FAST: A Functional Algorithm Simulation Testbed

Marios D. Dikaiakos

Research Report CS-TR-444-94
June 1994

FAST: A FUNCTIONAL ALGORITHM SIMULATION TESTBED

Marios D. Dikaiakos

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

June 1994

© Copyright by Marios D. Dikaiakos 1994
All Rights Reserved

Στον Δικαίο, τη Φρέγια

και τον Μιχάλη

Acknowledgements

I am indebted to my advisor, Kenneth Steiglitz. Ken has been an endless source of advice, help, and good spirits. I am also thankful to my co-advisor, Anne Rogers, for her enthusiasm, support, and her constructive criticism on my research. I am thankful to Kai Li, not only for reading this thesis and providing useful comments on its format, but also for always being willing to discuss and comment on my work.

Thanks to the staff of the Computer Science Department for keeping the computer systems up and running. Thanks to Vernon Holmes, Melissa Lawson, and Trish Killian Zabielski for their help with administrative details.

I am grateful to Alvaro Campos, Dimitris Gunopulos, Sarantos Kapidakis, Gordana Obuskovic, Leonidas Palios, and Christos Polyzois for their friendship, help and for the good times we had together. Thanks to Dimitris and Giota Doukas for being always hospitable and friendly, and for sharing with me, so often, the longing of greek summers. Thanks to Yiota Pappas for her affectionate support. Thanks to the members of the electronic-discussion list "macedonia," and especially to John Koukos, Danny Houdaverdis, Panos Sarantopoulos, and Stratos Safioleas, for the very many electronic discussions; I enjoyed every single bit of them. Last but not least, many thanks to my dearest friend Nina Obuskovic for her love and support.

It is difficult to find words to express my gratitude to my mother Freya and my father Dikaios for their affection, their sacrifices, and the unlimited support they gave me. This thesis is lovingly dedicated to them, and to my first friend, my brother Michalis.

I thank Intel Supercomputer Systems Division for providing an iPSC/860 for measurements. My thesis work has been funded by NSF Grants MIP-8912100 and

MIP-9201484, and by U.S. Army Research Office-Durham Contract DAAL03-89-K-0074.

Gift Silver Poem

I know that all this is worthless and that the language
I speak doesn't have an alphabet
Since the sun and the waves are a syllabic script
which can be deciphered only in the years of sorrow and exile
And the motherland a fresco with successive overlays
frankish or slavic which, should you try to restore,
you are immediately sent to prison and
held responsible
To a crowd of foreign Powers always through
the intervention of your own
As it happens for the disasters
But let's imagine that in an old days' threshing-floor
which might be in an apartment-complex children
are playing and whoever loses
Should, according to the rules, tell the others
and give them a truth
Then everyone ends up holding in his
hand a small
Gift, silver poem.

Odysseas Elytis
"The Tree of Light and The Fourteenth Beauty"

Abstract

Substantial progress in Parallel Scientific Computation will emerge from improvements in the effectiveness of current parallel machines, the development of new scientific algorithms, and the employment of more powerful multiprocessors. Successfully addressing these issues in a cost-effective way requires extensive experimentation. Nevertheless, the large computational complexity of scientific applications and the high cost of multiprocessor systems, make the quantitative and qualitative analyses of parallel workloads a difficult and costly endeavor.

In this thesis, I address some of the issues involved in the modeling and evaluation of parallel scientific computations. More specifically, I introduce Functional Algorithm Simulation, that is, simulation without performing the bulk of numerical calculations involved in the applications studied. Functional Algorithm Simulation is applicable in the evaluation of algorithms simulating complex systems, for which the core-set of time-consuming calculations and data-exchanges can be determined from input information, before the actual computations take place. To assess the principles of Functional Algorithm Simulation I built the *Functional Algorithm Simulation Testbed (FAST)*, a software prototype system for approximately simulating the parallel execution of such algorithms on uniprocessor workstations. FAST has been used to evaluate parallel executions of three interesting and important scientific algorithms: SIMPLE, a Computational Fluid Dynamics code; the Fast Multipole Method, and a modified version of the Barnes-Hut algorithm, which solve the N-Body problem and have applications in Computational Molecular Dynamics and Astrophysics. Experimentation with FAST shows that approximate simulation can give valid and useful results. FAST enables us to study parallel executions of much larger problem sizes

and more processors than those reported so far, with modest computing resources. Also, it allows us to collect detailed information characterizing parallel executions on various message-passing architectures, analyze the effects of communication overhead to parallel performance, and study the scalability of parallel algorithms.

Contents

iii

Acknowledgements iv

Abstract vii

1 Introduction 1
- 1.1 Functional Algorithm Simulation 4
  - 1.1.1 Principles of Functional Algorithm Simulation 4
- 1.2 A prototype testbed 6
- 1.3 Related Work 7
- 1.4 Contributions 9
- 1.5 Thesis Outline 12

2 A Functional Algorithm Simulation Testbed 13
- 2.1 Structure of FAST 13
- 2.2 Front-End 15
  - 2.2.1 Intermediate Representation 15
  - 2.2.2 Task-flow graphs 20
  - 2.2.3 Partitioning 22
- 2.3 Back-End 24
  - 2.3.1 Parallel Execution Model 24
  - 2.3.2 Clustering and Mapping 27
  - 2.3.3 Parallel Architecture Model 28

3 Clustering and Mapping 34
3.1 Clustering 36
3.2 Clustering Heuristics 40
3.2.1 Sarkar's Clustering Algorithm 41
3.2.2 Kim and Browne's Algorithm 42
3.2.3 Linear-Greedy Algorithm 42
3.2.4 Greedy Dominant Sequence Algorithm 43
3.3 Scheduling 44
3.3.1 TS/ETF Scheduling 45
3.3.2 CP/MISF Scheduling 46
3.4 Mapping 46
3.4.1 Sarkar's Algorithm 46
3.4.2 Yang and Gerasoulis' Algorithm 47
3.5 Experimental Results 48
3.5.1 Clustering 48
3.5.2 Mapping 52
3.6 Remarks 55

4 Message Ordering 56
4.1 Message Ordering 56
4.2 Optimal Orderings of Messages 58
4.3 An Algorithm for Optimal Message Ordering 63
4.4 Remarks 67

5 Validation of FAST on the SIMPLE CFD-kernel. 68
5.1 Functional Algorithm Simulation of SIMPLE 69
5.1.1 Derivation of the task-flow graph 69
5.1.2 Validating FAST 70
5.1.3 Validation experiments 72
5.1.4 Using FAST to collect profiles for SIMPLE 73
5.2 Scalability Analysis of SIMPLE with FAST 77
5.3 Some remarks on SIMPLE 82

6 **A Study of the Fast Multipole Method** 83
6.1 Setup of Functional Algorithm Simulations 86
6.1.1 N-body problem configurations 86
6.1.2 Derivation of the task-flow graph 87
6.1.3 Clustering and Mapping heuristics employed 90
6.1.4 Hardware parameters adopted 91
6.2 Remarks on the ad-hoc partitioning 92
6.3 Profiling the FMM 94
6.3.1 Remarks on the Scalability of the FMM 94
6.3.2 Profiles of parallel execution 95
6.3.3 Phases of the Fast Multipole Method 97
6.3.4 Resource requirements 98
6.4 Performance of the Fast Multipole Method on a Ring Multiprocessor 99
6.5 Performance of the Fast Multipole Method on Multirings 101
6.6 Some remarks on the FMM 103

7 **A Study of the Barnes-Hut Algorithm** 105
7.1 A modified version of the Barnes-Hut algorithm 106
7.2 Experiments with the Barnes-Hut algorithm 110
7.2.1 Profiles of parallel execution 111
7.2.2 Remarks on the Scalability of Barnes - Hut algorithm 114
7.2.3 Comparing the Barnes-Hut with the Fast Multipole Method . 115
7.3 Remarks on the Barnes-Hut algorithm 118

8 **Conclusions and Future Work** 120
8.1 Future Work 122

Bibliography 130

List of Tables

1 Pseudo-code. 16

2 Fraction of the Intermediate Representation for SIMPLE. 20

3 Hardware parameters used in FAST. 33

4 Clustering and Mapping algorithms' notation. 53

5 Description of the operations at quadtree-leaves. 88

6 Fraction of the Intermediate Representation for the FMM. 91

List of Figures

1	Front-end of the FAST.	15
2	Subset of the task-flow graph.	21
3	Implicit Partitioning.	22
4	Partitioning.	23
5	Back-end of the FAST.	25
6	Processor-node structure.	29
7	8-processor clique, binary hypercube, and double-ring.	30
8	Blocking vs. Non-blocking *Send*'s.	31
9	Different clustering choices.	37
10	Scheduling edges. The bold arrows denote the sequential paths of execution in the clusters.	38
11	Edge weights (blocking *Send*'s).	39
12	Scheduling problem: the bold edges represent the sequential thread of execution in each cluster.	44
13	Effects of clustering to parallel time.	49
14	Number of clusters.	50
15	Clustering with non-blocking *Send*'s.	51
16	Execution times for the Clustering heuristics.	52
17	Speedups for different clustering-mapping strategies (16 processors).	53
18	Execution times for the Mapping heuristics.	54
19	Example of communication between tasks and between processors.	57
20	Communication overheads under the blocking communication approach.	58
21	Message Ordering Alteration.	61

22	Message Ordering Alteration.	61
23	Message Ordering Alteration.	63
24	Induction Step.	67
25	Strip vs. Square partitioning.	69
26	Parallel Time and Speedup for SIMPLE on 4096-point grid.	71
27	Strip vs. Square partitioning of 2^{20} grid-point SIMPLE on a hypercube.	73
28	FAST results for 4096 grid-point SIMPLE on a 64-processor *clique* (PT = 220022 μs).	74
29	FAST results for 4096 grid-point SIMPLE on a 64-processor *hypercube* (PT = 232988 μs).	75
30	FAST results for 4096 grid-point SIMPLE on a 256-processor *clique*.	75
31	FAST results for 4096 grid-point SIMPLE on a 1024-processor *clique*.	76
32	Communication patterns for 64-processor clique, 4096 grid-points SIMPLE instance.	77
33	Speedups reported by FAST for SIMPLE executions.	78
34	Computation profile of runs on 4096-point grid with zero communication costs (1024 processors).	78
35	Fraction of sequential execution time that cannot be parallelized.	79
36	Speedups, average task-execution times and average message delays (4096-point grid).	80
37	Speedups, average task-execution times and average message delays (16384-point grid).	81
38	Decomposition of a two-dimensional space of particles and the corresponding quadtree created by the Fast Multipole Method.	84
39	Particle distributions (5000 particles).	86
40	Assignment of id's to quadtree nodes.	90
41	Task-flow subgraph (FMM).	92
42	Parallel Times derived with FAST for the FMM mapped on clique interconnections (uniform distributions, ad-hoc partitioning).	93
43	FAST results about performance of the parallel FMM on an *abundant clique*.	94

44 Execution Profiles for a 15,000-particles problem (Uniform distribution). 95
45 Execution Profiles for a 15,000-particles problem (Plummer distribution). 96
46 Execution Profiles for a 15,000-particles problem partitioned into 1024 blocks (Uniform distribution). 97
47 Communication Patterns for the 15,000-particle problem. 98
48 Speedups for a 10,000-particle problem. 99
49 Contribution of Congestion to Message Latency for a 10,000-particles problem (Plummer distribution). Notice the difference in the time-scale of the two diagrams. 101
50 Multiring Performance. 102
51 Efficiency of Multiring interconnections. 103
52 Distance-definitions for neighborhood calculation. 107
53 Execution Profiles for a 10,000-particle problem mapped on an abundant clique (Uniform distribution). 110
54 Execution Profiles for a 10,000-particle problem mapped on an abundant clique (Plummer distribution). 111
55 Execution Profiles for a 10,000-particle problem mapped on a 128-processor clique (Uniform particle distribution). 112
56 Ad-hoc partitioning (eight-processor case). 113
57 Communication Patterns for the 10,000-particle problem. 114
58 FAST results showing the performance of the parallel Barnes-Hut method on an abundant clique. 115
59 8,000-particle example (Uniform distribution). 116
60 Parallel Times reported by FAST for the FMM and the B-H algorithms mapped on clique architectures (8,000-particles example - Uniform distribution). 117
61 Parallel Times reported by FAST for the FMM and the B-H algorithms mapped on clique architectures (10,000-particles example - Plummer distribution). 117
62 Speedups reported by FAST for the FMM and the B-H algorithms mapped on clique architectures. 118

Chapter 1

Introduction

In this thesis we show that approximate simulation extends the flexibility and range of simulations of parallel scientific workloads substantially. Systems that incorporate approximate simulation techniques can be used to study parallel scientific algorithms applied on data-sets of practical interest and mapped to message-passing multiprocessors with tens to thousands of processors. Studies of this kind can provide accurate and detailed information about the algorithms examined. This information can be used to guide the design of parallel algorithms and the selection of cost-effective architectures.

Substantial progress in parallel scientific computation will emerge from improvements in the effectiveness of current parallel machines, the development of new scientific algorithms, and the employment of more powerful multiprocessors. Successfully addressing these issues requires extensive experimentation that will provide us with a broad and thorough understanding of parallel scientific computations. The high computational complexity of scientific applications and the high cost of multiprocessor systems, however, make the quantitative and qualitative analyses of parallel workloads a difficult and costly endeavor. Therefore, effectively modeling the execution of algorithms on massively parallel machines is a fundamental problem in parallel computing. The development of good modeling methodologies will enable researchers to generate, quantify, and evaluate computation and communication profiles characterizing the implementation of algorithms on multiprocessors. Knowledge of these

characteristics is valuable for the design of parallel algorithms and the understanding of parallel architectures: profile information describing the parallel execution of an algorithm can be used to identify bottlenecks and estimate their effects on overall performance. Analysis of realistic computation and communication characteristics can lead to better understanding of issues like the tradeoffs between parallelism and communication overhead and the scalability of algorithms. Moreover, it can enhance the exploration of different architecture and hardware tradeoffs and give useful directions for selecting or re-designing cost-effective, high-performance interconnection topologies that optimize the parallel execution of applications with common computation and communication characteristics.

A straightforward approach for collecting information of this kind is to profile programs running on parallel machines. Running programs that implement interesting parallel algorithms computing on realistic data-sets takes a significant amount of time. The volume of data collected to describe the computations and communications can also be large, imposing extensive storage and I/O requirements [48, 70]. Furthermore, software monitoring tools are very complex and, they cannot observe the behavior of hardware communication components easily [32]. In addition, results derived from profiling applications running on some multiprocessor are influenced, to a large extent, by the architecture and hardware characteristics of the specific multiprocessor. Scalability studies are restricted to the limited number of available processors. Monitoring real implementations is hindered by time considerations and the high-cost and low-availability of massively parallel computers. Monitoring also lacks the desirable flexibility for performing comparative studies and scalability analyses.

Simulators of parallel workloads are used to overcome some of the shortcomings of profiling. Exact simulation can be *trace-driven* or *direct*. The trace-driven approach is accomplished in two successive phases: *trace generation* and *trace simulation*. Trace generation collects trace data from a real parallel execution through hardware [2] or software [71] instrumentation. Trace simulation uses the collected trace to emulate the corresponding workload running on the parallel machine under consideration. If this machine is different from the one used to generate the trace, the results of the simulation can be inaccurate because the execution path in a parallel computation

may depend on the ordering of events on different processors, which in turn depends on the characteristics of the parallel machine [26]. Another drawback of the trace-driven approach is the large space and time required to generate and store the traces. In direct simulation, traces are not required since the simulator emulates instructions, messages, shared data accesses, and synchronization operations as they occur on the multiprocessor studied. Direct simulation incurs a significant slowdown with respect to uniprocessor execution. This is a major obstacle for using direct simulation to evaluate massively parallel computers. Exact simulation (trace driven or direct) represents a useful alternative to profiling when considering systems of small to medium size. The user has the option to change several parameters of the simulator, such as the number of processors, the processor speed, communication bandwidth, and cache size. Then, through simulation he can study the relationship between these parameters, the algorithm examined, and the performance measurements. Since the study of parallel executions require the simulation of millions of instructions and thousands of messages, such tasks are long-running and consume a lot of space. Thus, simulation of massively parallel machines (with hundreds or thousands of processors) is practically impossible, unless performed in a distributed fashion [57].

Theoretical models of parallel computation such as the various flavors of *PRAM* [25], *BSP* [73], and *LogP* [12] provide the algorithm designers with a convenient platform for formally expressing new algorithms and studying theoretically their performance. Nevertheless, they rely on unrealistic assumptions like zero communication costs and infinite bandwidth (in the case of PRAM's). Moreover, they have restricted capabilities for expressing communication characteristics (e.g. congestion) and evaluating the performance of parallel algorithms with irregular communication patterns and dynamic behavior.

More realistic methods of interconnection network analysis and performance evaluation rely on static-graph statistics such as average or maximum distance between nodes [16] and on analytical queueing models assuming uniform, random traffic [64]. Measurement of the performance of interconnection networks is carried out with simulations of queueing models or with the use of synthetic computation and communication benchmarks [64]. These methods represent useful ways of comparing different

interconnection strategies and proving general principles about network design. Nevertheless, they are based on assumptions that do not correspond to real-application traffic loads and communication patterns or to realistic hardware parameters.

1.1 Functional Algorithm Simulation

Methodologies for modeling and evaluating parallel computations should satisfy a certain number of requirements:

1. They should be *flexible* to allow the assessment of a large variety of parallel machines with processor numbers ranging from tens to thousands, with different interconnection topologies and message-passing interface primitives, various processor speeds and channel bandwidths, etc.

2. They should be *general* enough to apply in the study of a comprehensive group of algorithms running on realistic sets of data.

3. Implementing applications on parallel machines is a difficult, error-prone, and expensive task. Therefore we seek modeling techniques that allow algorithmic and architectural assessments with a *moderate cost* in terms of development and evaluation time, and invested money.

4. Results obtained by such modeling methodologies should be fairly *accurate* so as to guide the further development of algorithms and architectures.

Flexibility enables us to compare different architectures. Also, it helps us assess the effects of hardware characteristics and interconnection networks on parallel performance. Finally, the accuracy of results and their corroboration by real implementations or theoretical analyses is necessary for deriving proper understanding of the algorithms under investigation and on the hardware tradeoffs examined.

1.1.1 Principles of Functional Algorithm Simulation

We focus on computationally intensive scientific applications that simulate *complex systems* [22]. A complex system is defined with respect to a basic data-structure

which we call the *computational domain.* This is a set of non-decomposable entities, the *domain-nodes.* Every domain-node is labeled with one or more variables called *components.* The set of all the components across the computational domain represents the physical (or other) properties of the complex system that the simulation seeks to evaluate.

An algorithm simulating a complex system determines the computations that must be performed upon the components of domain-nodes for updating the system's state. The algorithm also determines the set of data-dependences among the different components of the system which must be carried out for the computation to proceed. The simulation of a complex system is performed for a number of iterations (or time-steps). After each iteration (time-step), the values of the components of the domain-nodes change and the complex system enters a different state. In a real implementation, for each iteration, the algorithm first constructs or updates the data-structures representing the computational domain. Then, it proceeds with the computation of component values at the domain-nodes.

To satisfy the requirements for modeling methodologies mentioned earlier, we developed a new approach exploiting two properties common to many of the algorithms of interest:

1. Most of the parallel execution time is spent in expensive numerical operations/procedures.

2. The number and the nature of these operations as well as the data-dependences and communication occurring among them can be determined at the initialization phase of the algorithms, after the input of data and the setup of the computational domains.

By generating the computational domain of an algorithm and identifying the set of "core" operations and communications performed upon it, we are able to derive profiles characterizing an algorithm without having to simulate all instructions involved in its parallel execution.

With this remark in mind we introduce *Functional Algorithm Simulation*, that is, simulation of parallel computations without actually simulating or executing most of

the numerical calculations involved. Our approach reproduces the *skeletons* of parallel computations and uses them to extract the basic computation and communication patterns sought. Therefore, Functional Algorithm Simulation 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* [12].

By not doing the numerical calculations, Functional Algorithm Simulation achieves orders of magnitude savings in terms of processor cycles and memory space. 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.

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 having applications in Computational Fluid Dynamics [42], Finite Difference Methods [22], Astrophysics and Computational Molecular Biology (tree-algorithms for the N-Body problem) [6, 28], Multigrid techniques [50] and Lattice-Gas computations [39].

1.2 A prototype testbed

To implement Functional Algorithm Simulation and test its capabilities, we developed a prototype system called *FAST (Fast Algorithm Simulation Testbed)*. FAST overcomes some of the difficulties imposed by the very high complexity of interesting scientific algorithms, collects profile information representative of the algorithms rather than the underlying mapping strategies and hardware-design choices, and allows a performance assessment of parallel machines with various sizes and different interconnection schemes.

We used FAST to study three scientific algorithms: the first is *SIMPLE*, a popular *Computational Fluid Dynamics* benchmark simulating the hydrodynamics of a pressurized fluid inside a spherical cell [55]. The second is the *Fast Multipole Method* [30], and the third is a modified version of the *Barnes-Hut* algorithm [6]. The Fast Multipole Method and the Barnes-Hut algorithm solve the *N-body problem* [22] for a two- or three- dimensional particle space and have applications in Astrophysics, Plasma Physics, Molecular Dynamics, and Fluid Dynamics.

SIMPLE has been used frequently for the assessment of parallel programming tools [45, 55, 60] and shows uniform computation and communication characteristics. We used it as a test-case for validating FAST and highlighting its flexibility and efficiency. The Fast Multipole and the Barnes-Hut algorithms exhibit irregular and non-uniform traits due to the long-range interactions of the N-body problem. Their high complexity makes the study of those traits a challenging task. With FAST we were able to collect computation and communication patterns from parallel executions of the FMM and BH on realistic data-sets, on message-passing systems with various sizes and different communication networks. Those patterns allowed us to derive interesting conclusions about the scalability and efficiency of the algorithms, and to compare the performance of different interconnection networks.

1.3 Related Work

In this section, we briefly discuss previous and ongoing research with motivation similar to ours. A number of researchers have been working on collecting communication and computation patterns from real-life applications. Their main goal is to understand and describe the parallel execution of these applications better and use this understanding to design interconnection structures and parallel computers better suited to computationally demanding problems.

One project developed a software monitoring tool for the hypercube architecture [32]. This tool allows the collection of communication traces, the measurement of communication localities and processor utilizations, the estimation of average message lengths, the existence of hot-spots, etc. It is used for performance evaluation,

workload characterization, and trace-driven simulation of a hypercube that runs realistic workloads corresponding to six CAD applications and numerical problems. The focus is on evaluating the hypercube and thus the results are representative of the hypercube architecture and interconnection scheme. It is difficult to assess the performance of the applications at hand for a wide range of processor numbers and different interconnection topologies. Furthermore, experimentation with larger, more time- and space-consuming applications will still experience the problems raised by the overall high complexity that we mentioned earlier.

A second approach comes from the DASH group at Stanford University [66, 68]. They evaluated the performance of parallel implementations of algorithms solving the N-body problem on the sixteen-processor shared-memory machine DASH [44]. Issues related to performance on larger machines and scalability were studied on a simulator of an idealized shared-memory multiprocessor architecture. Measurements of parallel time and speedup from real application data were used to compare different techniques for partitioning the N-body problem into parallel tasks. The results are representative of the shared-memory paradigm, the DASH architecture and interconnection network, and the partitioning techniques examined.

A third project is concerned with the design and development of a reconfigurable network evaluation testbed built with state-of-the-art, high-performance hardware [69]. This testbed consists of sixty-four user-configurable switch nodes used to construct a variety of interconnection networks. The topology, routing algorithm, and conflict resolution mechanism of the network can be varied. Furthermore, the switches contain hardware for monitoring and correlating network behavior with parallel application behavior. This approach can study communication performance and collect computation and communication patterns from real applications on various network setups. An interesting question is whether the testbed is scalable, namely whether its number of switches and processors can be increased while maintaining the flexibility of network reconfiguration.

Another tool named *FAST* has been developed in Berkeley to simulate large shared memory multiprocessors accurately at simulation speeds that are one to two orders of magnitude faster than comparable simulators [10]. This tool involves execution

driven simulation techniques which modify the object code of the application programs studied. These produce an augmented version of the code that does most of the simulation work.

A thorough study of eight scientific applications, running on four message-passing multiprocessors (512-processor Intel Delta, 256-processor nCube-1 and 64-processor nCube-2) is presented in [13]. The authors collected data describing floating-point operation, memory, I/O and communication requirements, and discovered existing traffic patterns. The study and analysis was performed for the complete applications, including the loading of input data and the storing of results. Several conclusions were derived about the common traits of the programs and about how these could influence machine design and scalability. It is pointed out that the results presented were necessarily affected by characteristics of the parallel programs and the parallel machines used.

Finally, the Wisconsin Wind Tunnel developed at the University of Wisconsin runs parallel shared-memory programs on a message-passing multiprocessor (CM-5) [57]. Concurrently with program execution, the WWT uses distributed discrete-event simulation to calculate the programs' execution times on a proposed shared-memory system. The WWT can be used to evaluate cache-coherent, shared-memory multiprocessors by simulating their execution on a powerful message-passing system.

1.4 Contributions

In this thesis, we address some of the issues involved in the modeling and evaluation of parallel scientific computations. We introduce *Functional Algorithm Simulation*, that is, simulation of parallel executions without performing the bulk of numerical calculations involved in the algorithms studied. To evaluate Functional Algorithm Simulation we have built a prototype system called FAST, and we use it to study three interesting and important scientific algorithms. Functional Algorithm Simulation extends the flexibility and the practical range of parallel execution simulations. More specifically, we simulate a large range of parallel *message-passing* systems with:

1. Tens to thousands of processors.

2. Various interconnection topologies, such as cliques, hypercubes, rings, and multirings.

3. Different hardware characteristics (processor speed, bandwidth of communication channels, communication overhead).

FAST simulations of practical parallel workloads can be accomplished with modest computing resources (Sparc and DEC workstations). The savings in terms of simulation time are impressive. For example, a functional algorithm simulation of an instance of SIMPLE took *0.63 secs* to complete on a Sparcstation. The same instance took *9.8 secs* to run on one iPSC/2 node. The slowdown factor of exact simulation (trace-driven of direct), that is, the number of cycles it takes to simulate a single cycle of execution for a single processor, ranges from *100's* to *1000's* [2, 26]. Therefore, we can easily deduce that functional algorithm simulation decreases the simulation time by at least *three orders of magnitude*.

The following table highlights some of the advantages of FAST. Data reported correspond to the SIMPLE application. The "Orca-C" and "Pingali&Rogers" rows of this table include data reported from implementations of SIMPLE with two different parallelizing compilers [45, 55]. The row labeled "Prcnt. difference in Parallel Time" presents the average percentage difference between the parallel times reported by FAST and the parallel times measured on the Orca-C and the Pingali&Rogers implementations. The row labeled "Execution Time" reports the running time of FAST when simulating an instance of SIMPLE (4,096 grid-points), and the running time of an actual execution of the same instance on a single iPSC/2 node.

	Max. problem size examined	**Max. number of processors**	**Prcnt. difference in Parallel Time**	**Execution Time (Single processor)**
Orca-C	4,096 (grid-points)	32	6.14	9.8 secs
Pingali&Rogers	4,098 (grid-points)	32	35.74	14.02 secs
FAST	2^{20} (grid-points)	4,096	-	0.63 secs

Experiments with FAST generated detailed computation and communication profiles representative of important scientific algorithms running on realistic data-sets. These profiles provided us with:

- New insights in the performance characteristics of the algorithms studied. For example, the study of SIMPLE revealed the inherently sequential parts of the algorithm. We used data derived from FAST to perform a quantitative study on the effects that these sequential parts have on the scalability of SIMPLE. Furthermore, we studied the effects of computation-to-communication ratio on the speedup curve, as the number of processors increases and the problem size remains constant. Functional Algorithm Simulation of the Fast Multipole Method and the modified Barnes-Hut algorithm revealed the inherently sequential parts of their parallel execution. We concluded that these parts are due to the hierarchical nature of the algorithms. Profiles derived with FAST showed that the two algorithms have irregular computation and communication patterns.

- Practical and realistic means for comparing the effects of different interconnection topologies on parallel performance. Simulations with FAST showed that the Fast Multipole Method and the modified Barnes-Hut algorithm, mapped to ring-based topologies, perform almost as well as when mapped to hypercubes and cliques.

- Cost-effective ways of evaluating the influence of different hardware parameters on the performance of parallel executions.

- Performance comparisons among different methods for partitioning sequential algorithms into parallel tasks and mapping those tasks on multiprocessors with various processor numbers.

To assess the validity of Functional Algorithm Simulation, we ran FAST simulations on data-sets which were used in other studies. Comparison between our simulation results, published data, and theoretical analyses provided sound corroboration of our method.

1.5 Thesis Outline

In Chapter 2 we discuss the prototype system FAST, which we built to assess the Functional Algorithm Simulation method. We explain how algorithms and architectures are modeled in FAST and how the descriptions of the algorithms are transformed into task-flow graphs. A portion of our system addresses the mapping of these graphs onto processors of parallel machines. Part of this chapter has been presented in [17] and will appear in [19]. Chapter 3 describes the various heuristics implemented in FAST to perform this mapping. Chapter 4 discusses a problem that arises when trying to estimate message latency under blocking or synchronous message-passing interface primitives. This work was presented in [18]. In Chapter 5 we present the functional algorithm simulation of SIMPLE; results from this simulation are used to verify the validity of FAST, and have been presented in [17]. The study of the Fast Multipole Method is described in Chapter 6, and the study of the Barnes-Hut technique is presented in Chapter 7. Finally, Chapter 8 summarizes our major conclusions and discusses future work.

Chapter 2

A Functional Algorithm Simulation Testbed

To test the Functional Algorithm Simulation method we developed a prototype system called *FAST* (*Functional Algorithm Simulation Testbed*). Parallel executions are modeled by FAST as directed-acyclic *task-flow graphs*. These are a special case of the *data dependence graphs* (DDG's) that are frequently used as abstract representations of parallel programs [23, 56, 63]. For a given algorithm and input data-set, FAST generates a task-flow graph, which encapsulates the computations and communications that comprise the parallel execution of this algorithm on the data-set. The nodes of the graph correspond to groups of numerical operations or numerical routines carrying out calculations required by the algorithm examined. Edges represent the data movements between these groups of operations. FAST uses the task-flow graph to extract information used to characterize, evaluate, and understand the parallel computation at hand.

2.1 Structure of FAST

In order to perform a functional simulation of an algorithm, one has to provide FAST with:

1. A sequential implementation of the algorithm, modified so as to generate at run-time the sets of calculations and data-dependences comprising the computation, instead of executing the actual computation. In the current version of FAST, the transformations necessary to convert the sequential program are performed by hand.

2. The input data that would be provided to a sequential implementation of the algorithm. For example, for the Fast Multipole Method the input to FAST is a number of particles (bodies), their positions, and algorithmic parameters such as the granularity of the hierarchical decomposition of the particle-space.

3. The target parallel architecture, which is described by specifying a number of processors, selecting one out of a set of available interconnection topologies and message-passing interface paradigms (e.g. asynchronous, blocking), and specifying hardware characteristics such as processor speeds and communication bandwidth.

4. The strategies for partitioning the problem and mapping it onto the target architecture.

FAST generates a task-flow graph describing the parallel execution of the algorithm at hand on the chosen architecture for the specific input data. From this graph, it extracts useful information such as the parallel-time of the execution, speedups achieved for a large range of processor numbers, utilization of processors and communication links, computation-to-communication ratios, distribution of message sources and destinations, and effects of congestion and communication overhead on parallel-time. All this information may well correspond to executions of practical interest since the user has the option to provide FAST with input data from "real-life" problems.

FAST is split in two parts: a front-end and a back-end. The front-end generates the task-flow graph encapsulating operations and data exchanges performed according to the algorithmic description for a given setup of algorithmic parameters and input data. The back-end performs the mapping of the task-flow graph onto a parallel architecture and generates a modified task-flow graph representing the parallel

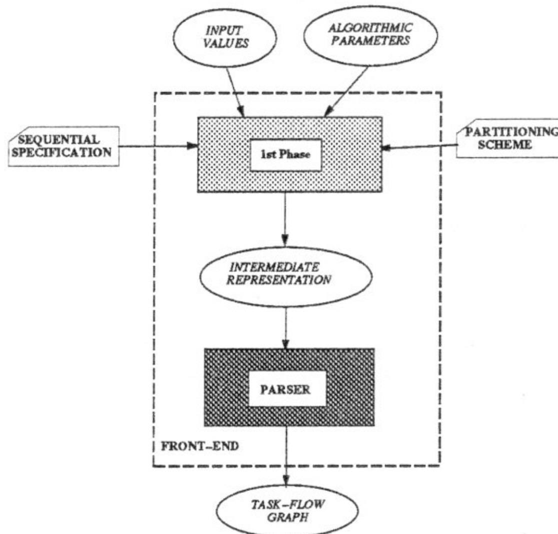

Figure 1: Front-end of the FAST.

execution on the chosen architecture.

