

# DPS: A Distributed Program Simulator and Performance Measurement Tool

Michael J. Lewis                      C. Mic Bowman

The Pennsylvania State University  
The Graduate School  
The Department of Computer Science \*

## 1 Introduction

The recent past has seen an intense effort to harness the power of networked workstations as an alternative to expensive supercomputers. However, the efficacy of parallel programming in a distributed system is limited by several factors. These include (1) the nature of the problem being attacked, (2) the design of the application program, (3) the speeds of the various components of the network, and (4) the communication between nodes in the system. A program that performs well in one particular environment may be extremely inefficient in another.

The design of an efficient parallel program generally relies on detailed knowledge about the network hardware on which the program will run. In the design phase of program development, issues such as the programming paradigm, the granularity of parallelism, and the cost of intertask communication must be addressed. Programmers must predict how a particular design will perform in practice, but complex program behavior makes it difficult to determine how characteristics of a particular network will affect each issue. Therefore, even careful consideration for hardware is insufficient to guarantee an efficient implementation.

DPS (Distributed Program Simulator) helps programmers build efficient parallel programs by alleviating the difficulty of matching applications to hardware. The user provides DPS with a program *blueprint* that describes the number of tasks, the location of tasks, and the characteristics of intertask communication within a parallel program. The method for specifying blueprints is simple and general. Blueprints for many different classes of parallel programs can be described without the complexity of a full implementation. DPS uses the information in the blueprint to run a real-time simulation of the program on the network for which it is intended. Upon completion of the simulation, DPS presents the user with a detailed *performance profile*. The profile can be used to identify bottlenecks and tune the algorithm. In this way, the programmer can develop a program that best matches the network hardware.

To simplify blueprint construction, DPS introduces a model that represents an application by its *shape*, *terrain*, and *texture*. These properties characterize the program based on (1) the decomposition of the program into *functional units*, (2) the interaction between functional units, (3) the grouping of functional units into processes, and (4) the mapping of processes onto machines in

---

\*Submitted in partial fulfillment of the requirements for the degree of Master of Science.

the network. All parallel programmers must address these issues; DPS simply mandates that it be done explicitly.

A key to the model is that the blueprint describes the relationship between events that generate messages. That is, the DPS user specifies when, where, and by whom the messages are sent, but not when or how they arrive. This hides the inherent complexity of predicting message arrival time in a real network. DPS simulates the specified communication by sending messages across the actual network hardware. Thus, it accurately reflects the influence of network hardware on the pattern of communication. Even though all communication must be explicitly specified in the blueprint, DPS saves the user from the difficult task of implementing the communication infrastructure before knowing if it is appropriate. With DPS, experimenting with different communication schemes is as easy as altering the blueprint file; this is simpler than reCOORDINATING the various processes in the system.

An inherent weakness in this approach is that the blueprint specifies a deterministic sequence of events associated with a particular execution of the program. Thus, if the execution of a parallel program depends on the order of message arrival, the programmer must construct one blueprint for each possible ordering of messages. This scheme restricts DPS's usefulness for only a small subset of parallel programs. DPS is effective for the majority that limit non-determinism, since there often exists a small number of important message orderings that the user can accurately identify and specify. DPS is not useful for simulating algorithms for which the programmer is unable to specify the important orderings. This situation could be remedied by dynamically altering the execution path of the simulation at runtime. But this would add complexity to DPS without a significant contribution to its functionality. DPS is a prototyping system whose effectiveness is due in large part to its simplicity. It is not intended to be a fully general parallel programming language. An attempt to turn it into one would compromise its simplicity and impair its utility.

The remainder of this paper is organized as follows. Chapter 2 introduces a model that encourages programmers to think about their applications in terms of shape, terrain, and texture. Fitting a program to this model leads directly to the specification of a blueprint file, whose format and function are described in Chapter 3. Performance profiles are described in Chapter 4, which also compares example simulations with the results of full implementations. Chapter 5 discusses DPS's implementation, including major design decisions and alternative designs, and Chapter 6 presents conclusions and ideas for improvements and future work. Throughout the paper, two classic parallel programming paradigms, "course grain data-parallel" and "bag-of-tasks", are used as illustrative examples to facilitate the explanation of DPS's functionality. When specifics are necessary in the discussion, the paper refers to a matrix multiplication application. Pseudocode for the two paradigms and for two matrix multiplication programs (`MatrixMultiply1` and `MatrixMultiply2`) that utilize the paradigms are presented in Appendix A, which also includes complete blueprints of both programs.

## 2 Modelling Parallel Programs

DPS's usefulness is due in large part to the simplicity with which programmers can specify their programs. This simplicity is a result of the way DPS suggests that programmers think about their applications. This chapter introduces a parallel programming model that decomposes a program into functional units and characterizes the program based on the interrelationship between the units (shape), the details of the computation and communication performed by each unit (texture), and

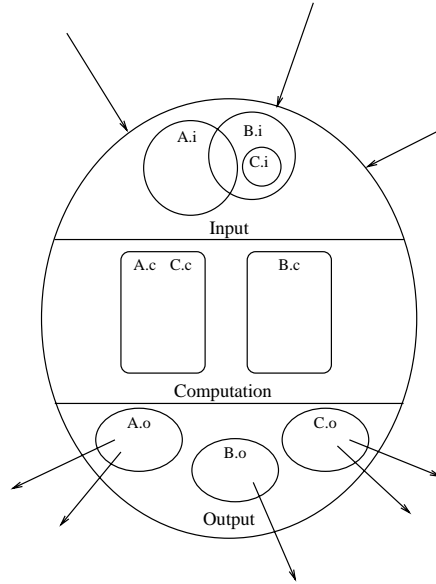


Figure 1: Functional Unit

the mapping of the units onto processors in the network (terrain).

## 2.1 Functional Units

Functional units are the building blocks of parallel programs. They receive messages, perform local computation, and generate messages for other functional units. Associated with each unit is a list of its *functionalities*, each consisting of three *functionality components*: input, computation, and output. Input components categorize incoming messages according to message *collections*. A collection is defined as a list of pairs containing message tags and their associated counts. A collection of  $i$  pairs,  $\langle mtag_i, count_i \rangle$ , is said to become *complete* when for all  $i$ ,  $count_i$  messages of type  $mtag_i$  have been received. When a collection is complete, it triggers the functional unit to operate according to the associated functionality. When triggered, the functional unit performs local processing according to the computation component, and generates messages as prescribed by the output component.

Figure 1 depicts a functional unit that consists of three functionalities, A, B, and C. All three contain specifications for each functionality component. Circles A.i, B.i, and C.i in the input portion of the functional unit represent collections, and intersect to indicate that a single message can contribute to the completion of several collections. Likewise, functionalities can share input, output, or computation components (e.g. A and C share the same computation component in Figure 1). Neither the interaction between functional units nor the details of their execution is shown in Figure 1; these aspects are described in the next two sections.

