

Distributed Simulation of Parallel DSP Architectures on Workstation Clusters

Alan D. George and Steven W. Cook, Jr.¹

High-performance Computing and Simulation (HCS) Research Laboratory
Electrical Engineering Department, FAMU-FSU College of Engineering
2525 Pottsdamer Street, Tallahassee, FL 32310
e-mail: george@hcs.eng.fsu.edu

The limits of sequential processing continue to be overcome with parallel and distributed architectures and algorithms. In particular, the use of parallel processing for high-performance signal processing and other grand-challenge applications is becoming the norm. However, the costs associated with such systems make it critical to be able to forecast system behavior before a hardware prototype is constructed. Processor libraries and other functional modeling and simulation techniques allow the complexities of actual systems to be portrayed in the form of software-based prototypes with significantly more accuracy than traditional simulation methods. Although the simulation times often grow beyond practical limits, distributed simulation methods have the potential to reduce these times in a scaleable fashion. One such method is to implement processor library models with a parallel programming language on perhaps the most flexible, available, cost-effective, and practical of all parallel computing platforms, the workstation cluster. This technique can in many cases achieve near-linear simulation speedup using existing computer resources.

Keywords: distributed simulation, parallel computing, signal processing, processor libraries

Introduction

As shown in previous work, fine-grain functional simulations of parallel digital signal processor (DSP) systems can rival the fidelity of the actual hardware, but at a fraction of the cost to construct [1]. Processor object libraries with a clock-cycle granularity are employed along with high-level language code to simulate and provide the connections between processors and other glue logic (e.g. address decoders, latches, and voters) so that a prototype can be constructed and exercised with actual machine-language programs running on the simulated processors. Furthermore, faults can be injected into the simulation-based prototype in order to determine system behavior in the presence of hardware or software failures, and in all cases the results are measured with an accurate accounting of execution time measured in clock cycles.

While these efforts have led to the study of several different parallel DSP architectures and systems, there are often occasions when the response time of the simulator itself can be of critical importance. The complexity associated with simulating dozens of VLSI processors and their glue logic with clock-cycle granularity is sufficient to tax the resources of any conventional workstation or personal computer. One promising approach to solving this problem is to apply techniques in parallel and distributed computing so that the resources of many processors can contribute to the computations in the form of a distributed simulation.

One particularly interesting, flexible, practical, and widely-available platform for parallel and distributed computing is the workstation cluster. By harnessing even the idle time of the workstations available at an average government, industrial, or university facility the potential exists to provide significant speedups in simulation time with virtually no added hardware or software costs [2].

The objective of this paper is to present the mechanisms used and the results obtained in the application of workstation clusters for the distributed simulation of parallel DSP computer systems. Using a test-bed consisting of conventional UNIX™ workstations connected by a 10-Mbps local area network (LAN) and parallel programming tools like Linda™, we have been able to realize a significant reduction in simulation time by segmenting the model and providing enough coarse-grain work at each workstation to compensate for the network communication overhead associated with LAN-based cluster computing.

Parallel DSP simulation with processor libraries

Developing complex parallel computer systems with advanced microprocessors can be a difficult, time consuming, and expensive task. By using simulation software like that provided by Motorola for its DSP microprocessor devices [3, 4], it is possible to design, implement, and evaluate such systems completely in software.

¹ Mr. Cook is now with Monsanto Inc. in Gonzalez, Florida.

For example, the simulation software for the DSP96002, a 32-bit, floating-point device with dual system ports, consists of processor libraries which enable uniprocessor and multiprocessor simulation to take place with single clock-cycle granularity [5]. A *processor library* is defined to be an object library consisting of special functions or subroutines which can be called from a high-level language and together serve to completely and accurately simulate one or more processors. By writing appropriate interface code in C to call these functions, almost any architecture can be simulated. Although not executing in real-time, the simulation can be used to keep an accurate count of clock cycles executed, thereby allowing simple scaling to determine actual real-time execution rates. *As opposed to analytical models and studies, these simulations exactly represent the real machines without having to actually build them in hardware.* The results are in effect prototype machines built in software.

Thus, programs constructed using these processor libraries are able to exactly duplicate all of the internal and external operations of the DSP96002, such as register and memory updates associated with program execution, updates to individual pins from both internal and external sources, exception processing, etc. The pipelined bus activity of the device is exactly simulated, and the program executed by each simulated processor is the same machine code compiled or assembled for the actual chip. During the simulation runs, the user can display and modify any registers or memory locations, thereby providing a level of test and debug beyond that associated with the real hardware.

In order to simulate a working multiprocessor, each processor is first created using the *dsp_new* function. This function allocates and initializes the device state data structure that will completely describe a particular DSP96002 device throughout its execution. Based on a device index, the *dsp_ldmem* function is used to load the object code for each processor. The *dsp_exec* function then enables an individual processor to execute a single clock cycle, so that simulation of parallel execution can be achieved for each clock cycle by calling *dsp_exec* for all processors in turn followed by calls to simulate communication between devices for that cycle. To accomplish the latter, other functions provide low-level support for accessing individual DSP96002 pins, ports, registers, and memory locations. These primitive functions can be used to describe the detailed connections which are found in multiprocessor interconnection networks. For example, constant value connections to pins are simulated by executing a *dsp_wpin* function call for each relevant processor during every clock cycle using this value as the data parameter. Pin-to-pin connections are handled by executing *dsp_rpin* and a *dsp_wpin* function calls on the source and destination processors respectively during every clock cycle. Port-to-port connections are similarly handled using the *dsp_rport* and *dsp_wport* functions. Additional functions are provided for simulation support, such as the ability to save the state of the processors to a set of files for later recall.

