SANDIA REPORT

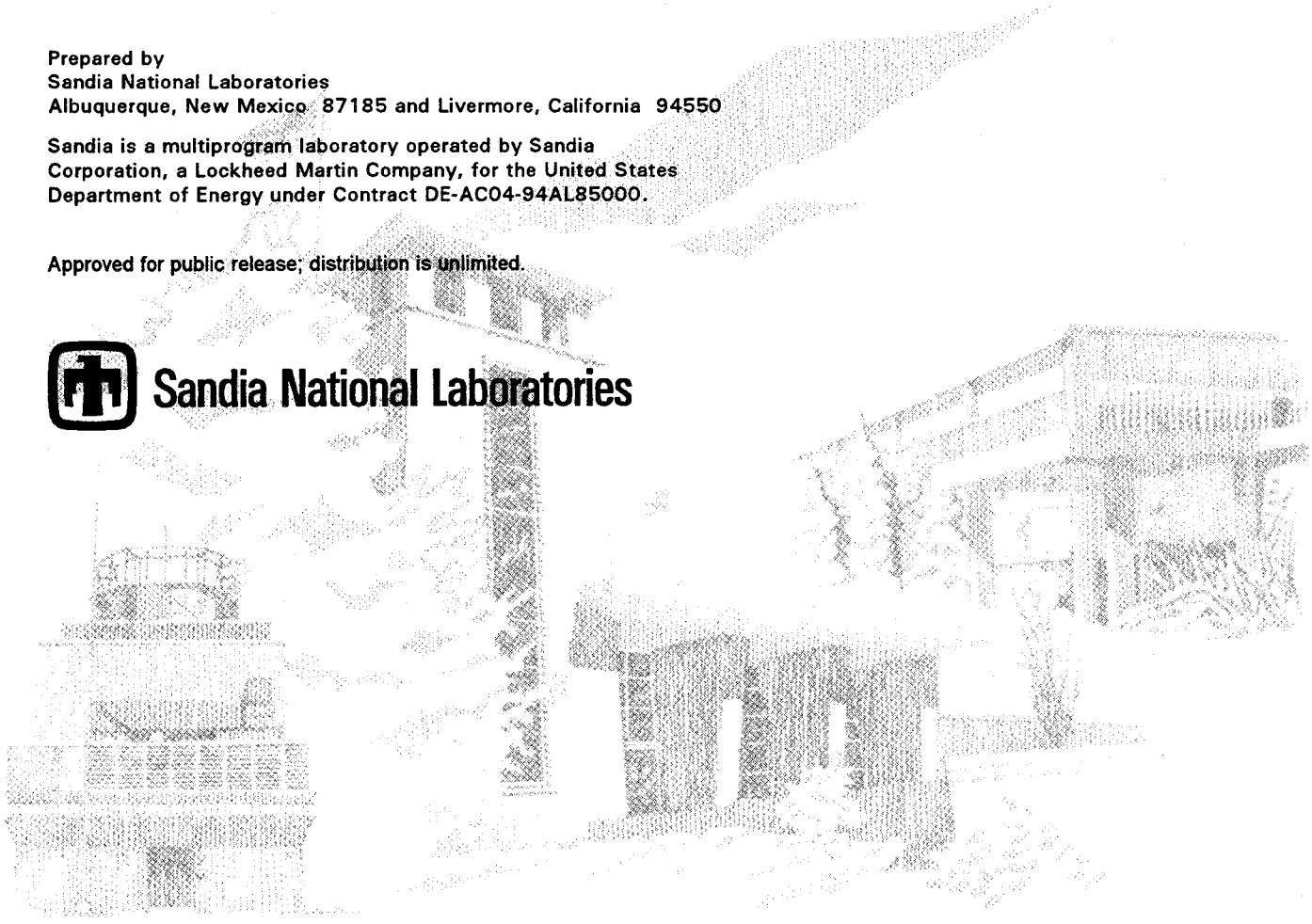
SAND96–2672 • UC–405
Unlimited Release
Printed April 1997

Massively Parallel I/O: Building an Infrastructure for Parallel Computing

David E. Womble, David S. Greenberg

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.



Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

SAND96–2672
Unlimited Release
Printed April 1997

Distribution
Category UC–405

Massively Parallel I/O: Building an Infrastructure for Parallel Computing [1]

David E. Womble
Applied and Numerical Mathematics Department
Sandia National Laboratories
Albuquerque, NM 87185

David S. Greenberg
Algorithms and Discrete Mathematics Department
Applied and Numerical Mathematics Department
Sandia National Laboratories
Albuquerque, NM 87185

Abstract

The solution of Grand Challenge Problems will require computations that are too large to fit in the memories of even the largest machines. Inevitably, new designs of I/O systems will be necessary to support them. This report describes our work in investigating I/O subsystems for massively parallel computers. Specifically, we investigated out–of–core algorithms for common scientific calculations present several theoretical results. We also describe several approaches to parallel I/O, including partitioned secondary storage and choreographed I/O, and the implications of each to massively parallel computing.

[1]This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000 and was performed at Sandia National Laboratories under the Laboratory Research and Development Program

1. Introduction

The solution of Grand Challenge Problems will require computations that are too large to fit in the memories of even the largest machines available today. The speed of individual procesors is growing too fast to be matched economically with increased memory size. Successful high performance programs will have to be designed to run in the presence of a memory hierarchy. Great efforts have already been made to optimize computations for the fastest end of the hierarchy, i.e., high speed registers and caches. The result has been the creation of optimized codes such as those in the BLAS[7]. At least as large an effort must be made to address the slow end of the hierarchy.

Traditionally the maintenance of the slow end of the memory hierarchy has been lumped under the category of I/O and left to the control of the operating system (OS). The result has been general purpose memory management routines that work fairly well in multitasking, workstation environments, but which are not appropriate in the massively parallel (MP) computing environment.

In this report, we will argue that more explicit management of disk I/O is necessary for high performance, although we expect that much of the management will eventually be packaged in libraries with system support. As an example, we will consider the LU factorization algorithm for solving dense linear systems. We will present a theoretical framework for explicitly managing I/O in the LU factorization algorithm and an implementation and results on the nCUBE 2 MP computer. We will then compare these results with OS managed I/O.

