Distributed Event-driven Simulation- Scheduling Strategies and Resource Management

Overeinder, B.J.

Publication date
2000

Chapter 4

APSE: Average Parallelism, Profile, and Shape Evaluation

Main Entry: **apse**
1 : APSIS 1
2 : a projecting part of a building (as a church) that is usually semicircular in plan and vaulted

—Merriam-Webster Dictionary

4.1 Introduction

Two measures of particular interest in parallel performance evaluation are *speedup* and *efficiency*. Notwithstanding the importance of these two measures, they do not reveal how well the available potential parallelism of the application is exploited. Nor do they help to understand why the performance may not be as good as expected. In particular, with the performance analysis and evaluation of PDES protocols we need a measure to compare the effectiveness of the different protocols. This can be done by a *relative* criterion that qualitatively compares the effectiveness of the protocols by measuring the speedup or the turn-around time. However, apart from a ranking of the different protocols, this does not answer the question how well we perform our task, that is, to comprehend how much of the potential parallelism is actually realized and what is the lower bound on the execution time. This lower bound on the execution time is the ultimate goal that the PDES protocols strive to achieve or at least to approach.

The potential or inherent parallelism of an application can be quantified as the *average parallelism* metric (Eager et al. 1989), which is a non-trivial upper bound to the asymptotic speedup of the software system (i.e., the speedup of the application with infinite resources and no synchronization costs). Thus, the average parallelism allows us to express the ability of a PDES protocol to exploit the potential parallelism in quantitative terms. For example, a statement that 80% of the available parallelism has been realized, indicates that 20% of the available parallelism has been wasted due to synchronization overhead. Note

that synchronization overhead has two components: communication overhead and PDES protocol overhead. The communication overhead is given by the communication substrate (hardware and software), while the PDES protocol overhead is determined by its efficacy to schedule simulation events in causal order.

In this chapter we propose to obtain the average parallelism of a simulation application by *critical path* analysis techniques. In this analysis, the discrete event simulation execution run is described by a task precedence graph, and well known techniques from graph theory and timing analysis are applied to determine a critical path in this graph. Besides the evaluation of the PDES protocol, the critical path analysis gives insight into the amount of available parallelism, and allows one to use the results of bottleneck analysis to improve the potential performance of the simulation application in a parallel environment.

The critical path analysis methodology is integrated within the APSIS simulation environment. A new and noteworthy feature is that the critical path analysis is performed on the task precedence graph that is obtained from a parallel execution. This is an important feature for spatially decomposed parallel simulations, where the decomposition strategy and the mapping of the application to the parallel architecture determine the available potential parallelism.

4.2 Characterization of Parallelism in Applications

The characterization of parallelism in applications can be described on different levels. At one extreme, the complete characterization of the parallelism in an application can be expressed in a *data dependency graph* (Veen 1986). In a data dependency graph, the concurrency is described on the level of arithmetic operations and assignments. Unfortunately, due to the level of detail, it is not practical to specify or analyze a full data dependency graph for programs of interest. At a less detailed level, portions of an application that are sequential (no internal parallelism) can be treated as tasks in a *task precedence graph* (Coffman 1976). Task precedence graphs are higher level than data dependency graphs, and are therefore more manageable.

At the opposite extreme, single parameter characterizations are very high level descriptions of the parallelism in an application. The *sequential fraction*, which is the fraction of the overall execution time that cannot be executed in parallel, was proposed by Amdahl (1967). Gustafson has argued that the limitation on the parallelism by the sequential fraction can be misleading, since in many applications, the parallel part of the computation grows with the number of processors, while the sequential portion does not grow, and hence the fraction typically decreases (Gustafson 1988).

Another single parameter characterization, the *average parallelism*, has been investigated by Eager, Zahorjan, and Lazowska (1989). The average par-

allelism gives a realistic upper bound to the asymptotic speedup of the parallel application when an unlimited number of resources are available. In the next sections, the average parallelism metric is defined formally. From this formal definition, we describe a method to obtain the average parallelism from the task precedence graph (also known as an event precedence graph in space-time diagrams) by critical path analysis.

4.2.1 The Average Parallelism Metric

The average parallelism metric can be defined in different equivalent ways. The specific definition of the average parallelism depends on the usage of the metric, that is, for the evaluation of the turnaround time, the speedup, or the efficiency. The common denominator in the equivalent definitions is the abstraction from the parallel hardware and the omission of all influences of system and communication overhead.

Definition 4.1 *The average parallelism, A, is defined as the average number of busy processors during the execution of the application, given an unbounded number of processors and no communication latency and other system overhead.*

Other equivalent definitions of the average parallelism measure are:

- The ratio of the total amount of work (expressed in time units as the total service time) and the theoretical turnaround time (without overhead and with ample processors allocated).
- The asymptotic speedup figures, if a hypothetical machine contains an unbounded number of available processors and there is no communication latency and other system overhead.

By deliberately keeping both the processor availability and the communication overhead out of the definition, the average parallelism reflects only the software characterization of parallelism. More precisely, we neglect the performance degradations caused by machine issues (lack of processors, communication delay and contention) to focus on the software factors of performance (non-optimality of the algorithm or program and software overhead). Just as the parallel machine size can be used as the hardware bound on parallelism, the average parallelism metric can be used as the software bound.

In addition to the average parallelism, there are several other parameters that provide some information about the available parallelism in the application. For example in Fig. 4.1, a graph is shown of the number of busy processors over the execution time of the application. We will refer to this as the *parallelism profile* of the application (Sevcik 1989). From the profile, the *shape vector* of the application is defined as the vector $p = (p_1, p_2, p_3, \ldots)$, where each p_i denotes the normalized fraction of execution time spent with degree of parallelism of i. Following the first alternative definition of Def. 4.1, the average parallelism can now also be determined by $A = \sum_i i \cdot p_i$.

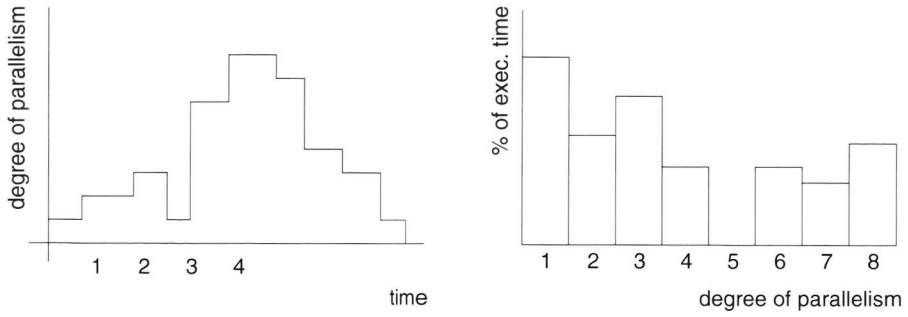


Figure 4.1: The parallelism profile of an application and its corresponding shape.

