Computer Sciences Department

Parallelizing Appbt for a Shared-Memory Multiprocessor

Douglas Burger
Sanjay Mehta

Technical Report #1286

September 1995

UNIVERSITY OF WISCONSIN MADISON

1286

Parallelizing Appbt for a Shared-Memory Multiprocessor

Doug Burger

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706 USA
dburger@cs.wisc.edu

Sanjay Mehta

Department of Chemical Engineering
University of Wisconsin-Madison
1415 Engineering Drive
Madison, Wisconsin 53706 USA
sanjaym@cae.wisc.edu

Abstract

The NAS Parallel Benchmarks are a collection of simplified computational fluid dynamic (CFD) applications. We have rewritten and parallelized Appbt—a CFD application that uses the solution of a block-tridiagonal system—to run efficiently on shared-memory multiprocessors. We tested our algorithm through simulation on the Wisconsin Wind Tunnel. We tested our code on two major types of shared-memory multiprocessors, one using a $\mathbf{dir_N NB}$ *directory protocol and one using the Scalable Coherent Interface cache-coherence protocol. We simulated the code with these protocols for machines ranging from 1 to 128 processors. We found that our parallelization methodology worked well for up to 128 processors, and will apparently scale to even larger systems. Further study is required to confirm this hypothesis, however.*

1 Introduction

NASA developed the NAS Parallel Benchmarks [3] for the performance evaluation of highly-parallel computers. These benchmarks consist of a set of kernels, and three computational fluid dynamic (CFD) applications. The benchmark designers stripped these codes of the complexities of a real CFD computation, leaving behind the core of the algorithm and the data movement and sharing that a true CFD application would contain. These applications thus mimic the way that actual large-scale CFD codes would behave on a real machine, without containing so much complexity that they are difficult to analyze or expensive to simulate.

We selected one of these benchmarks—Appbt (which stands for **app**lication with **b**lock-**t**ridiagonal solution)—to port to the Wisconsin Wind Tunnel [7]. WWT is a virtual prototyping system, which allows fast execution-driven simulation of a shared-memory multiprocessor running real scientific codes. Because of WWT's speed and flexibility, we were able to run our parallelized Appbt on a wide range of simulated machines.

The pseudo-application problem requires the steady-state solution of a set of nonlinear coupled partial differential equations—for a given set of boundary and initial conditions—on a 3-D rectangular region. Our implementation decomposes the spatial domain into a number of smaller ones and assigns a spatial region to each processor. The solution is divided into two separate stages. In the first stage, sharing occurs with neighboring points only in each dimension. There are no spatial dependencies in the first solution stage; all points depend on the neighboring values calculated during the previous time step. The second stage of the solution contains sharing along axes in each dimension. These dependencies are unidirectional at all times (although directions change 6 times per iteration). We use a software pipelining technique to speed execution during the second stage of the solution algorithm, since there are spatial dependencies along each dimensional axis.

To test the scalability and performance of our solution, we simulated execution of Appbt on three different types of shared-memory machines: $\mathbf{dir_N NB}$ [2], a full bitmap directory protocol, and the Scalable Coherent Interface [1, 5], both with and without its pairwise sharing option on. The sizes of these simulated machines ranged from 1 to 128 processors, and we used both processor and memory-constrained scaling techniques [9] to evaluate scalability.

2 The Appbt algorithm

This section briefly describes both the problem and the solution in mathematical terms. For a more complete description, both textual and formal, the reader is referred to the NAS report [3].

This work is supported in part by NSF PYI Award CCR-9157366, NSF Grant MIP-9225097, NSF Grant CCR-9207971, an unrestricted grant from Apple Computer, Inc., and donations from Thinking Machines Corporation, Xerox Corporation, and Digital Equipment Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618, with matching funding from the University of Wisconsin Graduate School.

2.1 Problem description

The simulated problem (in physical terms) represents the laws of conservation of mass, momentum, and energy applied to a fluid medium in motion. These laws are represented by the Navier-Stokes equations. The fluid region of interest may be the air element flowing past the wings of a spacecraft, or it may represent the hot gasses emerging from its engine. The time-dependent solution for the temperature, pressure, and velocity is desired at various points in the spatial domain. For performance evaluation of the algorithm, we are not interested in the actual physical problem. The simplified application can be reduced to a set of five coupled nonlinear partial differential equations, shown in Equation 1. These equations are associated with the Dirchlet type of boundary conditions, Equation 2, and initial conditions, given by Equation 3.

