BASIC SPARSE MATRIX COMPUTATIONS ON MASSIVELY PARALLEL COMPUTERS *

W. Ferng†, K. Wu ‡, S. Petiton §, Y. Saad ¶

February 8, 2010

Abstract

This paper presents a preliminary experimental study of the performance of basic sparse matrix computations on the CM-200 and the CM-5. We concentrate on examining various ways of performing general sparse matrix-vector operations and the basic primitives on which these are based. We compare various data structures for storing sparse matrices and their corresponding matrix – vector operations. Both SPMD and Data parallel modes are examined and a comparison of the two modes is made.

1 Introduction

As parallel processing is progressively gaining ground it is becoming commonplace to port a standard sequential code into a massively parallel computer. Among the types of computations encountered in scientific computing, few are more challenging to implement on high performance computers than sparse computations. One reason is that sparse matrices are often irregularly structured and this implies that costly irregular communications will be needed between processors. A secondary reason is that in sparse matrix computations, the number of floating point operations to be performed is often of the same order of magnitude as that of the number of data movements. As a result, it is often the case that computation times are bound by communication speeds rather than the arithmetic speeds. It is therefore far more important to keep communication costs down than it is in the dense matrix case.

One of the most common and basic operations in sparse matrix techniques is the matrix-by-vector product which arises, for example, in iterative methods for solving large linear systems and in eigenvalue algorithms. Analyzing performance of such operations

*This work was supported by Army Research Office contract number DAAL03-89-C-0038

†Army High Performance Computing Research Center, University of Minesota

‡Computer Science Department,University of Minnesota

§Site Experimental en Hyperparallélisme, Etablissement Technique Central de l'Armement, France, and Yale University, Computer Science Dept.

¶Computer Science Department, University of Minnesota

is important not only for the numerical methods just mentioned but also because it will give a good indication of the performance that one should expect to achieve in other related types of computations. For example, finite element or finite volume techniques on unstructured grids share basically the same computational kernels as general sparse matrix techniques. They are dominated by computational constructs that involve indirect addressing. Because these types of computations are so common in scientific computing, it is our view that a high performance computer should not be deemed satisfactory if it does not deliver a reasonable performance on the important class of *irregular computations*. In the past the performance of a computer has often been stated in terms of its achievable performance on dense computations. Unfortunately, this may give a view that is biased towards dense computations, and can be quite misleading considering that the two types of computations have completely different characteristics. Thus, the early CRAY machines which did not provide hardware instructions for scatter and gather operations, were quickly found lacking by users of sparse matrix techniques and the missing instructions were added within a short time by the manufacturer. In [14] a benchmark code specifically designed for testing the performance of computers on sparse matrix computations, was proposed and used to compare a number of shared memory computers. We believe that there is a need for designing a similar benchmark code for distributed memory computers in order to avoid that massively parallel computers be benchmarked by their performance on dense matrix computations.

Thinking Machines Corporation's new line of Connection Machines offers the distinct advantage that we can experiment with two significantly different modes of parallel computing, namely a particular SIMD mode referred to as the data-parallel mode, and a particular MIMD mode referred to as the Single Program Multiple Data (SPMD) mode. One of our goals in this paper is to compare the two models specifically for sparse matrix computations. Roughly speaking, the SPMD mode gives us the flexibility of programming communication by means of message passing between the nodes. The SIMD mode is far simpler to program but allows fewer choices regarding the organization of data transfers.

We should stress that the CM-5 available to us for the experiments is an α machine and that it is not equipped with vector processors. Nevertheless, as will be seen, some of the performances can be extrapolated quite easily. In addition, it is quite important to still examine the performance of the machine equipped with SPARC floating point arithmetic units in order to better understand the relations between arithmetic hardware and communication hardware and their impact on the overall speed. We will examine these relations more fully when the vector chips will be installed.

We start by describing the CM-5 and will briefly introduce the data parallel and SPMD programming paradigms. Then the formats used to store sparse matrices will be outlined as well as the various kernels for implementing matrix-by-vector multiplications. We will then describe a number of experiments in data parallel and then in SPMD mode.

2 Overview of the CM-2 and CM-5 Computers

In this section, we summarize the main architecture features and communication models for the Connection Machines CM-200 and CM-5. The different execution models will also be described and compared from the software point of view.

2.1 Connection Machine Model CM-2/CM-200

Architecture. The Connection Machine model CM-2 is an SIMD computer sometimes refered to as *data parallel computing system*. A Connection Machine system consists of a front-end computer, a large number of data processors, a few sequencers that control the data processors, an interprocessor communication network, and I/O controllers [1]. The system may be configured with up to 64K physical processors. Here, and throughout this paper, "K" stands for 1024.

Parallel processing instructions issued by the front-end computer are received by one of the sequencers, which interprets them to produce a series of single-cycle nanoinstructions. The nanoinstructions are then broadcast over the instruction bus to the data processors. Each data processor contains an arithmetic-logic unit (ALU), 64K or 256K bits of bit-addressable memory, communication and I/O interfaces, and optional floating-point accelerator. The floating-point accelerator consists of a floating-point interface chip and a floating-point execution chip, and is shared by every 32 data processors. Therefore, a fully configured CM-2 may contain 2K floating-point processors and 2 gigabytes of memory [1].

Communication Model. Each data processor of the CM-2 system has its own memory and the memory is bit-addressable. Each data processor can access its local memory at a rate of 5 Megabits per second [1]. Thus, a fully configured CM-2 has 2 gigabytes of memory that can in theory be accessed at about 300 gigabits per second.

The CM-2 system provides three forms of communication within the parallel processing unit: routing, NEWS, and scanning [1]. The *router* which allows any processor to communicate with any other processor is the most general communication mechanism. The *NEWS grid* is a faster but more structured communication mechanism. It allows processors to pass data in a multi-dimensional rectangular topology. Special hardware and the inherent hypercube topology of the CM-2 make this type of mechanism significantly faster than the general routing. *Scanning* is a more powerful operation on the NEWS grid that combines communication and computation, to perform certain "prefix" operations such as the computation of an inner product in all processors simultaneously.

To move data from their initial locations in local memories to specified locations in local memories of other hypercube nodes, the communication operations are decomposed into a number of memory to memory *swap* operations. Between swaps, data must be buffered in the local memories of intermediate nodes.

The communication hardware in the Connection Machine CM-2 comprises a large number of single bit data paths plus various mechanisms for writing data to and reading data from these paths. The architechure can be viewed in two different ways: the *CM*

chip model and the *Sprint chip model*. In the CM chip model, a CM-2 with p processors is regarded as a $(\log_2 p - 4)$ dimensional hypercube. Each CM chip has $(\log_2 p - 4)$ data paths attached to it. And there are $p/16$ such chips. In the Sprint chip model, a CM-2 with p processors is regarded as a $(\log_2 p - 5)$ dimensional hypercube and has $p/32$ such chips [4].

The CM-200 at AHPCRC. The Connection Machine Model CM-200 available to us for the experiments at the Army High Performance Computing Research Center (AHPCRC) is an upgraded version of the CM-2 which has a clock rate that is higher than that of the CM-2. It is configured with 32K data processors, 1K 64-bit floating-point processors, 1 Mbyte memory, and 4 sequencers.

2.2 Connection Machine Model CM-5

In the Fall of 1991, the Thinking Machines Corporation announced its latest massively parallel computer: the Connection Machine Model CM-5. One significant feature of the CM-5 is that it has been designed to support both SIMD and MIMD models. The basic components of the CM-5 include hundreds or thousands of parallel Processing Nodes (PNs), each with its own memory, one or more Control Processors (CPs), two global communication networks, high bandwidth I/O subsystems, and mass storage devices, e.g., the DataVault. In this section, we briefly summarize some of the most important aspects of this new massively parallel architecture from the Connection Machine CM-5 Technical Summary by Thinking Machine Corporation [2].

Processors. A CM-5 computer may consist of hundreds or thousands of processors. A system administrator may divide these processors into groups, known as *partitions*. There is a seperate processor, called *Control Processor* (CP), for each partition. A control processor is essentially like a standard high-performance workstation computer. It consists of a standard RISC microprocessor (a Sun SPARC microprocessor in the current release), associated memory and memory interface, a Network Interface (NI) providing access to the communication networks, and other devices and interfaces. A control processor acting as a partition manager (PM) controls each partition and communicates with the rest of the system through the communication networks. It runs a version of the UNIX operating system which allows multiple users to access the partition in a time sharing mode.

The basic components of a Processing Node include a RISC microprocessor (SPARC chip from Sun microsystems), a memory subsystem, and a Network Interface (NI) all connected to a standard 64-bit bus. The RISC microprocessor is responsible for the instruction fetch and execution, for processing data, and for controlling the NI. The memory subsystem consists of a memory controller, 2 Kbyte boot ROM, and either 8, 16, or 32 Mbyte of DRAM. The path from each memory to the memory controller is 72-bit wide. The microprocessor also has a 64 Kbyte cache that holds both instruction and data. Finally, the Network Interface connects a PN to the rest of the system through the Control Network and Data Network as is explained later in this section.

A PN may optionally contain an arithmetic accelerator. In this configuration, each PN has a full 32 Mbyte of memory, four banks of 8 Mbyte each. The memory controller is replaced by four vector units (VU), each with a dedicated 72-bit path to its associated memory bank, providing peak memory bandwidth of 128 Mbytes/sec per vector unit. The vector units execute vector instructions issued by the RISC microprocessor. Each vector unit has 32 Mflops peak 64-bit floating-point performance and 32 Mflops peak 64-bit integer performance. Together, each PN provides 512 Mbyte/sec memory bandwidth and a peak performance of 128 Mflops in 64-bit floating-point operations. However, we would like to emphasize that these vector units are not available in the current release of CM-5.

Communication Networks. Every control processor and parallel processing node in the CM-5 is connected to two scalable interprocessor communication networks, the *Control Network* (CN) and the *Data Network* (DN). In general, the Control Network is used for operations that involve all the processors at once, for example operations such as synchronization and instruction broadcasting. The Data Network is used for bulk data transfers where data has a single source and destination. A third network, called the *Diagnostics Network*, which keeps tabs on the status of the running hardware, is visible only to the system.

The Control Network also contains integer and logical arithmetic hardware for carrying out reduction operations, where every processor provides a value and all values are combined by the CN to produce a single result. CN operations may be overloaoed with the opertions by the processors themselves.

In theory, the Data Network provides enough bandwidth for every Network Interface to sustain data transfer rates of 20 Mbytes per second to any other NI within its group of 4; 10 Mbytes per second to any other NI within its group of 16; and 5 Mbytes per second to any other NI in the system. Thus, the best-case to worst-case performance ratio is a factor of at most 4.

At any time, any processor may send a message to any processor in the user task. This is done by first writing the destination processor number, and then the data to be sent, to the control registers in the Network Interface. Once the Data Network has accepted the message, it assumes all responsibility for delivery of the message to its destination. Moreover, the operation of the Data Network is independent of the PNs, which may carry out unrelated computations while the message are in transit.

The CM-5 at AHPCRC. The Connection Machine CM-5 available to us for the experiments at AHPCRC is an α-site machine with a total of 544 PNs each having 16 Mbyte of local memory. It can be configured into two partitions consisting of 512 and 32 PNs respectively, or three partitions consisting of 256, 256, and 32 PNs respectively.

2.3 SIMD versus SPMD

The CM-2 is a *Single Instruction Multiple Data* (SIMD) computer. The user program is compiled on the front-end computer, the data is mapped onto and resides in the data

processors. Each processor must execute the same instruction issued by the front-end and broadcast to all the processors at the same time. The user can write a program in low level language, e.g., Paris, to carry-out desired data mappings and interprocessor communications. However, it may be preferable and far more convenient to use the manufacturer's early version of FORTRAN-90, called CM-FORTRAN with a few communication primitives to achieve the same goal without sacrifing too much performance.

The CM-5 provides a similar programming environment and can also be viewed as a data parallel computer. In addition, it provides a *Single Program Multiple Data* (SPMD) execution model, which in essence is a restricted *Multiple Instruction Multiple Data* (MIMD) model. Next, we describe and compare the different execution models available on each machine.

Slicewise Execution Model on the CM-2 Beginning with version 1.0, the CM-FORTRAN compiler offers an alternative execution model called *slicewise model*, as opposed to the so-called *Paris* or *fieldwise model* [3]. The slicewise model takes full advantage of the registers and vector-processing capabilities of the 64-bit floating-point accelerator unit (FPU). All CM processors are organized into *Processing Elements* (PEs), each containing 32 bit-serial processors, some memory, one optional FPU chip, and other associated hardware. Thus, a Connection Machine executing in the slicewise model is using machine-size/32 PEs, or, 1-K PEs for a 32K CM.

When invoked for the slicewise mode, the compiler views the CM as a set of vector processors. Data is stored in memory in 32-bit words, with the memory of each of a node's 32 bit-serial processors holding a one-bit slice of a word, rather than the whole word. The compiler itself does not perform CM memory management or interprocessor communication. Instead, it calls the functions of a run-time library. The run-time system lays out arrays in CM memory differently depending on the number of PEs available to execute the program. The total number of memory locations allocated is a multiple of 4 times the number of PEs executing the program.

Under the Paris model, where all array operations are memory-to-memory, floating-point operations on double-precision numbers are twice as expensive as operations on single-precision numbers. In slicewise computations, double-precision floating-point arithmetic does not cost any more than single-precision arithmetic. However, loads and stores to/from memory of double-precision numbers will still cost twice as much as single-precision loads and stores [3].

Message-Passing Model on the CM-5. In the SPMD model on the CM-5, each PN holds an identical copy of the same program, called the node program, and executes its own copy concurrently. The host (control) processor can execute a seperate program independently. Access to the node program is possible via subroutine calls from the host program. Data can be exchanged among processors through the Data Network. There is no global synchronization necessary since the execution is data driven.

