Scientific Software Libraries for Scalable Architectures

Scientific Software Libraries for Scalable Architectures

S. Lennart Johnsson
Kapil K. Mathur

TR-19-94

August 1994

Parallel Computing Research Group

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

To appear in *Parallel Scientific Computing*, Springer–Verlag.

Scientific Software Libraries for Scalable Architectures

S. Lennart Johnsson
Thinking Machines Corp.
and
Harvard University

and

Kapil K. Mathur
Thinking Machines Corp.

Abstract

Massively parallel processors introduce new demands on software systems with respect to performance, scalability, robustness and portability. The increased complexity of the memory systems and the increased range of problem sizes for which a given piece of software is used, poses serious challenges to software developers. The Connection Machine Scientific Software Library, CMSSL, uses several novel techniques to meet these challenges. The CMSSL contains routines for managing the data distribution and provides data distribution independent functionality. High performance is achieved through careful scheduling of arithmetic operations and data motion, and through the automatic selection of algorithms at run–time. We discuss some of the techniques used, and provide evidence that CMSSL has reached the goals of performance and scalability for an important set of applications.

1 Introduction

The main reason for large scale parallelism is performance. In order for massively parallel architectures to deliver on the promise of extreme performance compared to conventional supercomputer architectures, an efficiency in resource use close to that of conventional supercomputers is necessary. Achieving high efficiency in using a computing system is mostly a question of efficient use of its memory system. This is the premise on which the Connection Machine Scientific Software Library [28], the CMSSL, is based.

Another premise for the CMSSL is the notion of scalability. Software systems must be designed to operate on systems and data sets that may vary in size by as much as four orders of magnitude. This level of scalability with respect to computing system size must be accomplished transparently to the user, i.e., the same program must execute not only correctly but also efficiently without change over this range in processing capacity and corresponding problem sizes. Moreover, programs should not have to be recompiled for various system sizes. This requirement will be even more important in the future, since over time the assignment of processing nodes to tasks is expected to become much more dynamic than today.

Robustness of software both with respect to performance and numerical properties are becoming increasingly important. The memory system in each node is becoming increasingly complex in order to match the speed of the memory system with that of an individual processor. The distributed nature of the total memory compounds the complexity of the memory system. It is imperative that software systems deliver a large fraction of the available performance over a wide range of problem sizes, transparently to the user. For instance, small changes in array sizes should not impact performance in a significant way. Robustness with respect to performance in this sense will increase the demands on the software systems, in particular on the run–time parts of the systems.

Robustness with respect to numerical properties is also becoming increasingly important. The

same software may be used for problem sizes over a very wide range. Condition numbers for the largest problems are expected to be significantly worse than for small problems. As a minimum, condition estimators must be provided to allow users to assess the numerical quality of the results. It will also be increasingly necessary to furnish software for ill–conditioned problems, and whenever possible, automatically choose an appropriate numerical method. Some parallel methods do not have as good a numerical behavior as sequential methods, and this disadvantage is often increasing with the degree of parallelism. The trade–off between performance and numerical stability and accuracy is very complex. Much research is needed before the choice of algorithm with respect to numerical properties and performance can be automated.

Portability of codes is clearly highly desirable in order to amortize the software investment over as large a usage as possible. Portability is also critical in a rapid adoption of new technology, thus allowing for early benefits from the increased memory sizes, increased performance, or decreased cost/performance offered by new technology. But, not all software is portable when performance is taken into account. New architectures, like MPPs, require new software technology that often lags the hardware technology by several years. Thus, it is important to exploit the architecture of software systems such that architecture dependent, nonportable software is limited to as few functions as possible, while maintaining portability of the vast amount of application software. One of the purposes of software libraries is to enable portability of application codes without loss of performance.

The Connection Machine Scientific Software Library today has about 250 user callable functions covering a wide range of frequent operations in scientific and engineering computation. In this paper we illustrate how the goals of high performance and scalability have been achieved.

The outline of the paper is as follows. In the next few sections we discuss memory systems for scalable architectures and their impact on the sequence to storage association used in mapping arrays to the memory system. We then discuss data representations for dense and sparse arrays. The memory system and the data representation defines the foundation for the CMSSL. We then present the design goals for the CMSSL and how these goals have been approached and achieved. The *multiple–instance* capability of the CMSSL is an extension of the functionality of conventional libraries in the spirit of array operations, and critical to the performance in computations on both distributed and local data sets. The multiple–instance feature is discussed in Section 6. Scalability and robustness with respect to performance both depend heavily on the ability to automatically select appropriate schedules for arithmetic/logic operations and data motion, and proper algorithms. These issues are discussed by specific examples. A summary is given in Section 8.

2 Architectural model

High performance computing has depended on elaborate memory systems since the early days of computing. The Atlas [23] introduced virtual memory as a means of making the main relatively slow memory appear as fast as a small memory capable of delivering data to the processor at its clock speed. Since the emergence of electronic computers processors have as a rule been faster than memories, regardless of the technology being used. Today, most computers, conventional supercomputers excepted, use MOS technology for both memories and processors. But, the properties of the MOS technology is such that the speed of processors is doubling about every 18 months, while the speed of memories is increasing at a steady rate of about 7%/yr.

