A Programming Model for Automated Decomposition on Heterogeneous Clusters of Multiprocessors *

Sean P. Peisert and Scott B. Baden

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

{peisert@sdsc.edu,baden@cs.ucsd.edu}

February 13, 2001

1 Introduction

Clusters of multiprocessors have emerged as a powerful tool for understanding technologically important physical phenomena [29]. Although various approaches to programming them have emerged [12, 5, 6, 25, 26, 24, 22], with few exceptions [18], heterogeneity has been conspicuously left out of programming models targeted to hierarchically constructed multicomputers. This is surprising, considering the flexibility offered by heterogeneous cluster designs. Even homogeneous configurations are likely to become heterogeneous with time, due to incremental upgrading. Nodes will come to have differing numbers of processors, varying clock speeds, different memory system configurations, or some combination of these.

Heterogeneity introduces a difficult challenge which could limit the effectiveness of clusters: the programmer would rather not be aware that the hardware is heterogeneous. In particular, heterogeneity introduces two implementation issues that complicate user code unnecessarily: how to assess the performance of the component nodes and how to share the workloads fairly.

In this paper, we will address the problem of how to treat heterogeneity arising in dedicated clusters of multiprocessors. These techniques generalize to clusters with uniprocessor nodes. We describe a programming model and library that may be used to program a heterogeneous cluster almost as if the cluster were homogeneous. The library, called Sputnik [28], is currently implemented in C++ on top of the KeLP infrastructure [21, 20, 10]. Sputnik relies on OpenMP to support parallelism within shared memory nodes, though the underlying model is not tied to a specific implementation of processor level parallelism.

*This work was supported in part by UC MICRO program award number 99-007 and by Sun Microsystems. Computer time on the Origin 2000 was provided by the National Computational Science Alliance. The development of KeLP was supported by NSF contract ACI-9619020, National Partnership for Advanced Computational Infrastructure.

We present results obtained by running an iterative finite difference solver on a two-node cluster of Origin 2000s. We find that Sputnik is able to effectively utilize the available parallelism, and that it meets the requirement of hiding a good deal of the underlying hardware heterogeneity from the programmer.

The paper is organized as follows. In Section 2, we introduce the software issues raised in a heterogeneous cluster of multiprocessors. Section 3 presents the Sputnik methodology and the API. Section 4 presents implementation details and section 5 presents empirical results. Section 6 presents discussions, conclusions and future work.

2 Programming Heterogeneous Clusters of Multiprocessors

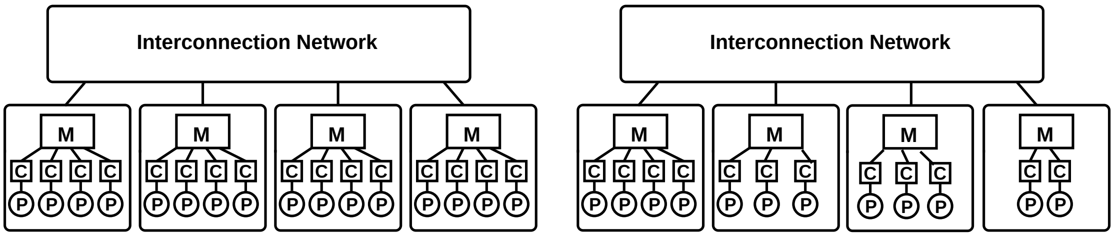
In this section we describe our underlying assumptions about the hardware, execution model, and our target application class.

2.1 System and Programming Assumptions

We define a cluster of multiprocessors as a collection of N multiprocessor nodes, numbered 0 through $N-1$, where each node i contains P_i processors. A homogeneous cluster, depicted in Fig. 1a, introduces a two-level locality model. Processors on the same node communicate quickly through shared memory, but processors on different nodes communicate much more slowly by passing messages.[1] Each node computes at a characteristic rate, r_i, which may be different from that of the other nodes. No node is slower than half the speed of the fastest node.[2] Data transfer rates within and between nodes are not uniform, even if we ignore the effects of contention. We will assume, however, that there is no processor contention, and that both the processors and the interconnect are dedicated.

[1]The logically shared memory depicted in the figure may in fact be physically distributed, but specifics of the physical interconnect are irrelevant here.

[2]This limitation may be somewhat liberal, but it helps avoid severely imbalanced memory and communication requirements.



(a) A two-tier cluster (b) A single tier cluster

Figure 1: Block diagrams of a homogeneous (a) and homogeneous (b) dual tier clusters. M=memory, C=cache, P=processor. On-processor cache memories are not shown.

