Abstracts of the
DIMACS Workshop on Parallel Algorithms
for Unstructured and Dynamic Problems
June 2–4, 1993

Wednesday, June 2, 9-10

Adaptive Problems and High Performance Fortran Compilers

Joel Saltz
University of Maryland

Over the past few years, we have developed methods that make it possible for a compiler to efficiently map many sparse and unstructured scientific problems to scalable multiprocessor architectures. These methods are implemented using compiler embeddable PARTI runtime preprocessing procedures. These procedures (1) coordinate interprocessor data movement, (2) manage the storage of and access to copies of off-processor data, (3) support a shared name space, and (4) couple runtime data and workload partitioners to compilers. We will describe a prototype Fortran D compiler which we are constructing in collaboration with groups at Syracuse and Rice Universities which uses PARTI procedures to handle unstructured computations. We will also describe experimental results obtained using the PARTI procedures to port molecular dynamics codes and unstructured mesh Euler solvers to the Intel iPSC/860, Delta and CM-5.

Wednesday, June 2, 10:30-11

Towards an Automated Approach to the Simulation of 3D Problems

Gary L Miller
CMU
Stan Eisenstat, Dafna Talmor, Shang-Hua Teng, Bill Thurston, and Steven Vavasis

The simulation of the 3-dimensional problems comprise many of the "grand challenge problems". Problems on the list include climate forecasting and turbulence. These problems require computer simulation of problems over highly unstructured domains. For the last several years we have been developing algorithms which will assist researchers in making these simulation on the next generation of parallel machines.

In particular, we have made progress on two major aspects of the problem: partitioning and formulatization/meshing. We have shown that meshes/combinatorial-graphs of size n that arise from from the finite element method can be partitioned into two roughly equal size pieces with a boundary of size at most $O(n^{(}d-1)/d)$ in d-dimensions. This result has application to efficient direct methods to solve the associated linear system. Also, the result can be used for placement of these systems on machines with sublinear bisection width such as fat-tree or mesh-connected machines.

It seems that one of the main stumbling blocks to complete automation of finite element method is mesh generation in 3-dimensions. We have been experimenting with an unstructured finite difference like method. The advantage with this method is that we do not need to generate unstructured meshes as used in the finite element method. We will show some numbers from preliminary experimentations with this method. They indicate that this method is competitive with finite element. Our methods should lend itself to problems which need a moving or changing mesh.

Wednesday, June 2, 11-11:30

Are Scalable Methods for Sparse Computations Practical?

Vijay K. Naik
IBM Research
Sesh Venugopal
Rutgers University

Extended Abstract

In the applications involving unstructured and sparse computations, the available parallelism at the level of arithmetic operations is large enough to keep hundreds of processors busy at all times during the entire computation. This abundant supply of parallelism has led to several interesting parallel algorithms for unstructured sparse matrix computations. However, the experience in the last several years indicates that the performance of these algorithms on the existing general purpose parallel architectures ranks close to the bottom.

The wide gap between the theoretically achievable performance and that observed in practice has led many to believe that the known algorithms for unstructured sparse matrix computations on general purpose parallel architectures are not scalable. The reasons for this belief are manifold. First of all, although there is an abundance of parallelism, it is most easily extractable at the smallest level of arithmetic operations, such as multiply or add. Identifying and scheduling these operations appropriately at run-time is prohibitively expensive. Even if this were possible, the modern RISC-based architectures deliver high performance only when cache utilization is high and the vector units are used efficiently. Both requirements demand data structures that use memory in a regular and structured fashion. This is not possible with the conventional representations for sparse matrix computations. Finally, if one were to parallelize the unstructured sparse matrix computations taking into account all the available parallelism, the amount of communication and the scheduling overheads would easily offset any gain in parallelism, even on a relatively small number of processors.

Clearly, the parallel sparse algorithms exploit parallelism at the fine grain levels, while the general purpose scalable architectures favor large or coarse grain parallelism. To achieve scalable performance using these algorithms on general purpose multiprocessor systems, the question to be answered is whether it is possible to set up a system that will provide an efficient and effective interface between the two. Currently we are working on developing such a system and, from the results on sparse Cholesky factorization, our answer to the above question is yes.

As a part of this work, we have developed a methodology for scalable sparse Cholesky factorization. The core of the methodology is based on the assumption that if the computations can be blocked in a structured manner, then the resulting implementations tend to be scalable. As a first step, we conducted a thorough study of block-based partitioning for sparse Cholesky factorization. Block partitioning for dense matrices is known to be more *scalable* as compared to column-based partitioning, and has *better cache utilization properties*. But exploiting these features in sparse Cholesky factorization is a complex task and requires sophisticated automatic tools. Our analysis of the computations for the sparse matrices arising from a class of structured grid problems indicates that block partitioning is more scalable than column partitioning. Using the analysis as a starting point, we developed a four step approach to handle complex unstructured sparse matrices. These steps are: *partitioning*, *allocation*, *scheduling of computation*, and *scheduling of communication*. Each of these steps or functions must be modular, scalable, and has to interface with the sparse matrix computations being manipulated. To accomplish this efficiently, we have conceptualized a a concise representation for the coarse-grain asynchronous computations that are built on the top of the dependencies from the fine-grain computations in the Cholesky algorithm. This representation is in the form of a block dependency directed acyclic graph called BDAG. It stores complete task and data partitioning information and precedence constraints, and exposes all the available parallelism. The representation is used to automatically schedule computation and communication, and would be useful for parallelizing any unstructured problem, assuming that the data partitioning has been done and that the processors use the "owner computes" rule.

For demonstrating the feasibility of our methodology, we have developed several tools that include, a *scalable partitioner* for sparse Cholesky factorization, a *general block dependency graph builder* for coarse-grain asynchronous computations, a *scalable pre-scheduler* to schedule the computations and communication represented by a BDAG, a set of *run-time primitives* to ease programming and improve performance for coarse-grain asynchronous computations, and a *scalable communication optimization algorithm* for message passing in asynchronous computations.

Using this methodology and set of tools, we have built a scalable software system for partitioning and scheduling block-based sparse Cholesky factorization on the iPSC/860. The system is called SHAPE, for Sparse Hybrid Automatic Parallelization Environment. We have employed SHAPE to carry out an extensive experimental study of block sparse Cholesky factorization on the iPSC/860. The results indicate that the methodology outlined here leads to scalable performances for a wide variety of test cases. Moreover, the results of this experimental study are helping us to gain a better understanding of block partitioning and scheduling, so that general principles for other sparse computations may be developed.

Wednesday, June 2, 11:30-12

An Experimental Comparison of Algorithms for Sparse Cholesky Factorization

Calvin Lin
Dept. of Computer Science and Engr.
University of Washington
Lawrence Snyder
Dept. of Computer Science and Engr.
University of Washington
W. Derrick Weathersby
Dept. of Computer Science and Engr.
University of Washington

Sparse Cholesky factorization is a common linear algebra computation that has been notoriously difficult to parallelize because its irregular data often causes poor locality of reference and poor load balance. This computation has been particularly difficult to implement on distributed memory machines. The goal of our work is to study this computation with respect to portability and scalability. In particular, we wish to understand whether a single algorithm can perform well on both shared and distributed memory MIMD computers, and we are interested in large problem sizes.

In this talk we describe two widely used sequential Cholesky algorithms and their parallel counterparts. We characterize these parallel algorithms in terms of their memory access patterns and flow of control, and we show results comparing these algorithms across a variety of MIMD multiprocessors – the Sequent Symmetry, the Intel iPSC/2, and the Kendall Square Research KSR-1. Our implementations use a non-shared memory programming model. We also compare our implementations against the shared memory Cholesky implementation from the SPLASH benchmark suite. Finally, we identify the performance limiting characteristics of the algorithms and implementations with respect to portability and scalability.

Wednesday, June 2, 1:30-2

Match and Move: Parallel Constructs for Sparse Matrix and Graph Algorithms

Thomas J. Sheffler
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

The **match** and **move** operators are data parallel primitives of sufficient power and generality to describe many algorithms on irregular and dynamic structures such as graphs and sparse matrices. The high level of programs written using **match** and **move** allow implementations of the operators to be fine-tuned for a particular architecture by making use of special features available. Because multiple implementation strategies may be possible for a given machine, programs written using **match** and **move** gain a level of implementation independence as well. The heightened abstraction of algorithms written in this manner not only ensures their portability, but allows radically different implementation strategies to be explored with no change to user level code.

Wednesday, June 2, 2-2:30

Dynamic Scheduling in Multilisp

Robert H. Halstead, Jr.
DEC Cambridge Research Lab

Multilisp is an extension of the Lisp dialect Scheme with additional operators and additional semantics for parallel execution. The design of Multilisp has been motivated by the needs of irregular, dynamic, "symbolic" computations.

The principal parallelism construct in Multilisp is the *future*, a concurrency and synchronization construct that exhibits some features of both eager and lazy evaluation. A high-performance Multilisp implementation called Mul-T runs on the Encore Multimax multiprocessor and has been used on up to 18 processors.

A Lisp-based approach is attractive for parallel computing since Lisp languages and systems assume significant clerical burdens, such as storage management, which otherwise require programmer effort. A parallel Lisp thus enables a programmer to focus more attention on the new problems introduced by using concurrency, instead of wasting time solving old problems like storage management whose solutions can be automated. Mul-T and other extant parallel Lisps can execute realistic applications with "industrial-strength" performance, but there are applications whose requirements they do not yet meet as well as they could.

Recent work with Multilisp has contributed new ideas about scheduling and speculative computation. The *lazy task creation* scheduling mechanism significantly reduces task-management overhead costs in fine-grained parallel programs by automatically and dynamically aggregating tasks into larger groups, according to the run-time load on the system. For programs amenable to speculative computing (which involves starting computations before their results are known to be required), a system of *sponsors* and task priorities can support faster execution. Both lazy task creation and sponsors allow the programmer to work at a higher level and concentrate on describing the algorithm that a multiprocessor should follow, rather than being forced to micromanage the scheduling and granularity of tasks by mingling explicit scheduling code with the program's computational instructions. This benefit is especially welcome when programming irregular, dynamic computations whose run-time parameters (such as the amount of available parallelism) may vary widely and be difficult to anticipate.

Both lazy task creation and sponsors have been implemented and measured on a range of applications. Lazy task creation has been implemented in Mul-T (by Eric Mohr and David Kranz) and has yielded significant benefits (up to a factor of 3 performance improvement) for several programs written with fine-grained tasks. A preliminary implementation of sponsors in Multilisp has been built (by Randy Osborne) and measured on several applications such as the Boyer theorem-proving benchmark, a subset of the EMYCIN rule-based inferencing engine, and a program that searches for a solution to the "8-puzzle." Performance improvements from using speculative computing in a simple and intuitive way range up to a factor of 26 (for the 8-puzzle problem) over the performance achieved with a simple program using ordinary futures.

