Yale University
Department of Computer Science

Data Parallel Programming and Basic Linear Algebra Subroutines

S. Lennart Johnsson

YALEU/DCS/TR-584
September 1987

This work has been supported in part by the Office of Naval Research under Contracts N00014-84-K-0043 and N00014-86-K-0564. Approved for public release: distribution is unlimited.

DATA PARALLEL PROGRAMMING AND BASIC LINEAR ALGEBRA SUBROUTINES [1][2]

S. LENNART JOHNSSON[3]

Abstract. Data Parallel programming is conceptually simple and provides powerful programming primitives as in shared memory models of computation. With an appropriate underlying architecture, primitives requiring global memory access do not require significantly longer execution times than primitives only requiring local access. In the data parallel programming model primitives are also available for expressing simple forms of local data interaction in relative coordinates, as for instance required in relaxation on multidimensional lattices.

One particular data parallel computer is the Connection Machine. In this paper, we describe some of the data parallel aspects of the programming languages provided on the Connection Machine. We comment on the implementation of level-1 and level-2 BLAS, and describe the implementation of one level-3 BLAS function. Matrix multiplication is discussed in detail. For matrices of the size of the machine or larger, only the kernel function is required. The matrix multiplication kernel yields a performance of up to 5.2 Gflops in single precision on the CM-2 with the floating-point option. For matrices considerably smaller than the machine, all three nested loops in a Fortran 77 program can be made parallel, and expressed with few instructions without any loop constructs.

Key words. Supercomputing, Parallel Computing, Basic Linear Algebra, and Mathematical software.

AMS(MOS) subject classifications. 68Q10, 68Q25, 65F30

1 Introduction

High performance computation requires a high storage bandwidth. Supercomputers with a performance of a few Gflops need an effective storage bandwidth of the order of at least one Tbit/sec. The memory width is 10^5 bits for 10 MHz memory chips. In a register or cache architecture the width of the primary storage needs to be substantial. We do not expect memory chips to become faster by more than a small constant factor. Hence, future supercomputers will require storage with an even greater width. For most designs in VLSI technology the processor speed is comparable to, or a small constant factor higher than, the speed of a standard memory chip. With a machine designed entirely in VLSI technology the number of processors is a fraction of the number of memories in a system having a good balance between processing capability and storage bandwidth. The fraction is essentially determined by the width of the processor. A balanced design with supercomputer performance in todays technology would require 100,000 1-bit processors, or 3,000

[1]To appear in Mathematical Aspects of Scientific Software, Springer Verlag

[2]Presented at the Institute for Mathematics and its Applications, the University of Minnesota, Minneapolis, March 23-27 1987.

[3]Thinking Machines Corp., 245 First Street, Cambridge, MA 02142, and Departments of Computer Science and Electrical Engineering, Yale University, New Haven, CT 06520

32-bit processors. With bandwidths in the order of a few Tbits/sec, or higher, it is increasingly difficult, and expensive, to have a bus for interconnecting memories with the processors. Interconnection networks are becoming common in parallel architectures. The Connection Machine described in some detail later has a total storage bandwidth of approximately 50 Gbytes/sec and $64k$ processors.

Programming of a computer with a large number of processors cannot be made in a way in which programmers concern themselves with every processor, and their synchronization. Indeed, it is desirable that the programmer does not need to be concerned with details of the architecture, neither for functionality, nor for performance. Communication should largely be hidden from the programmer, as in sequential programming models, but clearly requires efficient implementations. With a large number of processing elements and limited bandwidths at chip and board boundaries, it is inevitable that the width of the communication channels be small. With a bit-serial pipelined communication system the time for global communication is of the same order as the time for local communication. Implementing a shared memory model of computation is a realistic proposition. Shared memory models of computation provide powerful operators, such as concurrent read, and concurrent write. This allows for the efficient implementation of parallel prefix operations.

In a data parallel programming model the emphasis is on the data structures, and the interaction between elements of the data structure. Language primitives may be provided for particular data structures and simple interactions within these structures. An example is multi-dimensional lattices, and nearest-neighbor communication in such structures. Operators for subsets are also provided, such as copying an element to every processor that needs that element, finding the maximum or minimum in a set, or the sum of all elements in a set. Global operators are quite powerful programming primitives.

The data parallel programming model is particularly suitable for highly concurrent architectures with a communications system that allows fast access to any part of the storage from any processor. Ideally there is one processor for every primary object of the problem. In reality that is rarely the case. Multiple elements have to be mapped to the same processor. The notion of *virtual processors* is used to allow the algorithm designer and the programmer to work with the data parallel model and not concern themselves with the mapping of the abstract machine to the real machine. Mapping of *virtual processors* to *real processors* is handled at compile time, if possible. The maximum problem size that can be handled is determined by the total amount of storage, not by the number of *real processors*, as in any realistic programming model. Storage of a *real processor* is divided among *virtual processors*. The *virtual processors* assigned to a *real processor* time-share that processor. The scheme for assigning *virtual processors* to real processors affects the need for communication, and the *real processor* utilization. The most common assignment scheme is *consecutive* and *cyclic* [9]. For the matrix multiplication described in some detail later, either of these forms yield the same arithmetic and communication complexity, but a different control structure. Cyclic mapping is obtained by simply letting the low order bits of an address be the real processor address. In *consecutive* mapping the high order bits are used for real processor addresses. Cyclic mapping yields a better load balance for LU-decomposition [9].