The simplicity of a single parameter characterization like the average parallelism is intuitively appealing. However, the parallelism profile and shape vector provide more information about how the parallelism is available in the application. From the parallelism profile and shape vector, additional parameters such as the minimum and maximum parallelism, and the variance of parallelism can be determined. These additional parameters are very valuable for scheduling decisions of how many processors to allocate to an application. Depending on the load of the system, the policy can determine the optimal number of processors allocated to the application while maximizing the speedup or the efficiency, or makes a trade-off between speedup and efficiency (Sevcik 1989).

By the definition of the average parallelism metric, and the other characterizations like parallelism profile and shape vector, the hardware component is totally absent. As a consequence, these parallelism characterizations cannot *directly* be obtained from the execution of the parallel application under discussion. As all influences of hardware components are ruled out from the characterizations, we need to ensure that during the analysis of the parallelism only the software components of the execution run are included. The first alternative equivalent definition of Def. 4.1 indicates that the average parallelism is ratio of the total service time to the theoretical turnaround time. The theoretical turnaround time is typically the longest weighted path in the task dependency graph. Thus, given a graph representation of the software component of the parallel execution, we can apply well-known critical path analysis methods to obtain the average parallelism.

4.2.2 The Space-Time Model

For the analysis of the parallelism in the simulation application, we need a sufficient detailed representation of the execution of the simulation run. From the previous discussion, it appears that a task precedence graph, or more appropriate for discrete event simulation, an *event precedence graph* or *space-time*

diagram, suffices for our analysis. The space-time diagram representation of the execution of the simulation run is intrinsic to the software component of the simulation application, that abstracts from the execution mechanism that drives the discrete event simulation. In this respect, the space-time diagram of a sequential execution run is identical to the space-time diagram of a parallel execution run, conservative or optimistic.

The space-time diagram describes the execution run of both sequential and parallel discrete event simulations that adhere to the process-oriented world view, such as described in Section 1.2.7 and Section 2.2. For convenience, we recapitulate the essential characteristics. In process-oriented discrete event simulation, the system being modeled is viewed as being composed of a set of physical processes that interact at various points in real time. The simulation is constructed as a set of logical processes, where each specifies the behavior of some physical process in the system. All interactions between the physical processes are modeled by time stamped event messages sent between the corresponding logical processes.

Given a particular decomposition of the simulation into logical processes, the execution of events must follow two fundamental precedence constraints, also known as causality constraints (which are a further specification of the local causality constraint as defined in Section 2.2).

Definition 4.2 *We define the two precedence constraints in terms of predecessors and antecedents:*

(a) Event e is the (immediate) predecessor *of event e′ if (1) they are scheduled for the same logical process, and (2) timestamp* $V(e) <$ *timestamp* $V(e')$*, and (3) there is no other event e″ for the same logical process such that* $V(e) < V(e'') < V(e')$.

(b) Event e is the antecedent *of event e′ if the execution of event e causes the scheduling of event e′. Note that e and e′ may be scheduled for the same logical process.*

The space-time diagram describing the event precedences resulting from the process-oriented simulation is similar for sequential and parallel executions. As any discrete event simulation execution mechanism, sequential or parallel, must obey the causality constraint, the execution of the simulation application results in one unique and correct event order. An alternative view is that different execution mechanisms are different ways of "filling-in" the space-time diagram (Chandy and Sherman 1989). The various execution mechanisms differ in their strategy to accomplish the partial ordering of the event execution as defined by the precedence constraints. In the evaluation of the various parallel discrete event simulation protocols, we effectively determine the ability of the protocols to exploit the available parallelism in completing the space-time diagram as given for a particular simulation run. We can now use this space-time diagram to derive a protocol independent measure to quantify the effectiveness of the different protocols.

In the two dimensional space-time diagram, each event in the simulation, an independent sequential amount of work, is represented as a vertex (see Fig. 4.2). The two coordinates of each event consist of a spatial coordinate and a temporal coordinate. The spatial coordinate is the logical process LP_i where it is executed. Thus events placed on the same spatial position (vertically aligned in Fig. 4.2) occur in one logical process. The temporal coordinate is the simulation time at which the event occurs, the timestamp $V(e)$.

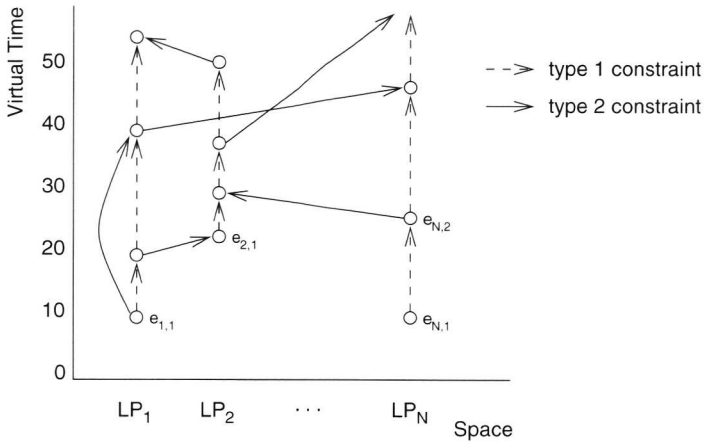


Figure 4.2: Space-time diagram depicting the events (vertices) with their dependency constraints (edges).

The precedence constraints formulated above, are represented by directed arcs. The dashed arcs in Fig. 4.2 represent the predecessor precedence constraint as defined in Def. 4.2(a): if two events e and e' are scheduled for the same logical process with timestamps $V(e)$ and $V(e')$ respectively, and $V(e) < V(e')$, then event e must be executed before e'. A continuous arc represents the exchange of an event message that must obey antecedent precedence constraint as defined in Def. 4.2(b): if event e causes the scheduling of event e', and consequently $V(e) < V(e')$, then event e must be executed before event e'. Notice that a process is allowed to send an event messages to itself.

There are some events in the simulation that have no predecessors; these events are called *initial* events (labeled $e_{1,1}$, $e_{2,1}$, and $e_{N,1}$ in Fig. 4.2). The events without antecedents are called *starting* events and are prescheduled before the execution of the simulation starts. The *terminal* events are the last scheduled events by their respective *LP*.

The resulting space-time diagram is a full description of the available parallelism in the discrete event simulation. Different analyses are possible, such as critical path computation, bottleneck analysis, etc. From these analyses, more high level parameters to characterize the parallelism can be obtained.

4.2.3 Critical Path Analysis

The first step in the determination of the average parallelism, parallelism profile, and shape vector, is the critical path analysis to determine the longest path in the space-time diagram. From the precedence constraints we construct a space-time diagram of the simulation run. The activities between the events are represented by the directed edges. We can label all activities in the space-time diagram with a weight $T(e_1, e_2)$, where e_1 is the event that preceded the activity, and e_2 is the event resulting from the activity. If e_1 is the immediate predecessor of e_2, then $T(e_1, e_2)$ expresses the service time for calculation; if e_1 is the antecedent of e_2, then it expresses the service time for communication. Note that in this scheme, the event vertices are *not* labeled with weights, but are rather instantaneous synchronization points in space-time.

