Programming Research Group

ALGORITHMIC AND PRACTICAL CONSIDERATIONS FOR DENSE MATRIX COMPUTATIONS ON THE BSP MODEL

Alexandros V. Gerbessiotis

PRG-TR-32-97

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Algorithmic and Practical Considerations for Dense Matrix Computations on the BSP Model*

Alexandros V. Gerbessiotis

Computing Laboratory,
Oxford University,
Wolfson Building,
Oxford, OX1 3QD,
UK.

October 1997

Abstract

The Bulk-Synchronous Parallel (BSP) model of computation has been proposed by L.G. Valiant as a unifying and bridging model for the design, analysis and implementation of parallel algorithms that are both scalable and portable, among diverse parallel machines. In this work we present direct BSP algorithms for dense matrix computations such as matrix multiplication, LU decomposition, determinant finding and matrix inversion. Two established distributions of an input matrix among the processors are examined and the particular theoretical and practical merits of each one are discussed.

As the BSP model is not just an abstract computational model, it can also be used as a programming paradigm for the implementation of parallel algorithms; towards this direction we implemented the aforementioned algorithms on top of the Oxford BSP Toolset, *BSPlib*, that provides library support for BSP programming. Our implementations were tested for scalability and portability on a variety of parallel systems that included the SGI Power Challenge, the IBM SP2 and the Cray T3D.

Our experimental results support our claims of efficiency, portability and reusability of the presented algorithms across the named hardware platforms, and reinforce our convictions that the BSP model can also be successfully used as a programming platform or, indeed, as a kind of programming paradigm.

*This work was supported in part by EPSRC(UK) under grant GR/K16999.

1 Introduction

The main reason for the moderate success of parallel computing has been the lack of a bridging and unifying model for parallel computation the way the von Neumann model has been for sequential computing [23]. The introduction of the Bulk-Synchronous Parallel (BSP) model of computation by L.G. Valiant [24, 25, 26] comes to address these limitations of parallel computing and presents itself as a possible bridge between parallel software and hardware.

1.1 The BSP Model

The BSP model, as described in [25], consists of three parts:

(1) a collection of *processor-memory components*,

(2) a *communication network* that can deliver messages point-to-point among the components, and

(3) a *facility for global synchronization*, in barrier style, of all or a subset of the components.

The parameters of such a machine are p, the number of processors/components, L, the synchronization periodicity, the minimal time between successive synchronization operations, and g the ratio of the total throughput of the whole system in terms of basic local computational operations to the throughput of the router in terms of words of information delivered.

The interconnection network, the router and the synchronizer set lower bounds on the minimal value that L can attain.

(1) A lower bound on the value of L is the time for a remote memory operation/message dispatch to become effective and is thus dependent on the properties of the router and the interconnection network.

(2) A lower bound on the effective value of L is also the time for barrier synchronization. It is erroneous and misleading to think that L is just the cost of barrier synchronization.

(3) Parameter L is also made large enough for the theoretical bound on g to be realizable.

The value of g is measured while the network is in a steady state of continuous message usage. In the BSP model, when we refer to time we use the abstract notion of a *time-step*, as opposed to a CPU instruction or cycle. As current microprocessors may execute more than one instructions at a time, we use the term time-step to refer to the time needed to perform an elementary local computational operation (such as a fetch from memory and a floating-point operation followed by a store operation, or a comparison and exchange). The definition of g in the BSP model relates to the routing of an h-relation, that is, the situation where each processor sends or receives at most h

messages; in practice this is equivalent to having each processor send or receive at most h words of information. Therefore, in the BSP model, g is the cost of communication so that an h-relation is realized within gh time steps, for any h such that $h \geq h_0$, where h_0 is a machine dependent parameter. Otherwise, the cost of communication is L. This way latency issues associated with small message size/messages are taken into consideration. In conclusion the cost of routing an h-relation is $\max\{L, gh\}$. L is thus not just the cost of barrier synchronization as latency issues associated with routing a small number of messages (or small-size messages) are taken into consideration in the value of parameter L. In practice, and in various libraries implementing the BSP model message requests issued/accepted by processors are of variable size. The h of an h-relation relates then to the total size of communicated data and g is expressed in terms of basic computational operations (sometimes, floating-point operations) or absolute time (seconds) per data-unit (a byte or word of information). If g is expressed in terms of basic computational operations it is naturally $g \geq 1$.

In practice, the parameter L of the BSP model not only hides the cost of communication when each processor sends/receives a small amount of information to/from few processors but also the cost of communication where each processor sends to/receives from a larger number of processors a small amount of information, or sends to/receives from few processors a larger amount of data.

In general, for any BSP computer, the values of L and g are likely to be non-decreasing functions of p.

Computation on the BSP model proceeds in a succession of *supersteps*. A superstep may be thought of as a segment of computation during which each processor performs a given task using data already available there locally before the start of the superstep. Such a task may include (i) local computations, (ii) message transmissions, and (iii) message receipts. Each superstep is charged $\max\{L, x + gh\}$ time steps, where x is the maximum number of basic computational operations executed by any processor during a superstep and h is the maximum number of messages transmitted or received by any processor. How the expression for charging combines the x with the h is not crucial if, as we do here [11], we assume that each superstep is pure computation or pure communication. In this way the two kinds of costs can be separated. We also use in this work the notion of a *stage*, as introduced in [11], which is defined to be a sequence of one or more supersteps performing a certain task. A stage should not be considered as part of the definition and description of the BSP model. Its existence facilitates the exposition of various algorithms only.

Under the BSP programming paradigm the objective is to minimize parallel computation and communication time, minimize the number of supersteps and maximize their size, and increase processor utilization.

1.2 Specifying the Performance of BSP Algorithms

Two modes of programming on the BSP model were envisaged in [25]: *automatic mode* where programs are written, say PRAM style, in a high level language that hides memory distribution from the user and *direct mode* where the programmer retains control of memory allocation.

In this paper we are primarily interested in the *direct mode* of programming since small multiplicative constant factors in runtime are important for matrix-related problems. We shall therefore emphasize multiplicative constant factors in this work. Although it is difficult to measure performance to high accuracy, since the operations that are counted have to be carefully defined, we can make such measures meaningful by measuring ratios between runtimes on pairs of models that have the same set of local instructions.

We specify the performance of a BSP algorithm $\mathcal{A}$ in three parts. First, we specify a sequential algorithm $\mathcal{A}^*$ with which we are comparing it and make explicit how we charge for the basic operations of both $\mathcal{A}$ and $\mathcal{A}^*$. Second, we specify two ratios π and μ. The former, π, is the ratio between the computation time $C_{\mathcal{A}}$, of the BSP algorithm, over the time $C_{\mathcal{A}^*}$ of the comparing sequential algorithm divided by p, i.e., $\pi = pC_{\mathcal{A}}/C_{\mathcal{A}^*}$. The latter, μ, is the ratio between the parallel time $M_{\mathcal{A}}$ required by the communication supersteps of the BSP algorithm and the computation time of $\mathcal{A}^*$ divided by p, i.e., $\mu = pM_{\mathcal{A}}/C_{\mathcal{A}^*}$. When we specify μ it is necessary to be explicit about the amount of information that can be conveyed in a single message. The term *relative work* w is defined as the sum $w = \pi + \mu$. For the expressions for π and μ to be concise and meaningful we sometimes consider upper bounds that thus also bound π and μ. We specify conditions on n, p, L and g that are sufficient for the algorithm to make sense and for the claimed bounds on π and μ to hold. Given these conditions and upper bounds on π and μ, a corresponding upper bound on w is also defined. It also implies a lower bound of p/w on the achieved speedup of the parallel computation. A lower bound on *efficiency* (as a percentage) of the parallel algorithm is thus $(100/w)\%$.

We also claim as corollaries some sufficient conditions for the most interesting optimality criteria, such as c-optimality, i.e., $\pi = c + o(1)$ and $\mu = o(1)$. All asymptotic bounds refer to the problem size as $n \to \infty$. The other parameters may also tend to ∞ if they are expressed in terms of n, though it is not assumed so.

We could get a more precise estimate of both the speedup and efficiency of a parallel algorithm and thus a more accurate estimate of its actual performance if we derived tighter upper bounds for π and μ (and consequently, for w) and closely matching lower bounds. Whereas for computation time and π this may be achievable in some simple instances, it is quite difficult to achieve for more complex algorithms whose expressions for computation time only become meaningful by eliminating (by taking upper bounds) terms that are deemed (asymptotically) insignificant. These terms may significantly influence the performance of the algorithm for small machine configurations and problem sizes. Such problems are exaggerated when expressions for μ are considered. The BSP charging policy for communication favors balanced communication. If unbalanced (skewed) communication takes place in the course of a parallel algorithm, the BSP cost for this communication is an upper bound of the actual communication. If a parallel algorithm is communication-bound or behaves as if it were communication-bound for small problem/processor sizes, then the gap between the observed and predicted (by the BSP cost model) efficiency can be quite significant.

It is also noted that in its current form, the BSP cost model ignores the effects of multiple memory hierarchies (caching), though it could be augmented to take them into consideration. For all these reasons, it is therefore application dependent whether good (asymptotic) algorithm performance prediction can lead to accurate parallel running-time estimation on real parallel machines.

The charging policy for the various operations is as follows. Each binary arithmetic operator over $\{+,-,*,/\}$ together with the necessary memory accesses is charged one time unit. One time unit is also charged for the binary *max* (maximum of two real numbers), and unary *abs* (absolute-value) operator along with their associated memory accesses.

The term slack in the context of algorithm design refers to the ratio of the problem size over the processor number of the BSP machine.

1.3 Contents of the paper

In this work we design and analyze new or previously known direct BSP algorithms for dense matrix computations such as matrix multiplication, LU decomposition, determinant finding and matrix inversion. Two established distributions of an input matrix among the processors are examined and the particular theoretical and practical merits of each one are discussed.

Particularly we first present the matrix multiplication algorithm described in [25]. Two matrices A and B of dimension $n \times n$ are distributed among the $p = p_i \cdot p_j$ processors of a BSP machine, for some integers p_i, p_j, so that each processor maintains a contiguous submatrix of dimension $n/p_i \times n/p_j$ of each matrix (in the subsequent discussion, we assume that $n/p_i, n/p_j$ are integer quantities). The algorithm is nevertheless memory inefficient. A memory efficient version of this algorithm, for the case $p_i = p_j = \sqrt{p}$, consists of $\sqrt{p}$ supersteps and achieves the same optimal (compared to the standard sequential algorithm) computation bound and communication time.

Algorithms for LU decomposition, determinant finding and matrix inversion are presented for two distributions of the input among the processors. One set of algorithms assumes a *scattered distribution* of the data among the processors to even out processor utilization [6]. Let $p = p_i \cdot p_j$ processors be divided into p_j groups each group P_j consisting of p_i processors. Particularly, $P_j = \{(j-1)p_i + l \ \mid 0 \leq l \leq p_i - 1\}$. Processor $(j-1)p_i + l$ is called the l-th processor of group P_j. This way matrix element (i,j), $0 \leq i,j < n$, is assigned to the $(i \bmod p_i)$-th processor of group $P_{(j \bmod p_j)}$. A scattered distribution refers to the special case $p_i = p_j = \sqrt{p}$. A second set of algorithms assumes a *wrapped-mapping* distribution of the input matrix [6, 16]. Wrapped mapping can be thought of as a special case of the distribution described above for $p_i = 1$ and $p_j = p$. Matrix algorithms when the input is under this distribution have been studied extensively [16]. All the presented BSP algorithms can be shown to be one-optimal for a wide range of the BSP parameters.

As the BSP model is not just an abstract programming model, it can also be used as a programming paradigm for the implementation of parallel algorithms; towards this direction we implemented the aforementioned algorithms on top of the Oxford BSP Toolset, *BSPlib*, that provides library support for BSP programming. Alternatively, one may use any other library that supports the simple communication and synchronization primitives required for programming on the BSP model. Our implementations are tested for scalability and portability on a variety of parallel systems that included the SGI Power Challenge, the IBM SP2 and the Cray T3D.

Although dense matrix algorithms based on the wrapped-mapping distribution of the input exhibit theoretical performance which is asymptotically, in many instances, worse than that of those assuming a scattered distribution, in practice, for machines with a small number of processors it exhibits performance as good as, and sometimes better, than scattered distribution based algorithms. The simplicity of programming algorithms based on a wrapped-mapping distribution of the input than on a scattered one is perhaps one other reason for its apparent superiority; locality of reference and the effectiveness and advantages of a caching hierarchy are also more effectively preserved and taken into advantage. For increasing processor configurations (say, $p > 32$) its drawbacks are becoming apparent.

Our experimental results support our claims of efficiency, portability and reusability of our algorithms across the named hardware platforms, and reinforce our conviction that algorithm design based on the BSP programming paradigm, can lead to programs that are scalable and portable among diverse hardware platforms. One can also estimate reliably (and depending on the algorithm, accurately) the performance of the algorithm implementation on the target platform and on other not readily available platforms, as long as the BSP parameters for these other platforms are only available, even when limitations due to the generality and general purposedness of the BSP model are scaled in.

