Scientific Computing on Bulk Synchronous Parallel Architectures

R. H. Bisseling and W. F. McColl*†‡§

December 1993

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 parallel architectures. The computations considered include the iterative solution of sparse linear systems, molecular dynamics, and the solution of partial differential equations on a multidimensional discrete grid. We analyse these computations in a uniform manner by formulating their basic procedures as a sparse matrix-vector multiplication.

Keywords: BSP, iterative methods, models of parallel computation, scalable parallel computing, scientific computing, sparse matrices, sparse matrix-vector multiplication.

1 Introduction

Bulk synchronous parallel (BSP) architectures [26] 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

*R. H. Bisseling is with the Department of Mathematics, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, The Netherlands. E-mail: bisseling@math.ruu.nl.

†W. F. McColl is with the Programming Research Group, Oxford University, Wolfson Building, Parks Road, Oxford OX1 3QD, UK. E-mail: Bill.McColl@comlab.oxford.ac.uk

‡This paper is a revised version of Preprint 836, Department of Mathematics, Utrecht University, December 1993. Last revision: November 1994.

§This work was initiated while the authors were employed as a research mathematician and a senior visiting scientist, respectively, at Koninklijke/Shell-Laboratorium, Amsterdam. A brief preliminary version of this paper appeared in *Proc. 13th World Computer Congress: Vol. I*, B. Pehrson and I. Simon, Ed. IFIP Transactions A, vol. 51. Amsterdam: Elsevier, 1994, pp. 509–514.

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, 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 synchronisation period. An ideal parallel algorithm has the values $a = 1$, $b = 0$, and $c = 0$; an algorithm with load imbalance or redundancy 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 the new value 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, if we distribute that grid by 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. The choice of a data distribution is one of the main means of influencing the performance of an 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 for example to the application of tiling techniques to reduce communication in discrete grid calculations.

We present experimental results for the multiplication $\mathbf{u} := A\mathbf{v}$ of a sparse matrix A and a dense vector $\mathbf{v}$. 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 $\mathbf{u}$ and $\mathbf{v}$ and the diagonal of A are distributed in the same way and that the matrix A is distributed in a so-called *Cartesian* manner [6]. This means that the p

processors used by the algorithm 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, we will show that 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. a numbering 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 will 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 [26] (and also to [27]). 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 the processors. Specialised broadcasting or combining facilities are not available. If we define a time step to be the time required for a single local operation, i.e. a basic operation such as a floating point addition or multiplication on locally held data values, then the performance of any BSP computer can be characterised by the following four global parameters:

$p =$ number of processors;
$s =$ processor speed, i.e. number of time steps per second;
$l =$ network latency, i.e. minimum possible number of time steps between successive synchronisation operations;
$g =$ communication throughput ratio, i.e. total number of local operations performed by all processors in one second divided by 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 time required to realise an h-relation in a situation of continuous message traffic is proportional to h. The proportionality

constant is the parameter g of the BSP computer: 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 non-local 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.

The BSP computer is a two-level memory model [22], i.e. each processor has its own physically local memory module and all other memory is non-local and 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 is 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 that is currently used, as in special purpose parallel computing [21]. Instead, performance of the communications network is described only in terms of its global properties, using the parameters l and g.

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 $h_{\rm s}$ be the maximum number of messages sent by any processor during S, and $h_{\rm r}$ be the maximum number of messages received by any processor during S. In the original BSP model [26], the cost of S is $\max\{w, gh_{\rm s}, gh_{\rm r}, l\}$ time steps. An alternative [12] is to charge $\max\{w + gh_{\rm s}, w + gh_{\rm r}, l\}$ time steps for superstep S. In this paper, we will charge $w + g \cdot \max\{h_{\rm s}, h_{\rm r}\} + l$ 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 are done in parallel and which ones in sequence. The difference is not crucial; for instance, our cost is an upper bound for the cost in the original model and it is at most three times higher than that cost. This worst-case factor of three is attained only in the case that all three terms w, $g \cdot \max\{h_{\rm s}, h_{\rm r}\}$, and l happen to be equal. Our choice of superstep cost is motivated by its convenience in obtaining a simple numeric expression for the cost of an algorithm on a BSP computer with unspecified 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. (For example, c is the number of supersteps of the algorithm.) The normalised BSP cost of an algorithm, which will be defined below, cf. (15), has the same simple form $a + bg + cl$.

For simplicity, we assume in this paper that a superstep either consists of steps of type (i), or steps of type (ii), but not both. This implies that either the first or the second term of the cost $w + g \cdot \max\{h_{\rm s}, h_{\rm r}\} + l$ is zero. This assumption follows naturally from our particular choice of cost charging, which reflects an implementation where computation and communication do not overlap. The effect of this additional assumption on the total cost of the algorithm is at most a doubling of c, because each mixed-type superstep is split

into two supersteps. In the complexity analysis of a BSP algorithm, consecutive supersteps of type (i) are combined so that they incur only once a charge of l; this is done because such computation supersteps do not require synchronisation between them. Supersteps of type (ii) are not combined; there may be situations in which it is beneficial to perform certain communications in separate supersteps with synchronisation in between.

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. (In this paper, we use the direct approach to BSP programming, where the thread locality is controlled directly by the user. This may lead to more favourable data distributions than those obtained by an automatic procedure such as hashing.) To achieve high efficiency on a BSP computer with a large l value, we must design sufficiently long computation and communication supersteps to amortise the synchronisation costs. Each computation superstep should have at least l operations. To achieve efficiency 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 GPL [23].) The resulting algorithms will therefore be efficiently implementable 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 [12, 26]. 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 parallel 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 led to the extensive use of linear algebra libraries such as LAPACK [2]. To achieve portability, many scientific computer programs rely on using the Basic Linear Algebra Subprograms (BLAS) for their matrix and vector operations. Today, efficient BLAS implementations are available for most computer architectures. Another reason for the importance of linear algebra is that it provides a powerful formalism for expressing scientific computations, including 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 [28].

