Performance Evaluation for Parallel Systems: A Survey

Lei Hu and Ian Gorton
Department of Computer Systems
School of Computer Science and Engineering
University of NSW, Sydney 2052, Australia
E-mail: {lei, iango}@cse.unsw.edu.au

UNSW-CSE-TR-9707 — October 1997

Abstract

Performance is often a key factor in determining the success of a parallel software system. Performance evaluation techniques can be classified into three categories: measurement, analytical modeling, and simulation. Each of them has several types. For example, measurement has software, hardware, and hybrid; simulation has discrete event, trace/execution driven, Monte Carlo; and analytical modeling has queueing network, Petri net, etc.. This paper systematically reviews various techniques, and surveys work done in each category. Also addressed and discussed are other issues related to performance evaluation. These issues include how to select metrics and proper techniques that are well suited for the particular development stage, how to construct a good model, and how to perform workload characterization. We also present fundamental laws and scalability analysis techniques. While many techniques discussed are common in both sequential and parallel system performance evaluation, our focus is on the parallel systems.

1 Introduction

Computer systems become more and more powerful. Many manufacturers design and implement various architectures ranging from microcomputers to supercomputers according to different application demands. Parallel computer architectures arise for many reasons. One of them is probably the limits imposed by the laws of physics on the development of semiconductor technology (see for example [Liu82]). Applications are getting more and more sophisticated: some problems must be solved within a limited amount of time. Weather forecasting is such a time-critical example. This forces computer scientists to seek innovative architectures and exploit parallelism in order to meet the demands of high speed computing.

Different systems have different architectures. Based on notions of instruction stream and data stream, Flynn [Flynn72] classified various computer architectures into four categories: SISD, SIMD, MIMD, and MISD. Of the four machine models, MIMD is most widely used for constructing machines for general-purpose computations. MIMD parallel computers can further be classified into share-memory multiprocessors and message-passing multicomputers.

With these different architectures, a problem that arises immediately is how to choose a system for a particular problem. Selecting a proper architecture for an application is problem oriented. One architecture that suits one kind of problems may not at all suit another. After a particular architecture is chosen, the following questions may be asked. How will the system perform? What criteria should be used to evaluate the performance? What techniques can/should we use to get the performance values? To answer these questions is the objective of performance evaluation.

According to [Ferrari86], performance evaluation dates back to 1965, when Alan Scherr submitted his Ph.D. thesis. Since then, great progress has been made in this area. It has become a distinct discipline independent of others, such as computer architecture, system organization, operating system, and so forth. Many textbooks are devoted to this subject (see for example [Drummond73; Sauer81; Jain91]).

Performance evaluation can be defined as assigning quantitative values to the indices of the performance of the system under study. So, what is performance?

To answer this question is not easy, for performance involves many aspects. Here, we do not mean to give our definition, for many authors have given different definitions from different perspectives (e.g., [Doherty70; Graham73]. In what follows, we try to list some factors that must be considered: *functionality*, *reliability*, *speed*, and *economicity*. First, functionality is the most important. Any successful system must do what its designer wants it to do. If the system cannot meet this basic demand, it is meaningless to talk about any other things. Second, a system must be as reliable as possible. Given a service request to a system, there exist two possible outcomes: the system services the request correctly or incorrectly. If it fails to service correctly, the probability that error occurs should be studied. If the probability is too high and cannot meet the design specification, the system should be redesigned or improved until the error probability limits to a reasonable level. Third, if a system can service the requests made to it correctly, how fast and efficient it completes the work becomes important (the speed factor). Because functionality and reliability are two basic issues that are often considered and solved by any system designer at a very early stage of the design life-cycle, speed becomes the most frequent research topic in the performance evaluation community. Speed is often reflected by response time and throughput rate; and efficiency is often reflected by utilization. Finally, a system is always designed and implemented to give its services at a given cost. Given a specification, how to design and implement the system at the lowest cost is what economicity should consider.

To evaluate the performance of a system or to compare two or more systems, one must first choose some criteria. These criteria are called *metrics*. Different metrics may result in totally different performance values. Hence, selecting proper metrics to fairly evaluate the performance of a system is difficult [Jain91]. To know metrics, their relationships and their effects on performance parameters is the first step in performance studies. Next, selecting proper *workload* is almost equally important. A system is

often designed to work in a particular environment with some workload. It is not proper to study the performance without considering the workload.

After choosing proper metrics and workload, one must consider what technique or techniques should be used. Generally, there are three techniques commonly used in performance evaluation. They are *measurement*, *simulation* and *analytical modeling*. All these techniques play equally important roles in performance studies. Each technique has its own advantages and disadvantages. The precision of the results obtained by each technique often varies from one technique to another. It is not fair to say that one is better than another. To select a technique or techniques to evaluate a system, many factors must be taken into consideration. One must have a clear idea that what goal he/she wants to achieve and at what stage the technique would be used. For example, at the early design stage when the system has not yet been constructed, measurement is obviously impossible, instead, a simple analytical model is practical. As the design process goes on, more and more details about the system are obtained. At this stage, simulation or more sophisticated analytical modeling techniques could be used. Finally, when the system design has been completed and a real system constructed, measuring becomes possible. Quite often, to make the evaluation results more convincing to the system's user, these techniques can be used together.

In the literature, there have appeared a number of excellent survey papers which review the work in a particular aspect discussed above. For example, [Plattner81] and [Power83] are program execution monitor survey papers. Plattner and Nievergelt survey the development of program execution monitors, and discuss basic concepts and requirements, design considerations for monitoring languages, and implementation and timing aspects. Power focuses on program execution monitoring tools. He discusses the design issues and a wide variety of techniques used by existing monitors. In addition, a new program execution monitor is presented, and the design and use of this new monitor are discussed. [Calzarossa93] discusses several methodologies and techniques for constructing workload models in terms of different system architectures, such as centralized systems, network-based systems and multiprocessor systems. [Trivedi94] addresses the analytical modeling techniques including Markov reward models, stochastic Petri net models, and hierarchical and approximate models. Various tools designed to solve these models for performance evaluation studies are also presented and described.

In this paper, we intend to address all the issues that a performance analyst must consider, review and compare in some detail typical techniques applied to performance evaluation, and survey some recent work done in this area. Although many techniques are common in both sequential and parallel system performance evaluation, our focus is on the parallel systems.

The paper is organized as follows. In section 2, we discuss some general issues that must be considered first in performance studies. These issues include what should be considered in modeling, and how to choose proper metrics and appropriate techniques. We also describe advantages and disadvantages of each technique in terms of different conditions the techniques are used. In section 3, we describe several well-known laws such as speedup, isoefficiency, scalability, and so on. These laws are often used in direct analysis of performance. Some commonly used metrics are also introduced. Section 4 describes workload characterization. Mainly, several statistical analysis techniques are introduced; their use in recent workload characterization research is also reviewed. Subsequent sections, sections 5, 6, 7, are dedicated to three different techniques in performance evaluation studies. They are *measurement*, *analytical modeling*, and *simulation*. Each of these sections are further divided into several subsections according to different types of each technique. Finally, section 8 concludes the paper by summarizing the main points addressed through this paper.

2 General Issues

2.1 Modeling

To study a system, as the first step, we must construct a model representing the system under study. We

then research the system's behavior and get knowledge about the system by solving the model using either simulation or theoretical analysis. This method is applicable to any branch of natural science and social science. Performance evaluation is no exception.

A model for a system can informally be defined as "a collection of attributes and a set of rules that govern how these attributes interact" [Mullender93].

Constructing a good model that is really useful to the problem requires many careful considerations and much hard work. A good model has at least two features. First, it must well describe the behavior of a system, and should be as accurate as possible. Second, it must be as simple as possible to solve. The first feature requests that the model include all the necessary details that define the behavior; and the second requests that the model exclude as many parameters as possible. These two features contradict each other. It is the dilemma faced by any practitioner. Including too many parameters may lead to an accurate but too complicated or even unsolvable model. An overly complicated model costs too much to be solved, and an unsolvable model is completely useless to the problem, however accurate it is. Hence, great care should be taken in selecting parameters and a reasonable trade-off should be made.

Before working on it, one must have a clear idea about what the performance study is for, what results are of interest, and what attributes of the system are to be studied. Simply put, what goal is to be achieved? Different situations may have different goals; and different goals may produce different models. Thus, careful examination and full understanding of the problem is necessary.

A model is to a system what a map is to a road system. A model is only representing the system, not the system itself, and it is obtained from various assumptions and different levels of abstraction. Assumptions and abstraction must correspond to the attributes to be studied, emphasize some useful features and ignore some others. The behavior of a system is described by parameters. However, not all the parameters have the same effect on the behavior. It is therefore necessary to list all the parameters first, then carefully examine and compare them, introducing those that most affect the behavior to be studied. That is what we mean by abstraction. This process can lead to a model that just includes the necessary parameters that exactly describe the behavior of interest, and excludes all those that have no or little effect on it.

Once a model has been completed, it can be solved. Quite often the model needs to be validated and modified if the result is too far away from the real world.

Collier [Collier92] summarized the following four steps the modeling process must follow.

(1) Decide what answers are sought.
(2) Reduce the complexity of the real world to its essence, eliminating the irrelevant and approximating the unmanageable.
(3) Translate the essence into a formally defined system within which one can make deductions.
(4) Test the deductions in the real world, to see if what the model teaches either illuminates or can be applied.

2.2 Selecting Metrics

To study the performance of systems or to compare different systems for a given purpose, we must first select some criteria. These criteria are often called metrics in performance evaluation. Different situations need different sets of metrics. Thus, selecting metrics are highly problem oriented. What is more, the same metric may have different weights in different situations. For example, response time is a commonly used metric, which reflects how fast a system completes given services. However, it may receive more attention in a real-time control system than in a network system. The latter may be more interested in such metrics as throughput.

Unfortunately, there is little work which discusses metrics selecting issues in the performance evaluation community [Jain91]. This is not to say that authors ignore the importance of selecting metrics. The probable reason for this phenomenon is that there are countless metrics people may use in various situations: no standard set of metrics has been defined, and no standard technique has been applied. They

greatly vary from problem to problem.

In what follows we discuss some principles and a general process of selecting appropriate metrics. The materials presented in this subsection are mainly from [Jain91]. We hope to see more efforts made and papers published on this subject.

Any system is designed to offer services. Often different services should be examined one by one. For a given service requested to the system, there will be three possible outcomes.

(1) The system completes the service correctly;
(2) The system completes the service with error;
(3) The system fails to perform the service.

After classifying the possible outcomes, the remaining work becomes easier. Each situation need an individual set of metrics. They may be considered separately.

For the first case, the system completes the service correctly, the performance can usually be measured by time-rate-resource metrics, which represent the time used by the system to complete the service, the rate at which the system performs the service, and the resource needed for the system to complete the service. In real examples, they may be response time, throughput, and utilization. These metrics are widely used in performance studies. A great deal of examples can be found in the literature, e.g., [Takahashi87] and [Mailles87].

For the second case, the system completes the service with error, the probability that error occurs should be measured. In many cases, different types of errors need to be classified, and each type studied separately.

For the last case, the system fails to perform the service, again, the probability of failure is of interest. A system often consists of several components. It is sometimes helpful to identify the source of failure. If further examination is necessary, the down component (software or hardware) can be modeled and studied. For example, if the failure comes from the CPU, a simple CPU model can be constructed using two states, an "UP" state and a "DOWN" state. If we assume that the time to failure and the repair time both obey the exponential distribution with parameters λ and μ respectively, then this stochastic process is a Markov chain. Some well developed techniques can now be used [Chimento87].

Given a number of metrics candidates, the following three features should further be considered: *low variability*, *nonredundancy*, and *completeness*. High variability may need more efforts to obtain a given degree of statistical confidence. Examining redundancy may reduce the number of metrics; select only one representative of those that have the same effects. For example, in a network router system, which receives packets from the sources and sends them to the destinations, the mean number of packets in the system can be given by the famous Little's Law:

$$\bar{n} = \lambda \bar{\tau}$$

where λ is the mean arrival rate, and $\bar{\tau}$ is the mean waiting time [Robertazzi94]. Thus, of the two metrics, $\bar{n}$ and $\bar{\tau}$, selecting one is enough. Finally, completeness is obvious. Any selected set of metrics should be able to describe all the possible outcomes related to the performance of interest. Jain's book [Jain91] also presents an example showing the use of the three feathers.

2.3 Performance Evaluation Techniques

There are many techniques that can be used in evaluating performance. Usually these techniques are classified into three categories: measurement, simulation, and analytical modeling [Brewer91; Jain91]. Some authors (e.g., [Ferrari83]) classifies simulation and analytical modeling into one, called modeling technique. This is because whether we use simulation or analytical modeling, we must first construct a model for the system under study. simulation is considered to be one powerful method to solve complex models [Lavenberg83]. This classification is based on the fact that measurement must be done on a real system, i.e., the system to be evaluated must exist and be available, while modeling does not require that. For the convenience of discussion, we use the former classification.

Each of the three techniques has several types. For example, measurement has software, hardware, and hybrid; simulation has discrete event, trace/execution driven, Monte Carlo; and analytical modeling has queueing network, Petri net, etc.. With so many alternatives, selecting one that is suitable for the problem is really difficult, and needs many serious considerations. Below, we discuss the issues that must be taken into account in selecting techniques to solve a problem. These discussions are actually summaries of some representative papers and books (e.g., [Lavenberg83; Jain91; Crovella94; Trivedi94]).

Before doing performance evaluation, considering the following factors is helpful in selecting a proper technique: *stage*, *accuracy*, and *cost*. Perhaps the most important factor one should consider over the others is stage. That is, at what design stage would the performance evaluation be studied, because different stages may need different techniques and accuracy.

Measurement techniques, as the name implies, are based on direct measurements of the system under study using a software monitor or/and hardware monitor. Therefore, the real system must be available. This technique is totally impossible when the system is only at design stage and has not yet been built. Thus measurement is oriented towards *performance tuning*. Simulation and analytical modeling are both based on abstract models instead of real systems. A model can be constructed by abstracting essential features of the design based on appropriate assumptions, before the system is built. Thus both techniques are oriented towards *performance prediction* (although they are sometimes used for tuning). It should be noted that performance prediction at very early stage during the design life-cycle is not only helpful but necessary as well, for this will lead to discovery of potential design flaws which could then be fixed as early as possible, thus avoiding costly improvement or even redesign once the system has been implemented.

It should be pointed out that simulation and analytical modeling can be used throughout the design life-cycle, not only at earlier stages. Often as the design matures and more and more details of the system are obtained, more accurate models can be constructed. In spite of this, analytical modeling techniques are sometimes improper at later stages. This is because at that time the model often involves so many details that it costs too much to solve, or in some cases it is mathematically unsolvable.

Since both simulation and analytical modeling use models, which require simplifications and assumptions and are only approximate representations of reality, approximation errors might be introduced. This is especially true with analytical modeling techniques. To make the model be mathematically solvable, higher level of abstraction and more assumptions must be made. For example, in many analytical models only exponential distributions are allowed for time-distributions (e.g., [Heidelberger83; Mak90]). When simulation is used, this restriction has gone. Many complex non-exponential distributions can easily be used. Briefly, simulation allows more complex and detailed models, thus resulting in more accurate solutions.

Because measurement is directly carried out on a real system, it appears to offer the most accurate results among the three techniques. As a matter of fact, this is not always true. The basic technique of measuring is using instrumentation, which will introduce performance overheads and may also affect the system's dynamic behavior [Jelly94]. This is called instrumentation perturbation. Different kinds of perturbations, direct or indirect, are described by Malony [Malony92]. Besides, the accuracy of measurement also depends on several other things, such as time of the measurement and the workload used. Thus, measurement techniques may offer very accurate data or very inaccurate data, all depending on how well the above factors are treated.

As for the cost, because measurement needs instruments it is the most costly method. Analytical modeling needs nothing, so it is the cheapest one. Simulation goes just in between these two.

To summarize, analytical modeling, with the lowest cost and accuracy, can best be used at earlier stages; simulation, with higher cost and accuracy, can be used at any stage; and measurement, with the highest cost and varying accuracy can only be used after the system has been built.

It is worth mentioning that these techniques should be used together, simultaneously or sequentially, to evaluate a system whenever possible. The results of simulation and analytical modeling would be more convincing if the inputs and parameters are determined based on previous measurement [Jain91]; and

analytical models could be validated by simulations (e.g., [Agarwal92; Menasce92].

3 Fundamental Laws and Scalability Analysis

One key problem faced by parallel programmers is how to effectively utilize the processing power provided by the underlying architecture in order to gain performance. Obviously, good performance can easily be achieved when multiple independent sequential job streams are run. It is, however, quite difficult to get good performance from parallel applications [Anderson89]. In the discussions that follow, we assume that the entire computer is devoted to a single parallel application.

In general, as more processors are added to a parallel processing system or used by a parallel algorithm, the proportion of processor time taken to perform useful work diminishes. For example, in the discrete Fourier transform problem, Rayfield and Silverman [Rayfield88] reported that the improvement in speedup keeps diminishing as more processors are used. This phenomenon, or performance loss, is caused by so-called parallel overhead. Rayfield and Silverman attributed this to interprocessor communication and the sequential part of computation. Agarwal [Agarwal92] gave two reasons for the decreasing processor utilization. First, the cost of each memory access increases because network delays increase with system size. Second, as we strive for greater speedups through fine-grain parallelism, the number of network transactions and synchronization delays also increases. A paper by Burkhart and Millen [Burkhart89] discusses and analyzes various sources of performance loss.

This section presents some fundamental laws which research the gain of performance. This gain involves several aspects, not only in time (as with Amdahl's Law), but also in problem size (as with Gustafson's Law) and in the use of resources (as with memory-bounded speedup). These fundamental laws include speedup, scalability, isoefficiency, and so on. These are useful metrics for measuring performance and widely used in the literature. The definitions of these performance measures given by authors from different perspectives are greatly different. Different definitions may lead to different performance results. As we will see later, speedup is such an example. Several formal definitions, such as speedup, efficiency, utilization, etc., can be found in [Lee80]. These are fundamental concepts in parallel processing.

For the convenience of our later discussions, we first give definitions for speedup and efficiency. *Speedup* is defined as the ratio of the execution time of the best possible serial algorithm (on a single processor) to the parallel execution time of the chosen algorithm on a *n*-processor parallel system. Thus, we can write

$$S(n) = \frac{T(1)}{T(n)}$$

Efficiency is defined as the ratio of speedup to the number of processors, or

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

3.1 Parallelism Profile and Asymptotic Speedup

This subsection describes some concepts useful in later discussions on speedup models.

Parallelism Profile We define *degree of parallelism* (DOP) as the maximum number of processors used to execute a program at a particular instant in time, given an unbounded number of available processors and other necessary resources. In real systems, the DOP may not always be achievable because of many limitations.

The DOP often changes at different periods of time during the execution cycle. The plot of the DOP over the execution time may be quite helpful in studies. This plot is called *parallelism profile*. Figure 1 shows the parallelism profile of a divide-and-conquer algorithm.

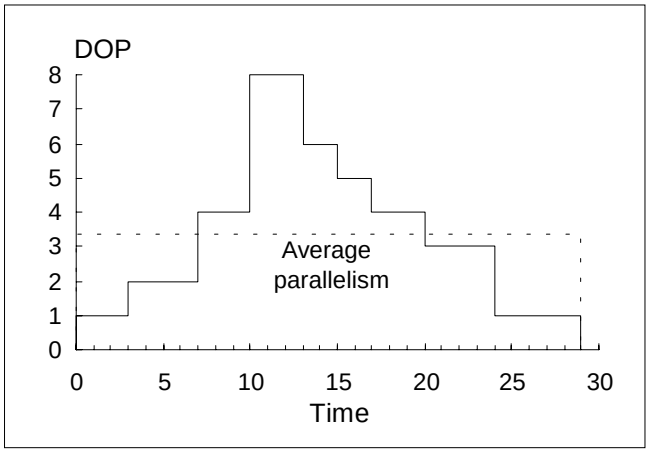

Figure 1. Parallelism profile of a divide-and-conquer algorithm

When i processors are busy during a period, we have DOP = i. The maximum DOP in a profile is denoted as m. We assume that a parallel processing system consists of n homogeneous processor, and $n >> m$ in the ideal case.

The *average parallelism* is a useful metric in studying speedup and efficiency. It is defined as the average number of processors that are busy during the execution time of the software system in question, given an unbounded number of available processors [Eager89].

According to this definition, the average parallelism can be written as

$$A = \left(\sum_{i=1}^{m} i \cdot t_i\right) \Big/ \left(\sum_{i=1}^{m} t_i\right)$$

Eager *et al.* [Eager89] carefully examine the average parallelism and investigate the tradeoff between speedup and efficiency. They show that

$$S(n) \geq \frac{nA}{n + A - 1}$$

and

$$E(n) \geq \frac{A}{n + A - 1}$$

$S(n)$ is lower bounded by $nA/(n + A - 1)$ and $E(n)$ by $A/(n + A - 1)$. We see that if $n << A$, $S(n) \rightarrow n$, and if $n >> A$, $S(n) \rightarrow A$. It should be noted that the average parallelism measure can only be used in a system without communication overhead.