We will present two different approaches to explicitly managed disk I/O. The first is partitioned secondary storage (PSS), in which each node in an MP computer maintains an independent memory hierarchy, including disk. The second is choreographed I/O (CIO), in which data is striped across disks in a parallel file system and groups of processors issue synchronized calls to the file system.

The purpose of this report is to describe our experiences in producing high performance codes that act on data sets that are too large to fit in core memory. Some of the lessons learned are algorithmic, i.e., ways to structure the code to reduce the I/O bottleneck. Others are systems related, i.e., features of the OS that can make producing high performance codes easier.

Throughout the paper, LU factorization is used as an example. It is a common kernel in many scientific applications, such as boundary element methods and electromagnetic scattering. The I/O required by LU factorization is structured and it should be possible for an I/O subsytem to achieve most of its practical peak bandwidth. It is thus a good first test of I/O algorithms and hardware, although it is not representative of all I/O required by scientific applications.

In Section 2, we describe the characteristics of I/O in scientific computing. In Section 3, we describe partitioned secondary storage. In Section 4, we describe the LU factorization algorithm and give theoretical upper and lower bounds on I/O required by the algorithm. We also describe a practical implementation using the concept of PSS on the nCUBE 2 and give performance results. In Section 5, we describe choreographed I/O, and in Section 6, we describe an experimental file system for the Intel Paragon that implements CIO. We

summarize the report in Section 7.

2. Characteristics of I/O

The I/O requirements of scientific applications logically divide into several classes: initial input of data, access to static data/databases, output of results, and maintenance of temporary memory.

The initial transfer of data into the machine can be via a high-speed network connection such as a HiPPI or ATM connection, by a directly connected disk, or from another computer. It is a one-time operation. Usually, hardware speed is the limiting factor. If the inital loading of data is done improperly, it can add significant overhead to a computation, but it is rarely in insurmountable bottleneck.

However, a mismatch between the format in which the data arrives and the format in which the program desires the data can be both a major programming inconvenience and can cause major inefficiencies in later stages if not corrected. The ability to convert data from one format to another is one of the major uses of the choreographed I/O approach described in Section 5. Of course it would be preferable if no conversion were necessary. However, the composing of individual programs or subroutines to create an application will, inevitably, require conversions since different tasks have different natural formats.

Another important issue for the inputting of data is efficiently spreading out data which will be accessed by many nodes. For example, having each node in a large parallel machine read the same data from the same file is much more expensive than having one node read the file and then broadcast the data to all the other nodes.

The question of how to share data which will be used by many nodes does not just occur during input. Many applications access large tables which contain properties of objects in the application (such as material properties or velocities within a section of siemic terrain). It may not be clear how to distributed this information at the beginning of a program or the best distribution may change over time. Furthermore, for different numbers of processors the preferred layout may differ. For example it may be desirable to have one complete copy of the table per some number of processors rather than one copy over all processors or one copy in each processor.

The output of results is similar in nature to the input of data. A conversion of format may be necessary but each item is written just once. Typically, an acceptable method of streaming the output can be found. In the case of repeated output of intermediate results (e.g., graphical data in a physical simulation) the output requirements are static.

The handling of temporary values is much more problematic. Temporary values must be both written and read. The order in which they are accessed can change over time. In addition, the critical path of the computation will rely on data being present in fast memory. If the management of temporary memory is not efficient, it can slow the whole computation to a crawl. Demand-paging virtual memory is currently the most popular method of maintaining temporary storage (and is provided by many MP vendors). While virtual memory simplifies the programming, no virtual memory system can perform as well as a code written by a programmer who understands the algorithm being implemented. In fact, the overhead of a virtual memory system often defeats the advantage of using a parallel

supercomputer, i.e., computational speed.

In the next sections we describe two paradigms for allowing the programmer (or library routine) more explicit control over the management of temporary external storage. These paradigms are designed to yield as natural a programming model as possible while at the same time allowing for highly efficient implementations.

3. Partitioned secondary storage

If our goal is to have high performance on large number of processors, we cannot pay the overhead for general purpose I/O service. Instead, we must understand the characteristics of our problems and tailor the I/O system to our problems. One approach to this is partitioned secondary storage (PSS).

For the large scientific codes written at Sandia, it has become apparent that the overhead of shared memory emulation is often large. A message passing paradigm is preferred because of its higher performance. The key aspect of this paradigm is that the programmer explicitly arranges for data which is used locally to be stored in local memory. PSS maintains the view of local storage: each processor has its own logical disk. The data on a processor's disk will be treated similarly to the data in its local memory, and the processor will have sole control of this data. Any sharing with other processors will be through explicit message passing.

PSS allows the application to control data locality. The programmer always knows where the data is and can therefore reliably plan the overlap of computation, message passing, and I/O. This control of the data meshes well with the message passing paradigm. If, as is often the case, the program has been parallelized by creating processes that work mostly on local data, we do not want the I/O system to destroy the locality in the search for general parallelism. Using PSS, the program can still be divided up so that the compute work is evenly balanced among the processors and so that the data can be reused as often as possible. This data reuse/locality is critical for good performance. PSS removes the impact of limits on local memory size by allowing the computation to be decomposed so that each process can be designed as if it had access to a large memory without destroying locality.

We remark here that the programming required to make effective use of PSS is more complicated than that required for shared memory emulation or for a virtual memory system. However, because PSS strictly adheres to the distributed memory paradigm, we expect that anyone programming a distributed memory machine using explicit message passing will be able to use PSS easily and effectively.

We also remark that the requirement that each processor has its own logical disk does not necessitate that each processor has its own physical disk. The use of virtual disks may not match the performance of separate physical disks. However, good performance should still be achieved because the demands on the operating system are minimal: it need only interleave standard file system requests to the virtual device.

There are several alternatives to the PSS paradigm, each with its own advantages and areas of applicability. One alternative is the shared parallel file system (PFS), which is in common use. This is typically a higher level approach than PSS and consequently requires more bookkeeping on the part of the OS. One advantage is that the format of the data on the disks is transparent to the programmer, so the programmer need not spend time tuning it.