2.2 Front-End

The task-flow graph generation is accomplished by the front-end in two phases (see Figure 1). The first one depends on the algorithm studied: given a sequential implementation, the user modifies it by inserting code that will produce the set of calculations and communications defining the corresponding parallel execution dynamically. The modified program is part of the first phase of FAST's front-end for the specific algorithm.

2.2.1 Intermediate Representation

Running the front-end on some appropriate input data-set produces the *computational domain* of the parallel execution at hand for the given input data. Simultaneously, it generates an architecture-independent *Intermediate Representation (IR)* of the parallel computation studied, representing the set of computations and communications that would be performed on the computational domain in a real implementation of

the algorithm. The Intermediate Representation is given as a group of *Intermediate Representation Routines* (*IRR*'s). Each IRR corresponds to a different domain-node of the computational domain and represents the computations and communications on the variables (components) of that node.

The Intermediate Representation Routines are expressed 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. Each of these groups represents a *basic computational block* of the algorithm. *Send/Receive* primitives correspond to data-dependences between IR-operations and represent the data-flow.

As an example we give a small piece of pseudo-code and present the steps taken to derive the code producing its Intermediate Representation. In this case, the computational domain is a two-dimensional grid with $siz_X \times siz_Y$ domain-nodes. The pseudo-code in Table 1 describes the calculation of variables at each point of the grid as a function of variables from this point and neighboring ones. In every domain-

```
           for j:=0 to siz_Y do
             for i:=0 to siz_X do
(I)              D[i,j] := R[i,j]+R[i,j-1]*(1-A[i,j])
(II)             A[i,j] := R[i,j]/D[i,j]
(III)            V[i,j] := R[i,j-1]*V[i,j-1]
             endfor
           endfor
```

Table 1: Pseudo-code.

node (i,j), the computation above requires the values `R[i,j-1]` and `V[i,j-1]` from the neighboring domain-node (i,j-1), as well as values of "local" variables, that is, `R[i,j]`, `A[i,j]`, and `D[i,j]`.

This pseudo-code can be readily parallelized with a parallel programming language like JADE. JADE follows the single-address space, sequential, and imperative programming paradigm [58]. Each piece of data allocated in the single shared memory of a JADE program is called a *shared object*; it is identified with the `shared` type

qualifier and can be accessed by all tasks. JADE uses the `withonly-do` construct to identify and spawn new tasks. This construct has the following syntactic form [59]:

> `withonly` {*access_declaration*} `do`
> (*parameters of task_body*)
> *task_body*

The *task_body* section contains the serial code executed when the task runs. The *parameters* section declares a list of shared objects from the enclosing environment that may be accessed by the task body. The *access_declaration* determines how the task body will access its parameters. JADE uses the `rd_wr` and `rd` primitives to specify access declarations: `rd_wr(var)` declares that variable `var` may be read and written by the corresponding task. Respectively, `rd(var)` declares that `var` may only be read, and that it will not be updated. We can describe the pseudo-code of Table 1 using JADE-constructs as follows:

```
typedef double shared *G_component;
toy_example(R, D, A, V, siz_X, siz_Y)
  G_component R, D, A, V;
  int siz_X, siz_Y, i, j;
{
for (i=0; i<siz_X; i++)
  for (j=0; j<siz_Y; j++) {
    withonly { /* access specification statements */
                 if (j-1 > 0) {
                        rd(R[i,j-1]);
                        rd(V[i,j-1]);
                 }
                 rd(R[i,j]);
                 rd_wr(D[i,j], A[i,j], V[i,j]);
    } do (R, D, A, V, i, j) { /* task_i,j */
         D[i,j] := R[i,j]+R[i,j-1]*(1-A[i,j]);
```

```
            A[i,j] := R[i,j]/D[i,j];
            V[i,j] := (R[i,j-1]*V[i,j-1];
         }
      }
   }
```

The task-granularity of the JADE program in this example does not necessarily represent the optimal way of parallelizing the pseudo-code of Table 1. Because of the overhead of task-creation and communication, it is conceivable that generating "coarser" tasks that compute the values of variables D, A, and V at blocks of the grid, rather than at single grid-points, will result in a more efficient implementation.

At run-time, this parallel program generates one task $task_{i,j}$ per domain-node (grid-point). $task_{i,j}$ performs the computations defined by the algorithm on the components (variables) of its domain-node. Furthermore, the access specifications determine the data-dependences between tasks $task_{i,j-1}$ and $task_{i,j}$. The former updates the values R[i,j-1] and V[i,j-1], and the latter uses those values in its calculations. The access specifications will be carried out through explicit messages between processors. For instance, $task_{i,j}$ will first receive the values R[i,j-1] and V[i,j-1] from $task_{i,j-1}$, then it will compute values D[i,j], A[i,j], and V[i,j], and finally it will send R[i,j] and the updated value of V[i,j] to $task_{i,j+1}$.

The computation that would be performed in an implementation of this example can be expressed abstractly as a set of routines representing the calculations on the grid-points. Each of these routines is expressed in an intermediate language format as follows:

```
Grid-point[i, j] :
   RECEIVE from [i,j-1]: R[i,j-1], 8 bytes
   RECEIVE from [i,j-1]: V[i,j-1], 8 bytes
   DAV_comp: DAV_COMP
   SEND to [i,j+1]: R[i,j+1], 8 bytes
   SEND to [i,j+1]: V[i,j+1], 8 bytes
```

DAV_comp denotes an IR-operation corresponding to the basic computational block of

instructions which compute `D[i,j]`, `A[i,j]`, and `V[i,j]`; that is, lines (I), (II), and (III) of Table 1. `DAV_COMP` is a constant representing the time spent for the calculation of values `D[i,j]`, `A[i,j]`, and `V[i,j]`. Such constants are expressed in terms of seconds, machine cycles, or instruction counts. In the current implementation of FAST we compute them manually by counting the corresponding machine instructions (e.g. three floating-point multiplications, one floating-point division, etc.). If necessary, the machine instruction counts are transformed into machine cycles or seconds by taking into consideration *cycles-per-instruction* figures from some specific hardware platform. The `SEND` and `RECEIVE` communication primitives implement the data-dependences defined by the algorithm at hand. Each such primitive, is assigned a number representing the size of the data it carries (in numbers of bytes). For instance, `R[i,j]` and `V[i,j]` are assumed to be 8-byte-long floating-point numbers and, therefore, `SEND`'s and `RECEIVE`'s carry 8 bytes of data.

The first part of FAST's front-end generates such an intermediate description of the algorithm examined for a certain input data-set. For instance, the following piece of C-code generates the Intermediate Representation for our example and can be used as the first part of a FAST simulator for this simple application.

```
for (i=0; i<siz_X; i++)
  for (j=0; j<siz_Y; j++) {
   printf("Grid-point[%d,%d]:\n", i, j);
   if (j-1 > 0) {
    printf("RECEIVE FROM [%d,%d]:R,%d words\n",i,j-1,siz_of(R[i,j-1]));
    printf("RECEIVE FROM [%d,%d]:V,%d words\n",i,j-1,siz_of(V[i,j-1]));
   }
   printf(\"DAV_comp: %lf microsec\n'', DAV_COMP);
   if (j+1 < siz_Y) {
    printf("SEND TO [%d,%d]:R,%d words\n",i,j+1,siz_of(R[i,j]));
    printf("SEND TO [%d,%d]:V,%d words\n",i,j+1,siz_of(V[i,j]));
   }
  }
```

The actual values that would be received, computed, and sent in a real implementation of this example are not of interest. Functional simulation focuses on extracting information about the *sets* of computations performed and about the sources, destinations, and sizes of messages exchanged; not about numerical values of the results.

Table 2 shows a typical subset of the Intermediate Representation as generated by FAST for the SIMPLE application. SIMPLE is a Computational Fluid Dynamics algorithm that computes the values of physical variables assigned to the points of a two-dimensional grid. Therefore, it describes calculations and message exchanges taking place at the points of such a grid.

Grid-point [0, 2]	
RECEIVE from [0, 0]:	8 bytes
RECEIVE from [0, 1]:	96 bytes
RECEIVE from [0, 3]:	96 bytes
RECEIVE from [1, 2]:	96 bytes
JcbR_comp:	257.96 μsec
DAV_comp:	445.04 μsec
ENPR_comp:	188.84 μsec
HEAT1_comp:	427.68 μsec
SEND to [0, 3]:	48 bytes

Table 2: Fraction of the Intermediate Representation for SIMPLE.

In a future version of FAST, instead of having the user manually writing the code that generates the Intermediate Representation, we could replace FAST's front-end with the front-end of a parallel programming language like JADE. The JADE compiler generates a task-flow graph according to the parallel program and the given input. This task-flow graph could then be fed to FAST's back-end. In that way the same parallel code could be used for functional simulation and actual execution.

2.2.2 Task-flow graphs

In the second phase of the front-end a simple parser transforms the Intermediate Representation into a *task-flow graph* G_{tf}, which follows the *Macro-Dataflow* model

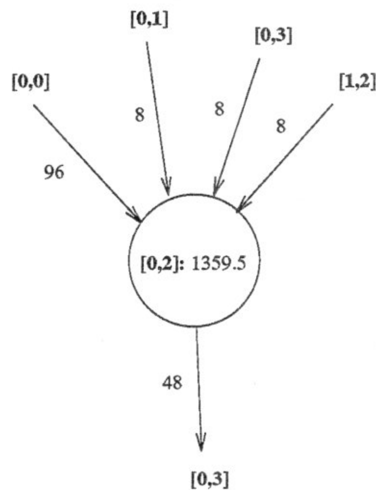

Figure 2: Subset of the task-flow graph.

of computation [62]. In this model, each task starts executing upon receipt of all incoming messages and continues to completion without interruption. Upon completion, it forwards its results to adjacent tasks. Formally, the task-flow graph is defined as a directed, acyclic, weighted graph $G_{tf} = G(V, E, \tau, W)$ where:

- V is the set of nodes of the G_{tf}, which represent indivisible sequential tasks. Every task v encloses a number of Intermediate Representation primitives that may be preceded by a number of *Receive* operations providing data from other tasks and may be followed by a number of *Send*'s which forward computation results to other tasks. The granularity of the task-flow graph follows naturally from the granularity of communication observed in the Intermediate Representation: the "boundaries" of the tasks are defined by *Send* and *Receive* primitives occurring in the IR.

- E is the set of edges, which correspond to *Send-Receive* pairs and represent the data dependencies between the nodes.

- $\tau(v)$, for $v \in V$ is the computation time of task v. It is equal to the sum of the constants corresponding to IR-operations assigned to node v. $\tau(v)$ is expressed in terms of seconds, or machine cycles, or instruction counts. If the code within task v depends only on input parameters and on the initial configuration of the

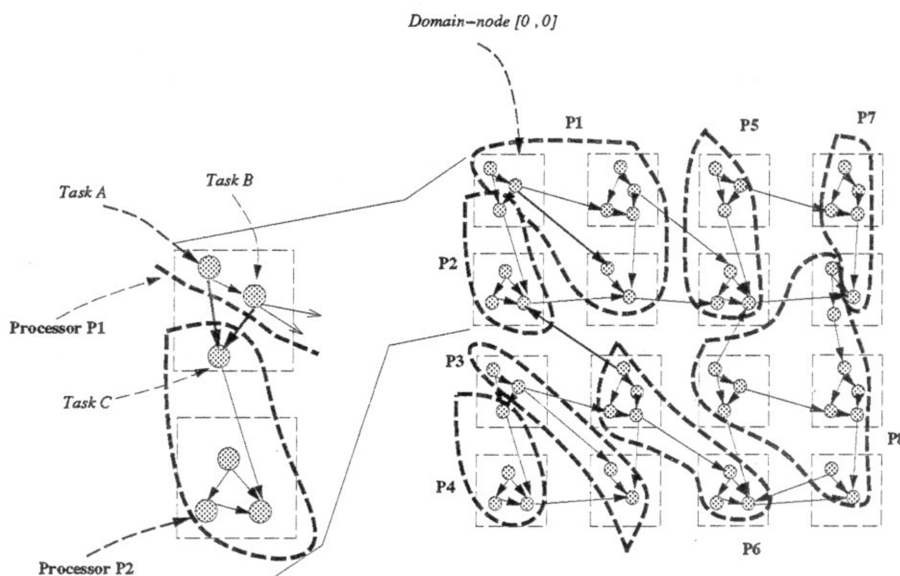

Figure 3: Implicit Partitioning.

data space, $\tau(v)$ can be estimated accurately from the algorithmic description, and from parameters of the processor architecture.

- W is the set of weights assigned to edges of G_{tf}. For every $e = (u, v) \in E$, $W(e)$ represents the number of bytes "carried" by edge e from its source to its destination.

For instance, the IR of Table 2 will be transformed into the portion of a task-flow graph shown in Figure 2.

2.2.3 Partitioning

For each Intermediate Representation Routine, the parser of FAST generates a set of tasks representing the processing performed on the variables (components) of the corresponding domain-node. These variables are considered to be "local" to those tasks. In Figure 3 (right), the computational domain is a two-dimensional grid containing 16 domain-nodes. For each domain-node, a number of tasks carries out the computations to be performed on the variables of the node. Clustering and mapping heuristics may be employed to partition the task-graph into a number of partitions

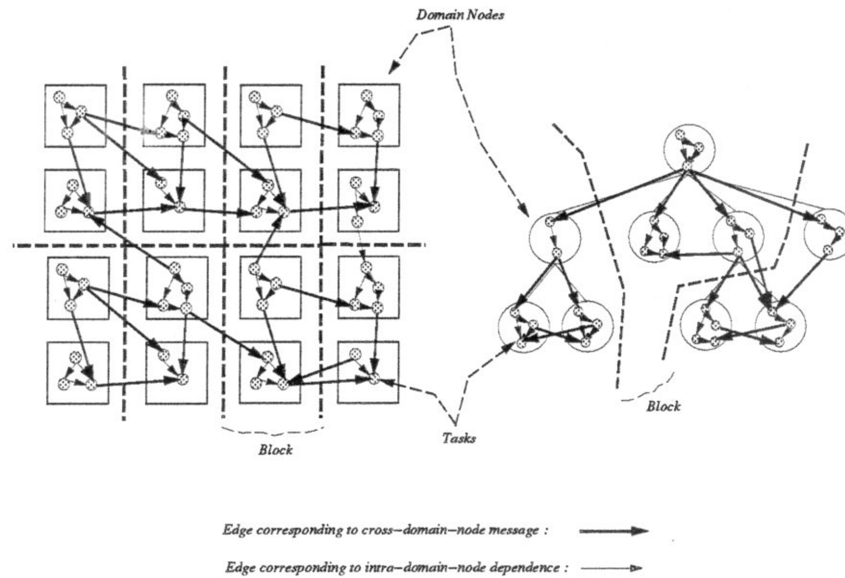

Figure 4: Partitioning.

equal to the number of available processors. Tasks belonging to each partition are merged into sequential threads of execution which are then assigned to the processors. We assume that tasks can access the variables of their domain-node through the local memory of the processor to which they will be assigned. If they use variables from other domain-nodes, the necessary values are brought into the local memory of the tasks by incoming messages from the appropriate tasks on the other domain-nodes. For example, in Figure 3, we show a partitioning of tasks into eight sets which are assigned to eight processors. For this partitioning, task C computes variables of domain-node $[0,0]$ and is mapped to processor $P2$. Variables of domain-node $[0,0]$ used by A, are considered to be available at the local memory of $P2$. Results from tasks A and B are carried to task C through messages represented by the edges (A,C) and (B,C).

The task-generation module in FAST does not make any further assumptions regarding the partitioning and placement of data. In practice, however, the parallelization of such problems oftenly starts with the "static" partitioning of data into blocks, and the assignment of blocks to processors. Subsequently, the tasks performing computations on each data-block are merged into sequential threads of execution

and mapped on the same processor.

To take into account the case where the computational domain is partitioned into blocks, the user of FAST is given the flexibility to have the system partition the domain into blocks, structured either according to a given geometry, or in compliance to some specified heuristic. For example, in Figure 4 (left), the computational domain is a two-dimensional grid. This is partitioned in blocks containing two domain-nodes each. In Figure 4 (right), the domain is a tree which is partitioned according to a heuristic assigning three tree-nodes per partition. FAST eventually merges tasks belonging to the same block of the computational domain into a sequential thread of execution. Each sequential thread is subsequently mapped onto a different processor.

2.3 Back-End

The back-end of FAST (see Figure 5) maps the task-flow graph G_{tf} onto an idealized architecture forming a fully-connected network (*"abundant" clique*). The supply of processors in this architecture is unlimited and in any given case is equal to the number of tasks G_{tf}. Clearly, the "abundant" clique does not represent a realistic choice for a parallel system, due to the unlimited number of available processors and communication links. However, it is a useful device for deriving architecture-independent characteristics of the algorithm, such as upper bounds on speedup, parallelism and communication profiles, and communication patterns.

2.3.1 Parallel Execution Model

The mapping to the "abundant" clique estimates the computation time of each task and the latency of each message, that is, send/receive overhead plus propagation delay plus congestion in the buffers of the network interfaces. The result of this mapping is expressed in terms of a new graph called the *parallel-execution graph.* Each node in the parallel-execution graph is assigned the computation time of the corresponding task and each edge is assigned the latency of the respective message. The parallel-execution graph is an abstraction of the parallel execution, which enables

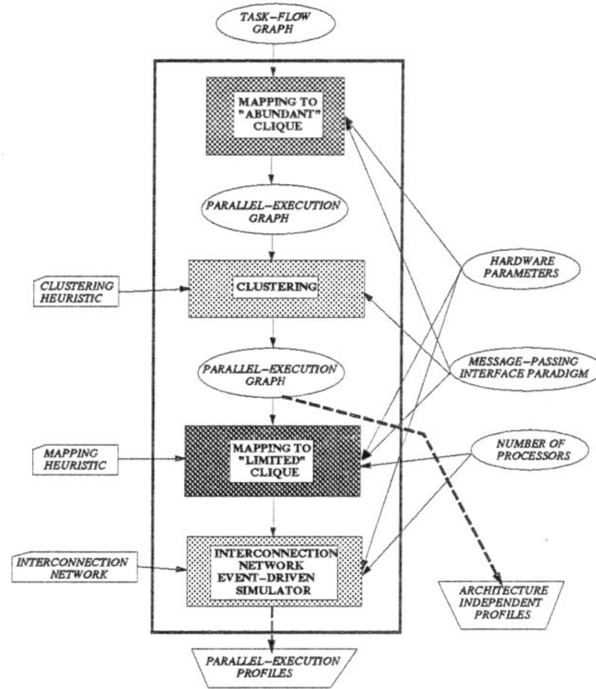

Figure 5: Back-end of the FAST.

us to estimate parallel time and available parallelism easily, and study the mapping of the parallel computation onto some realistic message-passing multiprocessor.

The parallel-execution graph is formally defined as follows:

$$G_{pe} \;=\; G(V,\; E_{pe},\; proc,\; T,\; D)$$

where:

1. V is the set of tasks.

2. $E_{pe} \;=\; E \cup E_{sch}$ is the set of edges. Edges in E correspond to explicit *Send/Receive* pairs. E_{sch} is a set of edges introduced in the graph to define the order of execution among tasks mapped on the same processor and with no program-determined dependences between them.

3. *proc*: A mapping from the set of task nodes V to the set of processors P: $\forall\; v \in V,\; proc(v)$ gives the processor in P that executes task v.

4. $T(v)$, $v \in V$ is the time it takes processor $proc(v)$ to perform v's computations.

5. $D(e)$, $e = (u, v) \in E$ is the weight assigned to edge e. $D(e)$ denotes the time-interval between the time that task u finishes its execution and the time that task v gains access to the data carried by edge e. If u and v are mapped onto different processors, $D(e)$ is equivalent to the interval between the time when $proc(u)$ has finished executing task u, and the time when message e has been loaded into the buffers of the destination processor's $proc(v)$ network interface. For a single-hop message, it is:

$$D(e) \;=\; t_{delay}(e) \;+\; S_{ov}(e) + W(e)/B + t_{congestion}(e) + R_{ov}(e)$$

where: $t_{delay}(e)$ is the delay between the time the sending processor issues the *Send* instruction initiating message e, and the time that this processor starts loading the message body to the buffers of its network interface. $S_{ov}(e)$ is the time it takes the sending processor to load its network interface's output buffers with the contents of message e and with control information (setup cost). $W(e)$ is the number of bytes carried by message e, B is the bandwidth of the communication links (in bytes per second), $t_{congestion}$(e) is the time e spends waiting in busy queues of the interconnection network, and $R_{ov}(e)$ is the time it takes a message to be loaded in the input buffers of the receiving processor's network interface. Additionally, we use $\delta(e)$ to denote the time it takes the message to propagate through the communication channels and then to be loaded into the input buffers of its destination's network interface. For one-hop messages this is equal to : $W(e)/B + t_{congestion}(e) + R_{ov}(e)$.

On the parallel-execution graph we can now define the *Parallel Time* as follows:

Definition 2.3.1 *The Parallel Time* (**PT**) *of a parallel-execution graph is defined as the weight of its critical path, i.e, of the path with the largest sum of node and edge weights. Let* $G_{pe} = G(V,\ E_{pe},\ proc,\ T,\ D)$ *be a parallel-execution graph, and* $\Pi = \{\pi_i \mid \pi_i = (v_1^i,\ v_2^i, \ldots, v_{m_i}^i)\}$ *be the set of paths in* G_{pe}, *where* $indegree(v_1^i) =$

$outdegree(v^i_{m_i}) = 0$. Let also:

$$time(\pi_i) = \sum_{j=1}^{m_i} T(v^i_j) + \sum_{k=1}^{m_i-1} D((v^i_j, v^i_{j+1}))$$

Then we define:

$$PT(G_{pe}) = \max_{\pi_i \in \Pi}\{ time(\pi_i) \}$$

□

As *Sequential Time* we define the time it takes to perform the computations described by the parallel-execution graph, when the tasks of the graph are mapped onto one processor. In other words:

$$ST(G_{pe}) = \sum_{i \in V} T(v_i)$$

The speedups reported in our work are computed as the ratio $ST(G_{pe})/PT(G_{pe})$.

2.3.2 Clustering and Mapping

In the absence of communication overhead, the optimal algorithm for assigning tasks to processors of an "abundant" clique is the straightforward one-to-one mapping. Nevertheless, in the realistic case where communication overhead cannot be ignored, the one-to-one mapping can be suboptimal. For instance, it is better to map long chains of tasks executing one after the other onto the same processor than to assign them to different processors. That way, we avoid the communication overhead involved in moving data from a task to its successor, and achieve better overall performance.

A tradeoff exists between parallelism and communication overhead for a given architecture (in this case the clique): a parallel execution that maps every node of the parallel-execution graph onto a different processor might not achieve minimum completion time because of communication delays and overhead. A sequential execution that maps all nodes of the task graph to the same processor, zeroes any communication overhead but neglects potential parallelism. Thus, it may also lead to suboptimal completion time. Between those two extremes there is a minimum completion time implementation that clusters together some tasks and maps them onto

the same processor. Such an implementation achieves the minimum communication overhead without sacrificing parallelism.

The problem of finding this optimal implementation is called *clustering* and has been proven to be NP-Complete [62]. In FAST, we implemented a number of clustering heuristics whose goal is the minimization of communication overhead without sacrificing parallelism or increasing parallel-time [20, 37, 63, 75]. The user can experiment with them while running the system and select the heuristic giving the best results. For the chosen clustering heuristic, the mapping of the clustered parallel-execution graph onto an "abundant" clique reveals all available parallelism of the application at hand, gives a lower-bound for the time span of the parallel execution, and can be used to derive an architecture-independent profile of the algorithm examined and study its scalability. Further details concerning the clustering heuristics and their effect on parallel performance can be found in the next chapter.

FAST subsequently maps the clustered parallel-execution graph onto a message-passing architecture. The number of processors is specified by the user of the system. This mapping problem is also NP-complete [51, 62]. To map the task-clusters onto processors we have implemented and incorporated in our system a number of heuristics similar to those included in the clustering stage [63].

The clustering and mapping stages can be bypassed if the user of FAST chooses to have the front-end partition the problem-instance at hand into a number of blocks equal to the number of available processors. In that case, task-nodes representing the operations performed on each block are clustered together into single threads of execution and mapped directly to the processors.

2.3.3 Parallel Architecture Model

A message-passing multiprocessor architecture is modeled in FAST as a set P of asynchronous, homogeneous nodes interconnected in some network topology IN. Each node is composed of two components: a processor that performs the actual computations, and a network interface that deals with sending messages to communication links, receiving messages that target the local processor, etc. (Figure 6). We adopt

the simplifying assumption that memory of the processing nodes is large enough so that the execution does not experience extra delays due to disk I/O. Moreover, we assume that buffer-space of the network interfaces is abundant, so that we will not have to worry about deadlocks and deadlock-avoidance protocols.

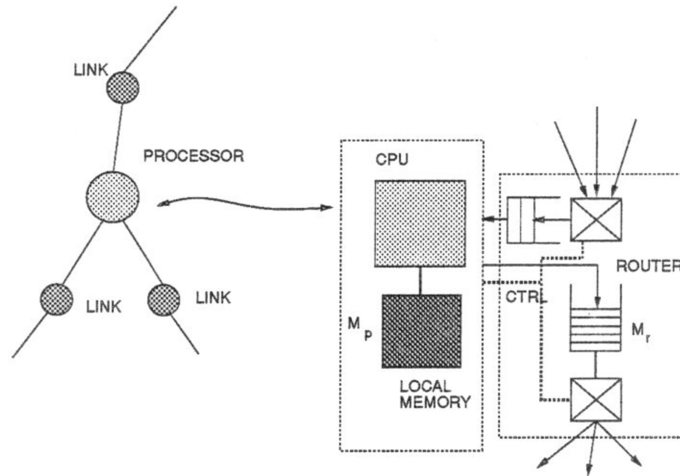

Figure 6: Processor-node structure.

Interconnection Network Topologies

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, a variety of multirings, and a binary hypercube (see Figure 7. The *limited clique* represents the fastest possible parallel implementation of the applications studied, for the number of available processors and the clustering and mapping heuristics adopted. With state-of-the-art technology and even for medium numbers of processors, the clique is neither a realistic nor a cost-effective interconnection scheme. But, it can be used to benchmark the efficiency of parallel executions on architectures with equal numbers of processors and sparser interconnection topologies. This approach is more accurate and more informative than the usual evaluation of efficiency as the ratio of measured speedup over the total number of processors.

The *single ring* interconnection simulated by FAST is bidirectional. Each direction has a functionality inspired by the SCI standard [64]. *Multiring* interconnections

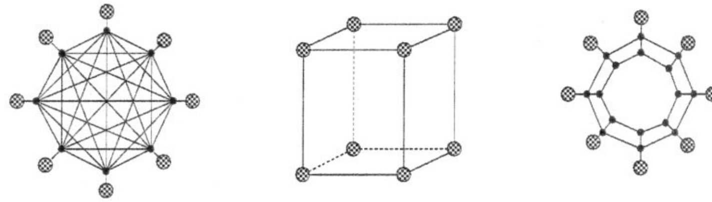

Figure 7: 8-processor clique, binary hypercube, and double-ring.

contain of a number of bidirectional rings functioning independently. For simplicity, we assume that a processor receives messages from all bidirectional rings and submits messages only to a specific bidirectional ring, defined by the processor number modulo the total number of single rings in the multiring network. Hardware parameters characterizing ring interconnections are similar to those chosen for the clique interconnection.

Message-Passing Interface Primitives

Point-to-point communications in parallel systems are implemented with *Send* and *Receive* primitives issued by parallel tasks. These primitives can be characterized as *blocking* or *non-blocking*, and as *synchronous* or *asynchronous*. Such characterizations determine the point in time when a communication primitive returns control to the task that called it. Also, they define the semantics of communication, and affect its performance.

More specifically, we encounter *blocking-asynchronous*, *blocking-synchronous*, and *non-blocking Send*'s. A blocking-asynchronous *Send* returns control to the calling process after the corresponding message has been loaded into the output buffer of the network interface at the *sending* processor. A blocking-synchronous *Send* returns after the message has been copied to the input buffer of the network interface at the *receiving* processor. A non-blocking *Send* returns before the corresponding message has been loaded into the outgoing buffer of the sending processor's network interface [14, 18, 53].

A blocking *Receive* returns after the corresponding message has arrived at the receiving processor, regardless of whether it has been dispatched from the sending

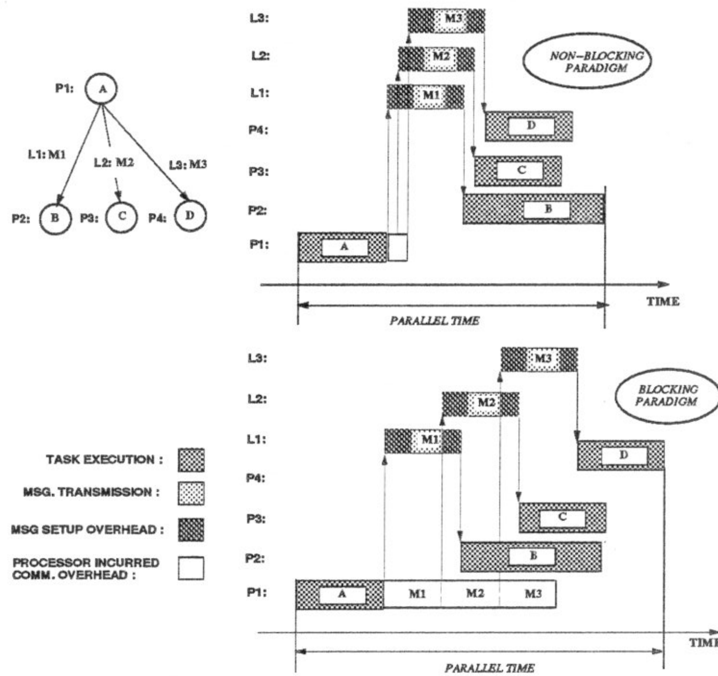

Figure 8: Blocking vs. Non-blocking *Send*'s.

processor with a synchronous or an asynchronous *Send*. A non-blocking *Receive* notifies the network interface that the receiving process is expecting a message, and then immediately returns control immediately to the calling process. When using a non-blocking *Receive*, a *Wait-for-receive* command has to be introduced at the point where the data of the incoming message are to be used by the program. *Wait-for-receive* is by definition a blocking primitive: a process that issues it, blocks until it receives the data required.

In the Macro-Dataflow model of computation the edges of the task-graphs represent pairs of *Send* and *Receive* primitives. *Send*'s can be either blocking or non-blocking, and synchronous or asynchronous. *Receive*'s must be blocking because of the definition of Macro-Dataflow. According to the non-blocking communication paradigm, messages are dispatched *simultaneously* at the end of the execution of a task. In contrast, according to the blocking paradigm, messages are transmitted *serially* from tasks with no overlap between the loading of buffers and the subsequent message-dispatches or computation (see Figure 8). FAST provides the user with the

choice of selecting a message-passing interface paradigm from the choices above.

Interconnection Network Simulators

We implemented an event-driven simulator to emulate message transmissions through binary hypercube and ring-based topologies. The simulator uses an event-queue and proceeds in synchrony with a routine searching the parallel-execution graph at hand in a breadth-first manner. When a node of the graph is marked as "visited" by the search routine, its outgoing messages become ready for transmission. These messages are denoted by the edges which start at the node and point to some node mapped at another processor.

Then, the simulator takes over and determines the order by which these messages are dispatched to the interconnection network, and the route they follow to reach their destinations. It updates the event-list by inserting events representing the corresponding message transmissions. These events are time-stamped appropriately to take into consideration the startup-time for a message dispatch (loading the outgoing buffers, testing control registers, etc.).

After the occurrence of an event representing a message-transmission, the simulator schedules a new event signaling the arrival of that message to the next node in its route. The time-stamp of the new event takes into consideration the propagation delay of the communication channel between the source and destination nodes, and the possible congestion delays due to contention in the channel or the corresponding network interfaces.

Upon occurrence of an event representing the arrival of a message to its destination, the simulator estimates the total latency of this message. Then, it uses this estimate to update the weight of the corresponding edge of the parallel-execution graph and triggers the breadth-first search routine to continue by checking the head-node of that edge.

Machine	S_{ov} (*in* μ*sec*)	R_{ov} (*in* μ*sec*)	Bandwidth
iPSC/860			(*in Bytes per* μ*sec*)
msg < 100bytes	37.25	37.25	2.3
msg > 100bytes	85.99	85.99	2.52
iPSC2			(*in 4Bytes per* μ*sec*)
msg < 100bytes	175	175	0.8
msg > 100bytes	330	330	1.44
CM-5			(*in Bytes per* μ*sec*)
	43	43	8.3

Table 3: Hardware parameters used in FAST.

