A scalable implementation of the NAS Parallel Benchmark BT on distributed memory systems

by V. K. Naik

In this paper, we describe an efficient and scalable implementation of the NAS Parallel Benchmark BT suitable for distributed memory systems such as the IBM Scalable POWERparallel Systems®. After describing the parallelization and data partitioning methods used, we outline some of the optimization steps used to realize good performance on individual processors and to reduce the communication overheads on the IBM SP1™ and SP2™ systems. We present performance results on up to 128 nodes of the SP1, and on the SP2 with wide nodes. We describe the performance on the standard Class A and Class B problem sets. To show the scalability of our parallelization methods, we present the performance of two additional data sets.

In 1991, the Numerical Aerodynamic Simulation (NAS) program at the NASA (National Aeronautics and Space Administration) Ames Research Center announced a set of applications and kernels for benchmarking highly parallel supercomputers.[1,2] These benchmarks are representative of the computations commonly encountered in aerophysics applications. Unlike many other benchmarks, these benchmarks are specified using the paper-and-pencil approach. The problems are completely specified (in text form), including the numerical methods to be used, but the benchmarking rules do not specify any particular implementation techniques or algorithms for parallelization. It is entirely up to the implementor to decide on the parallelization techniques, language constructs, the data structures, memory use, or processor allocation. Details of implementing the numerical methods are also left out. However, there are a few restrictions. One requirement is that the tests must be conducted with a specified set of input parameters and the test results must conform with the expected output within a specified level of tolerance. Another restriction is that all programs must be written in a high-level language such as FORTRAN or C; furthermore, no special-purpose library can be accessed for executing these benchmarks.

In the recent past, performance results characterizing various parallel platforms have been reported by many vendors as well as by researchers at NASA and elsewhere. The depth of these benchmarks and the fact that they capture the essence of typical large-scale computational fluid dynamics (CFD) applications have made these benchmarks popular, not only for the purpose of evaluating parallel supercomputing systems but also in demonstrating the viability of novel software and architectural concepts. As a result, the NAS parallel benchmark

Table 1 A summary of the application benchmark characteristics for the two problem sizes

	SP	BT	LU
Class A:			
Problem size	64 × 64 × 64		
Number of iterations	400	200	250
Number of floating-point operations ($\times 10^9$)	102.0	181.3	64.6
MFLOPS on Y-MP/1	216	229	194
Class B:			
Problem size	102 × 102 × 102		
Number of iterations	400	200	250
Number of floating-point operations ($\times 10^9$)	447.1	721.5	319.6
MFLOPS on C-90/1	627	572	493

suite has become one of the most widely used benchmarks in many areas of high-performance parallel computing. The NAS group at NASA Ames routinely compiles and distributes performance results on various parallel systems. For more details, please refer to Reference 3.

In this paper, we describe an efficient implementation of one of the three application benchmarks—the Block Tridiagonal (BT) benchmark—on distributed memory systems. In addition to describing the implementation strategies for scalable systems, we also describe steps for optimizing the code on the IBM scalable POWERparallel* 1 (SP1*) system and the IBM scalable POWERparallel 2 with wide nodes (SP2*-w) system. We consider single processor optimization steps as well as steps for reducing communication overheads. Besides describing the performance-enhancing techniques for SP1 and SP2-w, a goal of this paper is to examine the scalability of the SP architecture for computations characterized by the BT benchmark. For this we present results on up to 128 processors using four different data sets. In Reference 4, we have presented a similar study of a distributed memory implementation of the Scalar Pentadiagonal (SP) benchmark. An implementation of the Lower-Upper Diagonal (LU) benchmark is described in Reference 5.

As far as we know, our implementation of the BT benchmark, as described in this paper, has delivered the best performance on the SP1 and SP2 systems. Moreover, as of the writing of this paper, the performance of our implementation on the SP2-w with 32 or more processors is significantly better than the performance of any other implementation on any other parallel system (except for the Fujitsu VPP500**) with the same number of processors.[3] One objective of this paper is to describe our implementation in detail so other studies, including nonbenchmarking types, may benefit from its understanding.

First we present some background information on the NAS parallel benchmarks. In the section after that, we describe the mathematical and numerical problem solved by the BT benchmark. A sequential implementation of this benchmark is then outlined. Next we briefly describe the IBM SP systems used in this study. We describe our parallel implementation and the optimizations for performance in succeeding sections. The performance results are presented in the two sections just before the conclusions.

NAS parallel benchmarks

The NAS benchmark suite consists of eight problems: five kernels and three simulated CFD applications. For details and implementation rules, please refer to References 1 and 2. The kernels represent solvers in the form of compact problems. These kernels are relatively simple (each with a few hundred lines of sequential FORTRAN code) and are meant to provide insight into the performance of particular types of numerical computations.[6] The three simulated CFD applications are more complex than the kernels. Each of these applications consists of several data structures, and a typical sequential implementation of any of these benchmarks, in FORTRAN, results in a few thousand lines of code. The data dependencies imposed by the numerical methods in these applications and their computational requirements resemble closely those in the state-of-the-art CFD application codes. Thus, the implementation techniques of these applications are more typical of real CFD applications.[6] However, absent from the application benchmarks are complex boundary conditions and I/O operations that are typically present in many real CFD applications. For that reason, the three application benchmarks are also referred to as pseudo applications.

The three application benchmarks are: the Scalar Pentadiagonal (SP), Block Tridiagonal (BT), and Lower-Upper Diagonal (LU) benchmarks. For comparison of performance, NASA has defined two standard problem sets for each benchmark that are referred to as Class A and Class B size problems.[7]

In Table 1, we present a summary of the computational characteristics of the three application benchmarks under the two standard problem sets. The floating-point operation counts and the MFLOPS (millions of floating-point operations per second) specified in that table are from Reference 7. Note that the floating-point operation counts are for a particular implementation of these benchmarks and were measured using the Cray hardware performance monitor.[6]

According to the rules of the NAS parallel benchmarks, performance is to be reported in terms of the elapsed time (based on "wall-clock" time) to complete the computations over the required number of iterations. Another meaningful measure used in reporting performance is the ratio to the best execution times on a single node of a Y-MP** system (for Class A problems) or on a single node of a C90 system (for Class B problems).

Benchmark BT

In this section, we describe some of the salient points of the benchmark BT so as to facilitate a meaningful discussion on the implementation and performance issues. Most of the discussion in this section is based on Chapter 3 of Reference 1. For complete details, refer to that citation.

The problem solved in benchmark BT is that of computing the numerical solution for a synthetic system of five nonlinear partial differential equations (PDEs) representing some of the key characteristics exhibited by the Navier-Stokes equations. An implicit type of numerical solution is used in solving this system of PDEs. This method is used as a solver in many of the computational fluid dynamics (CFD) programs designed for the numerical solution of three-dimensional Euler/Navier-Stokes equations using finite-volume or finite-difference discretization on structured grids.

The system of five nonlinear PDEs is as follows:

$$\frac{\partial \mathbf{U}}{\partial \tau} = \frac{\partial \mathbf{E}(\mathbf{U})}{\partial \xi} + \frac{\partial \mathbf{F}(\mathbf{U})}{\partial \eta} + \frac{\partial \mathbf{G}(\mathbf{U})}{\partial \zeta} + \frac{\partial \mathbf{T}(\mathbf{U}, \mathbf{U}_{\xi})}{\partial \xi} + \frac{\partial \mathbf{V}(\mathbf{U}, \mathbf{U}_{\eta})}{\partial \eta} + \frac{\partial \mathbf{W}(\mathbf{U}, \mathbf{U}_{\zeta})}{\partial \zeta} + \mathbf{H}(\mathbf{U}, \mathbf{U}_{\xi}, \mathbf{U}_{\eta}, \mathbf{U}_{\zeta}) \quad (1)$$

where $\mathbf{U} = [u^{(1)}, u^{(2)}, u^{(3)}, u^{(4)}, u^{(5)}]^T$ is a vector function of temporal variable τ and spatial variables (ξ, η, ζ) that form the orthogonal coordinate system in $\Re^3$. **E**, **F**, **G**, **T**, **V**, **W**, and **H** are 5×1 vector functions whose components are prescribed as functions of **U** or **U** and its derivatives. The boundary conditions are of uncoupled Dirichlet type and the initial values of **U** at $\tau = 0$ are obtained by a transfinite, tri-linear interpolation of the boundary data.[1,8] The forcing function vector **H** is chosen such that, with the specified boundary and initial conditions, the system of PDEs given by Equation 1 satisfies an exact solution (a fourth order polynomial in ξ, η, and ζ) to **U**. The computations of the BT benchmark seek to obtain a discrete approximation to the steady-state solution of the PDEs, using a pseudo-time marching scheme (two-level, first-order accurate, Euler implicit) and a spatial discretizing procedure based on finite difference approximations (second-order accurate central difference approximations in each of the three coordinate directions). A linear fourth-difference dissipation term is added to the right side of Equation 1 so that the numerical scheme converges to a steady-state solution. If $\mathbf{U}^n$ and $\mathbf{U}^{n+1}$ are solutions at time-step n and $n + 1$, respectively, and $\Delta\mathbf{U}^n$ is $\mathbf{U}^{n+1} - \mathbf{U}^n$, then the numerical procedure involves the solution of a linear system of equations for $\Delta\mathbf{U}^n$ to determine $\mathbf{U}^{n+1}$. The linear system of equations has the following form:

$$(\mathbf{I} + \Delta\tau\mathbf{L})\Delta\mathbf{U}^n = \Delta\tau\mathbf{R} \quad (2)$$

where **L** consists of the flux Jacobian terms corresponding to the vectors **E**, **F**, **G**, **T**, **V**, **W** and the implicitly treated dissipation terms. **R** consists of the spatial difference terms for the vectors **E**, **F**, **G**, **T**, **V**, **W**, the forcing function vector **H**, and the added dissipation terms. In the above equation, the left-hand side (LHS) is the implicit part, and the right-hand side (RHS) is the explicit part.