The CM-5 provides a message-passing library, called CMMD, for interprocessor communications. The current release of CMMD supports primarily blocking message sending

and receiving. It permits concurrent processing in which synchronization takes place between matched pairs of sends and receives among processors. When not communicating, computing on each node processor proceeds asynchronously.

As was already mentioned, one advantage of the SPMD model over the SIMD model is that the node can be programmed to handle the communications among processors explicitly. In the SIMD model, the choice is limited (Paris is not available on the CM-5), and the data mappings and communications are not visible to the users. However, the SIMD model has the advantage of simplicity. One can essentially use the FORTRAN-90 like constructs with a few communication primitive routines to achieve a reasonable performance. There are currently a number of ongoing discussions regarding the future generations of FORTRAN, labelled under the generic name HPF FORTRAN, and it is quite conceivable that standards for such communication primitives will appear in the near future. The SPMD model may make the communication more efficient and flexible, but puts the burden of managing communications on the user.

3 Basic Sparse Matrix Computations

3.1 Storage formats for Sparse Matrices

There are perhaps as many ways of storing sparse matrices as there are scientific applications involving such matrices. However, we have elected to restrict our attention to just a few of them that have the best potential for delivering good performance on massively parallel computers. We consider six basic different storage formats and some variants of them. Using some of the terminology of SPARSKIT [13] these are the banded format (BND), the Compressed Sparse Row/Column (CSR/CSC) formats, the diagonal format (DIA), the Ellpack-Itpack generalized diagonal format (ELL), the Sparse General Pattern format (SGP), and the Coordinate format (COO) used by the CMSSL library.

Banded format(BND) Many matrices can be put in a banded form. Since the original matrix is rarely banded it is often necessary to pad the matrix with zero elements. Thus, we represent all the elements of a few super/subdiagonals that include all the nonzero elements in the matrix. The nonzero elements are usually stored in a rectangular array, either row-wise or column-wise as is done in Linpack.

Diagonal format(DIA) This scheme is motivated by the fact that the nonzero elements of many matrices are located in a small number of diagonals. As a result we can store those diagonals in a rectangular array together with their offsets with respect to the main diagonal. As with the banded format, a two-dimensional array is used to store nonzero diagonals. A separate array *offset* is used to store the offset of each diagonal.

Ellpack-Itpack format (ELL) This is a straightforward generalization of the diagonal storage scheme. We use a real array $aa(i, j), i = 1, n; j = 1, ncol$ to store the nonzero

elements a_{ij} of row i, for $i = 1, \ldots, n$. Thus, the maximum number of nonzero elements per row must not exceed *ncol*. If there are fewer than *ncol* elements in a row, the row $VAL(i, *)$ must be padded with zeros. We need an integer companion array *ja(i,j)* to store the column indices of the each nonzero element. This is clearly a more general scheme than the two prevous ones. However, it may be inefficient if many rows have fewer than *ncol* elements. A column-based scheme can also be defined similarly.

Sparse General Pattern format (SGP) This is a variation of the Ellpack-Itpack format introduced in [10] for distributed memory massively parallel machines using a data parallel programming model. It can be viewed as an augmented version of the row-oriented Ellpack-Itpack format which allows to transpose an $N \times nc$ Ellpack-Itpack matrix on $N * nc$ processors without communication to obtain the address of the virtual processor where each element is to be sent. This is useful if we need the pattern of both A and its transpose. In addition to the two 2-D arrays used for the Ellpack-Itpack row-wise format (i.e. $aa(1 : n, 1 : nc)$ and $ja(1 : n, 1 : nc)$), we use an additional 2-D integer array $ic(1 : n, 1 : nc)$ in which $ic(i, j)$ holds the index of a nonzero element in column i of the matrix, i.e., an element in the corresponding compressed column-oriented data-structure. In other words, SGP is the union of a full row-oriented ELL data structure for A together with a pattern only (real array omitted) column oriented ELL data structure for A. In a parallel programming model it is assumed that on each virtual processor will hold the element $aa(i, j)$ along with $ja(i, j)$ and $ic(i, j)$. As a result we only need to send the value stored in $aa(i, j)$ to the virtual processor $(ja(i, j), ic(i, j)$ to perform a matrix transposition operation. A minor restriction of this data structure is that we need to assume that the maximum number of nonzero elements in the rows is of the same order as that of the maximum number of nonzero elements in the columns. Note that when the matrix has a symmetric pattern, the additional array *ic* is not needed. We can also define similarly a column oriented SGP format but we omit the details.

Compressed Sparse Row/Column(CSR/CSC). In the row-based CSR format, an array $aa(i)$ is used to store the nonzero elements of a matrix, from row 1 to row n, in succession. We also need to store the column indices of each entry corresponding to the element $aa(i)$ in the integer array $ja(*)$. Finally, we need a pointer array $ia(i)$ which points to the beginnings of each row in aa, ja. Thus, the i-th row starts in position $ia(i)$ in the arrays aa, ja and ends in position $ia(i + 1) - 1$. This scheme is quite efficient in terms of storage. It is also very general and quite popular. The Compressed Sparse Column scheme is a similar scheme that is column oriented.

The Coordinate Format (COO) The coordinate format (COO) consists of three one-dimensional arrays: a real array $aa(1 : nnz)$ containing the non-zero elements of the sparse matrix in any order, where nnz is the number of non-zeros, and two integer arrays $ia(1 : nnz)$ and $ja(1 : nnz)$ containing the corresponding row indices and column indices, respectively. This scheme is as general as the CSR format, but not quite as efficient from the memory requirement point of view. On the other hand, it is attractive because of its

simplicity and the fact that it is very commonly used in software packages. In particular, the sparse matrix-vector product routine in the CMSSL library adopts a slight variant of the COO format.

3.2 Multiplication Algorithms

The computational kernels for performing sparse matrix operations such as matrix vector products are intimately associated with the data structures used. However, there are a few general approaches that are common to different algorithms for matrix - vector - products which we now consider. In the case of a dense matrix, the operation $y = Ax$ can be performed in many different ways, two popular schemes being (1) the inner product form described in Algorithm 3.1 and (2) the SAXPY form described by algorithm 3.2.

ALGORITHM **3.1 Dot product form – dense case**

```
do i = 1, n
   tmp = 0
   do j = 1, n
      tmp = tmp + a(i,j) * x(j)
   enddo
   y(i) = tmp
enddo
```

ALGORITHM **3.2 SAXPY form – dense case**

```
y(1:n) = 0.0
do j = 1, n
   do i = 1, n
      y(i) = y(i) + a(i,j) * x(j)
   enddo
enddo
```

In the sparse case, the multiplication can also be performed in one of these two modes. Thus, Algorithm 3.3 is a sparse translation of Algorithm 3.1 for the case where the matrix is stored in CSR format.

ALGORITHM **3.3 CSR format – Dot product form**
Sparse matrix-vector multiplication for CSR format.

```
do i = 1, n
   tmp = 0
   do j = ia(i), ia(i+1)-1
      tmp = tmp + a(j)*x(ja(j))
   enddo
   y(i) = tmp
enddo
```

Assuming that the matrix is stored by columns (CSC format) we can perform the matrix-vector product by the following algorithm.

ALGORITHM **3.4 CSC format – SAXPY form**
Sparse matrix-vector multiplication for CSR format.

```
y(1:n) = 0.0
do i = 1, n
   do j = ia(i), ia(i+1)-1
      y(ja(j)) = y(ja(j)) + x(i) * a(j)
   enddo
enddo
```

In the sparse case, a third possibility emerges, which consists of performing the product by diagonals. This third possibility bears no interest in the dense case. Again there are different variants related to different orderings of the loops in the basic FORTRAN program.

ALGORITHM **3.5 DIA format – dot product form**
Sparse matrix-vector multiplication for DIA format.

```
do i = 1, n
   tmp = 0.0d0
   do j = 1, ncol
      tmp = tmp + a(i,j)*x(i+offset(j))
   enddo
   y(i) = tmp
enddo
```

ALGORITHM **3.6 DIA format – TRIAD form**
Sparse matrix-vector multiplication for DIA format.

```
y = 0.0d0
do j = 1, ncol
   do i = 1, n
      y(i) = y(i) + a(i,j)*x(i+offset(j))
   enddo
enddo
```

Similarly there are also two basic ways of implementing a matrix vector product when using the Ellpack format.

ALGORITHM **3.7 Ellpack format – dot product form**
Matrix-vector multiplication using dot-products for the Ellpack format.

```
do i = 1, n
   yi = 0
   do j = 1, ncol
      yi = yi + a(j,i) * x(ja(j,i))
   enddo
   y(i) = yi
enddo
```

As will be seen later, there are two ways of implementing the above basic algorithm in SIMD mode on the CM2. We can either use a temporary array to store $x(ja(j,i))$ or we can use the SUM operation provided as part of the CM FORTRAN library.

ALGORITHM **3.8 Ellpack format – SAXPY form**
Matrix-vector multiplication based on SAXPY operation for the Ellpack format.

```
 y(1:n) = 0
 do i = 1, ncol
    do j = 1, n
       y(ja(j,i)) = y(ja(j,i)) + a(j,i) * x(i)
    enddo
enddo
```

ALGORITHM **3.9** *BND format Sparse matrix-vector multiplication for BND format. Variable* `p` *is the lower bandwidth of the matrix.*

```
do i = 1, n
   yi = 0
   do j = 1, bandwidth
      yi = yi + a(j,i)*x(i-p-1)
   enddo
   y(i) = yi
enddo
```

It is important to have an idea of which variant of the same basic algorithm will perform better in a 'generic case'. For example, can we expect the dot-product variants to perform generally better than the SAXPY one? To answer this question we tested two subroutines to carry out matrix-vector multiplication for the same matrix with completely random structure and random values on the CM-5. Table 20 shows the difference in the speed between the inner-product and the SAXPY forms. A detailed analysis is given in Section 5 for the CM-5.

In general we should expect the banded format to give the best performance for large enough bandwidth. The diagonal format should be slightly better than the Ellpack format because it involves less communication and also because of the regularity of the communication involved.

3.3 Basic Kernels

All the above algorithm are build up essentially from one of the following three types of level-1 BLAS type kernels with slight variations: dot-product, SAXPY's and TRIAD's. Note that the triad operation is not part of BLAS-1. These can be defined as follows.

1. DOT ($a = \sum x_i y_i$)
   a = 0
   do i = 1, n
      a = a + x(i) * y(i)
   enddo

2. SAXPY ($y_i = y_i + \alpha x_i$).
   do i = 1, n
      y(i) = y(i) + alpha * x(i)
   enddo

3. TRIAD ($y_i = y_i + a_i x_i$).
   do i = 1, n
      y(i) = y(i) + a(i) * x(i)
   enddo

As was seen in algorithms 3.3, 3.4, etc..., most of the kernels in sparse matrix computations, use versions of the above primitives that resort to indirect addressing. The indirect-addressing versions, are listed below. Note that there is a standard sparse BLAS defined by Grimes and Lewis [6] and that DOTI and sparse blas kernels DOTI and AXPYI given below follow the description of the standard.

1. DOTI ($a = \sum x_i y_{ind(i)}$).
   a = 0
   do i = 1, n
      a = a + x(i) * y(ind(i))
   enddo

2. SAXPYI ($y_{ind(i)} = y_{ind(i)} + \alpha x_i$).
   do i = 1, n
      y(ind(i)) = y(ind(i)) + alpha * x(i)
   enddo

3. TRIADI ($y_{ind(i)} = y_{ind(i)} + a_i x_i$).
   do i = 1, n
      y(ind(i)) = y(ind(i)) + a(i) * x(i)
   enddo

4 Numerical Experiments in Data Parallel Mode

In this section, we discuss the implementation and performance of a few algorithms for sparse matrix-vector multiplication with different storage formats on the Connection Machines using the data parallel model. We start with the CMSSL routines provided by the Thinking Machine Corporation, followed by the descriptions for implementing the Ellpack and the CSR formats. We then consider the algorithm for sparse matrices with special structure using the diagonal format. Numerical experiments and floating-point performance in terms of Mflops on both CM-200 and CM-5 are presented and discussed at the end of this section.

4.1 Using the CMSSL Routines

The version 2.2 of the Connection Machine Scientific Software Library (CMSSL) provides CM Fortran routines for general sparse and block sparse matrix-vector multiplications. In the new Beta Version 3.0, routines for so-called grid sparse matrix-vector manipuation which have special applications in the finite difference and finite element schemes, have been addad. The entry format for the CMSSL routines for general sparse matrices, is a Coordinate format, with the added restriction that the nonzero elements should be stored in row order contiguously. In addition to a, ia, and ja defined earlier for the COO format, a logical array $segment(1:nnz)$ must be supplied to specify the locations at which new rows start, and another logical vector $a\_mask(1:nnz)$ to indicate that the corresponding element of a is to be treated as a non-zero element of the sparse matrix. The following routines are used to compute the matrix-vector product:

- **sparse_matvec_setup:** This routine analyzes the sparsity of the matrix and returns the communication pattern, or *trace*, required by the sparse_matvec_mult routine.
- **sparse_matvec_mult:** This routine computes the matrix-vector product.
- **deallocate_sparse_matvec_setup:** This routine deallocates the extra CM storage space that was used by the setup routine.

The sparse_matvec_setup routine must be called before calling sparse_matvec_mult. However, one can follow one call to sparse_matvec_setup with multiple calls to sparse_matvec_mult, as long as the sparse matrices involved all have an identical sparsity pattern. Thus a pseudo-code for performing a sparse-matrix vector product employing the CMSSL routines is as follows,

ALGORITHM **4.1 COO format – using CMSSL**
Sparse matrix-vector multiplication for COO format using CMSSL routines.

```
In the preprocessing phase:
   call sparse_matvec_setup (a_mask, segment, ia, ja, x, trace,.....)
In the iteration phase:
```

```
         call sparse_matvec_mult(y,a,x,ja,ia,segment,a_mask,trace,....)
In the post processing phase:
   call deallocate_sparse_matvec_setup (trace, .....)
```

Despite the appearance in the above code, the three phases do not necessarily follow each other. The preprocessing is done only once, as long as the pattern of the matrix has not changed. The postprocessing is only done at the end of the iteration phase when we haved completed the work with the same sparsity pattern. In general, the preprocessing is done at the time the matrix is constructed, since in many applicationd the patterns remains the same throughout the computation. These routines are intended for general use, and as can be expected one may be able to obtain better performances by writing routines that either exploit the special structure of the matrix or adopt more efficient storage schemes.