A second advantage is that data may be shared among processors through the file system. The disadvantages of this paradigm are a reduction in performance due to overhead, the inability of the file system to optimize data placement based on future access patterns of the code, and access conficts if data is to be shared through the file system.

Another alternative is choreographed I/O. This is similar to the PFS with the addition that the file system provides synchronizing routines that allow the processors to control the placement of data on the disks during a write operation and the distribution of data during a read. An experimental file system incorporating choreographed I/O will be describe later in this report.

4. LU factorization

The solution of dense linear systems of equations is a critical kernel in many scientific applications, including boundary elements methods for partial differential equations and electromagnetic scattering. As such, it provides a good test case for out–of–core paradigms. Here we develop an LU factorization routine based on the PSS paradigm.

LU factorization is one of the most effective algorithms for the solution of dense linear sytems. In LU factorization, a matrix A is decomposed into the product of a lower triangular matrix L and an upper triangular matrix U. (If pivoting is required, L is logically lower triangular.) The basic LU factorization algorithm $[L, U] = \mathrm{LU}(A)$ from [5] is given below. The matrices A, L and U are divided into submatrices denoted by subscripts (e.g., $A_{1,1}$ denotes the upper left submatrix of A).

$$
\begin{aligned}
(1) \qquad [L_{1,1},\, U_{1,1}] &= \mathrm{LU}\,(\mathrm{A}_{1,1}) \\
U_{1,2} &= L_{1,1}^{-1} A_{1,2} \\
L_{1,2} &= 0 \\
U_{2,1} &= 0 \\
L_{2,1} &= A_{2,1} U_{1,1}^{-1} \\
[L_{2,2},\, U_{2,2}] &= \mathrm{LU}\,(\mathrm{A}_{2,2} - \mathrm{L}_{2,1}\mathrm{U}_{1,2})\,.
\end{aligned}
$$

Once the matrix is factored, the associated linear system can be solved with a forward substitution and a backward substitution using the matrices L and U respectively.

4.1. I/O complexity of LU factorization

Theoretical results on the I/O complexity of an algorithm can provide a guide in developing out-of-core algorithms; although, as we will demonstrate later, "good" out–of–core algorithms do not necessarily achieve optimal I/O complexity. Here we derive upper and lower bounds on I/O for optimal LU factorization algorithms.

There has been quite a bit of work on the I/O complexity of several algorithms, including permutation, sorting, FFT's and matrix–matrix multiplication[3, 2, 6, 8]. We can derive an upper bound for LU factorization (with or without pivoting) using the same techniques used to derive an upper bound for matrix–matrix multiplication in [8]. Specifically, we will develop (or quote) bounds for the following:

$T_{MM}(n)$ the I/O complexity of multiplying two $n \times n$ matrices,
$T_{TS}(n)$ the I/O complexity of computing $L^{-1}A$ (or equivalently the I/O complexity of computing AU^{-1}),
$T_{LU}(n)$ the I/O complexity of computing LU(A).

where A is an $n \times n$ matrix, L is an $n \times n$ lower triangular matrix and U is an $n \times n$ upper triangular matrix. All matrices are assume to be in either row-major or column-major order. The I/O complexities above will be bounded in terms of the following variables.

n size of the matrix to be factored,
M size of memory,
B total size of one I/O request (across all processors).

LEMMA 4.1 (VITTER, SHRIVER). *$T_{MM}(n) < C\frac{n^3}{B\sqrt{M}}$ for some constant C.*

Proof See [8] □

LEMMA 4.2. *$T_{TS}(n) < C\frac{n^3}{B\sqrt{M}}$, for some constant C.*

Proof Without loss of generality, we may assume that the triangular system under consideration is lower triangular. We divide the matrices L and A into four submatrices denoted by subscripts and write

$$\begin{aligned} L^{-1}A &= \begin{pmatrix} L_{1,1}^{-1} & 0 \\ -L_{2,2}^{-1}L_{2,1}L_{1,1}^{-1} & L_{2,2}^{-1} \end{pmatrix} \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \\ &= \begin{pmatrix} L_{1,1}^{-1}A_{1,1} & L_{1,1}^{-1}A_{1,2} \\ -L_{2,2}^{-1}L_{2,1}L_{1,1}^{-1}A_{1,1} + L_{2,2}^{-1}A_{2,1} & -L_{2,2}^{-1}L_{2,1}L_{1,1}^{-1}A_{1,2} + L_{2,2}^{-1}A_{2,2} \end{pmatrix}. \end{aligned}$$

The above calculations show that

$$\begin{aligned} T_{TS}(n) &< 6T_{TS}(n/2) + 2T_{MM}n/2 + C_1\frac{n^2}{B} \\ &< 6T_{TS}(n/2) + \frac{2}{8}C_2\frac{n^3}{B\sqrt{M}} + C_1\frac{n^2}{B}, \end{aligned}$$

where C_1 and C_2 are constants, and C_1n^2/B is a bound on the time to permute the submatrices into row-major or column-major order [8]. We recursively subdivide the problem until matrices fit into memory, that is, we subdivide the problem k times, where $k = \log_2(n^2/M)$. This yields $T_{TS}(n) < C\frac{n^3}{B\sqrt{M}}$. □

LEMMA 4.3. *$T_{LU}(n) < C\frac{n^3}{B\sqrt{M}}$, for some constant C.*

Proof We see from Eq. (1) that

$$T_{LU}(n) < 2T_{LU}(n/2) + 2T_{TS}(n/2) + T_{MM}(n/2) + C_1\frac{n^2}{B},$$

where C_1 is a constant, and C_1n^2/B is a bound on the time to permute the submatrices into row-major or column-major order. As in the proof of Lemma 2, we recursively subdivide the problem until it fits in memory and the result follows.□

Lemma (4.3) gives us an upper bound on the I/O complexity of LU factorization. We continue by developing a lower bound.

LEMMA 4.4. *[Hong, Kung] $T_{MM}(n) > C\frac{n^3}{B\sqrt{M}}$. for some constant C.*

Proof See [6].