The state equations:

$$\frac{\partial u}{\partial \tau} = \frac{\partial}{\partial \xi}E(u) + \frac{\partial}{\partial \eta}F(u) + \frac{\partial}{\partial \zeta}G(u) + \frac{\partial}{\partial \xi}T(u, u_\xi) + \frac{\partial}{\partial \eta}V(u, u_\eta) + \frac{\partial}{\partial \zeta}W(u, u_\zeta) + H(u, u_\xi, u_\eta, u_\zeta), \quad (\tau, \xi, \eta, \zeta) \in D_\tau \times D \tag{1}$$

with the boundary conditions:

$$B(u, u_\xi, u_\eta, u_\zeta) = u^B(\tau, \xi, \eta, \zeta), \quad (\tau, \xi, \eta, \zeta) \in D_\tau \times \partial D \tag{2}$$

and the initial conditions:

$$u = u^0(\tau, \xi, \eta, \zeta), \quad (\xi, \eta, \zeta) \in D, \text{ for } \tau = 0 \tag{3}$$

The variable u represents a vector of five components, i.e.

$$u = \left[u^1, u^2, u^3, u^4, u^5\right]^T \tag{4}$$

E, F, G, T, V, and W are the known functions of the state vector u and its spatial derivatives. Each of them is also a vector with five components. Equation 1 represents the most general form of governing equation, which can describe many kinds of physical phenomena with appropriate initial and boundary conditions.

The solution of the problem requires the evaluation of the state vector u at various points in the domain for each time step. Equation 1 is explicit in the time derivative, meaning that the value of the state vector at any time can be calculated in terms of the initial data. For the simulated problem, a polynomial type of solution containing up to the fourth power of the spatial variables is described. The forcing function H in Equation 1 is chosen such that the polynomial solution satisfies the state equations along with the boundary and initial data. Initial conditions for the interior points of the domain are obtained from the boundary conditions using transfinite, trilinear interpolation. For this particular problem, the domain of application, D, is a unit cube, which is symmetrical in all three spatial dimensions. Whatever computations are needed for one dimension can be easily extended to the other two dimensions.

2.2 Numerical solution

The approach followed here can best be described as a pseudo-time marching scheme with a spatial discretization procedure based on finite-difference approximation. A fourth-order numerical dissipation term is included to suppress high-frequency modes and allow the solution to converge to a steady state.

Time differencing is carried out using a first-order-accurate, Euler implicit scheme:

$$\Delta u^n = \Delta\tau(\partial \Delta u^n/\partial t) + \Delta\tau(\partial u^n/\partial t) + O(\Delta\tau^2) \quad \text{where } \Delta u^n = u^{n+1} - u^n \tag{5}$$

Expressions for the first derivative terms in Equation 5 are substituted from Equation 1. The resulting equation is non-linear in Δu^n.

$$\Delta u^n = \Delta\tau\left[\frac{\partial}{\partial \xi}\left(\Delta E^n + \Delta T^n\right) + \frac{\partial}{\partial \eta}\left(\Delta F^n + \Delta V^n\right) + \frac{\partial}{\partial \zeta}\left(\Delta G^n + \Delta W^n\right)\right] + \Delta\tau\left[\frac{\partial}{\partial \xi}\left(E^n + T^n\right) + \frac{\partial}{\partial \eta}\left(F^n + V^n\right) + \frac{\partial}{\partial \zeta}\left(G^n + W^n\right)\right] + \Delta\tau H^\circ \tag{6}$$

Using Taylor series expansion, the equation can be linearized without losing any temporal accuracy of difference scheme, i.e.

$$\Delta E^n = A^n(u)\Delta u^n + O(\Delta\tau^2) \tag{7}$$

where $A(u)$ is the Jacobian matrix $(\partial E/\partial u)$ of size 5×5.

Similar to Equation 7, expressions for other difference terms can be substituted, which will involve other Jacobian matrices (e.g. B, C, N, S, Q, etc.). All of them are 5×5 block matrices. After completing these linearizations, we obtain the following explicit equation in Δu^n:

$$\left\{I - \Delta\tau\left[\frac{\partial A^n}{\partial \xi} + \frac{\partial^2 N^n}{\partial \xi^2} + \frac{\partial B^n}{\partial \eta} + \frac{\partial^2 Q^n}{\partial \eta^2} + \frac{\partial C^n}{\partial \zeta} + \frac{\partial^2 S^n}{\partial \zeta^2}\right]\right\}\Delta u^n = \Delta\tau\left[\frac{\partial}{\partial \xi}\left(E^n + T^n\right) + \frac{\partial}{\partial \eta}\left(F^n + V^n\right) + \frac{\partial}{\partial \zeta}\left(G^n + W^n\right)\right] - \Delta\tau\varepsilon\left[h_\xi^4\left(\frac{\partial^4 u^n}{\partial \xi^4}\right) + h_\eta^4\left(\frac{\partial^4 u^n}{\partial \eta^4}\right) + h_\zeta^4\left(\frac{\partial^4 u^n}{\partial \zeta^4}\right)\right] + \Delta\tau H^\circ \tag{8}$$