An important application area of linear algebra is the solution of partial differential equations (PDEs) by finite difference, finite element, or finite volume methods. These

methods require the repeated solution of systems of linear equations $A\mathbf{x} = \mathbf{b}$, where A is an $n \times n$ nonsingular matrix, and $\mathbf{x}$ and $\mathbf{b}$ are vectors of length n. Usually, the matrix A is *sparse*, i.e., only $\mathcal{O}(n)$ of its n^2 elements are nonzero. Such systems can be solved by a direct algorithm, using for example Cholesky factorisation or LU decomposition, see [13]. An alternative approach is to use an iterative algorithm, based on successive improvements of approximate solution vectors $\mathbf{x}^{(k)}$. An important example is the conjugate gradient algorithm [18] for symmetric positive definite matrices. (A variety of iterative algorithms are available, in the form of a high-level implementation, in the TEMPLATES collection [3].) Iterative methods use the matrix A mainly in a multiplicative manner, computing matrix-vector products of the form $\mathbf{u} := A\mathbf{v}$ or $\mathbf{u} := A^T\mathbf{v}$. Iterative methods are increasingly becoming popular, because they enable the solution of very large linear systems such as those originating in PDE solving on 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 oil reservoir simulation, semiconductor device modelling, and aerodynamics computations. Iterative methods can be employed to solve these systems using quite reasonable amounts of computer time and memory. In contrast, direct methods will normally break down for these problems, because they create too many new nonzero elements in the matrix, leading to a prohibitive use of computer resources.

Another application area of linear algebra is ab initio quantum chemistry, and in particular the solution of the time-independent Schrödinger equation by the direct SCF method [1]. The dominant part of this computation is the calculation of two-electron integrals and their incorporation into a matrix (the Fock 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 [14].

These examples 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. For scientific applications that are not directly 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.

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

1. Sparse matrix-vector multiplication is the basis of iterative methods for the solution of sparse linear systems $A\mathbf{x} = \mathbf{b}$. At every iteration, the matrix A (and in certain cases A^T) is multiplied by a vector, and the resulting vector is used to update the current approximate solution. Similarly, it is also the basis for the Lanczos method [20], which is often used to solve sparse symmetric eigenproblems $A\mathbf{x} = \lambda\mathbf{x}$, see [13].

2. Sparse matrix-vector multiplication represents the execution of the finite difference

operator in certain PDE solvers. The finite-difference matrix may be formed explicitly, or, in matrix-free methods, it may only exist implicitly since the finite difference operator is applied directly to the current approximate solution. An example is the five-point Laplacian finite difference operator used to solve the Poisson equation on a two-dimensional grid of size $r \times r$. This operator can be formulated in matrix terms (see e.g. [25]) by defining an $n \times n$ matrix A, with $n = r^2$, by

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

The solution value of the PDE at a grid point $(k, m), 0 \leq k, m < r$, corresponds to component $x_i, 0 \leq i < n$, of the solution of a linear system $A\mathbf{x} = \mathbf{b}$, by the relation $i = kr + m$. In a matrix-free PDE solver, the equivalent of a sparse matrix-vector multiplication $\mathbf{u} := A\mathbf{v}$ will simply be executed by adding the values of $\mathbf{v}$ at the neighbouring grid points $(k+1, m)$, $(k-1, m)$, $(k, m+1)$, and $(k, m-1)$, and subtracting four times the value of $\mathbf{v}$ at the grid point (k, m), to produce the value of $\mathbf{u}$ at (k, m).

3. Sparse matrix-vector multiplication may be used to model two-particle interactions in molecular dynamics simulations and in many-body simulations in general. For example, consider a cubic molecular dynamics universe of size $1 \times 1 \times 1$ with periodic boundaries. The universe is filled with n particles, numbered $0 \leq i < n$. Each particle moves under the influence of 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 $F_i = \sum_{j=0}^{n-1} F_{ij}$, with $F_{ii} = 0$. 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 particle i one needs to know the position of the particle i itself and the positions of the particles with which it interacts. The need for information about particle positions can be expressed in an $n \times n$ matrix A, defined by

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

An analogy to the force computation from the positions of the particles is the matrix-vector multiplication $\mathbf{u} := A\mathbf{v}$, where $\mathbf{u}$ is a vector that models the force components and $\mathbf{v}$ is a vector that models the particle positions. For short-range potentials, there exists a cut-off radius $r_c > 0$, such that particles i and j do not interact if their distance is larger than 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. Efficient simulation methods exploit the sparsity to limit the total number of force computations. Furthermore, distributed memory parallel algorithms based on geometric parallelism exploit the sparsity also to reduce the number of communications of current particle positions [10].

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 [16] on parallel many-body simulations. 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 broadcast of all particle positions to all processors, which occurs in all-to-all methods, by partial broadcasts of these positions and partial combines of accumulated forces. Sparsity is used to reduce the total number of force computations F_{ij}, but it is not used to reduce the communication. Because of this, the method is most suited for long-range or medium-range potentials.

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 of radix r and dimension d. The adjacency matrix A of such a graph is sparse and its order $n = r^d$ grows rapidly with increasing radix or increasing dimension. 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 [8], which is widely being used to test sparse matrix algorithms. It contains a number of examples of matrices that occur in practical applications. We have included a small subset of the HB library in MLIB, mainly to facilitate our experiments on such practical matrices. For our specific purpose of modelling a wide range of scientific computations, the HB matrices are not particularly 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.

MLIB is a collection of 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 given in [7], i.e. 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. Presently, 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. The format of a matrix file is: first, a line containing the number of rows and the number of columns of the matrix; after that, the nonzeros, one per line; 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 $n \times n$ hypercube matrix with radix r, dimension d, and distance D, where $n = r^d$. 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 r^d points. The vertices are numbered lexicographically. Each vertex has directed edges to itself and to its immediate neighbours in each dimension. 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 $n \times n$ dense matrix. 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. Here, the density is defined as $\rho = nz(A)/n^2$, where $nz(A)$ is the number of nonzeros of A. The sparsity structure of A is generated by using the pseudo-random number generator `ran1` from [25]. (All random numbers used in this paper were produced by this generator.)

- `hb.x`, the matrix `x` from the HB collection [8]. For a description of the matrix, see [9]. The subset of the HB collection that is included in MLIB consists of five matrices from various application fields.

- `md`.$n.r_c^{-1}$, an $n \times n$ matrix which corresponds to a random configuration of n particles in a three-dimensional molecular dynamics universe with short-range potentials, see Section 3. The matrix element a_{ij} is nonzero if $||\mathbf{r}_i - \mathbf{r}_j||_2 \leq r_c$, where $\mathbf{r}_i$ is the position of particle i, r_c the cut-off radius, and $||\cdot||_2$ the standard Euclidean norm in three-dimensional space. 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`.$n.r_c^{-1}.\rho^{-1}$, an $n \times n$ matrix which corresponds to a random configuration of n particles in a three-dimensional molecular dynamics universe 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.