Since the speed of individual memory units, today primarily built out of MOS memory chips, is very limited, high performance systems require a large number of memory banks (units), even when locality of reference can be exploited. High end systems have thousands to tens of

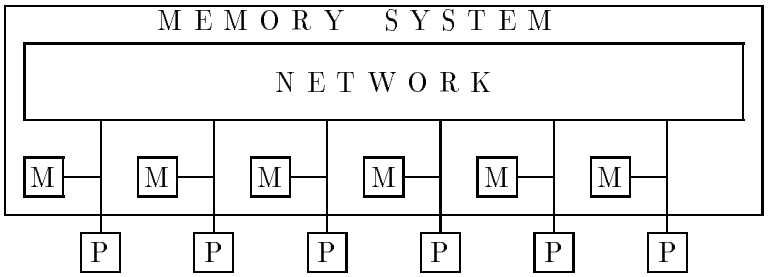

Figure 1: The memory system for distributed memory architectures.

thousands of memory banks. The aggregate memory bandwidth of such systems far exceeds the bandwidth of a bus. A network of some form is used to interconnect memory modules. The nodes in the network are typically of a low degree, and for most networks independent of the size of the network. A large variety of network topologies can be constructed out of nodes with a limited fixed degree. Massively parallel architectures have employed two– and three–dimensional mesh topologies, butterfly networks, binary cubes, complete binary trees, and fat–tree topologies.

The speed of the memory chips present the most severe restriction with respect to performance. The second weakest technological component is the communication system. Constructing a communication system with the capacity of a full crossbar with a bandwidth equal to the aggregate bandwidth of the memory system is not feasible for systems of extreme performance, and would represent a considerable expense even for systems where it may be technically feasible. Hence, with a constraint of a network with a lower bandwidth than that of the full memory system, in MPPs, processors are placed close to the memory modules such that whenever locality of reference can be exploited the potential negative impact upon performance of the limited network capacity is alleviated. This placement of processors and the limited network capacity has a fundamental impact upon the preferred sequence–to–storage association to be used for programming languages. This difference in preferred sequence–to–storage association is a major source of inefficiency in porting codes in conventional languages to MPPs.

The generic architectural model for MPPs used throughout this paper is shown in Figure 1. The local memory system is shown only schematically. As a minimum, the local memory hierarchy consists of a processor register file and DRAM, but quite often there is at least one level of cache, and sometimes two levels. In systems without a cache, such as the Connection Machine systems, the *mode* in which the DRAM is operated is important. In addition to the local memory hierarchy, the access time to the memories of other *nodes* (a processor with associated memory modules and network interface hardware) often is nonuniform. The source of the nonuniformity in access time may be directly related to the distance in the network, which is the case for packet switched communication systems. In circuit switched and wormhole routing systems, the distance in itself is often insignificant with respect to access time. However, the longer the routing distance the more likely it is that contention in the network will arise, and hence add to the remote access time.

3 Dimensionality of the address space

The one–dimensional address space used for the conventional languages is not suitable for most applications on MPPs. A linearized address space may also result in poor performance for multidimensional arrays or nonunit stride accesses in banked and interleaved memory systems. So–called bank conflicts are well known performance limitations caused by the combination of

data allocation strategies and access strides. For MPPs, for computations with a uniform use of the address space, a multi–dimensional address with as many dimensions as are being accessed uniformly is ideal. We discuss this claim through a number of simple, but important, examples. We first discuss computations dominated by operations on a single array, then consider operations involving two or three arrays.

3.1 The Fast Fourier Transform

For the fast Fourier Transform, the FFT, and many other hierarchical or divide–and–conquer methods, an address space with $\log_2 N$ dimensions may be ideal even for a one–dimensional array of extent N. All data references are to data within unit distance in such an address space. This view is particularly useful in the mapping of the arrays to networks of memory units, since it properly models the communication needs.

The FFT computations are uniform across the index space and the load–balance is independent of whether cyclic or consecutive allocation is used. However, the cyclic data allocation yields lower communication needs than the consecutive allocation by up to a factor of two for unordered transforms [13, 14]. The reason is that the computations of the FFT always proceed from the high to the low order bit in the index space. With the consecutive allocation the high order bits are associated with processor addresses and must be mapped to local memory addresses before local butterfly computations can be performed. Conserving memory in this remapping means that another remapping is required when the computations are to be performed on the dimensions that were moved from local memory to processor addresses in order to accommodate the move of the leading dimensions into local memory. In the cyclic allocation the leading dimensions are mapped to local memory from the start.

3.2 Direct methods for solution of systems of equations

LU and QR factorization only involves a single array, while the solution of triangular systems involves two or three arrays. Two important distinguishing features of dense matrix factorization are that all data references are "global", and that the computations are performed on a diminishing set of indices. The "global" references consists of pivot selection and the broadcast of the pivot row and column. If a block algorithm is used, sets of rows and columns are treated together, but it does not fundamentally change the reference pattern for the algorithm.