The fourth-order derivative terms in the above equation are the dissipation terms and may depend on the nature of spatial discretization used. All the spatial derivative terms can be discretized by using difference formulas on a mesh of uniform increments (h_ξ, h_η, h_ζ) in the three coordinate directions; two-point central difference for the first derivatives, three-point difference for the second derivatives and five-point for the fourth-order terms. After substitutions

```
for i = 1 to #iterations do
  jacx()
  btridx()
  BARRIER
  jacy()
  btridy()
  BARRIER
  jacz()
  btridz()
  BARRIER
  rhs()
  BARRIER
end do
```

Figure 1. Skeleton code for Appbt

are made for these spatial derivatives, the original system of differential equations reduces to a large system of algebraic equations. Such a large system is difficult to handle. For the solution in Appbt, an algorithm called 'Beam-Warming' is used to reduce this system to three factors (see Equation 9). These are then sequentially solved using forward elimination and backward substitution (Gaussian elimination). The final equation written below is expressed in terms of derivatives, rather than differences, for clarity.

$$\left\{I-\Delta\tau\left[\frac{\partial A^n}{\partial\xi}+\frac{\partial^2 N^n}{\partial\xi^2}\right]\right\}\times\left\{I-\Delta\tau\left[\frac{\partial B^n}{\partial\eta}+\frac{\partial^2 Q^n}{\partial\eta^2}\right]\right\}\times \quad (9)$$
$$\left\{I-\Delta\tau\left[\frac{\partial C^n}{\partial\zeta}+\frac{\partial^2 S^n}{\partial\zeta^2}\right]\right\}\Delta u^n = \text{RHS}$$

(everything on the right-hand side of Equation 8 is represented by RHS in Equation 9 and Equation 10.

$$\left\{I-\Delta\tau\left[\frac{\partial A^n}{\partial\xi}+\frac{\partial^2 N^n}{\partial\xi^2}\right]\right\}\Delta u_1 = \text{RHS} \quad (10)$$

The above equation is solved for the vector Δu_1, which represents the last two terms on the left-hand side of Equation 9. Equation 10 can be symbolically represented as $Ax = b$, where A, the coefficient matrix, corresponds to the term in the curly brackets of Equation 10. The substitution of the difference formulas for the derivatives in the above equation reduces it to a block-tridiagonal matrix with each block of dimension 5×5. The system of equations is solved and the solution obtained is used as the right-hand side for the next factor, Equation 11. This process is again repeated (see Equation 12), which produces Δu^n. This result can be used to obtain the state vector at the next timestep in accordance with Equation 13.

$$\left\{I-\Delta\tau\left[\frac{\partial B^n}{\partial\eta}+\frac{\partial^2 Q^n}{\partial\eta^2}\right]\right\}\Delta u_2 = \Delta u_1 \quad (11)$$

$$\left\{I-\Delta\tau\left[\frac{\partial C^n}{\partial\zeta}+\frac{\partial^2 S^n}{\partial\zeta^2}\right]\right\}\Delta u^n = \Delta u_2 \quad (12)$$

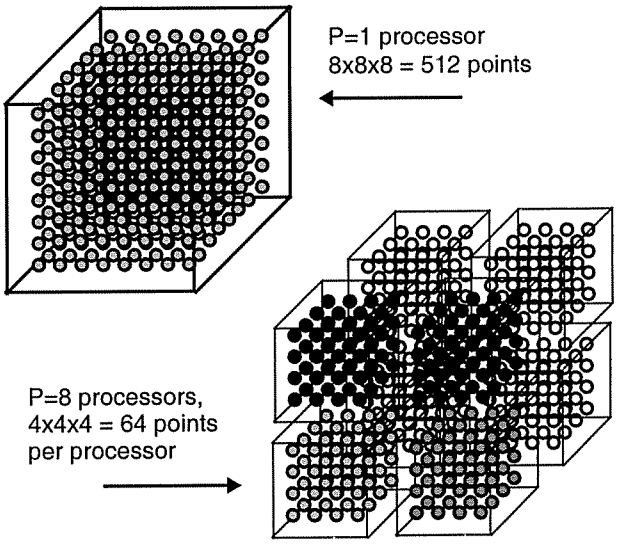


Figure 2. Spatial domain decomposition

$$\left[u^{n+1}\right]_{i,j,k} = \left[u^n\right]_{i,j,k} + \left[\Delta u^n\right]_{i,j,k}, \text{ for } (i,j,k)\in D_h \quad (13)$$

