The NAS Parallel Benchmark Kernels in MPL

Adam Ferrari, Adrian Filipi-Martin, and Soumya Viswanathan

[ferrari,adrian,vsv5y]@virginia.edu

Department of Computer Science, University of Virginia

Charlottesville, Virginia 22903

CS-95-39

September 1995

Abstract

The Numerical Aerodynamic Simulation (NAS) Parallel Benchmarks are a set of algorithmically specified benchmarks indicative of the computation and communication needs of typical large-scale aerodynamics problems. Although a great deal of work has been done with respect to implementing the NAS Parallel Benchmark suite on high-end vector supercomputers, multiprocessors, and multicomputers, only recently has the possibility of running such demanding applications on workstation clusters begun to be explored. We implemented a subset of the NAS benchmarks using the Mentat Programming Language, and ran performance tests on a cluster of workstations using the Mentat system. We compared our performance results to previous NAS benchmark tests using the Parallel Virtual Machine (PVM) system in a similar hardware environment. We found that due to algorithmic improvements, efficient communications provided by the Mentat system, and low introduced overheads even at the higher level of programming abstraction provided by Mentat, we observed significantly improved performance in a number of cases.

1. Introduction

The Numerical Aerodynamic Simulation (NAS) Program is an ambitious effort to significantly advance the state of high performance computing in order to facilitate research and development in the area of aerospace engineering.Towards this end, NAS has put forth the NAS Parallel Benchmarks, a collection of five kernels and three full applications indicative of the computation and communication needs of typical large-scale aerodynamics problems. These benchmarks are unique in their "pencil and paper" specification - the requirements, input sets, and reference results are describe entirely in [1]. This strategy is desirable in that it allows the programmer freedom to exploit the strong points of a target system while not being constrained by code portability issues. The algorithmic specification does introduce additional demands on programmer expertise, however, which can seriously affect the performance achieved on a given system.

Although a great deal of work has been done with respect to implementing the NAS Parallel Benchmark suite on high-end vector supercomputers, multiprocessors, and multicomputers, only recently has the possibility of running such demanding applications on readily available workstation clusters begun to be explored. Previous work in this area using the PVM[2] system demonstrated the feasibility of achieving performance levels within an order of magnitude of a Cray Y/MP-1 using a cluster of 8 mid-range workstations[3]. Unfortunately, the implementation of these relatively complex parallel applications is a difficult task in a low level programming environment such as PVM. In PVM, the programmer implements a set of cooperating processes which interact using send and receive primitives. Managing the state of such a collection of distributed, asynchronous processes "manually" is naturally quite difficult.

Given the great deal of computing power available in relatively inexpensive workstation clusters, and the degree to which the available performance is constrained by the complexity involved in low level message passing systems, it is clearly desirable to investigate the utility of a higher level programming model for use in such environments. A natural candidate for such an investigation is the Mentat system and associated Mentat Programming Language (MPL). MPL is an object-oriented language based on C++ intended to simplify the design and implementation of parallel programs. The programmer specifies C++ classes (with slight modifications) whose member functions may operate with various degrees of concurrency, yet may be used in the same natural fashion as sequential C++ routines[4]. The MPL compiler translates Mentat Programming Language code, and generates standard C++ source with embedded calls to Mentat system library routines. When the program runs, these embedded library calls interact with the Mentat run-time system to handle data dependencies, message passing, scheduling, and synchronization. This run-time system employs a macro-dataflow model which defines program execution as a dynamically constructed graph of data dependencies and computation vertices.

This model offers an attractive balance between exploiting the programmer's knowledge of the application and the compiler's ability to manage a complex state space with communication and synchronization constraints. We investigated the utility of this environment by implementing a subset of the NAS Parallel Benchmark suite in MPL. The use of a higher level programming system typically implies additional overhead, but often allows algorithmic refinements that would have been difficult to perceive at a lower level. It was our hope to find cases where higher levels of sophistication allowed performance improvements over the low-level, raw speed strategy.

