machines over a larger network. It is difficult to achieve true supercomputer performance from a workstation cluster unless the problem is embarrassingly parallel. For example, compare the results shown in Figure 5 from timings on a cluster of IBM RS6000s to the results shown in Figure 6 from timings on MPP systems; in fact, the cluster is not able to achieve any real speedups. However, small workstation clusters are well suited for solving large problems that cannot fit on a single machine. Single processor systems with virtual memory prove prohibitively slow due to excessive page-swapping with the hard disk. For example, an $N_x = N_y = N_z = 100$ problem (≈ 64 MBytes of data) on a single SPARCStation IPX requires almost one hour (3430 secs) to execute, while a 10 processor IPX cluster requires only 190 seconds, demonstrating superlinear speedup. In short, it is the ability to fit larger problems in memory, rather than overall parallel efficiency, that may make workstation clusters practical for large scale scientific computing.

REFERENCES

1. A. Beguelin, J. Dongarra, A. Geist, R. Manchek, V. Sunderam, *User's Guide to PVM: Parallel Virtual Machine*, ORNL/TM-11826, Mathematical Sciences Section, Oak Ridge National Laboratory, Sept. 1991.
2. K. R. Bennett, "Fast Direct Solution of Three-Dimensional Poisson and Helmholtz Problems on Distributed Memory Machines," *Proceedings of Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, Virginia, March 22-24, 1993.
3. K. R. Bennett and B. Bialecki, *Fourier Matrix Decomposition Methods for the Least Squares Solution of Singular Neumann and Periodic Hermite Bicubic Collocation Problems*. Submitted, 1993.
4. B. Bialecki, G. Fairweather, K. R. Bennett, *Fast Direct Solvers for Piecewise Hermite Bicubic Orthogonal Spline Collocation Equations,* SIAM J. Numer. Anal., 29, 1992, pp. 156-173.
5. J. C. Diaz, G. Fairweather, and P. Keast, *Algorithm 603 COLROW and ARCECO: FORTRAN Packages for Solving Certain Almost Block Diagonal Linear Systems by Modified Alternate Row and Column Elimination*, ACM Trans. Math. Softw., 9 (1983), pp. 376–380.
6. R. Pozo, " A Streams-Based Interface in C++ for Programming Heterogeneous Systems," *Proceedings of CRNS-NSF Workshop on Environment and Tools for Parallel Scientific Computing*, September 7-8, 1992, Saint Hilaire du Touvet, France.
7. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.
8. V. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience* Vol. 2, No. 4, Dec. 1990.

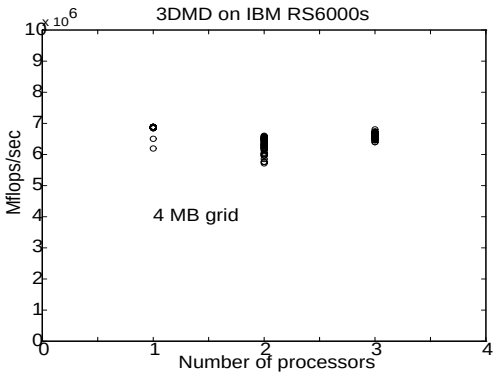

Figure 5. Scaled speedup for problems with running time proportional to the number of processors

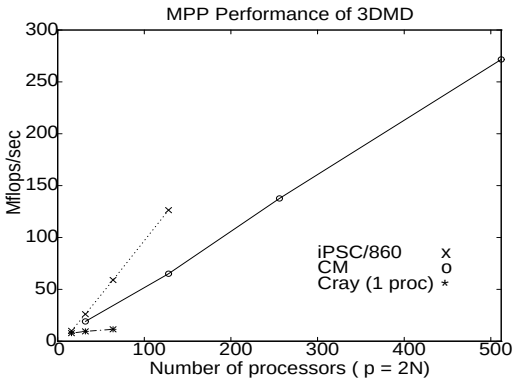

Figure 6. Performance for MPP machines using communication primitives (Cray 1 processor results included for comparison)

Although the time spent in communication is fairly negligible on MPP systems, it accounted for up to 50% of the elapsed time in the Ethernet/Workstation environments tested. This is in agreement with the general observation that large task granularity is necessary for reasonable performance on Ethernet configurations.