Wednesday, June 2, 2:30-3

Tao Yang and Apostolos Gerasoulis
Department of Computer Science
Rutgers University, New Brunswick, NJ 08903
Email: {tyang, gerasoulis}@cs.rutgers.edu

We consider the problem of automatic scheduling and code generation for irregular task graphs on massively parallel architectures. We introduce a new quantification of the granularity and use it to identify the impact of partitioning on optimal scheduling. We present heuristic scheduling algorithms and provide theoretical and experimental results to verify the performance of these algorithms. We propose an optimized method for executing a scheduled task graph on a distributed memory architecture based on an asynchronous communication model. We also prove the correctness of the generated code that assures data coherence and deadlock-free communication.

We have implemented the above algorithms on a compiler tool system named PYRROS. The input of this system is a set of C or Fortran modules with dependence information and the output is optimized parallel code for message-passing distributed memory machines such as nCUBE-1, nCUBE-2 and iPSC/860. We present the performance of PYRROS on nCUBE-2 and discuss its application to irregular problems arising from sparse matrix computation.

The references on this work can be obtained via anonymous ftp from machine cs.rutgers.edu under directory pub/gerasoulis.

[1] A. Gerasoulis and T. Yang, A comparison of clustering heuristics for scheduling DAGs on multiprocessors. *Journal of Parallel and Distributed Computing*, Special issue on scheduling and load balancing, Vol 16, No. 4, Dec. 1992, pp. 276-291.

[2] A. Gerasoulis and T. Yang, On the granularity and clustering of directed acyclic task graphs. To appear in *IEEE Transactions on Parallel and Distributed Systems.*, June 1993.

[3] T. Yang and A. Gerasoulis, PYRROS: Static scheduling and code generation for message passing multiprocessors. *Proc. of 6th ACM International Conference on Supercomputing*, Washington D.C., July, 1992, pp. 428-437.

[4] T. Yang and A. Gerasoulis, DSC: Scheduling parallel tasks on an unbounded number of processors. To appear in *IEEE Transactions on Parallel and Distributed Systems.*

[5] T. Yang, Scheduling and code generation for parallel architectures, Ph.D. Thesis, DCS-TR-299, Rutgers University, New Brunswick, NJ, May 1993.

Wednesday, June 2, 3:30-4

Exploiting Locality In Certain Geometric Algorithms

Abhiram Ranade
University of California, Berkeley

We present efficient randomized Butterfly network algorithms for finding convex hulls and the closest pair of points in two dimensions. For both problems, our algorithms run in time $O((N \log N/P) + \log^3 P)$, where N is the number of points, and P the number of processors in the Butterfly network. Our algorithms thus give linear speedups for each problems over the optimal sequential algorithms, provided $N = \Omega(P \log^3 P)$. Randomization is used only for fast sorting on the Butterfly (Flashsort). Indeed, for N polynomially larger than P, our algorithms can be made deterministic by using efficient deterministic sorting algorithms on the Butterfly. Previous work on these problems mostly used PRAMs, or used far less parallelism.

Our algorithms are portable to other networks besides the Butterfly, e.g. two dimensional grids. Indeed, it is possible to show that a simple simulation of the Butterfly algorithm on the grid will give the best grid algorithms for these problems. In other words, the algorithms exploit locality in a network independent manner.

Wednesday, June 2, 4-4:30

Collective Communications Operations for Unstructured, Dynamic Problems and their Suitablity for Optical Communication Networks

Larry Rudolph
Institute of Computer Science, Hebrew University, Israel

We discuss several extensions to the popular "loosely synchronous" model of parallel computation and discuss their relevence. We argue that the use of group communications are important for dynamic problems. Each process can be part of several groups and each collective communication must be performed by all the processes in the group. Thus, processes can work on different parts of the same problem with infrequent interaction of all the processes. In addition, we provide a mechanism that enables any processes to be replaced by a group of processes so that the group appears to the other processes as a single entity.

Each collective communication operation can be split into three parts: setup, initiate, and await. The idea is that the setup is executed, once outside the loop and the other two operations are performed within the loop. Not only can the software overheads of communication set-up be amortized in this fashion but so can the hardware overheads.

The talk will also review some methods for building interconnection networks based on free-space optics. The setup operation are particularly useful for such networks. In addition, the demands of unstructured and dynamic problems may need the power of free space optical networks. In such networks, potential links are fairly cheap, unlike traditional (electronic) networks in which every link is expensive. Of course, there is no free lunch – actual communication between nodes is not cheap.

Wednesday, June 2, 4:45-5:45, Poster Session

SOFTWARE SUPPORT FOR PARALLEL ADAPTIVE MESH REFINEMENT ALGORITHMS

U.R. Hanebutte
ICASE, Mail Stop 132
NASA Langley Research Center
Hampton, VA 23681-0001
and
J.J. Quirk
ICASE, Mail Stop 132
NASA Langley Research Center
Hampton, VA 23681-0001

Over the last few years, Adaptive Mesh Refinement (AMR) algorithms which dynamically match the local resolution of a computational grid to the numerical solution being sought have emerged as powerful tools for solving problems that contain disparate length and time scales. In particular, both Berger[1] and Quirk[2,3] have demonstrated the effectiveness of employing an adaptive, block-structured hierarchical grid system for simulations of complex shock wave phenomena. The goals of the work described in this abstract are two-fold. On the one hand, we wish to obtain an SPMD implementation of Quirk's algorithm without the need for rewriting major parts of the serial code, and on the other we wish to develop general concepts that will lead to software tools which would be applicable to a broader class of algorithms using adaptive data structures.

Briefly, the main issues of the AMR algorithm are the following. The bottom level of the hierarchical grid structure consists of a set of coarse grids which delineate the computational domain. Local refinement is achieved by embedding finer grids into the coarse grids. The finer grids then become the members of the next higher grid level. This process may be repeated indefinitely. To identify each grid, a global grid index is assigned to each grid of the structure. The flow solution is calculated on each level for every entire grid which is associated with that level. The algorithm requires the grids to be surrounded by a band of dummy-cells which have to be primed before the integration of each individual grid can be performed. The grid integration is a natural source for parallelism, since the integration is performed independently for each grid.

Although each individual rectangular grid is a static uniform mesh patch whose spacing and associated time step depend only on the grid level the overall grid structure is both irregular and dynamic. An adaption has to be performed frequently enough to ensure the consistent discretization of the moving flow throughout the entire calculation. When a grid level is to be adapted it implies that not only that level but all levels above have to be adapted and that the adaption will begin with the highest (i.e. the finest) level. To satisfy a required stability condition, the time step has to be reduced according to the refinement factor between grid levels. This results in a smaller time step on a finer grid level. Therefore, the number of integrations and adaptions increases with the grid level. This fact combined with the observation that the number of grids generally increases with grid level leads to the highest work load being on the finest grid level. Anticipated simulations will require a hierarchical grid structure containing several thousand grids, where the grid size can vary from a few to a few thousand grid points. An AMR algorithm performs grid to grid data exchange not only along grid boundaries, but also inside the grids when data is transferred between levels.

To achieve an SPMD implementation without major rewriting, it is important to preserve the global grid index. The global index however, must be supplemented by an local grid index pair consisting of processor location and local index of the grid. In order to ensure that an unidirectional mapping exists between global index and local index pair we impose the restriction that a grid resides completely on a single processor. This is not an unreasonable requirement, since a large grid can always be replaced by several smaller grids.

The data distribution strongly influences the parallel implementation strategy and so must be considered early on. Apart from data which is temporarily required during the adaption and integration steps one can divide the data structure of the algorithm into three categories. The first category consists of data describing the grid structure, this is a relatively small amount of data. For example, only two reference points (i.e. the lower left and upper right corner) have to be known in terms of global mesh coordinates to define the location of a grid. The second group is the connectivity information along the boundaries of each individual grid. This data is derived from grid structure data. And the third and largest category is the flow data, which has to be stored at each mesh point.

While the last two categories contain local data, the first category represents global data. Local data is distributed among processors in accordance with the processor location of the associated grid. The global grid structure data is replicated on all processors. This is feasible, since only a few thousand grids are contained within the grid structure and the amount of grid structure data per grid is small. Having the global grid structure locally available gives some advantages. First, load balance calculations for a grid structure does not require any communication, since each processor has the data describing the global structure, and is therefore able to generate the complete grid distribution table independently. Second, the connectivity information for each grid can be calculated simultaneously using the replicated data. Third and most important, almost all inter-processor data movements can be performed without the need for additional communication during the scheduling phase. This is true, for example, during the projection step after the solution is obtained on a grid level. There, each processor uses its own local copy of the global structure to determine the overlapping grid regions of the two adjacent levels for which it is responsible. For these regions, each processor will then either schedule all needed sends and receives, or perform internal data movements. However, a different communication paradigm has to be employed for the data transfer during the grid boundary update, since the connectivity information is not replicated. To prime grid boundary dummy-cells, the processor responsible for the grid will have to build first requests for all off-processor data that it needs. These requests are then exchanged between processors, followed by the exchange of the actual data. Depending on the adaption frequency, the connectivity information of a grid remains constant over some period of time. By adding a request type to each request, it is possible to minimize the request exchange by not resending unchanged requests.

While an SPMD implementation of the AMR algorithm is far from being complete, the general issues involved in such an algorithm are now understood, and required software tools are being designed. A preliminary study shows that the objective to obtain an SPMD implementation without major rewriting can be met. The existing serial routines which handle data transfer between grids are being supplemented with new runtime routines. These runtime routines analyze the required inter-grid data transfer and schedule the appropriate actions. In addition, for load balancing, a routine is called between the building of the grid structure and the actually priming of the grids with the flow data. The task of this routine is to assign each global grid index a local index pair, such that the work is distributed in the best possible manner. During the course of this project, it is planned to investigate different techniques in order to achieve best load balancing.

At the workshop, we will present these runtime routines in more detail and report our first experience with an SPMD implementation of a subset of the AMR algorithm.

References

[1] M.J.Berger and P.Colella, *Local adaptive mesh refinement for shock hydrodynamics.* J.Comput. Phys., **82**(1989), pp. 67–84.

[2] J.J.Quirk, *An adaptive grid algorithm for computational shock hydrodynamics.* Ph.D. thesis, College of Aeronautics, Cranfield Institute of technology (1991).

[3] J.J.Quirk, *An alternative to unstructured grids for computing gas dynamic flows around arbitrarily complex two-dimensional bodies.* NASA CR-189612, ICASE Report No. 92-7, (1992). To appear in Computers & Fluids.

