Executing Object-Oriented Parallel Programs on High Performance Simulators *

Suresh Srinivas, Dennis Gannon
Department of Computer Science, Lindley Hall 215,
Indiana University, Bloomington, IN 47405.
{ssriniva,gannon}@cs.indiana.edu

In the last few years several massively parallel computers based on the same microprocessors as workstations – such as Thinking Machines CM-5, Intel's Paragon Intel's Paragon [1], Cray's T3D [2] – have emerged.

This paper demonstrates that high-performance simulators – running on cost-effective workstations or servers – can be used for developing systems and application software for parallel machines such as these. Massively parallel machines are very good at running debugged, well tuned parallel programs fast. But they are not very good for software development. The development tools that come with these machines (compilers, debuggers, performance tools etc.,) are still in a state of infancy in comparison with those on workstations. Also massively parallel machines have restricted interactive use and are more expensive to do development on.

Using simulators on workstations or servers to develop systems and application software for parallel machines has several advantages. It allows reuse of familiar and robust workstation development tools – such as sequential debuggers for example `gdb` and programming environments for example CaseVision/Workshop – during the development process. Also simulators allow fine grain access to performance information that real machines rarely provide and this is very useful for performance tuning parallel applications.

Workstations and servers are good for different development tasks. High performance simulators running on workstations are fairly adequate for:

- Prototyping runtime systems and language extensions for high-level parallel programming languages.
- Debugging the parallel application for software bugs.

The reason being small problem sizes and machine sizes are almost always used for the above tasks. But determining performance bottlenecks or scaling problems in parallel applications usually requires them to be run on bigger problems or machine sizes. We find workstations are not suitable for this task since the physical memory size of the workstation turns out to be a limiting factor.

Medium size servers such as SGI's PowerChallenge [3] or Sun's SPARC 2000 [4] or HP's HP 9000 [5] – that are becoming commonplace – could be used for:

- Observing parallel performance and debugging the application for performance bottlenecks.
- Analyzing and predicting the behavior of the application on larger problem and machine sizes.

*This research is supported in part by ARPA under contract AF 30602-92-C-0135, the National Science Foundation Office of Advanced Scientific Computing under grant ASC-9111616. and the National Science Foundation and ARPA Grand Challenge Award, "The Formation of Galaxies and Large-scale Structure."

These servers have huge physical memory (several gigabytes) and therefore allow bigger problem and machine sizes to be simulated. For example a 20 processor SGI Challenge server with 2048 megabytes of memory can easily simulate 512 processors with 8 megabytes of memory. Since these servers typically have several processors, parallelizing and thus speeding up the simulation is also of interest. Studying parallel application performance by simulation also alleviates the problem of perturbation in application behavior which users of instrumentation have to deal with. The reason being instrumentation introduced into the simulation can be controlled.

This paper describes how parallel machines and applications can be simulated. It mostly focuses on our experiences in using a direct execution simulator in the context of pC++ [6], an object parallel language. Our experience includes developing runtime systems, language extensions, data structure visualization tools, and debuggers for pC++. This paper also describes strategies for speeding up the simulation and focuses on one such strategy where parallelism is used to speed up the simulation on a 20 processor SGI Power Challenge Server. The main contribution of this paper is to show the applications of direct execution simulation in the context of a high level parallel programming language.

1 Simulating parallel machines and parallel applications

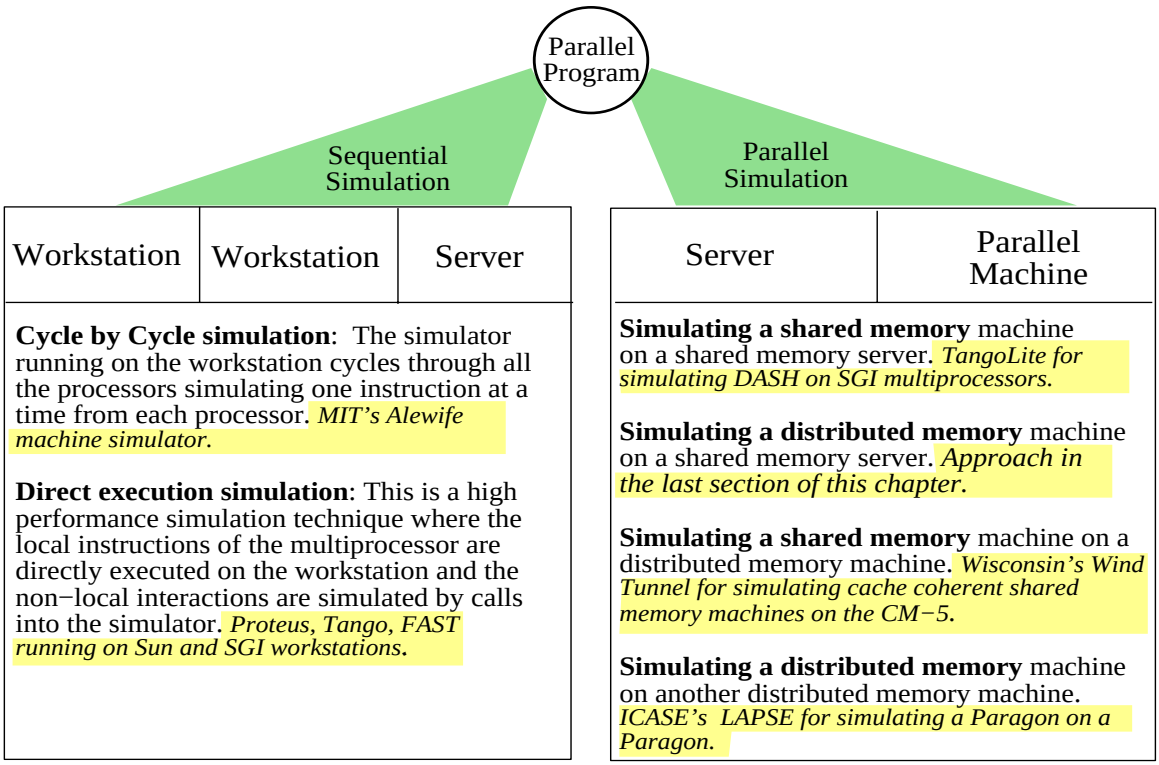

Figure 1: **SIMULATION EXECUTION OF A PARALLEL PROGRAM** can be performed on a workstation environment or an actual parallel machine. On the workstation environment there are two methods for performing the simulation. On the parallel machine there are four possibilities for simulation. The example systems in each of these categories are highlighted and are in italics.