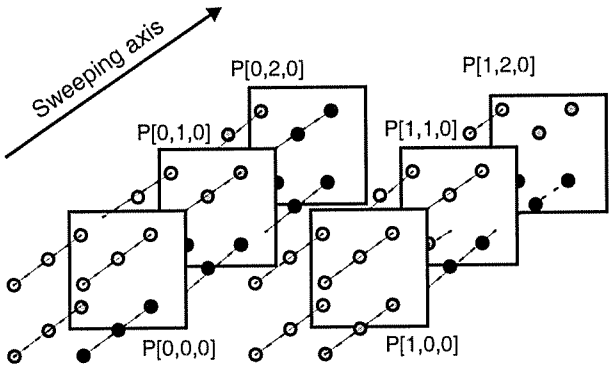
where D_h is the discretized spatial domain.

3 Parallelization algorithm

The Appbt code consists of seven main routines: **jacx**, **jacy**, **jacz**, **btridx**, **btridy**, **btridz**, and **rhs**. The first three compute the block-tridiagonal system for a sweep in each dimension, and the second three solve the block-tridiagonal systems computed in each of the jac routines using Gaussian elimination. The `rhs` routine computes the right hand side of the equation (see Equation 9). A skeleton of the main routines and barriers is shown in Figure 1.

The **jac** routines depend only on the values calculated during the previous timestep, and thus there are no spatial dependencies for the current timestep. However, since everything is essentially a loop-carried dependence, the only parallelism to be extracted is spatial.

We therefore decomposed the data spatially, assigning an equal number of internal points to each processor for computation. Since the "sweeps" through the data occur separately in each dimension, assigning contiguous sections of the original main array to each processors would have resulted in heavy communication for any two of the three dimensional sweeps in the **jac** routines, so we remapped the arrays. The domain was divided into subcubes (one per processor), and the arrays were remapped so that all points belonging to a subcube were contiguous in each array. The domain therefore consists of P subcubes, where P is the number of processors, and each data array consist of P discrete sections. Each section contains all of the points belonging to a processor's subcube.



A 4 x 2 x 9 problem, mapped to a 6-processor system (a 2 x 1 x 3 grid). Counter values for the above example are:

2 1 2 2

0 0 1 0

o = Finished
o = In progress
● = Not yet started

Figure 3. Software pipelining of FE & BS routines

Figure 2 illustrates such a decomposition, where equally-sized subcubes of points are assigned one each to eight processors. All communication for the **jac** phases of the program occurs only on the faces of the subcubes; interior points are used by the owning processor only.

In the second main phase of the program, comprised of the **btrid** routines, there are spatial dependencies within the current iteration. Each of these routines each have two phases: forward elimination and back substitution. Each routine "sweeps" through all of the points in a different dimension, with dependencies existing only in that dimension for the routine in question. Dependencies are backward during the forward elimination phase, and forward during the back substitution phase.

For example, in **btridx**, the sweep occurs in the x-dimension. For the forward elimination phase, each point (x, y, z) depends on the (x-1, y, z) just computed. When all the (x, y, z) have been computed, the back-substitution phase begins, and all (x, y, z) depend on the (x+1, y, z) just computed. Naively, this would only enable $P^{2/3}$ of the processors to be working at once.

To maximize the available parallelism, we implemented a software pipelining technique, in which a shared counter is associated with each axis in the direction of the current dimensional sweep. If the number of points in a processor's subcube is $i \times j \times k$ (the corresponding dimensions are x, y, z) with the sweep occurring in the x dimension, the processor has $j \times k$ subaxes to compute, each one of them i points long. When the counter for a subaxis is incremented, the next processor along that axis computes its section of that access, and increments the counter when complete. When the forward elimination phase completes for an axis, the same process is performed in reverse, with the processors computing the backward substitution and decrementing each counter as they complete the substitution for each subaxis. This technique is depicted in Figure 3, where 6 processors are each operating on a different axis, and thus are all operating concurrently.