Wednesday, June 2, 4:45-5:45, Poster Session

A portable software support system for irregular computations
(extended abstract)

Laxmikant V. Kalé, Amitabh B. Sinha, Attila Gursoy
Department of Computer Science,
University of Illinois,
Urbana, IL 61801.
email: {kale,sinha,gursoy}@cs.uiuc.edu

1 Introduction

In the last decade, parallel programming has emerged as a powerful new technology. Many large commercial parallel machines are available today, such as Intel iPSC/860 (and soon, the Paragon), NCUBE's 1024 processor machines, CM-5, etc. A large class of algorithms in science, engineering, operations research, and artificial intelligence can potentially benefit from parallel processing. However most of the applications that have been successfully solved using parallel machines have a very regular structure. Irregular computations, such as branch&bound, and adaptive grids constitute an important class of algorithms. The effective parallel solution of irregular computations poses greater problems because of the need for facilities for dynamic creation of work, and dynamic load balancing.

The features required of any software support for irregular computations are:

- Portability: The application program should be portable across the wide variety of parallel machines that are currently available.
- Dynamic load balancing: The system should provide support for automatically balancing load, in addition to the capabilities of user-controlled load balancing.
- Dynamic creation of work: A large number of irregular computations could be more easily solved using parallel computers if tasks could be inexpensively created dynamically. The support system should allow inexpensive, dynamic creation of tasks.
- Modularity and reuse: An application program could consist of different modules that may be written by different programmers; therefore the system should allow programs to be written in a modular fashion and the reuse of modules.
- Prioritization: In irregular computations, there may exist many different tasks in the system at the same time. Additionally, the order of processing of these tasks may be important. In such cases, the system should provide support for the prioritized execution of tasks.

Over the last six years, we have developed a machine independent parallel programming system called Charm. The system provides all the support needed by a software support system for irregular computation. Charm has been extensively used for successfully solving large irregular computations arising in branch&bound algorithms, CAD problems etc. In this abstract, we describe the basic features of Charm and an example of a branch&bound algorithm that was successfully solved using parallel computers.

2 Charm

Charm is a machine independent parallel programming system. Programs written using this system will run unchanged on MIMD machines with or without a shared memory. The system currently runs on Intel's iPSC/860, iPSC/2, NCUBE, Encore Multimax, Sequent Symmetry, ALLIANT FX/8, single-processor UNIX machines, and networks of workstations. It is being ported to a CM-5, Parsytec GC-el, and Alliant FX/2800.

Programs consist of potentially small-grained processes (called chares), and a special type of replicated processes, called branch-office chares. Charm supports dynamic creation of chares, by providing dynamic (as well as static) load balancing strategies. There may be thousands of small-grained chares on each processor, or just a few, depending on the application. Chares interact by sending messages to each other and via specific information sharing modes described below.

A charm program consists of chare definitions, message definitions, and declarations of specifically shared objects in addition to regular C language constructs. A chare definition consists of local variable declarations, entry-point functions that handle incoming messages and private function definitions. Some of the important Charm system calls are: *CreateChare(chareName,entryPoint,msg)* and *SendMsg(chareID,entryPoint,msg)*. CreateChare call is used to create an instance of a chare named as *chareName*. As all other Charm system calls, CreateChare is a non-blocking call, i.e., it immediately returns. Eventually the system creates an instance of chare *chareName* on some processor, starts its execution the *entryPoint* with the message *msg*. The SendMsg call deposits the message *msg* to be sent to the *entryPoint* of chare instance *chareID*.

The execution model of Charm is message-driven. The runtime system on each processor selects a message from the pool of available messages, in accordance with a scheduling strategy. If the message is directed to an existing chare, it invokes this chare with the message. If the message specifies creation of a new chare, the system creates it, and invokes it immediately with the message.

Messages represent just one mode of information sharing. Charm provides six information sharing modes, each of which may be implemented differently and efficiently on different machines. These include abstractions such as read-only variables, accumulators, distributed tables, and monotonic variables [1, 2].

Charm also provides a sophisticated module system that facilitates reuse, and large-scale programming for parallel software. From the point of this paper, the following properties of Charm are important:

1. Charm provides efficient portability across a range of parallel machines,
2. Charm allows for dynamic creation of processes and supports this with a suite of *dynamic load balancing* strategies.
3. Charm programs may specify (arbitrary, unbounded, levels of) priorities with messages and new processes.
4. Charm supports features such as distributed tables, which are very useful in many irregular computations.
5. Message driven execution in Charm enables creation of parallel programs that are highly *latency tolerant* (and tolerant of the unpredictability of the response times from remote processes, in general). Processors are never blocked waiting for a particular message. When one processor waiting for data from a remote process, another ready process may be scheduled for execution. Also, even a single process may wait for multiple data items simultaneously, and continue execution whenever any of the expected items arrive.
6. The *modularity* features supported in Charm permit flexible reuse of parallel software. Separately compiled library modules can be put together in a varying of combinations and contexts.

3 Application: Traveling Salesman Problem

The Traveling Salesman Problem (TSP) [3] is a typical example of an optimization problem solved using branch&bound techniques. In this problem a salesman must visit n cities, returning to the starting point, and is required to minimize the total cost of the trip. Every pair of cities i and j has a cost C_{ij} associated with them (if $i = j$, then C_{ij} is assumed to be of infinite cost).

We have implemented the branch&bound scheme proposed by Little, et. al. [4]. In Little's approach one starts with an initial partial solution, a cost function (C) and an infinite upper bound. A partial solution comprises a set of edges (pairs of cities) that have been included in the circuit, and a set of edges that have been excluded from the circuit. The cost function provides for each partial solution a lower bound on the cost of any solution found by extending the partial solution. The cost function is monotonic, i.e., if S_1 and S_2 are partial solutions and S_2 is obtained by extending S_1, then $C(S_1) \leq C(S_2)$. Two new partial solutions are obtained from the current partial solution by including and excluding the "best" edge (determined using some selection criterion) not in the partial solution. A partial solution is discarded (pruned) if its lower bound is larger than the current upper bound. The upper bound is updated whenever a solution is reached.

In the Charm implementation of the branch&bound solution of TSP, each partial solution is represented by a chare and the cost of the partial solution is the priority of the new chare message. A monotonic variable is used to maintain the upper bound.

Table 1 shows the execution times and the number of nodes generated for runs of a 60-city asymmetric TSP on the NCUBE/2 with a new load balancing strategy [7]. Notice that the number of nodes generated in this case are fairly constant for up to 512 processors and the speedups are good. The speedup numbers are estimated with the (normalizing) assumption that speedup is perfect up to 64 processors.

Figure 1: **The figure shows the execution times and the number of nodes generated for executions of a 60 city asymmetric TSP on the NCUBE/2 with upto 512 processors using the tokens strategy to balance load. In this case the cluster size is 16 processors.**

Processors	1 (estimated)	64	128	256	512
Time	19,366	302.6	151.1	86.2	42.1
Estimated Speedup	1	64	128	225	460
Number of Nodes expanded	-	85,165	84,030	93,816	85,420

We have developed a generalized branch&bound package that provides the parallel component of the branch&bound algorithm. Different branch&bound applications can be written on top of the parallel component by providing functions, such as *initialization*, *branching* and *lower-bounding*, for that particular application.

4 Discussion

We are using Charm (and its newer C++ based version [5]) to solve other irregular computations. These obviously include tree-structured computations, such as state space search, bi-directional search game tree search, and planning. (On many of these we have developed and demonstrated effective strategies [6, 9, 10, 11]). More importantly, we plan to focus on numerical computations with dynamic or irregular structure, such as finite element computations and adaptive grid refinements. Several other seemingly regular numerical computations also exhibit dynamic behavior due to indeterminacy in message arrival times [8] and critical path considerations.

In summary, Charm provides substantial support for solving irregular computations on parallel machines. We have illustrated its utility in the solution of a parallel branch&bound algorithm. Using Charm, researchers developing algorithms to solve irregular computations can leverage their efforts. They can concentrate on the algorithmic and application specific issues while leaving the task of machine dependent implementations, load balancing and prioritized scheduling to Charm.

References

[1] L. V. Kale. The Chare Kernel Parallel Programming System. In *International Conference on Parallel Processing*, August 1990.

[2] L. V. Kale *et. al.* Charm 3.2 Manual, available by anonymous ftp, along with the system, from a.cs.uiuc.edu, in pub/CHARM.

[3] Edward W. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[4] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11:972–989, 1963.

[5] L.V. Kale and Sanjeev Krishnan, *CHARM + + : A Portable Concurrent Object Oriented System Based on C++* Technical Report UIUCDCS-R-93-1796, March 1993, University of Illinois, Urbana, IL. (to appear in the Proc. of OOPSLA-93)

[6] L.V. Kale, *Parallel Problem Solving*, In *Parallel Algorithms for Machine Intelligence and Pattern Recognition*, 1989.

[7] Amitabh Sinha and L.V. Kale, *A Load Balancing Strategy for Prioritized Execution of Tasks,* International Parallel Processing Symposium, Los Angeles, Ca., April 1993.

[8] Edward Kornkven and Laxmikant Kale, *Dynamic Adaptive Scheduling in an Implementation of a Data Parallel Language.* Parallel Programming Laboratory Report #92-10, Department of Computer Science, University of Illinois at Urbana Champaign.

[9] B. Ramkumar, L.V. Kale, *Machine Independent AND and OR Parallel Execution of Logic Programs: Parts I and II* To appear in IEEE Transactions on Parallel and Distributed Systems.

[10] L. V. Kale and V. Saletore, *Parallel State-space Search for a First Solution,* Intl. Journal of Parallel Programming, Plenum Press, Vol. 19, 1990, pp. 251-293.

[11] Chin-Chau Low *Parallel Game Tree Searching with Lower and Upper Bounds.* M.S. Thesis Parallel Programming Laboratory Report #90-16 Department of Computer Science, University of Illinois at Urbana Champaign.

Wednesday, June 2, 4:45-5:45, Poster Session

Towards a virtual parallel branch and bound machine

Gerard Kindervater
Erasmus University, P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

Branch and bound algorithms have been implemented successfully on various types of parallel computers. However, the effort necessary to obtain good results must not be underestimated, and implementations tend to be highly architecture dependent. In our opinion, this situation is highly undesirable.

Therefore, we are presently working on the development of a high level 'parallel branch and bound machine', in which the user should bother himself only with problem specific parts of the branch and bound algorithm and where the machine should take care of an efficient execution on the parallel architecture to be used. We will present some of our ideas on how to achieve this goal.

Wednesday, June 2, 4:45-5:45, Poster Session

DYN-FO: A parallel, dynamic complexity class

Sushant Patnaik
University of Massachusetts, Amherst
Neil Immerman
University of Massachusetts, Amherst

Traditionally, computational complexity has worked in a "one-shot" paradigm: an algorithm is given a specific problem instance and solves it. All the Classic Complexity Classes: P, NP, PSPACE, etc. are defined in terms of the complexity of checking – upon presentation of an entire input – whether the input satisfies a certain property.