By the construction of the space-time diagram with the weights along the edges, the hardware component of the system is implicitly modeled as an infinite number of identical processors, each of unit speed. The synchronization between processors has zero overhead and the entire parallel computer is devoted to one single task. With these assumptions, the hardware component is neutral to the critical path analysis such that the resulting metric is a characteristic of the software component. By mapping cost functions to the weighted graph, the available parallelism on specific hardware platforms can be obtained for a specific process to processor allocation. For example, the influence of communication costs can be studied, maybe depending on processor distance, or the consequences of oversubscribing of processes to processors, and hence load balance or imbalance.

The extended space-time diagram with associated weights to the edges, is an acyclic directed graph, or *program activity graph* (PAG), in which a longest weighted path can be found. This path is called the critical path, and its length is the minimal time required to complete the execution of the parallel simulation.

With all the edges of the space-time diagram labeled with a weight, we can associate a *critical time* with each event.

Definition 4.3 *The critical time of an event e,* crit(e)*, is defined by:*

$$\text{crit}(e) = \begin{cases} 0 & \text{iff } ANCE(e) = \emptyset \\ \max_{e' \in ANCE(e)} \{\text{crit}(e') + T(e', e)\} & \text{otherwise} \end{cases}$$

where the ancestor set is:

$$ANCE(e) = \{e' \in E \mid e' \text{ is predecessor or antecedent of } e\}$$

It is clear that crit(e) is the earliest time the event e can complete execution under the assumption that no dependencies are violated. Consequently, the largest value of crit(e) among all the events in the simulation run will give us the lower bound on the completion time of the simulation run. The method for finding the crit(e) is essentially a topological sort. That is, each crit(e) can be calculated after all the crit(e') of its ancestors have been computed. An algorithm to calculate the critical times in a PAG is presented in Alg. 4.1 in Section 4.3.2.

Having determined the critical time for each event in a start-finish sequence, the critical path can be defined in reverse order:

- e with $max_{e \in E}\{\text{crit}(e)\}$ is on the critical path;
- if e is on the critical path, then $e' \in ANCE(e)$, resulting in the maximal critical time according to Def. 4.3, is on the critical path.

This method underlies both the computation of the average parallelism metric and the path enumeration for bottleneck debugging, that is an optional analysis part of the APSE tool.

4.3 Design and Implementation of APSE

The Average Parallelism, Profile, and Shape Evaluation (APSE) methodology is designed and implemented as a stand-alone tool. This allows for larger flexibility, and makes the APSE tool generally applicable for the analysis of any parallel program (see the discussion in Section 4.6).

4.3.1 Conceptual Tool Structure

We designed the APSE tool following the conceptual framework outlined by McKerrow (1988). In this conceptual framework, the performance measurement tool can be comprised of four sections:

Sensor Section This is the interface between the target process(es) and the measurement tool. The function of a *sensor* or *probe* is to detect events of interest and/or measure the magnitude of the quantities to be monitored, and store this data in some internal buffer. When these buffers fill up, it may become necessary to write their contents to secondary storage.

A software probe is typically a subprogram or a procedure call inserted in the target process. In the case of a software tool, the sensor section can be seen as the tools front-end. Sensors can be *internally driven*, meaning that the target process triggers the sensor to some accounting action, or *externally driven*, meaning that the sensor is triggered by the tool rather than the target process. An important advantage of internally driven sensors is that the data they collect is synchronized to the internal operations of the target process.

Transformer Section The transformer section performs essentially two functions. First, the (typically huge amount of) data coming from the probes is reduced to a subset of relevant data. What data is and is not relevant depends on the scope and goal of the experiment at hand. The word *reduce* in the preceding sentence may be misleading, for the amount of data reduction is strongly dependent on the requirements of the subsequent sections of the tool. The transformer may just store the data without change.

The second function of the transformer is to rearrange the data coming from the probes. The reduced set of data is structured to fit the requirements of successive phases of the tool.

Analyzer Section The data stored by the transformer is processed to produce the final output of the experiment, e.g., tables, graphs, etc. The analysis to be performed on the condensed and structured sensor data is determined by the experimental framework. For simple experiments the analysis may be not needed and can than be skipped (see Fig. 4.3), for very complex experiments the analysis can be done only after the data of several experiments is available.

With respect to the analyzer section, we can further classify tools either as *rigid* or *flexible*. A rigid tool has limited and fixed analysis capabilities, which cannot easily be changed or extended. The methods of Lin (1992) for computing the most critical path in a program activity graph is an example of a rigid analysis. On the other hand, the performance measurement environments described by Mink et al. (1990), Mohr (1990), or Reed et al. (1991, 1994, 1998) provide a generic toolbox of analysis methods, from which a specific analyzer section can be assembled.

Indicator Section The function of the indicator is to show the results of the experiment in a convenient way. Depending on the amount of sensor data, data reduction and the character of the analysis, the indicator may be less or more complex. In general, the more information must be presented by the indicator, the more visualization is needed to produce comprehensible results.

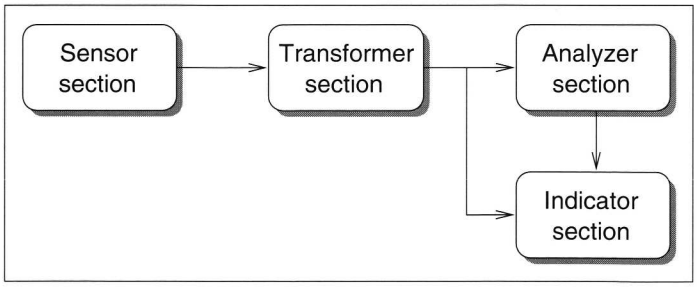


Figure 4.3: Conceptual structure of a measurement tool.

4.3.2 Overview of APSE

The overall design of the Average Parallelism, Profile and Shape Evaluation (APSE) tool is organized in according with the conceptual sections as described in the previous section. The functional structure is shown in Fig. 4.4.