4 Simulation methodology

Our simulation testbed was the Wisconsin Wind Tunnel [7], which is a parallel discrete-event simulator that runs on a Thinking Machines CM-5. WWT combines direct execution with low-overhead simulator traps, enabling simulation of codes approaching the size of real parallel programs.

The original Appbt code obtained from NASA was a serial version written in FORTRAN. Since WWT does not currently support FORTRAN libraries, we were obliged to convert the Appbt source to C before parallelizing it. We used the Bell Labs **f2c** translator for this purpose. The output of this translator is not intended to be human-readable, so we recoded the application by hand to make the code more understandable. **f2c** converts multidimensional FORTRAN arrays into one-dimensional C arrays, so we converted all of the array offsets into macros to improve their readability. The FORTRAN version allocated all memory statically, so the C version also statically allocates some hardcoded maximal problem size, out of which the experimenter can run equally-sized or smaller data sets. All of the FORTRAN library calls had to be translated by hand.

NAS provided automatic checking of results in the benchmark code for two problem sizes: $12 \times 12 \times 12$ and $60 \times 60 \times 60$ cubes, running for 60 iterations each. Since the simulation time of Appbt with the $60 \times 60 \times 60$ point data set is prohibitively long, we checked our implementation for computational correctness at each stage with the $12 \times 12 \times 12$ check.

We first verified that the translated C code executed correctly on a uniprocessor (A DECstation 3100 running Ultrix 4.3). We then implemented a uniprocessor version of the parallelized code, in which a loop tested the parallelism with each iteration representing a different processor. Finally, we ported the implementation to WWT, and implemented all barriers, locks, shared-memory allocation, etc., using the PARMACS macro package.

Although the parallelization algorithm assigns single contiguous regions of the data arrays to the points associated with each processor, the page placement algorithm that WWT uses assigns shared pages to processors round-robin. This means that uncached data that are supposed to be local to the "owning" processor will force remote

