Parallel Processing Letters

FUNCTIONAL ALGORITHM SIMULATION OF THE FAST MULTIPOLE METHOD: ARCHITECTURAL IMPLICATIONS

MARIOS D. DIKAIAKOS

Departments of Astronomy and Computer Science-Engineering,
University of Washington, ASTRONOMY, Box 351580, Seattle, WA 98195, U.S.A.

and

ANNE ROGERS

Department of Computer Science, Princeton University
Princeton, NJ 08544, U.S.A.

and

KENNETH STEIGLITZ

Department of Computer Science, Princeton University
Princeton, NJ 08544, U.S.A.

ABSTRACT

Functional Algorithm Simulation is a methodology for predicting the computation and communication characteristics of parallel algorithms for a class of scientific problems, without actually performing the expensive numerical computations involved. In this paper, we use Functional Algorithm Simulation to study the parallel Fast Multipole Method (FMM), which solves the N-body problem. Functional Algorithm Simulation provides us with useful information regarding communication patterns in the algorithm, the variation of available parallelism during different algorithmic phases, and upper bounds on available speedups for different problem sizes. Furthermore, it allows us to predict the performance of the FMM on message-passing multiprocessors with topologies such as cliques, hypercubes, rings, and multirings, over a wider range of problem sizes and numbers of processors than would be feasible by direct simulation. Our simulations show that an implementation of the FMM on low-cost, scalable ring or multiring architectures can attain satisfactory performance.

Keywords: Functional Algorithm Simulation. N-Body Problem. Performance Modeling.

1. Introduction

Functional Algorithm Simulation [5,14] is a new methodology for predicting the computation and communication characteristics of parallel algorithms for a class of scientific problems, without actually performing expensive numerical computations. To explore the Functional Algorithm Simulation we built the Functional Algorithm Simulation Testbed (FAST), a software prototype system that performs approximate simulations of parallel executions. FAST runs on uniprocessor workstations and has been used to evaluate a number of interesting and important scientific al-

gorithms [1,8,10] mapped onto message-passing multiprocessors. In this paper we present a case-study conducted with FAST for the parallel Fast Multipole Method [1], which solves the N-Body problem in two dimensions. We use the information derived with FAST to evaluate and analyze the relative performance of the algorithm on different interconnection topologies.

Performance is critical in parallel computing. Gaining an understanding of performance issues for challenging parallel algorithms like the FMM, however, requires the description and analysis of their parallel execution on realistic sets of data. This is a difficult task for many reasons, including the high complexity of these algorithms, the great volume of data needed to describe computations and communications performed during parallel execution, and the low availability of high-end multiprocessor systems.

Numerous research projects have collected and studied communication and computation patterns from challenging applications [3,4,17]. Most proceed by conducting instrumentation and performance monitoring on top of parallel software and hardware platforms. This approach has its obvious merits as it addresses the performance assessment of programs running on existing systems. It is accompanied, however, by inevitable drawbacks: the conclusions sought may be influenced by the underlying architectures, programming models, and implementation. Experimentation with different interconnection topologies requires the porting of applications to different multiprocessors, thus incurring a very high cost. Moreover, it is practically impossible to study the scalability of parallel algorithms and architectures on top of existing systems. In this paper, we show how Functional Algorithm Simulation addresses these issues in a study of the Fast Multipole Method. The remainder of the paper is organized as follows: Section 2 presents the basic concepts of Functional Algorithm Simulation and the structure of FAST. Section 3 describes the Fast Multipole Method and the study of the FMM with FAST; additionally, it presents an assessment of the parallel performance of the FMM on different interconnection networks. Finally, Section 4 summarizes our results and conclusions.

2. Functional Algorithm Simulation

Functional Algorithm Simulation [13] models and evaluates parallel executions by reproducing the *skeletons* of parallel computations and using them to extract their basic computation and communication patterns. It is basically a method for *approximately* simulating real parallel executions. It can also be considered as an *accurate* simulation of a theoretical model that accounts for communication costs and limited communication bandwidth, such as *LogP* [6]. The common algorithmic property necessary for Functional Algorithm Simulation to apply is the ability to determine the set of expensive calculations and data exchanges from input information, at the initialization phase of the algorithm, *before* the actual numerical computations take place. Another underlying assumption is that the initialization phase takes an insignificant portion of the overall parallel time. Both assumptions are valid for many important scientific algorithms [1,8,9,10].