All three pseudo-application benchmarks compute $\Delta\mathbf{U}^n$ in Equation 2 numerically, using an iterative method, and from that term advance the solution to $\mathbf{U}^{n+1}$. Only one iteration per time step is generally sufficient for the pseudo-time marching schemes used in all three benchmarks. The benchmark BT differs from the other two (SP and LU) in the manner in which the implicit operator in the LHS of Equation 2 is approximated and applied. In this benchmark, the implicit operator in Equation 2 is approximately factored using the Beam-Warming Approximate Factorization scheme.[9] The RHS

Figure 1 A functional outline of the BT benchmark

Set Boundary values of $\mathbf{U}_{i,j,k}$ for $(i, j, k) \in \partial D_h$
Set Initial values of $\mathbf{U}^0_{i,j,k}$ for $(i, j, k) \in D_h$
Compute forcing function vector, $\mathbf{H}^*_{i,j,k}$ for $(i, j, k) \in D_h$
Initialize $[\mathbf{RHS}]_{i,j,k}$ at $\tau = 0$ for $(i, j, k) \in D_h$

for $\tau = 1$ **to** Maxsteps **do**

i. **Perform** ξ-sweep:
Setup coefficient matrices:
i(a). $\mathbf{M}_\xi = (\mathbf{I} - \Delta\tau[\mathrm{D}_\xi(\mathbf{A})^{\tau-1} + D^2_\xi(\mathbf{N})^{\tau-1}])$
Solve linear system of equations for $[\Delta\mathbf{U}_1]_{i,j,k}$ for $(i, j, k) \in D_h$:
i(b). $\mathbf{M}_\xi\Delta\mathbf{U}_1 = [\mathbf{RHS}]^{\tau-1}$.

ii. **Perform** η-sweep:
Setup coefficient matrices:
ii(a). $\mathbf{M}_\eta = (\mathbf{I} - \Delta\tau[D_\eta(\mathbf{B})^{\tau-1} + D^2_\eta(\mathbf{Q})^{\tau-1}])$
Solve linear system of equations for $[\Delta\mathbf{U}_2]_{i,j,k}$ for $(i, j, k) \in D_h$:
ii(b). $\mathbf{M}_\eta\Delta\mathbf{U}_2 = \Delta\mathbf{U}_1$.

iii. **Perform** ζ-sweep:
Setup coefficient matrices:
iii(a). $\mathbf{M}_\zeta = (\mathbf{I} - \Delta\tau[D_\zeta(\mathbf{C})^{\tau-1} + D^2_\zeta(\mathbf{S})^{\tau-1}])$
Solve linear system of equations for $[\Delta\mathbf{U}_3]_{i,j,k}$ for $(i, j, k) \in D_h$:
iii(b). $\mathbf{M}_\zeta\Delta\mathbf{U}^{\tau-1} = \Delta\mathbf{U}_2$.

iv. **Update** solution to time-step τ:
$\mathbf{U}^\tau = \mathbf{U}^{\tau-1} + \Delta\mathbf{U}^{\tau-1}$

v. **Compute** $[\mathbf{RHS}]^\tau_{i,j,k}$ for $(i, j, k) \in D_h$

end for

(explicit part) of Equation 2 is unaffected by these modifications. The LHS takes the following form:

$$\mathrm{LHS} = \left[\mathbf{I} - \Delta\tau\left(\frac{\partial(\mathbf{A})^n}{\partial\xi} + \frac{\partial^2(\mathbf{N})^n}{\partial\xi^2}\right)\right] \cdot \left[\mathbf{I} - \Delta\tau\left(\frac{\partial(\mathbf{B})^n}{\partial\eta} + \frac{\partial^2(\mathbf{Q})^n}{\partial\eta^2}\right)\right] \cdot \left[\mathbf{I} - \Delta\tau\left(\frac{\partial(\mathbf{C})^n}{\partial\zeta} + \frac{\partial^2(\mathbf{S})^n}{\partial\zeta^2}\right)\right]\Delta\mathbf{U}^n \qquad (3)$$

where **A**, **B**, **C**, **N**, **Q**, and **S** are 5×5 flux Jacobian matrices for the specified problem. For the exact representation of these Jacobian matrices, please refer to Reference 1.

A functional description of the BT benchmark is shown in Figure 1. Observe that $\Delta\mathbf{U}^n$ is computed in the $(n + 1)$th iteration over Steps *i*, *ii*, and *iii*, where ξ, η, and ζ directional factors of the implicit operator are applied, respectively. Collectively, we refer to these steps as the *implicit phase* of the computations and the three sweeps as the *implicit sweeps*. In each of these sweeps, multiple independent systems of block tridiagonal equations (each block being a 5×5 matrix) are solved. If the spatial discretization has N_ξ, N_η, and N_ζ mesh points in ξ, η, and ζ directions, respectively, then, in the ξ-sweep, altogether $(N_\eta - 2)(N_\zeta - 2)$ independent systems of block tridiagonal equations are computed. Each of these block tridiagonal systems of equations has $5(N_\xi - 2)$ unknowns. Similarly, in the η- and ζ-sweeps, $(N_\xi - 2)(N_\zeta - 2)$ and $(N_\xi - 2)(N_\eta - 2)$ independent systems of block tridiagonal equations are solved, respectively. In Steps *i*(a), *ii*(a), and *iii*(a) of Figure 1, coefficient matrices are set up before performing the actual block tridiagonal solutions. Since in each sweep there are multiple independent block tridiagonal systems, $\mathbf{M}_\xi$, $\mathbf{M}_\eta$, $\mathbf{M}_\zeta$ represent multiple coefficient matrices in those setup steps.

As mentioned earlier, benchmark BT has two standard data sets: Class A and Class B. For the Class A data set, $N = N_\xi = N_\eta = N_\zeta = 64$, and for the Class B data set, $N = N_\xi = N_\eta = N_\zeta = 102$. In both cases, Maxsteps (see Figure 1) is 200.

A sequential implementation

The starting point for our parallel implementation was the sequential FORTRAN 77 implementation of the BT benchmark (written by S. Weeratunga) provided by NASA Ames as an example implementation. We found this code to be very well written, and we will refer to this version as the *unoptimized starting-point implementation*, or USI for brevity. For our parallel implementation on the IBM SP architecture, we modified this code significantly to extract parallelism and to get good floating-point performance. In the following, we first describe some of the key features of USI and then describe the optimizations we made to this code. Some of these optimizations are obvious, and we would like to note here that although we point out the obvious optimizations, we do not imply that the developers of USI were unaware of them. In our opinion, the main objective of USI was to bring clarity, modularity, and simplicity to the benchmark.

The computationally intensive steps in each iteration of the BT benchmark are Steps *i*, *ii*, and *iii* shown in Figure 1. In each of these steps, multiple independent block (5 × 5) tridiagonal systems are solved, after setting up their coefficient matrices. In the ξ-sweep, the block tridiagonal system has the following structure:

$$
\begin{aligned}
&[B_{1,j,k}][\Delta \mathbf{U}_1]_{1,j,k} + [C_{1,j,k}][\Delta \mathbf{U}_1]_{2,j,k} = [\mathbf{RHS}]_{1,j,k} \\
&[A_{i,j,k}][\Delta \mathbf{U}_1]_{i-1,j,k} + [B_{i,j,k}][\Delta \mathbf{U}_1]_{i,j,k} \\
&\qquad + [C_{i,j,k}][\Delta \mathbf{U}_1]_{i+1,j,k} = [\mathbf{RHS}]_{i,j,k}; \\
&\qquad 2 \le i \le N_\xi - 1 \\
&[A_{N_\xi,j,k}][\Delta \mathbf{U}_1]_{N_\xi-1,j,k} + [B_{N_\xi,j,k}][\Delta \mathbf{U}_1]_{N_\xi,j,k} \\
&\qquad = [\mathbf{RHS}]_{N_\xi,j,k} \qquad (4)
\end{aligned}
$$

In the above, [A], [B], and [C] are 5 × 5 matrices, $[\Delta \mathbf{U}_1]_{i,j,k}$ is a 5 × 1 column vector, $[B_{1,j,k}]$ and $[A_{N_\xi,j,k}]$ are identity matrices ([**I**]), and $[C_{1,j,k}] = [0]$. At $\tau = n + 1$ and for $2 \le i \le (N_\xi - 1)$, the (5 × 5) coefficient matrices are determined as follows:

$$
\begin{aligned}
[A_{i,j,k}] = &\\
&-\Delta\tau\left\{-\frac{1}{2h_\xi}[\mathbf{A}(\mathbf{U}^n_{i-1,j,k})] + \frac{1}{h_\xi^2}[\mathbf{N}(\mathbf{U}^n_{i-1,j,k})]\right\} \\
[B_{i,j,k}] = &\ \mathbf{I} + \Delta\tau\frac{2}{h_\xi^2}[\mathbf{N}(\mathbf{U}^n_{i,j,k})] \\
[C_{i,j,k}] = &\\
&-\Delta\tau\left\{\frac{1}{2h_\xi}[\mathbf{A}(\mathbf{U}^n_{i-1,j,k})] + \frac{1}{h_\xi^2}[\mathbf{N}(\mathbf{U}^n_{i-1,j,k})]\right\} \qquad (5)
\end{aligned}
$$

where $h_\xi = 1/(N_\xi - 1)$ is the mesh size in ξ direction. The structures of the block tridiagonal systems involved in the η- and ζ-sweeps are analogous to the above-defined structure for ξ-sweep.