In the remainder of this paper, we elaborate on the data parallel programming

model. The Connection Machine architecture is described, and some of the unique features of the programming languages are mentioned. We discuss implementation of level-1 and level-2 BLAS briefly, and treat in some detail a kernel for matrix multiplication, and the procedure for multiplying matrices of arbitrary shapes.

2 The Data Parallel programming model

In a data parallel programming model algorithms are devised and described, for the purpose of computing the solution with the structure of the problem domain in mind. Algorithms are expressed as a sequence of interactions between the elements of the structure. The type of interaction is often the same for large sets of elements. For instance, in solving partial differential equations a discretization of continuous space is introduced by some approximation scheme, like finite differences, or finite elements. Then, an algorithm is devised for the solution of the resulting systems of equations. A solution can be expressed concisely as a sequence of interactions between the nodal quantities in the discrete space. Nodal quantities may be vectors, or single elements. Interactions may be local; they may cover a large domain; or potentially they will cover the entire domain. If an iterative method is used for the solution, then the interaction between elements of the data structure is described by the stencil used for the approximation of the differential operators, or by the type of elements used for finite element problems. If an elimination method is used for the solution of the system of equations, then the first several steps in a nested dissection procedure often only involve local interactions. However, as the eliminations progress the interactions may extend over a larger domain. The more balanced the elimination tree, the further apart are typically the data elements involved in final elimination stages. For a regular lattice it is of the order of one side of the lattice. In terms of the number of points involved, it is of the order of $\sqrt{N}$ for a two-dimensional problem and $N^{\frac{2}{3}}$ for a three-dimensional problem.

Regardless of whether a direct or iterative technique is used for the solution of the system of equations, the type of interaction in most cases is the same for a very large set of nodes in the discrete space. For instance, the same stencil is typically used in the entire domain, with the exception for the boundary. We have also implemented some high resolution schemes for shocks in fluid flow calculations, which use the same discretization over the entire domain. In adaptive techniques for the solution of partial differential equations, the same approximation may still be used in large regions of the domain. It may even be the case that only the resolution differs. In direct solvers the pivot row (column) interacts with every other row (column) for variables yet to be eliminated. Interaction is the same for every element. If interaction is not identical for all elements, only a few types of interaction occurs in algorithms we are aware of. An essential characteristic of data parallel architectures is the ability to define classes of objects, and the operations thereupon.

