IPS-2: The Second Generation of a Parallel Program Measurement System

Barton P. Miller * *Morgan Clark* + *Jeff Hollingsworth* *
Steven Kierstead † *Sek-See Lim* * *Timothy Torzewski* ††

Abstract

IPS is a performance measurement system for parallel and distributed programs. IPS's model of parallel programs uses knowledge about the semantics of a program's structure to provide two important features. First, IPS provides a large amount of performance data about the execution of a parallel program, and this information is organized so that access to it is easy and intuitive. Second, IPS provides performance analysis techniques that help to automatically guide the programmer to the location of program bottlenecks.

IPS is currently running on its second implementation. The first implementation was a testbed for the basic design concepts, providing experience with a hierarchical program and measurement model, interactive program analysis, and automatic guidance techniques. This implementation was built on the Charlotte Distributed Operating System. The second implementation, IPS-2, extends the basic system with new instrumentation techniques, an interactive and graphical user interface, and new automatic guidance analysis techniques. This implementation runs on 4.3BSD UNIX systems, on the VAX, DECStation, Sun 4, and Sequent Symmetry multiprocessor.

Index Terms

Performance measurement, parallel and distributed programs, instrumentation, critical path analysis, message systems, shared-memory systems, UNIX.

---

* Miller, Hollingsworth, and Lim's address: Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, Wisconsin 53706.

+ Clark's address: AT&T Bell Laboratories, UNIX Software Operation, 190 River Rd. SFB-111, Summit, NJ 17901

† Kierstead's address: AT&T Bell Laboratories, 5555 Touhy Avenue, Rm 178-6, Skokie, Illinois 60077.

†† Torzewski's address: Digital Equipment Corporation, 310 Rockrimmon Blvd. South, Colorado Springs, CO 80919.

1. INTRODUCTION

IPS is a performance measurement system for parallel and distributed programs. IPS's model of parallel programs uses knowledge about the semantics of a program's structure to provide two important features. First, IPS provides a large amount of performance data about the execution of a parallel program, and this information is organized so that access to it is easy and intuitive. Second, IPS provides performance analysis techniques that help to automatically guide the programmer to the location of program bottlenecks.

IPS is currently running on its second implementation. The first implementation [1-3], was a testbed for the basic design concepts, providing experience with a hierarchical program and measurement model, interactive program analysis, and automatic guidance techniques. This implementation was built on the Charlotte Distributed Operating System [4]. The second implementation, IPS-2, extends the basic system with new instrumentation techniques, a powerful interactive and graphical user interface, and new automatic guidance analysis techniques. This implementation runs on 4.3BSD UNIX systems.

The next section presents an overview of the IPS concepts and model. In this section we describe the hierarchical program and measurement model of the IPS system. New techniques for instrumenting parallel programs are described in Section 3, including of the overhead caused by using IPS-2. Section 4 describes the graphical user interface. This interface is used to specify the program to be measured and to interactively inspect the performance results from the execution of the program. Section 5 discusses two automatic guidance techniques. Critical Path Analysis [2] is reviewed and new features are described. A new guidance technique, called Phase Behavior Analysis, in presented. Section 6 presents our conclusions and mentions ongoing research to develop new analysis techniques.

2. IPS OVERVIEW

IPS is based on a hierarchical model of parallel and distributed programs. A hierarchical model presents multiple levels of abstraction, provides multiple views of performance data, and has a regular structure. The objects in a hierarchical model are organized in well-defined layers separated by interfaces that insulate them from the internal details of other layers. Therefore, we can view a complex problem at various levels of abstraction. We can move vertically in the hierarchy, increasing or decreasing the amount of detail that we see. We can also move horizontally, viewing different components at the same level of abstraction.

In this section we review the sample hierarchy of IPS that is based on our initial target systems — the Charlotte Distributed Operating System and 4.3BSD UNIX. Charlotte is a distributed operating system written at the University of Wisconsin, running on VAX 11/750's connected via an 80 megabit/second token ring. Both Charlotte and 4.3BSD systems consist of processes communicating via messages. These processes execute on machines connected via high-speed local networks. The hierarchy presented here served as a test example of our hierarchy model and reflects our current implementation. It is easy to extend these ideas to incorporate new features and other programming abstractions. For example, in our Sequent multiprocessor implementation, we include light-weight processes (processes in the same address space) to our hierarchy with little effort. Our hierarchical structure can be also applied to systems such as HPC[5], which has a different notion of program structuring, or MIDAS[6], which has a 3-level programming hierarchy. The IPS paradigm would work with most systems that have regular, hierarchical decomposition of components.

2.1. The Program Hierarchy

An overview of our computation hierarchy is illustrated in Figure 2.1.

(A) *Program Level*

This level is the top level of the hierarchy, and is the level in which the distributed system accounts for all the activities of the program on behalf of the user. At this level, we can view a distributed program as a black box running on a certain system to which a user feeds inputs and gets back outputs. The general behavior of the whole program, such as the total execution time, is visible at this level; the underlying details of the program are hidden.

(B) *Machine Level*

At the machine level, the program consists of multiple threads that run simultaneously on the individual machines of the system. We can record summary information for each machine, and the interactions (communications) between the different machines. The machine level provides no details about the structure of activities within each machine.

(C) *Process Level*

The process level represents a distributed program as a collection of communicating processes. At this level, we can view groups of processes that reside on the same machine, or we can ignore machine boundaries and view the computation as a single group of communicating processes.

If we view a group of processes that reside on the same machine, we can study the effects of the processes competing for shared local resources (such as CPUs and communication channels). We can compare intra- and intermachine communication levels. We can also view the entire process population and abstract the process's behavior away from a particular machine assignment.

(D) *Procedure Level*

At the procedure level, a distributed program is represented as a sequentially executed procedure-call chain for each process. Since the procedure is the basic unit supported by most high-level programming languages, this level can give us detailed information about the execution of the program. The step from the process to the procedure level represents a large increase in the rate of component interactions, and a corresponding increase in the amount of information needed to record these interactions. Procedure calls typically occur at a higher frequency than message transmissions.

(E) *Primitive Activity Level*

The lowest level of the hierarchy is the collection of primitive activities that are detected to support our measurements. Our primitive activities include process blocking and unblocking by the scheduler, message send and receive, process creation and destruction, procedure entry and exit. Each event is associated with a probe in the operating system or programming language runtime that records the type of the event, machine, process, and procedure in which it occurred, a local time stamp, and event type dependent parameters.