Parallel DSP architecture example

By using the processor library approach coupled with code for interprocessor connections and glue logic, any number of parallel DSP architectures can be modeled and simulated. One architecture that has been studied recently is the FTDSP system which consists of a dual-mode architecture supporting both fault-tolerant (FT) and non-FT modes of operation. As shown in Figure 1a, the FT mode consists of three sets of cascaded processors (i.e. linear processor arrays or pipes), configured with majority voting at the outputs in a form known as triple modular redundancy (TMR) [6]. In Figure 1b, the non-FT mode is shown to be an adaptation of the FT mode whereby each linear array feeds the next, bypassing the voter, so that the system takes on a linear array architecture which is three-fold longer in length. These figures illustrate a sample system of nine processing elements (PEs) each consisting of a processor with local RAM, ROM, and interface logic, whereas the architecture itself is extendible to any size that is a multiple of three PEs.

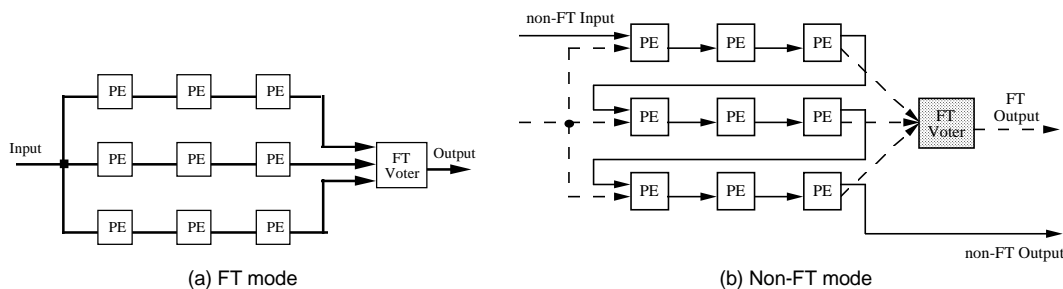


Figure 1. A 3-stage FTDSP Architecture in Fault-Tolerant and Non-FT Modes

Using the processor library techniques, a complete and fully-functional, software-based prototype has been constructed for this dual-mode system [7]. In addition to the simulation of the processors themselves, a number of interfaces also had to be simulated, including the PE-to-PE interfaces, the input-to-PE interfaces, and the PE-to-output interfaces with voting.

Based on a single workstation platform, a number of experiments have been conducted on this software-based prototype using basic DSP operations such as digital filtering and fast Fourier transformations [1]. These simulation studies provided the exact number of clock cycles required to initialize the processors, as well as the number of cycles required to compute each new output value.

While these results gave a clear indication as to actual real-time DSP processing rates which can be achieved by this architecture, the simulation runs would require many hours and often days to complete. For example, tests of a 3-stage FTDSP system (i.e. 9 PEs) conducted in 1991 on a 33-MHz 80386-based personal computer with 4MB of memory required almost five days to simulate one second of real-time execution. While this ratio may at first seem alarming, it should be noted that in reality many signal processing operations can be debugged and analyzed in periods of time measured in microseconds, and thus most simulation runs were measured in minutes or hours. While more recent simulation runs on a modern workstation have reduced the ratio of real-time to simulation-time from seconds-to-days to seconds-to-hours, there is clearly a need to address the often prohibitive run times associated with this high-fidelity simulation approach. One particularly attractive method to reduce this simulation time is with distributed simulation using a network of workstations coupled with a parallel programming environment such as Linda to form a workstation-cluster parallel computer.

Parallel Programming with Linda

Linda was originally conceived by David Gelernter and the first working version of Linda was developed with Nicholas Carriero [8-12]. Linda was first implemented on an AT&T S/net multicomputer and an Intel iPSC hypercube machine. Both of these computers are loosely-coupled distributed memory multiple-instruction, multiple-data stream (MIMD) computers. The first network-based implementation was on a cluster of DEC VAX/VMS computers connected via 10-Mbps Ethernet.

Fundamentally, Linda programs are either C or FORTRAN programs with extensions added for interprocessor coordination and communication via a logical shared-memory model. The C language version of Linda is referred to as C-Linda. C-Linda is an implementation for a loosely-coupled, distributed-memory multicomputer while another variant, Network C-Linda, is designed to operate on workstations connected by a network (e.g. Ethernet) in order to form a LAN-based multicomputer or cluster computer.

C-Linda functions as a coordination language. It provides the tools to extend the sequential code of a standard C program by adding four basic operations to provide access to the tuple space. Since these parallel operations are orthogonal to the high-level language, a programmer can make full use of the standard C operations and libraries. These extensions provided by C-Linda are independent of the system architecture and are portable across many different architectures.

The Linda paradigm is based on the model of a distributed-memory MIMD parallel computer. A *tuple*, the fundamental data object for Linda, consists of an ordered list of typed values. As illustrated in Figure 2, these tuples exist in a common tuple space, a shared virtual memory (SVM) logically accessed as an unordered associative memory of tuples and physically located on any of the computers currently active. Data is only transferred in and out of tuple space as a tuple, and cannot be altered within the tuple space. A tuple that contains static data is referred to as a data tuple, while a tuple that is under active evaluation is referred to as a live tuple. On a conventional sequential computer, memory storage is based on a byte or word and accessed by a physical address via two operations (i.e. read and write). By contrast, Linda accesses a tuple in tuple space by using one of several tuple space access operations along with a local name that can specify one or more values in the tuple space. These four operations are: *out*, *eval*, *in*, and *rd*. Since the Linda environment handles all the tuple space management functions, the programmer does not have to be concerned with how the tuple space is created or managed, or where it is physically located throughout the cluster of workstations.

