Performance Evaluation of Parallel Systems

Paolo Cremonesi [1]

Dipartimento di Elettronica e Informazione - Politecnico di Milano, Milano, Italy

Emilia Rosti [1]

Dipartimento di Scienze dell'Informazione - Università di Milano, Milano, Italy

Giuseppe Serazzi [1]

Dipartimento di Elettronica e Informazione - Politecnico di Milano, Milano, Italy and Università della Svizzera Italiana, Lugano, Switzerland

Evgenia Smirni [2]

Department of Computer Science - College of William and Mary, Williamsburg, VA, USA

Abstract

In this paper performance evaluation methodologies that have been applied in the analysis of parallel systems are reviewed together with the specific performance metrics. We concentrate on a few selected performance studies of parallel system components, i.e., processor, memory, interconnection network, input/output, and operating system. We demonstrate the utility of the performance evaluation methodologies for identification of system bottlenecks, performance forecasting, and future system design.

---

1 Introduction

Parallel computing has emerged as an indispensable tool for problem solving in many scientific domains during the course of the past twenty years as parallelism is the only avenue to achieve the required performance, which is vastly

[1] This work was partially supported by PQE2000 project and CNR CESTIA Center, Milano, Italy.
[2] This work was partially supported by a William and Mary Summer Research grant.

higher than what is possible with single processor systems only. Performance-driven studies have explored a wealth of alternatives with respect to all aspects of parallel systems aiming at improving the performance of the available systems by improving the performance of its individual components. Memory organizations have been investigated ranging from purely distributed memory to various levels of logically shared but physically distributed memory systems, with memory latency being a function of the data location. The advantages of software or hardware support for globally addressable memory providing various degrees of cache coherence have been considered. Additionally, significant research and development efforts have been put forth focusing on networking and storage systems that deliver high performance. In the area of interconnects, performance studies have driven the development of both highly specialized custom-made designs and generic high-speed networks. Input/output (I/O), being the last level of memory hierarchy, is another area where a variety of technologies have been studied to lessen the disparity between increases in processor speeds and memory access time.

The plethora of the available systems as well as their inherent complexity make performance evaluation of such systems a very challenging problem. The interaction of the system individual components (both hardware and software), as well as the complexity of the parallel applications further exacerbate the modeling and prediction problem. In this paper, we consider the role of performance evaluation methodologies for the design, analysis, and prediction of performance of parallel systems. To this end, we first present an overview of parallel workload characterization and its use in isolating and identifying hardware and software bottlenecks. Then, we concentrate on the various system components by viewing them in isolation. We illustrate the utility of different performance methodologies for the evaluation of each component via the form of self-contained case-studies.

This paper is organized as follows. Section 2 presents an overview of the role of performance evaluation and the utility of existing methodologies. In Section 3 we present an overview of the various architectural components of parallel systems. Operating systems components are reviewed in Section 4. Section 5 concludes this paper by summarizing the importance of performance evaluation as a scientific discipline for the analysis and design of parallel systems.

2 The Role of Performance

Quantitative evaluation and modeling of hardware and software components of parallel systems are critical for the delivery of high performance. Performance studies apply to initial design phases as well as to procurement, tuning, and capacity planning analyses. As performance cannot be expressed by quantities independent of the system workload, the quantitative characterization of resource demands of applications and of their behavior is an important part of any performance evaluation study. Among the goals of parallel systems performance analysis are to assess the performance of a system or a system component or an application, to investigate the match between applications requirements and system architecture characteristics, to identify the features that have a significant impact on the application execution time, to predict the performance of a particular application on a given parallel system, to evaluate different structures of parallel applications. In this section we briefly review the techniques most commonly adopted for the evaluation of parallel systems and the metrics used to characterize their performance and their workload.

2.1 Performance Evaluation Methodologies

Traditional performance methodologies, i.e., analytic modeling, simulation modeling, and experimental measurements naturally apply to the evaluation of parallel systems. Analytical techniques used for the evaluation of parallel system performance comprise queueing networks [15] and stochastic Petri Nets [2]. In order to extend the applicability of analytical techniques to the parallel processing domain, various enhancements have been introduced to model phenomena such as simultaneous resource possession, fork and join, blocking and synchronization. Hybrid modeling techniques allow to model contention both at hardware and software levels by combining approximate solutions and analytical methods [22]. However, the complexity of parallel systems limits the applicability of these techniques. Therefore, in spite of its computation and time requirements, simulation is extensively used as it imposes no constraints on modeling.