For many, if not most, applications of computers including: databases, text editors, program development, it is more appropriate to model the process as a dynamic one. There is a fairly large object being worked on over a period of time. The object is repeatedly modified by users and computations are performed.

Thus a dynamic algorithm for a certain class of queries is one that can maintain an input object, e.g. a database, and process changes to the database as well as answering queries about the current database.

Here, we introduce the complexity class, Dynamic First-Order Logic (DYN-FO). This is the class of properties S, for which there is an algorithm that can perform inserts, deletes and queries from S, such that each unit insert, delete, or query is first-order computable. This corresponds to the sets of properties that can be maintained and queried in first-order logic, i.e. relational calculus, on a relational database.

We investigate the complexity class DYN-FO. We show that some interesting graph properties including 1GAP, UGAP, and GAP (stCON) are in DYN-FO. We also show that k-vertex connectivity, k-edge connectivity and the computation of minimum spanning trees are in DYN-FO.

We also consider appropriate reductions for honoring dynamic complexity classes. We can prove that DYN-FO contains the regular sets. The expressive complexity of DYN-FO, in particular, whether it contains TC^0 (or even FO(Parity)), is still open.

Wednesday, June 2, 4:45-5:45, Poster Session

Work Efficient Techniques for the Parallel Execution of Sparse Grid-Based Computations

Jan F. Prins
University of North Carolina at Chapel Hill

Computations that consist of repeated evaluations of a local function to values arranged in a regular grid are easily and efficiently implemented on mesh-connected parallel computers using a hierarchical (block) domain decomposition. Local operations on the grid require only efficient near-neighbor communication on the mesh using such a decomposition.

When grids are large and elements requiring local function evaluation are sparsely distributed, the work performed using such a decomposition is non-optimal compared to an optimal sequential solution.

Use of a cyclic decomposition coupled with an algorithmic transformation of the program is shown to achieve work-efficient implementations for a broad class of sparse grid-based problems. The techniques are illustrated for a cellular automaton problem, and the performance of the implementation on a MasPar MP-1 is described.

Wednesday, June 2, 4:45-5:45, Poster Session

Neelakantan Sundaresan
and
Dennis Gannon
Department of Computer Science, Indiana University, Bloomington, IN-47405.

1 Introduction

We discuss programming language support for numerical problems whose algorithms are irregular in nature; dynamic in data structure and irregular in computation over that structure. We prefer object-oriented programming paradigm to a Fortran-like language because we can abstract out all the details of data layout, migration and efficient computation from the programmer. We discuss extensions to **pC++** [1], a *Collection* based object-oriented distributed parallel language, to achieve this task.

2 Problem Domain

We have studied the structure of adaptive mesh refinement techniques and hierarchical N-body methods. These constitute interesting classes of problems and algorithms where unnecessary data processing is eliminated by hierarchically refining the data domain and traversing this hierarchy based on some dependency conditions. This traversal can be parallel with multiple dependency criteria satisfied simultaneously. For instance, in the *V-cycle* scheduling of multilevel grid methods, the system cycles from the coarsest to the finest grid and back. All those nodes, which are ready for scheduling can be scheduled simultaneously.

3 pC++

Because C++ [2] is a de facto standard for a growing community of application designers,we have chosen it as our base language. Abstraction, performance, scalability and portability are the main objectives of **pC++**[1, 3]. The only extension to C++ is the *Collection* class and vector expressions. **pC++** supports both data-parallel and SPMD programming styles. Data mapping to processors is based on the High Performance Fortran model[4].

A *Collection* is a structured set of objects that are distributed over the processors and memories in a parallel system. *Collection*s are parameterized data types like the C++ templates classes where the element type of the *Collection* is the template parameter. Element methods can be executed in data-parallel fashion while the rest of the methods execute in SPMD mode.

The runtime model of **pC++** [5] is based on per-processor local collection and single active collection and single thread of computation per processor. Updates are based on "owner-computes rule". The only communication is to obtain part or whole of a remote element. Only one *Collection* is active at any instance and there is only one thread of computation per processor.

4 Support for Dynamic Structures and Adaptive Computation

A parallel DAG *Collection* and, its derived *Collection*, a parallel tree *Collection* form useful parallel data structures. The two-level specification of data alignment and distribution in **pC++** needs to be extended to support dynamic data structures (tree-shaped co-ordinate system and processor structure) and user-specified distribution and alignment. Realignment and redistribution become important for data locality and load balancing. One useful operator is *Traverse* which walks the structure in a way that the computation in a child node does not start before the parent computation is done. With adaptive computation, the corresponding *Collection* data structure changes shape over computational time; hence data redistribution and load balancing operators form an important part.

A level-wise parallel implementation of *Traverse* would mean loss of potential parallelism. Instead, each processor can have a dynamic state-list of all the nodes that it contains (a local sub-collection) with their states of execution. Each processor picks up an appropriate task from the list and takes necessary action according to what state the node task is currently in. Note that for any node its child may reside on a different processor, thus resulting in updating the list on another processor.

Since nested parallelism is natural to recursive parallel data structures, implementation of the same would require extension of this model to support multiple threads of computation per processor.

5 Conclusions

The support for dynamic parallel data structures for adaptive computation in **pC++** is very important in providing natural interface and efficient compilation. Our goal is to provide a good abstraction for such data structures and computation on such structures; mechanism for user-specified distribution, alignment, redistribution and realignment for such structures; and, support for nested parallelism and nested collections and efficient implementations of the same.

References

[1] Jeng Kuen Lee. *Object Oriented Parallel Programming Paradigms and Environments For Supercomputers.* PhD thesis, Indiana University, Bloomington, Indiana, 1992.

[2] Margaret Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, Reading, MA, 1990.

[3] François Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby Yang. Distributed pC++: Basic ideas for an object parallel language. Technical report, Indiana University, Bloomington, Indiana, 1992.

[4] High Performance Fortran Forum. Draft High Performance Fortran language specification, version 0.4. available by anonymous ftp from titan.cs.rice.edu, November 1992.

[5] François Bodin, Peter Beckman, Dennis Gannon, Shelby Yang, Allen Malony, and Bernd Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. Submitted to Supercomputing, '93.

Wednesday, June 2, 4:45-5:45, Poster Session

O^2SA Arrays for Parallel Sparse Matrix Factorization

Ricardo Telichevesky, Jacob K, White, and William J Dally
Massachusetts Institute of Technology

There is a great interest in exploiting the speed of parallel machines in order to accelerate sparse matrix factorization. For structured sparse matrices, approaches based on parallel nested dissection have been successful. Unfortunately, for applications like integrated circuit analysis, mechanical system simulation, and modeling electric power distribution, the associated matrices do not have such easily exploited structure.

Earlier work in this field [Huan79, Sada89] focused on multiprocessor scheduling, assuming that individual processors could access all sparse matrix data in uniform time. Other techniques were developed to reduce the total factorization time on a serial computer, but they present difficulties for parallel computers by constraining a parallel scheduling algorithm or requiring excessive main memory access speed. In order to alleviate these restrictions, we use the processor cache as a dynamic OSA structure, which evolves with the execution of the sparse matrix decomposition. In the beginning of the factorization, hundreds of small rows are scattered and overlapped, fitting in a small cache of size $O(n)$, allowing a large degree of freedom for parallel scheduling. Scattering into this small cache, on the other hand, insures fast code execution within each processing element. After all the updates to a target row are finished, its elements are gathered back in a dense vector and then other targets can be scattered in the cache. The factorization proceeds in this fashion, dynamically trading available cache space for concurrency. We call this approach the Overlapped-overlapped Scatter Array (O^2SA) technique.

We are currently developing a parallel computer with specialized datapath and control in each processor geared towards the execution of sparse matrix decomposition. It will profit from the O^2SA strategy by keeping an active data subset, necessary to achieve a high degree of concurrency in the sparse matrix factorization algorithm, in a small, fast memory which is tightly coupled with the floating point unit.

References

[Huan79] J. W. Huang, O. Wing, "Optimal Parallel Triangulation of a Sparse Matrix," *IEEE Trans. on Circuits and Systems,* pp. 726-732, Sept. 1979.

[Sada89] P. Sadayappan and V. Visvanathan, "Efficient Sparse Matrix Factorization for Circuit Simulation on Vector Supercomputers," *IEEE Trans. on Computer Aided Design,* pp. 1276-285, Vol. 8, No. 12, Dec. 1989

Wednesday, June 2, 4:45-5:45, Poster Session

Adaptive Parallelism with Piranha

Nicholas Carriero
David Gelernter
David Kaminsky
Jeffery Westbrook
Yale University

"Adaptive parallelism" refers to parallel computations on a dynamically changing set of processors: processors may join or withdraw from the computation as it proceeds.

Any parallel application might in principle gain if it is built as an adaptive program. In any computing environment (including a dedicated multiprocessor), such a program is capable of taking advantage of new resources as they become available, and of gracefully accommodating diminished resources without aborting. An adaptive parallel program might grow or shrink within a single multiprocessor, might encompass processors both within a multiprocessor and in a LAN, might jump from platform to platform and so on.

Networks of fast workstations are the most important setting for adaptive parallelism at present. Workstations at most sites are typically idle for significant fractions of the day, and those idle cycles may constitute in the aggregate a powerful computing resource. Although the Ethernet and comparable interconnects weren't designed to support parallel applications, they are capable in practice of supporting a significant range of production parallel codes. Ongoing trends make "aggregate LAN waste" an even more attractive target for recycling: desktop machines continue to grow in power; better interconnects will expand the universe of parallel applications capable of running well in these environments. For these reasons and others, we believe that adaptive parallelism is assured of playing an increasingly prominent role in parallel applications development over the next decade.

Adaptive parallelism is a poor fit to the assumptions underlying most parallel algorithm and program development models. Algorithms research targeted at adaptive parallelism is a necessity if algorithms are to keep pace with ongoing practical developments in parallel systems.

The "Piranha" system now up and running on a heterogeneous network at Yale [3] is a general-purpose adaptive parallelism environment. It has been used to run a variety of production applications, including applications in graphics, theoretical physics, electrical engineering and computational fluid dynamics.

A typical model for adaptive parallelism structures an application in terms of a fixed set of *processes* that are dynamically remapped among free processors (e.g. [2]). This approach, while theoretically appealing, has drawbacks in practice. General process migration is expensive, and it poses obvious difficulties on heterogenous networks of processors. Holding processes constant while the node pool contracts may lead to excessive process management overhead in the shrinking pool of active nodes.