4.2 Using the ELL and SGP Formats

We refer to the description of the formats and the basic algorithms given in Section 3. On a vector computer, such as the CRAY YMP, the SAXPY version of Algorithm 3.8 is generally preferable to the dot-product version of Algorithm 3.7 since better performance can be obtained form vectorizing the longer inner loop. However, this is not the situation on parallel computers, since fine grain parallelism can be exploited at little cost across a large number of processors. In particular parallelism across the row dimension can be easily exploited as is done in the following CM FORTRAN code segment.

ALGORITHM **4.2 ELL format – dot product version**
Sparse matrix-vector multiplication for ELL format.

```
forall ( i=1:n, j=1:ncol ) tmp(i,j) = x(ja(i,j))
y = SUM(a*tmp, dim=2)
```

Note that in order to get better performance, we use an extra temporary 2-D array $tmp(1:n, 1:ncol)$ to hold the entries gathered from vector x so that all the multiplications and summations can be done in parallel. This algorithm can be further simplified by using the **forall** construct in CM Fortran.

ALGORITHM **4.3 ELL format – using forall and SUM**
Sparse matrix-vector multiplication for ELL format.

```
forall (i = 1:n)  y(i) = SUM(a(i,1:ncol)*x(ja(i,1:ncol)))
```

Numerical experiments show that the dot-product based Algorithms 4.2 and 4.3 can be a few times faster than the SAXPY-based Algorithm 3.8, cf. Table 10.

However, no matter which algorithm is employed, the general communication caused by the gather operation $x(ja(i,j))$ dominates the cost. If the multiplication is required to be performed repeatedly but the sparsity pattern is not changing, one might want the system to calculate the communication pattern only once, "remember" it, and use

this information for later computations. The **sparse_util_gather** routine in the CMSSL library is provided for such purpose. Unfortunately, we are unable to implement this approach on the CM-5 due to a current bug in sparse_util_gather routine in CMSSL 2.2. Although the problem has been fixed in the new version 3.0, these routines are not available on the CM-5, since they have been designed for the CM-2 architecture only.

In order to perform the gather operation, the following routine must be applied to calculate and save the communication patterns, namely the *gather trace*.

- **sparse_util_gather_setup:** This routine analyzes the sparse pattern supplied by the application, allocates the CM memory space required for the preprocessing, calculates the communication pattern, and returns the information, *gather_trace*, required by the **sparse_util_gather** routine.

After the setup, the **sparse_util_gather** routine is used to move elements from the source vector x to a temporary 2-D array *tmp*, the element-by-element product $a * tmp$ is then computed in parallel and the intrinsic function **SUM** can be used to compute the summations along each row. One can follow one call to **sparse_util_gather_setup** with multiple calls to **sparse_util_gather** if the sparsity remains constant throughout the calls. The pseudo-code can be outlined as follows.

ALGORITHM **4.4 ELL format – using gather**
Sparse matrix-vector multiplication for ELL format.

```
In the preprocessing phase:
      call sparse_util_gather_setup (ja, gather_trace, ...)
In the iteration phase:
         call sparse_util_gather (tmp, x, gather_trace, ...)
         y = SUM(a*tmp, dim=2)
In the post processing phase:
      call deallocate_gather_setup (gather_trace)
```

Note that some auxiliary storage space needs to be allocated for these library routine calls. Also the **deallocate_gather_setup** routine is recommended to deallocate the extra CM storage space required by the gather routine at end of the computation.

An alternative way of doing the gathering operation is to use the communication compiler routines in the new CMSSL version 3.0 Beta. The *communication Compiler*, which was developed by Dahl [4], is a software facility for scheduling completely general communications on the Connection Machines. It produces output data structures which are used by a message delivery system to perform synchronous processor to processor message passing. The setup routine compute the *trace* for a communication pattern just once, and then the message delivery routines use it repeatedly in subsequent operations. Similarly to the gather routine, this feature can yield significant time saving in applications that use the same communication pattern repeatedly. The following routines are employed in our implementation.

- **comm_setup:** This routine calculates a message delivery trace for the specified operation with the user specified source and destination arrays and layout. The information about the trace is stored in the front end memory.
- **comm_get:** This routine gathers selected source array elements into a destination array using the communication trace computed by the setup routine.
- **deallocate_comm_setup:** This routine deallocates the CM and front end memory that the setup routine allocated to store a trace.

There are several methods provided by the communication compiler for trace compilation. Each method involves a trade-off between compilation time and message delivery performance gain, as well as a cost in memory usage. We employ the so called *Fast Graph* method, which optimizes the use of CM-200 hypercube topology by scheduling the use of individual wires by each message. The pseudo-code for computing the matrix-vector product using communication compiler routine can then be outlined as follows.

ALGORITHM **4.5 ELL format – Using the communication compiler**
Sparse matrix-vector multiplication for ELL format.

```
In the preprocessing phase:
      get_trace = comm_setup (tmp, ja, x, ......)
In the iteration phase:
         call comm_get (tmp, get_trace, x, ......)
         y = sum(a*tmp, dim=2)
In the post=processing phase:
      call deallocate_comm_setup (get_trace)
```

The communication compiler also provides a variety of **comm_send** routines for scattering selected source array elements to a destination array. However, the performance of these send operations is currently quite poor (cf. Table 9). Instead, we use the CM Fortran primitive routine **sum** to compute the summation required in the dot product, for each row.

The ideas behind the gather routine and the communication compiler get routine are similar. The numerical experiments show that better performance can be obtained by using the communication compiler. However, the **comm_setup** routine takes more time and memory for trace compilation, cf. 3.

Two-dimensional mappings of the vectors We now assume that the vector x is expanded as a 2D array, i.e., that the components $x(j)$ are duplicated horizontally on the nc columns of the j-th row of a 2-D array $x2D(1:n, 1:nc)$, with $x2D(:, j) = x(:)$, for $j = 1, \ldots, nc$. The vector $y(1:n)$ is also expanded similarly. The following algorithms assume that this expansion is done intially and will use vectors in their 2-D representation described above throughout.

ALGORITHM **4.6 ELL format - with 2D expansion of variables**
Sparse matrix-vector multiplication for the ELL format with redundant storage.

```
forall ( i=1:n, j=1:nc ) tmp(i,j) = x(ja(i,j),j)
tmp = a*tmp
y(1:n,k) = SUM(tmp, DIM=2)
y(1:n,1:nc) = spread(y(1:n,k),DIM=2,nc)
```

Note that the results are mapped directly on one column of virtual processors, the k^{th} so that an additional spread may be needed to prepare for the next matrix by vector product in a typical algorithm. Alternatively, we can also do a SPREAD-WITH-ADD operation along the second dimension which will deliver the result on each column. As it is written the algorithm assumes that the vector x is the result of previous computation and is already duplicated in each column.

The above algorithm performs two data parallel floating point computations, namely a elementwise matrix multiplication and a sum. The SUM requires communication between neighbouring virtual processors. Nevertheless, we often have a large *vpr* and many of these virtual communications are just local moves within the memory of the same physical processor.

The idea behind the above code, is that we would like to first perform the necesseary communication between the physical processors and then the triadic operations on each virtual processor. When we use a row-wise ELL or SGP versions described above, the massively paralelel operation `SUM(a*w,DIM=2)` can be done as a triadic operation on each PN and with just order $\log_2(p)$ communication steps between the PN's (p is the number of PN's).

We can also use an SGP column-wise format to compute a sparse matrix vector multiplication. Thus, the vectors are expanded such as $x(i,:) = x1D(:), \forall\ i = 1, nc$. In this case, the sparse matrix vector multiplication can be done as follows:

ALGORITHM **4.7 SGP column-wise format - with 2-D expansion of variables**

```
tmp = a * x
forall (i = 1:nc, j=1:n) tmp(jc(i,j),ia(i,j)) = tmp(i,j)
y(k,1:n) = SUM(tmp, DIM=1)
```

The mapping of the vector is such as the data $x(i,j)$ are aligned with the data $a(i,j)$. Notice than the algorithm 4.6 generates a general *get* (gather) between virtual processors while the above algorithm uses a *send* (scatter) operation as is typically the case with column oriented algorithms. This difference can be important because on the CM-2, the general router performs a *get* by using two consecutive sends: a first in which the address of the requesting node is sent from this requesting processor to all 'sending' processors and a second in which the data is sent from these processors to the requesting node. Thus, the second algorithm will require general communications which are often faster than those of the first one.

We observe that the possibility to transpose the sparse matrix on the geometry associated with the SGP format is well-suited to the algorithm 4.7. The one-to-one general communication is similar to this operation. In the classical Ellpack-Itpack format the information stored on the array $jc(i, j)$ is not available on each processor.

4.3 Using the CSR Format

When the matrix is stored in the CSR format, the matrix-vector product can be carried out by Algorithm 3.3. If this pseudo-code is compiled as it stands on the CM, all resulting computations will be executed on the front-end host of the CM-2 or the Control Processor (CP) of the CM-5, rather than on the hundreds or thousands of available processors. We can instead employ the *gather* and *scatter* operations provided by the CMSSL library, and the *scan_add* from the CM Fortran Utility Library. The corresponding alogrithm is outlined below.

ALGORITHM **4.8 CSR format – Data Parallel Model**
Sparse matrix-vector multiplication for CSR format.

```
y = 0.0d0
call sparse_util_gather ( tmp, x, gather_trace, .....)
tmp = a*tmp
call cmf_scan_add (tmp, tmp, cmf_upward, cmf_inclusive, .....)
call sparse_util_scatter ( y, scatter_pointer, tmp,
        scatter_trace, ..... )
```

In this algorithm, the *sparse_util_gather* routine is first called to gather the corresponding entries from the vector x into a temporary array *tmp*, then the multiplications are carried out element-by-element in parallel. The *cmf_scan_add* routine from the CM Fortran Utility Library is used to perform the summation for each row, and finally the *sparse_util_scatter* routine is used to retrieve the results. Note that the *sparse_util_gather_setup* and *sparse_util_scatter_setup* routines must be called to compute the communication patterns, *gather_trace* and *scatter_trace*, before this algorithm is called.

Instead of using *gather* & *scatter* operations, we have tried an alternative way of using *comm_get* and *comm_send_add* routines, known as the communication compiler routines, supplied in the new version CMSSL 3.0 Beta. However, the performance of this alternative is very unsatisfactory due to the poor performance of *comm_send_add*, cf. Table 9.

4.4 Using DIA Format

Special matrices that have a diagonal structure, can be stored in the diagonal (DIA) format and the matrix-vector multiplication can be performed by either one of the Algorithm 3.5 or 3.6. In order to accomplish the data movement related to shifting the vector x into $x(*+offset(j))$ in both algorithms, we can employ the EOSHIFT primitive

routine under the data parallel model on the CM. The matrix-vector product $y = Ax$ can be implemented in CM Fortran as following:

ALGORITHM **4.9 DIA format – data parallel model**
Sparse matrix-vector multiplication for DIA format.

```
y = 0.0d0
do j = 1, ndiag
   y = y + a(:,j)*EOSHIFT(x, dim=1, shift=offset(j))
end do
```

In some special cases, the number of diagonals is small and known in advance and the outer loop can be avoided. For example, the matrix-vector product for a tri-diagonal matrix can be written as

```
y =  a(:,1)*EOSHIFT(x, dim=1, shift=-1) + a(:,2)*x
     + a(:,3)*EOSHIFT(x, dim=1, shift=1)
```

Here we assume that the subdiagonal is stored in the first column of a, the main diagonal in the second column, and the superdiagonal in the third column. Since the communication required in shifting the vector x is a nearest neighbor communication, we can expect high performance from the above implemention. The numerical experiments confirm this expectation.

So far, we assumed that the sparse matrix A is mapped onto a 2-D mesh of CM processors using the mapping (:news,:news) as a default layout, that is, each virtual processor or PN holds one element of A.

An alternative is to store the sparse matrix in a 2-D array $diag(1 : ndiag, 1 : n)$. Rows of a are stored in columns of $diag$, that is, $diag = a^T$. We then map each column of $diag$ onto a virtual processor or PN by using the (:serial, :news) layout. We also map the vector x and y onto the same processors where rows of $diag$ reside. In this mapping, the only communication required would be the shifting operation of x. Multiplications and summations are carried out within each processor without any cross-processor communication. Numerical experiments show the improvement in performance over the previous mapping.

4.5 Numerical Experiments

We first show the performance of some basic linear algebra computations on the CM-200 and CM-5 using the Data Parallel model. These computations are fundamental to the sparse matrix-vector multiplications and many other applications. We examine the DOTPRODUCT, SAXPY, and TRIAD operations, and then compare the CM Fortran intrinsic function SUM and the Utility Library routine SCAN_ADD. We also compare the indirect addressing, sparse gather and get functions. Experiments of matrix-vector products using different storage formats are followed.

DOTPRODUCT, SAXPY and TRIAD. The dot-product is implemented in the intrinsic function DOTPRODUCT on the Connection Machines. SAXPY can be coded in a Fortran-90 like style

$$y(1:n) = \alpha * x(1:n) + y(1:n)$$

or simply by

$$y = \alpha * x + y$$

if x and y are conformable, that is, arrays of the same shape and size. A TRIAD can be coded in a similar way by replacing the scalar α by a vector a. Table 1 shows the Mflop performance of these operations in double precision floating point numbers. It is safe to predict that higher perofrmance can be obtained by increasing the array size n. In SAXPY, the front-end scalar α will be broadcasted to all processors, and in TRIAD, each processor can access its local memory simultaneously for operands. Therefore, the overhead of memory access is much less than for shared-memory architectures and consequently there is little difference in performance between SAXPY and TRIAD. From the table, we observe that the SPARC-based CM-5 with only 512 PEs is competitive with CM-200 with 32K processors in theses two computations. One must also note that high performance (over 100 Mflops) can only be achieved with very large arrays. For less than 20 Mflops are achieved even when the vector length n is as large as 65536.