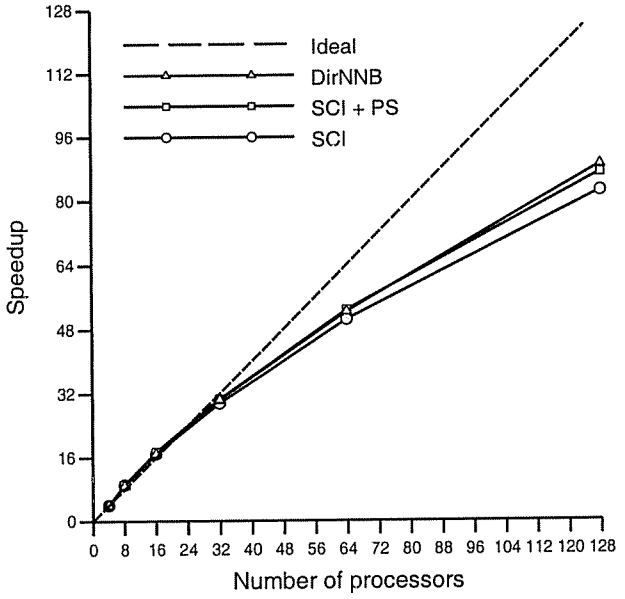


Figure 4. Constant data set speedups

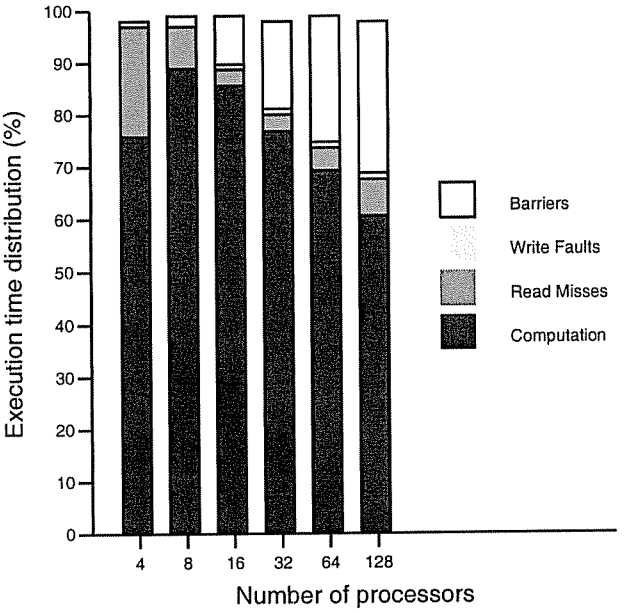


Figure 5. Constant execution time distributions

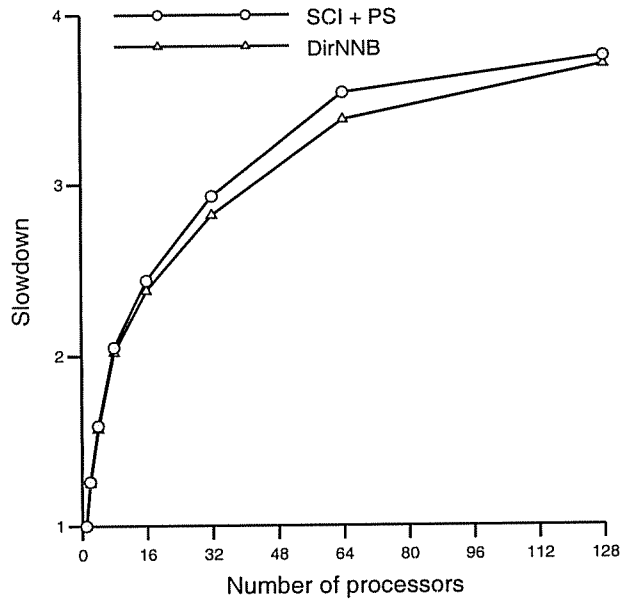


Figure 6. Scaled data set slowdowns

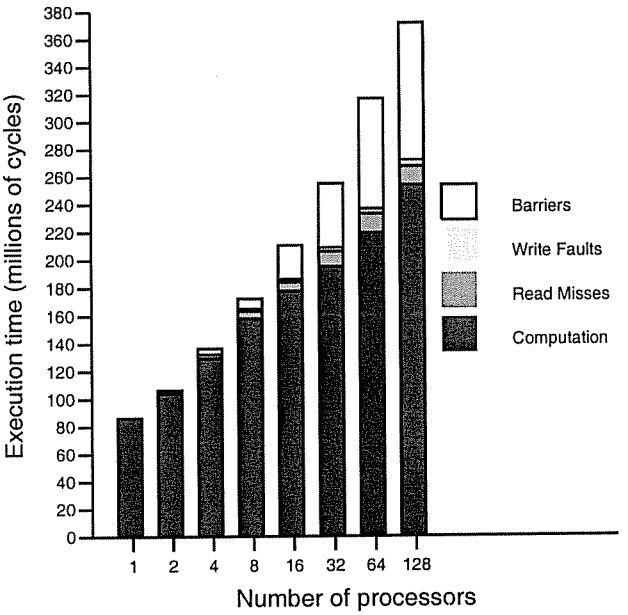


Figure 7. Scaled execution time distributions

accesses to load the data into the cache, thus preventing the attainment of the optimal possible performance under this parallelization scheme.

5 Demonstrative results

This section describes some simple experiments that evaluate the effectiveness of our parallelization algorithm. Our simulations all ran for 20 iterations (the default is 60). We also assumed a 1MB cache (with 64-byte blocks) per node. The interconnection network was not modelled; we assumed constant message latencies of 100 cycles. All subsequent descriptions of machine behavior refer to the simulated target machine and not the host CM-5.

Figure 4 shows speedups for three different systems; one running the $\mathtt{dir}_N\mathtt{NB}$ cache-coherence protocol, and two running variants of the Scalable Coherent Interface [1, 5] cache-coherence protocol. All experiments in this graph were run with a $24 \times 24 \times 24$ point data set.

$dir_N NB$ [2] is a full-bitmap directory protocol in which the home location of each cache block contains one bit for each processor in the system, to keep track of shared copies of the block. When a processor requests an exclusive copy of the block, an invalidation message is sent to each processor which owns a copy. Although this protocol is quite simple, the memory requirements quickly become prohibitively large as the number of processors increases (e.g. 128 bits per cache line in a 128-processor system).

The other two speedup lines shown in Figure 4 represent the Scalable Coherent Interface (SCI) base protocol [1, 5], running with and without one particular optional optimization called *pairwise sharing*. SCI is also an invalidation-based protocol, but it has constant memory overhead per cache line as the system size grows. SCI accomplishes this by maintaining the sharing nodes as a distributed linked list, in which all list pointers reside in the caches rather than main memory. The SCI pairwise-sharing option makes two-node sharing more efficient when the data in the line are written, maintaining the list when it would otherwise be broken down. The penalty for this optimization is sometimes having extra remote transactions when a third sharer joins the list.