## 2.2 Shape

The shape of a program describes the flow of execution and the interaction between functional units. Shape can be represented by a directed graph where vertices represent functional units and

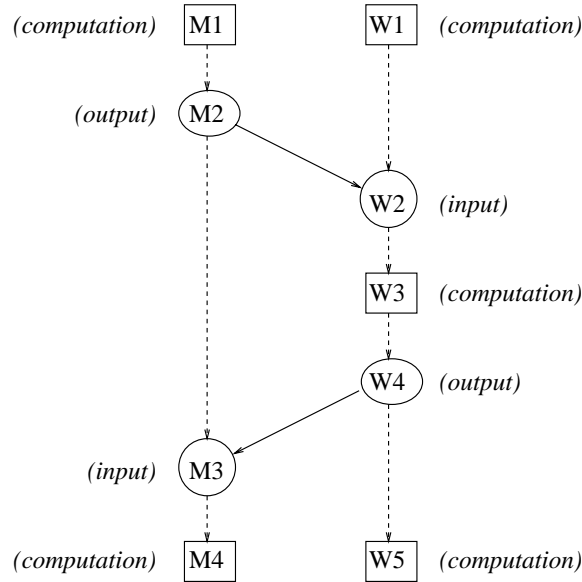


Figure 2: Shape of Course Grain Data-Parallel Paradigm

edges represent execution and communication dependencies. The execution of the program operates according to the characteristics of the vertices and continues along the edges of the graph. Input components finish operating and give way to computation components only when a message collection becomes complete. Thus, they are dependent upon the output components of other functional units. Computation components are directly dependent upon the completion of input components, and output components are dependent upon the completion of computation components. But empty functionality components allow edges between any pair of component types.

Figures 2 and 3 exhibit the shapes of the the “course grain data-parallel” and “bag-of-tasks” paradigms, respectively. Vertices W1-W5 represent functionality components of an instance of a Worker process, and M1-M5 represent those of the Master. Each of these components performs input, computation, or output, as indicated. Hashed arrows represent intraprocess control dependencies. A hashed arrow from vertex X1 to vertex X2 indicates that the execution of the component represented by X2 takes place immediately following that of the component represented by X1. A single vertex can be the source of several hashed arrows, one for each message collection that can be recognized by the functionality’s input component. The collection that becomes complete determines the arrow along which the flow of execution proceeds. Solid arrows represent interprocess control dependencies. A solid arrow leaves an output component vertex and enters an input component vertex, just as the messages it symbolizes travel from an output component of one process to an input component of another.

### 2.3 Texture

The texture of a program refers to the detailed properties of its execution. Associated with each computation and output component is a list of parameters describing the operation of that component. The rough outline of an algorithm can have shape, but a program takes on texture only when the specifics of its implementation are being considered. The characteristics of CPU utilization,

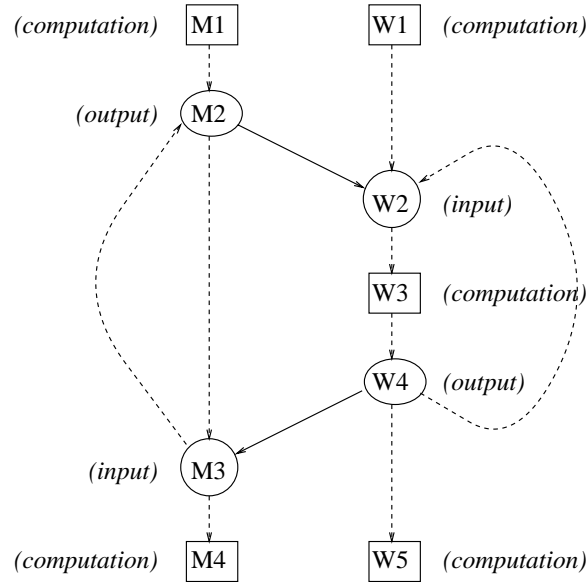


Figure 3: Shape of Bag-of-Tasks Paradigm

the number and size of messages, the packing and unpacking of data, and the transport protocol used for message delivery all contribute to a program’s texture.

Since texture is determined by a particular application, it is absent from the code outlines of Figures 15 and 16. Even the pseudocode of `MatrixMultiply1` and `MatrixMultiply2` does not present sufficient detail to completely specify the programs’ texture. The texture of the matrix multiplication applications details the size of the matrices sent between processes and the time needed to multiply vectors and write the resultant matrix. These details are described in Chapter 3’s description of a blueprint’s Computation and Output Specification Sections.

## 2.4 Terrain

The terrain of a program describes how functional units are mapped to a hardware environment. The selection of terrain can have a profound effect on program performance. For instance, a distributed system might consist of a group of workstations and a single supercomputer. An application containing a particularly compute-intensive portion would do well to include the supercomputer in the computation, but one with a fine grain decomposition might cause communication to be the factor most crucial for efficiency, thereby making the computing power of a supercomputer superfluous. Asymmetric network topologies, dissimilar network interface boards, and different processor speeds are subtle factors that can cause communication performance to vary significantly across different machines. Therefore, each of these factors must be taken into account in order to accurately predict performance.

For the matrix multiplication examples, the most natural choice of terrain maps each process to its own processor. However, with a fine grain decomposition and limited hardware resources, this might be impossible; several processors may have to support two or more processes. Figure 4 depicts an example that maps eight processes, P0-P7, onto six hosts, A-F; two processes are mapped to Hosts B and C and one to each of the others.

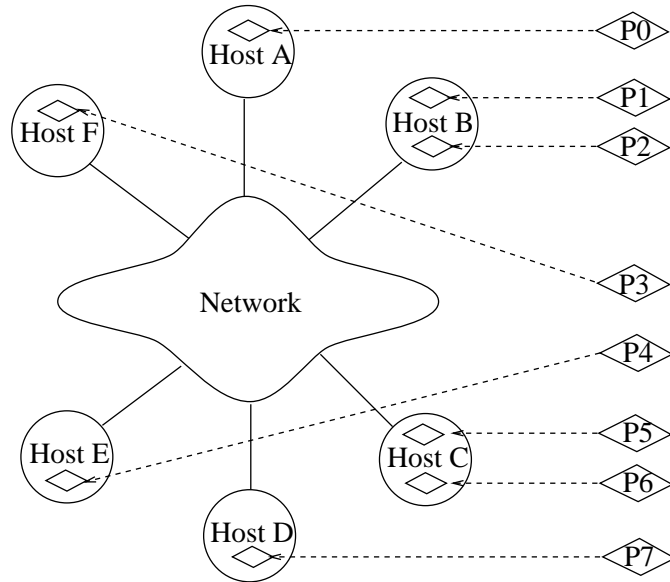


Figure 4: Example Terrain

### 3 User Interface

The user interface to DPS is a blueprint file that defines the characteristics of a program in terms of the model presented in Chapter 2. The Input, Computation, and Output Specification Sections define functionality components that are grouped together to form functionalities. These functionalities combine to form functional units which, in turn, make up the processes of the program. The Terrain Specification Section maps these processes onto hosts in the network to complete the program specification. The chapter refers to blueprints for `MatrixMultiply1` and `MatrixMultiply2` (Figures 19 and 20) for specific examples.

#### 3.1 Input Specification Section

The Input Specification Section consists of *input specification lines* that are marked with an ‘I’ as the first character of the line. The format of an input specification line is shown below:

```
I <itag> [ <count> <mtag>]*
```

An input specification line defines an input functionality component. An *itag* (input tag) is a character string that names the collection specified in the list of  $\langle count, mtag \rangle$  pairs that follows. A list containing  $i$  such pairs,  $\langle count_i, mtag_i \rangle$ , defines a collection that becomes complete only upon receipt of  $count_i$  messages of type  $mtag_i$  for each  $i$ .