We will first discuss the preferred dimensionality and shape of the address space, then load–balancing. Since the broadcast operations are performed both along rows and columns a one–dimensional partitioning makes one of these operations local, while for the other a complete pivot row or column must be broadcast. With a consecutive data allocation and a $\sqrt{N} \times \sqrt{N}$ nodal array for the factorization of a $P \times P$ matrix, the broadcast operations require the communication of $2 \times \frac{P}{\sqrt{N}}$ elements, instead of P elements for a one–dimensional partitioning. This argument is too simplistic in that the communication among the two axes is not the same. But, the conclusion is correct: a two–dimensional address space is desirable, and the shape of the local subgrid shall be close to square. Partial pivoting requires additional communication along one axis. Second, since not all indices are involved in all steps, the number of elements per node participating in the broadcast operation is not necessarily $\frac{P}{\sqrt{N}}$ and P, respectively. It depends upon what techniques are used for load–balancing as discussed in [19].

Note that for out–of–core factorization algorithms using panel techniques with entire columns in primary storage, the shape of the panel to be factored may be extremely rectangular. Hence, the shape of the processing array shall also be extremely rectangular to yield almost square subgrids.

A cyclic allocation guarantees good load–balance for computations such as LU and QR factorization, and triangular system solution. But, a good load–balance can be achieved also for consecutive mapping by adjusting the elimination order accordingly [19]. To allow for the use of level–2 LBLAS (Local BLAS [16]), blocking of rows and columns on each node is used. In LU factorization a blocking of the operations on b rows and columns means that b rows are eliminated at a time from all the other rows. The resulting *block–cyclic elimination order* yields the desired load–balance as well as an opportunity to conserve local memory bandwidth. A block–cyclic elimination order was first recommended in [7] for load–balanced solution of banded systems.

The result of the factorization is not two block triangular matrices, but *block–cyclic triangles*. A block–cyclic triangle can be permuted to a block triangular matrix. However, it is not necessary to carry out this permutation for the solution of the block–cyclic triangular system of equations. Indeed, it is desirable to use the block–cyclic triangle for the forward and back substitutions, since the substitution process is load–balanced for the block–cyclic triangles. Using block triangular matrices stored in a square data array (A) allocated to nodes with a consecutive data allocation scheme would result in poor load–balance. For details as well as modifications necessary for rectangular nodal arrays, see [19].

Note further that for triangular solvers the communication is again of the global nature, and the conclusions about the shape of the address space still applies [8, 19].

3.3 The Alternating Direction Implicit Method

In the Alternating Direction Implicit (ADI) Methods a multi–dimensional operator is factored into one–dimensional operators that are applied in alternating order. In its most common use, tridiagonal systems of equations are solved along each coordinate direction of a grid. Whether substructured elimination or straight elimination is used, the communication requirements along each coordinate axis is proportional to the area of the surface having the normal aligned with the axis of solution. Hence, regardless of the extent of the axes in the different dimensions, it is again desirable with respect to minimizing nonlocal references to minimize the surface area of the subgrids assigned to each node. For a more detailed discussion of ADI on parallel computers and cyclic reduction based methods as well as Gaussian elimination based methods for the solution of tridiagonal systems of equations see [12, 17].

3.4 Stencil computations

For stencil computations on three–dimensional arrays with a stencil symmetric with respect to the axis, the well known minimum "surface–to–volume" rule dictates that a three–dimensional address space shall be used for optimum locality of reference. For example, for a $512 \times 512 \times 512$ grid distributed evenly across 512 nodes, each node holds 256k grid points. With a 7–point, centered, symmetric stencil in three dimensions, the number of nonlocal grid points that must be referenced is $6 \times 64 \times 64$ for cubic subgrids of shape $64 \times 64 \times 64$. For the standard linearized array mapping used by Fortran 77 or C, the subgrids will be of shape $1 \times 512 \times 512$. References along two of the axis are entirely local, but the references along the third axis require access to $2 \times 512 \times 512$ nonlocal grid points. Thus, the linearized address space requires a factor of $\frac{1}{3} \times 8 \times 8 \approx 21$ more nonlocal references for the stencil computations.

Note that if the data array is of shape $256 \times 1024 \times 512$, it is still the case that the ideal local subgrid is of shape $64 \times 64 \times 64$. But, the ideal shape of processing nodes have changed from an $8 \times 8 \times 8$ array to a $4 \times 16 \times 8$ array. This example with simple stencil computations on a three–dimensional array has shown that a multi–dimensional address space is required in order to maximize the locality of reference. Moreover, the example also shows that the shape of the