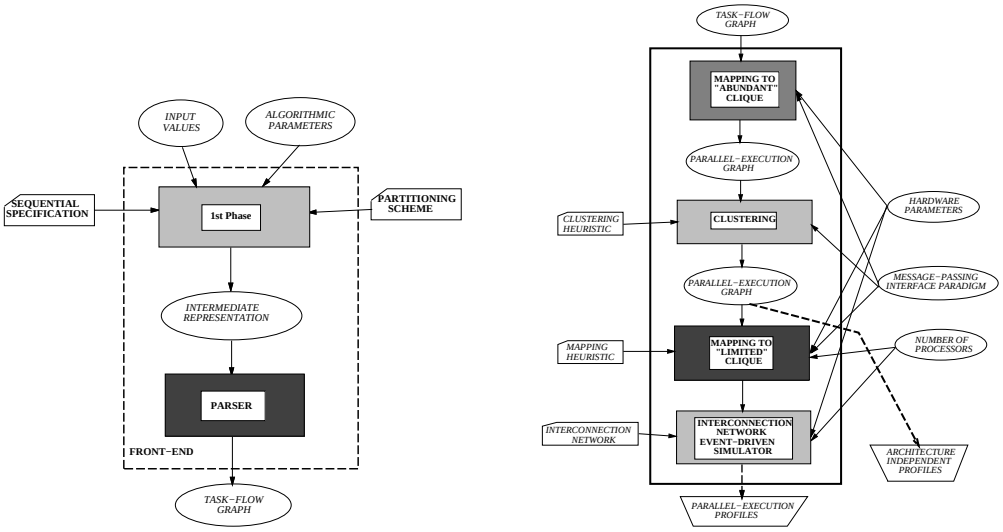

Figure 1: The Front-End and the Back-End of FAST.

2.1. Fast Algorithm Simulation Testbed (FAST)

FAST is a prototype system running on workstations and implementing the principles of Functional Algorithm Simulation. Its key structure is a weighted task-flow graph that describes computations occurring during the execution of the algorithms studied and reveals all medium-grain parallelism available. The difference between this approach and exact simulation (trace-driven or direct) lies in the fact that our system does not simulate every single instruction, but only procedure calls that would be performed in a real execution. FAST relies on knowledge of the algorithm and of the data-structures it constructs when provided with some specific set of input data. It uses this knowledge to interpret procedure calls as "black boxes" with known processing times and dependency constraints.

By not doing the numerical calculations, FAST achieves significant savings in terms of processor cycles and disk space. For example, a Functional Algorithm Simulation of an instance of the SIMPLE computational fluid dynamics benchmark [10] took *0.63 secs* to complete on a Sparcstation. The same instance took *9.8 secs* to run on one iPSC/2 node. Considering that exact simulation is approximately *100* to *1000* times slower than actual execution [12], we deduce that Functional Algorithm Simulation decreases the simulation time by *two* to *three* orders of magnitude. These savings enable us to increase the flexibility of simulations, study the performance and scalability of algorithms on parallel machines with thousands of processors, and compare the performance of different interconnection networks under realistic traffic loads.

FAST is split in two parts: a front-end and a back-end (see Figure 1), described in the following sections. A detailed description, along with information on validating its accuracy and its application on other algorithms can be found in [5,13,14].

2.2. Front-End

The task-flow graph generation is accomplished by the front-end in two phases (see Figure 1, left). The first one depends on the algorithm studied: given a sequential implementation, the user modifies it by inserting code that will produce dynamically the set of calculations and communications that define the corresponding parallel execution. The modified program is the first phase of FAST's front-end for the specific algorithm. Running this program on some appropriate input configuration produces an architecture-independent *Intermediate Representation (IR)* of the parallel execution. The Intermediate Representation is given in terms of a simple *intermediate language* comprised of *IR-operations* and *Send/Receive* communication primitives. Each IR-operation is an abstraction of a "medium-grain" group of successive numerical instructions. These groups correspond to the *basic computational blocks* of the algorithm. *Send/Receive* primitives correspond to data-dependences between IR-operations and represent the data-flow.