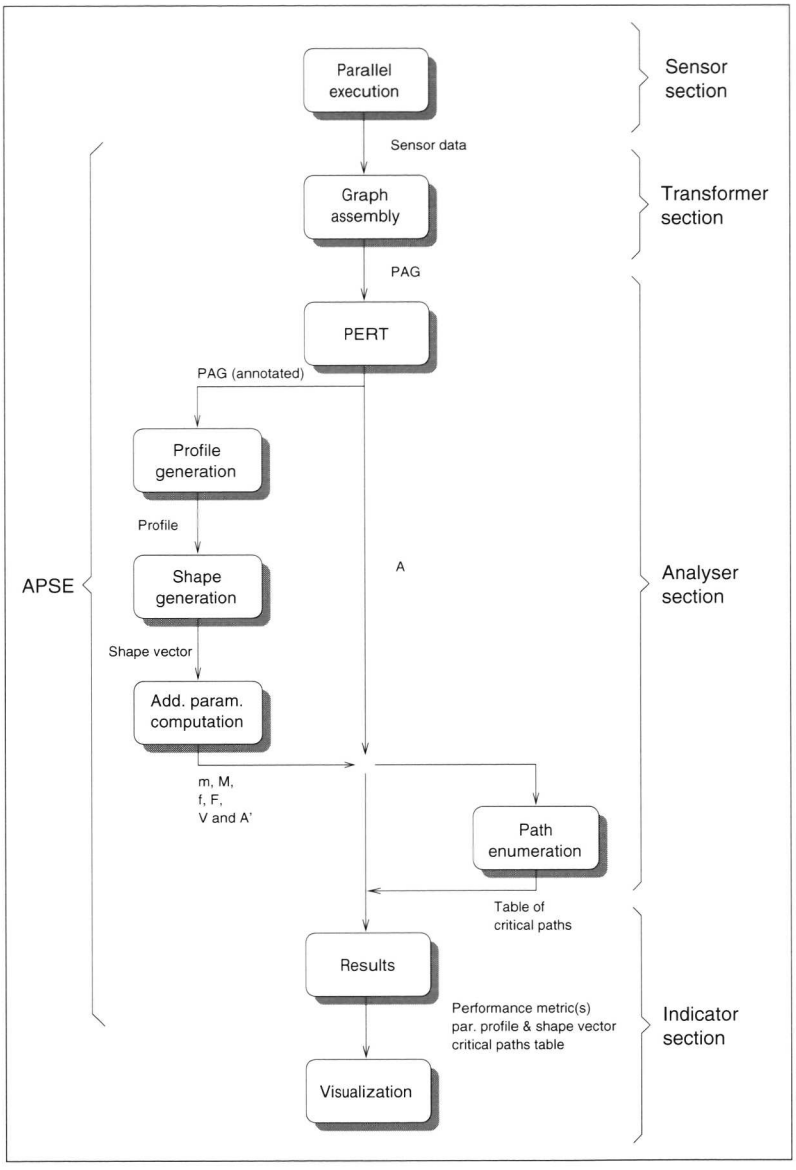


Figure 4.4: The functional structure of APSE (see for details, the text in Section 4.3.2).

The sensor section of the tool is located in a separate module and integrated with the parallel program. To keep interference (that is, program behavior perturbation due to monitoring code) as low as possible, we have isolated the recording of events from the process of analyzing them (see further on this

section). Thus, the average parallelism analysis is done postmortem, after the experimental run has finished.

The transformation section comprises the graph assembly module that constructs the program activity graph (PAG) from the sensor data. The analyzer section applies the Program Evaluation and Review Technique (PERT) algorithm to the PAG. Depending on user requests, the average parallelism metric is computed and optionally the profile and shape of the parallelism is generated. For performance bottleneck debugging, it is also possible to include the generation of a table of the most critical paths in the analysis. The indicator section includes the modules that present the results and provides an interface to visualization tools. The visualization of complex data sets such as the parallelism profile or the critical path in the PAG is not considered to be a part of APSE. However, an interface to Gnuplot is provided and more elaborate visualization with for example Tk/Tcl can be easily incorporated.

Sensor Section and Software Probes

The APSE interface with the experimental environment, in this discussion the APSIS simulation environment, is implemented by the software probes. The software probes are inserted to the Time Warp kernel (see Fig. 4.5); it is up to the experimentalist to identify the relevant trace events within the Time Warp kernel. In our study, the relevant trace events are: start and finish of a simulation event execution and the scheduling of a new simulation event. Although the number of software probes is very limited (currently six), it allows us to construct the PAG for further analysis.

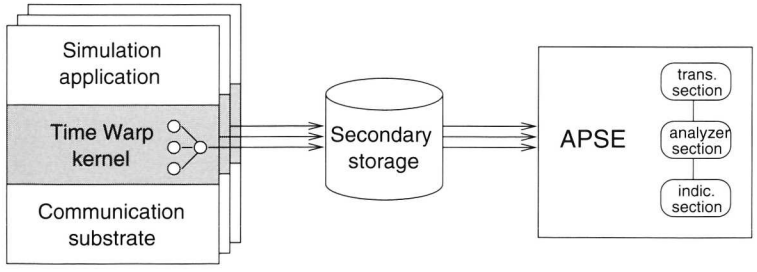


Figure 4.5: The sensor section of the APSE tool is integrated with the APSIS Time Warp simulation kernel. The trace data is written to the file system, and is analyzed off-line by the APSE tool.

The trace data gathered by the software probes is stored in a circular buffer data structure in main memory. If the circular buffer has reached its capacity, the trace data is written in a binary form to the file system. The interference of the periodic file system I/O to the program behavior and consequently the trace data is limited, as all the Time Warp protocol overhead including APSE buffer management is outside the traced program area. That is, only the committed

events in the simulation application are traced and recorded, and all erroneous events that are rolled back, and other overheads induced by the Time Warp protocol, are excluded from the trace data. This also implies that the trace data has to be committed in a similar way as events have to be committed. Only if the GVT swept past the events, the associated trace data is allowed to be written to the file system.

PAG Generation

The graph assembly module constructs the PAG from the trace event records in the input. This graph (see Fig. 4.2) contains two types of dependences among the events: intra-process and inter-process dependences, as described in Section 4.2.2. Every event record from the execution trace is converted into a PAG vertex, and the causality constraints are the edges in the graph. The graph is completed by adding two special vertices: the *source* (denoted by s) and the *sink* (denoted by t). The source is connected to all initial events and the sink is connected to all terminal events.

PERT Algorithm

The program activity graph is annotated by the Program Evaluation and Review Technique (PERT) algorithm. This algorithm annotates each vertex in the graph with its *maximal delay to sink* value. The original PERT algorithm computes for all vertices the *maximal delay from source* value rather than the maximal delay to sink, similar to the definition of critical time of an event (Def. 4.3). However, the path enumeration algorithm, as discussed in the next section, assumes the graph to be annotated with maximal delay to sink values. Since the objective is not to obtain the critical time for a specific event, but to look for the critical path, the choice is arbitrary.

Let crit(e) denote the "max-delay-to-sink" label of event e, and let $SUCC(e)$ be the set of successor events of e,

$$SUCC(e) = \{e' \in E \mid e' \text{ has } e \text{ as predecessor or antecedent}\}.$$

Furthermore, assume that each edge between e and e' in the PAG has a weight (or length) $T(e, e')$.

The PERT algorithm in Alg. 4.1 reflects a topological sort on the graph, since the sink is the start vertex, whose maximal delay to sink value is zero, and for each vertex in the graph that is not yet annotated, the maximal delay to sink value depends both on this value for all its successors, and on the weights of the connecting edges. Upon completion of the PERT algorithm, the maximal delay to sink value of the source vertex represents the weight of the most critical path in the graph.

By the construction of the PAG, the PERT algorithm computes the "max-delay-to-sink" time of the events while it adheres to the precedence constraints as defined in Section 4.2.2. If the user requested that a path enumeration be part of the analysis, the successors of each vertex are sorted in non-decreasing

maximal delay to sink value. This sorting is necessary in order to apply the path enumeration algorithm later on.