LEMMA 4.5. *$T_{LU}(n) > C\frac{n^3}{B\sqrt{M}}$, for some constant C.*

Proof We let A and B be $n \times n$ matrices and assume that AB is nonsingular. We also let L_A, L_B, and L_{AB} be the lower triangular factors of A, B and AB respectively, U_A, U_B and U_{AB} be the lower and upper triangular factors of A, B and AB, and I be the $n \times n$ identity matrix. We now suppose that there exists some algorithm with I/O complexity less than $C\,n^3/B\sqrt{M}$ for some constant C, use this algorithm to compute

$$\mathrm{LU}\begin{pmatrix} I & B \\ -A & 0 \end{pmatrix} = \left[\begin{pmatrix} I & 0 \\ -A & L_{AB} \end{pmatrix}, \begin{pmatrix} I & B \\ 0 & U_{AB} \end{pmatrix}\right]$$

from which we extract L_{AB} and U_{AB}. We further use this algorithm to compute

$$\mathrm{LU}\begin{pmatrix} L_{AB} & I \\ I & 0 \end{pmatrix} = \left[\begin{pmatrix} L_{AB} & 0 \\ I & L_{AB}^{-1} \end{pmatrix}, \begin{pmatrix} I & L_{AB}^{-1} \\ 0 & -I \end{pmatrix}\right]$$

from which we extract L_{AB}^{-1}. Finally,we use this algorithm to compute

$$\mathrm{LU}\begin{pmatrix} L_{AB}^{-1} & U_{AB} \\ I & 0 \end{pmatrix} = \left[\begin{pmatrix} L_{AB}^{-1} & 0 \\ I & L_{AB} \end{pmatrix}, \begin{pmatrix} I & L_{AB}U_{AB} \\ 0 & -U_{AB} \end{pmatrix}\right].$$

Because $AB = L_{AB}U_{AB}$, we have computed AB with less than $C\frac{n^3}{B\sqrt{M}}$ I/O operations. This contradicts Lemma 4.4 giving us the result.□

We summarize Lemmas 4.3 and 4.5 in the following theorem.

THEOREM 4.6. *LU factorization has I/O complexity $\Theta\left(\frac{n^3}{B\sqrt{M}}\right)$.*

Pivoting is a critical capability of any robust LU factorization code. While this presents a practical difficulty for the programmer, it does not affect the bounds presented above. Specifically, most algorithms implement column (row) partial pivoting, in which each column (row) must be searched for a pivot entry once during the factorization. Searching column i, $i = 1, \ldots, n$ requires the $n - i + 1$ entries below the diagonal. This requires at most $O(n^2)$ I/O operations for the search and another $O(n^2)$ operations if rows must be exchanged. It is clear that the bounds in Lemmas 3 and 5 are not affected.

4.2. Practical out–of–core LU factorization

The I/O complexity given in Theorem 4.6 is achieved by a recursive algorithm. However, this algorithm can be difficult to implement, and it is complicated to include pivoting, which is necessary for numerical stability. The more practical algorithm presented in this section does not achieve the optimal I/O complexity. However, because in our experience even the non-optimal amount of I/O can be almost completely overlapped with computations for problems of interest, it is not necessary to use an optimal I/O algorithm.

We begin by dividing the matrix A into b column blocks of size $n \times k$ where nk matrix elements fit in the size M matrix. We denote these blocks as A_i, $i = 1 \ldots b$. We denote the corresponding components of L and U by L_i and U_i. LU factorization can now be written as follows

L_i has approximately $nk - (i - 1/2)k^2$ entries and must be read $b - i$ times. Summing this for $i = 1 \ldots b$, we see that $O(n^4/M)$ entries must be read. Thus the I/O complexity

```
Begin (out-of-core LU factorization)
    For i = 1, ..., n
        Read A_i
        If i > 1, then
            For j = 1, ..., n
                Read L_j
                Update A_i with L_j
            End for
        End if
        Factor A_i to produce L_i and U_i
        Write L_i and U_i
    End for
End
```

FIG. 1. *Parallel LU factorization for processor q.*

of this algorithm is $O(n^4/MB)$. This differs from the theoretical upper bound by a factor of $n/\sqrt{M}$, which may be significant. However, the effect of additional I/O is mitigated in practice by overlaping the I/O with computation. In particular, we note that the read of L_{j+1} can be accomplished while L_j is being used in computations. This reduces the "visible" I/O to $O(n^2/B)$.

This algorithm has been implemented using the PSS constructs described in Section 3. The test machine for this implementation was the nCUBE 2 at Sandia National Laboratories. This machine has 1,024 nodes, each with a processor capable of 2.1 double precision Mflops/second (achievable using the BLAS library) and 4 MBytes of memory per node. The disk system consists of 16 one GByte disks, each with its own SCSI controller. The operating system used for these runs was the SUNMOS operating system developed at Sandia National Laboratories. Ideally, we would present experiments that make use of the entire capacity of the machine. Unfortunately, factoring a matrix of this size requires several hours of compute time. Instead, we present two medium–size runs to demonstrate the ability to overlap I/O with computation and several small–size runs to highlight the dependency of the performance on the available memory and the number of processors used.

# of col. blocks	col–block size (Gbytes/proc)	memory used	total I/O (Gbytes)	total time (sec)	total I/O time (sec)
14	.89	94%	4.99	7,226	85
27	.46	49%	8.61	10,773	87

TABLE 1
Scaling of $10,000 \times 10,000$ *matrices*

Table 1 shows the results of running our LU factorization algorithm for a $10,000 \times 10,000$ double-precision matrix (8 bytes per entry) on 64 processors varying the amount of memory

available to the algorithm. In the first run, each block of columns could be made large enough to cover the matrix with 14 blocks, while in the second run, only half of the memory was used, doubling the number of blocks necessary. The increase in the number of blocks almost doubled the amount of I/O done and significantly increased the total run time. The last column records the amount of time spent doing I/O that could not be overlapped with computation, which we note is almost constant as predicted above.