In the second phase of the front-end, a parser transforms the Intermediate Representation into a weighted *task-flow graph* which follows the *Macro-Dataflow* model of computation [11]. Task-nodes in the graph contain a number of Intermediate Representation primitives. Their "boundaries" are defined by *Send* and *Receive* primitives occurring in the IR. The tasks start executing upon receipt of all incoming data and continue to completion without interruption. Upon completion their results are forwarded to adjacent nodes. Edges correspond to *Send-Receive* pairs and represent the data dependencies between the nodes. The annotation of the task-flow graph is straightforward. Nodes are assigned the sum of the costs of their corresponding IR-operations. Edges are assigned weights that represent the number of bytes "carried" by those edges from their source to their destination nodes.

2.3. Back-End

The back-end of FAST (see Figure 1, right) receives the output from the front-end and maps the task-flow graph onto a message-passing multiprocessor architecture. The mapping process is accomplished in a number of successive steps: first, FAST maps the task-flow graph onto an idealized architecture with a number of processors equal to the number of tasks, forming a fully-connected network (*"abundant" clique*). The resulting graph is called the *parallel-execution graph* and is subsequently passed through *clustering*, a stage that seeks to minimize the communication overhead of the parallel execution, without sacrificing parallelism [11,16]. Clustering is NP-Complete [11]; therefore, FAST provides several different clustering heuristics [15]. After clustering, FAST performs a *mapping* of the clustered parallel-execution graph onto a message-passing architecture with a number of processors specified by the user. The mapping problem is also NP-complete [11]. To map the task-clusters onto processors, FAST includes another set of heuristics [15].

A few different interconnection schemes are available in the current version of FAST: a "limited" clique (a clique with a limited number of processors), a ring, various multirings, and a binary hypercube.

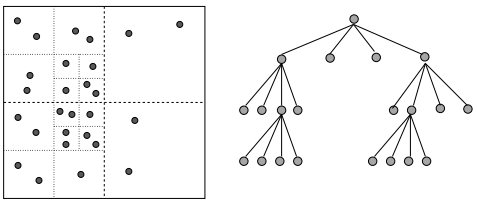

Figure 2: Decomposition of a two-dimensional space of particles and the corresponding quadtree created by the Fast Multipole Method.

In addition to the number of processors and the network topology, the user provides our system with a set of hardware parameters. These parameters are used to transform the weights assigned to nodes and edges of the task-flow graph into processing times and message latencies. They include cycles-per-instruction counts, clock-speeds, communication bandwidth, communication overhead, etc. In the current version of FAST, we used hardware parameters characteristic of INTEL's iPSC/2 and iPSC/860 multiprocessors.

3. Functional Algorithm Simulation of the FMM

3.1. The Fast Multipole Method

The Fast Multipole Method (FMM) [1] solves the N-body problem, i.e., it calculates the forces exerted on each particle by the whole ensemble of particles lying in a 2- or 3-dimensional data space. These forces determine new locations for the particles in each small time-step. Forces can be either gravitational or coulombic. The FMM has wide applications in astrophysics, molecular dynamics and computational fluid dynamics.

The brute-force method for N-body computations evaluates all pairwise interactions and thus its sequential complexity is $O(N^2)$ per time-step, where N is the number of particles (bodies). The FMM evaluates all interactions to within a fixed roundoff error and has an average time-complexity per time-step of $O(N)$. Its central strategy is the hierarchical decomposition of the data-space in the form of a quadtree (or octtree for the 3-dimensional case). This hierarchical decomposition is used to cluster particles at various spatial lengths and compute interactions with other clusters that are sufficiently far away by means of series expansions (see Figure 2).