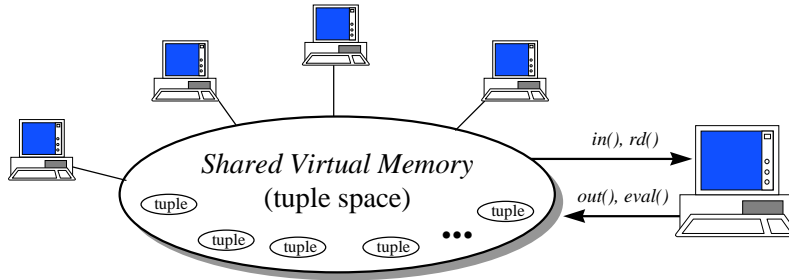


Figure 2. Shared Virtual Memory Model of Linda

The *out* and *eval* operations create new tuples; the difference is that the *eval* operation creates a live tuple while the *out* operation creates a data tuple. The *out* operation places a tuple in the tuple space. All of the fields are resolved into actual values before placing the data in the tuple. For example, the function $f(i)$ in Figure 3a would be evaluated first, and the resulting constant placed into tuple space. In contrast, the *eval* operation creates a process tuple to evaluate each argument. While the process executes on some particular worker (i.e. the first workstation to collect from tuple space and process the tuple), the tuple is considered to be an active or live tuple. As each process is completed, the results are placed back in the tuple space where they become a data tuple. As illustrated in Figure 3b, the *eval* operation creates a process to evaluate the function $f(i)$ before placing the results into tuple space.

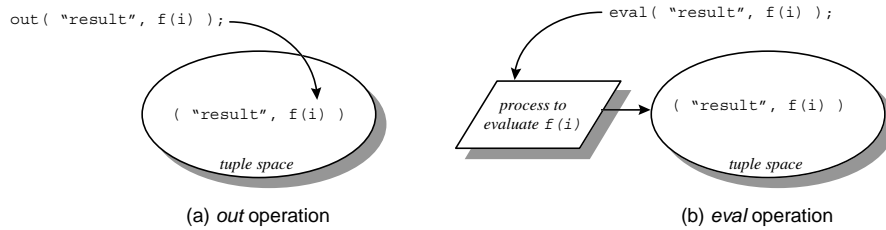


Figure 3. C-Linda *out* and *eval* operations

Both *in* and *rd* acquire information from the tuple space using a template as an argument. The template consists of one or more typed fields containing either *actuals* or *formals*. An actual is a field that contains a value, either a constant or an expression. A formal is a place holder for the data in the corresponding field of the tuple and begins with a question mark. The question mark indicates that a data value will be assigned to the same field of the template from a matching tuple. In Figure 4, the *in* operation removes a tuple from tuple space by searching for a tuple that matches the template. The *rd* operation behaves like the *in* operation, but leaves a copy of the tuple in tuple space.

Each formal in the template is set according to the value in the tuple. A template matches a tuple when three conditions are met. First, both the template and the tuple must have the same number of fields. Second, the types, values, and lengths of all the actuals in the template must be the same as the corresponding fields in the tuple. Third, the types and lengths of all formals in the template must match the types and lengths of the corresponding fields in the tuple. As illustrated in Figure 4, the *rd* operation and *in* operation search for a tuple matching the template. In each template, the placeholder *?i* is assigned the value of $f(i)$ from the matching tuple.

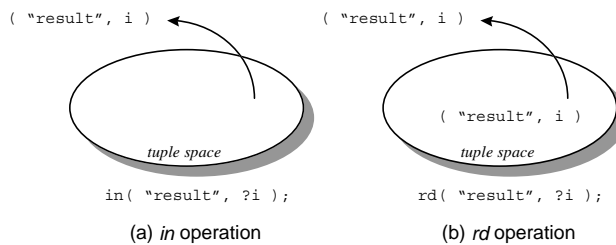


Figure 4. C-Linda *in* and *rd* operations

As a demonstration of C-Linda, a parallel version of the traditional "hello world" program is presented. While this parallel program does not perform any computations, it does provide a good demonstration of how to use the C language extensions for interprocessor communication.

```
#include <stdio.h>
real_main(argc,argv)          /* function executed by master */
int argc;
char *argv[];
{
    int worker, j, hello();

    worker = atoi(argv[1]);    /* get command line argument */
    for (j = 0; j < worker; j++) {
        eval("worker",hello(j)); /* create live tuples in tuple space */
    }                          /* to execute function hello() */
    for (j = 0; j < worker; j++) {
        in("done");           /* get tuple from all workers at end */
    }
    printf("Parallel version of Hello world finished.\n");
}

hello(task)                   /* function executed by each worker */
int task;
{
    char hostname[64];

    gethostname(hostname,64); /* get name of workstation we're on */
    fprintf(stderr, "Hello world from node %s, task %d.\n", hostname,task);
    fflush(stderr);
    out("done");             /* put tuple indicating completion */
    return(0);
}
```

This parallel program requires a one command line argument specifying the number of worker processes to create. Every C-Linda program requires a *real_main()* function in place of the traditional *main()* function. The first loop in *real_main()* creates the specified number of processes by entered live tuples into tuple space using the *eval* operation. As each worker (i.e. available workstation) executes a process it executes the function *hello()*, which prints out the node name and the task assigned. Before finishing the function, each process places a "done" data tuple into tuple space, which is then gathered by the master process in the second loop of *real_main()*. While this program is rather simplistic, the interprocessor communication functions, *eval*, *in*, and *out*, are used in the same manner for complex programs.

C-Linda and Network C-Linda are not without shortcomings. They are not fault-tolerant in the sense that if a processor assigned to a computation fails, the entire execution will fail. Furthermore, Network C-Linda lacks any provision for restricting its operation to idle workstations.

An alternate and adaptive version of C-Linda included in the tool set is Piranha Linda [13]. Like Network C-Linda, Piranha Linda is an extension of C-Linda that is designed to operate on workstations connected by a network. However, Piranha Linda is an adaptive form of a master-worker parallelism provided by standard C-Linda and is much more flexible in that the number of processors participating in a parallel program can change during execution in response to changing user demand on network nodes. The basic idea behind Piranha Linda is to make use of the idle workstations on a network. If additional workstations become available during the parallel computation, they can join the computation and add to the processing power of the program. At the same time, if the load on an active node increases due to the execution of a local program, that node may leave the computation without disturbing the ongoing computations.