Piranha, in contrast, is an adaptive version of master-worker parallelism [1]. Algorithm designers specify in effect a single general purpose "worker function." They do not create processes and their applications do not rely on any particular set of named active processes. When a processor becomes available, a new process executing the "piranha" function is created there; when a processor withdraws, a special "retreat" function is invoked, and the local piranha process is destroyed. Piranha functions perform *tasks*. A task consists of a fixed set of input data from which a worker function produces a set of output data and a collection of new tasks. Tasks may vary in complexity within an application, but a task is the finest grain of parallel execution. A worker executes exactly one task at any given time.

This approach has a number of advantages. Processes need never be moved around; the application collapses cleanly with little overhead into a degenerate "non-parallel" case, in which a single active process runs on a single participating node. Handling the degenerate case is important. Research should aim (we believe) at making adaptive parallelism a the default programming mode for *all* compute-intensive applications—which implies that the system must impose little or no overhead in the single-processor case.

References

[1] Carriero, N and Gelernter, D. *How to Write Parallel Programs: A First Course.* MIT Press (Cambridge: 1990).

[2] Chase, J.S., Amador, F.G., Lazowska, E.D., Levy, H.M., Littlefield, R.J. "The Amber System: Parallel Programming on a Network of Multiprocessors," Proceedings of the Twelth ACM Symp. on Operating Systems Principles, pp 147-158, 12/89.

[3] Gelernter, D. and Kaminsky, D., "Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha," Sixth ACM International Conference on Supercomputing, Washington D.C., July 19-23, 1992.

Thursday, June 3, 9-10

Performance/Memory Tradeoffs in Time Warp

Richard M. Fujimoto
College of Computing
Georgia Institute of Technology

The Time Warp mechanism has shown excellent promise in speeding up large scale discrete event simulation applications. A memory management protocol called Cancelback has been defined that enables Time Warp simulations to run with any fixed amount of memory that is at least that which is required by the corresponding sequential simulation. While it is generally acknowledged that performance will be poor if too little memory is provided, and good performance may be achieved with large amounts of memory, the performance/memory tradeoff in Time Warp executions when using cancelback are not well understood.

Our work presents results from an evaluation of the space-time tradeoffs in Time Warp augmented with the cancelback protocol for memory management. An implementation of the cancelback protocol on Time Warp has been developed that executes on a shared memory multiprocessor, a 32 processor Kendall Square Research Machine (KSR1). The implementation supports canceling back more than one object when memory has been exhausted. The limited memory performance of the system is evaluated for three different workloads with varying degrees of symmetry. These workloads provide interesting stress cases for evaluating limited memory behavior. It is observed that (i) depending on the available memory and asymmetry in the workload, canceling back several (called the *salvage parameter*) events at one time may improve performance significantly by reducing certain overheads, (ii) a performance nearly equivalent to that with unlimited memory can be achieved with only a modest amount of memory depending on the degree of asymmetry in the workload.

Thursday, June 3, 10:30-11

Abstractions for Parallel N-body Simulations
Sandeep Bhatt
Bellcore
Pangfeng Liu
Yale

Simulations involving irregular and adaptive structures, such as N-body simulations, typify applications which appear well suited for parallel machines, but which have generally been considered inconvenient to program efficiently.

We will discuss our recent experiences in implementing astrophysical simulations of several million particles on the Thinking Machines CM-5 and Intel hypercubes. We will describe our programming abstractions, discuss experimental results, and review some of the theoretical work on message-passing programs that was motivated by the project.

Thursday, June 3, 11-11:30

Arjen K. Lenstra, Thursday, June 3, 11-11:30

Factoring Integers using SIMD Sieves

Arjen K. Lenstra
Bellcore

In this talk I will discuss how massively parallel SIMD machines can be used for general purpose integer factoring. Our implementation is based on a surprisingly efficient random load balancing technique, and on more traditional pipelining strategies.

Thursday, June 3, 11:30-12

On Load Balancing Irregular Mesh Problems Using Field Functions
David M. Nicol
College of William and Mary
Dimitri Mavriplis
Institute for Computer Applications in Science and Engineering

Irregular meshes are increasingly used in scientific problems because they effectively focus the computation on regions of interest, where the solution changes rapidly in space. Such computations present interesting challenges for parallelization based on domain partitioning. The most popular existing approaches, binary dissection [1] and spectral dissection [2], are based on graph partitioning. Binary partitioning suffers from an artificial restriction that domain cuts be oriented along axis vectors (i.e., parallel to x, y or z axis). This (and insensitivity to communication costs) can led to partitions with excessive communication overhead. Spectral partitioning attacks the communication problem directly, by seeking graph partitions that balance the load and minimize the number of edges cut. Spectral methods have the drawback of high computational cost, and an underlying assumption of uniform execution weight per graph node.

In this talk we explore the idea of partitioning an irregular mesh by partitioning a continuous "field function" approximation to the workload. The idea is to partition a piecewise bilinear workload intensity function on the computation's domain such that the integral of this function over each resulting partition element is the same. This function arises naturally when using certain methods for mesh generation [3]. In fact, used in conjunction with this mesh generation method in 2D, our partitioning algorithm has a cost that is only linear in the number of mesh points. Furthermore, it has potential for parallelization, as well as guiding the parallelization of the mesh generation step.

1. M.J. Berger and S.H. Bokhari 1987. "A Partitioning Strategy for Nonuniform Problems on Multiprocessors" *IEEE Trans. on Computers* **C-36**, No. 5, May 1987, pp. 570-580.

2. A.Pothen and H.D. Simon and K.P. Liou 1990. "Partitioning sparse matrices with eigenvectors of graphs", *SIAM Journal of Mathematical Analysis and Applications*, **11**, 1990, pp. 430-452.

3. D.J. Mavriplis 1992. "An Advancing Front Delaunay Triangulation Algorithm Designed for Robustness", ICASE Technical Report 92-49, 1992.

Thursday, June 3, 1:30-2

Data Structures for Irregular Applications

Katherine Yelick
Soumen Chakrabarti
Etienne Deprit
Jeff Jones
Arvind Krishnamurthy
Chih-Po Wen
U.C. Berkeley

Irregular applications are challenging targets for parallelizing, and create the need for interesting distributed data structures other than arrays. We describe some of the major data structures that appear in some irregular applications, and the design decisions that are made in their representations. Unlike arrays, these data structures do not have simple mappings onto distributed memory multiprocessors. Moreover, because of the latency of interprocessor communication, an operation on a distributed data structure should not stall the invoking processor until the operation completes. This leads to a kind of *relaxed* consistency model for the data types, which is analogous to a weak consistency model for shared memory. In the discussion below, we begin with some background on what makes an application irregular, present an overview of the Multipol data structure library, and then give five examples of distributed data types that arise in at least one application.

Irregularity. Parallel programs may exhibit at least three different kinds of irregularity. The first kind of irregularity appears as *irregular control structures*, namely conditional statements and exception, which make it inefficient to use a synchronous data parallel model. A second source of irregularity is *data structures*, for example, unbalanced trees, graphs, and unstructured grids; in addition to being linked data structures, they may also change shape dynamically. These data structures lead to dynamic scheduling and load balancing requirements, since it is often impossible to predict the amount of computation that will be associated with a given data structure. The third type of irregularity comes from *communication*, which may be irregular in space, meaning that the patterns of communication are irregular, or irregular in time, meaning that there are no distinct communication and computation phases.

Implementing irregular applications on distributed memory machines requires mapping their major data structures across the machine. We are building a multiprocessor object library, called *Multipol*, of distributed data structures. Multipol is designed to provide high throughput data type operations with clean interfaces. The data types that exist so far use a combination of replication and partitioning to achieve the performance goals. The library hides much of the complexity, including concurrency control, message buffering, and consistency management. The library approach balances the trade-off between performance, which leads to machine specific implementations, and the conflicting goal of portability. While library implementations may be tuned to a particular architecture, the client application is portable across any machine on which the library is supported.

Bipartite Graph. The first example of a relaxed object comes from an electromagnetics problem, in which the main data structure is a bipartite graph [4]. The computation works in phases, reading one half of the graph while updating the other half, synchronizing, and then switching to the opposite side of the graph. Nodes of the graph are spread across processors. Operations on the graph allow for iteration over one of the two halves, writing to individual nodes, and forcing outstanding writes on either half of the graph to take effect. Thus, updates may happen any time between their initiation and the synchronization point at which the writes are forced. Although the algorithm is loosely synchronous, this mixture of communication and computation within a timestep is used to maximize the use of communication bandwidth.

Time Warp Object. The second example of a relaxed object is a *time-warp* object, first introduced by Reiher and Jefferson [5]. The essential data structure is the simulation state, which is partitioned and distributed across processors. For example, in a circuit simulator, the circuit is partitioned into subcircuits based on connectivity defined by direct voltage connections [7]. Although the contents of the state, its partitioning, and the algorithms for simulating a time step are specific to the simulation domain, the basic state operations are the same for any speculative simulation. The operations on the state are reading and writing a substate at a given timestep. Hidden within the data structure are mechanisms for handling speculative simulation, including rollbacks of incorrect states and garbage collection of old states. In this example, the relaxed nature of the data structure comes from speculation: the current value of the state is not necessarily seen when the object is read, but rather some approximation is used. In the end, only those steps based on legitimate values will have any effect on the result.

Relaxed Set. Another interesting example is a *set*, a container of unique, unordered objects. The insert operations on sets is difficult to implement with high throughput, because processes must check for duplicates, in some generalized sense, within the set and also guarantee that no other process is simultaneously inserting the same element. We use a test-and-test-and-set style interface, in which a processor checks for duplicates without holding a lock, and if that check succeeds, a lock is acquired and the set checked one more time. This data structure is used in a symbolic algebra application, in which a set of polynomials is being constructed [2, 3]. For this application, reading the elements of the set is much more common than inserting into the set, so a replicated design is used. The test for duplicate elements is more complicated than a simple equality test, and in many cases duplicates can be found by looking at only a subset of the elements. Therefore, the replicas of the set are kept only loosely synchronized, with a processor forcing the local copy to be made consistent only for the final test before a insert succeeds.

Task Queue. A common example of a relaxed object is a *task queue* is a container for holding and distributing tasks, e.g., [6] and others. In a sequential environment, a task queue would be a priority queue, but in a parallel environment, adhering to strict priorities leads to contention for the queue and high communication overhead. In many applications, strict priorities, or strictly FIFO behavior, is a performance requirement and not a correctness requirement. Thus, the task queue is really an unordered collection of objects in which priorities are used as hints; the priorities are locally, but not globally, observed. The implementation is a set of local priority queues distributed across processors. The task queue attempts to keep all queues partially full, so that a processor will not have to wait for a task to become available. A termination detection protocol is built into the task queue, so that the user can query the task queue to see if all tasks have been completed.