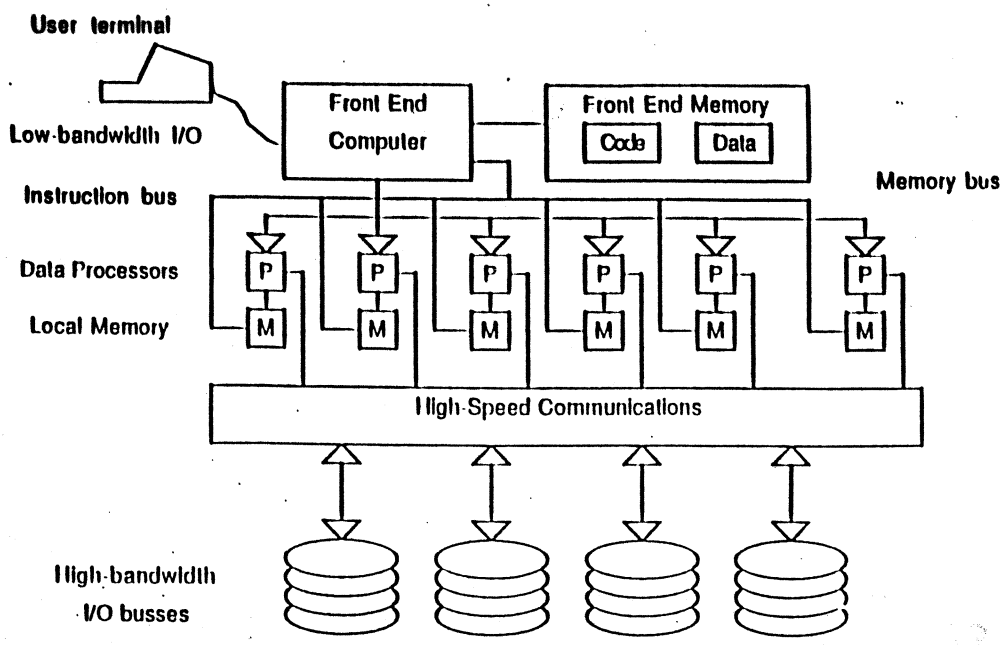

Figure 1: The Connection Machine system

3 The Connection Machine

3.1 Architecture

One example of a data parallel architecture is the Connection Machine. It has a total primary storage of 512 *Mbytes* using 256 *kbit* memory chips; and 2 *Gbytes* using 1 *Mbit* memory chips. Bandwidth to storage is approximately 50 *Gbytes/sec*, which is achieved at a modest $8MHz$ clock rate. Primary storage has $64k$ ports, and a simple 1-bit processor for each port. The Connection Machine model CM-2 also has facilities for adding hardware for floating-point arithmetic. With the floating-point option a performance of 5.2 Gflops has been measured for matrix operations. We describe the algorithm used to realize this performance. This routine is the kernel for multiplying arbitrarily shaped matrices.

The Connection Machine requires a host. Currently, two families of host architectures are supported: the VAX family with the BI-bus, and the Symbolics 3600 series. The Connection Machine is mapped into the address space of the host. The program code resides in the storage of the host. The host fetches the instructions, does the complete decoding of scalar instructions, and executes them. Instructions to be applied to variables in the Connection Machine are sent to a microcontroller, which decodes and executes instructions for the Connection Machine. The architecture is depicted in Figure 1

The difference between the nominal peak performance, and actual performance on the Connection Machine is largely due to the time spent in moving data between processors. Other overhead is low. The interprocessor communication capability depends on the communication pattern. For many linear algebra operations communication in a two-dimensional lattice is desirable. The lattice need not be a square lattice. The effective interprocessor communication bandwidth for two-dimensional lattice emulation is in the range $3 - 12$ *Gbytes/sec* for 4-byte messages. Overhead

Algorithm	Arithmetic op's/ element	Arithmetic ops/ Elem. comm.
Matrix-vector mpy	2	$\sqrt{M}$
Matrix-matrix mpy	$\frac{2}{3}\sqrt{N}$	$\sqrt{M}$
Relaxation 5-point stencil	1.6/iter.	$\sqrt{M}$
Radix-2 FFT	$1.25log_2N$	$1.25log_2(M/2)$
Sorting	$O(log_2N)$ - $O(log_2^2N)$	

Table 1: Number of operations per data element for a few operations on N elements in a machine with local storage M.

is approximately 15% for messages of this length. Communication time increases linearly with the message size. For communication in arbitrary patterns the Connection Machine is equipped with a router, which selects one of the shortest paths between source and destination, unless all of these paths are occupied. The router has several options for resolving contention for communication channels. The bandwidth for random permutations is a factor of 6 - 7 below that of the two-dimensional lattice emulation.

The degradation is due to bandwidth limitations at the chip boundaries. Performance measured as arithmetic or logic operations per unit time, benefits from increased granularity for many operations in that the arithmetic/logic operations grow faster than the communication as a function of the number of local data elements, as shown in Table 1.

3.2 Programming languages

Connection Machine programming languages are currently *Lisp, and a parallel version of C called C*. These languages are extensions of the familiar Lisp and C languages. The most essential extensions are the existence of a parallel data type, and the operations thereupon. Scans are among the operators included in the extensions. Concurrent read and concurrent write instructions are also supported. Members of a set of elements forming a parallel variable are operated upon concurrently by one instruction. No enumeration of the elements is required, and one or several loop levels disappears from the code, compared to languages not supporting array or set operations. The code becomes more compact, simpler, easier to debug, and one source of errors has vanished. Given that the programming languages for the Connection Machine are extensions of conventional languages the debugging tools, and the debugging process is similar to that for conventional architectures.

3.2.1 *Lisp

*Lisp is an extension of Common Lisp. There is one additional data type: a parallel variable, known as *pvar* . Parallel variables are defined through a (*defvar *pvar pvar-expression*) statement. In the current implementation a *pvar* is allocated across the

entire configuration of the Connection Machine. The same section of the storage of every processor is assigned to a given *pvar* . The *pvar-expression* is optional. If present, a value is computed for every element of the *pvar* . Any element of a pvar can be referenced by the statement (pref *pvar address*), which returns the element specified by *address* of the pvar specified by *pvar* . Assigning elements to pvars can be made through the function (*set *pvar-1 pvar-2*). The elements of *pvar-1* is set to the corresponding values of *pvar-2* . Individual elements of a pvar can be set by using the Lisp function setf : (setf (pref *pvar address*) *var*)

*Lisp provides two global addressing schemes: conventional binary addresses, known as *cube-address* , and addresses in multi-dimensional lattices, known as *grid-address* . References on the grid addresses can be made with both absolute coordinates, and relative addresses. The forms are

