Parallelizing a New Class of Large Applications over High-speed Networks

H.T. Kung, Peter Steenkiste, Marco Gubitoso, Manpreet Khaira

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Several large applications have been parallelized on Nectar, a network-based multicomputer recently developed by Carnegie Mellon. These applications were previously either too large or too complex to be easily implemented on distributed memory parallel systems. Parallelizing these applications was made possible by the cooperative use of many existing general-purpose computers over high-speed networks, and by an implementation methodology based on a clean separation between application-specific and system-specific code. We illustrate these points using our experience with parallelizing three real-world applications. The success in these applications clearly points out a new direction in parallel processing.

1. Introduction

Parallelizing large applications is a key concern for researchers in parallel processing. These applications typically involve large bodies of code, have substantial computation and memory requirements, and are of practical importance. If parallel architectures are to become widely used, porting such applications to parallel systems ought to be a routine activity.

There are numerous individual efforts in porting large applications onto various parallel systems. In most cases, the application is either developed specifically for the parallel machine, or it is almost completely rewritten to exploit the features of a specific parallel architecture. In both cases, the efforts require that the persons responsible for the porting are intimately familiar with the applications. Since large applications tend to involve substantial application-specific knowledge, this implies that often only application scientists themselves can do the porting. This definitely is not the best way of using their time, and as a result, many large applications have never been implemented on parallel machines.

What we need are parallel architectures and programming tools to provide direct and general support for parallelizing large applications. These tools should be at a higher level than send and receive primitives, or synchronization or shared data primitives. The objective is to be able to port large applications onto parallel machines without having to re-create the application code that captures the application knowledge.

Various research efforts have attempted to provide help in this area. Parallelizing compilers have some success in avoiding the rewrite of application code, but they deal mainly with inner-most loops rather than entire applications. Some projects have started to address parallel processing for large applications, but the solutions are either very ad hoc [8], or the application area is restricted. The work on LINDA [6, 11], for example, has provided tools capable of using existing application code, but their usage is restricted to applications satisfying a special computation model. Some programming environments such as //ELLPACK [13] support high-level descriptions for specifying applications in certain specialized areas, and allow new applications to use software and algorithms that were developed earlier for similar applications. Other work has concentrated on specific application areas such as vision [12, 16] or computations using triangular meshes [17].

This research was supported in part by the Defense Advanced Research Projects Agency (DOD) monitored by DARPA/CMO under Contract MDA972-90-C-0035, and in part by the Office of Naval Research under Contract N00014-90-J-1939.

The Nectar system [1] developed by Carnegie Mellon is intended to provide general support for parallelizing large applications. The system is a multicomputer built around a high-speed network. The use of existing general-purpose computers as its nodes and the high-bandwidth and low-latency network makes the system inherently suited for large applications. The system has allowed us to parallelize applications that were previously either too complex or too communication-intensive to be suited for parallel processing.

This paper describes the Nectar implementation of three applications: (1) COSMOS [4], a switch-level circuit simulator developed by Randy Bryant and his associates at Carnegie Mellon; (2) NOODLES [7], a solid modeling package developed by Professor Fritz Prinz' group in the NSF Engineering Design Research Center at Carnegie Mellon; and (3) a simulation of air pollution in the Los Angeles area, with Professor Greg McRae of the Chemical Engineering department. These three applications differ both in the application domain and in the programming model. Being able to port all these applications onto Nectar is an indication of the versatility of our approach. In fact, using similar methodologies several other large applications, not reported in this paper, have been successfully ported to Nectar by researchers at Carnegie Mellon. These include a parallel solid modeller from University of Leeds (called Mistral-3), distributed algorithms of finding exact solutions of travelling salesman problems, and a chemical flowsheeting simulation.

The strategy used to parallelize these applications is to maintain the serial code as much as possible, and to isolate communication and synchronization in a few routines. This approach simplifies the porting effort. We are in progress of developing libraries that provide the communication and synchronization support for several programming models. For applications that can use such a package, the porting effort will be limited to partitioning the sequential code and data at a high level, which a user can do quite easily. This methodology is illustrated using the three application examples.

In Section 2 we give an overview of the Nectar system. Our methodology for parallelizing these applications for Nectar-like systems is described in Section 3. Sections 4 to 6 describe the three applications and show how our methodology is used in porting them onto Nectar. Summary and concluding remarks are given in Section 7.

2. Nectar System Overview

The Nectar system developed by Carnegie Mellon is a multicomputer formed by linking together a number of *existing* machines by a *high-speed* network. Hosts are attached using powerful network coprocessors (CABs) that accelerate communication protocols. Therefore for Nectar a node is a CAB-host pair. The Nectar network (Nectar-Net) consists of 100 megabits per second fiber-optic links and 16×16 crossbar switches (HUBs). The network supports circuit switching, packet switching, multi-hop routing, and multicast communication. Figure 2-1 gives an overview of the Nectar system.