We found that our results were quite positive. In some cases, performance reported for the low level PVM implementation was quite similar to our own, as no significant basic algorithmic improvements were found. In other cases, most notably the integer sorting example, our performance significantly exceeded that of PVM. These speedups were due to algorithmic optimizations and refinements. Our overall conclusion was that, while the raw computational power available on workstation clusters is quite significant, harnessing this power is a difficult task which will be made more tractable by a high level programming system such as Mentat. Before discussing our performance results at length, we move to describing the benchmark kernels implemented and our experiences porting them.

2. Overview of Implemented Kernels

We implemented a subset of the NAS kernel level benchmarks consisting of the Embarrassingly Parallel Kernel (EP), the Integer Sorting Kernel (IS), the Multigrid Solver Kernel (MG), and the Conjugate Gradient Solver Kernel (CG). We now describe each of these benchmark kernels and our experiences implementing them in MPL.

In general, we took the strategy of modifying working versions of the code provided by NAS in order to ensure adherence to the NAS rules and specifications, and in hopes of avoiding duplication of much of the sequential numeric code already written for this benchmark set. We obtained versions of the entire benchmark suite implemented sequentially, for the Intel iPSC/860, and for PVM. The iPSC/860 and PVM versions are in most cases nearly identical. In some cases, these “legacy” codes were written in Fortran77 and were not suitable for code re-use. In all cases, however, the NAS implementations were a great aid in determining opportunities for parallelism. In all cases, we chose to parallelize the same elements of the benchmark as the NAS implementations, the difference primarily lying in the mechanisms used for expressing parallelism.

2.1. Kernel EP

Problem Description

The first and simplest benchmark implemented was the Embarrassingly Parallel Kernel (EP). The algorithm required by EP involves the generation of pairs of Gaussian random deviates and the tabulation of the number of pairs in successive square annuli. This process is performed by selecting random pairs (*x*, *y*) where *x* and *y* are evenly distributed on the interval (-1, 1). If the inequality $t = x^2 + y^2 \leq 1$ is satisfied, independent Gaussian deviates *X* and *Y* with mean zero and variance one are generated by letting

$$X = x\sqrt{\frac{(-2\log t)}{t}} \qquad \text{and} \qquad Y = y\sqrt{\frac{(-2\log t)}{t}} \ .$$

After all such pairs are generated, the number of pairs lying in unit width square annuli centered at the origin (within 10 units on each dimension) must be tabulated and output along with ΣX and ΣY over all *X* and *Y*. All operations from start to finish should be timed.

This algorithm is termed “Embarrassingly Parallel” as it requires minimal communication among processors operating in parallel to perform the required calculations. Since there are no inter-iterational dependencies, processors may simply execute a portion of the iterations in parallel, and combine results at the end. This problem is representative of a large class of Monte Carlo simulations.

Code Description

Our Mentat version of EP employs a regular Mentat object class with a single public member function to perform the prescribed computation. This mentat class is called `ep_object` and its public interface is the single member function `kernelEP()`. This member function accepts a record containing the range of iterations to perform, and returns a record containing the 10 sums described above and the object execution time. The `ep_object` has private member functions which implement the random number generation process described in [1].

The user interface to our Mentat EP kernel is implemented in the `ep_driver` program. This simple program accepts user input of the problem size parameters, generates the work partitions for the specified number of object invocations, invokes the `kernelEP()` member function the appropriate number of times, then gathers reference counts and performance results.

This simple version of the EP benchmark could be improved to allow objects to perform dynamic self scheduling or some other load balancing scheme, as load balance is certainly the primary issue for good performance here. Planned enhancements to the Mentat scheduling facilities in the near future (such as sender initiated dynamic scheduling for regular objects) may address this problem transparently to the use code.

2.2. Kernel IS

Problem Description

The Kernel IS benchmark is a parallel sort of *N* integer keys in a specified range meant to capture characteristic behaviors common to "particle method" codes. The keys to be sorted are generated by a sequential algorithm and then uniformly distributed among any worker processors, except in the case that the number of keys does not divide evenly. In this case, one worker will receive slightly fewer keys, as specified in the NAS benchmark. The initial distribution of these keys over the processors is rigidly specified as it can greatly affect the performance of the benchmark.

