High Performance Parallel Implementations of the NAS Kernel Benchmarks on the IBM SP2

R. C. Agarwal, B. Alpern, L. Carter, F. G. Gustavson, D. Klepacki [*], R. Lawrence, and M. Zubair
IBM T.J. Watson Research Center
P.O. Box 218,
Yorktown Heights, New York 10598

Abstract

Recently, researchers at NASA Ames have defined a set of computational benchmarks designed to measure the performance of parallel supercomputers. In this paper, we describe the parallel implementation of the five kernel benchmarks from this suite on the IBM SP2, a scalable, distributed-memory parallel computer. High-performance implementations of these kernels have been obtained by mapping the computation of these kernels to the underlying architecture of the SP2 machine. Performance results for the SP2 are compared with publicly available results for other high-performance computers.

1 Introduction

Recently, researchers at NASA Ames have defined a set of computational benchmarks for the performance evaluation of parallel supercomputers for large scientific applications [3, 4]. Known as the NAS parallel benchmarks, this set has become an increasingly recognized means of quantifying performance of high-performance computers on a range of algorithms of interest to many users of such machines. A key feature of these benchmarks is that the choice of data structures, algorithms, processor allocation and memory usage are left open to the discretion of the implementer. In other words, an implementer has the flexibility to design an algorithm which matches the target machine. This is an important feature which motivates researchers working in the area of high performance parallel algorithms to investigate efficient ways of implementing the benchmarks.

The NAS Kernel Benchmarks

The NAS benchmark suite consists of five kernel benchmarks and three simulated computational fluid dynamics application benchmarks. In this paper we focus on the implementation of kernel benchmarks on IBM SP2; the implementation of the simulated application benchmarks is discussed in [9, 10, 11].

The five kernel benchmarks vary in their computation and communication requirements:

- **EP:** This is an *Embarrassingly Parallel* kernel designed to measure primarily floating-point compute performance. It requires minimal interprocessor communication.

[*]IBM POWER Parallel Systems, Neighborhood Rd., Kingston, NY 12401

- **MG:** This is a 3-D multigrid kernel requiring highly structured interprocessor communication.
- **CG:** This is a conjugate gradient kernel to compute an approximation to the smallest eigenvalue of a large, sparse matrix. This kernel tests irregular long-distance communication.
- **FT:** This kernel solves a 3-D partial differential equation using FFTs and is a rigorous test of long-distance communication performance.
- **IS:** This is a large integer sort operation testing both integer computation speed and interprocessor communication. This kernel stresses the integer performance of the underlying node.

The IBM SP2 Parallel System

The SP2 is the second offering in IBM's Scalable POWERparallel family of parallel systems based on IBM's RS/6000 processor technology. The SP2 is a distributed-memory system consisting of up to 128 processor nodes connected by a High-Performance Switch. Three different processor nodes are available, based on RS/6000 Model 370, 390, and 590 CPU planars. (These nodes are also known as Thin 62, Thin 66, and Wide nodes, respectively). The Model 370 processor is based on the original POWER architecture while the 390 and 590 processors are based on POWER2 architecture. Each compute processor has at least 64 MB of local memory (wide nodes can have up to 2 GB of local memory per node). Each compute node has a locally attached disk.

All performance results reported in this paper are made on a wide-node SP2 system with POWER2 (Model 590) compute nodes. POWER2 processors have two floating point units and two fixed point units, and therefore can perform two fixed-point instructions and two floating-point instructions every cycle, if no dependencies exist. The SP2 wide nodes are clocked at 66.7 MHz, and thus each node has a peak floating-point performance of 266 MFLOP/s based on two multiply-add instructions per cycle. This model has a 256 KB four-way set-associative data cache with a cache line size of 256 bytes. An important feature of the POWER2 architecture is the availability of floating-point quad-word load/store operations. Using both fixed-point units, this results in an effective bandwidth of four double-words per cycle between cache and the floating-point registers. Furthermore, all POWER2 nodes can fetch a complete cache line from memory in eight cycles after the first word of the line arrives; on the SP2 wide nodes, this represents a local memory bandwidth of 32 bytes per cycle, or 2134 MB/s. The very high data access rates between cache and registers and between memory and cache make possible the very high sustained performance of the POWER2 nodes in the SP2. Many of the optimizations discussed in this paper are designed to exploit these capabilities.