Asymptotic Speedup Let W be the total amount of work of an application, W_i be the amount of work executed with DOP = i, and Δ be the *computing capacity* of each processor, which can be approximated by execution rate, such as MIPS or Mflops [Hwang93]. Thus, we have

$$W_i = i\Delta t_i$$

and

$$W = \sum_{i=1}^{m} W_i = \Delta \sum_{i=1}^{m} i \cdot t_i$$

By denoting the execution time of W_i on k processors as $t_i(k)$, we can write

$$t_i(1) = W_i / \Delta$$

$$t_i(k) = W_i / k\Delta$$

$$t_i(\infty) = W_i / i\Delta, \quad \text{for } 1 \leq i \leq m$$

Although an infinite number of processors can be used, the maximum number of processors that can be

used to execute W_i is still i.

If communication latency and other system overhead are not considered, then the execution times of W on a single processor and on an infinite number of processors are computed by

$$T(1) = \sum_{i=1}^{m} t_i(1) = \sum_{i=1}^{m} \frac{W_i}{\Delta}$$

$$T(\infty) = \sum_{i=1}^{m} t_i(\infty) = \sum_{i=1}^{m} \frac{W_i}{i\Delta}$$

The *asymptotic speedup* is formally defined as

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} W_i/\Delta}{\sum_{i=1}^{m} W_i/i\Delta} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} W_i/i}$$

This speedup is equivalent to the average parallelism A given above. Note that S_∞ is obtained under the assumption of having an infinite number of available processors and no communication latency. Thus the asymptotic speedup is the best possible speedup that can be achieved.

3.2 Three Speedup Modes

In this section, we consider three well-known speedup models. They are fixed-size, fixed-time, and memory-bounded speedup models.

Fixed-size Speedup In the asymptotic speedup model, we assume that the number of available processors is unbounded. If the number of processor n is taken into account, and $n < i$, then

$$t_i(n) = \frac{W_i}{i\Delta}\left\lceil \frac{i}{n} \right\rceil$$

Note that this equation still hold for $n \geq i$, in which case, $\lceil i/n \rceil = 1$.

Hence,

$$T(n) = \sum_{i=1}^{m} \frac{W_i}{i\Delta}\left\lceil \frac{i}{n} \right\rceil$$

and the speedup is

$$S(n) = \frac{T(1)}{T(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}\left\lceil \frac{i}{n} \right\rceil}$$

If communication latency and other system overhead are considered, the speedup becomes

$$S(n) = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}\left\lceil \frac{i}{n} \right\rceil + Q(n)}$$

where $Q(n)$ is the sum of all the overheads. Usually, it is different to derive a closed form for $Q(n)$ for it depends not only on the machine but also on the application and the problem allocation. In the later discussions, we assumed $Q(n) = 0$.

If we assume that the problem size, or workload, is fixed, and the workload only contains two parts, a sequential part (DOP = 1), and a perfectly parallel part (DOP = n), i.e., $W_i = 0$ for $1 < i < n$ in the parallelism profile, then the speedup becomes

$$S(n) = \frac{W_1 + W_n}{W_i + W_n/n}$$

Let α be $W_1/(W_1 + W_n)$, representing the percentage of the sequential portion. If the workload is normalized to 1, i.e., $W_1 + W_n = 1$, then $W_1 = \alpha$, and $W_n = 1 - \alpha$. The speedup is

$$S(n) = \frac{1}{\alpha + (1-\alpha)/n} = \frac{n}{1+(n-1)\alpha}$$

This is known as Amdahl's law. In 1967, Amdahl [Amdahl67] made the observation that if α is the sequential fraction in an algorithm, then no matter how many processors are used, the speedup is upper bounded by $1/\alpha$. α is called *sequential bottleneck*.

Two observations can be made from the speedup formulation. If $\alpha = 1$, $S(n) = 1$, i.e., no part can be parallelized. If $\alpha = 0$, $S(n) = n$, i.e., the given algorithm can perfectly be parallelized, thus obtaining the maximum speedup.

For a variety of parallel systems with any number of processors *n*, speedup close to *n* can be achieved by simply running the parallel algorithm on large enough problems. Much work has been done on that (see for example [Kumar87; Lee87]).

Fixed-time Speedup To study how small the turnaround time of a problem could be, Amdahl fixes the problem size, thus leading to the development of the fixed-size speedup, which is suitable for algorithms in which the problem size cannot be scaled. However, there exist many problems whose execution time is not so critical (although they may have time limitations). Instead, we emphasize their accuracy. When the machine size is increased and more computing power obtained, we may increase the problem size and perform more operations, thus obtaining more accurate solution, yet keeping the turnaround time unchanged. This observation lead Gustafson to develop the fixed-time speedup model.

Let W' be the total amount of scaled work, W_i' be the amount of scaled work with DOP = *i*, and m' be the maximum DOP of the scaled problem. We have $W = \sum_{i=1}^{m'} W_i'$. In order to keep the same turnaround time of the scaled problem as the sequential version (before scaled), $T(1) = T'(n)$ must be satisfied, that is

$$\sum_{i=1}^{m} W_i = \sum_{i=1}^{m'} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n) \qquad \text{(x)}$$

Thus the fixed-time speedup is

$$S'(n) = \frac{T'(1)}{T'(n)} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m'} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} W_i} \qquad \text{(xx)}$$

As with the fixed-size speedup, we assume the work only contains a sequential part and a perfectly parallel part. And we also assume the sequential part is independent of the problem size or system size, i.e., $W_1 = W_1'$. From Eq. (x), we have $W_1 + W_n = W_1' + W_n'/n$, thus $W_n' = nW_n$. The speedup becomes

$$S'(n) = \frac{W_i' + W_n'}{W_i + W_n} = \frac{W_1 + nW_n}{W_1 + W_n}$$

Let $\alpha = W_1$ and $1 - \alpha = W_n$, we rewrite $S'(n)$ as

$$S'(n) = n + (1-n)\alpha$$

This fixed-time speedup is known as Gustafson's scaled speedup [Gustafson88]. To keep the turnaround time unchanged, we have $W_n' = nW_n$. This means that the parallel part scales up linearly with the system size. Hence, Gustafson's law supports scaled performance.

In Amdahl's model, the load on each processor decreases as the number of processors increases, and finally the sequential part dominates the performance and the speedup is bounded by $1/\alpha$. In Gustafson's model, however, when the number of processors increases, the parallel part of the workload scales up linearly ($W_n' = nW_n$). Thus the sequential part is no longer a bottleneck. That is why Gustafson's law offers a good speedup. Two figures in Gustafson's original paper [Gustafson88] explain that clearly.

Some experiments were done at Sandia by Gustafson, Montry and Benner [Gustafson88a; Benner88], and near-linear speedup was obtained on a very large parallel system. On a 1024-processor hypercube

architecture, three high speedups: 1021, 1020, and 1016 were achieved for three practical applications.

One disadvantage of the fixed-time model is that not arbitrarily large number of processors can effectively be used to keep the turnaround time unchanged. Worley [Worley90] found that in order to keep the turnaround time fixed, no more than 50 processors can effectively be used for many common scientific problems. For some other problems, however, he found that near-linear time-constrained speedup can be obtained, which means that arbitrarily large instances of the problems can be solved within a fixed period of time by simply increasing processors. Gupta and Kumar [Gupta93] classify parallel systems which yield linear and sublinear time-constrained speedup curves.

Memory-Bounded Speedup Sun and Ni [Sun93] notice that in parallel systems the scaled problem size is limited by memory space. Under the assumption of distributed shared memory multicomputer systems, they develop the memory-bounded speedup.

The requirement of an algorithm consist of two parts. One is the memory requirement (*M*), and the other is the computation requirement or workload (*W*). For a given algorithm, these two are related to each other, i.e., $W = g(M)$ or $M = g^{-1}(W)$.

In distributed shared memory systems, the local memory capacity is fixed, but the total available memory increases linearly with the number of processors available. We write $W = \sum_{i=1}^{m} W_i$ as the workload for sequential execution on a single processor, and $W^* = \sum_{i=1}^{m^*} W_i^*$ as the scaled workload for execution on *n* processors. The following condition must be satisfied:

$$W^* = g(nM) = g(n\,g^{-1}(W))$$

The memory requirement for any active processor is thus bounded by $g^{-1}(\sum_{i=1}^{m} W_i)$.

Like Eq. (xx), we have

$$S^*(n) = \frac{\sum_{i=1}^{m^*} W_i^*}{\sum_{i=1}^{m^*} \frac{W_i^*}{i} \left\lceil \frac{i}{n} \right\rceil + Q^*(n)}$$

If we assume *g* is a semihomomorphism, i.e., $g(nM) = G(n)g(M)$, then the scaled parallel part is

$$W_n^* = g(nM) = G(n)W_n$$

Under the same assumption as is made with the fixed-time model, we can get the memory-bounded speedup as follows

$$S^*(n) = \frac{W_1^* + W_n^*}{W_1^* + W_n^*/n} = \frac{W_1 + G(n)W_n}{W_1 + G(n)W_n/n}$$

This memory-bounded speedup is actually a general speedup model. Note that

- If $G(n) = 1$, it becomes equivalent to Amdahl's law;
- If $G(n) = n$, it becomes equivalent to Gustafson's law;
- If $G(n) > n$, the computational workload grows faster than the memory requirement, and execution time is likely to be the scalable constraint. Thus, the memory-bounded model will likely offer a higher speedup than the fixed-time model does;
- If $G(n) < n$, the memory capacity is likely to be the scalable constraint when *n* is large. Thus the fixed-time speedup is likely better.

[Sun93] give an example on how to find $G(n)$ for an matrix multiplication algorithm. For Sun's other papers that address scaled speedup issues, interested readers are referred to [Sun90; Sun91].

In addition to the three famous speedup models described above, there still exist many other definitions of speedup, such as *generalized speedup*, *cost-related speedup*, *superlinear speedup*, and so on. They approach the problem in different ways. Some of them can be found in [Barton89; Gustafson91; Helmbold90; Sun91].

3.3 Isoefficiency Function

The isoefficiency concept was first introduced by Kumar and Rao [Kumar87]. It is especially useful in analyzing scalability of parallel algorithm-architecture combinations.

Let T_1 be the sequential execution time, T_P be the parallel execution time on p processors, and T_O be the total overhead (including idling, communication, and contention over shared data structures). We can see that

$$T_P = \frac{T_1 + T_O}{p}$$

The speedup is

$$S = \frac{T_1}{T_P} = \frac{pT_1}{T_1 + T_O}$$

The efficiency is then computed by

$$E = \frac{S}{p} = \frac{T_1}{T_1 + T_O} = \frac{1}{1 + T_O / T_1}$$

Let W be the workload or problem size, which could be measured by the number of operations the best sequential algorithm executes to solve the problem on a single processor, and t_c be the cost of each operation. Then $T_1 = t_c W$. The efficiency can be rewritten as

$$E = \frac{1}{1 + T_O / t_c W} \qquad \text{(x)}$$

The isoefficiency concept is based on the following observations. First, if the problem size (W) is fixed, the efficiency decreases as p increases because T_O increases with p. Second, if p is fixed, then the efficiency increases with the problem size because the overhead T_O grows slower than W for a given p. Thus it is possible to maintain a constant efficiency E ($0 < E < 1$) by increasing the problem size W proportionally with increasing the machine size.

Eq. (x) can be rewritten as

$$W = \frac{1}{t_c}\left(\frac{E}{1 - E}\right) T_O = K T_O$$

where $K = E/(t_c(1 - E))$, which is a constant for a given E. If the problem size needs to grow as fast as $f_E(p)$ to maintain an efficiency E, then $f_E(p)$ is defined to be the isoefficiency function of the algorithm-architecture combination for efficiency E, i.e., $f_E(p) = KT_O$. Different isoefficiency functions for different parallel systems can be found in the table of [Kumar87].

Isoefficiency is powerful in scalability analysis. A small isoefficiency function implies that small increments of the problem size are sufficient for effectively utilizing an increasing number of processors, and hence the parallel system is highly scalable. Conversely, a large isoefficiency function implies that the system is poorly scalable. For example, [Kumar87] shows that the parallel algorithm for solving the 0/1 knapsack problem given in [Lee87] is highly scalable for its isoefficiency function is $O(N \log N)$, while a frequently used parallel formulation of quicksort [Quinn87] has an exponential isoefficiency function, thus it is poorly scalable.

[Gupta91] presents scalability analysis of four different algorithms, which shows isoefficiency function is a proper metric for scalability analysis. For a detailed description of the isoefficiency concept and its applications, the following papers are recommended: [Grama93] and [Gupta93a].

3.4 Scalability Analysis

Scalability analysis plays an important role in performance evaluation of large parallel systems. On large

systems, one key issue is how to effectively utilize the processors provided by the system. Usually as more processors are used, the efficiency, utilization, and speedup etc. will drop. Scalability is such a metric that measures the capacity to effectively utilize an increasing number of processors.

A simple definition of scalability is that the performance of a parallel architecture increases linearly with respect to the number of processors used for a given algorithm. This definition involves two factors: architecture and algorithm, neither of which can be ignored. In fact, scalability studies determine the degree of matching between a computer architecture and an application algorithm. For different algorithm-architecture combinations, the scalability analysis may end up with different conclusions. An architecture may be scalable for one algorithm, but may not at all for another [Hwang93].

In the literature, lots of work on scalability analysis can be found. Kumar and Gupta [Kumar94] summarize the following situations where scalability analysis is found to be very useful.

- Selecting the best algorithm-architecture combination for a problem under different constraints on the growth of the problem size (workload) and the number of processors (machine size);
- Predicting the performance of a parallel algorithm and a parallel architecture for a large number of processors from the known performance on fewer processors;
- For a fixed problem size, determining the optimal number of processors to be used and the maximum speedup that can be achieved;
- Predicting the impact of changing hardware technology on the performance and thus help design better parallel architectures for solving various problems.

The notion of scalability is closely related to the notions of speedup and efficiency. To maintain the speedup and efficiency at a reasonable level, we have to increase the workload (W) properly with increasing the machine size (n). In different situations, the workload needs to be increased at different rates with respect to the number of processors. In some situations, if the workload needs to grow linearly w.r.t. n (which implies linear scalability in problem size), we say that the parallel system is highly scalable. On the other hand, if we need an exponential growth in the workload w.r.t. the machine size, then we say that the parallel system is poorly scalable, because to keep a constant efficiency or a good speedup, the increase in the problem size must be explosive, which is always limited by the memory capacity, and impossible in practice [Kumar87].

Gustafson's law, discussed earlier, supports the scaled speedup, which is obtained when the problem size grows linearly with the machine size ($W_n' = nW_n$). If the speedup curve is linear or near-linear with respect to the machine size, then the parallel system is considered to be scalable. Carmona and Rice [Carmona89; Carmona91] give detailed explanations for Amdahl's law and Gustafson's law. They provide new and improved interpretations of the serial and parallel fraction parameters commonly used in the literature. The interpretations are based on quantifiable expressions of these fractions. The authors also give new definitions of speedup and efficiency and introduce a general model of parallel performance.

The isoefficiency function is also a proper metric for analyzing scalability. An important feature of isoefficiency analysis is that in a single expression, it succinctly captures the effects of characteristics of a parallel algorithm as well as the parallel architecture on which it is implemented. As stated earlier, $W = f_E(p) = KT_O$, where $f_E(p)$ is the isoefficiency function. If $f_E(p)$ is small, then only small increments are required to effectively utilize an increasing number of processors. Thus the parallel system is considered to be highly scalable. In Kumar and Rao's framework, if an isoefficiency function exists, then the parallel system is considered to be scalable; otherwise it is unscalable. However, for some problems, the function would probably be very large. This means that the workload must be increased very fast w.r.t. the machine size. But in practice the problem size is bounded by memory capacity. Thus, from a practical viewpoint such systems should also be considered unscalable (even though their isoefficiency functions exist).

Nussbaum and Agarwal [Nussbaum91] give their definition for scalability. "For a given algorithm, an architecture's scalability is the ratio of the algorithm's asymptotic speedup when run on the architecture in

question to its corresponding asymptotic speedup when run on an EREW PRAM, as a function of problem size." The asymptotic speedups for the ideal machine (EREW PRAM) and for the given architecture are

$$S_I(s,n) = T(s,1)/T_I(s,n)$$

and

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

respectively. Thus the scalability is

$$\Psi(s,n) = \frac{S(s,n)}{S_I(s,n)} = \frac{T_I(s,n)}{T(s,n)}$$

where *s* is the problem size and *n* is the number of processors. Intuitively, the larger the scalability, the better the performance that the given architecture can yield running the given algorithm. This definition captures not only algorithm scalability (measured through the asymptotic speedup on an ideal machine), but architecture scalability as well. Note that, in the ideal case, $S_I(s,n) = n$, we have

$$\Psi(s,n) = \frac{S(s,n)}{n} = E(s,n)$$

This means that $\Psi(s,n)$ is a better metric for scalability than the efficiency $E(s,n)$ is. The reason is that the asymptotic speedup measures the parallelism inherent to the algorithm (refer to section 3.1). This definition is useful in comparing different architectures for a given algorithm. It is, however, useless in comparing different algorithm-architecture pairs for solving the same problem.

Karp and Flatt [Karp90] use serial fraction *f* as a metric for measuring the performance of a parallel system on a fix-sized problem. They define

$$f = \frac{1/S - 1/n}{1 - 1/n}$$

where *S* is the speedup, and *n* is the number of processors. Interestingly, if we rewrite this equation as

$$S = \frac{n}{1 + (n-1)f}$$

It becomes Amdahl's law, and *f* is just the serial fraction α. In general, a smaller *f* is desirable. If *f* increases with *n*, it is considered as an indicator of rising communication overhead, and thus an indicator of poor scalability. The authors explain the phenomenon that *f* decreases with increasing *n* as an anomaly caused by superlinear speedup effects or cache effects.

Zorbas *et al*. [Zorbas89] introduce the concept of an overhead function $\Phi(n)$. If the execution time $T(n)$ is given by

$$T(n) \le t_c(W_1 + W_n/n) \times \Phi(n)$$

then the smallest function $\Phi(n)$ that satisfies this equation is called the overhead function and is computed by

$$\frac{T(n)}{t_c(W_1 + W_n/n)}$$

In general, the overhead grows with the number of processors. Thus the rate at which $\Phi(n)$ grows reflects the degree of scalability. The smaller the rate, the more scalable the parallel system. And in the ideal case where $\Phi(n)$ remains fixed, the parallel system is ideally scalable.

Something like Nussbaum and Agarwal's method [Nussbaum91], where they compare the performance of a parallel system with that of the ideal machine, Van-Catledg [Van-Catledg89] compare the performance of a parallel system with that of a supercomputer (called reference machine) for the same problem. Let *W* be the problem size measured by the number of operations, and *s* be the serial fraction. Then the number of sequential and parallel operations equals sW and $(1-s)W$; and they become $G(k)sW$ and $F(k)(1-s)W$ after the problem size *W* is scaled up by a factor *k*. Thus, the parallel execution time on the parallel system

with *n* processors is given by

$$T(n) = t_c W\left(G(k)s + \frac{F(k)(1-s)}{n} \right)$$

Similarly, the execution time on the reference machine with n' processors is

$$T'(n') = t_c' W\left(G(k)s + \frac{F(k)(1-s)}{n'} \right)$$

The relative performance is defined as $T(n)/T'(n')$. If for some *n* the relative performance is 1, then *n* is used as *equal performance condition* to measure the scalability of the parallel system. The parallel system is considered more scalable with a smaller *n*. The author also shows, by examples, that a system with fewer faster processors is better than a system with more but slower processors. This is intuitively true because a system with more processors incurs more overheads and the probability that more processors are idle is higher.

In addition to those discussed above, there are still some useful concepts, such as *shape*, *execution profile*, *phases*, etc., which are more related to workload characterization, and will be described in the next section.

4 Workload and Workload Characterization

Any system is designed to be used in a specific environment with a specific workload. Therefore, the performance analysis of a system must be carried out with a certain workload under which the system is designed to run. Consequently, selecting proper workloads is important in performance evaluation. Workload is defined as the entity of all individual tasks, transactions, and data to be processed in a given period of time, or simply, the workload of a computer system is the user's demand from the system [Oed81]. A real workload is often very complex and unrepeatable, so it cannot be used properly for studies. Because of this, a test workload model must be designed. This model must have the following characteristics.

- It must be a representation of the real workload. That is the static and dynamic behavior of the real workload must be accurately captured. In other words, the test workload model must perform the same functions in the same proportions as the real workload, and it must demand the same system's resources in the same proportions and rates as the real workload;
- It must be easily reproduced and modified, such that the workload can be used repeatedly in different studies;
- It must be compact, such that the workload can be easily ported to different systems. This is quite useful in comparing different systems for the same purpose.

In this section, we first present several types of workloads that are traditionally used, and then review some techniques that are now commonly used in workload characterization.

4.1 Workload

Instruction Mix *Instruction mix* specifies the relative frequencies of different types of instructions. The types of instructions and their relative frequencies used in an instruction mix must be the same as those used in the real workload. Then the average execution time can be measured on the system to be tested. The Gibson mix [Gibson70] developed for use with IBM systems is perhaps the most widely known in industry. The instruction level of the mixes can be extended to statement level of high-level languages, called *statement mix*.