For a given input configuration of particles, the sequential FMM first decomposes the data-space in a hierarchy of blocks and computes local neighborhoods and interaction-lists involved in subsequent computations. Then, it performs two passes on the decomposition tree. The first pass starts at the leaves of the tree, computing *multipole expansion coefficients* for the gravitational field. It proceeds towards the root accumulating the multipole coefficients at intermediate tree-nodes. When the root is reached, the second pass starts. It moves towards the leaves of the tree, exchanging data between blocks belonging to the neighborhoods and interaction-lists

calculated at tree-construction. At the end of the downward pass all long-range interactions have been computed; subsequently, nearest-neighbor computations are performed to take into consideration interactions from nearby bodies. Finally, short- and long-range interactions are accumulated and the total forces exerted upon particles are computed. The algorithm repeats the above steps and simulates the evolution of the particle system for each successive time-step.

In hierarchical N-body methods in general, and FMM in particular, the largest portion of the computation time is spent in the force calculation procedure, that is, in the operations performed during the traversal of the decomposition tree. The time spent in the tree-construction phase is not significant. Moreover, parallelism can only be exploited within one time-step.

As mentioned earlier, FAST does not perform the bulk of the numerical calculations involved in the Fast Multipole Method computation. Hence, it has no means of computing the new locations of particles after one time-step. Execution of the FMM for a sequence of time-steps, however, can be studied with a sequence of FAST-simulations on input configurations of particles that correspond to the time-steps of interest. These configurations can be extracted from real N-Body simulations.

3.2. Setup of Simulations

We have employed two different input configurations of particles for our simulations. One corresponds to an approximately uniform particle distribution, representative of Molecular Dynamics applications, and the other corresponds to non-uniform particle distributions (Plummer), typical of Astrophysics simulations.

In addition to particle locations, two algorithmic parameters must be specified at the input of FAST: one is the number of multipole expansion coefficients sought and the other is the number of particles per quadtree leaf. In the simulations presented here, the size of the multipole expansions was set to ten coefficients. This guarantees highly accurate results for the corresponding actual FMM computation and, at the same time, maintains the low time complexity of the Fast Multipole algorithm with respect to the brute-force method.

The choice of the quadtree granularity affects many aspects of the parallel execution: the available parallelism and its granularity, communication overhead, the computation-to-communication ratio, and the overall parallel time. From our measurements with FAST, we concluded that small granularities (fewer than ten particles per quadtree leaf) lead to relatively high communication overhead, very small computation-to-communication ratios, and thus to larger parallel times. On the other hand, granularities larger than twenty particles per leaf result in larger sequential tasks and limit the available parallelism. In this paper, we present results derived with a quadtree granularity of *fifteen particles per leaf*, unless stated otherwise. This choice achieves good parallel time for the hardware parameters adopted, over a wide range of problem sizes.