In USI, for each of the Steps *i*, *ii*, and *iii*, first all coefficient matrices are evaluated and then $(N - 2)^2$ systems of equations are solved (as shown in Figure 1). For the $N \times N \times N$ problem, each setup step involves computing and storing $3N(N - 2)^2$ coefficient matrices, each represented by a 5 × 5 array. For ξ-sweep, these matrices are evaluated as shown in Equations 5. Note that the coefficient computations are primarily scalar-matrix multiplications. Each linear system has a block tridiagonal structure, and hence, the coefficient matrix $\mathbf{M}_\xi$ of Step *i* in Figure 1 is represented not by a full matrix, but by an array of triplets of 5 × 5 coefficient matrices ([A], [B], [C]), one triplet for each 5 × 1 block of unknowns (i.e., at each interior grid point). After the setup, $(N - 2)^2$ linear systems are solved using a regular Gaussian elimination algorithm, with no pivoting. These solutions take into account the block tridiagonal nature of the linear systems—all forward eliminations are computed first, storing the intermediate results in the same work arrays as the ones used for computing the coefficient matrices in the setup step; this is followed by all back-substitutions for that sweep. The same procedure is repeated in the other two directions as indicated in Figure 1. For storage efficiency, the array representing the **RHS** vector is used to hold the intermediate solutions ($\Delta \mathbf{U}_1$, $\Delta \mathbf{U}_2$, and $\Delta \mathbf{U}^{\tau-1}$).

In Step *iv*, the solution is advanced to the next time step by performing a vector (5 × 1) add at each interior grid point. Finally, the computations for evaluating the **RHS** vector at the new time step are performed in Step *v*. These computations are

equivalent to three regular sparse block (5×5) matrix-vector multiplications.

Clearly, the main data dependencies in the computations of one iteration are those that characterize the solution of a block tridiagonal system of

We report performance on up to 128 nodes for both the SP1 and SP2-w systems.

equations and those in a regular sparse block matrix-vector multiplication. The solution of the tridiagonal system of equations involves a forward elimination phase followed by a back-substitution phase. In the forward elimination, the computations at block i depend on the values computed at block $i - 1$. In the back-substitution phase, the computations at block i depend on the values computed at block $i + 1$. The RHS computations at a grid point (i, j, k) require values of the **U** vector at (i, j, k) and at 12 neighboring grid points: $\{(i \pm l, j \pm l, k \pm l) | l = 0, 1, 2\}$, (i.e., along a 13-point stencil).

Finally, associated with each grid point, altogether a total of 90 words (double precision) of memory is required: five for the solution vector, **U**, five for the right-hand side vector, **RHS**, and 75 variables for storing the coefficient matrices (work arrays).

IBM SP architecture

In this section, we briefly describe some of the architectural details of the IBM SP1 and SP2 systems used in this study. The details presented here are relevant to the optimization steps we describe in the following sections. We refer the interested reader to References 10 and 11 for further architectural details on SP1 and to References 12 and 13 for further details on SP2.

Each processing element on the SP1 is IBM's RISC System/6000* (RS/6000*) Model 370 processor, with a 32K-byte data cache and 62.5 MHz clock speed. Each processing element on the SP2 considered in this study is IBM's RS/6000 Model 590 processor, with a 256K-byte data cache and 66.5 MHz clock speed. There are other SP2 systems with different types of processors, but we do not consider those in this study. To avoid any ambiguity, in the rest of the paper we refer to the SP2 system used in this study as SP2-w. (The suffix "w" stands for *wide*—the qualification used to describe the type of processing elements used.) The SP1 processor belongs to the POWER Architecture*, whereas the SP2 processor belongs to the POWER2 Architecture*. Readers interested in the details of various aspects of the POWER Architecture should refer to the articles that appear in Reference 14 or 15. Details on the various aspects of the POWER2 Architecture can be found in the articles that appear in Reference 16 or 17. The differences between the SP1 and SP2-w processors that are relevant to this study are: (1) in the former case, the fixed-point and the floating-point units have one execution unit each, whereas in the latter case, each has two execution units; (2) in the former case, up to two floating-point operations can be performed per cycle (125 MFLOPS, peak), and in the latter case, the same is four floating-point operations per cycle (266 MFLOPS, peak); (3) the latter has the ability to perform quad-word load or store of two adjacent double precision references to or from two adjacent floating-point registers in one cycle; (4) the memory bus width in the former case is eight bytes, and the same in the latter case is 32 bytes.

On both the SP1 and SP2-w, the processors are interconnected via a High-Performance Switch, the details of which can be found in Reference 11. The processor-switch interface is managed by a special adapter card. The communication adapter cards used in the SP1 and SP2-w systems differ significantly. For additional details, refer to Reference 13. On both systems, explicit message passing is the parallel programming paradigm. The SP1 provides two communication protocols: MPL and MPL/p. In all the experiments presented in this paper on the SP1, we used the MPL/p protocol. In our experiments on the SP2-w, message passing was handled by the Parallel Operating Environment (POE) using the Message-Passing Library (MPL) protocol.[18]

In this paper, for both the SP1 and SP2-w systems, we report performance on up to 128 nodes, with each node having 128 MB of local memory. For the SP1 experiments, we used the system at the IBM Thomas J. Watson Research Center in Yorktown

Heights, New York, and, for the SP2-w experiments, we used the system at the NASA Ames Research Center. All parallel programs for which we report performance in this paper were written in the single-program, multiple-data (SPMD) style of programming in FORTRAN, using IBM's external user interface (EUI)[19] as the message-passing interface.

For the MPL/p protocol used in the SP1 experiments, the point-to-point message latency was measured (using ping-pong type point-to-point communication between pairs of processors) to be 30 μsec. The effective communication bandwidth under this protocol was measured to approach 8.5 MB/sec, asymptotically. For further details on the SP1 communication performance, please see Reference 20. For the SP2-w, where we used the MPL protocol, the message latency was observed to be 42 μsec, and the effective communication bandwidth approached 34 MB/sec, asymptotically, for unidirectional communication and 48 MB/sec (asymptotic) for bidirectional (exchange type) communication. For additional details on the SP2 communication software, please see Reference 18.

Optimizations for single processor performance

In achieving good performance on a parallel system, obtaining good single processor performance is an important step. However, single processor performance should not be obtained at the expense of available parallelism. In our implementation of the benchmark BT on the IBM SP systems, we made significant modifications to USI and optimized the code for the RS/6000 POWER (for SP1) and POWER2 (for SP2-w) architectures, without sacrificing the available parallelism. Here we describe some of these modifications. The parallel implementations of the BT benchmark for the SP1 and SP2-w differ only in the optimizations for single processor performance.

Computation of coefficient matrices. In USI, all the coefficient matrices are set up before performing the block tridiagonal solutions. As is evident from Equations 5, a significant amount of time is spent in setting up these matrices. We found two disadvantages with this approach: (1) large work arrays are necessary (75 words per grid point) and (2) locality in computations is reduced considerably when the block tridiagonal solutions are performed along directions involving strides (for example, in η- and ζ-sweeps). To overcome these difficulties, in each implicit sweep we merged the coefficient computation step with the solver step (e.g., Step i(a) and Step i(b) in Figure 1). The first advantage of this modification is the reduced memory requirement. Instead of 75 words of memory at each grid point in the entire three-dimensional domain, 75 words of memory per grid point along a one-dimensional line of grid points (corresponding to a single tridiagonal solution) are needed. This work space is reused in the subsequent solutions. The main performance benefit from this modification is that in performing the tridiagonal solutions in the η- and ζ-sweeps, the coefficient matrices can be accessed with single stride, thus, significantly reducing cache misses.

In achieving good performance on a parallel system, obtaining good single processor performance is an important step.

In addition to the above modification, we obtained performance improvements in the computations of the flux Jacobians (e.g., in computing $1/(2h_\xi)[\mathbf{A}(\mathbf{U}^n_{i-1,j,k})]$ and $(1/h_\xi^2)[\mathbf{N}(\mathbf{U}^n_{i-1,j,k})]$ of Equations 5). Performance was improved by carefully rearranging the computations so as to eliminate any duplications and also obtaining good register reuse. In the case of the implementation for the POWER2 Architecture (of the SP2-w nodes), further tuning was necessary to assist the compiler in maximizing the instruction-level parallelism to keep the dual arithmetic units of the floating-point unit (FPU) busy simultaneously and to take advantage of quad-load or store capabilities. Primarily, this tuning involved exposing computations for two independent grid points at the same time (but without exceeding the limitations imposed by the available number of registers or the cache size). Note that all our implementations are of single-program, multiple-data (SPMD) style, which means that the code remains the same for all nodes. Since not all processors may end up with same-size partitions (due to load imbalance), the number of grid points

assigned to processors may not all be the same. As a result, proper care must be taken in implementing the above-described optimizations.