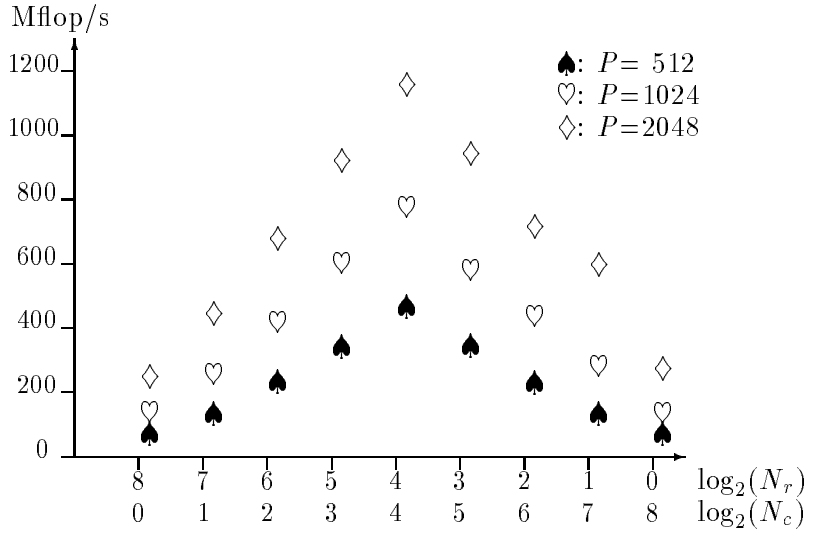

Figure 2: Influence of shared processor configuration on the performance for multiplication of square matrices of size P, 64–bit precision. The shape of the 256 processor Connection Machine system CM–200 is $N_r \times N_c = 256$.

address space, i.e., how the indices for each axis is split between a *physical* or *processor address* and a *local* memory address is very important. We have implicitly assumed a *consecutive* or *block* partitioning in the discussion above. A *cyclic* partitioning would in fact maximize the number of nonlocal references.

3.5 Matrix multiplication

We restrict the discussion to the computation $C \leftarrow C + A \times B$. In order to minimize the communication, a good strategy is to keep the matrix with the largest number of elements stationary [9, 21]. The other two operands are moved as required for the required indices of the three operands to be present in a given node at the same time. With this underlying strategy, the ideal shape of the address space is such that the stationary matrix has square submatrices in each node [9, 21]. This result can be derived from that fact that the required communication is all–to–all broadcast and/or reduction within rows or columns [20].

The ideal shape of the address space has been verified on the Connection Machine system CM–200 and is illustrated in Figure 2. It confirms that the optimal nodal array shape is square for square matrices. For the matrix shapes used in this experiment, a one–dimensional nodal array aligned with either the row or column axis, requires about a factor of six higher execution time than the ideal two–dimensional nodal array shape.

With the proper nodal array shape a superlinear speedup is achieved for matrix multiplication, since the communication requirements increase in proportion to the matrix size, while the computational requirements grow in proportion to the size to the $\frac{3}{2}$ power. The superlinear speedup achieved on the CM–5 is shown in Table 1.

4 Data representation

In the previous section we showed that linearized address spaces as used by the conventional languages is not compatible with the notion of locality of reference for MPP memory systems. We showed that multi–dimensional address spaces are required, and that the optimal shape of the address space can be derived from the data reference pattern. In this section we focus on the data

Number of Nodes	Matrix Size (P)	Mflop/s Overall	MFlop/s per node	Size for half peak perf. ($P_{1/2}$)
1	1088	62	62	128
32	5952	2266	71	1088
64	8704	4608	72	1408
128	12416	9933	78	1664
256	17664	19661	77	2816
512	24576	42395	83	3328

Table 1: Performance of matrix multiplication in Mflop/s on the CM–5.

representation and how, based on the selected representation, the desired data allocation can be realized. With the exception of the FFT, in all our examples a consecutive data allocation is either preferred, or the choice between cyclic and consecutive allocation immaterial with respect to performance. Thus, for simplicity, we will assume a consecutive data allocation in this section.

4.1 Dense arrays

For the allocation of dense arrays we have seen that subgrids with equal axes extents are either optimal or close to optimal for several common and important computations. Hence, as a default without sophisticated data reference analysis, an allocation creating subgrids with axes extents of as equal a length as possible is sensible and feasible.

4.2 Grid Sparse Matrices

For sparse arrays the data representation is less obvious, even for sparse matrices originating from regular grids. Such matrices typically consists of a number of nonzero diagonals. For instance, consider the case with a 7–point centered difference stencil in three dimensions. The stencil computation can be represented as a matrix–vector multiplication, $y \leftarrow Ax$, where x and y are grid point values and the matrix A represents the stencils at all the grid points. With an $N_1 \times N_2 \times N_3$ grid, with stride one along the axis of extent N_3 and stride N_3 along the axis of length N_2, the matrix is of shape $N_1N_2N_3 \times N_1N_2N_3$ with a nonzero main diagonal, a nonzero diagonal immediately above and below the main diagonal, two nonzero diagonals at distance N_3 above and below the main diagonal, and two nonzero diagonals at distance N_2N_3 above and below the main diagonal.

A common representation in Fortran 77 is either to use a set of one–dimensional arrays, one for each nonzero diagonal or a single one–dimensional array with the nonzero diagonals appended to each other. However, either of these representations are not suitable for MPP memory systems, since preservation of locality of reference for matrix–vector multiplication is likely to be lost.

