High Performance, Scalable Scientific Software Libraries

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
| Citable link   | http://nrs.harvard.edu/urn-3:HUL.InstRepos:25811002                                                                                                                                               |
| Terms of Use   | This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA |
High Performance, Scalable Scientific Software Libraries

S. Lennart Johnsson
Kapil K. Mathur

TR-19-93
October 1993

Parallel Computing Research Group
Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

To appear in Portability and High Performance Parallel Processing, John Wiley & Sons.
1

High Performance, Scalable Scientific Software Libraries

S. LENNART JOHNSSON and KAPIL K. MATHUR

Abstract

Massively parallel processors introduces 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 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.

1.1 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. This requirement is even more stringent in order for massively parallel processors (MPP) to deliver a significant cost/performance advantage over conventional supercomputers. Most MPPs are constructed out of mass...
produced, state-of-the-art components available at a fraction of the cost of the custom made, low integration-level parts used in conventional supercomputers.

Massively parallel processors are available in sizes from a few processors to a few thousand processors. Existing systems are often operated with partitions that vary in size during the day. Programs may even be developed on single nodes. Software must be designed to be operate on systems that may vary in size by as much as four orders of magnitude. This level of scalability must be accomplished transparently to the user, i.e., the same program must execute not only correctly but also efficiently over this range in processing capacity and corresponding problem sizes without change. 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.

Robustness of software both with respect to performance and numerical properties are becoming increasingly important. MPPs have the memory distributed among the processing nodes, which are interconnected by a network. Each processing node consists of communication circuitry, one or a few processors, and a local memory hierarchy. Today's high performance microprocessors used in MPPs have a processing capacity that exceeds the ability of MOS memories to deliver and accept data. By 1995, the speed of a high performance microprocessor may exceed that of DRAM chips by a factor of 10 or more. The memory system in each node will become more complex. Further, 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.

We have created a scientific software library for the Connection Machine systems CM-2, CM-200, and CM-5, the CMSSL (Connection Machine Scientific Software Library). The CMSSL today has about 250 user callable functions covering a wide range of frequent operations in scientific and engineering computation. In this paper we discuss how the goals of high performance, scalability, robustness, and performance have been achieved in CMSSL. The emphasis is on performance and scalability.

The outline of the paper is as follows. In the next section we state the major design goals for the CMSSL. The remaining sections discusses in some detail how these goals have been approached and achieved. Section 1.3 briefly introduces the software architecture followed by a discussion of language issues in Section 1.4. The data distribution in distributed memory architectures has a profound impact on performance. The issues in making the data distribution transparent to the calling program are discussed in Section 1.5, while the performance issues related to data distribution are discussed in Section 1.6. In order to guarantee high performance without reliance on sophisticated, interprocedural data dependence analysis, a high-level specification of the computation to be performed by the routine is necessary. The multiple-instance capability of the CMSSL is critical to the performance in computations on both distributed and local data sets. The multiple-instance feature is discussed in Section 1.7. Scalability and robustness with respect to performance both depend heavily on the ability to automatically select appropriate schedules for operations and data motion, and proper algorithms. These issues are discussed in Section 1.8. Communication functions are discussed in Section 1.9. A summary is given in Section 1.10.

### 1.2 Design goals of CMSSL

The ultimate 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 computation. 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 were

1. Consistent with languages with an array syntax, such as Fortran 90, Connection Machine Fortran and C*.
2. Data distribution independent functionality.
3. Multiple instance capability, i.e., operation on whole arrays in a way analogous to the way the language intrinsics operate on array data.
4. Support for all four conventional floating-point data types.
5. High Performance.
7. Robustness.
8. Portability.
9. Functionality supporting traditional numerical methods used in scientific and engineering computation.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mflop/s per node</th>
<th>Efficiency %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\ell_2$-norm</td>
<td>126</td>
<td>98</td>
</tr>
<tr>
<td>Matrix-vector</td>
<td>115</td>
<td>90</td>
</tr>
<tr>
<td>Matrix-matrix</td>
<td>115</td>
<td>90</td>
</tr>
<tr>
<td>Global</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\ell_2$-norm</td>
<td>126</td>
<td>98</td>
</tr>
<tr>
<td>Matrix-vector</td>
<td>80</td>
<td>63</td>
</tr>
<tr>
<td>Matrix-matrix</td>
<td>83</td>
<td>65</td>
</tr>
<tr>
<td>LU-factorization</td>
<td>61</td>
<td>48</td>
</tr>
<tr>
<td>Unstructured grid</td>
<td>37</td>
<td>29</td>
</tr>
</tbody>
</table>

Table 1.1  Peak local and global performance per node achieved for a few different types of computations on the CM-5. 64-bit precision.

Connection Machine Fortran, CMF, [Thi93a] and C* [Thi92] are languages with an array syntax designed and implemented by Thinking Machines Corp, for the programming of the Connection Machine systems [Thi91a, Thi91b]. CMF was modeled after Fortran 90 while this language still was in a proposal stage. Many of the new features in High Performance Fortran [For92] are derived from CMF. CMF will evolve towards HPF in the near future. C* is used as prototype language in an effort to develop a standard for C on parallel processors.

All of the goals enumerated above had an impact on the architecture of the CMSSL. The first four goals have also impacted the user interfaces. Today, the 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 1.1 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 a complete application [Tez93], while the other entries represent library functions by themselves.

Table 1.2 and Figure 1.1 provides excellent examples 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 [JHM92], while TeraFrac\(^2\) and MicMac\(^3\) are solid mechanics finite element codes ([MNT93a, BDM93]). 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.

---

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
High Performance, Scalable Scientific Software Libraries

Table 1.2 Performance in Mflop/s per node over a range of CM-5 system sizes. 64-bit precision.

For the unstructured grid computations the performance decreases by about 5%, an insignificant amount.

The focus of this paper is on the issues which are the basis for the high performance and scalability of CMSSL, and portability of application codes.

1.3 Software Architecture

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) [LHKK79, DCHH86, DCDH88]. Efficient implementations of this set of routines 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 [JM92, MJ94, Joh93b], DBLAS, is required in addition to a local BLAS, LBLAS, in each node [JO92]. 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 to mention a few.

1.3.1 External architecture

The goal of the CMSSL is to effectively support the solution of partial differential and integral equations, simulation of particle systems, and optimization problems. 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. Eigenanalysis is supported for both dense and sparse systems. For dense Hermitian systems, Householder reduction to a real symmetric matrix followed by an eigenanalysis of the real symmetric tridiagonal system is provided. Lanczos
<table>
<thead>
<tr>
<th>Elements</th>
<th>DOF</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36,364</td>
<td>4,558,800</td>
</tr>
<tr>
<td>2</td>
<td>73,728</td>
<td>946,275</td>
</tr>
<tr>
<td>3</td>
<td>147,456</td>
<td>1,846,467</td>
</tr>
<tr>
<td>4</td>
<td>294,912</td>
<td>3,671,619</td>
</tr>
<tr>
<td>5</td>
<td>559,824</td>
<td>7,272,963</td>
</tr>
<tr>
<td>6</td>
<td>1,179,648</td>
<td>14,461,827</td>
</tr>
</tbody>
</table>

**Figure 1.1** Scalability of CM–5 performance.
High Performance, Scalable Scientific Software Libraries

and Arnoldi techniques are available for sparse systems. Complex-to-complex, real-to-complex, and complex-to-real Fast Fourier transforms are supported for single and 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. In summary, the CMSSL contains

- Distributed and local BLAS (DBLAS and LBLAS)
  1. Level-1
  2. Level-2
  3. Level-3

- Sparse DBLAS
  1. for regular grids
  2. for irregular grids

- Banded direct equation solvers
- Dense direct equation solvers
- Iterative solvers
- Eigenanalysis
- Fast Fourier Transforms
- Ordinary differential equation solvers
- Statistical routines
- Communication functions
  1. for regular grids
  2. irregular grids
  3. global operations

- Stencil/convolution compiler
- Compiled routing

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. In the next section we briefly review the internal architecture. Specific issues are described in the subsequent sections.

1.3.2 Internal architecture

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 the following operational characteristics:

- Extraction of data distribution information.
- Algorithm selection.
- Execution through calls to
  - Local routines
  - Communication routines

It follows from the internal architecture of the CMSSL, that it also has the ability to serve as a nodal library. In fact, through the multiple-instance capability with each instance constrained to a node, only the nodal portion of the library is invoked. In a separately compiled nodal version of the CMSSL, the global data structure information is not required.

1.4 Languages with array syntax

Library routines operate on data structures defined in a calling program, whether used for input to, or output from, a routine. The language in which the calling program is written defines part of the environment to which a software library must conform. The hardware architecture clearly defines another part of the environment to which the library must conform and adapt. The most essential hardware characteristics of significance for high performance software is the memory architecture and the existence of pipelines. However, most high level languages abstract away the memory hierarchy. Memory is represented as a linearized address space with uniform access time. This feature is a major drawback in programming for performance, since compilers and runtime systems often are not capable in resolving this difference efficiently. In this section we discuss the collective computation aspects of languages with an array syntax. The nonuniform memory access time and how that has been resolved in the CMSSL is discussed in the next section.