2.2. The Measurement Hierarchy

The program hierarchy provides a uniform framework for viewing the various levels of abstraction in a distributed program. If we wish to understand the performance of a distributed computation, we can observe its behavior at different levels of detail. We chose a measurement hierarchy whose levels correspond to the levels in our hierarchy of distributed programs. At each level of the hierarchy, we define performance metrics to describe the program's execution. For example, we may be interested in parallelism at the program level, or in message frequencies at the process level. We can look at message frequencies between processes or between groups of processes on the same machine. This selective observation permits a user to focus on areas of interest without being overwhelmed by all the details of other unrelated activities. The hierarchical structure matches the organization of a distributed computation and its associated performance data.

2.3. The Structure of IPS

There are four basic components of IPS: *instrumentation probes*, *data pool*, *analyst*, and *user interface*. The instrumentation probes generate trace data when interesting events happen during the program execution. These probes are contained in the language runtime library and the operating system kernel. The data pool stores the trace data and caches intermediate results from the analysts. The data pool is resident in the memory of each machine. The analyst is a set of processes that summarizes and evaluates the measurement data. The user interface interacts with the user and presents the results.

Each machine contains a slave analyst that analyzes the trace data generated by the processes on that machine. The master analyst performs the program level analysis and coordinates with the slave analysts to synthesize the measurement and analysis data. In addition, it provides an interface with the user for the display of performance results. Figure 2.2 shows the basic structure of IPS.

3. INSTRUMENTATION TECHNIQUES

The overriding consideration in collecting performance data is efficiency. To efficiently gather data we must minimize the overhead, both in time and space. Collecting the trace information should not require much extra time, and the trace records should not take up much extra space, when compared to running the same programs without tracing them. The current version of IPS is based on software instrumentation. Hardware instrumentation would allow less intrusive monitoring of parallel programs. Currently, no monitoring tools are generally available, and we are investigating building our own hardware monitoring facility. The problem of how to efficiently correlate hardware-level monitoring with program-level analyses must also be investigated.

Programmers do not have to modify their programs to use IPS-2. Data is automatically collected from two sources: (1) modified† procedure call hooks used by gprof [7], and (2) a modified runtime library. Instrumentation is selected by a compiler option.

In this section, we first discuss the implementation of our new software instrumentation techniques, then present measurements on the performance overhead incurred when using IPS-2.

3.1. Implementation Issues

The initial version of IPS was limited in the type of performance data that it collected. Data for process, machine, and program level events was collected by tracing; that is, every important event was collected and recorded. Data for procedure level events was collected by periodic sampling. Events at the procedure level (specifically, procedure entry and exit events) occurred much more frequently than events at the other levels and sampling was used to keep the instrumentation space and time overhead manageable. The result of using sampling is that information at the procedure level was only approximate.

IPS-2 has improved the efficiency of event tracing so that we now use traces at all levels. This has two benefits. First, we get exact performance results at all levels of the hierarchy. Performance results at the procedure level have the same precision as results at the other levels. Second, IPS-2 has been extended to shared-memory, multiprocessor machines. The process interactions on such systems occur at a higher frequency than on loosely-coupled systems. The techniques used to trace procedure level events are used in the shared-memory environment to trace process interaction events.

We use several techniques to reduce both time and space requirements of event tracing. The most significant problem with the cost of tracing is the time needed to collect timestamps for each trace record. Each event that is traced by IPS requires the elapsed time (real time) and CPU time to be recorded. These times are typically accessed by using a operating system kernel call. Kernel calls are several orders of magnitude slower than procedure calls and add intolerable overhead if used for tracing procedure call events. All UNIX versions that we have examined require a kernel call to access at least one of these two types of time.

---

† Gprof collects data only on procedure entry. We make an extra pass over a program's assembly code to also monitor procedure exit.

The solution to this problem is to access clock values with simple memory references. The clock on most machines is stored either in the kernel's address space as one or more integer values or is accessible via memory-mapped clock device registers. In our VAX implementation, we modify UNIX to provide a kernel facility to map the clocks (both the process's CPU time and real time) into a process's address space (read-only). Processes read the clock at memory access speed. In our implementation for the Sequent Symmetry multiprocessor we use an auxiliary clock provided by the Sequent architecture. This is a hardware 1 MHz clock that can be mapped into a process's address space and read directly. A similar solution us used for CPU time, by directly (mapping and) reading the process's process table entry. The performance benefit of using memory-mapped clocks is quantified in Section 3.2, where we compare the overhead of reading a clock from memory to the overhead of reading it with a kernel call.

We use three methods to reduce the size of the traces. The first method addresses procedure calls and returns, which are usually the most frequently occurring traces. Process level traces (corresponding to kernel calls) generally need auxiliary information, such as return codes or message sizes, but procedure calls and returns need no information other than the timestamps and an identifier of the procedure that was called. Therefore procedure call traces are smaller than other types of traces. The second method is to shorten every trace record by encoding some of the information. To generate timestamps we read a two-word (64 bit) clock. We then compress the two words into a one-word timestamp for the trace records, and recreate the original timestamp at analysis time. No significant information is lost by this method, since the time between any two traces will not exceed the time represented in a single word. The third method is to encode multiple events in a single trace. For example, a ''lock'' synchronization operation on the Sequent has two events, one to *try* to acquire the lock (and possibly block), and another event to actually acquire it. For most cases, we can generate a single trace for these two events that includes the time difference between the two events.

Directly reading clocks can cause anomalies. One problem involves reading a multi-word clock. The clock might be updated between reads of the separate words. Detection and correction of this problem is straightforward, because the interval between a correct timestamp and a following incorrect timestamp appears to be negative. The incorrect value can be easily corrected. A second problem arises when different clocks have different resolutions. For example, in our Sequent implementation, the real time has a 1 microsecond resolution, while the process time has only a 10 millisecond resolution. This can cause a discrepancy when the process time is rounded to a value greater than was actually used. This problem is easy to detect, but hard to correct as the precise value of the process time is not known. Typically, computations must be based on the resolution of the least precise clock.

Tracing shared-memory inter-process communication is difficult. In the most general case, we would need to trace every memory reference in any shared areas in the processes' address spaces. This would be difficult and would require extensive hardware support. Instead we opted to trace only kernel calls relating to shared-memory synchronization mechanisms. For example, the Sequent supports semaphore operations. We trace semaphore blocking and restarting of blocked processes, but we do not trace memory references inside shared regions protected by semaphores.