Due to memory limitations on the CM-5, the smallest runs we were able to compete with the $24 \times 24 \times 24$ data set assumed 4 processors. The largest runs assumed 128 processors. The speedups are calculated using the 4-processor runs as the base case, assuming speedups of 4 for those over a uniprocessor run.[1] The serial initialization portion of each execution was not included in the speedup calculations, since this roughly constant quantity becomes proportionally much larger as the overall running time is greatly reduced by exploiting parallelism.

All three protocols exhibit similar speedups, with the SCI runs coming very close to or exceeding the speedups of the **$dir_N NB$** runs. The parallel efficiency (speedup/number of processors) remains high up to 64 processors, at about 80%. At 128 processors, however, the parallel efficiency drops to between 60% and 70%. This is not due to fundamental limitations in the parallelization algorithm, but to the increased quantity of required sharing. Since the data set size is held constant as the number of processors grows, the proportion of each processor's data that is shared also grows, from 54% of the points in the 32-processor case to 87% of the points in the 128-processor case. We were unable to experiment with larger data sets, however, due to memory limitations on our CM-5 (primarily for runs with few processors).

1. This is actually a pessimistic assumption because of super-linear cache effects. The 4-processor run would probably have a speedup of greater than 4 because of the small total cache in a 1-processor system.

Since we assumed 1MB caches per processor for all runs, we see a super-linear speedup for 8- and 16-processor runs. This is because the performance gains from the added SRAM in the system outstrips the performance losses due to parallelization overhead. Once the entire data set fits in the system cache, however, this effect no longer affects larger systems. The large ratio of cache to data set size may be unrealistic for larger data sets, but this mitigates the negative effects of WWT's round-robin page allocation. (The round-robin allocation eliminates the benefit of our having mapped contiguous portions of the shared arrays to individual processors).

Figure 5 depicts the decomposition of the parallel execution time into the time spent in barriers, write faults, read misses, and computation, as the number of processors is increased from 4 to 128. The cache-coherence protocol used for these results was SCI with pairwise sharing turned on. The large number of read misses for the 4- and 8-processor runs are due to the smaller amount of cache in the system (4 and 8 MB, respectively), since the total size for the $24 \times 24 \times 24$ problem is approximately 9.5 MB. The write faults and read misses for 16 processors and higher are mainly due to coherence misses. This explains why the cache misses increase again for the larger systems, since the percentage of the data that is shared grows with the system size when the data set size is held constant. Barrier overhead is the main contributor to the diminishing parallel efficiency. We attribute the growing increase in barrier overhead to the increased proportion of communication. More communication causes increased variance in subroutine execution times, increasing load imbalance and thus the average waiting time at each barrier.

One interesting phenomenon we observed was that performance of the larger runs was initially extremely poor for **$dir_N NB$**. Having all of the software pipelining counters allocated contiguously resulted in all of them being allocated on one page. Since WWT assigns fractions of global shared memory at a page granularity, one node became responsible for all of the shared counters. This caused massive contention at the memory directory for that node, with over three orders of magnitude more accesses at than the other directories. By padding the counter data structure so that each node owned approximately the same number of counters, the directory contention vanished and overall execution time was cut by as much as 50%. SCI without pairwise sharing showed some directory contention, but not enough to seriously inflate execution time. With pairwise sharing, very few requests needed to be sent to the counter directory at all (this is an artifact of the large caches we assumed, however). All results presented in this report assume padded counter structures.

We performed another set of experiments to try and negate the performance deterioration in large systems due to insufficiently large data sets. Instead of holding the problem size constant, as in Figure 4 and Figure 5, we used a form of *memory-constrained scaling* [9]. In this scaling strategy we kept the number of points constant per processor, scaling the total data set size linearly with the number of processors. Each processor was assigned a $6 \times 6 \times 6$ subcube of points. Since the Appbt algorithm is $O(N)$ (N in this example is $P \times 216$), the computation time does not explode with the increased data set sizes. As with the previous experiments, we discounted the serial initialization phase of the program when computing execution times and slowdowns.

Figure 6 shows the slowdowns for memory-constrained runs of $\texttt{dir}_\text{N}\text{NB}$ and SCI (with pairwise sharing on), as the system size and data sets are scaled. The slowdown is simply calculated as follows:

$$\text{Slowdown}_P = \frac{\text{T}_P}{\text{T}_1} \tag{14}$$

where P is the number of processors, T_1 is the execution time of the run with 1 processor, and T_P is the execution time of the run with P processors.