The CMSSL is designed for languages with an array syntax, such as Fortran 90. In such languages, many array operations, such as for instance, addition of arrays and elementwise multiplication of two arrays, are primitive operations. Addition of an array $A$ to and array $B$ is simply expressed as $A + B$, irrespective of the rank and shape of the two arrays. No explicit enumeration of array elements is required. The array shapes are known from the declaration of the arrays. Similarly, the array type is also known from the array declaration.

In compliance with this property of array languages, arrays are passed to subroutines in CMSSL through a descriptor that contains the information about shape and data type. Thus, neither is the shape information passed explicitly in the form of arguments, nor is the type passed explicitly through routine names or arguments. In this respect, CMSSL differs from conventional libraries, such as, for instance, the BLAS, Linpack, Eispack, and LAPack. An example of a routine interface is given below.
real array: \( y(N,M,K), x(N,K,L), A(M,L,N,K) \)

gen\_matrix\_vec\_mult(y, A, x, 2, 1, 2, 3, ier)

As can be seen from this example, the array type does not appear in the routine name. And there is no array shape information in the call. The arguments 2, 1, 2, and 3 are due to the multiple-instance capability of the CMSSL. The multiple-instance capability of the CMSSL is consistent with the idea of collective computation inherent in languages with an array syntax. Library 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 or multiplication of arrays are carried out through a single statement. To accomplish the same task in a Fortran 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 the loop nest, but also allows for parallelization and optimization without a sophisticated interprocedural data dependence analysis.

In the example above, \( y \) and \( x \) represent either single vectors, or (multidimensional) arrays of vectors, and \( A \) represents a matrix, or a (multidimensional) array of matrices. The rank of the array \( A \) must be one higher than the ranks of the arrays \( y \) and \( x \), which are of the same rank. The number 2 succeeding \( x \) states that the problem axis for \( y \) is the second axes of the array \( y \), i.e., the axis of extent \( M \). Similarly, the number 1 states that the problem row axis for \( A \) is axis 1 of the array \( A \), and the problem column axis is axis 2. The shape of each instance of \( A \) is \( M \times L \). The problem axis for \( x \) is axis 3 of the array \( x \). Thus, the above call defines multiple matrix–vector multiplications. Each instance consists of the multiplication of a \( M \times L \) matrix by a vector of length \( L \). There are \( N \times K \) such instances. The call is independent of the distribution of the arrays.

Restricting the arrays in an operation to have conforming shapes for the instances, allows for an implicit ordering of instances corresponding to the ordering of the axes in the arrays.

The use of higher dimensional arrays for a collection of vectors and matrices is often the preferred data representation in many applications. For instance, in Quantum Chromodynamics (QCD), computations are performed on a four dimensional regular lattice, where in each lattice point the state includes small matrices and vectors. It is natural to represent the matrices as six-dimensional arrays, and the vectors as five-dimensional arrays. Similarly, in finite difference methods for the solution of Navier–Stokes equations, a three-dimensional grid may be used for the spatial discretization, with the state in each grid point represented by vectors and matrices. It is natural to represent the matrices as five-dimensional arrays, and the vectors as four-dimensional arrays.

Finally, we remark that the CMSSL is a generic library for languages with an array syntax. The same library indeed supports applications written in either CMF or C*.

1.5 Data distribution independent functionality

In traditional computers and in shared memory parallel architectures, the data distribution is transparent to the programmer. The programming languages are based on...
the notion that the memory access time is constant. However, this is not true in conventional supercomputers, and certainly not in current distributed memory parallel computers. Bank conflicts on traditional supercomputers are a well known artifact of nonuniform memory access time. The only means a programmer has to affect the data distribution among memory units is through the array shape. The memory address space is linearized, which is one of the fundamental difficulties compilers and run-time systems must overcome.

The Connection Machine systems are distributed memory architectures, but may be programmed with a global address space. However, unlike traditional architectures, this address space is not linearized. It is treated as a multidimensional address space, where the dimensionality of the address space is determined for each array based on its shape, as discussed in the next section. The user can affect how the index space is mapped onto the physical memory through compiler directives. Compiler directives for control of the data distribution is the main contribution of High Performance Fortran over Fortran 90. The data distribution may have a significant impact on the locality of reference as well as on the time to move data between processing nodes for a given amount of locality. The potential benefits of exploiting locality of reference for three typical computations are quantified in Table 1.3. Table 1.3 [Joh90] shows the number of operations (and local references) per remote reference per processor for a few local memory sizes, assuming optimal locality of reference. For matrix multiplication and a block of size 100 x 100 (local storage 32k words), the reduction in the need for memory or communications bandwidth is a factor of 100 compared to no locality of reference. For the FFT, the reduction in required interprocessor communication bandwidth is a factor of 14 for a radix-16k algorithm. The significance of data locality of reference is clear. Table 1.3 also illustrates the fact that data motion is a much more significant issue for computationally efficient algorithms, such as the FFT. Computationally efficient algorithms require relatively few operations per data point. Data for the finite difference operator is based on a model that the ratio of operations to remote references follow the rule \( \frac{1}{\gamma} \left( \frac{\alpha \beta}{\gamma} \right)^\gamma \) for suitable values of \( \alpha, \beta, \) and \( \gamma. \) \( M \) is the local memory size. Exploiting locality reduces the required communication bandwidth by a factor of 8-100 at the chip boundary for these computations, a factor of 80-5000 at the board level, and at least a factor of 125 at the I/O interface. The value of exploiting locality is apparent, but the techniques for accomplishing this task are not.

Data distribution directives are orthogonal to the semantics of the program. The meaning of the program is unaffected by changes in the compiler directives, and no change of the code should be necessary for any change in any of the directives. Thus,
calls to a library must also be unaffected with respect to correctness, regardless of what directives are given. Moreover, it is highly desirable that a program can be executed on different size systems without recompilation. This was indeed one of the requirements for CMSSL. The requirement that programs should run on systems of different sizes implies that data distributions are not known until run-time. Much of the performance optimization cannot be made until run-time, even when the size and shape of arrays are known at compile-time.

The data distribution independent functionality of the CMSSL is accomplished without any information being passed explicitly in a call to a library routine. The layout information, together with information about data type and array shape, is kept in an array descriptor. Whenever the actual data layout is required either for correctness or performance, the library routines will make use of this information. However, the library interfaces are independent of layout directives, as shown in the following example, where both the calls to gen_matrix_vector_mult generate the same answers but the performance will differ.

\begin{verbatim}
DIMENSION A(81, 81, 4096), x(81, 4096), y(81, 4096)

CALL GEN_MATRIX VECTOR_MULT(y, x, 1, 1, 2, 1, ierr)

CMFS_LAYOUT A(:SERIAL, :SERIAL, x(:SERIAL), y(:SERIAL))
DIMENSION A(81, 81, 4096), x(81, 4096), y(81, 4096)

CALL GEN_MATRIX VECTOR_MULT(y, x, 1, 1, 2, 1, ierr)
\end{verbatim}

In the above example, SERIAL is a layout directive in Connection Machine Fortran that denotes that an array axis is allocated within the memory associated with a functional unit. Axis weights are used to control the length of the segment of an axis assigned to a node. Weights do not affect the total number of elements assigned to a node. It is also possible to directly specify the shape of the subarray assigned to a node [Thi93a].

1.6 Data allocation

When there are more data elements than processing nodes, several data elements are allocated to the same node, for an even data distribution. An even data distribution is a simple rule aimed at achieving load balance without sophisticated data dependence analysis. With uniform reference patterns the objective of good load balance is achieved. But, load balance may also be achieved for many nonuniform reference patterns depending upon how the elements mapped to a node are selected. Several schemes are used on the Connection Machine systems for data aggregation. Some of these schemes are part of the system software, other schemes are part of the CMSSL.

In this section we will first discuss data allocation schemes for regular arrays and meshes, then aggregation schemes for sparse matrices and irregular meshes. Following this discussion we address the issue of the placement of aggregates.
1.6.1 Data aggregation schemes

Consecutive (block) and cyclic distribution [Joh87], or combinations thereof [Joh85], are common schemes for data aggregation of regular arrays. In the CMSSL, data distributed on regular meshes, and common operations on such data, are treated in the same way as dense arrays are treated. The techniques we use for data aggregation for arbitrary sparse matrices and irregular grids are based on partitioning of arbitrary graphs.

1.6.1.1 One-dimensional arrays

In the consecutive distribution scheme, a one-dimensional array of $M$ elements is allocated to $N$ memory units by allocating $\left\lfloor \frac{M}{N} \right\rfloor \left(\left\lceil \frac{M}{N} \right\rceil\right)$ consecutive elements to the same memory unit. In cyclic distribution, element $i = M \mod N$ is allocated to memory unit $i$. The consecutive distribution scheme is currently used on the Connection Machine systems. Cyclic as well as consecutive distribution, and combinations thereof, are included in Fortran D [FHK+90], Vienna Fortran [CMZ92, ZBC+92], and the proposed High Performance Fortran. Cyclic distribution is traditionally used in banked memory systems [HP90]. With the stride being a multiple of the number of memory units, the memory system operates at the speed of a single bank instead of the full speed of the memory, which typically matches that of the processing speed. The performance degradation for memory access strides being a multiple of the number of memory banks is known as bank conflicts. We will discuss consecutive and cyclic allocation further in Section 1.6.1.5.