Hardware Parameters

In addition to the number of processors and the network topology, the user provides FAST with a set of hardware parameters. This set is used to transform the weights assigned to nodes and edges of the task-flow graph into processing times and message latencies. It includes 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 [33, 53], and of Thinking Machine's CM - 5 [72]. These parameters were either collected from experiments we ran on an 8-node iPSC/860 installed at Princeton or they were extracted from published performance analyses [9, 12, 74]. Table 3 presents some of the hardware parameters used by FAST to perform functional simulations of algorithms on different interconnection networks. S_{ov} denotes the start-up cost for a message, that is, the time it takes the sending processor to load its network interface's output buffers with the contents of the message and with control information. Similarly, R_{ov} is the time it takes a message to be loaded in the input buffers of the receiving processor's network interface.

Chapter 3

Clustering and Mapping

In this chapter we discuss the clustering, scheduling, and mapping heuristics implemented in the current version of FAST. Using data derived from functional algorithm simulations of the Fast Multipole Method we perform comparison studies of these heuristics. Our main conclusion is that, for the realistic and practical scheme employed by FAST to assign weights to the edges of the task-graphs, the clustering heuristics examined do not achieve substantial improvements in terms of task-graph execution time. Furthermore, the existence of communication overhead does not affect significantly the mapping of clustered task-graphs to parallel system. Our experiments show that the combination of clustering and mapping heuristics give consistently better results than one-phase mapping algorithms.

For every functional algorithm simulation, the front-end of FAST generates a task-flow graph, which describes the parallel execution at hand in an architecture-independent way. To capture the characteristics of parallel workloads running on multiprocessors, we need to map their corresponding task-flow graphs onto appropriate models for the parallel machines.

In practice, parallelization of applications is performed in a sequence of steps: problem partitioning, task generation, assignment of tasks to processors, and scheduling of tasks into sequential threads executed at each processor. A variety of factors, such as the parallel architecture, the application program, the parallel programming platform, and the run-time system, affect the implementation of these steps. There

are many different approaches to parallelizing an application, and it is not realistic to emulate all of them with FAST. It is reasonable, however, to map the task-flow graphs to the models of the parallel architectures as efficiently as possible, and have FAST generate execution profiles and performance measurements corresponding to highly efficient implementations. This is the case in real implementations, which try to achieve optimality with respect to performance. Therefore, using inefficient mapping methods in FAST would produce unrealistic profiles that could result in misleading conclusions about the algorithms.

Task-flow graphs generated by FAST are a special case of the *data dependence graphs* (DDG's) that are used frequently as abstract representations of parallel programs [23, 56, 63]. The nodes of DDG's correspond to single program instructions or sets of instructions, depending on the desirable DDG-granularity. Their arcs correspond to dependences, which enforce a partial order of execution on program statements. DDG's of various granularities are constructed automatically as internal program representations in systems like *SISAL* [62], Parafrase-2 [56], and *JADE* [59]. Moreover, they can be defined and constructed explicitly as in the *Pyrros* system [76]. Finally, in systems following the *Fork-Join*, *SPMD* and *hypercube* programming models, DDG's are not explicitly created but still are a meaningful tool for describing parallel executions [15, 33].

A key issue that arises in systems employing data dependence graphs, is related to the execution of these graphs on the processors of a parallel computer [41, 43, 49], [3, 8, 46]. There are many approaches for addressing this problem, most of which can be classified as *static* or *dynamic*. Static schemes apply in systems where the DDG's can be constructed before program execution. In that case, the user-program or the compiler can take advantage of information pertinent to the DDG for making decisions that will guide the assignment of graph-nodes to different processors, and the scheduling of tasks within each processor [4, 76]. It is not always possible, however, to create the DDG's before the program execution. In that case, execution of DDG's is accomplished with dynamic schemes that are enforced through the operating system or the hardware. These schemes are more common in practice [44, 59].

The availability of global information for the DDG's before the parallel execution

enables static schemes to perform optimizations such as reducing the communication overhead, eliminating the task scheduling overhead, and achieving load balancing [63]. These optimizations make the compilation of systems employing static schemes much slower and more complex. In dynamic approaches on the other hand, global information is not available and, consequently, the systems rely on local information about the program. Processor assignment and scheduling decisions are made at run-time. Complex criteria for such decisions would slow down execution time significantly; therefore, dynamic schemes rely on simple decision criteria that try to emulate the decisions that would be taken by the more complex static techniques, if the DDG's were known in advance.

We employ static schemes developed for executing DDG's on parallel architectures, in order to map the task-flow graphs generated by FAST to parallel architecture models. Usually, these methods take place in two phases [23, 63, 62]:

1. The *clustering* or *internalization* phase, seeks to minimize communication overhead and improve parallel time by deciding that certain tasks must go together on the same processor, even if other processors are available.

2. The *mapping* or *processor assignment* phase, maps the groups of tasks formed by the clustering phase to the processors of the parallel architecture at hand. At the same time, it seeks to preserve a low parallel time.

Concurrently with clustering and mapping, it is necessary to perform *scheduling* of tasks assigned to the same clusters or processors.

In this chapter, we discuss the heuristics we implemented in FAST to perform clustering, mapping, and scheduling of the task-flow graphs produced by the first phase of the system. Also, we present a comparison among the different heuristics with respect to their performance and cost.

3.1 Clustering

Clustering specifies the sequential units of computation in a parallel program by mapping tasks to clusters. A cluster is a set of tasks that execute sequentially on the

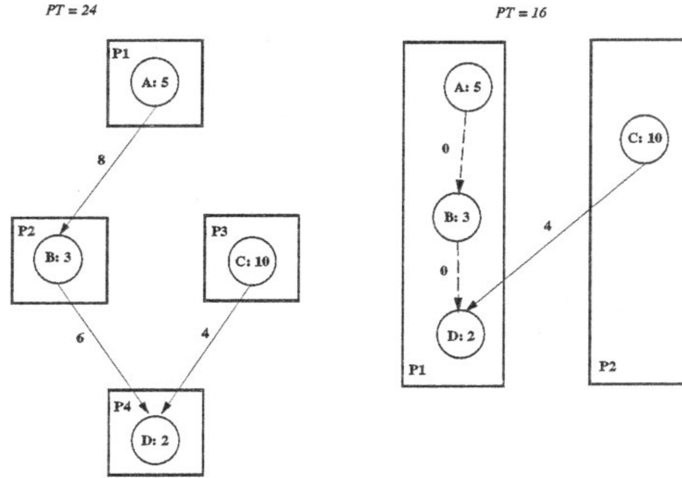

Figure 9: Different clustering choices.

same processor. The principle goal of clustering is to achieve the minimum parallel time for a given task graph on a clique architecture, with as many processors as tasks ("abundant" clique). Sarkar explained the reasoning of clustering as follows [62]:

> If two nodes are assigned to the same processor in this best-case situation with an unbounded number of processors available and the lowest possible overhead, then they should be assigned to the same processor in any schedule on the target architecture.

If communication overhead was zero, the trivial solution to clustering would assign each task to a different processor of an "abundant" clique. In the realistic case, however, a parallel execution that assigns every node of a task-graph on a different processor of an "abundant" clique might not achieve minimum completion time because of communication delays and overhead. For example, in Figure 9 (left), each task is assigned to a different processor and the parallel time of the execution graph is 24. In contrast, in Figure 9 (right), nodes A, B, and D are clustered to the same processor and the parallel time is only 16. This is because clustering eliminates the communication cost of messages (A, B) and (B, D).

For a formal definition of clustering, we consider a directed acyclic weighted parallel-execution graph $G_{pe} = G(V, E, proc, T, D)$ (see definition in Chapter

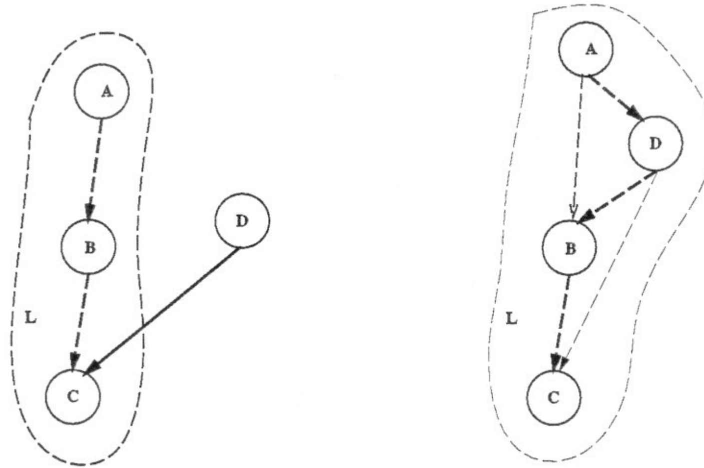

Figure 10: Scheduling edges. The bold arrows denote the sequential paths of execution in the clusters.

2). *Clustering* is the problem of partitioning the nodes of G_{pe} into clusters, and deriving the clustered parallel-execution graph with the *shortest* parallel time among all possible clustered graphs G'^{c}_{pe} mapped on "abundant" cliques. It has been proven that finding the optimal clustering of a directed acyclic graph that follows the Macro-Dataflow model of computation is NP-hard in the strong sense, if the cost function is the minimization of parallel time of the graph on an "abundant" clique architecture [62]. A number of heuristics have been developed to cope with the clustering problem [20, 37, 62, 75].

Clustering heuristics applied to G_{pe} will update its *proc* information to reflect the formation of clusters. If, for instance, nodes u and v clustered into the same cluster L, then $proc(u) = proc(v) = L$. Furthermore, clustering alters E, the set of edges of G_{pe}, by introducing new "scheduling" edges that express the scheduling priorities among nodes belonging to the same cluster. For example, in Figure 10, cluster L merges with node D. If task D is scheduled to run after task A and before task B, the edges (A, D) and (D, B) are inserted in the clustered graph to determine the new schedule.

Finally, clustering heuristics change the weights assigned to the edges of G_{pe}. For example, we consider a node $u \in V$ that sends $n+1$ messages to nodes $w_1, w_2, \ldots, w_k$,

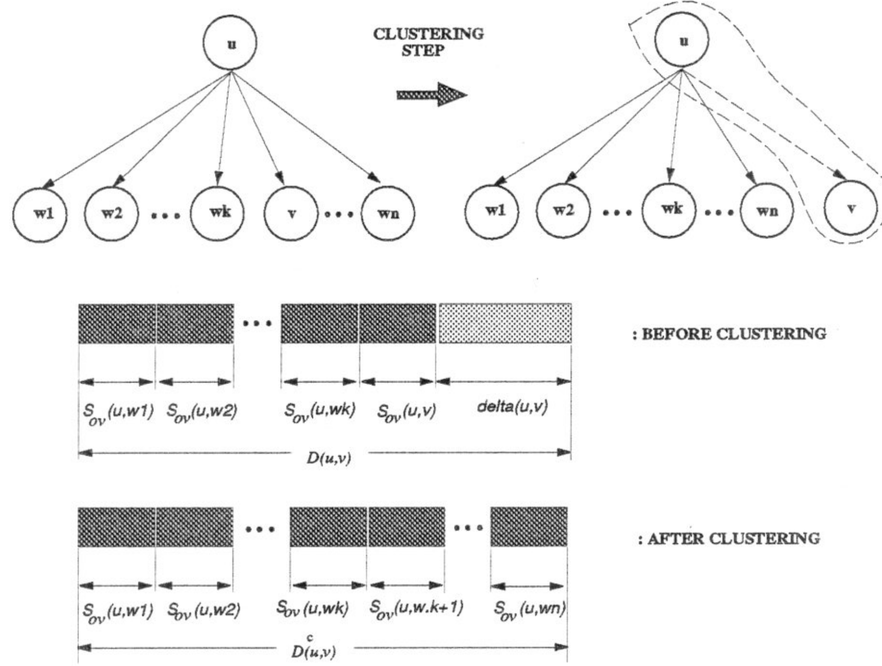

Figure 11: Edge weights (blocking *Send*'s).

v, $w_{k+1}, \ldots, w_n$, in that order. We assume that $proc(w_i) \neq proc(w_j) \neq proc(u) \neq proc(v)$, $\forall i \neq j$. If the clustering heuristic assigns nodes u and v to the same cluster, the weights of the outgoing edges $(u, w_1), \ldots, (u, w_k)$ of u will remain the same. The weight of (u, v) will be changed from:

$$D(u,v) \;=\; (\sum_{i=1}^{k} S_{ov}(u,w_i)) \;+\; S_{ov}(u,v) \;+\; \delta(u,v).$$

to:

$$D^c(u,v) \;=\; \sum_{i=1}^{n} S_{ov}(u,w_i)$$

(see Figure 11). The weights of edges $(u, w_{k+1}), \ldots, (u, w_n)$ will be reduced to

$$D^c(u,w_i) \;=\; D(u,w_i) \;-\; S_{ov}(u,v), \quad i = k+1, \ldots, n.$$

These formulas correspond to the case where the message-passing interface of the "abundant" clique provides a *blocking Send* communication primitive [14, 18].

The application of a clustering heuristic on a parallel-execution graph G_{pe}, transforms G_{pe} into a clustered parallel execution graph $G^c_{pe} = G(V_c, E_c, proc_c, T_c, D_c)$ where:

- V_c is the set of tasks in G^c; V^c is identical with V.
- $E_c = E \cup E_{sch}$, where E is the set of "scheduling" edges in G_{pe}, representing messages carrying information between tasks. E_{sch} is a set of edges introduced to the graph to define the order of execution among tasks assigned to the same cluster and with no program-determined dependences between them.
- $proc_c(v)$, $\forall v \in V$, is the number of the cluster that v belongs in.
- $T_c(v)$, $\forall v \in V$, is the processing time of task v.
- $D_c(e)$, $\forall e \in E$, is the weight assigned to edge e. For $e = (u, v)$, $D_c(e)$ represents the time between the completion of task u to the beginning of task v. If $proc(u) \neq proc(v)$, then $D_c(e)$ is equal to the latency of message e going from processor $proc(u)$ to processor $proc(v)$ of the "abundant" clique. Otherwise, $D_c(e)$ is equal to the time that processor $proc(u)$ of the "abundant" clique is busy transmitting messages to other processors, and therefore cannot continue with the execution of v (communication overhead).

3.2 Clustering Heuristics

The clustering heuristics implemented in FAST perform a number of refinement steps on the input parallel-execution graph. Each step performs a refinement on the output of the previous clustering step by merging two clusters, and scheduling their tasks within the newly formed clusters. In the initial parallel-execution graph, each task-node is a cluster by itself. The heuristics complete and report a final clustering when an end-condition is satisfied.

In FAST we implemented *edge-zeroing* heuristics with *no backtracking*. These algorithms proceed by merging *connected* nodes of the parallel-execution graph. Assigning

two connected nodes to the same cluster, eliminates the message that corresponds to the edge connecting them. After clustering, the message will be carried out through local memory *Write*'s and *Read*'s at the memory of the processor that executes the cluster. There is no backtracking, that is, once a cluster has been formed at one step of the heuristic, it cannot be split at a later step.

Various algorithms belonging to this class of clustering methods can be characterized with respect to:

1. The method for choosing which edge to eliminate.

2. The end-condition of the heuristic.

3. The scheduling heuristic employed when merging two clusters into a sequential thread of execution.

The choice of scheduling heuristic is orthogonal to the method for zeroing edges and to the end-condition. We present a number of clustering heuristics implemented in FAST first, and in the next section we talk about scheduling. A comprehensive discussion of clustering algorithms can be found in [24].

3.2.1 Sarkar's Clustering Algorithm

Sarkar's heuristic clusters a parallel-execution graph in a number of steps described below [62]:

1. Sort the edges $e \in E$ of the graph in descending order of their weights $D(e)$.

2. Merge the two clusters that include the head and tail node of the edge with the highest weight, if this change does not increase parallel time.

3. Repeat step 2 until all edges are scanned.

It is not difficult to see that the complexity of Sarkar's heuristic is $O(|E| \cdot (|V| + |E|))$. This results in very high execution times for large graphs. Therefore, we implemented also a variation of Sarkar's method that sorts the edges in descending order of their weights and examines only a percentage of them, starting from the one with the largest weight.

3.2.2 Kim and Browne's Algorithm

Kim and Browne's method takes a different approach to clustering [37]:

1. Mark all edges in the parallel-execution graph as *unexamined.*

2. Find the *critical path* in the graph composed of unexamined edges only. This is the path with the longest cumulative weight in the graph. In FAST, the cumulative weight of a path $(u_1, u_2), (u_2, u_3), \ldots, (u_{n-1}, u_n)$ is equal to $\sum_{i=1}^{n-1}\Big(T(u_i) + D(u_i, u_{i+1})\Big) + T(u_n)$.

3. Merge in the same cluster the nodes belonging to the critical path and mark all edges incident to nodes of the critical path as *examined.*

4. Apply steps 2 and 3 to the subgraphs formed by nodes and unexamined edges, until all edges are examined.

The complexity of Kim and Browne's heuristic is $O(|V| \cdot (|V| + |E|))$, since there are at most $|V|$ connected components in a graph and it takes $O(|V| + |E|)$ time to find the critical path in each component.

3.2.3 Linear-Greedy Algorithm

Kim and Browne's heuristic improves parallel time in the case where a simple scheme is used to assign weights to edges, and elimination of an edge results in zeroing its weight. Under the more realistic scheme employed in FAST, however, Kim and Browne's heuristic may result in clustered graphs with larger parallel times than the unclustered ones. With this consideration in mind, we modified this heuristic and introduced a version that we call *Greedy-Linear.* This algorithm is called "linear" because, as in Kim and Browne's method, it outputs clusters that are linear chains of task-nodes. The heuristic works as follows:

1. Mark all edges in the parallel-execution graph as *unexamined.*

2. Find the *critical path* in the graph composed of unexamined edges only.

3. For every edge of the critical path, cluster its adjacent nodes only if this does not result in a larger parallel time. Mark all the edges incident to nodes of the critical path as *examined*.

4. Apply steps 2 and 3 to the subgraphs formed by nodes and unexamined edges, until all edges are examined.

Testing whether the clustering of an edge results in a larger parallel time can be accomplished in constant time, without having to recompute the parallel time of the graph. Therefore, the complexity of this algorithm is $O(|V| \cdot (|V| + |E|))$ as well.

3.2.4 Greedy Dominant Sequence Algorithm

The clustering algorithm by Yang and Gerasoulis [75], identifies at every step the critical path of the graph, named *Dominant Sequence* (*DS*). The heuristic chooses one edge belonging to the DS and merges the clusters of its adjacent nodes, if this decision leads to a shorter parallel time. After the clustering, the algorithm computes the new DS. The complexity of Yang and Gerasoulis' heuristic is $O((|E|+|V|)\cdot log|V|)$. In FAST, we implemented a simpler, greedy version of this algorithm, which we call *Greedy Dominant Sequence* (*GDS*) algorithm:

1. Identify the Dominant Sequence of the graph.

2. Choose the edge of the Dominant Sequence whose elimination results in the largest decrease of parallel time. Merge the clusters of the nodes adjacent to the selected edge.

3. Repeat Steps 1 and 2 until there is no edge in the DS whose elimination can decrease parallel time.

Identifying the Dominant Sequence requires a depth-first search of the graph which takes $O(|E| + |V|)$ time. Choosing which edge of the Dominant Sequence to eliminate takes time proportional to the number of edges in the Dominant Sequence, that is, $O(|V|)$. The algorithm will perform $O(|V|)$ clusterings and, therefore, the total complexity of the Greedy Dominant Sequence is $O(|V| \cdot (|E| + |V|))$.

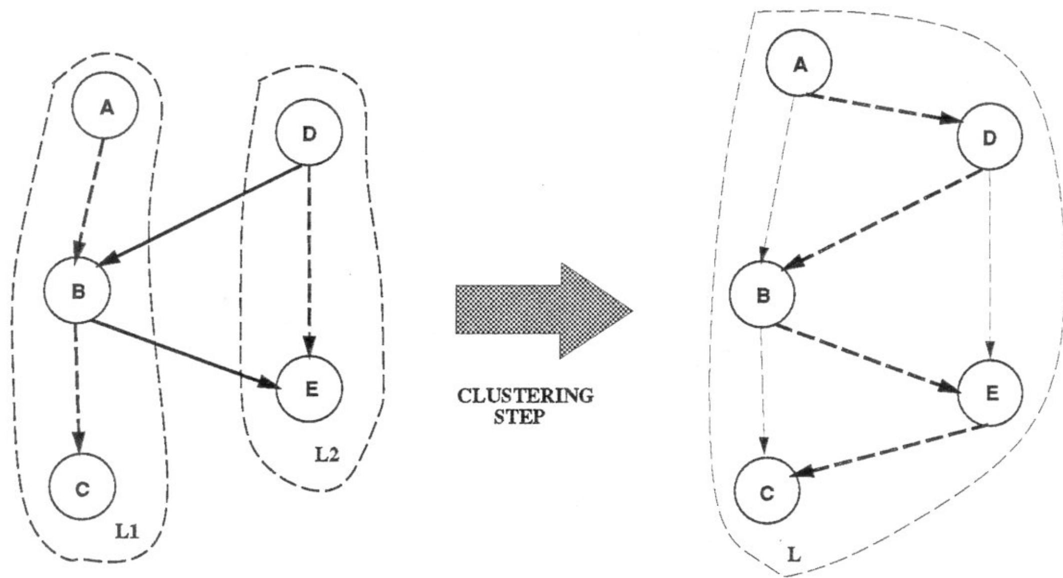

Figure 12: Scheduling problem: the bold edges represent the sequential thread of execution in each cluster.

3.3 Scheduling

The scheduling problem arises during the merging of two clusters, when their tasks have to be ordered according to some sequential order of execution. The example given in Figure 12, depicts the merging of clusters $L1$ and $L2$. In the resulting cluster L, tasks A, B, C, D, and E will constitute a sequential thread of execution. The scheduling algorithm should specify an ordering of tasks that achieves the shortest parallel time and, at the same time, complies to existing precedence constraints. For instance, if the decision made by the scheduling algorithm is to execute task D after task A, and task E before task C, we have to introduce scheduling edges (A, D) and (E, C) in the graph. These edges denote the priorities imposed by the scheduling algorithm. For general directed-acyclic parallel-execution graphs, the problem of finding the optimal task sequences that minimize overall parallel time is NP-Complete [63]. Consequently, we implemented in FAST two simple heuristics to perform scheduling.

These heuristics are variations of *Priority List Scheduling* [11]. According to Priority List Scheduling, each task is assigned a priority. Whenever a processor is available, a task with the highest priority is selected from the list of tasks and assigned to that

processor. The different schedulers assign priorities to nodes in different ways. Adam et al. showed that among all priority schedulers, level-priority schedulers are the best at getting close to the optimal schedule [1]. These schedulers use the level of each node to determine its priority.

Before presenting the scheduling heuristics, we introduce some useful notation. Given a directed-acyclic graph $G = G(V, E)$, we denote by V_i the set of "input" nodes, that is, nodes in V with no incoming edges. With V_o, we denote the set of "exit" nodes, that is, nodes in V with no outgoing edges. Also, $\forall u \in V$, $\Pi(u, V_o)$ represents the set of all possible paths in G from node u to some node in V_o. Given this notation, we define the *ptime* value of a node in the graph as follows:

$$ptime(u) \; = \; \max_{\pi \in \Pi(u, V_o)} \left(T(u) + D(u, w_1) + \sum_{i=1}^{n-1} (T(w_i) + D(w_i, w_{i+1})) + T(w_n) \right)$$

$\forall u \in V$, and for $\pi = (u, w_1), \ldots, (w_{n-1}, w_n)$. In other words, the *ptime* value of a node is the total weight of the longest path from this node to the "exit" nodes of the graph. Furthermore, we define the *stime* value of a node as follows:

$$stime(u) \; = \; \max_{\pi \in \Pi(V_i, u)} \left(\sum_{i=1}^{n-1} (T(w_i) + D(w_i, w_{i+1})) + T(w_n) + D(w_n, u) \right)$$

$\forall u \in V$, and for $\pi = (w_1, w_2), \ldots, (w_n, u)$. Finally, the *level* of a node in the graph is defined as follows:

$$level(u) \; = \; \max_{\pi \in \Pi(V_i, u)} ||\pi||,$$

where $||\pi||$ represents the number of edges in path π, that is, the length of π.

3.3.1 TS/ETF Scheduling

Topological Sort/Earliest Task First (*TS/ETF*) scheduling [49] performs a topological sort [65] of the parallel-execution graph and assigns *level* values to its nodes. If node u precedes node v in the topological-sort order, that is, $level(u) < level(v)$, then u will be assigned a higher priority than v. If, however, *level(u)* equals *level(v)*, then *TS/ETF* assigns a higher priority to the node with the smaller *stime* value. The relative priorities of nodes with equal *level* and *stime* values are assigned by *TS/ETF* randomly.

3.3.2 CP/MISF Scheduling

Critical Path/Most Immediate Successors First (*CP/MISF*) scheduling [35] computes the *ptime* values of the nodes in the task-graph. Then, it constructs a priority list of nodes according to the descending order of their *ptime* values. For nodes that have the same *ptime* value, *CP/MISF* assigns a higher priority to those with the larger number of immediate successors; that is, with the larger number of outgoing edges. In summary, *CP/MISF* sorts task-nodes according to the descending lexicographic order of the tuple:

$$(ptime(v), outgoing_edges(v)), \quad v \in V$$

The order that each node has in the resulting sorted list, represents its scheduling priority. FAST uses these scheduling priorities to merge different clusters and form sequential threads of execution.

3.4 Mapping

Clustering produces a clustered parallel-execution graph with a number of clusters usually much larger than the number of available processors of the target architecture. *Optimal Mapping* is the problem of finding an assignment of clusters to processors, leading to a parallel time shorter than the times derived by all other assignments, for the given number of processors [52]. The Optimal Mapping problem of a clustered directed acyclic graph has been proven to be NP-Complete [62]. Consequently, in order to perform mappings of the clustered parallel-execution graphs to the processors of the parallel architectures studied with FAST, we implemented two heuristics representative of those reported in the literature. The mapping algorithms are based on clustering and scheduling heuristics, and are quite similar to the methods presented in the previous sections.

3.4.1 Sarkar's Algorithm

Sarkar's heuristic is a modified version of the Priority List Scheduling algorithm [63]. It uses a list, *pblock*, of size P, where P is the number of available processors. *pblock*

entries are initially empty. When the algorithm completes, $pblock[i]$ contains the tasks assigned to processor i, for $i = 1,\ldots,P$. The algorithm creates a priority list of task-nodes, according to a topological-sort ordering of the graph. Then, at each step, it processes the next node T in the priority list. If T has not already been assigned to a processor, the algorithm performs the following tasks:

1. Choose a processor i, such that, the merging of clusters $proc_c(T)$ and $pblock[i]$ will result in a parallel time shorter than the one derived from the merging of $proc(T)$ with any other cluster $pblock[j]$.
2. Merge clusters $proc_c(T)$ and $pblock[i]$, and assign the result to $pblock[i]$.
3. Assign all task-nodes of cluster $proc[T]$ to processor i.
4. Reduce the number of clusters by one.

The algorithm completes when the total number of clusters in the graph becomes equal to P. It is not difficult to see that its computational complexity is $O(P \cdot |proc| \cdot (|V| + |E|))$, where $|proc|$ is the initial number of clusters.

Sarkar's mapping algorithm is slow because of the large constants involved in its complexity. We implemented a modified version to improve its running time, although without achieving a better asymptotic complexity. This version follows exactly the same steps as the original heuristic. It does not take, however, into consideration communication costs when calculating parallel time. We call it *SNC*, that is, *Sarkar's algorithm with No Communication Costs.*

We also implemented a version of the *Priority List Scheduling* heuristic, which applies directly to *non-clustered* graphs. This heuristic orders nodes of the graph according to the *TS/ETF* or the *CP/MISF* principle. Then, it traverses the priority list of nodes, and maps each task to the processor that will start executing it the earliest possible.

3.4.2 Yang and Gerasoulis' Algorithm

In [76], Yang and Gerasoulis introduced a fast heuristic for mapping a clustered graph to the processors of a parallel system. This algorithm seeks to optimize the

load-balancing of the available processors. It is comprised of four steps:

1. Estimate the average processing time, A, of the processors, as the sum of the processing times of all clusters, over the number P of processors.
2. Sort the clusters in an increasing order of their loads.
3. Assign each cluster with a processing time higher than the average A to a different processor.
4. Use a wrap mapping for the remaining clusters: number these clusters from 1 to their total number. Then, assign each of them on the processor whose number is equal to the number of the cluster modulo P.

The complexity of this method is $O(|V| \cdot \log |V| + |E|)$.

3.5 Experimental Results

In this section we present representative experimental results using the clustering, scheduling, and mapping heuristics employed in FAST. We give data derived when using the these heuristics on a parallel-execution graph with 1445 task-nodes and $12,860$ edges. This parallel-execution graph represents the parallel computation of the Fast Multipole Method (see Chapter 6) on 1000 bodies, with quadtree granularity of 5 bodies per leaf. Similar conclusions can be derived when using parallel-execution graphs representative of other problems. For the evaluation of clustering algorithms we used the *CP/MISF* scheduling heuristic. The results do not change considerably when using *TS/ETF*.

3.5.1 Clustering

Figure 13 shows the ratio of the parallel time of the clustered parallel-execution graph of our example, over the parallel time of the unclustered graph, for a number of different clustering techniques and for two message-passing interface paradigms (blocking or non-blocking *Send*'s): Sarkar's method; Greedy-Linear algorithm (*GL*);

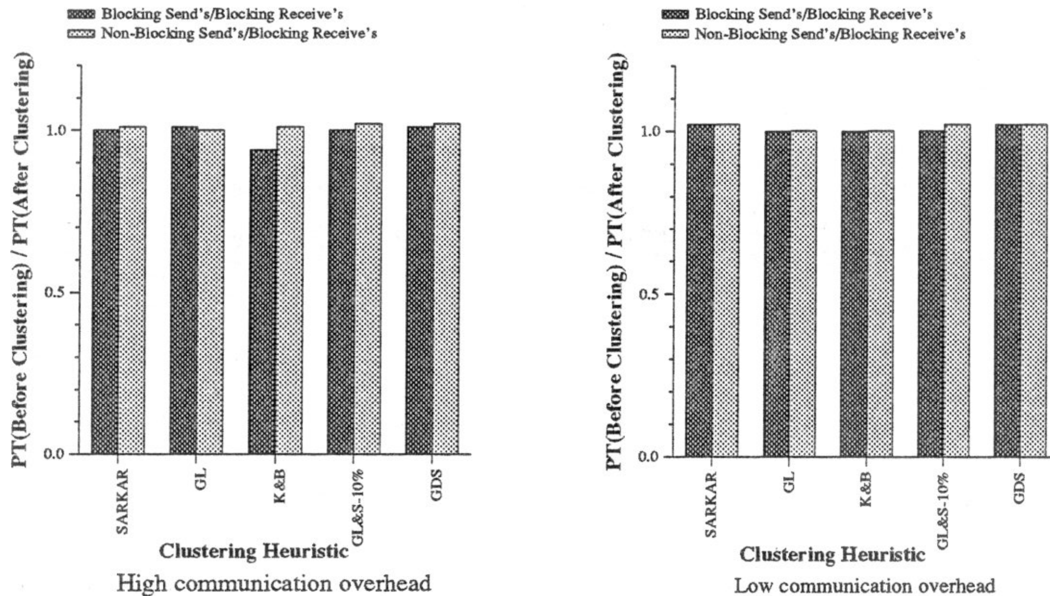

Figure 13: Effects of clustering to parallel time.

Kim and Browne's method (*K&B*); running Greedy-Linear on the graph and then applying Sarkar's heuristic for only the 10% of the edges (*GL&S-10%*), and Greedy Dominant Sequence approach. Clearly, the clustering heuristics do not improve the parallel time of the clustered graph significantly with respect to the parallel time of the unclustered graph. This remark is true regardless of the message-passing interface paradigm adopted (blocking or non-blocking *Send*'s), and for both high setup-cost as well as for low setup-cost communications. In the high setup-cost case, we used measurements from the $iPSC/860$, whereas in the low setup-cost case, we used overhead values ten times smaller.