Evaluating system performance via experimental measurements is the third alternative for parallel system performance analysis. Measurements can be gathered on existing systems by means of benchmark applications that aim at stressing specific aspects of the parallel system. Even though benchmarks can be used in all types of performance studies, their main field of application is competitive procurement and performance assessment of existing systems. Parallel benchmarks extend the traditional sequential ones by providing a wider set of suites that exercise each system component with a targeted workload. The Parkbench suite especially oriented to message passing architectures [5] and the SPLASH suite for shared memory architectures [27] are among the most commonly used benchmarks.

2.2 Performance Metrics

Historically, the *execution* or *elapsed time* has been one of the most popular metrics used to refer to the performance experienced by a particular job when run on a given system. The elapsed time is a high level metric since it captures the effects of the characteristics of the job, of the system components (e.g., CPU power, memory bandwidth, I/O transfer rate) and of the workload mix and mode of execution (e.g., under a dedicated or multiprogrammed system).

In parallel environments we must consider a significantly higher number of factors than those in sequential environments. New complexity measures refer to the number of processors, to the communication exchanged among the processors, to the I/O activity, to the structure of the applications, and to the simultaneous execution of the different activities. The performance metric most commonly used in parallel domains is *speedup*, which gives a measure of the improvement of performance experienced by an application when executed on a parallel system. As elapsed time, speedup is a high level metric since it captures the combined effects of all the factors typical of parallel computations.

Several others metrics are used to evaluate specific components of parallel architectures (e.g., GFLOPS as a measure of the CPU computational power, MB/s as a measure of a channel bandwidth). However, because of the specificity of such measures, overall system performance cannot be easily derived from their individual values. For this reason, in this section we focus on high level metrics adopted to characterize the structure and the scalability of parallel applications.

2.2.1 Application Structure Metrics

Parallel applications can be characterized in different ways depending on the objective of the study for which they are used. *Task graphs* are a commonly used model of parallel applications. Such a representation contains complete information about the application inherent parallelism. Nodes represent independent portions of sequential code, and arcs represent node data dependency and control flow. A node starts executing when it has received data and/or control signals from all (possibly at least one) of its predecessor nodes. Communication with its successor nodes terminates a node execution. As an example, Fig. 1(a) illustrates the task graph of a parallel version of the LU decomposition algorithm that uses the Gaussian elimination method with partial pivoting. Tasks k, k search for the pivot, tasks k, j for $j = k+1, ..., n$ reduce the other rows by linear combination with the pivotal coefficients. The progressive reduction in the number of active tasks is evident in the rows becoming one element shorter at each step.