We will assume that we have an existing application program that runs under SPMD parallelism and communicates by passing messages either explicitly under programmer control, or under the control of an application library or a compiler. However, many of our ideas generalize to a global, but distributed shared memory architecture.

Our application will execute trivially on a dual tier computer if we flatten the hierarchical machine organization into a ''single-tier" computer and run with one SPMD process per processor. However, we will assume that a more effective strategy is to unfold a second level of parallelism either by relying on lightweight processes, e.g. threads, one for each CPU [3], or by relying on a library or compiler to achieve a similar effect.

2.2 An Application

Consider an iterative finite difference (stencil) method that repeatedly sweeps a two dimensional $(M+2) \times (M+2)$ mesh according to the formula:

$$u_{i,j}^{new} = F(\rho(u_{i,j}^{old}))$$

There are two solution arrays u^{new} and u^{old}, and array indices i and j vary independently from 1 to M. (M is a user input affects the accuracy of the computed solution.) The function $F()$ returns the next approximation to the solution, and operates on a neighborhood of each point of the solution. $\rho()$ determines the neighborhood, depends on the application. A simple neighborhood would be nearest neighbors on the four Manhattan directions. After each iteration the old and new arrays reverse roles. The process terminates when the answer has converged to a user-supplied threshold.

If the nodes are running at the same speed, then we may partition the workload trivially by splitting the two dimensional domain into rectangular subdomains of equal size (or as nearly equal as possible given the the values of M and the number of nodes N). On the other hand, if the nodes run at different speeds, we must adjust the size of the partitions accordingly in order to balance the workload. This problem is reminiscent of load balancing strategies such as recursive coordinate bisection which has been applied to irregular problems on homogeneous architectures [14, 7]. More recently, irregular partitioning has been applied to heterogeneous architectures [18, 15].

With a partitioning strategy in hand, we are left with the problem of how to determine the relative running speeds of the nodes. In general, the rate at which a node computes on a given problem cannot be known in advance, as that quantity depends not only on the specific hardware used, but also on the

[3]In some cases, single-tiered programming might be more effective than dual-tiered programming, but we will not argue the merits of the two approaches here.

configuration and the application. In some cases the specific nodes to be used in the job will be determined only at the time the job is initiated according to decisions made by a job scheduler [17]. To complicate matters, the maximum running speed may be a non-linear function of the number of threads, and may be achieved with less than the maximum allowable number of processors. Lastly, various locality parameters, such as blocking factors, may need to be tuned, e.g. for cache, TLB, and so on.

As a result of these complications, we seek a strategy that automatically determines the optimal setting of various performance tuning parameters. The approach should enable a program written for homogeneous multicomputers to run efficiently on a heterogeneous configuration without entailing extensive recoding of the application. We define "efficient" to mean "competitive with hand coding."

The design of a programming methodology that enables applications to tune their performance to the hardware is elusive. Techniques currently exist for specific problems [8, 27, 23, 19], with some restricted to single processor or shared memory architectures. Our contribution is a programming methodology and a supporting user interface that formalizes the process. Our approach obviates the need for empirically-derived parameters, such as message start time and bandwidth, or a nominal floating point rate determined by measuring a fixed kernel [18]. The rationale for our approach is the notorious difficulty of predicting the performance of an application with a handful of parameters, which often depend on the application and the input. Instead, our approach is to search the performance-tuning parameter space.

Though our techniques have currently been tested on a finite difference problem, we believe they apply to other application classes, e.g. the Fast Fourier transform, which are enable to general blocked decompositions.

3 Sputnik

Sputnik is a programming methodology and a run time library for efficiently implementing scientific computations on heterogeneous clusters of multiprocessors. Sputnik's goal is to enable the programmer to write an application almost as if the hardware were *homogeneous.*

Sputnik supports a two-level model of parallel control flow. The first level runs a single process per node moving data between nodes. The second level runs multiple lightweight processes on each node, one per processor. Sputnik relies on a two-stage process for optimizing performance at these two levels. The first stage, called *ClusterDiscovery*, assesses the relative performance of each node by running the original application program individually on each node. During this stage Sputnik not only determines the speed of each of the nodes, but it attempts to optimize performance for a specified set of optimization criterion. In the current implementation of Sputnik, we restrict ourselves to optimizing the number of threads. However, other optimizations, e.g. blocking for cache, are possible.