The plot in Figure 14 shows the numbers of the clusters produced by the different clustering heuristics for the two message-passing interface paradigms. This is an interesting metric, since the performance of mapping algorithms depends on the number of clusters generated by the clustering heuristics which precede mapping. Clearly, mapping is faster for clustered graphs with fewer clusters. As expected, applying Sarkar's heuristic results in the smallest number of tasks. The reason is that the algorithm considers all edges in the graph for "zeroing." In contrast, the Greedy Dominant Sequence method results in a number of clusters almost identical to the initial number

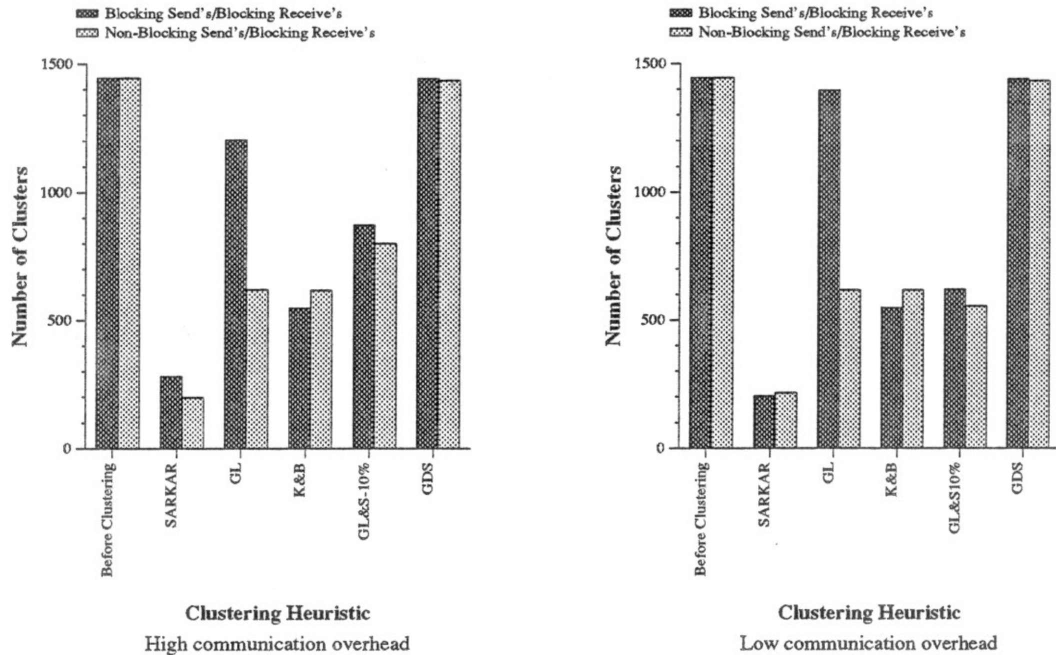

Figure 14: Number of clusters.

of tasks. *GDS* eliminates only edges belonging to the Dominant Sequence (that is, the critical path of the parallel-execution graph) and, thus, clusters few of the tasks belonging to the DS. Clustering these tasks, however, does not necessarily alter the DS. Hence, the algorithm can complete without further clustering.

The Greedy-Linear (*GL*) and Kim&Browne's heuristics do not check the Dominant Sequence only. Instead, after performing clustering on the DS, they proceed by clustering tasks belonging to the critical paths of the subgraphs formed when deleting edges adjacent to the initial DS. The *GL* method results in a small number of clusters in the case of non-blocking *Send*'s, and in twice as many clusters in the case of blocking *Send*'s. This is due to the fact that "eliminating" an edge on the critical path of a parallel-execution graph, will always result in a smaller cumulative weight for this path, if the message-passing interface paradigm provides for *non-blocking Send's.* This is not always the case with *blocking Send's* and, thus, there are fewer opportunities for the clustering heuristic to perform effective clusterings. For example, Figure 15 shows an unclustered parallel-execution graph where the critical path is the edge (A, B), and

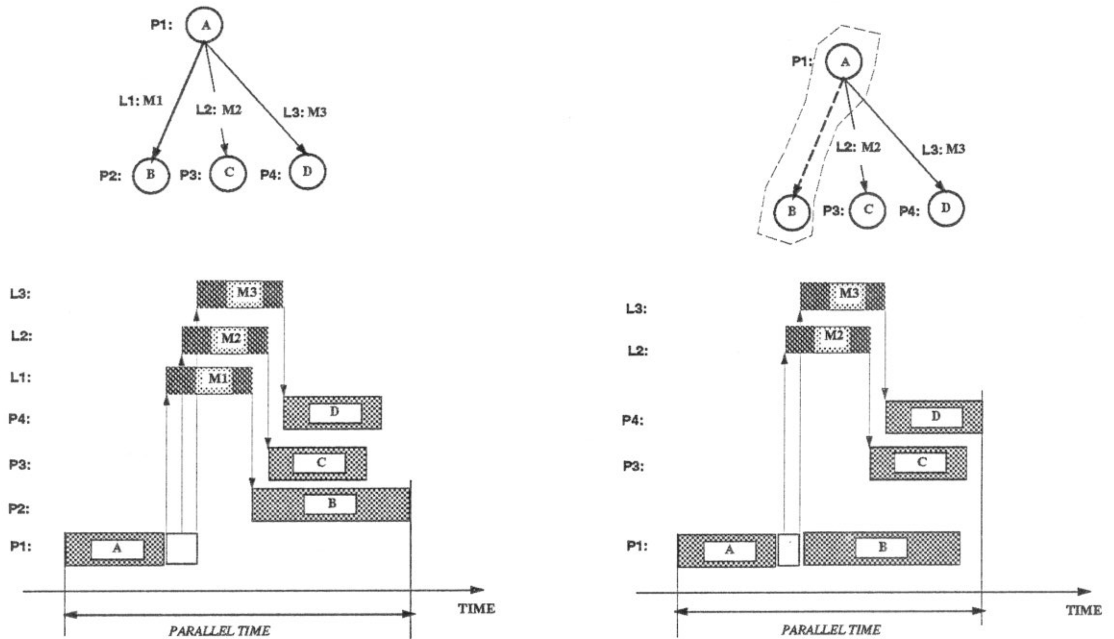

Figure 15: Clustering with non-blocking *Send*'s.

the *Send* primitives are non-blocking. Clustering tasks A and B results in a decrease of the parallel time of the graph. This would not be the case if the *Send* primitives were blocking (see Figure 11). Kim and Browne's method performs the clustering of linear chains of tasks, regardless of whether such an alteration will result in a larger parallel time. Therefore, the cluster-numbers reported for this algorithm are small, both for the blocking and the non-blocking paradigms.

Finally, *GL&S-10%* reports cluster numbers which are proportional to the numbers reported by Sarkar's algorithm. This is expected since, in its first pass, the method applies *GL* to the graph. This does not decrease the number of clusters substantially. The second pass applies Sarkar's heuristic, but only for 10% of the "heavier" edges.

In Figure 16, we present a diagram of execution-time measurements for the various clustering heuristics examined. The execution times represent measurements on FAST simulations running on a DEC-Alpha workstation. As expected, Sarkar's algorithm is substantially slower than the other heuristics, which have similar execution times.

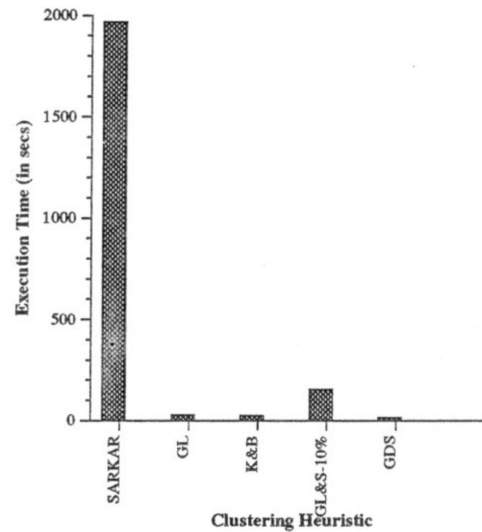

Figure 16: Execution times for the Clustering heuristics.

3.5.2 Mapping

To compare the mapping algorithms implemented in FAST, we applied them on the clustered parallel-execution graphs derived from the examples of the previous section, and mapped the clusters to 16 processors connected in a clique topology. In Figure 17, we give a diagram of speedups for different combinations of clustering and mapping algorithms. The speedup is a measure of the efficiency of the parallel computation described by the parallel-execution graph. Therefore, it can be used as a metric for the effectiveness of the mapping techniques examined. We give speedup results from eleven different combinations of clustering and mapping heuristics. *CP/MISF* is the scheduling heuristic employed for the merging of different clusters. Table 4 explains the notation used in the diagram of Figure 17.

From this figure, we can see that Sarkar's mapping algorithm gives better speedups than *SNC*. On the other hand, *SNC* reports better speedups than the Yang and Gerasoulis approach. This remark is true for all the clustering heuristics used, and both for blocking and non-blocking *Send*'s. Furthermore, we notice that the *SNC* approach performs almost as well as Sarkar's method, although it disregards communication costs in the parallel-execution graph. Therefore, the measured speedups are lower than the ideal linear speedups, mainly because of load-unbalancing, and because of

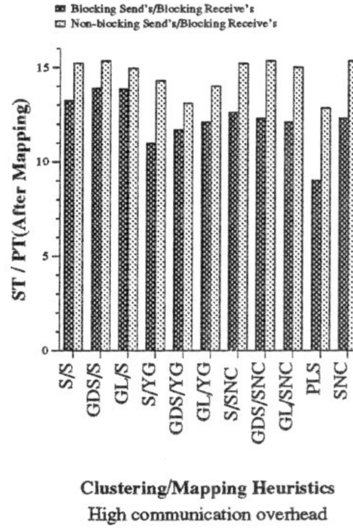

Figure 17: Speedups for different clustering-mapping strategies (16 processors).

Notation	Clustering Algorithm	Mapping Algorithm
S/S	Sarkar's	Sarkar's
GDS/S	Greedy Dominant Sequence	Sarkar's
GL/S	Greedy-Linear	Sarkar's
S/YG	Sarkar's	Yang&Gerasoulis
GDS/YG	Greedy Dominant Sequence	Yang&Gerasoulis
GL/YG	Greedy-Linear	Yang&Gerasoulis
S/SNC	Sarkar's	Sarkar's - No Communication Cost
GDS/SNC	Greedy Dominant Sequence	Sarkar's - No Communication Cost
GL/SNC	Greedy-Linear	Sarkar's - No Communication Cost
SNC	none	*SNC*
PLS	none	Priority List Scheduling, CP/MISF

Table 4: Clustering and Mapping algorithms' notation.

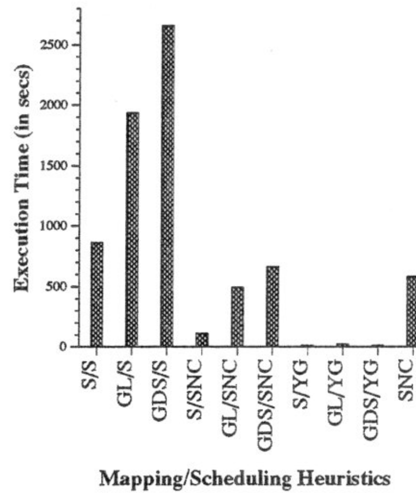

Figure 18: Execution times for the Mapping heuristics.

the data-dependences in the task-graph that result in almost sequential portions of the execution. The Priority List Scheduling algorithm (*PLS*) with no clustering, reports the lowest speedups. Using *SNC* on the parallel-execution graph of our example, with no prior clustering, results in speedups comparable to those derived when running *SNC* on the clustered graph. Another remark from Figure 17, is that the speedups reported from simulations of the non-blocking *Send*/blocking *Receive* primitives are, on the average, 20% higher than the speedups reported for blocking *Send*/blocking *Receive* primitives. This improvement is expected since the non-blocking *Send*'s incur smaller communication overhead to the processors of a parallel system.

In Figure 18, we present a diagram of measurements extracted from FAST for the execution times of the various mapping algorithms. Sarkar's algorithm is the slowest. *SNC*, which performs the mapping without taking into consideration communication delays and overhead, has a moderate execution time. Therefore, the very high time-complexity of Sarkar's approach is a result of the overhead for the estimation of communication costs while testing the different mapping choices at each step of the method. Furthermore, we notice that for the cases where we used Sarkar's clustering algorithm before the mapping, the cost of the mapping was smaller. This is because Sarkar's clustering heuristic results in low numbers of clusters.

3.6 Remarks

In this chapter we presented the clustering, scheduling, and mapping heuristics implemented in the current version of FAST. Using data derived from functional algorithm simulations of the Fast Multipole Method we performed a comparison study of these heuristics. From this study we conclude that, under the realistic scheme employed by FAST to assign weights to the edges of the task-graphs, the clustering heuristics examined do not achieve substantial improvements in parallel time. Nevertheless, clustering heuristics based on the methods introduced by Sarkar and by Kim and Browne, result in numbers of clusters that are significantly smaller than the initial numbers of tasks. Partitioning the task-graphs into a small numbers of clusters, expedites the mapping heuristics used after clustering.

Furthermore, we conclude that communication overhead does not play an important role in mapping clustered task-graphs to processors. The mapping heuristic *SNC*, which does not take into consideration communication costs, performs almost as well as *Sarkar*'s heuristic, with respect to the reported speedups. For the examples considered, the fast heuristic by Yang and Gerasoulis does not perform as well as slower heuristics, such as *SNC* and *Sarkar*'s. Preliminary experiments performed on graphs derived from the functional algorithm simulations of Barnes-Hut algorithm, however, show that Yang and Gerasoulis heuristic achieves competitive speedups. Finally, we conclude that the combination of clustering and mapping heuristics gives consistently better results than one-phase mapping algorithms, such as *Priority List Scheduling.*

Chapter 4

Message Ordering

In this chapter we show that, under the blocking communication assumption, message latency is largely determined by the order with which messages are transmitted from task-graph nodes. Consequently, this ordering affects the parallel time of the corresponding task-graph and, therefore, it affects the steps taken by the clustering, scheduling, and mapping heuristics discussed in Chapter 3. We consider the problem of finding message-transmission orderings that achieve minimum parallel times for task-graphs mapped on virtual clique architectures. We give a polynomial-time dynamic-programming algorithm for finding such optimal orderings. This algorithm has been incorporated in the procedures of FAST that estimate message-latency and parallel time. It has been used in the implementation of the clustering, scheduling, and mapping heuristics for the simulation of message-passing systems with blocking communication primitives.

4.1 Message Ordering

We extend the definition of a parallel-execution graph, so that it takes into consideration the ordering of message transmissions from the task-nodes. To this end, a parallel-execution graph is defined as follows:

$$G_{pe} = G(V,\ E_{pe},\ proc,\ \mathcal{O}(V),\ T,\ D)$$

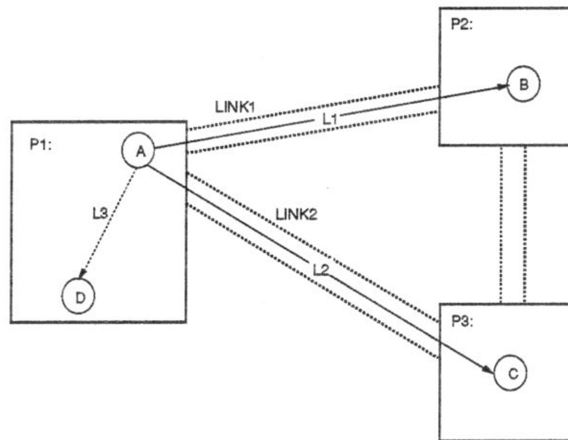

Figure 19: Example of communication between tasks and between processors.

where V, E_{pe}, $proc$, T, and D are defined exactly as in Chapter 2. $\mathcal{O}(u)$ defines an ordering among the messages dispatched from task u, $\forall v \in V$. We use the symbol "$\prec$" to denote the order of transmission between two messages. For instance, if node u sends a message to node w after having dispatched a message to node v, and no other message is sent by u in between, this is denoted as $(u, v) \prec (u, w)$.

As an example, consider Figure 19 where tasks A and D reside on Processor $P1$, B resides on $P2$, and C resides on $P3$. Task A performs its computation and then sends two messages to tasks B and C, in that order. Figure 20 shows the communication overheads incurred by the tasks. At the end of A's computation, processor $P1$ has to load the output buffers of its network interface so as to initiate messages $L1$ and $L2$. After this, it starts the computation of task D. Processors $P2$ and $P3$ wait until the incoming buffers of their network interfaces have been loaded with messages $L1$ and $L2$ respectively. Then they start executing by copying in memory the contents of the messages and then proceeding in the computation of tasks B and C. Under the blocking scheme adopted, the ordering of message-dispatches from processor $P1$ is important because it determines the starting time of tasks B and C on the other processors. From Figure 20 it is clear that if message $L2$ was sent before $L1$, the overall PT would be smaller since C's computation dominates the overall execution. Notice that D is mapped to the same processor as A and thus there is no communication overhead due to edge $L3$. The data exchange corresponding to $L3$ is

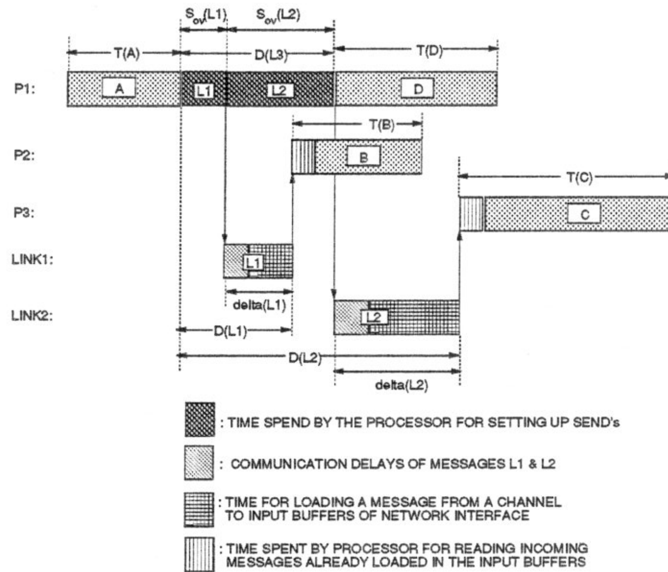

Figure 20: Communication overheads under the blocking communication approach.

performed implicitly through *Write* and *Read* operations in the local memory of $P1$. Nevertheless, D can start only after A has finished the transmission of messages $L1$, $L2$, and has freed Processor $P1$.

From the previous example it is clear that the order in which a task sends its messages affects the PT since it affects the D-weights assigned to the edges of the parallel execution graph G_{pe}. In order to define an optimal execution for a parallel algorithm, it is necessary to find an optimal ordering which, among all possible orderings, leads to a minimum PT for a specific clustering, mapping, and scheduling configuration.

4.2 Optimal Orderings of Messages

In this section we present definitions and lemmas characterizing the optimal ordering sought, and delineating the method proposed for computing such an ordering. First, we give a number of definitions for the elements of the parallel execution (weighted) graph, and the corresponding non-weighted graph $\widehat{G} = G(V, E_{pe})$. For every vertex $v \in V$, we define $height(v)$ as the length of the longest path in $\widehat{G}$ from v to the set of vertices with outdegree zero. Obviously for every $x,\ y\ \in V$ with $height(x) =$

$height(y)$ neither (x, y) nor (y, x) belongs to E_{pe}. Furthermore, we denote:

$$height(G_{pe}) = \max_{v \in V}\{height(v)\} \quad (1)$$

$$Succ(v) = \{w \in V \mid (v, w) \in E_{pe}\} \quad (2)$$

We modify the definition of *Parallel Time* on the parallel execution graph to take into consideration the ordering of messages :

Definition 4.2.1 *The Parallel Time* (**PT**) *of a parallel execution graph with a given message ordering $\mathcal{O}$ is defined as the weight of its critical path, i.e, of the path with the largest sum of node and edge weights. Let $G_{pe} = G(V, E_{pe}, proc, \mathcal{O}, T, D)$ be a parallel execution graph, and $\Pi = \{\pi_i \mid \pi_i = (v_1, v_2, \ldots, v_{m_i})\}$ be the set of paths in G_{pe}, where $indegree(v_1) = outdegree(v_{m_i}) = 0$. Then PT is defined formally by:*

$$PT(G_{pe} \mid \mathcal{O}) = \max_{\pi_i \in \Pi}\Big\{\sum_{j=1}^{m_i} T(v_j) + \sum_{k=1}^{m_i-1} D((v_k, v_{k+1}) \mid \mathcal{O})\Big\} \quad (3)$$

We use the notation $D((u, v) \mid \mathcal{O})$ only when we want to distinguish between two different orderings. Otherwise we drop the $\mathcal{O}$ for notational simplicity. The same holds for all our definitions. We define the *Start Time* for every node of the graph as the earliest time at which a task can start executing:

Definition 4.2.2 *The start time of every node $v \in V$ ($indegree(v) \neq 0$) in a parallel execution graph with a given message ordering $\mathcal{O}$, is defined as the largest among weights of all paths starting at a node with indegree zero, and ending at v, not including $T(v)$. In other words, for the set of paths $\Pi = \{\pi_i \mid \pi_i = (v_1, \ldots, v_{m_i}, v)\}$, where $(v_1, v_2), \ldots, (v_{m_{i-1}}, v_{m_i}), (v_{m_i}, v) \in E$, and $indegree(v_1) = 0$ it is:*

$$ST(v \mid \mathcal{O}) = \max_{\pi_i \in \Pi}\Big\{ \sum_{j=1}^{m_i} T(v_j) + \sum_{k=1}^{m_i-1} D((v_k, v_{k+1}) \mid \mathcal{O}) + D((v_{m_i}, v) \mid \mathcal{O}) \Big\}$$

If indegree(v) = 0 we assume that ST(v) = 0 according to the principle of no unforced idleness [62]. We can easily see that this definition for start time is equivalent to the following recursive one:

$$ST(v \mid \mathcal{O}) = \max_{\substack{u \in V \\ (u,v) \in E}} \Big\{ ST(u \mid \mathcal{O}) + T(u) + D((u,v) \mid \mathcal{O}) \Big\} \quad (4)$$

We also define the *Finish Time* of nodes in the graph:

Definition 4.2.3 *For every node $v \in V$ in a parallel execution graph with a given message ordering $\mathcal{O}$, the finish time $FT(v)$ of v is defined as the largest among weights (including $T(v)$) of all paths starting from v and ending at a vertex with outdegree zero. This is equivalent to:*

$$FT(v \mid \mathcal{O}) = T(v) + \max_{\substack{w \in V \\ (v,w) \in E}} \Big\{ D((v,w) \mid \mathcal{O}) + FT(w) \Big\} \quad (5)$$

Notice that $FT(v \mid \mathcal{O})$ is a lower bound on the time that elapses between the start time of v and the latest completion time among v's descendants with outdegree zero (it might be the case that the completion time of those descendants is determined by some other path in G_{pe} which does not include v). It is not difficult to deduce the following lemma relating PT to ST and FT:

Lemma 4.2.1 *The parallel time of a parallel execution graph G_{pe} with a message ordering $\mathcal{O}$, can be expressed as follows:*

$$PT(G_{pe} \mid \mathcal{O}) = \max_{v \in V} \Big\{ ST(v \mid \mathcal{O}) + FT(v \mid \mathcal{O}) \Big\} \quad (6)$$

Consider a parallel execution graph G_{pe} with an ordering $\mathcal{O}$ of messages, let v be a node in V, $Succ(v) = \{u, w_1, \ldots, w_m\}$, and u be the node in $Succ(v)$ with the *largest* Finish Time $FT(u)$ (Figure 21). We prove the following basic lemma:

Lemma 4.2.2 *Altering the message ordering*

$$\mathcal{O}(v) \;:\; (v, w_1) \prec \ldots \prec (v, w_j) \prec (v, u) \prec (v, w_{j+1}) \prec \ldots \prec (v, w_m) \quad (7)$$

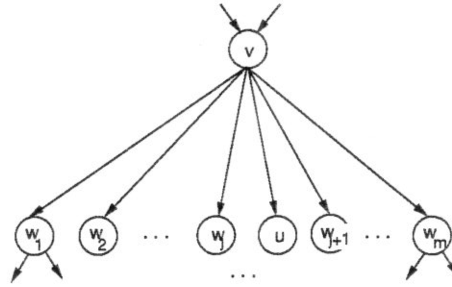

Figure 21: Message Ordering Alteration.

to

$$\mathcal{O}'(v) \; : \; (v,u) \prec (v,w_1) \prec \ldots \prec (v,w_m),$$

by dispatching message (v,u) *first does not increase the Parallel Time of the resulting graph, i.e.,*

$$PT(G_{pe} \mid \mathcal{O}) \; \geq \; PT(G_{pe} \mid \mathcal{O}') \tag{8}$$

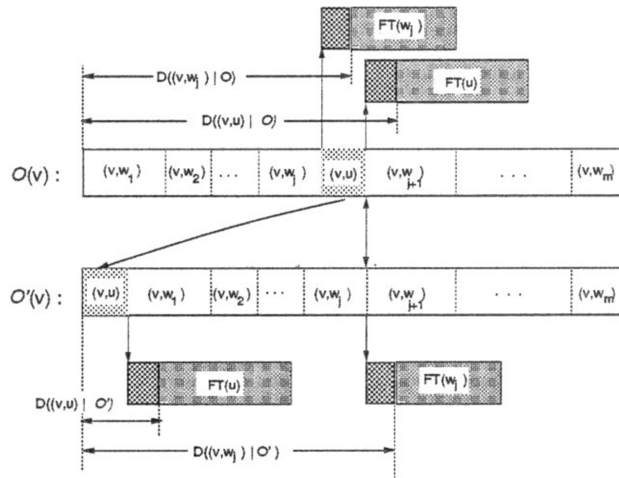

Figure 22: Message Ordering Alteration.

Proof: We give a proof based on Figures 21 and 22. In Figure 22 the slots marked (v, w_i), $i = 1, \ldots, j$ correspond to the overhead times $S_{ov}((v, w_i))$ incurred by processor *proc*(v) for dispatching v's messages. The ordering alteration described in the

lemma affects the D-values of edges (v, w_1),. . .,(v, w_j), (v, u) in Figure 21. Weights of paths not including these edges do not change. Thus, any possible modification in PT will come as a result of the changes in paths including one of (v, w_1),. . .,(v, w_j), (v, u). First, we look at paths including (v, u). From Figure 22 we can easily see that:

$$D((v, u) \mid \mathcal{O}) \;\geq\; D((v, u) \mid \mathcal{O}')$$

It is also true that $ST(v)$ and $FT(u)$ do not change because of the transformation from $\mathcal{O}$ to $\mathcal{O}'$. Moreover, by definition:

$$PT(G_{pe} \mid \mathcal{O}) \;\geq\; ST(v) + D((v, u) \mid \mathcal{O}) + FT(u)$$

Thus,

$$PT(G_{pe} \mid \mathcal{O}) \;\geq\; ST(v) + D((v, u) \mid \mathcal{O}') + FT(u)$$

which means that the PT cannot increase due to the change of (v, u)'s timing.

Next, we examine paths which include one of (v, w_1),. . .,(v, w_j). From Figure 22 we can also see that:

$$\forall\, i,\; 1 \leq i \leq j \quad D((v, w_i) \mid \mathcal{O}') \geq D((v, w_i) \mid \mathcal{O})$$

However, since u has the maximum FT value among FT values of nodes $\{u,\ w_1, \ldots, w_m\}$, it is true that $\forall i,\ 1 \leq i \leq j$:

$$D((v, u) \mid \mathcal{O}) + FT(u) \;\geq\; D((v, w_i) \mid \mathcal{O}') + FT(w_i)$$

Thus, although $\{w_1, \ldots, w_j\}$ receive messages from v under $\mathcal{O}'$ later than under $\mathcal{O}$, $PT(G_{pe} \mid \mathcal{O}')$ is not increased relative to $PT(G_{pe} \mid \mathcal{O})$. □

What Lemma 4.2.2 says, is that among the messages dispatched from some task, we can always schedule first the message that goes to the task with the largest FT, without worsening the overall PT.

We now give an extension of Lemma 4.2.2, which shows that if the first j messages from some task are sent to the tasks with the j largest FT values sorted in descending order, then PT won't be increased if we dispatch the message to the task with the $(j+1)$-st largest FT value, immediately after the j-th message. Let $v \in V$, $Succ(v) = \{w_1, \ldots, w_m\}$, and let w_k be the node in $Succ(v)$ with the $(j + 1) - st$ largest FT value (see Figure 23).

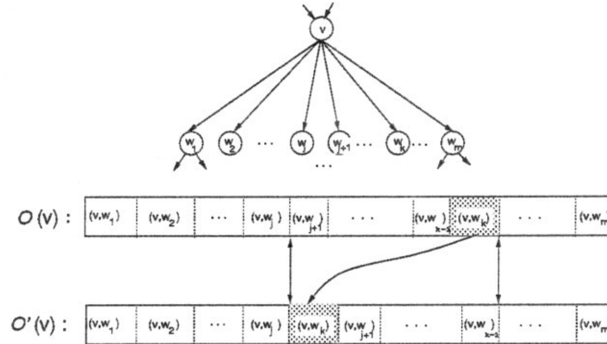

Figure 23: Message Ordering Alteration.

Lemma 4.2.3 *Assume that messages sent from task v are dispatched in the following order:*

$$\begin{aligned} \mathcal{O}(v) \quad : \quad & (v, w_1) \prec \ldots \prec (v, w_j) \prec \ldots \prec \\ & (v, w_k) \prec \ldots \prec (v, w_m) \end{aligned}$$

If this order is altered by dispatching message (v, w_k) immediately after (v, w_j):

$$\begin{aligned} \mathcal{O}'(v) \quad : \quad & (v, w_1) \prec \ldots \prec (v, w_j) \prec \\ & \prec (v, w_k) \prec (v, w_{j+1}) \prec \ldots \prec (v, w_m) \end{aligned}$$

the Parallel Time of the resulting graph is not increased, i.e.,

$$PT(G_{pe} \mid \mathcal{O}) \;\geq\; PT(G_{pe} \mid \mathcal{O}') \tag{9}$$

Proof: Notice that only $D((v, w_{j+1}))$, ..., $D((v, w_k))$ change because of the reordering. The proof is similar to the one of the previous lemma, and is omitted. □

4.3 An Algorithm for Optimal Message Ordering

In this section we give an algorithm that computes an optimal ordering of messages $\hat{\mathcal{O}}$ for a given parallel-execution graph G_{pe}. The algorithm breaks down the problem into

stages by processing nodes in ascending order of their *height*: it starts by computing the FT values of nodes with outdegree zero, i.e., of nodes with *height* zero. Then, after finding the optimal ordering for messages transmitted by nodes with *height* k, it computes the FT values of these nodes, and uses them to specify the optimal message ordering at nodes of *height* $k + 1$. The algorithm completes after having derived the optimal ordering of messages sent by nodes with indegree zero. It is a typical application of the *dynamic-programming* method [54].

We assume that the graph is given in an adjacency-list representation; $ordered[v]$, $v \in V$ is an array which shows whether node's v messages have been ordered or not; $tot_ovrhd[v]$ ($v \in V$) accumulates the communication overhead incurred by processor $proc(v)$ because of messages dispatched from task v; $Prev((v, w))$ gives the edge which precedes (v, w) in the ordering $\mathcal{O}(v)$ of v's outgoing edges, and SORT returns its input list of nodes sorted in descending order of their FT values.

Algorithm: *Optimal ordering of messages.*
Input: Parallel Execution graph G_{pe} with a specified clustering, mapping, and scheduling configuration.
Output: An optimal ordering $\widehat{\mathcal{O}}$ of messages of G_{pe}.

ORDER_MSG(G_{pe}):
1. **for** each vertex $v \in V[G_{pe}]$
   do if not $ordered[v]$
     then ORDER(v)

ORDER(v):
1. **for** each vertex $u \in Succ(v)$
   do if not $ordered[u]$
     then ORDER(u)
2. $\widehat{\mathcal{O}}(v) \leftarrow$ SORT($Succ(v)$)
3. $tot_ovrhd[v] \leftarrow 0$
4. **for** each $(v, w) \in \widehat{\mathcal{O}}(v)$
   if $(proc(v) \neq proc(w))$ {

$tot_ovrhd[v] \leftarrow tot_ovrhd[v] + S_{ov}((v,w))$

$D((v,w) \mid \hat{\mathcal{O}}(v)) \leftarrow tot_ovrhd[v] + \delta((v,w))$

}

5. **for** each $w \in \hat{\mathcal{O}}(v)$

if $(proc(v) == proc(w))$

$D((v,w) \mid \hat{\mathcal{O}}(v)) \leftarrow tot_ovrhd[v]$

6. $FT(v) \leftarrow T(v) +$

$\max_{w \in Succ(v)}\{D((v,w) \mid \hat{\mathcal{O}}(v)) + FT(w)\}$

7. $ordered[v]$ = **true**

The loop on line 1 in ORDER_MSG(G_{pe}) is executed $\Theta(|V|)$ times. The procedure ORDER is called exactly once for each vertex $v \in V$. During an invocation of ORDER(v), the call to SORT takes time $O(|Succ(v)| \cdot \log(|Succ(v)|))$. The loops on lines 4 and 5, as well as line 5 take time $\Theta(|Succ(v)|)$. Consequently, the total cost incurred by calls to ORDER is $O(|E_{pe}| \cdot \log(|E_{pe}|))$, and the overall time complexity of the algorithm is:

$$O(|V| + |E_{pe}| \cdot \log(|E_{pe}|)).$$

The algorithm does not make any distinction between different kinds of edges, namely between edges that correspond to actual messages sent between processors, and "pseudo-messages", i.e., edges joining tasks mapped onto the same processor. This does not affect the optimality of the resulting ordering. The overhead carried by "pseudo-messages" is null, the scheduling of their recipient tasks is predetermined, and is not affected by the ordering procedure. The following theorem proves the correctness of the algorithm.