One of the advantages of an instruction mix is that it is relatively easy to design and use. However, it has several disadvantages. Because it is designed to measure the speed of CPU, it is not suitable for evaluating the whole system's performance if the CPU is not the bottleneck. Furthermore, it cannot be properly used to compare two CPUs with different instruction sets (e.g., RISC and CISC). Because modern

computer systems often adopt new technologies, such as cache, pipeline, in pre-fetching instructions to get high efficiency, the problems are difficult to tackle properly with instruction mixes. Because of this, this technique becomes obsolete. Some early papers are available to the interested readers: [Knuth71], [ISE79], and [Febish81].

Synthetic Programs Synthetic programs are also called synthetic benchmarks which are programs designed to simulate real workload. They do no "useful" work, but consume amounts of system resources or request amounts of system services. The main advantages are that they can be quickly developed and easily modified to simulate a wide spectrum of real problems through a set of control parameters and they do not necessarily use real data files. The disadvantage of synthetic programs is that they are often too simple to accurately reflect some real system issues, such as disk caches.

Shein *et al*. [Shein89] proposed a synthetic program, named NFSStone, to measure the performance of SUN's NFS (Network File System) [Sandberg85]. It uses a single client that issues various requests for file operations in order to stress and measure the performance of the server. However, using only one client is not sufficient to stress the server. Another problem with the NFSStone is that the file and block sizes does not properly reflect the real workload, which may influence the accuracy.

Park and Becker [Park90] developed an I/O benchmark called IOStone. It performs data read and write on 400 files, the total size of which is 1 MB. The output of the program is throughput which is measured by IOStone/second. One problem with IOStone is that no I/O parallelism is introduced.

Noticing that most scientific applications often deal with large amount of data, Chen and Patterson [Chen93] describe a scientific benchmark which operates on very large files (100 MB) in large units (128 KB). Results from two sample scientific workloads are presented, but no other details are introduced in that paper.

Application Benchmarks Many computer systems are designed to be used in particular applications, such as airline reservations or transaction processing. To evaluate the performance of such systems or to compare them, the synthetic benchmarks are not sufficient. This demand gives rise to application benchmarks, which incorporate a set of representative functions the applications might use. Unlike a synthetic benchmark, an application benchmark is intended to do some real work as is done by standard programs, such as banking systems, editors, compilers, etc..

Since the 70', Debit-Credit benchmarks have become more and more popular. These application benchmarks are designed to compare transaction processing systems. The commonly used metric is price-performance ratio. The price includes the total cost to purchase, install, and maintain the system (both hardware and software) over a specified period of time. The performance is measured by throughput in terms of TPS (transactions per second). Each transaction may involve a database search, query answering, and database update operations. The database system used typically includes such records as account, teller, branch, and so on. Banking or business systems should be designed to deliver high TPS rates. The first benchmarks of this kind is known as TP1, which was originally proposed in 1985 [Anonymous85], and has already become the *de facto* standard for gauging transaction processing systems and rational database systems.

To define transaction processing benchmarks more precisely, the Transactions Processing Performance Council (TPC) was formed in August 1988 and proposed its first benchmark, TPC-A, in 1989 [TPC89]. This benchmark is actually based on TP1, and the throughput is measured in terms of TPS such that 90% of all transactions provide 2 seconds or less response time. TPC-A requires that all the requests be from the real terminals, and the average interarrival time of the requests is 10 seconds. A new version of the benchmark, named TPC-B, is later proposed [TPC90]. Like its previous versions, TPC-B uses a database containing records for accounts, tellers, and branches. It simulates typical account changes on the database by carrying out Debit-Credit transactions. The throughput is also measured in the same way as TPC-A. The price for the system and required storage, and a graph of throughput are reported at the end of test. Unlike TPC-A, the requests of which are from the real terminals, TPC-B generates requests using internal drivers,

which can generate the requests as fast as possible.

Some well-known benchmarks When it comes to the workload, we cannot ignore several well-known benchmarks, which are still commonly used in industry although they are rather old. Below, we briefly review some of them. The interested readers are referred to the literature for more details.

Whetstone is actually a synthetic benchmark [Hwang93]. It includes both integer and floating-point operations involving array addressing, subroutine calls, parameter passing, fixed/floating-point arithmetic, conditional branching, and trigonometric/transcendental functions. The operations are programmed according to the statistical data of about 1000 ALGOL programs. Because of this, it is considered to be a standard floating-point benchmark. The test results are measured in the number of *Kwhetstone/s* (Kilo Whetstone Instructions Per Second) that the system can perform. The main disadvantages of Whetstone benchmark are that it is compiler sensitive, and it is designed to test CPU performance: no I/O involved.

Dhrystone should also be considered a synthetic benchmark [Hwang93]. It is CPU-intensive, and designed to represent systems programming environment, thus no floating-point operations and I/O processing are exercised and the majority of the CPU time is spent in string manipulating. Since it was proposed, it has been updated for several times, and is considered to be a popular measure of the integer performance of modern processors [Jain91]. The results are measured in *Kdhrystones/s* (Dhrystone Instructions Per Second). The disadvantages found in Whetstone also apply to this benchmark.

The *LINPACK* benchmark was designed by Dongarra [Dongarra83]. It contains a number of programs that solve dense systems of linear equations using the LINPACK subroutine package. LINPACK is a general-purpose FORTRAN library of mathematical software for solving dense linear systems of equations of order 100 or higher. LINPACK benchmarks are compared based on the execution rate as measured in Mflops. In fact, many published Mflops and Gflops results are based on running the LINPACK code with prespecified compilers [Hwang93]. Many LINPACK benchmark results obtained by running on various computer systems can be found in [Dongarra92].

4.2 Workload Characterization

4.2.1 Overview

Workload characterization is the process of developing a workload model that can be used repeatedly. In order to design a workload model, it is essential to carefully study and understand the key characteristics and to know the limitations of the model. Oed and Mertens [Oed81] list and discuss in detail four basic levels, at which the performance of a computer system and the respective workload is influenced. They are *user*, *hardware*, *software*, and *organization*.

A workload model is a miniature workload with reduced information. The issues addressed in section 2.1 also apply to modeling workload. Often the amount of data collected (either by measuring or any other methods) is very large. Thus careful examination of classification of the data should be made to eliminate all the negligible parts and include only those that best describe the real workload.

In modeling workload, *representativeness* is one of the main characteristics that should be considered. The representativeness of a workload model can be determined by an equivalence relationship [Ferrari78]. Given a system *S*, a set of performance indices *L* can be obtained from a certain workload *W*. Formally, a function f_s can be defined for *S*, so that $L = f_s(W)$. If W_r is a real workload and L_r is the corresponding performance indices, then we have $L_r = f_s(W_r)$. Similarly, we have $L_m = f_s(W_m)$, where W_m is the workload model and L_m is the corresponding performance indices. W_m is considered to be equivalent to W_r if L_m falls within certain boundaries of L_r.

The methodologies and techniques applied for constructing workload models are highly related not only to the goals of the studies but to the system under study as well. The problems encountered in the workload characterization of centralized systems are well-known and have been approached and solved

quite a long time ago. However, similar conclusions cannot be drawn for more recent types of architectures, such as multiprocessor systems, client-server architectures, and so on [Calzarossa93]. In the parallel performance evaluation literature, we find it hard to summarize systematic ways for modeling workload. In the sequel we first review some general techniques that are commonly used in workload characterization for traditional system architectures. Some of them can also be useful to modern architectures. Then we survey some papers that address workload characterization for modern architectures.

It should be pointed out that workload characterization is related to other techniques of performance evaluation. The original data that are used are often obtained from measuring the system or from accounting routines of the system. However, the measured data cannot be directly used in performance studies without necessary processing. Actually, the techniques to be discussed below are used to process data. Workload characterization is also related to analytical modeling and simulation. For example, in [Pettey95], genetic algorithms (GAs) are used to accurately extract the assumed exponentially distributed customer class demands for a closed queueing network from the monitor data.

Before describing the techniques, we think it necessary to explain two terms that are commonly used in the workload characterization literature. They are *workload component* and *workload parameters*. Workload component (or workload unit) is often used to denote the entity that makes the service requests to the system under study. Examples of workload components are applications, programs, commands, machine instructions, and so on. Workload parameters (or workload features) mean the measured quantities, service requests, or resource demands, which are used to model or characterize the workload. Examples are transaction types, packet sizes, page reference pattern, and so on [Jain91].

4.2.2 Steps of Workload Characterization

Ferrari [Ferrari83] groups the main operations to be performed in implementing workload models into three phases: *formulation phase*, *construction phase*, and *validation phase*, each of which involves several operations. They are summarized as follows.

In the formulation phase, decisions should be made on what parameters are to be selected. This is often influenced by two factors: objectives of the study and the availability of parameters. Among the first operations to be performed is the definition of the workload basic components. After studying the objectives and verifying their availability, the parameters to be used to characterize the workload basic components are selected on the basis of the modeling level of detail. Two different levels are defined: the physical resource level and the functional level. If the model is constructed at the physical resource level, parameters, such as CPU time, memory space demand, number of I/O operations, etc., can be used. If the model is to be implemented at the functional level, the parameters that could be used are, for example, the number of compilations, and the number of executions of certain systems programs, and so on. Another issue that should be considered during this phase is the choice of the number of parameters to be used and the homogeneity of their values. This is very important in later statistical analysis, because they both will affect the degree of difficulty and accuracy of the resulting workload model.

In the construction phase, there are four fundamental operations. They are

(1) The analysis of parameters;
(2) The extraction of representative values;
(3) Their assignment to the components of the model;
(4) The reconstruction of the mixes of significant components;

During this phase, the main operation is using appropriate techniques to analyze the parameters. The workload may be considered as a set of vectors in an n-dimensional space, where n is equal to the number of the workload parameters. Each set of the parameter values is a point in the space. In order to reduce the number of points characterizing the workload, and identify groups of components with similar characteristics, a number of statistical techniques, which will be reviewed later in this section, can be

applied.

In the validation phase, the evaluation criteria for the representativeness of a model must be applied to establish the validity of the implemented model. The criteria should be determined according to the objectives of the study.

From the above description, we can see that the main considerations and the operations that an analyst should perform to build a workload model are almost the same as given in section 2.1. The readers can also find a similar summary in [Calzarossa93].

4.2.3 Workload Characterization Techniques

There exist many techniques (statistical methods) that have been successfully used in workload characterization. Again, there are many papers and books that address these techniques (see for example [Hartigan75; Oed81; Ferrari83; Jain91]). Summarized below are the most commonly used techniques. They are:

- Averaging
- Distribution Analysis
- Principal component analysis
- Clustering
- Markov models

Averaging Perhaps the simplest methods to characterize a parameter is to use a single value to represent the data obtained from measuring. The most intuitive method is averaging. There are a number of averaging methods, for example, *arithmetic mean*, *geometric mean*, *harmonic mean*, and so on. These techniques should be used under different conditions. Sometimes, to make the results more accurate pre-processing should be performed. For example, the outliers (whose value is very different from the typical ones) should be removed before averaging. Although in some cases this method is very useful and effective (for it need only one value to character a workload parameter), it is not very effective, or even absolutely useless in some others. For example, averaging is not sufficient if the data to be processed show a large variability. Variability is commonly measured by *variance*, *standard deviation*, *coefficient of variation* (COV), etc.. In a network router example, where source and destination addresses are used as parameters, the average address has no meaning at all. In such case, the most frequent value, or the top-*n* values should be specified.

Distribution analysis Given a number of workload parameters and their corresponding measured data, distribution analysis can be applied. The analysis can be made to each parameter independently (*single-parameter* analysis), or to some combinations of several parameters dependently (*multiparameter* analysis). The latter is often called *compound distribution* analysis. A commonly used method in distribution analysis is histogram. In this method, the complete parameter range is divided into several smaller subranges called *buckets* (or *cells*). Then the frequencies of the data items that fall within each bucket are counted.

With this, the minimum, maximum, mean, median and standard deviation etc. can be estimated. They can be used to characterize the distribution, or the frequencies can be directly used to generate random numbers with a similar distribution to that of the real parameter. The inherent problem of single-parameter analysis is the loss of the correlation information among different parameters. For example, a job that requires long CPU time may also requires a large number of I/O operations.

If the result of the single-parameter analysis cannot properly reflect the characteristics of the real workload because of the significant correlation between the parameters, the multiparameter analysis should be used. In multiparameter analysis, if n parameters are selected, an n-dimensional space (or correspondingly an n-dimensional matrix) can be created. As with the single-parameter analysis, the space is divided into a finite number of cells. If the whole range of each axis is divided into L parts, then the total

number of cells is L^n. Each component of the workload falls into one of the cells.

A workload model can be constructed from the distribution. A case study of this method can be found in the earlier literature (e.g. [Sreenivasan74]). Very often the number of components can be reduced using certain techniques [Ferrari83]. Thus, a test workload model with fewer number of data can be built. In spite of this, the resulting workload model is still too detailed. This restricts the usefulness of the method. Another drawback of this method is that it is difficult to plot the joint distribution when *n* is greater than two.

Principal component analysis This is a technique that transforms a set of variables into a set of principal components characterized by linear dependence on the variables in the original set and by orthogonality among its own variables. Because of its size reduction capability, it is often used in workload characterization [Ferrari83]. The main operations are to find the eigenvalues and the corresponding eigenvectors of the correlation matrix of a given set of variables based on a given set of their values.

Let $\{x_1, x_2,\ldots, x_n\}$ be a set of *n* parameters, and $\{y_1, y_2,\ldots, y_n\}$ be a set of factors. Jain [Jain91] lists the following conditions that must be satisfied.

(1) The *y*'s are linear combinations of *x*'s:

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

where a_{ij} is the loading of variable x_j on factor y_i.

(2) The *y*'s form an orthogonal set, i.e., their inner product is zero:

$$\langle y_i, y_j \rangle = \sum_k a_{ik} a_{kj} = 0$$

This condition means that y_i's are uncorrelated to each other.

(3) The *y*'s form an ordered set such that y_1 explains the highest percentage of the variance in resource demainds, y_2 a lower percentage, y_3 a still lower percentage, etc.. This order is useful, because according to the level of modeling detail, only the first few factors can be selected to classify the workload components.

There are many books that describe in detail this technique (see for example [Rummel70; Harman76]). Applications of this technique can be found in [Hunt71] and [Serazzi81; Serazzi85].

Clustering The aim of the statistical technique is to group the workload components into *classes* or *clusters* so as to make the differences between the members of the same class smaller than those between the members of different classes. The similarity is measured by a certain criterion. The criterion most commonly used are *Euclidean distance*, *weighted Euclidean distance*, and *Chi-square distance*. The concept between clustering and the compound distribution addressed above is similar. The difference is that clustering determines the cells by the workload components themselves rather than by previous division [Oed81].

After the workloads are classified, one member from each class may be selected to represent the class in later studies, or similar to the compound distribution, several representative components may be selected from each class, according to the size of the class (relative frequency). This way, the number of the workload components can greatly be reduced.

Everitt [Everitt74] described various clustering techniques. Also, various algorithms that can be applied to workload characterization can be found in [Hartigan75]. They fall into two classes: *hierarchical* and *nonhierarchical*. The minimum spanning tree method is one of the most widely used hierarchical algorithms, while the k-means method is one of the most widely used nonhierarchical algorithms. Numerous case studies of this technique can be found in the literature. See for example [Wight81; Calzarossa86].

Markov models Markov models have found wide applications in modeling stochastic processes. They are also very useful in workload characterization. Markov models treat the state transition of systems. If we assume that the next state of a system depends only on the current state, then we say that the system

follows a Markov model (memoryless property). The memoryless property of Markovian systems makes them relatively simple to analyze. Markov models are usually described by a *transition matrix*, the elements of which are the probabilities of the next state given the current state. Although none of systems in the real world exactly follows this model, it does provide a good approximation of the reality.

In workload characterization, we sometimes need not only the number of service requests of each type but their order as well. Markov models provide a powerful means to deal with this requirement. When using Markov models in workload characterization, the states of the system can be defined in various ways according to the objective of the study. For instance, in certain cases we may view the service requests as the system states, where the next request can be determined by the present request. In others, we may view the execution of components as the states. If the system is executing component *i*, then we say that the system in state *i*. An example can be found in [Haring83], where a Markov chain is constructed at the task level. The nine different states of the system that correspond to the software resources used by the run are identified. The states include seven software resources employed (e.g., FORTRAN compiler, Editing, etc.) and two fictitious states (BEGIN and END). In real applications, the transition probability matrix can be worked out from the measured data. In the test workload we can simulate the probabilities with the help of a random number generator.

Detailed introductions to Markov models can be found in many books (See e.g. [Kleinrock75]). The descriptions of the technique used in workload characterization are given in more detail in [Ferrari83; Jain91]. And an application example is presented in [Agrawala78].

4.2.4 Parallel Workload Characterization Techniques

In the previous subsections, we described the main operations one should perform to model a workload, and a number of techniques that are traditionally applied to workload characterization. In this subsection, we survey some relatively new work on workload characterization for parallel architectures. Although the traditional techniques reviewed earlier can also be used, they are sometimes not sufficient and new techniques should be introduced. In what follows, we limit ourselves to the work which uses the techniques not yet reviewed in the previous subsection.

Parallel algorithms can be characterized in many ways. One example is the parallelism profile (see section 3.1), which is also useful in workload characterization, and will further be discussed later in this subsection. *Task graphs* are usually used for representing data-driven computations. A task graph is a directed graph which consists of nodes and arcs. Each node represents a sequence of computations (tasks), while the arcs represent data dependencies. Formally, a directed graph $G = (N, A)$ is defined as a finite nonempty set N of nodes and a collection A of ordered pairs of distinct nodes form N; each ordered pair of nodes in A is called a directed arc or simply arc. If (i, j) is a directed arc, then it is an outgoing arc from node i, and an incoming arc to node j [Bertsekas89].

From a task graph, Calzarossa and Serazzi [Calzarossa93] identify the following measures. They are *N*, *in-degree*, *out-degree*, *depth*, *maximum cut*, and *problem size*. Here *N* is the total number of nodes in the graph; the in-degree and out-degree are the average number of incoming arcs and outgoing arcs of all the nodes, respectively; the depth is the longest path between input and output nodes, which is directly related to the execution time of the algorithm; the maximum cut is the maximum number of arcs taken over all possible cuts, which reflects the maximum theoretical parallelism that can be obtained during the execution; and the problem size is a measure of the size of the data.

Task graphs for representing workload are widely used in the literature (see for example [Mak90]). [Calzarossa93] gives two examples to illustrate a workload characterization process, where two typical parallel algorithms are analyzed. One is the block decomposition matrix multiplication from [Fox88], and the other is the LU decomposition from [Lord83]. The asymmetric task graph of LU decomposition indicates a high probability the maximum number of processors (maximum cut) will be used only for a small fraction of the global execution time. A table shows all the values of the above measures for both of

the examples.

A task graph and its corresponding measures are useful in characterizing a parallel algorithm. They describe the complexity of the structure of an algorithm. However, what they capture are the inherent parallel characteristics of the algorithm and system-independent. Because of this, they are called static indices. As we have seen earlier, the performance of a parallel system is determined not only by the algorithm but by the underlying system architecture as well. Static indices are not sufficient in many cases, since they are lacking in representing the parallelism exploited by the algorithm on a given system and its behavior as the execution progresses. Consequently, dynamic metrics are introduced to characterize the behavior of a parallel system more succinctly. Metrics of this kind are appropriate for the evaluation of the match between algorithms and architectures.

In what follows, we examine some of the dynamic metrics and their applications to workload characterization.

The *parallelism profile* has been described in section 3.1. Here we only take a brief look at its applications to workload characterization. It gives the number of busy processors (or DOP) as a function of time, which can be theoretically derived assuming an ideal machine with an unbounded number of available processors. From the parallelism profile, we can obtain a number of inherent properties of a parallel program such as the average parallelism, the fraction of the sequential portion in the program, the variation of parallelism, and maximum and minimum parallelism. These statistical data are useful to characterize the workload. Since in practice the number of available processors are always bounded, and the bounded number of processors determines the algorithm behavior, fixing the maximum number of available processors is more practical. In this case, one can obtain the parallelism profile through measuring. Ghosal *et al.* [Ghosal91] give a parallelism profile example for the quicksort algorithm on a 16 transputer machine.

A parallelism profile can further be decomposed into a *computation profile* and a *communication profile*, which describe the number of processors performing computation and communication, respectively. Then the computation phase and communication phase can be identified, which are useful in characterizing workload.

For the convenience of analysis, the parallelism profile can be rearranged by accumulating the time spent at each degree of parallelism. The resulting cumulative plot is referred to as *shape* [Sevcik89]. The following figure shows the shape of the parallelism profile shown in Figure 1 (section 3.1).

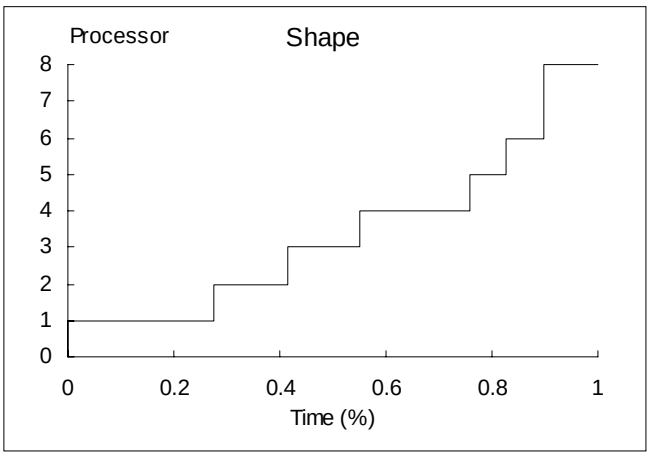

Figure 2. The shape derived from parallelism profile

A shape reflects more intuitively the interested properties (e.g., the average parallelism, the variation of parallelism, etc.) presented above. The average value can be effectively used to characterize the program

behavior if the variability is small. Otherwise, the distribution of the values is required. Two extreme values can be clearly seen in a shape, i.e., the period of time during which the number of busy processors is one (DOP = 1); and the period of time during which the number of busy processors is the maximum number of available processors. The first time reflects the fraction of execution time during which the algorithm can only be executed sequentially, while the second reflects the fraction of time during which the algorithm can be executed concurrently by all the available processors.