Algorithm 4.1 The PERT algorithm: critical time computation.

```
{s = source; t = sink;}
∀e ∈ E − {t} : crit(e) := undefined; {initialization}
crit(t) := 0;
repeat {topological sort}
  for all e ∈ E do
    if crit(e) = undefined ∧ ∀e' ∈ SUCC(e) : crit(e') = defined then
      crit(e) := max_{e'∈SUCC(e)}{crit(e') + T(e, e')};
    end if
  end for
until e = s;
{crit(s) represents the length of the most critical path}
```

Path Enumeration Algorithm

The critical path enumeration algorithm used in APSE finds the K most critical paths in the annotated PERT PAG (Yen et al. 1989). Path extracting algorithms are a very important part of parallel performance bottleneck debugging. The path enumeration algorithm is conceptually a K iterative process. In the ith iteration, the ith most critical path is expanded in the graph. The algorithm as presented in Alg. 4.2 uses a table (*PATHS*) to store the most critical paths, a dynamic threshold (T) to be able to prune paths that will not reside in the paths table, and an overloaded function nextnode:

1. nextnode(e): returns the first vertex in the sorted successor list of e;
2. nextnode(e, e'): returns nil, when e' is the last vertex in the successor list of e, or returns the vertex next to e' in the sorted successor list of e otherwise.

Let crit'(e) denote the "delay-from-source" label of vertex e. This value is easily computed for each element of any (partial) path, since it is just the sum of all the weights associated with the edges leading to the element. Therefore, we have left out the computations of the crit'(e) in the pseudo-code.

Each vertex has its successors sorted in non-increasing maximal delay to sink value, hence the most critical path can be found by starting at the source, and always taking the first successor in each vertex successor list, until the sink is reached.

Path enumeration is realized by creating variants of the most critical path. Such a variant is realized by making a copy of the current path, and then applying a process of tracing backward and forward on the copy. A new critical path is completed when the forward trace arrives at the sink.

The backward trace starts at the sink and traces the path in the direction of the source, until it arrives at a vertex where an alternative route can be

taken. Comparing the sum of the two delays crit'(e) and crit(e') and the weight $T(e, e')$ with the threshold provides the answer whether the alternative path is suitable. The backward trace has two termination conditions:

1. a vertex has been found in the graph from which a variant path can be followed;

2. no vertex has been found, and the algorithm terminates—this happens when the backward trace has proceeded all the way to the source.

The forward trace consists of following the first successor of each vertex every time, until the path is complete again (i.e., the last vertex of the path equals the sink vertex). The new path that is created this way is inserted in the paths table.

Algorithm 4.2 The path enumeration algorithm.

$T := 0$; {initialize threshold}
$P := (s = e_0, e_1, e_2, \ldots, e_q = t)$, where $e_{i+1} = \text{nextnode}(e_i)$;
$j := q - 1$;
while $j > 0$ **do**
  insert P to *PATHS*;
  $T = \min_{P \in PATHS}\{\text{crit}(P)\}$;

  {backward trace}
  find largest $j, 0 \leq j < q$ such that:
    $\text{nextnode}(e_j, e_{j+1}) \neq \text{nil}$ and
    $\text{crit}'(e_j) + T(e_j, e_k) + \text{crit}(e_k) > T$, where $e_k = \text{nextnode}(e_j, e_{j+1})$;

  {forward trace}
  $P := (s = e_0, e_1, e_2, \ldots, e_q = t)$, where e_i is:
    $0 \leq i \leq j : e_i$ as in the existing P,
    $i = j + 1 : \text{nextnode}(e_j, e_{j+1})$,
    $i > j + 1 : \text{nextnode}(e_{i-1})$;
end while
{*PATHS* contains the K most critical paths}

Profile and Shape Generation

The parallelism profile is a plot of the degree of parallelism versus time. The profile can be generated by rearranging and reducing the information from the PAG. For each activity in the PAG, i.e., an intra-process dependency edge, two entries are inserted in the profile list: one for the activation of the activity and one for its termination. When the list is sorted on times of activation and termination, the parallelism profile can be obtained by counting activation and termination events: parallelism increases with an activation event and

decreases with a termination event. From this profile the shape vector is generated by associating the normalized fraction of execution time spent with a certain degree of parallelism in a histogram. Given the notation in Def. 4.2.1, we can compute the following metrics from the shape vector:

Minimum parallelism $m = \min\{i \mid p_i \neq 0\}$

Maximum parallelism $M = \max\{i \mid p_i \neq 0\}$

Fraction sequential $f = p_1$

Average parallelism $A = \sum_{j=m}^{M} j \cdot p_j$

Variance in parallelism $V = \sum_{j=m}^{M} j^2 \cdot p_j - \left(\sum_{j=m}^{M} j \cdot p_j\right)^2$

The advantage of the presentation of the execution trace as a parallelism profile over a PAG comes from the compactness of the representation and its resulting in more comprehensible results. For huge execution traces even visualization will not suffice to provide insight into the trace structure if the PAG is considered. This is due to both the amount of information contained in the PAG, and the complexity of the structure of the PAG. Actually, the parallelism profile can be used complementary to the PAG in order to identify performance bottlenecks. Periods with a relative small degree of parallelism are readily recognized and direct the detailed study of the complex PAG to areas where performance bottlenecks appear.

4.4 Experiments, Validation, and Assessment

In this section we present some experiments to validate the correctness of the APSE critical path analysis, and to assess the use of critical path analysis in average parallelism evaluation. The validation and assessment is shown by two simple simulations: unidirectional and bidirectional message routing over a ring embedded in a two-dimensional torus topology. In Section 5.5 the APSE analysis will be applied to a complex problem, namely the Ising spin simulation.

4.4.1 Unidirectional Ring

The unidirectional ring simulation is realized using the APSIS environment. To generate an event trace of the parallel simulation, an instrumented version of the Time Warp simulation library is made available (see also Fig. 4.5). The event trace of a logical process is buffered in memory, and during the fossil collection phase of committed events, the buffers are flushed to the file system. In this respect, the event trace represents the sequence of correct events that are executed in causal order, thus the event in the event trace *must* be committed before they are written to file.

The embedding of the ring into the two-dimensional torus is shown in Fig. 4.6. From the starting LP 0, the event messages are routed to their east neighbor for odd rows in the torus, and to their west neighbor for even rows in the torus. If the message hits one of the east or west boundary LPs, it is routed to the south neighbor. Note that if the message arrives at the south-west corner LP, the message is routed south to LP 0, which completes one single round through the ring. The unidirectional ring simulation routes at every instance of time *one single message* through the embedded ring. A parameter determines the number of times the message is routed through the ring. As at each instance only one single message is present in the system, and hence only one LP can be active, we expect an average parallelism of one.

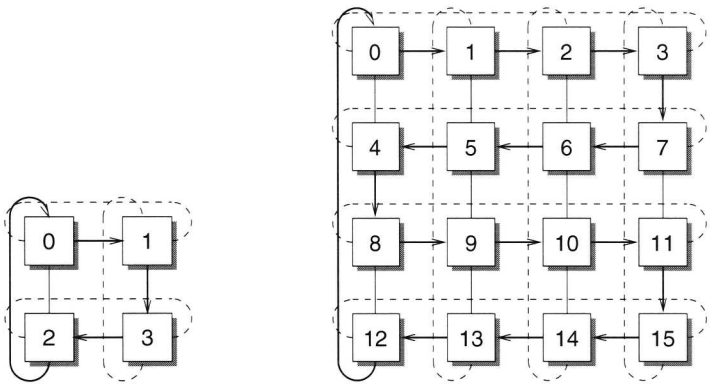


Figure 4.6: Ring mapped on two-dimensional torus topology.