Piranha programs do not use the *eval* operation to invoke new processes. Instead, process execution is controlled by the Piranha system via three routines: *feeder*, *piranha*, and *retreat*. All Piranha programs must contain these three routines. Like C-Linda and Network C-Linda, the *in*, *out*, and *rd* operations are retained.

The *feeder* routine, which runs only on the master node, is responsible for creating, distributing, and collecting the results. The *piranha* routine runs on all the worker nodes and executes a task that reads in the data tuple, performs the task, and creates zero or more new tasks. When a processor becomes available, the *piranha* function is initiated on the newly available node by the master node. The *retreat* function is performed whenever a *piranha*

process must terminate in the middle of execution because the system on which it is running is required for other purposes. The *retreat* is only executed on the worker nodes.

Distributed Simulation Examples

Using the dual-mode FTDSP system, several parallel programs have been created to simulate the FT and non-FT modes of operation in a distributed manner. One program simulates the system in a non-FT mode of operation using two or more workstations while two others simulate the system in a FT mode of operation using several workstations with two different partitioning strategies of opposing degrees of granularity (i.e. computation-to-communication ratio).

In constructing the parallel program code, the specialist parallelism approach is employed. This method allows each workstation involved in the distributed simulation to function as an element of a pipeline by getting its input from the previous workstation and sending its output to the next workstation. In so doing, a message-passing communication interface is used requiring each workstation to explicitly request a data transfer.

Using the DSP96002 simulator software tools and the Linda programming language as the parallel paradigm, distributed multicomputer simulations can be performed. While the DSP96002 object functions can be retained from the sequential simulation [1], several additions and adaptations are necessary to allow the network of workstations to work together for distributed simulation. The following subsections present a brief overview of the issues that are involved in creating these distributed simulations running on a cluster of workstations. This discussion is divided into issues associated with the distributed simulation of the non-FT mode of the system and those with the FT mode using either of two different simulation partitioning approaches. Complete C-Linda and DSP96002 assembly-language source code is available from the authors.

Distributed Simulation of non-FT Mode

The non-FT linear array can be partitioned, where each partition will be simulated on a different workstation. Figure 5 illustrates this scenario for a linear array broken into partitions. While the figure only shows two partitions and workstations being used, in general there can be a number of workstations, each with the same or perhaps a different number of PEs (e.g. if the second workstation is more powerful than the first, we might choose to ask it to simulate a greater number of simulated PEs). The new interface occurs where the last PE of a partition must communicate with the first PE of the next partition. In general, from two to N partitions and workstations can be used where N equals the number of processing elements (PEs) in the simulated architecture, and each partition does not have to contain the same number of PEs.

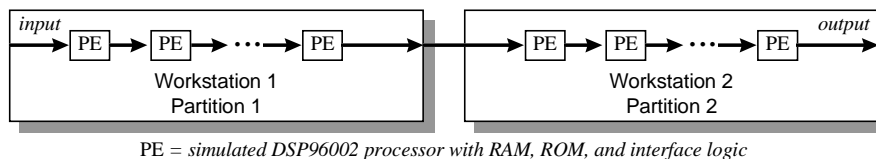


Figure 5. Distributed Simulation of the FTDSP Linear Array Architecture in non-FT Mode

Most of the interfaces developed and used with the sequential simulation of parallel DSP architectures remain the same. These include the system input-to-PE interface, the PE-to-PE interface, and system PE-to-output interface. The PE-to-PE interface is used to link the PEs within each partition. In a similar manner, the first PE of the first partition uses the system input-to-PE interface and the last PE of the last partition uses the system PE-to-output interface. The only new interface needed is the PE-to-PE interface between PEs to be simulated on different workstations in the distributed simulation.

Since each PE of the linear array must communicate on every simulated clock cycle to emulate the many pin and port connections in hardware, the new interface is broken down into two perspectives of a master/slave interface: the last PE of partition k (i.e. the master PE of the interface) and the first PE of partition $k+1$ (i.e. the slave PE of the interface). Then, using the Linda tuple access functions *in* and *out*, this interface is constructed to allow the PEs to communicate through tuple space (see Figure 6).

This interface consists of two concurrent sets of handshaking operations using tuple space. First, the workstation responsible for simulating the master PE of the inter-workstation, PE-to-PE interface (i.e. the last PE of partition k) gathers all the data and status bits involved in the simulated hardware link, including the control, address,

and data pins of the master PE. The workstation then stores values in temporary variables, assembles them into a tuple, and places the tuple into tuple space using an *out* operation. Since any number of workstations can be placing similar tuples into tuple space, an extra field is included in the tuple with the address of the destination workstation (i.e. the workstation responsible for simulating the interface's slave PE). When ready, the receiving workstation collects this tuple via an *in* operation and uses the values contained within the tuple to update the appropriate input pins associated with the slave PE. In the event the receiver attempts to read the tuple before the sender has placed it into tuple space, the receiver blocks until the tuple arrives.

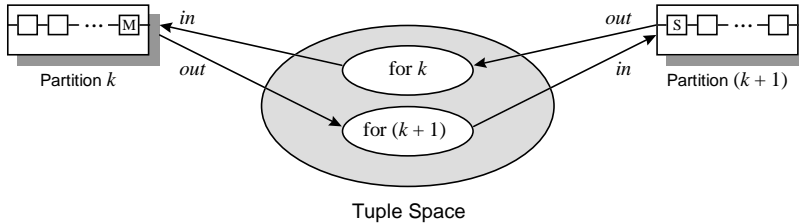


Figure 6. Tuple Space Access for Simulating non-FT Mode

Meanwhile, the workstation responsible for simulating the slave PE (i.e. the first PE of partition $k+1$) gathers all control, address, and data bits of the slave PE that the master PE needs for the simulated hardware link, stores them in temporary variables, constructs a tuple with an extra field indicating the address of the destination workstation (i.e. the workstation responsible for simulating the interface's master PE), and places the tuple into tuple space via an *out*. When ready, the receiving workstation reads and processes the tuple to update the appropriate input pins of the master PE. This process of using tuple space for handshaking and data transfer between workstations involved in a PE-to-PE interface continues for the duration of the simulation run, collecting and swapping simulated pin values all the while.