As the objectives of algorithm design on the BSP are *scalability*, *portability* and *predictability of performance*, the algorithms we suggest may not lead to the best possible implementation on a specific platform. One may improve their performance by directly using specific features and interfaces of each machine. Our aim is to show that all three objectives can be realized without significant loss of performance. This is similar to the portability one obtains by programming in a high-level language as opposed to assembly.

Our intention with this work is not to present original ideas for new algorithms for dense-matrix problems – this is the context of numerical-analysis related research – but to show how existing algorithms can be parallelized in an easy, perhaps straightforward, way following the BSP programming paradigm, and programmed. In addition, we intend to show that algorithm performance can be analyzed in a general way within the BSP framework to allow reliable prediction, to the degree this is possible even for sequential machines, of their performance not only on the target platform but also on other not readily available ones — for such a task, the BSP parameters p, L and g for these platforms need only be available — even when limitations due to the generality and general purposedness of the BSP model are scaled in.

In the discussion to follow, when needed, we assume that $\sqrt{p}$ is an integer and that $n/\sqrt{p}$ is also integer. If condition $p = p_i \cdot p_j$, for some integers p_i and p_j is ever assumed to hold, then $n/p_i, n/p_j$ are also assumed to be integer quantities.

1.4 Primitive Operations

In the algorithms below, in various stages, a broadcasting operation involving p or $\sqrt{p}$ processors needs to be performed.

One can use the algorithm in [25, 11] to realize the broadcast of a single message in l supersteps, $0 < l \leq \lg p$, each superstep involving the replication of the message by a single processor $p^{1/l}$ times. If the size of the message to be broadcast is greater, traditional - folklore - pipelining techniques can be used to split the large message into smaller ones (the number of them equals the depth of the broadcasting tree to be used) and pipeline them through the tree of the former algorithm. Each processor sends or receives messages of total size at most twice the size of the message to be broadcast. Such ideas have been used in the derivation of one case of Lemma 1. Alternatively, for large-message broadcasts, one can use the following - folklore - algorithm. Split the large message into p smaller ones of equal size, send one such message to each processor, and then, each processor transmits the received message to every other processor. Again, each processor sends or receives messages of total size at most twice the size of the message to be broadcast. Such ideas are also used in the derivation of the other case of Lemma 1. This algorithm is also implied in the work by [1] and its first published reference in the context of the BSP model appeared in [20].

Lemma 1 ([12]) *There exists a p-processor BSP algorithm for broadcasting an n-word vector, that for any integer $2 \leq t \leq p$, requires time at most*[1]

$$M^n_{brd}(p) = 2\max\left\{L, g(p-1)\left\lceil\frac{n}{p}\right\rceil\right\} \cdot [n \geq p] + (2h-1)\max\{L, gt\lceil n/h\rceil\} \cdot [n < p],$$

for communication, where $h = \lceil \log_t((t-1)\lceil p/n\rceil + 1)\rceil - 1$. The overall time complexity of the algorithm is denoted by $T^n_{brd}(p) = M^n_{brd}(p)$.

The utilization of the algorithm implied by Lemma 1 can lead to the elimination of a multiplicative factor of p (if $l = 1$) or $p^{1/l}$ ($l > 1$) that would have appeared in the communication time related to broadcasting if the algorithms in [25, 11] were used instead.

2 Matrix Multiplication

We commence our examination of dense matrix algorithms with the problem of multiplying two $n \times n$ matrices A and B. The first BSP algorithm for this problem which we describe below was presented in the seminal work of [25]. It works for $p \leq n^2$. Each processor is assigned the task of computing an $n/\sqrt{p} \times n/\sqrt{p}$ submatrix of the product $A \times B$. The input matrices A and B are divided into p block-submatrices, each one of dimension $m \times m$, where $m = n/\sqrt{p}$. We call this distribution of the input among the processors *block* distribution. This way, element $A(i,j)$, $0 \leq i < n, 0 \leq j < n$, belongs to

[1]In general, if S is any statement that can be true or false, the bracketed notation [S] evaluates to 1 if S is true, and 0 otherwise.

the $(j/m) * \sqrt{p} + (i/m)$-th block that is subsequently assigned to the memory of the same-numbered processor. Let A_i (respectively, B_i) denote the i-th block of A (respectively, B) stored in processor i. With these conventions the algorithm in [25] can be described as follows.

```
begin MULT_A (C,A,B,n,p)
1.   Let m = n/√p ;
     Each processor is also assigned a unique processor number q;
2.   Let p_i = q mod √p ; p_j = q/√p ; C_q = 0;
3.   a_l ← A_{p_i + l*√p}, 0 ≤ l < √p;
4.   b_l ← B_{p_j*√p + l}, 0 ≤ l < √p;
5.   for 0 ≤ l < √p do
         C_q = C_q + a_l × b_l;
end MULT_A
```

Figure 1: Procedure MULT_A.

The following Proposition describes the performance of the aforementioned algorithm.

Proposition 1 *Algorithm* MULT_A *for multiplying two $n \times n$ matrices A and B stored according to the block distribution requires, for any $p \leq n^2$, computation time $C_{mul}(n)$ that is given by*

$$C_{mul}(n) = \max\{L, \frac{(2n-1)n^2}{p}\},$$

and communication time $M_{mul}(n)$ that is given by the expression

$$M_{mul}(n) = \max\{L, g\frac{2n^2}{\sqrt{p}}\}.$$

We shall compare the performance of a BSP matrix-multiplication algorithm to that of a standard sequential one which requires $(2n-1)\ n^2$ additions and multiplications. For a sufficient range of the BSP parameters algorithm MULT_A is one-optimal. This is summarized in the following theorem.

Theorem 1 *Algorithm* MULT_A, *for any $p \leq n^2$, has*

$$\pi \leq 1 + (Lp)/(2n^3),$$

and

$$\mu \leq g\sqrt{p}/n + (Lp)/(2n^3).$$

If $L = o(n^3/p)$ and $g = o(n/\sqrt{p})$, then $\pi = 1 + o(1)$ and $\mu = o(1)$ (one-optimality).

One immediately realizes that algorithm MULT_A is not memory efficient since it requires more local memory per processor – by a factor of $\sqrt{p}$ – than the required one.

begin MULT_B (C,A,B,n,p)
1. Let $m = n/\sqrt{p}$;
Each processor is also assigned a unique processor number q;
2. Let $p_i = q \bmod \sqrt{p}$; $p_j = q/\sqrt{p}$; $C_q = 0$;
3. **for** $0 \leq l < \sqrt{p}$ **do**
begin
4. $a \leftarrow A_{((p_i+p_j+l) \bmod \sqrt{p})*\sqrt{p}+p_i}$;
5. $b \leftarrow B_{((p_i+p_j+l) \bmod \sqrt{p})+p_j*\sqrt{p}}$;
6. $C_q = C_q + a \times b$;
end
end MULT_B

Figure 2: Procedure MULT_B.

Algorithm MULT_B shown in Figure 2 is the memory efficient variant of MULT_A. Its number of supersteps though, is increased by a factor of $\sqrt{p}$.

The difference between MULT_B and MULT_A is, whereas the latter one remotely reads all the submatrices it needs in one superstep, the former one reads one submatrix (from A, and one from B) at a time. The communication is performed in a round-robin like fashion (lines 4 and 5 in Figure 2) so that at each superstep each processor sends or receives at most $2n^2/p$ matrix elements, i.e. an $(2n^2/p)$-relation is routed. The performance of algorithm MULT_B is summarized in the following proposition.

Proposition 2 *Algorithm* MULT_B *for multiplying two $n \times n$ matrices A and B stored according to the block distribution requires, for any $p \leq n^2$, computation time $C_{mul}(n)$ that is given by*

$$C_{mul}(n) = \sqrt{p}\ \max\{L, \frac{(2n-1)n^2}{p^{3/2}}\}$$

and communication time $M_{mul}(n)$ that is given by the expression

$$M_{mul}(n) = \sqrt{p}\ \max\{L, g\frac{2n^2}{p}\}$$

For a sufficient range of the BSP parameters, algorithm MULT_B is one-optimal. This is summarized in the following theorem.

Theorem 2 *Algorithm* MULT_B*, for any $p \leq n^2$, has*

$$\pi \leq 1 + (Lp^{3/2})/(2n^3),$$

and

$$\mu \leq g\sqrt{p}/n + (Lp^{3/2})/(2n^3).$$

If $L = o(n^3/p^{3/2})$ and $g = o(n/\sqrt{p})$, then $\pi = 1 + o(1)$ and $\mu = o(1)$ (one-optimality).

3 LU Decomposition

The decomposition of an $n \times n$ dense matrix $A = [a_{ij}]$ into a lower-triangular matrix L and upper-triangular U, so that $A = L \times U$, is a method with a great deal of applications in numerical analysis, and parallel algorithms that efficiently decompose such a matrix A have been studied for a long time [16]. The sequential version of LU decomposition works in n phases, whereas in phase k the k-th column of matrix A is factored. For stability reasons, one may use partial pivoting, under which in phase k, row k is exchanged with that row, among rows $k, \ldots, n$, whose element of the k-th column has the maximum absolute value. For more details on LU decomposition we refer to any of standard numerical analysis books such as [13].

In this section we design and analyze on the BSP model two parallel dense LU decomposition algorithms. The two variants differ in the way data are distributed among the processors. They also complement the BSP algorithm for Gauss-Jordan elimination presented in [11] and Cholesky factorization [7, 8]. One algorithm assumes a *scattered distribution* of the data among the processors to even out processor utilization [6]. Let $p = p_i \cdot p_j$ processors are divided into p_j groups each group P_j consisting of p_i processors. Particularly, $P_j = \{(j-1)p_i + l \ \mid 0 \leq l \leq p_i - 1\}$. Processor $(j-1)p_i + l$ is called the l-th processor of group P_j. This way matrix element (i, j), $0 \leq i < n, 0 \leq j < n$ is assigned to the $(i \bmod p_i)$-th processor of group $P_{(j \bmod p_j)}$. A scattered distribution refers to the special case $p_i = p_j = \sqrt{p}$. The second algorithm assumes a *wrapped-mapping* distribution of the input matrix [6, 16]. Wrapped mapping can be thought of as a special case of the distribution described above for $p_i = 1$ and $p_j = p$. We shall compare the performance of the BSP LU decomposition algorithms, when partial pivoting is used, to that of sequential LU decomposition without pivoting, which requires $2n^3/3 + O(n^2)$ arithmetic operations [13].

3.1 Scattered distribution

A BSP algorithm for LU decomposition, based on a scattered distribution of the input matrix is similar to the one presented in [11] for Gauss-Jordan elimination and in [7, 8] for dense Cholesky factorization. In each phase a column k of the input matrix is decomposed, and the corresponding elements of L and U are stored in the memory locations previously occupied by the elements of column k of A.

Proposition 3 *There is a BSP algorithm for the LU decomposition of an $n \times n$ non-singular matrix A whose elements are stored on the p processors of a BSP machine using a scattered distribution, that for $p \leq n^2$ requires parallel time (computation time $C_{LU}(n)$ and communication time $M_{LU}(n)$) which is given by the following expression.*

$$\begin{aligned} C_{LU}(n) + M_{LU}(n) \quad &\leq \quad \frac{n(n-1)(2n-1)}{3p} + \frac{7n(n-1)}{2\sqrt{p}} + (5 + \sqrt{p})n \\ &+ \quad 5Ln + \frac{3n^2 g}{\sqrt{p}} + 2\sqrt{p}ng + 2png + 5ng. \end{aligned}$$

Consequently, the following theorem is derived.

Theorem 3 *There is a BSP algorithm for the LU decomposition of an $n \times n$ non-singular matrix A whose elements are stored on the p processors of a BSP machine using a scattered distribution, that for $p \leq n^2$ has*

$$w = \pi + \mu \leq 1 + \frac{15Lp}{2n^2} + \frac{21\sqrt{p}}{4n} + \frac{3(5+\sqrt{p})p}{2n^2} + \frac{9\sqrt{p}g}{2n} + \frac{3p^{3/2}g}{n^2} + \frac{3p^2 g}{n^2} + \frac{15pg}{2n^2}.$$

Proof (of Proposition 3): The p processors of the BSP machine are split into $\sqrt{p}$ groups P_j, $0 \leq j \leq \sqrt{p}-1$, of cardinality $\sqrt{p}$. The processors of group P_j are numbered as follows.

$$P_j = \{(j-1)\sqrt{p} + l \mid l = 0, \ldots, \sqrt{p}-1\}.$$

Processor $(j-1)\sqrt{p} + l$ of group P_j will be called the l-th processor of group P_j.