Solution of block tridiagonal systems. We replaced the generic Gaussian elimination (with no pivoting) algorithm used in USI by the Thomas algorithm. The Thomas algorithm is a special case of Gaussian elimination and is used to obtain an efficient solution to tridiagonal and pentadiagonal systems, when no pivoting is involved.[21] To explain this algorithm, consider the following ith equation $2 \leq i \leq N_\xi - 1$, in a block tridiagonal system given by Equations 4 for some values of j and k:

$$[A_i][\mathbf{u}_{i-1}] + [B_i][\mathbf{u}_i] + [C_i][\mathbf{u}_{i+1}] = [\mathbf{r}_i] \tag{6}$$

where $[A_i]$, $[B_i]$, and $[C_i]$ are 5×5 matrices, and $[\mathbf{u}_i]$ and $[\mathbf{r}_i]$ are 5×1 column vectors. For convenience, in the above equation we have dropped j and k suffixes. In a conventional Gaussian elimination (with no pivoting) algorithm, at the ith step of the forward elimination phase, the following computations are performed:

$$\begin{aligned}
[Q_i] &= [X_{i-1}][C_{i-1}] \\
[\bar{B}_i] &= [B_i] - [A_i][Q_i] \\
[X_i] &= [\bar{B}_i]^{-1} \\
[\bar{\mathbf{r}}_i] &= [X_{i-1}][\mathbf{p}_{i-1}] \\
[\mathbf{p}_i] &= [\mathbf{r}_i] - [A_i][\bar{\mathbf{r}}_i]
\end{aligned} \tag{7}$$

and in the back-substitution phase the following computations take place:

$$\begin{aligned}
[\bar{\mathbf{p}}_i] &= [\mathbf{p}_i] - [C_i][\mathbf{u}_{i+1}] \\
[\mathbf{u}_i] &= [X_i][\bar{\mathbf{p}}_i]
\end{aligned} \tag{8}$$

where $[\bar{B}_i]$, $[Q_i]$, $[X_i]$ are 5×5 matrices, and $[\mathbf{p}_i]$, $[\bar{\mathbf{p}}_i]$, and $[\bar{\mathbf{r}}_i]$ are 5×1 column vectors. When $[X_1]$, $[\mathbf{p}_1]$, and $[\mathbf{u}_{N_\xi}]$ are known, the solution to the block tridiagonal system is determined.

In the Thomas algorithm, the block tridiagonal system represented by Equations 4 is rearranged so that the ith equation has the following form:

$$[\mathbf{u}_i] = [\mathbf{p}_i] - [Q_i][\mathbf{u}_{i+1}] \tag{9}$$

Thus, at the ith step of the forward elimination phase, the following computations are performed:

$$\begin{aligned}
[\bar{B}_i] &= [B_i] - [A_i][Q_{i-1}] \\
[X_i] &= [\bar{B}_i]^{-1} \\
[Q_i] &= [X_i][C_i] \\
[\bar{\mathbf{r}}_i] &= [\mathbf{r}_i] - [A_i][\mathbf{p}_{i-1}] \\
[\mathbf{p}_i] &= [X_i][\bar{\mathbf{r}}_i]
\end{aligned} \tag{10}$$

and, in the back-substitution phase, the following computations take place:

$$[\mathbf{u}_i] = [\mathbf{p}_i] - [Q_i][\mathbf{u}_{i+1}] \tag{11}$$

where $[\bar{B}_i]$, $[Q_i]$, $[X_i]$ are 5×5 matrices and $[\mathbf{p}_i]$, and $[\bar{\mathbf{r}}_i]$ are 5×1 column vectors. When $[Q_1]$, $[\mathbf{p}_1]$, and $[\mathbf{u}_{N_\xi}]$ are known, the solution to the block tridiagonal system is determined.

Comparing the steps in the conventional Gaussian elimination (Equations 7 and Equations 8) with those in the Thomas algorithm (Equations 10 and Equation 11), we notice that in the former case the information is carried forward in the $[X]$ matrix and $[\mathbf{p}]$ vector, whereas in the latter case, this is accomplished by matrix $[Q]$ and vector $[\mathbf{p}]$. Moreover, when we analyze the above-described Gaussian elimination algorithm, we find that at the ith step (both forward elimination and back-substitution phases combined), there are seven pairs of triangular "solves" (counting a lower triangular solve followed by an upper triangular solve as one pair) involving a 5×5 matrix and a 5×1 vector, whereas in the Thomas algorithm only six pairs of triangular solves are performed. (Notice that in the back-substitution phase of the Thomas algorithm, no triangular solves are performed.) In addition, both algorithms perform one matrix-matrix multiply-and-add, two matrix-vector multiply-and-adds, and one factoring of a matrix. (All matrices are 5×5, and all vectors are 5×1.) Thus, by using the Thomas algorithm, we avoid performing N pairs of triangular solves (a lower triangular solve plus an upper triangular solve) in the solution of each block tridiagonal system of equations. Recall that altogether $3N^2$ such systems are computed in advancing the solution by one time step. A lower and upper triangular pair involves 20 multiply-and-add operations (daxpy type) and five divides.

In addition to reducing the operation count, the Thomas algorithm offers somewhat better locality of computations. For example, notice from Equations 10 that immediately after $[X_i]$ is factored, it is used in the ten triangular solves. In the Gaussian elimination, however, the ten triangular solves use $[X_{i-1}]$ from the previous step. Such improvements in the locality of computations have a significant effect on the performance of cache-based RISC processors.

On the RS/6000 (as with many other microprocessors), floating-point reciprocals and divisions are significantly more expensive than floating-point multiplications. Keeping this in mind, we further improved the performance of each block tridiagonal solution by reducing the number of floating-point reciprocal and division operations, at the expense of an increased number of floating-point multiplications. Each upper triangular solve involves five divisions. However, in all six upper triangular solves performed in step i of the Thomas algorithm (see Equations 10), the same diagonal elements of $[X_i]$ are involved. Moreover, in the matrix factoring step where $[X_i]$ is computed, the same reciprocals are needed. When the 5×5 matrix is factored, we store the reciprocals of the diagonal elements of $[X_i]$ instead of the "normal" values, and use these values in all upper triangular solves involving $[X_i]$. Thus, in the ith step of a block tridiagonal solution, we replaced 30 divisions by that many multiplications. Note that this optimization is possible even when regular Gaussian elimination is used (as in USI).

Optimization of RHS computations. After the computations in the implicit sweeps, computations of **RHS** are the next most expensive to evaluate. These computations correspond to Step v in Figure 1. We made modifications to USI to improve the single processor performance in this part of the code as well. Most of these changes consisted of fine tuning of the computations so as to reduce the number of operations, especially floating-point divisions, improving the register reuse, and by exploiting the POWER2 Architecture (in case of the SP2-W). In some cases, the cost of floating-point divisions was minimized at the expense of additional memory usage by computing the reciprocal once and using it as a multiplier in more than one operation. We made a few significant changes in the loop nesting to improve cache performance. We outline that modification below.

For each time step, RHS is evaluated in three parts. In the first part, ξ-direction flux differences are computed and to these, fourth-order dissipation terms in ξ direction are added. In the second and the third parts, similar computations for η and ζ directions are performed. In computing the RHS components in the η or ζ direction, memory access to the data structure for the **U** vector involves nonunit strides, which gives rise to a high rate of cache misses. We mitigate this situation by rearranging the nested loop structure in these parts of the computations. In performing the η-direction computations, in USI, the innermost loop is first over the second dimension of the main arrays involved and then over the first dimension. We rearranged these so the innermost loop is over the first dimension. The net effect is that the flux differences are now computed over a plane of grid points (ξ-η plane) at a time, instead of one grid point at a time. This reduces the cache misses to a large extent at the expense of having larger work arrays. Similarly, in the third part, where ζ-direction contributions are computed, the original nesting of loops is such that the third dimension of the main arrays is varied in the innermost loop. To improve the cache misses in this part, we rearranged the loops so the innermost loop is over the first dimension and the next level of looping is over the third dimension of the arrays.

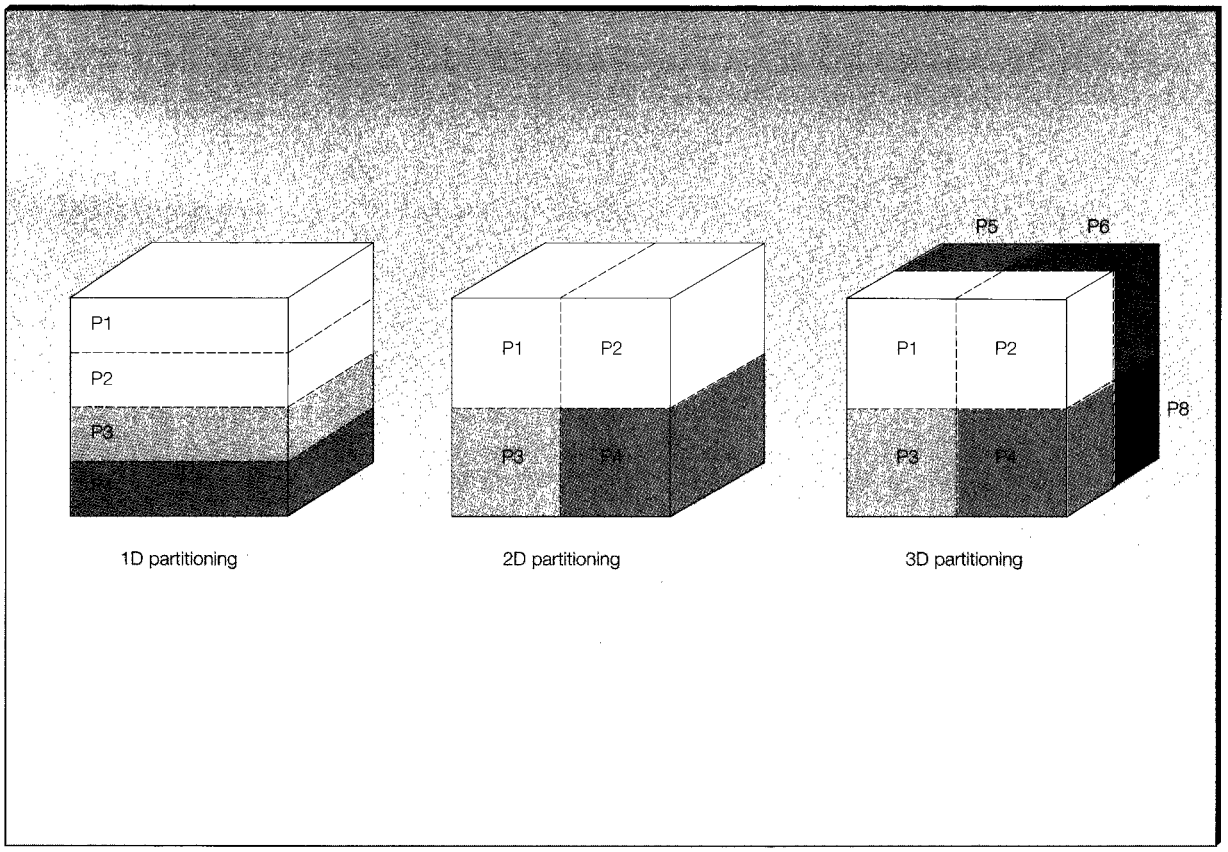
Parallel implementation

For the parallel implementation of the BT benchmark, we use SPMD style of programming and take advantage of the available data parallelism. In the following discussion, we assume $N \times N \times N$ to be the size of the grid.