Theorem 4.3.1 *An optimal ordering $\hat{\mathcal{O}}$ of messages is achieved on a parallel-execution graph G_{pe}, if for every task $v \in V$ its outgoing messages are dispatched according to the descending order of the FT-values of destination tasks.*

Proof: We assume that we have a parallel execution graph $G_{pe} = G(V, E_{pe}, proc, \tilde{\mathcal{O}}, T, D)$ where $\tilde{\mathcal{O}}$ is an optimal ordering not necessarily the same as $\hat{\mathcal{O}}$. We use

induction on $height(v)$ to show that if we alter $\widetilde{\mathcal{O}}(v)$ to $\widehat{\mathcal{O}}(v)$, we get a parallel time $PT(G_{pe} \mid \widehat{\mathcal{O}}) \leq PT(G_{pe} \mid \widetilde{\mathcal{O}})$.

First, we consider all $v \in V_1$, i.e., $v \in V$ with $height(v) = 1$. According to Lemma 4.2.2 we can dispatch first the message sent from v to the $w \in V_0$ with the largest FT value, without increasing the parallel time. Furthermore, given Lemma 4.2.3 we can dispatch second, third, and so on, the messages sent from v to the nodes with the second, third, and so on largest FT values respectively, again without increasing PT. This argument holds for all $v \in V_1$. In this way we order the messages sent from tasks v according to the descending order of the FT values of the recipient tasks. Thus, altering $\widetilde{\mathcal{O}}(V_1)$ to $\widehat{\mathcal{O}}(V_1)$ does not increase PT:

$$PT\Big(G_{pe} \mid \widetilde{\mathcal{O}}(V - \mathcal{V}_1), \widehat{\mathcal{O}}(\mathcal{V}_1)\Big) \leq PT(G_{pe} \mid \widetilde{\mathcal{O}}) \tag{10}$$

where $\mathcal{V}_i$ is defined as $V_0 \cup \ldots \cup V_i$. Our induction hypothesis is that for some k with $1 < k < height(G_{pe})$, it is true that:

$$\begin{aligned} PT\Big(G_{pe} \mid \widetilde{\mathcal{O}}(V - \mathcal{V}_k), \widehat{\mathcal{O}}(\mathcal{V}_k)\Big) &\leq \\ PT\Big(G_{pe} \mid \widetilde{\mathcal{O}}(V - \mathcal{V}_{k-1}),\ \widehat{\mathcal{O}}(\mathcal{V}_{k-1})\Big)& \end{aligned} \tag{11}$$

For the induction step we apply the same transformations as in our induction basis over all $v \in V_{k+1}$ (Figure 24), altering $\widetilde{\mathcal{O}}(V_{k+1})$ to $\widehat{\mathcal{O}}(V_{k+1})$. Clearly the orderings $\widetilde{\mathcal{O}}(V - \mathcal{V}_{k+1})$ of messages at nodes $V - \mathcal{V}_{k+1}$, and $\widehat{\mathcal{O}}(\mathcal{V}_k)$ at $\mathcal{V}_k$ are not affected. Considering Lemmas 4.2.2 and 4.2.3 it is not difficult to see that the following is true:

$$\begin{aligned} PT\Big(G_{pe} \mid \widetilde{\mathcal{O}}(V - \mathcal{V}_{k+1}), \widehat{\mathcal{O}}(\mathcal{V}_{k+1})\Big) &\leq \\ PT\Big(G_{pe} \mid \widetilde{\mathcal{O}}(V - \mathcal{V}_k),\ \widehat{\mathcal{O}}(\mathcal{V}_k)\Big)& \end{aligned} \tag{12}$$

Hence we conclude that for $1 \leq i \leq height(G_{pe})$:

$$\begin{aligned} PT\Big(G_{pe} \mid \widetilde{\mathcal{O}}(V - \mathcal{V}_i), \widehat{\mathcal{O}}(\mathcal{V}_i)\Big) &\leq \\ PT\Big(G_{pe} \mid \widetilde{\mathcal{O}}(V - \mathcal{V}_{i-1})\ \widehat{\mathcal{O}}(\mathcal{V}_{i-1})\Big)& \end{aligned} \tag{13}$$

From (13), and (10) we have:

$$PT(G_{pe} \mid \widehat{\mathcal{O}}) \leq PT(G_{pe} \mid \widetilde{\mathcal{O}}),$$

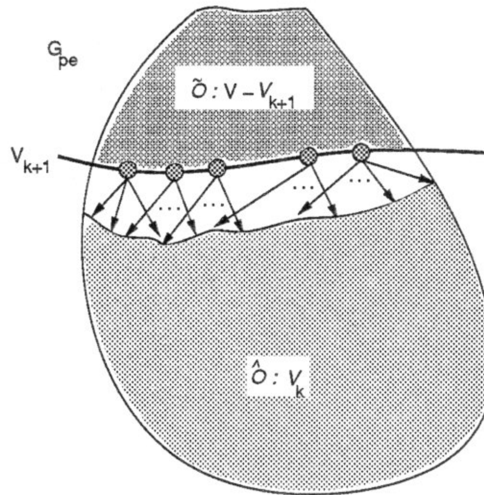

Figure 24: Induction Step.

but since $\tilde{\mathcal{O}}$ is optimal by assumption, it can only be that:

$$PT(G_{pe} \mid \hat{\mathcal{O}}) \;=\; PT(G_{pe} \mid \tilde{\mathcal{O}}) \tag{14}$$

which proves that $\hat{\mathcal{O}}$ is also optimal. □

4.4 Remarks

In this chapter we showed that under the blocking communication paradigm, the order of message transmission affects the overall task-graph execution (parallel) time. We proved that for some given clustering of the task-graph, there exists an ordering of messages which results in a minimum parallel time. Finally, we introduced a polynomial-time algorithm which finds such optimal orderings of messages for given task-graphs. This algorithm has been combined with clustering heuristics employed in FAST, to cluster task-graphs generated when simulating parallel systems with blocking communication.

Chapter 5

Validation of FAST on the SIMPLE CFD-kernel.

In this chapter we present a case-study performed with FAST on a computational fluid dynamics kernel called SIMPLE. This study is used as a validation of our technique. We compare the Parallel Time and Speedup figures reported by FAST with those reported independently by other researchers who implemented SIMPLE on iPSC/2 multiprocessors. We show that FAST reports numbers which are reasonably close to those measured on real implementations, and follow the same trends for parallel systems with increasing numbers of processors. Furthermore, we present data from simulations of SIMPLE for problem sizes much larger than those reported so far and study the inherent computation and communication characteristics of the application.

The SIMPLE computation simulates the hydrodynamics of a pressurized fluid inside a spherical cell. The state of the problem is modeled in terms of a cylindrical coordinate space. Values of various physical quantities are maintained at a number of points in the coordinate system. Because of the spherical symmetry of the problem, the physical domain is reduced to a quarter of an annular region projected onto a two-dimensional plane. The two-dimensional projection is transformed into Cartesian coordinates. In order to solve the equations that simulate the motion of the fluid, time is discretized into a sequence of steps and the physical domain is discretized into a finite number of nodes.

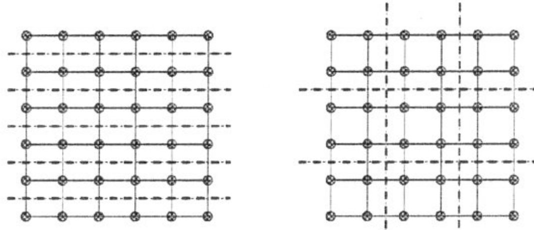

Figure 25: Strip vs. Square partitioning.

After these transformations, the computer simulation of each time-step solves a number of equations on a two-dimensional grid with size proportional to the size of the spherical cell examined, and updates the values of various physical variables. Every time-step in SIMPLE can be split in five successive phases, according to different types of data dependencies and communication patterns occurring in the algorithm [42]: *Delta* phase (computation of the length of the next time-step), *Hydro* phase (computation of new acceleration, velocity, coordinates, Jacobian, volume of revolution, new density, artificial viscosity, new energy, and pressure), *Heat* phase (computation of the new temperature and heat), *Energy1* phase (computation of energy and work), and *Energy2* phase (computation of total heat, and error).

Computation and communication patterns are identical across different time-steps. Therefore, it suffices to simulate and study one time-step only [60].

5.1 Functional Algorithm Simulation of SIMPLE

5.1.1 Derivation of the task-flow graph

For the functional simulation of SIMPLE, the user defines at the input of FAST the size of the two-dimensional grid and a scheme for partitioning it. Subsequently, the first part of FAST's front-end generates the grid, its partition, and the Intermediate Representation of basic computational blocks and message exchanges representing the computation of SIMPLE on the specific grid for one time-step. This Intermediate Representation is transformed by the second part of FAST's front-end into a task-flow graph, as explained in Chapter 2.

The partitioning scheme splits in an ad-hoc way the two-dimensional grid into a designated number of blocks according to some designated geometry, e.g., square blocks or strips (see Figure 25).

Figures presented in the following sections correspond to a square-block-partitioning unless stated otherwise. We set the number of partitions to be equal to the number of available processors. Thus, the mapping of tasks to processors was straightforward and FAST's clustering and mapping stages were bypassed.

We do not give any results from functional simulations of SIMPLE with automatic partitioning of the grid performed by the clustering and mapping heuristics. One reason for this is that the computational structure of the problem is static and uniform. So there is no need to use "expensive" partitioning techniques. Furthermore, preliminary experiments with automatic partitioning reported poor parallel performance. In the absence of ad-hoc grid-partitioning, FAST generated task-flow graphs of very fine granularity. The clustering, scheduling, and mapping heuristics employed were not very successful in splitting those graphs into coarse-grain subgraphs, mapping the result on the available processors and, at the same time, maintaining high processor utilization and load-balance.

5.1.2 Validating FAST

We chose to study SIMPLE because there exist a number of published performance measurements by different research groups [42, 60]. Therefore, we can use it to assess the validity of the Functional Algorithm Simulation method and the accuracy of FAST. We compare published measurements and estimates of parallel time and speedup with the respective figures reported by our system. The data reported in [42, 60] were collected from implementations of SIMPLE on the iPSC/2 multiprocessor. Thus, for the functional simulations of SIMPLE, FAST was provided with the hardware parameters of the iPSC/2 [9, 34].

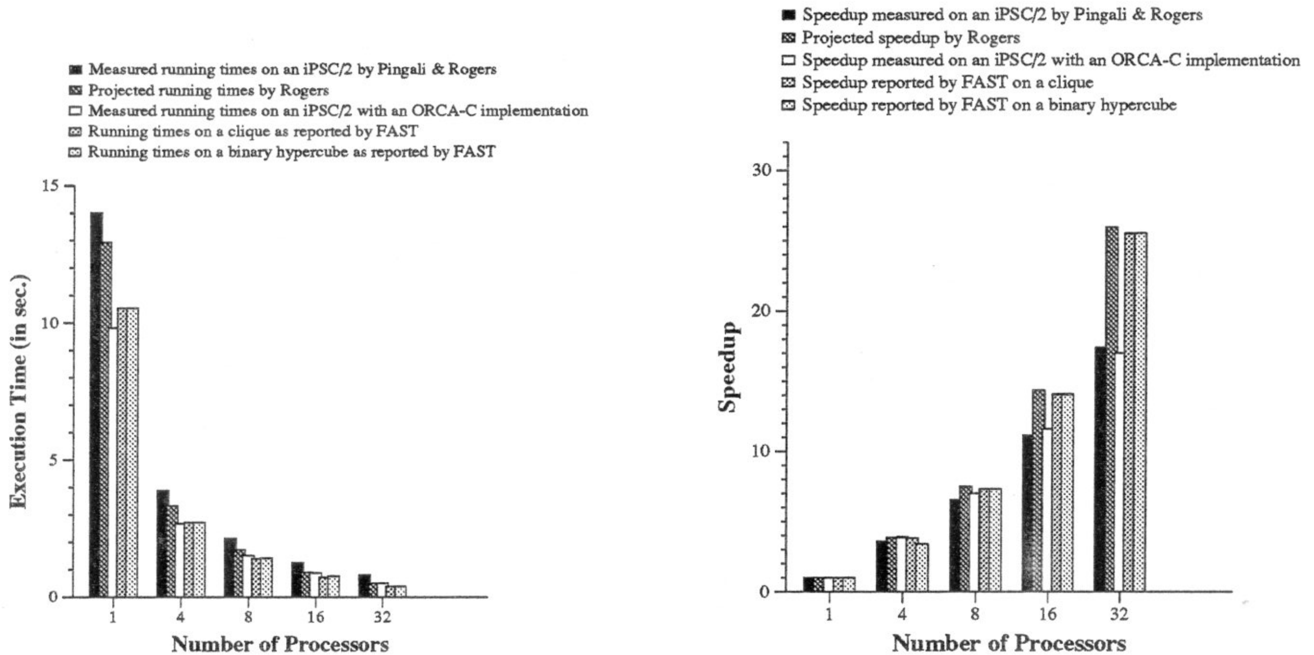

Figure 26: Parallel Time and Speedup for SIMPLE on 4096-point grid.

Comparisons cannot be strict for various reasons: as far as parallel time is concerned, different implementations of SIMPLE, even on the same type of multiprocessor might report varying execution times because of differences in machine configurations and software platforms. Moreover, various analyses are based on different assumptions influencing the parallel times they report. Finally, simulations capture the parallel executions approximately, and therefore it is expected that their figures will not be identical to data measured on actual implementations.

Speedup comparisons presuppose that all speedups are calculated with respect to the same sequential time. This is not always true because of differences in system and implementation software, and hardware. For instance, in the right diagram of Figure 26 the speedups are computed as the ratio of parallel times over the sequential times presented in the left diagram (values "Execution Time" when the "Number of processors" is equal to one). However, the sequential times differ. Nevertheless, a comparison is invalid if we focus on observing trends and ranges of values, rather than absolute numbers.

5.1.3 Validation experiments

Figure 26 (left), shows the parallel execution times of SIMPLE on a 4096-point grid for one time-step. The plots labeled "Measured running times" and "Projected running times" present data from [60]. The "Measured running times" correspond to a SIMPLE-implementation with Pingali and Rogers' parallelizing compiler. The "Projected running times" were extracted from a model of a hand-written implementation of SIMPLE which takes into consideration the extra overhead representative of the functional language used in [60] (e.g. time spent for array allocations), as well as times spent for floating-point computations and message transmissions. This model is optimistic because it assumes that the workload can be divided evenly among all processors and that the processors are fully utilized during parallel execution. Times reported by FAST correspond to two different interconnection networks: a (limited) clique and a binary hypercube. As expected, the clique and the hypercube show almost identical performance characteristics since communication overhead is dominated more by the high setup times and less by propagation and congestion delays. Furthermore, the model of communication implemented in the iPSC/2 hypercube guarantees almost uniform communication among all processors, which makes it equivalent to a clique [34, 53]. Figure 26 (right), gives the respective speedups, plus the speedup achieved with an ORCA-C implementation of SIMPLE by Lee, Lin and Snyder [42].

From these diagrams, we can see that our system reports numbers that are quite close to measurements and projections by other researchers. In fact, the parallel times given by FAST are slightly smaller than those of the optimistic model for the hand-written implementation. This does not come as a surprise since the code that generated the Intermediate Representation of the parallel execution was hand-written, and also we did not take into consideration other parameters besides computation and communication times (for instance, system overhead).

FAST gives us the flexibility to experiment with much larger problem sizes and more processors than those reported so far. Also, it allows us to try different partitioning schemes and collect more detailed information with respect to the parallel executions of the algorithms studied. For example, we present diagrams derived from

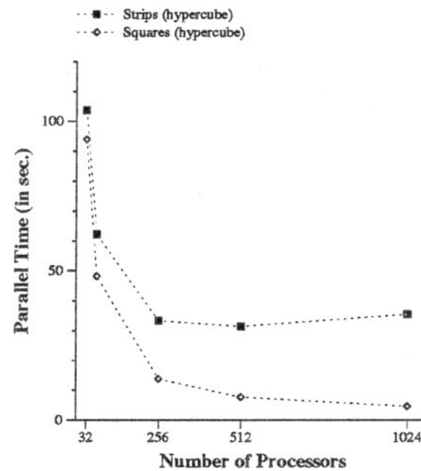

Figure 27: Strip vs. Square partitioning of 2^{20} grid-point SIMPLE on a hypercube.

the functional simulations of SIMPLE on problems with as many as 2^{20} grid points. Figure 27 compares two partitioning approaches: strips vs. squares. In the former, the computation grid is split into strips whereas in the latter it is split into square blocks. Square-block partitioning results in more messages with smaller size than strip partitioning. The computation on strip-partitioned grids has longer sequential parts, larger message sizes, and more overall transmitted data which increase with problem-size. As expected, square-partitioning results in better performance. The plots in Figure 27 are similar in shape to the ones reported in [42] for the iPSC/2 and the NCUBE.

5.1.4 Using FAST to collect profiles for SIMPLE

To illustrate the use of FAST further, we used it to gather a profile of 4096 grid-point SIMPLE executions on parallel systems with 64, 256, and 1024 processors. In Figure 28, we present the variation of busy processors and channels during a parallel execution on a 64-processor *clique* (solid line). Moreover, we give an approximate estimate for the duration of the five algorithmic phases of SIMPLE during this execution; the nodes of the task-flow graph can be ascribed to one of these phases. The dotted line represents the temporal variation of the average of phase-numbers ascribed to tasks which are active at each moment. The y-axis for the phase-diagram is drawn on the

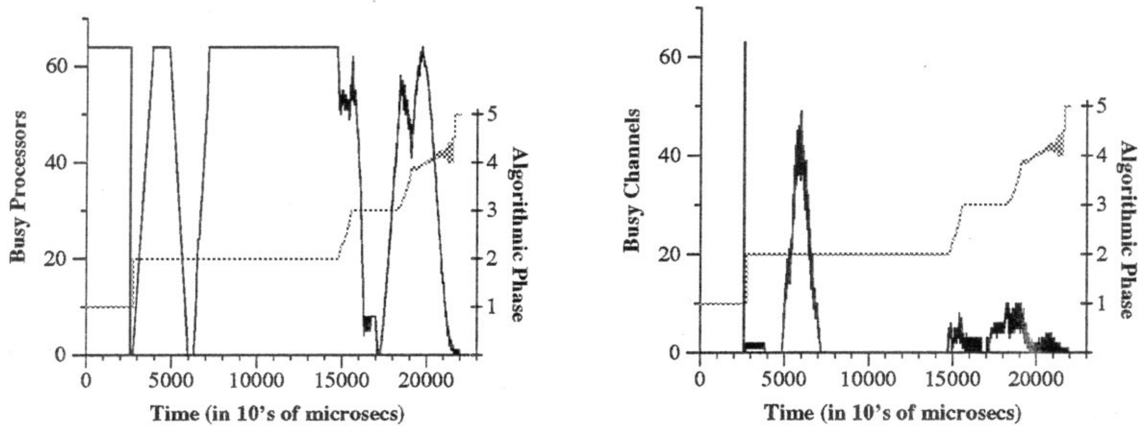

Figure 28: FAST results for 4096 grid-point SIMPLE on a 64-processor *clique* (PT = 220022 μs).

right part of the plot.

From the left diagram in Figure 28 we can see that available parallelism is very high in the first phase when all processors simultaneously compute a tentative duration for the next time-step. Then, they send their outcome to one processor which calculates and broadcasts back the global time-step. This results in the first spike of busy links in the right diagram of Figure 28 and the corresponding sharp drop in the number of busy processors. At the beginning of the second phase (*Hydro*), processors compute acceleration, velocity, and new coordinates for their assigned parts of the grid, and forward the new values to their neighbors. This explains the high number of active tasks, the subsequent sharp drop in processor activity and the corresponding second spike of busy communication-links. Parallelism starts increasing as processors resume their local execution of the second phase, after having sent and received messages to and from neighboring processors. The third phase (*Heat*), consists of a short, highly parallel computation and two "sweeps" of the grid along its horizontal and vertical directions. Thus, it is mostly sequential and requires limited data-exchange between grid-points [42]. Parallelism drops sharply and active links are few. In the fourth phase (*Energy1*), computations are local in nature and parallelism is high. Finally, the fifth phase is very short and its effect on overall parallel time is negligible. It performs a fast aggregate operation over all grid-points to accumulate the total energy

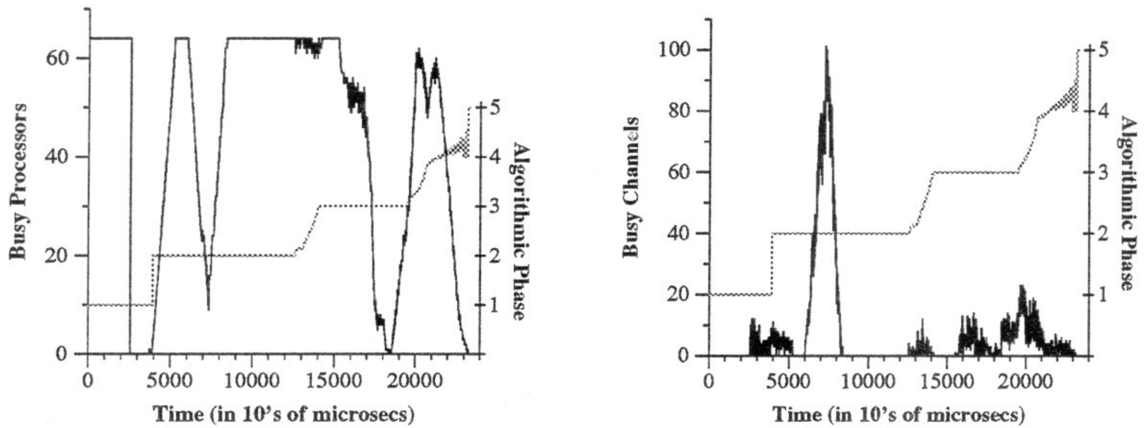

Figure 29: FAST results for 4096 grid-point SIMPLE on a 64-processor *hypercube* (PT = 232988 μs).

and work.

An almost identical computation profile is derived from the functional simulation of a 4096 grid-point SIMPLE simulation running on a 64-processor *binary hypercube* (Figure 29, left). The communication profile is quite similar, although the number of busy channels in the hypercube is larger. This is because the average number of hops per message (and thus of links used by a message) on a hypercube is twice that on the clique, where all messages are one-hop.

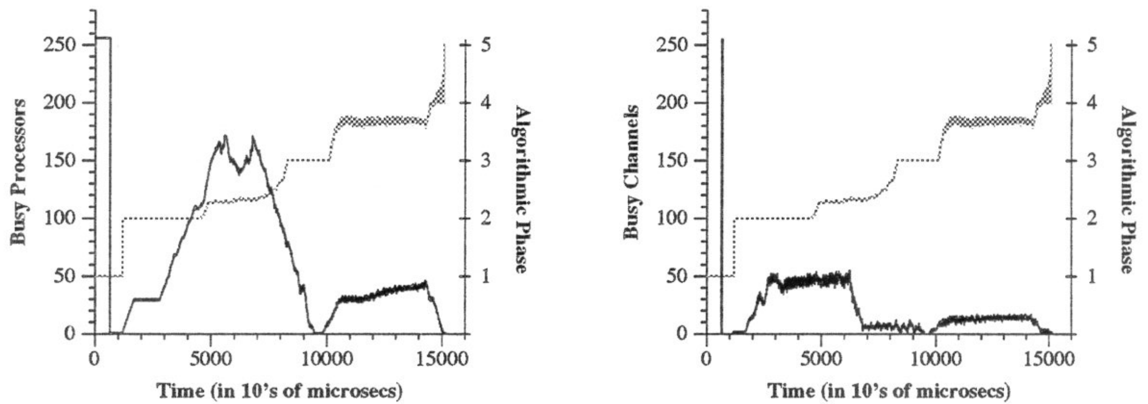

Figure 30: FAST results for 4096 grid-point SIMPLE on a 256-processor *clique*.

In Figure 30 (left) we present a busy-processors diagram derived with FAST for a SIMPLE execution on a 256-processor clique. We notice that the shape of the

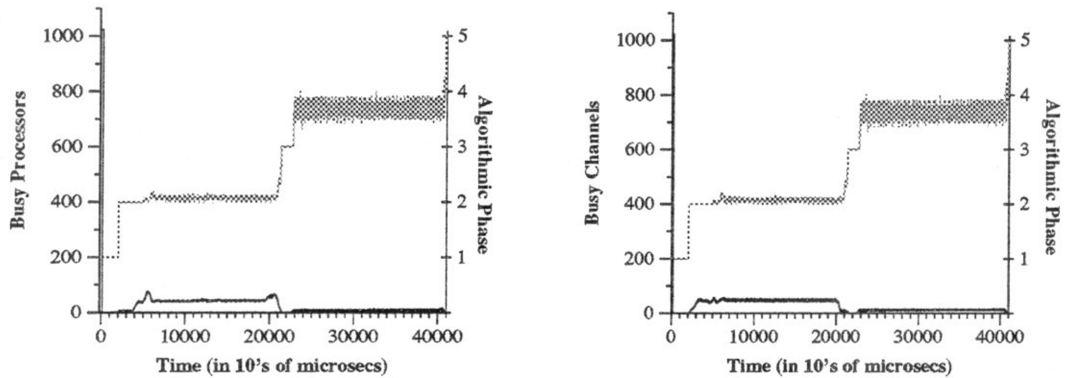

Figure 31: FAST results for 4096 grid-point SIMPLE on a 1024-processor *clique*.

plot is different from that of Figure 28: the second peak in processor activity has disappeared. Moreover, average parallelism in the fourth phase is much smaller than expected. These differences are mainly due to higher communication overhead. For instance, at the end of the first phase, one processor broadcasts the value of the new time-step. According to the blocking communication-model adopted [18, 53], the network interface of this processor sets up channels for outgoing messages sequentially. Messages dispatched later have a greater latency than messages dispatched earlier. Therefore, their arrival times at the receiving processors vary. This variance increases with the number of messages broadcasted, i.e., with the total number of available processors. Consequently, the beginning of the next phase does not occur simultaneously in all processors and hence this phase is "spread" over time. In the 64-processor case, the "spread" is not significant: processors start computing the *Hydro* phase almost synchronously and thus we notice a peak in processor activity at its start. On the other hand, in the 256-processor case, *Hydro* phase does not start simultaneously and there is no peak in processor activity because of it. Similar remarks can be extended to the 1024-processor case. Figure 31 presents a profile for a problem instance of 4096 grid-points mapped onto 1024 processors. Because of higher communication overhead, the parallel-time in the 1024-processor case is greater than parallel-times achieved with fewer processors (e.g., 64 and 256).

Finally, Figure 32 presents *communication patterns* collected with FAST, corresponding to a SIMPLE instance of 4096-point grid, mapped onto a 64-processor clique.

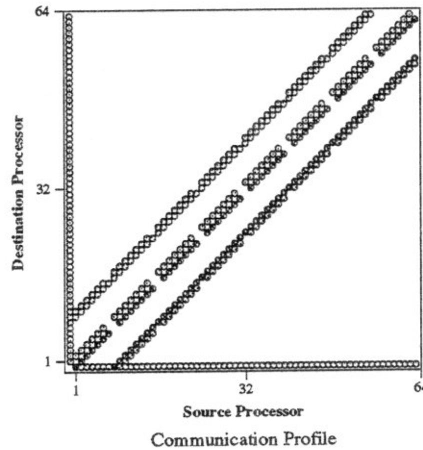

Figure 32: Communication patterns for 64-processor clique, 4096 grid-points SIMPLE instance.

A circular mark in position (i, j) of this diagram signals the existence of messages sent from processor i to processor j. The intensity of the mark is proportional to the number of messages sent from i to j. As expected, except for the broadcasts and aggregates involved in the computation of the global time-step and error, and the synchronization between the two "sweeps" of *Heat* phase, communication is regular and local.

5.2 Scalability Analysis of SIMPLE with FAST

FAST gives us the opportunity to study the scalability of SIMPLE. Figure 33 (left) presents maximum speedups extracted from a large set of FAST-runs simulating different problem and machine sizes. From this diagram we conclude that if there are abundant available resources, parallel executions of SIMPLE can achieve speedups increasing with problem-size. However, the rate of increase of the maximum-attainable speedup decreases with problem-size.

The right diagram in Figure 33 presents speedup curves for cliques (dotted lines) and hypercubes (solid lines) with 16 to 1024 processors. We notice that there is a point in the number of processors after which the speedups saturate. We explore this further, by performing simulations and collecting profile information when all

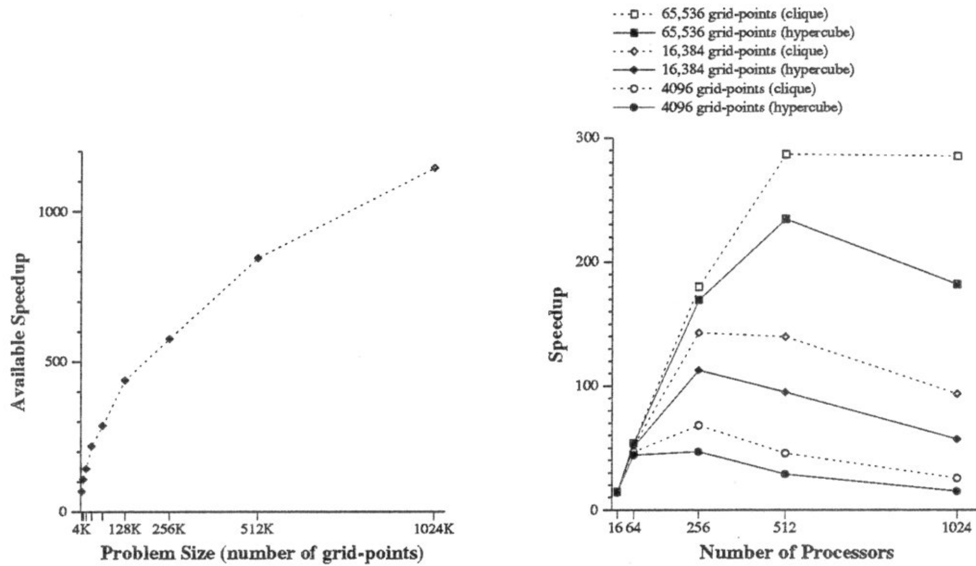

Figure 33: Speedups reported by FAST for SIMPLE executions.

communication delays are turned off. In Figure 34, we present the busy-processor diagram derived from a functional simulation of 4096-point grid SIMPLE-run on a 1024-processor clique with communication latencies set to zero. Parallel time is reduced by 90% with respect to the case of non-zero communication costs (compare this diagram with the left plot of Figure 31 and notice that here we use a different time-scale). From Figure 34 we can see that there are two inherently sequential parts in the algorithm: besides the third phase (Heat), which is largely sequential, there

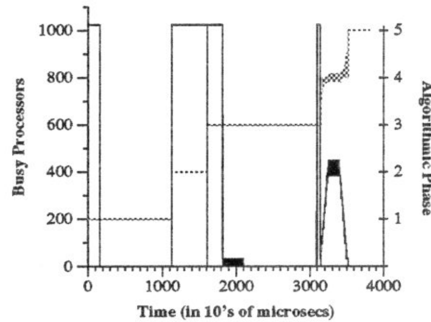

Figure 34: Computation profile of runs on 4096-point grid with zero communication costs (1024 processors).

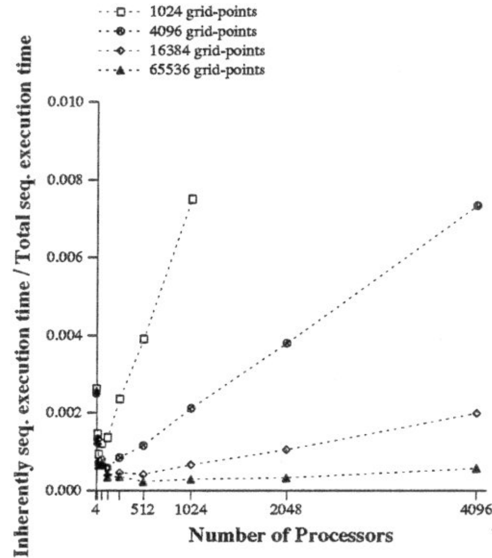

Figure 35: Fraction of sequential execution time that cannot be parallelized.

is a long sequential part in the first phase of SIMPLE (*Delta*). This corresponds to the time when one processor computes the global time-step by taking the minimum over all tentative time-steps computed locally by other processors. Of course, this operation could be split among different processors although this is not clearly a better choice in the presence of communication overhead.