The elements of column i, $0 \leq i < n$, of A are assigned to group $P_{i \bmod \sqrt{p}}$ so that the j-th element of this column is assigned to the ($j \bmod \sqrt{p}$)-th processor of that group. Let $columnp(i)$ also identify the set of $\sqrt{p}$ processors allocated to store elements of column i. Each processor therefore gets n^2/p elements of A. Let $rowp(i)$ be the set of $\sqrt{p}$ processors that hold the elements of row i of A, and let $proc(a_{ij})$ be the processor that holds element a_{ij} or whatever element is stored in location (i,j) of A in subsequent iterations of the algorithm.

The algorithm is described in terms of operations performed in a succession of stages.

Procedure LU_S

begin
for $k = 1$ **to** n **do**
  begin
Stage 1: [lines 3-4 in Figure 3] A pivoting operation is performed. Each processor q in $columnp(k)$, finds the element a_{lk} with maximum absolute value such that $l \geq k$ and $proc(a_{lk}) = q$. Let e_q be that element and its row number be r_q. Processor q then communicates the pair (e_q, r_q) to processor $proc(a_{kk})$.
[**Analysis of Stage 1:** The first part of the pivoting operation involves the finding of a local maximum. At most $\sqrt{p}$ processors are involved (those of $columnp(k) = P_{k \bmod \sqrt{p}}$). The computation part requires time at most $\max\{L, \lceil (n-k)/\sqrt{p} \rceil\}$, whereas the communication part requires time at most $\max\{L, 2g\sqrt{p}\}$, if two memory words suffice for the communication of the pair (e_q, r_q).]

Stage 2: [lines 5-6 in Figure 3] Processor $proc(a_{kk})$ finds the maximum *pivot* among the received e_j values and broadcasts it and the row r to which *pivot* belongs, to every other processor, by replicating the message at the source.
[**Analysis of Stage 2:** The computation part requires time at most $\max\{L, \sqrt{p}\}$ whereas the communication part requires time at most $\max\{L, 2gp\}$.]

Stage 3: [line 7 in Figure 3] The processors in $rowp(k)$ and $rowp(r)$ exchange the elements of rows k and r they hold.

begin LU_S (A,n,p)
1. Each processor is assigned a unique processor number q;
   Let $p_i = q \bmod \sqrt{p}$; $p_j = q/\sqrt{p}$;
2. **for** $k = 0$ to $n-1$ **do**
   begin
3. for each processor q such that $p_j = (k \bmod \sqrt{p})$
   let $e_q = a_{lk}$, where $a_{lk} = \max_i |a_{ik}|$, $i \geq k, proc(a_{ik}) = q$; and
   let $r_q = l$;
4. Processor q communicates (e_q, r_q) to $proc(a_{kk})$;
5. If $q = proc(a_{kk})$ let $pivot = \max_i e_i$, $i = (k \bmod \sqrt{p})\sqrt{p} + l$, $0 \leq l < \sqrt{p}$;
6. Processor q broadcasts *pivot* as well as the row r to which
   the pivot element belongs, to every other processor;
7. The processors that hold elements of row k (those having $p_i = k \bmod \sqrt{p}$)
   and row r, exchange the elements of these rows;
8. Each processor that holds an element a_{kj}, $j \geq k$ of row k
   divides it by *pivot* and broadcasts it to the processors of group $P_{j \bmod \sqrt{p}}$;
   Simultaneously, each processor that holds element a_{ik}, $i \geq k$,
   broadcasts it to every processor such that $p_i = i \bmod \sqrt{p}$;
9. The decomposition step for column k is performed and matrix
   A is updated as required;
   end

end LU_S

Figure 3: Procedure LU_S.

[**Analysis of Stage 3:** A $(n/\sqrt{p})$-relation needs to be communicated in time at most $\max\{L, gn/\sqrt{p}\}$.]

Stage 4: [line 8 in Figure 3] Each processor in $rowp(k)$ divides each element a_{kj}, $j \geq k$, it holds by *pivot* and broadcasts it to the $columnp(j)$ processors of group $P_{j \bmod \sqrt{p}}$. Simultaneously, each processor in $columnp(k)$ broadcasts element a_{ik} to the $rowp(i)$ processors.
[**Analysis of Stage 4:** Time at most $\max\{L, \lceil (n-k)/\sqrt{p} \rceil\}$ is required for the computation part. The communication part requires time at most $2M_{brd}^{\lceil (n-k)/\sqrt{p} \rceil}(\sqrt{p})$.]

Stage 5: [line 9 in Figure 3] Column k is factored.
[**Analysis of Stage 5:** This step requires time at most $\max\{L, \lceil (n-k)/\sqrt{p} \rceil\}$.]

Stage 6: The lowest right $(n-k) \times (n-k)$ submatrix of A is updated as required.
[**Analysis of Stage 6:** Time at most $\max\{L, 2\lceil (n-k)/\sqrt{p} \rceil \lceil (n-k)/\sqrt{p} \rceil\}$ is required for this step.]

end

end

The parallel time of the algorithm is thus bounded above by the following quantity.

$$
\begin{aligned}
C(n)+M(n) &\leq \sum_{k=1}^{n}(3\lceil (n-k)/\sqrt{p}\rceil + \sqrt{p} + 2\lceil (n-k)/\sqrt{p}\rceil \lceil (n-k)/\sqrt{p}\rceil) \\
&+ \sum_{k=1}^{n}\left(2g\sqrt{p} + gp + 2g\lceil n/\sqrt{p}\rceil + 2M_{brd}^{\lceil (n-k)/\sqrt{p}\rceil}(\sqrt{p})\right) \\
&\leq \frac{n(n-1)(2n-1)}{3p} + \frac{7n(n-1)}{2\sqrt{p}} + (5+\sqrt{p})n + 5Ln \\
&+ \frac{3n^2 g}{\sqrt{p}} + 2\sqrt{p}ng + 2png + 5ng.
\end{aligned}
$$

■

3.2 Wrapped mapping distribution

A BSP algorithm for LU decomposition, based on a wrapped-mapping distribution of the input matrix is presented here and its structure is similar to the one presented in [8] for dense Cholesky factorization.

Proposition 4 *There is a BSP algorithm for the LU decomposition of an $n \times n$ non-singular matrix A whose elements are stored on the p processors of a BSP machine using a wrapped-mapping distribution, that for $p \leq n$ requires parallel time (computation time $C_{LU}(n)$ and communication time $M_{LU}(n)$).*

$$
C_{LU}(n) + M_{LU}(n) \leq \frac{n(n-1)(2n-1)}{3p} + \frac{n(n-1)}{2p} + 2n^2 + 4Ln + \frac{n^2 g}{2} + (2p+1)ng + \frac{n^2}{p}g.
$$

Consequently, the following theorem can be shown.

Theorem 4 *There is a BSP algorithm for the LU decomposition of an $n \times n$ non-singular matrix A whose elements are stored on the p processors of a BSP machine using a wrapped-mapping distribution, that for $p \leq n$ has*

$$
w = \pi + \mu \leq 1 + \frac{6Lp}{n^2} + \frac{3p}{n} + \frac{3}{4n} + \frac{3pg}{4n} + \frac{3pg}{2n^2} + \frac{3p^2 g}{n^2} + \frac{3g}{2n}.
$$

Proof (of Proposition 4): The p processors of the BSP machine are assigned at most $\lceil n/p \rceil$ columns each. Column i, $0 \leq i < n$, is assigned to processor ($i \bmod p$). The algorithm is described in terms of operations performed in a succession of stages. It is very similar to the previous one except for the simplification of the corresponding communication phases.

Procedure LU_W

begin

```
begin LU_W (A,n,p)
1.   Each processor is assigned a unique processor number q;
2.   for k = 0 to n − 1 do
       begin
3.     for each processor q such that q = (k mod p)
         let pivot = a_lk, where a_lk = max_i |a_ik|, i ≥ k; and
         let r = l;
4.     Processor q broadcasts (pivot, r) to every other processor;
5.     Processors exchange the elements of rows k and r;
6.     Each processor divides element a_kj, j ≥ k of row k by pivot;
       Simultaneously, processor (k mod p) broadcasts to every processor
       elements a_ik, i ≥ k;
7.     The decomposition step for column k is performed and matrix
       A is updated as required;
       end
end LU_W
```

Figure 4: Procedure LU_W.

for $k = 1$ **to** n **do**

begin

Stage 1: [line 3 in Figure 4] A pivoting operation is performed. Processor ($k \bmod p$) that holds column k finds the element a_{lk} with maximum absolute value such that $l \geq k$. Let *pivot* be that element and let r be its row number.
[**Analysis of Stage 1:** The first part of the pivoting operation involves the finding of the pivot element. The computation time required is at most $\max\{L, \lceil (n-k) \rceil\}$.]

Stage 2: [lines 4 in Figure 4] Processor ($k \bmod p$) broadcasts the pair ($pivot, r$) to every other processor, by replicating the message (pair) at the source.
[**Analysis of Stage 2:** This communication requires time at most $\max\{L, 2gp\}$.]

Stage 3: [line 5 in Figure 4] Processors exchange the elements of rows k and r they hold.
[**Analysis of Stage 3:** A (n/p)-relation needs to be communicated in time at most $\max\{L, gn/p\}$.]

Stage 4: [line 6 in Figure 4] Every processor divides each element a_{kj}, $j \geq k$, it holds by *pivot*. Simultaneously, processor ($k \bmod p$) broadcasts elements a_{ik}, $i \geq k$ to the remaining processors.
[**Analysis of Stage 4:** Time at most $\max\{L, \lceil (n-k)/p \rceil\}$ is required for computation and at most $2M_{brd}^{(n-k)}(p)$ for communication.]

Stage 5: [line 7 in Figure 4] Column k is factored.
[**Analysis of Stage 5:** This step requires time at most $\max\{L, n-k\}$.]

Stage 6: [line 7 in Figure 4] The lowest right $(n-k) \times (n-k)$ submatrix of A is updated as required.

[**Analysis of Stage 6:** This step requires time at most $\max\{L, 2(n-k)\lceil (n-k)/p \rceil\}$.
]

end

end

The parallel time of the algorithm can thus be shown to be bounded above by the following quantity.

$$\begin{aligned} C(n) + M(n) &\leq \frac{n(n-1)(2n-1)}{3p} + \frac{n(n-1)}{2p} + 2n^2 - n + 4Ln \\ &+ \frac{n(n-1)g}{2} + ng + 2png + \frac{n^2}{p} g. \end{aligned}$$

■

3.3 Determinant finding

Given the LU decomposition of a matrix, its determinant can be computed in additional $\max\{L, n\} + \max\{L, ng\}$ time. Each processor first sends any diagonal elements it has (by design, the diagonal elements of L are 1; those of U are the ones stored in A's diagonal memory locations) to, say, processor 0 where they are multiplied. Care should be taken that the sign of the determinant of the original matrix is adjusted properly. This is straightforwardly handled, by keeping track of the number of row exchanges during the course of the decomposition.

4 Matrix Inversion

The inverse of an $n \times n$ non-singular matrix $A = [a_{ij}]$ can be computed by a variety of methods one of which is LU decomposition followed by the solution of an appropriate system of linear equations through back-substitution [13]. An alternative way is to use a Gauss-Jordan elimination method. In this section we present a straightforward modification of the Gauss-Jordan elimination BSP algorithm of [11] that allows for in-place computation of the inverse of a matrix, when the input is distributed among the processors according to the scattered distribution. We also present, as a supplement, a variant of this algorithm under a wrapped-mapping distribution of the input. Both algorithms use a partial pivoting scheme and they are very similar to the ones for LU decomposition; the minor differences are in the size of the broadcast data from row and column k and the processing of the resultant matrix A.

4.1 Scattered distribution

A BSP algorithm for matrix inversion, based on a scattered distribution of the input matrix is identical (except for the way broadcasting is performed) to the one presented in [11] for Gauss-Jordan elimination. The algorithm works in phases, and in phase k, a column k of the input matrix is eliminated; the inverse of matrix A is gradually built in-place. We shall compare the performance of the BSP inversion algorithms base on Gauss-Jordan elimination with partial pivoting, to that of sequential inversion (based on the same method) without pivoting, which requires $2n^3 + O(n^2)$ arithmetic operations [13].

Proposition 5 *There is a BSP algorithm for the inversion of an $n \times n$ non-singular matrix A whose elements are stored on the p processors of a BSP machine using a scattered distribution, that for $p \leq n^2$ requires parallel time (computation time $C_{inv}(n)$ and communication time $M_{inv}(n)$) which is given by the following expressions.*

$$C_{inv}(n) + M_{inv}(n) \leq \frac{2n^3}{p} + \frac{13n^2}{2\sqrt{p}} + \sqrt{p}n + 5Ln + 6n + \frac{5n^2 g}{\sqrt{p}} + 2\sqrt{p}ng + 2png + 5ng.$$

Consequently, the following theorem can be proved.