1.6.1.2 Multi-dimensional arrays

The distribution of two-dimensional arrays to memory units for contention-free access was studied carefully in the context of the Burroughs Scientific Processor (BSP) [BK71, KS82, Law75, LV82]. In all distribution schemes considered for the BSP, a two-dimensional index space was converted to a one-dimensional (linear) index space through row or column skewing in combination with the standard row and column major orderings. The resulting one-dimensional array was, in all cases, mapped to the memory units by the cyclic distribution scheme. It was shown that rows, columns, diagonals, and blocks can only be accessed concurrently when the number of memory units is a prime number. The resulting design, which includes algorithms for address generation and routing, is known as the Prime Memory System [LV82].

One important assumption in the Prime Memory System was that the interconnection network can support the communication of one word per memory unit every clock cycle to each processor, i.e., the network has the capacity of a full crossbar between processors and memory units. In current distributed memory architectures, this assumption is not valid. The network capacity may be up to two orders of magnitude less than the aggregate peak capacity of all memory and processing units. Therefore, memory and processing units are grouped together in distributed memory architectures, and exploiting locality of reference in computations is important to alleviate the network capacity from becoming a bottleneck. The use of a multidimensional address space is necessary, in general, to achieve the desired level of locality of reference [Joh90].
A Cartesian address space of the same dimensionality as the problem space, yields the desired locality of reference for relaxation methods and convolution, when combined with grouping of data by the consecutive scheme. For example, a three-dimensional address space and consecutive allocation, reduces the communication requirements for explicit methods for the solution of partial differential equations by up to two orders of magnitude (Table 1.3).

1.6.1.3 Dimensionality of the address space

The ideal dimensionality of the address space is not only a function of the dimensionality of the data array, but also of the data reference pattern. In the case of relaxation methods with simple stencils or convolution kernels, an address space with the same dimensionality as the data array is sufficient to minimize the number of remote references. However, other schemes, such as most hierarchical or divide-and-conquer based methods, may benefit from a higher dimensional address space even on computations on single arrays. Thus, in computing the FFT on \( N \) data elements, an address space in the form of a binary cube with \( \log_2 N \) dimensions is ideal. All data references are to data within unit distance in such an address space.

Most computations involve operations on more than one array. A simple example is matrix multiplication: \( C \leftarrow A \times B \). With \( A \) being of shape \( P \times Q \) and \( B \) of shape \( Q \times R \), the index space for the computation is three-dimensional (Figure 1.2) even though the index space for each of the operands is two-dimensional. Depending on the relative sizes and shapes of the three operands, a one-dimensional, two-dimensional, or three-dimensional address space is optimal for the computations [Joh93a]. The fact that a three-dimensional address space is desirable when there is many more processing nodes than matrix elements is well known. However, a three-dimensional address space may also be beneficial with respect to performance when there are many more matrix elements than processing nodes [JH89a, Joh93a]. In a distributed memory architecture, the affect of the dimensionality of the address space on performance can be very significant (more than one order of magnitude).

1.6.1.4 Shape of the multidimensional address space

The shape of a multidimensional address space, i.e., the shape of the local subarrays, has a significant impact on the performance for many computations. For a \( d \) dimensional data array with equally frequent data references along all axes, and the same reference pattern for each array element, it is well known that the number of remote references is minimized when the lengths of the segments of all axes mapped to a memory unit are the same [FJL+88, Joh90]. Thus, for two-dimensional arrays, the local subarray should be a square for minimum communication needs, and for three-dimensional arrays the local data set should be a cube, and so on.

Computations such as LU and QR factorization and the solution of triangular systems of equations, require global communication operations for selecting pivots, distributing pivot rows and columns, and related operations. No nearest neighbor operations in a Cartesian space are required. The ideal shape of the address space for the factorization is one in which shape of the subarrays is close to a square [LJ93]. The deviation from a square is due to the slight difference in communication needs along
the two array axes. The ideal array shape is also somewhat effected by whether the communication system can be modeled as a node limited or link bound system. In a node limited communication system, the data rate with which a node can send and receive data is independent of the number of channels per node. In the link bound model, the data rate per node is determined by the number of communication channels (links) per node participating in the communication.

For the Alternating Direction Implicit (ADI) methods, which are based on the solution of sets of tridiagonal systems of equations in alternating directions along array axes, the ideal shape of the address space is such that the shape of the local subarrays have axes' of equal length [JSS87, JH90].

For matrix multiplication, we first notice that in order to minimize the communication, a good strategy is to keep the matrix with the largest number of elements stationary [Joh93a, MJ94]. 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 [Joh93a, MJ94]. The ideal shape of the address space has been verified on the Connection Machine system CM-200 and is illustrated in Figures 1.3 and 1.4. Figure 1.3 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.

Figure 1.4 is more interesting in that all three operands have different shapes. The
High Performance, Scalable Scientific Software Libraries

Figure 1.3 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 $N_r \times N_c = 256$.

aspect ratio for the product matrix is four, while the aspect ratios for the multiplier and the multiplicand were varied by a ratio of up to 16. With a 256 node system, and an algorithm that keep the product matrix stationary, the results show that making the submatrices of the product matrix square tends to yield the best performance. Deviations from the predicted optimal shape is due to some artifacts of the Connection Machine geometry manager. The geometry manager determines the data distribution at run-time, either from a set of default rules as indicated in previous sections, or based on compiler directives. The fact that the geometry manager does not always find the ideal shape is the reason that some of the cases in Figure 1.4 have optimal subarray shapes for the stationary matrix that are not square. Forcing them to be square for the multiplication would require a reallocation of the product matrix, which would be more costly than the gain in communication time for the multiplication phase. For the rectangular matrices in our example, the best to worst performance ratio as a function of the nodal array shape was nearly 20.

Determining an ideal nodal array shape and optimum algorithm as a function of the operand shape should not be required from the user. The ideal shape is dependent not only upon array shapes and sizes, but also on the number of processing nodes, which may not be known until run-time. On the Connection Machine systems the default data distribution strategy attempts to make the lengths of all segments mapped to a memory unit as equal as possible. This mapping is ideal for

- the explicit solution of partial differential equations, when the difference “stencil”, or “molecule”, is symmetric.
- the ADI method
- nearly optimal for LU and QR factorization and the solution of triangular systems of equations
- optimal with respect to the stationary matrix in matrix multiplication (including the degenerate case of matrix-vector multiplication).
Figure 1.4 Influence of shared processor configuration on multiplication of rectangular matrices of shape $P \times Q \times R$, 64-bit precision. The shape of the 256 processor Connection Machine system CM-200 is $N_r \times N_c = 256$.

For operations on multiple arrays, or when the default layout is sufficiently far from optimal to warrant a data reallocation, a reallocation to an optimal layout for the requested computation must be made within the CMSSL routine being called. This decision is based on knowledge of the layout of the input and output arrays, a collection of candidate algorithms, and the performance characteristics of those.

1.6.1.5 Cyclic vs consecutive data allocation

In the previous section we showed that the shape of the address space may have a significant impact on performance. In this section we discuss some of the issues in choosing a cyclic or consecutive allocation, and show that in fact the most often quoted justification for cyclic data allocation does not need such an allocation for load-balance, or any other performance related reason.

A cyclic data distribution is often advocated as a technique to achieve load balance for computations in which the index space is not used uniformly, such as in LU and QR factorizations and triangular systems solution [LC88, LC89, vdG91]. However, in these particular examples, load balance can be achieved through a suitable scheduling of operations with a consecutive data allocation. In fact, the same load balance is achieved for

- consecutive data allocation and cyclic elimination order, and
- cyclic data allocation and consecutive elimination order

In the consecutive elimination order, the elimination proceeds through the address...
space in consecutive order (with data exchanges as required by pivoting). The cyclic elimination order is illustrated in Figures 1.5 and 1.6, which show one cycle and the start of the second cycle in the elimination process for a $4 \times 4$ nodal array and a block size of two. Pivot rows and columns are indicated by numbers, eliminated elements by periods, and all other elements are shown as an asterisk.

As can be seen from Figures 1.5 and 1.6, the result of the factorization is not two block triangular matrices, but two block-cyclic triangles. The substitution process is load-balanced for the block-cyclic triangles, in the same way as the block-cyclic elimination is load-balanced.

Thus, by suitably adjusting the schedule of operations to the data allocation, load-balance can be achieved for the computations most often used to justify cyclic data allocation. Matrix multiplication, which uses the index space uniformly and is naturally load-balanced for sufficiently large operands, can be performed equally well under cyclic and consecutive data allocation. But, the choice of cyclic or consecutive data allocation may have a significant impact on the communication requirements, even for computations that are load-balanced. For relaxation methods, a cyclic data allocation increases the communication requirements significantly by increasing the surface area for a given volume of data assigned to a node, and should be avoided. In contrast, cyclic data allocation is advantageous with respect to communication for the FFT, which also is naturally load-balanced with respect to arithmetic. For the FFT, the butterfly computations proceed from the most significant bit in the index space to the least significant bit. Thus, with the most significant bits allocated to local memory, no communication is required for the leading bits [JJK92, HJIR87, Swa87, TS91].

