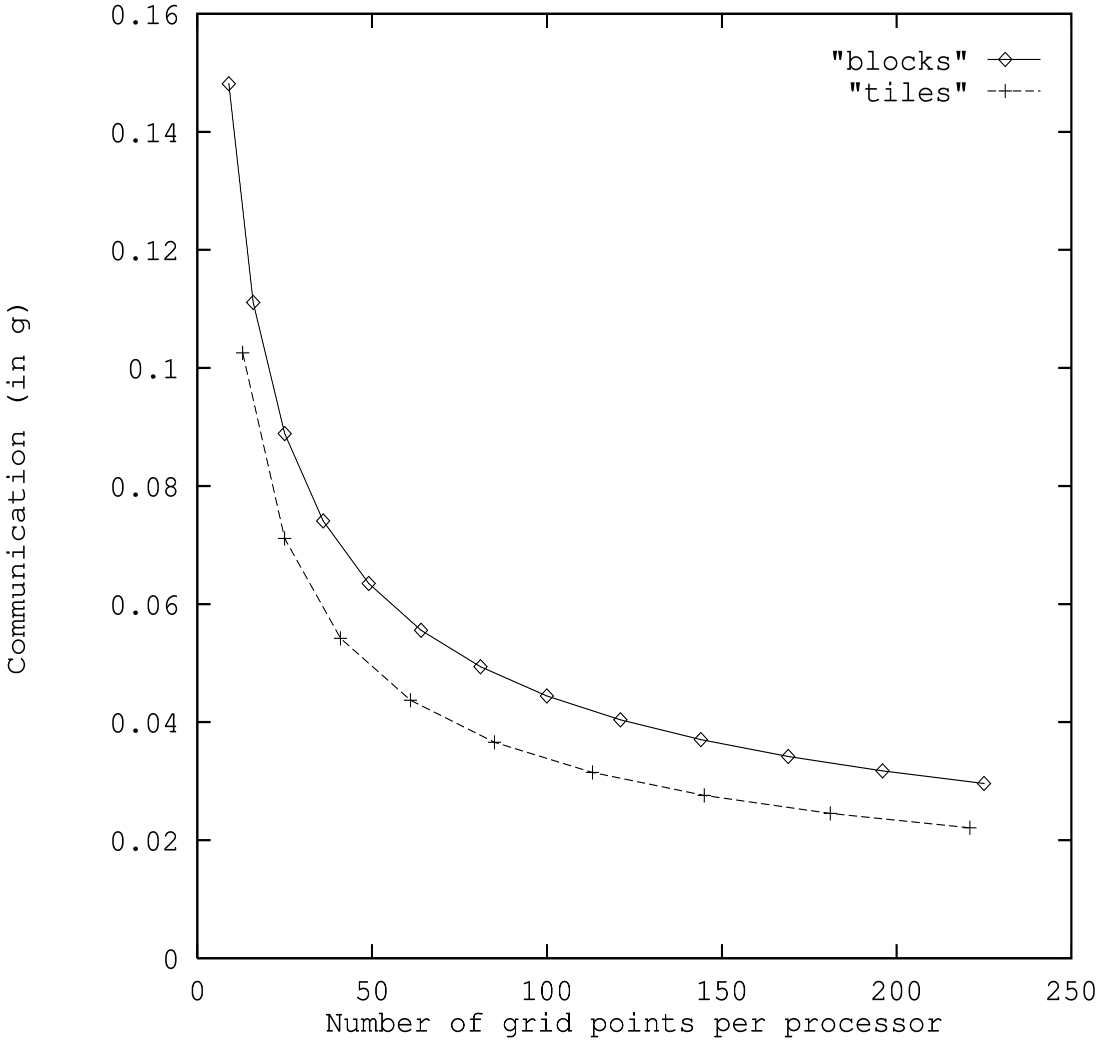

Figure 3: Communication cost comparison between block distribution and tile distribution of two-dimensional hypercube matrices

particles to subdomains and hence to processors.

The results for the structure independent distributions show that they achieve a good load balance but that they suffer from large amounts of communication. Even the best distributions of this type, block/grid and "diagonal", need BSP computers with a low value of g, $g \leq 3$, to prevent communication dominance. One can view these distributions as being based on so-called particle parallelism. Another approach is to distribute particles by using geometric parallelism, see [12] for an extensive discussion. This leads to structure dependent distributions as given in the lower part of the table. These distributions have lower communication requirements, but the price to be paid is a possible deterioration of the load balance, due to an inhomogeneous particle density.