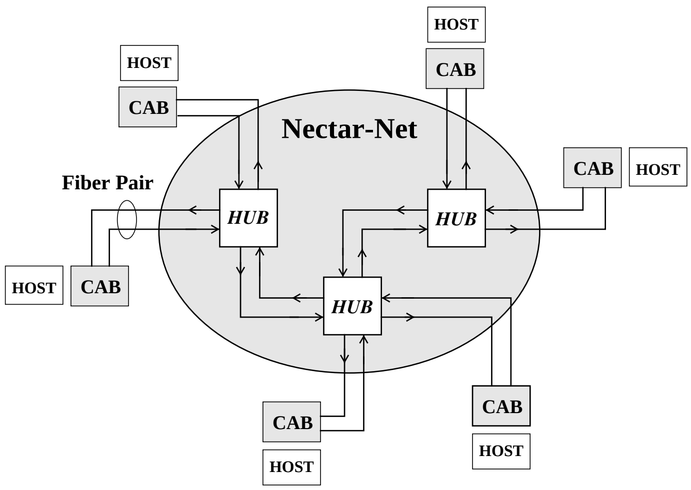

Figure 2-1: Nectar system at Carnegie Mellon

Currently the Nectar system has 26 hosts, mostly Sun 4 workstations. The network contains a 26 km Nectar connection to a Westinghouse facility, which hosts the CRAY Y-MP of the Pittsburgh Supercomputing Center. The systems software includes a CAB run-time system that supports multiprogramming using light-weight threads [10] and manages message buffers using a mailbox mechanism.

Besides high-bandwidth communication (100 megabits/second per link), Nectar features low-latency communication. The existing Nectar has the following measured performance: the latency to establish a connection through a single HUB is under one microsecond; excluding the transmission delays of the optical fibers, the latency for a message sent reliably between processes on two CABs is under 100 microseconds; and the corresponding latency for processes residing in host nodes is under 200 microseconds. This high-bandwidth and low-latency network can sustain the communication bandwidth required by nodes operating at high speeds and allow concurrent processing of small-grain computations at multiple nodes [15].

In the area of protocol software, the CAB currently supports several transport protocols with different reliability/overhead trade-offs [9]. In particular, we have implemented a number of Nectar-specific protocols, plus a CAB-resident version of the Internet standard protocol TCP/IP.

The Nectar system is used on a daily basis by both applications programmers and computer science researchers. To further develop the Nectar technology, Carnegie Mellon is working with Network Systems Corporation to develop a *gigabit Nectar* system capable of sustaining 1 gigabit per second or higher speed end-to-end communication.

3. A Methodology for Parallelizing Large Applications

Implementing large applications on Nectar-like architectures involves considerations along many dimensions. These include partitioning the application to exploit parallelism, mapping and distributing global data, ensuring data consistency, enforcing required synchronization, performing load balancing, and possibly providing fault tolerance for applications with long execution times. Here we describe an implementation methodology that can simplify this task. This methodology has been successfully used in the Nectar implementation of the three applications described in the subsequent sections.

The basic principle is the separation of *application code* and *system code*. The application code has all the application-specific knowledge, but does not include any code related to parallel processing. The system code provides communication and synchronization operations required for the parallelization on a specific architecture.

When parallelizing an application starting with a serial implementation, the first step is to partition the application in units of work, called *tasks*; this collection of tasks makes up the application code. The code for each task is a sequential program to be executed on a single node. If the code for a task already exists, it can be reused in the parallel implementation. The next step is to identify the synchronization and communication requirements between the tasks. Their implementation makes up the system code.

The system code is a separate module with a clean interface to the application code. This makes it possible to reuse the system code when parallelizing other applications. For the three applications described in this paper, the system code was developed specifically for the application, but we are in the process of packaging the system code in the form of a library that can be linked in by other applications. The system code implements a specific communication style for the parallel computation in hand. For a given application, the programmer selects the appropriate module for the system code, based on the application needs.

This strategy for parallelizing applications based on strictly separating application and system code has several advantages. First, it naturally supports reuse of existing application code. For large applications, for which extensive application-specific knowledge is embedded in existing code, this is the only practical approach. The cost of rewriting and maintaining different versions of the application for different systems can be prohibitive, no matter what the payoff in performance due to parallel processing might be. Even if the application is implemented originally on a parallel system, for maintenance reasons one would want to keep most of the code system independent. Second, this approach is not limited to a single communication model, as is shown below by the example applications. Different modules for system code can be provided to support a range of synchronization and communication styles. As mentioned earlier, the Nectar project is in the processing of building up a library of these modules.