	CM-200			CM-5		
n	DOT	SAXPY	TRIAD	DOT	SAXPY	TRIAD
16384 (16K)	4.34	3.54	3.54	4.51	3.30	3.30
65536 (64K)	17.20	12.25	12.25	17.28	10.74	10.99
524288 (512K)	126.87	72.98	72.97	119.20	65.73	67.69
1048576 (1024K)	240.94	134.33	134.31	224.89	122.17	125.57
2097152 (2048K)	457.64	248.80	248.76	423.95	228.26	230.51

Table 1: Mflop performance of DOTPRODUCT, SAXPY, and TRIAD on a 32K-processor CM-200 and a 512-processor CM-5 without vector units

SUM and SCAN_ADD. It is usually necessary to compute the sum of entries accross the rows, columns, or intervals of arrays in the sparse matrix-vector multiplications. There are two primitives, SUM and SCAN_ADD, provided in the CM Fortran. They provide similar but not identical functions. In Table 2, we compare the Mflop performance of these two routines. In the 2-D cases, there is not much difference in summing accross the rows or columns. We only show the results of summing accross the rows. From the table, one may observe that SUM is about twice faster than SCAN_ADD in the 1-D case, and 60% faster in the 2-D case.

1-D ($n \times 1$)				
	CM-200		CM-5	
n	SUM	SCAN_ADD	SUM	SCAN_ADD
16384 (16K)	2.21	1.09	2.34	1.14
65536 (64K)	8.79	4.29	9.27	4.30
524288 (512K)	68.49	31.45	67.88	28.99
1048576 (1024K)	133.47	59.57	128.38	54.29
2097152 (2048K)	255.97	112.52	241.65	96.34
2-D ($n \times n$)				
	CM-200		CM-5	
n	SUM	SCAN_ADD	SUM	SCAN_ADD
1024 (1K)	95.96	58.71	94.25	53.92
2048 (2K)	321.14	208.63	302.01	170.57
4096 (4K)	1089.91	680.45	1007.76	569.02
6144 (6K)	2248.05	1371.66	2123.75	1158.34
8192 (8K)	3766.70	2282.99	3501.49	1949.31

Table 2: Mflop performance of SUM and SCAN_ADD on a 32K-processor CM-200 and a 512-processor CM-5 without vector units

Indirect Addressing. Communication costs due to indirect addressing can be quite high for sparse matrix computations on massively parallel computers, see illustration in Table 8. Since it is unavoidable to use indirect addressing in the sparse matrix context, we examine three different ways of handling the indirect addressing on the Connection Machines. A 2-D integer array of indices, ja, is generated from the 5-point finite difference scheme over a rectangular domain, We then compare the timing (in ms) for performing the following operations:

- forall(i=1:n,j=1:ncol) tmp(i,j) = x(ja(i,j))
- call sparse_util_gather (tmp, x, gather_trace, gather_trace_mask)
- call comm_get (tmp, get_trace, x, ier)

We are unable to perform sparse_util_gather and comm_get routines on the CM-5 since the CMSSL 3.0 is not available on the CM-5 at present. From Table 3, one can see that the sparse_util_gather routine is about 50% faster than the array index format, and the comm_get routine is twice faster than the sparse_util_gather routine. The time spent in the gather setup is moderate. On the other hand, the setup time for the comm_get routine is extremely high. Therefore, its use can only be justified in some specific applications. We also like to comment that the communication cost on the CM-5 is greatly improved when compared to the CM-200.

	CM-200					CM-5
$nx \times ny \times nz$	x(ja)	gather setup	gather	get setup	get	x(ja)
$128 \times 128 \times 1$	3.92	8.46	3.07	9.24×10^3	1.58	1.48
$256 \times 256 \times 1$	12.90	24.13	8.58	4.80×10^4	4.65	5.27
$512 \times 512 \times 1$	50.63	89.83	34.63	4.93×10^5	16.22	20.80
$25 \times 25 \times 25$	5.40	11.33	3.68	1.66×10^4	2.02	3.73
$32 \times 32 \times 32$	8.66	17.46	6.13	2.95×10^4	2.56	3.86
$64 \times 64 \times 64$	70.97	128.71	44.84	4.56×10^5	16.99	29.18

Table 3: Comparison of Timing(in ms) for Indirect Addressing on a 32K-processor CM-200 and a 512-processor CM-5 without vector units

Sparse Matrix-vector Product. We experimented with two types of sparse matrices to test the performance of matrix-vector products on the CM-200 and CM-5 using data paralllel (SIMD) models. We used the COO, CSR, ELL and DIA storage schemes to store the test matrices.

Problem 1. The sparse matrices arise from the elliptic partial differential equation with Dirichlet boundary condition on a rectangular domain using centered difference schemes. Both 2-D and 3-D cases are considered.

Problem 2. Random symmetric band matrix with different bandwidths. All the non-zero entries are clustered around the main diagonal.

Although both types of matrices have a special structure, we treat them as general sparse matrices when they are stored in COO, CSR or ELL format.

In Table 4 and 5, we show the performance in Mflops for Problem 1. Tables 6 and 7 are for test problem 2. If the matrix is stored in COO format, we use the matrix-vector product routines supplied in the CMSSL library. When the DIA format is used, we experiment with both row- and column-oriented mappings as was explained earlier. For the ELL format, both the gather routine and the communication compiler get routine for trace compilation are tested and compared on the CM-200. Since we are unable to use these routines on the CM-5 at present, we only compare the row- and column-oriented data mappings using plain CM-Fortran code on the CM-5.

As can be seen from the tables, the best performance is obtained with the column-oriented DIA format. Clearly, this format exploits the special structure of the matrices. There is no inter-processor communication when summing up the results of the triads in the algorithms. It can be as much as 5 times faster than the the CMSSL library routine using the very general COO format (eg., $64 \times 64 \times 64$ finite difference data on CM-200), and about twice as fast as the algorithm using a row-oriented DIA format (by comparing the columns DIA_row and DIA_col in tables $4 \sim 7$). For the band matrix with large bandwidth, e.g. 31, the ratio between the column-oriented data mapping and the row-oriented data mapping can be as much as 6.5 on the CM-200 and 8.5 on the CM-5 when $n = 1,024K$. The performance decreases dramatically when the bandwidth

increases. This is due to the increasing cost in the communication performed by EOSHIFT operations. In the special tri-diagonal cases, we code the multiplication explicitly, hence we are able to get over 340 Mflops on the CM-200. The performance on the CM-5 is about half of that on CM-200. We think this is because the lack of floating-point accelerators on the CM-5 and fewer processors; 512 processors vs. 32K processors with 1K floating-point accelerators.

When the matrix is treated as a general sparse matrix, the ELL format using the communication compiler get routine achieved better performance than the others. It is about twice faster than the gather routine, and 5 times faster than the CSR format and CMSSL library routine for the finite difference data. It is also interesting to observe that the performance of ELL format using communication compiler get routine increases when the bandwidth increases in the band matrix data, while that of ELL format using gather routine keeps steady, and that of DIA format using EOSHIFT routine decreases. However, we need to point out that the communication compiler setup routine takes much more time for trace compilation while the setup time for gather is negligible (less than one second in most cases). Moreover, the communication compiler setup routine requires a lot more memory space. We were unable to test some of the large data set due to the memory problem. On the CM-5, we compare both the row- and column-oriented ELL format. Unlike the DIA format, there is not much of differences in the performance. This is because the gneral communications involved in the indirect addressing $x(ja(i,j))$ dominates the cost. Also the performance is about the same with variant bandwidths in the band matrix data set.

In Table 8, we show the time spent in multiplications, gather operation, and summation of each row when and the matrix is stored in ELL format. In Table 9, we show some timing results for communication compiler routines when the matrix is stored in CSR format and communication compiler get and send_add routines are used.

We should point out that although the sparse matrices in both test problems have similar characteristics; namely all non-zero entries belong to just a few diagonals, better performance can be obtained when all the diagonals are clustered around the main diagonal, like those band matrices in problem 2. Similar observation has been made in [15].

4.6 Analysis of the data parallel ELL/SGP algorithms

In this section we analyze the performance on the CM-5 of the algorithms that utilize the Ellpack and the variant SGP in a data parallel programming model. We evaluate the performance of each part of these two versions. The building blocks in both routines are the following operations.

1. The massively parallel multiplication.

2. the reduction with addition along each dimensions of the two concerned geometries.

3. the global time of the sparse matrix-vector multiplication.

2-D Finite Difference						
$nx \times ny \times nz$	CMSSL	CSR	ELL_gather	ELL_get	DIA_row	DIA_col
$32 \times 32 \times 1$	0.11	2.52	4.12	4.45	1.88	8.30
$64 \times 64 \times 1$	9.29	8.99	39.57	39.81	25.26	32.49
$128 \times 128 \times 1$	18.44	17.96	46.93	67.26	50.22	70.83
$256 \times 256 \times 1$	24.20	24.33	65.07	78.13	65.24	96.77
$512 \times 512 \times 1$	26.40	26.37	65.24	73.65	69.59	105.095
3-D Finite Difference						
$25 \times 25 \times 25$	21.14	20.47	50.23	67.98	27.13	37.12
$32 \times 32 \times 32$	22.52	22.80	65.47	100.74	53.72	85.28
$64 \times 64 \times 64$	22.97	23.47	71.50	96.72	65.19	115.09
$32 \times 64 \times 64$	23.48	23.67	71.54	101.73	63.94	111.96
$8 \times 64 \times 64$	22.56	22.95	66.23	100.74	56.83	93.36
$64 \times 64 \times 8$	21.84	21.33	67.81	99.13	49.43	75.01

Table 4: Mflop performance with finite difference data on CM-200 with 32K processors

For testing purposes we will use two test matrix patterns with different values of N, nc and 3 different geometries. These three are: the standard geometry [:,:] which is the one used by default by the system, the geometry [1:,10:] which is equivalent to the serialization of the second dimension during the mapping onto physical processors, and the geometry [10:,1:] which is equivalent to the serialization of the first dimension during the mapping between the virtual data parallel model and the physical processor.

4.6.1 Data parallel element-wise multiplication

Figure 1 shows the performance of the massively parallel operation $V = V * W$, which in CM-FORTRAN denotes the element-wise data parallel multiplication of two arrays. If V and W are N by nc arrays, the resulting V is defined by $v_{ij} := v_{ij} * w_{ij}$ for $i = 1, \ldots, N; j = 1, \ldots, nc$. There is no communication between the processors, i.e., we only perform the operation $V = V * W$. Note that these performances are independent of the geometry and the matrix pattern.

We observe that when the size of the data parallel 2D array is larger than 1 Mega words, the performance decreases. This corresponds to 16K bytes on each physical processor, which is precisely the size of the SPARC cache. We will study the performance of the floating point arithmetic for the two cases, namely when the data fits in cache and when it does not.

2-D Finite Difference						
$nx \times ny \times nz$	CMSSL	CSR	ELL_row	ELL_col	DIA_row	DIA_col
$32 \times 32 \times 1$	6.22	6.22	14.06	14.33	4.12	5.05
$64 \times 64 \times 1$	19.54	19.31	37.68	38.15	13.62	17.86
$128 \times 128 \times 1$	39.49	39.33	60.38	65.55	33.23	54.25
$256 \times 256 \times 1$	35.51	23.04	66.71	71.62	46.05	86.68
$512 \times 512 \times 1$	*	59.13	67.46	69.15	48.05	97.14
3-D Finite Difference						
$25 \times 25 \times 25$	29.96	39.32	45.19	45.19	20.33	27.80
$32 \times 32 \times 32$	43.74	23.06	68.49	69.41	34.30	68.71
$64 \times 64 \times 64$	36.58	34.31	69.72	54.10	40.94	96.49
$32 \times 64 \times 64$	48.90	51.57	72.20	73.99	39.78	59.95
$8 \times 64 \times 64$	47.60	48.26	68.61	72.14	34.93	69.58
$64 \times 64 \times 8$	46.24	46.53	66.79	60.62	34.82	69.24

Table 5: Mflop performance with finite difference data on a CM-5 with 512 PNs, without vector units

4.6.2 Data parallel reduction with addition operation

Tables 11, 12 and 13 show the performances obtained for the data-parallel reduction with addition. We compute $s_k = \sum_{j=1}^{j=l} a(k,j)$ or $s_l = \sum_{j=1}^{j=k} a(j,l)$ with $l, k = N$ or $l, k = nc$ depending of the underlying geometries.

Table 11 shows the performance obtained using an N by nc geometry, with N equal to one million. We observe that the order of the results are similar for the mappings [:,:] and [10:,1:]; even if we have a ratio of 9 when $nc = 32$ along the second dimension. The variation of the performance corresponding to the reduction along the first dimension is better with the [10:,1:] mapping than the standard one but the reduction along the other dimension is worse. The variation is just amplified. The standard mapping does not map the second dimension well for the concerned computation and dimension sizes. It seems to generate much more communication between physical processor when performing the reduction along the second dimension. With a N by nc geometry, more elements of the same column of the matrix, reside on the same physical processor when using a [10:,1:] mapping, than with the standard [:,:] mapping. As the *vpr*, i.e., the ratio of the virtual to physical processors, is equal to 2K * nc, using 512 PNs, the number of communications is reduced along the first dimension but is increased along the second. The impact on the times is shown in the Table.

The reduction along the short dimension is fast when we use a [1:,10:] mapping. In this case we can map all the elements of each row of the matrix, using a N by nc geometry, on the same physical processor and there is no communication when computing the reduction with addition along this dimension.