6. CONCLUSIONS

The SMP interface greatly simplified the communication kernel in this application example. Because SMP works with three-dimensional grids, the communication interface was more natural and closely matched the mathematical abstraction of the application. The communication code was easier to design, debug, and tailor with SMP, compared to traditional send/receive message-passing interfaces. Further, the SMP interface did not incur any measurable overhead. After running various experiments with a prototype SMP library, we were not able to detect any performance degradation. (This observation was to be expected, since SMP functions avoid data copying and are typically inlined by the C++ compiler.)

As observed with most Ethernet-based network clusters, the communication bandwidth between machines is a bottleneck. Network-based solutions are only practical for problems with large granularity. Our 3DMD application contains considerable all-to-all global communication which typically consumes about 50% of the overall execution time. Thus we see reasonable speedups only for small ($p < 8$) workstation clusters.

As expected, our wide-heterogeneous-network performances were not as efficient as homogeneous local-area networks. There are two reasons for this. One is that all data packets to have pass through XDR (Internet RFC 1014) data translation to ensure a consistent floating-point representation. This introduces excess message overhead compared to homogeneous cases, but the process is completely processor-local and, with today's powerful workstations, the overhead is negligible compared to the network bottleneck. The more serious problem is the increased network traffic encountered in accessing remote

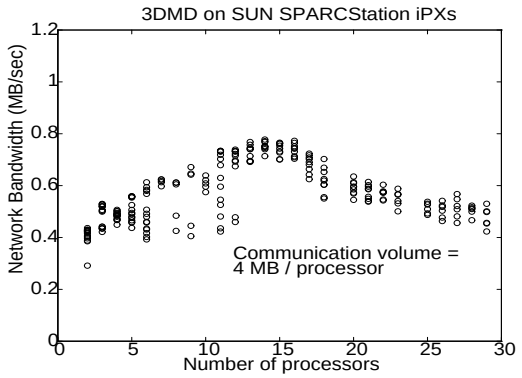

Figure 3. Network performance for increasing problem sizes

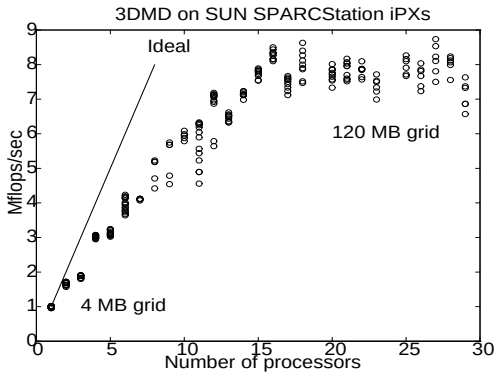

Figure 4. Scaled speedup for problems with running time proportional to the number of processors

iments on computing environments which are typical of those found in university and industrial research centers and do not require specialized hardware or network modifications. We used elapsed wall-clock time for our measurements, as opposed to CPU time, or any other"virtual" elapsed time. Ultimately, this is how users judge whether a program runs quickly or slowly. (Measuring CPU time was once a widespread practice for timing serial programs on heavily loaded multiuser systems, but such measurements hide network communication cost, since idle time while processors wait for messages is not counted.) Elapsed time may sometimes be misleading when many users are sharing a single CPU, but in our case experiments typically ran on idle workstations overnight, so this posed little problem.

The SUN4 network performance studies were made on a Ethernet cluster of 32 SUN 40 MHZ/ 8 MB SPARCStation IPXs at the University of Tennessee. Measurements were made in the late evening/early morning hours, when network traffic and workstation utilization was relatively light, but no attempt was made to alter or modify the everyday network environment of these machines – no network barriers were used in the timing, nor did we restrict access to these machines in any way. Each timing measurement for a particular processor/problem size configuration was taken at least 10 times over a span of several days to establish statistical means and variances and ensure some consistency in results. In our experiments, the performance showed less than 10% variance.

The effects of communication bandwidth on a lightly loaded Ethernet network are shown in Figure 3. The network typically saturates around 800 KB/sec, but sustains 500 KB/sec performance even for large ($p \approx 30$) configurations. It should be noted that in the single-processor case, no PVM or SMP++ communication calls are made – the transpose is performed at the memory-to-memory transfer rate of the IPX, about 3.3 MB/sec.

Figure 4 illustrates the measured speedups for problem sizes that scale with the number of processors. In this case, we chose the problem size so that each processor receives roughly 4 MBytes of data. The results show that speedups diminish after 15 processors. Fairly good efficiency ($> 80\%$) is achieved on clusters of between 2 and 10 processors.

Table 2
Code for the 3-D matrix transposition communication.