Finally, we note that by implementing the same system modules on different architectures, porting applications across these architectures will be easier. The implementations of the system modules on the different architectures can be optimized for the architecture. For example, a system module that provides support for load balancing can use different task granularities in different implementations to deal with differences in the computation speed to communication latency ratio. We expect that this approach to portability will be more effective than trying to provide compatibility at a low level such as send and receive primitives.

An alternative to our task-based approach, the automatic extraction of parallelism as is done successfully for FORTRAN DO-loops, is not practical for entire applications. It is unlikely that compilers can characterize the way the program updates complex data structures or the complex control flow of a substantial application. The task-based approach is an intermediate solution between automatic parallelization and rewriting the entire application. The programmer specifies the parallelism explicitly based on his or her understanding of the application. Since most of the application, if not all, is executed in sequential stretches of code, most of the application details can be ignored in the parallelization process.

The EXPRESS environment [14] developed at Caltech and Parasoft Corporation is based on a similar methodology. The main difference is that we try to provide communication support at a higher level than that provided by the EXPRESS libraries which are at the level of SEND and RECEIVE primitives and global synchronization primitives.

4. COSMOS: A Logic Simulation Application

Logic simulation is both a time-consuming and a memory-intensive process. For this reason, many large circuits cannot be simulated in their entirety on single computers. A multicomputer such as Nectar can solve the problem by linking together many existing systems to increase both the computational power and the memory.

We describe a parallel implementation of a logic simulator on Nectar. The simulator is COSMOS [4], a high-performance logic simulator developed at Carnegie Mellon over the past several years. COSMOS is the successor of MOSSIM [5], a widely used simulator in industry at present. The key feature of COSMOS is that it compiles the circuit into executable code, instead of interpreting a representation of the circuit at run time. COSMOS first partitions the circuit into a number of channel-connected subcircuits and derives a boolean representation of the behavior of each subcircuit. It translates this representation into C language evaluation procedures and declarations of data structures describing the connections between the subcircuits. This circuit code is then compiled and linked with a COSMOS kernel and user interface to generate the simulator program. COSMOS runs about an order of magnitude faster than MOSSIM; the cost is a relatively slow and memory-intensive compilation phase.

Circuits are simulated a clock *phase* at a time, and at the beginning of each phase, external signals such as clocks can change. The simulation of a phase consists of a number of *simulation steps*. During each step, subcircuits whose input signals have changed since the previous step are evaluated; for the first step of each phase, external signals determine which subcircuits are evaluated. The simulation of a phase is finished when all signals are stable, so the number of steps in a phase depends both on the circuit and on the input signals.

The goal of the COSMOS implementation on Nectar, called Nectar-COSMOS, is to simulate large single-chip or multi-chip circuits which can have as many as one million transistors. An initial version of Nectar-COSMOS in May 1990 could already handle circuits with hundreds of thousands transistors. More recently, we have used Nectar-COSMOS to simulate the latest design of the 650,000-transistor iWarp chip [2, 3], jointly developed by Carnegie Mellon and Intel. For the previous fabrication runs, the full-chip simulation of iWarp was infeasible on any single computers available to Intel. Using Nectar-COSMOS, a full-chip simulation of the iWarp chip is possible.

4.1. Mapping COSMOS onto Nectar

In the Nectar-COSMOS implementation, the subcircuits are statically distributed over the Nectar nodes; the subcircuits placed on the same Nectar node form a *unit*. The connectivity information for the subcircuits is used to determine what signals have to be communicated between the nodes on every simulation step. Each node runs a copy of the simulator, i.e., the COSMOS code corresponding to the subcircuits assigned to the node. After each simulation step, the node sends the output signals to other nodes that need them. A node can start on the next simulation step once it has received the necessary input signals from the other nodes.

To detect the end of a phase, distributed termination detection is required, since the circuit is partitioned, and no Nectar node has access to all signals. At this point, a centralized algorithm is used: after a small number of steps, all nodes report to the master, who determines whether the circuit is stable. Note that once the circuit is stable, simulation steps are very fast since no circuits are evaluated, so doing a few extra steps seems to be acceptable. Nevertheless, better algorithms are being studied.

Because all the nodes work on one phase at a time, the simulation time for a phase is determined by the slowest node. Therefore, when placing subcircuits over Nectar nodes, it is important that all resulting units have nearly the same simulation time. Using sequential measurement for subcircuits, and experience, it is usually possible to achieve a reasonable balance.