The High-Performance Switch is a multi-stage packet switch providing a peak point-to-point bandwidth of 40 MB/s in each direction between any two nodes in the system. For the wide-node system, the sustained application buffer to application buffer transfer rate is approximately 35 MB/s for a uni-directional transfer measured as one half of the time necessary for a round-trip "ping" operation between two compute nodes. The latency (*i.e.* the time for a zero-byte message) measured in the same manner is approximately 40 microseconds on the SP2. In the case where a compute processor simultaneously sends and receives different messages, the aggregate (incoming plus outgoing) bandwidth at this node is approximately 48 MB/s on the wide-node system. This is the transfer rate observed when two nodes exchange long messages, a common communication operation in many parallel algorithms.

2 The Embarrassingly Parallel Benchmark

In this benchmark, two-dimensional statistics are accumulated from a large number of Gaussian pseudo-random numbers, which are generated according to a particular scheme that is well suited for parallel computation. This problem is typical of many "Monte-Carlo" applications. Since it requires very little communication, this benchmark measures the compute performance of the underlying node. The problem has been defined in two sizes: class A, whose size is $n = 2^{28}$, and class B problem size is four times bigger. Because the problem scales very nicely, it is sufficient to restrict ourselves to the class B problem on a single processor.

Statement of the EP Problem
Generate pairs of Gaussian random deviates, also called two independent normally distributed variables, according to a specific scheme, see [7], and tabulate the number of pairs in successive square annuli.

Set $n = 2^{30}$, $a = 5^{13}$, and $s = 271828183$. Generate the pseudo-random floating point values r_j in the interval (0, 1) for $1 \leq j \leq 2n$ using the scheme described in the papers [12, 3]. Then for $1 \leq j \leq n$ set $x_j = 2r_{2j-1} - 1$ and $y_j = 2r_{2j} - 1$. Thus x_j and y_j are uniformly distributed on the interval (-1, 1). Next set $k = 0$. Then beginning with $j = 1$ test to see if $t_j = x_j^2 + y_j^2 \leq 1$. If not, reject this pair and proceed to the next j. If this inequality holds, then set $k = k + 1$, $X_k = x_j\sqrt{(-2\log t_j)/t_j}$ and $Y_k = y_j\sqrt{(-2\log t_j)/t_j}$, where log denotes the natural logarithm. Then X_k and Y_k are independent normally distributed variables with zero mean and unit variance. Approximately $n\pi/4$ pairs will be constructed in this manner. Finally, for $0 \leq l \leq 9$ tabulate Q_l as the count of the pairs (X_k, Y_k) that lie in the square annulus $l \leq max(|X_k|, |Y_k|) < l + 1$, and output the ten Q_l counts.

On a p processor machine, every processor generates the statistics for a set of n/p points. This is done in parallel without any inter-processor communication. The only communication in this problem is to add the 10 sums from various processors at the end, which is insignificant. Thus, the only optimization we did was to improve the performance of the single node. We now summarize some of the major techniques employed to improve the single node performance. The details can be found in [2].

An Improved Random Number Generator
We used an improved random number generator which utilizes the fused multiply-add unit of RS/6000. On RS/6000 and POWERPC machines, during the multiply-add operation, all 106 bits of the product are added to the 53 bit operand. This results in the best possible accuracy. On a POWER2 node, the new random number generator generates approximately 40 million random numbers per second.

A Table Based Algorithm for Generating Gaussian deviates and their classification
The Gaussian deviates are generated using the function $f(t) = \sqrt{(-2\log t)/t}$. We subdivide the interval (0,1) into a set of discrete points $t_i = ih$ where h is the sub-interval size. We

construct the table of points $f(t_i)$. The function $f(t)$ is a monotonic function in the range of interest $0 < t < 1$. Now $t = 0$ is a singularity of $f(t)$. However, only a very small fraction of the random numbers have t close to zero. Therefore, we handle the t values in first interval separately as a rare event.