Operations that directly involve the operating system can cause problems when creating traces. For example, to trace the times when a process is blocked awaiting a free processor, the scheduler inside the operating system kernel will generate trace records. A potential race condition arises, as both the operating system and the process may be trying to write a trace record. This issue will be addressed in an upcoming version of IPS that includes scheduler blocking time measurements.

3.2. Performance

This section presents measurements of the overhead on application programs caused by using IPS-2. The results presented were taken from Microvax-II workstations and from the Sequent Symmetry multiprocessor.

Two programs were measured, a parallel sort program and a parallel solution to the Traveling Salesman Problem [8]. The sort program was based on a divide-sort-merge algorithm. It was run on randomly generated lists, from 1000 to 8000 records. Each run of the sort program was repeated 10 times (with a different randomly generated list of records), so actual sort times are 1/10 those reported. The Traveling Salesman program used a branch-and-bound algorithm. This program was run for a problem size (number of cities) of 16, over several input data sets. The sort program was run on the Microvax and the Traveling Salesman program was run on both the Microvax and Sequent. For each input/problem size, all programs were run three times: (1) without any tracing, (2) with IPS tracing, and (3) with UNIX ''gprof'' [7] procedure call profiler tracing. For each run of a program, elapsed time and CPU time were recorded. Procedure call rates and trace log sizes were also calculated from the IPS runs. These results are summarized in Figures 3.1 and 3.2.

The first result to examine is the percent overhead (as calculated from the elapsed times). The overhead for programs run under IPS-2 ranges from 10-45%. This compares favorably with the overhead from the standard UNIX profiler, gprof. The percent overhead under IPS-2 increased, predictably, with the frequency of procedure calls. The two test programs that we measured consisted of relatively small procedures (average size, 25 lines, including white space and comments), so we should expect overhead results for other programs to be as good or better than those in the figures.

Note the two sets of IPS-2 performance times in Figure 3.2. Each program on the Sequent was run twice, once with instrumentation code using a memory-mapped clock to sample CPU time and once using a kernel call (‘‘getrusage()’’) to obtain CPU time information. We can see the substantial penalty in having to enter the operating system for timing information.

Figures 3.1 and 3.2 also shows the size of the trace generated by the various program runs. Examples range from 206K bytes, to a relatively large trace of 1.4 Mbytes in 25 seconds. The maximum rate at which traces were generated in these runs was about 56K bytes/second. At these rates, memory can hold a substantial part of the trace and the disk write operations needed to flush the trace buffer are infrequent.

4. USER INTERFACE

The first version of IPS had a simple textual user interface. This interface provided access to the IPS facilities, but was limited in two ways. First, the interface did not allow the programmer to visualize the program model. The hierarchical model has an intuitive visual representation and the textual interface could not use this. Second, the textual interface did not allow for graphical display of performance results. The ability to graph performance metrics over time and to graphically compare performance results gives the programmer valuable information.

The IPS-2 interface allows the programmer to specify both the structure of the program to be measured and the performance results to be displayed. The programmer starts in a graphic editor mode. The editor allows the programmer to modify the structure of the program, save and re-edit it, or execute the program. After the program has executed, the programmer interacts with a flexible user interface to display any combination of performance metrics for nodes in the program tree. The programmer can display performance metrics in tabular or graphical form, or use the automatic guidance techniques, Critical Path Analysis and Phase Behavior Analysis. In addition, standard gprof-style profiling data is available at each level of the hierarchy. Figures 4.1 and 4.2 show an example of a session with IPS-2.

The programmer starts with a single window showing a program level node (the triangle node in the window with the tree in Figure 4.1). To this program node, the programmer can add machine nodes. Each machine node represents a host machine on which the processes of the program will run. In the example, these machines are called ‘‘grilled’’ and ‘‘havarti’’. The programmer can also specify parameters (using pop-up property sheets), such as account names and home directories, for these machines. Next, the programmer specifies the initial processes to run on each machine (‘‘test2a.swb’’ and ‘‘test2b.swb’’). For each process, the programmer can specify the executable file to be run in the process, parameters to the process, and input and output files. Figure 4.1 shows the program tree with the property sheet for machine ‘‘havarti’’. After the program specification is completed, it can be saved for later use.

IPS-2 can now be used to run the program. IPS-2 will transfer (if necessary) each executable file to the correct host machine, start the processes, monitor them, and report back when they have completed. A new program tree will be displayed with additional information from the program execution. New process nodes may appear as a result of dynamic process creation and procedure level nodes will appear for each procedure executed in the program (nodes such as ‘‘getData’’ and ‘‘calc1’’ in Figure 4.2).

Large programs can spawn many processes and call many procedures. IPS-2 provides functionality to mananage the display complexity in the tree window. Single mouse-button and keyboard commands can be used to: (1) hide all descendants of a node, (2) hide a single node, or (3) show the immediate children of a node. There are also commands to show only those nodes that contribute more than a certain percentage to the total CPU time or critical path. In addition, a horizontal scroll bar is provided at the bottom of the window.

The table at the bottom, left corner of Figure 4.2 shows a metric table for process ‘‘test2a.swb’’. Various performance metrics have been displayed for this process. Added to this table was a list of all child nodes, i.e., the procedures that ran in this process. Any combination of nodes and metrics can be displayed in a table.

In the center of the screen is a graph of the ‘‘CPU Time’’ metric for the whole program (out of 200%, because there are 2 machines), and superimposed on this display is the graph of the same metric for machine ‘‘grilled’’. The graphs can be zoomed to get more detail, panned to examine individual portions of the program history, and enlarged to show more detail. The window on the bottom right hand corner of the screen displays graphs of multiple metrics, message rate and CPU time. Any combination of metrics and nodes can be displayed in single graph.

An important aspect of this interface is its simplicity. There are few commands and menus, and the structure of the commands and displays matches a programmer’s notion of the structure of the program.

5. AUTOMATIC GUIDANCE TECHNIQUES

A major goal of the IPS system is to provide program performance analysis techniques that guide the programmer in the search for performance problems. We provide the programmer with information to directly locate performance bottlenecks. In this section, we briefly outline our first guidance technique (Critical Path Analysis) and then describe new features for this analysis. We then describe a new technique called Phase Behavior Analysis, and show how it interacts with the metric tables and Critical Path Analysis.