Using the timings obtained from stage one, the second stage, called *ClusterOptimizer*, partitions the dataset non-uniformly, according to the relative speed of each node. Sputnik then runs the program using the optimal partitionings and other optimizations made in stage one. We can see then, that the first stage of optimization manages the second level of parallelism, and the second stage of optimization manages the first level of parallelism.

3.1 ClusterDiscovery

ClusterDiscovery performs two tasks. It measures node performance and it carries out performance optimizations. Sputnik assumes that optimizations are parameterized. It searches a parameter space of possible optimizations, choosing the optimal parameter configuration. When evaluating optimizations, Sputnik measures application performance directly. It does not model performance using empirically determined parameters, e.g. peak floating point rate and message start time, and attempt to use these parameters to estimate performance. The advantage of this approach is to eliminate nearly all built in assumptions about the hardware. For example, the nodes need not be multiprocessors, and Sputnik does not need to be told how many processors there are on each node. It need not know characteristics of the individual CPUs, e.g. cache configurations. However, we must take care to prune the performance-tuning parameter space to avoid long search times.

3.2 ClusterOptimizer

The *ClusterOptimizer* uses the optimizations found in the ClusterDiscovery stage and decomposes the underlying computational data according to the relative performance of each node in the cluster. A node discovered to have better performance than other nodes will therefore work on a larger portion of the overall problem. Depending on the size of the problem as a whole, the cache sizes, and the amount of communication taking place, there are a variety of different decomposition schemes available. From the viewpoint of the programmer, the exact partitioning is not known, except that the partitionings are blocked. Unlike standard blocked partitionings, the partitionings employed by Sputnik may not be regular.

3.3 Limitations

Sputnik assumes that the application is represented as a uniform multidimensional array. It does not currently support general sparse structures, e.g. sparse matrices or graphs. Computations are further assumed to be amenable to general blocked decompositions. This constraint admits finite difference computations, including multi-level and adaptive methods, and spatially subdivided particle methods. However, it does not apply to matrix linear algebra, which employ block cyclic decompositions. A different kind of abstractions is required [13].

When executing the ClusterOptimizer stage, it is possible that the optimizations previously made in ClusterDiscovery will no longer be valid. However, we assume that such non-linearities are benign.

4 Implementation

4.1 Library Design

The Sputnik API was implemented in C++ using the KeLP infrastructure [21, 20, 10], which is a C++ class library. Like KeLP applications, Sputnik applications are written in a mixture of C++ of Fortran. C++ handles both data decompositions expressed by ClusterOptimization and the accompanying data motion between processes. Fortran handles the numerical computations.

KeLP supports the irregular decompositions needed by Sputnik, which were discussed in §2. It runs on top of MPI [3]. Intra node parallelism is managed with OpenMP [4]. Thus, management of parallelism across and within nodes is handled with two distinct programming models.

The Sputnik API is designed so that the `main()` routine of a program is moved, mostly, to a user-defined routine called `SputnikMain()`. The real `main()` does initialization and calls a routine called `SputnikGo()`. `SputnikGo()` acts as a kind of "shell" that calls `SputnikMain()` over and over to determine the optimal number of threads per node, and makes the final run with the optimized configuration. `SputnikMain` returns a double. The value of that double should be the time it takes for the kernel to run. For example:

```
double SputnikMain(int argc,char ** argv, double * SputnikTimes) {
    double start, finish;


    ...
    <declarations, initializations>
    ...
```

```
    start = MPI_Wtime();  // start timing
    kernel();             // call the kernel function
    finish = MPI_Wtime(); // finish timing
    ...
    return finish-start;
}
```

Essentially everything that was in `main()` can now be in `SputnikMain()` with the addition of timing calls. A typical `main()` might now look like this:

```
int main(int argc, char **argv)
{
   MPI_Init(&argc, &argv); // Initialize MPI
   InitKeLP(argc,argv);    // Initialize KeLP

   // Call Sputnik's main routine, which in turn will
   // then call SputnikMain().
   SputnikGo(argc,argv);

   MPI_Finalize();         // Shut down MPI
   return (0);

}
```

The call that sets the number of threads per node is actually set in `SputnikGo()` and is not seen by the user. The number of threads actually employed in a loop should be tested by calling `OMP_GET_MAX_THREADS()`. In this way, the programmer can determine whether OpenMP is doing a good job of parallelizing the code.