In each unit, we distinguish two types of subcircuits – *boundary modules* and *interior modules*. Interior modules are only connected to subcircuits within the same unit, while boundary modules are connected to subcircuits in other units. Each Nectar node first simulates the boundary modules, since the simulation of these subcircuits will produce results needed by other nodes for the next simulation step. After this is done, the host simulates the interior modules, while the CAB sends out the results produced previously by the simulation of the boundary modules. Thus Nectar-COSMOS can take advantage of the CAB to overlap computation with communication, thereby reducing the total execution time.

4.2. Results and further work

To evaluate the performance of Nectar-COSMOS, we simulated a 30×30 maze routing chip implemented using dynamic CMOS, and consisting of 170,000 transistor. Table 4-1 shows the results using dedicated Sun 4/330 hosts. A cycle consists of 4 phases, each of which requires about 8 steps to reach stability. We observe a close to linear speedup up to 3 nodes. (The work of using more than 3 nodes is in progress.) The chip could not be be simulated on a single workstation because we were not able to generate the simulator due to memory limitations. Based on the simulation of a single column of the maze router, we estimate that if we could generate the simulator on the workstation, a single-node simulator would take approximately 1.32 seconds per cycle.

Number of nodes	1	2	3
Time/cycle (seconds)	1.320 (estimate)	0.676	0.474
Speedup	1	1.95	2.78

Table 4-1: Nectar-COSMOS speedup for 30×30 maze router

Table 4-2 gives some insight in the structure of the maze router chip. The chip consists of a 30 by 30 array of identical cells plus a circuit for clock distribution. In Nectar-COSMOS, each node gets a block of columns, and one node also gets the extra burden of the clock distribution circuit. This mapping results in a good balance of modules across nodes. It also has the advantage that most of the subcircuits are interior modules as indicated in the table. Using this fact, the Nectar-COSMOS implementation is able to overlap most of its the communication overhead with simulation computation. Consequently, a node spends less than 10% of its total execution time solely on communication. This explains the good speedup observed. Note that this speedup is obtained in spite of the fact that there is little activity in the maze router chip. As shown in Table 4-2, the number of modules evaluated during a cycle is only about 50% higher than the number of modules in the circuit.

Number of nodes	1	2	3
Boundary modules	-	45	40
Interior modules	-	19276	12884
Module evaluations per cycle	60300	30200	20140

Table 4-2: 30×30 maze router analysis

The results for the maze router chip show that it is possible to speed up circuit simulation using a Nectar-like system. However, Nectar-COSMOS can be communication intensive when a large of nodes are used, thus limiting the speedup that can be achieved. Further evaluation of COSMOS on Nectar is needed, using more realistic circuits such as iWarp. These circuits have more potential parallelism, since they are larger and probably have more activities, but they have the drawback that they are not regular, thus making it harder to distribute them evenly across the network nodes.

Future work will concentrate on finding efficient mappings of subcircuits to nodes. Tradeoffs involve balancing the load, maximizing the number of interior nodes, and minimizing communication. A complication is due to the fact that COSMOS does not simulate subcircuits whose inputs have not changed since the previous time step. This optimization makes the execution time, data dependent, and for some circuits this might influence how the subcircuits should be mapped onto the processors.

Even though COSMOS is a significant real application (about 50,000 lines of code in the COSMOS compiler chain and kernel), the porting of COSMOS to Nectar was relatively easy. The reasons are that the sequential COSMOS had already partitioned the main data structure (the circuit), and that the sequential implementation already existed on the same workstations that form the Nectar nodes. As a result, a mapping where each node runs a copy of the original program (a simulator), and operates on part of the input data (circuit) is natural and required very few changes to the original program. The only change is that the simulator now gets input signals, and returns output signals in a slightly different format.

The main effort in Nectar-COSMOS was in implementing the system code that is specific to the parallel implementation. Its function is to communicate the signals between the nodes; as much of the work is done on the CAB to overlap the communication overhead with the processing on the CAB. It also supports communication between the master and the simulators for initialization and termination detection. The system code is implemented as a separate module.

As mentioned earlier, an important feature in Nectar-COSMOS is that we are able to hide the communication latency by simulating the subcircuits in the right order (i.e., simulate external circuit modules before internal ones). We hope that some of these techniques will extend to other parallel simulators.

5. NOODLES: A Geometric Modeling Application

NOODLES is a geometric modeling system [7] developed by the NSF-sponsored Engineering Design Research Center (EDRC) at Carnegie Mellon. NOODLES models objects of different dimensions as a collection of basic components such as vertices, edges, and faces. As a result, NOODLES can represent both real objects, and non-manifold objects, that exist only in abstract models and cannot be actually built, such as a single edge of zero thickness. Non-manifold objects simplify some higher-level operations on models, such as testing whether two objects touch in a point (their intersection is a vertex). Applications of NOODLES include integrated systems for computer design, knowledge-driven manufacturability analysis, and rapid tool manufacturing. We describe a Nectar implementation of NOODLES, called *Nectar-NOODLES*, developed jointly by the EDRC and School of Computer Science at Carnegie Mellon.