Fortran/PVM Version

```
      msgsize = 2*nsubx*nslices
      vecleng = 2*nsubx

      do 10 i = 0,ntids-1
        if (i .ne. me) then
          indcol = i*nslices+1
          call pvmfinitsend(PVMDEFAULT, info)
          do 20 is = 1,nslices
            call pvmfpack(REAL8, f1(1,indcol,is),
     *                    msgsize, 1, info)
 20       continue
          call pvmfsend(tids(i+1), me, info)
          call pvmfrecv(-1, i, info)
          do 30 is = 1,nslices
            call pvmfunpack(REAL8, f1(1,indcol,is),
     *                      msgsize, 1, info)
 30       continue
        end if
 10   continue
```

C++/SMP++ Version

```
for (i=0; i<ntids; i++)
{
  Grid3DRef G(f1(Index(1,2nx),
                 Index(0,nslices-1)+i*nslices,
                 Index(1,2nx)));

  msgout(tid[i], me) << G << flush();
  msgin (SMPMessageType(i)) >> G;
};
```

Table 2 we provide a comparison between the Fortran/PVM code and the C++/SMP code. The 3D matrices are distributed along a one-dimensional decomposition in the z direction. Thus, each processor receives a $2N_x \times 2N_y \times 2N_z/p$ slab. The submatrix shape that is communicated in each message is $2N_x \times 2N_y/p \times 2N_z/p$. Since this is not a contiguous memory region, the Fortran code (Table 2) must be broken to individual communication calls that send and receive each $2N_x \times 2N_y/p$ plane separately. SMP, on the other hand, recognizes 3D grids as fundamental objects, supports distributed vectors and matrices, and can perform communication on any arbitrary subgrid. Therefore, it is a natural choice for this application. In the C++/SMP code segment shown in Table 2, the subgrid reference G is explicitly named and is used both as a source and destination. The `Index()` object is used to denote index ranges, i.e. `A(Index(1,M))` is similar to the Fortran-90 subscript notation `A(1:M)`. The `msgout()` function specifies the processor destination (`tid[i]`) and message type (`me`). Similarly, the `msgin()` expression denotes a receipt of a particular message of type `i` and stores the message contents into the subgrid specified by G.

The SMP++ library used in this implementation is built atop the Parallel Virtual Machine (PVM) environment [1], [8]. This software environment easily configures networks of workstations and supercomputers via a simple message passing library. The PVM software package is freely available and widely used.

5. PERFORMANCE RESULTS

We illustrate the performance of this implementation by providing timing results for various architecture configurations. Our goal in testing was to accurately reflect the performance of the solver under realistic conditions. Therefore, we conducted our exper-

the parallel operations on its resident subvectors without communicating with other processors. However, Step 2 requires subvectors which are allocated across processors. If the data array is transposed across two dimensions, a task requiring global ("all-to-all personalized") communication, each subvector will then lie completely within a single processor. In Fig. 2, the layout after this transposition is indicated on the right. At this point, Step 2 may proceed, with its calculations being performed in parallel without communication. After Step 2, the data is again transposed, so that the layout is as it was originally, allowing Step 3 to proceed, again without interprocessor communication.

4. SMP++

To accomplish the initial distribution of subproblems to processors and subsequent data transpositions, we use the SMP++ library, which serves to abstract, rather than replace, the basic message-passing communication model. This interface supports no `send()` or `recv()` calls; instead communication is accomplished by inheritance of basic message stream classes and the overloading of the standard stream *insertion* operator `>>` and the *extraction* operator `<<` for input/output (I/O) in C++. The approach is based on the notion that interprocessor communication is simply another form of I/O. There is a strong analogy between how programs interact with one another via files and how processes communicate with one another via messages. Thus, the syntax used to denote the reading of a data value from a generic stream, for example, can be made to appear the same whether the data originates from a file, a pipe, input from the user's terminal, or a message from another processor. This simple, yet powerful, idea unites message-passing and stream I/O at a higher-level of abstraction, and has allowed for the creation of an extendable heterogeneous programming interface. The benefit of this interface is that it may be used without changing the basic layouts of existing distributed-memory programs or sacrificing run-time performance.