5.1. Critical Path Analysis

Our first guidance technique was based on identifying the path through the program that consumed the most time [2]. This *critical path* identifies the parts of the program responsible for its length of execution (based on traces of the program’s execution history). This information is more precise than just a profile of the execution times of each part of a program. The critical path identifies the parts of the program (including CPU times, synchronization and communication delays) that cause the execution time. If we speed up the events along the critical path, we speed up the whole program.

Critical Path Analysis (CPA) can identify program parts that occur most frequently in the critical path, and can further identify the most frequent *sequences* of events along the critical path. The ability to locate frequent sequences allows us to detect bottlenecks spread across several procedures or across several processes or machines. The results of the Critical Path Analysis can be displayed at the different levels of abstraction: we can observe the most frequent elements of the path at the program, machine, process, and procedure levels.

To perform CPA, we construct a graph of the program's activities (a *Program Activity Graph*, or *PAG*) from the trace information generated during execution. This graph represents the time dependencies among the various parts of the program and is built from the program traces using only those records that show an interaction between two processes (inter-process communication and process creation events). Other records only appear in the PAG as elapsed time. Nodes in the PAG represent events (e.g., inter-process communication and process creation) and arcs represent observed timings.

A slave analyst handles the traces from the processes on its machine. It first builds one subgraph per process, and then uses the trace information to combine these subgraphs with the subgraphs for the other processes (on the same machine and on others). Slaves compute these results concurrently. Finally, we add global initial and final nodes to combine all the subgraphs into a single PAG for the whole program.

After constructing the PAG, we find the critical path (the longest time-weighted path through the graph) using a distributed algorithm based on one by Chandy and Misra[9] and adapted to our problem for the original version of IPS [1]. The adaptation focused on two areas. First, Chandy and Misra represented each node with an analyst process. Since PAG's can contain tens of thousands of nodes, that number of processes would be unworkable on current operating systems. In our implementation, a single slave analyst represents the PAG subgraph for all processes that ran on that slave's machine. Second, Chandy and Misra designed their algorithm to find the shortest path through a (directed) graph. Since the PAG is acyclic (all arcs represent a forward progression of time), shortest path algorithms apply equally well to the problem of searching for the longest path through the PAG.

Figure 5.1 illustrates a simple PAG. In this figure, time progresses from top to bottom. Processes A and B ran on one machine, and Process C on another. Arcs are weighted with time values, and the critical path is marked with double lines.

The master analyst is responsible for requesting that the Critical Path Analysis be performed, consolidating the information gathered from that analysis, and presenting it to the user. Since it is impractical to consider a graphical display of the thousands of nodes that can make up the critical path, we present critical path information to the user statistically. For example, at the process level, we present a table, sorted by percentage of total time, of how much of the critical path execution time was due to CPU time in each process, and how much was due to inter-process communication between each pair of processes. Similar presentations are available at the program, machine and procedure levels. The windows at the top, right corner of Figure 4.2 show critical path results for the process and procedure levels of our test program.

It is possible to have a PAG in which the longest and second longest paths do not overlap (except at beginning and end). In this case, improving the critical path may have little affect on the program's performance. Fortunately, experience has shown that the longest path and second longest path have substantial overlap. There is still the question: how much improvement will we really get by fixing something that lies on the critical path?

While this question can not be answered in general, the critical path analysis provides a feature that can help. For any element(s) on the critical path, we can change their weight to zero and recalculate the critical path. We can then compare the length of the new path with the original critical path. This is only an approximation of the affect of a change to the program, but it provides some insight about the change.

For example, Figure 5.2, top right corner, shows the critical path table for the procedure level. We have selected the procedure that contributes the largest time on the path (‘‘calc2’’ in process ‘‘test2a.swb’’) and assigned its weight to zero. This creates a new context (‘‘Context 1’’), which is based on the original PAG, but with all of the weights for ‘‘calc2’’ event edges set to zero. To the left of the original critical path table in Figure 5.2 is a window with a new critical path table, based on the modified PAG. We can see that eliminating ‘‘calc2’’ can substantially change the critical path. The length of the path has changed from 2.85 to 1.20, indicating that the execution time might be substantially improved if ‘‘calc2’’ could be made more efficient. The contents of the critical path have also changed – procedure ‘‘getdata’’ in process ‘‘test2b.swb’’ is now the major contributor to the critical path.

5.2. Phase Behavior Analysis

Programs go through different phases during the course of their executions. For example, a master/slave parallel program might have the following phases: (1) the master process sets the initial problem, (2) the slave processes are initialized, (3) the master distributes pieces of the problem to each slave, (4) the slaves compute their piece of the program, (5) the master reaps the partial results and combines them. Steps (3)-(5) are repeated until a solution is reached. Each of these phases has different execution characteristics. The goal of the Phase Behavior Analysis is to automatically identify phases in the program's execution history. Once these phases are identified, we can then use our other analysis techniques, *focusing* on each phase as a separate problem. Each phase represents a simpler subproblem, which should be easier to evaluate and improve its execution.

Intuitively, a phase is a period of time when the program is performing the same activity. For our performance tool, we define the phase as a period of time where some combination of performance metrics maintain consistent values. For example, in the graph in the center of Figure 4.2, CPU time is displayed for an entire program. For this single metric, we can observe periods of low CPU usage and periods of high CPU usage. In the Phase Behavior Analysis, we take several such graphs (for different metrics, such as message frequency or procedure call frequency, or for different parts of the program) and identify common periods between these graphs.

Our detection algorithm inputs raw metric curves that are derived from the trace data generated by the instrumented programs. Each metric curve is represented by a list of discrete values for a finite number of points in time, summarized from the total execution period of the program. The algorithm works in three steps: *smoothing*, *segmenting*, and *combining*. The smoothing step reduces spikes from the raw metric curves. The segmenting step determines the potential segment boundaries in the execution history graph for a single performance metric. The combining step identifies the phases in the overall program execution from the common segment boundaries in a list of metrics.

5.2.1. Smoothing

The goal of the smoothing step is to simplify the segmenting step by reducing spikes in the performance data. The current smoothing function is a sliding window average, weighting the center point most and the edges of the window least. A window size of 9 (empirically determined) suppresses spikes that results from the fine granularity of the trace data collected. The smoothing function has the same effect as a low pass filter. Increasing the window size effectively lowers the cutoff frequency. Each smoothed curve is normalized with respect to the maximum value of that metric (as constrained by physical and operating systems characteristics of the machines). The smoothed and normalized metric curve is then used to compute segment boundaries.