The *execution profile* is defined as the number of processors kept busy as a function of time, given a fixed number of available processors [Kumar88]. The execution profile is similar to the parallelism profile. The difference is that in the parallelism profile the number of available processors is unbounded. Periods of homogeneous processor utilization are manifested in execution profiles. These periods, called phases, can usually be correlated with the algorithms implemented in the underlying parallel code.

Two classes of phases are defined: *stationary phase* and *transitional phase*. A stationary phase is a contiguous subsequence of an execution profile which has roughly uniform processor activity. A transitional phase is a contiguous subseuqence of an execution profile which constitutes an abrupt change in processor activity [Carlson92; Carlson94]. Three special types of stationary phases are of interest. They are:

(1) a single processor is utilized;
(2) all processors are utilized;
(3) less than all, but more than one processor is utilized.

It is quite clear that identification and analysis of phases are important both in workload characterization and in scalability analysis. [Carlson94] proposes an algorithm for off-line detection of phases, and [Carlson92] describes three families of methods for smoothing execution profile data. They are useful in automatically detecting phases in execution profiles.

The *execution signature* [Dowdy88; Leuze89; Dowdy94] is defined as the execution time of the parallel algorithm, $T(p)$, as a function of the number of processors. It can further be divided into two components:

$$T(p) = T_{comp}(p) + T_{comm}(p)$$

where $T_{comp}(p)$ is the computation part, called the *computation signature*, and $T_{comm}(p)$ is the communication overhead, called the *communication signature*. Usually, $T_{comp}(p)$ is a monotonically decreasing function, while $T_{comm}(p)$ is an increasing function with respect to p. Several examples of these signatures for different algorithms are given in [Ghosal91]. From these examples, it can easily be seen that when p is small, the execution time is computation dominant, after p becomes greater than a certain value the execution time becomes communication dominant.

Ghosal *et al.* define the cost function as

$$C(p) = p\frac{T(p)}{T(1)} = \frac{p}{S(p)}$$

where $S(p)$ is the speedup. $C(p)$ incorporates the number of processors scaled by the speedup achieved for that many processors. A higher speedup implies a lower cost. Ideally, if $S(p) = p$ (a linear speedup), then $C(p) = 1$. On the other hand, if $S(p) = 1$ (no speedup), then $C(p) = p$, i.e., the cost equals the number of processors allocated. The *efficacy* $\eta(p)$ describes how well the processors are used. It is defined as

$$\eta(p) = \frac{S(p)}{C(p)} = \frac{S^2(p)}{p}$$

It increases to a maximum and then decreases, thus having a distinct maximum.

The *processor working set* (*pws*) is defined as the minimum number of processors that achieves the absolute maximum of $\eta(p)$. The *pws* coincides with the number of processors corresponding to the "knee" of the execution time-efficiency profile [Eager89]. Usually, $\eta(p)$ and $S(p)$ reach to the maximum at different values of p. Let $p_{allocated}$ be the number of processors allocated to a program. If $p_{allocated} \geq$ pws, then

the marginal benefit in allocating one more processor is less than the associated cost. The *pws* obtained from $\eta(p)$ maximizes the performance index *power*, which is defined as the ratio of the throughput to the response time [Kleinrock79].

Queueing network models are widely used in modeling computer systems. When using a queueing network, the workload is characterized by device demands that are placed on system resources by customers. In general, the amount of parameters used to characterize the workload is quite large. For example, for a queueing network with *K* devices and *N* customers, *N* * *K* parameters are needed [Dowdy90].

The costs of solving the network and collecting the parameters will be reduced, if the number of the parameters can be reduced. Pettey *et al*. [Pettey94; Pettey95] propose genetic algorithms (GAs) which group all the customers in the system into classes, according to the measured data. The customers in the same class exhibit roughly the same behavior. Because the customer demands are assumed to have exponential distributions, the traditionally used k-means method is no longer suitable. GAs are search techniques based on the principles of survival of the fittest and genetic recombination found in population genetics. It is shown that a GA can determine the demands quite accurately, given the number of classes. The technique is also effective at determining the number of classes.

In [Dowdy90], single customer classes are considered. The authors propose a new workload characterization technique that requires the *K* average device demands for the single-class counterpart model (of a multi-class queueing network). A segregation measure is developed. It is said that if an appropriate segregation measure can be found, the single-class counterpart model can be constructed and solved, and the resulting single-class throughput can then be modified to approximate closely the actual multi-class throughput.

5 Measurement

5.1 Overview

Measurement is one of the three techniques that are used in performance evaluation (see section 2.3). It is based on direct measurements of the system under study using a software or/and hardware monitor. Because of this, the term *monitoring* is interchangeably used (the subtle difference between the two terms is described in [Svobodova76].

The purpose of the measurements is to find information of interest, that can be used later to improve the performance. This process is often called *performance tuning*. For example, through monitoring, the frequently used segments of the software can be found. Optimizing the segments can improve the performance of the whole program. Similarly, the resource utilizations of the system can also be obtained, and performance bottleneck can be identified. That is useful in system tuning. Another purpose of the measurements, as has been mentioned in the previous sections, is that the measured data are often useful in modeling a system or characterizing workload. Measurement is also one of the approaches to validate a system model.

A monitor is a tool which is used to observe the activities on a system. In general, a monitor performs three tasks: data acquisition, data analysis, and result output. The data recorded by a monitor include hardware related data, e.g. processor usage throughout program run, message latency, and software data, e.g. process times, buffer usage, load balancing overhead [Jelly94]. Traditionally, monitors are classified as *software* monitors, *hardware* monitors, and *hybrid* monitors. Software monitors are made of programs that detect the states of a system or of sets of instructions, called *software probes*, capable of event detection. Hardware monitors are electronic devices to be connected to specific system points (the connections are made with probes) where they detect signals characterizing the phenomena to be observed [Ferrari83]. A hybrid monitor is a combination of software and hardware. Many examples of software monitors can be found in the literature (see e.g. [Plattner81; Power83; Malony92]). Examples of hardware monitors are

COMTEN and SPM [Ibbett78]. [Ries93] describes the Paragon performance monitoring environment that uses a hardware performance monitoring board. And examples of hybrid monitors are Diamond [Hughes80] and HMS [Hadsell83].

Each class of monitors has its own advantages and disadvantages. Selecting an appropriate monitor involves various aspects, e.g., cost, overhead, accuracy, availability, information level, etc.. In general, hardware monitors have received less attention than software monitors [Power83]. This has been shown by the fact that the number of existing software monitors are far greater than that of hardware monitors. Jain [Jain91] describes and compares software and hardware monitors in more detail. He also discusses some important issues that must be carefully considered by any monitor designer, and lists some terms that are frequently used in the literature. Some of them that will be used in the paper are given below.

An *event* is a change in the system state. Examples are process context switching, beginning of seek on a disk, and arrival of a packet. A *trace* is a log of events usually including the time of the event, the type of event, and other important parameters associated with the event.

Overhead is one of the key problems in designing a monitor. Any monitor more or less consumes system resources, or subtracts energy from the system being measured [Ferrari83]. For example, the measured data may be recorded on the secondary storage. And a software monitor needs a number of instructions to be executed by the CPU every time it is activated, thus consuming CPU time. This consumption of system resources is called overhead. The overhead depends on different situations. Sometime it may be quite large, and greatly affect the accuracy of the measurement results. Therefore, reducing the overhead to the minimum is one of the goals of monitor design.

Usually there are two ways to activate a monitor. One is using events. When an event occurs, the monitor is activated to capture the data about the state of the system. This gives a complete trace of the executing program. Because of this, this technique is also called *tracing*, and the monitor using this technique is sometimes called an *event-driven* monitor. The other method is using a timer. The monitor is activated by clock interrupts. This technique is known as *sampling*. One main problem with tracing is that it can produce too much data. This is particularly serious with parallel systems, which contain a great many processors. Most of the data provided are often not necessary. This makes it difficult for the user to focus on the only small amount of useful performance related data. Hence, tools become especially important to reduce the data and filter out all unnecessary data [Anderson89; Burkhart89].

Because of their importance, two monitors, *program monitors* and *accounting logs*, need to be mentioned here. These two monitors are frequently used in measuring performance. Accounting logs are often part of the system software. Although the primary aim is not for monitoring, they provide useful information about system usage and performance, and are used as software monitors [Jain91]. The main advantage is that they are built in, and need no extra effort to develop. The main disadvantage is that they are general-purpose, and do not provide information for specific measurements.

Program execution monitors (also known as program optimizers or program execution analyzers) are tools for measuring specific details about the execution of programs [Power83]. They help improve the performance of the programs by pinpointing where a program spends its time, finding out what program paths are actually executed, identifying what subroutines are called, and so forth. Besides, debugging is another important motivation [Burkhart89].

As has been seen from the above descriptions, monitors can be classified in many ways. For example, we can make the classification in terms of the implementation, triggering mechanism, functions, and so on. Because of the objective of this paper, in what follows we limit our focus on monitoring parallel systems.

5.2 Monitoring Parallel Software

Parallel software systems have their own features, which are different from those of sequential software. To measure such systems, more efforts must be made than to measure sequential ones. The problems described above also apply to parallel program monitoring. Besides, several new problems will arise.

Furthermore, the performance data we are interested in and should be measured in parallel systems are also different.

Jelly and Gorton [Jelly94] present two specific problems with parallel system measurement, i.e., the determination of a 'global' clock in order to effect the time stamping operation, and the fact that the monitoring process may significantly affect or perturb the dynamic behavior of the system being measured.

Malony *et al.* [Malony92] carefully examine the instrumentation *perturbations*. They analyze different sources of instrumentation perturbations. The primary source of the perturbation is the execution of additional instructions. However, ancillary perturbations include disabled compiler optimizations, memory conflicts, and additional operating system overhead. Collectively, these perturbations can increase the measured system's execution time, change memory reference patterns, reorder events, and even cause register interlock stalls.

The authors also determine the magnitude of performance perturbations as a function of instrumentation frequency and execution mode (sequential, vector, and parallel) by conducting a series of instrumentation experiments on the Alliant FX/80 vector multiprocessor. Their results show that perturbations in execution times for complete trace instrumentation can exceed three orders of magnitude.

The perturbations exist in every monitoring experiment, and are impossible to get rid of, thus minimizing the effect of perturbations is a key effort in conducting experiments. Based on the experimental data, they derived a perturbation model that can approximate true performance from instrumented execution. It is reported that with appropriate models of performance perturbation, these perturbations in execution time can be reduced to less than 20% while retaining the additional information from detailed traces.

The perturbation is with no doubt the primary problem, however, there exist several other problems that must be considered in designing tools that monitor parallel programs. There are many parallel programming models available on computing systems (see e.g. [Andrews91]), so tools should be highly flexible. In tuning applications system effects must be distinguished from application bottlenecks [Ries93].

The primary motivation of building multiprocessor systems is to cost-effectively improve system performance. As we have seen earlier, given a parallel architecture, how to effectively use the computing power is the main effort in programming. It is obviously not efficient to obtain only a small amount of parallelism while a large number of processors are available. In practice, however, large parallelism implies large overhead. This is a tradeoff faced by any parallel programmer. Consequently, measuring the parallelism of the parallel program becomes one of the main concerns in monitoring parallel software (see for example [Kuck74; Nicolau84; Kumar88; Chen89]). This helps the designer know which part of the program can effectively exploit the processors, and which part is the bottleneck.

Kuck *et al.* [Kuck74] report some early measurements of parallelism in ordinary FORTRAN programs. They statically analyze the programs, and then determine which statements can be executed in parallel. The parallelism available in a program is estimated by taking the average of the parallelism found on each trace in the program. The problem inherent in static analysis is the lack of the necessary performance data, such as loop iteration counts, and even the most careful estimates are often inaccurate [Power83].

Kumar [Kumar88] describes a software tools, named COMET (concurrency measurement tool), which is developed for measuring the total parallelism in large FORTRAN programs. Parallelism in a program is obtained by actually executing the program, monitoring the statements executed and the flow of values between them, and then deducing the maximum concurrency possible.

The program to be measured is first modified/extended automatically by COMET by inserting a set of *shadow variables* and *tracking statements*. There is one shadow variable for each variable in the original program for recording the time at which the current value of the variable was computed. The tracking statements are used to initialize and manipulate the shadow variables. The modified/extended program is then compiled and executed. In the monitoring process, the tracking statements trace the dynamic execution sequence of the original statements of the program.

Four large FORTRAN programs (SIMPLE, VA3D, etc.) are observed, and their execution profiles are

plotted. Average parallelism is also reported. From the profiles, some useful phases can be identified (e.g., sequential parts, parallel computations). The phases are helpful in performance tuning, performance prediction, and workload characterization.

The idea introduced in [Kumar88] is used and extended in [Chen89], which describes the MaxPar execution driven simulator used to measure and identify the inherent parallelism in real application programs. The concept of dependences is described in that paper, which is useful in measuring the inherent parallelism of programs.

Several techniques for detecting phases in execution profiles are proposed in [Carlson92; Carlson94] (see also section 4.2.4). Three special types of stationary phases are emphasized. For example, the phase during which a single processor is used, where the bottleneck of the program is likely located. Minimizing this phase may improve the performance. On the other hand, the phases during which all processors are utilized denote pure computation. Since there is no synchronization or communication in these phases (all processors are computing), it is assumed that these phases will scale linearly.

PICL (Portable Instrumented Communication Library) [Geist90; Geist90a] is used to generate execution trace information. Data obtained by using PICL are used to generate the execution profiles.

There are two problems with focusing purely on the parallelism. One is that spinning processes appear to be busy, but they are actually doing no useful work. The other is that it is different to relate the periods of poor parallelism to specific sections of code that can be changed [Anderson89].

The observations lead much work to focus both on the parallelism and on the synchronization. For example, a trace of interprocessor synchronization event with a timestamp of when the event occurred can be recorded, and the behavior of the program can be completely reconstructed from such a trace for the tuning purpose [Fowler88]. Monit [Kerola87] is a tool used to measure the behavior of higher-level objects, such as the number of busy processors or the number of threads waiting to enter each critical section.

Quartz [Anderson89] is a tool for tuning parallel program performance on shared memory multiprocessors, the idea of which is from gprof (a sequential UNIX tool). The tool efficiently measures just those factors that are most responsible for performance and relate these metrics to one another and to the structure of the program. Two mechanisms of thread (or lightweight process) synchronization are considered: lock (for mutual exclusion) and condition (or barrier). Both may cause the thread to wait. A new measurement is introduced, i.e., *normalized processor time*, which is defined as:

$$\sum_{i=1}^{P} \frac{\textit{Processor Time with i processors concurrently busy}}{i}$$

where P is the number of processors.

The procedures are then sorted by their normalized processor time plus that of the work done on their behalf, which may focus the programmer's attention on those areas of the program that have the greatest impact on performance. One of the advantages reported by the authors is that the tool not only provides guides of *where* improvements can be made but also *when* they can be done to improve performance. Indeed Quartz has many advantages, but, as has been pointed out by the authors, there are also some limitations. For example, when threads execute at the same time, it equally weights each even though only one is on the critical path.

Burkhart *et al.* [Burkhart89] describe an extensible, integrated family of monitoring tools (PEM) in project M^3. They propose a family of various monitoring facilities which can be used both in isolation and in combination. When used in multimonitor mode, several monitor tools operate simultaneously and independently, but their results are brought into relationship at the end of the experiment.

Their monitor system may be subdivided into two types of functional units: the universal "monitor base" and an extensible set of "monitoring agents". The monitor base is a centralized unit that has four interfaces, while the monitoring agents realize the actual monitor functions. Three monitors are described. They are trap monitor, mailbox monitor, and bus count monitor. They perform different types of monitoring. For example, a trap monitor can be used for debugging purposes, a mailbox monitor for

analysis of synchronization traffic, and a bus monitor for measurements of bus load.

One advantage of this system is that it provides an integrated environment consisting of the programming language and compiler, the configuration subsystem, the input/output management, the measurement database, and so on. Extensibility is another advantage, which makes it easy to add new monitors. The disadvantage is that it is based on a single user/single application environment. Multiple applications are not supported. Another problem with the implementation of such integrated environments is the cost. As reported by the authors, roughly three man-years have been spent in designing and implementing the monitor base and a dozen monitor tools.

6 Analytical Modeling

There are various performance modeling formalisms that can be used in modeling parallel systems. These formalisms can be classified into two classes: *deterministic* and *probabilistic* formalisms [Jonkers94]. In deterministic models, all quantities are fixed. In probabilistic models, some degree of uncertainty exists, and stochastic quantities are included in the models. Queueing networks and Petri nets, which will be discussed in this section, belong to the latter class.

Before reviewing the two formalisms for the performance evaluation purpose, we would like to mention two papers. One was written by Meyer [Meyer92], and one by Trivedi [Trivedi94]. Meyer surveys the performability issues and the history of the concept, and presents some pointers to the future. Trivedi addresses reliability, performance and performability evaluation tools and techniques from the user's perspective. He discusses different approaches to performance and reliability evaluation, which include non-state space, Markov reward models, stochastic Petri net models, and hierarchical and Approximate models. We highly recommend this two papers to the interested readers, both for the techniques and for the rich set of useful references.

6.1 Queueing Networks

Queueing theory is one of the most powerful mathematical tools for making quantitative analyses of systems. Here the systems have a general meaning (not only the computer systems). There are a wide spectrum of systems that can be analyzed using queueing theory. Telephone switching systems are an example. As a matter of fact, queueing theory was first developed to analyze the statistical behavior of such systems.

Using this modeling technique, one always faces a dilemma between the accuracy and efficiency. Sometimes, the resulting models may become too complex to solve, or in some cases even though the models are solvable, their solutions may be computationally too expensive. Fortunately, if the application domain is restricted to computer systems, this problem can be alleviated, and many benefits can be obtained. An appropriate subset of networks of queues can be selected, and evaluation algorithms can be designed to obtain meaningful performance measures with an appropriate balance between accuracy and efficiency [Lazowska84].

Queueing network modeling can be viewed as a small subset of the techniques of queueing theory, which is selected and specialized to model computer systems [Lazowska84]. Because of this, in this section, we restrict ourselves to the queueing networks and the solution techniques that are developed for modeling computer systems. Those readers interested in queueing theory and its applications are referred to [Kleinrock75]. Volume I of the book thoroughly introduces classical queueing theory, and volume II deals with applications (e.g., the analysis of computer systems and communication networks).

Queueing systems are used to model processes in which customers arrive, wait their turn for service, are serviced, and then depart. In order to characterize such systems, the following six components should be specified:

(1) The interarrival time probability density function.
(2) The service time probability density function.

(3) The number of servers.
(4) The amount of buffer space in the queues (or system capacity).
(5) The population size.
(6) The queueing discipline.

A short notation is often used in the queueing literature to specify these parameters. It is A/S/m/B/K/SD, each field of which corresponds to the six characteristics listed above, respectively. In general, if there is no buffer space or population size limitations and the queueing discipline is FCFS, we simply write A/S/m. The most widely used distributions for A and S are:

M — Exponential distribution (Markov)
D — All customers have the same value (Deterministic)
G — General (Arbitrary distribution)

For example, M/M/1 stands for a queueing system with the interarrival time and the service times exponentially distributed, and with only one server.

A queueing system is described by its states, which are often represented by the number of customers in the system (including the number in the queue and in the server). For example, if the system is in state *n*, the arrival of a new customer will cause the transition to state *n*+1. In the context of the performance evaluation of computer systems, the state of the system can actually be thought of as the number of packets in a router, the number of new calls in a telephone exchange computer, the number of client requests in a server, and so forth. As we will see, in analyzing a queueing system, finding the probability of the system being in state *n* (i.e., the probability of having exactly *n* customers in the system) is the key step. This probability is often referred as to *equilibrium probability*. Many other performance indices can then be derived from this probability.

6.1.1 Analysis of a Single Queue

In this subsection, we first take a brief look at systems having only one queue. Such systems can be used to model individual resources in computer systems.

A *birth-death process* is a Markov process in which the state transitions happen to the adjacent states only. It is the foundation of many queueing systems. The following figure shows the state transition diagram for a birth-death process.