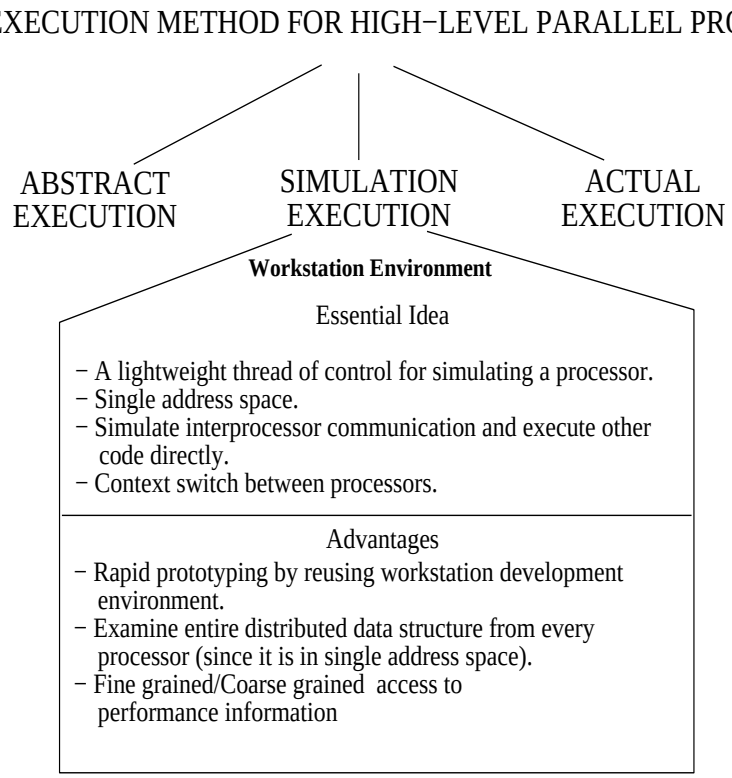

Figure 2: **AN OVERVIEW OF THE APPROACH** described in this paper. There are 3 basic execution models for parallel programs. Simulation execution is the middle ground approach. It is less accurate than actual execution but provides detailed information.

Simulation is an important tool for designing all aspects of parallel computer systems. It has a broad spectrum of applications ranging from virtual prototyping of new parallel computers as in Wisconsin's Wind Tunnel [7] to studying parallel applications as in LAPSE [8].

Simulating multiprocessors on a uniprocessor can be done in several ways as **Figure 1** shows. The simplest (and the slowest) way is to cycle through all the processors simulating one instruction at a time from each processor. This gives a great deal of accuracy but is very slow. The simulator ASIM for the Alewife [9] machine at MIT is one such example. Cycle by cycle simulators by virtue of their accuracy are used often by parallel computer architects in studying architectural design and tradeoffs.

Direct execution simulation [10] is another way of simulating multiprocessors. It is much faster compared to the cycle by cycle instruction simulators. In direct execution simulation the local instructions of the multiprocessor are executed directly on the workstation and nonlocal instructions are simulated by procedure calls. Nonlocal instructions include message passing instructions which cause access to other nodes of the multiprocessor being simulated. Since the local instructions are executed directly, the assembly code has to be augmented to keep track of the number of instruction cycles simulated in the local portions of the code.

Several direct execution simulators are available today: Proteus [11] from MIT, TangoLite [12] from Stanford, FAST [13] from Berkeley. The advantages and disadvantages of direct execution simulation has been extensively addressed in the paper [14] .

Performing the simulation on a parallel machine is an attractive proposition as well as **Figure 1** shows. Since

parallel machines come in two flavors (shared memory and distributed memory) there are 4 possibilities for simulation. The final section in this paper is about executing a parallel program on a simulated distributed memory machine; the simulation is performed on a shared memory server.

Trace-driven simulation is another technique for simulating multiprocessors. Here traces generated on one multiprocessor system are used in the simulation of a multiprocessor system with different characteristics. This has been predominantly used in evaluating memory-system performance and requires that the environment for trace generation be very close to the system being simulated.

Figure 2 highlights the approach described in this paper.

2 Direct execution simulation of pC++ programs

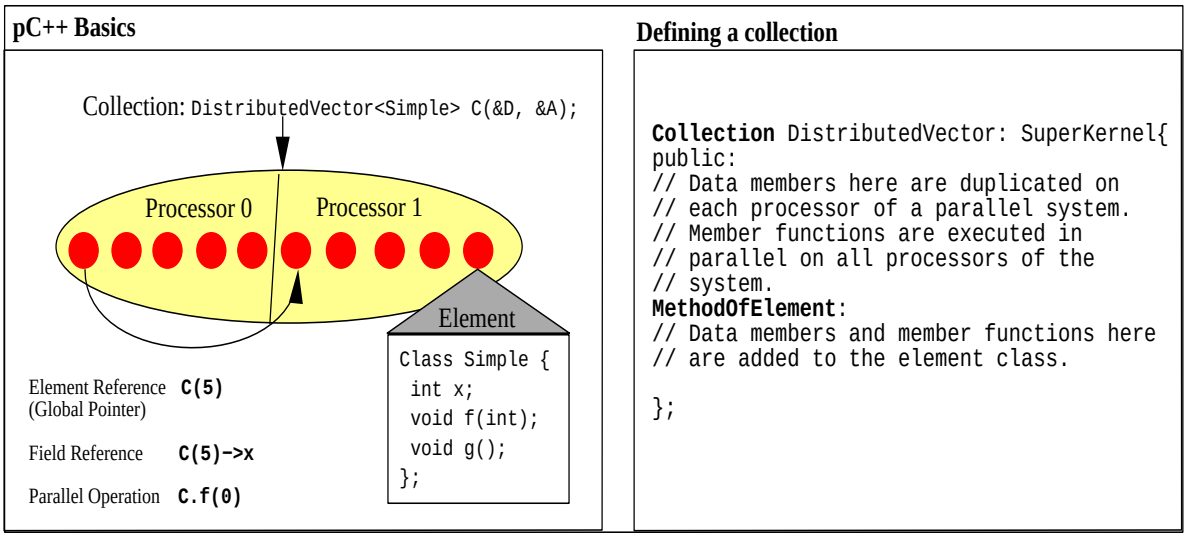

Figure 3: **IN pC++ A DATA STRUCTURE CALLED** *COLLECTION* is used to describe a set of objects distributed over the processors of a parallel machine. One of the base collections that pC++ collection library provides – for building other collections – is the `SuperKernel` collection. It builds arrays of element objects and provides global name space for the element objects.
To access an individual member of a collection, one can use the overloaded operator `()` which returns a *global* pointer to an element. For example, `C(5)` returns a global pointer to the 5th element in the collection. A global pointer is a pointer that spans the entire address space of a distributed memory machine. But the most important feature of a collection is the ability to apply a function in parallel across all the element objects. For example, `C.f(0)` is a parallel application of the method `f()` to all the elements of the collection with argument `0`.
The box on the right illustrates how a collection is defined. There are two types of data and member functions in a collection definition. Data members and functions labeled as *MethodOfElement* represent new functions and data that are added to each element class. The member functions are invoked just as if they are member functions of the element class. Other data member not labeled as MethodOfElement are duplicated on each processor and the functions are invoked in the SPMD (Single Program Multiple Data) mode.