5.2.2. Segmenting

An execution history graph, G_m, for metric m can be divided into segments, $S_{m,i}$, where $S_{m,i}$ starts at time t_i and ends at t_{i+1} $(t_i < t_{i+1})^{\dagger}$. A new segment is started at time t_i when values for the metric m during $S_{m,i-1}$ differ significantly from the values immediately after time t_i.

To derive segments, we define a *boundary curve*, B_m, for metric m that shows the likelihood that any given point on the metric curve is at the end of a segment. To calculate B_m, we first calculate a *step* function to show the range of values for m. The step function, $h_{m,i}$, for metric m at time t_i is the difference in value of m between the previous minimum (maximum) and the following maximum (minimum). Figure 5.3a shows the step function for the metric curve in Figure 5.3b. Next, we define two variables for computing the first derivative of the metric curve: time and value increments. The time increment, Δt_i, is the difference between the present time, t_i, and the previous time, t_{i-1}, in which the metric was sampled. The value increment, $\Delta V_{m,i}$, is the difference in the value of the metric m at time t_i and t_{i-1}, as shown in Figure 5.3a. Thus, the first derivative of the metric curve at time t_i is approximated by $\frac{\Delta V_{m,i}}{\Delta t_i}$.

The boundary curve is derived by multiplying the absolute value of the first derivative of the metric curve with the step function, $h_{m,i}$. Thus the boundary curve, B_m, at time t_i, is defined

$$B_{m,i} = abs\left(\frac{\Delta V_{m,i}}{\Delta t_i}\right) \times h_{m,i}$$

The greater the value of $B_{m,i}$, the greater the probability that the corresponding point on the metric curve is at the end of a segment. We identify segment boundaries as the peaks of the boundary curve that are greater in value than some threshold.

5.2.3. Combining

After the boundary curves for each metric have been computed, they must be combined. If $B_{m,i}$ is high at time t_i for most of the metric curves, then there is a high probability that t_i is an endpoint of a phase. The combining function identifies the most common boundaries and generates the program phases based

† The notations here are used to represent discrete data rather than some continuous function of time.

on this combined list of metrics. The combining function sums up the boundary curves of each of the metric curves to compute the segment boundaries from the aggregate boundary curve. Hence, the aggregate boundary curve, B, at time t_i, is defined

$$B_i = \sum_{m \in M} B_{m,i} = \sum_{m \in M} abs\left(\frac{\Delta V_{m,i}}{\Delta t_i}\right) \times h_{m,i}$$

where M is the set of all the metrics used.

There is a phase boundary for the program at time t_i if the first derivative of the aggregate boundary curve is zero and B_i is greater than some threshold. The programmer interacts with the IPS-2 to determine a reasonable threshold value. If the threshold is too low, there will be too many phases and the results will not be useful. If the threshold is too high, there will be too few phases. Figure 5.4 shows a close-up of the graph of the CPU time and message frequency metrics for the program, and the corresponding boundary curve.

Note that the only manual step in identifying phases is setting the threshold. This is done by adjusting the slide bar on the left side of Figure 5.4. We are currently experimenting with heuristics to set this value automatically. Once we have identified the phases, we use the performance metrics and Critical Path Analysis to study these phases. We are investigating the use of Phase Behavior Analysis to find patterns and periods in a program's phases.

5.2.4. Using Phases with Other Analyses

IPS-2 can automatically identify phases or they can be specified manually. Once a phase has been identified and selected, we can use the other facilities in IPS-2 to study the behavior of that specific phase. We can display metric tables for a phase, and display the portion of the critical path that lies within the phase.

For example, we measured the execution of a shared-memory, parallel, database join program that runs on the Sequent Symmetry. The graph of total CPU time for one execution is shown in Figure 5.5. Note that there is a start-up interval of low CPU use. We identify two phases, phase ''A'' representing the start-up and phase ''B'' for the main computation. Figure 5.6 shows the procedure-level critical path table for the entire program (top right window), and below it, critical path tables for phases ''A'' and ''B''. We have re-sized these tables to show only the top eight entries; a scroll bar is used to see the others. We can see that the start-up phase (‘‘A’’) is dominated by procedure ‘‘random_shuffle’’ (used for initialization), but this procedure is not an important part of phase ‘‘B’’. Other changes in the critical path reflect the different type of work done in the different phases.

6. CONCLUSIONS

IPS-2 is a running [10] system whose design and features benefited from the experience gathered in the first (Charlotte Distributed Operating System) implementation. The first implementation of IPS provided useful insights in how to design a parallel program performance measurement tool. Using the semantic structure of the program produces a hierarchical model for the program and performance data. This model resulted in a system that was intuitive to use and provided large amounts of information. The model also allowed for the construction of analysis techniques that help guide the programmer to the cause of program bottlenecks.

IPS-2 uses this foundation to make several new advances. The new instrumentation techniques provides more detailed and precise information about the program. The implementation now includes both distributed and shared-memory systems. The graphical user interface simplifies use of the system and significantly improves the presentation of performance results. The Phase Behavior Analysis presents a new type of guidance technique: a focusing technique that allows more precise use of other analyses.

IPS-2 has been used in several performance studies, and we are gaining experience with several larger numerical applications. The Critical Path Analysis seems to have a real benefit, reducing the need to look through piles of statistics. We are just beginning to get experience with the Phase Behavior Analysis. To date, IPS-2 has been used to: (1) gather data to parameterize analytical performance models of parallel systems, (2) measure parallel database join algorithms, (3) evaluate code generated by parallelizing compiler algorithms, and (4) measure parallel search programs and network flow programs. The feedback that we have received from these studies has helped to improve the quality of the analyses and interface.

The strengths of IPS-2 are shown in the comments that we commonly receive. First, IPS-2 does not require modification of the user’s program. All instrumentation is automatically inserted at compile/link time. Second, IPS-2 has exposed performance problems in places not expected by the programmer. Third, IPS-2 seems to be easy to use; learning the basic features takes about 15 minutes.

IPS-2 is an evolving system. We are currently working on Critical Path Analysis advances, hardware instrumentation, browsing tools, refining Phase Behavior Analysis, kernel instrumentation, and new guidance techniques.

(1) The Critical Path work is to investigate second-longest, third-longest, etc., critical paths, and comparing and correlating information from these paths. We would like to compute these multiple paths efficiently.

(2) Hardware instrumentation has the potential to greatly reduce execution time overhead. We are currently instrumenting our instrumentation to better understand the type of data that we gather. This information will be used in the design of a hardware data collection facility.