The tables $f(t_i)$ is used to decide the bin number l where this random pair lies. Thus, unless we are close to a bin boundary, there is no need to compute $f(t)$ with high precision. Thus for most of the cases when we are not close to a bin boundary we use the table values $f(t_i)$ and $f(t_{i+1})$ bordering $f(t)$ to get the bin number.

Performance Tuning for Power Architecture

A large amount of inefficiency in the generic code supplied by NASA is due to the overheads associated with the conversion from a floating point number to an integer. This conversion is required to get the table index and the bin index. The compiler calls a function routine which takes many cycles. In this conversion, the function routine has to take into account all possibilities including negative integers and overflows. However, in our case, the integers are always small and positive. For a 52-bit positive IEEE floating point number x, $2.0^{52} + x = 2.0^{52} + int(x)$, provided that the arithmetic is done in the chop mode. If this result is stored back into memory, then its low order 32 bits represent the integer part of x, when $x \leq 2^{31} - 1$. These bits are then loaded in a fixed point register. This store/load combination could introduce several cycles of delay and the code must be scheduled to do useful work during this period.

We also did some additional tuning which is generally applicable to high performance RISC workstations [2].

In the NAS parallel benchmark results report, [3], changes were made to the EP benchmark. In Section 2.1 of [3] the benchmark authors state that "the intent of the EP benchmark is to provide an accuracy and performance check on the FORTRAN LOG and SQRT intrinsic ..." and thus they made two changes. Briefly, the changes *disallows* the use of table look-up and also the construction of the composite function SQRT(-LOG(X)). In Tables 2a and 2b of [3] EP timing based on the table look-up approach are given for Cray C90, Cray T3D, IBM SP1, and IBM RS6000-590. Here, for brevity, we only mention that the table based approach is about 4.5 times faster on SP2 machine than the non-table based approach. The results reported in this paper (See Table 2 ahead) are for the non-table based approach.

3 The Multigrid Benchmark

Statement of the MG Problem

The MG kernel is a V-cycle multigrid algorithm used to obtain the approximate solution to the Poisson equation, $\nabla^2 u = v$, on a $256 \times 256 \times 256$ regular grid with periodic boundary conditions and a specified spatial distribution for v. Four multigrid iterations are done, starting with an initial iterate $u = 0$. Each iteration consists of the following two steps, where $K = 8 = \log_2(256)$:

$$\begin{aligned} r &= v - Au && (\textit{evaluate residual}) \\ u &= u + M^K r && (\textit{apply correction}). \end{aligned}$$

Here, M^K denotes the V-cycle multigrid operator [3] and A denotes the trilinear finite element discretization of the Laplacian operator ∇^2.

Figure 1 illustrates the serial implementation of the MG algorithm. Here, k denotes the grid level such that the k-th grid has 2^k points in each dimension, and $z_k \equiv M^k r_k$. The coefficients of the operators A, P, Q, and S shown in Figure 1 are given in [3]. These operators represent three-dimensional, nearest-neighbor, 27-point stencils, but with some coefficients set to zero. A single layer of "ghost points" is added around the exterior boundary of the computational mesh to facilitate evaluation of these stencils for points on the external boundary. Values of the solution at these ghost points are updated using explicit copies of active data points consistent with the specified periodic boundary conditions. These copies are performed after each update (e.g. for $r_k, z_k, \cdots$) on each grid level.

Parallel Implementation

The parallel implementation of the MG kernel is a data parallel algorithm applied at each grid level. The computational grid is subdivided into subdomains using a three-dimensional block data decomposition, with a one-to-one mapping of subdomains to processors. Processors are logically configured in a three-dimensional grid; for example, 32 processors are configured as a $2 \times 4 \times 4$ processor grid. Each processor applies the stencil operations to grid points in its subdomain, requiring interprocessor communication to access data needed for the evaluation of the points on the boundary of the subdomain. This decomposition is applied only on (finer) grids above a specified grid level. For grids at or below this level (i.e. on coarser grids), the distributed data is combined and replicated in each processor, and each processor then does the computation for all points in these coarser grids.

Figure 2 summarizes the parallel implementation of the down cycle (residual restriction) included in Figure 1. In Figure 2, r_k denotes the residual for points for the specified subdomain, and R_k is the residual for all points on level k.