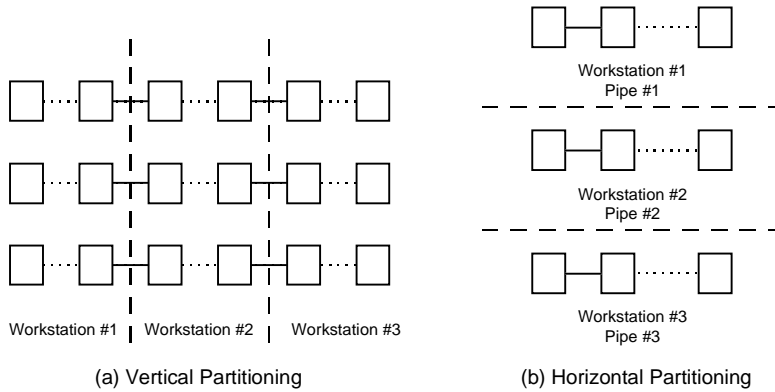


Figure 7. Two Partitioning Approaches for FT Mode (with three workstations)

Distributed Simulation of FT Mode

Two approaches are taken in segmenting the FT linear array by using either horizontal or vertical partitions (see Figure 7) and both support performance and fault injection simulations. The first approach uses vertical partitioning to split the pipes up into stages, requiring each workstation to perform in a lock-step fashion passing information after each simulated clock cycle much like the non-FT distributed simulation. The second approach uses horizontal partitioning and allows one or several workstations to simulate each single pipe in a virtually independent fashion, only exchanging information when necessary (e.g. for TMR voting purposes at the end of the pipes).

Vertical Partitioning

Like the method for distributed simulation of the non-FT linear array, the FT linear array simulation uses the same system input-to-PE and PE-to-output interfaces as the sequential simulation. The only new interface needed is the interface between the PEs on different workstations. However, since there are three pipes instead of just one single pipe, the problem is slightly different.

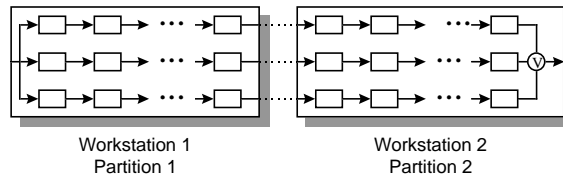


Figure 8. Vertical Partitioning of FT Mode

Figure 8 illustrates how the FT linear array is segmented into vertical partitions. While the figure only shows two workstations being used, in general there can be a number of workstations. By partitioning the configuration in this method, the problem can be approached in the same manner as the non-FT linear array. As illustrated in Figure 9, the key difference is that each tuple placed into the tuple space during the master/slave handshaking process discussed in the previous section now contains the data, address, and control bits for *three* simulated PE-to-PE hardware interfaces instead of just one. In addition, the final workstation is also responsible for simulating the hardware voting.

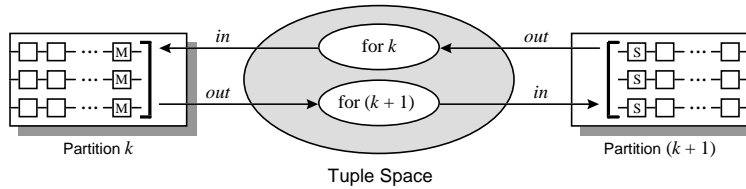


Figure 9. Tuple Space Access of FT Mode with Vertical Partitioning

Horizontal Partitioning

While the vertical approach for simulating the FT linear array consists of partitioning *across* each pipe, the horizontal method allows one or several workstations to simulate *each* pipe in its entirety. For example, when three workstations are used, each pipe is simulated by a single workstation as shown in Figure 10.

For this technique, the output status of the last PE in each pipe is placed into tuple space after each simulated clock cycle. One of the workstations (e.g. the most powerful), referred to as the *collector* workstation, then gathers this information to perform the simulation of hardware voting. At the same time, the collector workstation places a new system input value into tuple space that is then gathered by the workstations when ready for the simulation of the next clock cycle. Thus, the collector workstation is responsible for performing the voting, simulating the first pipe of the FT linear array, and for providing the input value to the other workstations.

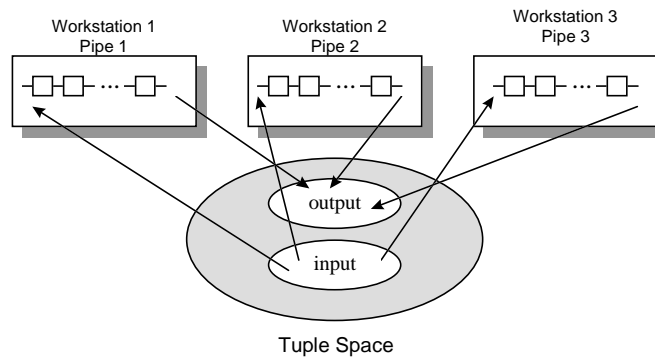


Figure 10. Tuple Space Access of FT Mode with Horizontal Partitioning

More workstations can be added to this simulation by allowing two or more workstations to simulate a single pipe, thus using a hybrid partitioning scheme. In this case, the workstations collaborating on the simulation of a particular pipe communicate for the master/slave, PE-to-PE handshaking as before. For example, six workstations can be used instead of three. Each pipe can be broken into two partitions, where one workstation is responsible for each partition. Two workstations are then responsible for a single pipe and communicate in the same manner as the non-FT linear array by using the tuple space. In this configuration, the collector workstation is still responsible for

gathering the status of the last PE in each pipe from tuple space and for providing an input value to the workstations simulating the first PE of each pipe.