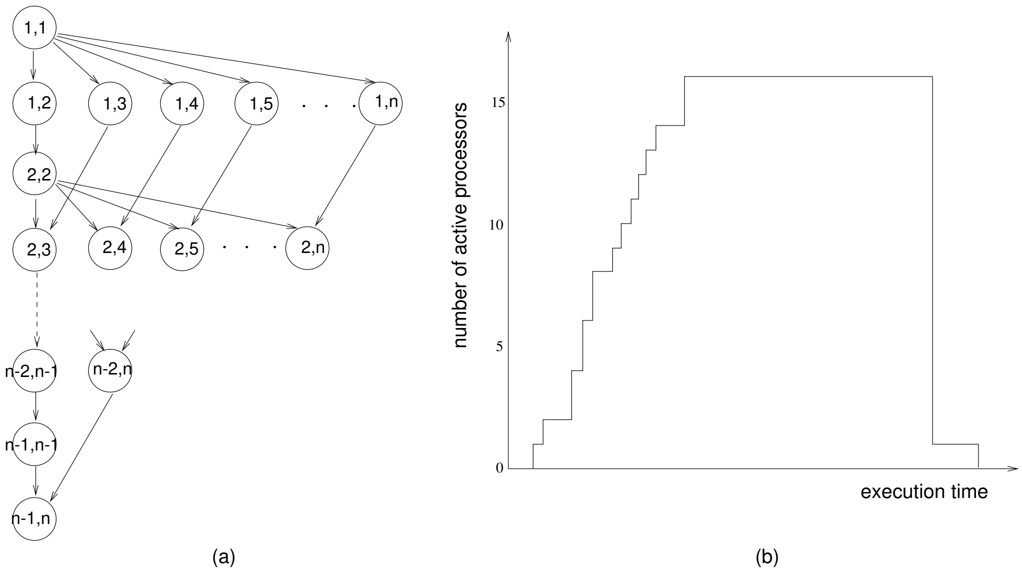

Fig. 1. Task graph (a) and a qualitative execution profile for a 16 processor system (b) for the LU decomposition algorithm.

Software performance metrics are divided into *static* and *dynamic* metrics according to the aspects of applications they describe [23]. Static metrics are derived from the model adopted for the description of the parallel application. They capture the application inherent characteristics with respect to parallelism and synchronization. Dynamic metrics are obtained from application execution. They summarize the application behavior when it executes on an actual or simulated system.

The *problem size* is a static metric that indicates the data set size of the application. Different parameters are used as problem size, depending on the nature of the problem. A set of static metrics derivable from the analysis of a task graph includes:

- *number of nodes*: total number of nodes in the task graph; it is a measure of the sequential complexity of the application and is an indicator of the computational granularity;
- *degree of synchronization*: average in-degree (the number of incoming arcs) and out-degree (the number of outgoing arcs) of the nodes; it is a measure of the synchronization complexity of the application;
- *longest path length*: number of nodes in the longest path; it may give a lower bound on the execution time;
- *maximum cut*: maximum number of arcs taken over all possible cuts from the input node to the output node; it represents the maximum parallelism in the task graph.

Dynamic metrics describe the run time behavior of an application in terms of concurrent activities on various processors. They capture the effects of the interaction between the application and the underlying parallel architecture. Unlike static metrics, dynamic metrics are bound to the particular system on

which the application is run. Some of the metrics listed below refer to communication activity, assuming a message passing environment. In a shared memory environment, parameters related to the access to shared variables should be considered instead. A set of dynamic metrics includes:
- *computation time*: total time consumed in computation;
- *communication time*: total time spent in communication;
- *execution time*: total time to complete the execution;
- *communication volume*: number of messages and size of data exchanged among the processors;
- *input/output volume*: number of I/O operations and data volume transferred to/from the file system;
- *average parallelism*: mean number of busy processors, either computing or communicating;
- *execution profile*: number of busy processors as a function of the execution time (see Fig. 1(b) for a qualitative example).

2.2.2 Scalability Metrics

Scalability metrics describe the application characteristics in terms of relative gain or loss in performance as a function of the number of the allocated resources. A popular measure of how effectively an application uses a given parallel system is *speedup*. Let T_1 and T_p be the elapsed time of an application when executed on a single processor and on p processors of a multiprocessor system, respectively. The speedup of an application as a function of the number p of processors, is given by:

$$S_p = \frac{T_1}{T_p} \tag{1}$$

When T_1 is either the elapsed time required to execute the *best* sequential version of the application or the elapsed time required by a sequential version of the application considered as *benchmark*, speedup measures the absolute merits of a parallel algorithms with respect to the "optimal" sequential version or with respect to a sequential "benchmark". When T_1 is the elapsed time required to execute the given application on a single processor of a p-processor system, speedup is a measure of the relative merit of an application with respect to different numbers of processors of a given parallel system. In what follows we adopt the latter definition of speedup since it is commonly used for the evaluation of actual applications.