The communication operation involves the exchange of subdomain boundary data with neighboring processors in the logical processor grid. Given a subdomain grid of $N_s \times N_s \times N_s$, a total of $6(N_s)^2 + 12(N_s) + 8$ words must be exchanged with processors holding adjacent subdomains. A naive implementation is to do 26 distinct communication operations, involving message lengths of 1, N_s, and $(N_s)^2$ words. This can be rather inefficient on typical message-passing architectures with point-to-point communication times given by the usual $T_{comm} = \alpha + \beta \times msglen$. The preferred approach is to do only 6 communication operations (two in each of three coordinate dimensions) involving $(N_s + 2)^2$ words for each operation. Note that single "cornerpoint" values are moved in successive communication steps as part of the much longer messages. Furthermore, in certain phases of the algorithm, it is necessary only to communicate in the positive (or negative) coordinate directions, therefore requiring only 3 distinct communication steps. All such communication is implemented using non-blocking, double-buffered, point-to-point send and receive operations. This communication operation also serves to enforce periodic boundary conditions, and is invoked at precisely the same phases in the parallel algorithm as the copy operation is done in the serial implementation.

The gather operation requires interprocessor communication in order to concatenate the distributed data and then replicate the concatenated data in each processor. In practice, we use $k_{cutoff} = 3$, and we observe that the time necessary to perform the gather operation is negligible since $k_{cutoff} << K$ $(= 8)$.

An analogous procedure is used during the up cycle, in which the global values are distrib-

```
$u = u^{(0)} \equiv 0$                               $(initialize\ solution)$
$for\ i = 1, NITER\ do$                              $(do\ NITER\ (= 4)\ multigrid\ iterations)$
    $r_K = v - Au^{(i-1)}$                           $(evaluate\ residual\ on\ fine\ grid)$

    $for\ k = K,\ K-1,\ \cdots, 1\ do$               $(down\ cycle)$
        $r_k = P\ r_{k+1}$                           $(restrict\ residual)$
    $enddo$                                          $(end\ down\ cycle)$

    $z_1 = S\ r_1$                                   $(apply\ smoother\ on\ coarsest\ grid)$
    $for\ k = 1, \cdots, K\ do$                      $(up\ cycle)$
        $z_k = Q\ z_{k-1}$                           $(prolongate)$
        $r_k = r_k\ -\ A\ z_k$                       $(evaluate\ residual)$
        $z_k = z_k\ +\ S\ r_k$                       $(apply\ smoother)$
    $enddo$                                          $(end\ up\ cycle)$
    $u^{(i)} = u^{(i-1)} + z_K$                      $(apply\ correction\ on\ fine\ grid)$
$enddo$                                              $(end\ multigrid\ iterations)$
```

Figure 1: Outline of serial MG algorithm

```
$for\ k = K,\ K-1,\ \cdots, 1\ do$                   $(down\ cycle\ for\ this\ subdomain)$
    $if\ k > k_{cutoff}\ then$
        $r_k = P\ r_{k+1}$                           $(restrict\ subdomain\ residual)$
        $communicate(r_k)$                           $(stencil\ communication)$
    $endif$
    $if\ k = k_{cutoff}\ then$
        $gather(r_{k+1} \rightarrow R_{k+1})$        $(form\ global\ residual\ in\ each\ processor)$
        $R_k = P\ R_{k+1}$                           $(restrict\ global\ residual)$
    $endif$
    $if\ k < k_{cutoff}\ then$
        $R_k = P\ R_{k+1}$                           $(restrict\ global\ residual)$
    $endif$
$enddo$                                              $(end\ down\ cycle)$
```

Figure 2: Outline of parallel algorithm for down cycle

uted (i.e. $distribute(R_{k-1} \rightarrow r_{k-1})$) at the cutoff grid level. This operation does not require interprocessor communication since each processor merely resumes computation on only the data in its subdomain. Stencil communication is required after each update step as in the down cycle.

4 The Conjugate Gradient Benchmark

Statement of the CG Problem

The CG benchmark "computes an iterative approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. Within the iteration loop, the core procedure is to solve a linear system of equations via the conjugate gradient method (CGM). This kernel is typical of unstructured grid computations in that it tests irregular long distance communication, employing unstructured matrix-vector multiplication [4]."