As will be seen in the next section, this distributed simulation allows each workstation to simulate a pipe at its own pace and at the same time decreases the amount of interprocessor communication. However, it does have the disadvantage of placing the responsibility of performing the voting of the outputs and supplying the input values on the collector workstation.

Simulation Results

A number of distributed simulation runs were conducted with the dual-mode, fault-tolerant, linear array multicomputer architecture using various DSP applications (e.g. digital filtering, fast Fourier transforms, etc.) and tested to ensure that each software prototype was functioning correctly in terms of verification and validation. In collecting the various times required to simulate a certain number of clock cycles, the time required to simulate one clock cycle is relatively independent of the application being executed. For example, 10,000 simulated clock cycles using a FFT algorithm takes approximately the same amount of simulation time as 10,000 simulated clock cycles of a simple data propagation test. Therefore, after verification and validation were complete, a simple data propagation test was used for the performance tests to determine the amount of time involved in a simulation, but the results are representative of any program used.

The time required to perform DSP multicomputer simulations is highly dependent upon the number of PEs in the architecture. For example, the sequential simulation of 15,000 clock cycles of a 36-PE, non-FT linear array requires more time in comparison to 15,000 clock cycles of a 6-PE, non-FT linear array. Figure 11 illustrates this behavior by graphing the amount of time required for a sequential simulation as a function of the number of PEs involved in the simulation.

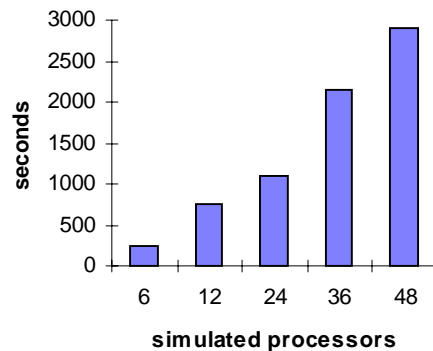


Figure 11. Sequential Simulation Time (on a SPARC-2/40 for 15,000 simulated clock cycles)

While this graph only displays the amount of time associated with 15,000 clock cycles, many other DSP applications require the simulation of more clock cycles. For example, when simulating a 512-point FFT running on a FTDSP architecture with nine PEs, the maximum number of clock cycles per output set is 37,514 [7]. For this application, several hundred-thousand clock cycles were simulated to get an idea of how it performs. The time required to simulate this number of clock cycles can approach several hours using a single SPARC-2/40 workstation.

The workstations involved in the distributed simulations consisted of five SPARC-2/40 workstations and one SPARC-10/40 workstation connected by a 10-Mbps Ethernet. Since the SPARC-10 workstation provides approximately twice the performance of a SPARC-2 workstation, care was taken to ensure that this increased performance level did not affect the results by only using the SPARC-10 workstation in situations where its speed had no impact. For example, many of the simulations involved each workstation simulating an equal number of PEs. In this scenario, the speedup of the SPARC-10 is not a factor since all the workstations must work together in a lock-step fashion by communicating after each simulated clock cycle. The slowest workstation used will set the pace of the distributed simulation. While this homogeneous approach was employed in order not to skew the performance measurements, for practical purposes we could take advantage of the more powerful workstation to perform extra duties (e.g. handle more PEs, be the collector workstation in FT mode simulation with horizontal partitioning, etc.).

The performance of the distributed simulations is measured in terms of the speedup factor [14], which for an n -processor system is defined as

$$S(n) = T(1) / T(n)$$

where $T(n)$ is the execution time for the n -processor system and $T(1)$ is the execution time for the uniprocessor system. Similarly, the system efficiency for an n -processor system is defined as

$$E(n) = S(n) / n = T(1) / nT(n)$$

For this situation, the speedup reflects the decrease in simulation time associated with each additional workstation used in the simulation. The ideal speedup factor for two workstations is two, with three workstations is three, etc. (i.e. an efficiency of 1.0).

The key factor affecting the speedup of the simulations is the communication overhead involved. Since the DSP96002 multicomputer simulations require the simulation of port and pin connections, each workstation must exchange information after every simulated clock cycle. Simulating this level of fidelity adds up to a large amount of time spent on communication overhead. For the Linda parallel programming environment, each tuple access (i.e. an *in* and *out* combination) takes about five milliseconds [2]. Thus, the network communication between workstations becomes an important factor. The horizontal partitioning approach of the FT mode simulation attempts to compensate for the communication overhead by allowing each workstation to simulate a single pipe or a portion of a pipe.

In the following sections, the speedup results from the distributed simulations were obtained by varying the number of workstations involved in the simulation between one and a maximum value of six. This maximum value was chosen for two reasons. First, the number of workstations available was limited to this number. Second, as the results will show, the use of more than six workstations yields no significant improvement in the simulation time for the cases studied.

Non-FT Mode Simulation Results

For the non-FT linear array, the number of PEs being simulated is initially set at six. Then this number is increased to twelve, and from here twelve PEs are added until a maximum of forty-eight PEs in the linear array is reached. In some DSP applications, it may not be practical to develop a linear array consisting of dozens of PEs. However, varying the number of PEs in this fashion illustrates the time savings obtainable from a distributed simulation. Figure 12 compares the speedup results as a function of the number of workstations used in the simulation.