Band Matrices						
band	n	CMSSL	ELL_gather	ELL_get	DIA_row	DIA_col
3	16384 (16K)	20.85	50.12	74.04	96.14	170.13
	65536 (64K)	49.36	105.46	148.86	137.80	281.82
	262144 (256K)	61.94	110.55	197.72	148.84	332.54
	524288 (512K)	64.39	110.55	209.13	150.50	341.06
	1048576 (1024K)	65.70	110.66	*	151.42	344.68
7	16384 (16K)	37.95	90.10	141.54	55.20	97.81
	65536 (64K)	57.16	121.65	240.46	76.41	160.39
	262144 (256K)	65.09	129.95	290.06	83.83	189.51
	524288 (512K)	66.44	130.94	300.34	85.01	194.35
	1048576 (1024K)	67.05	131.28	*	85.54	196.33
11	16384 (16K)	41.33	89.71	187.99	44.27	93.15
	65536 (64K)	57.66	117.23	288.95	59.89	159.25
	262144 (256K)	65.32	133.29	332.39	64.67	190.01
	524288 (512K)	66.58	135.92	*	65.49	195.38
	1048576 (1024K)	67.14	137.05	*	65.87	197.63
31	16384 (16K)	36.78	59.79	293.71	22.75	79.56
	65536 (64K)	50.73	90.09	371.56	28.39	149.19
	262144 (256K)	62.80	127.26	*	30.18	187.36
	524288 (512K)	65.29	136.49	*	30.47	194.81
	1048576 (1024K)	66.48	141.23	*	30.62	198.24

Table 6: Mflop performance with band matrices on CM-200 with 32K processors

The performance of the reduction along the other dimension is just a little worse than with the others mapping. It is the worst possible case for the reduction along the first dimension for this geometry. As a result we conclude that the standard by-default mapping should not be used for this prefix operation along the shorter dimension.

We note that the data does not fit in the cache but that performance is better than that shown in Figure 1. The difference may come from the fact that we do the reduction with addition and some optimizations of register usage can be done, in contrast with the case of the data parallel diadic operation.

Table 12 shows the performance obtained using an nc by N geometry. We remark again that the geometry [:,:] and [10:,1:] give similar variations. We now have higher performances along both dimensions using this geometry. We may conclude that the first dimension always yields a better mapping than the second with respect to these operations. Using a [1:,10:] mapping of the nc by N geometry, we obtain the same sort of variations than with the [10:,1:] mapping of the N by nc geometry discussed above, but it is not completely symmetric as we would expect.

Band Matrices						
band	n	CMSSL	ELL_row	ELL_col	DIA_row	DIA_col
3	16384 (16K)	37.44	54.63	54.78	39.82	70.88
	65536 (64K)	60.15	67.22	68.12	74.27	152.43
	262144 (256K)	64.20	69.73	71.10	83.94	189.15
	524288 (512K)	63.02	66.62	69.63	85.87	179.01
	1048576 (1024K)	58.52	63.81	64.29	75.30	165.80
7	16384 (16K)	40.37	72.68	72.78	26.29	52.58
	65536 (64K)	72.06	75.49	75.58	38.99	89.39
	262144 (256K)	68.63	70.81	75.43	41.71	100.08
	524288 (512K)	66.55	71.17	71.81	39.24	106.03
	1048576 (1024K)	64.22	70.85	70.11	38.60	155.84
11	16384 (16K)	61.49	76.65	76.53	20.90	51.55
	65536 (64K)	74.11	80.97	81.45	29.53	90.53
	262144 (256K)	68.48	75.06	73.62	30.79	101.30
	524288 (512K)	63.08	74.22	72.50	29.38	100.64
	1048576 (1024K)		74.05	71.33	29.07	100.29
31	16384 (16K)	56.99	79.69	79.81	11.84	50.59
	65536 (64K)	54.72	80.56	80.86	13.65	86.10
	262144 (256K)	32.35	76.85	74.47	13.04	110.98
	524288 (512K)	31.24	78.62	73.86	13.16	106.99
	1048576 (1024K)		77.63	73.94	13.20	102.35

Table 7: Mflop performance with band matrices on a CM-5 with 512 PNs without vector units

Table 13 presents the results of identical tests but with a smaller matrix, i.e. $N =$ 128 K. Now, the data fits in the cache for $nc \leq 8$, cache but the performances are not higher because the *vpr* is smaller. Then, the percentage of time spent performing the floating point operation is just a little smaller than with a larger *vpr* but the time spent communicating is constant and depends only on the number of physical processors in this case.

4.6.3 Data parallel sparse matrix vector multiplication

The two versions of sparse matrix-vector multiplications examined here are the row and column versions of the SGP algorithms 4.6 and 4.7. Note that the first version uses a **get**(gather) operation and the second uses a one-to-one **send** operation for the general communications generated.

In this section we will use two types of matrices to test the performance on the CM-5. The terminology used here is borrowed from [11, 10].

	Multiplication		Gather		SUM		Total
$nx \times ny \times nz$	ms	%	ms	%	ms	%	ms
$256 \times 256 \times 1$	0.302	3.10%	8.609	85.47%	0.019	0.19%	10.073
$32 \times 32 \times 32$	0.234	3.33%	6.140	87.63%	0.014	0.20%	7.007

Table 8: Timing (ms) Results for Finite Difference Data in ELL format using Sparse_util_gather routine on CM-200 with 32K processors

	Multiplication		GET		SEND_ADD		Total
$nx \times ny \times nz$	ms	%	ms	%	ms	%	ms
$128 \times 128 \times 1$	1.6	0.02%	1.7	0.25%	7870	99.96%	6873
$16 \times 16 \times 16$	0.6	0.005%	1.5	0.01%	12111	99.99%	12113

Table 9: Timing (ms) Results for Finite Difference Data in CSR format using Communication Compiler GET, SEND_ADD routines on CM-200 with 32K processors

C-distributed matrices. A matrix with a *C-distributed pattern* has a uniform distribution of nc diagonals across the matrix in addition to a (non zero) main diagonal. Thus, we have $a(i,j) \neq 0$ only when $j = i + k * nc$ with $k = -\lfloor \frac{i-1}{nc} \rfloor, \frac{N}{2} - \lfloor \frac{i-1}{nc} \rfloor - 1$, $\forall\ i \in [1, N]$. Therefore, for the same nc, we have the same number of non zero elements, namely, nc, along each row or column. The distance between the physical processor holding a given diagonal and that holding the main diagonal can reach values close to $\frac{N}{2}$ for standard mappings.

C-diagonal matrices. A matrix with a C-diagonal pattern has $nc - 1$ diagonals to the right of, and in addition to, the main diagonal. Thus, $a(i,j) \neq 0$ for only $j = i + k$ with $k = 0, \ldots, nc - 1$, $\forall i \in [1, N]$ with $j \leq N$. The distance from non-zero elements to the main diagonal are always $\leq nc$.

We saw in algorithms 4.6 and 4.7, described in previous sections, that communication

$nx \times ny \times nz$	dot-product form	forall & sum	saxpy form
$128 \times 128 \times 1$	62.75	61.67	20.38
$256 \times 256 \times 1$	70.79	67.95	19.17
$25 \times 25 \times 25$	42.96	44.73	9.13
$32 \times 32 \times 32$	71.39	68.70	16.20

Table 10: Comparison of Mflop Performance for Finite Difference Data in ELL format on CM-5 with 512 PNs without vector units

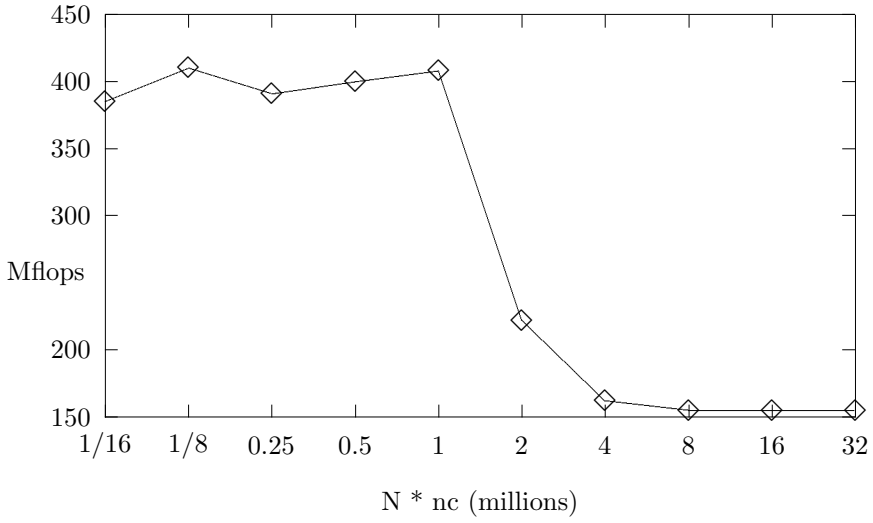

Figure 1: Performance of the element-wise matrix product operation $V = V * W$, on a 512 PN CM-5.

times depend on the distance between the processor holding the non-zero element located on a given diagonal and that holding the nonzero element located on the main diagonal. We need to send (resp. get) data from the column (resp. row) of the virtual processor j to the column (resp. row) of virtual processor i, in order to compute the result $a_{i,j-k}x_j$, where k is the offset. Let us assume that $nc \ll N$. With classical 2D geometries as described above, we can expect communication between processors that are farther apart for C-distributed matrices, with $\|i-j\|$ larger than $\frac{N}{2}$, than for C-diagonal matrices, with $\|i-j\| \leq nc, \forall(i,j)$. These results were already observed in the previous work [11, 10] for the CM-2.

Table 14 presents the performance obtained for the multiplication of a sparse matrix of size one million with nc non zero elements per row and column. We observe that the

weight	dimension	$nc = 4$ (sec., **Mflops**)	$nc = 8$ (sec., **Mflops**)	$nc = 16$ (sec., **Mflops**)	$nc = 32$ (sec., **Mflops**)
[:,:]	first	(0.016 ,**262**)	(0.034 ,**246**)	(0.081 ,**207**)	(0.162 ,**207**)
	second	(0.062 ,**67**)	(0.167 ,**50**)	(0.192 ,**87**)	(0.233 ,**144**)
[1:,10:]	first	(0.016 ,**262**)	(0.041 ,**205**)	(0.082 ,**204**)	(0.165 ,**203**)
	second	(0.023 ,**182**)	(0.034 ,**246**)	(0.054 ,**310**)	(0.096 , **350**)
[10:,1:]	first	(0.016 ,**262**)	(0.031 ,**270**)	(0.062 ,**270**)	(0.123 ,**273**)
	second	(0.158 ,**26**)	(0.364 ,**23**)	(0.764 ,**22**)	(2.085 , **16**)

Table 11: Reduction with addition along the first or the second dimension on 512 PN CM-5 without vector units. The geometry is **N by** nc, with N = 1 Mega. The data can't fitted into the cache.

weight	dimension	$nc = 4$ (sec., **Mflops**)	$nc = 8$ (sec., **Mflops**)	$nc = 16$ (sec., **Mflops**)	$nc = 32$ (sec., **Mflops**)
[:,:]	first	(0.021 ,**199**)	(0.081 ,**103**)	(0.111 ,**151**)	(0.171 ,**197**)
	second	(0.016 ,**262**)	(0.031 ,**270**)	(0.062 ,**270**)	(0.124 ,**270**)
[1:,10:]	first	(0.064 ,**65**)	(0.17 ,**48**)	(0.38 ,**45**)	(0.786 ,**43**)
	second	(0.016 ,**262**)	(0.031 ,**271**)	(0.062 ,**270**)	(0.123 , **273**)
[10:,1:]	first	(0.022 ,**190**)	(0.040 ,**210**)	(0.070 ,**240**)	(0.131 ,**256**)
	second	(0.016 ,**262**)	(0.032 ,**262**)	(0.063 ,**267**)	(0.124 , **270**)

Table 12: Reduction with addition along the first or second dimension on 512 PN CM-5 without vector units. The geometry is *nc* **by N**, with N = 1 Mega. The data can't be fitted inot the cache.

weight	dimension	$nc = 4$ (sec., **Mflops**)	$nc = 8$ (sec., **Mflops**)	$nc = 16$ (sec., **Mflops**)	$nc = 32$ (sec., **Mflops**)
[1:,10:]	first	(0.002 ,**262**)	(0.004 ,**263**)	(0.008 ,**262**)	(0.017 ,**247**)
N by nc	second	(0.003 ,**175**)	(0.004 ,**263**)	(0.006 ,**350**)	(0.012 ,**350**)
[1:,10:]	first	(0.008 ,**65**)	(0.018 ,**59**)	(0.044 ,**48**)	(0.091 ,**47**)
nc by N	second	(0.002 ,**262**)	(0.004 ,**262**)	(0.007 ,**300**)	(0.016 , **262**)
[10:,1:]	first	(0.002 ,**262**)	(0.004,**262**)	(0.008 ,**262**)	(0,016 ,**262**)
N by nc	second	(0.018 ,**30**)	(0.039 ,**27**)	(0.083,**26**)	(0.765 , **55**)
[10:,1:]	first	(0.003 ,**175**)	(0.005 ,**210**)	(0.009 ,**234**)	(0.017 ,**245**)
nc by N	second	(0.002 ,**263**)	(0.004 ,**263**)	(0.008 ,**263**)	(0.017 , **245**)

Table 13: Reduction with addition along the first or second dimension on 512 PN CM-5 without vector units. The geometry is *nc* **by N**, with N = 128 K. The data can be fitted into the cache for $nc \leq 8$ and can't be fitted into the cache when $nc>8$.

variation of performance for the C-distributed pattern matrix with respect to the mapping is poor. The speed for these matrices is small as expected. The performance for C-diagonal pattern matrices is better but smaller that for reductions and data parallel array multiplications. The best performances are obtained with a [:,:] or a [1:,10:] mapping of the N by nc geometry, as was indicated in our previous discussion of the reduction with add operation. Nevertheless, the variation of the performance with respect to nc, i.e., the vpr, are different. The general communication seems not to have the same evolution with respect to these mappings. The gap between the performances shown for the reduction with addition and these matrix vector products is certainly due to the general communications.