(pref *pvar address*)
(pref-grid *pvar grid-address*)
(pref-grid-relative *pvar relative-grid-address*)

and for concurrent access

(pref !! *pvar-expression cube-address-pvar*)
(pref-grid !! *pvar-expression grid-address-pvars border-pvar*)
(pref-grid-relative !! *pvar-expression relative-grid-address-pvars border-pvar*)

where *grid-address* is the address of a lattice point, *relative-grid-address* the relative address of a lattice point. *grid-address-pvars* and *relative-grid-address-pvars* must contain as many address pvars as there are dimensions in the lattice. The *border-pvar* field is optional. If it is provided, then if a processor p references a processor outside the defined lattice, then instead the value of the pvar *border-pvar* in processor p is returned. Standard operators in Common Lisp are extended to pvars by the suffix !!. The following example illustrates the implementation of Jacobi iteration for the 5-point stencil

```
(*set new
     (*!!
          (!!0.25)
          (+!!
               (pref-grid-relative !! old (!! -1) (!! 0))
               (pref-grid-relative !! old (!! 0) (!! -1))
               (pref-grid-relative !! old (!! 0) (!! 1))
               (pref-grid-relative !! old (!! 1) (!! 0))
               rhs
          )
     )
)
```

The operation *set is a local memory movement in all selected processors. The corresponding global operation is *pset , which like pref comes in three forms depending on the addressing scheme.

(*pset *combiner value-pvar destination-pvar cube-address*)

(*pset-grid *combiner value-pvar destination-pvar grid-address-pvars*)
(*pset-grid-relative *combiner value-pvar destination-pvar relative-grid-address-pvars*)

The content of *value-pvar* will be written into *destination-pvar* of the processor(s) specified by *cube-address* , *grid-address-pvars* , or *relative-grid-address-pvars* , respectively. Depending upon the addresses that are specified, several processors may be writing into the same memory location. The field *combiner* specifies the type of combining that is desired in such a case. Examples of *combiners* are add, max, and min.

In general, whether or not an operation shall be carried out is a function of the state. Examples of conditionals in *Lisp are *all, *when, *if, and *cond.

*all *body*
*when *pvar body*
*if *pvar-expression then-form else-form*
*cond {(*pvar* {form}*)}*

In the *all statement the body is evaluated for the entire set of processors; *when subselects the processors for which *pvar* is non-NIL from the currently selected set. For the *if statement the subset of processors of the currently selected set for which the *pvar* is non-NIL executes the *then-form.* The *else-form* is optional. The function *cond evaluates *all* clauses. Subselection is based on the *pvar-expression* .

*Lisp also has some powerful global operators. Of particular interest for numeric applications are *min , *max , and *sum , and *scans.* The *min , *max , and *sum operators compute the minimum, maximum, and the sum of the values in a pvar in the currently selected set of processors. Scans are specified by

(scan !! *pvar function* : direction *segment-pvar* :include-self)
(scan-grid !! *pvar function* :dimension : direction *segment-pvar* :include-self)

The *segment-pvar* divides the address space into non-overlapping segments. The scan operation is concurrently applied to all segments for which the *segment-pvar* is true. A scan is a parallel prefix operation, with the prefix specified by the *function* field. If the function is *plus*, then every pvar location will contain the sum of the pvar elements in processors with lower addresses in its segment, including its own original pvar value, if include-self is true. The scanning can also be made for decreasing addresses by specifying the direction to be backward. Note that for non-associative operators such as floating-point addition, precision may be lost.

3.2.2 C*

C* is an extension of C with a strong influence from C++. Objects that are of the same nature are in C* members of the same domain . Conceptually, a domain is similar to a class in C++. In a data parallel model of computation a processor is associated with every instance of a domain . Every member of a domain has the same storage layout. Referencing a domain implies a selection of processors. Only processors associated with an instance of the referenced domain remain active.

There are two new data types: mono and poly . Data of type poly are allocated in the Connection Machine storage. Data belonging to any domain is by default of type poly . Data that is not of type poly is of type mono , and resides in the storage of the host machine.

There are only very few new operators in C*. Most C operators are extended to C* by the distinction between mono and poly . Communication between the host and the Connection Machine is implicit. Broadcasting from the host occurs if a mono value is assigned to a poly variable. Reduction occurs if the combined result of the elements of a poly variable is desired. The result of the reduction is a mono value in the host. Interaction between the elements of a single or several poly variables results in various communication patterns in the Connection Machine. C* does not currently allow the programmer to specify particular communication patterns, such as lattice communication.

3.2.3 *Fortran

The *Fortran language that is being planned for the Connection Machine is similar to the proposed Fortan 8X standard. This standard has array constructs and operations on such constructs. In particular, the maxval, minval, and sum functions correspond to the *min , *max , and *sum functions already available in *Lisp, and will be supported in *Fortran. Many of the same comments have been made for *Lisp and C* applies to *Fortran.