The increase in total time in Table 1 is almost entirely due to increased interprocessor communication. The amount of communication required is $O(n^2 b\sqrt{p})$, where b is the number of column blocks and p is the number of processors. Thus, the memory size is important because is defines the grain size for the computation, but not because it affects the amount of visible I/O.

p	number of column blocks	column block size (bytes/proc)	memory used	total I/O (bytes)	total time (sec)
16	4	534,800	55%	71,417,856	231
16	8	242,400	26%	131,866,624	254
32	2	525,312	55%	33,619,968	117
32	4	262,656	28%	71,467,008	126
32	8	131,328	14%	131,923,968	141
64	2	262,656	28%	33,619,968	67
64	4	131,328	14%	71,532,544	77
64	8	65,664	7%	132,136,960	99

TABLE 2
Scaling of 2048×2048 *matrices*

Table 2 shows the dependence of the total run time and total I/O on both the number of processors and the memory used for a $2,048 \times 2,048$ matrix. The non-overlapped I/O time is not shown because the small size of the matrix allowed effective caching of data by the disk software in some cases. (Again, the small size was chosen to allow us to do a larger number of runs.) The results again show that the total I/O is inversely proportional to the amount of memory available to the program. The increase in total time, however, is the result of increased interprocessor communication. The data in Table 2 does show that the total I/O is almost independent of the number of processors. Thus, the algorithm scales well to large numbers of processors.

5. Choreographed I/O

In Sections 3 and 4 we showed how a simple I/O paradigm, Partitioned Secondary Storage, could be used to yield a high performance implementation of the LU factorization kernel. In this paradigm, the I/O is kept entirely local to each process – any aspects of I/O which are not local must be implemented through the message passing rubric. In this section we will examine a complementary paragidm, Choreographed I/O. The idea of choreographed I/O is that all the processes work together to perform an operation on secondary storage.

The paradigmatic example of this sort of operation is file reordering. In the file reordering operation the processes cooperate to logically read the entire file in one format and write

it in another. For many reorderings, such as row major to column major or reblocking an array, it has been proven that multiple passes through the file are necessary in order to avoid highly inefficient small sized reads or writes[4, 1, 8, 9]. It thus becomes desirable to utilize an algorithm which is specially tailored to make optimal use of I/O operations [2, 3, 9]. These algorithms attempt to bring in large chunks of files at once, rearrange these chunks and then output them again in large chunks. Typically the algorithms assume that there are D disks, each with a natural block size B (the physical size of a block which can be read from a disk in one I/O operation). Thus if the system has a total memory size of M then $\frac{M}{BD}$ I/O operations can be used to fill the memory. Some reordering of the read blocks followed by some writes are then necessary before any more data can be read. In the referenced paper it is assumed that the memory is monolithic and the cost of reordering data within memory is discounted. On a parallel machine the memory is likely to be distributed and one must be aware of the internal memory reordering costs. The issuing of the I/O requests must also be carefully managed. If the algorithm requires $\frac{M}{BD}$ I/O operations (each of which accesses all D disks) then the resulting data must be partitioned across the P memories of the P processors.

Choreographed I/O is exactly the process of managing such I/O requests. It allows the processes as a group to request data be read from many disks and distributed among the processes. Interprocess communication routines can then be used to reorder the data within the processes followed by a choreographed write. In the next section we describe a prototype file system, the Whiptail File System, which was implemented at Sandia in order to test out choreographed I/O.

6. The Whiptail File System

In order to study choreographed I/O we implemented a new file system on Sandia's Intel Paragon computer and the SUNMOS operating system. The goal was to produce a system which was small and quick like the whiptail lizard of New Mexico, hence the name whiptail file system.

6.1. The low-level system

For various technical reasons we could not build the features we desired directly on Intel's PFS parallel file system. On the otherhand, the PFS system promised to deliver much higher performance than the standard unix file system UFS. Thus we built our own low-level file system inside of files opened through PFS on each individual disk. Our low level file system maintains an inode-style directory of WFS files on each disk. Parallel files are thus striped across the disks by having entries in the directories of all appropriate disks.

In order to access the low-level file system users of WFS start by running the command

wfsnewfs -l disklist -t blocks -m numdisks,

where disklist is a list of the PFS file names on each disk, blocks is the number of blocks to use on each disk, and numdisks is the number of disks to stripe across.

Commands for listing wfs files (wfsls), copying files (wfscp), and deleting files (wfsrm) are provided but the low-level system was designed to be the minimum required in order to support our research into the higher level primitives which are useful for choreographing

I/O.

In the next section we will assume that a user has access to a low-level file system with the mechanism described above.

6.2. The high-level system

Within a parallel program a user can access the WFS via the following set of commands.

Each program is expected to initialize its use of WFS by calling start_wfs and to clean up by calling shutdown_wfs.

Between calls to start and shutdown the system, a user can open, close, rename, and delete files via the routines open_pfile, close_pfile, rename_pfile, and delete_pfile. As with standard Unix a file is opened by giving a filename and a file type. The type can be r, w, or rw depending on whether the file is to be read-only, write-only, or read-write, respectively. In addition, WFS requires that a maximum file size in blocks be given. This last requirement could be removed if the low-level system has more flexibility than the rudimentary system described above (and implemented at Sandia). The open command returns a file descriptor which can be used in read, write, and close commands. The rename and delete functions used file names.

An open WFS file can be accessed in two distinct methods: block access, stripeload access.

Direct block access. The block access routines, read_block, write_block, and iread_block, provide a means of directly accessing each block of a file. They are most likely to be useful to a program which is laying out the data in a specific manner across disks. Each command is given a file descriptor, a pointer to the block of data to be read or written, a disk number, and a block offset. Thus any given block on any given disk can be directly addressed. The iread version is non-blocking. An iwrite version was not implemented because the current system would necessarily treat it exactly like a blocking version.