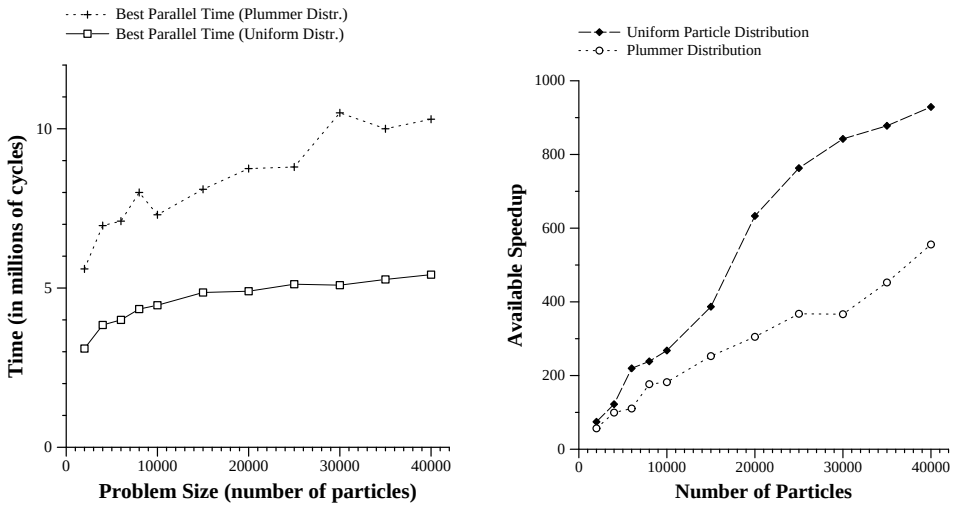

Figure 3: Data about the limiting scalability of the parallel FMM

3.3. Profiling of the FMM

The clustering stage of FAST (Figure 1) creates the clustered parallel-execution graph, which describes the processing of the Multipole Method on a parallel architecture with a clique interconnection and as many processors as task-clusters. From the parallel-execution graph we can easily estimate the parallel time of the execution, and thus measure the speedups achieved on an abundant clique and assess the scalability of the parallel algorithm. In Figure 3 (left), we present parallel execution times for problems of 2,000 to 40,000 bodies distributed according to uniform and Plummer distributions. The times in this diagram correspond to the minimum estimates from a set of FAST experiments with different clustering heuristics.

In Figure 3 (right), we present the *available speedup* sustained by a parallel implementation of the Fast Multipole Method, as the problem size increases and abundant hardware resources are accessible. This represents an estimate of the scalability of the algorithm with respect to problem size. In that sense, the parallel FMM is scalable because for larger problem sizes, greater speedups can be achieved if more processors, memory, and links are available. The parallel FMM would not be scalable if speedups leveled off for larger problem sizes; this would signal the existence of significant sequential parts, rapidly increasing with the problem size. Moreover, we notice that in the non-uniform case (Plummer distribution) the available speedup increases more slowly with problem size than in the uniform case: non-uniformity results in higher and "tighter" decomposition trees and thus in less available parallelism.

From the parallel-execution graph we can also extract the profile of active tasks and busy links during parallel execution on the abundant clique. This profile reveals characteristics inherent to the algorithm at hand and is not influenced by partitioning and mapping to a specific multiprocessor. Experiments with differ-

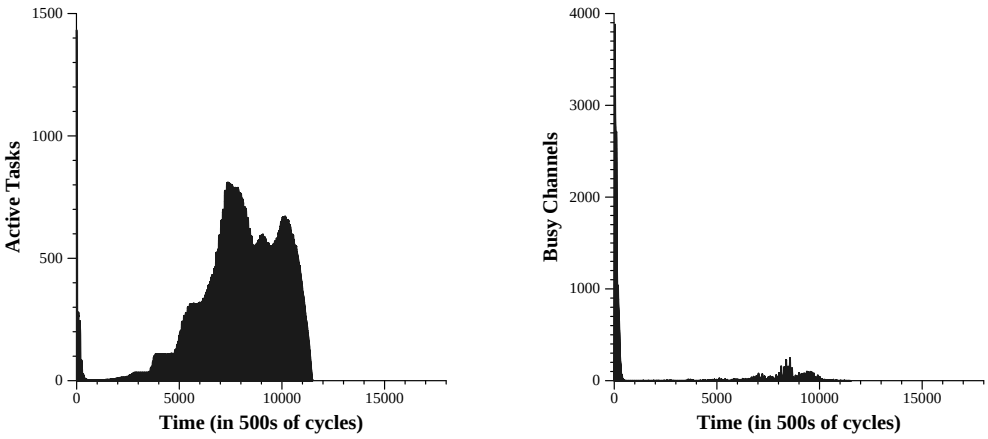

Figure 4: Execution Profiles for a 15,000-particles problem (Uniform distribution).

ent clustering heuristics led to similar results and conclusions (see [5]). Figure 4 presents such a profile for a parallel execution of the Fast Multipole Method on a problem instance with 15,000 particles distributed according to the uniform distribution, fifteen particles per quadtree-leaf, and a ten coefficient approximation. In this case we used a clustering technique combining heuristics from [11] and [16]. Profiles from non-uniform distributions have similar shape.