This section describes the details of direct execution simulation for high-level pC++ programs. It starts off with a brief description of pC++ and N-body applications written in it. Next it describes the architecture of the system for direct execution simulation of pC++ programs. It then runs through a complete example showing the transformations performed on an actual pC++ program. It concludes with a description of the runtime system

One Dimensional Particle list

```
Class Segment {
 public:
 int pcount; // Number of particles
 double x[pcount],y[pcount],z[pcount];
 double vx[pcount],vy[pcount],vz[pcount];
 double mass[pcount];
};

Collection ParticleList: SuperKernel {
 public:
  ParticleList(Distribution *D, Align *A);
  void sortParticles();
 MethodOfElement:
  void pushParticle(Mesh<MeshElement> &G);
  void updateGridMass(Mesh<MeshElement> &G);
    :
    :
};
```

Three Dimensional Mesh

```
Class MeshElement {
 public:
 double mass[nz], position[nz];
};

Collection Mesh : SuperKernel {
 public:
 Mesh(double lx, double ly, double lz ...);
 void computePotential();
  :
  :
};
```

The Simulation Loop

```
main()
{
 Processors P;
 Distribution DP(seg_count, &P, BLOCK);
 Align AP(seg_count, "[ALIGN(particlelist[i], DP[i])]");
 Distribution DM(x_count, &P, BLOCK, WHOLE);
 Align AM(x_count, y_count, "[ALIGN(mesh[i], DM[i])]");

 ParticleList<Segment> particleList(&DP, &AP);
 Mesh<MeshElement>     mesh(&DM, &AM);

 // Initialize particle list
 ...
 // main loop

 for(int i=0; i < nsteps; i++) {
  mesh.computePotential();
  particleList.pushParticle(mesh);
  if (needed) particleList.sortParticles();
  particleList.updateGridMass(mesh);
 }
 ...
}
```

Figure 4: **THE 1D PARTICLE LIST AND THE 3D MESH** are the primary distributed data structures in the particle mesh code. The one dimensional particle list comprises of segments and each segment has several particles in it. For each particle the position (*i.e.*, `x`, `y` and `z` co-ordinates), velocities (*i.e.*, velocities in the `x`, `y`, and `z` directions `vx`, `vy`, `vz`), and mass (`mass`) are stored. The three dimensional mesh is logically a three dimensional array of mesh points each containing the value of density and position.
The function `sortParticles()` is a routine that sorts the particles in lexicographic order according to their positions. The function `pushParticle()` uses the gravitational force to update the positions and velocities of the particles. The argument passed to `pushParticle()` is a collection designed for the mesh data structure. The function `updateGridMass()` is used to add the mass of a particle to the total mass of the mesh point to which it is closest.
The mesh element, which contains an array of mesh points that have the same x and y coordinates but different z coordinates is shown on the box to the right. In the `Mesh` collection the function `computePotential()` computes the gravitational potential using the total mass at each mesh point and `lx`, `ly`, `lz` are the numbers of mesh points in the three x, y, and z dimensions, respectively.
The main loop involves parallel computation on both `Mesh` and `ParticleList`. First the potential is computed in parallel on the grid. Second, the particle velocities and positions are updated. If particles have moved to new grid points, the appropriate data structure updates must then be made. If the particles have moved a lot since the last time they were sorted, the particles are sorted again. Third, the particle masses are accumulated in their corresponding points for the next iteration step.

implementation.

2.1 pC++ basics

pC++ [6] is a simple extension to C++ that supports a *data parallel* style programming. To accomplish this, pC++ provides a very simple mechanism to build "collections of objects" of a base *element* class. Member functions from this element class can be applied to the entire collection (or a subset) in parallel.

pC++ has two basic extensions to the C++ language:

- A mechanism to declare a distributed data structure comprising of objects distributed over the parallel machine.

- A mechanism to describe how operations can be invoked over a set of objects in parallel as well as to refer to an individual object.

Figure 3 succinctly illustrates the basics of pC++.

2.1.1 N-body application codes

This section illustrate the use of collections to define parallel data structures for the N-body simulation codes which are a part of the GC^3, the Grand Challenge Cosmology Consortium, project. More details can be found in [15]. The primary reason for describing it here is that these application programs were the basis for language extensions and data structure visualizations performed under the simulation environment describe later on.

There are two distributed data structures that natural for the N-body code: a one-dimensional particle list and a three dimensional mesh. Both of these distributed data structures are illustrated in **Figure 4** along with the skeleton of the simulation loop.

2.2 Architecture for direct execution simulation of pC++ programs

Figure 5 illustrates the architecture for direct execution pC++ programs.