The inner iteration of the CGM computes the product of a sparse matrix with a vector:

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

and then uses the result $\mathbf{y}$ to update the $\mathbf{x}$ vector. Unless P, the number of processors, is very large, the cost of the update is insignificant. The size N of this problem is the length of $\mathbf{x}$. Another parameter, k, measures the sparsity of the matrix; each row and column of $\mathbf{A}$ has about k nonzero elements. For the class B (class A) problem, N is $75,000$ $(14,000)$ and k is approximately 183 (133).

Parallel Implementation

Our tuning of this kernel had three components:

1. selecting an approach that minimizes communication and computation costs,
2. using the memory hierarchy of the processing nodes effectively, and
3. striving for maximal use of the processors on the inner loop.

The choice of a general approach was considerably simplified by the decision of the NAS Benchmark Committee to disallow approaches based on factoring the sparse matrix $\mathbf{A}$. The choice narrows to either a one-dimensional or a two-dimensional decomposition of this matrix. Each decomposition requires the same amount of computation but the one-dimensional decomposition does not scale well [8]. It requires communicating NP values per matrix vector product as opposed to about $2NP^{\frac{1}{2}}$ values in the two-dimensional approach.[1]

Lewis and van de Geijn [8] give a very nice communication algorithm for the two-dimensional decomposition approach. The P processors are logically arranged in a $P_1 \times P_2$ grid, where both P_1 and P_2 are powers of two. The data are partitioned so that the processor in the i-th row and j-th column owns length–N/P subvectors $\mathbf{x}_{ij}$ and $\mathbf{y}_{ij}$ of $\mathbf{x}$ and $\mathbf{y}$, as well as an $N/P_1 \times N/P_2$ submatrix $\mathbf{A}_{ij}$ of $\mathbf{A}$. It computes the product:

$$\mathbf{y}_i^j = \mathbf{A}_{ij}\mathbf{x}_j$$

[1]To simplify analysis, each processor is assumed to communicate with itself. The program requires slightly less communication.

where $\mathbf{x}_j$ is the concatenation of all the $\mathbf{x}_{ij}$'s for a fixed j. Construction of the $\mathbf{x}_j$'s from the $\mathbf{x}_{ij}$'s is done by an elegant hypercube-based communication strategy. It requires $\log_2 P_1$ rounds of communications, with a combined length of N/P_2 doublewords per processor. After each processor computes its $\mathbf{y}_i^j$, accumulation and distribution of the $\mathbf{y}_{ij}$'s is accomplished by a similar communication strategy. Altogether each global matrix vector multiplication requires Nk multiply-adds, communication of $N(P_1 + P_2)$ values, and $P \log_2 P$ messages.

The key to effective use of the memory hierarchy on each node is keeping $\mathbf{x}_j$ in cache during the matrix-vector multiplication. This can be achieved by an appropriate factoring of the number of processors P into P_1 and P_2. This argues for making P_2 large, but if $P_1 \ll P_2$ then the communication cost will mount. In addition, there is a measurable loop overhead for the inner loop that gets amortized over k/P_2 multiply-adds. This also argues for keeping P_2 from getting too big. Careful balancing of the various costs results in the choices for P_2 given in the following table:

	The total number of processors, P.									
Cache Size	1	2	4	8	16	32	64	128	256	512
32K	1	1	2	8	8	8	8	16	16	16
64K	1	1	4	8	8	16	16	16	16	16
128K	1	2	4	8	8	8	8	8	16	16
256K	1	2	4	4	4	4	8	8	16	16

Table 1. The number of processors in a row, P_2 for the class B problem.

The inner loop computes the dot product of $\mathbf{x}_j$ with one row of the local submatrix. This entails about k/P_2 multiply-adds. Since the matrix is sparse, each multiply-add requires three loads: a value ¿from $\mathbf{A}_{ij}$, the value's column index, and the corresponding value from $\mathbf{x}_j$. If the column index were used to subscript the vector directly then it must be shifted (multiplied by the size of a doubleword) in order to be used. This additional fixed-point cycle can be eliminated using the Fortran 90 pointer construct. Thus, each multiply-add requires 3 loads, each taking one cycle on the POWER architecture. On the POWER2 architecture, which has two fixed-point units, the three loads take 1.5 cycles.