(3) IPS-2 currently provides no way to browse through the raw trace data or critical path. We are currently designing browser functions to allow the programmer to intelligently select and display parts of the (potentially huge) trace files.

(4) IPS-2 can measure application programs, but not the operating system kernel. Instrumenting the kernel is more difficult than applications, but it will allow us to get system-level performance data. We will also be able to study an application along with its effect on the operating system.

(5) We are investigating new analyses for studying the contention for such resources as the CPU, memory, and communication channels.

ACKNOWLEDGEMENTS

We are grateful to Bruce Irvin for his work on the DECStation implentation of IPS-2. to Joann Ordille for her shared-memory parallel join program used in Section 5, and to Doug DeGroot for his helpful comments on preparing the final version of this paper.

Research supported in part by National Science Foundation grant CCR-8815928, Office of Naval Research grant N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.

7. REFERENCES

[1] B. P. Miller and C.-Q. Yang, ‘‘IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs,’’ *7th Int’l Conf. on Distributed Computing Systems*, pp. 482-489 Berlin, (September 1987).

[2] C.-Q. Yang and B. P. Miller, ‘‘Critical Path Analysis for the Execution of Parallel and Distributed Programs,’’ *8th Int’l Conf. on Distributed Computing Systems*, pp. 366-375 San Jose, Calif., (June 1988).

[3] C.-Q. Yang and B. P. Miller, ‘‘Performance Measurement of Parallel and Distributed Programs: A Structured and Automatic Approach,’’ *IEEE Transactions on Software Engineering* **12**(15) pp. 1615-1629 (December 1989).

[4] Yeshayahu Artsy, Hung-Yang Chang, and Raphael Finkel, ‘‘Interprocess Communication in Charlotte,’’ *IEEE Software*, (1987).

[5] T.J. LeBlanc and S.A. Friedberg, ‘‘Hierarchical Process Composition in Distributed Operating Systems,’’ *Proc. of the 5th Int’l Conf. on Distributed Computing Sys.*, pp. 26-34 (May 1985).

[6] C. Maples, ‘‘Analyzing Software Performance in a Multiprocessor Environment,’’ *IEEE Software*, pp. 50-63 (July 1985).

[7] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick, ‘‘gprof: a Call Graph Execution Profiler,’’ *Proceedings of SIGPLAN ’82 Symposium on Compiler Construction*, pp. 120-126 (1982).

[8] N. Lai and B.P. Miller, ‘‘The Traveling Salesman Problem: The Development of a Distributed Computation,’’ *Proc. of the 1986 Int’l Conf. on Parallel Processing*, pp. 417-420 St Charles, Ill., (August 1986).

[9] K. M. Chandy and J. Misra, ‘‘Distributed Computation on Graphs: Shortest Path Algorithms,’’ *Communications of the ACM* **25**(11) pp. 833-837 (November 1982).

[10] J. Hollingsworth, B.P. Miller, and R.B. Irvin, ''IPS User's Guide,'' *Computer Sciences Technical Report*, University of Wisconsin−Madison, (December 1989).