Data partitioning strategies. As noted earlier, the computationally intensive sections of the BT benchmark are Steps *i*, *ii*, *iii*, and *v* shown in Figure 1. For this reason and because these steps encapsulate important data dependencies that affect scalability, we primarily focus on the performance of these steps in our parallel implementation.

We considered three grid-level partitioning schemes that are suitable for parallelizing the computations: 1D unipartitioning, 2D unipartitioning, and 3D unipartitioning. These three are shown schematically in Figure 2. In each case, a single contiguous partition of grid points is assigned to each processor, and hence we refer to this type of partitioning as unipartitioning. In 1D unipartitioning,

Figure 2 Three block-based partitioning schemes



the grid is partitioned along one of the three spatial dimensions; under 2D unipartitioning, the grid is partitioned along two of the three spatial dimensions; and it is partitioned along all three dimensions in the 3D unipartitioning case. In each case, the associated arrays are distributed along the direction of grid partitioning. These partitioning schemes accomplish the same array distributions as the block distribution constructs of High Performance FORTRAN (HPF). In Reference 22, we consider another class partitioning scheme called multipartitioning which, in HPF terminology, is a type of block-cyclic partitioning scheme. We do not consider multipartitioning schemes in this paper. We refer the interested reader to References 23–25 for a detailed analysis and discussion on the communication, load imbalance, and scalability properties of these partitioning schemes in the context of CFD applications. For the purpose of this discussion, we note the following points for these partitioning schemes. Along the dimensions where partitions are made, there are data dependencies across processors that affect the solution of block tridiagonal systems of equations and the evaluation of RHS vectors. Thus, values must be communicated among processors during forward elimination and back-substitution phases. Also, at the end of the update computations in Step *iv* of Figure 1, the updated values at the partition boundaries must be communicated among (logically) neighboring processors so that the computations of RHS in Step *v* can be completed. Note that the communication during the forward elimination or back-substitution phases is one-directional, whereas the communication after Step *iv* can take place in an exchange fashion. With the 1D partitioning, each processor communicates with at most two other processors. However, the adjacent surface area

Table 2 Effect of data distribution methods on performance (Class A problem set and the SP1 system)

No. of PEs	1D		2D		3D	
	Partitioning	Time (sec.)	Partitioning	Time (sec.)	Partitioning	Time (sec.)
8	1 × 1 × 8	589.7	1 × 2 × 4	504.5	2 × 2 × 2	446.3
16	1 × 1 × 16	350.0	1 × 4 × 4	298.1	2 × 2 × 4	250.7
32	1 × 1 × 32	219.3	1 × 4 × 8	182.2	2 × 4 × 4	143.8

among partitions does not reduce when more partitions are made. Thus, this partitioning is the least scalable of all.

In Table 2, we compare the performance of the three partitioning schemes on an 8-, 16-, and 32-node SP1 using Class A problem set. The "Partitioning" columns indicate the data-partitioning strategy used. The first number indicates the number of partitions to be made along the first spatial array dimension (i.e., along the ξ direction), and so on. Thus, for example, with eight processors and under 1D partitioning, eight slices were made along the third spatial dimension of the distributed arrays (the slowest varying dimension, which is along ζ direction) and no partitions were made along the other two spatial dimensions. Note that the same code was run on all platforms, and in all cases the same communication optimizations described below were included. Clearly, the 3D partitioning has the superior performance among all three partitioning methods.

Another important decision in data partitioning is that of the number of partitions to make along each dimension of partitioning. For example, with 16 processors and 3D partitioning, one may partition the grid into $2 \times 2 \times 4$ or $4 \times 2 \times 2$ or as $2 \times 4 \times 2$. This has the effect on the locality of computations within a processor. Although we do not show the results here, significant performance gains are realized by partitioning along the slowest varying dimension than along the fastest varying dimension.

We used the 3D partitioning in all the experiments reported in the rest of this paper. Whenever the number of processors, P, is not a perfect cube of an integer, we factor the number P into three integers that are relatively close to $\sqrt[3]{P}$ and partition the grid so that the fastest varying dimension (ξ in the current example) has the least number of partitions and the slowest varying dimension (ζ) has the most number of partitions.

Parallel block tridiagonal solutions. As described in the previous section, we used the Thomas algorithm for performing the block tridiagonal solutions. As with the generic Gaussian elimination, the Thomas algorithm is highly sequential when used for solving a single block tridiagonal system. However, under the 3D data-partitioning scheme, the multiple independent systems ($(N-2)^2$ in our example) in each implicit sweep can be computed in parallel. This parallelism across multiple solutions helps in mitigating the adverse performance effects of using the Thomas algorithm on multiple processors. In the ξ direction, we first complete the forward sweep for all $(N-2)^2$ systems and then perform the backward sweeps. Following this, the same procedure is repeated along η and ζ directions. To improve single processor computational performance and to reduce communication overhead, in the forward elimination phase we bundle together the $[Q_i]$ matrix (5×5) and $[\mathbf{p}_i]$ vector (5×1) corresponding to the last grid point of a line segment assigned to a processor.

In summary, in the implicit computations along the ξ direction (and analogously, along η and ζ directions), each processor sweeps over all the segments of grid lines in its block partition, first in the forward direction and then in the backward direction. In the forward sweep over a segment of grid lines, the coefficient matrices are computed on the fly, the intermediate values needed in the forward elimination phase are received from the neighboring processor (that performs the forward elimination on the earlier part of the grid-line), the forward elimination computations on that line segment are completed, and then the intermediate values computed at the last grid point are sent off to the processor performing the next segment of the same line of grid points. Following this, the same is repeated over the next segment of grid lines in that partition. After completing the forward sweep over all the grid lines, the backward sweep is performed in a similar fashion. In the backward sweep, first

the values of the $[\mathbf{u}_{i+1}]$ are received for each line segment from the neighboring processor and, at end of the computations over that line segment, values of vector $[\mathbf{u}_k]$ are sent to the neighboring processor in the direction of the backward sweep (k is the first grid point of the line segment assigned to a processor). This completes the implicit computations in the ξ direction. The procedures for the implicit computations along the η and ζ direction are analogous.

The communication costs of the parallel implementation are as follows. In the forward sweep over each segment of grid points, a processor requires 30 words of information from a neighboring processor (unless the segment corresponds to the beginning part of the block tridiagonal system), and that processor sends away 30 words of information to the processor working on the subsequent segment of that line of grid points. Similarly, in the backward sweep over each segment of grid line, a processor receives five words of information and sends away five words of information, at the beginning and at the end of the back sweep over a grid-line, respectively. In the forward and back sweeps, only one message needs to be sent and received per grid-line segment.