Each floating-point unit is capable of executing a multiply-add every cycle, but the result is not ready until the third cycle. Thus, accumulation of a dot-product into a single register requires 2 cycles per multiply-add. This would be a bottleneck on the POWER2 architecture, so the alternate terms of the dot-product are accumulated in two separate registers which are added together at the end of the loop. Assuming the input vector fits in cache, the resulting code can run on the POWER (POWER2) architecture at about 3 cycles (1.5 cycles) per multiply-add, plus the cost of cache misses on the data structures representing the $\mathbf{A}$ matrix. This cost depends on the cache size, but ranges from 3/8 cycles (on a wide-node SP2) to 3/2 cycles (on an SP1) per multiply-add.

5 The 3-D Fourier Transform Benchmark

In this benchmark a Poisson partial differential equation (PDE) is solved using 3-D forward and inverse discrete Fourier transform (DFT). Basically, this benchmark first requires a forward 3-D fft computation. Then in a loop the transformed data is multiplied by a coefficient array followed by an inverse 3-D fft computation. We now give a brief description of the benchmark for the Class B problem size. For details one can refer to [4].

Statement of the FFT Problem

Initialization phase:

Set $n_1 = 512$, $n_2 = 256$, $n_3 = 256$, and $\alpha = 10^{-6}$. Generate $2 \times n_1 \times n_2 \times n_3$ 64-bit real numbers using the pseudorandom generator outlined in [4]. Assign these real numbers to a complex array, $U(0 : n_1 - 1, 0 : n_2 - 1, 0 : n_3 - 1)$ such that two consecutive real elements are assigned to an element of U.

Forward phase:

Compute $V(0 : n_1 - 1, 0 : n_2 - 1, 0 : n_3 - 1)$, the 3-D DFT of the array U using the *FFT* algorithm.

Inverse phase:

```
for $t = 1$ to $20$ do
multiplication step
    for $0 \leq i < n_1$, $0 \leq j < n_2$, $0 \leq k < n_3$
    $W(i, j, k) = e^{-4\pi\alpha^2(i'^2+j'^2+k'^2)t} V(i, j, k)$
    Where $i'$ is defined as $i$ for $0 \leq i < n_1/2$ and
    $i - n_1$ for $n_1/2 \leq i < n_1$. Similar definitions hold
    for $j'$ with $n_2$ and $k'$ with $n_3$.
inverse 3-D DFT step
    Compute $X(0 : n_1 - 1, 0 : n_2 - 1, 0 : n_3 - 1)$,
    the 3-D inverse DFT of the array $U$
    using the FFT algorithm.
checksum step
    Compute the complex checksum
    $\sum_{i=0}^{1023} X(q, r, s)$, where
    $q = i \bmod n_1$, $r = 3i \bmod n_2$, and $s = 5i \bmod n_3$.
end for
```

Parallel Implementation

The major effort in parallelizing the above benchmark is to efficiently implement the 3-D inverse DFT computation inside the for loop. An over all description of our algorithm is as follows. Initially the 3-D data is assumed to be distributed across the processors along the third dimensions. We first do FFT computation along the first dimension. This computation does not require any communication. Second, we move data between the processors such that the new distribution across the processors is along the first dimension. Third, we do FFT computation along the second and third dimensions locally on a node without any communication. The implementation details can be found in [1].

On a single node we use FFT routines of the Engineering and Scientific Subroutine Library (ESSL) [13]. We implemented the complete benchmark using 1-D FFT routines along with some data movement routines blocked for cache. This results in a better cache behavior of the RS/6000 node.