A natural representation for grid–sparse matrices and grid–point vectors is to tie the representation directly to the grid rather than the matrix representation of the grid. Grid–point vectors are represented as multi–dimensional arrays with one axis for each axis of the grid, plus an axis for the grid–point vector. The grid–axes extents are the same as the lengths of the corresponding grid axes. A grid–sparse matrix is represented in an analogous way. The matrix represents interaction between variables in different grid points. As an example of the grid based representation of a grid–sparse matrix we consider the common 7–point stencil in three dimensions. Each of the stencil coefficients are represented as three–dimensional arrays A,$\cdots$,G, of nodal values of shape (LX,LY,LZ).

No. of partitions	No. of shared edges	% of total	No. of shared nodes	% of total
8	5186	2.4	2735	13.4
16	8005	3.7	4095	20.1
32	11553	5.3	5747	28.2
64	16055	7.3	7721	37.9
128	21502	9.8	9827	48.2

Table 2: Partitioning of a tetrahedral mesh between concentric spheres.

The corresponding vectors for the operation $y = Ax$ may be represented as X(LX,LY,LZ), Y(LX,LY,LZ), and the computation $y = Ax$ as

$$\begin{aligned} Y(x,y,z) &= A(x,y,z)X(x,y,z) + B(x,y,z)X(x-1,y,z) + C(x,y,z)X(x+1,y,z) + \\ &\quad + D(x,y,z)X(x,y-1,z) + E(x,y,z)X(x,y+1,z) + \\ &\quad + F(x,y,z)X(x,y,z-1) + G(x,y,z)X(x,y,z+1) \end{aligned}$$

5 Representation and allocation of arbitrary sparse matrices

The representation and allocation of arbitrary sparse matrices is a very difficult topic subject to research. Two general partitioning techniques of significant recent interest are the recursive spectral bisection technique proposed by Pothen *et al.* [25] and the geometric approach proposed by Miller *et al.* [24]. The recursive spectral bisection technique has been used successfully by Simon [26] for partitioning of finite volume and finite element meshes. A parallel implementation of this technique has been made by Johan [4].

The spectral partitioning technique is based on the eigenvector corresponding to the smallest nonzero eigenvalue of the Laplacian matrix associated with the graph to be partitioned. The Laplacian matrix is constructed such that the smallest eigenvalue is zero and its corresponding eigenvector consists of all ones. The eigenvector associated with the smallest nonzero eigenvalue is called the *Fiedler vector* [3]. Grid partitioning for finite volume and finite element methods is often based on a dual mesh representing finite volumes or elements and their adjacencies (or some approximation thereof) rather than the graph of nodal points [4]. One advantage of the spectral bisection technique is that it is based on the topology of the graph underlying the sparse matrix. It requires no geometric information. However, it is computationally quite demanding. The results of applying the spectral bisection technique to a model problem is reported in [4] and shown in Table 2. A planar grid of tetrahedra between concentric cylinders. with 20,374 nodes, 107,416 tetrahedra, and 218,807 faces is partitioned using the spectral bisection algorithm. The numbers of shared nodes and edges as a function of the number of partitions are given in the table.

The results of applying the spectral bisection technique on a more realistic finite element application [6] are summarized in Table 3. The spectral bisection technique in this example offered a reduction in the number of remote references by a factor of 13.2. The speedup for the gather operation was a factor of 13 and of the scatter operation the speedup was a factor of 9.6 (the scatter operation includes the time required for addition which is unaffected by the partitioning).

Another important aspect of computations with arbitrary sparse matrices is that unlike for dense and grid sparse matrices, address computations cannot be performed by incrementing addresses using fixed strides. For arbitrary sparse matrices, indirect addressing is required. It frequently is the most time consuming part on uniprocessors. On a distributed memory machine, the address

Operation	Standard allocation	Spectral bisection
Partitioning	—	66
Gather	298	23
Scatter	445	46
Computation	180	181
Total time	923	316

Table 3: Gather and scatter times in seconds on a 32–node CM–5 for 50 time steps with a 1–point integration rule for finite element computations on 19,417 nodes and 109,914 elements.

computations do not only involve the computation of local addresses, but routing information as well. In an iterative (explicit) method, the underlying grid may be fixed for several or all iterations. For such computations it is important with respect to performance to amortize the cost of computing the addresses over as many iterations as possible. Cacheing this information and reusing it later is important for performance [28].

In an arbitrary sparse matrix, there is no simple way of encoding the global structure. Yet, arbitrary sparse matrices may still have some local structure resulting in a block sparse matrix. Taking advantage of such a block structure for both economy in data representation, data storage and efficiency of operations, is significantly simplified by explicitly representing the blocks [28].

6 Multiple–instance computation

The multiple–instance capability of the CMSSL is consistent with the idea of collective computation inherent in languages with an array syntax. We have already seen how it arises naturally in the ADI method. CMSSL routines are designed to carry out a collection of high level computations on independent sets of operands in a single call, in the same way addition of arrays are carried out through a single statement, or intrinsic functions are applied to each entry in an array in Fortran 90. To accomplish the same task in an F–77 or C library, the call to a library routine would be embedded in a set of nested loops. The multiple–instance capability not only eliminates loop nests, but also allows for parallelization and optimization without a sophisticated interprocedural data dependence analysis. The multiple–instance feature for parallel computation is necessary for the desired degree of optimization, which goes beyond the capabilities of state–of–the–art compiler and run–time systems.