Hash Table. The final example of a relaxed object is a global hash table. This is used in a Phylogeny tree application, which computes the evolutionary tree of a set of species, subject to certain constraints [1]. The algorithm is essentially a divide and conquer algorithm, which proceeds by subdividing the set of species, finding smaller phylogeny trees on the subsets, and combining them into a larger tree. However, the same intermediate problem might be generated multiple times during the search. The global hash table is used to store the phylogeny trees for subsets of species, so that they only need to be computed once. The hash table is partitioned, as opposed to being replicated like the set. This has the advantage of minimizing memory utilization, because the table is potentially very large, but it requires communication for most table lookups. Future designs of both the set and the hash table may use a hybrid approach, in which there is some amount of dynamic caching, but complete copies are not used.

The Multipol library is an ongoing project. To date, we have a small set of data structures and complete applications that use them. The implementations have all been done on a CM5 multiprocessor and performance of the applications is very good. The library will grow as new data structures are identified. In addition, we are exploring opportunities for using high level languages and advanced compilation technology for applications that use the library.

References

[1] Richa Agarwala and David Fernandez-Baca. A polynomial-time algorithm for the phylogeny problem when the number of character states is fixed. Draft manuscript.

[2] Soumen Chakrabarti. A distributed memory gröbner basis algorithm. Master's thesis, University of California, Berkeley, CA, 1992.

[3] Soumen Chakrabarti and Katherine Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *Principles and Practice of Parallel Programming*, San Diego, CA, 1993. to appear.

[4] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. Submitted for publication, 1993.

[5] Peter Reiher and David Jefferson. Dynamic load management in the time warp operating system. *Transactions of the Society for Computer Simulation*, 7(2):91–120, June 1990.

[6] Amitabh Sinha and L. V. Kale. A load balancing strategy for prioritized execution of tasks. In *Proceedings of the Workshop on Dynamic Object Placement and Load Balancing, ECOOP'92*, June 1992.

[7] Chih-Po Wen. Parallel timing simulation on a distributed memory multiprocessor. Master's thesis, University of California, Berkeley, CA, 1992. Master's Report.

Thursday, June 3, 2-2:30

Runtime Support for Symbolic Parallel Algorithms

Suresh Jagannathan
NEC Research Institute
James Philbin
NEC Research Institute
Monika Rauch NEC Research Institute

Despite significant advances in hardware construction and parallel algorithm design, there remains a large class of algorithms which have typically performed poorly on conventional stock multiprocessors. These problems (broadly classified under the name *symbolic programming*) include combinatorial optimization, simulation, symbolic algebra, graph algorithms, expert systems, alpha-beta search, etc. Symbolic programming applications exhibit characteristics not found in more well-structured numerical algorithms: data is generated dynamically and often have irregular shape and density, data sets typically consist of objects of many different types and structure, the natural unit of concurrency is usually much smaller than can be efficiently supported on stock hardware, and sensible decomposition of the program into parallel threads of control often cannot be achieved by mere examination of the source text.

Since different kinds of symbolic algorithms impose vastly different requirements on the underlying implementation vehicle, efficient realization of these applications is often difficult in the context of a single system. Applications implemented via a high-level parallel language, for example, perform well only if their runtime characteristics match closely with the strengths of the runtime system, and the optimizations performed by the language's compiler. Any reasonable language implementation invariably will be tailored specifically for the concurrency constructs defined by the language; if these constructs are not sufficiently general and not suitable for expressing the paradigm of interest, efficiency suffers and program complexity increases. For most parallel languages, optimizations incorporated into an implementation to support a particular programming methodology often are too specialized to be applicable to others.

One obvious way of addressing some of these problems is to implement symbolic algorithms via generic thread package or operating system-defined library routines. Such approaches offer greater extensibility and generality than available within a high-level language optimized for a particular paradigm. However, efficiency is compromised in two obvious respects: (1) because there is no compile-time or runtime system sensitive to the semantics of the concurrency abstractions being implemented, there is little or no optimization performed on these abstractions; (2) because scheduling and migration policies, storage management, etc., are managed exclusively by generic thread routines, building customized implementations of different paradigms is problematic. The latter shortcoming makes this approach especially ill-suited for implementing diverse parallel symbolic algorithms.

In this paper, we propose an alternative implementation strategy. STING is a parallel dialect of Scheme that permits the efficient expression of a number of diverse concurrency paradigms typically used by symbolic parallel algorithms. STING achieves this goal by (a) providing a high-level abstraction for defining scheduling decisions for lightweight threads created by applications, (b) aggressively supporting data locality in its storage model, and (c) allowing user control over local and inter-address space management. The goal of the STING design is to provide a small collection of efficient building blocks upon which many high-level concurrency abstractions can be easily implemented. In this sense, it supports the functionality of a multi-threaded operating system environment. However, because its abstractions are completely integrated into Scheme, opportunities for compile-time and runtime optimization can be identified and exploited within the context of a high-level programming language. As we discuss below, this kind of flexibility is crucial in the context of the applications cited above.

We describe a series of applications implemented in STING that use different process structure (coarse and fine-grained), different synchronization protocols (message passing, distributed data structures, and dataflow), and different scheduling regimes (FIFO, LIFO, randomized, global and local queues). Our results indicate that despite the different runtime requirements imposed by these applications, efficiency is not compromised. No alteration to the STING runtime system is necessary in order to implement these algorithms. We therefore envision STING as a basis upon which diverse concurrency paradigms for symbolic computing may be efficiently implemented and competitively analyzed.

References

[1] Thomas Anderson, Edward Lazowska, and Henry Levy. The Performance Implications of Thread Management Alternatives for Shared Memory MultiProcessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[2] Eric Cooper and Richard Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie-Mellon University, June.

[3] Eric Cooper and J.Gregory Morrisett. Adding Threads to Standard ML. Technical Report CMU-CS-90-186, Carnegie-Mellon University, 1990.

[4] David Greenberg. *Full Utilization of Communication Resources*. PhD thesis, Yale University, 1991. Also published as Yale University Computer Science Technical Report, YALEU/DCS/TR-860.

[5] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[6] Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 128–137, March 1990.

[7] Suresh Jagannathan and James Philbin. A Customizable Substrate for Concurrent Languages. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.

[8] Suresh Jagannathan and James Philbin. A Foundation for an Efficient Multi-Threaded Scheme System. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, June 1992.

[9] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 110–21. Association for Computing Machinery SIGOPS, October 1991.

[10] James Philbin. *The Design of an Operating System for Modern Programming Languages*. PhD thesis, Dept. of Computer Science, Yale University, 1992.

[11] Mitch Wand. Continuation-Based MultiProcessing. In *Proceedings of the 1980 ACM Lisp and Functional Programming Conference*, pages 19–28, 1980.

Thursday, June 3, 2:30-3

Experiences with a Parallel Data Flow Algorithm

Dr. Barbara G. Ryder
Department of Computer Science, Rutgers University

Current optimizing compilers for parallel machines can take hours to compile programs of reasonable size. A large proportion of the time is spent in computing data flow and program dependence information, which is crucial in the generation of efficient code. For usability, these long compilation times must be greatly shortened.

We propose to utilize the resources of the parallel target machine for the analysis phase of the compilation process. Our parallel data flow analysis algorithms are based on a family of data flow analysis algorithms called hybrid algorithms. Hybrid algorithms embed natural parallelism that can be systematically parallelized. This talk reports our implementation experiences with our algorithm on intraprocedural and interprocedural data flow problems.[1]

[1] The research reported was done jointly with Dr. Yong-fong Lee, and supported, in part, by National Science Foundation grants CCR-8920078 and CCR-9023628-2/5.

Thursday, June 3, 3:30-4

Recovering Structure in Damaged Networks

Bruce Maggs
NEC Research Institute

This talk shows that a parallel computer whose underlying communication network is a tree, mesh, or butterfly can sustain many faults and still emulate a fault-free computer of the same size and type with little slowdown. The type of faults that we consider are failures of the nodes (processors) and edges (communication links) of the network. We assume that faults are static and detectable, and that information regarding the locations of the faults can be used to reconfigure the network. We show, for example, that even if an adversary is allowed to place a set of $N^{1-\epsilon}$ worst-case faults in an $N \times N$ mesh, for any fixed $\epsilon > 0$, the network can still emulate a fault-free $N \times N$ mesh with only constant slowdown. Before this research, there was no example of any connected bounded-degree network that could sustain more than a constant number of worst-case faults without suffering more than constant slowdown.

The research described in this talk was done in collaboration with Richard Cole, Tom Leighton, and Ramesh Sitaraman.

Thursday, June 3, 4-4:30

Competitive Distributed Paging
Baruch Awerbuch
Lab. for Computer Science, MIT.
Yair Bartal
Dept. of Computer Science, Tel-Aviv University,
Amos Fiat
Dept. of Computer Science, Tel-Aviv University,

The goal of this paper is to extend the virtual memory concept from the traditional uni-processor to distributed shared-memory environment. In other words, the programmer can use the convenient Parallel Access Machine (PRAM) abstraction to write the program, which will be then compiled automatically to run on a network-based distributed memory machine.

Namely consider the problem of distributed paging in a distributed-memory multi-processor (or a distributed file server) where space sharing between processes is allowed. The traditional file server is thus replaced by the "aggregate cache", i.e. the conjunction of the local caches. In the context of parallel multiprocessors, this setting is referred to as *Cache-Only Memory Architecture* (COMA), and it has has been employed in a number of recently developed parallel machines such as the Data Diffusion Machine and the KSR1 machine.

In such scenario, it is the joint responsibility of all the processors to do efficient space-sharing of their local caches, maintaining the overall global consistency constraints. The goal of the paging algorithm is to minimize number of remote network accesses in the presence of dynamic and unpredictable access pattern. We show how an online paging algorithm whose performance, *on each input sequence*, is within a logarithmic factor from that of the a prescient strategy, optimized on that instance. We also show that logarithmic gap between online and offline is inherent.

Thursday, June 3, 4:45-5:45, Poster Session

Mapping of Branch and Bound Algorithms onto
Shared and Distributed Memory Multiprocessors

Tarek S. Abdelrahman
Department of Electrical and Computer Engineering
University of Toronto

In this paper our results of mapping Branch and Bound (BB) algorithms with a best-first search strategy onto shared and distributed memory multiprocessors will be presented. BB algorithms present a challenge in their mapping onto multiprocessors because of their irregular and dynamic nature. A BB algorithm expands a search tree whose shape is irregular and cannot be predicted in advance. This implies that the division of work among the processors must be done dynamically at run-time. Furthermore, a BB algorithm requires some global knowledge about the lower bounds of subproblems to maintain its efficiency. This is difficult to achieve in distributed memory and can result in contention in a shared memory.

Experimental results from a 64-processor hypercube for three application problems (Travelling salesman, 0-1 integer programming and 8-puzzle) will be presented. The results show the performance of a new parallel BB algorithm, and compare it to the performance of two other algorithms that have been proposed in the literature. The factors affecting the performance of the three algorithms will be discussed. The new algorithm achieves superior and scaling performance by utilizing a tradeoff between the cost of communication and the cost unnecessary computations.