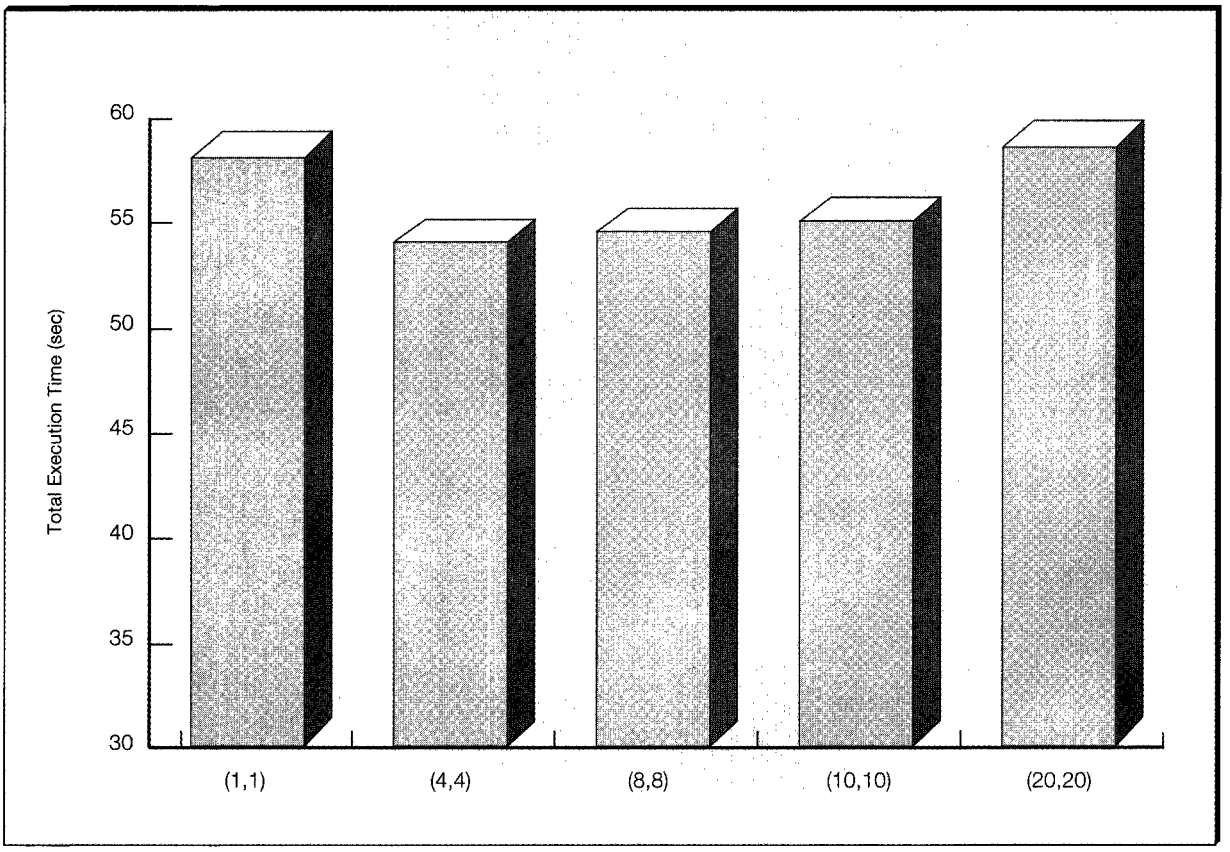
When the information is sent and received in this manner, the total number of messages sent or received by a processor is quite large. For example, if $n_x \times n_y \times n_z$ are the dimensions of the block of grid partition assigned to a processor, then in the implicit part of the computations in each iteration, altogether a processor would send or receive $2n_xn_y + 2n_yn_z + 2n_xn_z$ messages. When the message latency costs are relatively high as compared to the CPU speed, this overhead turns out to be significant. Hence, to further reduce the number of messages sent, we combine messages from several forward or backward sweeps together and send that out as one message. We refer to this as *coalescing of line solves*. If ℓ lines are coalesced, the total number of messages communicated in the implicit computations is reduced by a factor of ℓ. However, such a coalescing of line solves increases the number of line solves being serialized by the same factor. Thus, there is a trade-off between reducing the number of messages and maintaining adequate parallelism. In Reference 22, the performance effects of this trade-off are analyzed in detail. Here we present experimental data to support that analysis.

In Figure 3, we show the variations in the total execution time (for Class A problem set) on a 128-node SP1 as the number of lines coalesced together is changed. The pair (x, y) under each bar indicates the number of lines coalesced in the forward (x) and the backward (y) sweeps. Thus, on the 128-processor SP1, the minimum execution time is obtained by sending one message in the forward sweep after four line segments are computed and by sending one message in the backward sweep after four line segments are computed. The total execution time drops from 55.8 seconds (1,1) to 48.0 seconds (4,4) (a gain of 16 percent). Note also that the total execution time drops quickly in the beginning as the number of lines coalesced is increased from one, and after the minimum is reached it rises gradually. Thus, it is not necessary to get the exact optimum values for the number of lines to coalesce together; approximate values are sufficient to obtain good performance. We conducted similar experiments on other processor configurations. We found that for 64 and 128 processors, coalescing four lines gives the best performance, and on 8, 16, and 32 processors, coalescing eight lines together gave the best performance. In general, the higher the processor speed compared to the latency, the larger the number of line solves that need to be coalesced to minimize the total execution time. For exact relations, which involve several other parameters, including the number of grid points, number of partitions, number of operations at each grid point, and bandwidth, please refer to Reference 22.

Performance results

In the previous section, we presented some of the highlights of our parallel implementation of the BT benchmark. In this section, we summarize the current best performance results for the BT benchmark using the two standard size problems (i.e., Class A and Class B) on the SP1 and SP2-w. We performed all our SP1 measurements on the 128-node SP1 system at the IBM T. J. Watson Research Center. All our SP2-w measurements were performed on the 128-node SP2 system at the NASA Ames Research Center. For comparison, in our results we have included the Cray Y-MP and Cray C90 performance on these benchmarks.[7] For a compiled list of the performance of various other parallel systems on this benchmark, please refer to References 3 and 26.

Figure 3 Effect of number of lines coalesced together on the performance of a 128-node SP1



The performance of the BT benchmark using the Class A problem set is shown in Table 3, and the Class B results are shown in Table 4. In both cases, the performance is reported on 8 through 128 processors, except for the Class B problem set on the SP1 for which the performance is reported on 16 through 128 processors. In the case of Class A, we compare the SP1 and SP2-w performance with Y-MP/1 performance, and in the case of Class B, we compare the performance with that of a single processor C90. The Y-MP and C90 performance figures are from Reference 3. Note that on the Class A problem set, the 128-node SP2-w is able to deliver performance that is over 39 times higher than that of a single Y-MP node and over 5.67 times higher than the performance of an eight-node Y-MP. Similarly, on the Class B problem set, the 128-node SP2-w is able to deliver performance that is close to 19 times the performance of a single node C90 and close to 1.5 times the performance of a 16-node C90.

In Tables 3 and 4, we also report the MFLOPS delivered by the SP1 and SP2-w on the two problem sets. To determine the true (and meaningful) MFLOPS delivered by a system, it is necessary to accurately measure the number of actual floating-point operations performed by the hardware and that contributed directly toward the solution of the problem. A simple count of the floating-point operations in the code is far from accurate. Similarly, counting the floating-point operations in the assembly code has difficulties. To overcome some of these difficulties, we have used the floating-point operation counts for the BT benchmark as reported in Reference 7. These counts were obtained for the benchmarks tuned for the Cray systems and were measured using the performance monitoring hardware on the corresponding Cray systems (Cray Y-MP count for the Class A problem and C90 count for the Class B problem). Using these operation counts and the observed execution times on the various SP1 and SP2-w configurations, we have

Table 3 Performance comparisons for benchmark BT (Class A problem set)

System	No. of Processors	Time for 200 Iterations (sec.)	Ratio to Y-MP/1	Equivalent Y-MP/1 (MFLOPS)
Y-MP	1	792.4	1.00	229.0
	8	114.0	6.95	1,591.6
C90	1	356.9	2.22	508.4
	16	28.4	27.91	6,391.4
SP1/(MPL/p)	8	446.3	1.78	406.6
	16	250.7	3.16	723.8
	32	143.8	5.51	1,261.9
	64	83.4	9.50	2,175.8
	128	48.0	16.49	3,777.3
SP2-w/(MPL)	8	206.7	3.83	877.9
	16	112.9	7.02	1,607.3
	32	61.8	12.82	2,936.2
	64	34.7	22.84	5,229.4
	128	20.1	39.42	9,027.8

Table 4 Performance comparisons for benchmark BT (Class B problem set)

System	No. of Processors	Time for 200 Iterations (sec.)	Ratio to C-90/1	Equivalent C-90/1 (MFLOPS)
C90	1	1,261.4	1.00	572.0
	16	96.4	13.10	7,484.4
SP1/(MPL/p)	16	993.5	1.27	726.2
	32	514.5	2.45	1,402.3
	64	275.6	4.58	2,617.9
	128	154.3	8.17	4,676.0
SP2-w/(MPL)	8	862.8	1.46	836.3
	16	440.6	2.86	1,637.6
	32	226.8	5.56	3,181.3
	64	119.2	10.58	6,053.0
	128	67.0	18.83	10,769.0

reported the MFLOPS as the equivalent Y-MP and C90 MFLOPS in Tables 3 and 4, respectively. We note here that since our implementations of the BT benchmark were done independent of the implementations on the Cray systems, the MFLOPS we report may not be true RS/6000 MFLOPS. However, since the same problem is solved in both cases, the equivalent MFLOPS as computed here are a good measure for comparing two different systems. Using this measure, we observe that on the 128-node SP2-w system, we are able to realize over 10 GFLOPS (billions of floating-point operations per second) on the Class B problem set.

Scalability of the SP architecture

One objective in any benchmarking study is to be able to systematically compare the performance (and, more generally, some specific characteristics of a system) of dissimilar architectures in performing some standard set of computations. The results presented in the previous section help toward meeting that objective for the BT benchmark. Another objective for scalable architectures such as the SP series is to quantify the scalability of the system. Architectural scalability may be defined in various ways. For this study, we use a restrictive

definition of it as the ability to sustain the performance characteristics with the incremental changes in the system resources such as the number of processors, memory (DRAM), and the communication subsystem. Ideally, architectural scalability should be quantified independent of any application-specific characteristics. However, without performing any meaningful computations, it is almost impossible to assess the architectural scalability of the system as a whole. The approach we have taken here is that of quantifying the architectural scalability for a given type of computations—specifically, for the BT type of computations. Results presented in this section should help in determining, qualitatively if not quantitatively, the consistency of the system performance characteristics when more processors are added to the system and problems are solved without making any changes to the application implementation (but possibly by varying the problem sizes to fully utilize available resources). The results are more representative of the computations similar to those in the BT benchmark. However, some generalizations can be made. We make additional comments on this in the concluding section.