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

Table 1: Matrix library MLIB

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.fs5411	541	4285
hb.steam2	600	13760
hb.jpwh991	991	6027
hb.sherman2	1080	23094
hb.lns3937	3937	25407
hb.gemat11	4929	33185
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

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 $\phi_0 : \{0,1,\ldots,n-1\} \to \{0,1,\ldots,q_0-1\}$ and $\phi_1 : \{0,1,\ldots,n-1\} \to \{0,1,\ldots,q_1-1\}$, such that matrix elements are distributed according to

$$a_{ij} \longmapsto \text{processor}(\phi_0(i), \phi_1(j)),\ \text{for } 0 \leq i,j < n. \tag{4}$$

Note that the elements of a matrix row are mapped to processors with the same first coordinate and that the elements of a matrix column are mapped to processors with the same second coordinate. This property reflects the row-wise and column-wise nature of most linear algebra operations. As a consequence, communication is often needed only within a subset of q_0 or q_1 processors. The two-dimensional numbering of the processors originates in special purpose algorithms for mesh networks [4, 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

$$u_i \longmapsto \text{processor}(\phi_0(i), \phi_1(i)),\ \text{for } 0 \leq i < n. \tag{5}$$

This distribution scheme is sufficiently general in that it includes most commonly used distribution methods. It is also sufficiently restrictive in that it imposes efficient communication patterns. The following two examples illustrate the generality of the scheme. The first example concerns a Laplacian operator on a discrete grid. Often, domain partitioning is used to split the grid into blocks of grid points with the aim of allocating complete blocks to processors. Since each grid point corresponds to one vector component, this amounts to distributing a 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

$$\phi_0(i) = \phi_1(i) = i \bmod \sqrt{p},\ \text{for } 0 \leq i < n, \tag{6}$$

where $q_0 = q_1 = \sqrt{p}$. This distribution is optimal for linear algebra computations such as dense LU decomposition [6, 17]. It is also known under other names such as *scattered square decomposition* [11], *cyclic storage* [19], and *torus-wrap mapping* [17]. (The term "grid distribution" should not be confused with the term "grid" used in the context of

$\{\ A:\ n \times n,\ \mathrm{distr}(A) = (\phi_0, \phi_1),\ \mathbf{v}:\ n,\ \mathrm{distr}(\mathbf{v}) = \mathrm{distr}(\mathrm{diag}(A))\ \}$
$\{\ I_{st} = \{i : 0 \le i < n\ \wedge\ \phi_0(i) = s\ \wedge\ (\exists j : 0 \le j < n\ \wedge\ \phi_1(j) = t\ \wedge\ a_{ij} \neq 0)\}\ \}$

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

{ local sparse matrix-vector multiplication }
for all $i : i \in I_{st}$ **do**
    $u_{it} := (\sum_j a_{ij} v_j : 0 \le j < n\ \wedge\ \phi_1(j) = t\ \wedge\ a_{ij} \neq 0)$;

{ fan-in }
for all $i : i \in I_{st}$ **do**
    send u_{it} to processor $(s, \phi_1(i))$;

{ summation of partial sums }
for all $i : 0 \le i < n\ \wedge\ \phi_0(i) = s\ \wedge\ \phi_1(i) = t$ **do**
    $u_i := (\sum_k u_{ik} : 0 \le k < q_1\ \wedge\ I_{sk} \ni i)$;

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

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

PDE modeling.) The general distribution scheme (4)–(5) leaves much freedom in choosing particular mappings, and this can be exploited to achieve a good load balance and to reduce communication. A detailed discussion and motivation of this scheme is given in [4].

Figure 1 presents a sparse matrix-vector multiplication algorithm for a BSP computer. The algorithm consists of four supersteps: a fan-out of input vector components to the processors that need them; 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, a summation of partial sums to compute the local part of the output vector. The fan-out and fan-in are communication supersteps; the multiplication and summation are computation supersteps. The computations of this algorithm are determined by the choice of data distribution and by our decision not to communicate any matrix elements. The communications are derived from the computations on the basis of the "need to know". The only communication needed is that of input vector components and of partial sums used to compute output vector components. The input and output vectors are required to be distributed in the same manner to facilitate 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 use only the nonzero elements of the matrix; second, communications are needed only because of nonzeros elements.

The notation of Fig. 1 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 s and t. The data are described using global indices. (In an implementation, it may be convenient to convert these to local indices.) The statements of the program are described using expressions such as $a_{ij} \neq 0$, but in an implementation such tests are avoided by using suitable data structures. This means that local vector components are easily accessible and that local matrix nonzeros are available through a sparse data structure that provides row-wise access. This sparse data structure stores only those rows that are locally non-empty, i.e. rows $i \in I_{st}$. A suitable data structure is the collection of sparse row vectors [7], with pointers only to non-empty rows. The communication supersteps 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, so that there is no need to include explicit "receive"-statements in the program text. It is also assumed that messages from a processor to itself are not sent, even though they may occur in the program text. The destination address of a message is determined by the sending processor. This can be done using pre-computed information, based on the sparsity pattern of A. In an efficient implementation, 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 an h-relation. Let $h_{\rm r}(s,t)$ be the number of components v_j received by processor (s,t) and $h_{\rm s}(s,t)$ the number of components sent. Then define

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

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

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

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

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

$$r_{it} = |\{j : \ 0 \leq j < n \ \wedge \ \phi_1(j) = t \wedge \ a_{ij} \neq 0 \ \}|, \tag{10}$$

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

$$w(s,t) = (\textstyle\sum_i (2r_{it} - 1) : i \in I_{st}), \tag{11}$$

since there are r_{it} multiplications and $r_{it} - 1$ additions for the local part of row i. The maximum amount of work of any processor is

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

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

The third superstep is similar to the first, except that partial sums u_{it} are communicated instead of vector components v_j. The fourth superstep is similar to the second; its local work load $w(s,t)$ is determined as follows. Let

$$s_i = |\{k : 0 \leq k < q_1 \ \wedge \ I_{\phi_0(i),k} \ni i\}|, \tag{13}$$

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

$$w(s,t) = (\textstyle\sum_i (s_i - 1) : 0 \leq i < n \ \wedge \ \phi_0(i) = s \ \wedge \ \phi_1(i) = t \ \wedge \ s_i > 0). \tag{14}$$

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

The BSP cost as defined above can be used to compare the efficiency of different distributions of the same matrix. To compare efficiency for different matrices, however, 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

$$T_{\text{seq}} = (\textstyle\sum_i (2r_i - 1) : 0 \leq i < n \ \wedge \ r_i > 0), \tag{16}$$

where

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

for $0 \leq i < n$. 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}/p of a perfectly parallelised sequential algorithm. The normalised BSP cost is an expression of the form $a + bg + cl$, where a, b, and c are scalars which depend on the algorithm, the number of processors, and 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 $1 + 0g + 0l$. (The normalised BSP cost $C(p)$ is the inverse of the commonly used efficiency measure $E(p) = T_{\text{seq}}/pT(p)$. We propose to use $C(p)$ because it leads to simpler expressions in g and l.)