Diagrams like this help us identify the inherently sequential parts of the algorithm, representing the bottlenecks to further improvement of performance according to Amdhal's law. An estimate of the inherently sequential parts of SIMPLE is given in Figure 35. It displays the ratio of the portion of SIMPLE's *parallel execution* that runs sequentially in the absence of communication overhead, over the total sequential execution time. For a given problem size, the non-parallelizable portion of the algorithm increases with the number of processors. According to Amdhal's law, this results in a decrease in the efficiency of parallel executions as processor numbers increase. It represents one of the reasons for the saturation of speedups observed in Figure 33 (right).

To explore this further, in Figure 36 (left) we give a diagram of the upper bound

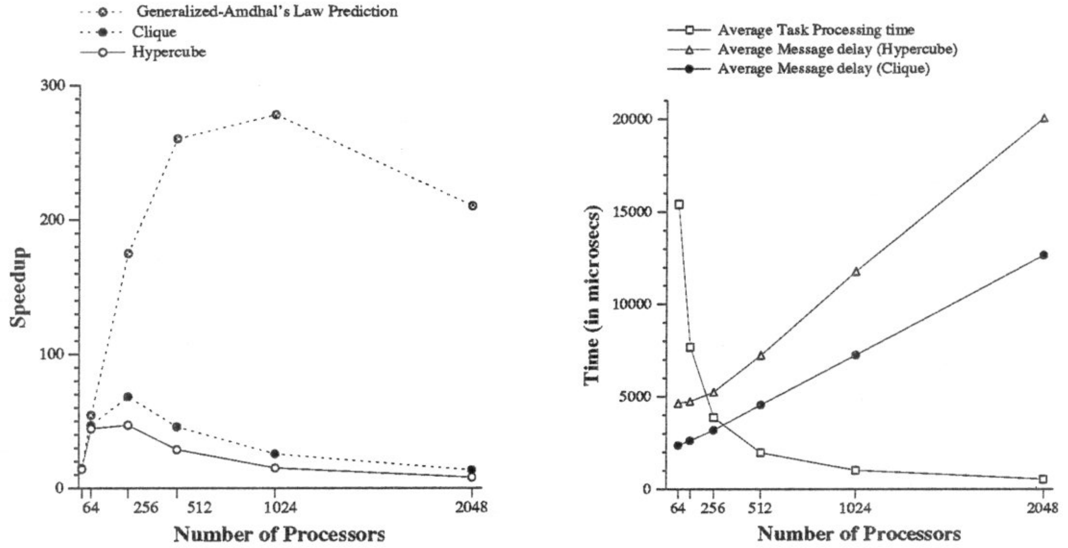

Figure 36: Speedups, average task-execution times and average message delays (4096-point grid).

on speedup derived from a generalized version of Amdhal's law. Also, we give a diagram of speedups reported by FAST for the 4096-point grid case on the clique and the hypercube. The simple version of Amdhal's law, as applied to parallel processing, estimates speedup according to the following equation [22]:

$$speedup \;=\; 1 \;/\; (f \;+\; \frac{(1-f)}{P}) \tag{15}$$

where f is the fraction of the sequential computation time that cannot be parallelized and P is the number of available processors. This equation gives only a rough upper bound on speedup, since it assumes that parallel execution is comprised of a purely sequential part and a parallelizable part during which all available processors are fully utilized. However, as we can easily see from previous diagrams (e.g. Figure 34), this is not the case. Moreover, f is dependent on the problem size and the number of partitions of the grid (that is, the number of available processors). Therefore, we use instead the following generalized version of Amdhal's law to estimate a tighter upper bound on speedup:

$$speedup \;=\; \frac{1}{\sum_1^P f_i/i} \tag{16}$$

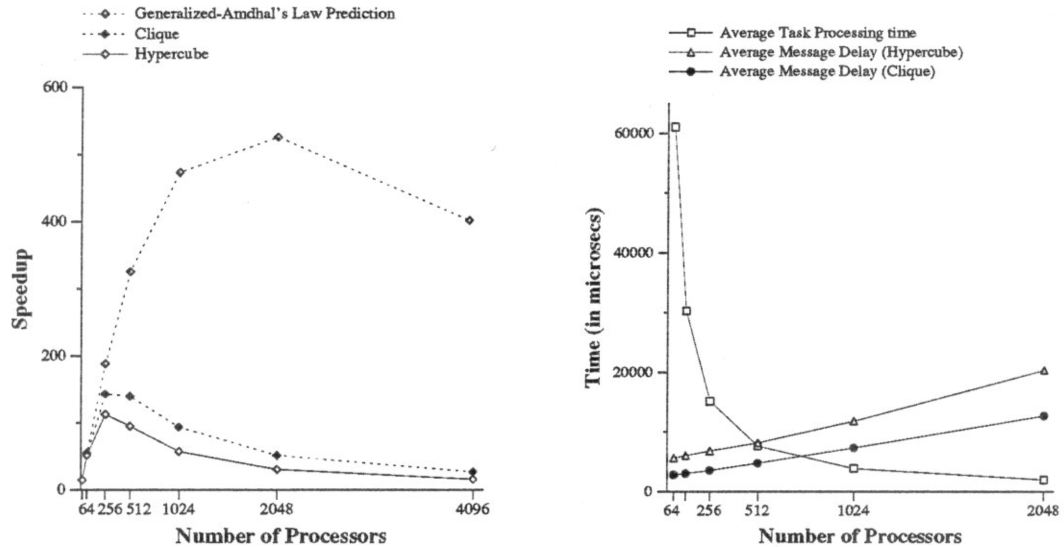

Figure 37: Speedups, average task-execution times and average message delays (16384-point grid).

where f_i is the fraction of the sequential computation time which attains a speedup of i on a P-processor parallel implementation with zero communication costs. f_i can be easily computed by FAST as the product of i times the portion of the parallel execution during which there are exactly i busy processors, over the total sequential execution time. From the plot in Figure 36 (left) we see that the speedup reported by the generalized version of Amdhal's law saturates when the number of processors exceeds 1024. On the other hand, speedups corresponding to functional simulations of cliques and hypercubes saturate for numbers of processors beyond 256. This discrepancy is due to communication costs which have not been accounted for in the Amdhal's law speedups.

The right diagram in Figure 36 displays the change of average task processing time and average message delay with respect to the number of available processors for a 4096-point grid. With the increase in machine-size, task-granularities decrease and average message delays increase, in both clique and hypercube interconnection. From Figure 36 (left) we notice that speedup-saturation occurs at the point where the average communication delay exceeds average task execution time. This remark is confirmed in Figure 37, which presents similar diagrams for a problem size of 16,384

grid-points.

5.3 Some remarks on SIMPLE

Using FAST, we were able to gain further insight into the SIMPLE application by:

1. Collecting detailed data on the variation of computation and communication "intensities" during parallel execution and associating them with the phases of the algorithm.

2. Identifying inherently sequential parts of the algorithm and analyzing their effects on speedup and scalability.

3. Studying the way communication overhead affects performance and scalability.

This insight is important for tuning parallel code for SIMPLE to achieve greater efficiency. It can guide the selection of hardware platforms that will maximize the cost-effectiveness of SIMPLE implementations. Moreover, it gives us a measure of the performance improvements we should expect when investing in communication hardware and software to decrease communication overhead.

Finally, we conclude that we can achieve satisfactory speedups and efficiencies when executing large SIMPLE instances on parallel hypercubes of medium size. As the number of processors increases, attained efficiency drops because of inherently sequential parts in the algorithm. Speedup decreases both because of the sequential parts of the algorithm, and increasing communication overhead.

Chapter 6

A Study of the Fast Multipole Method

In this chapter we present a study of the Fast Multipole method solving the *N-Body* problem. So far, besides theoretical results, there is little experimental information on the behavior of this algorithm. Using FAST, we collect detailed information about its computation and communication profiles and show that it can be implemented efficiently in sparse interconnection topologies, such as multirings.

Computer simulations of many physical phenomena in scientific fields like Molecular Dynamics, Plasma Physics, Astrophysics, and Fluid Dynamics require the solution of the N-Body problem, that is, the evaluation of long-range interactions in electrostatic or gravitational systems of particles (bodies). The N-body problem is of great importance to computational sciences for its wide applicability and for its interesting computational traits, such as, the high volume of numerical calculations involved, the non-uniform structure of the particle data-space, and the inherently unstructured, long-range communication requirements.

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. Consequently, for the large numbers of particles occurring in practical applications, the brute-force method is very slow. Therefore, several people have suggested fast hierarchical approaches for solving the N-body problem [5, 6, 30, 7, 27].

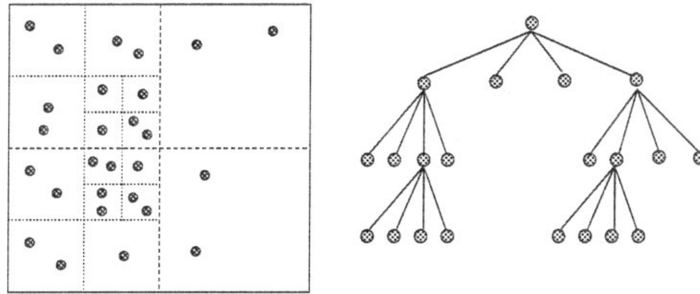

Figure 38: Decomposition of a two-dimensional space of particles and the corresponding quadtree created by the Fast Multipole Method.

One of these approaches is the Fast Multipole Method (FMM) that seeks to solve the N-body problem in two or three dimensions. It does so by approximating the sum of pairwise interactions (electrostatic or gravitational) between each particle and all other particles. The forces are used to compute the new locations that the bodies assume after a small time-step. The FMM evaluates all interactions to within a fixed roundoff error in an average sequential time of $O(N)$ per time-step. 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 38).

For a given input configuration of particles, the sequential FMM first decomposes the data-space in a hierarchy of blocks. It starts with an empty square cell, in the case of a two dimensional problem, or a cubical cell, in the case of a three dimensional problem. The initial cell is large enough to contain all bodies of the system and the algorithm proceeds by loading bodies into it. At any point, if the number of bodies loaded in one cell exceeds a chosen threshold, the cell is subdivided into four square (2-D) or eight cubical (3-D) cells. This process is continued to as high a level as required. The result is a quadtree (2-D) or an octtree (3-D) decomposition of the particle space. Given this decomposition, the algorithm computes a set of necessary neighborhoods involved in subsequent computations. Each block (node) *ibox* of the decomposition tree has three kinds of neighborhoods:

1. The *Close Neighborhood*, which is comprised of *nearest neighbors*, that is, blocks

that share a common border with *ibox*.

2. The *Interaction List*, which is the set of blocks that are children of nearest neighbors of *ibox*'s parent, and do not belong to the *Close Neighborhood* of *ibox*.

3. The *Inverse Interaction List*, that is, the set of blocks whose *Interaction List* includes *ibox*.

The algorithm performs two passes on the tree. The first starts at the leaves with the computation of expansion coefficients and proceeds towards the root accumulating the 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 defined at the beginning. At the end of the downward pass all long-range interactions have been computed and 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. This allows the calculation of new locations of the particles. The algorithm repeats the above steps and simulates the evolution of the particle-system for each successive time-step. Further details can be found in the literature [29, 30]. Depending on the structure of the decomposition tree, the Fast Multipole Method can be described as *adaptive* or *non-adaptive*. In the former, the decomposition tree is unbalanced whereas in the latter it is balanced. The non-adaptive method is applicable in the case where the input particles are uniformly and regularly distributed in space. The adaptive case is more general and applies to non-uniform distributions of particles as well.

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 at the tree-construction phase is not significant [21, 27, 40]. Moreover, parallelism can be exploited only within one time-step. Therefore, we focus our attention in parallelizing the force-computation phase.

We perform functional simulations of an *adaptive* parallel two-dimensional Fast Multipole Method [30]. The principles of the parallel version of the algorithm are

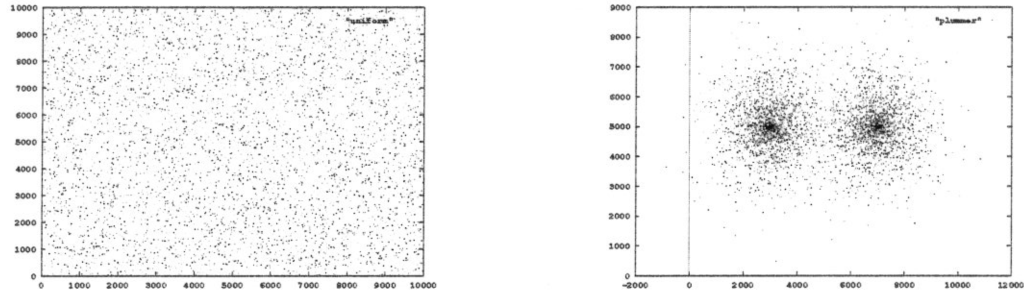

Figure 39: Particle distributions (5000 particles).

incorporated into the front-end of FAST: the parallel algorithm is described in terms of operations performed on the nodes of the quadtree and in terms of information exchanges between them. A similar approach in parallelizing the Multipole Method is described in [36].

6.1 Setup of Functional Algorithm Simulations

6.1.1 N-body problem configurations

We employed two different input configurations of particles for our simulations. One corresponds to an approximately uniform particle distribution, representative of some Molecular Dynamics applications (see Figure 39, left). The second corresponds to non-uniform particle distributions (Plummer), typical of Astrophysics simulations (see Figure 39, right).

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 maximum number of particles per quadtree leaf, that is, the granularity of the hierarchical decomposition. For the specification of the number of coefficients, the tradeoff between running time and desired numerical accuracy should be taken into consideration; it is conceivable that seeking larger series expansions in the FMM computation results in more numerical calculations and longer messages. 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 as compared with the brute-force method.

Experiments with FAST showed that 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-five particles per leaf result in larger sequential tasks and limit the available parallelism. In this chapter, we present results derived when the quadtree granularity was set to 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.

6.1.2 Derivation of the task-flow graph

The description of the Fast Multipole Method, which is hand-coded in the first part of FAST's front-end, generates a quadtree that decomposes the particle-space according to the particle distribution and the quadtree-granularity specified at the input. Subsequently, it computes the various neighborhoods that are defined by the Multipole Method for each node of the quadtree. Then, it produces an Intermediate Representation of the set of operations and message exchanges constituting this computation, instead of performing the actual numerical calculations and data handling.

As an example, in Table 5 we give a description of the computation that the Fast Multipole Method performs upon the leaves of the quadtree. For each leaf *ibox*, the algorithm first computes the multipole expansion coefficients, Φ_i, $i = 1, \cdot, accu$, of the field produced by the particles in *ibox* (step 1 in Table 5). *accu*, is the number of these coefficients, and is given as a parameter to the functional algorithm simulation. This computation is represented by the IR-operation **FORMME**. As soon as the computation of the coefficients is accomplished, each leaf provides its parent

Node [ibox]

1. **FORMME**: Compute the *accu* coefficients for the multipole expansion of *[ibox]*: $\Phi_i(ibox),\ i = 1, \ldots, accu$.
2. If *parent(ibox)* is not the root of the tree
   SEND to *parent(ibox)*:
   - Total charge in *ibox*.
   - $\Phi_i(ibox),\ i = 1, \ldots, accu$.
   - Location z_0 of *ibox*'s center.
3. $\forall$ *jbox*, $jbox \in$ INV_INTER_LIST(*ibox*)
   SEND to *jbox*: $\Phi_i(ibox),\ i = 1, \ldots, accu$.
4. If *parent(ibox)* is not the root of the tree
   RECEIVE from parent: $\tilde{\Psi}(parent(ibox)),\ i = 1, \ldots, accu$
   (multipole expansion coefficients shifted to *ibox*'s center).
5. $\forall$ *jbox*, $jbox \in$ INTER_LIST(*ibox*)
   RECEIVE from *jbox*: $\Phi_i(jbox),\ i = 1, \ldots, accu$.
6. **SHILME**: Shift the coefficients received from *ibox*'s interaction list to the center of *ibox*.
7. **ADDME**: Accumulate the multipole coefficients received from *ibox*'s interaction list and *parent(ibox)*.
8. **EVALME**: Evaluate the potential at the particles of *ibox*, attributed to the fields represented by the multipole expansions computed.
9. **EVALLOCAL**: For each particle in *ibox* compute the potential caused by the other particles in it.
10. $\forall$ *jbox*, such that *jbox* is a nearest neighbor of *ibox*
    SEND to *jbox*: charges and locations of particles within *ibox*.
11. $\forall$ *jbox*, such that *jbox* is a nearest neighbor of *ibox*
    RECEIVE from *jbox*: charges and locations of particles within *jbox*.
12. **EVALNN**: For each particle in *ibox*, compute the potential caused by the interaction with the particles of nearest neighbor boxes.
13. **EVALPOT**: Accumulate direct and far-field terms together for all particles in *ibox*.

Table 5: Description of the operations at quadtree-leaves.

node with the computed coefficients, the total charge (or total weight) of its bodies, and its location in the data-space (step 2). Moreover, it sends the multipole coefficients to nodes that belong in the *Inverse Interaction List* of *ibox* (step 3). Multipole coefficients are forwarded from the leaves towards the root of the tree and merged along the way. When the root is reached, the merging completes and the algorithm starts descending towards the leaves and performing appropriate transformations on the merged coefficients. When *ibox* is reached again, it receives the coefficients $\tilde{\Psi}$ from its parent (step 4). The $\tilde{\Psi}$ coefficients constitute the series expansion of the field generated by particles of the data-space that are "far away" from *ibox*, that is, all particles except those belonging to the nearest-neighbors and the *Interaction List* of *ibox*. Subsequently, *ibox* receives the multipole coefficients from its *Interaction List* (step 5), and then shifts the corresponding series expansion to its center (**SHILME**, step 6). The $\tilde{\Psi}$ coefficients are merged with those received from the *Interaction List* (**ADDME**, step 7). The series expansion thus formed, is used to evaluate the potential (gravitational or electrostatic) at the particles of *ibox*, caused by particles that reside in other blocks of the tree, except the nearest-neighbors of *ibox* (**EVALME**, step 8). The next step computes the potential at each particle of *ibox*, generated by the remaining particles in *ibox* (**EVALLOCAL**, step 9). The computation of the Fast Multipole Method on *ibox* completes with the evaluation of the potential due to nearest-neighbors (steps 10, 11, 12, and 13). In this description, **FORMME**, **SHILME**, **ADDME**, **EVALME**, **EVALLOCAL**, **EVALNN**, and **EVALPOT** are *IR-operations* that correspond to some of the basic computational blocks of the algorithm.

It is clear that the information regarding the quadtree and the neighborhoods of its nodes is sufficient to define the computation of the algorithm on quadtree-leaves for one time-step. The same remark holds for the internal nodes of the quadtree. Table 6 presents the IR for the computation at a leaf of a quadtree, extracted from a small functional algorithm simulation of the FMM. The leaf's id, 031031, determines its position in the quadtree: it is the child of node 03103, at position 1 (see Figure 40). Numbers associated with IR-operations correspond to i860 machine cycles.
In the second part of FAST's front-end, a simple parser generates a task-flow graph from the IR. Figure 41 presents the portion of the task-flow graph which corresponds

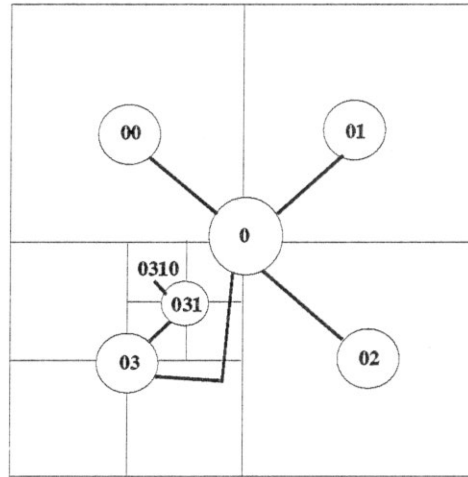

Figure 40: Assignment of id's to quadtree nodes.

to the IR-operations of Table 6. This task-flow graph is subsequently transformed into a parallel-execution graph, as described in Chapter 2, and fed into the back-end of FAST which clusters and then maps the graph onto some message-passing parallel architecture.

6.1.3 Clustering and Mapping heuristics employed

For the functional algorithm simulation of the FMM we experimented with various clustering and mapping heuristics. The heuristics of choice were the ones leading to the shortest parallel execution times as reported by FAST, and requiring at the same time a reasonable amount of processing time to complete. For data presented in subsequent sections we used the clustering technique *GL&S-10%* (see Chapter 3, section 3.5) unless stated otherwise.

Besides the clustering and mapping heuristics of FAST's back-end, in the front-end we implemented a simple ad-hoc technique that partitions the decomposition tree of the particle space into subtrees of equal size. Each available processor is assigned an equal number of subtrees. This technique is similar to the static partitioning heuristics applied in various implementations of the Fast Multipole Method [27, 40]. It is much faster than the automatic partitioning (clustering and mapping) heuristics

Tree Node [031021]	
FORMME:	915
SEND to [03102]:	16 bytes
RECEIVE from [03102]:	80 bytes
RECEIVE from [0312]:	80 bytes
SHILME:	23921.96
ADDME:	350
EVALME:	1348
EVALLOCAL:	40.5
SEND to [031023]:	80 bytes
RECEIVE from [031023]:	80 bytes
EVALNN:	40.5
EVALPOT:	33

Table 6: Fraction of the Intermediate Representation for the FMM.

implemented and achieves competitive parallel execution times for problem instances with uniform distribution of particles. However, as expected, it results in poor parallel performance for highly non-uniform distributions. Nevertheless, similar ad-hoc techniques can be applied to non-uniform distributions easily [68].

6.1.4 Hardware parameters adopted

For hardware parameters, we used values derived from Intel's iPSC/860 interconnection network and the 40Mhz, i860 microprocessor [38]. We measured the speed and overhead of the blocking communication primitives *csend* and *crecv* for various message sizes and expressed them in terms of i860 machine cycles. Furthermore, cycles-per-instruction figures from the i860 manual were used to estimate the timing of operations. We did not take into consideration potential instruction pipelining. Time is expressed in terms of i860 machine cycles.

These hardware parameters characterize an architecture with very fast processors, fast communication links, and high message-initialization and message-receive overheads. But, with FAST it is very easy to experiment with different hardware setups as, for instance, those with low message initialization/receive overheads, faster links,

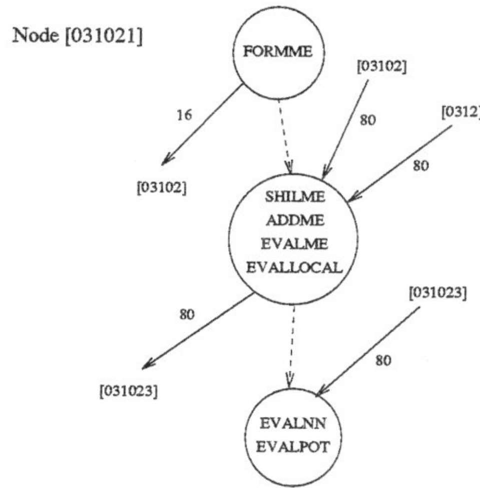

Figure 41: Task-flow subgraph (FMM).

etc.

As mentioned in Chapter 2, the current implementation of FAST provides a few different interconnection schemes available for simulation: a clique, a single ring, multirings, and a binary hypercube.

6.2 Remarks on the ad-hoc partitioning

We ran a large number of functional simulations to test the ad-hoc partitioning technique. For each run the number of partitions was set equal to the number of available processors and therefore FAST's clustering and mapping stages were bypassed. Figure 42 (left) presents parallel times derived with FAST for the case where the partitioned problem was mapped on clique interconnections.

According to a theoretical analysis presented in [27], the complexity of the parallel algorithm running on uniformly distributed particle-space is :

$$O(N/p) \; + \; b\log_4 p \; + \; c(N,p)$$

where N is the number of particles, p is the number of processors, and $c(N,p)$ is a lower order term which includes things like the communication and synchronization overhead. For a constant $p << N$ the complexity becomes $O(N)$. Simulation results

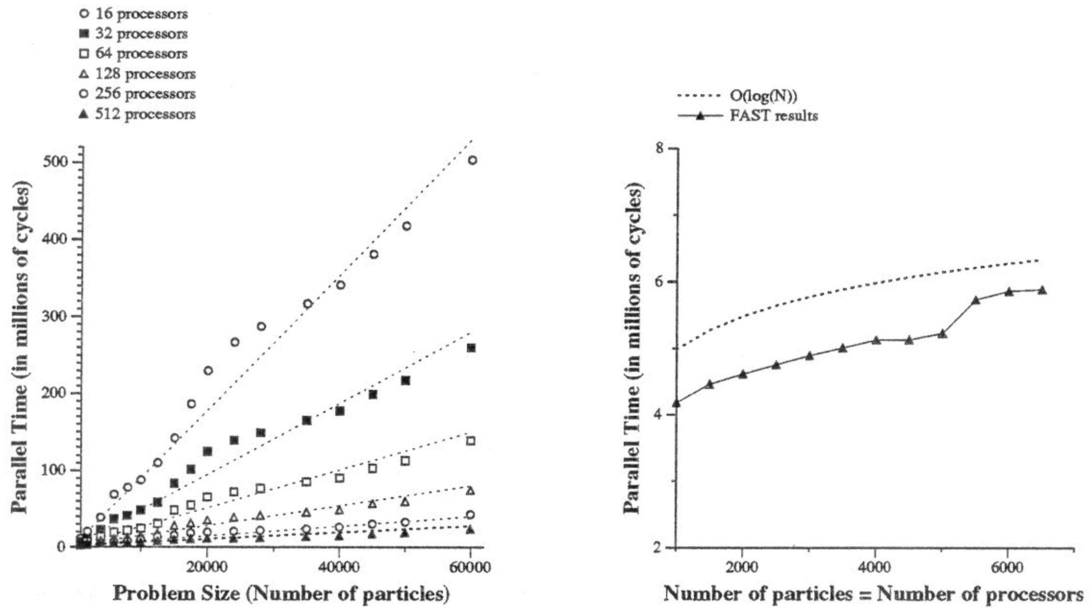

Figure 42: Parallel Times derived with FAST for the FMM mapped on clique interconnections (uniform distributions, ad-hoc partitioning).

over a wide range of problem sizes presented in Figure 42 (left) concur with this theoretical prediction.

When the number of processors is $O(N)$ the complexity of the parallel algorithm becomes $O(\log N)$. In Figure 42 (right) we present a plot of parallel times reported by FAST for problem instances of 1000 to 6500 uniformly distributed particles, mapped onto cliques with 1000 to 6500 processors (that is, one particle per processor). Although this diagram could not possibly correspond to a practical implementation, it gives us the opportunity to compare and confirm FAST's results with theoretical predictions on the parallel Fast Multipole Method.

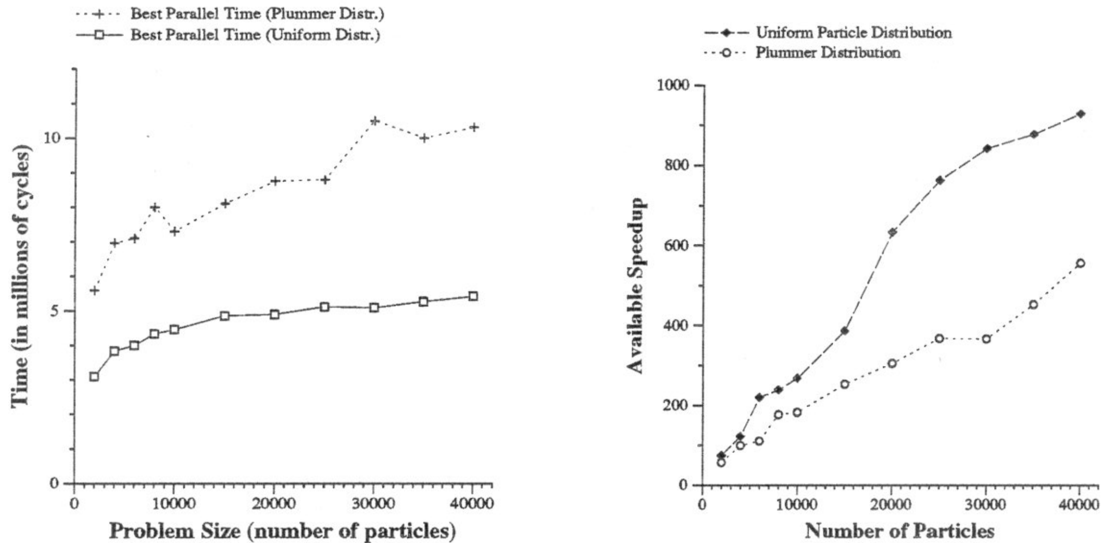

Figure 43: FAST results about performance of the parallel FMM on an *abundant clique*.

6.3 Profiling the FMM

6.3.1 Remarks on the Scalability of the FMM

The clustering stage of FAST (see Figure 5) 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 (*"abundant clique"*). We can calculate the parallel time of the parallel-execution graph and thus measure the speedups achieved on the abundant clique. In Figure 43 (left) we present parallel execution times for problems of 2,000 to 40,000 particles distributed according to uniform and Plummer distributions. The times in this diagram correspond to the minimum estimates from a set of FAST experiments with various quadtree granularities (numbers of particles per quadtree leaves). No partitioning in the fist phase of the Front-End was performed on the quadtrees. Instead, clustering was employed on the parallel-execution graphs to minimize communication overhead.

Furthermore, in Figure 43 (right) we present *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

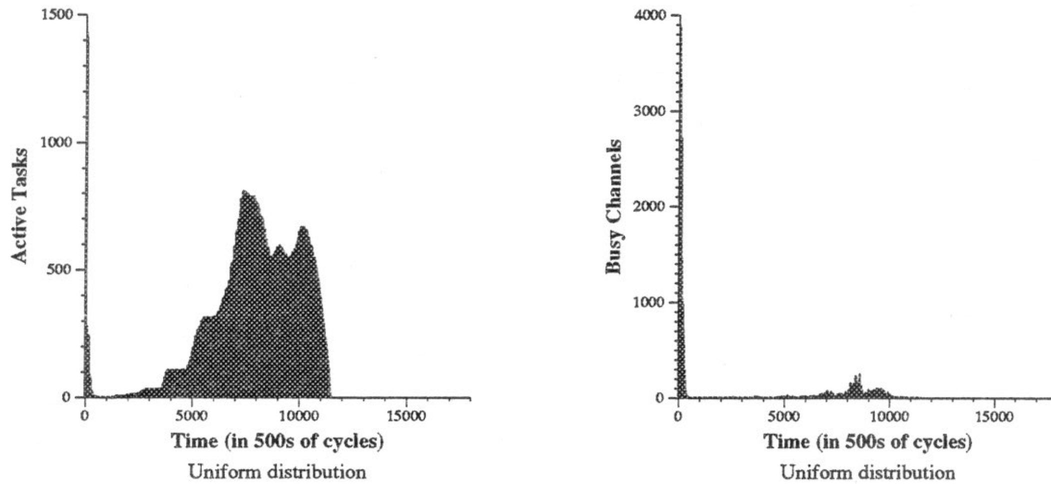

Figure 44: Execution Profiles for a 15,000-particles problem (Uniform distribution).

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. This can be verified from Figure 43. The parallel algorithm would not be scalable if speedups leveled off for larger problem sizes; this would signal the existence of significant sequential parts in the algorithm, rapidly increasing with the problem size. 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 "narrower" decomposition trees and thus in less available parallelism.

6.3.2 Profiles of parallel execution

From the parallel-execution graph we can also extract a more detailed profile of the algorithm studied, consisting of the numbers of active tasks and busy links during parallel execution in the abundant clique. This profile depends on the method by which the front-end constructs the graph, the configuration of the input data-space at hand, and the clustering heuristic adopted. Nevertheless, the task-graph generation is defined according to the inherent communication granularity of the algorithm.

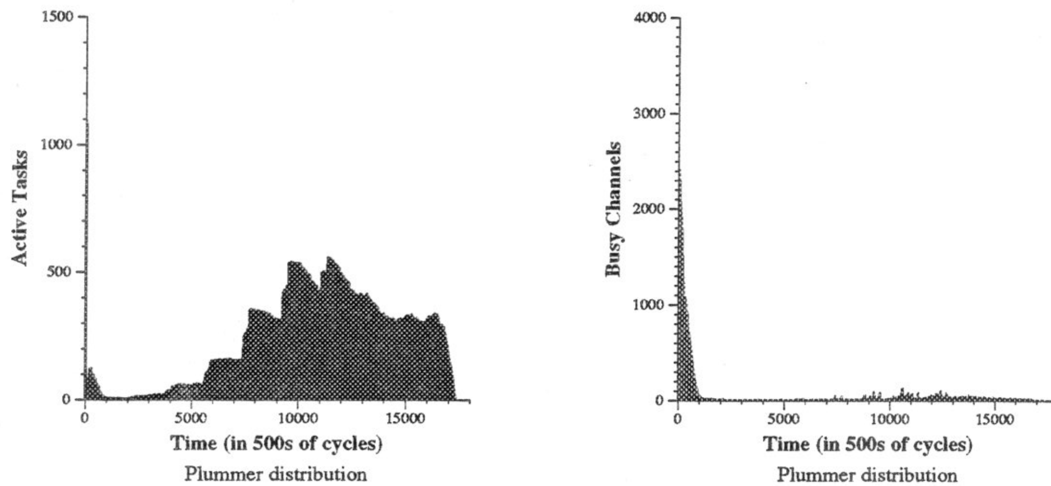

Figure 45: Execution Profiles for a 15,000-particles problem (Plummer distribution).