To quantify the architectural scalability for a given type of computations we define a new term called *scalability factor*. If a problem with $w(p_1)$ amount of work is computed in time $t(p_1)$ on p_1 processors and a similar problem with $w(p_2)$ amount of work is computed in time $t(p_2)$ on p_2 processors, the scalability factor, σ, for the system with p_2 processors with respect to (w.r.t.) the system with p_1 processors is given by

$$\sigma(p_2, p_1) = \frac{t(p_1)}{t(p_2)} \times \frac{w(p_2)}{w(p_1)} \times \frac{p_1}{p_2}$$

The scalability factor, $\sigma(p_2, p_1)$, is a relative efficiency measure weighted by the problem sizes. It indicates the efficiency of a system with p_2 processors w.r.t. a system with p_1 processors. When p_1 is 1 and $w(p_1)$ and $w(p_2)$ are the same, σ gives the efficiency of the parallel system with p_2 processors. When p_1 is smaller than p_2, $w(p_1)$ is at most equal to $w(p_2)$, and the load distributions are identical, then $\sigma(p_2, p_1)$ is at most 1, and it is desired to be close to 1. Note that when one is interested in the scalability of the architecture, the application-specific effects such as the degree of parallelism (or sequentiality) inherent to an implementation or the effects of the data-partitioning strategies used must be eliminated. Although it is impossible to completely eliminate some of these effects, by using the same algorithms, the same partitioning strategies, and the same implementation methods to solve different size problems, one can gain enough information about the architectural scalability. In our experiments, the implementation of the BT benchmark is such that the parallelization algorithms remain the same, and the data-partitioning strategies are parameterized by the number of processors and the problem size. Thus, we could use the same implementation even when processor configurations or the problem sizes were changed. To avoid (as far as possible) variation in the cache behavior because of change in the array sizes, we used array sizes to appropriately fit the corresponding partition size.

We studied the scalability of the SP architecture using two reference point performance results: one with a 16-processor system and the other with a 32-processor system. We considered four problem sizes: $64 \times 64 \times 64$, $80 \times 80 \times 80$, $102 \times 102 \times 102$, and $126 \times 126 \times 126$. Recall that the $64 \times 64 \times 64$ and $102 \times 102 \times 102$ problem sizes are specified in the NAS parallel benchmark suite. For the $80 \times 80 \times 80$ problem we used a time step of 0.0005, and for the $126 \times 126 \times 126$ problem we chose a time step of 0.00005; in each case we performed 200 iterations, just as in the standard problem sizes. (Note that for the $64 \times 64 \times 64$ and the $102 \times 102 \times 102$ problems, the prescribed time steps are 0.0008 and 0.0003, respectively. Note further that, under "normal" conditions, the size of the time step does not affect the total number of floating-point operations performed, which is determined by the number of iterations. However, an appropriate time step value should be chosen for numerical stability.)

Determining the scalability factor using two different problem sizes requires an estimate of the ratio of the work associated with the two problem sizes. In general, for the CFD problems, such as those represented by the BT benchmark, the work is proportional to the number of grid points. However, because of the complex nature of the computations, the amount of work is not exactly linear in the number of grid points. To simplify the issue, we assumed the time to solve a problem on one processor to be representative of the work associated with that problem regardless of the number of processors used to solve that problem. Speedups and efficiency measures are typically based on

Table 5 Scalability of SP1 on benchmark BT

No. of PEs	Execution Time for 200 Iterations (sec.)				Scalability w.r.t. 64 × 64 × 64 on	
	64^3	80^3	102^3	126^3	16 PEs	32 PEs
16	**250.7**	485.6	993.5	—	**1.000**	—
32	*143.8*	**267.5**	514.5	970.3	**0.965**	*1.000*
64	83.4	*149.6*	**275.6**	500.8	**0.976**	*0.990*
128	48.0	84.1	*154.3*	**273.3**	**0.987**	*1.000*

Table 6 Scalability of SP2-w on benchmark BT

No. of PEs	Execution Time for 200 Iterations (sec.)				Scalability w.r.t. 64 × 64 × 64 on	
	64^3	80^3	102^3	126^3	16 PEs	32 PEs
16	**112.9**	217.9	440.6	838.1	**1.000**	—
32	*61.8*	**116.8**	226.8	429.5	**0.996**	*1.000*
64	34.7	*63.6*	**119.1**	220.3	**1.017**	*1.001*
128	20.1	35.6	*67.0*	**119.8**	**1.014**	*0.989*

such an assumption. The difference in our case is that we did not necessarily use the best possible algorithms or implementation techniques suitable for the single processor case (since our aim was to achieve best possible parallel performance). Instead, the same parallel implementation of the BT benchmark, but without any communication constructs, was used for measuring the execution time on one processor (with suitable changes in the data array sizes).

The memory requirements of the single processor executables get very large; e.g., 92.8 MB, 174.7 MB, 360.6 MB, and 677.9 MB are the memory requirements of the four problem sizes considered. To avoid paging effects on the execution time, we ran these one-processor experiments on one of the processors of a specially configured two-node SP2-w system, which was identical in all aspects to the larger counterparts, except that each node had 1-GB main memory.

Another difficulty in these single processor runs was the amount of time required to complete each run, which was prohibitively long. Since we only had a limited amount of machine time available, we ran each of the four problems for only 20 iterations instead of the full 200 iterations. Since the same set of computations are performed in each iteration, we found that running the benchmark for only 20 iterations and linearly projecting the observed time to 200 iterations was reasonable. (This truncation to 20 iterations was done only for the single processor runs; all other runs for which the results are reported in this paper were carried out for 200 iterations.) The single processor execution times for the four problem sizes—64 × 64 × 64, 80 × 80 × 80, 102 × 102 × 102, and 126 × 126 × 126—were respectively found to be 1606, 3307, 6893, and 13 830 seconds. Thus, the ratios of the work associated with the four problem sizes turn out to be 1:2.1:4.3:8.6. The scalability factors for the SP1 and SP2-w can be computed using these work ratios and the execution times of the four data sets.

The performance of the SP1 and SP2-w on the four data sets is shown in Tables 5 and 6, respectively. In these tables, we highlight scalability factors for 32-, 64-, and 128-processor systems with respect to a 16-processor system, using the 80 × 80 × 80, 102 × 102 × 102, and 126 × 126 × 126 problem sizes, respectively. (See the numbers in **bold font**.) We also highlight scalability factors for 64-, and 128-processor systems with respect to a 32-processor system, using the 80 × 80 × 80, 102 × 102 × 102 problem sizes, respectively. (See the numbers in *italic font*.) In each of these two cases, the data partition size per processor remains the same; i.e., the problem size is doubled when the number of

processors is doubled. In the first case, the memory requirements per processor for the four processor partition sizes are: 9.7 MB (16 processors), 10.7 MB (32 processors), 10.2 MB (64 processors), and 10.6 MB (128 processors). In the latter case (where scalability is computed with respect to a 32-processor system), the memory requirements per processor are: 6.5 MB (32 processors), 5.9 MB (64 processors), and 6.6 MB (128 processors). Thus, the selected problem sizes assured no change in the memory utilization per processor.

We observe from the scalability factors presented in Tables 5 and 6 that the SP architecture scales well on the BT type of computations when the memory utilization is maintained at the same level. Note that this outcome is obtained despite the fact that the memory utilization on each processor is less than 10 percent of the available memory (which was 128 MB per node for both the SP1 and SP2-w). In some cases, the SP2-w architecture seems to scale superlinearly; i.e., the scalability factors are greater than 1.0. However, this is because of the load imbalance effects that we could not completely eliminate from our experiments. (In the BT benchmark, the $64 \times 64 \times 64$ grid essentially involves working on a $62 \times 62 \times 62$ grid, whereas the $102 \times 102 \times 102$ grid involves working on a $100 \times 100 \times 100$ grid. Thus, the severity of load imbalance in computing the $64 \times 64 \times 64$ size problem on 16 processors is more than that in computing the $102 \times 102 \times 102$ size problem on 64 processors.)

In the above, we considered the scalability when the memory utilization remained unchanged. Scalability factors, when memory utilization is varied, can also be computed from the information in Tables 5 and 6 and other information presented earlier in this section. In particular, by considering the execution times in the same column (i.e., the execution times for the same problem size), one can compute relative speedups and efficiencies. Consider two extreme cases: $64 \times 64 \times 64$ and $126 \times 126 \times 126$. On the SP2-w, in the former case, the scalability factor for the 128 processors w.r.t. 16 processors is 0.70 and, in the latter case, the same is 0.87. Again both of these figures are quite respectable considering the fact that in the former case only 2.7 MB memory per node was required when the problem was solved on 128 processors and, in the latter case, the same was 10.6 MB per node.

Conclusions

In this paper, we have described an efficient and scalable method for implementing the BT benchmark on distributed memory systems. Using the strategies described here, we have implemented this pseudo-application benchmark on the IBM SP systems (the SP1 and the SP2 with wide nodes) and have presented performance results on up to 128 processors. The experimental results indicate that the SP architecture delivers good performance on this benchmark, both in terms of raw performance and scalability. To get the level of performance that we obtained, we used a combination of techniques that included the use of efficient sequential algorithms, the use of scalable partitioning strategies, the use of algorithms to reduce the number of messages, the use of improved data structures to reduce memory requirements and memory references, and some tuning for high cache and register utilization. We concentrated our efforts primarily in three areas: (1) developing a good sequential implementation, (2) using appropriate data distribution methods, and (3) reducing the number of messages communicated where possible. A noteworthy aspect of this study is that we realized good performance on both the SP1 and SP2-w systems using implementation methods that are identical as far as the parallelization techniques, parallel algorithms, and data partitioning strategies are concerned. In terms of the parallel implementation effort, these issues encompass the bulk of the development and programming effort. The implementations for the SP1 and SP2-w differed only in the single processor optimization techniques used in the computationally intensive solver sections. This is an important point since it shows that scalability is achievable even at the software level and that it is possible to reuse code without sacrificing performance, which is critical in the development of large, complex application software.