The blueprint for `MatrixMultiply1` defines three simple message collections—one consisting of two `Mat.m` messages, one consisting of a single `RsltMat.m` message, and one, named `DoneM.i`, that consists of four `RsltMat.m` messages. The collections are selected to define important message groups in the algorithm. Since `MatrixMultiply2` returns the product matrix one entry per message rather than in four “addend” matrices, many more messages must be received before the Master possesses the entire resultant matrix; this is reflected in the `DoneM.i` itag of `MatrixMultiply2`’s blueprint.

```

#define VECTORSIZE 10

void Multiplyvectors()
{
    int A[VECTORSIZE],B[VECTORSIZE];
    int value=0,i;
    for (i=0;i<VECTORSIZE;i++) {
        value += A[i]*B[i];
    }
}

static struct tt_struct tt[] = {
    {"multiply_vectors",Multiplyvectors},
};

```

Figure 5: Example userfile.c

### 3.2 Computation Specification Section

The Computation Specification Section consists of *computation specification lines* that are marked with a ‘C’ as the first character of the line. The format of a computation specification line is shown below:

```
C <ctag> [sleep|busywait <time>] |[call <function>] |[exec <program>]
```

A computation specification line defines part of a computation component of a functionality. A *ctag* (computation tag) is a character string that names the computation characteristics specified in the arguments that follow.

Computation simulation is carried out in one of four ways. The simplest two allow the user to tell DPS to **sleep** or **busywait** for a specified number of seconds. The simplicity of these methods comes at the expense of accuracy, since the programmer is required to estimate the duration of processing sections. To increase the accuracy of the simulation, the user may use **exec** to specify a program to be executed, or **call** to specify a user-defined function to be called. This is particularly useful when specific portions of the program have been implemented. For subprogram execution, the user supplies the name of an executable file. For function calls, he compiles into DPS the code for the function and a translation table mapping character strings to function pointers. DPS currently allows only functions that take no arguments and return “void”. This functionality will be augmented in future versions of DPS.

All four options are present in the two example blueprints. Figure 5 shows a user file that contains vector multiplication code that is called by Worker processes of MatrixMultiply2 when they execute the `Mult.c` computation functionality. Both blueprints contain computation specification lines that use **exec** to cause DPS to execute preexisting programs named `read_mats` and `print_matrix`. Finally, `Add.c` and `Assign.c` are examples of computation components that cause DPS to simulate by **busywaiting** and **sleeping**, respectively.

Option		Arguments	
Description	Flag	Values	Default
Message Tag	-m	<mtag>	0
Message size	-s	<numbytes>	0
Number of messages	-n	<nummessages>	0
Packing	-p	on off	on
XDR encoding	-e	on off	on
Routing style	-X	Pvmd TCP	Pvmd

Figure 6: Output Specification

### 3.3 Output Specification Section

The Output Specification Section consists of *output specification lines* that are marked with an ‘O’ as the first character of the line. The format of an output specification line is shown below:

```
O <otag> <rcv_ptag> [ -flag <arguments>]*
```

An output specification line defines an output component of a functional unit. An *otag* (output tag) is a character string that names the communication specified in the arguments that follow. A *rcv\_ptag* is a string that refers to one or more processes that are named and defined in the Process Specification Section. Messages generated by the output component named *otag* are sent to all processes whose *ptag* contains a prefix that matches *rcv\_ptag*; this is due to the way DPS names processes and is discussed in more detail in Section 3.6. Specifying the *rcv\_ptag* as **ACK** causes the output messages to be sent to the process whose message caused the collection to become complete. A table of the flags and arguments of an output specification line is presented in Figure 6. Nummessages of size numbytes are sent to all processes that match the *rcv\_ptag* as described above. Each message is identified by the tag *mtag* and is packed and XDR encoded according to the **-p** and **-e** flags. Messages are sent using either the TCP or UDP transport protocol, depending on the routing style defined by the **-X** flag.

The *otags* defined in the blueprints of `MatrixMultiply1` and `MatrixMultiply2` reflect the number and size of the messages in each program. The “course grain data-parallel” version delivers entire matrices whose size is 400 bytes—(10 rows)\*(10 columns)\*(4 bytes per integer). The two original matrices are sent together but in separate messages, indicated by the **-n 2** part of the `SndMats.o` *otag*. The “bag-of-tasks” version returns the resultant matrix an entry at a time. It contains different output components for matrices, (row, column) pairs, and (row, column, value) triples. The options used to specify packing, encoding, and routing style are not shown in the example blueprints, thus the default settings are used.

### 3.4 Functionality Specification Section

The Functionality Specification Section consists of *functionality specification lines* that are marked with an ‘F’ as the first character of the line. The format of a functionality specification line is shown below:

```
F <ftag> <itag> <ctag>[,<ctag>]* <otag>[,<otag>]* <nextfutag>
```

An *ftag* is a character string that names the functionality specified in the arguments that follow. The *itag* and the list of *ctags* and *otags* identify the input, computation, and output components of the functionality. A *nextfutag* identifies the functional unit that is to begin operating once functionality *ftag* has finished computing and generating its output. Specifying the *nextfutag* as **SAME** causes the current functional unit to continue operating. Any of the four tag fields may contain the string **NULL**. For input, this indicates that the functional unit can begin operating according to functionality *ftag* without being triggered by the completion of a message collection. A **NULL** *ctag* list or *otag* list causes the corresponding computation component or output component, or both, to be skipped by functionality *ftag*. A **NULL** *nextfutag* indicates that the functional unit and the process to which it belongs have completed. In this way, the user specifies the conditions under which each process halts. If more than one collection becomes complete and triggers functionalities with conflicting *nextfutags*, only one of the tags is chosen. When conflicts occur, a **NULL** *nextfutag* takes precedence over any other, and a **SAME** *nextfutag* is always overridden. Conflicts between *nextfutags* that specify two different functional units are settled by selecting the tag that belongs to the functionality that appears first on the functional unit’s specification line, whose format is described in Section 3.5 below.

The Functionality Specification Sections of the two blueprints group the appropriate functionality components together to cause DPS to operate as specified in the pseudocode of Figures 17 and 18. For instance, the *ftag* **fw1.f** of `MatrixMultiply1` defines a functionality that receives matrices (**Mats.i**), performs an appropriate part of the multiplication (**Mult.c**), and sends a result back to the Master (**SndRslt.o**). The process halts after this functionality operates (*nextfutag* is **NULL**). Functionality **mf3.f** of `MatrixMultiply2` indicates that once the Master process receives the messages that cause **DoneM.i** to be satisfied, the resultant matrix can be printed according to **Print.o**. **NULL** values in the final two tag fields of **mf3.f** ensure that no subsequent output is generated, and that no next functional unit operates.

### 3.5 Functional Unit Specification Section

The Functional Unit Specification Section consists of *Functional unit specification lines* that are marked with a ‘U’ as the first character of the line. The format of a functional unit specification line is shown below:

```
U <futag> [ <ftag> ]+
```

An *futag* (functional unit tag) is a character string naming a functional unit that operates according to the functionalities identified by the *ftags* that follow. The functional units of the matrix multiplication applications contain between one and three functionalities.

### 3.6 Process Specification Section

The Process Specification Section consists of *process specification lines* that are marked with a ‘P’ as the first character of the line. The format of a process specification line is shown below:

```
P <ptag> <N> [ <futag>]*
```

A process specification line causes  $N$  processes, named `ptag[0] – ptag[N-1]`, to be built from instantiations of the functional units identified by the futags that follow. This allows processes to be cloned easily, and because users can cause messages to be sent to all processes that contain a specified `rcv_ptag` substring, it also makes for a simple method of defining process groups.