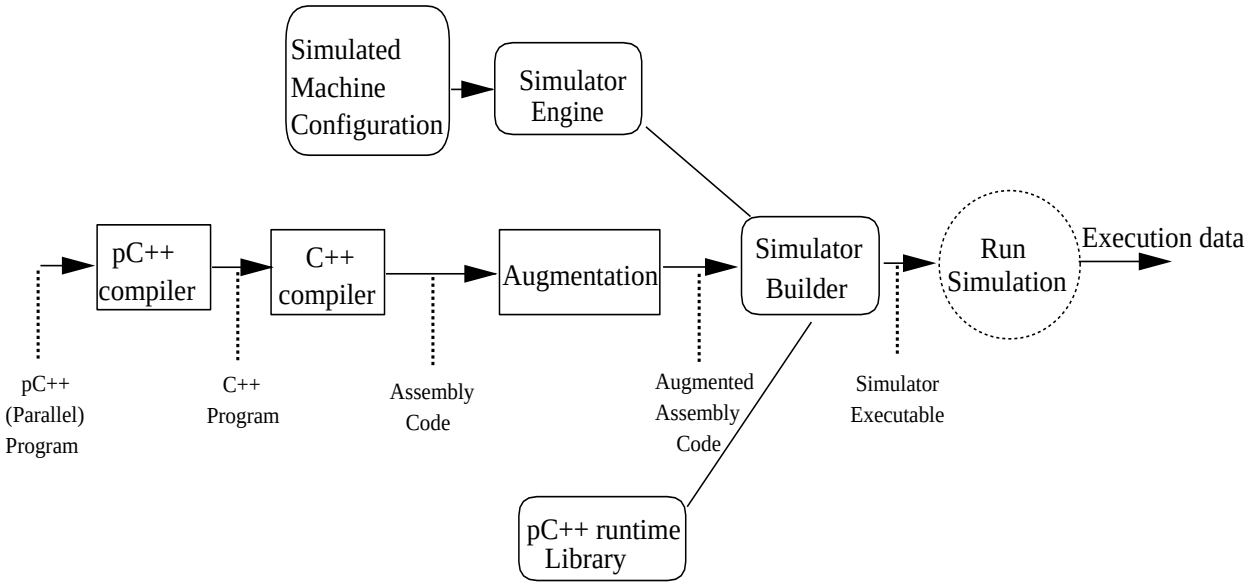

Figure 5: **ARCHITECTURE FOR DIRECT EXECUTION SIMULATION** of pC++ programs: The pC++ program is first preprocessed by the pC++ compiler to produce C++ code as shown in **Figure 6**. The C++ code is then converted into assembly code by the native C++ compiler. This assembly code is then augmented with additional assembly code – as shown in **Figure 7** – that is mainly responsible for a) Counting instruction cycles in the direct executed assembly code, b) Context switching between the simulated processors when the time quantum for the processor that is simulated expires. The assembly code is then linked in with the simulator engine and the pC++ runtime library to give a executable program. This program when executed mimics the execution of pC++ program on the simulated parallel machine. The simulated machine configuration provides information on the number of processors in the simulated machine, the type of interconnection network, and it's characteristics *etc.*, .

The simulator engine works quite like an operating system. It multiplexes a single resource (the workstation's processor) among the processors and network of the simulated machine. Also it interleaves the execution of the

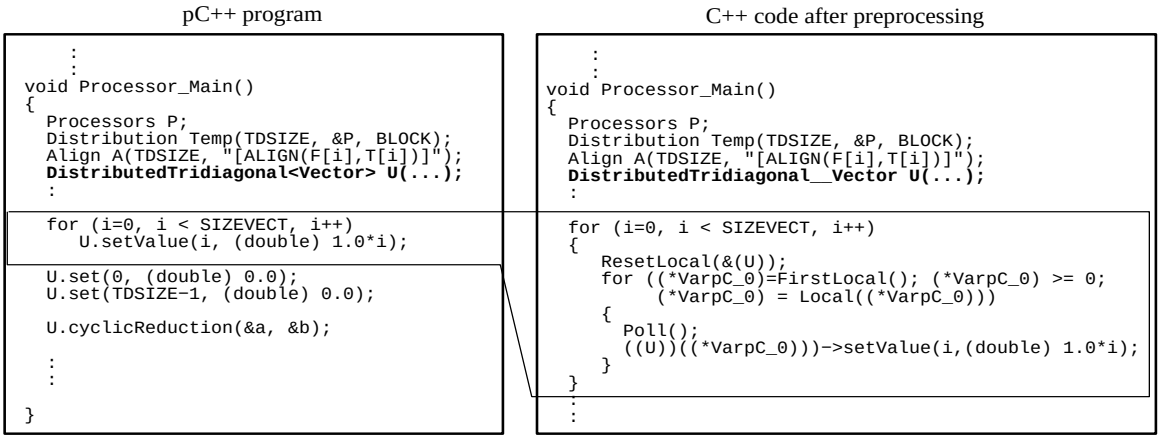

Figure 6: **EXAMPLE ILLUSTRATING pC++ COMPILER TRANSFORMATIONS** on a portion of the cyclic reduction program. The collection (distributed data structure) declaration is shown in bold face. The transformation that is applied to the parallel method invocation is shown in the enclosed box. The pC++ compiler converts the parallel operation on the distributed data structure into a loop which performs the operations on all the local elements on the processor. The compiler temporary is a pointer to an integer due to an artifact of the implementation under the simulator.

parallel application program with the simulation of the parallel machine such as simulating the message passing between the simulated processors. It is based on an event driven simulation mechanism.

The pC++ runtime library manages all the inter-processor communication. The runtime library is implemented using the primitives the simulator provides for inter-processor communication and is described more in detail in the next section.

The sequential implementation runs on the Silicon Graphics machines and Sun SPARC's.

2.3 An example

This section steps through an example - a pC++ application from the test suite - to show how the various components of the architecture transform the program. **Figure 6** illustrates the transformations that are performed by the pC++ compiler and **Figure 7** illustrates the transformations introduced by the augmentation phase.

2.4 The runtime system

A key idea in the design of pC++ is to enable programmers to build distributed data structures whose data movement operators can be closely linked to the semantics of the datatype. The programmer thinks abstractly about the distributed data structure and does not worry about the inter-processor communication in sharp contrast to the message passing paradigm where explicit send's and receives are used.

It is then the responsibility of the runtime system and the compiler to provide the abstraction of distributed data structures. The runtime system has to manage:

- The partitioning of the distributed data structure among the processors of the parallel machine.
- The references to portions of the distributed data structure that are not local to the processor causing the references.

As shown in the simulation architecture figure the augmented program is linked in with the pC++ runtime system (written using the simulator primitives) and the code that corresponds to the configuration of the parallel machine being simulated to produce an executable.

This executable when run simulates the execution of the high−level pC++ parallel program on a Sun SPARC workstation.

The augmentation phase has been modified to work on SGI workstations as well.