When we use C-diagonal pattern matrices, the general communication represents, on the average, one third of the global time. In contrast, it represents almost more than 90 per cent of the global time for C-distributed matrices.

Table 15 presents performances for the nc by N geometry. We observe again that the performance of the C-distributed case is better when we have a [:,:] or a [1:,10:] mapping

weight	matrix	$nc = 4$ (sec., **Mflops**)	$nc = 8$ (sec., **Mflops**)	$nc = 16$ (sec., **Mflops**)	$nc = 32$ (sec., **Mflops**)
[:,:]	C-distr.	(5.8 , **1.5**)	(7.1 , **2.4**)	(9.7 , **3.5**)	(24 , **2.8**)
	C-diag.	(0.171 , **49**)	(0.402 , **42**)	(0.66 , **51**)	(1.143 , **59**)
[1:,10:]	C-distr.	(4.06 , **2.1**)	(6.9 , **2.5**)	(25 , **1.4**)	(30 , **2.3**)
	C-diag.	(0.146 , **58**)	(0.358 , **47**)	(0.709 , **48**)	(1.408 , **48**)
[10:,1:]	C-distr.	(13.5 , **0.7**)	(8.02 , **2.1**)	(5.88 , **5.8**	(14.09 , **4.7**)
	C-diag.	(0.268 , **32**)	(0.581 , **29**)	(1.19 , **29**)	(3.4 , **21**)

Table 14: Sparse matrix - vector multiplication on 512 PN CM-5 without vector units; N = 1M. The geometry is **N by** *nc*. The data can't be fitted into the cache.

than otherwise.

weight	matrix	$nc = 4$ (sec., **Mflops**)	$nc = 8$ (sec., **Mflops**)	$nc = 16$ (sec., **Mflops**)	$nc = 32$ (sec., **Mflops**)
[:,:]	C-distr.	(7.4 , **1.2**)	(1.7 , **9.8**)	(2.9 , **12**)	(5.9 , **12**)
	C-diag.	(0.16 , **53**)	(0.397 , **43**)	(0.74 , **45**)	(1.5, **45**)
[1:,10:]	C-distr.	(1.55 , **5.4**)	(1.21 , **14**)	(1.37 , **25**)	(3.3 , **21**)
	C-diag.	(0.20, **42**)	(0.48 , **35**)	(1.0 , **34**)	(2.11 , **32**)
[10:,1:]	C-distr.	(7.4 , **1.2**)	(7.3 , **2.3**)	(8.7 , **3.9**	(13 , **5.1**)
	C-diag.	(0.16 , **53**)	(0.36 , **47**)	(0.73 , **46**)	(1.4 , **47**)

Table 15: Sparse matrix - vector multiplication on 512 PN CM-5 without vector units; N = 1M. The geometry is *nc* **by N**. The data can't be fitted into the cache.

Table 16 shows that we obtain higher performances when the data fits in the cache. We observed that it is not so important for the reduction but it is still crucial for the data parallel array multiplication. The performance of the matrix vector operation is difficult to analyze in depth because each of the three parts investigated have different evolutions with respect to *nc* (and then on the *vpr*), the pattern of the matrix and the mapping. When the data fits into cache and when the geometry and the pattern are well-adapted to the mapping we can reach 70 Megaflops. Nevertheless, when the data does not fit into cache the data parallel multiplication performance decreases as does the global performance. The mapping seems to be important for the performance of the reduction operation and for the C-distributed matrix vector multiplication. General communications are more difficult to analyze but they still constitute the bottleneck. There is a big gap in performance between the data parallel multiplication, the reduction with add, and the sparse matrix multiplication.

When the size of the matrix is larger than one million or smaller than 128 K, the variation of the performance can reach a ratio approximately 5 for the C-distributed pattern matrix or 3 for the C-diagonal pattern matrix, for $nc \leq 32$, see Table 17.

weight	$nc = 4$ (sec., **Mflops**)	$nc = 8$ (sec., **Mflops**)	$nc = 16$ (sec., **Mflops**)	$nc = 32$ (sec., **Mflops**)
[:,:]	(0.015 , **70**)	(0.034 , **62**)	(0.66 , **64**)	(0.17, **48**)
[1:,10:]	(0.02, **53**)	(0.043 , **49**)	(0.096 , **44**)	(0.238 , **36**)
[10:,1:]	(0.015 , **70**)	(0.030 , **70**)	(0.062 , **68**)	(0.16 , **52**)

Table 16: Sparse matrix - vector multiplication on 512 PN CM-5 without vector units; **N = 128 K**. The geometry is nc **by N**. The data can be fitted into the cache when $nc \leq 8$. The matrix is C-diagonal.

	C-distributed pattern			**C-diagonal pattern**		
N	$nc = 4$	$nc = 16$	$nc = 32$	$nc = 4$	$nc = 16$	$nc = 32$
4M	0.6	*	*	45	*	*
1M	1.5	3.5	2.8	49	51	59
128K	3.12	9.7	16	58	65	53
16K	3.2	10	17	32	65	62
2K	2.8	9.4	16.4	17	33	44

Table 17: Sparse matrix - vector multiplication on 512 PN CM-5 without vector units. The geometry is N by nc, [:,:]. The data can't be fitted into the cache when $N>128K$ and can be fitted on the cache when $N<128K$. For $N = 128K$, the data can be fitted into the cache only when $nc = 4$. Performance are in Megaflops.

4.6.4 General Remarks

The peak performance obtained for the sparse matrix vector multiplication in data parallel mode is approximately 70 Megaflops on 64 bits word arithmetic using CM-FORTRAN. We emphasize that this is obtained on a CM-5 with no vector units and no optimizing compiler. It is difficult to understand exactly the variations in performance at times. The difficulties with the cache will disappear when the vector units will be installed, since data will be stored in memory. The communication performance will not change in the new configuration. As a result, for the C-distributed pattern matrix, the performance will not increase very much because the major part of the computational time is spent during the general communications. For C-diagonal pattern matrices only one third of the global time is spent during the general communications and as a result we expect to obtain an interesting gain in speed with the vector units. However, we will not be able to exceed a three-fold improvement in achievable performance. Thus, since C-diagonal matrices yield the best performances in our study, we may extrapolate that we will not be able to exceed a performance of about 200 Mflops on a 512 CM-5 for the general sparse matrix-vector product, in data parallel mode. This is a performance that would be obtained from CM-FORTRAN, without any optimization in communication, such as communication pre-processing as is done via the use of the communication compiler on the CM-2. However, we note that precisely because these performance are obtained without communication optimization, we may achieve a good speed up over the CM-2, in situations

where the sparsity pattern changes dynamically, as is the case in many time-dependent PDE applications.

5 Experiments on the CM-5 in SPMD mode

A remarkable feature of the CM-5 is its ability to support both SIMD and MIMD modes. In this section we will discuss how to implement matrix-vector product routines in SPMD mode and briefly examine the capability of the SPARC chips which currently constitute the processing units of each node of the CM-5.

Before we present each individual matrix-vector product routine, we would like to briefly mention some unique aspects of SPMD programs. The first is the data splitting, dividing the matrix among different processors. In conventional and SIMD type of program, the programmer needn't split the data. SPMD (MIMD) program on the other hand does require the programmer explicitly handle this issue. Some of the common methods of splitting data are block-row, block-column, interleaving, etc. These type of data distribution schemes must be based on information on the matrix. There are cases where we know more about the origin of the matrices, in which cases a mapping by subdomains, i.e., a *domain decomposition approach*, may be preferable.

Tightly linked to the distribution of data is the question of communication or exchange of data among different processors. From a pure matrix operation viewpoint, the communication can be arranged according to the array indices. On the other hand, if a domain decomposition approach is used communication is dictated by the need to exchange data across the boundaries of the domains.

On the Processing Node itself, local memory layout and the flow of arithmetic operations should be carefully designed so that the architecture of the processor is best used. From this *local* viewpoint an SPMD program is very much like a sequential program.

We recall that all the matrix-vector multiplication routines are constructed as in the power iterations, i.e., we model the vector iteration $x^{(i+1)} = Ax^{(i)}$. Each iteration includes a copying process and a multiplication process ($x(1:n) = y(1:n)$, $y = Ax$), rather than just a single multiplication. This type of structure resembles the structure of most iterative linear system solvers. This copying will put these routines at a disadvantage if the MFLOP rate is the measure of the performance, since the copying of a data element will require a nonnegligible time but it is not counted as a floating point operation.

In the rest of this section, the first part is on the performance of the basic computational kernels defined in section 3.3. This part shows on the one hand the hardware capability of the CM-5, and on the other hand it also shows which of these primitives can form faster matrix-vector multiplication routines. The worst and best types of sparse matrices are presented in the two following sections. Then an in-depth look at the issues of how to map the matrix onto the processors and how to perform communications are discussed for one type of application. Finally, a short summary is presented.

5.1 Performance of the Basic Computational Kernels

As indicated in section 3.3, all the matrix-vector multiplication routines considered in this paper involve three types of basic kernel operations. There are four variations for each type of operation. An one-letter prefix and an one-letter suffix are used to distinguish these variations. The first letter in the name of the function indicates the precision of the function, the letter `S` indicates single precision (32-bit floating point arithmetic), and the letter `D` double precision (64-bit floating point arithmetic). The suffix letter `I` is used to indicates that the primitive is with indirect addressing. For example, `SDOT` is the name of the function that performs dot-product in single precision without indirect addressing; `DDOT` is the double precision dot product operation; `SDOTI` is the single precision dot product with indirect addressing; and finally `DDOTI` is the double precision dot product with indirect addressing. Similarly for the SAXPY and the TRIAD. We refer to Section 3.3 for the definitions of these operations. All of these basic primitives are invoked in one or a few of the sparse matrix-vector multiplication routines. Those without indirect addressing are invoked in the special data structures such as the banded storage scheme.

We show in figures 2, 3, 4 and 5 the speed of these functions (in Mflops) versus the array size. The speed is measured for *a single processing node* (PN) separately and not for a whole partition. Accordingly, the array size is the number of elements on each PN also. The figures are self-explanatory.

We can make the following comments.

1. The basic speeds and the number of clock cycles per iteration for not too large array sizes are listed in table 18. The clock rate of the SPARC chip used is 33MHz.

2. Among the three functions, the dot-product gives the best megaflop rate. Notice that *DOT* is much faster than *AXPY*, which in turn is only slightly faster than the *TRIAD*.

3. When the array size exceeds a certain limit, the performance of all these BLAS1 type of operations is severely reduced. We can see from the figures that the speeds of all the functions fall between the 0.5 and 1 Mflops marks, when the array size becomes larger than 8K. We list separately in table 19 the speed and the number of clock cycles per iteration corresponding to operations with these large size arrays.

According to the technical specification, the SPARC chips used on CM-5 has a peak rate of about 5 Mflops which is roughly consistent with the speeds observed here. The SPARC chip has a combined cache (used for both instruction and data) of 64K bytes. For single precision arrays, the cache can hold 2 arrays of size 8K. The program used in the test is very small, so most elements of the data array used in `SDOT` and `SAXPY` will be in cache. This can be verified in figure 2. When the arrays are too large to be stored in cache, since in each run of all three operations the arrays are not reused, most the elements of the arrays must be brought into cache at the time of use. The main memory is much slower than that of the cache, hence the severe decrease in performance.

speed (Mflops)				
addressing	direct		indirect	
precision	single	double	single	double
DOT	5.1	4.5	3.9	3.3
SAXPY	3.2	2.9	2.9	2.3
TRIAD	3.0	2.4	2.6	2.2
clock cycles per iteration				
addressing	direct		indirect	
precision	single	double	single	double
DOT	13	15	17	20
SAXPY	20	23	23	29
TRIAD	22	27	25	30

Table 18: Performances of kernel operations when arrays fit in the cache.

speed (Mflops)				
addressing	direct		indirect	
precision	single	double	single	double
DOT	1.0	1.0	1.1	1.3
SAXPY	0.8	0.9	0.6	1.1
TRIAD	0.6	0.6	0.5	0.6
clock cycles per iteration				
addressing	direct		indirect	
precision	single	double	single	double
DOT	65	65	60	53
SAXPY	84	72	103	61
TRIAD	112	114	132	110

Table 19: Performance of kernel operation when arrays do not fit in cache.

Since the write buffer is write-through, a write could take a lot more CPU cycles than a read from cache. The inner-product operation requires 2 reads per iteration (as shown in the pseudo-Fortran code), and virtually no write. The *AXPY requires 2 reads and 1 write; and the TRIAD 3 reads and 1 write. On the other hand the number of floating point operations for the same array size is the same for all these operations. This explains why the dot-product performs better than both SAXPY and the TRIAD in all cases, while the SAXPY is only slightly better than the TRIAD. From the tables of the clock cycles per iterations in table 18, it seems that the SAXPY takes about 8 clock cycles longer than DOT and the TRIAD takes 2 clocks cycles longer than the SAXPY. For the case where data is directly read from main memory, the differences are not very consistent among different variations of the functions.

We are unable to explain the consistent and important drops in performance for `DDOTI`

Figure 2: Speed versus array length. Single precision data, no indirect addressing.

in Figure 5. For array sizes of exactly 256, each iteration of `DDOTI` seems to take about 12 more clock cycles than for nearby array sizes. There are other fluctuations in the figures, but these are much smaller. Most of these fluctuations seem to occur when the total data array size is about 6K bytes.

We now go back to the issue of performance of matrix-vector products. Table 20 shows the speed (in Mflops) of the arithmetic operations used in matrix-vector subroutines with matrix in Ellpack format. Each PN has 2560 rows and 80 nonzero elements per row. The nonzeros have been randomly distributed, i.e. at given row the column indices of the nonzeros are randomly chosen. This shows the difference between using the dot-product form and the SAXPY form in an artificial but representative case.