From these plots it is clear that the parallel execution has three phases: a short phase at the beginning is defined by a large number of active tasks indicating a high degree of available parallelism. This is followed by a long period during which the available parallelism is very low. The execution ends in a third, long phase where the number of active tasks is high. Similar remarks hold for the channel utilizations in the abundant clique. The first phase of the parallel execution corresponds to the upward step of the Fast Multipole algorithm: in the beginning, many tasks calculate the multipole coefficients at the leaves of the decomposition tree in parallel; the results are sent to tasks accumulating these coefficients in nodes at lower levels of the tree and so on. As the algorithm moves towards the root, the number of internal nodes decreases logarithmically and thus the number of parallel tasks drops very quickly. In the second phase, the algorithm moves from the root of the tree to its nodes, exchanging messages between nodes belonging to the same neighborhoods. The available parallelism is very small initially and increases as the algorithm approaches the quadtree leaves. In the final phase, nearest-neighbor computations and message exchanges take place.

Another interesting observation relates to resource utilization of the abundant clique. For instance, the parallel-execution graph of the above example has 2384 clusters and therefore, the corresponding abundant clique would have 2384 processors and $5,681,072$ unidirectional links. However, the average number of busy processors over the parallel execution time is 308, that is, 12% of the processors in the abundant clique. The average number of used links over time is as low as 51,

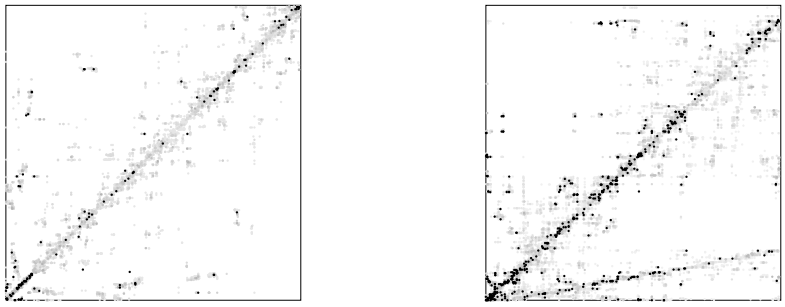

Figure 5: Matrix Communication Patterns for the 15,000-particle problem.

which corresponds to 0.08% of the total number of links. Therefore, it is conceivable that a much sparser architecture with fewer processors could achieve essentially the same speedups as the abundant clique.

This can be seen also from Figure 5: the plot to the left presents communication patterns for the case where bodies are distributed uniformly, whereas the plot to the right presents the patterns extracted from the non-uniform distribution case. The horizontal and vertical axes correspond to processors of the abundant clique, that is, to task-clusters of the clustered parallel-execution graph. Points in the plot represent the occurrence of messages sent between the task-clusters; darker points correspond to larger numbers of messages between the corresponding clusters.

3.4. Hypercube, Ring and Multiring Performance for the Fast Multipole Method

The communication patterns plotted in Figure 5 show that destination clusters of messages dispatched from any task-cluster tend to belong to small neighborhoods and have numberings close to the number assigned to the source cluster. This observation suggests that a ring interconnection might achieve speedups comparable to those achieved on a clique.

Using FAST, we mapped the clustered parallel-execution graphs onto clique, hypercube, and ring interconnections with various numbers of processors. From these experiments, we found that ring interconnections represent a cost-effective architectural choice for implementing the Fast Multipole Method in parallel. For example, in a 10,000-particle simulation with 15 particles per quadtree leaf and ten coefficients, a 128-processor ring achieves 50% of the speedup of the clique with only 1.57% of its links. In a 256-processor configuration, the ring achieves the 40% of the clique speedup with only the 0.78% of its links.

Speedups measured on rings are not as high as the ones achieved on cliques, simply because the ring is a much sparser interconnection and thus link-contention causes extra delays in message propagation times. This is confirmed by the diagrams in Figure 6, which display the average message delay and message congestion measured with FAST for the $10,000$-particle example (Plummer distribution). Con-