Throughout the duration of the benchmark, the value associated with each key may never move from its originating worker's memory. Instead, copies of the keys are transferred, and ranks for each key are computed (i.e. what the resulting key index would be in a sorted array). The resulting ranks need not correspond to a stable sort. An outline of the algorithm is:

```
1. Sequentially generate the keys on a single worker.
2. Distribute the keys to the other workers.
3. Begin timing.
4. do $i$ = 1...10
        a. Modify the sequence of keys, making the following two changes:
                $\mathrm{Key}_i \leftarrow i, \quad \mathrm{Key}_{i+10} \leftarrow 2^{19} - i$
        b. Compute the tank of each key, as described below.
        c. Perform a partial verification test.
5. End timing.
```

As indicated, this ranking process must be performed 10 times during the course of the benchmark, using an NAS specified permutation of the array for each iteration. As a direct result of the nature of this algorithm, it is expected that the majority of its run-time in a distributed environment will involve network communication. The pattern of communication is a fully connected graph with communication typically being frequent and relatively low-volume. As a result, this algorithm is better suited to shared memory systems. Despite this fact, the benchmark is very useful for comparing the relative communication overheads of different distributed memory parallel processing systems.

Code Description

The basis of our implementation was the PVM version of the kernel used to generate results reported in [2]. This code was written in ANSI C and thus provided us with a direct launching point for the C++/MPL version. During the course of the implementation, we devised a number of refinements which resulted less overall communication. The work of the benchmark is encapsulated in the sequential mentat class, `is_object`. A front end driver program, `is_driver`, was implemented to interact with the benchmark user, initialize worker objects, and gather performance results.

It is worth noting that because of the specified key generation scheme, there is an uneven distribution of keys. Because the range is 16 times smaller than the number of keys, and the keys are generated using a linear congruential recursion random number generator, there will be more numbers in the middle of the range. As we will describe, this has load balance implications for our implementation which are not addressed. This is an area of possible improvement in our version of IS.

The basic job of the `is_object` is to perform the key ranking phase described above in parallel. In order to achieve data parallel style execution of this activity, objects are assigned subranges of the global key range which they are responsible for ranking. Prior to the ranking phase, the objects send copies of all local keys in the appropriate range to the object at which they should be ranked. Similarly, each object receives a set of keys from each other object containing keys in the locally assigned range. After the rankings are performed by each object for their respective range, the results are returned to their owners and the process is complete. It is in this basic method where we found the PVM version used un-needed messages requiring extra synchronization. Key member functions of the `is_object` class include:

`is_init()`

The purpose of this method is to allow the `is_driver` process to communicate the problem parameters to the worker objects.

`create_seq()`

This method is invoked on a designated worker object to perform the sequential key generation algorithm required by the benchmark. After the keys are generated, they are distributed to the other workers using the `give_sequence`() member function.

`nas_is_benchmark()`

This member function performs the main loop of the algorithm described above. There are three main activities performed during each iteration in order to facilitate the parallel ranking strategy. First, the local keys are partitioned into sub-arrays based on the object by which they should be ranked. After being partitioned, the keys are sent to their appropriate destination using the `give_keys`() member function. Next, after all of the keys in the object's ranking range have been obtained, the object ranks the keys. This phase represents the computation element of the benchmark – a relatively undemanding bucket sort which can easily result in a very fine granularity. The third phase is the transfer of the ranks to the appropriate key owners.

We observed correct reference counts using our version of IS on smaller problem sizes, but had trouble running the largest problem size due to memory constraints. This problem was also encountered by the PVM group that reported results for a reduced IS problem size in [2].