Both `MatrixMultiply1` and `MatrixMultiply2` contain a single Master process, named `Master.p[0]`, and four Workers, named `Worker.p[0]`, `Worker.p[1]`, `Worker.p[2]`, and `Worker.p[3]`. Each process in both paradigms is simple enough to be specified within a single functional unit. Multiple functional units are only necessary when the same message can have different meanings depending upon the state of the receiving process when that message arrives. To handle this situation, a different functional unit can be specified for each separate state within a process, and the operation associated with the receipt of the message can be different in each functionality. Since this is not a property of either of the matrix multiplication programs, a single functional unit for each process suffices.

### 3.7 Terrain Specification Section

The Terrain Specification Section consists of *terrain specification lines* that are marked with a ‘T’ as the first character of the line. The format of a terrain specification line is shown below:

```
T <ptag> <htag> [ <variable> <operator> <value>]*
```

Each process in the program is associated with its own terrain specification line. A `ptag` is a character string that names the process whose mapping to a host in the network is defined in the arguments that follow. An *htag* (host tag) is the logical host on which process `ptag` will run. Two processes whose terrain specification lines contain the same `htag` will be run on the same host; if the `htags` for two separate processes are different, then the processes will be executed on different hosts. The logical hosts are mapped onto real machines in the network based on the list of `(variable, operator, value)` triples that follow. Each triple forms a constraint that must be satisfied in the mapping. Such a mapping may not exist due to limited resources in the user’s network configuration or to a user’s error in the mapping specification. If this is the case, DPS will halt without carrying out the simulation and will return an appropriate error message to the user. The variables and values currently supported by DPS are shown in Figure 7. The value used to specify processor speed refers to the value held in the `hi_speed` field of the `hostinfo` structure in PVM. See [0] for details.

The blueprints for `MatrixMultiply1` and `MatrixMultiply2` demonstrate how the Terrain Specification Section can be used to map processes to hosts in a real network. For this example, the HEAT network at Sandia National Laboratories, California serves as the target network. HEAT (Heterogeneous Environment and Testbed) is a dedicated cluster computing platform of 50 workstations consisting of 10 of each of the following: IBM/RS6000’s, HP-9000 PA-RISC’s, SGI IRIS Indigos, DEC Alphas, and Sun SPARCstation 10’s. The workstations are currently networked via

variables	operators	values
NAME	=	Any legal hostname in the system
SPEED	=, <, >, <=, >=	Any positive integer
TYPE	=, !=	SGI, HP, SUN, IBM, DEC

Figure 7: Terrain Specification

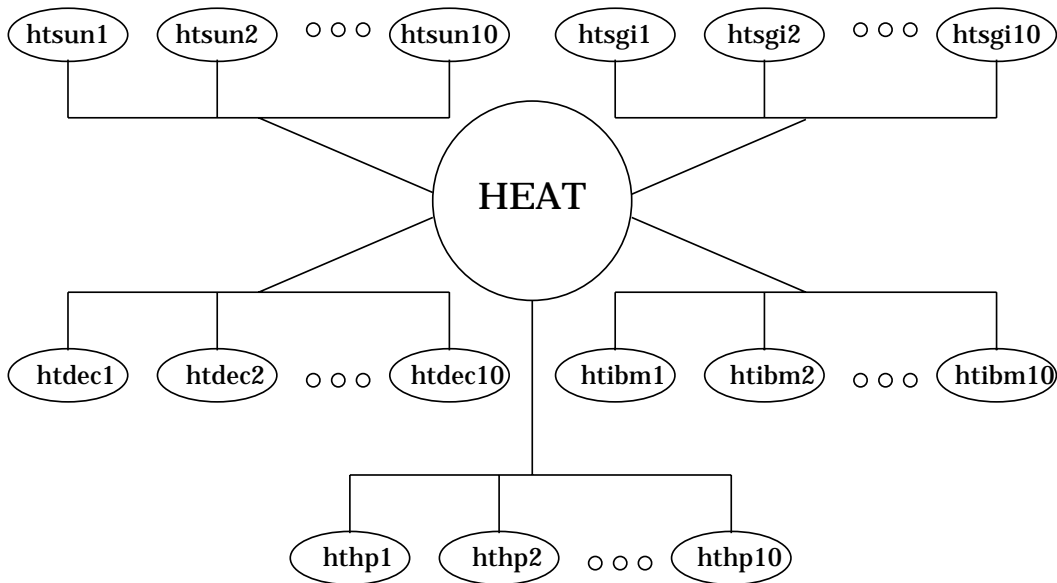


Figure 8: The HEAT Computing Cluster

a 100 Mbps DEC FDDI Gigaswitch and several DEC Concentrator 500's. The network is depicted in Figure 8.

The mapping presented in MatrixMultiply1's blueprint illustrates the full functionality of the Terrain Specification Section. The four distinct htags cause DPS to map the five processes onto four different machines. `Master.p[0]` will be run on an SGI whose `SPEED` variable is greater than or equal to 1000. `Worker.p[0]` will be executed on the machine named `htsun2`. Processes `Worker.p[1]` and `Worker.p[3]` will be run on the same host, an SGI named `htsgi3`. (Providing the machine name and architecture type may seem redundant, but this could be done to ensure that a particular machine is of a specified type.) Finally, DPS will map `Worker.p[2]` to a machine that is neither an SGI nor an IBM. The Terrain Specification Section of MatrixMultiply2's blueprint (Figure 20) presents a more realistic and useful example of mapping processes to machines. It ensures that all processes run on separate processors by assigning distinct htags to each process. It forces the Master process to execute on the machine named `htsgi1` and the Worker processes to execute on machines of each of the other four architecture types. The programmer could use this configuration and the resulting performance profile to identify the process that has completed the most work, thereby suggesting a better terrain.

## 4 Performance Profiles

DPS produces output in the form of performance profiles that contain information about the timing and communication characteristics of a program. Some profiles contain only the total execution time of the simulation. More sophisticated profiles expose the dynamics of an algorithm by reporting finer grain details about the simulation. Section 4.1 describes the format of a performance profile and the method by which users select the detail that it includes. Section 4.2 compares the results of a DPS simulation of MatrixMultiply2 to those of a real program that implements the algorithm.

### 4.1 Format

A performance profile is organized according to the structure of the program it describes, as represented in a blueprint. The profile includes information about incoming messages, outgoing messages, and the duration of functionality components. The format of a performance profile is depicted in Figure 9. A profile contains lines, labelled in Figure 9 by the letters 'P', 'U', and 'F', that represent three different granularity levels—process, functional unit, and functionality. By default, only those lines that describe the program at the process granularity level are included in the profile. Two macros, `PP_FU` and `PP_FUNC`, determine the information collected about functional units and functionalities. These macros are defined in the user configuration file "userfile.c". If `PP_FU` is defined, then DPS includes functional unit profile lines along with the process profile lines. If `PP_FUNC` is defined, then all three profile line types are included. Future versions of DPS will allow users to specify profile contents on a per tag basis. That is, the user will be able to select the components of the simulation in which he is most interested.