The difference between inner-product and SAXPY (or TRIAD) forms for performing matrix-vector products, is quite consistent with the results obtained in the previous tables and figures. Because of this, the dot-product form is preferred for the current configuration of CM-5. This may change when the vector chips are installed.

	32 PNs	256 PNs	512 PNs
SDOT	77.6	208.8	364.2
SAXPY	35.4	161.9	291.2

Table 20: Speed (in Mflops) of arithmetic operations in matrix-vector subroutines.

Table 21 shows the speed in Mflops of the CMMD reduction and scan operations as provided in the CMMD library. These two functions are used here only to sum up the number contributed from each PN. As is, they only accept one number from each PN, which is not as general as the reduce and scan functions used in the data parallel programs. The reduction returns the total to either all PNs or the host. The scan puts a running tally

Figure 3: Speed versus array length. Double precision data, no indirect addressing.

Figure 4: Speed versus array length. Single precision data, with indirect addressing.

Figure 5: Speed versus array length. Double precision data, with indirect addressing.

on each PN. They may be used for other purposes. From the table, it is evident that the speed of the reduce and scan operations is proportional to the number of the processors in the partition. This indicates that the these two global communication functions take roughly a fixed amount of time for different size partitions. The performance of these two functions suggests that we should not use them directly to construct our matrix-vector product routines as in data parallel mode.

32 PNs	integer	real*4	real*8
reduce	4.4	0.77	0.78
scan	3.1	0.62	0.62
256 PNs	integer	real*4	real*8
reduce	36.8	6.3	6.3
scan	24.6	4.8	4.8

Table 21: Speed (in Mflops) of CMMD's reduction and scan operations.

5.2 Matrix-Vector Products with General Sparse Matrices

The first consideration when implementing parallel matrix-vector multiplication on distributed memory computers is to decide how to map the matrix onto the processors. In SPMD programs, the mapping must be programmed explicitly by the user. As a starting point here, we will assume the matrix has a general sparsity pattern, and no additional information about the problem structure is available. The most straight-forward way of mapping matrix under this given condition is to split the matrix according the row or

column indices. More complicated schemes may include blocking, or distributing the matrix in such a way that every processor will have the same number of non-zero elements of the matrix, etc. For simplicity we will first consider only block-row and block-column type splitting. Other schemes will be discussed in subsequent sections.

Suppose the matrix is of size $n \times n$, there are p processors. Then each PN will have $m = \lceil n/p \rceil$ rows (or columns) of the matrix. In block-row(column) scheme, each PN will have m consecutive rows (or columns). In interleaving row (column) scheme, each PN will have m rows (columns) with indices that are p apart. For example, processor i will have row (column) $i, p+i, 2p+i, \ldots$. For the block version of data splitting, each PN will also have m consecutive elements of the vector. In the case of interleaving schemes, the vector will also be distributed in the interleaving fashion, processor i will have $i, p+i, 2p+i, \ldots$th element of the vector x.

Now that we have the matrix distributed in block-row, or block-column. We may further arrange the local arrays (on each PN) so that a given algorithm will access most of its data from consecutive memory locations. The next consideration is communication. We will analyze what is needed by both block-row and block-column splitting schemes.

One point should be made clear before we continue. For the multiplication $y = Ax$, if initially x_i is on processor j, y_i is required to be generated on the same processor. This demand makes the power iteration easy to implement in the simple way described earlier. Under this restriction, when the matrix is split block-row-wise, each PN will obtain the needed elements x_i. For every a_{ij} on a given PN, if x_j is not located on this processor, it must be moved from some other processor to this PN. In case block-column splitting of the data, we can form partial summation of $y_i = \sum a_{ij}x_j$ before doing any data exchange. If x_i belongs to a given PN, all the partial summations of y_i must be passed to this processor directly or indirectly. For the sake of clarity, we will directly pass the partial sum to the desired destination, then perform the final sum there. If this is done, the block-row scheme and the block-column scheme basically require the same amount of data movement if the matrix has a symmetric sparsity pattern.[1]

To estimate the lower bound of performance for matrix-vector product routines, we now will consider what is the worst case communication requirement as outlined above. The most demanding case for communication is when every element x_i is needed by every PN in case of the block-row version, or every PN produces a partial sum for every y_i in case of the block-column version. In both cases, there is a simple way of accomplishing the required data movement, which is to use 'personalized all to all communication' (or 'total exchange' as it is sometimes called) to obtain the whole vector x or y from each processor. Table 22 show the speeds of the subroutine using this communication scheme. We know that the communication rate for this operation is about 0.87 MegaBytes/sec [16]. Assuming that arithmetic operations take basically no time, the execution time for the procedure in double-precision (8-Byte words), would be at least $8n/0.87\mu sec$, where n is the total number of rows. There are total $2 * n * nc$ number of floating point operations

[1] A matrix has a symmetric sparsity pattern when $a_{ij} \neq 0$ iff $a_{ji} \neq 0, \ \forall i, j$.

executed. Therefore the estimated speed of the procedure should be approximately

$$R_{comm} = 0.22 \times nc \ \text{ Mflops}$$

where nc is the number of nonzero per row. This gives the upper limit of the Mflop rate for the scheme using all-to-all communication with an infinitely fast arithmetic unit, which can be considered as a worst case scenario for the communication overhead.

In table 22, DIA means that the matrix used in multiplication is diagonal format (see algorithm 3.5). The CSR refers to the multiplication for general sparse matrices stored in the CSR format. Algorithm 3.3 is used in this case. ELLi and ELLs are both for matrices stored in Ellpack format, ELLi uses algorithm 3.7 and ELLs uses algorithm 3.8. All such matrices have been generated to have a random sparsity pattern. 'Total exchange' is used as the means of communication. As can be seen the above estimate for the bound on speed, R_{comm} is reasonable. In other words, the communication will dominate the execution time if no special structure of the matrix is exploited in order to reduce the communication time.

# of PNs	nc	DIA	ELLi	ELLs	CSR	R_{comm}
256	64	12.59	12.46	11.98	11.10	14.08
256	16	3.24	3.22	3.16	3.00	3.52
32	80	12.40	13.36	11.02	13.64	17.60
256	80	15.26	14.90	14.76	15.36	17.60
512	80	15.12	15.38	15.38	15.64	17.60

Table 22: Speed of general sparse matrix-vector multiplication subroutines.

5.3 Banded Matrices

Having seen the worst case, now we turn our attention to an special class of sparse matrix, namely banded matrix, which will require very little data exchange in general.

For not too large bandwidth, the communication will be limited to between nearest neighbors only, for either block-column of block-row data mappings. Suppose the total bandwidth is $2b + 1$, the upper bandwidth and the lower bandwidth are both b, a given processor i would only need to obtain b elements of a vector (either x or y) from processor $i - 1$ and $i + 1$. Assuming each processor has more than b elements of the desired array. Though the amount of data to be moved is the same for both data splitting schemes, as we will see, the block-row version is slightly favorable than the block-column version. This is because (1) the block-row version can be easily constructed to have longer dot-product loop; (2) the block-column version must use some extra memory in the process of communication. Due to these technical reasons, we will be using block-row version for the experiments. (A comparison will be made in later sections.)

The experimental results are listed in table 23. The time required for communication is generally much smaller than that needed for arithmetic operations. This is evident

from the data $(t_c + t_b)/t_a$ in the table, where the time t_a is the time spent in arithmetic operations, t_b is the time in copying the array y into x, and t_c is the communication time. In almost all the cases we tested, only nearest neighbor communication is required. Besides the low communication cost, the arithmetic operation for banded matrix is also faster than other types. As we can see from algorithm 3.9, multiplications with banded matrices do not require indirect addressing.

Other data splitting schemes, such as interleaving, are not tested here, because they require more data exchanges than simple block-row scheme.

size		32 PNs		256 PNs		512 PNs	
n	band	Mflops	t_z/t_a	Mflops	t_z/t_a	Mflops	t_z/t_a
16k	3	31.0	0.74	77.2	4.18	149.0	6.23
16k	11	67.0	0.40	115.4	2.04	441.0	2.83
16k	31	68.8	0.22	415.6	1.17	805.2	1.43
16k	51	85.4	0.12	515.0	0.90	976.8	1.04
64k	3	35.1	0.53	191.4	1.20	379.5	2.00
64k	11	60.8	0.21	441.6	0.64	832.0	0.81
64k	31	82.0	0.10	643.8	0.37	1308.5	0.47
64k	51	88.6	0.08	712.2	0.27	1457.2	0.36
128k	3	32.6	0.40	229.0	0.78	447.4	0.93
128k	11	61.9	0.19	483.9	0.44	800.0	0.44
128k	31	83.6	0.09	614.6	0.17	1194.1	0.20
128k	51	89.2	0.06	672.4	0.16	1352.0	0.18
256k	3	28.0	0.30	243.9	0.54	528.9	0.73
256k	11	58.3	0.17	444.0	0.25	1072.4	0.31
256k	31	80.3	0.09	622.1	0.13	1243.4	0.16
256k	51	87.4	0.05	670.6	0.11	1371.0	0.12

Table 23: Speed of banded matrix-vector multiplucations subroutines. Here T_a is the time spend for performing arithmetic operations and $t_z = t_c + t_b$ is the total time needed for communicating and copying the arrays.

5.4 Grid Matrices

From the two previous sections we have seen two extreme cases, on one side, the general sparse matrix, the speed of the multiplication routines is low; on the other side the faster routines can only take on special type of matrices, namely banded matrices. In this section we will concentrate matrices from finite difference discretization of PDEs on rectangular domains(see figure 6). The focus here is to try to take advantage of the special sparsity structure and the knowledge of the original physical problem.

For later discussions, we will assume the matrix has symmetric non-zero pattern. This is true for the 5-point matrix we are considering. Given this, all the communication can be

Figure 6: An example of a 2-D grid and a neighborhood of a grid point.

arranged in terms of `swap`s, which are CMMD functions that perform a send and receive at the same time. This is one of the more efficient ways to accomplish the data exchange.

First let us assume we only know the matrix has a general diagonal structure. To speed up the communication, we need to find which processor must communicate with which and save the information. This idea is generally applicable. In our experiments we implemented this with block-row data splitting. For simplicity, if one element x_i is needed, the PN that has it will send out all the elements of array x that belong to the processor. Also we used algorithm 3.6, the TRIAD form of DIA matrix-vector multiplication. In our discussions this is denoted as routine **A**.

Instead of continuing exploring other methods of splitting matrix according to the indices, we will turn to the schemes that involve more information from finite element grids. Some of the information that is obvious from observing the grid may not be easily obtained from analyzing the matrix. For example, there is only a fixed number of diagonals, and the offset of each diagonal is fixed when the size of the grid is given. We don't need to keep an offset array, and consequently the indirect addressing in the multiplication algorithm 3.5 is removed. This should speed up the arithmetic operations. More significantly, the communication may be arranged much more efficiently according to the relations shown by the grid.

Matrices arising from 5-point discretizations of elliptic problems on 2-D rectangular grids have five diagonals. If the matrix is divided in block-rows, a total of 6 possible swaps are needed for each processor. The two diagonals with offset ± 1 requires one element of x from neighboring nodes. For the two far away diagonals, the elements required may possibly be located on two separate processors, therefore four possible swap operations are needed. However, by looking at the neighborhood of a grid point (fig 6b), we can see that there are only four neighbors for each grid point, in other word besides x_i only

for other elements of x are needed in order to compute y_i. We should be able to require no more than four swaps for each PN in two dimensions, and no more than 6 in three dimensions.

We would now like to look at the communication needs of routine **A**. For 2-D case, each PN will perform possible six swaps, receive $6m$ array elements. (m is the number of variables on each processor.) And for 3-D case, a processor could possibly perform 10 swaps, receive $10m$ elements for other PNs.

One splitting that will require four swaps per PN for 2-D grid and 6 swaps for 3-D grid is the row (column) interleaving scheme. For 2-D case, each node will receive $4m$ array elements. For the 3-D case, $6m$. The advantage of this type of splitting is that the work is uniformly divided among the PNs and the communication pattern is simple. One disadvantage is that each element y_i will require exactly 4 (6) x_i's from other processors. Worse yet may be the need to consider that the processors form a ring. Because the CMMD provides mainly blocking message passing functions, it is not possible to have all the nodes paticipate in the data exchange at the same time. Suppose each PN i is to exchange a message with processor $i - j$. If the processors (say p of them) are treated as an open linear array, a simple algorithm will accomplish the task.

1. Divide the processors in to two groups. If $\lfloor i/j \rfloor$ is even, then $i \in$ even group, and $\lfloor i/j \rfloor$ is odd, $i \in$ odd group.

2. Each processor $i \in$ the even group swaps the content of the message with PN $i + j$. Each processor i in the odd group swaps with PN $i - j$.

3. Each processor $i \in$ the even group swaps with PN $i - j$. And all processors $i \in$ the odd group swap with Pn $i + j$.

4. If $i - j$ or $i + j$ out of range $(0, p - 1)$, PN i will not be sending/receiving anything.

However, in the case of a ring, $i - j < 0$ means i should be expecting some data from $p + i - j$. Above algorithm will not complete the task in one run. Two runs are needed if p/j is not a even number. A program is implemented using this scheme. We will refer to it as routine **B**.