2.3. Kernel MG

Problem Description

The Multigrid Solver Kernel (MG) involves executing four iterations of the V-cycle multigrid algorithm described below to obtain an approximate solution u to the discrete Poisson problem

$$\nabla^2 \cdot u = v$$

on a three dimensional grid with specified boundary conditions (possible grid dimensions are 32, 64 and 128 and 256). The algorithm starts by setting $v = 0$ except at certain NAS specified points[1]. The iterative solution begins by setting $u = 0$. Each of the four iterations consists of the following two steps, in which $k = \log_2(grid\_size)$:

$$r = v - (A \cdot u)$$
$$u = u + M^k \cdot r$$

Here M^k denotes the V-cycle multigrid operator, defined to by:

```
$z_k = M^k r_k$:
    if $k > 1$
        $r_{k-1}$   =  $P\ r_k$              (restrict residual)
        $z_{k-1}$   =  $M^{k-1} r_{k-1}$     (recursive solve)
        $z_k$       =  $Q\ z_{k-1}$          (prolongate)
        $r_k$       =  $r_k - A\ z_k$        (evaluate residual)
        $z_k$       =  $z_k + S\ r_k$        (apply smoother
    else
        $z_1$       =  $S\ r_1$              (apply smoother)
```

The coefficient vectors A, P, Q and S here are constants specified by NAS. Each vector has four coefficient values c_0, c_1, c_2 and c_3. The c_0 coefficient is the central coefficient of the 27-point operator, when these coefficients are arranged as a 3x3x3 cube. Thus c_0 is the coefficient that multiplies the value at the grid-point (i, j, k), while c_1 multiplies the six values at grid points which differ by one in exactly one index, c_2 multiplies the next closest twelve values that differ by one in exactly two indices, and c_3 multiplies the eight values located at grid points that differ by one in all three indices. The residual norm is then calculated using the formula:

$$\|r\|_2 = \left[\left(\sum_{i,j,k} r_{i,j,k}\right)/N^3\right]^{1/2}$$

The value of this residual should agree with the reference value provided by NAS.

Code Description

The framework of our Mentat MG implementation is a master-worker model employing data parallel execution. A main driver program called `mg_driver` interacts with the user, creates worker objects, initializes their state and then collects results and reports residuals and performance statistics. The real work of the kernel is encapsulated in the `mg_object` sequential mentat class. A collection of these worker objects cooperate to perform the multigrid operations described above. The data movement requirements of this algorithm form a logical ring where communication is primarily to and from nearest neighbors. The problem data consists of $n \times n \times n$ grids which are partitioned into $k \times n \times n$ chunks where $k = n/N$, N being the number of objects for a given run (i.e. partitioning is done along one dimension only). Any operations on the border values of a worker objects's data set require values assigned to another processor. This implementation resolves this issue by keeping a neighboring processor's border points "in shadow" using a scheme to update them before they go out of date.

The `mg_object` class implements a worker object that will cooperate with other `mg_objects` in solving the multigrid problem specified. Thus `mg_object` contains private variables and member functions that are required for the execution of the kernel including three dynamically allocated three dimensional, double precision floating point arrays that represent *u*, *v* and *r*. Private data members also include four statically allocated, single dimensional, double precision floating point arrays that represent the coefficient vectors *A*, *P*, *Q* and *S*. There are also member variables for the dimensions of the grid, dimensions of each worker's data set, number of objects being used, and Mentat names of other workers. The member functions include an initialization function, a function to begin the iterative process, functions for each phase of the V-cycle multigrid, routines for communication of border values and functions to gather and integrate norm values computed by each worker. The key member functions of the `mg_object` include:

`mg_init`():

This function initializes internal data of the `mg_object` class, including the dimension of the grid (*N*), and the number of objects used (*p*).

Once the data members have been initialized, the three 3-dimensional, double precision arrays are allocated with dimensions $my\_N \times N \times N/p$. The coefficient arrays *A, P, Q* and *S* are initialized with values specified in the benchmark. The *v* grid is initialized with zeroes in all positions except where specified by the benchmark, where the initial value is either +1.0 or -1.0 (see the Kernel Description section for details).

`kernelMG`():

This function encompasses the main operations required to solve the multigrid problem. First the norm is computed prior to beginning the iterative process. The next step is to go through four iterations of two main steps viz. evaluating the residual and applying correction via the V-cycle multigrid operator. The evaluation of the residual is done by function `resid`()

and function multigrid() carries out the recursive multigrid operations. At the end of four iterations, the norm is computed again and the results are reported.

Once `kernelMG()` is done executing, the kernel operations have been performed and the required norm has been computed.