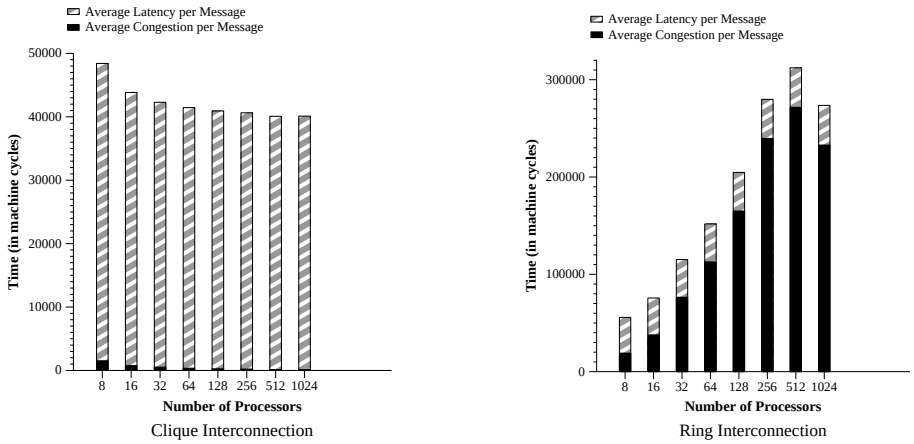

Figure 6: Contribution of Congestion to Message Latency. Note the difference in the y-axis (time scale) of the two diagrams.

gestion figures correspond to average time spent by each message while waiting in queues because of link and network interface contention. From these graphs it is clear that congestion constitutes the largest portion of message latencies measured in the rings. On the other hand, communication contention in the cliques is practically nonexistent.

The above remarks suggest that spending extra hardware to reduce ring contention might result in substantially improved speedups for the ring interconnection. A straightforward way to reduce contention is by using multiring instead of single-ring communication networks. Building such interconnections is feasible and much cheaper than building cliques or hypercubes with the same number of processors.

Our functional simulations proved that multirings are also cost-effective: Figure 7 presents the speedups reported by FAST for the $10,000$-particle problem mapped on hypercubes and multirings with two to sixteen rings. It is clear that an increase in the number of rings improves significantly the attained speedups. Moreover, it enhances the cost-effectiveness of the ring implementation: for instance, in a 128-processor machine running the parallel FMM on $10,000$ particles distributed according to the Plummer model, the four-ring achieves 83% of the speedup of the clique with only 6.3% of its links. As another example, the 512-processor four-ring achieves a speedup slightly larger than the one attained by a 256-processor clique.

4. Conclusions

In this paper, we described a case study of a parallel version of the Fast Multipole Method with FAST, a software system implementing a new approach for modeling the parallel execution of a class of important scientific applications. FAST enabled us to perform an architecture-independent analysis of the algorithm over a large number of realistic data sets and for various algorithmic parameters. Furthermore, it provided us with useful information regarding communication patterns occurring

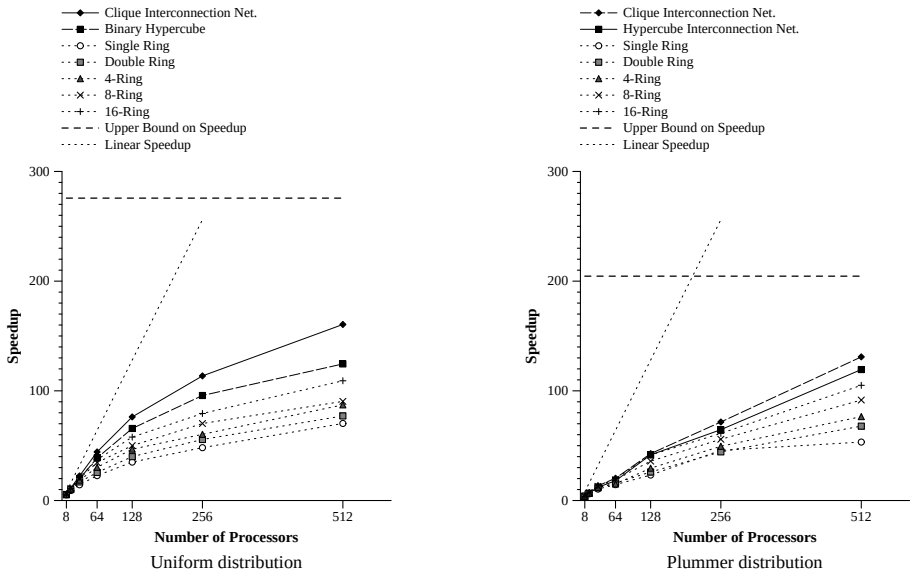

Figure 7: Multiring Performance.

in the parallel execution, the variation of available parallelism during different algorithmic phases, and upper bounds on available speedups for different problem sizes. This information suggested that an interconnection topology as simple as the ring can achieve satisfactory performance.