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 distributions in a wide range of problem areas. Our results can be used to predict the execution time on an actual BSP computer, provided that the machine parameters s, g, and l 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 values a, b, and c of the normalised BSP cost $a + bg + cl$ for the matrices from Table 1, for five different data distributions. (Additional results can be found in [5].) The value of c is the same for all five distributions, because c depends only on the number of supersteps, the number of processors, and the amount of work of the sequential algorithm. The data distribution is fully determined by the matrix distribution, because we distribute vectors in the same manner as the diagonal of the matrix. The matrix distributions are as follows. The PRAM distribution is obtained by assigning the nonzero elements of the matrix randomly to the processors. This distribution is non-Cartesian, since there need not exist mappings ϕ_0 and ϕ_1 that satisfy (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 acceptable on machines with a low value of g [26]. All other distributions are Cartesian, see (4), and square, i.e. $q_0 = q_1 = \sqrt{p}$.

The random/random distribution randomly assigns an identifier $\phi_0(i)$, $0 \leq \phi_0(i) < q_0$, to each matrix row i, and, independently, an identifier $\phi_1(j)$, $0 \leq \phi_1(j) < q_1$, to each matrix column j. This is done in such a way that an equal or near-equal number of rows ($\lceil n/q_0 \rceil$ or $\lfloor n/q_0 \rfloor$) is assigned to each identifier, and similarly for columns. This is equivalent to randomly permuting the rows and columns, and then distributing the matrix according to the square grid distribution (6). This random permutation procedure was proposed by Ogielski and Aiello [24] for use in a parallel algorithm for sparse matrix-vector multiplication. Ogielski and Aiello [24] also show that, with high probability, the random/random distribution has a good load balance. (Our algorithm differs from the algorithm of [24] , and also from a similar algorithm of Hendrickson, Leland, and Plimpton [15], in that we reduce communication by exploiting sparsity and by choosing a vector distribution that matches the distribution of the matrix diagonal. The algorithms of [15, 24], however, require the same amount of communication in the sparse case as in the dense case. In the present work, output vectors are distributed in the same way as input vectors. This is also achieved in the algorithm of [15], by using an extra communication operation, a vector transposition. In the algorithm of [24], the output and input vectors are distributed differently: $\mathbf{u}$ is distributed by a row-wise lexicographic ordering and $\mathbf{v}$ by a column-wise one. Repeated use of this algorithm may necessitate vector redistributions.)

The next two distributions are deterministic. The grid/grid distribution is the square grid distribution of (6). The block/grid distribution is defined by

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

Table 2: Normalised BSP cost $a + bg + cl$ for five different data distributions with $p = 100$

Cost variable	a					b					c
Distribution	PRAM	rand/ rand	grid/ grid	block/ grid	diag	PRAM	rand/ rand	grid/ grid	block/ grid	diag	all
hyp.2.10.1	1.44	1.41	4.26	1.07	1.26	1.55	0.99	4.61	0.46	0.68	0.0186
hyp.2.10.2	1.34	1.16	2.43	1.03	1.15	1.31	0.29	1.59	0.16	0.17	0.0035
hyp.2.10.3	1.20	1.07	1.74	1.03	1.12	0.81	0.09	0.52	0.06	0.06	0.0011
hyp.3.10.1	1.04	1.04	3.21	1.01	1.02	0.95	0.42	3.65	0.31	0.39	0.0002
hyp.3.8.1	1.14	1.13	3.52	1.02	1.08	1.10	0.58	4.39	0.39	0.47	0.0018
hyp.20.4.1	1.02	1.03	8.82	1.00	1.02	0.93	0.64	2.35	0.18	0.61	0.0001
hyp.30.3.1	1.08	1.09	8.46	1.00	1.05	1.01	0.74	3.08	0.21	0.68	0.0011
hyp.50.3.1	1.03	1.04	8.46	1.00	1.02	0.94	0.69	3.08	0.19	0.67	0.0002
hyp.50.2.1	1.29	1.33	7.78	1.00	1.19	1.24	1.02	4.44	0.27	0.84	0.0178
hyp.100.2.1	1.14	1.16	7.78	1.00	1.10	1.04	0.85	4.44	0.24	0.77	0.0044
hyp.200.2.1	1.06	1.08	7.78	1.00	1.05	0.96	0.77	4.44	0.23	0.73	0.0011
dense.100	2.14	1.12	1.41	1.00	1.00	2.57	0.33	0.91	0.09	0.09	0.0201
dense.500	1.12	1.01	1.08	1.00	1.00	0.42	0.04	0.18	0.02	0.02	0.0008
random.1000.1000	2.10	2.18	4.21	1.88	2.13	3.27	2.89	14.60	2.26	2.54	0.3010
random.1000.100	1.48	1.46	4.00	1.29	1.28	1.77	1.10	6.37	0.74	0.73	0.0210
random.1000.10	1.28	1.11	1.49	1.09	1.08	1.10	0.16	0.91	0.09	0.09	0.0020
hb.fs5411	1.69	2.50	6.41	2.42	2.24	2.55	1.29	6.96	1.17	1.03	0.0498
hb.steam2	1.56	1.40	3.11	1.11	1.22	1.75	0.74	3.98	0.25	0.40	0.0149
hb.jpwh991	1.53	1.58	5.52	1.48	1.39	1.62	1.22	6.79	0.71	0.88	0.0362
hb.sherman2	1.47	1.34	3.79	1.27	1.27	1.55	0.62	3.91	0.27	0.42	0.0089
hb.lns3937	1.25	1.27	4.61	1.62	1.21	1.26	0.91	7.37	0.84	0.76	0.0085
hb.gemat11	1.23	1.24	4.30	1.28	1.18	1.38	0.95	7.64	0.58	0.80	0.0065
md.6000.20	1.21	1.25	5.78	1.19	1.18	1.11	0.91	6.70	0.80	0.80	0.0091
md.6000.10	1.14	1.11	2.83	1.07	1.07	1.09	0.43	3.27	0.34	0.34	0.0013
md.6000.8	1.11	1.08	2.04	1.05	1.05	0.97	0.24	1.80	0.18	0.18	0.0007
mdr.6000.10.2000	1.13	1.11	2.71	1.06	1.06	1.07	0.39	2.96	0.30	0.30	0.0012
mdr.6000.8.1000	1.11	1.07	1.91	1.05	1.04	0.95	0.21	1.61	0.16	0.16	0.0006

$$\phi_0(i) = \begin{cases} \lfloor \frac{i}{\ell_1} \rfloor, & \text{for } 0 \leq i < r\ell_1, \\ r + \lfloor \frac{i - r\ell_1}{\ell_0} \rfloor, & \text{for } r\ell_1 \leq i < n, \end{cases} \tag{19}$$