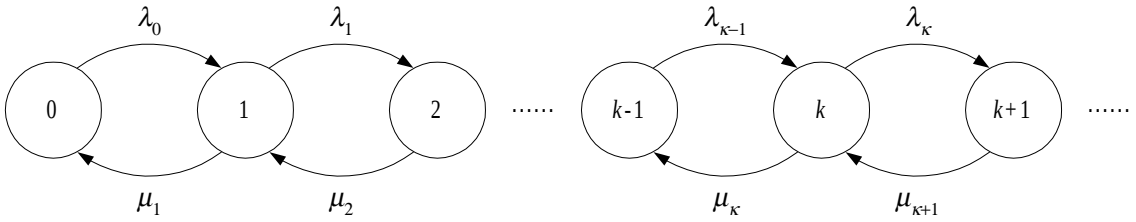

Figure 3. A state transition diagram for a birth-death process

When the system is in state *n*, it has *n* customers. The new arrivals take place at a rate λ_n. The service rate is μ_n. The following important equation gives the equilibrium probability p_n of a birth-death process:

$$p_n = \frac{\lambda_0 \lambda_1 \cdots \lambda_{n-1}}{\mu_1 \mu_2 \cdots \mu_n} p_0, \quad n = 1, 2, \ldots, \infty$$

where p_0 is the probability of being in state 0 (no customer in the system). Using the condition that the sum

of all probabilities equals 1, p_0 can be computed using only λ's and μ's. That is

$$p_0 = 1 \Big/ \left(1 + \sum_{n=1}^{\infty} \prod_{j=0}^{n-1} \left[\lambda_j / \mu_{j+1}\right]\right)$$

The M/M/1 queueing system is the most commonly used type of queues. It is illustrated in the following figure.

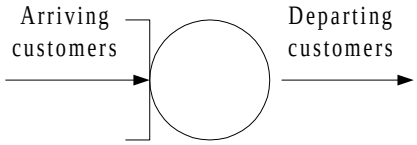

Figure 4. An M/M/1 queueing system

Its remarkable property is the memoryless property (because of the exponential distribution). This property makes the analysis quite easy, and also provides a reasonable approximation to the reality. The M/M/1 system is a special case of the birth-death process with the following correspondence:

$$\lambda_n = \lambda, \quad n = 0, 1, \ldots, \infty$$

$$\mu_n = \mu, \quad n = 1, 2, \ldots, \infty$$

Thus, we have

$$p_n = \rho^n p_0, \quad n = 1, 2, \ldots, \infty$$

where $\rho = \lambda/\mu$, which is known as the *traffic intensity*. To make the system *stable*, ρ must be less than 1, otherwise, the queue will grow without bound (*unstable*). After some substitutions, we have

$$p_n = (1-\rho)\rho^n, \quad n = 0, 1, \ldots, \infty$$

With this, we can have all the statistics about the system. For example, the server's utilization is given by the probability of having one or more customers in the system. That is $U = 1 - p_0 = \rho$. That is why ρ is sometimes called utilization by some authors (e.g., [Robertazzi94]). The mean number of customers in the system is given by

$$N = E[n] = \sum_{n=1}^{\infty} n p_n = \frac{\rho}{1-\rho}$$

Using Little's law [Little61], $N = \lambda T$, the mean response time, T, is computed by

$$T = \frac{N}{\lambda} = \frac{1}{\mu - \lambda}$$

Other statistics include the mean length of the queue, the mean waiting time, the idle time of the server, the busy period, and so on. They can also easily be computed using the above results.

The M/M/*m* queueing system is an extension of M/M/1 system, in which there are *m* identical servers and one queue. It can be used to model multiprocessor systems. Using the following correspondence:

$$\lambda_n = \lambda, \quad n = 0, 1, \ldots, \infty$$

$$\mu_n = n\mu, \quad 1 \le n \le m-1$$

$$m\mu, \quad n \ge m$$

we can easily get p_0 and p_n, and other useful statistics. One important performance parameter of this system is what is known as *Erlang's C formula*, that computes the probability that all the servers in the system are busy, or equivalently, the probability that a new customer has to wait in the queue.

An M/M/*m*/*B* system is similar to the M/M/*m* system. The difference is that the former has a buffer limitation. In this case, if a new customer arrives, and finds no buffer left, it leaves, and never comes back again. Because of this, this kind of systems are called *loss* systems. The analysis of such systems is quite similar to that of M/M/*m* systems.

Besides the above queueing systems, there are also other kinds of systems that are useful in modeling

computer systems. Examples are M/M/∞, M/G/1, M/G/∞, M/D/1, G/M/1, G/G/m, and so on. Their analyses can be performed in a similar way described above.

6.1.2 Queueing Networks

Single queueing systems are important in queueing theory, because they are fundamental. The analysis results are useful for other advanced systems. However, they are not powerful enough in modeling real computer systems, where there are often more than one queue required. In such cases, queueing networks should be used. A queueing network is a network of queues, where customers leaving a queue often arrive at another queue for another service.

Queueing networks can be classified into two types: *open queueing networks*, and *closed queueing networks*. An open queueing network has external arrivals and departures, whereas an closed queueing network is just the opposite. The analysis of queueing networks are often based on the product form solution.

Similar to the analysis of single queueing systems where the state probability is used, the joint state probability is useful in analyzing queueing networks. It is denoted as

$$P(n_1, n_2, \ldots, n_M),$$

where M is the number of queues included in the network, and n_1, n_2, ..., n_M are the numbers of customers in different queues. If the joint probability can be expressed as

$$P(n_1, n_2, \ldots, n_M) = \frac{1}{G(N)} \prod_{i=1}^{M} f_i(n_i)$$

then this network is a product form network. $G(N)$ is a normalizing constant, and $f_i(n_i)$ is some function of the number of customers. Product form networks are easier to analyze. Much work has been done to extend the product form solutions in order to use them in various networks. Some classical papers on this subject are [Jackson64], [Gordon67], and [Baskett75].

6.1.3 Fundamental Operational Laws

Buzen and Denning [Buzen76; Denning78] describe some influential laws, known as *operational laws*. These are general laws and can be applied to any networks, because there is no need to assume any distributions about the interarrival times and service times. The word operational means directly measured. The *operational quantities* are the quantities that can be directly measured during a finite observation period. Note that these quantities vary from one observation to another. However, certain relationships always hold in every observation.

The following table lists the notations of the operational quantities, with the explanations.

T	Length of an observation interval
A_k	Number of arrivals observed
C_k	Number of completions observed
λ_k	Arrival rate
X_k	Throughput
B_k	Busy time
U_k	Utilization
S_k	Service requirement per visit
N	Number of jobs in the system
Q_k	Number of jobs in the kth device
R_k	Response time per visit
Z	Think time of a terminal user
V_k	Number of visits per job

D_k	Total service demand per job

Here, the subscript k corresponds to device k in the system. The following relationships obviously hold.

$$\begin{aligned}
\lambda_k &= A_k/T \\
X_k &= C_k/T \\
U_k &= B_k/T \\
S_k &= B_k/C_k = U_kT/C_k \\
V_k &= C_k/C \\
D_k &= V_kS_k = B_k/C = U_kT/C
\end{aligned}$$

With the above fundamental relationships, we can get the following operational laws.

The Utilization Law:	$U_k = X_kS_k = XD_k$
Little's Law:	$N = XR$, or $Q_k = X_kR_k$
The Response Time Law:	$R = N/X - Z$
The Forced Flow Law:	$X_k = V_kX$

In getting these laws and the law to be presented later, there is an assumption used. That is *job flow balance* assumption. It assumes that the number of arrivals to a device equals the number of departures from the device.

6.1.4 Analysis of Queueing Networks

In this subsection, we consider *convolution algorithm* and *mean value analysis* (MVA). They are the most widely used product-form solution methods for queueing networks [Jonkers94].

Gordon and Newell [Gordon67] showed that the product form solution for the distribution of the number of customers at the service centers can be computed as

$$P(n_1, n_2, \ldots, n_M) = \frac{D_1^{n_1} D_2^{n_2} \cdots D_M^{n_M}}{G(N)}$$

where D_i is the total service demand per job for device i, n_i is the number of customers at the service center i, the numbers of the customers at all the centers add to N, and $G(N)$ is a normalizing constant such that the probabilities for all states sum to 1. That is

$$G(N) = \sum_n (D_1^{n_1} D_2^{n_2} \cdots D_M^{n_M})$$

where n represents all possible state vectors $\{n_1, n_2, \ldots, n_k\}$. Usually to make the value of $G(N)$ moderate (not too large or small), the D_i's should be scaled by some factor α. That is $y_i = \alpha D_i$. Thus we have

$$G(N) = \sum_n (y_1^{n_1} y_2^{n_2} \cdots y_M^{n_M})$$

or

$$G(N) = \sum_n \prod_{i=1}^{M} y_i^{n_i}$$

The problem with this formula is that to compute $G(N)$ we must enumerate all the permutations of possible states. The convolution algorithm is thus developed by Buzen [Buzen73] to solve this problem. The idea is based on the following fact. If

$$g(n,k) = \sum_n \prod_{i=1}^{k} y_i^{n_i}$$

then we can rewrite it as a recursive form:

$$g(n,k) = g(n,k-1) + y_k\, g(n-1,k)$$

where the initial values are

$$g(n,0) = 0, \quad n = 1,\ 2,\ \ldots,\ N$$

$$g(0,k) = 1, \quad k = 1,\ 2,\ \ldots,\ M$$

Obviously, $G(N)$ can be computed in NM steps. Once $G(N)$ is obtained, other performance statistics can easily be derived.

In many cases, the distribution is often not required, and what we need is only main performance values. Therefore, a simple analysis method can be used. That is MVA. For open queueing networks with *fixed-capacity service centers* (single server with exponentially distributed service time), we have

$$R_k = S_k(1 + Q_k)$$

Then, using the operational laws, we can have the following results.

$$Q_k = U_k / (1 - U_k)$$

and

$$R_k = S_k / (1 - U_k)$$

In *delay centers* (infinite servers with exponentially distributed service time), we have the similar, but rather simple results:

$$R_k = S_k$$

$$Q_k = U_k$$

Note that the queue length denotes the number of jobs currently being serviced since there is no need for them to wait in the queue (infinite servers available).

Mean value analysis (MVA) is developed to analyze closed queueing networks. The original paper was written by Reiser and Lavenberg [Reiser80]. It gives the mean performance in a similar way described above. The main idea is based on a recursive algorithm. Given a system with N customers, its performance is computed using the performance for $N - 1$ customers. The process goes on until the performance with no customers ($N = 0$) is reached, which can be easily obtained ($Q_k(0) = 0$).

The method for analyzing the network with fixed-capacity service centers is summarized below. The response time with N jobs can be computed by

$$R_k(N) = S_k[1 + Q_k(N - 1)]$$

This equation, along with the operational laws, allows us to obtain the following performance.

$$R(N) = \sum V_k R_k(N)$$

$$X(N) = N / [R(N) + Z]$$

$$X_k(N) = X(N) V_k$$

$$Q_k(N) = X(N) V_k R_k(N)$$

The equations also apply to delay centers except the response time, which is

$$R_k(N) = S_k$$

Once again, the queue length of delay centers means the number of jobs currently being serviced.

The problem with this recursive algorithm is that if N is large, it is computationally too expensive, especially in the case where the performance for smaller N's is not required. This problem can be solved using the iterative Bard/Schweitzer approximation of MVA [Bard79; Schweitzer79]. The algorithm is based on the assumption that as the number of jobs in a network increases, the queue length at each device increases proportionately. This way, the recursion in MVA can be avoided.

In addition to the basic methods described above, there are still other methods. LBANC (Local Balance Algorithm for Normalizing Constants) was inspired by MVA, and is also quite useful for analyzing queueing networks with a product form solution. It uses equations closely related to those of MVA but also uses the same normalizing constants as Convolution. It yields unnormalized mean queue lengths as intermediate results. With these, we can get $G(N)$, and then normalized mean queue lengths [Lavenberg83]. For solving very large queueing networks, if they satisfy certain conditions, they can first be decomposed

and then sovled. This method is known as *hierarchical decomposition* [Chandy75].

Interesting progress of queueing networks that has been made is the concept of *negative customers* which was developed by Gelenbe *et al.* [Gelenbe91a; Gelenbe91b]. In this new queueing paradigm, customers are classified into two types: positive customers and negative customers. A positive customer acts as the normal customers, but a negative customer does differently. When a new negative customer arrives at a queue, it decreases the queue length by one if there is at least one positive customer waiting in the queue. And it will be cleared from the network if the queue is empty. It should be noted that this paradigm only makes sense for open queueing networks.

This type of models finds its application in modeling the systems where the already issued requests need to be canceled. In this case, positive customers represent the requests, while negative customers represent the commands that terminate the requests. Another application might be in modeling communication network systems, where packets should be discarded after a certain age.

So far we have summarized and briefly described different queueing systems, including single service centers and queueing networks (both open and closed). Actually, these systems are most widely used in modeling and analyzing computer systems. We also addressed several techniques for solving the queueing network systems, which have product form solutions. For systematic descriptions on these models and their solutions, the interested readers are referred to [Lavenberg83; Lazowska84; Robertazzi94]. These books not only address the techniques but also different applications and ways to model various computer hardware and software systems.

To model parallel systems, the modeling technique must have the ability to describe synchronization between parallel processes. Two forms of synchronization can be distinguished: *mutual exclusion* and *condition synchronization* [Andrews91a]. Product form queueing networks can describe mutual exclusion but cannot express condition synchronization [Jonkers94]. Because of this, other techniques must be used together with queueing models. Usually, task graph models are applied, which are used to express condition synchronization.

In what follows we briefly look at some work done in predicting parallel performance. In fact much more work has been done in this area. Here, we only select several representative papers to show the applications. The research focuses are mainly on two aspects. One is on looking for different methods of describing systems under consideration, and the other is on finding cost-effective approximate solutions with respect to the corresponding modeling methods.

The work done by Heidelberger and Trivedi [Heidelberger82; Heidelberger83] is considered to be representative of earlier work using queueing network models to predict parallel performance. In their work, analytic queueing models of programs with internal concurrency are considered.

The system under investigation consists of m active resources such as processors and channels. The workload consists of a set of statistically identical jobs where each job consists of a primary task and zero or more statistically identical secondary tasks. The secondary tasks are spawned by the primary task sometime during its execution and execute concurrently with it, competing for system resources. The tasks execute independently of one another, and do not require synchronization, except for queueing effects.

The queueing network model of the system has two chains, one closed and the other open. The closed chain models the behavior of primary tasks, while the open chain models the behavior of secondary tasks. Because of the existence of concurrency within a job, the queueing network model of the system under study does not belong to the class of product form networks. An iterative technique is described for solving a sequence of product form queueing networks such that upon convergence, the solution to the product form network closely approximates the solution to the system under investigation. In [Heidelberger83], the primary tasks wait for their secondary tasks to complete; and the secondary tasks wait for all of their siblings to finish execution or merge with the parent (primary task) if all have finished execution. Two approximate solution methods for the performance prediction of such systems are developed. The problem with both papers is that only very simple task structures are considered. It is not enough to apply to parallel computations with more general task structures.

Mak and Lundstrom [Mak90] use task graph to model parallel computations. Resources in a concurrent system are modeled as service centers in a queueing network model. The two models are used as inputs to the prediction method which gives as output performance measures of the system, which include expected execution time of the parallel computation and the concurrent system utilization. Three types of service centers are supported. They are single-server, multiple-server, and infinite-server (or delay center). Given the application models, a two-step method is developed to determine the system completion time, i.e.

(1) Estimate the residence time for each task.
(2) Using the task residence times, reduce the precedence graph to determine system completion time, task starting, and ending times.

A case study is given, and the outputs obtained both by analysis and simulation are presented. The analysis of accuracy is also examined.

Relatively new research is reported in [Jonkers94]. A more systematic way, called the *Glamis* (GeneraLized Architecture Modelling wIth Stochastic techniques) methodology, is described by the authors. Like the work in [Mak90], two kinds of models are supported: machine models and program models.

A tuple notation is defined for a machine model. Every item in the tuple represents a queueing model building block. A general block is denoted by $Q_k^m(D)$, where *m* is the number of service centers in the block, *k* is the number of servers in every service center, and *D* is the service demand of a job on each of the service centers for one visit to the block. As for programs, *SPS-programs* are considered. The corresponding task graph is called *SPS-graph*. The class of SPS-graphs is a subclass of the well-known class of *series-parallel graphs* (SP). A model of an SPS-program consists of a sequence of descriptions of its parallel sections. A program section is characterized by its workload on the system resources. Because the service demand for one visit to a building block is specified in the machine model, only the visit count of a job to every block is needed to describe the workload. With the two models, an iterative algorithm is proposed to solve the models. The main performance measure is the completion time of a parallel program.

6.2 Petri Nets

6.2.1 Overview

Petri net theory is another formalism which is widely used in performance evaluation studies. One of the main advantages is that it supports graphical representation which greatly helps the modeler understand the behavioral aspects of the system by playing so-called “token games”. Many computational problems can be partitioned into a set of concurrently executing sequential processes. Proper synchronization must be used between these processes. Petri nets (PNs) are proposed, and further developed by many authors for the well-known fact that they are very well suited for the representation of concurrency, synchronization, communication, and cooperation.

Petri net research has significantly contributed to the performance evaluation field in recent years (see for example [Molloy82; Vernon85]); analysis and evaluation tools have been developed [Dugan85; Chiola92]. The simulataneous use of the formalism for analyzing qualitative and quantitative aspects of systems is approved [Balbo92]. Petri net models are also used to investigate the presence of desirable or undesirable properties, such as deadlocks, safety, and so on, in a system (e.g., [Leveson87; Murata89]). Leveson and Stolzy use a Time Petri net to analyze such properties as safety and fault-tolerance. Murata *et al.* detect deadlocks in Ada tasking programs using structural and dynamic analysis of Petri nets.

Research is also done to involve performance evaluation using PNs through the application design life-cycle so that design flaws can be discovered and corrected as early as possible, thus avoiding later costly modifications or even redesign. In this case, PNs tools are often integrated into development environments both for the specification of the systems under design and for the performance prediction [Buchholz92; Ferscha92; Gorton93]. Such environments have in common that they all support structured and hierarchical

design.

Much work has also been done to investigate the relationships of PNs with other formalisms and high level programming languages. Florin *et al.* [Florin91] describe the analogy between stochastic Petri nets and queueing networks. They give out the following equivalence:

- a place is equivalent to a queue. Tokens in a place are equivalent to customers in a queue.
- a transition is equivalent to a server.

This way, a grouped departure of customers corresponds to the weight on an arc which goes out from a place to a transition, while the arrival of customers corresponds to the weight of an arc which goes from a transition to a place. Some differences are also presented in that paper. For example, as has been mentioned earlier, queueing networks can only model mutual exclusion but not condition synchronization, whereas Petri nets can model both of them. The comparison with Markov chains is also described in terms of the complexity.

Mainly Ada and CSP-like programming languages are considered in researching their relations with Petri nets. The work is mainly concentrated on how the programs written in these languages can be modeled with PNs, and how Petri net models can be automatically translated into the language versions. For example, De Cindio and Botti [De Cindio91] investigate the equivalence between PNs and concurrent languages like CSP and CCS (Occam2 actually uses CSP models [Hoare85]). Instead, Taylor [Taylor83] and Shatz and Cheng [Shatz87] consider Ada. Taylor proposes a static analyzer for Ada programs. It inputs the program code and outputs the program call graph. Balbo *et al.* [Balbo92a] report part of a project for the development of a Petri net based programming environment for the DISC programming language that includes the CSP parallel constructs.

Although Petri nets have many good properties which make them superior to other formalisms (at least at some important aspects), they also have some serious problems. As noticed by many autohers (e.g., [Dutheillet89; Buchholz92; Granda92]), one main problem is that when the complexity of the parallel systems to be modeled increases, the corresponding reachability graph of the Petri net will become very huge (because of the explosion in the number of states of complex parallel systems). This makes the problem of analysis unmanageable, due to the size of the Markov process that is generated. In many cases, simulation techniques can be employed for the evaluation of the overall behavior of a model and for quantitative analysis, but they are computationally expensive and yield only approximative results. In addition, like testing simulation is not a formal analysis method, and thus cannot prove properties of a system [McLendon92].

To solve the problem addressed above is one of the major research efforts that have been made in this area. Many techniques have been proposed in order to reduce the dimension of the state space that must be analyzed to obtain the desired performance indices. Three main approaches are summarized by Granda *et al.* [Granda92]. They are

(1) to find subclasses of Petri nets which have a product form solution similar to that of queueing networks (see the last subsection), while the modeling capability of PNs (at least part of it) is still conserved. Examples can be found in [Marsan86; Balbo88].
(2) to employ colored Petri nets or high-level Petri nets.
(3) to use lumping techniques. [Granda92] is one of the examples, where lumping techniques are used to develop methods of evaluating the performance of parallel systems modeled by means of UGSPN.

In his Guest Editor's Introduction for a special issue on Petri net modeling of parallel computers, Chiola [Chiola92a] classifies the papers into the following four classes:

- Modeling methodologies;
- Modeling tools and formalisms;
- Parallel processing applications;
- Reduction techniques.

This actually reflects the current tendency in the Petri net research area.

Since Petri nets were first proposed by C. A. Petri in his Ph.D. dissertation, much work has been done

to extend and modify Petri's original models in order to apply them to solve various kinds of problems. One solid progress, we think, is introducing time features into standard Petri nets. Those Petri nets are known as *timed Petri nets*, one main advantage of which is the possibility of combining qualitative and quantitative analysis by using the same model for both purposes.

In the sequel, we first present standard Petri nets, the notation and the terminology. Then we describe timed Petri nets with different firing rules. We will restrict ourselves to *stochastic Petri nets* (SPNs) and *generalized stochastic Petri nets* (GSPNs), since they are the most widely used classes and have received much attention in the literature.