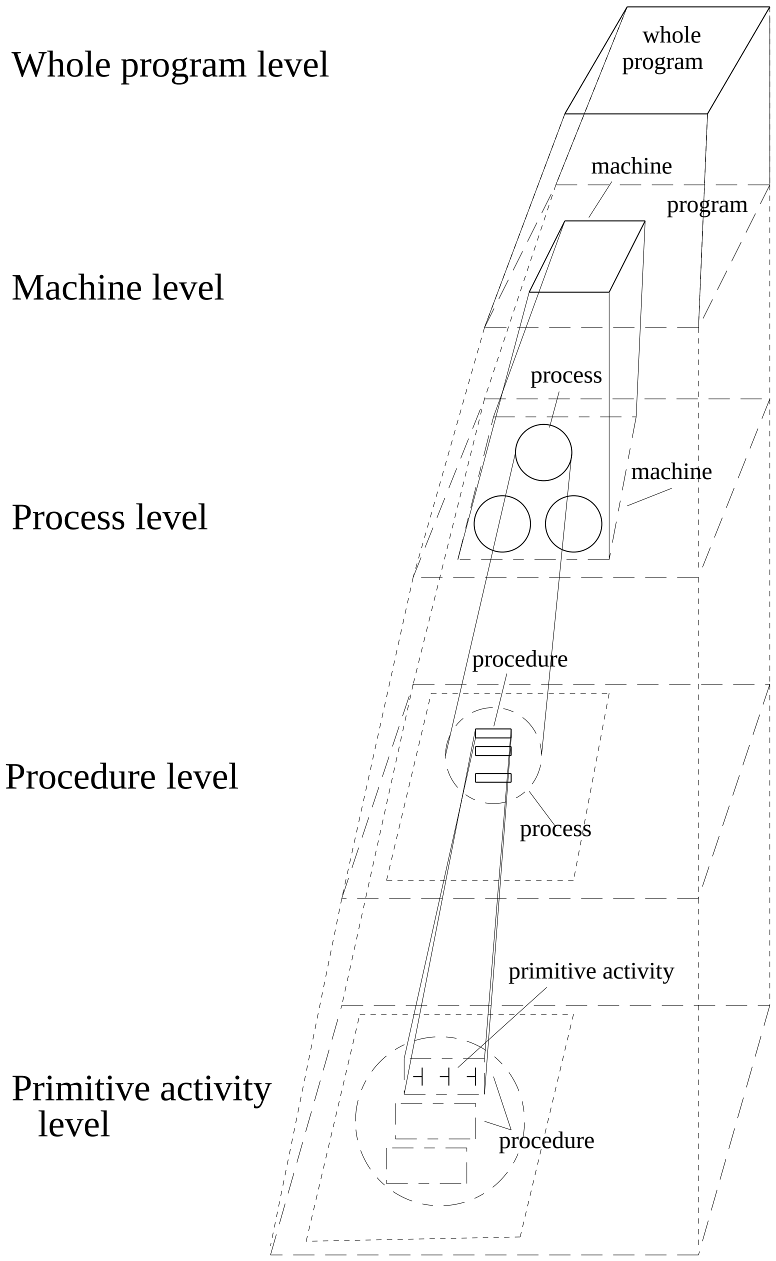

Figure 2.1: IPS Program Hierarchy

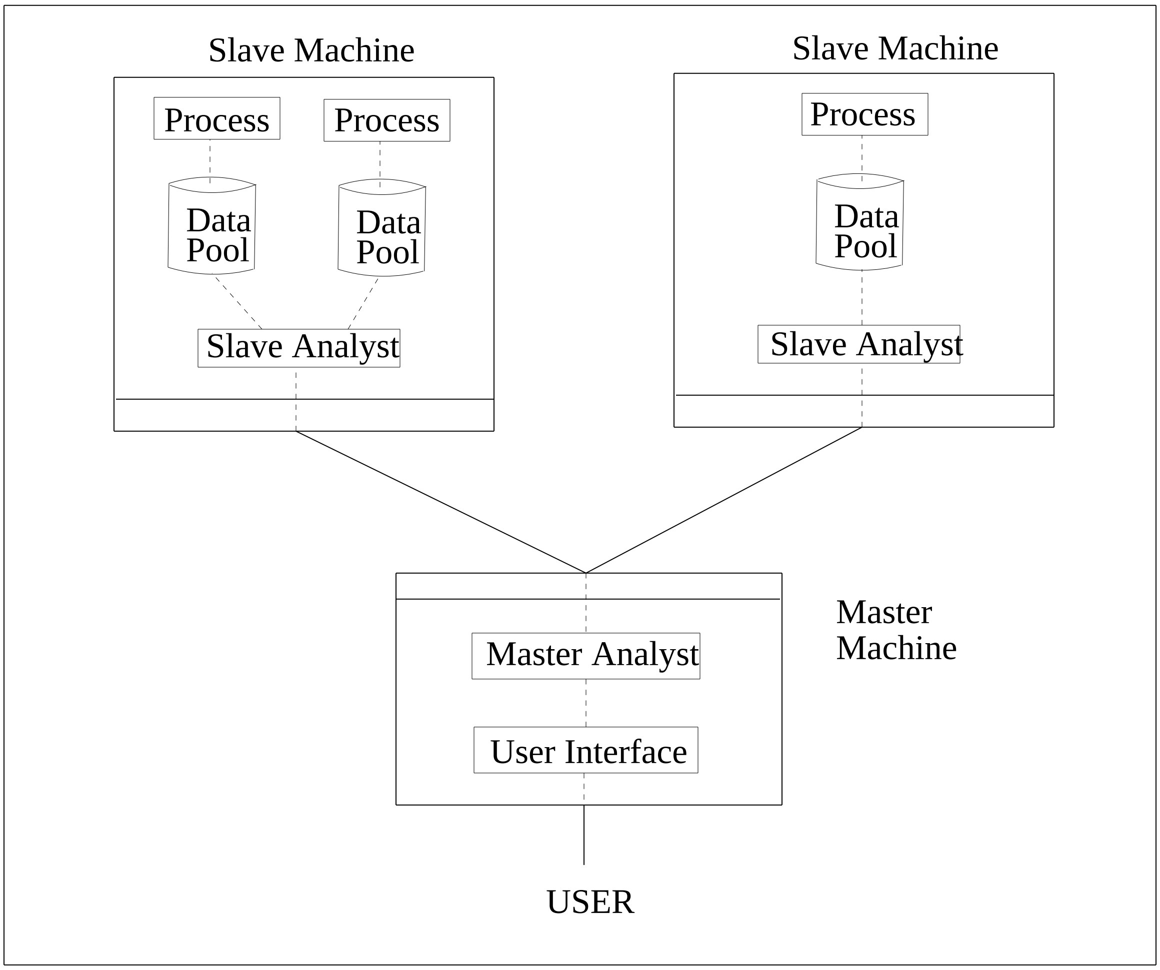

Figure 2.2: The Basic Structure of IPS

# Records	Untraced Elapsed Time	Untraced CPU Time	IPS Elapsed Time	IPS CPU Time	IPS Overhead	IPS Trace Size	gprof Elapse Time	gprof CPU Time	gprof Overhead	Proc. Calls/ Second
1000	3.95	3.38	4.91	4.84	24%	206568	5.60	5.30	42%	2325
3000	7.34	8.33	10.35	12.40	41%	541852	12.05	13.60	64%	3590
4000	9.39	10.86	13.39	16.25	43%	709056	15.82	17.83	68%	3764
5000	11.61	13.49	15.94	20.02	37%	888664	19.40	21.94	67%	3811
6000	13.27	16.00	19.28	23.82	45%	1062116	22.13	25.83	66%	3750
7000	15.61	18.74	22.11	27.56	41%	1233348	26.20	30.43	67%	3955
8000	17.87	21.42	25.33	32.00	41%	1408264	30.37	34.69	69%	3911

Figure 3.1: Overhead Measurements – Parallel Sort

All times in seconds; trace size in bytes. Program run on 2 Microvaxes, connected via an Ethernet.

	Untraced		IPS				gprof			
Config	**Elapsed Time**	**CPU Time**	**Elapsed Time**	**CPU Time**	**Overhead**	**Trace Size**	**Elapse Time**	**CPU Time**	**Overhead**	**Proc. Calls/ Second**
VAX	50.60	9.05	55.49	11.42	10%	441592	53.80	10.54	6%	142
Sequent w/ mem-map clock	7.91	7.18	8.51	8.05	7%	443008	8.25	7.52	4%	906
Sequent w/o mem-map clock			11.16	10.84	41%	443008				

Figure 3.2: Overhead Measurements − Traveling Salesman

All times in seconds, trace size in bytes. Microvax version run in 1 process, Sequent version run in 8 processes (on 8 CPUs). Problem size of 16 cities.