The repartitioning, one of the primary features of the Sputnik API is a modification of the distribution functions of an existing KeLP library called DOCK, which automatically decomposes a rectangular region across processors. Whereas DOCK supports uniform BLOCK decompositions, Sputnik supports non-uniform ones. Sputnik inherits DOCK's run time `Processor` and `Decomposition` objects. The latter has a `distribute()` member function, which under Sputnik contains an additional array argument specifying the running times of each of the processors:

```
Processors3 P;
Decomposition3 T(domain);
T.distribute(BLOCK,BLOCK,BLOCK,P,double Times[]);
```

4.2 Limitations

Currently only single- dimensional partitions are supported, since our initial target architectures have only a small (two to four) number of nodes. The distribute() function readily generalizes to multiple dimensions, however (DOCK supports multi-dimensional partitionings)

The Sputnik API also requires that the application is written in C++, at least as a wrapper, though the kernel(s) of the program may be written in either C, C++, or Fortran and linked in. Finally, Sputnik depends on the fact that the cluster has a thread-safe implementation of MPI installed as well as OpenMP for both Fortran and C++.

4.3 Experimental Testbed

Our experimental testbed consisted of a pair of SGI-Cray Origin 2000 multiprocessors located at the National Center for Supercomputing Applications (NCSA). The two machines ran v. 6.5 of the IRIX OS. We used KeLP version 1.3a and compiled all code with the native MIPSpro f77 and CC compilers, v. 7.3.1m, with command line options as shown in Table 2. We used round robin page placement and turned page migration on using the following environmental variable settings: `_DSM_MIGRATION =ALL_ON` and `_DSM_PLACEMENT = ROUND_ROBIN`.

Runs were collected during a special dedicated time slot running on two machines called *balder* and *aegir*. The machine configurations are given in Table 1. The two machines are connected by an SGI Gigabyte System Network (GSN) interconnect supporting a maximum bandwidth of 800 MB /sec. and a theoretical latency of less than 30 μs. Experimental results showed the actual latency to be much closer to 140 μs and the bandwidth less than 100 MB /sec.

4.4 Optimizations

Sputnik remembers past configurations, saving the results of ClusterDiscovery in a small database on disk. As a result, future runs of the program on the same cluster will not have to "re-discover" performance each time and can simply run with the optimal settings. This technique allows us to reduce ClusterDiscovery execution time to zero, though in practice, this may not be so important in iterative methods where we expect to be able to optimize performance with just the first few of many mesh sweeps. In future work, we

Characteristic	*balder*	*aegir*
Processors	256	128
Main Memory	128 GB	64 GB
CPU Type and Clock Speed	250 MIPS R10000	
Cycle Time	4.0 ns	
Processor Peak Performance	500 MFLOPS	
L1 Cache Size	32 KB	
L2 Cache Size	4 MB	
Operating System	IRIX 6.5	

Table 1: Specifications for the two Origin2000 machines, *balder* and *aegir*.

Compiler	Version	Command line flags
MIPSpro f77	7.3.1m	-mp -O3 -mips4 -r10000 -64
MIPSpro CC	7.3.1m	-mp -lmpi -lm -lftn -lcomplex -O3 -r10000 -64

Table 2: Compiler command line options

hope to formalize this process. For example, FFTW provides for "words of wisdom" that can be used to amortize the cost of optimizations. [9].

5 Results

5.1 An application

To validate Sputnik, we ran Redblack3D, an iterative finite difference code that solves Poisson's equation in three dimensions subject to Dirichlet boundary conditions. Redblack3D uses Red/Black ordering with Gauss-Seidel iterations to solve the equations. We began with publicly available implementation of Redblack3D [10] written using the KeLP infrastructure [21, 20, 10].

Recalling that management and control of parallelism is handled in C++ via KeLP, we modified the application to invoke the Sputnik API, and to unfold a second level of parallelism within the Fortran kernels. The changes to the Fortran code were trivial, and involved the introduction of a handful of OpenMP directives. Since these modifications would be needed on a homogeneous platform, we consider only the C++ portion of the code.

The existing RedBlack3D code executes an outer iteration loop, and within each iteration it repeatedly calls a KeLP routine to carry out inter node communication, and a Fortran routine to carry out computation. Sputnik requires that we move the entire main program inside a call to `SputnikGo()`, which invokes the Sputnik shell. Each call to `SputnikGo()` returns the time spent executing the program. Thus, we insert calls to time the execution `SputnikGo()`[4] In all, we added or modified about 25 lines of code to an application

[4] This included some output, which took negligible time.

containing 596 uncommented lines of C++. These changes were restricted to the single top-level source module, which contained 149 uncommented lines of C++.