This graph clearly shows a high initial overhead for scaling up to 32 processors, which levels off once the problem and system size exceed a hundred or so. $\texttt{dir}_\text{N}\text{NB}$ outperforms SCI slightly for the medium-size systems, but their performance converges for the largest run that we were able to do.

Figure 7 shows the absolute execution times for the SCI experiments depicted in Figure 6. The time spent for computation increases proportionally to the log of the number of processors. The increasing number of read misses and write faults is due to the greater percentage of processor subcubes that have shared faces—and therefore a higher percentage of shared points—in larger systems. 20% of the processor subcube faces are shared in a 4-node machine, whereas 66% of the subcube faces are shared in a 128-node machine.

The barrier time for these experiments increases similarly to the constant data set experiments. The percentage of total time that processors spend waiting at a barrier increases quickly up to 64 nodes, and then begins to level off as the variance in load imbalances ceases to grow substantially. This levelling-off is the biggest single reason for the flattening of the slowdown curve in Figure 6. Had we mapped the "owned" pages to the appropriate owning processor, we predict that the execution time would level off almost completely and would eventually show no increase as the number of processors increased further. This prediction discounts network latencies and contention, which would likely both increase with larger systems.

6 Future Work

The Appbt results presented in this report are for only the basic parallelized algorithm, with naive data placement and unaggressive optimization. We have undertaken several more aggressive implementation efforts, one of which is complete. These include:

- A version using SCI as the base protocol, extended with some of the more aggressive SCI options [6]. This work is underway.
- A Tempest-interface-compatible [8] version that implements two step-wise improvements: one that using **`signal & wait`** rather than shared counters during the solution routines, and one that uses an application-specific update protocol instead of the default library invalidate protocol. Both protocol extensions were written specifically for Appbt. This implementation is complete, and results have appeared in the literature [4].
- A variant of the Tempest code where one address-space per node is multithreaded using Solaris kernel threads and a home-grown user-level thread package (Cyklos). Threads overlap communication with computation by switching on remote cache misses and synchronization points. This work is underway.

Results from these optimized implementations will appear in a subsequent publication. We also plan to run these codes on larger simulated machines.

7 Summary

This report has described Appbt, one of the NAS Parallel Benchmarks. We have given a brief description of the problem, followed by a more detailed description of the method we used to parallelize this code for a shared-memory multiprocessor.

We have also presented some demonstrative results, which used the Wisconsin Wind Tunnel to simulate the execution of our parallelized code on several shared-memory multiprocessors. We showed that—for both a $\texttt{dir}_\text{N}\text{NB}$ machine and two versions of an SCI machine—the scalability of the code up to 128-processor machines was quite good. We also showed that the algorithm scaled well when the problem size was scaled along with the machine.

Finally, we presented a list of optimizations that should permit even more efficient scaling and execution on shared-memory or hybrid shared-memory machines. These optimizations will be evaluated in the near future.

Acknowledgments

The authors thank Profs. Mark Hill, Sangtae Kim, and

Mary Vernon, who taught the computational science class out of which this work grew. We also thank Profs. David Wood and Jim Goodman for supporting this work, and finally Babak Falsafi and Alain Kägi for their generous assistance.

References

[1] Scalable Coherent Interface (SCI). ANSI/IEEE Std 1596-1992, August 1993.

[2] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[3] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002 Revision 2, NASA Ames Research Center, August 1991.

[4] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[5] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Scalable Coherent Interface. *IEEE Computer*, 23(6):74–77, June 1990.

[6] Alain Kägi, Nagi Aboulenein, Douglas C. Burger, and James R. Goodman. Techniques for Reducing the Overheads of Shared-Memory Multiprocessing. In *Proceedings of the 9th International Conference on Supercomputing*, July 1995.

[7] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 48–60, May 1993.

[8] Steven K. Reinhardt, James L. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, April 1994.

[9] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *IEEE Computer*, 26(7):42–50, July 1993.

Appendix A

The original Appbt FORTRAN code was written by Sisira Weeratunga at NASA Ames Research Center. NASA requires that you obtain a NAS license to get the original FORTRAN version of the code. However, NASA has given us permission to freely redistribute our "significantly changed" code. The C version of the serial Appbt code is available via ftp at the following location:

```
ftp.cs.wisc.edu
/wwt/Benchmarks/nas/nas.serial.tar.Z
```

The parallelized C version of the code, which assumes the presence of the PARMACS macro package, is available at:

```
ftp.cs.wisc.edu
/wwt/Benchmarks/sc94/appbt/src/nas.parallel.tar.Z
```