Subsequently, FAST allowed us to estimate the parallel performance of the FMM on message-passing multiprocessors with hypercube, ring and multiring interconnection topologies. Performance figures derived from the mapping of the FMM to cliques were used to evaluate the effectiveness of the ring and multiring implementations. Our simulations showed that an implementation of the Multipole algorithm on scalable ring or multiring architectures is cost-effective.

Acknowledgements

This work was supported by NSF grants MIP-8912100, MIP-9201484, and ASC-9110766.

References

1. L. Greengard and V. Rokhlin, A fast algorithm for particle simulation, *Journal of Computational Physics*, **73** (1987), 325–348.

2. J.P. Singh, J.L. Hennessy and A. Gupta, Implications of Hierarchical N-body Methods for Multiprocessor Architecture, **CSL-TR-92-506**, *Computer Systems Lab, Stanford University*, 1992.

3. R. Cypher, A. Ho, S. Konstantinidou and P. Messina, Architectural Requirements of Parallel Scientific Applications with Explicit Communication, in *Proc., 20th Annual*

International Symposium on Computer Architecture, San Diego, CA., May 1993, 2–13.

4. S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis and D. Wood, The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers, **1122**, *Computer Sciences Dept., University of Wisconsin-Madison*, Nov. 1992.
5. M. D. Dikaiakos, FAST: A Functional Algorithm Simulation Testbed, Ph.D. Thesis, Princeton University, 1994.
6. D. Culler, R. Karp, D. Patterson et al, LogP: Towards a Realistic Model of Parallel Computation, in *Proc., Fourth ACM Sigplan Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.
7. J. Barnes and P. Hut, A Hierarchical O(N)log(N) force-calculation algorithm, *Nature*, **87** (1990) 161–170.
8. A.W. Appel, An efficient program for many-body simulation, *SIAM J. Sci. Stat. Computing*, **6** (1986) 85–103.
9. T. Chan, Hierarchical Algorithms and Architectures for Parallel Scientific Computing, *Proc., International Conference on Supercomputing*, Amsterdam, Netherlands, September 1990, 318–329.
10. C. Lin and L. Snyder, A Portable Implementation of SIMPLE, *International Journal of Parallel Programming*, **20** (1991), 363–401.
11. V. Sarkar, Partitioning and Scheduling Parallel Programs for Multiprocessors, (MIT Press, 1989).
12. B. Boothe, Fast Accurate Simulation of Large Shared Memory Multiprocessors, **UCB/CSD 92/682** *Computer Science Division, University of California at Berkeley.*
13. M. Dikaiakos, A. Rogers and K. Steiglitz, Functional Algorithm Simulation: Implementation and Experiments, **TR-429-93**, *Dept. of Computer Science, Princeton University*, June 1993.
14. M. Dikaiakos, A. Rogers, and K. Steiglitz, FAST: A Functional Algorithm Simulation Testbed, in *Proc. of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems – MASCOTS'94* (IEEE Press 1994), 142–146.
15. M. Dikaiakos, A. Rogers and K. Steiglitz, A Comparison of Techniques Used for Mapping Parallel Algorithms to Message-Passing Multiprocessors, in *Proc. of the 6th IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, Oct. 1994, IEEE Computer Press, 434–442.
16. A. Gerasoulis and S. Venugopal and T. Yang, Clustering Task Graphs for Message Passing Architectures, in *Proc. of the 1990 International Conference on Supercomputing*, 447–457
17. M. Dikaiakos and J. Stadel, A Performance Study of Cosmological Simulations on Message-Passing and Shared-Memory Multiprocessors. (*Submitted for Publication, March 1995*).