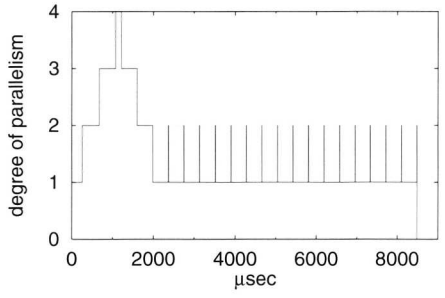
The experiments are executed on the Distributed ASCI Supercomputer (DAS)*, a 200-node parallel platform composed of four distributed clusters connected by ATM (Vetter 1995), and where the nodes within each cluster are connected by a Myrinet System Area Network (SAN) (Boden et al. 1995). All experiments are performed on one single cluster.

The results of the parallel simulation on four processors of the unidirectional ring routing of five messages are shown in Table 4.1 and Fig. 4.7. From Table 4.1 we can observe that although we expect an average parallelism $A = 1$, we have measured an average parallelism $A = 1.34$. Similarly remarkable is the fraction sequential $f = 0.79$ and fraction maximum parallelism $F = 0.02$. This can be explained by Fig. 4.7(a), the parallelism profile, that shows a transient behavior in the degree of parallelism during the initialization phase of the simulation. The initialization of the LPs, not the initialization of the simulation library, is also accounted for in the event trace. The initialization is not dependent of any event and occurs in parallel. After real time 2000 μsec the parallelism profile in Fig. 4.7(a) shows a degree of parallelism of 1, as expected.

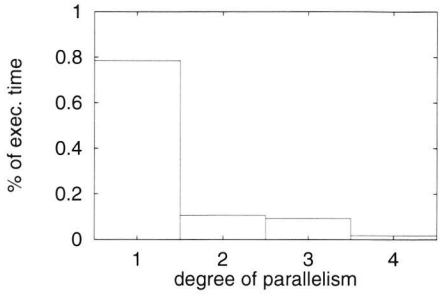
*http://www.asci.tudelft.nl/das/das.shtml or http://www.cs.vu.nl/das/

$A =$	1.34	$f =$	0.79
$m =$	1	$F =$	0.02
$M =$	4	$\sigma^2 =$	0.51

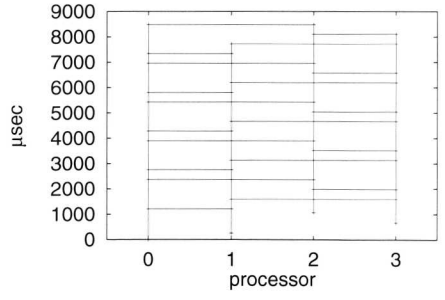
Table 4.1: Average A, minimum m, and maximum parallelism M, fraction sequential f, fraction maximum parallelism F, and variance in average parallelism σ^2 for simulation of five messages over unidirectional ring on four processors.



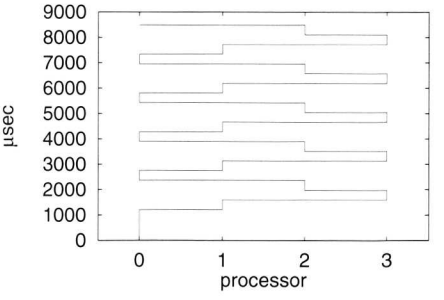
(a) Parallelism profile of unidirectional ring simulation.



(b) Shape of unidirectional ring simulation.



(c) Program activity graph of unidirectional ring simulation.



(d) Critical path of unidirectional ring simulation.

Figure 4.7: The parallelism profile, the shape, the program activity graph, and the critical path of the parallel simulation of the unidirectional ring on four processors.

The program activity graph in Fig. 4.7(c) depicts the activities and intra- and inter-dependencies between the events that trigger the activities. From Fig. 4.7(c) one can see how the event message travels from LP 0 to LP 1, LP 3,

LP 2, and back to LP 0. The y-axis in the figure shows the progress in real time, i.e., the real time accounted for the activity by the LP. The critical path in Fig. 4.7(d) shows the activities and the intra- and inter-dependencies that are along the critical path in the parallel simulation. The figure shows clearly how the critical path follows the message routing through the torus. As there is only one event message in the system, the critical path should follow the event message through the parallel simulation.

In Table 4.2 and Fig. 4.8 results are presented of the parallel simulation on sixteen processors of unidirectional ring routing of 100 messages. The average parallelism and fraction sequential in Table 4.2 are (almost) reflecting the sequential behavior of the simulation. The average parallelism $A = 1.04$ and the fraction sequential $f = 0.98$. The effects of the transient behavior in the degree of parallelism are nearly nullified by the long simulation run, see also the small peak around 0 in Fig. 4.8(a). In Table 4.2 the maximum parallelism, M, of five instead of sixteen can be explained by the gauging of the initialization activities that are relative to the first event message exchange (send or receive). Depending the duration of the initialization activity and the event inter-dependencies, the activities will overlap with each other in real time and hence accounts for the degree of parallelism. This effect can be seen in Fig. 4.7(c).

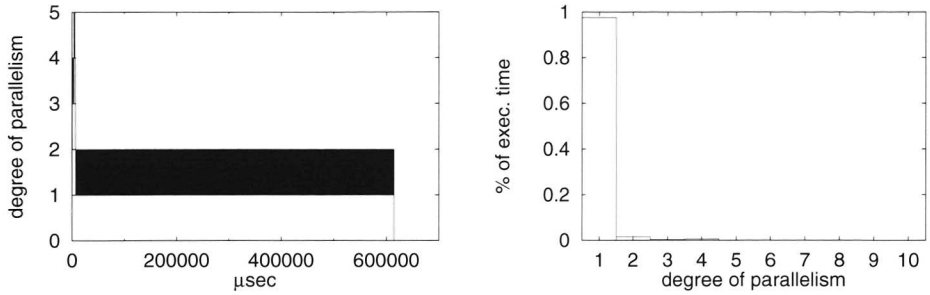
$A =$	1.04	$f =$	0.98
$m =$	1	$F =$	$0.45 \cdot 10^{-3}$
$M =$	5	$\sigma^2 =$	0.9

Table 4.2: Average, minimum, and maximum parallelism, fraction sequential, fraction maximum parallelism, and variance in average parallelism for simulation of 100 messages over unidirectional ring on sixteen processors.

4.4.2 Bidirectional Ring

The framework of the bidirectional ring experiment is similar to the unidirectional ring simulation. The bidirectional ring through which messages are routed is embedded into a two-dimensional torus, as shown in Fig. 4.6. The bidirectional ring differs from the unidirectional ring in that the initiating LP 0 routes two messages to its neighbor: one message to its east neighbor, and one message to its west neighbor. On arrival of a message, it is forwarded along the same direction as it was received from, to the neighboring LP. At every instance the bidirectional ring simulation routes two messages through the embedded ring, and hence we expect an average parallelism of two.