Theorem 5 *There is a BSP algorithm for the inversion of an $n \times n$ non-singular matrix A whose elements are stored on the p processors of a BSP machine using a scattered distribution, that for $p \leq n^2$ has*

$$w = \pi + \mu \leq 1 + \frac{13\sqrt{p}}{4n} + \frac{5Lp}{2n^2} + \frac{p^{3/2}}{2n^2} + \frac{3p}{n^2} + \frac{5\sqrt{p}g}{2n} + \frac{p^{3/2}g}{n^2} + \frac{p^2 g}{n^2} + \frac{5pg}{2n^2}.$$

Proof: The terminology used in the proof of Proposition 3 is used here as well. The algorithm is described in terms of operations performed in a succession of stages.

Procedure INVERSE_S

begin
for $k = 1$ **to** n **do**
begin
Stage 1: [lines 3-4 in Figure 5] A pivoting operation is performed. Each processor q in *columnp*(k), finds element a_{lk} with maximum absolute value such that $l \geq k$ and $proc(a_{lk}) = q$. Let e_q be that element and its row number be r_q. Processor q then communicates the pair (e_q, r_q) to processor $proc(a_{kk})$.
[**Analysis of Stage 1:** The first part of the pivoting operation involves the finding of a local maximum. At most $\sqrt{p}$ processors are involved (those of $P_{k \bmod \sqrt{p}}$). The computation part requires time at most $\max\{L, \lceil (n-k)/\sqrt{p} \rceil\}$, whereas the communication part requires time at most $\max\{L, 2g\sqrt{p}\}$, if two memory words suffice for the communication of the pair (e_q, r_q).]

Stage 2: [lines 5-6 in Figure 5] Processor $proc(a_{kk})$ finds the maximum *pivot* among the received e_j values and broadcasts it and the row r to which *pivot* belongs, to every other processor, by replicating the message at the source.
[**Analysis of Stage 2:** The computation part requires time at most $\max\{L, \sqrt{p}\}$ whereas the communication part requires time at most $\max\{L, 2gp\}$.]

begin INVERSE_S (A,n,p)
1. Each processor is assigned a unique processor number q;
   Let $p_i = q \bmod \sqrt{p}$; $p_j = q/\sqrt{p}$;
2. **for** $k = 0$ to $n-1$ **do**
   begin
3. for each processor q such that $p_j = (k \bmod \sqrt{p})$
   let $e_q = a_{lk}$, where $a_{lk} = \max_i |a_{ik}|$, $i \geq k, proc(a_{ik}) = q$; and
   let $r_q = l$;
4. Processor q communicates (e_q, r_q) to $proc(a_{kk})$;
5. If $q = proc(a_{kk})$ let $pivot = \max_i e_i$, $i = (k \bmod \sqrt{p})\sqrt{p} + l$, $0 \leq l < \sqrt{p}$;
6. Processor q broadcasts $pivot$ as well as the row r to which
   the pivot element belongs, to every other processor;
7. The processors that hold elements of row k (those having $p_i = k \bmod \sqrt{p}$)
   and row r, exchange the elements of these rows;
8. Each processor that holds an element a_{kj}, $\forall j$, of row k
   broadcasts it to the processors of group $P_{j \bmod \sqrt{p}}$;
   Simultaneously, each processor that holds element a_{ik}, $\forall i$,
   broadcasts it to every other processor such that $p_i = i \bmod \sqrt{p}$;
9. Elements of row and column k are divided by $pivot$ and matrix
   A is updated as required;
   end

end INVERSE_S

Figure 5: Procedure INVERSE_S.

Stage 3: [line 7 in Figure 5] The processors in $rowp(k)$ and $rowp(r)$ exchange the elements of row k and r they hold.
[**Analysis of Stage 3:** A $(n/\sqrt{p})$-relation needs to be communicated in time at most $\max\{L, gn/\sqrt{p}\}$.]

Stage 4: [line 8 in Figure 5] Each processor in $rowp(k)$ for each element a_{kj}, $\forall j$, broadcasts it to the $columnp(j)$ processors. Simultaneously, each processor in $columnp(k)$ broadcasts element a_{ik}, $\forall i$, to the $rowp(i)$ processors.
[**Analysis of Stage 4:** The communication requires time at most $2M_{brd}^{\lceil n/\sqrt{p}\rceil}(\sqrt{p})$.]

Stage 5: [line 9 in Figure 5] Column and row k are factored.
[**Analysis of Stage 5:** This step requires time at most $\max\{L, 2\lceil n/\sqrt{p}\rceil\}$.]

Stage 6: Matrix A is updated as required (a Gauss-Jordan step modified for an inverse computation).
[**Analysis of Stage 6:** This step requires time at most $\max\{L, 2\lceil n/\sqrt{p}\rceil\lceil n/\sqrt{p}\rceil\}$.]
end
end
■

4.2 Wrapped mapping distribution

A BSP algorithm for matrix inversion, based on a wrapped-mapping distribution of the input matrix is presented here and its structure is similar to the one presented in [8] for dense Cholesky factorization and in this work for LU decomposition.

Proposition 6 *There is a BSP algorithm for the inversion of an $n \times n$ non-singular matrix A whose elements are stored on the p processors of a BSP machine using a wrapped-mapping distribution, that for $p \leq n$ requires parallel time (computation time $C_{inv}(n)$ and communication time $M_{inv}(n)$) which is given by the following expression.*

$$C_{LU}(n) + M_{LU}(n) \leq \frac{2n^3}{p} + \frac{7n^2}{2} + 4Ln + 2n^2 g + ng + 2png + \frac{n^2}{p} g.$$

Theorem 6 *There is a BSP algorithm for the LU decomposition of an $n \times n$ non-singular matrix A whose elements are stored on the p processors of a BSP machine using a scattered distribution, that for $p \leq n$ has*

$$w = \pi + \mu \leq 1 + \frac{2Lp}{n^2} + \frac{7p}{4n} + \frac{pg}{n} + \frac{pg}{2n^2} + \frac{p^2 g}{n^2} + \frac{g}{2n}.$$

Proof: The p processors of the BSP machine are assigned $\lceil n/p \rceil$ columns each. Column i, $0 \leq i < n$, is assigned to processor $(i \bmod p)$. The algorithm is described in terms of operations performed in a succession of stages. It is very similar to the previous one except for the simplification of the corresponding communication phases.

```
begin INVERSE_W (A,n,p)
1.  Each processor is assigned a unique processor number q;
2.  for k = 0 to n − 1 do
        begin
3.      for processor q such that q = (k mod p)
            let pivot = a_lk, where a_lk = max_i |a_ik|, i ≥ k; and
            let r = l;
4.      Processor q broadcasts (pivot, r) to every other processor;
5.      Processors exchange the elements of rows k and r;
6.      Each processor divides element a_ik, ∀i, of column k
        by pivot;
        Simultaneously, processor (k mod p) broadcasts to every processor
        elements a_ik, ∀i;
7.      Matrix A is updated as required;
        end
end INVERSE_W
```

Figure 6: Procedure INVERSE_W.

Procedure INVERSE_W

begin
for $k = 1$ **to** n **do**
  begin
Stage 1: [line 3 in Figure 6] A pivoting operation is performed. Processor $(k \bmod p)$ that holds column k finds the element a_{lk} with maximum absolute value such that $l \geq k$. Let *pivot* be that element and let r be its row number.
[**Analysis of Stage 1:** The first part of the pivoting operation involves the finding of the pivot element. The computation time required is at most $\max\{L, \lceil (n-k) \rceil\}$.]

Stage 2: [lines 4 in Figure 6] Processor $(k \bmod p)$ broadcasts the pair $(pivot, r)$ to every other processor, by replicating the message (pair) at the source.
[**Analysis of Stage 2:** This communication requires time at most $\max\{L, 2gp\}$.]

Stage 3: [line 5 in Figure 6] Processors exchange the elements of rows k and r they hold.
[**Analysis of Stage 3:** A (n/p)-relation needs to be communicated in time at most $\max\{L, gn/p\}$.]

Stage 4: [line 6 in Figure 6] Every processor divides each element a_{ik}, $\forall i$, it holds by *pivot*. Simultaneously, processor $(k \bmod p)$ broadcasts elements a_{ik}, $\forall i$ to the remaining processors.
[**Analysis of Stage 4:** The computation part requires time at most $\max\{L, \lceil n/p \rceil\}$. The communication part requires time at most $M^{n}_{brd}(p)$.]

Stage 5: [line 7 in Figure 6] The lowest right $(n-k) \times (n-k)$ submatrix of A is updated as required.
[**Analysis of Stage 5:** This step requires time at most $\max\{L, 2(n-k)\lceil (n-k)/p \rceil\}$.
]
  end
end■

5 Experimental Results

The BSP model, unlike other models of parallel computation, is not just an architectural-oriented theoretical model; it can also serve as a paradigm for programming parallel computers. The fundamental concept introduced by the BSP model is the notion of the superstep, and that all remote memory accesses occur between supersteps as part of a global operation among the processors; the results of these accesses become effective at the end of the current superstep.

The BSP model has been realized as a library of functions for process creation and destruction, remote memory access and message passing, and global, barrier-style, synchronization [15, 22, 14]. The abstraction offered by the BSP model is such that any library offering such facilities can be used for programming according to the BSP programming paradigm. The Oxford BSP Library [22], the Green BSP library [15] and the the Oxford BSP Toolset [14, 17] are some of the libraries that specifically allow programming according to the BSP paradigm.

In the following sections we present our implementations of the algorithms discussed earlier, as well as variations of them, on diverse hardware platforms; a shared-memory machine, a Silicon Graphics Power Challenge and two distributed-memory machines, an IBM SP2 system and a CRAY T3D. We extensively discuss below our experience and results obtained from our experiments. The commented source code of all these implementations can be found in the primary author's Web page [9]. In addition, one can find utility functions that allow for the distribution and collection of matrices among any number of processors for a variety of input-matrix distributions.

Our intention with this work is not to present original ideas for new algorithms for dense-matrix problems – this is the context of numerical-analysis related research – but to show how existing algorithms can be parallelized in an easy, perhaps straightforward way, following the BSP programming paradigm, and programmed. In addition their performance can be analyzed in a general way to allow reliable prediction, to the degree this is possible even for sequential machines, of their performance not only on the target platform but also on other not readily available ones — the BSP parameters p, L and g, for these platforms need only be available — even when limitations due to the generality and general purposedness of the BSP model are scaled in.

We establish that parallel algorithm design following the BSP paradigm can lead to a programming style which is easy to grasp and allows for the writing of programs that are scalable and transportable among diverse hardware platforms, without significant loss of performance. In our examples, the same source code developed in one of the three machines only needed to be recompiled to the other two ones, without any modifications. Towards this goal, in the appendix, we present the source code for the memory efficient matrix multiplication algorithm. The algorithm is no more complicated than its sequential counterpart. As hierarchical (main, primary cache, secondary cache) memory seems to be the norm for parallel, as well as sequential machines, some observations of ours on the effect of caching led to significant improvements in the performance of some of the presented algorithms.

As the objectives of algorithm design on the BSP model are *scalability*, *portability* and *predictability of performance*, the algorithms we suggest may not lead to the best possible implementation on a specific platform. One may improve their performance by directly using specific features and interfaces of each machine. Our aim is to show that all three objectives can be realized without significant loss of performance. This is similar to the portability one obtains by programming on a high-level language as opposed to assembly. Our results show that our parallel implementations are very close, in terms of efficiency, to our sequential ones, and perform as good or better that other similar parallel implementations As we would like to present this work as an introductory note on BSP programming emphasis is given on *clarity*, *relative efficiency*, *portability* and *predictability of performance*, over *absolute efficiency* on a specific platform.

5.1 Implementations

The implementations of the aforementioned algorithms presented here are written in standard ANSI C; only recompilation of the source code is required to execute it on the hardware platforms utilized. The available machines consisted of a four-processor Silicon Graphics Power Challenge shared-memory system, a four-processor IBM SP2 distributed-memory system and a 64-processor distributed-memory Cray T3D. The manufacturer-supplied C compiler (`cc`) is used through the *BSPlib* frontend, and the source-code is compiled with the `-O3` compiler option set. The effort required to learn the basics of *BSPlib* is minimal; one needs to understand the semantics of no more than 10-15 functions before starting riting a BSP program, and half of these function are straightforward and related to process creation, destruction and identification, and to access of the BSP parameters for a particular machine as modeled by the implementor of *BSPlib*.

In our implementations, array elements are of the `double` ANSI C data type. We chose the implementation of an array as an one-dimensional object. An $n \times n$ array A is presented in our code by an n^2-element one-dimensional ANSI C array a. Then element (i, j), $0 \le i < n, 0 \le j < n$, of A is represented by element $a[j * n + i]$ of a, thus implying a column-major storage of A into a.

Depending on the algorithm and the memory available on each platform, matrix sizes of 256, 512, 1024, 2048 and 2560 are used on one- and four-processor machine configurations on the SGI Power Challenge and the IBM SP2 machines. On the Cray T3D, one- (wherever possible), four-, 16- and 64-processor machine configurations are used.