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

This distribution allocates rows in consecutive blocks to processors. The first r blocks contain $\lceil n/q_0 \rceil$ rows and the remaining $q_0 - r$ blocks contain $\lfloor n/q_0 \rfloor$ rows. Columns are allocated in a cyclic fashion. The block/grid distribution was proposed by Bisseling [4] as a suitable distribution for iterative linear system solvers. It has the property that the matrix diagonal is distributed over all the processors, so that the diagonal can easily be matched with a vector distribution. (The square grid distribution (6) does not have this advantage, because it distributes the diagonal over only $\sqrt{p}$ processors.)

The diagonal distribution is obtained by randomly distributing the matrix diagonal over the processors, with the constraint that each processor obtains $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ diagonal elements. This determines the distribution of the complete matrix, because Cartesian matrix distributions are fully determined by the distribution of their diagonal, see (4). The resulting distribution has a well balanced matrix diagonal and this is expected to lead to well balanced computation and communication. The diagonal distribution is equivalent to a distribution obtained by first performing a random symmetric permutation on the matrix, then assigning each diagonal element a_{ii}, $0 \leq i < n$, to a processor identifier $i \bmod p$, and after that translating this one-dimensional identifier into a two-dimensional one. (Hendrickson, Leland and Plimpton [15] propose to use a random symmetric permutation to preserve the matrix diagonal, which is often dense. They exploit this to overlap their vector transposition with computations related to the matrix diagonal.) The cost results given for the diagonal distribution and also for the PRAM and random/random distributions are the averages over 100 runs of the distribution program. The observed standard deviations are small and hence we consider the results to be reliable.

The results of Table 2 show that it is relatively easy to obtain a low normalised computation cost a. Taking $a \leq 2$ as our efficiency criterion, we may conclude that most distributions are efficient with respect to computation. The exception is the grid/grid distribution, which leads to excessive work loads on diagonal processors (s, s) in the fourth superstep, because these processors are the only ones that participate in this superstep. A breakdown of the total cost into the contributions of the separate supersteps confirms this analysis. Table 2 also shows that imposing suitable constraints on the distribution decreases a. The best random distribution with respect to computation is the diagonal distribution, which imposes an equal spreading of the matrix diagonal over the processors and hence causes a good load balance in the fourth superstep. The diagonal distribution clearly outperforms the other two random distributions, which are less restrictive. The PRAM distribution does not impose any constraints except for an identical distribution of matrix diagonal and vectors. The random/random distribution imposes Cartesianity and it also imposes an equal distribution of rows over processor sets $(s, *)$ and, independently, an equal distribution of columns over processor sets $(*, t)$, but it does not impose any connection between the row distribution ϕ_0 and the column distribution ϕ_1. The superiority of the diagonal distribution is due to the connection through the equi-distributed matrix

diagonal. (The inferiority of the grid/grid distribution is due to an overly restrictive connection, namely $\phi_0 = \phi_1$.) The block/grid distribution is a deterministic distribution with the good properties of the randomised diagonal distribution. The block/grid distribution may be somewhat better or worse than the diagonal distribution, depending on the sparsity structure of the matrix. For the hypercube matrices, the nonzeros are positioned along diagonals, and the block/grid distribution handles this diagonal structure particularly well.

The results of Table 2 also show that it is quite difficult to achieve a low normalised communication cost bg, if one cannot exploit any structural knowledge about the matrix. Taking $bg \leq 1$ as our efficiency criterion, we observe that using the best distribution, block/grid, on a BSP computer with a low value of $g = 10$, we can still solve only four problems efficiently, namely the relatively dense problems `hyp.2.10.3`, `dense.100`, `dense.500`, and `random.1000.10`. Most problems need a value of g in the range of 2–10. The results show that the block/grid distribution and the diagonal distribution are superior with respect to communication to the other three distributions. The reason for this is twofold. First, because they are Cartesian, they limit the number of processors that receive a particular input vector component to $q_0 - 1$. They limit the number of non-local partial sums received for each output vector component to $q_1 - 1$. Because the distributions are square, they limit both numbers to $\sqrt{p} - 1$. This compares favourably to the limit of $p - 1$ which holds for the PRAM distribution. Square Cartesian distributions may reduce communication by up to a factor of $\sqrt{p}/2$ compared to a pure row or column distribution, see the analysis of [4]. This is confirmed by additional experiments comparing the random/random distribution to a random row distribution with $q_0 = p$ and $q_1 = 1$, see [5]. Second, because the block/grid distribution and the diagonal distribution impose an equal distribution of the matrix diagonal and hence of the vectors, they lead to more balanced h-relations in the communication supersteps. This balance is much better than in the case of the grid/grid distribution, where the diagonal processors are the only processors that send data in the fan-out, and also the only ones that receive data in the fan-in. The gain in communication performance of the diagonal distribution compared to the grid/grid distribution is about a factor of $\sqrt{p}$. The block/grid distribution and the diagonal distribution perform equally well for problems that have a random nature, such as the `random`, `md`, and `mdr` matrices, and for dense problems. For problems that have some local structure that is reflected in the matrix, the block/grid distribution is sometimes able to discover part of this structure and to exploit it. This can be observed for the `hyp` matrices, `hb.steam2`, and `hb.sherman2`, which are all derived from multidimensional discrete grids. Obviously, the random nature of the diagonal distribution prevents discovery of any structure.

The normalised synchronisation cost cl of the sparse matrix-vector multiplication is low, because the multiplication algorithm has only four supersteps. The value of c is

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

Taking $cl \leq 1$ as our efficiency criterion, we note that problems with more than 200,000 nonzeros can be solved efficiently on a 100-processor BSP computer with $l \leq 1000$.

Table 3: Normalised communication cost bg for various distributions of two-dimensional hypercube matrices. The value of b is shown.