The basic operation in NOODLES is the *merge* operation. Using this operation, complicated objects can be built by intersecting a pair of simpler ones. The merge operation does a pairwise geometric test on components in both input objects, and it breaks up components if they intersect. These tests are separated in *stages*, depending on the type of the comparison test. For example, in the first stage all vertices of one model are compared with those of the other. In the subsequent stages, vertices are compared with edges, and so on.

Geometric tests may yield updates to the database of the models. These changes will influence the tests to be done in subsequent stages, so the computation in each stage depends on the results of earlier tests. For example, if two edges intersect, they will be replaced with four non-intersecting edges, which will be used in later tests. Thus, the number of tests are data-dependent, and the cost of the tests depends on the models.

5.1. The parallelization of NOODLES

Updates in each stage are intrinsically sequential, but the geometrical tests that produce these updates can be performed in parallel. When merging two models, each with n components, the total number of operations needed for updates is $O(n)$ or $O(n \log n)$, while that for geometric tests is $O(n^2)$. Thus, for large models with large n, the speedup resulting from parallelizing geometric tests can be substantial. The goal of Nectar-NOODLES is to allow interactive use of NOODLES, even for large designs.

Because the execution time of the various tests and updates in NOODLES is very much data dependent, the distribution of work across Nectar nodes is done dynamically at runtime. Nectar-NOODLES uses a central load balancing strategy: a master node keeps a central task queue, and slave nodes execute tasks that they receive from the master. Each task consists of a series of basic geometrical tests, where each basic test consists of comparing one component of one model with a class of components in the other model, for example, an edge of one model with all vertices of the other model.

The master node has two functions. First, it manages the transition between the stages, including the updates to the database. This function is NOODLES specific and runs on the workstation host. Second, the master node manages the dynamic load balancing during each stage. This function involving internode communication is not specific to NOODLES, and is implemented on the CAB of the master node. The advantage of placing the task queue manager on the CAB is that it can respond to requests faster: it can handle about 10000 requests per second. The workstation of the master node operates as a slave during each stage.

Nectar-NOODLES cannot rely on a straightforward partitioning of the input data space, as was done in Nectar-COSMOS. In order to accommodate non-manifold models, NOODLES uses an intricate data structure with a large number of pointers. Distributing this data structure over the Nectar nodes would require a total rewrite of NOODLES. Nectar-NOODLES avoids this by giving each Nectar node a copy of the geometric models being merged.

The copies of the models on the nodes are kept consistent by updating all the models at the same time and in the same order. To make this possible, updates to the models are not done ‘‘on the fly’’ as in the sequential NOODLES, but all updates are delayed until the end of the stage. When a geometric test indicates that an update is needed, the slave sends an update request to the master. The master collects the updates, and at the end of each stage, it sends the list of updates for that stage to all the slaves, which use the information to update their copies of the geometric models. To allow nodes to send updates to each other, global names were added to each entry in the NOODLES database. The NOODLES code was not changed: it still operates on its original data structure using local pointers (which can be different on all the nodes). The translation between local pointers and global names is done using a table lookup at the interface between NOODLES and the system code.

It is interesting to note that an implementation of NOODLES on a shared-memory parallel processor would probably have the same structure as Nectar-NOODLES. Because of the complexity of the data structures, it would not be possible to update the database while other nodes are doing geometric tests. The shared-memory implementation would have to batch updates between the testing stages in exactly the same way as Nectar-NOODLES does. The main benefit of a shared memory implementation would be that the updates could be done by a single node, which is slightly simpler than broadcasting an update request to all the slaves.

The structure of Nectar-NOODLES lends itself well to a robust implementation. If a slave node goes down during a session, no information is lost. The master can simply reissue the task that slave was working on to one of the remaining slaves, and the session can continue without interruption. This should make it possible to use a large number of nodes reliably, although the master would of course remain a single point of failure. In the current implementation, we have not yet implemented this robust scheme, although the master does ignore nodes that do not respond during initialization.

5.2. Results

Table 5-1 shows the speedup for Nectar-NOODLES merging two models, each consisting of two spheres. Each of the models has about 3500 components. The hosts are dedicated Sun 4/330 workstations. The speedup is relative to a single-node Nectar-NOODLES. The different columns show the results for various task sizes, starting with 1 test per task to 30 tests per task. The size of a test ranges from 3 milliseconds in the early stages to as high as 50-150 milliseconds in the later stages (6 milliseconds average). We observe a similar speedups for all task sizes. Even with one test per packet, we do not observe any degradation of performance. The Nectar net and the load balancer are fast enough to support tasks as small as a few milliseconds. As the task size increases, the speedup drops slightly, and this effect becomes stronger as the number of nodes increases. The reason is that the load balancing becomes less effective in the later stages, which have a smaller number of larger tasks.