Stripeloads. The stripeload routines, read_independent_stripeloads, write_independent_stripeloads, read_consecutive_stripeloads, and write_consecutive_stripeloads, allow the coordinated effort to read or write data from *all* the disks in a coordinated manner. Each striped access specifies an equal number of blocks to be retrieved from each disk (and a per disk offset block from which to start). The data in these blocks is then distributed to buffers specified for each processor. The current implementations logically concatenates the data for each disk in canonical order and distributes this data across a logical concatenation of the processor buffers (again in canonical order.) The "consecutive" functions remove the need to specify block offsets by having the file system maintain a file pointer for reads and writes which is automatically adjusted after each operation. Seek routines are supplied to adjust these pointers if desired.

WFS also supplies a variety of helper and maintenance routines. Further details of routine syntax and semantics can be found in the Appendix.

6.3. Using Choreographed I/O

As has already been mentioned the purpose of CIO is to facilitate I/O which involves the coordinated processing of large data files. Examples of such coordinated processing are sorting, reformatting files, matrix reblocking, and matrix arithmetic. The read and write routines described above are specially tailored to work with I/O optimal algorithms for these problems[2, 9, 10]. A typical I/O optimal algorithm is expressed in passes over the data. In each pass all the blocks of a file are read into memory in *memory loads*, i.e. as many blocks as will fit in memory at a time while leaving room for control and buffering. Each memory load is permuted (in the case of sorting or reformatting) or acted on (in the case of matrix arithmetic) and then written back to disk. The algorithms are carefully crafted so that memory loads can be read and written by accesses equal numbers of blocks from each disk. The striped routines of WFS allow exactly these accesses.

Shriver and Wisniewski implemented several of these routines on top of WFS and found that the algorithms were in fact easily expressed (see chapter 4 of [10]. Wisniewski further explored the issue of dealing with the internal structure of memory within an MP machine. The literature on I/O optimal algorithms had typically assumed a flat, global memory and thus ignored the issue of rearranging data or computing with data between reads and writes.

In chapter 5 of [10] several alternatives are discussed for mitigating the effects of internal memory structure. On possibility is to simply note that in the worst case every processor will need to send data to every other processor. This communication pattern, often called all-to-all communication, is well study and library routines exist for it. However, all-to-all communication is notoriously expensive in terms of network bandwidth require. In order to reduce the communication one can try several approaches: redirecting data to permuted locations during the I/O read (called tagging), clustering of blocks within subsets of the processors, and factoring the permutation.

In the tagging approach each block read from disk is partitioned into pieces bound for specific processors and sent directly to their destination. System support features such as *portals* in the Puma operating system would allow data to flow directly into the proper user address. Other systems might require copying data. Coordination to keep from flooding system buffers can also be problematic. A final concern is that tagging can create many small messages so the interconnection network and system software must be able to handle small messages efficiently.

In the cluster approach the algorithm is modified to cause all data read within a cluster of processors to be written by some processor in the same cluster. In other words the data need only be permuted within processor clusters. Variations of the BMMC permutation techniques are used to ensure clustering for reordering problems.

The factoring approach applies the I/O optimal, multipass techniques to the internal memory as well as to the disks. Instead of having to permute all of internal memory the factoring reduces the communication to simple, efficient primitives applied over multiple passes.

Further work remains to be done to determine what is the best practical approach to internal memory mapping. However, we expect that a mixture of system software primitives and the approaches above will lead to efficient algorithms on MP systems such as those at

Sandia.

7. Summary

In this report, we have discussed several options for parallel I/O focusing on scientific applications. Partitioned secondary storage (PSS) is one of the options. In PSS, each processor has its own disk (or section of a disk) and does not have direct access to any other processors' disks. This means that programs cannot share data indirectly through the disk system, and the programmer must control the placement and communication of data. This adheres to the distributed memory, message passing paradigm, which is known to be an effective paradigm on massively parallel computers.

Another option presented for parallel I/O was choreographed I/O (CIO). This is a methods for coordinating (choreographing) large data transfers from parallel disks to parallel memories. The data is striped across disks and groups of processors issue synchronized calls. This is especially useful in supporting I/O intensive kernels such as out–of–core sorting, permuting, FFT's and matrix multiplication.

We also discussed out–of–core LU factorization, which is an important kernel in scientific applications. We presented a theoretical analysis of the I/O requirements and a practical implementation. In the process, we showed that a good implementation is not necessarily an optimal (in terms of I/O complexity) implementation. In this case, the simpler, but non–optimal, implementation allowed the necessary I/O to be overlapped with the computations. We note here, however, that this observation is machine dependent and is limited to matrices where the computation dominates, and for today's machines, this includes matrices of several hundred thousand in each dimension.

Finally, we discussed the Whiptail file system, an experimental file system designed to test the practical implementation of CIO. This implementation is currently limited by the hardware capabilities of the parallel machine; however, initial testing has been done. This testing has shown that the potential for high–performance I/O exists, but that algorithms that use it must still be designed with distributed memory in mind (if it is to be implemented on a distributed memory machine).

Acknowledgements. We would like to thank Stephen Wheat, Rolf Riesen, Mack Stallcup and the rest of the PUMA Operating System development team for their technical support during the implementation of the out–of–core LU factorization. We would also like to thank Liddy Shriver, Len Wisniewski, Bruce Calder and Ryan Moore for their work on the Whiptail File System.

REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[2] T. H. Cormen. *Virtual Memory for Data-Parallel Computing.* PhD thesis, MIT, 1993.

[3] T. H. Cormen, T. Sundquist, and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR94-223, Dept. of Computer Science, Dartmouth College, July 1994. Preliminary version also appeared in Proceedings of the Fifth Symposium on Parallel Algorithms and Architectures.

[4] R. A. Floyd. Permuting information in idialized two-level storage. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 105–109. Plenum, 1972.

[5] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 2nd edition, 1989.

[6] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the Symposium on the Theory of Computing*, pages 326–332, 1981.

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

[8] J. S. Vitter and E. A. M. Shriver. Optimal disk I/O with parallel block transfer. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC '90)*, pages 159–169, May 1990.

[9] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2/3):148–169, August and September 1994.

[10] L. F. Wisniewski. *Efficient design and implementation of permutation algorithms on the memory hierarchy*. PhD thesis, Dartmouth University, 1996.

Appendix.