Interleaving schemes only take advantage of the local structure of the grid points. Next we will introduce one-way dissection. We always try to let each PN have equal number of variables. In general one-way dissection does not require all the subdomain to be the same size. For 2-D case, onw-way dissection will divide a $m \times n$ grid into subdomain of size $m/p \times n$ or $m \times n/p$, where p is number of processors. For 3-D case, it will divide a domain of $l \times m \times n$ into $l/p \times m \times n$ or $l \times m/p \times n$ or $l \times m \times n/p$. In our implementation, a slightly different dissection is used. The difference is in how 3-D case is handled. In our implementation, a 3-D grid of size $l \times m \times n$ is treated as a $l \times mn$ 2-D grid. When the number of the processors is large, this will balance the load on each processor better than the straight one-way dissection method. To keep the locality in the computation, we try to number the grid point on each PN in consecutive order, or in other word we will always divide a $m \times n$ grid into $m \times n/p$ subdomain. There are still two ways of splitting the

matrix. The first is the block-row. The program implementing this will be called **C** in later discussions. The other one is block-column. It will be called routine **D**. These two routines will need 2 swaps for a $m \times n$ grid, each PN will receive $2m$ elements of x (or y). For a $l \times m \times n$ grid, possible six swaps are needed, up to $2l(1+mn/p)$ elements of x (or y) maybe received. The communication requirement shown here is slightly more than that of pure one-way dissection. Let us assume that the communication time can be modeled as $t = t_s + n/b$, where t_s is start-up time, and b is bandwidth, n is the message size. When the start-up time is very high one-way dissection is the most economical method. However, when the start-up time is small, then minimizing the total number of bytes passed would also roughly minimize communication time [12]. In the 2-D case, dividing the domain into hexagonal subdomains would normally be best in that it provides the tessalation which requires the least amount of data to be passed [12]. Since this optimal subdivision is not easy to implement or generalize, we simply use squares. We try to select two parameters q and r such that $p = q * r$ and $q : r = m : n$ for a $m \times n$ grid. Each processor will then have an $m/q \times n/r$ block of the grid. For this, one matrix-vector multiplication (in row-wise version) requires four swaps and $2(m/q+n/r)$ elements of x. In case of $l \times m \times n$ grid, we will try to find q, r, s, such that $p = q * r * s$ and $q : r : s = l : m : n$. Each PN will have a grid of $l/q \times m/r \times n/s$. Again, under row-wise splitting, six swaps are needed and each processing node will receive $2(l/q * m/r + l/q * n/s + m/r * n/s)$ elements of x. We will refer to this as routine **E**.

Table 24 shows the performance of the different routines described above. Table 25 shows the time used in communication (averaged across all the PNs). Table 26 shows the actual number of swaps (communication start-ups) for different grid size. Table 27 shows the exact number of bytes received by a typical PN in the partition. From all the tables we can see that routine **B** needs to move more data, and spends more time in communication than any other routine. Routine **E** moves the least among of data, and also spends the least amount of time in communication. Routine **C** and **D** require the same number of start-ups and move the same number of bytes, but due to some technical detail in the programs, routine **D** spend slightly more time than **C** in communication. Routine **A** spend much less time in communication than **B** even for the case where they both move same amount of data and use same number of swaps. For example when a 512 node partition is used to work on a grid of size $120 \times 60 \times 60$, for each matrix-vector multiplication, each PN does six swaps and receives 23040 bytes of data, routine **A** spends $18.5ms$ in communication versus $28.4ms$ for **B**. This is due to the fact that in the interleaving scheme (**B**) the PNs are conceptually on a ring (see previous section). However, if it is possible to pair all the PNs into send-receive pairs, then there is no difference whether the PNs are viewed as on a open line or closed one. An illustration of this can be found in table 27 for the case of 32 PNs with grid $32 \times 32 \times 32$, where routine **B** passes same size of data as **A**, and on the average the two routines spend about same amount of time in communication.

Regarding the overall performance of the routines, the one based on SAXPY (or TRIAD) is obviously slower than the one based on dot-products. Comparing routines **A** and **B**, **A** spends less time in communication, but the overall Mflops rate of **A** is clearly

lower than **B**. This is because **A** is based on algorithm 3.6, uses TRIAD rather than dot-product as in the **B**. This is consistent with what was presented in earlier sections. Comparing between **C** and **D** we may again conclude that the block-row versions of matrix-vector product routines will outperform the the block-column versions.

grid size			**A**	**B**	**C**	**D**	**E**
nx	nx	nz					
32 PNs							
128	128	1	11.5	36.4	67.7	29.8	50.7
25	25	25	11.0	16.4	38.1	22.5	36.4
30	30	30	11.4	19.5	39.9	23.5	46.4
32	32	32	14.8	44.0	50.4	27.5	53.2
512 PNs							
1024	512	1	148.0	556.6	600.1	488.8	1047.0
100	50	50	99.2	151.1	416.5	331.6	639.0
120	60	60	105.0	181.1	442.7	341.4	897.5
128	64	64	128.4	349.1	462.5	356.5	996.3

Table 24: Speed of different algorithms for some median size grid matrices.

grid size			**A**	**B**	**C**	**D**	**E**
nx	nx	nz					
32 PNs							
128	128	1	2.8	2.7	1.0	1.4	1.7
25	25	25	5.1	11.1	3.6	4.2	2.0
30	30	30	8.0	15.3	5.2	6.3	2.3
32	32	32	5.0	5.0	4.9	5.0	2.3
512 PNs							
1024	512	1	6.1	5.0	5.0	5.1	1.7
100	50	50	11.8	20.7	6.3	6.8	2.1
120	60	60	18.5	28.4	9.2	10.0	2.2
128	64	64	14.0	15.8	11.3	12.4	2.2

Table 25: Time (ms) in communication for different data distribution schemes.

5.5 Summary

In this section, we tested various algorithms for doing matrix-vector product on a massively parallel machine using SPMD program. The first type of matrix-vector product routine is for matrix with random sparsity structure, in which communication demand is high. And most of the computer time is spent in moving the desired data between

grid size			**A**	**B**	**C**	**D**	**E**
nx	nx	nz					
32 PNs							
128	128	1	2	4	2	2	4
25	25	25	4	6	6	6	6
30	30	30	4	6	6	6	6
32	32	32	2	6	2	2	6
512 PNs							
1024	512	1	2	4	2	2	4
100	50	50	4	6	4	4	6
120	60	60	6	6	6	6	6
128	64	64	4	6	4	4	6

Table 26: Number of communication start-ups required by different routines.

processors. We give an estimate of the communication time, and therefore the total time of the execution and what could be expected in terms of Mflops. On this, we would like to point out that it is not necessary to pass the partial sum of y_i to the final destination then perform the final summation. It is conceivable that one may perform the summation as the data is being passed as is done in the data parallel mode. However, there are still uncertainties regarding this possibility and we have not implemented this option.

High Mflops rates can be reached for banded matrices, because of the low communication to arithmetic ratio achieved in this case. When a matrix in nearly banded, or can be put in banded form without padding with too many zeros this can constitute a good alternative to general sparse matrices.

For a class of real applications, finite discretization of PDE on regular grid, we presented various different ways of taking advantage of the special matrix structure and domain structure. In our tests, we have shown that dissecting the matrix based on the domain structure makes the handling of communication much easier, and can significantly reduce the communication cost. Even though interleaving is easy to implement and can enhance load balancing among the processors, it was not a very efficient method for our test problems because it involves the longest subdomain interfaces among all possible splittings. Standard one-way dissections and other domain decomposition approaches work much better on regular grids. For some medium size problems on a 512 node partition of the CM-5, GFLOPs rates can be achieved for real application even without the vector units.

grid size			**A**	**B**	**C**	**D**	**E**
nx	nx	nz					
32 PNs							
128	128	1	4096	4096	1024	1024	384
25	25	25	8000	12000	4200	4200	1848
30	30	30	13920	20880	7200	7200	2432
32	32	32	8192	8192	8192	8192	2560
512 PNs							
1024	512	1	8196	16384	8192	8192	512
100	50	50	8000	12000	4800	4800	1848
120	60	60	23040	23040	8640	8640	2432
128	64	64	16384	24576	9216	9216	2560

Table 27: Message size (in bytes) actually passed by each routine.

grid size			32 PNs		256 PNs		512 PNs	
nx	ny	nz	Mflops	t(ms)	Mflops	t(ms)	Mflops	t(ms)
32	32	1	8.5	1.20	9.4	1.09	9.5	1.08
128	128	1	60.2	2.72	122.4	1.34	136.2	1.20
512	512	1	71.9	36.5	531.6	4.93	943.0	2.78
16	32	32	45.6	5.03	111.5	2.06	223.6	1.03
8	64	64	53.7	8.54	191.8	2.39	365.8	1.26
128	64	64	72.4	101.4	493.7	15.6	224.8	20.4
128	128	128	77.0	381.2	571.1	51.4	986.4	29.8

Table 28: Performance of variant (**E**) for grid matrices.

grid size			32 PNs			256 PNs			512 PNs		
nx	ny	nz	t_a	t_b	t_c	t_a	t_b	t_c	t_a	t_b	t_c
32	32	1	9	2	89	3	2	95	4	2	94
128	128	1	42	10	48	15	3	82	9	2	89
512	512	1	81	12	7	55	12	33	42	9	49
16	32	32	54	8	38	21	3	76	21	3	76
8	64	64	65	9	26	27	4	69	21	3	76
128	64	64	89	5	6	77	6	17	64	6	30
128	128	128	92	5	3	86	5	9	78	9	13

Table 29: Distribution of the execution times for version (**E**) with respect to arithmetic operations t_a, local array copying t_b, and communication t_c.

6 Conclusion

In this section we summarize our main results and draw some tentative conclusions. Our first overall conclusion regarding sparse matrix computations on massively parallel computers is that reasonable performance can be obtained with average effort, i.e., from standard FORTRAN augmented with the usual communication primitives. Using either CM FORTRAN or CMMD FORTRAN we can achieve a good fraction of the peak performance for certain matrices. However, one notable difference with dense computations, is that performance depends critically on the type of matrices considered. In sparse computations, it is important to provide the possibility of taking advantage of the structure of the matrix if one wishes not to sacrifice a great deal in performance. Thus, there can be a big difference between the performance that can be obtained from a matrix stored in diagonal format and one that is stored in the more general CSR format. Generally speaking, we also observed that in data parallel mode it is important to take advantage of two-dimensional mappings of the data in conjunction with the variants of the ELLPACK format. The CMSSL scatter and gather routines allow to enhance performance substantially. In SPMD mode, there is currently no means of exploiting knowledge on the communication pattern to speed-up later matrix – vector produtcs via saved traces. This is because of the dynamic nature of the current implementations of the communication routines, although improvements are possible. On the other hand a rather important feature of the communication provided on the CM-5 is the near insensitivity of communication speed relative to the logical distance between processors. In other words, the network tends to provide a good approximation to a fully connected *universal* computer. There is certainly a price to pay for this capability, partly in terms of hardware complexity, partly because it may be achieved at the expense of limiting certain achievable communication speeds for special cases.

Regarding the comparison between the two modes, it is clear that the SPMD mode has a slight edge in terms of achievable performance, mainly because one can better take advantage of communication patterns. On the other hand, programming is much easier in data parallel mode.

The experiments in this paper were made on a preliminary version of the CM-5 which is not equipped with vector chips [2]. It is interesting to notice that the timings were in many cases dominated by communication and as a result, we anticipate that the vector units will make essentially no difference in those cases. It is a rule of thumb that for sparse computations performance can often virtually be stated in terms of communication speeds alone [7]. With communication latencies generally high and difficult to reduce to acceptable levels, this problem is a challange to algorithm designers and manufacturers alike.

We would like to discuss the possible improvements that can be made to bring the communication times down to much lower levels. First, it is feasible to combine messages to exploit the fact that the best comminucation speed is obtained for messages whose length is a multiple of 32 Bytes, or four 64-bit foating point numbers. Some of these

[2] As of the time of this writing the vector units are planned to be installed by the end of August 1992.

optimisations are possible on the CM-5 [5]. Another possible improvement [5], may come from exploiting the fact that even though communication in the higher levels of the data network can be viewed as indeterministic, they can be deterministic in the first level. As a result optimization tools similar to the communication compiler may be developed for this limited but important level.

References

[1] *Connection Machine Model CM-2 Technical Summary*, Thinking Machine Corporation, Cambridge, Massachusetts.

[2] *The Connection Machine CM-5 Technical Summary*, Thinking Machine Corporation, Cambridge, Massachusetts.

[3] *CM-Fortran User's Guide.* Thinking Machine Corporation, Cambridge, Massachusetts.

[4] E. Denning Dahl, *Mapping and Compiled Communication on the Connection Machine System*, Proc. Fifth Distributed Memory Computing Conference, Charleston, SC, 1990.

[5] E. Denning Dahl, *Personal communication, 1992.*

[6] D. S. Dodson, R. G. Grimes, and J. G. Lewis. Sparse extensions to the Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 17(2):253–263, June 1991.

[7] K. Gallivan, W. Jalby, A. Malony, and H. Wijshoff. Performance prediction for parallel numerical algorithms. *International Journal of High Speed Computing*, 3:31–62, 1991.

[8] S. Hammond and R. Schreiber, *Mapping Unstructured Grid problems to the Connection Machine*, Tech. Report 90.22, RIACS, 90.

[9] M. Misra and P. Kumar, *Efficient VLSI implementation of iterative solutions to sparse linear systems*, Tech. Report 246, Institute for Robotics and Intelligent Systems, University of Southerm California, 1988.

[10] S. Petiton, *Massively Parallel Sparse Matrix Computation for Iterative Methods*, Tech. Report YALE/DCS/878, Department of Computer Science, Yale University, 1991.

[11] S. Petiton and C. Weill-Duflos, *Very sparse preconditioned conjugate gradient on massively parallel architectures*, in Proceeding of 13th World Congress on Computation and Applied Mathematics, 1991.

[12] D.A. Reed and R.M. Fujimoto, *Multicomputer Networks, Message-Based Parallel Processing* section 7.2, The MIT Press, 1987.

[13] Y. Saad, SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[14] Y. Saad and H. Wijshoff. A benchmark package for sparse matrix computations. In J. Lenfant and D. De groot, editors, *Proceedings of ICS conference 1988, St Malo, France*, pages 500–509. ACM, 1988.

[15] Joel Saltz, Serge Petiton, Harry Berrryman, and Adam Rifkin, *Performance Effects of Irregular Communication Patterns on Massively Parallel Multiprocessors*, 1 Journal of Parallel and Distributed Computing, 13 (1991), pp. 202–212.

[16] K. Wu and Y. Saad, Experiments with the CM-5 message passing primitives, AHPCRC internal report, 1992.