We discuss the significance of the multiple–instance capability with respect to performance and simplicity of user code by considering the computation of the FFT along one of the axes of a two–dimensional array of shape $P \times Q$. We assume a *canonical data layout* in which the set of processing nodes are configured as an array of the same rank as the data array and of a shape making the local subarrays approximately square. The nodal array shape is $N_r \times N_c (= N)$.

With the FFT performed along the P–axis, the computations on the two–dimensional array consist of Q independent FFT computations, each on P data elements. We consider three different alternatives for the computation:

1. Maximize the concurrency for each FFT through the use of a canonical data layout for one–dimensional arrays of size P.

2. Compute each FFT without data relocation.

3. Compute all Q FFTs concurrently through multiple–instance routines.

Alternative 1 corresponds to the following code fragments:

```
FOR J = 1 TO Q DO
    TEMP = A(:,J)
    CALL FFT1(TEMP,P)
    A(:,J) = TEMP
ENDFOR

SUBROUTINE FFT1(B,N)
ARRAY B(N)
FFT on a one–dimensional array
END FFT1
```

The concurrency in the computation of the FFT is maximized. The data motion prior to the computation of the FFT on a column is a *one–to–all personalized communication* [11]. The data redistribution corresponds to a change in data allocation from A(:,:) to TEMP(:) and back to the original allocation, one column at a time. The arithmetic speedup is limited to $\min(N, P)$ for transforms on the P–axis.

In alternative 2, the data redistribution is avoided by computing each instance in–place. An obvious disadvantage with this approach is the poor load–balance. The speedup of the arithmetic is proportional to $\min(N_r, P)$ for a transform along the P–axis.

```
FOR J = 1 TO Q DO
    CALL FFT2(A,P,Q,J)
ENDFOR

SUBROUTINE FFT2(B,N,M,K)
ARRAY B(N,M)
In–place FFT on column K of array B
END FFT2
```

Finally, using the CMSSL FFT corresponds to Alternative 3. All different instances of the FFT represented by the Q columns are treated in–place in a single call. The concurrency and data layout issues are managed inside the FFT routine. The CMSSL call is of the form CALL FFT(A, DIM = 1), where DIM specifies the axis of the array A subject to transformation. The actual CMSSL call has additional parameters allowing the calling program to define the subset of axes for which forward transforms are desired, for which axes inverse transforms are desired, and for which axes ordered transforms are desired [28].

```
FORALL J DO
    CALL FFT2(A(:,J))
ENDFOR
```

The third choice is clearly preferable both with respect to communication and arithmetic load–balance. Note that with a single–instance library routine and canonical layouts, Alternative 1 would be realized. Further, for particular situations, a noncanonical layout will alleviate the communication problem, but in many cases the communication appears somewhere else in the application code. Thus, we claim that our discussion based on canonical layouts reflects the situation in typical computations.

Operation	Mflop/s per node	Efficiency %
Local		
ℓ_2–norm	126	98
Matrix–vector	115	90
Matrix–matrix	115	90
Global		
ℓ_2–norm	126	98
Matrix–vector	80	63
Matrix–matrix	83	65
LU–factorization	61	48
Unstructured grid	26	20

Table 4: Peak local and global performance per node and efficiencies achieved for a few different types of computations on the CM–5. 64–bit precision.

7 CMSSL

The primary design goal for the Connection Machine Scientific Software Library, CMSSL, is to provide high level support for most numerical methods, both traditional and recently developed methods, such as hierarchical and multi–scale methods and multipole and other fast so–called N-body algorithms used for large scale scientific and engineering computations. High level support in this context means functionality that is at a sufficiently high level that architectural characteristics are essentially transparent to the user, yet that a high performance can be achieved. Specific design goals for the CMSSL include consistency with languages with an array syntax, such as Fortran 90, Connection Machine Fortran and C*, functionality that is independent of data distribution, multiple instance capability, support for all four conventional floating–point data types, high performance, scalability across system and problem sizes, robustness, portability, and functionality supporting traditional numerical methods. These goals have had an impact on the architecture of the CMSSL. The first few goals have also impacted the user interfaces. Version 3.2 of CMSSL has about 250 user callable functions. The library exists on the Connection Machine systems CM–2, CM–200, and CM–5. The CM–5 version consists of about 0.5 million lines of code, and so does the CM–2 and CM–200 version.

Table 4 gives a few examples of how the goal of high performance is met by the CMSSL. The table entry for unstructured grid computations actually represent complete applications [27, 5, 22, 1], while the other entries represent library functions by themselves. Table 5 provides excellent data of how the goal of scalability has been met by the CMSSL, as well as the CM–5 architecture over a range of a factor of a thousand in system size. ENSA[1] is an Euler and Navier Stokes finite element code [5], while TeraFrac[2] and MicMac[3] are solid mechanics finite element codes ([22, 1]). To first order, the performance per node is independent of the system size, thus demonstrating excellent scalability. For some computations, like matrix multiplication, the efficiency actually increases as a function of system size. For the unstructured grid computations the performance decreases by about 5%, an insignificant amount.