Moreover, since the goal of clustering is to "prune" unnecessary communications without restricting the available parallelism, the choice of clustering heuristic should not drastically alter the profile information. Indeed, experiments with different clustering heuristics led to similar results and conclusions. Also, the profile is independent of the coarser-grain problem partitioning and the parallel architecture on which the algorithm will be executed. Therefore, the profile reveals characteristics inherent to the algorithm at hand and is not influenced by partitioning and mapping to a specific multiprocessor. Figure 44 presents such a profile for a parallel execution of the Fast Multipole Method on a problem instance with 15,000 uniformly distributed particles, fifteen particles per quadtree-leaf, and a ten coefficient approximation.

Similarly, Figure 45 gives the profile in the case where FMM is used on 15,000 particles distributed according to the non-uniform Plummer distribution. In both cases we have used the same clustering technique, *GL&S-10%*, which is a combination of heuristics presented in references [24, 24]. As we can easily see, the profiles have basically the same shape and reveal common characteristics of the different parallel executions.

In Figure 46 we give a profile of the parallel execution generated by FAST for a problem of 15,000 uniformly distributed particles. In this case we did not use any clustering or mapping heuristics. Instead, the quadtree was statically partitioned into

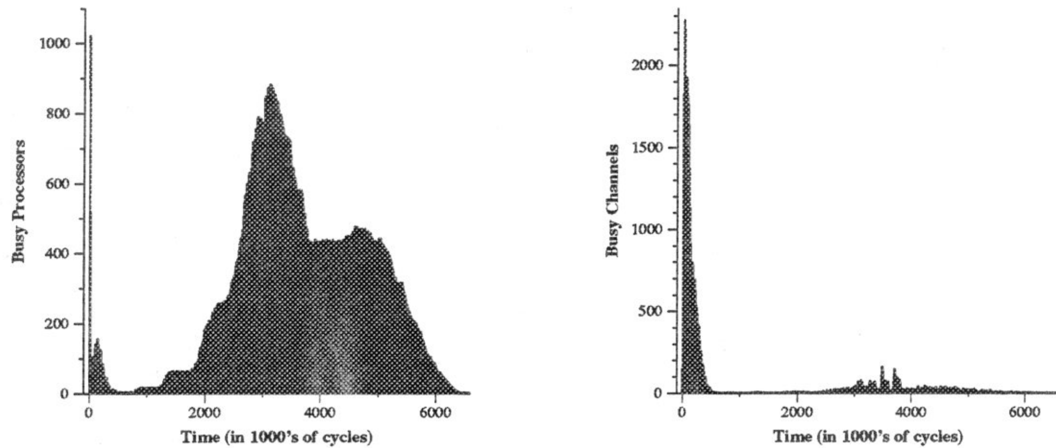

Figure 46: Execution Profiles for a 15,000-particles problem partitioned into 1024 blocks (Uniform distribution).

1024 blocks. Tasks corresponding to these blocks were mapped onto a 1024-processor clique. This profile is similar to the ones extracted from clustered parallel-execution graphs mapped onto abundant cliques.

6.3.3 Phases of the Fast Multipole Method

From the plots above it becomes clear that the parallel execution has three phases: at the beginning there is a short phase 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.

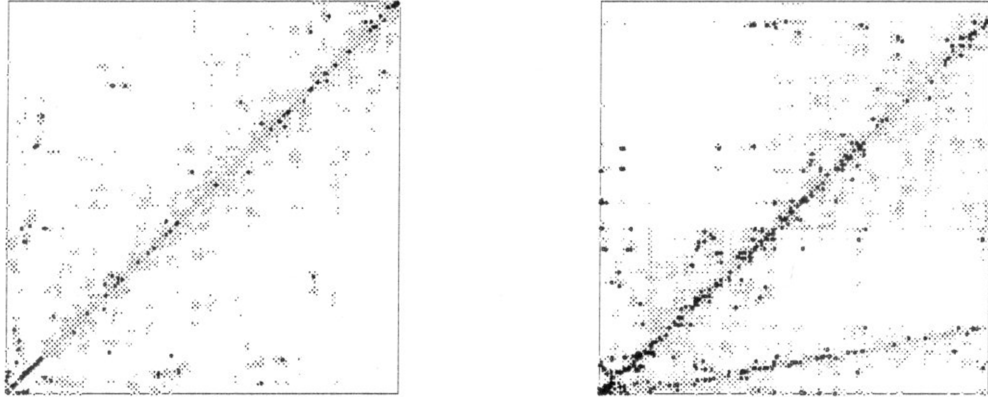

Figure 47: Communication Patterns for the 15,000-particle problem.

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.

6.3.4 Resource requirements

Another interesting observation from these profiles relates to resource utilization of the abundant clique. For instance, the parallel-execution graph of the above example in the uniform particle-distribution case has 2384 clusters and therefore, the corresponding abundant clique would have 2384 processors and 5,681,072 unidirectional links. However, as we can infer from the execution profiles, even if we could build such a multiprocessor most of its resources would remain unused for most of the parallel execution time. In this example, the average number of busy processors over the parallel execution time is 308, that is, a 1.29% of the processors in the abundant clique. In contrast, the average number of used links over time is as low as 51, which corresponds to the 0.08% of the total number of links. Therefore, it is conceivable that a much sparser architecture with fewer processors or links could achieve essentially the same speedups as the abundant clique.

This can be seen also from Figure 47 which shows the communication patterns for the above example: the plot to the left presents communication patterns for the case

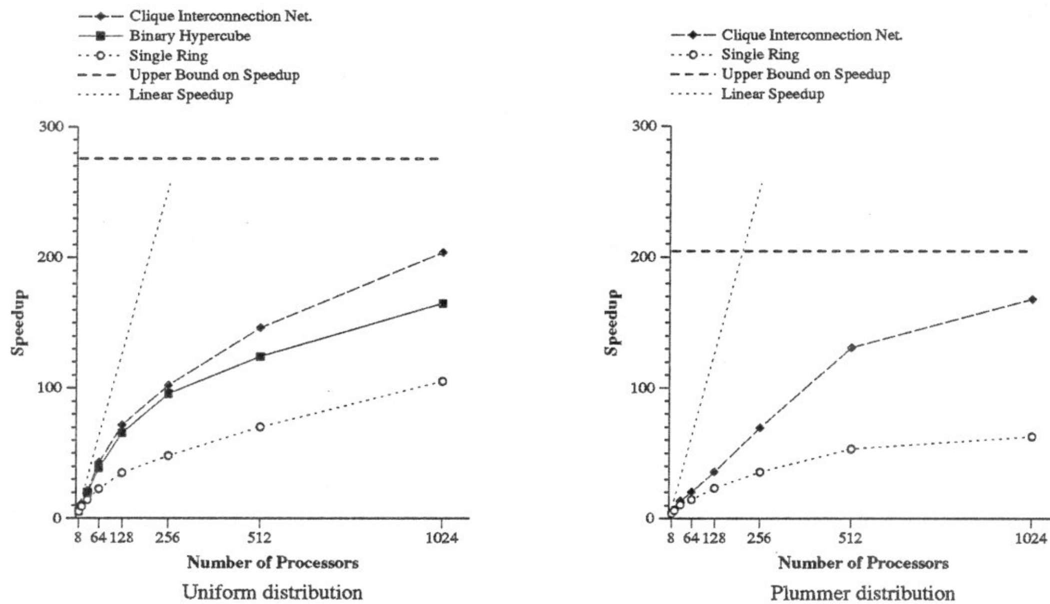

Figure 48: Speedups for a 10,000-particle problem.

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. The particular task-cluster numbering scheme used has better localizing characteristics than a random one.

6.4 Performance of the Fast Multipole Method on a Ring Multiprocessor

Communication patterns presented in Figure 47 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 (see Figure 47). This observation suggests that a ring interconnection might achieve speedups comparable to those achieved on a clique. We mapped the clustered parallel-execution

graphs onto ring interconnections with various numbers of processors and studied ring performance for the parallel Fast Multipole Method. We also evaluated the performance of clique and binary hypercube interconnections with equal numbers of processors in order to assess the relative effectiveness of ring implementations.

As an example, we present in Figure 48 the speedups measured by our system for a problem instance of 10,000 particles, fifteen particles per quadtree leaf, and a ten coefficient approximation. Each plot also includes the upper bound on speedup for this problem size and the respective particle distribution. The upper bound on speedup corresponds to executions on the abundant clique when all communication delays and overheads are set to zero. From these diagrams we see that in a 128-processor machine, the ring interconnection 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. Therefore we conclude that ring interconnections represent a cost-effective architectural choice for implementing the Fast Multipole Method in parallel. The cost-effectiveness of the rings becomes more obvious if we take into account the relative costs of rings and cliques having equal numbers of processors. Even with state-of-the-art technology, the implementation complexity of medium- or large-size cliques is prohibitive. In comparison, rings are cheap and easily scalable.

Speedups measured on rings are not as high as the ones achieved on cliques or hypercubes, 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 49 which display the average message delay and message congestion measured with FAST for the 10,000-particle example (Plummer distribution). Congestion 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 results from simulations on uniform distributions are similar.

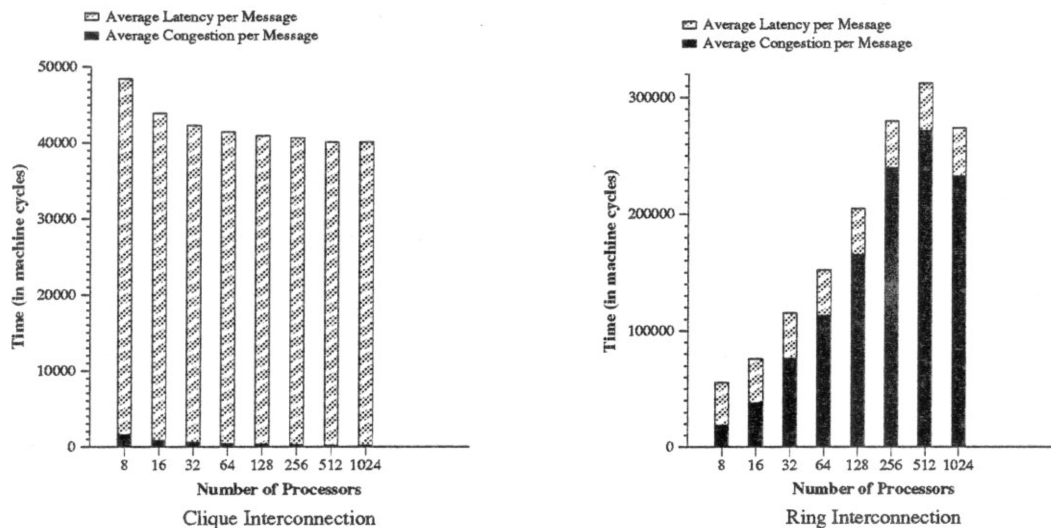

Figure 49: Contribution of Congestion to Message Latency for a 10,000-particles problem (Plummer distribution). Notice the difference in the time-scale of the two diagrams.

6.5 Performance of the Fast Multipole Method on Multirings

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 with the same number of processors. We performed a number of functional simulations over a wide range of problem instances (from 2,000 to 10,000 particles) to examine the performance and cost-effectiveness of multirings. In this section, we present results derived from a representative simulation of 10,000 particles.

Our functional simulations proved that multirings are also cost-effective: Figure 50 presents the speedups reported by FAST for the 10,000-particle problem mapped onto multirings with two to sixteen rings. Figure 51 gives the ratios of parallel times measured for the problem instance above, on cliques and multirings with the

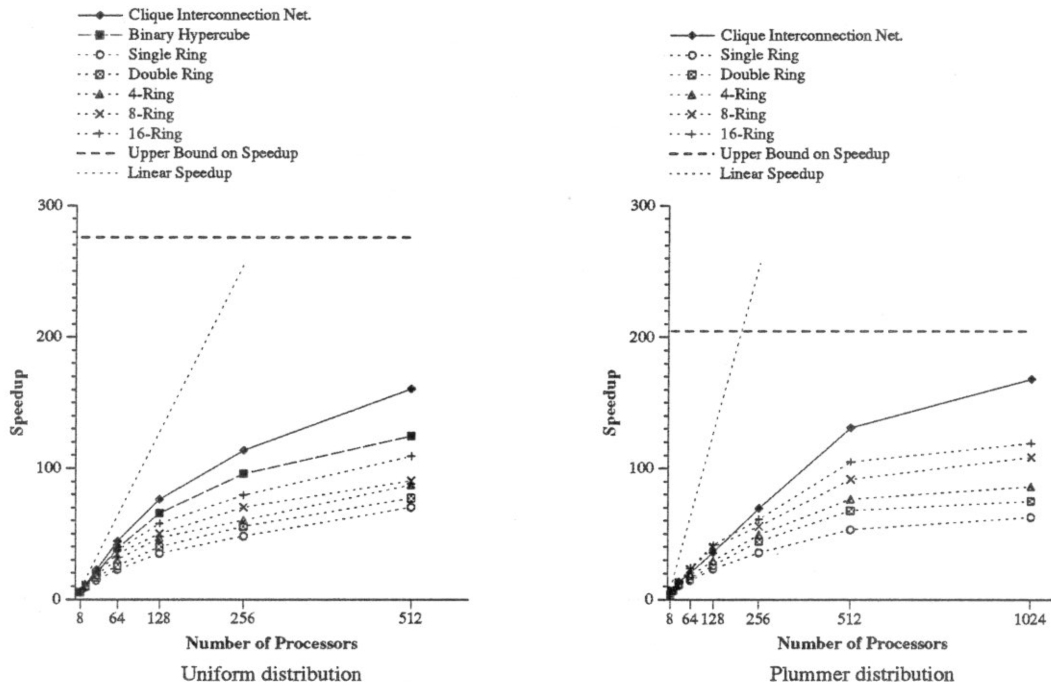

Figure 50: Multiring Performance.

same number of processors. This is used as a measure of the efficiency of multiring interconnections: a ratio equal to one indicates that the multiring structure achieves the same performance as a clique with an equal number of processors.

It is clear that an increase in the number of rings significantly improves 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 4-ring achieves 83% of the speedup of the clique with only 6.3% of its links. As another example, the 512-processor 4-ring achieves a speedup slightly larger than the one attained by a 256-processor clique. Although it has twice as many processors as the clique, it is feasible to implement it with state-of-the-art technology or to embed it in an existing multiprocessor. In contrast, the implementation of the 256-processor clique is not feasible.

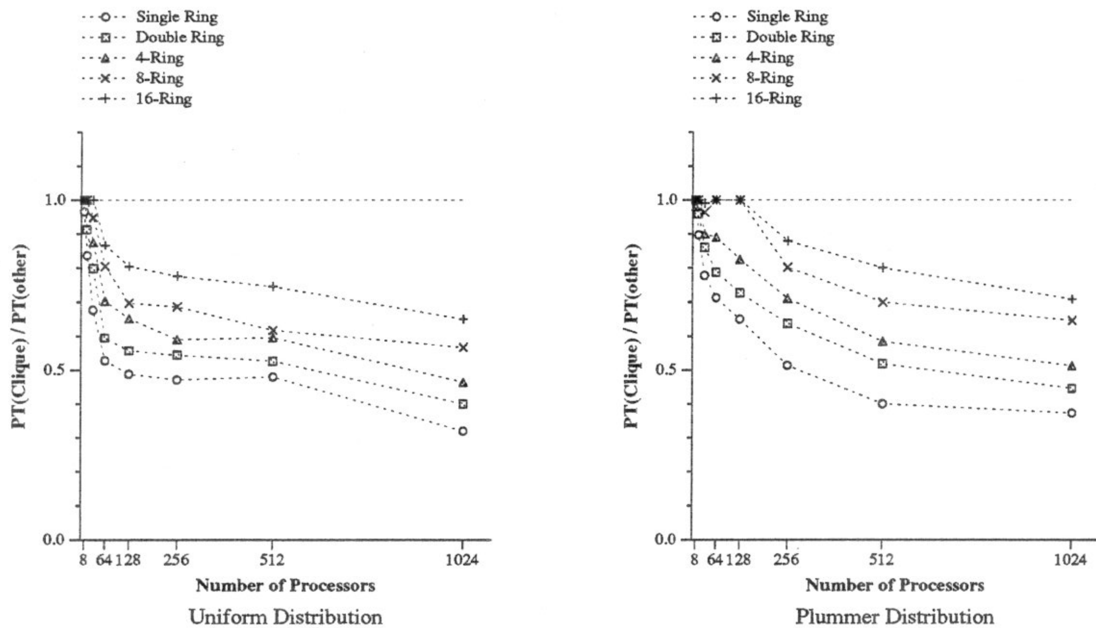

Figure 51: Efficiency of Multiring interconnections.

6.6 Some remarks on the FMM

We used FAST to perform a study of the parallel Fast Multipole Method and collect performance measurements, and computation and communication profiles over a wide range of problem sizes corresponding to practical cases. Using these data we were able to better understand the characteristics and limitations of the algorithm and derive conclusions about its scalability and its resource requirements. Needless to say that it would be much more difficult and expensive to collect such information from trace-driven simulations or monitoring of real executions.

We concluded that the algorithm is scalable in the sense that as the problem size increases, considerably higher speedups can be achieved if abundant processors and communication links are available. However, speedup values for clique interconnections are small with respect to "linear" speedup. This is a result of the hierarchical nature of the FMM which performs top-down and bottom-up visits of a quadtree data-structure, computing data on tree-nodes; the potential for parallelism is very small during the processing of tree-nodes near the root of the tree.

Communication traffic is irregular; all experiments showed a large burst in message transmissions at the beginning of the parallel execution and relatively few transmissions afterwards. Communication patterns display the existence of communication locality between processors: each processor interacts through message transmission and receipt only with a very small set of processors.

FAST allowed us to estimate the performance of the parallel implementation of the FMM on message-passing multiprocessors with hypercube, ring and multiring interconnection networks. Performance figures derived from the mapping of the FMM on cliques were used to evaluate the effectiveness of the hypercube, ring and multiring topologies. Our simulations showed that an implementation of the Multipole algorithm on scalable ring or multiring architectures is cost-effective and attractively simple. Hypercube implementations proved to be very effective since their performance measurements were almost identical to the ones derived for the cliques.

Chapter 7

A Study of the Barnes-Hut Algorithm

In this chapter we present a modified version of the Barnes-Hut algorithm for solving the N-Body problem, and evaluate its behavior with FAST. The Barnes-Hut algorithm is a popular method for solving the N-body problem in two or three dimensions [30]. At each time-step of the N-body simulation, the algorithm constructs a hierarchical tree, decomposing the particle space into blocks, as in the Fast Multipole Method. Each node of the tree maintains the total mass (or charge) of the corresponding block and the position of its center-of-mass (or center-of-charge). Having constructed the decomposition tree, the force on any particle p is approximated by a simple recursive calculation. The calculation starts at the root of the tree, which contains the entire set of particles, and recursively visits the blocks of the decomposition. If l is the length of a block, *ibox*, under consideration, and D is the distance from the block's center-of-mass to p, the algorithm compares l/D with θ, where $\theta \approx 1$ is a fixed accuracy parameter. Particle p is considered to be "well separated" from *ibox*, if:

$$l/D < \theta. \tag{17}$$

In that case, the force applied to p from the particles of *ibox* is approximated by the force that would be applied to p from a mass (charge) equal to the total mass (charge) of the block's particles and placed at the block's center-of-mass. If $l/D \geq \theta$,

the calculation continues one level down and recursively examines the children of *ibox*, with respect to p.

The number of interactions for each particle is, on the average, on the order of $\log N$ for large N, assuming a homogeneous distribution of particles in space. Therefore, the average complexity of the algorithm is $O(N \cdot \log N)$ [6]. If θ is set to zero, then the algorithm becomes identical with the brute-force $O(N^2)$ method.

Parallel algorithms for the Barnes-Hut technique usually partition the data-space into a number of subsets of particles equal to the number of available processors, and assign each partition to a different processor [61, 66, 67]. The methods used for partitioning seek to improve the load-balancing of parallel executions [61, 68]. If the decomposition trees are too large to fit in the memory of a single processor, special precautions are taken to distribute the appropriate parts of the tree to processors [31, 47, 61, 66, 68].

In contrast with the cases of the Fast Multipole Method and SIMPLE, constructing the computational domain for a Barnes-Hut instance is not sufficient to predict the set of computations and communications performed during parallel execution of that instance. To predict this set, it is necessary to compare the location of each particle with respect to the appropriate decomposition blocks, according to the Separation criterion (17). Such comparisons, however, require the execution of a large portion of the algorithm. This contradicts the assumptions and goals of Functional Algorithm Simulation. Nevertheless, we can reorganize the algorithm so it can be studied with FAST.

7.1 A modified version of the Barnes-Hut algorithm

A number of different implementations of the Barnes-Hut technique seek to improve its performance on vector and data-parallel machines by regarding the tree-walk used to compute long-range forces as a device for establishing interaction lists for every particle [31, 47, 77]. Each of these lists includes a collection of blocks and particles

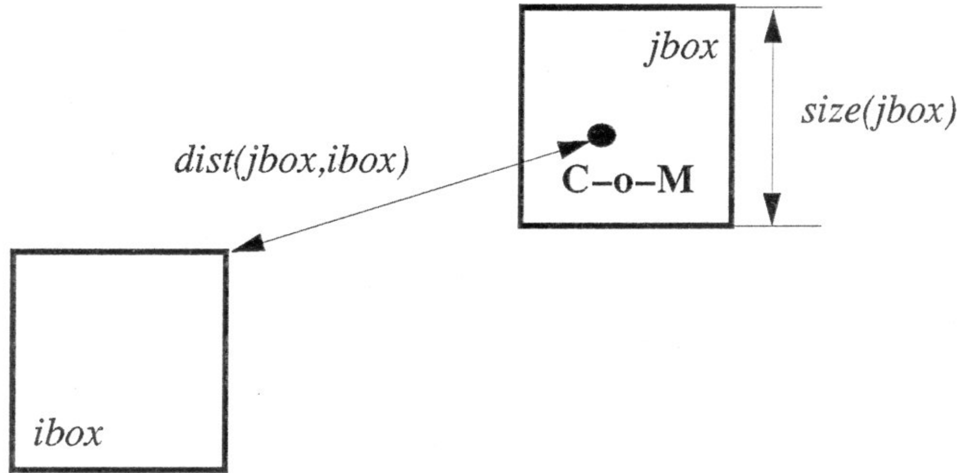

Figure 52: Distance-definitions for neighborhood calculation.

interacting with the corresponding particle. The interaction lists are constructed, either for single particles by walking through the tree from node to node [31], or for groups of particles which are close together in space [7].

In our study, we consider a combination of these techniques for two-dimensional problems. Each internal quadtree-node *ibox*, has the following interaction lists:

1. The *Distant Interaction List (DIL)*, that is, the set of nodes *jbox* which are located at the same level of the quadtree as *ibox*, and satisfy the Separation criterion:
$$\frac{size(jbox)}{dist(jbox, ibox)} < \theta. \tag{18}$$
size(jbox) is the length of *jbox*'s side, *dist(jbox, ibox)* is the distance between the center-of-mass of *jbox* and the perimeter of *ibox* (see Figure 52), and θ is the accuracy parameter of Barnes-Hut.

2. The *Close Interaction List (CIL)*. This is the set of nodes that belong to the same level of the quadtree as *ibox*, and *do not* satisfy Criterion (18).

Each *leaf* of the quadtree has the following interaction lists:

1. The *Distant Interaction List (DIL)*, that is, the set of nodes *jbox* which are located at the same or higher level of the quadtree as *ibox*, and satisfy the

Separation criterion (18).

2. The *Nearest Neighbor Interaction List (NNIL)*. This list includes quadtree *leaves* that share a common border with *ibox*. Also, it includes leaves that do not share a common border with *ibox* but are close to it, according to Criterion (18).

To compute the Distant and the Close Interaction Lists of an *internal* node *ibox* we consider the Close Interaction List of *ibox*'s parent, *CIL(parent(ibox))*, and create a list of nodes *CDIL(ibox)* such that:

$$CDIL(ibox) \;=\; N_i \cup N_l,$$

where:

$$N_i = \{u \mid parent(u) \in CIL(parent(ibox))\},$$

and

$$N_l = \{v \mid v \in CIL(parent(ibox)),\ and\ v\ is\ a\ quadtree\ leaf\}.$$

In other words, *CDIL(ibox)* includes the children of nodes that belong to the Close Interaction List of *ibox*'s parent. Also, it includes the nodes that belong to the *CIL(parent(ibox))* and do not have any descendants, that is, the quadtree leaves of *CIL(parent(ibox))*. We estimate the separation between *ibox* and each node of *CDIL(ibox)* using Criterion (18). Nodes close to *ibox* are assigned to *CIL(ibox)*, together with *ibox*'s siblings. Nodes well separated from *ibox* constitute its Distant Interaction List.

We follow the same procedure to compute the Distant and the Nearest Neighbor Interaction Lists for each quadtree leaf *ibox*. Quadtree nodes well separated from *ibox* are assigned to its Distant Interaction List. Quadtree leaves that are close to *ibox* according to Criterion (18) are assigned to $NNIL(ibox)$. For internal nodes that are near to *ibox*, we examine recursively the location of their children with respect to *ibox*, and update accordingly *ibox*'s Interaction Lists.

DIL(ibox) is used for the approximation of interactions between blocks of particles that are far away according to the Separation criterion (18). For example, assume that *jbox* belongs to *DIL(ibox)*. Then the sum of the forces exerted on the particles

of *ibox* by the particles of *jbox*, is approximated by the force that would be applied on these particles by a body located at the center-of-mass of *jbox*, with mass (charge) equal to the total mass (charge) inside *jbox* (see Figure 52).

The force-approximations from all the members of *DIL(ibox)* are added together, and the result is used to compute $F_{dist}(ibox)$. This represents the approximation of interactions between the particles of *ibox* and particles of nodes that are well separated from *ibox*, *or* from its ancestor nodes in the quadtree. In other words, to calculate $F_{dist}(ibox)$, we accumulate the approximations of interactions from *DIL(ibox)*, and add the result to $F_{dist}(parent(ibox))$.

With these definitions in mind, we present a description of the modified version of Barnes-Hut used in our study:

Modified Barnes-Hut

1. Construct the quadtree.

2. Start at the leaves of the tree and move towards the root, computing the center-of-mass of each block of the decomposition by combining the centers-of-mass of its children blocks.

3. From the root of the tree move toward the leaves, computing the *Distant*, *Close*, and *Direct* Interaction Lists, and the F_{dist} values of quadtree nodes.

4. Compute the interactions between every particle in every quadtree leaf, and the remaining particles belonging either to the same leaf, or to members of the leaf's Direct Interaction List.

5. Compute the interaction between each particle in every quadtree leaf *ibox*, and the particles of nodes well separated from *ibox* by applying force $F_{dist}(ibox)$ to the particles of *ibox*.

6. Estimate the new positions of particles and go to step 1.

Generating the computational domain for the modified version of Barnes-Hut and calculating the interaction lists presented earlier, enables us to predict the set of

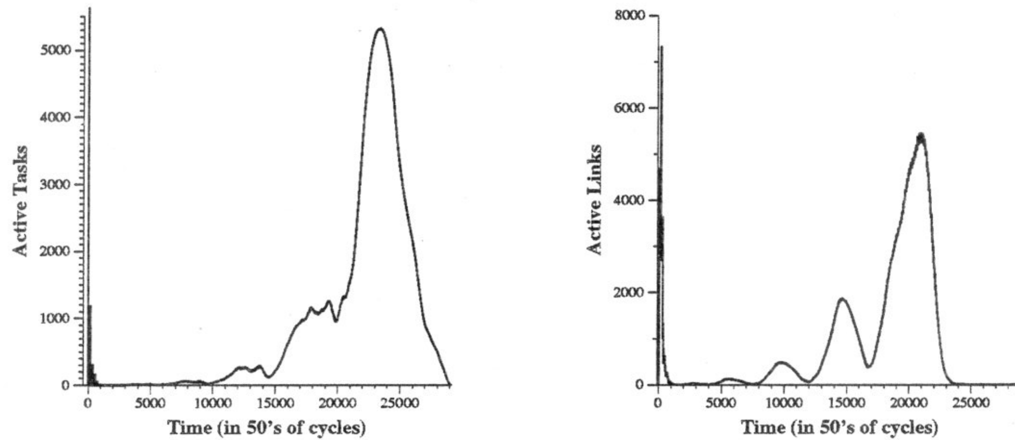

Figure 53: Execution Profiles for a 10,000-particle problem mapped on an abundant clique (Uniform distribution).

computations and communications that take place in a parallel execution of this algorithm. This enables us to study this version of the algorithm using Functional Algorithm Simulation.

7.2 Experiments with the Barnes-Hut algorithm

In our experiments with FAST we employed the same hardware parameters as in the case-study of the Fast Multipole method, that is, parameters derived from Intel's iPSC/860 interconnection network and the 40Mhz, i860 microprocessor [38].

The accuracy parameter θ was set to 0.6 or 0.9. Experimentation has shown that forces computed with an accuracy parameter as large as $\theta = 1$ are still accurate to approximately 1% [6]. Therefore, values of θ equal to 0.6 and 0.9 are expected to correspond to calculations of high accuracy. Clearly, the smaller the value of θ, the more accurate and expensive the Barnes-Hut calculation is. The granularity of the quadtrees was set to a maximum of 3 particles per quadtree leaf.

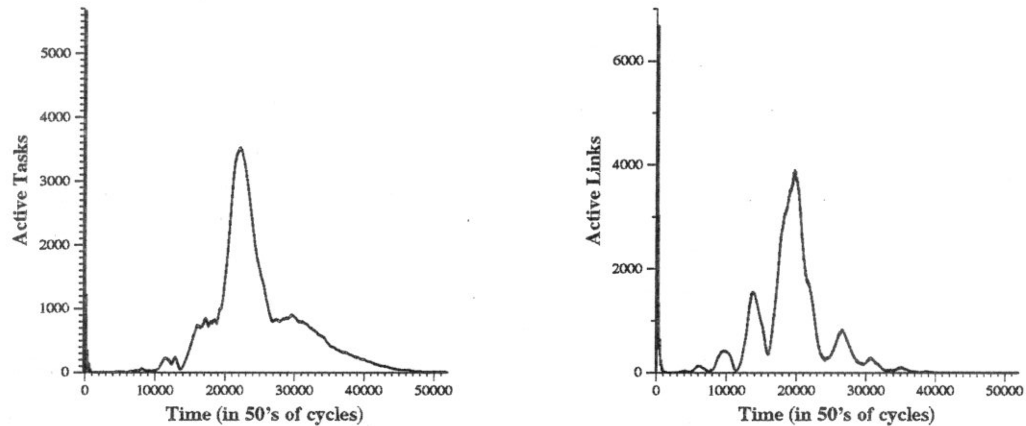

Figure 54: Execution Profiles for a 10,000-particle problem mapped on an abundant clique (Plummer distribution).

7.2.1 Profiles of parallel execution

From the parallel-execution graphs generated by FAST we can easily extract profiles characterizing the computations and communications of the Barnes-Hut technique. The profiles presented in this section correspond to parallel-execution graphs which have been clustered with the *GL* heuristic (see Chapter 3) and then mapped on an abundant clique. Figure 53 (left) shows the active tasks derived from a functional algorithm simulation of a problem instance with 10,000 particles distributed uniformly, with $\theta = 0.6$. Figure 53 (right) presents the diagram of active links. Notice the different scale of the two diagrams.

Diagrams in Figure 54 present the profiles of a parallel execution of the modified Barnes-Hut technique mapped on an abundant clique, for 10,000 particles distributed according to the non-uniform Plummer distribution (see Figure 39 in Chapter 6). The accuracy parameter was set to 0.6.