1.6.1.6 Sparse matrices – regular grids

The purpose of sparse matrix techniques is to take advantage of the zero/nonzero structure of a matrix to reduce both storage and arithmetic needs. Address calculation is often a significant portion of the time required in sparse matrix operations. Some storage schemes attempt to reduce this time at the expense of storing some zero elements and performing (redundant) arithmetic operations on these elements. For highly regular sparse matrices, storage schemes and address calculations can be much simplified compared to arbitrary sparse matrices. Scheduling of operations for efficient use of memory hierarchies is also much simplified for highly regular sparse matrices. Therefore, the data representation and the algorithms for sparse matrices with a "stripe structure", i.e., a structure with nonzeros appearing on diagonals that are not necessarily adjacent, are typically different from those for arbitrary sparse matrices. On massively parallel architectures, it is advantageous to maintain this difference, but the representation we use is different from the ones used on sequential machines. Our representation simplifies preservation of locality of reference in many important applications.

We refer to sparse matrices originating from difference approximations on regular domain discretizations as Grid Sparse matrices. The structure of these matrices is either a band or a collection of non-zero diagonals separated by a number of zero diagonals. The matrix entries may be single elements, vectors, or small matrices. Explicit methods for the solution of partial differential equations and iterative methods
Figure 1.5  The first three steps of block cyclic elimination.
Figure 1.6 Steps four through six of block cyclic elimination.
for the solution of linear systems of equations, require matrix–vector multiplication or matrix–matrix multiplication where one or all operands are sparse. It is common to represent the nonzero matrix elements as a collection of one-dimensional arrays, or sometimes as a dense matrix with the number of columns equal to the number of nonzero diagonals [Saa93]. Each array, or each column in the two-dimensional representation, represents a linear ordering of all grid points. If the sparse matrix entries are vectors or matrices, then the dimensionality of the arrays is increased by one or two dimensions.

Representing the nonzero diagonals in this traditional manner is not suitable for distributed memory architectures, the linear ordering of grid point values makes the task to preserve locality of reference more complex than necessary. Relaxation methods for linear system solution and explicit methods for partial differential equations, typically access data in some local neighborhood of the grid point to be updated. The task of preserving locality of reference is much simplified if the adjacency in the grid is preserved in the data representation.

In the CMSSL, grid point data and operations thereupon are represented as dense arrays, thereby allowing the techniques for preserving proximity in mapping dense arrays to processing nodes to be used also for grid sparse problems. Grid point values are represented as multi-dimensional arrays with one axis for each axis of the grid. The axes extents are the same as the corresponding grid axes. There may be additional axes if grid point data consists of vectors or matrices.

A grid sparse matrix represents interactions between variables in different grid points. For instance, the common three-point stencil in one dimension yields a tridiagonal matrix representing interactions between adjacent grid points along one array axis, a five-point stencil in two dimensions yields a matrix with five nonzero diagonals with a row or column ordering of the grid points. Similarly, a seven-point stencil in three dimensions yields a matrix with seven nonzero diagonals, defining the interaction between adjacent grid points in three dimensions.

In the CMSSL representation of grid sparse matrices, there is a multi-dimensional array for each nonzero diagonal in the matrix representation. Each such array represent the interaction between a pair of grid points. The array shape is the same as that of the array of grid point values. As for the grid point data, if the stencil has nonscalar entries for the interaction, then additional data array axes are used to represent the nonscalar quantities.

Representing grid sparse matrices based on the underlying grid structure preserves locality of reference not only within an array, but also between arrays, if all arrays with the same shape are partitioned in the same way. This rule is the default on the Connection Machine systems. Allocating the stencil coefficients, represented by the nonzero diagonals in the grid sparse matrix, to the same processing node as the grid point data for which it is used to compute a new value, implies that communication is only required for the boundary of subarrays. The required data motion can be expressed through a polymesh (Pshift) operation [GBJ94], which is a generic communication primitive that can be optimized with respect to communication and local data motion. An example of the call for a seven-point stencil in three dimensions is shown below.

19/9/1994 17:53—PAGE PROOFS for John Wiley & Sons Ltd (using jwwebmc01, Vers 01.01 May 1992)—P4
High Performance, Scalable Scientific Software Libraries

\begin{verbatim}
DIMENSION A(512,512,512), B(512,512,512), C(512,512,512), D(512,512,512),
E(512,512,512), F(512,512,512), G(512,512,512), X(512,512,512),
Y(512,512,512)

CALL GRID_SPARSE_MATRIX_VECTOR_MULTIPLICATION(ier, setup, y_axes, coeff_axes, x_axes,
Y, X, A, B, C, D, E, F, G)
\end{verbatim}

1.6.1.7 Sparse matrices – irregular grids

For computations involving arbitrary sparse matrices and irregular grids, taking advantage of locality of reference is a much more difficult problem than for grid sparse matrices. The basic idea is still the same: assign variables and elements of the sparse matrix, or the grid, to processing nodes based on a partitioning of the underlying graph. However, the representations of the matrix and the nodal variables are quite different from the grid sparse case, and so are the partitioning techniques. As was the case for grid sparse problems, partitioning based on subdomains preserve locality of reference when the computations use data with indices in some local neighborhood of the data being updated. This property holds for most iterative methods for the solution of sparse systems of equations, and for explicit methods for the solution of partial differential equations.

Two general partitioning techniques of significant recent interest are the recursive spectral bisection technique proposed by Pothen et al. [PSL90] and the geometric approach proposed by Miller et al. [MTV91, MTTV92, Ten91]. The recursive spectral bisection technique has been used successfully by Simon [Sim91] for partitioning of finite volume and finite element meshes. A parallel implementation of this technique has been made by Johan [Joh92a]. This implementation is now part of the CMSSL.

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 [Fie73, Fie75a, Fie75b]. 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. The reason for using a volume or element based graph is that the computations are naturally organized as volume or elementwise computations. These computations exhibit locality of reference within the volumes or elements and can often be performed as a (large) collection of dense matrix operations. Communication is required when passing data between the global representation, and the representation of the collection of local elements [JM90, MNT93b]. The purpose of the partitioning is to minimize this communication.

For finite element computations, the adjacency in applying the spectral bisection method has been approximated by elements that share faces. This adjacency accurately represents the communication requirements for finite volume methods. However, in finite element methods, communication is also required between elements sharing edges and corners.

Thus, for finite element and finite volume computations the graph to be partitioned
have nodes representing individual elements or volumes, and edges representing shared faces. Based on this partitioning, a partitioning of the set of nodal values is carried out. Nodal points internal to a partition are mapped to the processing node to which the partition is assigned. Boundary nodes must be assigned to one of the partitions among which they are shared, or replicated among the partitions among which they are shared. Only boundary nodes require communication.

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 geometric partitioning technique by Miller et. al. holds promise to be computationally less demanding than the spectral decomposition technique, but relies on geometric information and geometric properties of the graph [MT90, MV91]. Geometric information is typically available for meshes generated for the solution of partial differential equations, but may not be present in other applications. The geometric approach is currently being implemented as part of the CMSSL. Since we do not yet have much experience with the geometric partitioning technique we only report the results from the spectral bisection technique.

The results of applying the spectral bisection technique to two model problems are reported in [Joh92a, JH92a] and shown in Tables 1.4 and 1.5. One of the model problems consists of a planar triangular mesh between an outer ellipse and an inner double ellipse. The other problem is a grid of tetrahedra between concentric cylinders. The planar grid has 8,307 nodes, 16,231 triangles, and 24,537 edges. The numbers of shared nodes and edges as a function of the number of partitions are given in Table

<table>
<thead>
<tr>
<th>Number of partitions</th>
<th>Number of shared edges</th>
<th>% of total</th>
<th>Number of shared nodes</th>
<th>% of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>188</td>
<td>0.8</td>
<td>195</td>
<td>2.4</td>
</tr>
<tr>
<td>16</td>
<td>381</td>
<td>1.6</td>
<td>396</td>
<td>4.8</td>
</tr>
<tr>
<td>32</td>
<td>752</td>
<td>3.1</td>
<td>773</td>
<td>9.3</td>
</tr>
<tr>
<td>64</td>
<td>1483</td>
<td>6.0</td>
<td>1479</td>
<td>17.8</td>
</tr>
<tr>
<td>128</td>
<td>2154</td>
<td>8.8</td>
<td>2101</td>
<td>25.3</td>
</tr>
</tbody>
</table>

Table 1.4 Partitioning of a planar mesh with inner boundary in the form of a double ellipse.