With respect to scientific and engineering computations, the architectural dependence on traditional architectures has mostly been captured in a set of matrix utilities known as the BLAS (Basic Linear Algebra Subprograms) [2, 18]. Efficient implementations of this set of routines

[1] Developed at the Division of Applied Mechanics, Stanford University

[2] Developed at the Division of Engineering, Brown University and Tech. Univ. Denmark

[3] Developed at the Department of Mechanical Engineering, Cornell University

Number	Dense matrix operations				Unstructured grid computations		
Nodes	ℓ_2–norm	MV	MM	LU-fact	ENSA	TeraFrac	MicMac
1	126	83	62	68			
32	126	80	71	61	25	26	30
64	125	74	72	60	25	26	31
128	125	76	78	60	26	24	29
256	125	68	77	59	24	25	32
512	125	68	83	59	24	25	32
1024				58		26	

Table 5: Performance in Mflop/s per node over a range of CM–5 system sizes. 64–bit precision.

are architecture dependent, and for most architectures is written in assembly code. Most scientific codes achieve high performance when built on top of this set of routines. On distributed memory architectures a distributed BLAS [10, 21, 15], DBLAS, is required in addition to a local BLAS, LBLAS, in each node [16]. Moreover, a set of communication routines are required for data motion between nodes. But, not all algorithms parallelizes well, and there is an algorithmic architectural dependence. Thus, architectural independence of application programs requires higher level functions than the DBLAS, LBLAS, and communication routines. Hence, the CMSSL includes a subset of functions corresponding to traditional libraries, such as Linpack, Eispack, LApack, FFTpack and ITpack.

The external architecture of the CMSSL is similar to conventional library systems in that there exists a set of matrix utilities similar to the BLAS, a set of sparse matrix utilities supporting operations on regular and irregular grids, dense and banded direct solvers, and iterative solvers. Fast Fourier transforms are supported for multidimensional transforms. In addition, CMSSL also includes a few statistical routines, and a routine for integration of systems of ordinary differential equations, and a simplex routine for dense systems. The CMSSL also contains a communications library. Such libraries are unique to distributed memory machines. The CMSSL also contains tools in the form of two special compilers, a stencil compiler and a communications compiler. Novel ideas in the CMSSL can be found at all levels: in the internal architecture, in the algorithms used, in the automatic selection of algorithms at run–time, and in the local operations in each node.

The CMSSL is a "global" library. It accepts global, distributed data structures. Internally, the CMSSL consists of a set of library routines executing in each node and a set of communication functions. The communication functions are either part of the Connection Machine Run–Time System, or part of the CMSSL. All communication functions that are part of the CMSSL are directly user accessible, and so are the functions in each node. For the global library, these functions are called directly and are transparent to the user and the distributed nature of the data structures is transparent to the user. The internal structure of the CMSSL supports data distribution independent functionality, automatic algorithm selection for best performance for the BLAS, FFT and a few other functions as well as user specified choices for many other functions. The execution is made through calls to local routines and communication functions. It follows from the internal architecture of the CMSSL, that it also has the ability to serve as a nodal library.

8 Summary

The CMSSL has been designed for performance, scalability, robustness and portability. The architecture with respect to functionality follows the approach in scientific libraries for sequential architectures. Internally, the CMSSL consists of a nodal library and a set of communication and data distribution functions. CMSSL provides data distribution independent functionality and has logic for automatic algorithm selection based on the data distribution for input and output arrays and a collection of algorithms together with performance models.

The performance goals have largely been achieved both for the local and global functions. Particular emphasis has been placed on reducing the problem sizes offering half of peak performance. Some peak global performance data were given in Section 7. Scalability is excellent. The performance per node has been demonstrated to be largely independent of the number of nodes in the systems over a range of a factor of one thousand (Table 5, Section 7).

Robustness with respect to performance is achieved through the automatic selection of algorithm as a function of data distribution for both low level and high level functions.

CMSSL offers portability of user codes without loss of performance. CMSSL itself has an architecture amenable to portability. It is the same on all Connection Machine platforms. Code for maximum exploitation of the memory hierarchy is in assembly language, and thus has limited portability. Some algorithmic changes were also necessary in porting the library to the CM–5. These changes are largely due to the differences in the communication systems, but also due to the MIMD nature of the CM-5.

Acknowledgement

Many people have contributed to the CMSSL. We would like to acknowledge the contributions of Paul Bay, Jean-Philippe Brunet, Steven Daly, Zdenek Johan, David Kramer, Robert L. Krawitz, Woody Lichtenstein, Doug MacDonald, Palle Pedersen, and Leo Unger all of Thinking Machines Corp., and Ralph Brickner and William George of Los Alamos National Laboratories, Yu Hu of Harvard University, Michel Jacquemin of Yale University, Lars Malinowsky of the Royal Institute of Technology, Stockholm, and Danny Sorensen of Rice University.