Assembly code after compilation

```
_Processor_Main:
.stabn 68,0,1845,LM1010
LM1010:
        !#PROLOGUE# 0
        save %sp,−4096,%sp
        add %sp,−2456,%sp
        !#PROLOGUE# 1
.stabn 68,0,1847,LM1011
LM1011:
LBB60:
.stabn 68,0,1848,LM1012
LM1012:
        add %fp,−56,%o0
        add %fp,−12,%o2
        mov 257,%o1
        mov 1,%o3
 call___12DistributioniP10Processorsi,0
        nop
        :
        :
```

Assembly code after augmentation

```
_Processor_Main:
.stabn 68,0,1845,LM1010
LM1010:
        !#PROLOGUE# 0
        save %sp,−4096,%sp
        ta      0x20
        sethi   %hi(_stackmin_),%l0
        ld      [%l0 + %lo(_stackmin_)],%l1
        subcc   %sp, %l1, %g0
        bgu     L11251
        ta      0x21
        call    _SimStack
        nop
L11251:
 ! block 0: 8 instructions
        ta      0x20
        sethi   %hi(_cycles_), %l3
        ld      [%l3 + %lo(_cycles_)], %l1
        subcc   %l1, 8, %l1
        st      %l1, [%l3+%lo(_cycles_)]
        bpos    L11252
        ta      0x21
        call    _SimQuantum
        nop
L11252:
        add %sp,−2456,%sp
        !#PROLOGUE# 1
.stabn 68,0,1847,LM1011
LM1011:
LBB60:
.stabn 68,0,1848,LM1012
LM1012:
        add %fp,−56,%o0
        add %fp,−12,%o2
        mov 257,%o1
        mov 1,%o3
        call___12DistributioniP10Processorsi,0
        nop
        :
```

Figure 7: **EXAMPLE ILLUSTRATING AUGMENTATION OF SPARC ASSEMBLY** code of the same cyclic reduction program. The augmentation phase introduces: **a)** Stack checks at the beginning of the subroutine. **b)** Cycle counting code for the basic blocks. **c)** Calls to `SimQuantum` to determine if the processor that is being simulated has exceeded its time quantum. The assembly code shown here is a small portion of the complete assembly code for the program.

- The termination of parallel operations on the distributed data structure using barrier synchronization.

The runtime system was completely written using the primitives that the simulator engine provided. The simulator engine provides mechanisms for interrupting a processor thread from another thread and the runtime system manages the non-local distributed data structure references using that mechanism. A tree barrier synchronization routine synchronizes the parallel operations. **Figure 8** contains a portion of the runtime system software. This contains the core of the implementation.

One of our first applications after was to build a simple debugger for pC++ on top of `gdb`. The debugger written as an extension to the gdb mode `gdb.el` in `elisp`. The new primitives allowed:

- Displaying the entire distributed data structure. For example `pc++print mesh.mass` would display the `mass` field in the mesh collection.
- Allowing break points on the distributed data structure methods. For example `pc++break mesh::computePotential()` would place a breakpoint on the element method `computePotential`.

Remote Access Management

```
int AskForData(int where, int index, int offset,
               int info, int *data1, int *data2 ...)
{
  if (where != mynode()) {
    *message_flag = FALSE;
    SUM_METRIC(no_of_nonlocal, 1.0);
    send_ipi(where, 2, READ_DATA_MSG, ...);

    while (*message_flag == FALSE) {
     CYCLE_COUNTING_OFF;
     PassControl(); /* Return to Simulator Engine */
     CYCLE_COUNTING_ON;
    }
    *message_flag = FALSE;
    *data1 = *read_buf1;
    :
    :
}
```

Spawning the SPMD threads

```
static
void spawn_spmd(int argc, Word argv[])
{
 thread_create_and_wakeup((FuncPtr) Processor_Main,
                          PM_STKSIZE ...);
}
GLOBAL void usermain(int argc, char **argv)
{
  InitRunTime();
  InitBarrier();
  SPAWN_SPMD = define_new_ipi((FuncPtr)spawn_spmd,
                              "ProcessorThread");
  for (i=1; i < NO_OF_PROCESSORS; i++)
   send_ipi(i, 2, SPAWN_SPMD, ...);
  Processor_Main();
  PassControl();
}
```

Figure 8: **PORTION OF RUNTIME SYSTEM SOFTWARE** for direct execution simulation of pC++ programs. The box on the right illustrates the creation of lightweight threads of control on each of the processors. `Usermain` is called from the simulator engine by Processor 0 and it performs initialization for barrier synchronization as well as the various tables for the runtime system. All the other processors would be in an idle state at that point of time. Processor 0 then interrupts all the other processors and request them to spawn the thread of control corresponding to `Processor_Main` which is the routine provided by the pC++ programmer. This portion of the code is highlighted; `send_ipi` is a system call for sending an interprocessor interrupt. On receipt of the interrupt the remote processor executes a handler which in this case is the routine `spawn_spmd` and creates the lightweight thread corresponding to `Processor_Main`. The box on the left illustrates the portion of the runtime system which manages the remote accesses by using the `send_ipi` system call. It is called from the `SuperKernel` class which was introduced in the pC++ basics section. Notice that while the processor is waiting for the data which it requested it transfers control to the simulator engine and also indicates not to count the cycles.

The extensions are really simple. They know about the de-mangling of pC++ collections by the pC++ compiler and by virtue of executing in a single address space they can display entire distributed data structures.

The simulator with the debugger was useful in fixing a bug in one of the NASA's NAS benchmarks (the CFD benchmark) – that our group rewrote in pC++ – which was not apparent earlier. All of the pC++ test suite programs (7 medium size parallel programs) were moved to work under the simulator.

Coupling the runtime system with the simulator also allowed detailed high level (distributed data structure level) information to be recorded. For example it becomes possible to determine the communication to computation ratio of time spent in the distributed data structure `mesh` or `particleList` in the N-body code. Since parallelism is restricted to distributed data structure operations this gives an accurate profile of the distributed data structure operations that are using the parallel computer's (computation and communication) resources.

Our experience suggests that the development time for sophisticated runtime systems and debuggers for parallel programming languages are much shorter using a parallel simulator than on an actual machine. Also much of the software infrastructure on sequential workstations can be leveraged to give a better parallel program development environment.

3 Experiences with direct execution simulation of pC++ programs on workstations

Our experience with direct execution simulation on workstations in the context of pC++ has been in experimenting with runtime systems, language extensions, debugging, and data structure visualization. This section describes briefly the experiences with prototyping language extensions, data structure visualization, and strategies for improving the speed of the simulation.

3.1 Prototyping language extensions

As mentioned in the previous section the distributed data-structures in pC++ are called Collections and they are homogeneous and contain classes as elements. If `c` is a collection of type `C<E>` then any processor thread may access the *i*th element of `c` by the syntax `c(i)`. The () operator is overloaded to provide this functionality.