The single node version of Nectar-Noodles is almost 30% slower than the sequential Noodles, because some of the geometric tests may be performed more than once in the parallel implemenation. This duplication happens because updates to the database are delayed until the end of each stage and as a result possible redundant operations are not deleted in time. This

Number of nodes	1	5	10	15	30
2	1.94	1.93	1.92	1.90	1.89
3	2.83	2.81	2.81	2.79	2.75
4	3.61	3.62	3.60	3.54	3.47
5	4.40	4.38	4.33	4.29	4.17
6	5.15	5.10	5.04	4.94	4.75
7	5.84	5.78	5.71	5.54	5.35
8	6.40	6.38	6.27	6.07	5.76

Table 5-1: Nectar-NOODLES speedup for different task sizes (expressed in tests per task)

Number of nodes	Time (seconds)	Speedup
1	0.427	1.00
2	0.208	2.05
3	0.142	3.01
4	0.120	3.55
5	0.105	4.07
6	0.100	4.25
7	0.098	4.34

Table 5-2: Speedup for ray tracing application using Noodles load balancing package

illustrates a difficulty in parallelizing code that uses complex data structures. A parallel implementation of NOODLES on a shared memory machines would have the same problem.

As in the case of COSMOS, NOODLES was mapped onto Nectar by running a version of the sequential program on every node. Again, very few changes had to be made to the existing code, thus simplifying the porting of this relatively large application, which has about 12,000 lines of code. Almost all the code that is specific to the parallel implementation is in a separate module. Because of the complexity of the data structures, the data could not be partitioned, but had to be replicated, thus loosing one of the benefits of using a multicomputer (more memory). The low communication latency on Nectar made dynamic load balancing very effective, even for relatively small task sizes.

5.3. Building a load balancing package

We are currently in the process of implementing the load balancing code that was developed for Nectar-NOODLES as a separate package. As a demonstration of the usefulness of using such an application-independent package, a second application, ray tracing, has been ported very quickly to Nectar using this package. This application allows us to evaluate the load balancing packet with smaller packet sizes.

Table 5-2 shows the results. The application consists of 1024 tasks, each taking about 400 microseconds; the sequential part of the code takes little time (about a millisecond). We notice that the speedup curve flattens at about 5 nodes with a speedup of 4. This shows the limitations of a central load balancing scheme: the master node can handle a new request about every 100 microseconds, so the minimal execution time is about 100 milliseconds for an application with 1000 tasks. Using more nodes will require a coarser partitioning of the problem and a larger problem.

6. Simulation of Air Pollution in Los Angeles

Flow field problems, such as weather forecasting and tracking of spills, are computationally intensive and can benefit from parallel processing. As a first step, we have implemented a parallel program on Nectar which tracks pollutant particles in the atmosphere of the Los Angeles area. The input to the program are the wind velocities recorded at 67 weather stations around the Los Angeles (LA) area once every hour. The program calculates the traces of pollutant particles that are released in some initial locations.

Computing the particle traces given the wind conditions is a two phase process. The first phase consists of computing the wind velocity at each point of a 80×30 grid on the geographic area concerned, for every hour, given the measurements from the weather stations and precomputed weights. This problem involves interpolating from the measurements, as well as solving the conservation of mass equations across the grid. In the second phase, each particle is tracked as it moves about the grid; this requires an interpolation in both space and time. The time step used in this phase is 30 seconds.

6.1. Parallel implementation over Nectar

When partitioning this program over Nectar, we tried to maintain the structure and code of the original sequential program as much as possible. In the first phase, a task consists of calculating the wind velocities at each point in the grid for a given hour. The second phase is parallelized by partitioning the particles among the processor: each processor tracks the motion of a set of particles for the duration of the simulation. The two phases are pipelined: while some processors are tracing particles at time T, other processors are calculating the wind velocities for the following few hours. In the initial implementation, load balancing was done statically: the hours and particles were divided among the processors before computation begins.

6.2. Results and evaluation

Table 6-1 shows the results for parallelizing the particle tracking on Nectar using 1, 2, 3, 4, 6 and 8 nodes. Again, the hosts are dedicated Sun 4/330 workstations.