### 4.2 Experimental Verification

DPS's usefulness depends on its ability to accurately predict the performance of real programs. This section compares the results of DPS's simulation of MatrixMultiply2 with a corresponding

P)  $P$  Processes:  $ptag_0, \dots, ptag_p, \dots, ptag_P$   
P) .  
P) .  
P) Process  $p$ :  $ptag_p$   
P) CPU Time:  $user_p$  user,  $sys_p$  sys,  $real_p$  real.  
P) Received a total of  $bytes.in_p$  bytes in  $msgs.in_p$  messages  
P) Sent a total of  $bytes.out_p$  bytes in  $msgs.out_p$  messages  
U)  $FU_p$  Functional Units:  $futag_{p,0}, \dots, futag_{p,fu}, \dots, futag_{p,FU_p}$   
U) .  
U) .  
U) Functional Unit  $fu$ :  $futag_{p,fu}$   
U) Became active  $active_{p,fu}$  times.  
U) Received a total of  $bytes.in_{p,fu}$  bytes in  $msgs.in_{p,fu}$  messages  
U) Sent a total of  $bytes.out_{p,fu}$  bytes in  $msgs.out_{p,fu}$  messages  
U) CPU Time:  $user_{p,fu}$  user,  $sys_{p,fu}$  sys,  $real_{p,fu}$  real.  
F)  $F_{p,fu}$  Functionalities:  $ftag_{F_{p,fu},0}, ftag_{p,fu,f}, \dots, ftag_{p,fu,F_{p,fu}}$   
F) .  
F) .  
F) Functionality  $f$ :  $ftag_{p,fu,f}$   
F) Triggered  $triggered_{p,fu,f}$  times.  
F) CPU Time:  $user_{p,fu,f}$  user,  $sys_{p,fu,f}$  sys,  $real_{p,fu,f}$  real.  
F) .  
F) .  
U) .  
U) .  
P) .  
P) .

Figure 9: Performance Profile Format

	10X10		50X50		100X100	
Workers	DPS	MM2	DPS	MM2	DPS	MM2
1	0.56	0.61	13.53	12.60	63.64	57.22
2	0.32	0.38	7.87	7.29	59.17	55.74
3	0.31	0.37	8.70	8.21	69.47	64.86
4	0.36	0.42	10.46	9.58	49.26	41.86
5	0.54	0.61	13.53	12.44	53.55	48.03
6	0.65	0.71	22.28	20.17	64.72	58.72
7	0.72	0.78	17.26	15.95	75.60	67.68
8	0.77	0.81	23.75	21.73	88.39	79.88

Figure 10: Timing Results Table

real implementation, MM2, of the same algorithm.

For the tests whose results are presented here, the blueprints were similar to the one presented in Figure 20. However, all computation components were simulated using the `call` option; this isolated the cost of DPS’s performance measurements by eliminating the difficulty of predicting computation times. Each process was run on its own SGI machine, and the number of Workers and the problem size were varied, along with the blueprint lines that depended on those factors. Tests were repeated 100 times each. The table in Figure 10 reports the average wall clock time, in seconds, of the Master process. Figure 11 depicts the results graphically.

The results suggest that DPS simulates MatrixMultiply2 with a high degree of accuracy. Not only are the simulation times close to those of the corresponding real implementation, but important unpredictable dynamics of the program’s execution are also in evidence. For instance, the 100X100 version performs best with four Workers, while the 50X50 version’s performance is highest with only two. This could be due to any of several factors, but the important point is that DPS predicts the phenomenon.

Figure 10 represents the initial results in what will be an extensive performance evaluation of DPS. The problem size is relatively small, and, admittedly, the paradigm fits well with DPS’s model. While it is clear that further successful testing must precede the pronouncement of DPS as a success, the results do show that the current implementation at least exhibits the potential to accurately simulate real parallel programs.

## 5 Implementation

This section describes the interesting aspects of DPS’s implementation. DPS is composed of a *coordinator* process that spawns and manages a set of *simulator* processes. The coordinator manages but does not take part in the simulation. It parses and interprets the user’s blueprint, spawns the simulators, sends them information about how to operate, and waits for them to return performance results. After the arrival of results from all simulators, the coordinator compiles a performance profile in the format requested by the user. Figure 12 depicts the shape of DPS. Sections 5.1 and 5.2 detail the implementation of the coordinator and simulators, respectively.

DPS utilizes the Parallel Virtual Machine[0, 0] (PVM) message passing library for interprocess