In the first version of the pC++ language there was no support for remote execution at the element level. Such a support was needed for implementing one of the N-body astrophysics codes in pC++. The idea is that just as a possible non-local element can be accessed from another element with the syntax `c(i)`, it should also possible to run a method `f` on a possible non-local element using the syntax `c(i)->f()`. This technique is used in implementing the function `pushParticles` to update particles on remote processors.

The implementation was non-trivial and is described in more detail in the next paper. The entire language extension was written and debugged under the simulator using compiler transformations and extensions to the runtime system. The compiler transformations were implemented using Sage++[17], the object oriented toolkit for building program transformations systems for Fortran90 and C++. The runtime system was extended to handle remote execution messages. The final debugged version was moved to work on the CM-5 fairly quickly.

3.2 Data Structure Visualization with direct execution

The direct execution simulator for pC++ programs was the substrate for many of our distributed data structure visualization experiments [18]. The idea was to show using visualization and animation the time based evolution of the distributed data structure and the inter-processor communication that occurs in distributed data structure references. The following figure (**Figure 9**) shows the snapshot from an animation of particle densities in a distributed adaptive particle mesh during the course of a N-body simulation. The adaptive version of the code is similar to the version that described in the introductory section but there are two meshes a coarser mesh and a finer mesh.

The visualizations and animations were extremely useful and they revealed several bugs in pC++ programs. Two examples are the bug in the copying stage of an adaptive mesh refinement computation and a bug in the distribution portion of the runtime system.

Our experiences show that direct execution simulators provide an alternative and fast development time for the visualization of high-level parallel programs than traditional methods which were performed by collecting traces by executing the parallel program on an actual parallel machine. Using direct execution simulation both interactive and post-mortem visualizations can be performed unlike the traditional method which allowed only post-mortem visualization. Also collecting traces are a lot simpler on workstations since familiar sequential I/O mechanisms can be used in contrast to the more complex I/O on parallel machines.

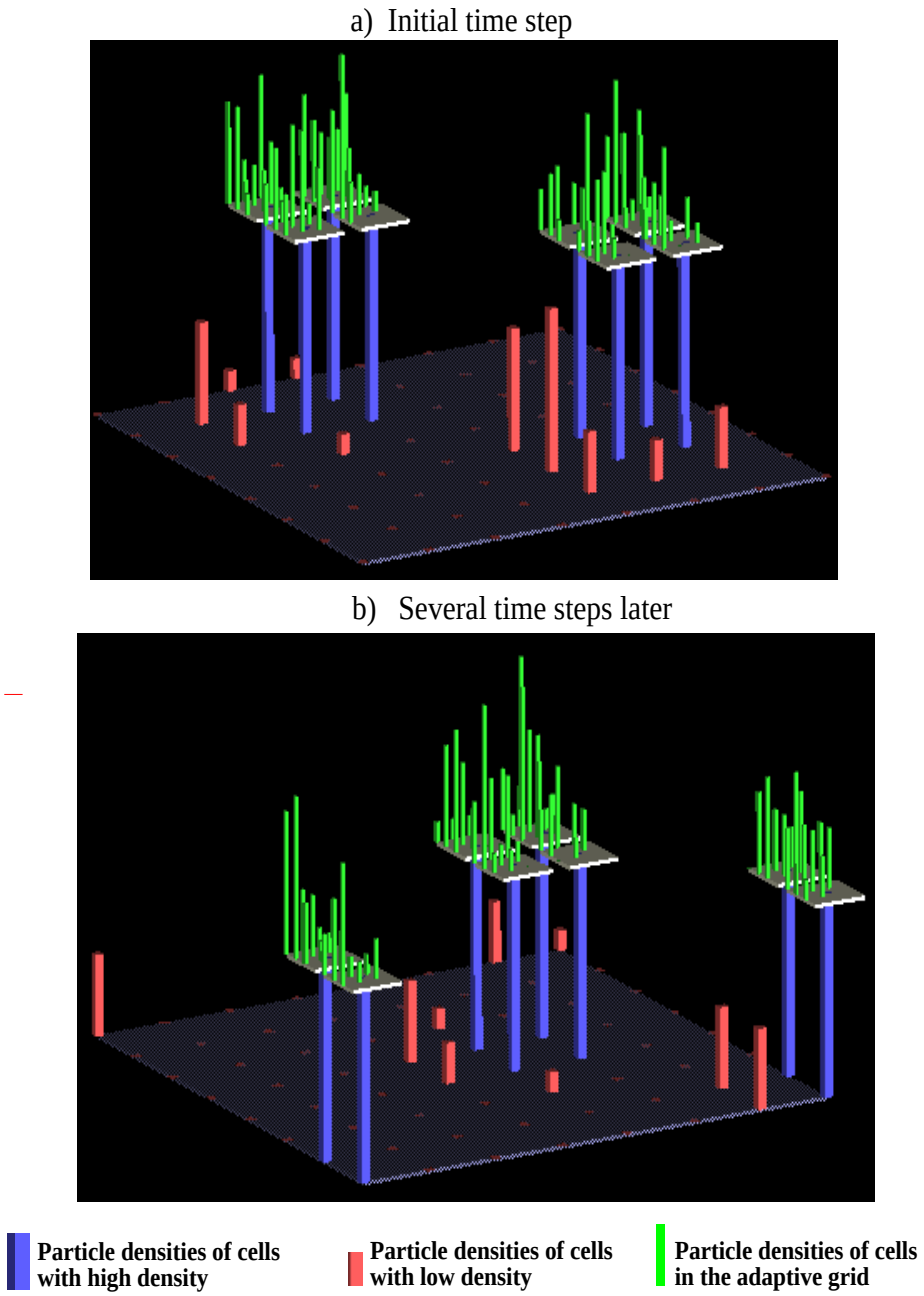

Figure 9: **SNAPSHOTS FROM THE ANIMATION** of particle densities in an N-body pC++ program (view in color).

4 Strategies for improving the speed of the simulation

High performance is an important factor in building a fast parallel architecture simulator. There are several possible ways of improving the performance of a direct execution simulator:

- Performing fast context switches among the simulated processors or reducing the total number of context switches.
- Using parallelism.

This section discusses how parallelism can be used to improve the speed of the simulator. The applications for this as mentioned in the introduction are in studying and debugging the performance of parallel applications under large problem sizes or large machine configurations.