This model supports both blocking and non-blocking messages, with integer-based message types. Such message-passing interfaces are the most widely supported by commercial systems. Sending messages consists of three steps: (1) providing mailer information, destination process(es) and message type, (2) packing the information into the message stream, and (3) explicitly flushing the buffer to move the information out onto the communications network. The predefined stream `msgout` plays a similar role to standard output stream `cout`; that is, information sent to `msgout` is buffered until the stream is explicitly flushed. Thus the message can be built piecewise while controlling message granularity for communication performance. The operator `<<` adds data elements to this buffer and the `flush()` operator physically sends the message to its destination. For example,

```
msgout(proc,msgtype) << i << j << x << flush();
```

packs three data elements (two integers and a floating point) to the process denoted by `proc` with the integer-valued `msgtype`. Message elements of fundamental data types (`char`, `int`, `float`, `double`) are typically passed by value. Therefore, the insertion operator `<<` can accept expressions of arbitrary complexity. The reader is referred to the SMP overview[6] for more details.

To illustrate how SMP simplifies the message passing interface in this application, consider the communication kernel used in the three-dimensional matrix transpose. In

Table 1
THE OSC-MD ALGORITHM

1. Compute:	a.	$\vec{g}^* = (I_x \otimes Z_y^T B_y^T \otimes I_z)\vec{f}$
	b.	$\vec{g} = (Z_x^T B_x^T \otimes I_y \otimes I_z)\vec{g}^*$
2. Solve:		$(\Lambda_x \otimes I_y \otimes B_z + I_x \otimes \Lambda_y \otimes B_z + I_x \otimes I_y \otimes A_z)\vec{v} = \vec{g}$
3. Compute:	a.	$\vec{u}^* = (I_x \otimes Z_y \otimes I_z)\vec{v}$
	b.	$\vec{u} = (Z_x \otimes I_y \otimes I_z)\vec{u}^*$

3. A DISTRIBUTED MEMORY MD IMPLEMENTATION

The highly parallel nature of the OSC-MD algorithm is evident if the data vector is arranged as a three-dimensional array throughout the computations. Each step of the algorithm can then be viewed as independent operations on a collection of subvectors of this array. For example, the system in Step 2 can be viewed as $4N_xN_y$ independent almost block diagonal (ABD) linear systems of order $2N_z$, where the right hand sides are subvectors of the three-dimensional array which run in the direction corresponding to the space variable z. (Note: We solve these systems using the package COLROW [5], which takes advantage of the sparse structure of the ABD systems.) Assuming that the data layout is fixed, the direction of the subvectors which may be handled independently in each step are indicated in Figure 1.

In the heterogeneous SMP++ implementation presented here, we have parallelized the algorithm across one of the dimensions to reduce the amount of global communication required. With this approach, each processor holds "slices" of the data array, as depicted for a small example in Fig. 2. In this figure, each processor contains the elements of the array located vertically above it. Bold lines are used to help visualize the group of elements contained on each processor. In Steps 1a and 1b, each processor can complete

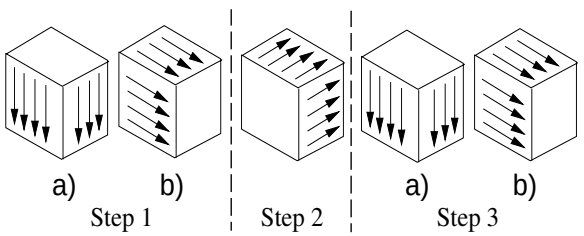

Figure 1. Data usage pattern in the matrix decomposition algorithm

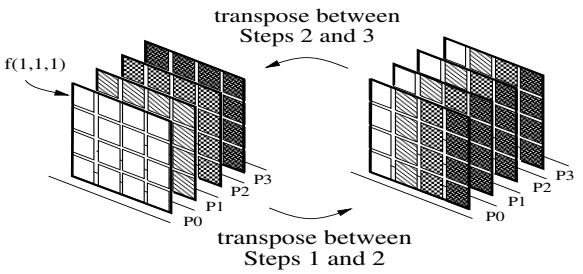

Figure 2. Layout of 3-dimensional data array in processors

here, in which the problem has been discretized using OSC, comes from the ability to provide explicit formulas for the eigensystems corresponding to the one dimensional Hermite cubic collocation problems and the ability to carry out forward and inverse transformations using fast Fourier transforms (FFTs). In this paper, we describe how the efficiency can be increased on distributed memory systems by exploiting the inherent parallelism of this algorithm.