Table 5 indicates that cubic subdomains are optimal among the orthogonal partitioning schemes, i.e., those schemes that split each dimension separately. Note that for non-cubic subdomains such as slabs or piles, choosing a square Cartesian distribution (with $q_0 = q_1$) improves the performance significantly. This is also done by Hendrickson and Plimpton [18] in the case of particle parallelism. For cubic subdomains, communication requirements are already reduced to such a low level, that this procedure, based on aggregation of partial sums, does not lead to further improvement. Note that the cut-off radius $r_c = 0.1$ of this matrix is quite large compared to the subdomain size. For partioning into slabs this implies that particle information must be sent to 20 other processors (so that it pays to aggregate information); for piles it must be sent to 4–8 other processors, depending on the position; and for near-cubes to 2–6 processors. (In our discussion we ignore the symmetry of particle interactions, which may be used to reduce the computation and the communication by a factor of two.)

It is possible to further improve the distribution by allowing cuts of the domain in any direction. This can be done efficiently by taking a suitable sphere packing lattice [7] and assigning particles to the nearest centre of a sphere. (Sphere packing lattices have been used in other areas of scientific computing; for instance, it has been proposed [5] to use them to decrease anisotropy in pseudo-spectral PDE solving on multidimensional grids.) This method splits the universe into Voronoi cells, each of which corresponds to a processor. Figure 4 shows the communication cost for the matrix `md.6000.20`, which represents 6000 particles with a cut-off radius of 0.05. For the cube distribution, the universe is split into cubes of size $p^{-1/3} \times p^{-1/3} \times p^{-1/3}$. This perfect splitting is, of course, only possible if the number of processors p is a cube. For the sphere packing distribution, we used a body-centred-cubic (bcc) lattice, defined by three basis vectors $\mathbf{v}_0 = (2, 0, 0)$, $\mathbf{v}_1 = (0, 2, 0)$, and $\mathbf{v}_2 = (1, 1, 1)$. The lattice is scaled by a factor $\lambda = (p/2)^{-1/3}$. This leads to a perfect splitting of the box if p equals two times a cube. The figure shows that the bcc distribution is slightly superior. (Note that this figure is based on experiments for one randomly generated `md` matrix and not on the average for a set of randomly generated matrices.) We chose the bcc lattice for this experiment because it has been conjectured that it solves the sphere covering problem. Our results indicate that sphere packing techniques may be useful in distributing physical domains over the processor of a parallel computer. This holds in