4 The BLAS

Level-1 BLAS are operations on a vector, or pairs of vectors. Vectors are *pvars* or poly variables in the data parallel model. Dot products are easily expressed in both *Lisp and C*. For instance, the *Lisp expression is *sum (*!! x y). The BLAS-1 copy routine is simply a *set . The level-1 BLAS routines does not require any loop constructs. Required data movement is implicit in the instructions, and the efficiency of level-1 BLAS a consequence of being at the level of machine instructions.

The level-2 BLAS routines can be written with two nested loops in Fortran 77. One of the operands is a matrix. With the machine configured as a two dimensional array the assignment of matrix elements to processors is simple. The level-2 BLAS routine has one or two vector operands. With the vectors aligned with one of the axis, a transposition may be (but need not be) required. The vector transposition is a single instruction. For matrix-vector multiplication one *copy-scan-grid* and one *plus-scan-grid* will suffice in addition to a concurrent multiplication over all matrix elements. For a rank-1 update, two copy-scan-grid instructions and the concurrent multiplication suffice. For the solution of a triangular system of equations loops can be avoided, if the inverse is represented instead of the triangular factors. The forward or backsolve then become matrix-vector multiplications. Otherwise, one loop is required.

Level-3 BLAS attempts to provide a standard interface for matrix-matrix operations. In the remainder of this paper we provide some insight into the techniques and considerations in implementing matrix multiplication on a data parallel archi-

tecture.

5 Multiplying arbitrarily shaped matrices

The computation we consider for the remainder of this paper is $A \leftarrow B \times C + D$, where the matrix B is a $P \times Q$ matrix and C a $Q \times R$ matrix and A and D $P \times R$ matrices. Particular issues we focus on are processor utilization for arbitrarily shaped matrices, effective use of the communication capabilities of an architecture such as the Connection Machine, and software engineering issues in terms of suitable primitives, in the spirit of the BLAS.

A general matrix computation can take several standard forms, such as an inner-product covered by the level-1 BLAS [16]; an outer product or rank-1 update covered by the level-2 BLAS (GER) [4]; an AXPY (level-1 BLAS) [16], or triad; or a matrix-vector product (GEMV) [4]. These are degenerate cases of matrix-matrix multiplication (MM), and level-3 BLAS [3]. Non-degenerate matrix-matrix multiplication can be expressed in terms of the inner-product operation or the AXPY. The former is suitable for architectures with fast inner-product instructions, and the latter for architectures with pipelined arithmetic units.

None of the BLAS versions directly address the issues in parallel computation. However, block algorithms in part are the justification for introducing the level-3 BLAS work well on shared memory multiprocessors [8]. For distributed storage architectures a variety of systolic algorithms have been proposed for dense and banded matrices [1,2,15,17,10,12,11,6]. These algorithms define synchronization between different data streams, and the alignment between those streams. Systolic algorithms are typically presented for one element per processor, but are easily generalized to a submatrix of each operand per processor [9,5]. For systolic type algorithms constant storage suffice. A reduction in the communication time is possible in architectures with a high communication overhead, if data aggregation is allowed [13]. For minimal communication time communication buffers and temporary storage of the order of $O(\sqrt{\frac{PQ}{N}})$ (or $O(\sqrt{\frac{PR}{N}})$, or $O(\frac{QR}{N})$) is necessary. In a data parallel architecture such as the Connection Machine [7], communication overhead is very low, and constant storage algorithms are preferable. Performance related issues are the utilization of processors and the communication bandwidth for arbitrarily shaped matrices. From a software engineering point of view, it is important to find programming/algorithm primitives that are useful for a variety of matrix shapes.

Matrix multiplication consists of three nested loops. In a data parallel architecture there is the potential for performing all three concurrently. In such a case the multiplication of a $P \times Q$ and a $Q \times R$ matrix can be performed in a time proportional to $log_2 Q$, if the number of processors $N \geq PQR$. Parallelizing the loop in the Q direction is beneficial, if there are more than PR processing elements. Our matrix multiplication algorithm for matrices of arbitrary shapes distinguishes between several cases. The kernel function that we describe here is based on an algorithm by Cannon [1]. This algorithm is devised for two-dimensional square lattices.