2. THE MATRIX DECOMPOSITION ALGORITHM

We restrict our attention to three-dimensional elliptic partial differential equations of the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + L_z u = f(x,y,z), \quad (x,y,z) \in \Omega = (a_x, b_x) \times (a_y, b_y) \times (a_z, b_z), \tag{1}$$

where

$$L_z u = p(z)\frac{\partial^2 u}{\partial z^2} + q(z)\frac{\partial u}{\partial z} + r(z)u,$$

and in each direction either Dirichlet, Neumann, or periodic boundary conditions are specified. After imposing a rectangular grid on the physical domain, if we take N_x, N_y and N_z to be the numbers of mesh subintervals in each coordinate direction, the method of OSC results in a system of $8N_xN_yN_z$ equations which can be written in the tensor product form

$$(A_x \otimes B_y \otimes B_z + B_x \otimes A_y \otimes B_z + B_x \otimes B_y \otimes A_z)\vec{u} = \vec{f}, \tag{2}$$

where each of the component matrices A_x, B_x, A_y, B_y, A_z and B_z are almost block diagonal of order twice the number of mesh subintervals in the corresponding coordinate direction. For details of the OSC method with Hermite cubic polynomials, refer to [4]. Here, we note only that in contrast to traditional finite difference methods, this collocation method gives an approximate solution of the differential equation which is defined over the entire spatial domain while maintaining fourth order accuracy at the nodal points of the mesh and it has been noted that this collocation approximation has superconvergent derivatives at these same nodal points.

The matrix decomposition algorithm relies on the knowledge of matrices Z_* and Λ_* which satisfy the properties

$$Z_*^T B_*^T A_* Z_* = \Lambda_*, \qquad Z_*^T B_*^T B_* Z_* = I_*,$$

where $*$ denotes either x or y, I_* is an identity matrix of order $2N_*$ and Λ_* is diagonal. These Z and Λ matrices, chosen for the x and y coordinate directions in accordance with the specified boundary conditions, are used to decouple the system (2), as described in the algorithm presented in Table 1.

With the Hermite cubic basis functions chosen in the standard way (cf. [4]), the A_* and B_* matrices have only four non-zero entries in each row, and this sparsity can be exploited in the matrix multiplications in Step 1, and in the linear system solutions in Step 2. Since the Z_* matrices are composed of blocks of sines and cosines (see [3, 4]), fast Fourier transforms may be used for the matrix multiplications involving these matrices in Steps 1 and 3. Steps 1 and 3 can be computed in $O(N_xN_yN_z(\log N_x + \log N_y))$ operations, and Step 2 requires $O(N_xN_yN_z)$ operations.

Fast Three-Dimensional Elliptic Solvers on Distributed Network Clusters

Roldan Pozo [a]
Karin Remington [b]

[a]University of Tennessee, Department of Computer Science, Knoxville, Tennessee, 37996-1301 pozo@cs.utk.edu

[b]Oak Ridge National Laboratory, P.O. Box 2008, Oak Ridge, Tennessee 37831-6367 kbennett@msr.epm.ornl.gov

We examine the performance of a fast direct orthogonal spline collocation solver for three-dimensional Poisson, Helmholtz and more general separable linear elliptic problems on heterogeneous parallel architectures. We distribute the problem across a given parallel architecture using SMP++, an object-oriented message passing interface in C++ built upon the Parallel Virtual Machine (PVM) environment. Preliminary performance results are provided for various architecture configurations. These results indicate reasonable performance on small clusters of processors, and demonstrate the utility of a distributed approach over a sequential approach when faced with large problems.

1. INTRODUCTION

The solution of the Helmholtz equation, or Poisson's equation as a special case, often forms the computational core of large-scale modeling techniques for grand challenge applications. We describe the implementation of a fast direct method based on a matrix decomposition (MD) technique for the efficient solution of such problems on heterogeneous systems using SMP++[6], an object-oriented message-passing interface in C++. In Section 2, we give a short introduction to the MD technique and orthogonal spline collocation (OSC), and we present the OSC-MD algorithm which we have implemented. In Sections 3 and 4, we briefly describe the distributed memory implementation of this algorithm and the SMP++ software used to facilitate this implementation. Finally, we present the results of performance tests of the software on various configurations of clustered workstations.

In the two dimensional case, fast direct MD techniques transform the corresponding approximation problem with respect to one variable, thus reducing the problem to a collection of independent one-dimensional (1-D) subproblems. This decoupling is accomplished based on the knowledge of eigensystem information, and is particular to the discretization method used (e.g. finite differences, finite element Galerkin, or collocation). Once these subproblems are solved, the solution of the original approximation problem is obtained by applying the inverse of the transformation first applied. This can be extended to three dimensions by transforming with respect to two of the variables, again reducing the problems to a set of 1-D subproblems. The sequential efficiency of the algorithm we present