Number of nodes	Time (seconds)	Speedup
1	125	1.0
2	65	1.9
3	44	2.9
4	34	3.7
6	23	5.4
8	19	6.6

Table 6-1: Speedup for LA pollution simulation

The speedup shown in the table are encouraging, but since the distribution of work is done statically, performance degrades quickly if the load on the (shared) nodes changes during the execution. To avoid this degradation, we are currently implementing dynamic load balancing for both phases. For the first phase, processors receive the next hour to be simulated from a master. This gives good performance except that slow nodes might make the latency between the two phases too large; for this reason, the master should replace nodes that are too slow. For the second phase, a load balancing process monitors the progress of each of the phase two processors, and moves particles from slow processors to fast processors, if the difference in simulated time on the slaves becomes too large. If the network environment does not change, each slave will trace its particles with minimal disruption by the load balancer.

The communication bandwidth of this application increases linearly as more processors are added to the system, since every processor in the second phase must have all the information computed by the processors in the first phase. Because we use the Nectar (hardware) multicast facility between phases one and two, the communication overhead per simulated hour remains constant for each node. Eventually, the constant communication overhead will limit the number of nodes that can be used effectively for a fixed problem size.

The total number of bytes sent can be reduced by using a new mapping in which the grid is partitioned across the processors. For first phase, each processor calculates the wind velocities for its part of the grid, for all hours, i.e., we partition in space instead of in time. For the second phase, each processor traces the particles in its area at any given time, i.e., we divide the area instead of the particles. This mapping significantly reduces the communication requirements, but it has several disadvantages. First, it require more fine-grained interactions between the processors working on the same phase: for phase one, processors have to interact when solving the conservation of mass equations, while for the second phase, communication is needed when particles cross the partitioning boundaries.

Second, this mapping is more complicated to implement, because the structure of the program is changed more dramatically: we are parallelizing over an innerloop, while the first mapping parallelized over the outerloop. Third, load balancing becomes much more difficult. Not only is the workload in the second phase no longer static, but moving work between processors as part of a dynamic load balancing strategy is much more complex.

The LA simulation program is an example of a medium size static application (2500 lines of FORTRAN). Our implementation shows that a very simple mapping that preserves the program structure is the most appropriate for a network environment: there is a good match between the resulting coarse-grained parallelism and an architecture with a small number of powerful nodes. Finer-grain parallelism should be exploited in more tightly coupled multiprocessors, i.e., inside a node of the multicomputer. We plan to work on this type of hierarchical decomposition using more accurate simulation programs and using iWarp arrays connected by Nectar as the computing engines.

7. Summary and Concluding Remarks

As demonstrated by the Nectar implementation of the three applications described in the paper, it is now possible to parallelize some applications which were too complex or too large for previous parallel processing approaches. The two factors that make this possible are the emerging class of network-based multicomputers such as Nectar and a systematic approach to parallelizing large applications.

Because Nectar uses existing general-purpose computers as nodes, parallelized applications can make immediate use of system software and application code that are already exist for these computers. Using existing systems as nodes also has the advantage that users can work in a familiar environment, for example, a UNIX workstation. As a result, in spite of the high complexity of the applications described in the paper, the implementation of them on Nectar has taken relatively little effort.

An important requirement for network-based multicomputers is good network performance. The bandwidth and latency characteristics of Nectar, for example, are similar to those of the current generation custom-made multicomputers. Because of these features, there is a large class of applications for which internode communication is no longer bottleneck, and parallelization over a network becomes practical, as is demonstrated in the NOODLES and LA pollution simulation applications. The programmer does not need to worry too much about communication overheads in doing load balancing. This is a reason why it has been relatively easy to achieve good speedups for the parallelized applications.

Our approach to parallelizing large applications emphasizes the use of existing application code and the development of general supports for large-grain parallelization. Most of the application code should be architecture independent, and the system code that implements synchronization and communication may be reused. Our goal is to provide applications with programming support at a higher level than sends, receives, and low-level synchronization primitives. This effort complements other existing efforts of providing general supports for parallelizing kernels of computation for more tightly coupled multiprocessors. The combined capability will significantly increase the applicability of parallel processing, we believe.

However, much work needs to be done in support of this new opportunity, especially in the area of tool building. We plan to undertake some of this work in the near future. Results reported in this paper should be viewed as a progress report of our efforts in the important area of providing general support for parallelizing large applications.

Acknowledgements

A large number of people at Carnegie Mellon contributed to the implementation of the Nectar prototype system. Professor Fritz Prinz of the Mechanical Engineering Department, Professor Greg McRae of the Chemical Engineering Department, and Professor Randy Bryant and Dr. Carl Seger of the School of Computer Science directed the three applications described in this paper. The University of Sao Paulo sponsored Marco Gubitoso’s visit to Carnegie Mellon University in 1989-90. This work would not have been possible without help from all these parties.