5.1 Data Allocation

For the kernel by Cannon the processor set is partitioned into a two-dimensional array of $N_r \times N_c = \lfloor\sqrt{N}\rfloor \times \frac{N}{\lfloor\sqrt{N}\rfloor}$ processors. For $log_2 N$ even the processing array is square, otherwise it is rectangular with one side twice the length of the other. For matrices such that P, Q and R are all smaller than the lattice dimensions one matrix element can be assigned to each processor. At the other extreme the number of matrix elements in all dimensions may exceed the number of processors in that dimension. Multiple elements have to be assigned to every processor, with a strategy to minimize the maximum storage per processor. Two apparent schemes are *cyclic* and *consecutive* assignment [9]. The two assignment schemes can be illustrated as follows, assuming $P = 2^p$, $Q = 2^q$, and $R = 2^r$ for simplicity.

$$\underbrace{(u_{p-1}u_{p-2}\ldots u_{n_r}}_{vp}\underbrace{u_{n_r-1}u_{n_r-2}\ldots u_0}_{rp}\underbrace{v_{q-1}v_{q-2}\ldots v_{n_c}}_{vp}\underbrace{v_{n_c-1}v_{n_c-2}\ldots v_0)}_{rp}$$

$$\underbrace{(u_{p-1}u_{p-2}\ldots u_{p-n_r}}_{rp}\underbrace{u_{p-n_r-1}u_{p-n_r-2}\ldots u_0}_{vp}\underbrace{v_{q-1}v_{q-2}\ldots v_{q-n_c}}_{rp}\underbrace{v_{q-n_c-1}v_{q-n_c-2}\ldots v_0)}_{vp}$$

The real address field is made up of $n_r + n_c = n$ address bits obtained by concatenating the real processor row address field with the real processor column address field.

In the *consecutive* assignment each processor is assigned a block matrix of size $2^{p-n_r} \times 2^{q-n_c}$ of matrix B, a block matrix of size $2^{q-n_r} \times 2^{r-n_c}$ of matrix C, and a block matrix of size $2^{p-n_r} \times 2^{r-n_c}$ of matrix A. Low order bits are used for *virtual processor* addresses of both the row and the column address fields. In the *cyclic* assignment the low order bits are instead used for *real processor* addresses, and high order bits for *virtual processors*. Mixed assignment schemes can be used as well [14], but will not be considered here. In the cyclic assignment the matrices are "tiled" with tiles of size $2^{n_r} \times 2^{n_c}$. Each such tile represents a slice of storage across all processors [9].

5.2 A Kernel for Concurrent Matrix Multiplication

In the algorithm by Cannon the inner products defining the elements of A are accumulated *in-place*. Denote the storage cells for A, B, C and D by E, F and G. The algorithm has two phases. A set-up phase in which the operands are aligned, and a multiplication phase. In the set-up phase the shifting yields: $F(i,j) \leftarrow F(i,(i+j) \pmod{2^k})$, $G(i,j) \leftarrow G((i+j) \pmod{2^k}, j)$, $E \leftarrow D$ for $(i,j) \in \{0,1,2,...,2^k-1\} \times \{0,1,2,...,2^k-1\}$. Clearly $F(i,j) \times G(i,j)$ is a valid product for all i and j. In the multiplication phase the following operations are carried out: $E(i,j) \leftarrow E(i,j) + F(i,j) \times G(i,j)$, $F(i,j) \leftarrow F(i,(j+1) \pmod{2^k})$, $G(i,j) \leftarrow G((i+1) \pmod{2^k}, j)$, $i,j = \{0,1,2,...,2^k-1\}$.

We distinguish between the following cases:

1. The operands covers the processor array exactly.

2. The number of elements in each of the two dimensions of the operands are multiples of the number of processing elements in the corresponding dimension.

3. For each of the matrices the number of elements in at least one dimension is less than the number of processors assigned to that dimension.

5.2.1 Matrices Perfectly Matching the Mesh Size

The first case corresponds to the kernel function for $N_r = N_c$. If the number of processors is an odd power of two then a mesh with aspect ratio two is the closest to a square mesh. With square matrices each real processor has to simulate two virtual processors. Note, that with the same storage scheme for all matrices and no matrix dimension exceeding the processor mesh $Q = min(N_r, N_c)$. Hence, either $P = max(N_r, N_c)$ and $R = min(N_r, N_c)$ or vice versa. The multiplication can be performed by first replicating the matrix C in the row direction $\frac{max(N_r,N_c)}{min(N_r,N_c)}$ times if $P > R$, else replicating B if $R > P$. The kernel algorithm is then applied concurrently to the $\frac{max(N_r,N_c)}{min(N_r,N_c)}$ meshes of size $min(N_r, N_c) \times min(N_r, N_c)$.

5.2.2 Virtual processors

In the second case each *real processor* has several *virtual processors*. In the *consecutive* assignment each *real processor* has a block matrix that with matrix dimensions being powers of two are of size $2^{p-n_r} \times 2^{q-n_c}$ for matrix B, of size $2^{q-n_r} \times 2^{r-n_c}$ for matrix C, and of size $2^{p-n_r} \times 2^{r-n_c}$ for matrix A. For the case of $N_r = N_c = \frac{n}{2}$ each *real processor* performs a block matrix operation requiring $2 \cdot 2^{p+r+q-\frac{3}{2}n}$ operations. Then, a rotation of the matrices B and C is performed blockwise, and the process repeated $2^{\frac{1}{2}n}$ times. The time for arithmetic is $\frac{PQR}{N}$, and the data transfer time proportional to $\frac{(P+R)Q}{\sqrt{N}}$. In the bit-serial pipelined communication system of the Connection Machine the overhead in the communication is negligible. If $N_r \neq N_c$, then with $N_r > N_c$ $2^{n_r-n_c}$ rotations are performed in the direction of N_r for every rotation in the N_c direction. For $N_r < N_c$ the situation is the opposite. The arithmetic time is the same as in the case of a square mesh, but the communication time is $(\frac{P}{N_c} + \frac{R}{N_r})Q$.