`multigrid():`

`Multigrid()` is the implementation of the V-cycle multigrid algorithm specified in the definition of M^k above. This is a recursive function, where the recursion stops when $k \leq 1$. Each phase of the operator corresponds to a different function call. From the definition of M^k, we see that if $k \leq 1$, the only operation to be carried is smoothing. This corresponds to the function `smooth()`. This function takes two grids z and r and adds the product of the coefficient vector S and r to z. The product is carried out as a 27-point operator as explained in the Kernel Description section.

If $k \geq 1$, then a series of calls are made to functions corresponding to the operations described above. The first call is to `restrict()`. This function takes in two grid parameters r_k and r_{k-1}. The size of r_{k-1} is 1/8 the size of r_k. The residual is then restricted by taking the product of coefficient vector P with r_k. Since the dimensions of r_{k-1} and r_k are not the same, only every other value of r_k is considered in the product.

After restricting the residual, a recursive call is made to `multigrid()`. The function "recurses in" till $k \leq 1$. The function then "recurses out" and a function call is made to `prolongate()`. `Prolongate()` again is a straightforward coding of the prolongate step specified in Table 2. The next call is to `resid()`, which evaluates the residual and then a call to `smooth()` is made. Once the "recursion out" process is complete, the function `multigrid()` ends.

Since this is a stencil problem, all the communication takes place between neighboring processors and during computation of border values. A worker running on processor *p* will be communicating with workers on processors *p-1* and *p+1*, where additions are done modulo number of processors[1]. Each worker object has a "shadow region" which contains neighboring values that it might need and these values are continuously updated. At end of each phase of the V-cycle multigrid operation, border values have to be communicated.

2.4. Kernel CG

Problem Description

The NAS Conjugate Gradient Kernel (CG) estimates the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of non-zero elements. The basic algorithm employed towards this end is:

```
$x = [1, 1,..., 1]^T$;
(start the timer here)
**do** $it = 1, niter$
    Solve the system $Az = x$ and return $\|r\|$ as described below
```

1. Note: The terms worker *p* and processor *p* are used interchangeably

```
        $\zeta = \lambda + 1/(x^T z)$
        Record $it$, $\|r\|$, and $\zeta$
        $x = z/\|z\|$
od
(stop the timer here)
```

This process results in the computation of the eigenvalue estimate, ζ and the residual, $\|r\|$ on each iteration. The final values computed for these should conform to NAS supplied values. The solution to the system $Az = x$ is to be performed in the following way

```
$z = 0, r = x$
$\rho = r^R r$
$p = r$
do $i = 1, 25$
        $q = Ap$
        $\alpha = \rho/(\rho^T q)$
        $z = z + \alpha p$
        $\rho_0 = \rho$
        $r = r - \alpha q$
        $\rho = r^T r$
        $\beta = \rho/\rho_0$
        $p = r + \beta p$
od
compute residual norm given by $\|r\| = \|x - Az\|$
```

They key opportunities for parallelism in this process are the matrix-vector multiplication (e.g. the first line of the loop above), dot product (e.g. the second line of the loop above), and vector-vector addition operations (e.g. the third line of the loop above). NAS rules require that the sparse array representation data structures used in their sample implementation be used, as well as their provided Fortran77 routine `makea()` which constructs this structure.

Code Description

Our code design for CG follows the basic data-parallel master-slave pattern used in the NAS sample implementation. A `cg_driver` routine interacts with the user, creates and drives `cg_object` workers, and finally gathers performance results. Of the operations mentioned above, our implementation performs the matrix-vector multiplication and dot product in parallel, while serializing the vector-vector addition.

The data members of the CG object class include the sparse *A* matrix being solved (stored as a one dimensional array of non-zero elements), arrays corresponding to the vectors utilized in the algorithm above, double precision floats storing the current eigenvalue estimate and residual, and vectors describing the objects problem partition’s global problem indices. Additional data members are included to facilitate cooperation including the number of objects for the run, an array of other `cg_object` workers with whom to work, and various performance statistics.