Distribution	block/grid	diag	100×1	50×2	25×4	10×10	tile
hyp.50.2.1	0.27	0.84	—	0.23	—	0.089	0.071
hyp.100.2.1	0.24	0.77	0.22	0.12	0.064	0.044	0.039
hyp.200.2.1	0.23	0.73	0.11	0.06	0.032	0.022	0.018

7 Results for structure dependent distributions

Table 3 shows the normalised communication cost bg for hypercube matrices of distance one and dimension two, distributed by various domain partitionings of the corresponding hypercube graph. These distributions exploit the structure of the matrix. For comparison, the table also includes the results from Table 2 for the best structure independent distributions, i.e. block/grid and diagonal. The $P \times Q$ distribution is obtained by splitting the first dimension of the hypercube graph into P subdomains and the second dimension into Q subdomains. For radix r, this partitions the hypercube graph into $PQ = p$ rectangular blocks, each with $r/P \times r/Q$ grid points. For the structure dependent distributions of these hypercube matrices, we choose $q_0 = p$ and $q_1 = 1$, because we found no advantage in using other values. The value of p is fixed at $p = 100$. The distribution of the grid points and hence of the corresponding vector components uniquely determines the distribution of the matrix.

The results of Table 3 show that it pays to exploit structure: orthogonally splitting each of the two dimensions of the domain reduces b by a factor of up to thirty, compared to the structure independent diagonal distribution. Furthermore, the results show that the lowest communication cost for separate dimension splitting is achieved if the resulting blocks are square. This is a surface-to-volume effect, because communication grows as the number of points near the surface of a block and computation as the number of points within the volume. Partitioning the domain into square blocks of size $r/\sqrt{p} \times r/\sqrt{p}$ reduces 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 `hyp.200.2.1`, when comparing the 100×1 distribution with the 10×10 distribution.

It is possible to improve the distribution further, by partitioning the domain along suitable lines, not necessarily parallel to the coordinate axes. For example, a two-dimensional toroidal grid can be split into *digital spheres* of the form

$$B_R(\mathbf{a}) = \{\mathbf{x} \in \mathbf{Z}^2 : ||\mathbf{x} - \mathbf{a}||_1 \leq R\}, \tag{22}$$

where the norm is defined by $||\mathbf{x}||_1 = |x_0| + |x_1|$. In other words, all grid points with a Manhattan distance of at most R to the centre $\mathbf{a}$ of such a sphere are allocated to the same processor. Figure 2 illustrates this distribution for an ideal case. The infinite grid $\mathbf{Z}^2$ can be partitioned by choosing all integer linear combinations of the vectors $(R+1, R)$ and $(-R, R+1)$ as centres of spheres of radius R. It can be shown that these spheres are mutually disjoint and that they cover the complete grid $\mathbf{Z}^2$. This partitioning of the infinite

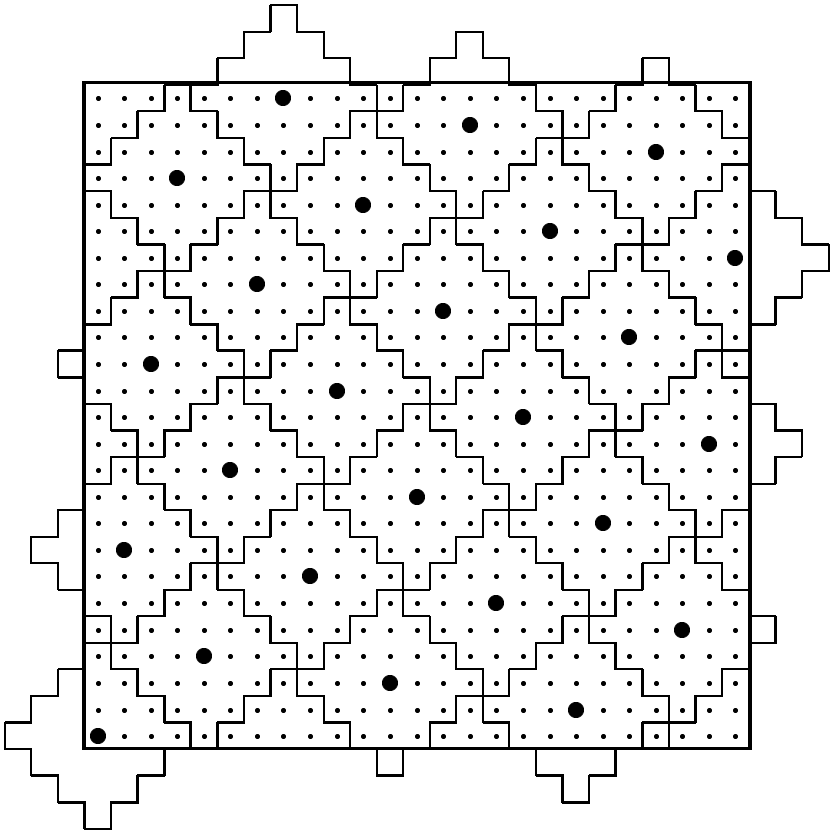

Figure 2: Partitioning of a 25×25 grid into 25 digital spheres of radius $R = 3$

grid $\mathbf{Z}^2$ can be used to partition finite subgrids. We call this procedure *tile partitioning*, because it originates in a tiling of the plane $\mathbf{R}^2$. The last column of Table 3 presents the communication cost for tile partitioning (or approximate tile partitioning, see below).

For the five-point Laplacian operator on a two-dimensional grid, tile partitioning reduces communication asymptotically by a factor of $\sqrt{2}$ compared to square block partitioning. This can be seen as follows. For each digital sphere, $2R + 1$ values from grid points to the left of the sphere must be received, $2R + 1$ values from points to the right, one from the grid point above the sphere, and one from the point below it. This means that $4R + 4$ values must be received, and, by a symmetry argument, that the same number of values must be sent. Therefore, a $(4R + 4)$-relation must be performed during the fan-out. (There is no fan-in, because $q_0 = p$ and $q_1 = 1$.) The number of points of the digital sphere is $N = 2R^2 + 2R + 1$. For each point, five floating point multiplications and four additions are needed. Therefore, $b = (4R + 4)/(18R^2 + 18R + 9) \approx 2\sqrt{2}/(9\sqrt{N})$ for tile partitioning. For a square block of $N = m^2$ points, a $(4m)$-relation must be performed. Therefore, $b = 4m/(9m^2) = 4/(9\sqrt{N})$ for square block partitioning. The value of b for the example of Fig. 2 is $b \approx 0.071$, which compares favourably to the value of $b \approx 0.089$ for the corresponding square block partitioning. The tile distribution over 100 processors of the matrix `hyp.50.2.1` in Table 3 has the same value $b = 0.071$ as the tile distribution over 25 processors of the 25×25 grid of Fig. 2. This can be explained by viewing a 50×50 grid