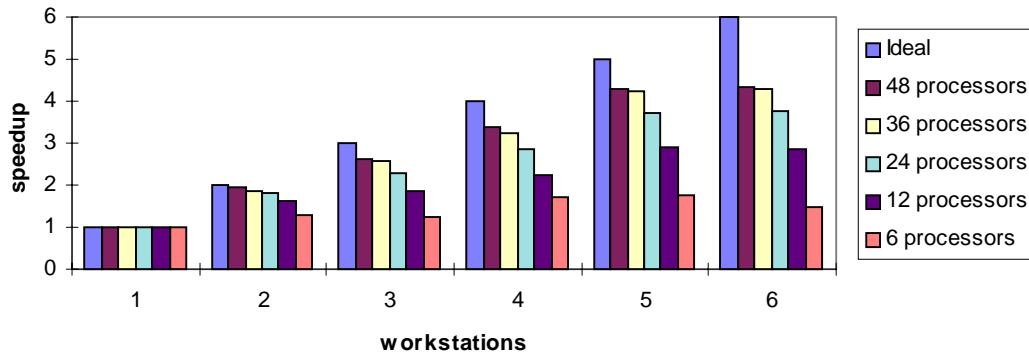


Figure 12. Distributed Simulation Results for the non-FT Mode

Since the FTDSP architecture requires a fine-grain distributed simulation, much of the time involved is communication between the workstations. Even though it is understood that fine-grained computations often contain an abundance of parallelism, they also require more communication overhead [14]. In this distributed simulation, the overhead is caused by the relatively high latencies associated with communication via TCP/IP over 10-Mbps Ethernet.

The effect of this overhead is lessened somewhat by increasing the number of PEs being simulated on a single workstation. By doing this, each workstation is performing more work between each exchange of information via the tuple space, thus increasing the granularity ratio of computation-to-communication and reducing the impact of the relatively large communication latencies. In general, the best results are obtained when each workstation is simulating three or more PEs.

With only six PEs in the linear array, the reduction in time of the simulation is limited and any increase in the number of workstations results in little improvement in the speedup. In particular, with six workstations simulating six PEs (i.e. one PE per workstation), the speedup obtained is insignificant. These results improve as the number of PEs in the architecture is increased up to a point where any additional PEs yield very little improvement.

For example, with six workstations the granularity ratio of the distributed simulation is no longer as effective since an insufficient degree of computation is required to compensate for the increased latencies imposed by the network communication. From this point, each additional workstation involved in the simulation produces no decrease in simulation time as compared to the peak values achieved with five workstations.

FT Mode Simulation Results

For the FT mode of the FTDSP architecture, the number of PEs being simulated is initially set at three PEs per pipe, or nine PEs in total. Three PEs per pipe is chosen because it simplifies the partitioning task when using vertical partitioning. After nine PEs, the number is increased to four PEs per pipe or a total of twelve, and from here twelve are added to the total number each time until a maximum of forty-eight PEs or sixteen PEs per pipe is reached.

With vertical partitioning of a simulated architecture with nine PEs (i.e. three PEs per pipe), the maximum number of workstations that can be employed is three because this method of partitioning allows a workstation to simulate no fewer than three PEs. In the situation where twelve and twenty-four PEs are simulated, the maximum number of workstations employed is four since it is impossible to use more than this number and still have each workstation simulate at least three PEs.

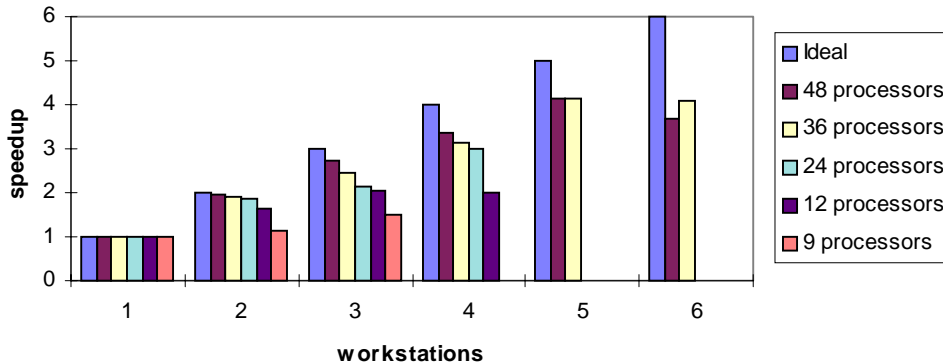


Figure 13. Distributed Simulation Results for the FT Mode with Vertical Partitioning

As with the non-FT linear array, there is very little speedup obtained when few PEs are being simulated per workstation (see Figure 13). For example, with nine PEs and three workstations performing the simulation, the speedup obtained is insignificant. These results improve as the number of PEs in the FT linear array is increased up to a point where any additional workstations yield very little improvement. Once again, the bandwidth of the network becomes a factor when the communication overhead is at a maximum (i.e. six workstations) resulting in a larger variance from the ideal speedup. In general, the results indicate that the more PEs being simulated on a workstation, the better the speedup results.

As with the non-FT mode of the architecture, when partitioned vertically the FT mode requires a fine-grained simulation. This type of simulation results in a large amount of communication overhead and the use of any more than six workstations results in an insignificant reduction in simulation time. To avoid this communication overhead, horizontal partitioning attempts to partition the problem in such a way as to create a medium-grained simulation.

With horizontal partitioning, the number of workstations participating in the simulation is first set at three and later six. Figure 14 displays the speedup results of these simulations. The results are an improvement over vertical partitioning when three workstations are used since the communication overhead is greatly reduced. However, when six workstations are used, the results are almost identical to vertical partitioning. One advantage of using three workstations with this approach is that each workstation can function independently of the others. The benefit of reduced interprocessor communication is improved speedup.