`Cg_object` member functions are defined which perform the basic state initialization, computation direction, and result gathering for a worker. Key member functions include:

`cg_init():`

This is the first member function invoked on a worker object. It sets up the worker's internal state, initializes the sparse matrix data structure, then proceeds to perform the main (outer) loop described above. After the appropriate number of iterations of this algorithm, the object records its performance statistics returns.

`makea():`

As mentioned above, this routine is basically a direct, line for line translation into C++ of the NAS supplied Fortran77 routine which generates the sparse matrix. Technically speaking, we ought not have altered the original version of makea. The wording of the NAS rules is quite specific in requiring that the Fortran77 version remain part of the resulting implementation verbatim. While it is certainly not difficult to link and call Fortran routines from MPL, it is unlikely that various Fortran compilers would use the same array indexing conventions as a C++ compiler. Recall that Fortran arrays by default are indexed from 1 to *N*, while C derivative languages index from 0 to *N*-1. While we found that the standard Sun4 `f77` compiler did employ the same array indexing method at the compiled level, we felt that it was safest and most portable to translate the routine to C++, and in doing so we did not alter the semantics at all.

`cg_solve():`

This member function encapsulates the conjugate gradient sparse system solution described above (the second algorithm given in the problem description).

`dot_product():`

The first parallel operation performed by the conjugate gradient method solver is a vector-vector dot product. This member function first computes the local partial dot product, then employs additional member functions to gather the results at a designated gatherer object, which sums them and returns the full dot product. While this can result in some speedup given the large vector size (14,000), in general, this operation is relatively fine grained and would not likely perform well with large numbers of objects.

`matrix_vector_mutiply():`

The next operation required by the conjugate gradient method solver loop is a matrix-vector dot product. This member function employs related member functions to gather a complete copy of the vector multiplicand at all worker objects. Partial vector results are then computed, sent to a gatherer object, summed sequentially, and distributed to all objects using a simple gathering member function.

A number of other member functions are included which perform various sequential numeric operations and performance calculation. While we were able to verify correct execution of CG for the small problem size reference value provided, we experienced problems running the full problem size due to it’s very large memory requirements.

3. Performance of Implemented Kernels

We now move to a discussion of the performance results obtained for the kernel set we implemented. We ran our tests using Mentat 2.8 and the corresponding version of `mplc` with “-O2” optimization.The testbed employed was network of eight Sun4c class workstations with 28 Mb of memory each. The network communication was via 10Mbps Ethernet. The eight workstations were all on the same subnet. Each of our performance tables includes results for PVM on various platforms (from [2]), the Intel iPSC/860, and a single processor of the Cray Y-MP (from[1]) for comparison. Our performance numbers reflect the best time of eight runs of each application in order to mask the secondary effects of varying user load and background processes.

3.1 EP Kernel Performance

Performance results of the EP Benchmark Kernel are largely determined by the processing power of the hosts used. The main challenge to good processor utilization in this case is load balance. For example, the PVM implementation employed a 10 second pilot computation to assign

Platform	**Time (secs)**
Mentat 8 sun4c Ethernet	1429
PVM 16 SS1+ Ethernet	1603
PVM 8 RS/6000 FDDI	342
PVM 8 SGI Gigaswitch	446
Cray Y-MP/1	126
i860/32	102
i860/64	51
i860/128	26

TABLE 1. Kernel EP Results

better load partitions to the participating processes. Our implementation could certainly have been improved by a static run-time scheme such as this, or a dynamic self scheduling scheme.

In our homogeneous, lightly loaded testbed, we saw good processor utilization. In a heterogeneous environment in the presence of widely variable user loads, this version of the code would likely suffer from poor efficiency. Despite this obvious opportunity for enhancement, we spent the bulk of our effort on the more substantial kernels. As mentioned, Mentat scheduling enhancements will soon be available which will improve the load balance of applications based on regular objects, which will transparently improve the dynamic load balancing properties of this application.

3.2 IS Kernel Performance

While the benchmark specifications for the Integer Sort NAS Parallel Benchmark call for two problem sizes, Class A and the larger Class B, our tests were performed on a smaller problem size. A problem size of 2^{19} integer keys in a range of $[0, 2^{19})$ was run due to memory constraints on local machines. A positive side to this problem size was that it allowed a more direct comparison with the PVM results presented in [2].