5.2 Experiments

Our experimental strategy was designed to conserve scarce dedicated time. Thus, we carried out some of our experiments on a single system emulating a virtual cluster by running two MPI processes on the one node. We also we ran across the two machines, but we did not employ thread optimization in ClusterDiscovery. Rather, we set the number of threads manually. This was necessary because we restricted our runs to just 3 mesh sweeps. In practice we would run for dozens or hundreds of mesh sweeps (in particular, if we used multigrid to accelerate the smoother), but we would only utilize a few mesh sweeps to determine the optimal decomposition. Thus, the cost of ClusterOptimization would be amortized over the length of the run. [5] This strategy enables us to demonstrate the effectiveness of the irregular partitionings employed by Sputnik, while avoiding long ClusterDiscovery times in the short runs we used to collect our experiments.

In these experiments we set the number of threads on each machine to predetermined values. We ran each computation for three mesh sweeps, and repeated each run once, for a total of two runs each. Our strategy was to set the number of threads on one machine (*balder*) to a fixed amount, while varying the number of threads on the other machine (*aegir*) to emulate various degrees of heterogeneity. This enabled us to establish whether or not the irregular decompositions were robust with respect to the degree of heterogeneity. We ran three experiments. In the first experiment, we set the problem size N = 761. With 761 $\times$ 761 $\times$ 761 unknowns, and two 64-bit floating point numbers per unknown, the total amount of storage (excluding ghost cells) was 6.72 GB. We set the number of threads on `balder` to $\tau_{balder} = 32$, and ran with various number of threads on *aegir*, τ_{aegir} = 16, 20,24, 28, and 32. Thus, the ratios of machine speeds varied from 2:1 down to 1:1. (As noted previously, the slowest node never runs at less than half the speed of the fastest node). We also scaled up the power of the virtual cluster, running with the same problem size and $\tau_{balder} = 48, \tau_{aegir}$ = 24, 30, 36, 42, 48, and on a few larger problems with with up to 128 threads on *balder* and 96 on *aegir*.

Our results are presented in Fig. 2 and in tabular form in Tab. 3. They reveal that Sputnik is able to balance workloads according to our expectations. Moreover, Sputnik's effectiveness is insensitive over the tested range of heterogeneity. Fig. 2 presents just the computation times, and shows that the workloads have been well balanced. We see three sets of bars for each value of τ_{aegir}. The left-most bar (black) gives the original computation time, excluding communication. The middle bar (gray) shows the computational time

[5]Furthermore, if we use a feature of Sputnik that "remembers" the optimal performance parameters from a previous run, then this time would be reduced to zero.

after Sputnik has assessed the performance of the node, and partitioned accordingly. The final bar (white) shows the ideal time based on the computation time measured during ClusterDiscovery. The differences between the predicted and actual timings are small. They never differ by more than about 5% in the $\tau_{balder} = 32$ runs, while the relative difference decreases as the degree of heterogeneity decreases. (The differences are much smaller for the $\tau_{balder} = 48$ runs — about 1%). The discrepancy between the measured and predicted running time is small, and we suspect that this is due to a non-linearity effect that results from basing our predictions on a uniformly partitioned workload. [6] When we repartition the workload, we increase the footprint of the workload assigned to the faster machine. In particular, memory access strides in will be larger, which could result in increased miss rates in the TLB and in the caches. Ideally we might repeat the ClusterDiscovery stage, but the resultant improvement is not likely to be significant.

Tab. 3 also shows the total times including communication. Significantly, we note that Sputnik prevents the slower node (*aegir*) from idling for significant amount of time. For example, with 16 threads and $\tau_{aegir} = 32$, *balder* remains idle for 40 seconds while waiting for *aegir* to catch up. With 24 threads and $\tau_{aegir} = 48$, *balder* remains idle for about 25 seconds. We can see this by glancing at Fig. 3. The total running times without Sputnik ("Original Total") are nearly identical; both processors will complete in about the same time, due to the wait on communication. But the actual computation time ("Original Compute") on one of the nodes is much smaller. Next, we see that with balanced workloads, the total run time is somewhere in between the two extremes, and similarly for the computation times. The workloads are not *perfectly* balanced, however, they are close enough.

We also performed a few experiments with larger numbers of threads and larger problem sizes. These are shown in Tab. 4. Sputnik performed well with large numbers of threads per system as well, balancing the workloads to within about 5% of the predicted time or better.