The timing results obtained are discussed later in this section. The results depicted in the following tables are in general averages over at least four experiments. Timing is obtained through the use of the real-time clock function `bsp_time()` (it provides wall-clock time) of the BSP Toolset [14]. The matrices factored, multiplied or inverted are symmetric ones with no zero elements. Each of the table summarizing the obtained results shows for a given algorithm and for varying processor configurations the corresponding execution time on particular problem sizes. Besides timing results we also provide megaflop rates for the particular machine configuration on the particular implementation and problem size. Thus, the column labeled `Mfl` gives the *effective megaflop rate* (per processor) of the computation. The effective megaflop-rate for a particular matrix algorithm is the ratio of the total elapsed real-time over the number of floating-point operations per processor. This, of course, may underestimate the actual number of floating point operations performed. For the sake of an example, the effective megaflop-rate for the matrix multiplication algorithms is taken to be $2n^2(n-1)/p$. Timing results are truncated to one-digit precision except perhaps when the indicated time is less than one second; it may then appear that two instances of the same problem with the same time to have different effective megaflop rate.

Some interesting conclusions can be derived from these results. The SGI machine primarily, as well as the Cray T3D secondarily, exhibits, in some problem instances, superlinear speedup mainly because of caching; on the other hand, an increase in problem size beyond a certain threshold leads to a significant drop of performance. Given that the size of the secondary cache remains the same independent of machine configuration, an increase in the number of participating processors leads to an increase in the size (over all processors) of the primary (on-chip) cache with a subsequent increase of the effective megaflop rate (up to a certain threshold) of a particular machine. Matrix multiplication is the problem that takes full advantage of caching and gives the highest megaflop rate for any of our implementations; it is also the problem that suffers from a significant drop in performance when the effects of caching deteriorate with increasing problem (matrix) size. Given the column-major storage orientation of the matrices involved in the computations, it is significant that loops in the implementation programs be arranged so that they take advantage of this orientation. From our experience, this makes quite a difference in the performance of matrix multiplication on the SGI and the IBM SP2 and improves performance on the other implementations. Although peak performance of the SGI Power Challenge seems to be higher than those of the other two machines, the sustained megaflop performance of the IBM SP2, for the problem instances and the implementations examined, seems to be extremely good and higher than that of the SGI.

Finally in parts of the presentation below we shall require the BSP parameters of the machine configurations tested.

In an earlier version of this work these values were obtained from [18, 19]. Those values, however, were not reliable and in some cases significantly underestimated the true performance of the version of *BSPlib* used in these experiments. The underestimation of the BSP parameters coupled with the overestimation of communication time under the BSP cost model led to a good match between predicted (under the BSP cost model) and observed (by our experiments) running time. These problems have been reported in detail in [10]. In this work more representative values for the BSP parameters for the test machines when modeled as BSP computers are used. The SGI Power Challenge of the experiments is reported to have approximately $(p, L, g) = (4, 90\mu sec, 1.10\mu sec/double)$. For the IBM SP2, the corresponding values are $(p, L, g) = (4, 700\mu sec, 0.5\mu sec/double)$. The CRAY T3D is reported to behave as a BSP machine with sets of parameters $(p, L, g) = (4, 72\mu sec, 0.30\mu sec/double)$, $(16, 130\mu sec, 0.55\mu sec/double)$, and $(64, 364\mu sec, 0.85\mu sec/double)$ for the three configurations of our experiments. The units for L and g are converted from μsec to flops so that they can be used with our formula by multiplying each value with the effective Megaflop rate for $p = 1$ of the corresponding problem/algorithm studied.

5.2 Matrix Multiplication

We first implemented algorithms MULT_A and MULT_B. In the multiplication of two matrices, say A and B to form the product $A \times B$, the operations performed may be viewed as vector products of rows of A and columns of B. As, in our implementation, both A and B are stored in column-major form, one may improve performance if both matrices A and B are accessed this way; MULTT_A and MULTT_B, which are variants of MULT_A and MULT_B respectively, are implementing this feature by first transposing matrix A and then modifying the loop of matrix multiplication accordingly. As a consequence, the computation cost of algorithms MULTT_A and MULTT_B should include an extra cost-factor that reflects the cost of transposition of at most $n^2/\sqrt{p}$ matrix elements. As it is observed below, this cost is insignificant compared to the obtained gains in performance.

Algorithms MULT_A and MULT_B exhibit almost the same performance, and so do MULTT_A and MULTT_B. The $\sqrt{p}$ supersteps of MULT_B and MULTT_B are no imped-

Algorithm MULT_A				
	$p=1$		$p=4$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	0.37	88.5	0.11	72.2
512	3.0	88.0	0.84	79.2
1024	823.1	2.6	6.4	82.9
2048	8566.0	2.0	1702.0	2.5

Algorithm MULTT_A				
	$p=1$		$p=4$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	0.25	128.8	0.08	99.4
512	2.1	127.4	0.60	109.9
1024	66.3	32.3	4.57	117.2
2048	575.3	29.8	128.2	33.4

Table 1: Execution time for MULT_A and MULTT_A on the SGI Power Challenge

Algorithm MULT_B				
	$p=1$		$p=4$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	0.37	88.7	0.11	71.6
512	3.0	88.3	0.8	79.5
1024	799.5	2.6	6.4	83.0
2048	8566.0	2.0	1724.6	2.4

Algorithm MULTT_B				
	$p=1$		$p=4$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	0.25	128.9	0.08	101.7
512	2.1	127.5	0.60	110.8
1024	62.4	34.3	4.57	117.2
2048	575.3	29.8	128.0	33.5

Table 2: Execution time for MULT_B and MULTT_B on the SGI Power Challenge

iment to their performance compared to the benefits of efficient memory usage. We shall restrict our discussion below on MULT_B and MULTT_B; identical conclusions hold for the pair MULT_A and MULTT_A.

Tables 1 and 2 depict the results obtained on the SGI Power Challenge. For problem sizes $n = 256$ and $n = 512$ the effects of using transposition of the left-hand matrix in the multiplication is the most likely factor for the superiority of MULTT_B over MULT_B. For $n = 1024$ and $n = 2048$ and $p = 1$ the performance of MULT_B drops rapidly from an effective rate of 70-90Mfl to an effective rate of 2-3Mfl, whereas that of MULTT_B drops from a rate of 100-130Mfl to a sustainable and more reasonable rate of 30-35Mfl. Whereas MULTT_B possibly loses the benefits of one cache hierarchy, the pattern of memory accesses of MULT_B, which requires access to the rows of one array and the columns of the other one, apparently destroys locality both in primary and secondary cache levels.

For the $p = 4$ case, MULTT_B is consistently around 50% faster than MULT_B for $n = 256, 512$ and 1024. The drop in performance observed between $n = 512$ and $n = 1024$ for $p = 1$ is observed between $n = 1024$ and $n = 2048$ for $p = 4$. The only explanation is the effect of primary cache as in the four-processor machine it is four times larger, and its effects thus deteriorate for larger problem sizes on a four-processor machine than on a uniprocessor one. Full cache utilization seems to be achieved around $n = 1024$ as it is evident from the tables. For large problem sizes the Mfl rate for a parallel configuration is indeed higher than that of a uniprocessor one; highly parallelizable code thus may lead to superlinear speedup.

For small problem sizes, the observed efficiency of MULTT_B is 79 % and 88 % for $n = 256$ and $n = 512$ respectively, compared to the BSP model predicted one of 64 % and 78 % . The gap between predicted and observed efficiencies is mainly contributed to the caching effects that result in superlinear speedup for larger problem sizes. We also note that the effective value of g for the particular machine is four times larger than the asymptotic one observed when large messages are communicated in the network.

Tables 3 and 4 give the corresponding results obtained for the IBM-SP2. The memory efficient version of either MULT_B or MULTT_B are at least as good or better than MULT_A or MULTT_A respectively. Both MULTT_A and MULTT_B are consistently 5-15 times faster than MULT_A and MULT_B respectively. IBM SP2, for the problem ranges tested, exhibited a consistent performance in the range of 40-50Mfl for MULTT_A and MULTT_B. It seems that the IBM SP2 has a better, more consistent and sustainable cache hierarchy than the SGI Power Challenge. In addition, the effective megaflop rate for the implementations MULTT_A and MULTT_B increased with n (though, slowly) perhaps showing the strengths of the memory hierarchy of the IBM SP2; although the Power Challenge excels in peak performance, IBM SP2 is faster in sustained one. Similar observations have been made in [8]. The speedup observed for both MULTT_A and MULTT_B are in the range 3.3-3.8 and for both algorithms it is an increasing function of the examined problem size n. For small problem sizes, the observed efficiency of say, MULTT_B is 83 % and 91 % for $n = 256$ and $n = 512$ respectively, compared to the BSP model predicted one of 66 % and 80 %.

Algorithm MULT_A				
	$p = 1$		$p = 4$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	4.2	7.9	1.1	7.5
512	33.7	7.9	8.5	7.8
1024	598.2	3.5	67.6	7.9
2048	-	-	1214.3	3.5

Algorithm MULTT_A				
	$p = 1$		$p = 4$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	0.75	44.3	0.23	35.7
512	5.7	46.4	1.60	41.7
1024	45.3	47.3	11.8	45.2
2048	355.6	48.3	92.0	46.6

Table 3: Execution time for MULT_A and MULTT_A on the IBM SP2

Algorithm MULT_B				
	$p = 1$		$p = 4$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	4.2	7.9	1.1	7.5
512	33.6	7.9	8.5	7.8
1024	599.1	3.5	67.6	7.9
2048	-	-	1205.9	3.5

Algorithm MULTT_B				
	$p = 1$		$p = 4$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	0.76	43.9	0.20	36.3
512	5.7	46.4	1.50	42.1
1024	45.3	47.3	11.8	45.4
2048	355.6	48.3	91.8	47.3

Table 4: Execution time for MULT_B and MULTT_B on the IBM SP2

On the Cray T3D we present results on MULT_B and MULTT_B only, as MULT_A and MULTT_A exhibit no better performance. The corresponding timing results for the CRAY T3D are summarized in Tables 5 and 6. Conclusions similar to those drawn for the IBM SP2 (and the SGI Power Challenge) are drawn for this platform as well. The effective Mfl performance for MULTT_B is significantly higher than that of MULT_B (on the average, by a factor of 1.7). and the effective megaflop rate increases with increasing processor-pool size implying a contribution from the increased available cache memory. The obtained speedups of MULTT_B are high, efficiency never drops below 95% , and for $n = 256$ a speedup of 68 is attained on 64 processors; for $n = 1024$, the highest observed speedup of 86 is attained. The corresponding figures for MULT_B are lower than those of MULTT_B but still are pretty satisfactory.

In all cases, the obtained efficiency figures are better than the predicted lower bounds.

It seems that in all three machines, MULTT_B fully exploits the machine memory/caching hierarchies; the implementation, nevertheless, is highly portable, concise, and easy to grasp and comprehend.

We present the source code for MULTT_B in the appendix to also support our claims earlier that BSP programming allows for programs that are portable and scalable; in addition, they happen to be quite efficient compared to their corresponding sequential counterparts.

Algorithm MULT_B								
	$p = 1$		$p = 4$		$p = 16$		$p = 64$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	4.1	7.9	1.1	7.8	0.28	7.4	0.03	13.9
512	34.0	7.8	8.4	7.9	2.1	7.7	0.56	7.4
1024	289.8	7.4	68.4	7.8	16.9	7.9	4.3	7.7
2048	-	-	-	-	136.8	7.8	33.8	7.9

Table 5: Execution time for MULT_B on the Cray T3D

Algorithm MULTT_B								
	$p = 1$		$p = 4$		$p = 16$		$p = 64$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	2.3	14.3	0.58	14.4	0.15	13.7	0.03	15.1
512	20.7	12.9	4.7	14.1	1.16	14.4	0.30	13.5
1024	202.7	10.5	41.7	12.8	9.4	14.1	2.3	14.3
2048	-	-	-	-	83.5	12.8	19.0	14.1

Table 6: Execution time for MULTT_B on the Cray T3D

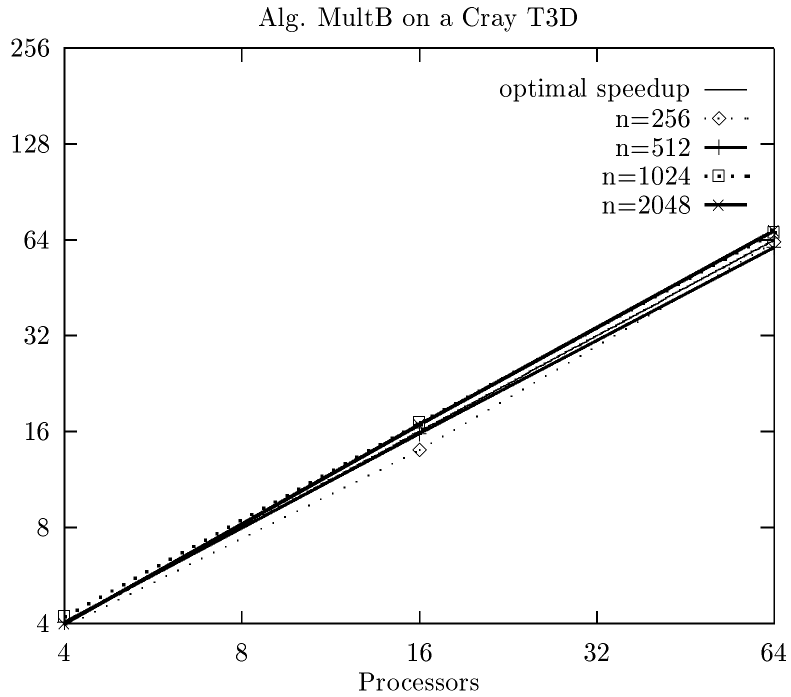

Figure 7: Speedup for MULT_B on the Cray T3D.