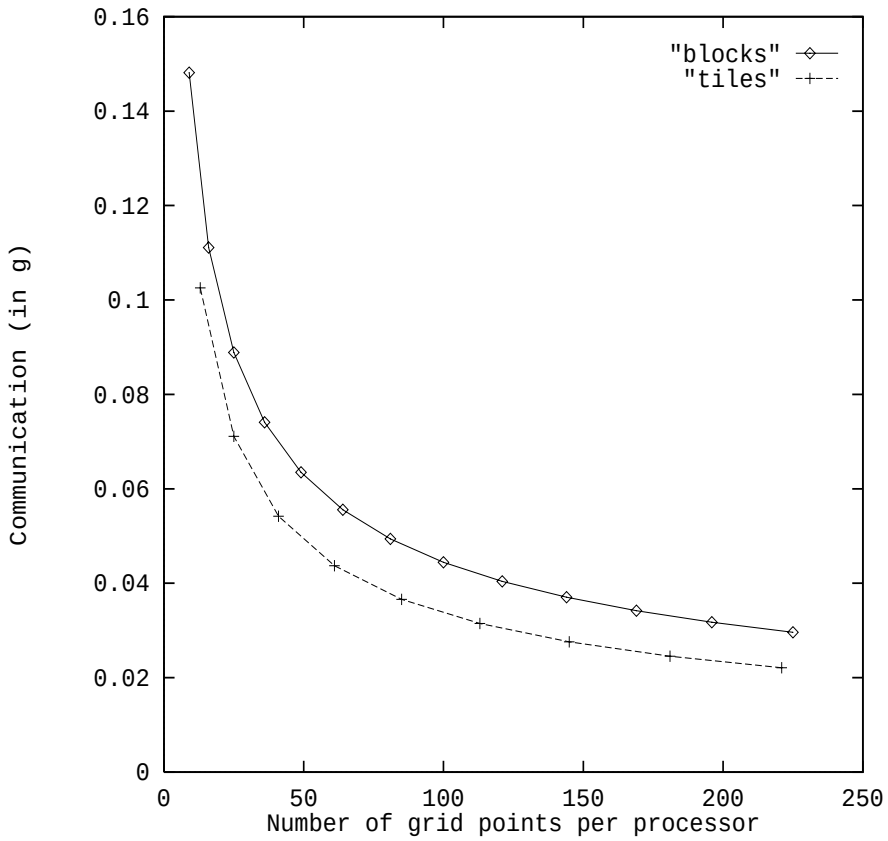

Figure 3: Normalised communication cost bg for tile distribution and square block distribution of two-dimensional hypercube matrices

as being composed of four subgrids of size 25×25, each tiled as in Fig. 2. Fig. 3 shows the normalised communication cost bg as a function of the number of grid points per processor N, for the tile and block distributions. Results are given for values of N that permit a perfect partitioning. The figure shows that the tile distribution is superior.

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 numbers of grid points per processor, i.e. for squares, 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 imperfect case, a good distribution can still be obtained by approximating a tile distribution. One way of doing this is by the following geometric method. First, the grid is split in square blocks and each block is cut along its diagonals; this produces four triangles per block. (Nearly square blocks are padded with dummy points to make them square.) After that, the grid points within a triangle are assigned to the nearest block boundary, with fair tie-breaking by using symmetry. Each boundary is identified with a processor. Two blocks with a common boundary each contribute one

triangle to the processor of that boundary. This procedure was used to obtain the tile results for `hyp.100.2.1` and `hyp.200.2.1` in Table 3. For example, the 100×100 grid is cut in 49 blocks, with boundaries of length 14 or 15. The largest block has 225 grid points. Splitting the blocks in four parts produces 196 triangles, with at most 57 grid points per triangle. Adjacent triangles of neighbouring blocks are combined and then assigned to a processor. This partitions the grid among 98 processors; each processor has at most 113 grid points. The normalised computation cost of this distribution is $a = 1.13$, for 100 processors. (Note that two processors are idling.) The normalised communication cost is reduced from $0.044g$ for square blocks to $0.039g$ for tiles, at the expense of an increase in computation cost due to load imbalance. For the 200×200 grid, computation cost increases by a factor of 1.05, whereas communication cost decreases by a factor of 1.27. In that case, tiling is already beneficial for $g \geq 11$.

The surface-to-volume ratio of cubic blocks in higher dimensions is worse than in two dimensions, for a fixed number N of grid points per block. This is because a larger fraction of the grid points lies at the boundary of the block. The normalised communication cost for cubic block partitioning of a d-dimensional grid with radix r is

$$b = \frac{2dp^{1/d}}{(4d+1)r} \approx \frac{N^{-1/d}}{2}. \tag{23}$$

This implies that in higher dimensions communication time may well dominate computation time on most parallel machines. The use of tiling techniques can reduce communication time by a small constant. Although small, this may be useful in achieving maximum efficiency for certain three-dimensional computations.

Table 4 shows the normalised 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 or equal to $r_c = 0.1$. The upper part of the table repeats the cost results from Table 2 for several structure independent distributions. The lower part of the table presents cost results for structure dependent distributions that assign particles on the basis of their position to rectangular subdomains and hence to processors. (In our discussion we ignore the symmetry of particle interactions, which may be used to reduce the amount of computation by a factor of two.)

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. Another approach is to distribute particles using geometric parallelism, see [10] for an extensive discussion. This leads to structure dependent distributions such as those in the lower part of the table. These distributions require less communication, but they are more susceptible to load imbalance caused by an inhomogeneous particle density.