Finally, we ran ClusterDiscovery on a single node with two virtual machine processes. We indeed found that when presented with a maximum number of threads, that Sputnik would sometimes choose to use fewer than the maximum number of threads in order to achieve optimal performance.

[6] Another possibility is that the variations are simply noise; we were not able to repeat our runs a sufficient number of times to factor out variations due to memory layout and thread scheduling. However, the two runs generally agreed closely.

Threads	Original Compute		New Total		New Compute		Predicted
aegir	*balder*	*aegir*	*balder*	*aegir*	*balder*	*aegir*	
16	47.7	87.7	66.4	65.7	65	60.4	61.8
20	48.4	74.4	63.1	63.8	61.3	58.0	58.6
24	51	62.7	58.4	59.5	56.0	55.4	56.2
28	50	55.6	54.4	53.8	50.3	51.4	52.6
32	51.2	48.7	51.6	52.57	48.0	50.6	49.9

Threads	Original Compute		New Total		New Compute		Predicted
aegir	*balder*	*aegir*	*balder*	*aegir*	*balder*	*aegir*	
24	37.6	62.9	50.6	49.9	43.8	46.3	47.06
30	37.6	50.1	45.5	45.5	43.4	42.7	42.69
36	36.1	43.5	42.3	42.3	37.8	39.7	37.8
42	36.5	38.4	40.6	40.7	36.8	35.4	36.8
48	37.4	34.3	41.4	42.3	33.3	36.2	33.3

Table 3: Redblack3D timings with 32 and 48 threads on *balder* and varying numbers of threads on *aegir*

N	Threads		Original Compute		New Total		New Compute		Predict
	balder	*aegir*	*balder*	*aegir*	*balder*	*aegir*	*balder*	*aegir*	
949	64	32	57.1	99.3	74.6	74.6	71.5	71.8	72.5
1163	128	64	59.6	108	83.3	83.7	80.3	75.5	76.8
1163	128	96	65.6	73.5	72.9	72.7	65.5	68.8	69.3

Table 4: Redblack3D timings for larger problems and larger numbers of threads per system

6 Discussion and Conclusions

We have presented a programming methodology and API called Sputnik, which is able to adaptively partition an interactive finite difference solver on a heterogeneous cluster of multiprocessors. Our experiences demonstrate that heterogeneous clusters of multiprocessors need not be difficult to program and that they can be treated almost as if they were heterogeneous. As clusters of multiprocessors appear to be the near-term future for supercomputing, ways are needed to address the evolution of these machines. Sputnik is one of these ways.

It should be easily possible to convert an application written with KeLP to Sputnik so long as a good OpenMP implementation exists and the kernel is amenable to parallelization with OpenMP. Following the Sputnik Model, the Sputnik API library could be readily adapted to work with different software technologies. For instance, instead of KeLP one should be able to adapt Sputnik to run applications based on MPI, so long as the application supports irregular blocked decompositions.

Like others who have developed architecture cognizant methodologies [23] or libraries e.g. PHiPAC [8] and Atlas [19]. we do not rely on a performance model with empirically-determined parameters. By compar-