The parallel programming model that is used in pC++ is similar to the BSP [19] model. The parallel program alternates between a sequential single threaded mode and a parallel multi-threaded mode which has local computation, communication and global synchronization. An implementation on a distributed memory machine would duplicate the globals as well as the single thread on all the processors.

Given this model there are two opportunities for parallelizing the simulation of the parallel application running on a distributed memory machine:

- Executing the single threaded portion in parallel.
- Using a parallel discrete event simulation for the executing the parallel multi-threaded portions.

Figure 10 illustrates the programming model and parallelized implementations of the simulator.

Our implementation at the present time is based on parallelization using the first method and runs on a 20 processor SGI Power Challenge recently acquired by Indiana University. The parallel execution driven simulator was implemented using the functionality provided by the SGI's IRIX parallel processing library. The library provides functions that allow processes to be spawned and these processes can run on different processors but they can share the same address space, it also provides parallel synchronization primitives such as semaphores.

Parallelization a sequential simulator involves several changes to the augmentation portion of the simulator:

- The parallel portions of the application do not need context switching and the augmentation in the parallel portions should not include the context switching code. This also has the effect of reducing the total number of context switches.
- The parallel portions have to maintain a per-thread cycle counter instead of a global cycle counter in the sequential simulator. Also additional code to perform a addition of all the per-thread cycle counters has to be introduced when the parallel portions terminate.

Also referring to the program globals in the parallel portion of the simulator is handled differently. The parallelized implementation at present does not work for pC++ programs but has been shown for simple parallel C programs. A compiler pre-processor is needed for performing the mapping. It has to perform transformation to generate additional code to create parallel threads of execution as shown in the first simulation execution model figure. Using that we would be able to do scalability studies on large-scale pC++ programs.

5 Related work and Future directions

This paper is unique in illustrating the applications of direct execution simulation in the context of a high level programming language. It also shows the importance of both sequential versions and parallel implementations of simulation for developing application and system software for parallel machines.

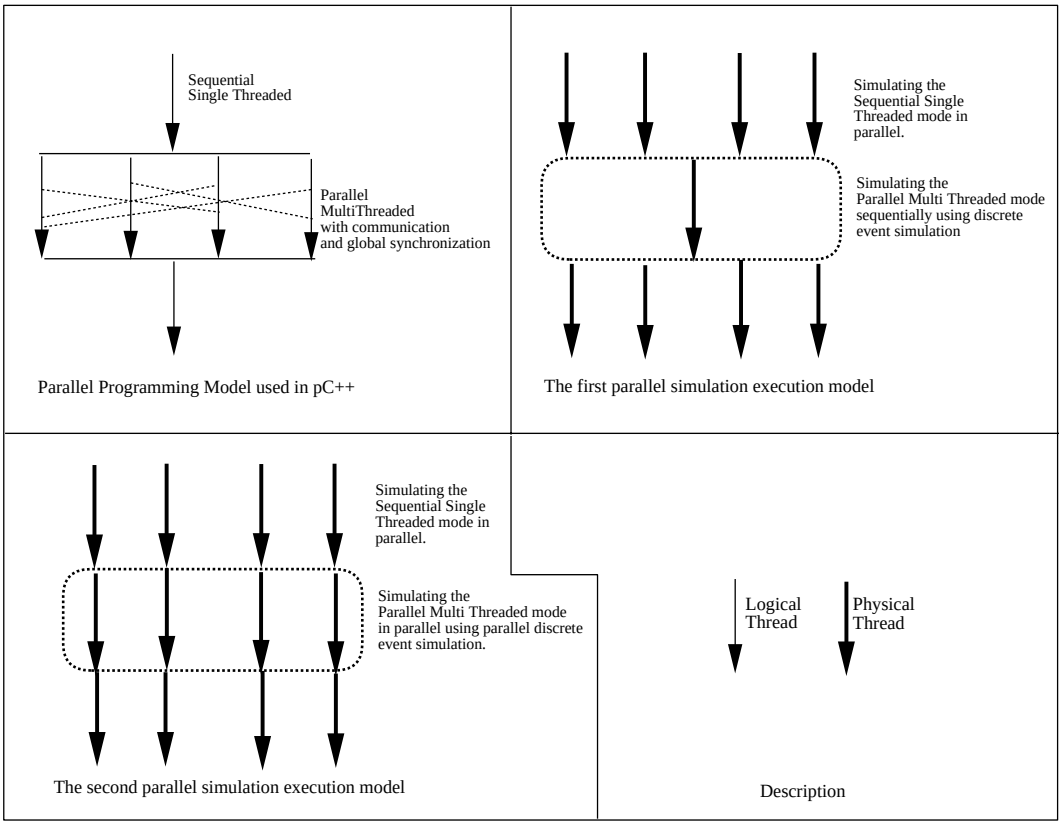

Figure 10: **PROGRAMMING MODEL AND MAPPINGS TO PARALLEL SIMULATION EXECUTION** on distributed memory machines for high-level parallel programs. The box on the upper left represents the application programmers conceptual model. The box on the upper right and the lower left illustrate the use of parallelism for simulating their execution on distributed memory machines. The approach in this section is as described in box on the upper right.

The system is built on top of the Proteus [11] from MIT. Proteus has support for simulating both shared memory multiprocessors and distributed memory multiprocessors. Only the distributed memory support in Proteus is used for all of our simulations.

FAST [13] from Berkeley is another sequential execution driven simulator that concentrates on managing context switching well to simulate large shared memory multiprocessors. Tango [20] and TangoLite [12] from Stanford are both memory system simulators. Tangolite supports both execution-driven simulation and trace-driven simulation.

None of the sequential simulators except Proteus for some degree discuss how they have applied the simulator to enable the development of better parallel software. Also by virtue of running on workstations they do not allow scalability studies on large-scale parallel applications.

Wisconsin Wind Tunnel (WWT) [7] from University of Wisconsin by using execution-driven, parallel discrete-event simulation allows evaluation of larger and more realistic applications and system software. They have performed several interesting studies using the WWT. But the WWT has several disadvantages:

- It is specific to Wisconsin's CM-5. Their implementation is based on several kernel modifications and other researchers cannot use their simulator easily.

- They choose to perform the simulation on a massively parallel machine (the CM-5) which has very high cost and more controlled (i.e less interactive) access.

LAPSE [8] is much more general but has the restrictive goal of performing scalability and performance analysis only on Intel Paragon applications. It has a similar disadvantage of choosing a massively parallel machine (the Intel Paragon) to perform the simulation. But the results obtained from both Wisconsin Wind Tunnel and LAPSE really show the potential for parallel execution driven simulators.