In the case of *cyclic* assignment, it is clear that every real processor has the same number of virtual processors as with *consecutive* assignment. It can also be shown that the amount of work for each communication is the same. In addition to performing the work for each virtual processor, data for different virtual processors are also in the correct location for a multiply-add operation. The control structure is different from the *consecutive* assignment, however.

For local matrix multiplication, the issues with respect to performance are the traditional ones, with block methods being preferable for cache based architectures, and AXPY based algorithms preferable for pipelined, and register architectures. The performance of our implementation of Cannon's algorithm is extended to handle virtual processors is 5.2 Gflops with 256 virtual processors per real processor (or 4k x 4k matrices).

S_r	no. of block copy	no. of block additions
P_r	$\frac{R_r}{P_r}$	$\frac{Q_r}{P_r}$
Q_r	$\frac{R_r+P_r}{Q_r}$	none
R_r	$\frac{P_r}{R_r}$	$\frac{Q_r}{R_r}$

Table 2: Number of block copy and addition for one product matrix.

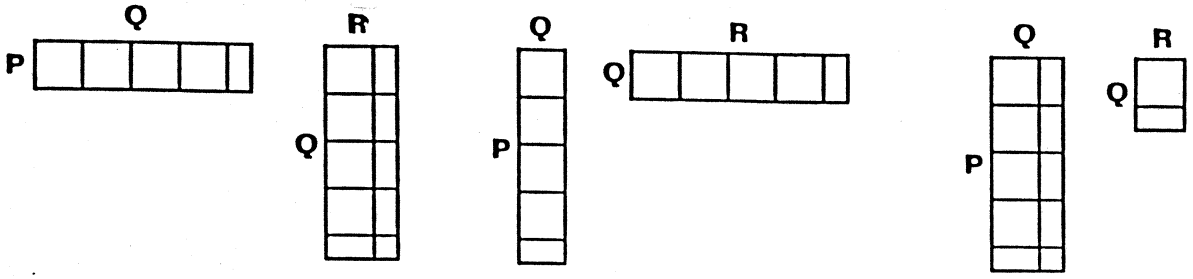

Figure 2: Multiplying arbitrary matrices on a large array

5.2.3 Small Matrices in a Large Mesh

This case is relevant not only for matrices with fewer elements than the number of processors, but also handles the case when P, Q, or R are not multiples of the number of processors in the respective direction. In the *cyclic* assignment there are $PmodN_r$ rows in the last row of "tiles" of B with rows being allocated in the direction of N_r. Similarly, there are $QmodN_c$ columns in the last columns of "tiles" of B. In the *consecutive* partitioning some processors have one more matrix element than others. Let $PmodN_r = P_r, QmodN_r = Q_r, QmodN_c = Q_c$, and $RmodN_c = R_r$. Assume $N_r = N_c$. Since $P_r, Q_r, R_r < N_r$ there are more processors than matrix elements, and there is a possibility to parallelize the loop in the Q direction.

Operands are partitioned into squares of side $S_r = min(P_r, Q_r, R_r)$. There is a total of $\lceil \frac{P}{S_r} \rceil \lceil \frac{Q}{S_r} \rceil \lceil \frac{R}{S_r} \rceil$ such squares. Each such processing square receives a $S_r \times S_r$ block from B and C through a copy operation. The number of block copy and additions are summarized in Table 2.

All multiplication on blocks of size $S_r \times S_r$ are performed concurrently. Each block matrix multiplication uses the kernel routine and requires S_r multiplication steps. In general, a summation of corresponding elements in some of the blocks are necessary to complete the computation. For instance, if $P_r = 1, R_r = 1$ and $Q_r > 1$, then the computation is an inner-product. The blocks are of size 1×1 and a summation of the product of all blocks is required. Figure 2 illustrates the three different situations summarized in Table 2.

The number of processors used in the block algorithms is $\frac{PQR}{S_r}$. If the total number of processors can be partioned into several such sets of subarrays, say M, then the loop on Q_r can be subdivided further at most that many times. The

loop on Q_r is parallelized in the algorithms above, except if $S_r = Q_r$. This further parallelization of the multiplication is obtained by creating multiple instances of the algorithms just described by copying, and by rotating the different copies $k\frac{S_r}{M}, k = \{1, 2, \ldots, M-1\}$ steps with respect to the "original". This rotation can be done while copying. Multiplication then proceeds in $\frac{S_r}{M}$ steps, followed by block addition of all copies.