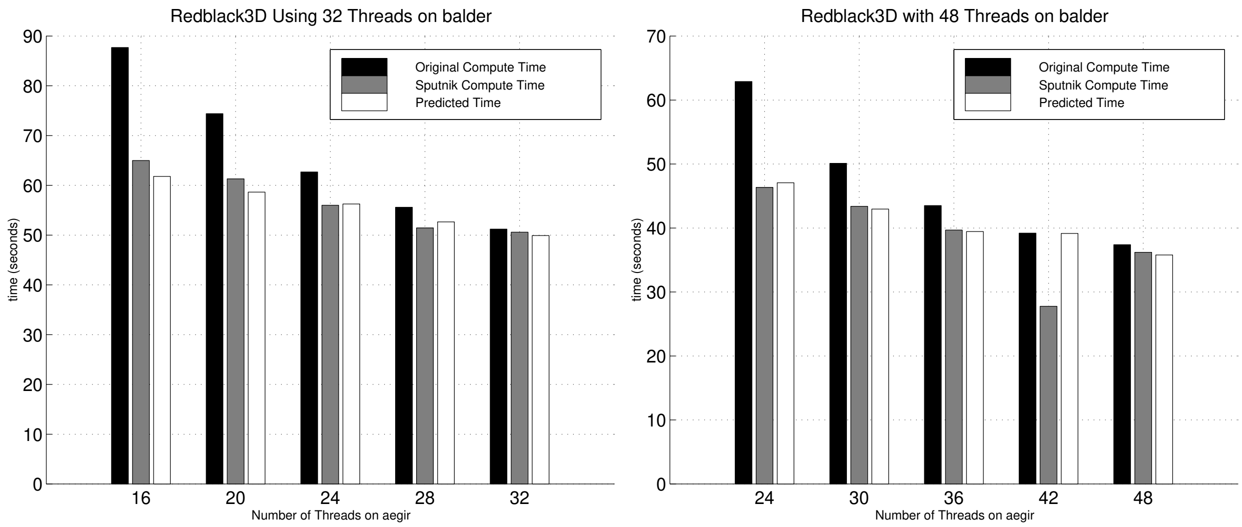


Figure 2: Redblack3D timings with 32 and 48 threads on *balder*, and varying numbers of threads on *aegir*

ison, Crandall has investigated a performance advisory system which relies on a parameterized performance model [18]. The advantage of an approach which is parameter-free is that to admit extension to new types of performance optimizations. For instance, the current implementation of Sputnik carries out only one optimization, namely, it determines the optimal number of threads. In the future, we plan to explore alternative optimizations, such as tiling. Sputnik could also be adapted to work in a dynamic environment as well, where instead of sampling just once, at the beginning, testing and sampling could happen continuously throughout the run of the program to optimally execute long-running programs, tuning throughout the run of the program. Moreover, since actual running times are used, it is possible to accommodate a highly irregular cluster with slow and fast interconnects. (But the more general scheduling problem on highly heterogeneous architectures, involving remote resources, requires a different kind of support [15]).

Our initial experiments were carried out on a cluster with just two nodes and many processors. In future work we plan to extend the API to handle a larger number of "smaller" nodes, and to test other types of applications such as multigrid and the Fast Fourier transform.

Acknowledgments

The authors wish to thank Faisal Saied for arranging access to the dedicated Origin 2000 at NCSA, and for the many members of the technical staff at NCSA who gave advice on using the Origin. Thanks also go to Dan Shalit for providing assistance with KeLP, and Dr. Paul H. J. Kelly for early suggestions on how to improve Sputnik. Thanks also go to Uppsala University in Sweden for generously allowing use of the Yggdrasil cluster in the early stages of the research.

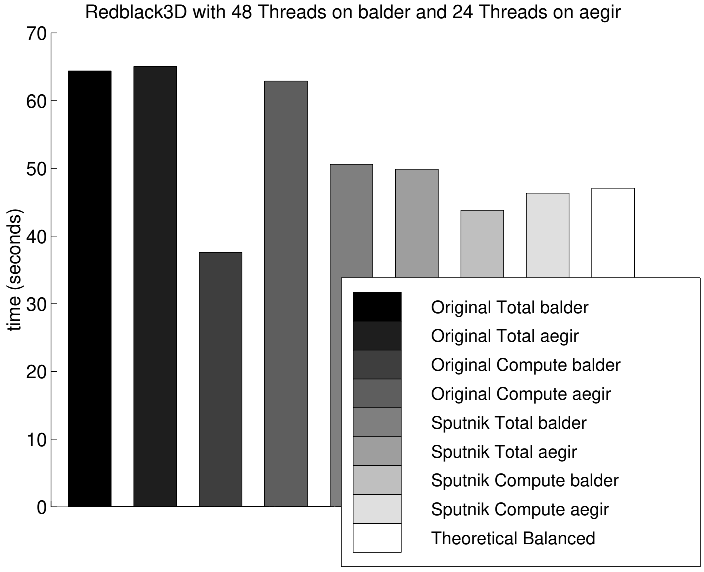


Figure 3: Detailed timings of RedBlack3D timings with 48 threads on *balder* and varying numbers of threads on *aegir*, using the Sputnik library.

References

[1] S. B. Baden and S. J. Fink, "The Data Mover: A Machine-independent Abstraction for Managing Customized Data Motion," LCPC '99, August 1999.

[2] S. J. Fink, S. B. Baden, and S. R. Kohn, "Efficient Run-Ti me Support for Irregular Block-Structured Applications," J. Par. Distrib. Comput., 50(1-2), April-May 1998, pp 61-82.

[3] Argonne National Laboratories, "MPI - The Message Passing Interface Standard," `http://www-unix.mcs.anl.gov/mpi/`.