Both WWT and LAPSE provide support for one type of parallel machine. WWT allows simulation of shared memory multiprocessors and provides much support for cache-coherency. And LAPSE provides support only for distributed memory and message passing applications. We believe with the convergence in technology that support in parallel simulators for both kinds of machines are important.

The massively parallel machines are still in a state of flux [1] and providing parallel simulators for developing parallel software on them makes it too restrictive. As we mentioned in the introduction, high performance servers using the symmetric shared memory multiprocessing technology are a much better platform for building parallel simulators.

Input/Output (I/O) which is well understood on sequential machines is rather poorly understood on parallel machines. We also believe that simulators that support simulation of the I/O subsystem (all the way down to the simulation of parallel RAID disks) would be invaluable in building better I/O systems as well as implementing better abstractions for I/O in parallel programming languages. This could also be the substrate for designing the future parallel object oriented databases which would be based on object persistence.

6 Summary

This paper described our experiences with direct execution simulation of parallel machines on workstations and servers, in the context of the high level object oriented parallel language pC++. It also showed that implementing a simulator on a parallel server allows the simulation of large parallel machine configurations and large-scale applications.

To summarize the lessons learned from our experiences:

- By using simulation for development the existing software infrastructure on workstations can be leveraged resulting in shorter development cycles.
- Implementing a high level parallel language's runtime system on top of a simulator allows programs in that high level language to be executed on simulated machines. This execution yields detailed as well as high level information (for example distributed data structure level information) about the program execution. This information can be used for visualizing parallel program behavior and for identifying potential performance bottlenecks caused by distributed data structures.
- Execution driven simulation on workstations is sufficient for debugging small scale parallel applications, prototyping runtime systems, and extending high level parallel languages.
- Execution driven simulation on high performance servers makes possible the simulation of large parallel machine configurations and large-scale applications since servers have large physical memories. This enables the scalability analysis and performance tuning of large-scale parallel applications.

[1] Considering the recent demise of both Thinking Machines and Kendall Square Research

- Parallelizing simulators is important for reducing the simulation time. Implementing the simulator on a parallel server makes it accessible and inexpensive compared to implementations on massively parallel machines like the CM-5 or Paragon.
- With the convergence of shared memory and message passing technologies simulators should soon provide support for simulating both shared-memory and distributed memory parallel machines.

References

[1] Intel Supercomputing System Division. *Paragon Users Guide*. Available on World Wide Web (WWW) http://www.ccsf.caltech.edu/paragon/man.html.

[2] Cray Research Incorporated. *Online information on the Cray T3D Product*. Available on the WWW from `http://www.cray.com/PUBLIC/product-info/mpp/CRAY_T3D.html`.

[3] Mike Galles and Eric Williams. Performance optimizations, implementation and verification of the sgi challenge multiprocessor. Technical report, Silicon Graphics Computer Systems. Available on WWW from http://www.sgi.com/tech/challenge_paper.html.

[4] Online information on SPARC 2000. On WWW from http://www.sun.com/smi/bang/2000.html.

[5] Online information on HP 9000. Available on WWW from http://www.sun.com/smi/bang/2000.html.

[6] D. Gannon, S. X. Yang, and P. Beckman. *User Guide for a Portable Parallel C++ Programming System, pC++*. Computer Science Department, Indiana University, Available on WWW from http://www.cica.indiana.edu/sage/home-page.html, 1994.

[7] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David Wood. The Wisconsin Wind tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, May 1993. Available from http://www.cs.wisc.edu/p/wwt/Mosaic/wwt.html.

[8] Phillip Dickens, Philip Heidelberger, and David Nicol. Parallelized direct execution simulation of message-passing parallel programs. Technical Report ICASE/94/94-50, Institute for computer applications in science and engineering, NASA Langley, 1994. Available on WWW from ftp://ftp.icase.edu/pub/techreports/94/94-50.ps.Z.

[9] Anant Agarwal et al. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Workshop on Scalable Shared-Memory Multiprocessors*. Kluwer Academic Publishers, June 1990. Available on the World Wide Web from http://cag-www.lcs.mit.edu/alewife/.

[10] R.G. Covington, S.Dwarkadas, J.R.Jump, J.B.Sinclair, and S.Mandala. Efficient simulation of parallel computer systems. In *International Journal in Computer Simulation*, volume 1, pages 158–168, 1991.

[11] Eric A. Brewer, Chrysanthos N Dellarocas, Adrian Colbrook, and William Weihl. Proteus: A high peformance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991. Available on WWW from file://ftp.lcs.mit.edu:/pub/supertech/papers/.

[12] Stephen R Goldschmidt. *Simulation of multiprocessors: Accuracy and Performance*. PhD thesis, Stanford, June 1993. Available on the WWW from ftp://meadow.stanford.edu:/pub/tango.

[13] Bob Boothe. Fast acurate simulation of large shared memory multiprocessors. Technical Report UCB:CSD-93-752, University of California at Berkeley, June 1993. Available on WWW from http://cs-tr.cs.berkeley.edu/TR/UCB:CSD-93-752.

[14] Eric A. Brewer and William E. Weihl. Developing parallel applications using high-performance simulation. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 158–168, May 1993. Available on WWW from file://ftp.lcs.mit.edu:/pub/supertech/papers/WPDD93-simulation.ps.Z.

[15] D.Gannon, S.Yang, S.Srinivas, V.Menkov, and P.Bode. Object-oriented methods for parallel execution of astrophysics simulations. In *Proceedings of Mardigras94 Teraflop conference*, February 1994. Available from gannon@cs.indiana.edu.

[16] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Supercomputing 93*, Available on WWW from http://www.cica.indiana.edu/sage/home-page.html, 1993. IEEE Computer Society.

[17] D.Gannon, P.Beckman, F.Bodin, J.Gotwals, S.Narayana, S. Srinivas, and B.Winnika. Sage++: An object oriented toolkit for program transformations. In *Proceedings of Oonski 94*, April 1994. Available by anonymous ftp from moose.cs.indiana.edu:pub/sage/oonski94.ps.

[18] Suresh Srinivas. Visualizing distributed data structures. In *To appear in Proceedings of Frontiers of Massively Parallel Computation*, McLean, Virginia, February 1995. Available from ssriniva@cs.indiana.edu.

[19] L.G. Valiant. A bridging model for parallel computation. *CACM*, 33(8):103–111, August 1990.

[20] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor simulation and tracing using tango. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, May 1993.