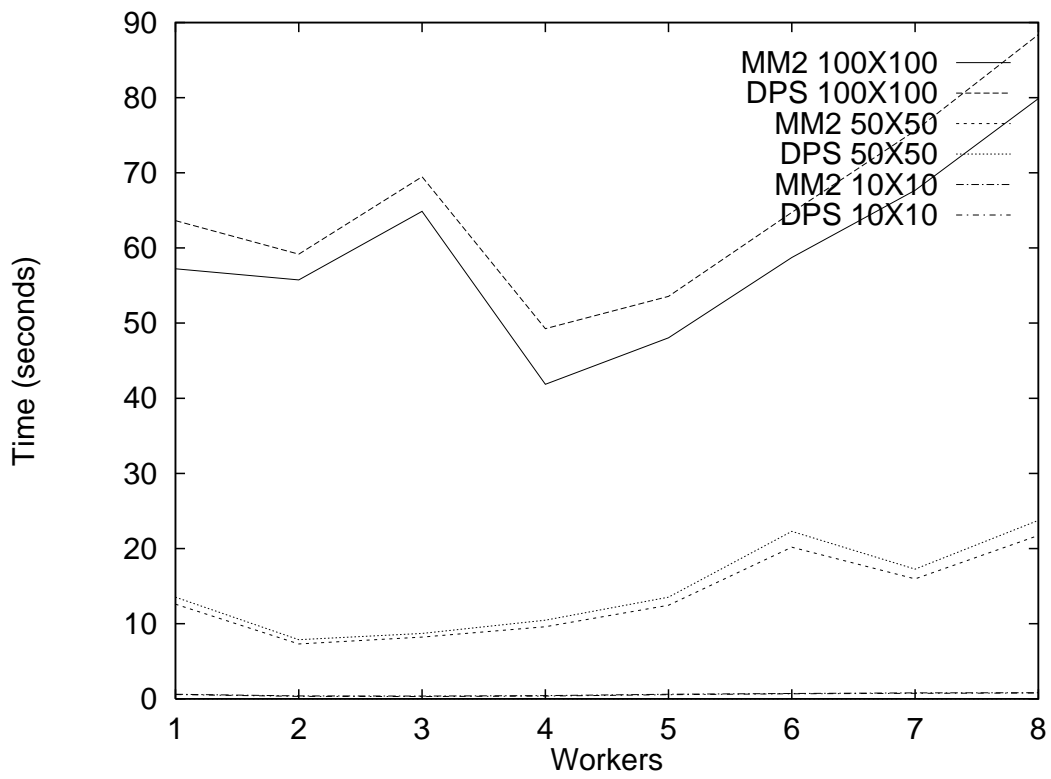


Figure 11: Timing Results Graph

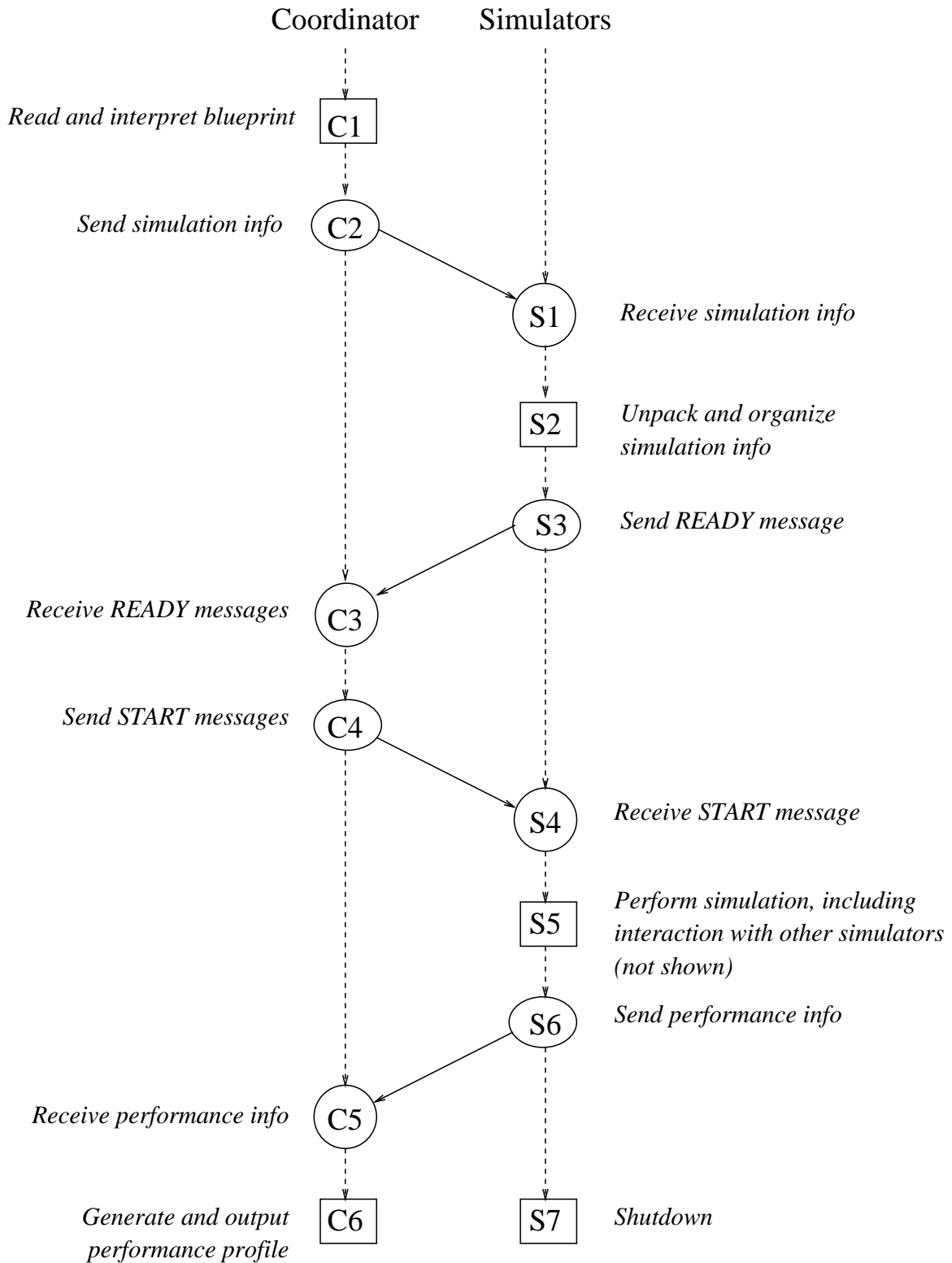


Figure 12: Shape of DPS

communication. PVM facilitates the design and implementation of applications that wish to exploit a heterogeneous set of networked machines. The PVM library, available to the C or Fortran application programmer, includes routines to spawn and kill processes, pack and unpack message buffers, send and receive messages, and synchronize processes through group communication. Users can configure their own pool of processors and may either assign machines to tasks or allow PVM to do so. Task to task communication is realized in one of two ways. By default, messages are sent through a local TCP socket to the PVM daemon process (*pvm*) on the node where the message originates. From there, they are forwarded to the remote *pvm* via a UDP network socket, and then passed on to the remote task through another local TCP socket. As an alternative, the user can cause the *pvm*s to be bypassed by having PVM set up direct TCP connections between tasks.

DPS is written in the C programming language, and has been compiled and tested on the following five architectures and operating systems: Sun SPARCStation 10's running SunOS 4.1.3, SGI Indigos running IRIX 4.0.5F, DEC Alphas running OSF1, IBM/RS6000's running AIX, and HP-9000 PA-RISC's running HP-UX.

## 5.1 Coordinator Process

The first job of the coordinator is to read, parse, and interpret the blueprint file. Since the content of each line is simple and well-defined, automatic lexicographical analyzers and parsers are unnecessary; the file is simply read a line at a time and parsed with C library calls according to the first character on the line. Separate singly linked lists of different structures are maintained for input components, computation components, output components, functionalities, functional units, processes, hosts, and messages. The structures that make up the lists contain pointers to entries in other lists to build the program specification from the components (e.g. an entry in the functional unit list points to a list of structures, each of which contains an *ftag* and a pointer to an entry in the functionality list). With this scheme, information is stored once and can be accessed easily across lists. While the blueprint is being parsed, only the tags that name each component, and the modifying arguments on computation, output, and terrain specification lines are entered into the structures. Once the entire blueprint has been read, the coordinator traverses the lists and fills in the cross-list pointers. This allows lines to appear in any order in the blueprint.

The coordinator is also in charge of creating a *message map*. In the blueprint, users distinguish message types with *mtag* character strings. Rather than using these tags to distinguish the messages in the implementation, the coordinator maps the strings to integer message identifiers (*mid*'s) between 0 and N, where N is the number of different tags. The identifiers tag the messages in the **type** parameter of `pvm_send()`. This scheme is more efficient than sending the strings themselves within the body of each message, since the **type** argument is immediately available through the `pvm_bufinfo()` routine upon receipt of a message, and need not be explicitly unpacked from the message body. Furthermore, the identifiers themselves can be used to index arrays. This is a useful feature since simulators must check for collection completion after the arrival of every message. Per-message overhead must be kept to a minimum so that performance is not significantly affected by management of the simulation.

The coordinator's next task is to map processes onto hosts in the network according to the lists of constraints in the terrain specification lines. To do so, the coordinator creates a boolean *assignment matrix* logically indexed by *htags* and *hostnames*. *Htags* are logical host names from the terrain specification lines. *Hostnames* are names of machines in the user's PVM configuration.

		Logical Hosts (htags)			
		A	B	C	D
Real Hosts (hostnames)	<i>htsgi1</i>	1	0	0	0
	<i>htsgi2</i>	1	0	0	0
	<i>htsgi3</i>	1	0	1	0
	<i>htibm1</i>	0	0	0	0
	<i>htibm2</i>	0	0	0	0
	<i>htibm3</i>	0	0	0	0
	<i>htsun1</i>	0	0	0	1
	<i>htsun2</i>	0	1	0	1
	<i>htsun3</i>	0	0	0	1

Figure 13: Assignment Matrix for MatrixMultiply1

The list of hostnames is obtained via a call to the `pvm_config()` routine. To fill the matrix, the coordinator traverses the list and identifies the hostnames that satisfy each constraint. Matrix entry  $M[i,j]$  becomes 1 if  $hostname_i$  satisfies all constraints on  $htag_j$ , and 0 otherwise. A valid mapping exists if and only if each  $htag_i$  can be assigned a unique hostname,  $hostname_j$ , such that  $M[i,j]$  is 1. The coordinator determines an assignment and fills the hostname field of process list entries. If no assignment exists, the user receives an error message. Figure 13 depicts the assignment matrix that results from the Terrain Specification Section of MatrixMultiply1. Entries are circled to indicate one possible mapping of htags to hostnames.

Next, the coordinator starts simulators on the appropriate hosts and sends each the information it needs. For every simulator, the coordinator traverses the input component, computation component, output component, functionality, and functional unit lists and marks those entries that are pertinent to that particular simulator. Marked entries are packed into a single message and sent to the simulator, which unpacks them and recreates corresponding lists with appropriate entries. The coordinator waits for the simulators to send a READY message indicating that they have unpacked the information and are ready to begin operating. Once a READY message from each simulator has been received, the coordinator multicasts a BEGIN message triggering the start of the simulation. The coordinator then waits to receive performance measurements from all simulators. When they arrive, a performance profile is generated as described in Section 5.3.