[4] OpenMP Architecture Review Board, `http://www.openmp.org/`.

[5] S. B. Baden and S. J. Fink, "Communication Overlap in Multi-tier Parallel Algorithms," SC98, Orlando FL, Nov. 1998.

[6] S. B. Baden and S. J. Fink, "A Programming Methodology for Dual-tier Multicomputers. *IEEE Trans. on Software Eng.,* March 2000.

[7] S. B. Baden, "Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors," *SIAM Journal on Scientific and Statistical Computing*, vol. 12, pp. 145–157, January 1991.

[8] J. Bilmes, K. Asanović, C. Chin, and J. Demmel, "Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology," in *Proceedings of International Conference on Supercomputing*, (Vienna, Austria), July 1997.

[9] The Fastest FFT in the West, `http://www.fftw.org`.

[10] S. B. Baden and D. Shalit, `http://www-cse.ucsd.edu/groups/hpcl/scg/kelp/`.

[11] S. B. Baden, D. Shalit, R. B. Frost. "KeLP User Guide Version 1.3," Department of Computer Science and Engineering, University of California, San Diego, Jan. 2000, `ftp://ftp.cs.ucsd.edu/pub/scg/KeLP/UserGuide_1.3.pdf`.

[12] D. A. Bader, and J. Ja'Ja', "SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs)," Tech. Rep. CS-TR-3798, Univ. of Maryland Inst. for Advanced Computer Studies-Dept. of Computer Sci, Univ. of Maryland, May 1997.

[13] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Load Balancing Strategies for Dense Linear Algebra Kernels on Heterogeneous Two-dimensional Grids," IPPS'2000, Cancun Mexico, May 2000.

[14] M. J. Berger and S. H. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Transactions on Computers*, vol. C-36, pp. 570–580, May 1987.

[15] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, "Application-Level Scheduling on Distributed Heterogeneous Networks," Proc. Supercomputing 1996, November 1996.

[16] F. Cappello and O. Richard, "Performance characteristics of a network of commodity multiprocessors for the NAS benchmarks using a hybrid memory model," PACT 99, July 1999.

[17] W. Cirne and F. Berman, "Adaptive selection of partition size for supercomputer requests," in *Proceedings of 6th Workshop on Job Scheduling Strategies for Parallel Processing*, (Cancun, Mexico), May 2000.

[18] P.E. Crandall and M. J. Quinn, "A Partitioning Advisory System for Networked Data-Parallel Processing," Concurrency: Practice and Experience, 7(5), Aug. 1995, pp. 479-495.

[19] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project," submitted for publication, 1999, `http://www.netlib.org/utk/people/JackDongarra/PAPERS/atlas_pub.pdf`.

[20] S. J. Fink, "A Programming Model for Block-Structured Scientific Calculations on SMP Clusters," UCSD CSE Department/Ph.D Dissertation, June 1998.

[21] S. J. Fink, S. B. Baden, and S. R. Kohn, "Efficient Run-Time Support for Irregular Block-Structured Applications," Journal of Parallel and Distributed Computing, 1998.

[22] I. Foster and N. T. Karonis, "A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems," Proc. SC 98, Orlando FL, Nov. 1998.

[23] K. S. Gatlin and L. Carter, "Architecture-Cognizant Divide and Conquer Algorithms," SC 99 Conference, Portland, OR, Nov. 1999.

[24] S. S. Lumetta, A. M. Mainwaring, and D. E. Culler, "Multi-Protocol Active Messages on a Cluster of SMPs," Proc. SC 97, San Jose, CA Nov. 1997.

[25] J. May, B. de Supinski, B. Pudliner, S. Taylor, "Final Report Programming Models for Shared Memory Clusters," Lawrence Livermore National Labs, 99-ERD-009, January 13, 2000.

[26] J. May and B. R. de Supinski, "Experience with Mixed MPI/Threaded Programming Models," Lawrence Livermore National Labs, UCRL-JC-133213.

[27] N. Mitchell, L. Carter, J. Ferrante, and K. Högstedt, "Quantifying the Multi-Level Nature of Tiling Interactions," LCPC 1997.

[28] S. P. Peisert, "A progrmaming model for automated decomposition on heterogeneous clusters of multiprocessors," Master's thesis, Dept. of Computer Science and Engineering, University of California, San Diego, 2000.

[29] P. R. Woodward, "Perspectives on supercomputing: Three decades of change," *IEEE Computer*, vol. 29, pp. 99–111, Oct. 1996.