5.3 LU decomposition

We first implemented a generalization of algorithm LU_S where the processor geometry expressed in terms of p_i and p_j is variable (note that $p = p_i \cdot p_j$). This way LU_S becomes a special case when $p_i = p_j = \sqrt{p}$ and LU_W a special case when $p_i = 1$ and $p_j = p$. We also separately programmed a direct implementation of LU_W to examine its possible efficiency over the more general previous implementation. It is possible to run our implementations when partial pivoting is enabled or not. Our experiments included running algorithm LU_S for $(p_i, p_j) = (1,1)$, $(p_i, p_j) = (2,2)$ and $(p_i, p_j) = (1,4)$, and algorithm LU_W, with partial pivoting both enabled and disabled. This way we could observe the contribution of pivoting related operations to the total running time. We test our implementations on two non-singular matrices with non-zero elements. When partial pivoting is enabled the number of row exchanges is about $n/2$, where n is the dimension of the input matrix. With pivoting disabled, execution time of LU_S with $(p_i, p_j) = (1,4)$ is faster than that of LU_W; with pivoting enabled it is the other-way around. In the tables below we provide the faster timing results and thus columns indicating the LU_W variant may report results obtained either using the direct implementation of LU_W or the instance of the generalized implementation of LU_S for $p_i = 1$ and $p_j = p$. In most cases such differences are insignificant (less than 2-4%).

From its algorithm description, LU_W resembles the sequential implementation (minus the broadcasting operation), and the reported time for $p = 1$ for both LU_W and LU_S are comparable to those of a direct sequential implementation; therefore, for con-

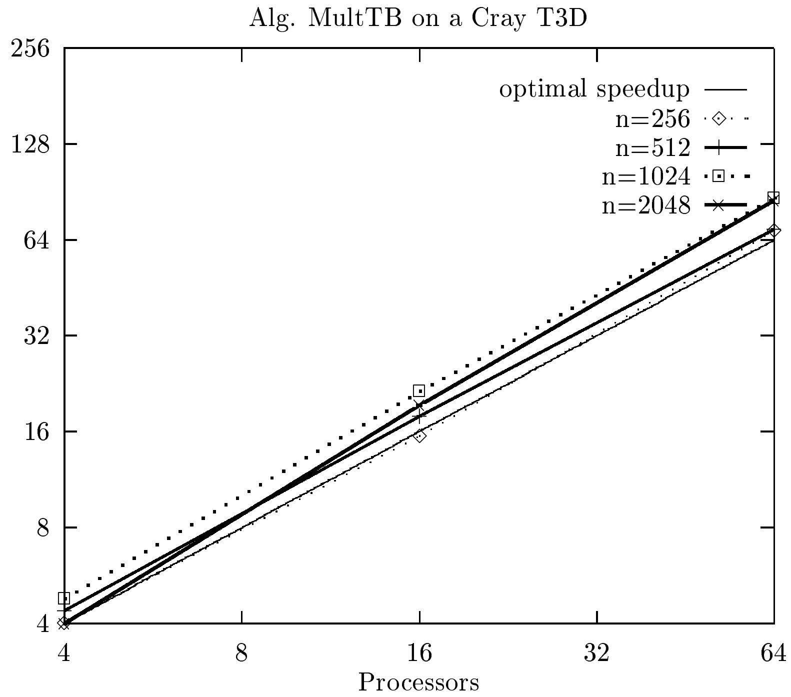

Figure 8: Speedup for MULTT_B on the Cray T3D.

venience in the compilation of our experimental results, the reported $p = 1$ time reflects that obtained from the LU_S implementation.

The effective megaflop rate for LU decomposition used in our experiments is taken to be $(2/3)n^3/p + O(n^2/p)$.

Table 7 presents the results obtained on the SGI Power Challenge with pivoting both disabled and enabled. The first two columns give the time and effective megaflop rate of our sequential implementation ($p = 1$), which, by the previous discussion, is the time for LU_S, $(p_i, p_j) = (1, 1)$, the following two columns provide the results on four processors of the scattered distribution implementation (LU_S) and the following two columns those obtained for the wrapped mapping distribution (the best results between LU_S, $(p_i, p_j) = (1, 4)$, and LU_W). These columns are followed by similar ones that depict performance with pivoting enabled. Table 8 contains timing results on the IBM SP2.

With reference to Table 7 we note that the effective peak megaflop performance of LU decomposition is nowhere as high as it is for matrix multiplication. Nevertheless, it sustained a high performance in the range of 19-25Mfl, about 60% of the sustained performance of MULTT_A and MULTT_B. Its peak is observed, as it is also the case for matrix multiplication, for the $n = 1024$ problem size. Due to the effects of caching, efficiency [16] is high, and in many instances superlinear speedup is achieved.

The *efficiency* of an implementation is computed according to its definition in [16]; first the speedup is obtained by dividing the $p = 1$ time over the $p = 4$ one and then

	No Pivoting						With Pivoting					
	$p = 1$		$p = 4$				$p = 1$		$p = 4$			
	LU_S		LU_S		LU_W		LU_S		LU_S		LU_W	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate
512	3.5	25.4	1.0	21.1	1.0	21.5	3.8	23.5	1.2	18.2	1.0	21.3
1024	33.3	21.5	7.4	24.2	7.4	24.0	35.9	19.9	8.1	22.0	7.5	23.8
2048	295.2	19.4	72.4	19.8	72.4	19.8	306.9	18.6	78.0	18.4	75.6	19.0
2560	571.0	19.6	145.7	19.2	145.5	19.2	588.2	19.0	152.1	18.4	147.7	19.0

Table 7: Execution time for LU_S and LU_W on the SGI Power Challenge

expressing the speedup as a percentage of the optimal one. Despite the large number of row exchanges (and thus, communication) involved in pivoting operations, the contribution of such operations to the overall execution time is kept small (in the range of 5-10% for most cases).

The case of the IBM SP2 experiments is more interesting to examine as they do not suffer from the perhaps inconsistent cache behavior of the SGI Power Challenge. The effective megaflop rate for LU decomposition is also consistently kept at around 60% of that of matrix multiplication and also kept growing for increasing matrix sizes. The predictability of the BSP cost model finds a strong advocate on the IBM SP2. Table 9 gives the predicted and observed efficiency for LU_S and LU_W when pivoting is enabled. The predicted efficiency is a lower bound derived from the upper bounds on w derived in Theorems 3 and 4 and the BSP values reported earlier for the particular configuration of the IBM SP2 used in our experiments. We note that the observed efficiency is in many instances very close to the lower bound (predicted efficiency). Only for problem size $n = 512$ is predicted efficiency higher than the observed one. This is mainly attributed to the computational overhead for small problem sizes that is not scalable and is insignificant for larger sizes. We also point out that the gap between predicted and observed efficiencies is no more than four percentage points.

	No Pivoting						With Pivoting					
	$p = 1$		$p = 4$				$p = 1$		$p = 4$			
	LU_S		LU_S		LU_W		LU_S		LU_S		LU_W	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate
512	2.9	30.1	1.2	17.6	1.3	17.3	3.4	26.3	1.9	11.9	1.4	16.0
1024	23.7	30.1	7.1	25.3	7.2	24.8	25.9	27.7	8.8	20.3	7.5	23.9
2048	186.8	30.6	50.4	28.4	50.7	28.3	195.2	29.3	56.2	25.5	52.3	27.4
2560	365.2	30.6	96.5	29.0	97.1	28.8	378.5	29.5	105.2	26.6	99.1	28.3

Table 8: Execution time for LU_S and LU_W on the IBM SP2

Table 10 and 11 summarize the results obtained on the Cray T3D. The former table shows the performance of LU_S and LU_W without pivoting and the latter one shows the performance of these two algorithms when pivoting is enabled. It should be pointed out, however, that in the latter case comparison of the obtained values cannot be drawn as for the same problem size but varying number of processors a different number of pivoting-related row exchanges are performed in LU_S. For the sake of an example, the number of exchanges with $n = 1024$ varied from 521 ($p = 1$) to 877 ($p = 64$)

n	LU_S		LU_W	
	Observed %	Predicted %	Observed %	Predicted %
512	44	31	57	37
1024	73	62	84	69
2048	86	85	92	89
2560	89	88	94	92

Table 9: Observed and Predicted (lower bound) Efficiency on an IBM SP2

in the test input matrix. We thus do not comment on these results except to note that performance is increasing with problem size as overhead becomes insignificant; this occurs for $n = 2048$, where efficiency is higher than 70%. Both the theoretical analysis as well as the experiments confirmed that LU_S yields better performance than LU_W. Table 12 shows the observed and predicted efficiencies for LU decomposition with pivoting enabled on the Cray T3D. Despite the discrepancies caused by the aforementioned row exchanges, the theoretically obtained (worst case) values reflect the actual behavior of the implementations. Only in one case $p = 16, n = 512$ is the predicted efficiency higher than the observed one by two percentage points. The larger the slack (n/p) the smaller the gap between predicted and observed efficiency is as the BSP cost model overestimate of the skewed communication involved becomes a less contributing factor. When no row interchanges took place during pivoting operations, the observed and predicted efficiencies also conformed to the theoretical predictions in all cases. The Mfl rate for LU_S (with or without pivoting) is increasing for increasing n throughout all the experiments. In addition, for the former case, the asymptotic rate is relatively high, even when compared to that of matrix multiplication, despite differences in caching patterns.

		$p = 1$		$p = 4$		$p = 16$		$p = 64$	
n	*Algorithm*	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate
512	LU_S	7.82	11.4	1.98	11.4	0.63	9.0	0.68	2.1
	LU_W			2.11	10.7	0.80	7.5	1.1	1.6
1024	LU_S	75.3	9.5	16.0	11.2	4.0	11.2	2.0	5.6
	LU_W			19.0	9.4	5.4	8.5	3.2	4.0
2048	LU_S	605.6	9.4	150.0	9.5	33.0	10.9	9.7	9.3
	LU_W			152.1	9.4	39.7	9.2	14.2	6.8
2560	LU_S	1171.	9.5	274.1	10.2	63.4	11.0	17.5	10.0
	LU_W			293.6	9.5	77.7	9.0	27.8	6.7

Table 10: Execution time for LU_S and LU_W on Cray T3D

Finally, we compare the performance of our dense LU decomposition implementations to another one that can be found in [3] which was also written on top of *BSPlib*. A comparison of performance on both the SGI Power Challenge and the IBM SP2 for $p = 4$

n	*Algorithm*	$p=1$ Time (sec)	$p=1$ Mfl rate	$p=4$ Time (sec)	$p=4$ Mfl rate	$p=16$ Time (sec)	$p=16$ Mfl rate	$p=64$ Time (sec)	$p=64$ Mfl rate
512	LU_S	9.0	9.9	2.5	8.7	1.1	5.2	1.07	1.3
	LU_W			2.2	10.2	0.90	6.6	1.5	1.2
1024	LU_S	81.4	8.8	18.7	9.5	5.7	7.9	3.2	3.5
	LU_W			19.3	9.3	5.7	8.2	4.2	3.1
2048	LU_S	643.3	8.9	162.7	8.8	40.0	9.0	13.8	6.5
	LU_W			152.8	9.4	40.6	9.0	17.5	5.5
2560	LU_S	–	–	298.3	9.3	74.9	9.3	24.1	7.3
	LU_W			294.8	9.5	77.1	9.2	30.5	6.1

Table 11: Execution time for LU_S and LU_W with pivoting on Cray T3D

n	$p=4$ Observed %	$p=4$ Predicted	$p=16$ Observed %	$p=16$ Predicted %	$p=64$ Observed %	$p=64$ Predicted %
512	87	86	52	53	14	12
1024	100	95	90	79	40	32
2048	99	97	100	92	73	62
2560	-	98	99	94	77	71

Table 12: Observed and Predicted (lower bound) efficiency for LU decomposition on a Cray T3D with partial pivoting

follows in Table 13.