## 5.2 Simulator Processes

The first job of each simulator is to unpack the information sent by the coordinator and to recreate, in its own address space, subsets of the input component, computation component, output component, functionality, and functional unit lists. The simulator fills in cross-list pointers (just as the coordinator does after reading the blueprint), sends a READY message to the coordinator, and waits to receive a BEGIN message before starting the simulation. Every simulator maintains

a message table, indexed by message identifier, that contains a count of the number of messages of each type that have arrived at the process. Upon the arrival of each new message, this table is referenced to determine if a collection associated with the active functional unit has been completed by the incoming message.

Each simulator maintains a pointer to a the *active functional unit*—a member of the functional unit list. When the active functional unit pointer is set for the first time, or is changed to reflect the nextfutag after the completion of a functionality, a message table *snapshot* is taken. The snapshot saves the message count associated with each message identifier, and is used to determine when collections have become complete. When a message arrives, each collection  $j$  consisting of a list of  $i$  pairs,  $\langle mtag_i, count_i \rangle$ , is examined to determine the number of messages that have arrived since the last time collection  $j$  became complete. To do so, the simulator associates with each collection  $j$ , a variable,  $completed_j$ , that counts the number of times collection  $j$  has become complete since the current functional unit was loaded. The following condition determines if collection  $j$  has become complete:

$$\forall i (MT[mid_i] - SS[mid_i]) \leq ((completed_j + 1) * count_i)$$

MT is the message table, indexed by message identifier, whose entries contain counts of the number of messages of each type that have arrived at the process. SS is the snapshot of the message table that was taken when the current functional unit first became active. If the condition is satisfied, then  $completed_j$  is incremented and the associated computation and output components are simulated as described below. The only other time a functionality gets triggered is when it contains a NULL itag and becomes the active functional unit.

A computation component is defined by a list of ctags. The simulator traverses the list and operates according to the specification in each member. To **sleep**, the simulator sets an interval timer, calls the **pause()** system call, catches the SIGALRM signal generated by the timer, and then continues processing. This scheme, as opposed to the **sleep()** system call, allows simulation times to be estimated and specified in fractions of seconds. Simulators **busywait** in a similar manner—a spinlock is waited on by the simulator, and turned off by the SIGALRM interrupt handler. For the **call** option, the file **userfile.c** is compiled into DPS. It contains user-defined functions and a table that maps character strings to pointers to these functions. The simulator locates the string in the table, and calls the appropriate function. The **system()** system call is used to run subprograms for the **exec** option.

The simulation of output functionality components is carried out as follows. During the startup phase (before the simulation begins), a separate buffer is packed for each different size message that the process may have to send. This allows the user to turn off the packing option and still send arbitrary size messages. The simulator traverses the associated list of otags, setting the appropriate routing style with **pvm\_advise()**, and the appropriate packing type with **pvm\_initsend()**. It then packs (if packing is on) the appropriate number of messages and sends each to all processes in the list of tids associated with the output component.

When the simulation is complete, the simulators send performance results to the coordinator, which compiles them and returns a performance profile to the user. The method used to gather performance information is described in Section 5.3 below.

### 5.3 Generating Performance Profiles

Entry type	Associated information
PROCESS_BEGIN	ptag, time
PROCESS_END	ptag, time
FU_BEGIN	futag, time, snapshot
FUNC_BEGIN	ftag, time
FUNC_END	ftag, time

Figure 14: Execution History Log Entry Types

To generate performance profiles, each simulator builds an *execution history log*. The log collects state and timing information corresponding to events that occur during the simulation. Figure 14 lists the five types of log entries and the associated information logged with each. Every entry contains the wall clock time and the amount of system time and user time used by the process when the entry is logged. A `PROCESS_BEGIN` entry is logged when the simulation is begun, and a `PROCESS_END` entry is logged when it completes. An `FU_BEGIN` entry is added each time a new functional unit becomes active. In addition to the timing information, an `FU_BEGIN` log entry contains a pointer to the snapshot of the message table that is taken upon activation of the functional unit. `FUNC_BEGIN` and `FUNC_END` entries mark the triggering and completion of functionalities, respectively.

The execution history log contains sufficient information for the generation of a performance profile. The difference between the time logged in the `PROCESS_BEGIN` entry and the time logged in the `PROCESS_END` entry determines the duration of the process. Similarly, differences between logged times in `FUNC_BEGIN` and `FUNC_END` entries and between consecutive `FU_BEGIN` entries, determine the duration of functionalities and functional units, respectively. The information about incoming messages to each functional unit can be derived from differences between consecutive snapshots in the table. Information about outgoing messages can be determined by the count of the number of times each functionality is triggered. The remaining profile information is obtained by counting the various entries in the table. The interpretation of performance results can be time consuming, but since it is done after the simulation is complete, it does not add important overhead.

As mentioned in Section 4.1, user’s specify the granularity of the profile they receive by creating an appropriate macro definition in the file named “userfile.c”. This allows the implementation to avoid creating log entries that are unnecessary for generating a particular profile, by placing the code within `#ifdef` preprocessor statements. This method is better than simply allowing the user to specify a granularity level variable on the command line, because it avoids decisions about whether to log information. These decisions can add nontrivial overhead, especially if the time for which a functionality operates is short.

## 6 Conclusions

This paper describes DPS, a distributed program simulator and performance measurement tool. It introduces a model in which parallel programs are characterized by their decomposition into functional units and by the operation of and interaction between these units. The model is designed

to suggest the straightforward specification of a program in a blueprint—a file that DPS uses to run a real-time simulation of the program on the network hardware for which it is intended. Results returned to the user by DPS in performance profiles are used to tune the algorithm before its implementation.

DPS works—it provides a simple and general method of specifying parallel programs, and it simulates these programs accurately. The results presented in Chapter 4 show that DPS simulations come close to the performance of real implementations of the programs they simulate. DPS also makes an effective tool for measuring communication performance. [0] reports experiences with utilizing a predecessor of DPS to simulate simple communication intensive parallel “applications” designed solely to measure communication performance. By altering the message packing style, processor type, MTU size, transport protocol, and network traffic, the authors characterize the effect of each factor on throughput and latency.

Although effective and useful in its current form, there exist many areas for improvement of and future work on the DPS project. First of all, DPS has proven useful for relatively simple applications, but has yet to be used for a real, more complex, one. This will be the true test of DPS’s utility. The representation of shape and the high degree of modularity allowed in the specification of functional units suggests a graphical user interface that would make program specification even simpler. Users could draw a directed graph similar to those in Figures 2 and 3, and could click on the various components to define their texture. Other improvements would allow functional units to add semantics to messages and to pass parameters to and receive results from user-defined functions. This would give processes on separate machines more power to influence each other’s operation, thereby widening the range of programs that could be effectively simulated. Yet another improvement would allow the user to define and use variables within the blueprint file. Currently, scaling an application size can require the alteration of several blueprint lines. Using variables in lines that depend on such factors as the application size would simplify the alteration of the blueprint.

The model introduced for DPS need not be used simply to enable the specification of algorithms and programs that have already been conceived. Because it is the right way to think about parallelism, it is also useful for designing parallel algorithms and for implementing parallel programs. Many of the benefits of the object oriented programming paradigm are extended to parallel programming by this model. Functional unit specifications in a blueprint are similar to object oriented class definitions; the specifications define templates that are instantiated within specific processes for the simulation. Like objects in the object-oriented programming model, functional units export an interface defined by the set of messages they accept, and the messages determine the functionality with which the units operate. Thus, it is not difficult to envision the extension of DPS into a fully general object-oriented parallel programming language.