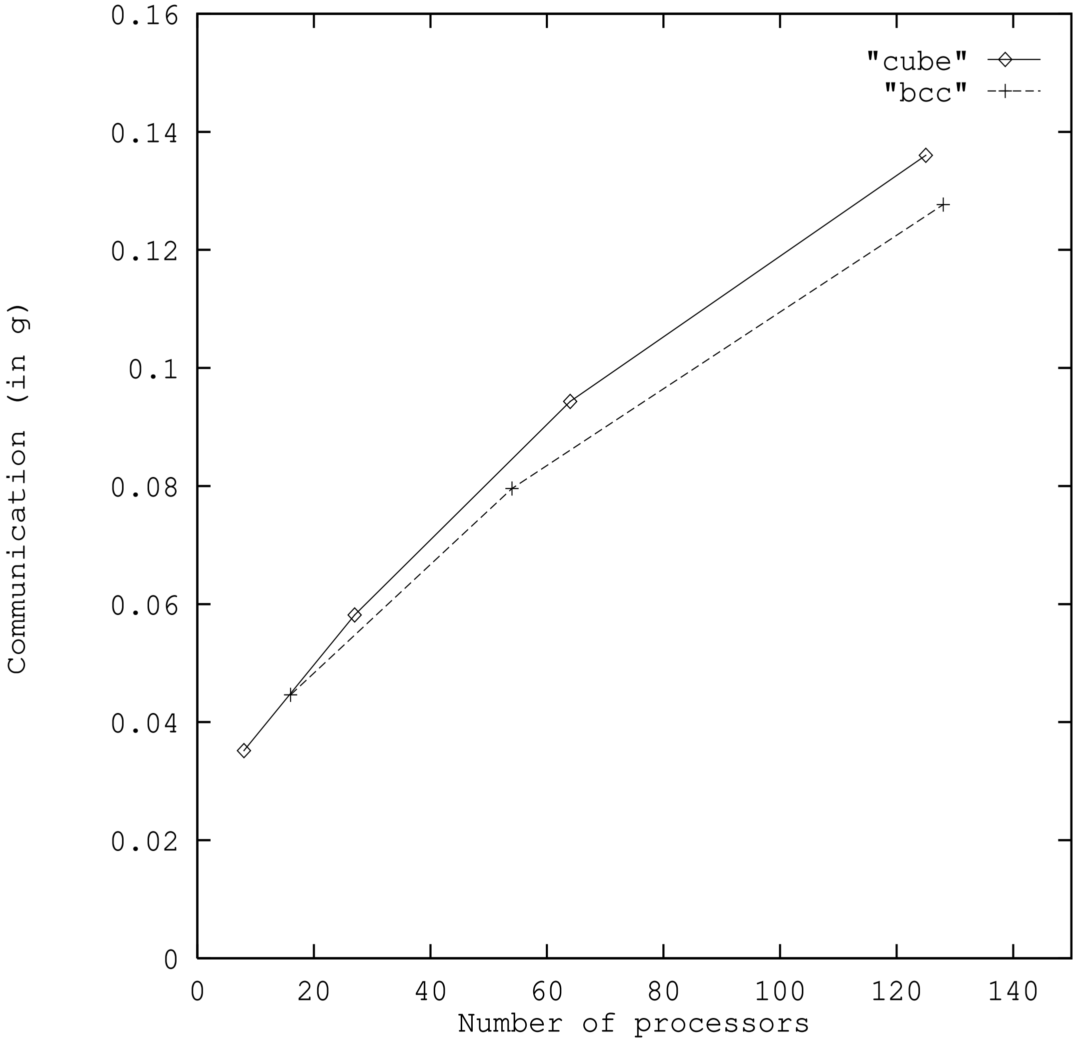

Figure 4: Communication cost comparison between cube distribution and bcc sphere packing distribution for molecular dynamics matrix `md.6000.20`

distribution	q_0	q_1	comp.	comm. (in g)	synch. (in l)
PRAM			1.14	1.087	0.0013
random row	100	1	1.26	0.541	0.0007
eq. random row and column	10	10	1.11	0.426	0.0013
block/grid	10	10	1.07	0.338	0.0013
diagonal	10	10	1.07	0.337	0.0013
slabs of size $0.01 \times 1.0 \times 1.0$	100	1	1.34	0.320	0.0007
slabs of size $0.01 \times 1.0 \times 1.0$	10	10	1.28	0.259	0.0013
piles of size $0.1 \times 0.1 \times 1.0$	100	1	1.41	0.108	0.0007
piles of size $0.1 \times 0.1 \times 1.0$	10	10	1.41	0.081	0.0013
near-cubes of size $0.2 \times 0.2 \times 0.25$	100	1	1.54	0.075	0.0007
near-cubes of size $0.2 \times 0.2 \times 0.25$	10	10	1.54	0.087	0.0013

Table 5: Normalised BSP cost for distributions with $p = 100$ of the matrix `md.6000.10`

particular for a BSP computer, because it liberates us from considerations of network locality. Therefore, there is no need for rigid partitioning schemes that produce highly regular domains. Further investigation of this issue is needed; for instance, there may exist better lattices for our purpose. Furthermore, to be useful in practice, finite-size effects must be taken care of.

8 Conclusion

The BSP model provides a new theoretical foundation for the development of scalable parallel computing systems. It offers a robust framework within which we can unify the various classes of parallel computers which are being produced (distributed memory architectures, shared memory multiprocessors, networks of workstations). The model permits and encourages the development of efficient parallel algorithms and programs which are both scalable and portable.

In this paper we provide the first theoretical and experimental analysis of the efficiency with which a wide range of important scientific computations can be performed on bulk synchronous architectures. The computations considered include the iterative solution of sparse linear systems, molecular dynamics, linear programming, and the solution of partial differential equations on a multidimensional discrete grid.

Our analysis shows that the exploitation of knowledge about the underlying structure of the problem is the key to achieving efficient parallel computations on a BSP computer. We have shown that grid computations and molecular dynamics simulations are feasible on BSP computers with realistic values for the machine characteristics g and l. Therefore, the BSP computers that can be built in the foreseeable future will be able to solve problems from several important problem classes. Highly irregular scientific computing problems without a known structure are much harder to solve on BSP computers. We have introduced two distributions, block/grid and "diagonal", see Section 6, that perform reasonably well on a variety of such problems. Our results show that structure independent parallel computations require extremely high communication performance and demand values of g that at present are difficult to achieve. This holds even more for the PRAM approach, which completely ignores the problem structure.

Providing a library of parallel algorithms to solve general sparse problems is a first step towards efficient parallel scientific computing, but to make further progress, this should be combined with developing algorithms that find structure in the problems, see e.g. [27] and [17]. The BSP model facilitates developing such algorithms, because it focuses attention on the partitioning of the problem to be solved and not on the mapping to any particular hardware.

The initial techniques and results described here show clearly that the network independent approach of the BSP model gives rise to a whole range of interesting new theoretical questions concerning load balancing, communication complexity, and domain partitioning for parallel scientific computing. In contrast to the many network specific (e.g. hypercube, mesh, or butterfly) process mapping and domain decomposition methods which were developed over the last decade, the techniques and results described here have an advantage in that they are of relevance to any parallel computing system.

References

[1] J. Almlof, K. Faegri, and K. Korsell, *J. Comput. Chem.*, **3** (1982), 385.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, PA, 1992.