If $N_r \neq N_c$, then Q_r and Q_c may not be the same. However, the number of rotations required for the direction with the larger number of processors is higher. For every rotation in the direction of $min(N_r, N_c)$, $\frac{max(N_r,N_c)}{min(N_r,N_c)}$ rotations are required in the other direction. This in effect guarantees that after the phase where all "complete" layers are treated for the direction with the maximum number of layers, the remainder in the two directions are the same.

6 Summary

The Data Parallel programming model provides a simple conceptual framework for programming highly concurrent architectures. The model also provides powerful instructions. Indeed, several of the level-1 BLAS routines are single instructions. None of the routines require loop constructs. Level-2 BLAS requires two or three instructions; with the exception of the triangular system solvers, no loop constructs are required. If the inverse of the triangular matrix is available instead of the matrix, then the triangular solve can be accomplished in one or two instructions. Otherwise, a single loop is required.

Level-3 BLAS requires three nested loops in Fortran 77. In a data parallel programming model a single loop is required, in general. Indeed, matrix multiplication can be performed without loops, if there is a sufficiently large number of processors. In this paper, we have used a simple mesh algorithm as the kernel. This kernel requires a single loop, and the communication operation is single step rotations, ignoring the initial data alignment. The kernel executes at a rate of 5.2 Gflops on the Connection Machine model CM-2. The loop is in the Q direction for multiplying a $P \times Q$ and a $Q \times R$ matrix. This loop can be parallelized given that there is a sufficient number of processors. Parallelizing this loop implies the introduction of segmented copy-scans and plus-scans. Matrix multiplication can be performed without loops in the data parallel model of computing for sufficiently many processors.

References

[1] L.E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, 1969.

[2] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM J. Computing*, 10:657–673, 1981.

[3] Jack J. Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. *Preliminary Proposal for a Set of Level 3 BLAS.* Technical Report Technical

Memorandum, Argonne National Laboratories, Mathematics and Computer Science Division, January 1987.

[4] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. *An Extended Set of Fortran Basic Linear Algebra Subprograms.* Technical Report Technical Memorandum 41, Argonne National Laboratories, Mathematics and Computer Science Division, November 1986.

[5] Geoffrey C. Fox, S.W. Otto, and A.J.G. Hey. *Matrix Algorithms on a Hypercube I: Matrix Multiplication.* Technical Report Caltech Concurrent Computation Project Memo 206, California Institute of Technology, dept. of Theoretical Physics, October 1985.

[6] Donald E. Heller. *Partitioning Big Matrices for Small Systolic Arrays*, pages 185–199. Prentice-Hall, 1985.

[7] W. Daniel Hillis. *The Connection Machine.* MIT Press, 1985.

[8] William Jalby and Ulrike Meier. *Optimizing Matrix Operations on a Parallel Multiprocessor with a Memory Hierarchy.* Technical Report , Univ. of Illinois, Center for Supercomputer Research and Development, February 1986.

[9] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, 4(2):133–172, April 1987. (Report YALEU/DCS/RR-361, January 1985).

[10] S. Lennart Johnsson. *Computational Arrays for Band Matrix Equations.* Technical Report 4287:TR:81, Computer Science, California Institute of Technology, May 1981.

[11] S. Lennart Johnsson. Highly concurrent algorithms for solving linear systems of equations. In *Elliptic Problem Solving II*, Academic Press, 1983.

[12] S. Lennart Johnsson. Vlsi algorithms for doolittle's, crout's and cholesky's methods. In *International Conference on Circuits and Computers 1982, ICCC82*, pages 372–377, IEEE, Computer Society, September 1982.

[13] S. Lennart Johnsson and Ching-Tien Ho. Matrix multiplication on boolean cubes using generic communication primitives. In *Parallel Processing and Medium Scale Multiprocessors*, SIAM, 1987. (Presented at the ARMY workshop on Medium Scale Parallel Processing, Stanford University, January 1986, Report YALEU/DCS/RR-530, March 1987).

[14] S. Lennart Johnsson and Ching-Tien Ho. Matrix transposition on boolean n-cube configured ensemble architectures. *SIAM J. on Algebraic and Discrete Methods*, . To appear. YALE/DCS/RR-572. (Revised edition of YALEU/DCS/RR-494 November 1986.).

[15] H.T. Kung and Charles E. Leiserson. *Algorithms for VLSI Processor Arrays*, pages 271–292. Addison-Wesley, 1980.

[16] C. L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM TOMS*, 5(3):308–323, September 1979.

[17] Uri Weiser and Al Davis. *Mathematical Representation for VLSI Arrays.* Technical Report UUCS-80-111, Universit of Utah, Department of Computer Science, September 1980.