<table>
<thead>
<tr>
<th>Number of partitions</th>
<th>Number of shared edges</th>
<th>% of total</th>
<th>Number of shared nodes</th>
<th>% of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5186</td>
<td>2.4</td>
<td>2775</td>
<td>13.4</td>
</tr>
<tr>
<td>16</td>
<td>8005</td>
<td>3.7</td>
<td>4095</td>
<td>20.1</td>
</tr>
<tr>
<td>32</td>
<td>11553</td>
<td>5.3</td>
<td>5747</td>
<td>28.2</td>
</tr>
<tr>
<td>64</td>
<td>16055</td>
<td>7.3</td>
<td>7721</td>
<td>37.9</td>
</tr>
<tr>
<td>128</td>
<td>21502</td>
<td>9.8</td>
<td>9827</td>
<td>48.2</td>
</tr>
</tbody>
</table>

Table 1.5 Partitioning of a tetrahedral mesh between concentric spheres.
Table 1.6  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.

14. The grid for the concentric spheres consists of 20,374 nodes, 107,416 tetrahedra, and 218,807 faces.

The results of applying the spectral bisection technique in a more realistic finite element application [MJH94] are summarized in Table 1.6. 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. The address computation may be fairly complex. It frequently is the most time consuming part on uniprocessors. On a distributed memory machine, the address 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. CMSSL provides a mechanism for caching this information and reusing it later.

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. Thus, CMSSL supports both representation and operations on element sparse and block sparse matrices.

1.6.2 Allocation of aggregates

The consecutive and cyclic distribution schemes and the dimensionality of the address space for regular and grid sparse arrays, and spectral or geometric partitioning for arbitrary sparse matrices and irregular grids, determines which elements are grouped together for a node. The data reference pattern determines the communication needs of each node.

The total demand on the communication system is not only affected by how the
data is grouped together, but also how the groups are assigned to nodes. Ideally, the
groups of data are allocated to the nodes such that the contention is minimized. To
accomplish this task, both the data reference pattern and the network topology must be
taken into account, as well as the routing scheme. Optimal allocation of data is a hard
problem. Moreover, the allocation may need to be dynamic to efficiently accommodate
different phases of a computation. We will briefly discuss the mapping of groups of data
below. Routing issues and how they have been addressed in the Connection Machine
systems and in CMSSL will be discussed in a later section.

1.6.2.1 Allocation of regular arrays

For nearest neighbor references along data array axes, such as for typical difference
stencils on regular grids, proximity preserving data array embeddings of array parti-
tions to nodes are ideal, if such embeddings exist. Such embeddings do exist in nodal
arrays of a dimensionality that is at least as high as that of the data array [Ros75].
Thus, two-dimensional mesh configured machines can embed meshes of at most two
dimensions preserving adjacency, while three-dimensional meshes of processing nodes
allows adjacency preserving embeddings for grids of up to three dimensions. Binary
cube networks of nodes allow for adjacency preserving embeddings of meshes with
a dimensionality up to the number of cube dimensions. On the Connection Machine
system CM-200 [Thi91a], the default embedding of data arrays preserve adjacency.

Though data arrays can be embedded preserving adjacency in meshes with sufficient
dimensionality, such embeddings are not possible in some other common networks,
such as butterfly networks, and binary tree networks [RS88]. The fat–tree network of
the CM-5 [Lei85] does not allow an embedding of higher dimensional arrays preserving
adjacency.

1.6.2.2 Allocation of irregular arrays

For irregular grids, mapping of groups of data to processing nodes such that proximity
is preserved, is a much more difficult problem than the mapping of regular arrays.
Instead of attempting to find the best possible map, it may be more profitable to
search for a map that is guaranteed to have an acceptable worst case behavior. A
randomized data placement [RBJ88, Ran87] reduces the risk for bottlenecks in the
routing system. The randomized placement of data achieves the same communication
load characteristics in a single (deterministic) routing phase as randomized routing
achieves in two phases [Val82, VB81]. Randomized placement is an option in some
CMSSL routines [Thi93b].

The Connection Machine system CM-200 does not support randomized routing. The
routing on the CM-5 randomizes the path selection on the message path towards the
root, followed by a deterministic path from the point of turnaround to the destination.

Figures 1.7 and 1.8 give examples of the performance improvements achieved on the
CM-2 through the use of randomized data allocation in a finite element computation
on an unstructured grid. The horizontal axis shows the number of degrees of freedom
and elements, while the vertical axis denotes the execution time. Each element has 24
degrees of freedom. The performance improvement for the gather instruction due to
randomization is in the range 2.1 - 2.4. The improvement is increasing with the problem
Figure 1.7 Gather with binary and randomized addresses. 8K CM-2.

Table 1.7 The effect of randomization on gather and scatter performance. Times in msec on an 8K CM-200.

size. Figure 1.8 shows the execution times for two methods of accumulating the product vector: using the combining features of the router, and accumulation after the routing operation. Randomization of the addresses improved the router combining time by about a factor of two, but performing the routing without combining is even more effective. Table 1.7 gives the gather scatter times with and without randomization for a solid mechanics application [MNT93b] on the CM-200. The performance enhancement is a factor of 1.5 – 2.2, which in our experience is typical. It is rarely the case that randomization has caused a performance degradation.
1.6.3 A summary of data allocation issues

The assignment of data to memory units affects load–balance, communication requirements, network contention, and the performance of the computations in each node (by affecting the ability to carry out local blocking and pipelining of operations).

- Consecutive distribution preserves locality of reference along data array axes, and is suitable for stencil–like reference patterns. It also offers the possibility to improve the efficiency of the operations in each node by increasing the chance for good cache behavior through optimal blocking, and through long vectors for pipelined processors.
- Cyclic distribution significantly increases the communication requirements for relaxation methods and explicit methods for the solution of partial differential equations, and shall be avoided. Cyclic distribution may offer a reduction in the communication requirements for the FFT by a factor of two. Cyclic distribution is not required for load–balance in LU and QR factorization, or for the solution of triangular systems of equations.
- Grid sparse problems can be represented as dense matrix problems. The partitioning techniques for such problems yield good locality of reference for typical operations on regular grids and grid sparse matrices.
- Arbitrary sparse matrices and irregular grids can be successfully partitioned into subdomains using recursive spectral bisection and geometric partitioning.
- An optimum distribution of partitions requires a data dependence analysis and an understanding of optimum embeddings and routing algorithms for the network at hand. For irregular computations, and when proximity preserving embeddings may not be possible, minimizing the contention through randomized distribution has shown to be effective.
Finally, we remark that data distributions cannot be determined until run-time, not only because the index sets may not be known until run-time, but it is highly desirable not to constrain the number of processors used for execution at compile time. Library routines must always yield a correct result. Library routines must also offer as good a performance as possible, given the specified data distributions of the input and output objects. The decisions of what algorithm to choose for a given function, and whether or not a redistribution shall be performed before and after executing the function, must be made at run-time.

1.7 Multiple-instance computation

The multiple-instance capability removes loop nests in the calling program and provides for additional optimization opportunities in carrying out the specified operation without sophisticated data dependence and interprocedural analysis. Even with information about all instance computations readily available, choosing an optimal (parallel) algorithm is a very complex task. For sequential machines, the level-3 BLAS was introduced to provide a sufficient degree of freedom compared to the level-1 BLAS, to allow for a desired level of optimization for cache based architectures. Much in the same spirit, the multiple-instance capability of the CMSSL provides interfaces at a sufficiently high level to perform the necessary performance optimizations without extensive reliance on compilers and run-time systems.

We illustrate the multiple-instance capability in computing the FFT along one of the axes of a two-dimensional array of shape $P \times Q$. With a canonical data layout, the set of nodes are configured as a two-dimensional array of shape $N_r \times N_c$, where $N_r N_c = N$. Figure 1.9 illustrates the layout of a two-dimensional data array in row and column major order on a $2 \times 4$ nodal array.

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. Maximizing the concurrency for each FFT through the use of a canonical data layout for one-dimensional arrays of size \( P \).
2. Computing each FFT without data relocation.
3. Computing all \( Q \) FFTs concurrently through multiple-instance routines.

Alternative 1 can be expressed by the following code fragments:

```fortran
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
```

For Alternative 1, a temporary one-dimensional array with a canonical layout is created for each column \( A(:, J) \). 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 (scatter) within processing node rows for the row major ordering. In one-to-all personalized communication, a node sends a unique piece of data to all other nodes. Upon completion, an all-to-one personalized communication (gather) is required within processing node rows. The data redistribution is illustrated in Figure 1.10. The redistribution corresponds to a change in data allocation from \( A(:, :) \) to \( A(:, SERIAL) \) 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 the column major ordering, a skewing is required prior to the one-to-all personalized communication within columns, as well as after the all-to-one personalized communication within columns that follows the FFT computation. The skewing step is shown in Figure 1.11.

Alternative 2 can be described by the following code fragments:

```fortran
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
```

In Alternative 2, the data redistribution is avoided by computing each instance.
Figure 1.10  Data redistribution for load-balanced column processing. Nodes labeled in row major order.

Figure 1.11  The skewing step in data redistribution for load-balanced processing of columns in a column major labeling of a two-dimensional nodal array.
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.