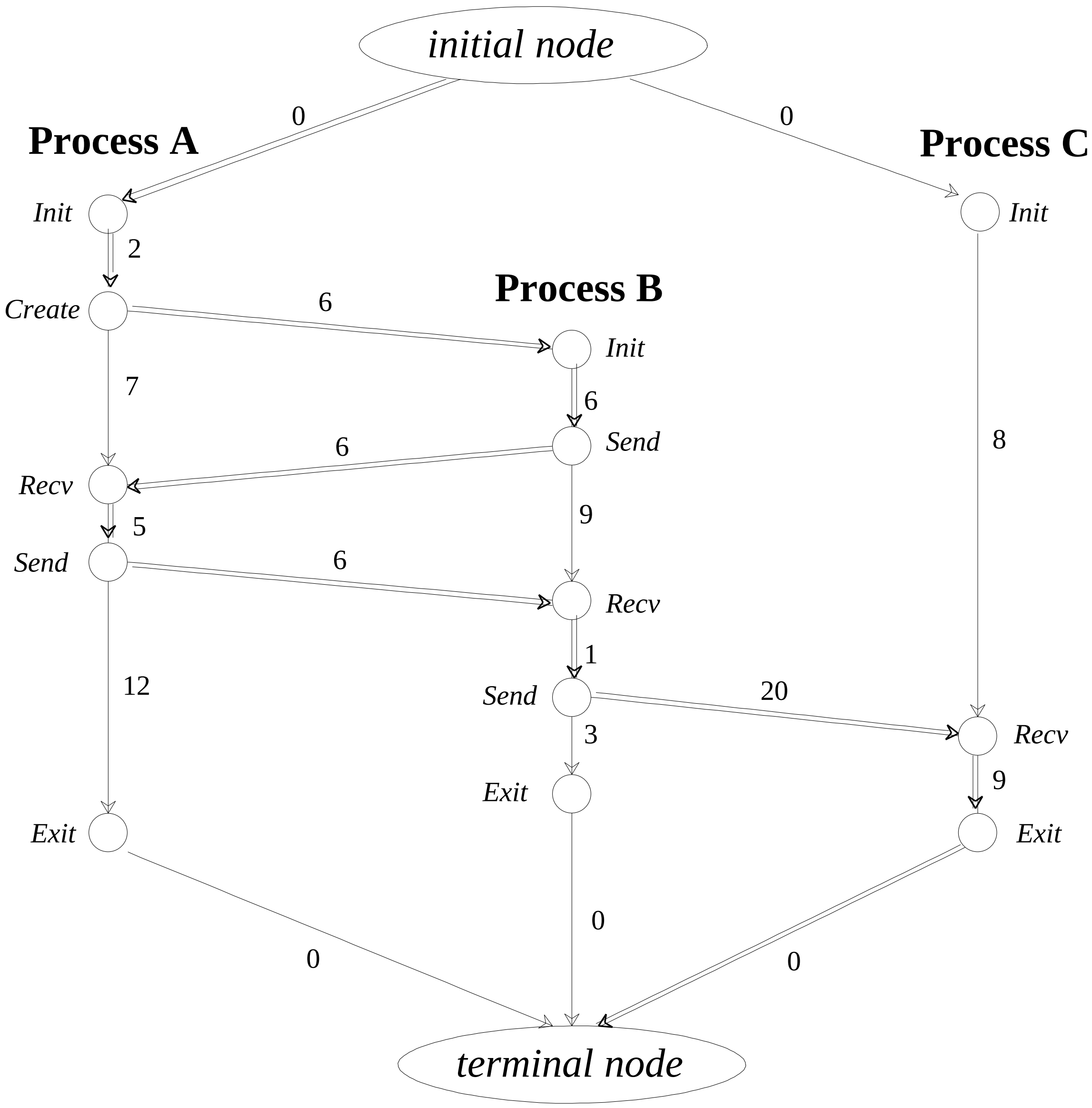

Figure 5.1: Sample Program Activity Graph

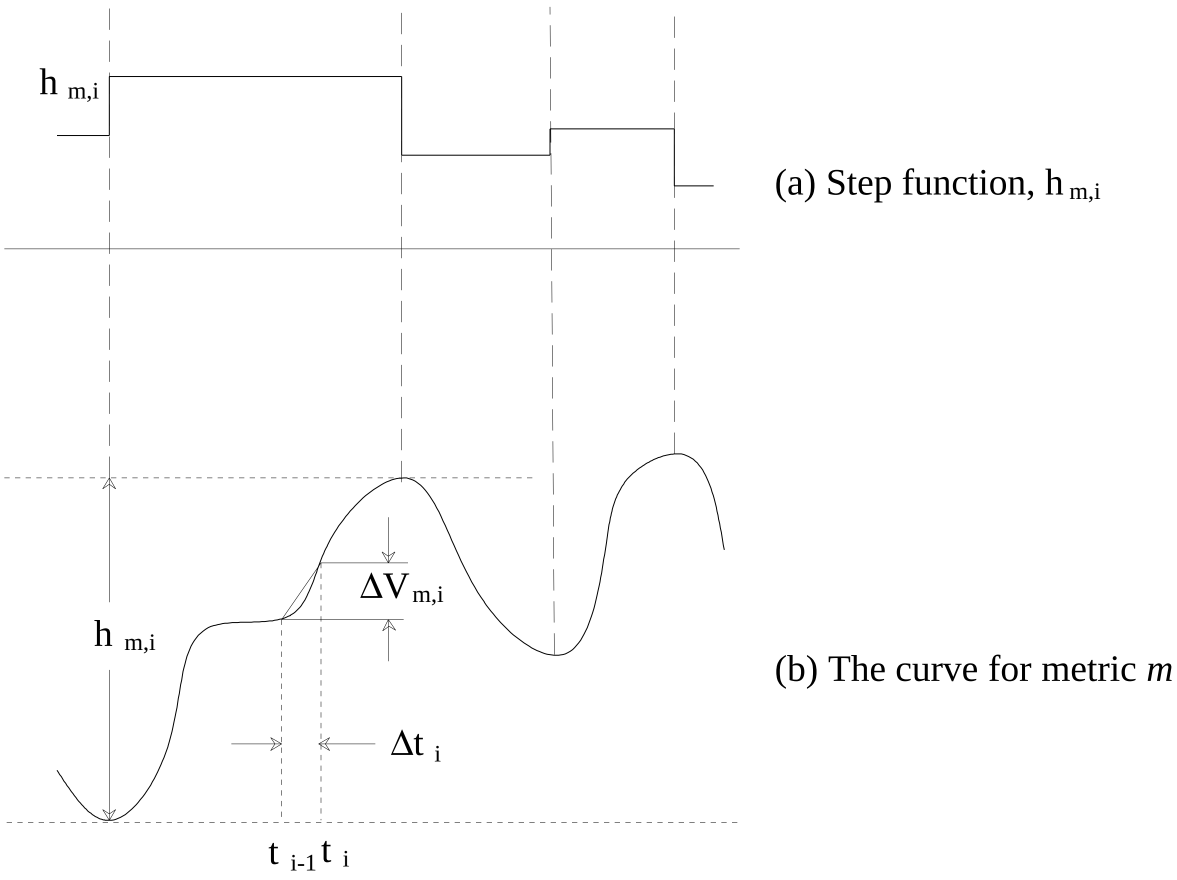

Figure 5.3: Definitions Used in Boundary Curve Calculation