We will also present experimental results of mapping the algorithms onto a Kendall Square Research multiprocessor and show that the performance of the algorithms is limited by contention over shared data structures, and by the lack of spatial and temporal locality in the algorithm. The results suggest that even on a shared memory multiprocessor, it will be more advantageous to use strategies similar to those employed on distributed memory multiprocessors.

Thursday, June 3, 4:45-5:45, Poster Session

SIMD Tree Search: The Untold Story

Srinivas Aluru
Ames Laboratory, Iowa State Univ.
John Gustafson
Ames Laboratory

In this paper, we bring to light some of the subtle issues in SIMD tree search that have been ignored so far by researchers. SIMD tree search can be broken down into three parts: A work generation phase to distribute work to the idle processors, a search phase during which processors search their respective subtrees and a triggering mechanism to decide when to switch between the two phases. Although both the work generation phase and the triggering mechanism are discussed in the literature, the search phase has been taken for granted to be fully parallelizable. We demonstrate that the problem of processors being idled by control differences during the search phase could be significant in complicated tree search problems involving a large number of pruning rules. We present techniques to alleviate this problem and compare their performance by implementing a complicated search problem on the SIMD MasPar computer.

Thursday, June 3, 4:45-5:45, Poster Session

Systems and Algorithms for Parallel and Distributed Symbolic Algebraic Computation
Bruce W. Char Jeremy Johnson
Department of Mathematics and Computer Science
Drexel University
Philadelphia, PA 19104
B. David Saunders
Department of Computer and Information Science
University of Delaware
Newark, DE 19711

The goal of our research is the effective parallelization of symbolic algebra in a variety of parallel architectures. To further this goal, we are studying algebraic operations of fundamental importance and general usefulness: basic linear algebra operations, high-level symbolic polyalgorithms, and cylindrical algebraic decomposition, among others. We observe that for many types of typical algebraic computation, problems of similar size have non-uniform run time and memory usage, with a priori prediction of resource requirements difficult. Furthermore, these problems offer an extensive choices in granularity due to the hierarchical nature of the algebraic domains in which they are posed. Opportunistic scheduling and dynamic load-balancing would seem to be useful strategies in making effective use of parallel and distributed computing in computer algebra.

Our vehicle for exploration is *Sugarbush*, a version of the Maple symbolic mathematics system using C/Linda. Its design, evaluation, and refinement is an integral part of the project. We are interested in having *Sugarbush*provide scalable libraries of parallel routines: easily accessible large and medium-grained parallelism for multiprocessor, multicomputer, or distributed computation architectures. Its nature allows straightforward inclusion of other C routines as modules. *Sugarbush*should provide inclusion of other C routines as modules. *Sugarbush*should provide accessible parallelism – large and medium grain parallelism on multiprocessor, multicomputer and distributed computation architectures through Maple, and smaller-grained parallelism for polynomial operations on shared memory multiprocessors through links to other algebraic routines such as those provided by SAC2-C.

Thursday, June 3, 4:45-5:45, Poster Session

An Almost Regenerative Simulation of Video Teleconferencing Traffic

David M. Cohen
Daniel P. Heyman
Bellcore
Morristown, NJ 07960

This note discusses the application of an almost regenerative method to the simulation of a Markov chain that takes a very long time to converge. The Markov chain models cell loss in a buffer used to multiplex video teleconferencing traffic on an Asynchronous Transfer Mode (ATM) network. For a cell-loss-rate of the order of 10^{-8}, even runs of 10^{12} cells didn't always see the same cell-loss-rate.

The regenerative method [2] [4] could not be used here to find the steady state probability because the time between regenerative states is too large (10^{30} events). Instead, an almost regenerative method was tried using a massively parallel processor and it gave results that agreed with predictions from regenerative theory and with some very long simulation runs. Unlike the work of Crane and Iglehart [3] and Gunther and Wolff [5], this process is discrete time and required a different way to define the regenerative event.

References

[1] D. M. Cohen and D. P. Heyman, "Performance Modeling of Video Teleconferencing in ATM Networks", to appear in *IEEE Transactions on Circuits and Systems for Video Technology*

[2] M. A. Crane and D. L. Iglehart, "Simulating Stable Stochastic Systems: III. Regenerative Processes and Discrete-Event Simulations," *Operations Research*, **23** (1975), pp 33 - 45.

[3] M. A. Crane and D. L. Iglehart, "Simulating Stable Stochastic Systems, IV: Approximation Techniques," *Management Science*, **21**, (1975), pp 1215-1224.

[4] G. S. Fishman, "Estimation in Multiserver Queuing Simulation," *Operations Research*, **22** (1974), pp 72 - 78.

[5] F. L. Gunther and R. W. Wolff, "The Almost Regenerative Method for Stochastic System Simulations," *Operations Research*, **28** (1980), pp 375 - 387

Thursday, June 3, 4:45-5:45, Poster Session

Impact of Memory Contention on Dynamic Scheduling on NUMA Multiprocessors

M. D. Durand
Bellcore, Morristown, NJ
T. Montaut, L. Kervella, W. Jalby
IRISA, Rennes, France

Self-scheduling is a method for dynamic task scheduling in parallel programs, in which each processor acquires a new block of tasks for execution whenever it becomes idle. To get the best performance, the block size must be chosen to balance the scheduling overhead against the load imbalance. To determine the best block size, better analytical models of self-scheduling are needed.

We present experimental measurements of self-scheduling on a Non Uniform Memory Access (NUMA) machine (a BBN TC2000). The results show that on NUMA machines, memory latencies are large and variable. Hence, task duration is extremely unpredictable and dynamic methods such as self-scheduling are imperative. Previously published models to predict running time and optimal block size were tested using our experimental results. Although the models gave good predictions for small block sizes, for large block sizes the models fail, underestimating the running time by almost a factor of 2. We present an upper bound on the running time and use it to explain this failure. Possible solutions are discussed.

Thursday, June 3, 4:45-5:45, Poster Session

Yu Gong
Assistant Professor
Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849

In the last decade, a great deal of efforts has been made towards evaluating the performance of the Time Warp (TW) mechanism, an optimistic mechanism for scheduling Parallel Discrete Event Simulations (PDES). The performance of this mechanism, however, is still not well understood. A large number of analytical evaluations assumed two processor systems and/or single logical process per processor. Recently, the system with multiple processors and single process per processor was analyzed. The most realistic setting for the TW mechanism would be a system with multiple processors and multiple processes per processor. In this setting, processes in the same processor compete for the shared resource. Thus, the progress speed of a PDES program is determined by rollback, communication delay, AND processor sharing. Analysis of this general system is difficult due to the extremely high analytical complexity. In our work, an approximate analysis is conducted. We cope with the analytical complexity using a decomposition technique, which groups multiple messages into one logical "message" in order to reduce the system size. The interaction of the resulting "messages" which have special statistical behaviors, is studied in the second step. Using this divide-and-conquer method, we are able to derive closed form formulas for the performance measures of interest such as speed up and probability of rollback in each process, etc. The analytical results are compared with experimental results obtained from a TW testbed (on a 32 processor BBN Butterfly) in Georgia Institute of Technology. The results show a close match between the two results, especially on the speedup performance.

Thursday, June 3, 4:45-5:45, Poster Session

Performances of Parallel Branch and Bound Algorithms with Best-First Search
Bernard Mans
Carleton University, Ottawa, Canada & INRIA, France

We analyse the performances of the Branch and Bound algorithm (B&B) with best first search strategy, which is commonly used to solve NP-hard combinatorial optimization problems. We study the three main anomalies on the expected speed-up which may occur when the parallelism is introduced to increase the speed of the execution : detrimental, acceleration and detrimental acceleration.

We recall why the best evaluation is not always sufficient to distinguish the best node to choose with best first search strategy. In order to analyse such anomalous speed-up, we define three basic rules designed for the distinct selection of same bound nodes: the fifo, the lifo and the consistent rules.

We derive tight bounds of each of the potential parallel executions, and therefore give the sufficient and necessary conditions regarding the predisposition for each of the three rules to reach anomalous parallel behaviour. A propriety of "proneness to anomaly", which allows several comparisons and choices, is deduced.

Results presented can be also found in INRIA-RR-1716 (French Release).

Thursday, June 3, 4:45-5:45, Poster Session

Communication Structures for Parallel Computing in Graphs

Bojana Obrenic
University of Massachusetts

Emulations are algorithms for automatic translation of parallel programs designed for one interconnection network to another. Graphs that underly many practical problems appear "unstructured" because of the sophisticated communication dependencies between objects in these problems. Such graphs are difficult to emulate on realistic networks, whose highly regular structure reflects various implementation concerns. This structural mismatch is analogous to the traditional dichotomy between assembly-level instruction sets (which correspond to emulator edge-sets) and user-level computation descriptions (which correspond to problem graphs.) Many current emulation algorithms resemble long and tricky assembly-level programs that match specific problems to individual networks.

We define communication structures as graphs whose edge-sets are specified only implicitly, by specifying communication primitives (emulation sequences) that the graph can perform, along with the time-cost of performing them. Such structures serve as intermediate emulators that break the original emulation problem into two independent phases. In the first phase the user-level communication is expressed as a composition of the primitives of the chosen communication structure. In the second phase these primitives are ported to a specific emulator. By manipulating generic communication primitives instead of specific edge sets, the user-level communication programming gains coherence and readability. By focusing on the support of selected communication primitives, programming specific emulators is simplified.

Cell graphs are communication structures which we have introduced and used in the analysis and conceptual design of parallel architectures. We present new emulators and efficient emulation algorithms for: *(a)* arbitrary bounded-degree graphs having sublinear separators, *(b)* virtual, dynamically reconfigurable parallel architectures.

REFERENCES

1. S.N. Bhatt, F.R.K. Chung, J.-W. Hong, F.T. Leighton, B. Obrenić, A.L. Rosenberg, E.J. Schwabe (1992): Optimal emulations by butterfly-like networks. *J. ACM*, to appear.
2. B. Obrenić (1992): An approach to emulating separable graphs. *Math. Syst. Th.*, to appear.
3. B. Obrenić (1993): Cell graphs for managing communication in parallel computing. Ph.D. Dissertation, Univ. Massachusetts, in preparation.

Friday, June 4, 9-10

Subway, a communication compiler for the Maspar MP-x computers.
Robert Schreiber
Research Institute for Advanced Computer Science
Petter Bjørstad
University of Bergen

Unstructured communication on distributed memory parallel computers is difficult. The problem of multiplying a sparse matrix by a vector contains all the essential difficulties, encompasses most of the applications, and can be viewed as perhaps *the* fundamental kernel in parallel scientific computing. There are two basically different situations: the matrix is used once, or the matrix (or at least its nonzero structure) is used repeatedly. In the former situation, no distributed memory machine has been efficient. In the later, several preprocessing optimizations can make these machines efficient.