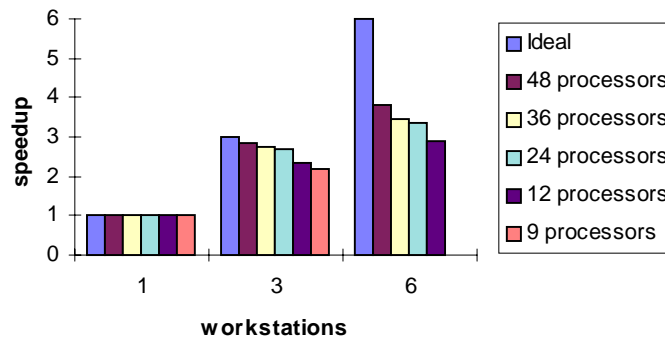


Figure 14. Distributed Simulation Results for the FT Mode with Horizontal Partitioning

With horizontal partitioning and three workstations, the simulation is transformed into a medium-grained computation with a reduced communication overhead. While the exchange of information is still required between the collector workstation and the other workstations to gather the status of the last PE in each pipe and to pass the input value to the first PE in each pipe, it still achieves a reduction in the amount of communication overhead as compared to the other methods.

When six workstations are used with horizontal partitioning (i.e. vertical partitioning within horizontal partitions), the results are almost identical to vertical partitioning due to the increase in the amount of communication overhead. While six workstations still offer a reduction in simulation time, they also cause the largest variance from the ideal speedup.

Conclusions

While functional simulation tools such as processor libraries provide researchers and designers with the benefit of testing prototype uniprocessor and multiprocessor systems without incurring the increasingly high costs associated with hardware development, the simulation times can quickly become impractical. However, by taking advantage of widely available computing resources in the form of networked workstations with parallel programming and coordination languages like Linda, these functional simulation tools and others can be partitioned and mapped onto workstation clusters. Depending upon such factors as the architecture being simulated, the computing resources available, the parallel computing tools employed, and the algorithms developed for partitioning the problem, the results of these distributed simulations can be extremely valuable. In the best cases, speedups approaching a linear increase with respect to the number of workstations used can be achieved. And, since many of the software tools are available at low or no cost and most government, industrial, and academic institutions have networked workstations with some idle time, these simulation speedups can be attained without any additional computing investments.

This paper has presented one method with several variations by which the distributed simulation of parallel computer systems, for DSP or other applications, can be performed on workstation clusters to significantly reduce simulation time. The test configuration consists of a signal processing multicomputer architecture with fault-tolerant and non-fault-tolerant modes of operation. First, the Linda parallel programming environment was adapted for use in distributed simulations. By using the C language extensions for interprocessor communication, a method is created that allows the simulated PEs to communicate via the tuple space. The results indicate that a parallel programming language such as Linda can indeed be readily adapted for use in distributed simulations using the processor library approach to functional simulation. Next, several variants of distributed simulation were performed and compared to their equivalent sequential simulation. While significant speedup was obtained in many cases, the results are most favorable when each workstation is responsible for simulating several PEs. Distributed simulations are shown to reduce the simulation time required for multicomputer configurations if each workstation is provided with enough work (e.g. three or more PEs per workstation) to compensate for network overhead.

A limiting factor in all of these distributed simulations is the network latency and throughput. While it is true that an increase in the number of workstations results in a decrease in the simulation time, it also results in a reduction in system efficiency (i.e. an increase in the span between ideal and realized speedup). One area of future research is to investigate the benefits of using a state-of-the-art network. Presently, the use of more than six workstations in the simulation does not result in a significant improvement in the speedup due to the relatively low throughput and high latency of TCP/IP over 10-Mbps Ethernet. The results of the simulations performed in this

study could be significantly improved by making use of a low-latency, high-bandwidth interconnect such as ANSI/IEEE Standard 1596-1992, the Scalable Coherent Interface. With the potential for sub-microsecond latency and 8-Gbps throughput between nodes, this advanced interconnect could make it feasible to employ fine-grain parallelism with many more than six workstations, perhaps as many as one workstation for each simulated PE.

References

1. George, A., "Simulating Microprocessor-based Parallel Computers using Processor Libraries," *Simulation*, Vol. 60, No. 1, pp. 37-42, February 1993.
2. Anderson, T., D. Culler, and D. Patterson, "A Case for NOW (Networks of Workstations)," *IEEE Micro*, Vol. 15, No. 1, pp. 54-64, February 1995.
3. Kloker, K., B. Lindsley, S. Liberman, P. Marino, E. Rushinek, and G.D. Hillman, "The Motorola DSP96002 IEEE Floating-Point Digital Signal Processor," *Proceedings of the IEEE International Conference on ASSP*, May 1989.
4. *DSP96002 IEEE Floating-Point Dual-Port Processor User's Manual*, Publication No. DSP96002UM/AD, Motorola Inc., Austin, Texas, 1989.
5. *Motorola DSP96002 Digital Signal Processor Simulator Reference Manual*, Motorola Inc., Austin, Texas, 1989.
6. Johnson, B., *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.
7. George, A. and L. Hawkes, *Microprocessor-Based Parallel Architecture for Reliable DSP Systems*, CRC Press Inc., Boca Raton, FL, 1992.
8. Ahuja, S., N. Carriero, and D. Gelernter, "Linda and Friends," *IEEE Computer*, Vol. 19, No. 8, pp. 26-34, August 1986.
9. Gelernter, D., "Domesticating Parallelism," *IEEE Computer*, Vol. 19, No. 8, pp. 12-19, August 1986.
10. Carriero, N. and D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed," *ACM Computing Surveys*, pp. 323-357, September 1989.
11. Carriero, N. and D. Gelernter, *How to Write Parallel Programs*, The MIT Press, Cambridge, Massachusetts, 1992.
12. *C-Linda User's Guide and Reference Manual Version 2.5.2*, Scientific Computing Associates, 1993.
13. Carriero, N., E. Freeman, D. Gelernter, and D. Kaminsky, "Adaptive Parallelism and Piranha," *IEEE Computer*, Vol. 28, No. 1, pp. 40-49, January 1995.
14. Hwang, K., *Advanced Computer Architecture*, McGraw-Hill, New York, NY, 1993.