Our time for this benchmark reflects a significant improvement over the numbers reported for all PVM configurations, even those based on much higher-performance communication substrates. This performance gain is a largely a result of improvements in the key ranking algorithm facilitated by Mentat, which employed fewer barrier synchronization points than the PVM version. In addition to this, the Mentat message passing facilities appear to achieve somewhat better communication performance in this application. Given that this kernel is communication bound, the improved message passing performance also led to better results.

Platform	Time (secs)	Comm. Volume	Comm. Time (secs)	Number of msgs
Mentat 8 sun4c Ethernet	201*	n.a.	n.a.	n.a.
PVM 16 SS1+ Ethernet	607*	150MB	595	10115
PVM 8 RS/6000 FDDI	674	560MB	610	2491
PVM 8 SGI Gigaswitch	770	560MB	720	2491
Cray Y-MP/1	11			
i860/32	26			
i860/64	17			
i860/128	14			

TABLE 2. Kernel IS Results

*. Reduced problem size (2^{21} keys in the range 0 to 2^{19}).

3.3 MG Kernel Performance

Our MG implementation was similar in its basic algorithm to both the iPSC/860 and PVM versions, and thus we expected to achieve roughly similar performance. In fact, somewhat improved performance was observed (note, we did run on processors configured with more mem-

Platform	Time (secs)	Comm. Volume	Comm. Time (secs)	Number of msgs
Mentat 8 sun4c Ethernet	104*	n.a.	n.a.	n.a.
PVM 16 SS1+ Ethernet	198*	96MB	154	2704
PVM 8 RS/6000 FDDI	229	192MB	162	1808
PVM 8 SGI Gigaswitch	264	192MB	112	1808
Cray Y-MP/1	22			
i860/128	8.6			

TABLE 3. Kernel MG Results

*. Reduced problem size ($128 \times 128 \times 128$).

ory than on the comparable PVM configuration). This result indicates an encouragingly low level of introduced overhead by Mentat as compared to the lower level, hand-coded version of the application, as well as better communications performance

3.4 CG Kernel Performance

As in the MG kernel case, our CG implementation was quite similar to the provided NAS version in its exploitation of opportunities for parallel execution. Unfortunately, we observed memory constraints which prevented running the full sized tests, making performance comparison difficult. Given the observed results, however, it seems likely that our performance would compare favorably to that of the PVM implementation.

Platform	Time (secs)	Comm. Volume	Comm. Time (secs)	Number of msgs
Mentat 8 sun4c Ethernet	46.0*	n.a.	n.a.	n.a.
PVM 16 SS1+ Ethernet	701	370MB	480	37920
PVM 4 RS/6000 FDDI	285	130MB	192	7116
PVM 9 SGI Gigaswitch	130	250MB	101	19756
Cray Y-MP/1	12			
i860/128	7.0			

TABLE 4. Kernel CG Results

4. Conclusions

Given these performance results and our experiences porting the NAS benchmarks, we have concluded that the Mentat programming model was useful in harnessing the available computational power of local workstation clusters and applying that power to realistic scientific applications. We found the overhead introduced by Mentat was not large when compared to hand-coded applications using the low level PVM system. In some cases, the higher level programming abstraction provided a vehicle for algorithmic refinements which resulted in overall improved performance. Further work on the NAS benchmarks in MPL could include porting the remaining kernel, a three dimensional PDE solver using forward and inverse fast Fourier transforms, as well as the NAS application-level benchmarks.

References

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon, at al. “The NAS Parallel Benchmarks”, RNR Technical Report RNR-94-007, March 1994

[2] V.S. Sunderam, “PVM: A Framework for Parallel Distributed Computing”, *Journal of Concurrency: Practice and Experience*, 2(4), pp. 315-339, December 1990

[3] S. White, A. Alund, V.S. Sunderam, “Performance of the NAS Parallel Benchmarks on PVM Based Networks” Report RNR-94-008, May 1994

[4] The Mentat Research Group, “Mentat 2.8 Programming Language Reference Manual”

[5] The Mentat Research Group, “Mentat 2.8 User’s Manual”