Table 4 indicates that, among rectangular subdomains, cubic ones are the best with respect to communication. (It is possible to improve the distribution further by cutting the domain in a skewed fashion. This would lead to polyhedron-shaped subdomains instead

Table 4: Normalised BSP cost $a+bg+cl$ for various distributions of the molecular dynamics matrix `md.6000.10`

Distribution	q_0	q_1	a	b	c
PRAM			1.14	1.09	0.0013
random/random	10	10	1.11	0.43	0.0013
block/grid	10	10	1.07	0.34	0.0013
diagonal	10	10	1.07	0.34	0.0013
regions of size $0.01 \times 1.0 \times 1.0$	100	1	1.34	0.32	0.0007
regions of size $0.01 \times 1.0 \times 1.0$	10	10	1.28	0.26	0.0013
regions of size $0.1 \times 0.1 \times 1.0$	100	1	1.41	0.11	0.0007
regions of size $0.1 \times 0.1 \times 1.0$	10	10	1.41	0.08	0.0013
regions of size $0.2 \times 0.2 \times 0.25$	100	1	1.54	0.08	0.0007
regions of size $0.2 \times 0.2 \times 0.25$	10	10	1.54	0.09	0.0013

of rectangular blocks. This subject is beyond the scope of the present paper.) For cubic domain partitionings it is best to choose a row distribution of the matrix, i.e. $q_0 = p$ and $q_1 = 1$. For less favourable domain partitionings, choosing a square matrix distribution, i.e. $q_0 = q_1 = \sqrt{p}$, reduces communication. Note that the cut-off radius $r_c = 0.1$ of the matrix `md.6000.10` is quite large compared to the subdomain size. In this case, this implies that for regions of size $0.01 \times 1.0 \times 1.0$, particle information must be sent to 20 processors, so that it pays to aggregate information. For regions of size $0.1 \times 0.1 \times 1.0$, such information must be sent to 6–8 processors, depending on the positions of the particles, and for regions of size $0.2 \times 0.2 \times 0.25$, to 2–7 processors.

8 Conclusion

The BSP model provides a new foundation for the development of scalable parallel computing systems. It offers a robust framework within which we can unify the three main classes of currently available parallel computers: distributed memory architectures, shared memory multiprocessors, and networks of workstations. The model permits and encourages the development of parallel algorithms and programs which are efficient, 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 parallel architectures. The computations considered include the iterative solution of sparse linear systems, the solution of partial differential equations on a multidimensional grid, and molecular dynamics. The introduction of the normalised BSP cost $a + bg + cl$ enables a uniform analysis of these and other computations.

Our analysis shows that the exploitation of knowledge about the underlying structure of the problem is often crucial in achieving efficient parallel computations on a BSP computer. In this paper, we demonstrate that low-dimensional grid computations and molecular dy-

namics simulations are feasible on BSP computers with realistic values of the machine characteristics g and l. Therefore, the BSP computers that can be built in the foreseeable future will be able to solve structured problems from several important problem classes. Highly irregular scientific computing problems without a known structure are much more difficult to solve on BSP computers. For such problems, the block/grid distribution and the diagonal distribution perform better than others. Nevertheless, 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 even the matrix structure.

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. Almlöf, K. Faegri, Jr., and K. Korsell, "Principles for a direct SCF approach to LCAO-MO *ab-initio* calculations," *J. Comput. Chem.*, vol. 3, no. 3, pp. 385–399, 1982.

[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*. Philadelphia, PA: SIAM, 1992.

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadelphia, PA: SIAM, 1994.

[4] R. H. Bisseling, "Parallel iterative solution of sparse linear systems on a transputer network," in *Parallel Computation*, A. E. Fincham and B. Ford, Ed. Oxford, UK: Oxford University Press, 1993, pp. 253–271.

[5] R. H. Bisseling and W. F. McColl, "Scientific computing on bulk synchronous parallel architectures," Preprint 836, Dept. of Mathematics, Utrecht University, The Netherlands, Dec. 1993.

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

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

[8] I. S. Duff, R. G. Grimes, and J. G. Lewis, "Sparse matrix test problems," *ACM Trans. Math. Softw.*, vol. 15, no. 1, pp. 1–14, 1989.

[9] I. S. Duff, R. G. Grimes, and J. G. Lewis, "Users' guide for the Harwell-Boeing sparse matrix collection (Release I)," Technical Report RAL 92-086, Rutherford Appleton Laboratory, Oxfordshire, UK, Dec. 1992.

[10] 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.*, vol. 106, no. 1, pp. 101–107, 1993.

[11] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors: Volume I, General Techniques and Regular Problems*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[12] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *J. Parallel Distrib. Comput.*, vol. 22, no. 2, pp. 251–267, 1994.

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

[14] M. F. Guest, P. Sherwood, and J. H. van Lenthe, "Parallelism in computational chemistry. I. Hypercube-connected multicomputers," *Theor. Chim. Acta*, vol. 84, no. 4/5, pp. 423–441, 1993.

[15] B. Hendrickson, R. Leland, and S. Plimpton, "An efficient parallel algorithm for matrix-vector multiplication," Technical Report SAND 92-2765, Sandia National Laboratories, Albuquerque, NM, March 1993. To appear in *Int. J. High-Speed Comput.*

[16] B. Hendrickson and S. Plimpton, "Parallel many-body simulations without all-to-all communication," Technical Report SAND 92-2766, Sandia National Laboratories, Albuquerque, NM, March 1993. To appear in *J. Parallel Distrib. Comput.*

[17] B. A. Hendrickson and D. E. Womble, "The torus-wrap mapping for dense matrix calculations on massively parallel computers," *SIAM J. Sci. Comput.*, vol. 15, no. 5, pp. 1201–1226, 1994.

[18] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *J. Res. Nat. Bur. Stand.*, vol. 49, no. 6, pp. 409–436, 1952.

[19] S. L. Johnsson, "Communication efficient basic linear algebra computations on hypercube architectures," *J. Parallel Distrib. Comput.*, vol. 4, pp. 133–172, 1987.

[20] C. Lanczos, "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators," *J. Res. Nat. Bur. Stand.*, vol. 45, no. 4, pp. 255–282, 1950.

[21] W. F. McColl, "Special purpose parallel computing," in *Lectures on Parallel Computation*, A. M. Gibbons and P. Spirakis, Ed. Cambridge, UK: Cambridge University Press, 1993, pp. 261–336.

[22] W. F. McColl, "General purpose parallel computing," in *Lectures on Parallel Computation*, A. M. Gibbons and P. Spirakis, Ed. Cambridge, UK: Cambridge University Press, 1993, pp. 337–391.

[23] W. F. McColl, "An architecture independent programming model for scalable parallel computing," in *Portability and Performance for Parallel Processing*, T. Hey and J. Ferrante, Ed. Chichester, UK: John Wiley and Sons, 1994, pp. 43–69.

[24] A. T. Ogielski and W. Aiello, "Sparse matrix computations on parallel processor arrays," *SIAM J. Sci. Comput.*, vol. 14, no. 3, pp. 519–530, 1993.

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

[26] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[27] L. G. Valiant, "General purpose parallel architectures," in *Handbook of Theoretical Computer Science: Volume A, Algorithms and Complexity*. J. van Leeuwen, Ed. Amsterdam: Elsevier, 1990, pp. 943–971.

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