The communications functions and some of the numerical routines in the CMSSL relies heavily on algorithms developed under support of the ONR to Yale University under contracts N00014-84-K-0043, N00014-86-K-0564, the AFOSR under contract AFOSR-89-0382 to Yale and Harvard Universities, and the NSF and DARPA under contract CCR-8908285 to Yale and Harvard Universities. Support for the CMSSL has also been provided by ARPA under a contract to Yale University and Thinking Machines Corp.

References

[1] A. J. Beaudoin, P. R. Dawson, K. K. Mathur, U.F. Kocks, and D. A. Korzekwa. Application of polycrystal plasticity to sheet forming. *Computer Methods in Applied Mechanics and Engineering*, in press, 1993.

[2] Jack J. Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. Technical Report Reprint No. 1, Argonne National Laboratories, Mathematics and Computer Science Division, August 1988.

[3] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25:619–633, 1975.

[4] Zdenek Johan. *Data Parallel Finite Element Techniques for Large–Scale Computational Fluid Dynamics*. PhD thesis, Department of Mechanical Engineering, Stanford University, 1992.

[5] Zdenek Johan, Thomas J.R. Hughes, Kapil K. Mathur, and S. Lennart Johnsson. A data parallel finite element method for computational fluid dynamics on the Connection Machine system. *Computer Methods in Applied Mechanics and Engineering*, 99(1):113–134, August 1992.

[6] Zdenek Johan, Kapil K Mathur, S. Lennart Johnsson, and Thomas J.R. Hughes. An efficient communication strategy for Finite Element Methods on the Connection Machine CM–5 system. *Computer Methods in Applied Mechanics and Engineering*, 113:363–387, 1994.

[7] S. Lennart Johnsson. Fast banded systems solvers for ensemble architectures. Technical Report YALEU/DCS/RR-379, Dept. of Computer Science, Yale University, March 1985.

[8] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distributed Computing*, 4(2):133–172, April 1987.

[9] S. Lennart Johnsson. Minimizing the communication time for matrix multiplication on multiprocessors. *Parallel Computing*, 19(11):1235–1257, 1993.

[10] S. Lennart Johnsson. *Parallel Architectures and their Efficient Use*, chapter *Massively Parallel Computing: Data distribution and communication*, pages 68–92. Springer Verlag, 1993.

[11] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.

[12] S. Lennart Johnsson and Ching-Tien Ho. Optimizing tridiagonal solvers for alternating direction methods on Boolean cube multiprocessors. *SIAM J. on Scientific and Statistical Computing*, 11(3):563–592, 1990.

[13] S. Lennart Johnsson, Ching-Tien Ho, Michel Jacquemin, and Alan Ruttenberg. Computing fast Fourier transforms on Boolean cubes and related networks. In *Advanced Algorithms and Architectures for Signal Processing II*, volume 826, pages 223–231. Society of Photo-Optical Instrumentation Engineers, 1987.

[14] S. Lennart Johnsson, Michel Jacquemin, and Robert L. Krawitz. Communication efficient multi-processor FFT. *Journal of Computational Physics*, 102(2):381–397, October 1992.

[15] S. Lennart Johnsson and Kapil K. Mathur. Distributed level 1 and level 2 BLAS. Technical report, Thinking Machines Corp., 1992. In preparation.

[16] S. Lennart Johnsson and Luis F. Ortiz. Local Basic Linear Algebra Subroutines (LBLAS) for distributed memory architectures and languages with an array syntax. *The International Journal of Supercomputer Applications*, 6(4):322–350, 1992.

[17] S. Lennart Johnsson, Yousef Saad, and Martin H. Schultz. Alternating direction methods on multiprocessors. *SIAM J. Sci. Statist. Comput.*, 8(5):686–700, 1987.

[18] 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.

[19] Woody Lichtenstein and S. Lennart Johnsson. Block cyclic dense linear algebra. *SIAM Journal of Scientific Computing*, 14(6):1257–1286, 1993.

[20] Kapil K. Mathur and S. Lennart Johnsson. All–to–all communication. Technical Report 243, Thinking Machines Corp., December 1992.

[21] Kapil K. Mathur and S. Lennart Johnsson. Multiplication of matrices of arbitrary shape on a Data Parallel Computer. *Parallel Computing*, 20(7):919–951, July 1994.

[22] Kapil K. Mathur, Alan Needleman, and V. Tvergaard. Ductile failure analyses on massively parallel computers. *Computer Methods in Applied Mechanics and Engineering*, in press, 1993.

[23] N Metropolis, J Howlett, and Gian-Carlo Rota, editors. *A History of Computing in the Twentieth Century.* Academic Press, 1980.

[24] Gary L. Miller, Shang-Hua Teng, William Thurston, and Stephen A. Vavasis. Automatic mesh partitioning. In *Sparse Matrix Computations: Graph Theory Issues and Algorithms.* The Institute of Mathematics and its Applications, 1992.

[25] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, 1990.

[26] Horst D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[27] Tayfun Tezduyar. Private communication, 1993.

[28] Thinking Machines Corp. *CMSSL for CM Fortran, Version 3.1*, 1993.