Alternative 3 can be illustrated by the code fragment:

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

Alternative 3 is used in the CMSSL. All different instances of the FFT represented by the different columns are treated concurrently in-place. However, a `FORALL` statement is not used, since its use in Connection Machine Fortran [Thi93a] is limited to a single assignment. The concurrency and data layout issues are managed inside the FFT routine, below the Fortran level of programming. The call to the multiple-instance library routine is consistent with a language with array syntax and of the form

```fortran
CALL FFT(A, DIM = 1)
```

The argument `DIM` specifies the axis of the array $A$ to which the function shall be applied. The actual CMSSL call to the FFT routine 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 [Thi93b].

The three outlined alternatives have the following performance characteristics.

**Alternative 1.** $Q$ one-to-all and all-to-one personalized communications within rows for a row major ordering. These communications correspond to the data redistribution $A(;;)$ to $A(;;, SERIAL)$ and back to $A(;;)$, one column at a time. For column major ordering, a skew operation is required in addition to the personalized communication. With $N$ nodes, the arithmetic speedup is proportional to $\min(N, P)$ for a transform along the $P$-axis.

**Alternative 2.** In-place, single-instance computation. No excess data motion. With $N_r$ nodes along the $P$-axis, the arithmetic speedup is proportional to $\min(P, N_r)$.

**Alternative 3.** Multiple-instance, in-place computation. No excess data motion. The arithmetic speedup is proportional to $\min(N, PQ)$.

The third choice is clearly preferable both with respect to communication and computation. The communication time in Alternative 1 is determined by the time for one-to-all and all-to-one personalized communication. For a $P \times Q$ data array allocated to a $N_r \times N_c$ nodal array, the communication time in a node limited communication system is proportional to $2Q \frac{P}{N_c}$. In a link bound communications system, such as the Connection Machine system CM-200, the optimal communication time for the one-to-all and all-to-one personalized communication operations, require a time of $2Q \frac{P}{N_c \log_2 N}$. The time required for the FFT computation itself is proportional to $\frac{PQ}{N_r N_c}$ in the node limited model, and to $\frac{PQ}{N_r N_c}$ in the channel limited model [JHJR87, JJK92, JK92, Swa87, TS91]. Thus, the data redistribution time in Alternative 1 far exceeds the communication time required for the FFT computation.
Table 1.8  Optimal communication times for FFT computation on the Connection Machine system CM–200.

<table>
<thead>
<tr>
<th>Function</th>
<th>Alternative 1</th>
<th>Alternative 2</th>
<th>Alternative 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data realloc.</td>
<td>$2Q N_{rev} N_c^{p}$</td>
<td>$Q N_c^{P}$</td>
<td>$Q N_{c}$</td>
</tr>
<tr>
<td>FFT</td>
<td>$Q N_c^{P}$</td>
<td>$Q N_c^{P}$</td>
<td>$Q N_{c}$</td>
</tr>
<tr>
<td>Total</td>
<td>$Q N_c^{P}$</td>
<td>$Q N_c^{P}$</td>
<td>$Q N_{c}$</td>
</tr>
<tr>
<td>Bit–reverse</td>
<td>$Q N_{rev} N_c^{P}$</td>
<td>$Q N_{rev}$</td>
<td>$Q N_{rev}$</td>
</tr>
</tbody>
</table>

Table 1.9  Arithmetic speedup for FFT computation on the Connection Machine system CM–200.

The communication times are summarized in Table 1.8, and the arithmetic speedups in Table 1.9.

Note that with a single-instance library routine and canonical layouts, Alternative 1 would be realized.

1.8 Automatic algorithm selection

One of the novel features of CMSSL is the automatic selection of algorithms. This is made both at the local level and at the global level. The purpose of the selection is to maximize performance by preserving locality of reference. We discuss this feature for matrix operations, both at the local and global level.

1.8.1 Local algorithm selection

We use matrix–vector multiplication to illustrate one of the novel features of CMSSL, namely run-time loop partitioning and loop reordering. The need for these features arises from the local memory hierarchy in each node of the Connection Machine systems. There is a single data path between each memory unit and the floating-point unit with which it is associated. This is true for the CM-2, the CM-200, and the CM-5. It is also true for most other multiprocessors based on standard microprocessors. The register set forms the first level in the memory hierarchy. The next level for the Connection Machine systems is DRAM pages. DRAM is operated in page mode, which allows one memory access per processor cycle for accesses within a page. However, access to a different page results in a page fault, which results in a two cycle access time for the CM-2 and CM-200, while on the CM-5 a DRAM page fault incurs an access time of five cycles, a significant performance loss. A third level of the local memory hierarchy exists on the CM-5, and is introduced through translation lookahead buffers,
Figure 1.12 The efficiency of matrix-vector multiplication in 64-bit precision in each vector unit for a CM–5. The matrix shape is $P \times Q$.

TLB. Not all of the node memory is mapped at any given moment. Thus, the object of the loop partitioning and loop reordering is to

- maximize the use of data while in registers
- minimize the number of DRAM page faults
- minimize the number of TLB replacements

Figure 1.12 shows the impact of DRAM page faults on the local matrix-vector multiplication performance on the CM–5. The matrix shape is $P \times Q$. For the top curve, the stride along the $P$–axis is one and the innermost loop is along this axis. The template that fits in the register file is of shape $P \times Q$, with the second loop being of length $Q$. The templates are treated vertically before they are treated horizontally. Thus, the matrix is scanned by vertical panels. With this loop order, changing the stride along the $Q$–axis to one for $Q = 1024$ yields a stride of 1024 along the $P$–axis. The performance decreases by about a factor of 3.5. Changing the loop order for the tile such that the stride is one in the inner loop, leads to an inner–product like algorithm, and substantially improved performance. However, since the inner-product is not a particularly efficient operation on the CM–5, the performance is not quite as good as for stride one on the $P$–axis and an AXPY algorithm. This change in loop order/algorithm is performed automatically in the CMSSL, and the performance degradation is limited to about 30% instead of 75%.

Another effect that is visible in the top curve (labeled “pqpq_p_stride_1”) in Figure 1.12, although small, is the effect of TLB thrashing. This is the reason for the performance degradation for large values of $P$. Figure 1.13 gives an example where the effect of TLB thrashing is much more severe. The performance is reduced by almost a factor of two. The TLB thrashing can be reduced by introducing yet another level of loop partitioning/blocking. As seen in Figure 1.13 the performance can be restored to close to peak performance at a very small expense.
The efficiency of DAXPY in each vector unit for a CM-5. The vector length is $P$.

In general, each call to a CMSSL routine handles multiple matrix-vector multiplications, and an instance loop is included in determining tile shape and looping order. Thus, in the case of matrix-vector multiplication, the tile is a three-dimensional box of a size that fits in the register file. The shape of the box is a function of $P$, $Q$, and the number of instances, desired vector length, looping overhead, and strides along the different axis. The looping over boxes is determined so as to minimize DRAM page faults and TLB thrashing, as illustrated in Figure 1.14.

1.8.2 Global algorithm selection

We again use matrix operations for illustration. The idea that the operand with the largest number of elements should be kept stationary is very plausible for matrix-vector multiplication. An obvious algorithm is:

- Align the input vector with a row of the matrix
- Broadcast the input vector along columns
- Perform local matrix-vector multiplication
- Reduce along rows
- Align the result with the allocation of the output vector

Depending upon the relative layouts of the matrix and the vectors, no alignment may be required. However, with the canonical layout an alignment is required for both vectors, since a one-dimensional nodal array shape is used by default for vectors, while a two-dimensional nodal array shape is used for the matrix. (Except for matrices of extreme shape, this is indeed optimal.) For a one-dimensional nodal array shape also for the matrix, either the broadcast or the communication for the reduction is unnecessary. For the ideal (also the default) layout, the matrix-vector multiplication in the CMSSL is implemented using all-to-all communication as described in Section 1.9.2.
Figure 1.14 Tiling of the index space for optimum locality of reference.

For matrix-matrix multiplication, it is intuitive that if, for instance, the input instead of a vector was a matrix with two, or even a few columns (but many rows), the basic algorithmic strategy should not change much. Thus, for the computation $C \leftarrow A \times B$, where $C$ and $B$ are vector-like, $B$ should be aligned with $A$ and broadcast and partial products accumulated spatially and aligned with $C$. Similarly, if $C$ and $A$ are vector-like, but $B$ has a moderate aspect ratio, $B$ should be stationary and $A$ be aligned and broadcast. And, if both $A$ and $B$ are vector-like, but $C$ has a moderate aspect ratio, then $C$ should be stationary and $A$ and $B$ aligned and broadcast [MJ94].

In the CMSSL, a choice of algorithm as indicated above is made automatically in the matrix multiplication routine. The user need not be concerned with specifying what particular algorithm to choose for what matrix shapes and what machine size. The result is shown in Figure 1.15. In the bottom part of the plot, where the performance is relatively flat, a matrix-vector type algorithm is used, while an algorithm with the matrix $C$ stationary is used for the part with a rapidly increasing performance as a function of matrix size [MJ94].

### 1.9 Communications lib

The following communication primitives have been found important in the implementation of the CMSSL.

- One-to-all reduction/copy
Figure 1.15 Performance of the matrix multiplication function in the Connection Machine Scientific Software Library for the multiplication of a $P \times P$ matrix by a $P \times R$ matrix on Connection Machine system CM-200, 64-bit precision.