Obtaining good single processor performance is an important part of parallelizing an application for performance. To realize the significance of single processor performance, note that when we compiled and ran USI without any modifications (but using all possible compiler optimization options) on a Class A data set, the overall performance was about 20 MFLOPS on a single processor of the SP1. After the modifications described in this paper, we managed to raise the single processor performance to about 53 MFLOPS. To obtain good performance on a single processor, we carefully analyzed the

computationally intensive parts of the benchmark and replaced the conventional Gaussian elimination by the Thomas algorithm to reduce the operation count. We performed loop level optimizations and rearranged individual segments of computations so as to keep the dual arithmetic units (ALU) in the floating-point unit (FPU) simultaneously busy (in the case of the SP2 processor). Optimizations were also incorporated to improve register and cache reuse. We emphasize here that although the modifications described in this paper were performed manually, most of these techniques are well known. With some familiarity of the compiler and architecture, but without being an expert (and certainly without resorting to assembly line programming), most of our performance gains are realizable. Although familiarity with the RS/6000 architecture helped us in fine tuning the performance, the concepts we used are general and are applicable to many other RISC-based architectures.

Similarly, the 3D data distribution scheme that we used is the same as the block distribution scheme proposed and implemented by modern compilers for distributed memory systems (such as FORTRAN-D). We also made use of the computation and communication trade-off typically observed in parallel implicit computations such as the BT benchmark. Thus we could reduce the detrimental effects of relatively high latency in message passing on the SP architecture. Although this aspect of the optimization is somewhat specialized for incorporation into parallelizing compilers, we believe that this concept can easily be incorporated into special-purpose tools, particularly for CFD applications.

We conclude this paper with a comment on the scalability aspect. A commonly accepted meaning of scalability is the ability to deliver a level of performance that is in proportion to the available resources. Typically, on distributed memory systems, as the processors are added the amount of memory (an important resource) also increases proportionately. When users want to upgrade a parallel system to a larger one, in addition to speed, they are also interested in solving larger applications by making use of all the available resources. As it has been pointed out in various studies on scalability, this factor should be taken into account. This implies that performance of a smaller system on a smaller problem should be compared with the performance of a larger system on a correspondingly larger problem. The performance metric such as speedup shows the effects on the performance as resources are increased without changing the problem size. Since speedup is truly a measure of effective speed, its practicality is limited, especially in the context of an application benchmark. When speed is the sole criterion, one may in principle just replace the CPU by another one that has correspondingly faster clock speed. Scalability implies more than speed, and as such one should conduct experiments with proportionately larger problem sizes on larger systems, as we have shown in this paper. A second point is regarding comparison of two different systems for scalability. The performance results compiled by the NAS group give an excellent indication of achievable performance on various systems (e.g., see Reference 3). The raw figures in those tables can be used to understand the level of performance achievable on a given system of certain configuration. However, one cannot use these performance figures to compare the scalability of two different systems in a meaningful fashion. One meaningful way of comparing the scalability of various systems, that does not involve porting the same implementation to all systems, is to compare the scalability factors as we have done in this paper.

Finally, an objective of benchmarking is to obtain enough information about the system behavior so as to be able to make intelligent estimations about the performance of the system when used to solve problems that are more complex but have similar computations as in the benchmark. One may argue that the raw performance and the scalability results presented in this paper are useful in understanding the system behavior when the computations performed have characteristics similar to those in the BT benchmark. There is no doubt that no single benchmark or application can characterize the system behavior completely, and the performance of a whole suite of benchmarks and applications must be studied in a similar manner to understand the system behavior. However, the results presented in this paper have applicability beyond the narrow range of the BT benchmark. Further work needs to be done to quantify the generality of these results.

Acknowledgments

The author is grateful to Idajean Fisher and Bob Walkup of the Center for Scalable Computing Solutions at the IBM T. J. Watson Research Center for their expert assistance in completing the ex-

periments on the 128-way SP1 system. The author is also grateful to the NASA Ames Research Center and to the helpful staff at NAS for allowing the use of their SP2 system. The author would also like to thank Wei-Hwan Chiang and Mike Pettigrew of the IBM POWER Parallel Division for their help in completing the single processor runs on an SP2 system with large memory nodes.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Fujitsu America, Inc., or Cray Research, Inc.

Cited references

1. D. Bailey, J. Barton, T. Lasinski, and H. Simon, *The NAS Parallel Benchmarks*, NASA Ames Research Center Technical Report, No. RNR-91-002 (1991).
2. D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS Parallel Benchmarks," *International Journal of Supercomputer Applications* **5**, 63–73 (1991).
3. D. Bailey, E. Barszcz, L. Dagum, and H. Simon, *NAS Parallel Benchmark Results 10-94*, NASA Ames Research Center Technical Report, No. RNR-94-006 (1994).
4. V. Naik, "Performance of NAS Parallel Application-Benchmarks on IBM SP1," *Proceedings of Scalable High Performance Computing Conference*, IEEE (1994), pp. 121–128.
5. V. Naik, "Performance of NAS Parallel Benchmark LU on IBM SP Systems," to appear in *Proceedings of Parallel CFD'94*, A. Ecer et al., Editors, Elsevier (1995).
6. D. Bailey, E. Barszcz, L. Dagum, and H. Simon, "NAS Parallel Benchmark Results, *IEEE Parallel and Distributed Technology* **1**, 43–51 (1993).
7. D. Bailey, E. Barszcz, L. Dagum, and H. Simon, *NAS Parallel Benchmark Results 10-93*, NASA Ames Research Center Technical Report, No. RNR-93-016 (1993).
8. W. Gordon, "Blending Function Methods for Bivariate and Multivariate Interpolation," *SIAM Journal of Numerical Analysis* **8**, 158 (1971).
9. R. Beam and R. Warming, "An Implicit Finite Difference Algorithm for Hyperbolic Systems in Conservation Form," *Journal of Computational Physics* **23**, 87–110 (1976).
10. *IBM 9076 Scalable POWERparallel 1: General Information Manual*, GH26-7219-00, IBM Corporation (1993); available through IBM branch offices.
11. C. Stunkel, D. Shea, D. Grice, P. Hochschild, and M. Tsao, "The SP1 High-Performance Switch," *Proceedings of the 1994 Scalable High Performance Computing Conference* (1994), pp. 150–157.
12. T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, "SP2 System Architecture," *IBM Systems Journal* **34**, No. 2, 152–184 (1995, this issue).
13. C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Grice, P. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker, "The SP2 High-Performance Switch," *IBM Systems Journal* **34**, No. 2, 185–204 (1995, this issue).
14. *IBM RISC System/6000 Technology*, SA23-2619, IBM Corporation (1990); available through IBM branch offices.
15. Special issue on the IBM RISC System/6000 processor, *IBM Journal of Research and Development* **34**, No. 1 (1990).
16. *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*, SA23-2619, IBM Corporation (1990); available through IBM branch offices.
17. Special issue on POWER2 and PowerPC, *IBM Journal of Research and Development* **38**, No. 5 (1994).
18. M. Snir, P. Hochschild, D. D. Frye, and K. J. Gildea, "The Communication Software and Parallel Environment of the IBM SP2," *IBM Systems Journal* **34**, No. 2, 205–221 (1995, this issue).
19. V. Bala, J. Bruck, R. Bryant, et al., "The IBM External User Interface for Scalable Parallel Systems," *Parallel Computing* **20**, 445–462 (1994).
20. W. D. Gropp and E. Lusk, "Experiences with the IBM SP1," *IBM Systems Journal* **34**, No. 2, 249–262 (1995, this issue).
21. J. Strikwerda, *Finite Difference Schemes and Partial Differential Equations*, Wordsworth and Brooks, Inc., Pacific Grove, CA (1989).
22. N. Naik, V. Naik, and M. Nicoules, "Parallelization of a Class of Implicit Finite Difference Schemes in Computational Fluid Dynamics, *International Journal of High Speed Computing* **5**, 1–50 (1993).
23. V. Naik, "Performance Effects of Load Imbalance in Parallel CFD Applications," *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, J. Dongarra et al., Editors, SIAM (1992), pp. 425–434.
24. V. Naik, "Scalability Issues for a Class of CFD Applications," *Proceedings of Scalable High Performance Computing Conference*, IEEE (1992), pp. 268–275.
25. V. Naik, N. Naik, and M. Nicoules, "Implementation of Implicit Scheme Based CFD Applications on Message Passing Multiprocessor Systems," in *Parallel CFD: Implementations and Results Using Parallel Computers*, H. Simon, Editor, The MIT Press (1992), pp. 97–125.
26. D. Bailey, E. Barszcz, L. Dagum, and H. Simon, "NAS Parallel Benchmark Results," *Proceedings of the 1994 Scalable High Performance Computing Conference*, IEEE (1994), pp. 111–120.

Accepted for publication February 13, 1995.

Vijay K. Naik *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: vkn@watson.ibm.com).* Dr. Naik is a research staff member in the Parallel Applications Methods and Analysis group at the Thomas J. Watson Research Center. His current research interests include development of application interfaces and runtime systems for load balancing and resource sharing on scalable high-performance multiprocessor systems. He is also interested in the characterization of architectures and applications for high-performance computing and development of systems that are tuned to application characteristics. Prior to joining IBM, he was a staff scientist at ICASE, NASA Langley Research Center. He received a Ph.D. and A.M. in computer science in 1988 and 1984, respectively, both from Duke University. In 1982, he received an M.S. from the University of Miami and, in 1980, a B.Tech. from the Indian Institute of Technology, Madras, both in mechanical engineering.

Reprint Order No. G321-5569.