A. Using WFS in Application Programs

The Whiptail File system is easy for the programmer to use. In this appendix we describe all that is necessary to create a WFS program using WFS routines, execute the application code, as well as some hints for debugging the program.

Creating a Whiptail File System

Before any program uses WFS routines, the programmer must create and initialize a new WFS. At the system prompt, the programmer enters the `wfsnewfs` command. The `wfsnewfs` command requires that the programmer specify several parameters. The `-l` option takes a specified *file system name* of a Unix file that contains the names of the PFS or UFS files that will serve as the raw disk storage file on each of the disks. The `-t` and `-m` options allow the programmer to specify the number of blocks per disk to allocate for the new WFS and the number of disks on which the file system stripes the files. Other optional parameters permit the specification of the number of inodes, files, inode blocks, and directory blocks. Please note that all of these parameters can be specified at file system creation time, but cannot be changed after system initialization. For example, we can create a new WFS which uses 1024 blocks on each of 5 disks with the command

```
% wfsnewfs -l disklist -t 1024 -m 5
```

where `disklist` is a file that contains the names of the 5 PFS or UFS files used to store the file system information and data.

Once a file system has been created, the programmer has the ability to list the files in the file system using the `wfsls` command. The `wfsls` command requires that the programmer specify a single parameter, the name of the file that serves as storage for the first disk in the file system. If you prefer to remember only the file system name, the command

```
% wfsls `head -1 disklist`
```

also performs the same function.

There are commands that allow the application to copy and delete files from the system prompt. To copy a Unix file to WFS, the programmer uses the `wfscp` command. The `wfscp` command requires that the programmer provide the name of the Unix file to be copied, the name of the new WFS file, and the file system name. Currently, `wfscp` is not implemented, but the programmer can use Unix `cp`. To delete a file at the system prompt, the programmer uses the `wfsrm` command. The `wfsrm` command takes two parameters: the name of the file to be removed and the file system name.

The `matgen` utility generates data of different types of records. The standard use of the utility is to generate matrices of data, but it can also be used to generate files.

The usage is

```
% matgen -f ZFS-file -r rows -c cols -n num-type -t matrix-type
```

where `num-type` is `1` for integers or `2` for doubles and `matrix-type` can have the following values:

1 = fill the matrix with `1`'s
2 = fill the matrix with `0`'s
3 = fill the matrix with random numbers
4 = fill the matrix with consecutive numbers, starting with `0`.

Programming

When writing a program that uses WFS routines, the programmer must include the file `wfs.h`. This allows for use of the WFS routines. (These routines are presented later in this appendix.)

The function `start_wfs()` must be called before any of the file system routines are. The function `shutdown_wfs()` should be called before the program terminates.

The Paragon supports the `mynode()` and `numnodes()` routines to return the caller's logical order among the processors and the number of processors that were allocated for the application. Since WFS must be run in heterogeneous mode, the substitution of `my_group_offset()` for `mynode()` and `my_group_size()` for `numnodes()` is needed.

Compile the program code for the Paragon and link it with the `diskserver` code.

Executing

To run a program, the processors need to be divided into two groups: application-compute nodes and disk-server nodes. The number of application-compute nodes are left up to the programmer; the number of disk-server nodes should be the number of disks that the file system was created with. Dividing the nodes into two groups is achieved with a heterogeneous load operation. For example, with an executable of `a.out` that takes 3 parameters, we can execute with the command

```
% yod -F data4202500.compute80.disk5
```

where `data4202500.compute80.disk5` is the following Unix file

```
17x5
yod -sz 16x5:0,0 a.out matA2050 matB2050 4202500
yod -sz 5x1:0,16 diskserver -f disklist
```

It is important that the disk servers are the last submesh in the heterogeneous load.

The MACHINE WIDTH environment variable must be set to the number of groups of 16 processors the specific Paragon has. At Sandia National Labs, this is 4 for zia and 112 for acoma.

Debugging

The following might help in debugging:

- If the program seems like it is in an infinite loop, control-C will halt the processing and return control to operating system. This will close open files.
- If the program was supposed to create files during its run, the `wfsls` command line utility can be used to list the files in the specified file system. The command line utility `wfsdump` will print the contents of a parallel file.

WFS Routines

File System-Level Routines

WFS has routines which work on the file system:

```
void start_wfs (void);
int shutdown_wfs (void);
```

The routine `start_wfs()` allows a program to access files stored on WFS. It is a collective call. It starts the file system. The routine `shutdown_wfs()` closes the files currently open and exits. It returns `ERROR` and `NO_ERROR`.

Basic File Routines

WFS provides the following basic routines which allow programmers to manipulate a file as a unit in the file system:

```
int open_pfile (char *filename, char *type, int file_size_in_blocks);
int close_pfile (int fd);
int rename_pfile (char *new_filename, char *old_filename);
int delete_pfile (char *filename);
```

The routine `open_pfile()` returns the file descriptor of the file or the value `-1`. The parameter `type` specifies the way that the file will be used: `r` for reading, `w` for writing, and `rw` for reading and writing. The parameter `file_size_in_blocks` specifies the amount of space to pre-allocate to the file. If the file is being opened for for read only, the values specified should be `NOT_APPLICABLE`. This is a collective call.

The routine `close_pfile()` returns `NO_ERROR` or an error from `wfs_errors.h` if the file referenced to by the file descriptor `fd` could not be closed correctly. This is a collective call. The routine `rename_pfile()` returns `ERR_NONE` if there was no error when renaming the file. If there were errors (e.g., the `new_filename` already exists), an error code is returned (e.g., `ERR_EXISTS`). The routine `delete_pfile()` returns an error code identifying whether or not an error was encountered when deleting `filename`.

Block Access

WFS provides the following routines for a processor to directly read or write any one particular block of a file:

```
int read_block (int fd, void *buffer_pointer, int disk_num, int block_offset);
int write_block (int fd, void *buffer_pointer, int disk_num, int block_offset);
int iread_block (int fd, volatile int *read_flag, unsigned int *error, void *buffer_pointer,
                 int disk_num, int block_offset);
```

The `read_block()` and `write_block()` routines wait until the read or write of the requested block completes before returning. The programmer specifies the desired block by the `disk_num` and `block_offset` parameters for the file specified by the file descriptor (`fd`). The `buffer_pointer` identifies the local memory location for the read or written data. These routines return `ERROR` if there was an error during execution or 0 if there is no error.