6 The Integer Sort Benchmark

Statement of the IS Problem
This benchmark computes ranks, $r_0, r_1, \ldots, r_{n-1}$, for a given set of n integer keys, $k_0, k_1, \ldots, k_{n-1}$ with m-bits each. A rank r_i is the position of key k_i in the sorted ascending order. In other words $r_i \leq r_j$ implies $k_i \leq k_j$ for $0 \leq i, j \leq n-1$. Here, we focus on the ranking problem (m, n) on a p processor machine with $p \geq n/2^m$ and $n \geq 2^m$. The two ranking problems for the benchmark are: $(19, 2^{23})$ for Class A, and $(21, 2^{25})$ for Class B. The choice of an algorithm depends on the relative values of m, n, and the number of processors p. The proposed algorithm is efficient for any number of processors. However, the communication part of the algorithm could be improved for $p < 16$. For smaller values of p, the communication part can be significantly reduced by sending the count arrays from each processors, instead of sending the key values. For $p < 16$, at each node, the number of keys are more than the size 2^m of the total count array. Thus it is more efficient to send counts instead of keys.
Parallel Implementation
The central idea of the algorithm is as follows. At each node, we sort keys into a certain number of buckets based on some key bits. For many reasonable distribution of keys, sorting on the middle bits assures nearly uniform key density distribution across all buckets. If there are nb buckets, nb/p of them are sent to each processor using a global transpose communication routine. Here, we assume p to be a power of two. Each processor receives buckets from all processors corresponding to some of the middle key bits. At this stage, for each key value, there are on the average $n/2^m$ keys. For the NAS IS problems this average is 16. Therefore, for efficiency, we implement a distribution count sort [6, 5]. At the same time, at no additional cost, we implement sub-bucket sorting on the high order bits. Each processor assigns sub-bucket ranks to all keys received by it. These ranks along with all the sub-bucket counts are sent back to the originating processor using another global transpose of the same size. The originating processor computes the final rank by adding the sub-bucket rank to the global rank offset for that sub-bucket. In the region of interest, i.e., $n/2^m \geq 1$, both computing and communication scale with the number of processors.

7 Results

To date, two different specifications of the NAS parallel benchmarks have been defined. Class A refers to the original problem dimensions, while Class B denotes a more recent specification in which problem sizes have been increased by a factor of 4 in most of the kernels [3].

Table 2 summarizes the performance results for the Class B kernels. As is the convention, results are reported as ratios to the respective single-processor Cray C-90 time [3]. However, in the caption of Table 2 we have added the single-processor Cray C-90 times for each of the benchmarks. This allows one to compute wall clock times for each of the benchmarks. The SP2 results are for a wide-node system with 128 MB of memory per node, and were obtained using the user-space communication protocol in the IBM Message Passing Library [14] included as part of the IBM Parallel Environment [15]. For comparison, we have also included the performance results for other scalable parallel machines; these results are taken from [3, 16]. Note that the IBM SP2 EP results are based on the latest NAS rules which also require additional checksums [3]. The Intel Paragon performance results correspond to the better of the two environments, OSF1.2 or SunMos, wherever applicable.

In general, the SP2 performance per processor exceeds that of the other parallel machines shown in Tables 2 for all processor configurations for which SP2 results are available. This performance is due primarily to the very high sustained performance of the SP2 wide nodes, combined with the relatively high sustained interprocessor communication bandwidth for these problems. One of the referee's noticed that for the EP benchmark (Table 2), which requires very little communication, the Intel Paragon performance is almost as good as that of the SP2, whereas for the more communication-intensive benchmarks in Table 2, the Paragon's performance is much worse than that of the SP2. Here is an explanation. First, in [16, p.5] the NAS benchmark authors note that the SunMos-turbo operating system for the for the Paragon allows both i860 processors on the node to be used for computation (in regular SunMos and OSF the second processor is used purely for communication). Additionally, Intel Paragon results in Table 2 do not include the computation of the two new check sums which adds extra time to the IBM numbers. Also, NAS ground rules require that only official vendor libraries be used for the intrinsic functions such as square root and log. Recently, IBM has made available an alternate library. With the new library we have been able to decrease the IBM SP2 EP numbers in Table 2 by nearly a factor of two.

8 Conclusion

In this paper, we have discussed the novel techniques used to implement the NAS kernel benchmarks on the IBM SP2 scalable parallel system. The central idea of these techniques is to match the implementation of these kernels with the underlying architecture of the SP2. Performance results have shown that the SP2 implementations compare very well with results currently available on other scalable parallel machines.

Acknowledgement: The authors are indebted to Mark Smith (Power Parallel Systems, Kingston, NY) for performing the SP2 runs used to obtain the results for this paper. Agarwal, Gustavson and Zubair acknowledge Mahesh Joshi, Alok Kothari and Prasad Palkar for their help in running and tuning various versions of the EP, IS and FFT kernels.