5.4 Matrix inversion

With respect to the implementation of matrix inversion the same methodology is followed with LU decomposition. The same test matrices are also used in the experiments.

The effective megaflop rate for matrix inversion is taken to be $2n^3/p + O(n^2/p)$.

Table 14 presents the results obtained on the SGI Power Challenge with pivoting enabled nad disabled. The first two columns give the time and effective megaflop rate of our sequential implementation ($p = 1$), which is the time for INVERSE_S, $(p_i, p_j) = (1, 1)$, the following two columns give the corresponding results on four processors of the scattered distribution implementation (INVERSE_S) and the following two columns those obtained for the wrapped mapping distribution (the best results between INVERSE_S, $(p_i, p_j) = (1, 4)$, and INVERSE_W). A similar table, containing the results on the IBM SP2 follows it.

With reference to Table 14 we note that the effective peak megaflop performance of matrix inversion are comparable to that for LU decomposition. The same patterns in the variation of that rate are also observed (a peak at $n = 1024$). Due to the effects of caching, efficiency [16] is high (in most cases at least 90%), and in many instances

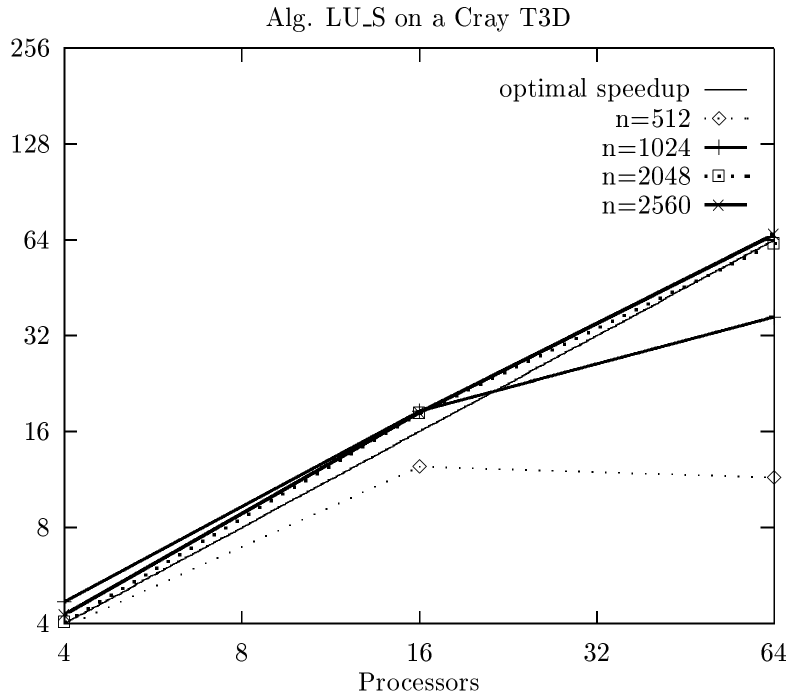

Figure 9: Speedup for LU_S (no pivoting) on the Cray T3D.

superlinear speedup is achieved. Efficiency is increasing as a function of matrix size up to $n = 1024$, where superlinear speedup is observed for the various test cases. After that point, with the decreasing contribution of caching, efficiency started to drop. Despite the large number of row exchanges (around $n/2$) involved in pivoting operations, the contribution of such operations to the overall execution time is kept small (in the range of 5-10% for most cases).

The case of the IBM SP2 experiments is more interesting to examine. In Table 15, the effective megaflop rate for matrix inversion is consistently at around 60% of that of matrix multiplication, is a bit higher than that of LU decomposition, and also kept growing for increasing matrix sizes. For large matrix sizes, efficiency is high. The observed efficiency in our experiments matched that predicted by the BSP model, as this is also

n	SGI		IBM SP2	
	Ours sec	[3] sec	Ours sec	[3] sec
512	1.2	2.8	1.9	4.2
1024	8.1	15.0	8.8	19.5
2048	78.0	112.5	56.2	100.6
2560	152.1	215.2	105.2	175.9

Table 13: Comparison with other dense LU decomposition algorithms

the case for the LU decomposition. Table 16 gives the predicted and observed efficiency for INVERSE_S and INVERSE_W when pivoting is enabled. The predicted efficiency is a lower bound derived from the upper bounds on w derived in Theorems 5 and 6 and the BSP values reported earlier for the particular configuration of the IBM SP2 used in our experiments. As in the case of LU decomposition, it shows how the BSP model can be effectively used in predicting algorithm performance from the analysis of an algorithm and the usage of the BSP parameters for a particular architecture/machine configuration.

	No Pivoting						*With Pivoting*					
	$p = 1$		$p = 4$				$p = 1$		$p = 4$			
	INVERSE_S		INVERSE_S		INVERSE_W		INVERSE_S		INVERSE_S		INVERSE_W	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate
512	10.4	25.8	2.8	24.2	2.8	23.9	10.7	25.2	3.0	22.3	2.9	22.9
1024	107.0	20.1	21.6	24.9	21.8	24.7	108.0	19.7	22.4	24.1	21.9	24.6
2048	882.2	19.5	236.0	18.2	236.2	18.2	898.2	19.1	240.6	17.9	239.5	17.9
2560	1726.	19.4	478.3	17.5	477.4	17.6	1855.	18.1	483.9	17.3	479.7	17.5

Table 14: Execution time for INVERSE_S and INVERSE_W on the SGI Power Challenge

	No Pivoting						*With Pivoting*					
	$p = 1$		$p = 4$				$p = 1$		$p = 4$			
	INVERSE_S		INVERSE_S		INVERSE_W		INVERSE_S		INVERSE_S		INVERSE_W	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate
512	8.9	30.2	2.7	24.3	2.9	23.3	9.3	28.9	3.4	19.7	3.1	21.4
1024	70.6	30.4	19.0	28.4	19.4	27.7	74.1	29.0	21.0	25.6	20.4	26.4
2048	557.4	30.8	145.0	29.7	146.6	29.3	577.2	29.8	154.8	27.8	153.8	28.0
2560	1088.	30.8	280.5	29.9	282.7	29.7	1124.	29.8	297.7	28.4	295.9	28.4

Table 15: Execution time for INVERSE_S and INVERSE_W on the IBM SP2

The observations related to the performances of the LU decomposition algorithms on the Cray T3D hold for the inversion algorithms as well. Table 17 depicts the performance of both INVERSE_S and INVERSE_W. As the latter one has been consistently outperformed by the former one, we did not include any experiments of it when pivoting caused row exchanges. Thus the performance of INVERSE_S only is depicted in Table 18. The effective megaflop rate is consistently at least as good as that of the LU decomposition and the

n	INVERSE_S		INVERSE_W	
	Observed %	Predicted %	Observed %	Predicted %
512	68	56	73	45
1024	88	81	90	75
2048	93	92	94	91
2560	94	94	95	93

Table 16: Observed and Predicted (lower bound) efficiency for matrix inversion on an IBM SP2

ratio of three in the performance of the two algorithms is also observed and maintained in the observed running times. Because of the high running time, only problem sizes up to $n = 2048$ are included. Efficiencies in excess of 100% are also observed in certain instances, especially in the case of INVERSE_S. Such occurrences are mainly due to caching; such effects seem to peak for $n \leq 1024$.

Table 19 gives the predicted and observed efficiency for INVERSE_S and INVERSE_W when no row exchanges occurred in pivoting and 20 when they do occur; it is noted that in the latter case, both timing results and efficiencies are skewed by the fact that the for fixed n, the number of exchanged rows increases with p thus making it complicated to draw any accurate conclusions. The predicted efficiencies are computed from the expressions in Theorems 5 and 6 and the BSP values reported earlier for the Cray T3D of our tests. Despite the problems mentioned, the two sets of values are close to each other supporting the claim of predictability of BSP performance.

		$p = 1$		$p = 4$		$p = 16$		$p = 64$	
n	*Algorithm*	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate
512	INVERSE_S	23.8	11.3	5.47	12.4	1.5	11.4	0.93	4.7
	INVERSE_W			6.11	11.2	1.8	9.5	1.3	3.7
1024	INVERSE_S	225.3	9.5	48.1	11.2	11.2	12.0	3.9	8.7
	INVERSE_W			57.1	9.4	15.1	9.1	5.8	6.3
2048	INVERSE_S	1809.	9.4	453.6	9.4	96.8	11.1	24.3	11.1
	INVERSE_W			452.9	9.5	117.0	9.3	41.1	6.8

Table 17: Execution time for INVERSE_S and INVERSE_W on Cray T3D

		$p = 1$		$p = 4$		$p = 16$		$p = 64$	
n	*Algorithm*	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate
512	INVERSE_S	24.4	11.0	5.9	11.4	1.9	9.0	1.29	3.4
1024	INVERSE_S	228.6	9.4	49.5	10.9	12.5	10.8	4.9	6.9
2048	INVERSE_S	1836.	9.3	459.0	9.3	100.9	10.6	27.7	9.7

Table 18: Execution time for INVERSE_S with pivoting on Cray T3D

6 Conclusion

In this work we first introduced some well known linear algebra sequential algorithms, described parallel algorithms for them (existing or new ones) in the context of the BSP model, and analyzed their performance in terms of the BSP parameters and the problem (matrix) size n. We then implemented these algorithms on three different parallel platforms (same portable source code was recompiled thrice) and reported the experimental results obtained.

n	$p = 4$		$p = 16$		$p = 64$	
	Observed %	Predicted	Observed %	Predicted %	Observed %	Predicted %
512	100	94	98	75	40	27
1024	100	97	100	90	90	56
2048	99	98	100	95	100	81

Table 19: Observed and Predicted (lower bound) efficiency for matrix inversion on a Cray T3D with no pivoting

n	$p = 4$		$p = 16$		$p = 64$	
	Observed %	Predicted	Observed %	Predicted %	Observed %	Predicted %
512	100	94	81	75	30	27
1024	100	97	100	90	72	56
2048	99	98	100	95	100	81

Table 20: Observed and Predicted (lower bound) efficiency for matrix inversion on a Cray T3D with partial pivoting

The following conclusions can be drawn from our previous discussion.

- The experiments have strengthened our convictions that algorithm design based on the BSP model is plausible, easy to follow, leads to algorithms that can be portable, scalable and, without much programming effort, efficient.

- The performance of BSP algorithms in diverse hardware platforms can be reliably predicted using the BSP parameters; one only needs an estimate of L and g for given p and the problem size n to predict algorithm performance for a particular machine. This way, one can reliably get a lower bound on the efficiency of a particular BSP algorithm implementation through the relative work parameter $w = \pi + \mu$.

- Whereas performance of a particular BSP algorithm can be reliably predicted, one *should not* expect the BSP cost model to accurately predict the running time and behavior of a particular implementation. Accurate prediction is difficult even for sequential algorithms due to the existence of varying caching hierarchies that the BSP cost model does not currently model; adding parallelism and the side-effects of communication introduces two more difficulties. This became apparent in matrix multiplication in particular, where caching effects led to superlinear speedup in performance; on the other hand, prior knowledge of data distribution (even for a sequential algorithm) led to a significant increase in performance (use of transposition in matrix multiplication). One should also be careful how the BSP parameters L and g are computed for a particular machine. Optimistic assumptions in the derivation of these values [18] can lead to considerable discrepancies between predicted and observed performance estimation; more realistic assumptions that also

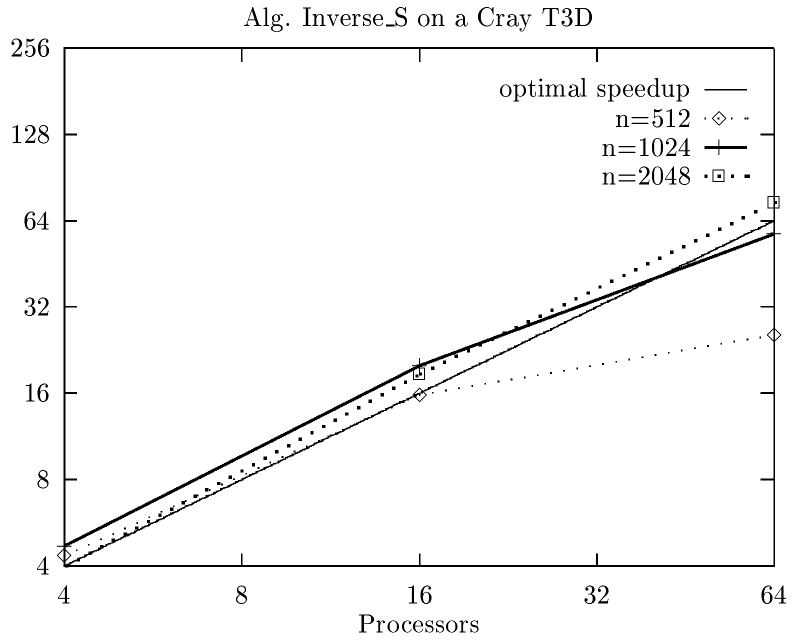

Figure 10: Speedup for INVERSE_S (no-pivoting) on the Cray T3D.

take into consideration the patterns of communication observed in real-life applications may yield sets of values for L and g that are closer to the ones observed in practice.