References

1. Emmanuel A. Arnould, Francois J. Bitz, Eric C. Cooper, H. T. Kung, Robert D. Sansom and Peter A. Steenkiste. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, ACM/IEEE, Boston, April, 1989, pp. 205-216.

2. Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross, H. T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper, Linda Rankin, P. S. Tseng, Jim Sutton, John Urbanski, and Jon Webb. iWarp: An Integrated Solution to High-Speed Parallel Computing. Proceedings of Supercomputing ’88, IEEE Computer Society and ACM SIGARCH, Orlando, Florida, November, 1988, pp. 330-339.

3. Shekhar Borkar, Robert Cohn, George Cox, Thomas Gross, H.T. Kung, Monica Lam, Margie Levine, Brian Moore, Wire Moore, Craig Peterson, Jim Susman, Jim Sutton, John Urbanski, and Jon Webb. Supporting Systolic and Memory Communication in iWarp. Proceedings of the 17th Annual International Symposium on Computer Architecture, ACM/IEEE, Seattle, May, 1990, pp. 70-81. Also published as CMU Technical Report CMU-CS-90-197.

4. Randy E. Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, and Thomas Sheffler. COSMOS: A Compiled Simulator for MOS Circuits. Proceedings of the Design Automation Conference, ACM/IEEE, June, 1987, pp. 9-16.

5. Randy E. Bryant. "Boolean Analysis of MOS Circuits". *IEEE Transactions on Computer Aided Design CAD-6*, 4 (July 1987), 634-649.

6. Nicholas Carriero and David Gelernter. Applications Experience with Linda. ACM Symposium on Parallel Programming: Experience with Applications, Languages, and Systems, ACM, July, 1988, pp. 173-187.

7. Young Choi. *Vertex-based Boundary Representation of Non-Manifold Geometric Models*. Ph.D. Th., Carnegie Mellon University, 1989.

8. E. Clementi, J. H. Detrich, S. Chin, G. Corongiu, D. Folsom, D. Logan, R. Caltabiano, A. Carnevali, J. Helin, M. Russo, A. Gnudi and P. Palmidese. Large-Scale Computations on a Scalar, Vector and Parallel ‘‘Supercomputer’’. In *Structure and Dynamics of Nucleic Acids, Proteins, and Mambrans*, Plenum Press, 1986, pp. 403-449. Edited by E. Clementi and S. Chin.

9. Eric C. Cooper, Peter A. Steenkiste, Robert D. Sansom, and Brian D. Zill. Protocol Implementation on the Nectar Communication Processor. Proceedings of the SIGCOMM ’90 Symposium on Communications Architectures and Protocols, ACM, Philadelphia, September, 1990, pp. 135-143. Also published as CMU Technical Report CMU-CS-90-153.

10. Eric C. Cooper and Richard P. Draves. C Threads. Tech. Rept. CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June, 1988.

11. David Gelernter. "Generative Communication in Linda". *ACM Transactions on Programming Languages and Systems 7*, 1 (January 1985), 80-112.

12. Leonard G. C. Hamey, Jon A. Webb, and I-Chen Wu. "An Architecture Independent Programming Language for Low-Level Vision". *Computer Vision, Graphics, and Image Processing 48* (November 1989), 246-264.

13. E. N. Houstis, J.R. Rice, N.P. Chrisochoides, H.C. Karathanasis, P.N. Papachiou, M.K. Samartzis, E. A. Vavalis, Ko Yang and S. Weerawarana. //Ellpack: A Numerical Simulation Programming Environment for Parallel Mimd Machines. Proceedings of the 1990 International Conference on Supercomputing, June, 1990, pp. 96-107. Also published as ACM SIGARCH Computer Architecture News, Volume 18, Number 3, September 1990.

14. K. Ikudome, G. C. Fox, A. Kolawa, and J. W. Flower. An Automatic and Symbolic Parallelization System for Distributed Memory Parallel Computers. Proceedings of the Fifth Distributed Memory Computing Conference, IEEE, April, 1990, pp. 1105-1114.

15. Kung, H. T. Heterogeneous Multicomputers. Carnegie Mellon Computer Science: A 25-Year Commemorative, Reading, Massachusetts, 1990, pp. .

16. Jon A. Webb. Architecture-Independent Global Image Processing. 10th International Conference on Pattern Recognition, IEEE, Atlantic City, NJ, June, 1990, pp. 623-628.

17. Roy D. Williams. DIME: A Programming Environment for Unstructured Triangular Messhes on a Distributed-Memory Parallel Processor. Proceedings of the Third Conference on Hypercubes, Concurrent Computers, and Applications, Pasadena, California, January, 1988, pp. 1770-1787.