References

[1] R. Agarwal, F. Gustavson, and M. Zubair, "An Efficient Parallel Algorithm for the 3-D FFT NAS Parallel Benchmark", *in the proceedings of SHPCC '94'*, pp.129-133.

[2] R. Agarwal, F. Gustavson, and M. Zubair, "A Very High Performance Algorithm for NAS EP Benchmark", *High-Performance Computing and Networking*, Edited by Wolfgang Gentzsch and Uwe Harms, Springer-Verlag, 1994, pp.164-169.

[3] D. Bailey, E. Barszcz, L. Dagum, and H. Simon, "NAS Parallel Benchmark Results 3-94," Report RNR-94-006, NASA Ames Research Center, March 1994.

[4] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan,

and S. Weeratunga, "The NAS Parallel Benchmarks," Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[5] F.G. Gustavson, "Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition", *ACM Transaction on Mathematical Software*, Vol. 4, No. 3, September 1978, pp.250-269.

[6] D.E. Knuth, *The Art of Computer Programming, Volume III: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

[7] D.E. Knuth, *The Art of Computer Programming, Volume II, 2nd Edition: Semi Numerical Algorithms* , Addison-Wesley, Reading, Mass., 1973.

[8] J.G. Lewis, and R.A. van de Geijn "Distributed Memory Matrix-Vector Multiplication and Conjugate Gradient Algorithm," Proceedings of *Supercomputing '93*, pp. 484-492.

[9] V.K. Naik, "Performance of NAS Parallel Application-Benchmarks on IBM SP1", *in Proceedings of Scalable High Performance Computing Conference, IEEE*, 1994.

[10] V.K. Naik, "A Comparative Study of the NAS Parallel Application Benchmarks", *to appear in the Proceedings on Parallel CFD'94.*

[11] V.K. Naik, "An Efficient Implementation of NAS Parallel Benchmark BT on Distributed Memory Systems", Submitted to IBM Systems Journal, 1994

[12] D.H. Bailey and P.O. Frederickson, "Performance results for two of the NAS parallel benchmarks," *in the proceedings of Supercomputing '91*, pp.166-173, 1991.

[13] ESSL, Guide and Reference, Order number SC23-0526-01, IBM Corp., 1994.

[14] "IBM AIX Parallel Environment - Parallel Programming Reference", SH26-7228-0, IBM Corporation, 1994.

[15] "IBM AIX Parallel Environment - Operation and Use", SH26-7230-0, IBM Corporation, 1994.

[16] D. Bailey, E. Barszcz, L. Dagum, and H. Simon, "NAS Parallel Benchmark Results 10-94," NAS Technical Report NAS-94-001, NASA Ames Research Center, October, 1994.

Kernel	No. of Procs	IBM SP2	Cray T3D	TMC CM-5E	Intel Paragon
EP	8	1.20			
	16	2.40	1.32		
	32	4.84	2.64	3.95	
	64	9.54	5.29	7.85	8.75
	128	19.30	10.58	15.97	17.66
	256		21.15		34.24
MG	8	1.31			
	16	2.50	0.57		
	32	4.60	1.24	1.8	
	64	8.34	3.01	3.3	0.9
	128	14.36	5.75	5.6	1.8
	256		10.49		2.8
CG	8	0.74			
	16	1.31	0.21		
	32	1.91	0.41	0.3	
	64	2.88	0.74	0.6	
	128	4.59	1.44	1.3	0.9
	256		2.45		1.76
FT	8				
	16	1.33			
	32	2.4		1.4	
	64	4.47	3.12	2.8	
	128	8.75	6.08	3.7	2.3
	256		11.70		4.2
IS	8	0.65			
	16	1.17			
	32	1.88	0.51	0.4	
	64	3.64	1.00	0.8	1.1
	128	6.49	1.97	1.5	1.8
	256		3.95		2.2

Table 2: Performance results in ratios to C-90/1 for Class B NAS Kernel Benchmarks
The C-90/1 times (in seconds) used to compute these ratios are: 185.26 (EP), 37.77 (MG), 122.90 (CG), 127.44 (FT) and 12.92 (CG).