The `iread_block()` routine does not wait for the reading of the block to complete. An `iread_block()` call passes a pointer to a `read_flag` variable which gets incremented when the reading of the block has completed. Blocks are read in the order that they are requested. Thus, the programmer can poll the `read_flag` variable to determine if the reading of a particular block has completed. If an error occurs during the execution of the `iread_block()` routine, a specified `error` variable will contain the appropriate error code.

Independent Access

WFS provides independent-access routines which allow each processor to simultaneously access a portion of one or more stripeloads of data. WFS supports the following independent-access routines:

```
int read_independent_stripeloads (int fd, int num_stripeloads, int *block_offset_array,
                                  void *buffer_pointer, int buffer_size);
int write_independent_stripeloads (int fd, int num_stripeloads, int *block_offset_array,
                                   void *buffer_pointer, int buffer_size);
```

The programmer must specify the file descriptor (`fd`), the number of blocks to be read per disk (`num_stripeloads`), a block offset for each disk to be accessed (`block_offset_array`), a buffer space in the local memory of the calling processor for the read or written records (`buffer_pointer`), and the size of the buffer space (`buffer_size`). The `buffer_size` must be a sufficiently-large integral number of blocks.

The independent-access routines are *collective-access* routines; that is, all the processors must call this routine to collectively access a quantity of data. The responsibility for receiving or providing the quantity of data read or written, respectively, is distributed over all the processors. Thus, each processor specifies the same `num_stripeloads` and `block_offset_array` parameters to collectively access the same data. The `buffer_pointer` and `buffer_size` parameters, however, may differ on each processor (e.g., the number of processors does not equally divide the number of records requested.)

Consecutive Access

WFS provides consecutive-access routines which allow each processor to simultaneously access a portion of one or more stripeloads of data. WFS supports the following consecutive-

access routines:

```
int read_consecutive_stripeloads (int fd, int num_stripes, void *buffer_pointer, int buffer_size);
int write_consecutive_stripeloads (int fd, int num_stripes, void *buffer_pointer, int buffer_size);
```

The programmer needs to specify the file descriptor (`fd`), the number of stripes requested (`num_stripes`), the buffer space in the local memory for the read or written records (`buffer_pointer`), and the size of the buffer (`buffer_size`). The consecutive-access routines are also collective-access routines.

WFS uses two separate consecutive file-access pointers, one for reading and one for writing. Upon opening a file, WFS initializes the consecutive file-access pointers to point to the first stripe. After the user performs a consecutive read or write on the file, WFS increments the appropriate consecutive file-access pointer. The WFS interface includes the following routines to reset the consecutive file-access pointers:

```
wfs_seek_read (int fd, int stripe_number);
wfs_seek_write (int fd, int stripe_number);
```

The programmer must specify a `stripe_number` to designate the new location in the file (`fd`) for the consecutive file-access pointer. The routines return `ERR_NONE` if no error was found; otherwise, they return `ERR_NOFENT`, meaning that the file is not found in the directory.

File and Configuration Information Routines

WFS supports the following file and configuration information routines:

```
int num_of_disks (int fd);
int generate_unique_filename (char *filename);
int sizeof_file_in_blocks (int fd);
int gsizeof_file_in_blocks (int fd);
int sizeof_block (int fd);
int starting_disk (int fd);
```

The routine `num_of_disks()` returns the number of disks that the file referenced by the file descriptor `fd` is striped across. It is an individual call.

The routine `generate_unique_filename()` will generate a filename in `filename` that does not match any filenames in the current file system. The file name generated will be of length `FILENAME_LENGTH`. This is a collective call; all nodes will have the same file name generated. This routine returns `ERROR` if there is an error or `ERR_NONE` otherwise.

The routines `sizeof_file_in_blocks()` and `gsizeof_file_in_blocks()` will return the number of blocks pre-allocated to the file. The routine `sizeof_file_in_blocks()` is an individual call; the routine `gsizeof_file_in_blocks()` is a collective call. The routines return `ERROR` if there was an error.

The routine `sizeof_block()` returns the size of the block, in bytes, that file is written to the disk system using.

The routine `starting_disk()` returns the disk number which represents the first disk that the file is written to.

Error and Synchronize Routines

WFS provides routines to print WFS errors and synchronize the application-compute nodes:

```
void wfs_perror (char *msg);
void subsync (void);
int wfs_sync (void);
int gwfs_sync (void);
```

The routine `wfs_perror()` prints the string `msg` followed by the last WFS error to occur and its error string. It is an individual call.

The routine `subsync()` allows the application-code nodes to be synchronized together.

The routines `wfs_sync()` and `gwfs_sync()` force the writing of a file to disk. The routine `wfs_sync()` is an individual call; the routine `gwfs_sync()` is a collective call.

B The Intel Paragon

A programmer needs some knowledge of the Paragon to write code using the WFS routines. See [Int93] for additional information.

Data is read from disk in chunks of data that is 65536 bytes = 64 KB long; we call this a block.

The Paragon has three types of nodes: service nodes, compute nodes, and I/O nodes. The service nodes run OSF and yod; the compute nodes run SUNMOS; the I/O nodes run OSF and fyod.

The Paragon consists of a set of nodes interconnected by a mesh. Some of these nodes may be connected to a RAID disk controller. Thus, the *I/O nodes* become dedicated to

performing any requests for access to data residing on the RAIDs. The programmer can make these I/O requests from any of the *compute nodes* which runs the application.

Nodes are allocated to an application; different programs can be loaded into sub-meshes of the allocated nodes. The nodes can be divided into sub-meshes of nodes (i.e., groups of nodes that will be running the same process) by an heterogeneous load operation.

Communication between nodes happens by sends and receives. A node posts a receive if it is expecting a message from another node. SUNMOS has buffer space, called communication space (or comm space), in which the nodes stores messages that have not had receives posted yet.