6.2.2 Standard Petri Nets

A Petri net is defined as a five-tuple: $\Phi = (P, T, I, O, \mu_0)$. P is a set of places; T a set of *transitions*; I an *input function*; O an *output function*; and μ_0 an *initial marking*. The input function I is a mapping from the transition t_i to a bag of places $I(t_i)$, the elements of which are called *input places*. Similarly, the output function O maps a transition t_i to a bag of places $O(t_i)$, the elements of which are called *output places*. Places may contain *tokens*. μ_0 specifies the initial placement of tokens on the places of the defined Petri net.

In the graphical representation of PNs, places are drawn as circles, transitions as bars, and tokens as black dots. *Arcs* that connect the input places to transitions are *input arcs*, and that connect transitions to the output places are *output arcs*.

A transition is *enabled* if each of its input places contains at least as many tokens as there exists arcs from that place to the transition. Any enabled transition may *fire* at will, removing all enabling tokens from its input places and depositing a token in each of its output places. The firing of a transition needs no time (the instantaneous firing feature). The placement of tokens in the net is changed by a firing, and thus a new marking is reached. The *reachability set* for a Petri net is defined as a set of all markings (states) that can be reached from the initial marking μ_0 by means of a sequence of transition firings. If two or more transitions are enabled at a certain marking, any one of them may fire. The choice of a transition to fire is indeterminate, which is why Petri nets are useful for modeling parallel applications. As a matter of fact, much research work has been done around this matter. Different firing rules are proposed to make the Petri net models suited for different modeling situations (we will discuss this later). For a formal definition, readers are referred to [Peterson81].

When Petri nets are used to model systems, the states of the system are defined by the markings. Transitions are used to represent events, and tokens in a place represent the condition that the corresponding transition fires (or the event occurs). An event can be either an atomic or non-atomic event. For example, it can be a single instruction, a sequence of instructions, or even a process. See for example [Ferscha92]. That is why Petri nets can be used to support hierarchical designs.

As an example, we define the following Petri net:

$P = \{P_1, P_2, P_3, P_4\}$
$T = \{t_1, t_2, t_3\}$
$\mu_0 = (1, 0, 0, 0)$
$I(t_1) = \{P_1\}$ $O(t_1) = \{P_2, P_3\}$
$I(t_2) = \{P_3\}$ $O(t_2) = \{P_1\}$
$I(t_3) = \{P_2, P_3\}$ $O(t_3) = \{P_4\}$

The corresponding graph is shown in Figure 5.

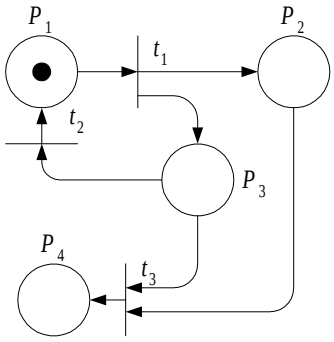

Figure 5. A sample Petri net

This net models a simple protocol, where one message is to be sent. It works as follows. When a message is sent, a timer is started. If the message is correctly received by the receiver, an acknowledgment is sent by the receiver back to the transmitter. If the transmitter does not receive the ACK before the timeout, it re-transmits the message, repeating the above process. Here, t_1 represents the event SEND_MESSAGE and START_TIMER, t_2 represents the event TIMEOUT, and t_3 represents the event ACK_RECEIVED, indicating that the sent message has been successfully received by the receiver.

The initial state of the system is μ_0. The condition that t_1 can fire is that there is a token in P_1. The firing of t_1 removes the token in P_1, and places one token in P_2 and one token in P_3. The firing condition of t_3 is that both P_2 and P_3 have a token, which defines the state that a message has been sent out and the ACK has not yet been received. At this state, t_2 and t_3 are enabled. If t_2 fires before t_3 (implying that either the sent message or the ACK is somehow damaged), the system returns to the initial state, and t_3 is disabled.

From this example, we see that t_2 and t_3 share the same input place, P_3. This means that the two transitions are in conflict: only one of them can fire, because the firing of one will disable the other. This property of Petri nets is useful to model mutual exclusion. The choice of t_2 and t_3 is non-deterministic. This class of PNs (standard PNs) does not specify the firing order of the transitions in conflict. In what follows, we describe other classes of Petri nets extended by introducing time into the standard Petri nets, where various firing rules are proposed.

6.2.3 Extended Petri Nets

Standard PNs can only be used for qualitative analysis of the modeled systems. In order to analyze quantitative properties, time must be introduced. In the literature, there have been several proposals for extending standard PNs to include time.

One of the first efforts in this direction is represented by E-nets [Noe73], which associates with each transition a fixed time to specify the delay between the enabling and the firing. Ramchandani [Ramchandani74] associated a firing finite duration with each transition of the net. The firing rule of standard PNs is modified to account for the time it takes to fire a transition. Another modification is that a transition must fire as soon as it is enabled. The resulting nets are called *timed Petri nets*.

Merlin [Merlin74; Merlin76] defined *time Petri nets* (TPNs) by using two values of time to define a range of delays for each transition. Two mappings are defined. They are Min: $T \rightarrow R$ and Max: $T \rightarrow R$, where R is the set of non-negative real numbers. $\text{Min}(t_i) \leq \text{Max}(t_i)$ for all i such that $t_i \in T$.

$\text{Min}(t_i)$ specifies the minimal time the transition t_i must wait (after it is enabled) before it can fire. $\text{Max}(t_i)$ specifies the maximum time during which the transition t_i can be continuously enabled without being fired and it must fire before the time has elapsed, unless it is disabled before its firing by the firing of another transition. One advantage of the model is that it retains the instantaneous firing feature of untimed Petri nets while also providing a very flexible modeling tool. Note that the TPN is more general than the

untimed PN. If we set the Min times of all transitions to zero, and the Max times to ∞, then these two models are equivalent. This approach has also been used by Berthomieu and Diaz [Berthomieu91]. They use an enumerative method to exhaustively validate the behavior of the model.

Razouk [Razouk83] used firing times along with enabling times. A transition fires after the enabling time has elapsed. During the firing of the transition, the tokens are "absorbed", and not available to other transitions. After the transition finishes firing, the tokens are placed into the output places, thus enabling other transitions.

Instead of associating time with transitions, time can also be associated with places [Sifakis77]. In fact, the models with transition delays and place delays are equivalent since one can be translated into the other [Leveson87]. In addition to the above, other proposals exist to associate time features with tokens (token-timed PNs) or even arcs (arc-timed PNs) [Ferscha95].

The stochastic Petri net (SPN), independently proposed by several authors [Ajmone-Marsan84], is another class of Petri nets. In this model, a random firing time is associated with each transition. SPNs play an important role in performance studies, because they are very powerful modeling tools. And what is more, they are closely related to Markov chains, which lay a solid foundation for the analysis of SPNs. Over the years, many authors have proposed various models of the stochastic Petri net. After careful examination, we can find that all the models are different only in the way that they modify the firing rule.

The changes are summarized as follows.

(1) A random firing time is associated with each transition. The distribution may be either arbitrary or exponential. The latter is generally more useful, because of the memoryless property.
(2) New transitions, named *immediate transitions*, are defined. They differ from classical transitions in that they must fire as soon as they are enabled. If there exist more than one immediate transitions that may be enabled simultaneously, a probability density function must be specified, according to which a firing transition can be selected. These transitions are often called *random switchs*.
(3) New arcs, such as *probabilistic arcs* and *inhibitor arcs*, are defined. A probabilistic arc looks like this:

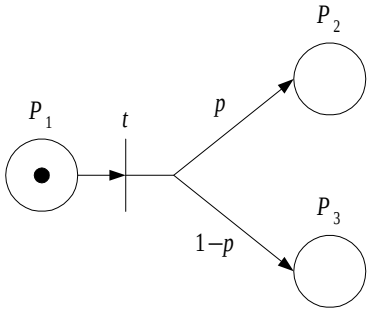

Figure 6. A stochastic Petri net with a probabilistic arc

The firing of transition t removes the token from P_1, and deposits a token either in P_2 with probability p, or in P_3 with probability $1 - p$. An inhibitor arc connects a place to a transition. It is drawn as a line terminating with a small circle rather than an arrowhead.

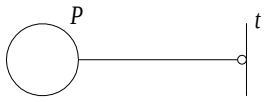

Figure 7. A Petri net with an inhibitor arc

Transition t is enabled only if P is empty.

In what follows, we present several examples of SPNs, which are generally considered to be

representative. The SPNs proposed by Florin and Natkin [Florin81] are obtained by associating with transitions arbitrarily distributed firing times. Under some conditions, the reachability graph can be regarded as a semi-Markov process. Dugan's ESPNs (Extended Stochastic Petri Nets) [Dugan84] uses additional features, such as immediate transitions, inhibitor arcs, and probabilistic arcs.

Molloy [Molloy82] uses exponentially distributed firing times associated with transitions. The author shows that the Petri nets so extended are isomorphic to continuous time Markov chains (MCs), thus can be solved as such. The markings of such a SPN correspond to MC states. The *sojourn time* in each state is also an exponentially distributed variable. This model is very useful and convenient. The only thing users need to do is to construct the models, which can then automatically be translated into MCs, and solved.

GSPNs (Generalized Stochastic Petri Nets) are proposed by Ajmone Marsan *et al*. [Ajmone-Marsan84], which are one of the most popular and powerful classes of SPNs. Based on the Molloy's SPNs, GSPNs are obtained by introducing some additional features. The transitions are partitioned into two subsets: *timed* and *immediate*. A timed transition has an exponentially distributed random firing time, which specifies the amount of time the transition must remain enabled before firing. An immediate transition fires in no time. The use of inhibitor arcs is also allowed, which may reduce the number of random switches. *Tangible markings* enable timed transitions only, and *vanishing markings* enable immediate transitions. The time spent in any vanishing marking is deterministically zero. The choice of timed transitions to fire next is slightly more difficult, which is determined by the firing rates (λ's) associated with the transitions. Informally, we can explain that as follows. Each time when a timed transition is enabled, a timer associated with it starts counting down at a constant speed. The initial value of the timer is obtained by sampling the corresponding random variable (which obeys an exponential distribution). The transition whose timer goes to zero first is the one selected to fire. The dynamic operation of a GSPN is equivalent to a continuous-time stochastic process. An example of using GSPNs in parallel program performance study can be found in [Balbo92a].

DSPNs (Deterministic and Stochastic Petri Nets) described in [Ajmone-Marsan87] are another class of SPNs, which allow both deterministic and random firing times, and thus are useful for many practical situations. The analysis tool has also been published [Lindemann94]. An application can be found in [Donatelli94].

Although SPNs have many advantages, there are several problems. A global 'must' firing rule changes the properties of the nets so that results for an identical untime Petri net need not be valid for the timed net [Buchholz92; Heiner94]. Another problem is the complixity of the underlying Markov chains. To solve the problems, higher level models (e.g., colored nets, and Pr/T nets) are proposed.

Recently colored Petri nets [Jensen81] receive more attention in the literature. For other versions of colored nets, we have colored SPNs and colored GSPNs, which are obtained by introducing token colors. Colors are used to model defferent behavior patterns of entities in a system, and different parts of a system with a similar or equivalent structure. Colored nets can be unfolded into the equivalent place-transition nets for the analysis purposes. Examples of colored GSPNs can be found in [Balbo92; Buchholz92]. The nets are solved by first unfolding the reprersentation into a GSPN, and then the usual manner can be used.

7 Simulation

7.1 Simulation Overview

Simulation is a widely used technique in performance evaluation. It provides a powerful way to predict performance before the system under study has not been implemented. It can also be used to validate analytical models, as is done in, for example, [Agarwal92; Menasce92].

There are a variety of simulations presented in the literature: emulation, Monte Carlo simulation, trace-driven simulation, execution-driven simulation, and discrete-event simulation.

An example of emulation is using one available processor (host processor) to emulate the instruction

set of another processor (target processor) which is not available or under design. This type of simulation is sometimes called by some authors *instruction-level simulation* [Covington91] or *cycle-by-cycle simulation* [Boothe93]. Programs written for the target processor are simulated by the simulator running on the host processor to study the behavior of the programs if they were executed on the target processor. The simulator is actually an assembly language interpreter. Because of its features, emulation can provide very detailed information about the behavior of the target processor if every single instruction of the target processor is simulated in great detail. And furthermore, it is also accurate in interleaving global events (we will discuss this later in this section). Those are advantages of emulation. However, this type of simulation is sometimes inaccurate in instruction timing when used to simulate more complex processors with pipeline. Another disadvantage is the low efficiency, because usually a number, sometimes even hundreds of, host instructions must be executed to emulate just one target instruction. This overhead is especially intolerable when we want to study parallel architectures with many processors. This alone is enough to limit the use of this method, because of which this paper will not discuss this methods in detail. Interested readers can find several examples in [O'Krafka89; Dellarocas91].

Monte Carlo simulation is a static simulation where the simulated systems do not change their characteristics with time. One application can be found in [Schweitzer93]. Computer systems are dynamic systems, and do not belong to this category.

A trace-driven simulation system consists of two components: an event generator (or trace generator) and a simulator. The event generator produces a trace of execution events, mostly addresses, which are used as input to the simulator. The simulator consumes the traced data and simulates the target architecture to estimate the time token to perform each event on the architecture under study. This technique has been widely used for the study of memory organizations, especially cache memories. Usually, the executions of programs are monitored to obtain traces, as done in most trace-driven simulation systems. However, the partitioning into two distinct components allows the event generator to be designed independently of the simulator, thus different techniques might be used. For example, models can also be used to produce synthetic traces.

Trace-driven simulations have many advantages. For example, a trace is generally obtained from real program executions, thus representing more accurate workload than other techniques; and quite often the simulation results have less variance, because a trace is a deterministic input [Sherman73].

Among the disadvantages of the technique, the most serious one is that it is difficult to simulate parallel architectures. When the trace-driven technique is used to study uniprocessor architectures, it provides accurate results, because the trace of a uniprocessor application does not depend on the simulated architecture. However, this is obviously not the case for the multiprocessor architectures, where the program execution path often differs on different architectures, or even varies from one execution to another on the same architecture. This means that the ordering of events and even which events occur cannot be guaranteed as recorded in the traces, thus invalidating the observed traces. This problem has been observed by many authors (e.g. see [Davis90; Covington91; Gefflaut93]). Gefflaut and Joubert show an example to illustrate it. In spite of this, the trace-driven technique has still been used in multiprocessor studies. Examples are [Eggers89; Kumar89; Holliday92]. In [Konas94], trace-driven simulation is incorporated together with other simulations, such as execution-driven simulation and critical path simulation, into an integrated simulation environment, call Chief.

Discrete-event simulation is used to simulate systems that can be described using discrete event models. There exist many examples of such systems. For instance, the arrivals and departures of packages at a network router can be modeled as a discrete event system. Discrete-event simulation is very well suited for studying queueing systems. Andrews and Olsson [Andrews93] present a simple example, where a simple multiprocessor architecture is simulated. The system consists of several processors competing to access a common memory bus. The technique is also powerful in simulating Petri nets. An example can be found in [Ferscha94], where timed Petri net performance models are studied using a Time Warp based scheme. Nance [Nance85] summarizes the following areas of interest in the discrete event simulation

community. They are:

- Random number generation;
- Modeling concepts;
- Simulation programming;
- Statistical analysis.

One of the drawbacks of simulation is the performance. Large simulations take enormous amounts of time on sequential machines, which limits the application of the technique. This problem gives rise to the research on parallel discrete event simulation (PDES). PDES can undoubtedly reduce the simulation time by partitioning the simulation among several processors, but it also introduces several new problems, such as deadlock. Two different approaches are proposed. One is known as *conservative* approach [Chandy79], and the other *optimistic* (or Time Warp) approach [Jefferson85]. Misra's survey paper [Misra86] describes the basic idea of PDES, and techniques of deadlock detection. Fujimoto's paper [Fujimoto90] surveys existing approaches and analyzes the merits and drawbacks of various techniques, and addresses a variety of amendments. These two papers are highly recommended to the interested readers, who want to know the principles, the existing problems, the techniques of solving the problems, and the state of the art.

For the limited space, we devote the rest of this section only to the execution-driven simulation. We do so because execution-driven simulation is a new technique, and to the best of our knowledge there is no review paper about this technique. In what follows, we survey several well-known execution-driven simulation systems, and review the techniques that are used in these systems.

7.2 Execution-driven Simulation

7.2.1 Overview

Execution-driven simulation is a relatively new simulation technique. It uses an available machine to emulate the program execution on some target machine. This way, information can be gathered, which reflects the behavior of abstract and unimplemented machines. This approach is very useful for system designers and programmers to study different algorithms, architectures, and the interactions between hardware and software. For example, different algorithms can be evaluated by simulating them on the same architecture, and the performance of the same algorithm can also be investigated on different architectures. Execution-driven simulation has many advantages over other simulation techniques such as trace-driven and instruction-level simulations.

To understand the execution-driven simulation, it is important to know *local events* and *global events*. A local event, or local computation, is private to a particular process, which is not visible to any other processes in the application. The occurrence of a local event does not affect the behaviors of other processes, and is also unaffected by the local events within other processes. Floating-point operations on local variables are such examples. A global event, sometimes called globally visible event, is such an operation that can be observed by some other processes. Global operations may change the execution path through the parallel application. Typically, there are two types of global operations: accesses to shared memories (e.g., read or write shared variables) and synchronization operations (e.g., message passing, barrier synchronization, etc.).

Execution-driven simulation views a parallel computation as consisting of global events separated by local computations [Davis90]. The basic idea of execution-driven simulation is to interleave the execution of the application with the simulation of the target architecture. The execution of the application is controlled by the simulator. The simulator only emulates global events on the target machine model, and lets the host CPU directly execute local computations, thus avoiding interpreting every single instruction of the target CPU, which would otherwise spend the majority of the simulation time. This explains the high efficiency of the execution-driven simulation, and is what differentiates the execution-driven simulation and the instruction level simulation. Covington *et al.* [Covington91] explain this differentiation in more

detail. However, execution-driven simulation is not suitable in the case where a new instruction set needs to be studied, instead this technique is geared towards higher level aspects of the target architecture.

Generally, execution-driven simulation systems work as follows. At the first, a process is scheduled to execute, and continuously performs its local computations. As soon as the process reaches some point where a global operation needs to be executed, it halts its execution, yields control to the simulator, and waits for an acknowledgment from the simulator to resume the execution. When the simulator gains the control, it simulates the operation issued by the process on the target architecture model, and then resumes the execution of the delayed process, or most likely other processes previously blocked to wait for the event to occur. An example of two processes operating on a shared variable to enter critical sections can be found in [Gefflaut93]. This scheduling policy is much like the way used in many lightweight thread packages.

To control the execution of the application, semaphores can be used. Each of the simulated processes has a semaphore associated with it. When a process attempts to execute a global operation, it simply blocks on the semaphore, and yields control to the simulator, which simulates the event, estimates the time needed, and then resumes the execution by incrementing the semaphore at an appropriate time. This scheme is used by Gefflaut and Joubert in SPAM [Gefflaut93].

The progression of time must be simulated as it would occur on the target architecture. Usually two schemes can be used. One is using a global system clock, which is complex and slow. The other is using local clocks. This approach is used by TANGO developed in Stanford University [Davis90]. There is one clock associated with each process in the application. The clock represents the elapsed time (the simulated time, not the actual execution time) of the corresponding process since the beginning of the simulation. The scheduler always elects a process with the smallest clock value to execute. That is why execution-driven simulation can ensure the correct ordering and timing of the events, which is more difficult for trace-driven simulation to achieve.

7.2.2 Code Augmentation

In order to make the application be able to interact with the simulator, and estimate the simulated time, the code must first be augmented. Code augmentation is a critical part of execution-driven simulation systems, because it directly affects the performance of simulation systems. Boothe [Boothe93] lists the following three purposes:

(1) Time counting: Code is inserted to estimate the execution time of each basic block (including local computations) if it was executed on the target machine.
(2) Statistics gathering: Code is inserted to collect such statistics as counts of the number of times that certain pieces of code are executed.
(3) Event call-outs: Code is inserted into points where global events need to be executed to call out to the simulator, which can then process the events.

In addition to the above, several new ideas about code augmentation are introduced in FAST developed in UC, Berkeley [Boothe93]. Two of them are worth mentioning: *in-line context switching* and *virtual registers*. In general, process context should be saved when the process is suspended, and restored when it is re-scheduled. The process context includes all the registers and other parameters that describe the state of the process. Context switching is quite time-consuming. The idea of in-line context switching is based on the observation that only a small number cycles are executed in a basic block before returning control to the simulator. During this period, only a few registers will be used and modified. Therefore, it is enough to load and store only those registers used in the basic block.

The idea of virtual registers is related to in-line context switching. Because not all the registers are loaded, a register (`r8` in the example of [Boothe93]) could be loaded into any unused physical register as long as the later use was also changed to use the same register. The benefit is that one can have more virtual registers than there are physical registers.