[3] R. H. Bisseling, Parallel iterative solution of sparse linear systems on a transputer network, In: A. E. Fincham and B. Ford (eds) *Proc. IMA Conf. on Parallel Computing, Oxford 1991*, Oxford University Press, Oxford, UK, 1993, pp. 253–273.

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

[5] R. H. Bisseling and R. Kosloff, Optimal choice of grid points in multidimensional pseudospectral Fourier methods, *J. Comput. Phys.*, **76** (1988), pp. 243–262.

[6] R. H. Bisseling and J. G. G. van de Vorst, Parallel LU decomposition on a transputer network. In: G. A. van Zee and J. G. G. van de Vorst (eds.), *Parallel Computing 1988, Shell Conference*, Lecture Notes in Computer Science, **384**, Springer-Verlag, Berlin, 1989, pp. 61–77.

[7] J. H. Conway and N. J. A. Sloane, *Sphere Packings, Lattices, and Groups*, Springer-Verlag, New York, 1988.

[8] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

[9] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford, UK, 1986.

[10] I. S. Duff, R. G. Grimes, and J. G. Lewis, Sparse matrix test problems, *ACM Trans. Math. Softw.*, **15**

[11] I. S. Duff, R. G. Grimes, and J. G. Lewis, Users' guide for the Harwell-Boeing sparse matrix collection (Release I). Technical Report TR/PA/92/86, CERFACS, Toulouse, France, 1992.

[12] K. Esselink, B. Smit, and P. A. J. Hilbers, Efficient parallel implementation of molecular dynamics on a toroidal network. Part I. Parallelizing strategy, *J. Comput. Phys.*, **106** (1993), pp. 101–107.

[13] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving problems on concurrent processors*, Vol. 1, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[14] A. V. Gerbessiotis and L. G. Valiant, Direct bulk-synchronous parallel algorithms. In: O. Nurmi and E. Ukkonen (eds.), *Proc. Third Scandinavian Workshop on Algorithm Theory 1992*, Lecture Notes in Computer Science, **621**, Springer-Verlag, Berlin, 1992, pp. 1–18.

[15] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Second Edition, The Johns Hopkins University Press, Baltimore, MD, 1989.

[16] M. Guest, P. Sherwood, and J. van Lenthe, Concurrent supercomputing at ERC Daresbury Laboratory, *Supercomputer*, **36** (1990), pp. 89–103.

[17] B. Hendrickson and R. Leland, An improved spectral graph partitioning algorithm for mapping parallel computations. Technical Report 92-1460, Sandia National Laboratories, Albuquerque, NM, 1992.

[18] B. Hendrickson and S. Plimpton, Parallel many-body simulations without all-to-all communication. Technical Report 92-2766, Sandia National Laboratories, Albuquerque, NM, 1993.

[19] M. R. Hestenes and E. Stiefel, Methods of conjugate gradients for solving linear systems, *J. Res. Nat. Bur. Stand.*, **49** (1952), pp. 409–436.

[20] S. L. Johnsson, "Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures", *J. Parallel Distrib. Comput.* **4** (1987), pp. 133–172.

[21] N. Karmarkar, A new polynomial-time algorithm for linear programming, *Combinatorica*, **4** (1984), pp. 373–395.

[22] C. Lanczos, An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, *J. Res. Nat. Bur. Stand.*, **45** (1950), pp. 255–282.

[23] W. F. McColl, Special purpose parallel computing. In: A. M. Gibbons and P. Spirakis (eds.), *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation*, Cambridge University Press, Cambridge, UK, 1993, pp. 261–336.

[24] W. F. McColl, General purpose parallel computing. In: A. M. Gibbons and P. Spirakis (eds.), *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation*, Cambridge University Press, Cambridge, UK, 1993, pp. 337–391.

[25] W. F. McColl, GL: An architecture independent programming language for scalable parallel computing. Technical Report 93-072-3-9025-1, NEC Research Institute, Princeton, NJ, 1993.

[26] A. T. Ogielski and W. Aiello, Sparse matrix computations on parallel processor arrays, *SIAM J. Sci. Comput.*, **14** (1993), pp. 519–530.

[27] A. Pothen, H. D. Simon, and K.-P. Liou, Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, **11** (1990), pp. 430–452.

[28] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, *Numerical Recipes in Pascal*, First Edition, Cambridge University Press, Cambridge, UK, 1989.

[29] Y. Saad and M. H. Schultz, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, *SIAM J. Sci. Stat. Comput.*, **7** (1986), pp. 856–869.

[30] L. G. Valiant, A bridging model for parallel computation, *Commun. ACM*, **33** (1990), pp. 103–111.

[31] L. G. Valiant, General purpose parallel architectures. In: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science: Volume A, Algorithms and Complexity*, North-Holland, Amsterdam, 1990, pp. 943–971.

[32] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, PA, 1992.