The preprocessing consists of placement and routing. In the placement phase, the data are assigned to the nodes so as to minimize some easily obtained estimate of the resulting communication burden, subject to a load balance constraint. Two essentially different ideas have proven useful: mapping rows (equivalent to mapping the bodies on an N-body problem or the vertices of a graph) and mapping blocks (equivalent to mapping the forces of an N-body problem or the edges of a graph). The former is appropriate only to matrices whose structure is well behaved — having small separators — such as those that come from PDE applications. We consider such matrices in the talk. The later has lead to some very interesting implementations of sparse matrix-vector product lately by Aiello and Ogielski, Lewis and van de Geijn, and Hendrickson, Leland, and Plimpton.

Given a placement, the set of communications (messages) required by the sparse matrix vector product is determined. The problem is to implement these messages with the underlying architecture as efficiently as possible. The tools used are packetizing, routing and scheduling of the messages. Work of this kind for MIMD machines due to Johnsson, Ho, Mathur, Saltz, van de Geijn, Fox and others addresses issues of maximal utilization of the wires in the machine (exploitation of the bandwidth) and minimization of the number of distinct messages (avoidance of latency). Similar work for the CM-2 by Dahl also involved maximal use of the machine bandwidth. Much of this work assumed an underlying hypercube topology. All of it showed very significant speedups compared to naive message passing relying on dynamic routing by the hardware.

Petter Bjorstad and I have developed a router/scheduler for arbitrary communication on the Maspar. We ignore completely the hardware router and instead use the 8-nearest neighbor grid communication, which has a bandwidth that is 16 – 50 times greater than the router's. The Maspar is a synchronous, SIMD, 128 X 128, Cartesian torus (a city consisting of stations) in which a collection (a train) of messages of identical size, and going in the same direction, can leave the processors at each cycle. Our work provides preliminary answers to these questions:

* Should the routing be adaptive or oblivious?

* How important are diagonal connections in a mesh?

* Should long distance messages (express trains) be used?

* Is fan-out (from one source too several destinations) important?

* What is the storage cost of such a system?

* Is the preprocessing time excessive?

* Can a massively parallel SIMD system be useful for unstructured meshes and other irregular problems?

Friday, June 4, 10:30-11

Very Large Travelling Salesman Problems

David Applegate
AT&T Bell Labs
William Cook
Bellcore
Bernhard Korte
Universität Bonn
Mike Schäfer
Universität Bonn

We describe new parallel methods for obtaining approximate solutions to very large travelling salesman problems, together with bounds showing that the proposed solutions are not far from optimality. With our implementation on a network of unix workstations, we were able to find a solution to a 10,907,064 city problem (arising from a VLSI application) and a proof that the length of the solution is no more than 4 percent from the optimal length.

Friday, June 4, 11-11:30

Scalable Parallel Search Algorithms for Combinatorial problems

Vipin Kumar
University of Minnesota

Search algorithms are the best known sequential algorithms for many important Discrete Optimization Problems. We have developed parallel formulations of depth-first search, depth-first branch-and-bound, Best-first search, iterative deepening-A* algorithms for a variety of architectures. These formulations were tested in the context of several abstract and practical problems such as the 15-puzzle, the n-Queens problem, the Travelling Salesperson problem, the vertex cover problem, VLSI floor plan optimization, VLSI test pattern generation, tautology verification. For many of these problems, linear speedup was demonstrated on a variety of commercially available parallel computers including a 1024 processor Ncube, 128 processor BBN Butterfly, and a network of 16 SUN workstations. This work led to development of several new methods of distributing work among different processors as well as concurrent data structures which are useful for a variety of parallel algorithms.

This talk summarizes our research on development of parallel search algorithms for solving Discrete Optimization problems, and the evaluation of their performance and scalability on various parallel architectures.

Friday, June 4, 11:30-12

Branch-and-Bound and Backtrack Search on Processor Arrays
Christos Kaklamanis
DIMACS
Giuseppe Persiano
Harvard University

We investigate the parallel complexity of the backtrack and branch-and-bound search on mesh-connected arrays.

We present an $\Omega(\sqrt{dN}/\sqrt{\log N})$ lower bound for the time needed by a *randomized* algorithm to perform backtrack and branch-and-bound search of a tree of depth d on the $\sqrt{N} \times \sqrt{N}$ mesh, even when the depth of the tree is known in advance. The lower bound holds also for algorithms that are allowed to move tree-nodes and create multiple copies of the same tree-node.

For the upper bounds we give *deterministic* algorithms that are within a factor of $O(\log^{\frac{3}{2}} N)$ from our lower bound. Our algorithms do not make any assumption on the shape of the tree to be searched, do not know the depth of the tree in advance, and do not move tree-nodes nor create multiple copies of the same node.

The best previously known algorithm for backtrack search on the mesh was randomized and required $\Theta(d\sqrt{N}/\log N)$ time. Our algorithm for branch-and-bound is the first algorithm that performs branch-and-bound search on a sparse network. Both the lower and the upper bounds extend to higher-dimension meshes.

Friday, June 4, 1:30-2

Min-Cost Flows: A Case Study in Increasing Parallelism in Solving Unstructured Graph Problems

B. Narendran
Renato De Leone
Prasoon Tiwari
Computer Sciences Department
University of Wisconsin-Madison

We discuss the parallelization of the ϵ-relaxation algorithm for the min-cost flow problem, and its implementation on the Connection Machine CM-5. Due to the unstructured nature of the computations, a naive parallelization results in very low and variable parallelism. The effectiveness of a set of algorithmic augmentations to the naive parallelization is investigated. These augmentations work by either eliminating non-parallel iterations of the basic algorithm, or increasing the parallelism of the remaining iterations. An important issue is when and how often these augmentations are to be applied to get the maximum benefit. Given the lack of structure, adaptive schemes that decide based on the current dynamic state of the algorithm are found to work the best.

Friday, June 4, 2-2:30

Shared Memory Implementation of Network Flow Algorithms

Richard Anderson
University of Washington

The motivating question for this research was to determine if the network flow problem could be solved "effectively" on a shared memory parallel machine. The goal was to demonstrate a reasonable parallel speedup over a good sequential implementation of a network flow algorithm.

Several studies have demonstrated conclusively that Goldberg's Push-Relabel algorithm is the fastest known sequential algorithm. Golberg's algorithm operates by moving flow towards the sink using a series of local operations. However, to get good practical performance, it is necessary to periodically perform a global relabelling of the graph. (This is best called a heuristic since it does not improve worst case run time, but gives several orders of magnitude speedup in practice).

In an earlier work [Anderson, Setubal, SPAA 92], we considered an implementation of Goldberg's algorithm on a Sequent Symmetry shared memory multiprocessor. One of the main algorithmic results of this work was to devise a method for incorporating global relabelling into a parallel algorithm. The speedup results that we achieved were relatively good, we saw parallel efficiency of up to 50processors. The difference between this and full speedup were due to lock overheads, lack of parallelism in the application, and hardware effects such as bus contention.

In our current work, we are attempting to extend the results to a KSR shared memory machine. Although both of these machine classify themselves as shared memory machines, they have drastically different architectures. The Sequent Symmetry is a bus based parallel machine, so that processors communicate to global memory over a single channel. The processors themselves have relatively small caches. This has the effect, on applications such as network flow, of providing uniform cost memory access. The KSR architecture is to divide all of the global memory between the processor caches, so that there is a very big difference between local and non-local memory access. However, in constrast to a distributed memory machine, the user does not have explicit control on where the memory resides.

We will be studying whether or not it is possible to get reasonable performance on the KSR. The memory access pattern in a network flow algorithm is unstructured, which substantially increases the challenge in getting good performance on parallel machine that does not provide uniform cost memory access. The initial KSR runtimes are not particularly encouraging. The figures for one 10000 node graph are:

Processors	Time	Speedup
1	20.7	1.00
2	25.1	0.82
4	18.9	1.09
8	12.9	1.60
16	9.6	2.15
24	10.7	1.93

However, increasing the number of vertices to 90000 gives figures with improved speed up numbers:

Processors	Time	Speedup
1	471	1.00
2	365	1.29
4	278	1.69
8	191	2.46
16	148	3.18

(The one processor algorithm can be sped up by about 25% by removing locks, at which point it is essentially the sequential implementation). These results are from a direct port of the Sequent code, with not attempts at optimization on the KSR. One important feature is the increase in runtime between the one and two processor cases. This is presumably do to the cost of transfering data between the two processors.

The size of graph chosen was a bad case for achieving parallel speedup results since the graph is small enough to fit in the memory of a single processor. However, when the graphs get bigger than what can fit in a single processor memory, the one processor algorithm slows down (giving better observed speedups!). This raises a number of experimental issues as to what types of performance is being measured. The interesting cases are graphs that are large enough to require a moderate amount of computation and are larger than a single processor's memory. However, it is turning out to be a problem to understand even the single processor performance on large graphs. (A factor of 3 difference in runtime has been observed on the same input, depending upon whether or not the operating system intervenes and moves the thread from one processor to another.) Software system upgrades have also changed performance substantially (in the correct direction!), which has made it difficult to get a consistent set of experimental numbers.

Friday, June 4, 2:30-3

Shared Memory Support for Unstructured Problems
Phillip B. Gibbons
Michael Merritt
AT&T Bell Laboratories

The performance of a multiprocessor is heavily dependent on its ability to tolerate large latencies in interprocessor communication. Multiprocessors that support a shared memory abstraction often provide each processor with a cache for shared data, so that subsequent references to the same data can be fast. For problems with regular structure, long cache lines can be used to fetch a block of memory locations, exploiting the fact that subsequent reads or writes will likely reference the same cache line, and hence be fast. Unfortunately, unstructured and dynamic problems do not exhibit such good spatial locality, and other techniques are needed to overcome the large memory latency.

We consider shared memories in which scattered memory locations can be fetched or updated in a pipelined manner. These nonblocking shared memories permit a processor to have multiple reads/writes in progress at a time, obtaining good performance even in the absence of spatial locality. We describe a simple programming abstraction, based on sequential consistency, for nonblocking shared memories. We show how particular implementations of nonblocking shared memories, including those providing caches and/or supporting weak consistency models, can be formally specified and proven correct.

Friday, June 4, 3-3:30

Designing Algorithms for a General Model of Parallel Computation

Clyde Kruksal
University of Maryland
Pilar de la Torre
University of New Hampshire

One difficulty in designing parallel algorithms is the large number of different models and machines. Algorithms (programs) are redesigned for each new model (machine). We are developing a general parallel computer model that will allow algorithms to be designed once, and have them apply to many specific machines. We will show how our techniques can be applied to solving some irregular problems.