Code augmentation can be carried out at different code levels. Typically, we have two types. One is source code level augmentation, and the other is object code level augmentation. The source code level augmentation is usually done by the user when writing the application by calling library routines provided in the simulation engine. This method is adopted by many systems. Examples are PROTEUS [Brewer91; Brewer92] and SPAM [Gefflaut93], where a superset of C is used. Actually this is not code augmentation, because it is done directly by the user. It is so called by us because we think that in the ideal case, source code should be kept unchanged, and should be compiled and optimized in the original form as it would be written for a shared memory multiprocessor. The major problem with this method is that it will seriously affect the system's behavior, either directly or indirectly. For example, any change in the source code might also change the way the compiler uses the registers and optimizes the code, or more indirectly change the cache hits. PROTEUS even requires new operators for shared memory references to be simulated. In TANGO, static shared variables are not allowed for the convenience of simulation. This means that users have to dynamically allocate shared variables from the heap, and reference them indirectly through pointers. This forces users to change their programming style as would otherwise be used in writing their original applications.

The low level (object code level) code augmentation performs the augmentation on the object file, after the application has been compiled. Because of this, it can avoid the problems stated above, and the users can program in exactly the same way as they would do on the target machine. Note that the techniques such as in-line context switching and virtual registers described above can only be used with this level of augmentation. FAST is an example of successful use of this technique. PROTEUS also applies some code augmentations on the assembly language, but it is only for the purpose of timing basic blocks. The drawback of the low level augmentation is that it is difficult to implement, and less portable.

7.2.3 Performance

Basically two criteria should be considered to evaluate simulation systems, That is accuracy and efficiency.

The accuracy means how accurate the simulator can predict the time needed by the application if it was executed on the target machine. In execution-driven simulation, the time prediction involves the estimation of the time needed by the basic blocks of local instructions, and the time needed by the global operations. The two times are usually treated differently.

The time estimating for the global events is done by the simulator as it emulates the operations when the application yields control to the simulator at global events. This time greatly depends on the target architecture being simulated, and also on the abstraction level at which the architecture is modeled. The accuracy improves as more details are included in the model. Including more details, however, will affect the simulation efficiency, which we will discuss later in this section.

The direct execution of basic blocks of local instructions ignores the simulated time required to execute them. Therefore extra code should be added during the code augmentation which estimates the execution time. Several techniques can be used to achieve this.

PROTEUS uses assembly language augmentation to add code which increments a global cycle counter by the number of cycles required to execute that block. PROTEUS also allows users to add arbitrary monitoring or debugging code to the application, which needs extra time to execute, thus changes the behavior of the system. To solve this problem, the authors provide mechanisms to allow users to turn on and off cycle counting within cycle-counted code. Primitives are also designed to let users increase or decrease the cycle counter by a delta. The purpose of this is for users to adjust the simulated time. For example, users can subtract out the extra cycles due to the monitoring code. In practice, however, it is not easy for users to figure out the amount of time introduced by the extra monitoring or debugging code.

In Rice Parallel Processing Testbed (RPPT) [Covington91], a utility program called a *profiler* is designed to perform code augmentation for timing basic blocks. This approach, called *standalone* or *native profiling*, is used only when the instruction set of the target processor is the same as that of the host

processor. When the two instruction sets are different, as will happen in most cases, a *cross-profiling* technique is used. In this approach, compilers for a common source language for both the target and host machines are required. The assembly language outputs of the target and host compilers are called the target assembly language representation (TALR) and host assembly language representation (HALR), respectively. Timing analysis of basic blocks in the TALR is then made, and the cycle counts so obtained are assigned to the corresponding basic blocks in the HALR. This technique can obviously give higher accuracy because the target instruction set is taken into account. The problem is that there is not always a close correspondence between basic blocks in the two representations generated by different compilers. This may lead to some errors.

The efficiency is another main consideration in design simulation systems. The simulation time is often quite considerable, which becomes the main obstacle of the application of the simulation technique. The efficiency of a execution-driven simulator is often measured by *slowdown factor*, which is defined as the ratio of the time needed by the simulator to simulate the application to the execution time of the application if it was executed on the target machine.

The slowdown factor mainly results from the following three sources.

(1) Overhead incurred by code augmentation;
(2) Time required by the simulator to simulate global operations;
(3) Time required by context switching.

The influence on the slowdown caused by the overhead is quite small, but the latter two times cannot be overlooked. For example, the access time of shared variables cannot be directly predicted during code augmentation because of contention in the interconnection module and at the memory modules. Instead simulation routines must be called to perform detailed simulation. This is very time-consuming. Because of this, some execution-driven simulators are restricted to message-based systems. Examples are RPPT and SPAM.

Efforts are made by many authors to reduce the simulation time for global events. For example, PROTEUS is designed using the modular structure. That promotes multiple implementations of a given module, which allows users to switch between very accurate versions and very fast versions.

Context switching between the simulated processes and the simulator is quite frequent, thus fast context switching is very important to improve the performance. Using lightweight threads packages is a favorite choice by many authors (e.g. PROTEUS, RPPT, FAST, etc.). Because the threads run in the same address space, the time needed for context switching is much shorter than the time required by conventional heavyweight processes. TANGO was first designed using heavyweight process. New versions will use lightweight threads according to the authors.

8 Summary

This paper has systematically reviewed techniques used in evaluating parallel system performance, and surveyed various work done in this area.

To study performance, some general issues must first be considered. We must first choose some criteria called *metrics*. Different metrics may result in totally different performance values. To know metrics, their relationships and their effects on performance parameters is the first step. A system is often designed to work in a particular environment with some *workload*, so selecting proper workload is almost equally important.

Many statistical analysis techniques can be used in workload characterization:

- Averaging,
- Distribution Analysis,
- Principal component analysis,
- Clustering, and
- Markov models.

These techniques are common to both sequential and parallel systems. New techniques and metrics are introduced for parallel systems. They are:

- Task graph,
- Profiles and shape,
- Stationary phase and transitional phase,
- Execution signature, and
- Processor working set (pws).

After that, we can use one of the following three techniques to carry out the performance evaluation:

- Measurement,
- Analytical modeling, and
- Simulation.

The availability and precision of each technique depend on the stage during the system development life cycle. At the early design stage when the system has not yet been constructed, measurement is impossible, instead, a simple analytical model or simulation is practical. As the design process goes on, more details and knowledge are obtained; simulation or more sophisticated analytical modeling techniques could be used. Finally, when the system design has been completed and a real system constructed, measuring becomes possible. These techniques can be used together, thus making the evaluation results more convincing to the system's user.

Besides, some useful fundamental laws, such as speedup, isoefficiency, scalability, and so on, may be used in direct analysis of performance and scalability.

References

[Agarwal92] Agarwal, A., "Performance tradeoffs in multithreaded processors," IEEE Transactions on Parallel and distributed Systems, vol. 3, no. 5, pp. 525-539, September 1992.

[Agrawala78] Agrawala, A. K. and Mohr, J. M., "A markovian model of a job," Proc. CPEUG, pp. 119-126, October 1978.

[Ajmone-Marsan84] Ajmone Marsan, M., Conte, G. and Balbo, G., "A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems," ACM Transactions of Computer Systems, vol. 2, no. 2, pp. 93-122, May 1984.

[Ajmone-Marsan87] Ajmone Marsan, M. and Chiola, G., "On Petri nets with deterministic and exponentially distributed firing times," in: Rozenberg, G. (ed.), Advances in Petri Nets, Lecture Notes in Computer Science, vol. 266, no. 24, pp. 132-145, 1987.

[Amdahl67] Amdahl, G. M., "Validity of the single processor approach to achieving large scale computing capabilities," Proc. AFIPS Conference, Reston, VA, vol. 30, pp. 483-485, April 1967.

[Anderson89] Anderson, T. E. and Lazowska, E. D., "Quartz: a tool for tuning parallel program performance," Performance Evaluation Review, Special Issue, 1990 ACM SIGMETRICS, vol. 18, no. 1, pp. 115-125, May 1990.

[Andrews93] Andrews, G. R. and Olsson, R. A., The SR Programming Language: Concurrency in Practice, Benjamin/Cummings, Redwood City, California, 1993.

[Andrews91] Andrews, G. R., "Paradigms for process interaction in distributed programs," ACM Computing Surveys, vol. 23, no. 1, pp. 49-90, March 1991.

[Andrews91a] Andrews, G. R., Concurrent Programming: Principles and Practice, Benjamin/Cummings, Redwood City, California, 1991.

[Anonymous85] —, "A measure of transaction processing power," Datamation, vol. 30, no. 7, pp. 112-118, April 1985.

[Balbo88] Balbo, G., Bruell, S. C., and Ghanta, S., “Combining queueing networks and generalized stochastic Petri nets for the solution of complex models of system behavior,” IEEE Transactions on Computers, vol. 37, pp. 1251-1268, October 1988.

[Balbo92] Balbo, G., Chiola, G., Bruell, S. C. and Chen, P., “An example of modeling and evaluation of a concurrent program using colored stochastic Petri nets: Lamport’s fast mutual exclusion algorithm,” IEEE Transactions on Parallel and Distributed Systems, vol. 3, no., 2, pp. 221-240, March 1992.

[Balbo92a] Balbo, G., Donatelli, S. and Franceschinis, G., “Understanding parallel program behavior through Petri net models,” Journal of Parallel and Distributed Computing, vol. 15, pp. 171-187, 1992.

[Bard79] Bard, Y., “Some extensions to multiclass queueing network analysis,” in: Arato, M., Butrimenko, A. and Gelenbe, E. (eds.), Performance of Computer Systems, North-Holland, 1979.

[Barton89] Barton, M. and Withers, G., “Computing performance as a function of the speed, quantity, and cost of the processors,” Proc. Supercomputing’89, pp. 759-764, 1989.

[Baskett75] Baskett, F., Chandy, K. M., Muntz, R. R. and Palacios, F. G., “Open, closed, and mixed networks of queues with different classes of customers,” Journal of the ACM, vol. 22, no. 2, pp. 248-260, April 1975.

[Benner88] Benner, R. E., Gustafson, J. L. and Montry, G. R., “Development and analysis of scientific application programs on a 1024-processor hypercube,” SAND 88-0317, Sandia National Laboratories, Albuquerque, N. M., February 1988.

[Berthomieu91] Berthomieu, B. and Diaz, M., “Modeling and verification of time dependent systems using time Petri nets,” IEEE Transactions on Software Engineering, vol. 17, no. 3, pp. 259-273, March 1991.

[Bertsekas89] Bertsekas, D. P. and Tsitsiklis, J. N., Parallel and Distributed Computation: Numerical Methods, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Boothe93] Boothe, B., “Fast accurate simulation of large shared memory multiprocessors (revised version),” Technical Report UCB/CSD-93-752, Computer Science Division, University of California, Berkeley, June 1993.

[Brewer91] Brewer, E. A., Dellarocas, C. N., Colbrook, A. and Weihl, W. E., “PROTEUS: A high-performance parallel-architecture simulator,” Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, 1991.

[Brewer92] Brewer, E. A., Dellarocas, C. N., Colbrook, A. and Weihl, W. E., “PROTEUS: A high-performance parallel-architecture simulator,” Performance Evaluation Review, vol. 20, no. 1, pp. 247-248, June 1992.

[Buchholz92] Buchholz, P., “A hierarchical view of GCSPNs and its impact on qualitative and quantitative analysis,” Journal of Parallel and Distributed Computing, vol. 15, pp. 207-224, 1992.

[Burkhart89] Burkhart, H. and Millen, R., “Performance-measurement tools in a multiprocessor environment,” IEEE Transactions on Computers, vol. 38, no. 5, pp. 725-737, May 1989.

[Buzen73] Buzen, J. P., “Computational algorithms for closed queueing networks with exponential servers,” Communications of the ACM, vol. 16, no. 9, pp. 527-531, September 1973.

[Buzen76] Buzen, J. P., “Fundamental laws of computer system performance,” Proc. SIGMETRICS’76, Cambridge, MA, pp. 200-210, 1976.

[Calzarossa86] Calzarossa, M. and Ferrari, D., “A sensitivity study of the clustering approach to workload modeling,” Performance Evaluation, vol. 6, pp. 25-33, 1986.

[Calzarossa93] Calzarossa, M. and Serazzi, G., “Workload characterization: a survey,” Proceedings of the IEEE, vol. 81, no. 8, pp. 1136-1150, August 1993.

[Chandy75] Chandy, K. M., Herzog, U. and Woo, L., “Parametric analysis of queueing networks,” IBM Journal of Research and Development, vol. 19, no. 1, pp. 36-42, 1975.

[Carlson92] Carlson, B. M., Wagner, T. D., Dowdy, L. W. and Worley, P. H., "Speedup properties of phases in the execution profile of distributed parallel programs," in: Pooley, R. and Hillston, J. (eds), Computer Performance Evaluation - Modeling Techniques and Tools, Antony Rowe, Ltd., 1992.

[Carlson94] Carlson, B. M. and Wagner, T. D., "An algorithm for off-line detection of phases in execution profiles," in: Haring, G. and Kotsis, G. (eds.), Computer Performance Evaluation — Modeling Techniques and Tools (Proceedings of the 7th International Conference, Vienna, Austria), pp. 253-265, Springer-Verlag, May 1994.

[Carmona89] Carmona, E. A. and Rice, M. D., "A model of parallel performance," Technical Report AFWL-TR-89-01, Air Force Weapons Laboratory, 1989.

[Carmona91] Carmona, E. A. and Rice, M. D., "Modeling the serial and parallel fractions of a parallel algorithm," Journal of Parallel and Distributed Computing, vol. 13, pp. 286-298, 1991.

[Chandy79] Chandy, K. M. and Misra, J., "Distributed simulation: A case study in design and verification of distributed programs," IEEE Transactions on Software Engineering, vol. SE-5, no. 5, pp. 440-452, September 1979.

[Chen89] Chen, D. -K., "MaxPar: An execution driven simulator for studying parallel systems," Master's Thesis, University of Illinois at Urbana-Champaign, 1989.

[Chen93] Chen, P. M. and Patterson, D. A., "Storage performance — metrics and benchmarks," Proceedings of the IEEE, vol. 81, no. 8, pp. 1151-1165, August 1993.

[Chimento87] Chimento, P. F. and Trivedi, K. S., "The performance of block structured programs on processors subject to failure and repair," in: Gelenbe, E. (ed.), High Performance Computer Systems, North-Holland, Amsterdam, pp. 269-280, 1988.

[Chiola92] Chiola, G., "GreatSPN 1.5 software architecture," in: Balbo, G. and Serazzi, G. (eds.), Computer Performance Evaluation, pp. 121-136, 1992.

[Chiola92a] Chiola, G., "Special issue on Petri net modeling of parallel computers: guest editor's introduction," Journal of Parallel and Distributed Computing, vol. 15, pp. 169-170, 1992.

[Collier92] Collier, W. W., Reasoning about Parallel Architectures, Prentice-Hall, 1992.

[Covington91] Covington, R. G., Dwarkadas, S., Jump, J. R., Sinclair, J. B. and Madala, S., "The efficient simulation of parallel computer systems," International Journal in Computer Simulation, vol. 1, pp. 31-58, 1991.

[Crovella94] Crovella, M. E., "Performance prediction and tuning of parallel programs," Ph.D. thesis, Department of Computer Science, College of Arts and Science, University of Rochester, Rochester, New York, 1994.

[Davis90] Davis, H., Goldschmidt, S. R. and Hennessy, J., "TANGO: A multiprocessor simulation and tracing system," Technical Report CSL-TR-90-439, Stanford University, July 1990.

[De Cindio91] De Cindio, F. and Botti, O., "Comparing Occam2 program placements by a GSPN model," Proc. 4th International Workshop on Petri Nets and Performance Models, Melbourne, Australia, December 1991.

[Dellarocas91] Dellarocas, C. N., "A high-performance retargetable simulator for parallel architectures," Technical Report MIT/LCS/TR-505, MIT, June 1991.

[Denning78] Denning, P. J. and Buzen, J. P., "The operational analysis of queueing network models," Computing Surveys, vol. 10, no. 3, pp. 225-261, September 1978.

[Doherty70] Doherty, W. J., "Scheduling TSS/360 for responsiveness," AFIPS Proc. FJCC, pp. 97-111, 1970.

[Donatelli94] Donatelli, S., Franceschinis, G., Ribaudo, M. and Russo, S., "Use of GSPNs for concurrent software validation in EPOCA," Information and Software Technology, vol. 36, no. 7, pp. 443-448, 1994.

[Dongarra83] Dongarra, J. J., "Performance of various computers using standard linear equations software in a FORTRAN Environment," Computer Architecture News, vol. 11, no., 5, 1983.

[Dongarra92] Dongarra, J. J., "Performance of various computers using standard linear equations software," Technical report, Computer Science Department, University of Tennessee, 1992.

[Dowdy88] Dowdy, L. W., "On the partitioning of multiprocessor systems," Technical Report, Department of Computer Science, Vanderbilt University, Nashville, TN, 1988.

[Dowdy90] Dowdy, L. W., Krantz, A. T. and Leuze, M. R., "Using a segregation measure for the workload characterization of multi-class queuing networks," CMG'90 International Conference on Management and Performance Evaluation of Computer Systems, pp. 543-551, December 1990.

[Dowdy94] Dowdy, L. W. and Leuze, M. R., "On modeling partitioned multiprocessor systems," International Journal of High Speed Computing, vol. 6, no. 1, pp. 31-53, March 1994.

[Drummond73] Drummond, M. E., Evaluation and Measurement Techniques for Digital Computer Systems, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[Dugan84] Dugan, J. B., "Extended stochastic Petri nets: applications and analysis," Ph.D. thesis, Department of Electrical Engineering, Duke University, July 1984.

[Dugan85] Dugan, J. B., Ciardo, G., Bobbio, A. and Trivedi, K., "The design of a unified package for the solution of stochastic Petri net models," International Workshop on Timed Petri Nets, Torino, Italy, IEEE Computer Society Press, pp. 6-13, July 1985.

[Dutheillet89] Dutheillet, C. and Haddad, S., "Regular stochastic Petri nets," Proc. 10th Intern. Conf. Application and Theory of Petri Nets, Bonn, Germany, June 1989.

[Eager89] Eager, D. L., Zahorjan, J. and Lazowska, E. D., "Speedup versus efficiency in parallel systems," IEEE Transactions on Computers, vol. 38, no. 3, pp. 408-423, March 1989.

[Eggers89] Eggers, S. J., "Simulation analysis of data sharing in shared memory multiprocessors," Technical Report UCB/CSD89/501, University of California, Berkeley, 1989.

[Everitt74] Everitt, B., Cluster Analysis, Heinemann Educational Books, London, 1974.

[Febish81] Febish, G. J., "Experimental software physics" in: Ferrari, D. and Spadoni, M. (eds.), Experimental Computer performance Evaluation, North-Holland, Amsterdam, pp. 33-55, 1981.

[Ferrari78] Ferrari, D., Computer Systems Performance Evaluation, Prentice-Hall, 1978.

[Ferrari83] Ferrari, D., Serazzi, G. and Zeigner, A., Measurement and Tuning of Computer Systems, Prentice-Hall, Englewood Cliffs, NJ, 1983.

[Ferrari86] Ferrari, D., "Considerations on the insularity of performance evaluation," Performance Evaluation Review, vol. 14, no. 2, pp. 21-32, August 1986.

[Ferscha92] Ferscha, A., "A Petri net approach for performance oriented parallel program design," Journal of Parallel and Distributed Computing, vol. 15, pp. 188-206, 1992.

[Ferscha94] Ferscha, A. and Chiola, G., "Accelerating the evaluation of parallel program performance models using distributed simulation," in: Haring, G. and Kotsis, G. (eds.), Computer Performance Evaluation — Modelling Techniques and Tools (Proceedings of the 7th International Conference, Vienna, Austria), pp. 231-252, Springer-Verlag, May 1994.

[Ferscha95] Ferscha, A., "Performance Analysis of Parallel Systems," Tutorial Notes, HICSS-28, Hawaii, 1995.

[Florin81] Florin, G. and Natkin, S., "Evaluation Based upon Stochastic Petri Nets of the Maximum Throughput of a Full Duplex Protocol," in: Girault C. and Reisig W. (eds.), Informatik Fachberichte 52, Springer-Verlag, 1982.

[Florin91] Florin, G., Fraize, C. and Natkin, S., "Stochastic Petri nets: properties, applications and tools," Microelectronics and Reliability, vol. 31, no. 4, pp. 669-695, 1991.

[Flynn72] Flynn, M. J., “Some computer organizations and their effectiveness,” IEEE Transactions on Computers, vol. 21, no. 9, pp. 948-960, 1972.

[Fowler88] Fowler, R. J. and LeBlanc, T. J., “An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors,” Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, May 1988.

[Fox88] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J. and Walker, D., Solving Problems on Concurrent Processors, Vol. I, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Fujimoto90] Fujimoto, R. M., “Parallel discrete event simulation,” Communications of the ACM, vol. 33, no. 10, pp. 30-53, October 1990.

[Gefflaut93] Gefflaut, A. and Joubert, P., “SPAM: A multiprocessor execution driven simulation kernel,” Technical Report 1966, INRIA, Mars 1993.

[Geist90] Geist, G. A., Heath, M. T., Peyton, B. W. and Worley, P. H., “PICL: a portable instrumented communication library, C reference manual,” Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.

[Geist90a] Geist, G. A., Heath, M. T., Peyton, B. W. and Worley, P. H., “A users’ guide to PICL: a portable instrumented communication library,” Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, TN, August 1990.

[Ghosal91] Ghosal, D., Serazzi, G. and Tripathi, S. K., “The processor working set and its use in scheduling multiprocessor systems,” IEEE Transactions on Software Engineering, vol. 17, no. 5, pp. 443-453, May 1991.