## A Matrix Multiplication Example

The appendix consists of six figures related to the two paradigms and the matrix multiplication programs referred to throughout the paper. Figures 15 and 16 present outlines of the “coarse grain data-parallel” and “bag-of-tasks” paradigms, Figures 17 and 18 present pseudocode for matrix multiplication programs that use the paradigms, and Figures 19 and 20 contain two example blueprints.

```

Master
Startup();
for i = 1 to numworkers
    Send(Worker[i],work[i]);
for i = 1 to numworkers
    Receive(&Worker,&results[i]);
Write(results);
Shutdown();

```

```

Worker
Startup();
Receive(Master,&work);
results = Do(work);
Send(Master,results);
Shutdown();

```

Figure 15: Pseudocode for Course Grain Data-Parallel Paradigm

```

Master
Startup();
for i = 1 to numworkers
    Send(Worker[i],work[i]);
j = 0;
for i = numworkers to numworkchunks
    Receive(&Worker,&results[j++]);
    Send(Worker,work[i]);
for i = 1 to numworkers
    Receive(&Worker,&results[j++]);
Write(results);
Shutdown();

```

```

Worker
Startup();
while (1)
    Receive(Master,&work);
    results = Do(work);
    Send(Master,results);
Shutdown();

```

Figure 16: Pseudocode for Bag-of-Tasks Paradigm

```

Master(A,B)
ZeroMatrix(SUM);
for i = 1 to numworkers
    Send(Worker[i],A,B);
for i = 1 to numworkers
    Receive(&Worker,&C);
    Addtomatrix(SUM,C);
WriteMatrix(C);

Worker(id)
ZeroMatrix(C);
Recieve(Master,&A,&B)
for i = 1 to A.size
    for j = 1 to B.size
        if ((j % id) == 0) C[i,j] = Multiplyvector(A[i,*],B[* ,j]);
Send(Master,C);

```

Figure 17: Pseudocode for MatrixMultiply1 (Course Grain Data-Parallel Version)

## References

- [1] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Labs, Oak Ridge, TN, May 1993.
- [2] Brian K. Grant and Anthony Skjellum. The PVM System: An In-Depth Analysis and Documenting Study – Concise Edition. Technical Report TR UCRL-JC-112016, Lawrence Livermore National Laboratories, Livermore, CA, 1992.
- [3] Michael J. Lewis and Raymond E. Cline Jr. PVM Communication Performance in Switched FDDI Heterogeneous Distributed Computing Environments. *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 13–19, October 1993.

```

Master(A,B)
for i = 1 to numworkers
    Send(Worker[i],A,B);
for i = 1 to A.size
    for j = 1 to B.size
        Receive(&Worker,&value,&row,&col);
        if (row>=0) C[row,col] = value;
        Send(Worker,i,j);
for i = 1 to numworkers
    Receive(&Worker,&value,&row,&col);
    if (row>=0) C[row,col] = value;
for i = 1 to numworkers
    Send(Worker[i],SHUTDOWN);
WriteMatrix(C);

```

```

Worker(id)
Receive(Master,&A,&B);
Send(Master,0,-1,-1);
while (continue)
    Receive(Master,&row,&col);
    value = Multiplyvector(A[row,*],B[*],col);
    Send(Master,value,row,col);

```

Figure 18: Pseudocode for MatrixMultiply2 (Bag-of-Tasks Version)

```

# Blueprint for MatrixMultiply1
#
# 10X10 integer matrices; 4 Workers
#
#
# Input Specification Section
#
# itag    count mtag
# -----
I Mats.i    2    Mat.m
I Rslt.i    1    RsltMat.m
I DoneM.i   4    RsltMat.m
#
# Computation Specification Section
#
# ctag    type    argument
# -----
C Mult.c    call    mult_routine
C Read.c    exec    read_mats
C Add.c     busywait 1.5
C Print.c   exec    print_matrix
#
# Output Specification Section
#
# otag    rcv_ptag argument list
# -----
O SndRslt.o Master.p -t RsltMat.m -n 1 -b 400
O SndMats.o Worker.p -t Mat.m    -n 2 -b 400
#
# Functionality Specification Section
#
# ftag    itag    ctag    otaglist    nextfutag
# -----
F fw1.f Mats.i    Mult.c    SndRslt.o NULL
F fm1.f NULL      Read.c    SndMats.o SAME
F fm2.f Rslt.i    Add.c     NULL      SAME
F fm3.f DoneM.i   Print.c   NULL      NULL
#
# Functional Unit Specification Section
#
# futag    ftags
# -----
U worker.fu fw1.f
U master.fu fm1.f fm2.f fm3.f
#
# Process Specification Section
#
# ptag    N    futaglist
# -----
P Worker.p 4    worker.fu
P Master.p 1    master.fu
#
# Terrain Specification Section
#
# ptag    htag constraint list
# -----
T Master.p[0] A    TYPE = SGI SPEED >= 1000
-F Worker.p[0] B    NAME = htsum2
C Worker.p[1] C    TYPE = SGI
C Worker.p[2] B    TYPE != IBM TYPE != SGI
T Worker.p[3] C    NAME = htsgi3

```

Figure 19: Blueprint for MatrixMultiply1

```

# Blueprint for MatrixMultiply2
#
# 10X10 integer matrices; 4 Workers
#
# Input Specification Section
#
# itag          count mtag
# -----
I GetMats.i      2  Mat.m
I GetRowCol.i    1  Pair.m
I GetVal.i        1  Val.m
I DoneM.i        104 Val.m
I DoneW.i         1  Halt.m
#
# Computation Specification Section
#
# ctag          type  argument
# -----
C Read.c         exec  read_mats
C Mult.c         call  mult_vectors
C Assign.c       sleep 0
C Print.c        exec  print_matrix
#
# Output Specification Section
#
# otag          rcv_ptag argument list
# -----
O RtrnVal.o     ACK      -t Val.m -n 1 -b 12
O SndRowCol.o   ACK      -t Pair.m -n 1 -b 8
O SndMats.o     Worker.p -t Mat.m -n 2 -b 400
O HaltWkrs.o    Worker.p -t Halt.m -n 1
#
# Functionality Specification Section
#
# ftag  itag          ctag          otaglist  nextfetag
# -----
F wf1.f GetMats.i  NULL          RtrnVal.o  SAME
F wf2.f GetRowCol.i Mult.c        RtrnVal.o  SAME
F wf3.f DoneW.i    NULL          NULL        NULL
F mf1.f NULL       Read.c         SndMats.o  SAME
F mf2.f GetVal.i   Assign.c       SndRowCol.o SAME
F mf3.f DoneM.i    Print.c         HaltWkrs.o NULL
#
# Functional Unit Specification Section
#
# futag          ftags
# -----
U worker.fu wf1.f wf2.f wf3.f
U master.fu mf1.f mf2.f mf3.f
#
# Process Specification Section
#
# ptag          N futaglist
# -----
P Worker.p 4 worker.fu
P Master.p 1 master.fu
#
# Terrain Specification Section
#
# ptag          htag constraint list
# -----
T Master.p[0] A  NAME = htsgi1
T Worker.p[0] B  TYPE = HP
T Worker.p[1] C  TYPE = SUN
T Worker.p[2] D  TYPE = IBM
T Worker.p[3] E  TYPE = DEC
#

```

Figure 20: Blueprint for MatrixMultiply2