We see from the plots in Figures 53 and 54 that the parallel executions on the abundant clique start with a large number of active tasks having very short execution times. In both cases, the number of these tasks is equal to the number of leaves of the quadtree. Each of the tasks computes the center-of-mass of a quadtree-leaf. After computing the centers-of-mass of the leaves, the algorithm proceeds towards the root of the quadtree computing centers-of-mass of internal nodes. This explains the fast

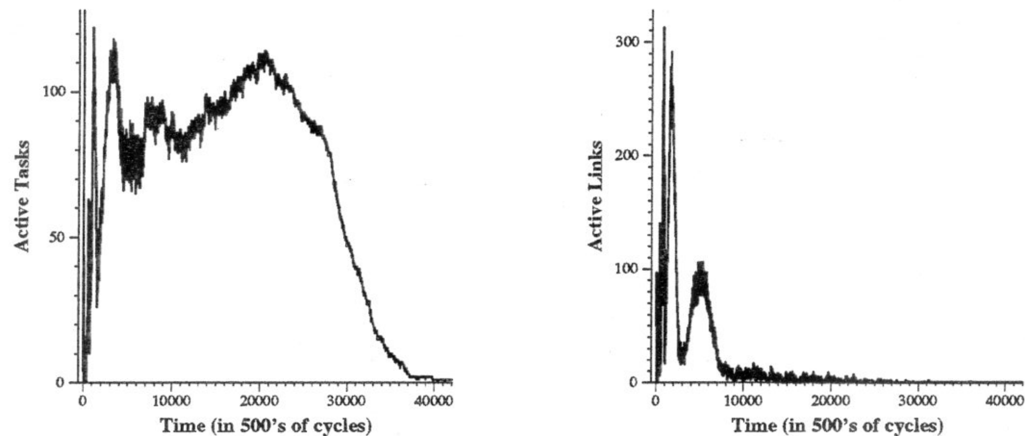

Figure 55: Execution Profiles for a 10,000-particle problem mapped on a 128-processor clique (Uniform particle distribution).

decrease in the number of active tasks: at each step towards the root of the tree, the number of active tasks is decreased by a factor of four.

After reaching the quadtree root the computation starts descending the tree, computing the Interaction Lists of its nodes. At the beginning of this downward phase available parallelism is low. Available parallelism starts increasing as the computation approaches the leaves. At the end, there is a peak in the number of active tasks which corresponds to the calculation of forces due to inter-leaf and near-neighbor interactions. In the case of uniformly distributed particles (Figure 53), the quadtree is almost balanced and the peak number of active tasks equals the number of leaves of the tree. When the quadtree is not balanced (Plummer distribution, Figure 54), however, the force calculation phase is spread in time. The peak computing activity occurs when the algorithm reaches the tree-level containing the largest number of leaves and, consequently, the peak number of active tasks is less than the total number of leaves. For instance, in the example above the average depth of quadtree leaves is 11.96, whereas the depth of the tree is 16.

The diagrams of "Active Links" are similar to the plots of "Active Tasks." Peaks in link activity follow the peaks of task activity in the first phase of the algorithm when the tasks compute centers-of-mass and then forward the results towards the root of the quadtree. In the last phase, peaks in link activity precede the peaks in task

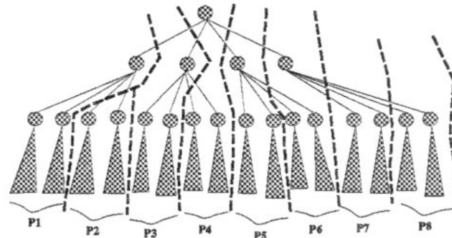

Figure 56: Ad-hoc partitioning (eight-processor case).

activity as tasks receive information about particle location and mass from nearby nodes, and then perform the force computations.

Figure 55 shows the computation and communication profiles derived from a functional algorithm simulation of a problem with 10,000 uniformly distributed particles mapped to a 128-processor clique. The corresponding quadtree was split into 128 blocks with an ad-hoc heuristic that seeks to assign an equal load to each processor by assigning an equal number of particles per processor as described in Chapter 6 (see also Figure 56). The "Active Tasks" diagram of Figure 55 differs from the diagram of Figure 53, which presents the profile of the same problem instance mapped to an abundant clique. The difference is in the profile of the second, "downward," phase of the algorithm. As we can see from Figure 53 (left), the average available parallelism during this phase is much larger than 128. Therefore, that phase incurs most of the slowdown with respect to the abundant clique implementation, when mapping the problem to a 128-clique. In contrast, the difference between the two diagrams in the first, "upward," phase of the algorithm is minimal; for the largest part of this phase, the available parallelism is very small. Therefore, most of the processors of the abundant and the 128-processor cliques stay idle. Similar remarks hold for the "Active Links" diagrams. The information exchange corresponding to the link-activity of the diagram in Figure 53, take place within the first $15,000$ time-units in Figure 55 (left).

Finally, Figure 57 shows the communication patterns for the 10,000-particle example (uniform distribution). The horizontal and vertical axes correspond to processors of the abundant clique, that is, to task-clusters of the parallel-execution graph. Each point (x, y) in the plot represents the existence of a message from the task-cluster x to task-cluster y.

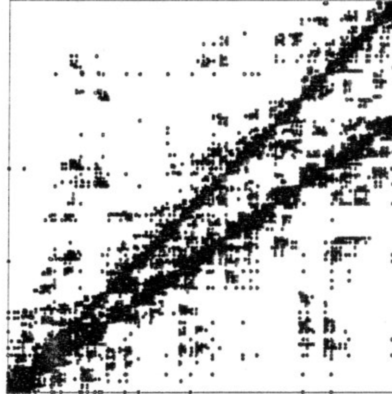

Figure 57: Communication Patterns for the 10,000-particle problem.

7.2.2 Remarks on the Scalability of Barnes - Hut algorithm

To study the scalability of the Barnes-Hut method we ran a number of functional algorithm simulations for increasing problem sizes. We used the *GL&S-20%* clustering heuristic on the parallel-execution graphs derived, and mapped the clustered graphs on the appropriate abundant clique architectures. In Figure 58, we present a diagram of the speedups measured on the abundant clique, versus the problem size (number of particles) for the uniform and Plummer particle-distributions. The speedups reported by FAST increase with problem size in both cases. The rate of speedup-increase is constant for the uniform distribution of particles, whereas it decreases with increasing problem size for the non-uniform Plummer distribution. Furthermore, speedup values are larger for the uniform distribution. This is expected since uniform distributions of particles result in balanced decomposition trees and it is easier to parallelize the computation on a balanced, as opposed to an unbalanced tree.

Figure 59 presents the speedups reported by FAST for a number of functional algorithm simulations of a Barnes-Hut instance with 8,000 particles distributed uniformly, $\theta = 0.9$, and a quadtree-granularity of three particles per leaf. For these simulations the problem was mapped on clique and hypercube multiprocessors with 8 to 2048 processing elements. For the mapping we used two different methods: the *GL&S-20%* clustering heuristic combined with the mapping technique by Yang and

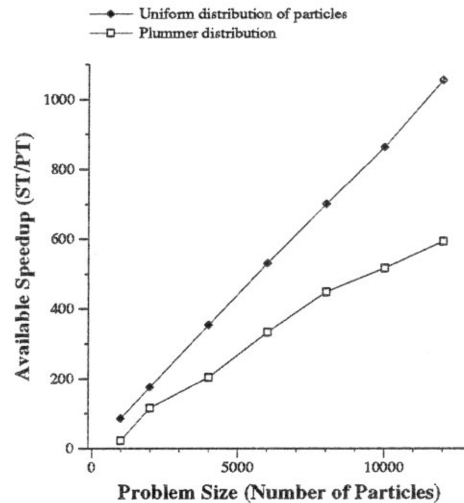

Figure 58: FAST results showing the performance of the parallel Barnes-Hut method on an abundant clique.

Gerasoulis; and the straightforward ad-hoc approach, which partitions the quadtree into a number of blocks equal to the number of available processors. From these diagrams we can see that the binary hypercube and the ring interconnections perform almost as well as the clique, for machine sizes up to 512 processors, although the hypercube and the ring are practical interconnections, and more easily scalable than the clique. Comparing the diagram to the left (*GL&S-20%* mapping) with the diagram to the right (*ad-hoc* mapping) we conclude that, for this case, the ad-hoc partitioning gives better speedups than the more expensive *GL&S-20%* heuristics, for the hypercube and ring interconnections. In contrast, *GL&S-20%* results in better performance for the clique. Finally, we can see that there is no point in using more than 512-1024 processors in a ring; or more than 1024-2048 in a hypercube when considering an 8,000-particle problem.

7.2.3 Comparing the Barnes-Hut with the Fast Multipole Method

Both the Barnes-Hut and the Fast Multipole algorithms seek approximate solutions to the N-Body problem. It is interesting, therefore, to compare their performance.

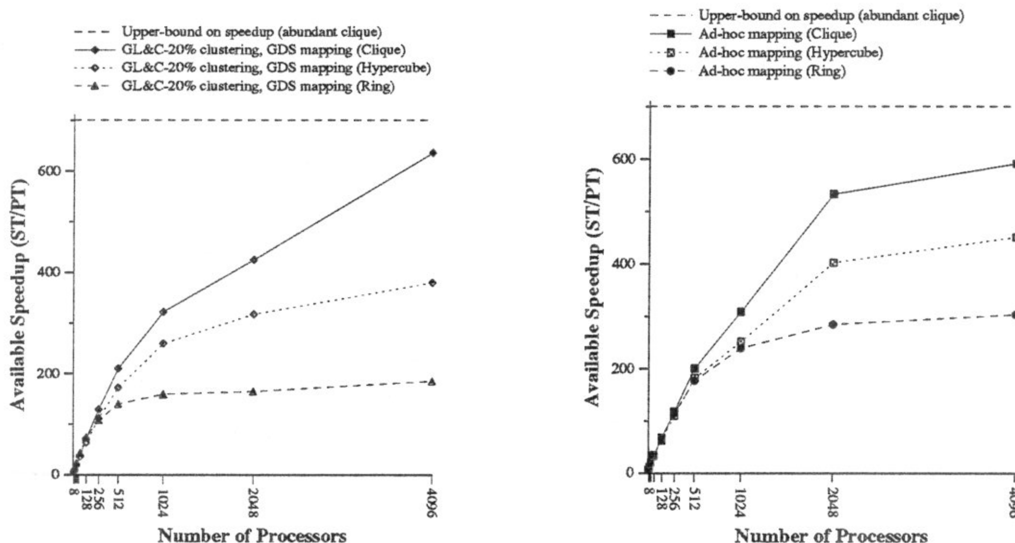

Figure 59: 8,000-particle example (Uniform distribution).

Comparisons cannot be strict, however, because the two methods define and estimate the error of approximation differently.

Here we present data extracted from functional algorithm simulations of an N-Body problem with $8,000$ particles distributed uniformly, and $10,000$ particles distributed according to the Plummer distribution. For the Fast Multipole Method we used 10 multipole coefficients and for the Barnes-Hut algorithm we set θ to 0.9. The sequential time given by FAST for the FMM was 1.8 times larger than the sequential time reported for the Barnes-Hut. Figure 60 presents diagrams of parallel times derived with FAST for both algorithms, for machine sizes from 1 to 1024 processors forming a clique network, and for the 8,000-particle example. Notice the different scales of the two plots. Figure 61 contains the plots for the 10,000-particle case. To map the FMM instance to the clique, we used the *GL&S-20%* clustering heuristic and the *SNC* mapping heuristic. For Barnes-Hut, we used the same clustering approach and the Yang and Gerasoulis mapping method. From the diagrams of Figures 60, 61 it is clear that the Barnes-Hut algorithm is faster than the Fast Multipole Method for the examples studied. This is consistent with remarks by other researchers [47, 68]

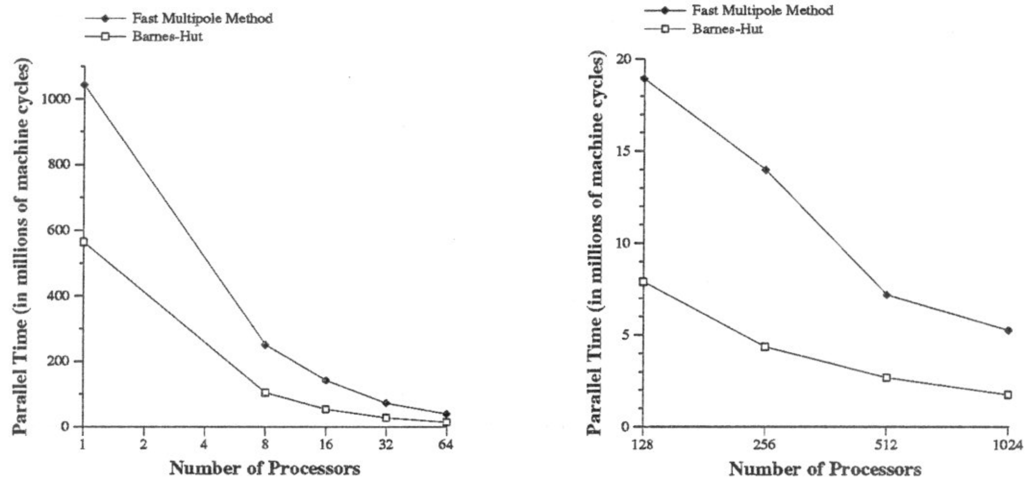

Figure 60: Parallel Times reported by FAST for the FMM and the B-H algorithms mapped on clique architectures (8,000-particles example - Uniform distribution).

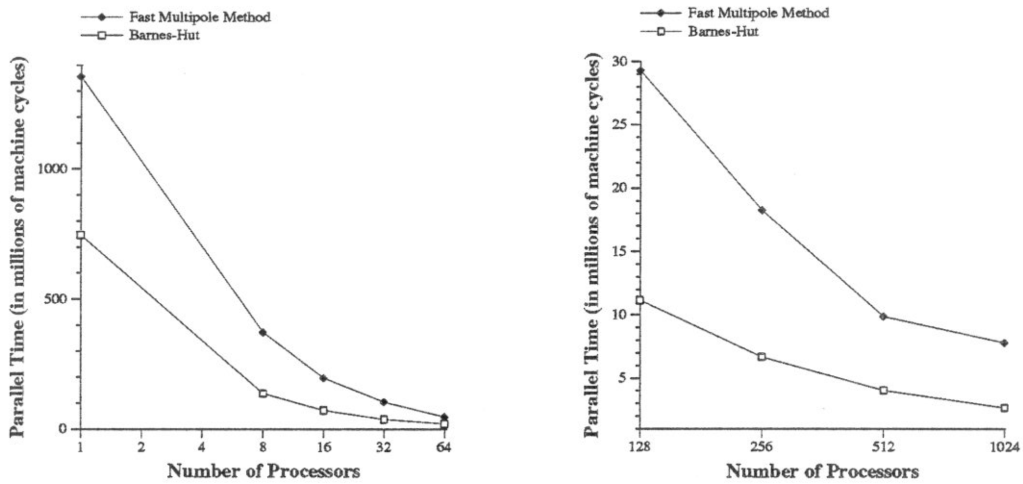

Figure 61: Parallel Times reported by FAST for the FMM and the B-H algorithms mapped on clique architectures (10,000-particles example - Plummer distribution).

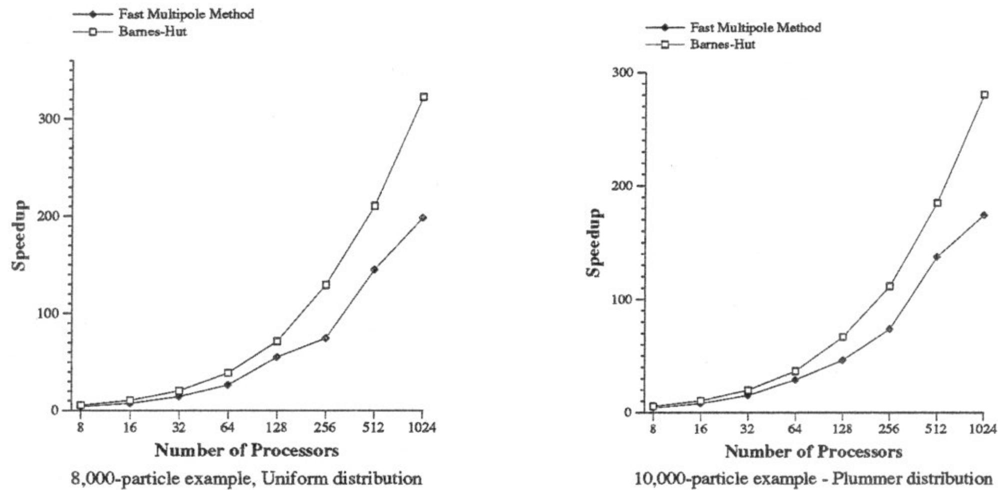

Figure 62: Speedups reported by FAST for the FMM and the B-H algorithms mapped on clique architectures.

who note that, in spite of the better average asymptotic complexity of the Fast Multipole Method, which is $O(N)$ vs. $O(N \cdot \log N)$ for Barnes-Hut, the constant factors in its complexity are substantially higher than those of Barnes-Hut.

A diagram of speedups reported by FAST for the two methods and the two examples is presented in Figure 62. From these plots we can see that Barnes-Hut also scales better than the FMM with increasing numbers of processors for a constant problem size. The reason is that the computations performed in the internal leaves of the quadtree are more "expensive" for the FMM than for the B-H, and that parallelism is limited in the lower levels of the tree. Therefore, the portion of the computation that has an inherently small potential for parallelism is greater in the FMM.

7.3 Remarks on the Barnes-Hut algorithm

To perform functional algorithm simulations of the original version of the Barnes-Hut algorithm [6], we would have to execute the largest part of its computation. To avoid this, we studied a modified version of the method, which is similar to modifications seeking to improve its performance on vector and data-parallel machines [31, 47, 77].

We used our prototype system FAST to explore the scalability of the modified

Barnes-Hut, discover its computation and communication profiles, and compare its performance with the performance of the Fast Multipole Method. It turns out that, for the examples studied, the modified Barnes-Hut algorithm is faster and more scalable than the Fast Multipole Method.

Chapter 8

Conclusions and Future Work

The high complexity of interesting scientific computations, and the high cost of high-end multiprocessor systems make extensive experimentation with computationally challenging applications and large parallel machines difficult or practically impossible. The development of parallel scientific computations can be enhanced substantially when effective tools modeling the execution of parallel workloads become available. This thesis addresses issues pertinent to the development of such tools, for modeling and evaluation of parallel scientific computations at levels of abstraction higher than the ones provided by exact simulation and profiling.

More specifically, we introduce Functional Algorithm Simulation, that is, simulation without performing the bulk of numerical calculations involved in the applications studied. Functional Algorithm Simulation is applicable in the evaluation of algorithms simulating complex systems, for which the core-set of time-consuming calculations and data-exchanges can be determined from input information, before the actual computations take place. To assess the principles of Functional Algorithm Simulation we built the *Functional Algorithm Simulation Testbed (FAST)*, a software prototype system for approximately simulating the parallel executions of such algorithms on message-passing systems. FAST runs on uniprocessor workstations with modest resources.

Experimentation with FAST shows that using approximate simulation to model parallel scientific computations at a higher level of abstraction, can give accurate and

useful results. FAST enables us to model computationally challenging algorithms in a cost-effective way. Applying this modeling approach on practical data-sets produces realistic computation and communication profiles, which expose essential characteristics of the corresponding parallel algorithms. Also, these profiles can be used to clarify several aspects of algorithmic scalability, and to reveal intrinsic bottlenecks affecting parallel performance.

FAST provides a cheap and fast means for assessing the effects that different interconnection topologies have on the performance of parallel algorithms. Data collected from FAST simulations can be used to compare interconnection schemes and study related tradeoffs. Finally, such data can be used to estimate the effects of various hardware parameters on parallel performance. For example, it is trivial to change a parameter determining processor clock speed, communication channel bandwidth, or message-initiation cost; running FAST with a different parameter-set can help us understand the relative importance of each parameter to achieving good parallel performance for the applications under consideration.

With FAST we studied three interesting and important parallel algorithms. The first is SIMPLE, a Computational Fluid Dynamics code. SIMPLE provided us with solid evidence about the validity of FAST. Furthermore, using data derived with FAST, we discovered parts of the algorithm that are inherently sequential, affect its parallel performance, and restrict scalability. The other two methods examined are the Fast Multipole Method and a modified version of Barnes-Hut algorithm. These algorithms solve the N-Body problem and have applications in many scientific fields. FAST enabled us to understand and quantify the potential for parallelism that the two methods have, for problem sizes and configurations of practical interest. We showed that as problem size increases, the available parallelism also increases. We derived upper bounds on speedup which are more informative and more meaningful than the "linear speedup" criterion used frequently to assess the efficiency of parallel applications.

Communication patterns collected for the Fast Multipole Method suggest that good parallel performance can be achieved on a sparse interconnection network. Further simulation results with FAST prove that the performance of the algorithm on

multiring topologies is almost as good as on hypercube and clique interconnections, although the latter are more expensive and less scalable than the former. Similar conclusions were derived for the modified Barnes-Hut algorithm. Finally, a comparison between the two methods shows that the version of the Barnes-Hut studied is more scalable and takes less time than the Fast Multipole Method.

8.1 Future Work

Our experience with FAST underlines the need for software tools that scientists can use to design, develop, and evaluate new computational algorithms. To this end, we plan to pursue cost-effective ways of studying the parallel computation of challenging applications and develop *Modeling and Evaluation Tools for Parallel Computations (MET-PCs)*. As a first step, we intend to incorporate in FAST a general parallel-programming model used to program current message-passing multiprocessors. Our goal is to be able to use the same parallel program for Functional Algorithm Simulation and actual parallel execution. Having such a tool will provide scientists with the opportunity to describe parallel algorithms and study their inherent properties in reasonable time and with modest computing resources.

For the design of *MET-PC*, it is important to : (1) explore, understand, and formalize the computational needs of computational scientists; (2) define the requirements and expectations they have from modeling and simulations tools to be; (3) develop effective measures for capturing the basic qualitative and quantitative characteristics of practical parallel computations; (4) experiment with existing profiling tools. This information will guide the choice of a suitable level of detail for the *MET-PC* under development, the selection of an appropriate parallel programming platform, and the design of the right experiments for deriving proper insights into the algorithms studied. Implementing a *MET-PC* requires expertise and experimentation with existing profiling and parallel programming tools.

Bibliography

[1] T. Adam, K. Chandy, and J. Dickson. A Comparison of list schedulers for parallel processing systems. *Communications of the ACM*, 17:685–690, December 1974.

[2] Anant Agarwal, R.L. Sites, and M. Horowitz. ATUM: A New Technique for capturing Address Traces Using Microcode. In *The 13th Annual International Symposium on Computer Architecture*, June 1986.

[3] R. Agrawal and H.V. Jagadish. Partitioning techniques for large-grain parallelism. *IEEE Transactions on Computers*, 37(12):1627–1634, December 1988.

[4] S.B. Shukla and D.P. Agrawal. Scheduling Pipelined Communication in Distributed Memory Multiprocessors for Real-time Applications. In *The 18th Annual International Symposium on Computer Architecture*, pages 222–231, May 1991.

[5] A.W. Appel. An efficient program for many-body simulation. *SIAM J. Sci. Stat. Computing*, 6:85, 1985.

[6] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[7] Joshua E. Barnes. A Modified Tree Code: Don't Laugh; It Runs. *Journal of Computational Physics*, 87:161–170, 1990.

[8] J. Blazewicz, M. Drabowski, and J. Weglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transactions on Computers*, pages 389–393, May 1986.

[9] L. Bomans and D. Roose. Benchmarking the iPSC/2 hypercube multiprocessor. *Concurrency: Practice and Experience*, 1(1):3–18, September 1989.

[10] Bob Boothe. Fast Accurate Simulation of Large Shared Memory Multiprocessors. Technical Report UCB/CSD 92/682, Computer Science Division, University of California at Berkeley, April 1992.

[11] E.G. Coffman. *Computer and Job-Shop Scheduling Theory.* Wiley, 1976.

[12] D. Culler, R. Karp, D. Patterson, et al. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM Sigplan Simposium on Principles and Practice of Parallel Programming*, May 1993.

[13] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *20th Annual International Symposium on Computer Architecture*, pages 2–13, May 1993.

[14] Robert Cypher. Message-Passing Models for Blocking and Nonblocking Communication. In *DIMACS Workshop on Models, Architectures, and Technologies for Parallel Computation, Technical Report 93-87.* DIMACS Center, Rutgers University, September 1993.

[15] R. Cytron, J. Lipkis, and E. Schonberg. A Compiler-Assisted Approach to SPMD Execution. In *Supercomputing*, pages 398–406, November 1990.

[16] W.J. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[17] M. Dikaiakos, A. Rogers, and K. Steiglitz. Functional algorithm simulation: A new approach for modeling the parallel execution of scientific applications. In *DIMACS Workshop on Models, Architectures, and Technologies for Parallel Computation, Technical Report 93-87.* DIMACS Center, Rutgers University, September 1993.

[18] M.D. Dikaiakos, A. Rogers, and K. Steiglitz. Message Ordering in Multiprocessors with Synchronous Communication. In *1992 International Conference on Parallel Processing*, volume III, pages 196–203, 1992.

[19] M.D. Dikaiakos, A. Rogers, and K. Steiglitz. FAST: A Functional Algorithm Simulation Testbed. In *International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems - MASCOTS'94*. IEEE-Press, 1994.

[20] H. El-Rewini and T.G. Lewis. Scheduling Parallel Program Tasks onto Arbitrary Target Machines. *Journal of Parallel and Distributed Computing*, (9):138–153, 1990.

[21] G. Fox, P. Hipes, and J. Salmon. Practical Parallel Supercomputing: Examples from Chemistry and Physics. In *Supercomputing '89*, pages 58–70, 1989.

[22] G. Fox, M. Johnson, et al. *Solving Problems on Concurrent Processors*, volume I. Prentice Hall, 1988.

[23] A. Gerasoulis, S. Venugopal, and T. Yang. Clustering Task Graphs for Message Passing Architectures. In *1990 International Conference on Supercomputing*, pages 447–457, 1990.

[24] A. Gerasoulis and T. Yang. A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors. *Journal of Parallel and Distributed Computing*, 16:276–291, 1992.

[25] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[26] S. Goldschmidt and J. Hennessy. The Accuracy of Trace-Driven Simulations of Multiprocessors. In *1993 ACM Sigmetrics*, pages 146–157, May 1993.

[27] L. Greengard and W. Gropp. A Parallel Version of the Fast Multipole Method. In Garry Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 213–222. SIAM, 1987.

[28] L. Greengard and V. Rokhlin. A fast algorithm for particle simulation. *Journal of Computational Physics*, 73:325–348, 1987.

[29] L. Greengard and V. Rokhlin. On the efficient implementation of the Fast Multipole Algorithm. Technical Report 602, Department of Computer Science, Yale University, 1988.

[30] Leslie Greengard. *The rapid evaluation of potential fields in particle systems.* PhD thesis, Yale University, Cambridge, Mass., 1988.

[31] Lars Hernquist. Vectorization of Tree Traversals. *Journal of Computational Physics*, 87:137–147, 1990.

[32] J.-M. Hsu and P. Banerjee. Performance Measurement and Trace Driven Simulation of Parallel CAD and Numeric Applications on a Hypercube Multiprocessor. In *17th Annual International Symposium on Computer Architecture*, pages 260–269, May 1990.

[33] Intel. *iPSC/860 Manual - Concurrent Programming*, May 1991.

[34] Intel Corporation. *iPSC/2 User's Guide*, October 1989.

[35] H. Kasahara and S. Narita. Practical Multi-processor Scheduling. *IEEE Transactions on Computers*, C-33:1023–1029, 1984.

[36] J. Katzenelson. Computational Structure of the N-Body Problem. *SIAM Journal of Scientific and Statistical Computing*, 10(4):787–815, July 1989.

[37] S.J. Kim and J.C. Browne. A General Approach to Mapping of Parallel Computations upon Multiprocessor Architectures. In *International Conference on Parallel Processing*, volume 3, pages 1–8, 1988.

[38] L. Kohn and N. Margulis. The i860 64-bit Supercomputing Microprocessor. In *Supercomputing '89*, pages 450–456, November 1989.

[39] S. Kugelmass and K. Steiglitz. A Scalable Architecture for Lattice-Gas Simulations. *Journal of Computational Physics*, vol. 84:311–325, October 1989.

[40] J.F. Leathrum and J.A. Board Jr. The Parallel Fast Multipole Algorithm in Three Dimensions. Technical report, Department of Electrical Engineering, Duke University, April 1992.

[41] F.F. Lee. Partitioning of Regular Computation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 9:312–317, 1990.

[42] J. Lee, C. Lin, and L. Snyder. Programming SIMPLE for Parallel Portability. In *Preliminary Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing*, 1992.

[43] S-Y. Lee and J.K. Aggarwal. A Mapping Strategy for Parallel Processing. *IEEE Transactions on Computers*, C-36(4), April 1987.

[44] D. Lenoski, J. Laudon, L. Stevens, T. Joe, D. Nakahira, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *19th Annual International Symposium on Computer Architecture*, May 1992.

[45] C. Lin and L. Snyder. A Portable Implementation of SIMPLE. *International Journal of Parallel Programming*, 20(5):363–401, 1991.

[46] V.M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on Computers*, pages 1384–1397, November 1988.

[47] Junichiro Makino. Vectorization of a Treecode. *Journal of Computational Physics*, 87:148–160, 1990.

[48] A.D. Malony and D.A. Reed. Hardware-Based Performance Monitor for the Intel iPSC/2 Hypercube. In *1990 International Conference on Supercomputing*, pages 213–226, 1990.

[49] S. Manoharan and P. Tranish. Assigning Dependency Graphs onto Processor Networks. *Parallel Computing*, 17:63–73, 1991.

[50] L. Matheson and R. Tarjan. A Critical Analysis of Multigrid Methods on Massively Parallel Computers. In *Fourth European Multigrid Conference*, 1993.

[51] Tin-Fook Ngai. Runtime Resource Management in Concurrent Systems. Technical Report CSL-TR-92-504, Computer Systems Laboratory, Stanford University, January 1992.

[52] M.G. Norman and P. Thanisch. Mapping in Multicomputers. *ACM Computing Surveys*, 25(3):264–302, September 1993.

[53] S.F. Nugent. The iPSC/2 Direct-Connect Communications Technology. In *Concurrent Supercomputing - The second Generation - A Technical Summary of the iPSC/2 Concurrent Supercomputer*, pages 59–68. Intel.

[54] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and Complexity.* Prentice Hall, 1982.

[55] K. Pingali and A. Rogers. Compiler Parallelization of SIMPLE for a distributed memory machine. Technical Report 90-1084, Department of Computer Science, Cornell University, 1990.

[56] C. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, 1988.

[57] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. Technical Report 1122, Computer Sciences Dept., University of Wisconsin-Madison, November 1992.

[58] M. Rinard, D. Scales, and M. Lam. Heterogeneous Parallel Programming in Jade. In *Supercomputing*, 1992.

[59] M. Rinard, D. Scales, and M. Lam. Jade: A High-Level Machine-Independent Language for Parallel Processing. *Computer*, 26(6):28–38, June 1993.

[60] A. Rogers. Compiling for Locality of Reference. Technical Report PhD Thesis 91-1195, Department of Computer Science, Cornell University, 1991.

[61] John Salmon. Parallel N logN N-body Algorithms and Applications to Astrophysics. In *36th IEEE Computer Society International Conference*, pages 73–78. IEEE, 1991.

[62] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, 1989.

[63] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. *SIGPLAN Notices*, 21(7):17–26, July 1986.

[64] S.L. Scott, J.R. Goodman, and M.K. Vernon. Performance of the SCI Ring. In *19th Annual International Symposium on Computer Architecture*, pages 403–414, May 1992.

[65] R. Sedgewick. *Algorithms.* Addison Wesley, 1988.

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

[67] J.P. Singh, John Hennessy, and A. Gupta. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *Computer*, 26(7):42–50, July 1993.

[68] J.P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load Balancing and Data Locality in Hierarchical N-body Methods. Technical Report CSL-TR-92-505, Computer Systems Lab, Stanford University, 1992.

[69] D. Smitley, F. Hady, and D. Burns. Hnet: A High-performance Network Evaluation Testbed. In *1992 International Conference on Parallel Processing*, volume I, pages 276–279, August 1992.

[70] C.B. Stunkel and W.K. Fuchs. TRAPEDS: Producing Traces for Multicomputers via Execution Driven Simulations. In *International Conference on Measurement and Modeling of Computer Systems*, pages 70–78, May 1989.

[71] C.B. Stunkel and W.K. Fuchs. TRAPEDS: Producing Traces for Multicomputers via Execution Driven Simulations. In *International Conference on Measurement and Modeling of Computer Systems*, pages 70–78, May 1989.

[72] Thinking Machines. *The Connection Machine CM-5 Technical Summary*, October 1991.

[73] L. G. Valiant. A Bridging Model for parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[74] T. von Eicken, D. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *19th Annual International Symposium on Computer Architecture*, pages 256–267, May 1992.

[75] T. Yang and A. Gerasoulis. A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors. In *Supercomputing 91*, 1991.

[76] T. Yang and A. Gerasoulis. PYRROS: Static scheduling and code generation for message passing multiprocessors. In *6th ACM International Conference on Supercomputing*, pages 428–437, July 1992.

[77] F. Zhao and S.L. Johnsson. The Parallel Multipole Method on the Connection Machine. *SIAM Journal of Sci. and Stat. Comput.*, 12:1420–1437, 1991.