- All-to-all reduction/copy
- Gather/scatter
- One-to-all personalized communication
- All-to-all personalized communication
- Dimension(index) permutation
- Generalized shuffle permutations
- Scan/parallel prefix
- Lattice emulation
- Butterfly emulation
- Data manipulator network emulation (PM2I network emulation)
- Pyramid network emulation
- Bit-inversion
- Index reversal ($i \leftarrow N - i$).

Below we will discuss the need for some of the communication primitives above, and the techniques used to achieve high utilization of the communications bandwidth.

1.9.1 Broadcast

Broadcast and reduction from a single source to subsets of nodes, holding an entire row or column, are critical for the efficiency of computations such as LU and QR-factorization. In fact, many concurrent broadcast (and reduction) operations are desired in these computations as illustrated in Figure 1.16. Whether or not these
broadcast operations imply communication that interfere, depends upon the network topology and how the index space is mapped to the nodes.

On a binary cube network, entire subcubes are often assigned to a data array axis. In such a case, the broadcasts within the different columns in the index space do not interfere with each other, and the concurrent broadcast operation degenerates to a number of broadcasts within disjoint subsets of nodes. However, in other networks, such as the fat-tree, such a data mapping may not be feasible, and the simultaneous broadcast from several sources to distinct subsets of nodes may require a more complex routing for optimal bandwidth utilization. For instance, with the layout currently used on the CM-5, a two-dimensional array is either mapped to the nodes in row or column major order. A row major ordering implies that broadcast within rows can take place without interference between rows, since they are mapped to separate subtrees. However, for broadcast within columns contention occurs since the CM-5 fat-tree exhibits a reduction in bi-section width for the first two levels.

Global broadcast and reduction is used, for instance, in the conjugate gradient method. Even for the conjugate gradient method, several simultaneous broadcast operations may be required due to multiple-instance computation.

In implementing a broadcast algorithm it is important to exploit the bandwidth. A simple spanning tree may not accomplish this task. Consider a binary cube network (used in the CM-2 and CM-200). On an n-cube, using a binomial tree to broadcast $M$ elements from a node to all other nodes requires a time of $nM$ with the communication restricted to one channel at a time. The time is proportional to $M$ with concurrent communication on all channels of every node, all-port communication. However, the lower bounds for the two cases are $M$ and $\frac{M}{n}$, respectively [JH89b]. Thus, the binomial tree algorithm is nonoptimal by a factor of $n$ in both cases.

Multiple spanning trees rooted at the same node can be used to create lower bound algorithms [JH89b]. The basic idea is that the source node splits its data set into $\frac{M}{n}$ disjoint subsets and sends each subset to a distinct neighbor. Then, each of these neighbor nodes broadcasts the data set it received to all other nodes (except the original source node) using spanning binomial trees. By a suitable construction of the
trees, the $n$ binomial trees are edge disjoint, and the full bandwidth of the binary $n$-cube is used effectively.

The multiple spanning binomial tree algorithm is used for broadcasting on the Connection Machine systems CM-2 and CM-200. This broadcast function is part of the Connection Machine Run-Time System. The performance is illustrated in Figure 1.17 [Joh91]. As expected, the time to broadcast a given size data set decreases with the number of nodes to which the set is broadcast. Thus, broadcast exhibits a logarithmic speedup with respect to the number of nodes.

The CM-5 communication system is of the node limited variety and the optimizations are different.

1.9.2 All-to-all broadcast/reduce

Another important communication primitive is the simultaneous broadcast from each node in a set to every other node in the same set, *all-to-all broadcast* [JH89b]. This communication is typical for so called direct $N$-body algorithms, but it is also required in many dense matrix algorithms. In all-to-all reduction, reduction operations are performed concurrently on different data sets, each distributed over all nodes such that the results of the different reductions are evenly distributed over all nodes. Here we will illustrate their use in matrix-vector multiplication.

With the processing nodes configured as a two-dimensional nodal array for the matrix, and as a one-dimensional nodal array for the vectors, both all-to-all broadcast and all-to-all reduction are required in evaluating the matrix vector product. Figure
Figure 1.18 Data allocation on a rectangular nodal array.

1.18 illustrates the data allocation for both row major and column major ordering of the matrix. The data allocation shown in Figure 1.18 is typical on Connection Machine systems.

For a matrix of shape $P \times Q$, allocated to a two-dimensional nodal array in column major order, an all-to-all broadcast [FIL+88, JH89b, SW87a, SW87b] is required within the columns of the nodes for any shape of the nodal array and for any length of the matrix $Q$-axis.

After the all-to-all broadcast, each node performs a local matrix-vector multiplication. After this operation, each node contains a segment of the result vector $y$. The nodes in a row contain partial contributions to the same segment of $y$, while different rows of nodes contain contributions to different segments of $y$. No communication between rows of nodes is required for the computation of $y$. Communication within the rows of the nodes suffices.

The different segments of $y$ can be computed by all-to-all reduction within processor rows, resulting in a row major ordering of $y$. But, the node labeling is in column major order, and a reordering from row to column major ordering is required in order to establish the final allocation of $y$. Thus, for a column major order of the matrix elements, matrix-vector multiplication can be expressed as:

- All-to-all broadcast of the input vector within columns of nodes
- Local matrix-vector multiplication
- All-to-all reduction within rows of nodes to accumulate partial contributions to the result vector
- Reordering of the result vector from row major to column major order.

All-to-all broadcast or reduction is required also when a one-dimensional nodal array configuration is used for the matrix [MJ92a].

As in the case of broadcast from a single source, all-to-all broadcast on an $n$-cube can be performed in a time proportional to the lower bound. With each node initially holding $M$ elements, a time of $M(N-1)$ is required for communication restricted to a single port at a time, and a time of $\frac{M(N-1)}{n}$ is required for all-port communication.
Problem | Gather | Scatter |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>std alloc.</td>
<td>random alloc</td>
<td>std alloc.</td>
</tr>
<tr>
<td>3200 20-node brick elements</td>
<td>75</td>
<td>50</td>
</tr>
<tr>
<td>864 8-node brick elements</td>
<td>5.6</td>
<td>3.7</td>
</tr>
</tbody>
</table>

Table 1.10 The effect of randomization on gather and scatter performance. Times in msec on an 8K CM–200.

[JH89b]. A simple, yet optimal, all-port algorithm for all-to-all broadcast uses n Hamiltonian paths for each node. For all-to-all broadcast, the Hamiltonian paths need not be edge-disjoint [JH89b, BJ92].

1.9.3 Gather/Scatter

Gather and scatter operations on regular grid data represented as one or multidimensional arrays, as well as irregular grid data, is critical for the performance of many scientific and engineering applications. On the Connection Machine Systems, gather and scatter operations on regular grids are supported through PSHIFT (for polyshift) [GBJ94, TH93b], which allows the programmer to specify concurrent shift operations in one or both directions of one or multiple axes. On the CM–2 and CM–200 with Gray-coded axes, PSHIFT concurrently performs all the data exchanges requiring communication between nodes. In effect, PSHIFT provides an effective means of emulating lattices on binary cubes. (A further level of optimization is provided by the so called stencil compiler [BHMS91], which in addition to maximizing the concurrency in internode communication (using PSHIFT), avoids unnecessary local memory moves and uses a highly optimized register allocation in order to minimize the number of load and store operations between local memory and the register file in the floating-point unit.)

For gather and scatter operations on unstructured grid computations, the general router on the Connection Machine systems is used. However, two means of improving the performance are provided; randomization of the data allocation, and savings of the routing information for repetitive gather/scatter operations. Table 1.10 [MJ92b] summarizes the effect of randomization of the data allocation for a few meshes.

The performance enhancement of 1.5 – 2.25 is typical.

1.9.4 Personalized communication

In one-to-all personalized communication, a node sends a unique piece of data to every other node. An example is matrix computations where a node holds an entire column, which may need to be redistributed evenly over all the nodes as in some algorithms for matrix-vector multiplication [MJ92a]. In all-to-all personalized communication, each node sends unique information to all other nodes. Personalized communication is not limited to matrix transposition, but encompasses operations such as bit-reversal, transposition or bit-reversal combined with a code change (such as the conversion
between binary code and binary–reflected Gray code), and bit–inversion. We now illustrate the significance of personalized communication in computing the FFT on a multiprocessor.

In computing the FFT on distributed data, one possibility is to exchange data between nodes and have one of the nodes in a pair compute the “top” and the other compute the “bottom” of the butterfly requiring data from the two nodes. This type of algorithm is used on the Connection Machine systems CM–2 and CM–200 [JK92].

When there are two or more elements per node, then an alternative is to perform an exchange of data between nodes, such that each node in a pair computes one complete butterfly. The sequence of exchanges required for the FFT amounts to a shuffle, as illustrated below, where the | separates node address bits to the left and local memory address bits to the right:

**Example 1.**

<table>
<thead>
<tr>
<th>Address Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>(543210)</td>
</tr>
<tr>
<td>(534215)</td>
</tr>
<tr>
<td>(543214)</td>
</tr>
<tr>
<td>(54213)</td>
</tr>
<tr>
<td>(54312)</td>
</tr>
<tr>
<td>(543211)</td>
</tr>
<tr>
<td>(543210)</td>
</tr>
</tbody>
</table>

Thus, the end result of the sequence of exchanges is a shuffle on the node address field. Each step is equivalent to the transposition of a collection of 2 × 2 matrices.