The results of the APSE analysis of the parallel simulation on four processors of the bidirectional routing of five messages in both directions are shown in Table 4.3 and Fig. 4.9. The average parallelism $A = 2.30$ is larger than the expected $A = 2.0$, but again this can be explained by the transient behavior of the degree of parallelism during initialization of the LPs, see also Fig. 4.9(a).



(a) Parallelism profile of unidirectional ring simulation.

(b) Shape of unidirectional ring simulation.

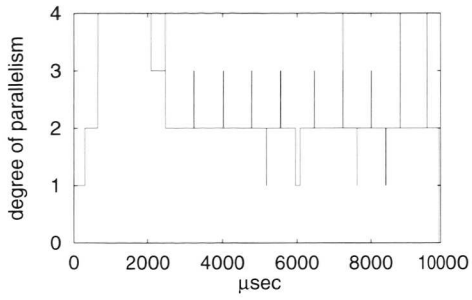
Figure 4.8: The parallelism profile and the shape of the parallel simulation of the unidirectional ring on sixteen processors.

The shape of the bidirectional ring in Fig. 4.9(b) shows a peak at degree of parallelism of two, but shows also a large percentage of execution time a degree of parallelism of four. This is due to the initialization phase.

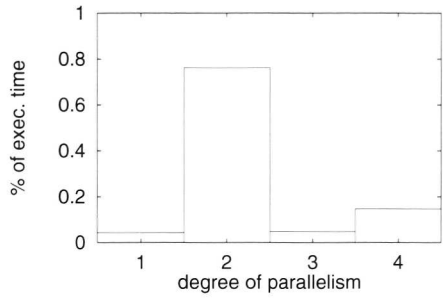
$A =$	2.30	$f =$	0.04
$m =$	1	$F =$	0.15
$M =$	4	$\sigma^2 =$	0.59

Table 4.3: Average, minimum, and maximum parallelism, fraction sequential, fraction maximum parallelism, and variance in average parallelism for simulation of five messages over bidirectional ring on four processors.

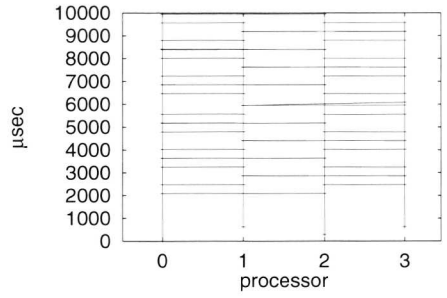
Interesting to note is the relatively large period of degree of parallelism of one at execution time 6000 μsec. If two messages pass each other halfway on their round through the bidirectional ring, the simulation of the two messages are sequentialized. This is enforced by the timestamps of the event messages. The forwarding activity of the LPs takes approximately the same amount of time, as can be seen in Fig 4.9(c) by the vertical segments between message departure and arrival. However, due to some perturbation in the activity execution time (due to underlying operating system activities, virtual memory management, etc.) one of the two messages lags behind. When the event messages meet each other halfway, the first message must wait for the second to arrive before it can be forwarded. The simulation of the lagging event message accounts for the degree of parallelism of one for a period. Note also the short period of inactivity of LP 3 in Fig 4.9(c) at 6000 μsec, and how the send and receive times differ in the figure to compensate for the difference in activity execution times. In the ideal case the send and receive times overlap with each



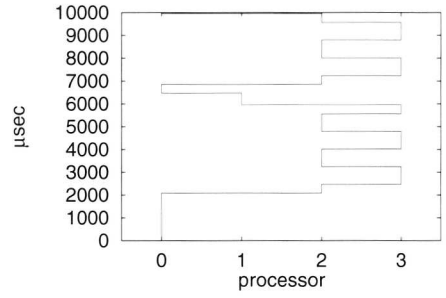
(a) Parallelism profile of bidirectional ring simulation.



(b) Shape of bidirectional ring simulation.



(c) Program activity graph of bidirectional ring simulation.



(d) Critical path of bidirectional ring simulation.

Figure 4.9: The parallelism profile, the shape, the program activity graph, and the critical path of the parallel simulation of the bidirectional ring on four processors.

other as each event activity execution takes the same amount of real time.

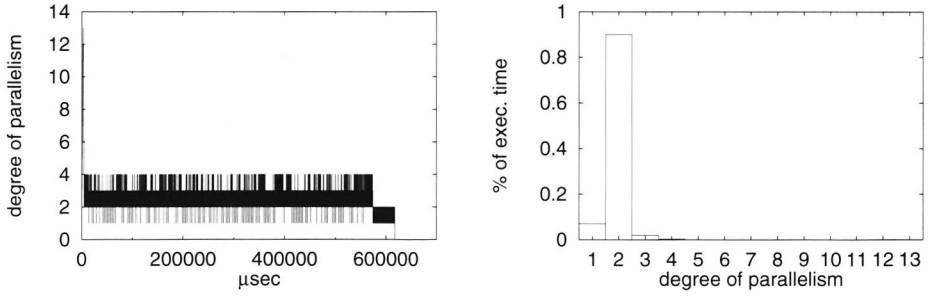
Figure 4.9(d) depicts the critical path through the bidirectional ring simulation. In the bidirectional ring simulation the critical path does not necessarily follow one single event message, but can switch between the event messages. See for example how the critical path switches between LP 2 and LP 3 during execution time interval $[2000, 6000]\,\mu$sec.

The results for the parallel simulation on sixteen processors of the bidirectional ring routing of 100 messages are presented in Table 4.4 and Fig. 4.10. The average parallelism $A = 2.0$ is in correspondence with the theoretical expected value. The maximum degree of parallelism is thirteen, which can be explained in a similar way as for the unidirectional ring study, i.e., gauge effect.

$A =$	2.0	$f =$	0.07
$m =$	1	$F =$	$0.9 \cdot 10^{-4}$
$M =$	13	$\sigma^2 =$	0.22

Table 4.4: Average, minimum, and maximum parallelism, fraction sequential, fraction maximum parallelism, and variance in average parallelism for simulation of 100 messages over bidirectional ring on sixteen processors.

The parallelism profile in Fig 4.10(a) shows how after the short initialization phase, the degree of parallelism is two with many spikes to three and four due to small perturbations in the event activity execution time. Although the parallelism profile shows almost black areas between degree of parallelism of two and four, the contribution to the average parallelism is small as can be seen in Fig. 4.10(b), where the degree of parallelism of three is 2% and degree of parallelism of four is 0.3% of the execution time. The relative large period of degree of parallelism of one at the end of the execution is resulting from the finalization phase.



(a) Parallelism profile of bidirectional ring simulation.

(b) Shape of bidirectional ring simulation.

Figure 4.10: The parallelism profile and the shape of the parallel simulation of the bidirectional ring on sixteen processors.

4.5 Related Work

Critical path analysis has shown to be a comprehensive method to quantify the inherent parallelism or average parallelism of a parallel program. The critical path analysis technique, or more generally event tracing, is incorporated in a wide range of performance evaluation environments such as Pablo (Reed et al. 1993) and Paradyn (Miller et al. 1995). The basic critical path analysis of the

event trace does not differ for the various analysis tools, but the manner to incorporate the analysis with the parallel program execution does. Event tracing is not limited to simulation application. Any program can define "events" such as a function call, interrupt, or trap to a kernel, and record this event with a timestamp (in this case the real time). Although the APSE performance analysis is presented in the context of parallel discrete event simulation, any parallel program can be instrumented for performance analysis with APSE (Overeinder and Sloot 1995).