In [3] simple arguments concerning the limits of speedup from parallelization were proposed. In its original form, Amdahl's law states that speedup is

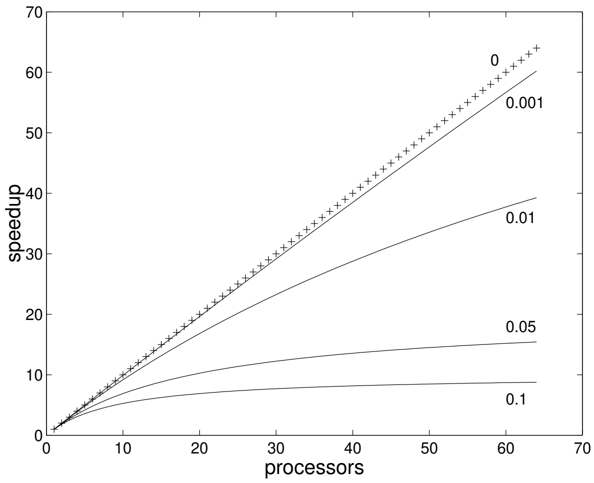

Fig. 2. Speedup curves S_p obtained with Eq. 2 for different values of f_{seq}.

bounded by:

$$S_p \leq \frac{1}{f_{seq} + (1 - f_{seq})/p} \tag{2}$$

where f_{seq} is the fraction of elapsed time that is inherently sequential. Figure 2 presents various speedup curves that are obtained using Eq. 2 for different values of f_{seq}. Note that very small values of f_{seq} are required to achieve significant speedups since an application cannot be executed faster than its sequential part. Ideally, when $f_{seq} = 0$, the speedup is linear, i.e., $S_p = p$. Two upper bounds on speedup can be identified from Eq. 2 that are expressed in terms of both the structure of the application by the parameter f_{seq} (software bound) and the characteristics of the architecture by the number p of processors (hardware bound)[7].

Other scalability metrics can be derived from speedup. We briefly review two of them, namely *efficiency* E_p and *efficacy* η_p, defined as:

$$E_p = \frac{S_p}{p} \quad , \quad \eta_p = \frac{S_p^2}{p} \tag{3}$$

Efficiency represents the fraction of time the p processors are usefully employed, or, equivalently, the speedup per allocated processor. The maximum value of efficiency (i.e., $E_p = 1$) is achieved when the speedup is linear. Efficacy takes into account both the gain and the cost of a computation [9]. The value of p corresponding to the maximum of η_p is the processor working set of the application, that is, the number of processors that maximizes the speedup per unit of cost, and identifies an optimal operating point.

With the evolution of parallel computing, applications manipulating large volumes of data have become common. With I/O intensive software, the I/O power of the system can significantly affect the application performance. Parallel file systems manage application I/O activities by taking advantage of disks that can operate in parallel. In order to take into consideration both computation and I/O scalability, new definitions of the performance indices described above are needed.
The speedup, defined as the ratio of the elapsed time $T_{1,1}$ on a single processor and a single disk to the elapsed time $T_{p,d}$ on p processors and d disks, is given by:

$$S_{p,d} = \frac{T_{1,1}}{T_{p,d}} \tag{4}$$

which represents a *speedup surface* in the space $(p, d, S_{p,d})$. Similar extensions are derived for efficiency and efficacy.
A generalization of Amdahl's law (Eq. 2) to the three dimensional space to include processors and disks can be derived from Eq. 4 [24]. Define f_{seq}^{CPU} and $f_{seq}^{\mathrm{I/O}}$ to be the sequential fractions of the computational and I/O components of the elapsed time, respectively, and f_{par}^{CPU} and $f_{par}^{\mathrm{I/O}}$ to be the corresponding parallel fractions. The speedup can be described by:

$$S_{p,d} = \frac{1}{f_{seq}^{\mathrm{CPU}} \; + \; f_{par}^{\mathrm{CPU}}/k'(p,d) \; + \; f_{seq}^{\mathrm{IO}} \; + \; f_{par}^{\mathrm{IO}}/k''(p,d)} \tag{5}$$

where $k'(p,d)$ and $k''(p,d)$ are scaling functions that allow to consider the different sources of overhead (e.g., communications, bottlenecks, synchronizations). Figure 3 illustrates the measured speedup behavior for a chemistry applications, namely QCRD (Quantum Chemical Reaction Dynamics) [28]. The figure shows that speedup does not necessarily increase as the numbers of allocated disks and processors increase, indicating the presence of severe bottlenecks on both resources.

3 Architectural Components

In this section we focus on performance issues associated with the various architectural components, i.e., processors, memories, interconnection network, and I/O devices.