[Gibson70] Gibson, J. C., “The Gibson mix,” IBM TR 00.2043, June 1970.

[Gelenbe91a] Gelenbe, E., “Product form networks with negative and positive customers,” Journal of Applied Probability, vol. 28, pp. 656-663, 1991.

[Gelenbe91b] Gelenbe, E., Glynn, P. and Sigman, K., “Queues with negative arrivals,” Journal of Applied Probability, vol. 28, pp. 245-250, 1991.

[Gordon67] Gordon, W. J. and Newell, G. F., “Closed queueing systems with exponential servers,” Operations Research, vol. 15, no. 2, pp. 254-265, 1967.

[Gorton93] Gorton, I., Jelly, I., and Gray, J., “Parallel software engineering with PARSE,” Proceedings of 17th International Computer Software and Applications Conference, Phoenix, USA, pp. 124-130, November 1993, IEEE

[Graham73] Graham, R. M., “Performance prediction,” Advanced Course on Software Engineering, Springer-Verlag, pp. 395-463, 1973.

[Grama93] Grama, A., Gupta, A. and Kumar, V. “Isoefficiency Function: A Scalability Metric for Parallel Algorithms and Architectures,” Technical Report, University of Minnesota, 1993.

[Granda92] Grander, M., Drake, J. M. and Gregorio, J. A., “Performance evaluation of parallel systems by using unbounded generalized stochastic Petri nets,” IEEE Transactions on Software Engineering, vol. 18, no. 1, pp. 55-71, January 1992.

[Gupta91] Gupta, A. and Kumar, V., “Scalability of Parallel Algorithms for Matrix Multiplication,” Technical Report TR 91−54, University of Minnesota, November 1991, (Revised April 1994).

[Gupta93] Gupta, A. and Kumar, V., “Performance Properties of Large Scale Parallel Systems,” Journal of Parallel and Distributed Computing, vol. 19, pp. 234-244, 1993.

[Gupta93a] Gupta, A. and Kumar, V., “The Scalability of FFT on Parallel Computers,” IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 8, August 1993.

[Gustafson88] Gustafson, J. L., “Reevaluating Amdahl’s law,” Communications of the ACM, vol. 31, no. 5, pp. 532-533, May 1988.

[Gustafson88a] Gustafson, J. L., Montry, G. R. and Benner, R. E., “Development of parallel methods for a 1024-processor hypercube,” SIAM Journal on Scientific and Statistical Computing, vol. 9, no. 4, pp. 609-638, 1988.

[Gustafson91] Gustafson, J., Rover, D., Elbert, S. and Carter, M., “The design of a scalable, fixed-time computer benchmark,” J. Parallel Distrib. Comput., vol. 11, pp. 338-401, August 1991.

[Hadsell83] Hadsell, R. W., Keinzle, M. G. and Milliken, K. R., “The hybrid monitor system,” Technical Report RC9339, IBM Thomas J. Watson Research Center, New York, 1983.

[Haring83] Haring, G., “On stochastic models of interactive workloads,” in: Agrawala, A. K. and Tripathi, S. K. (eds.), Performance’83, North-Holland, Amsterdam, pp. 133-152, 1983.

[Harman76] Harman, H. H., Modern Factor Analysis, University of Chicago Press, Chicago, 1967.

[Hartigan75] Hartigan, J. A., Clustering Algorithms, Wiley, New York, 1975.

[Heidelberger82] Heidelberger, P. and Trivedi, K. S., “Queueing network models for parallel processing with asynchronous tasks,” IEEE Transactions on Computers, vol. C-31, no. 11, pp. 1099-1109, November 1982.

[Heidelberger83] Heidelberger, P. and Trivedi, K. S., “Analytic queueing models for programs with internal concurrency,” IEEE Transactions on Computers, vol. C-32, no. 1, pp. 73-82, January 1983.

[Heiner94] Heiner, M., Ventre, G. and Wikarski, D., “A Petri net based methodology to integrate qualitative and quantitative analysis,” Information and Software Technology, vol. 36, no. 7, pp. 435-441, 1994.

[Helmbold90] Helmbold, D. P. and McDowell, C. E., “Modeling Speedup (*n*) greater than *n*,” IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 2, pp. 250-256, April 1990.

[Hoare85] Hoare, C. A. R., Communicating Sequential Processes, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[Holliday92] Holliday, M. A. and Ellis, C. S., “Accuracy of memory reference traces of parallel computations in trace-driven simulation,” IEEE Transactions on Parallel and Distributed Systems, vol. 3, no. 1, pp. 97-109, January 1992.

[Hughes80] Hughes, J. H., “Diamond — A digital analyzer and monitoring device,” Performance Evaluation Review, vol. 9, no. 2, pp. 27-34, 1980.

[Hunt71] Hunt, E., Diehr, G. and Garnatz, D., “Who are the users? An analysis of computer use in a university computer center,” AFIPS Conf. Proc. SJCC, vol. 38, pp. 231-238, May 1971.

[Hwang93] Hwang K., Advanced Computer Architecture: Parallelism, Scalability, Programmability, McGraw-Hill, New York, 1993.

[Ibbett78] Ibbett, R., “The hardware monitoring of a high performance processor,” in: Benwell, N. (ed), Computer Performance Evaluation, Cranfield Institute of Technology, UK, pp. 274-292, December 1978.

[ISE79] —, “CPU power analysis report,” Institute for Software Engineering, Menlo Park, Calif., 1979.

[Jackson64] Jackson, J. R., “Job shop like queueing systems,” Management Sciences, vol. 10, no. 1, pp. 131-142, 1964.

[Jain91] Jain, R., The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, John Wiley & Sons, New York, 1991.

[Jefferson85] Jefferson, D. R., “Virtual time,” ACM Transactions on Programming Languages and Systems, vol. 7, no. 3, pp. 404-425, July 1985.

[Jelly94] Jelly, I. and Gorton, I., “Software engineering for parallel systems,” Information and Software Technology, vol. 36, no. 7, pp. 381-396, 1994.

[Jensen81] Jensen, K., "Coloured Petri nets and the invariant method," Theoretical Computer Science, vol. 14, pp. 317-336, 1981.

[Jonkers94] Jonkers, H., "Queueing models of parallel applications: the Glamis methodology," in: Haring, G. and Kotsis, G. (eds.), Computer Performance Evaluation — Modeling Techniques and Tools (Proceedings of the 7th International Conference, Vienna, Austria), pp. 123-138, Springer-Verlag, May 1994.

[Karp90] Karp, A. H. and Flatt, H. P., "Measuring parallel processor performance," Communications of the ACM, vol. 33, no. 5, pp. 539-543, 1990.

[Kerola87] Kerola, T. and Schwetman, H., "Monit: A performance monitoring tool for parallel and pseudo-parallel programs," Proc. 1987 ACM SIGMETRICS Conference, May 1987.

[Kleinrock75] Kleinrock, L., Queueing Systems, Vol. I: Theory, Wiley, New York, 1975.

[Kleinrock79] Kleinrock, L., "Power and deterministic rules of thumb for probabilistic problems in computer communications," Int. Conf. on Communications, pp. 43.1.1-43.1.10, June 1979.

[Knuth71] Knuth, D. E., "An empirical study of FORTRAN programs" Software — Practice and Experience, vol. 1, pp. 105-133, 1971.

[Konas94] Konas, P., Poulsen, D. K., Beckmann, C. J., Bruner, J. D. and Yew, P.-C., "Chief: A simulation environment for studying parallel systems," International Journal of Computer Simulation, special issue on Computer Architecture Simulation, 1994.

[Kuck74] Kuck, D. J. *et al.*, "Measurements of parallelism in ordinary FORTRAN programs," Computer, vol. 7, pp. 37-46, January 1974.

[Kumar87] Kumar, V. and Rao, V. N., "Parallel depth first search. part II. analysis," International Journal of Parallel Programming, vol. 16, no. 6, pp. 501-519, 1987.

[Kumar88] Kumar, M., "Measuring parallelism in computation-intensive scientific/engineering applications," IEEE Transactions on Computers, vol. 37, no. 9, pp. 1088-1098, September 1988.

[Kumar89] Kumar, M. and So, K., "Trace driven simulation for studying MIMD parallel computers," 1989 International Conference on Parallel Processing, pp. I-68-I-72, 1989.

[Kumar94] Kumar, V. and Gupta, A., "Analyzing Scalability of Parallel Algorithms and Architectures," Journal of Parallel and Distributed Computing, 1994.

[Lavenberg83] Lavenberg, S. S. (ed.), Computer Performance Modeling Handbook, Academic Press, 1983.

[Lazowska84] Lazowska, E. D., Zahorjan, J., Graham, G. S. and Sevcik, K. C., Quantitative System Performance, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Lee80] Lee R. B., "Empirical results on the speed, efficiency, redundancy and quality of parallel computations," Proceedings of the 1980 International Conference on Parallel Processing, pp. 91-100, August 1980.

[Lee87] Lee, J., Shragowitz, E. and Sahni, S., "A hypercube algorithm for the 0/1 knapsack problem," Proceedings of International Conference on Parallel Processing, pp. 699-706, 1987.

[Leuze89] Leuze, M. R., Dowdy, L. W. and Park, K. H., "Multiprogramming a distributed-memory multiprocessor," J. Concurrency Practice, 1989.

[Leveson87] Leveson, N. G. and Stolzy, J. L., "Safety analysis using Petri nets," IEEE Transactions on Software Engineering, vol. SE-13, no. 3, pp. 386-397, March 1987.

[Lindemann94] Lindemann, C., "DSPNexpress: a software package for the efficient solution of deterministic and stochastic Petri nets," Perform. Eval., 1994.

[Little61] Little, D. C., "A proof for the queueing formula: $L = \lambda W$," Operations Research, vol. 9, no. 3, pp. 383-387, 1961.

[Liu82] Liu, M. T. and Rothstein, J., “Introduction: parallel and distributed processing,” IEEE Transactions on Computers, vol. C-31, no. 11, pp. 1033-1035, November 1982.

[Lord83] Lord, R. E., Kowalik, J. S. and Kumar, S. P., “Solving linear algebraic equations on an MIMD computer,” J. ACM, vol. 30, no. 1, pp. 103-117, 1983.

[Mailles87] Mailles, D. and Fdida, S., “Queueing systems with flag mechanisms,” in: Fdida, S. and Pujolle, G. (eds.), Modelling Techniques and Performance Evaluation, North-Holland, 1987.

[Mak90] Mak, V. W. and Lundstrom, S. F., “Predicting performance of parallel computations,” IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 3, pp. 257-270, July 1990.

[Malony92] Malony, A. D., Reed, D. A. and Wijshoff, H. A. G., “Performance measurement intrusion and perturbation analysis,” IEEE Transactions on Parallel and Distrubuted Systems, vol. 3, no. 4, pp. 443-450, July 1992.

[Marsan86] Marsan, M. A., Balbo, G., Chiola, G. and Donatelli, S., “On the product-form solution of a class of multiple-bus multiprocessor system models,” J. Syst. Software, vol. 1, no. 2, pp. 117-124, 1986.

[McLendon92] McLendon, W. W. and Vidale, R. F., “Analysis of an Ada system using coloured Petri nets and occurrence graphs,” in: Jensen, K. (ed.), Application and Theory of Petri Nets 1992, pp. 384-388, June 1992.

[Menasce92] Menasce, D. A. and Barroso, L. A., “A methodology for performance evaluation of parallel applications on multiprocessors,” Journal of Parallel and Distributed Computing, vol. 14, pp. 1-14, 1992.

[Merlin74] Merlin, P. M., “A study of the recoverability of computer system,” Ph.D. Thesis, Dep. Comput. Sci., Univ. of California, Irvine, 1974.

[Merlin76] Merlin, P. M. and Jarber, D. J., “Recoverability of communication protocols — Implications of a theoretical study,” IEEE Trans. Commun., vol. COM-24, no. 9, pp. 1036-1043, September 1976.

[Meyer92] Meyer, J. F., “Performability: a retrospective and some pointers to the future,” Performance Evaluation, vol. 14, pp. 139-156, 1992.

[Misra86] Misra, J., “ Distributed discrete-event simulation,” Computing Surveys, vol. 18, no. 1, pp. 39-65, March 1986.

[Molloy82] Molloy, M. K., “Performance analysis using stochastic Petri nets,” IEEE Transactions on Computers, vol. C-31, no. 9, pp. 913-917, September 1982.

[Mullender93] Mullender, S. (ed.), Distributed Systems (second edition), ACM Press, New York, 1993.

[Murata89] Murata, T., Shenker, B. and Shatz, S. M., “Detection of Ada static deadlocks using Petri net invariants,” IEEE Transactions on Software Engineering, vol. 15, no. 3, pp. 314-326, March 1989.

[Nance85] Nance, R. E., “Simulation classics (of the discrete event kind),” Performance Evaluation Review, vol. 13, no. 3-4, pp. 11-13, November 1985.

[Nicolau84] Nicolau, A. and Fisher, J. A., “Measuring the parallelism available for very long instruction word architectures,” IEEE Transaction on Computers, vol. C-33, no. 11, November 1984.

[Noe73] Noe, J. D. and Nutt, G. J., “Macro E-nets representation of parallel systems,” IEEE Trans. Comput., vol. C-22, no. 8, pp. 718-727, August 1973.

[Nussbaum91] Nussbaum, D. and Agarwal, A., “Scalability of parallel machines,” Communications of the ACM, vol. 34, no. 3, pp. 56-61, March 1991.

[Oed81] Oed, W. and Mertens, B., “Characterization of computer system workload,” Computer Performance, vol. 2, no. 2, pp. 77-83, June 1981.

[O’Krafka89] O’Krafka, B. W., “An empirical study of three hardware cache consistency schemes for large shared memory multiprocessors,” Technical Report UCB/ERL M89/62, Electronics Research Laboratory, University of California, Berkeley, May 1989.

[Park90] Park, A. and Becker, J. C., "IOStone: A synthetic file system benchmark," Computer Archit. News, vol. 18, no. 2, pp. 45-52, June 1990.

[Peterson81] Peterson, J. L., Petri Net Theory and the Modeling of Systems., Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Pettey94] Pettey, C. C., Wagner, T. D. and Dowdy, L. W., "Applying genetic algorithms to extract workload classes," CMG'94 International Conference on Management and Performance Evaluation of Computer Systems, pp. 880-887, December 1994.

[Pettey95] Pettey, C. C., Wagner, T. D. and Dowdy, L. W., "Using GAs to characterize workloads," (under review) International Genetic Algorithm Conference, 1995.

[Plattner81] Plattner, B. and Nievergelt, J., "Monitoring program execution: A survey," IEEE Computer, vol. 14, pp. 76-93, November 1981.

[Power83] Power, L. R., "Design and use of a program execution analyzer," IBM Systems Journal, vol. 22, no. 3, pp. 271-294, 1983.

[Quinn87] Quinn, M. J., "Designing efficient algorithms for parallel computer," McGraw Hill, New York, 1987.

[Ramchandani74] Ramchandani, C., "Analysis of asynchronous concurrent systems by timed Petri nets," Ph.D. thesis, Massachusetts Inst. Technol., Project MAC, Tech. Rep. MAC-TR-120, Feb. 1974.

[Rayfield88] Rayfield J. T. and Silverman, H. F., "System and application software for the Armstrong multiprocessor," IEEE Comput. Mag., vol. 21, pp. 38-52, June 1988.

[Razouk83] Razouk, R. R., "The derivation of performance expressions for communication protocols from timed Petri net models," Technical Report 221, Dep. Inform. Comput. Sci., Univ. of California, Irvine, November 1983.

[Reiser80] Reiser, M. and Lavenberg, S. S., "Mean value analysis of closed multichain queueing networks," Journal of the ACM, vol. 27, no. 2, pp. 313-322, April 1980.

[Ries93] Ries, B., Anderson, R., Auld, W., Breazeal, D., Callaghan, K., Richards, E. and Smith, W., "The Paragon performance monitoring environment," Proceedings of the conference on Supercomputing'93, pp. 850-859, 1993.

[Robertazzi94] Robertazzi, T. G., Computer Networks and Systems: Queueing Theory and Performance Evaluation (second edition), Springer-Verlag, New York, 1994.

[Rummel70] Rummel, R. J., Applied Factor Analysis, Northwestern University Press, Evanston, Ill., 1970.

[Sifakis77] Sifakis, J., "Petri nets for performance evaluation," in: Beilner, H. and Gelenbe, E. (eds.), Measuring, Modeling, and Evaluating Computer Systems (Proc. 3rd Symp., IFIP Working Group 7.3), North-Holland, Amsterdam, pp. 75-93, 1977.

[Sandberg85] Sandberg, R., Goldbert, D., Kleiman, S., Walsh, D. and Lyon, B., "Design and implementation of the Sun network file system," Proceedings of the Summer Usenix Conderence, 1985.

[Sauer81] Sauer, C. H. and Chandy, K. M., Computer Systems Performance Modeling, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Schweitzer79] Schweitzer, P., "Approximate analysis of multiclass closed networks of queues," Proceedings of International Conference on Control and Optimization, Amsterdam, 1979.

[Schweitzer93] Schweitzer, R., McHugh, J., Burns, P. J. and Zeeb, C., "Daylighting design via Monte Carlo with a corresponding scientific visualization," Proceedings of the conference on Supercomputing'93, pp. 250-259, 1993.

[Serazzi81] Serazzi, G., "A functional and resource-oriented procedure for workload modeling," Proc. Performance'81, North-Holland, Amsterdam, pp. 345-361, 1981.

[Serazzi85] Serazzi, G., “Workload modeling techniques,” in: Abu El Ata, N. (ed.), Modelling Techniques and Tools for Performance, Analysis, North-Holland, Amsterdam, pp. 13-27, 1985.

[Sevcik89] Sevcik, K. C., “Characterizations of parallelism in applications and their use in scheduling,” Proc. ACM SIGMETRICS and Performance’89, pp. 171-180, May 1989.

[Shatz87] Shatz, S. M. and Cheng, W. K., “A Petri net framework for automated static analysis of Ada tasking,” J. Systems Software, vol. 8, pp. 343-359, October 1987.

[Shein89] Shein, B., Callahan, M. and Woodbuy, P., “NFSStone — A network file server performance benchmark,” Proc. USENIX Summer Tech. Conf., pp. 269-275, 1989.

[Sreenivasan74] Sreenivasan, K. and Kleinman, A. J., “On the construction of a representative workload,” Communications of the ACM, vol. 17, no. 3, pp. 127-133, March 1974.

[Sherman73] Sherman, S. W. and Brown, J. C., “Trace-driven modeling: review and overview,” Proc. Symp. on Simulation of Computer Systems, pp. 201-207, 1973.

[Sun90] Sun, X. -H. and Ni, L. M., “Another view on parallel speedup,” Proceedings of Supercomputing’90, New York, 1990.

[Sun91] Sun, X. -H. and Gustafson, J., “Toward a better parallel performance metric,” Parallel Comput., vol. 17, pp. 1093-1109, December 1991.

[Sun93] Sun, X. -H. and Ni, L. M., “Scalable problems and memory-bounded speedup,” Journal of Parallel and Distributed Computing, vol. 19, pp. 27-37, 1993.

[Svobodova76] Svobodova, L., Computer Performance Measurement and Evaluation Methods: Analysis and Applications, Elsevier, New York, 1976.

[Trivedi94] Trivedi, K. S., Haverkort, B. R., Rindos, A. and Mainkar, V., “Techniques and tools for reliability and performance evaluation: problems and perspectives,” in: Haring, G. and Kotsis, G. (eds.), Computer Performance Evaluation — Modelling Techniques and Tools (Proceedings of the 7th International Conference, Vienna, Austria), pp. 1-24, Springer-Verlag, May 1994.

[Takahashi87] Takahashi, Yutaka, Takine, Tetsuya and Hasegawa, Toshiharu, “Throughput analysis of a hybrid protocol for ring networks,” in: Fdida, S. and Pujolle, G. (eds.), Modelling Techniques and Performance Evaluation, North-Holland, 1987.

[Taylor83] Taylor, R., “A general purpose algorithm for analyzing concurrent programs,” Communications of the ACM, vol. 26, May 1983.

[TPC89] TPC Benchmark A Standard Specification, Transaction Processing Performance Council, November 1989.

[TPC90] TPC Benchmark B Standard Specification, Transaction Processing Performance Council, August 1990.

[Van-Catledg89] Van-Catledge, F. A., “Towards a general model for evaluating the relative performance of computer systems,” International Journal of Supercomputer Applications, vol. 3, no. 2, pp. 100-108, 1989.

[Vernon85] Vernon, M. K. and Holliday, M. A., “A generalized timed Petri net model for performance analysis,” Proc. International Workshop on Timed Petri Nets, IEEE Computer Society Press, New York, pp. 181-190, July 1985.

[Wight81] Wight, A. S., “Cluster analysis for characterizing computer system workloads — Panacea or Pandora?,” Proc. CMG’81, New Orleans, LA, pp. 183-189, 1981.

[Worley90] Worley, P. H., “The effect of time constraints on scaled speedup,” SIAM Journal on Scientific and Statistical Computing, vol. 11, no. 5, pp. 838-858, 1990.

[Zorbas89] Zorbas, J. R., Reble, D. J. and VanKooten, E. R., “Measuring the scalability of parallel computer systems,” Supercomputing’89 Proceedings, pp. 832-841, 1989.