With respect to the specific parallel simulation instrumentation for event trace generation and critical path analysis, we can distinguish between (i) sequential versus parallel execution event traces, and (ii) on-line versus off-line critical path analysis. Sequential execution event trace generation supports early performance evaluation of the potential parallelism available in the simulation, even before a parallel program is implemented. On the other hand, parallel execution event trace generation allows for the computation of a more realistic bound that takes into account all the overheads associated with the parallel simulation of the model, except those specific to different simulation protocols. On-line versus off-line critical path analysis is a practical trade-off (Hollingsworth 1998). As the computation of the critical path is expensive, off-line (post mortem) approaches minimize the intrusiveness of the analysis onto the execution behavior. However, the off-line approach requires memory and disk space proportional to the number of events and communication operations performed. Therefore, to make critical path analysis practical for long running simulations, on-line analysis is necessary.

Berry and Jefferson (1985), and later Salmi et al. (1994), presented an off-line method for critical path analysis of sequential event traces. The inherent parallelism computation is included in the modeling process of the (sequential) simulation to assess the parallel potential. The APSE critical path analysis is similar, but within the APSIS environment, the event trace is collected from the parallel simulation execution. The analysis approach of Jha and Bagrodia (1996) is orthogonal to the previous ones. They describe an Ideal Simulation Protocol (ISP), based on the concept of critical path, which experimentally computes the best possible execution time for a simulation model on a given parallel architecture. The ISP requires an event trace from a sequential execution to eliminate protocol specific overheads, that is, the ISP knows the identity of the next message it is going to execute, thus no rollback or unnecessary blocking.

On-line critical path analysis has been used in a number of parallel simulators. Livny (1985) incorporated an on-line critical path algorithm in the Distributed System Simulation (DISS)[†] simulator. In the study it is assumed that all events have the same execution time, which is defined to be the unit time. During the distributed simulation, the global optimal execution instance of an event is computed, and the total events that have been executed is counted. The inherent parallelism is than defined as the ratio between the total events executed and the optimal execution instance of the last event in the simula-

[†]DISS is a predecessor of HLA

tion. Lin (1992) proposed a critical path analysis algorithm that is integrated with the sequential simulation. The method described by Lin is similar to the algorithm proposed by Livny, but the algorithm of Lin can be used to study load balancing under different event scheduling policies. These results are extended by Wong et al. (1995) by considering both the inherent parallelism and the parallel simulation protocol overhead. Lim et al. (1999) developed a set of performance prediction tools, including a critical path analyzer. In their approach the sequential simulator informs the analyzer about the execution time, whereupon the analyzer models the progress of the parallel execution.

Jefferson and Reiher (1991) and Srinivasan and Reynolds (1995) reported about super-critical speedup of optimistic simulation protocols. Although critical path analysis establishes a lower bound on the completion times of parallel discrete event simulations, at least one optimistic protocol can complete in less than the critical path time in a nontrivial way. For example Time Warp with lazy cancellation can achieve super-critical speedup, although this is of more theoretical than practical interest. The parallel simulation using Time Warp with lazy rollback might include erroneous causal execution of events which are accepted as the incorrect execution order does not influence the correct result of the simulation. This incorrect execution order can potentially be shorter than the (correct) critical path through the simulation, and hence beating the critical path time.

Other techniques than critical path analysis are also used in performance evaluation of parallel simulations. Ferscha and Johnson (1996) present an incremental code development process that supports early performance of Time Warp protocols and several of its optimizations. The set of tools represent a test bed for a detailed sensitivity analysis of the various Time Warp execution parameters. The performance engineering activities range from performance prediction in the early development stages, to measurements of performance metrics of the preliminary or final program. Liu et al. (1999) propose a Scalable Simulation Framework (SSF) to predict the performance of a given model, using given features of the simulator, without having to run, or even build, the model. Balakrishnan et al. (1997) present a framework for performance analysis of parallel discrete event simulators which is based on a Workload Specification Language (WSL). WSL is a language that allows the characterization of simulation models using a set of fundamental performance-critical parameters. The WSL presentation can be translated (using a simulator-specific translator) to different simulation back-ends. The ultimate goal of this project is to provide a standard benchmark suite that studies the performance space of the simulators using realistic models.

4.6 Summary and Discussion

The average parallelism analysis and the associated parallelism characterization obtained from the analysis, such as parallelism profile, shape, and critical path, provides a deeper understanding of the behavior of parallel programs in

general, and parallel simulations in particular. The APSE framework is an environment independent, stand-alone parallelism analysis workbench, which analyzes program traces generated by instrumented message passing libraries or simulation libraries. The instrumentation of a library is fairly simple as special APSE monitor functions are provided to record events into a buffer and to flush the buffer to the file system. For example, for the instrumentation of the APSIS Time Warp simulation library, we had to add four appropriate calls at three places: one place to record the start of an event, one place to record the finish of an event, and finally at one place to omit rollback activities from the event trace.

The APSE analysis framework has been applied to two fairly simple simulation problems: unidirectional ring simulation with one message, and bidirectional simulation with two messages. For long running simulations, the measured average parallelism was in correspondence with the theoretically expected value. For short runs the average parallelism was higher due to the transient behavior of the degree of parallelism during LP initialization that dominates the results. The critical path figure of the unidirectional ring simulation depicts how the critical times of the events in the parallel simulation depend on the single event message that is routed through the system. For the bidirectional ring simulation we see that in the general case where a number of events are executed in parallel, the critical path meanders through the program activity graph and not necessarily follows one single activity through the system.

The inherent parallelism made available in the software system should not be a goal itself. Given two different implementations of one simulation application, the execution of these simulations can result in different inherent parallelism measures. This fact does not imply that the implementation with a higher degree of inherent parallelism will perform better than the implementation with a lower degree of parallelism. Independently of the ability of the PDES protocol to exploit the available parallelism, the parallel architecture must be able to bring this parallelism to expression. An important characterization of the available parallelism is the grain size or granularity of the parallelism: the amount of computation involved between two synchronization points. For example, if a high degree of parallelism indicates a fine grain level of parallelism, it is difficult for a distributed memory parallel architecture to achieve proper speedup figures. On the other hand, the implementation with a lower degree, but coarse grain, parallelism can behave well on a distributed memory parallel architecture and outperform the first implementation. Resuming, the performance measures made to evaluate the effectiveness of the PDES protocol are also influenced by the granularity of the parallelism in relation to the parallel architecture. It is also the responsibility of the parallel program designer to tailor the granularity of the parallelism to the architecture.

The parallelism evaluation by APSE is also applicable to parallel programs in general. For example, by instrumentation of communication libraries such as MPI or PVM, the parallel application generates program traces that can

be analyzed by the APSE tool. For example, the APSE parallelism evaluation has been successfully applied in a study of parallel sorting algorithms (van den Brink 1997).