- There have been various attempts to more accurately predict parallel performance by extending the BSP model. The E-BSP model [4] is one such approach and is more explicit and specific about the communication network of a particular hardware platform, and the patterns of communication involved in routing. The attractiveness of the BSP cost model is its simplicity and generality; introducing more parameters to describe the performance of a communication network under various patterns of communication increases the complexity of describing the performance of all but the simplest algorithms with perhaps only small gains in prediction accuracy. Such an approach may also be problematic as it is the main reason parallel computing failed in the past: the attempt to realize for a particular algorithm those patterns of communication that are optimal for a given platform (and communication network), whereas they may lead in significant degradation in performance, if utilized in other platforms (portability vs. efficiency). Another variant of the BSP model described in [5] introduces one more parameter, B, related to message size, and associates g with that message size, enforcing this way coarse-grained communication of messages of size equal to B. The original BSP model does not elaborate in detail between fine-grained and coarse-grained communication. If small h-relations are communicated (where "small" is to mean less than some parameter h_0, usually assumed to be equal to L/g) a cost L is assigned to such a communication; no mention of message size is inferred. Presumably, for BSP to be an abstract and general-purpose model, details of how communication is performed efficiently are left to the BSP library implementor. In practice, the generality of the BSP model works well if one interprets the cost model so as to absorb in L not only "small" h-relations but also "small" messages; therefore, the value of B is reflected in the choice of L and g (as is, h_0 as well) without the need of introducing an extra parameter.

- For the current generation of computer platforms, one may gain in efficiency by exploiting *parallelism*; in many instances in our experiments the parallel implementations exhibited higher megaflop-ratings than the sequential ones. This was mainly a result of improved cache utilization.

- No attempt was made to optimize our code by sidestepping the communication primitives offered by *BSPlib* and using equivalent, highly-optimized, platform specific ones; neither has it been attempted to optimize the sequential code to take full advantage of the capabilities of each CPU on the named platforms. The former could be easily incorporated in *BSPlib* for a specific function; the programmer hopes and expects the library implementor to make the correct choice. If the latter optimizations were performed, then both our sequential and parallel implementations would have benefited from them; it is hoped that the compiler optimizer performs as many optimizations as possible and the programmer is always aware of the fact that ANSI C programs may be slower that assembly language one. Our aim in this work is to show how one can write, with minimum programming effort, portable, readable, and scalable BSP programs that can also be quite efficient. The ANSI C implementation of algorithm MULTT_B is given in the appendix to support these claims.

7 Acknowledgements

It is a pleasure to thank Jonathan M. D. Hill for his assistance in obtaining the experimental results on the Cray T3D. The support of the Edinburgh Parallel Computing Centre in granting us access to the Cray T3D there is gratefully acknowledged.

References

[1] M. Barnett, D. Payne, R. van de Geijn, and J. Watts. Broadcasting on meshes with wormhole routing. *Journal of Parallel and Distributed Computing*, 35(2):111–122, 1996.

[2] R. H. Bisseling and W. F. McColl. Scientific computing on Bulk-Synchronous Parallel architectures. Preprint 836, Department of Mathematics, University of Utrecht, December 1993.

[3] R. H. Bisseling. `http://www.math.ruu.nl/people/bisselin/software.html`. BSPPACK version 0.2, June 1, 1997.

[4] B. H. H. Juurlink and H. A. G. Wijshoff. The E-BSP model: Incorporating unbalanced communication and general locality into the BSP model. Technical Report TR-95-44, Leiden University, 1995.

[5] A. Baumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *Proceedings of the Annual European Symposium on Algorithms*, 1995.

[6] 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: General Techniques and Regular Problems.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[7] A. V. Gerbessiotis. Topics in parallel and distributed computing. PhD Thesis (Technical Report TR-03-93), Aiken Computation Laboratory, Harvard University, 1993.

[8] A. V. Gerbessiotis. Practical Considerations of Parallel Simulations and Algorithm Design on the BSP Model. Submitted for Publication.

[9] A. V. Gerbessiotis. Web page:
`http://www.comlab.ox.ac.uk/oucl/people/alex.gerbessiotis.html`

[10] A. V. Gerbessiotis and F. Petrini. Network Performance Assessment under the BSP Model. To appear in the Workshop for Constructive Methods for Parallel Programming 1998, Marstrand/Gotenborg, Sweden.

[11] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel algorithms. *Journal of Parallel and Distributed Computing,* 22:251-267, 1994.

[12] A. V. Gerbessiotis and C. J. Siniolakis. Primitive Operations on the BSP Model. Technical Report PRG-TR-23-96, Computing Laboratory, Oxford University, October 1996.

[13] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The John-Hopkins University Press, Baltimore, MD, 1989.

[14] M.W. Goudreau, J.M.D. Hill, K. Lang, W.F. McColl, S.D. Rao, D.C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for a BSP Worldwide standard. BSP Worldwide, `http://www.bsp-worldwide.org/`, April 1996.

[15] M. Goudreau, K. Lang, S. Rao and T. Tsantilas. The Green BSP Library. Technical Report CR-TR-95-11, University of Central Florida, 1995.

[16] G. A. Geist and M. T. Heath. Matrix Factorization on a Hypercube. In *Proceedings of First SIAM Conference on Hypercube Multiprocessors*, (ed. M. T. Heath), SIAM Publications, 1986.

[17] J. M. D. Hill and D. Skillicorn. Lessons learned from implementing BSP. To appear in High Performance Computing and Networking (HPCN'97), Lecture Notes in Computer Science, Springer-Verlag, April 1997.

[18] J. M. D. Hill. In `http://www.bsp-worldwide.org/implmnts/oxtool/`, September 1997.

[19] J. M. D. Hill. Personal Communication, September 1997.

[20] B. H. H. Juurlink and H. A. G. Wijshoff. Communication primitives for BSP computers. *Information Processing Letters,* 58:303-310, 1996.

[21] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes.* Morgan Kaufmann, California, 1991.

[22] R. Miller. A library for Bulk-Synchronous Parallel programming. In *Proceedings of the British Computer Society Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing*, December 1993.

[23] D. Talia. Parallel computation still not ready for the mainstream. *Communications of the ACM,* 44(7):98-99, July 1997.

[24] L. G. Valiant. Bulk-synchronous parallel computers. In *Parallel Processing and Artificial Intelligence*, (M. Reeve and S. E. Zenith, eds.), Wiley, 1989.

[25] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM,* 33(8):103-111, August 1990.

[26] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science,* (J. van Leeuwen, ed.), North Holland, 1990.

[27] L. G. Valiant. A combining mechanism for parallel computers. In *Parallel Architectures and Their Efficient Use*, pages 1-10, LNCS 678, Springer-Verlag, 1993.

A Algorithm MULTT_B in ANSI-C

Below, the source code for the matrix multiplication algorithm MULTT_B tested is given. Initially, two global $n \times n$ matrices A_g and B_g are distributed among p processors. The p processors are divided into $\sqrt{p}$ groups of $\sqrt{p}$ processors each. Element (i,j) of A_g or B_g is stored in the $i/\sqrt{p}$-th processor of the $j/\sqrt{p}$ processor group, that is, processor $(j/\sqrt{p}) * \sqrt{p} + i/\sqrt{p}$.

Function `_multiply_par` requires five arguments, the two input matrices A and B which contain the n^2/p elements of A_g and B_g that are local to a particular processor, the result C that will hold n^2/p elements of the product C_g, the dimension n, and $\sqrt{p}$ (which must be an integer).

As we have previously mentioned, A, B and C are ANSI-C pointers to a `double` data type. We store matrices in the form of an on dimensional array. This way, element (i,j) of a two-dimension matrix is stored in position $j * n + i$. All indices are in the range $0, \ldots, n-1$.

The *BSPlib* allows for SPMD (Single Program Multiple Data) programming. The program that calls `_multiply_par` spawns p processes each executing the same code. Each such process is assigned a unique identifier that can be accessed by `bsp_pid()`. The number of available processes is available through a call to `bsp_nprocs()`. Among the variables used, `nprocs` and `pid` hold the number of available processors p and the processor identifier of a process.

```
void _multiply_par(double *A,double *B,double *C,int n,int p_sqrt )
{
 register int    nprocs=bsp_nprocs(); /* # of processors */
 register int    pid=bsp_pid();       /* processor identifier */
```

Variables a and b will hold local copies of A and B that will be fetched during the course of the algorithm (lines 4 and 5 of Figure 2). Variable `ni` holds $n/\sqrt{p}$ and variables `ppi` and `ppj` hold respectively, the index of processor `pid` within processor group `ppj`, and the processor group to which processor `pid` belongs.

```
 double          *a,*b;                /* local storage  */
 register double tmp;                  /* temporary variable */
 register int    i,j,k,l;              /* indices */
 register int    t1,t2,                /* temporary indices */
                 ppi,ppj,              /* index and processor group*/
                                       /* for processor  pid */
                 ni;                   /* n/sqrt(p) */
```

In the following code segment `ni`, `ppi` and `ppj` are computed.

```
 ni=n/p_sqrt;
 ppi=pid % p_sqrt; /*index of pid within its (column block) group */
 ppj=pid / p_sqrt; /* (column-block) group pid belongs to */
```

In lines 14-18, allocation of memory space for `a` and `b` is performed. Function `check_if_null()` checks whether a returned pointer by `malloc()` is empty or not and in the former case prints an informative diagnostic message.

```
14.  /* Allocation and checking */
15.  a = (double *)malloc(ni*ni*sizeof(double));
16.  check_if_null((void *)a,"in multiply_par","a");
17.  b = (double *)malloc(ni*ni*sizeof(double));
18.  check_if_null((void *)b,"in multiply_par","b");
```

A registration of variables that hold data to be communicated is required in *BSPlib*. This is performed in lines 19-21.

```
     /* Registration */
19.  bsp_pushregister(A,ni*ni*sizeof(double));
20.  bsp_pushregister(B,ni*ni*sizeof(double));
21.  bsp_sync();
```

An initialization of the result matrix C is performed in lines 22-24.

```
     /* Initialization */
22.  for (j=0;j<ni;j++)
23.    for (i=0,t1=j*ni;i<ni;i++)
24.      C[t1 + i]= 0.0;
```

The number of communication rounds (supersteps) for this matrix multiplication routine is $\sqrt{p}$; line 25 initiates the loop of line 3 or Figure 2. In round l the required submatrices stored in processor $((p_i + p_j + l) \bmod \sqrt{p}) * \sqrt{p} + p_i$ (for A) and $((p_i + p_j + l) \bmod \sqrt{p}) + p_j * \sqrt{p}$ (for B) are fetched. These operations correspond to lines 4 and 5 of Figure 2. The syntax of `bsp_get(processor,source,offset,destination,size)` is as follows: `processor` identifies the processor from whom data will be obtained, `source` is a pointer to the structure that will provide the data from `processor`, `offset` is an offset in bytes of the first address of the data to be transferred from `source`, `destination` is the destination address in `pid` of the data that will be transferred and `size` is the size of transferred data in bytes.

```
25. for (l=0;l<p_sqrt;l++) {
     /* Efficient non-conflicting communication */
     /* A 2n^2/p relation is communicated */
26.  bsp_get(((ppi+ppj+l)%p_sqrt)*p_sqrt+ppi,A,0,a,ni*ni*sizeof(double));
27.  bsp_get(((ppi+ppj+l)%p_sqrt)+ppj*p_sqrt,B,0,b,ni*ni*sizeof(double));
28.  bsp_sync();
```

In lines 29-34 array a is transposed so that the following multiplication loop be performed more efficiently.

```
     /* Transpose a */
29.    for (i=0,t2=0;i<ni;i++,t2+=ni)
30.       for (j=i,t1=i*ni;j<ni;j++,t1+=ni)  {
31.         tmp=a[t2+j];
32.         a[t2+j]=a[t1+i];
33.         a[t1+i]=tmp;
34.       }
```

Lines 35-44 perform the local multiplication of line 6 of Figure 2.

```
     /* Multiplication */
35.      for (j=0;j<ni;j++) {
36.        t1=j*ni;
37.        for (i=0;i<ni;i++) {
38.            t2=i*ni;tmp=0.0;
39.            for (k=0;k<ni;k++)
40.              tmp += (b[t1+k]*a[t2+k]);
41.            C[t1+i] += tmp;
42.          }
43.      }
44.    } /* end of (loop) l-th superstep */
```

A de-registration of variables is performed, followed by a deallocation of variables.

```
45.  bsp_popregister(B);
46.  bsp_popregister(A);

47.  free((void *)b);
48.  free((void *)a);
}
```