In practice, for a one–dimensional transform, there are typically several local memory bits. For performance, under many models for the communication system, minimizing the number of exchange steps is desirable, i.e., instead of performing bisections it is desirable to perform multisectiions including all local memory bits. Thus, for instance, with two local memory bits, four–sectioning should be carried out, as shown in Example 2. Multisectioning is used for the FFT on the CM–5.

**Example 2.**

<table>
<thead>
<tr>
<th>Address Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>(6543210)</td>
</tr>
<tr>
<td>(6543265)</td>
</tr>
<tr>
<td>(6543243)</td>
</tr>
</tbody>
</table>

19/9/1994 17:53—PAGE PROOFS for John Wiley & Sons Ltd (using jwebmc01, Vers 01.01 MAY 1992)—P4
Example 2 was deliberately chosen such that the exchanges cannot simply be treated as digit exchanges with increased radix for the digit, but must indeed be treated as exchanges with digits of different radices. Moreover, the last few exchange steps were made such that the final order represents an $m$-step shuffle on the nodal address bits, where $m$ is the number of bits used to encode the first exchange. This node address ordering requires a local memory shuffle to restore the original local memory ordering. (In practice, it may, in fact, be preferable to avoid the local memory reordering by performing the last exchange such that local memory is normally ordered, which would leave the node addresses in an order corresponding to two shuffles: one $m$-step shuffle on all $n$ node address bits, one $n \mod m$ shuffle on the last $m$ bits.)

In multidimensional FFT, all of local memory should be considered in performing the multisectioning, see [Joh92b].

The FFT produces the results in bit-reversed order with respect to the indices. Thus, establishing a normal index map in the output domain requires an unshuffle with bit-reversal. Figure 1.19 [Joh92b] shows an example.

The lower bounds for all-to-all personalized communication depends upon the network and communication system. For a binary $n$-cube with $M$ data elements per node, the lower bound is $\frac{nM}{2}$ for communication restricted to a single port per node and $\frac{nM}{2}$ for all-port communication [JH89b]. Balanced spanning trees [HJ89] provide for optimal all-to-all personalized communication with all-port communication. A balanced spanning tree has $N/n$ nodes in each of the $n$ subtrees of the root. The use of $n$ rotated spanning binomial trees rooted in each node also yields the desired complexity [JH89b]. The CM-5 network is node limited, and the communication time is simply limited by the bandwidth at the node. However, the order in which the data is sent and retrieved from the network has a measurable impact on the performance.

In our FFT example above, several all-to-all personalized communications were performed in succession. In such a case, it may be of interest to minimize the time elements spend in transition from source to destination in order to minimize pipeline delays. Algorithms with a minimal transition time are presented in [JH92c].

Bit-reversal with an equal number of dimensions assigned to the node address field and the local memory address field constitutes one form of all-to-all personalized communication. The performance on various sizes of the CM-2 is shown in Figure 1.20 [Joh91]. As expected, the execution time is almost independent of the machine size for a fixed size data set $Mn$. The increase in the execution time is largely due to the fact that local memory operations cannot be performed in parallel. Thus, there is a term proportional to $n$ in addition to the constant term.
Two step reordering after 4-section based radix-2 FFT. First step, bit-exchange between nodes; second step, bit-exchange between local memory and node addresses. \( p \leq 2n - m \).

All-to-all personalized communication on the Connection Machine.
Index reversal is another important permutation used for instance in the computation of real-to-complex FFT. For this computation, the standard algorithm requires that data with indices $i$ and $N - i$, $0 \leq i < N$ be operated upon in a preprocessing or postprocessing step for the FFT [Swa86, Tem80]. In binary-coded data, the index reversal required for the FFT corresponds to a two's-complement subtraction (bit complement plus one).

However, in the case of the real-to-complex FFT on a one-dimensional array with binary-coded data, the first step in one of the most common algorithms is to perform a complex-to-complex FFT on the array viewed as a half-size, one-dimensional array of complex data points. The result is shown in Figure 1.21. The Figure also shows that the postprocessing matching indices $i$ and $N - i$ correspond to bit-inversion in subcubes of the form $00\ldots01xx\ldots x$, with the inversion being performed on the bits denoted by $x$.

If there are more than one complex data point per node, then the communication requirements depend upon how the indices are aggregated to the nodes. In consecutive data allocation, the communication pattern between nodes is the same as if there was only one element per node. In a cyclic data allocation, the communication for the first complex local memory location across all nodes is the same as if there was a single element per node. The communication for the second and all subsequent complex local memory locations is bit-complement on the entire node address.

Bit-inversion also occurs in the alignment of the operands in matrix-matrix multiplication on three-dimensional nodal array configurations [Joh93a].

Concurrent communication for bit-inversion on binary cubes is straightforward. For
instance, multiple exchange sequences starting in different dimensions and progressing through the dimensions in increasing (or decreasing) order cyclicly can be used.

1.9.6 Shuffle operations on binary cube networks

Computing the FFT through multisectioning results in an $m$-step shuffle on the index space, where $m$ is the number of bits encoding the first digit exchange. Restoring the original index order corresponds to an unshuffle (except for the FFT which in itself implements a bit-reversal). Reshaping the nodal array for a given data array also represents a general shuffle operation. For instance, changing the allocation

\[
(a_3 a_2 m_2 a_1 a_0 m_1 m_0)
\]

\[
(0 2 1 0 3 2 3 1)
\]

to the allocation

\[
(m_2 m_1 m_0 a_2 a_1 a_0)
\]

\[
(0 2 1 0 3 2 3 1)
\]

where $x_i$ and $y_i$ denote bits encoding an $x$-axis and $y$-axis, respectively, and $a_i$ denotes nodal address bits and $m_i$ local memory address bits, constitutes a generalized shuffle, or dimension permutation. The dimension permutation is: $a_3 \leftrightarrow a_1 \leftrightarrow m_1 \leftrightarrow a_2 \leftrightarrow a_0 \leftrightarrow m_2 \leftrightarrow a_3$. In this example, the reshaping resulted in a single cycle on the dimensions. In general, the reshaping may result in several cycles, just as the $m$-step shuffle in general can be decomposed into several cycles.

A shuffle can be implemented as a sequence of successive pairwise dimension exchanges starting in any position. In a binary cube, such exchanges imply communication in two cube dimensions for each step, when both dimensions in an exchange are nodal address dimensions. However, it is also possible to use a fixed memory dimension for each exchange. If the first exchange is repeated as a last exchange, then the result is a shuffle on all bits but the fixed exchange dimension. For a shuffle on $n$ bits, the first alternative requires $n-1$ exchanges while the last requires $n+1$ exchanges. Thus, at the expense of two additional exchanges, each exchange only involves one nodal address dimension. In [JH92b] we present algorithms that are nonoptimal by two exchanges at most, regardless of the number of cube dimensions in the shuffle and the data elements per node. The algorithms use multiple exchange sequences (embeddings), exploiting the fact that a shuffle can be performed as a sequence of exchanges starting at any bit and proceeding in order of decreasing dimensions cyclicly.

1.10 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
Table 1.11  Peak performance per node for some BLAS functions for the CM-5.

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of instances</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>DDOT</td>
<td>44.01</td>
</tr>
<tr>
<td>DNORM2</td>
<td>125.85</td>
</tr>
</tbody>
</table>

Table 1.12  Problem size for half of peak performance for BLAS functions local to a CM-5 node.

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of instances</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>DSCAL</td>
<td>16</td>
</tr>
<tr>
<td>DAXPY</td>
<td>11</td>
</tr>
<tr>
<td>DDOT</td>
<td>35</td>
</tr>
<tr>
<td>DNORM2</td>
<td>44</td>
</tr>
</tbody>
</table>

functions. Table 1.11 provides some evidence for this claim for the CM-5. Particular emphasis has been placed on reducing the problem sizes offering half of peak performance, Table 1.12.

Some peak global performance data were given in Section 1.2. For the distributed DBLAS, communication may degrade the peak performance with about 25%.

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 1.2, Section 1.2.

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.

19/9/1994 17:53—PAGE PROOFS for John Wiley & Sons Ltd (using jwebmc01, Vers 01.01 MAY 1992)—P4
The communications functions and some of the numerical routines in the CMSSL relies mainly 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-8808285 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


[Joh92a] Zdenek Johan. Data Parallel Finite Element Techniques for Large-Scale Com-


19/9/1994 17:53—PAGE PROOFS for John Wiley & Sons Ltd (using jwcbmc01, Vers 01.01 MAY 1992)—P4
High Performance, Scalable Scientific Software Libraries


19/9/1994 17:53—PAGE PROOFS for John Wiley & Sons Ltd (using jwcbmc01, Vers 01.01 MAY 1992)—P4