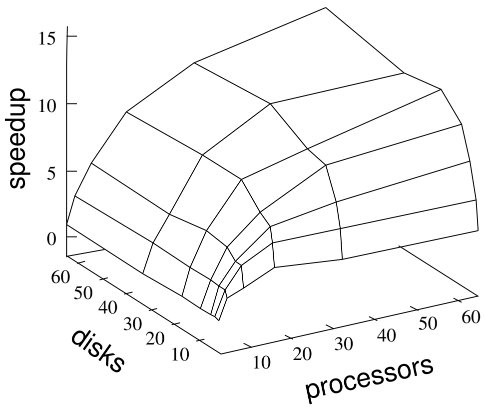

Fig. 3. Speedup surface of QCRD, a computational chemistry application.

3.1 Historical perspective

High-end single vector processor architectures and parallel computers based on slower but cheaper processors dominated the early days of parallel computing The first parallel computers, built at the end of the 70s, were all fine grain Single Instruction Multiple Data (SIMD) machines, with hundreds of single-bit processors. In the early 1980s Multiple Instruction Multiple Data (MIMD) systems dominated the parallel computing arena. Such systems were shared-memory, bus-connected multiprocessors with moderate parallelism (i.e., in the order of tens). Both these SIMD and MIMD machines were much slower than the first supercomputer available at that time, the single vector processor CRAY-1. New fine-grain SIMD systems were introduced at the end of the 80s, with the most notable being the Thinking Machines CM-2, equipped with 64K one-bit processors, that outperformed the fastest vector systems for certain computations. However, the range of applications that could run efficiently on the CM-2 was limited. In the early 90s, several massively parallel MIMD systems, based on commodity off the shelf (COTS) microprocessors with distributed memory were developed, powered by the availability of high speed interconnects.

Several factors drive recent design trends in parallel architectures. The performance of commercial microprocessors has advanced at tremendous rate and the gap between COTS components and vector processors has been greatly reduced. With symmetric multiprocessor (SMP) technology a small number of processors share one or more memory banks on a common bus. The need to exploit all the levels of parallelism in an application (e.g., instruction level, task level, process level) leads toward a shared-space programming model, in which application threads can access all of the virtual memory available.

3.2 Processors

Performance of COTS microprocessors has been increasing at a rate of nearly 50% per year since the mid 80's, while dedicated supercomputer microprocessors performance has been increasing at a rate of roughly 25% per year. As a result, the gap between off-the-shelf and dedicated microprocessors is rapidly decreasing and is projected to disappear within the next decade. The evolution of COTS microprocessor architecture is driven by two forces: the increasing number of transistors per processor chip, and the increasing gap between processor speed and memory access time.

The primary technology advance is a reduction in the basic VLSI feature size. If we look at the raw computing power of a chip (total transistor switching per second), transistor capacity has increased more than an order of magnitude comparing to clock rate over the past two decades. The simplest analysis of this technological trend suggests that the basic single-chip building block will provide increasingly larger capacities, so that more components can be packed on the chip (e.g., memory).
Divergence between capacity and speed is more pronounced in memory technology. From 1985 to 1995, the capacity of DRAM chips has increased a thousand-fold, quadrupling every two years, while the memory cycle time has improved only by a factor of two. Processor design attempts to exploit the huge numbers of available transistors to reduce performance degradation due to memory latency, which ranges from few nanoseconds for a first level on-chip cache to milliseconds for mass memory, i.e., I/O devices. *Caches* are used to reduce the memory access latency, and *thread-level* parallelism and *speculative* execution allow the processor to overlap memory access time with other useful work.

The IA-64 architecture: a case study
Multi-threading architectures provide special hardware to support thread level parallelism in which individual threads can simultaneously exploit instruction-level (ILP) parallelism. With ILP, several instructions are found to be executed at the same time. By providing multiple functional units on which to execute instructions, performance is expected to improve. However, branch instructions, which introduce control dependencies, and memory latency limit the effectiveness of ILP. In the IA-64, Intel's upcoming 64-bit architecture [6], two key architectural features – predication and control speculation – help compilers to extract instruction-level parallelism effectively. The IA-64 implements the Explicitly Parallel Instruction Computing (EPIC) design philosophy [10], which is based on:
-*Explicit parallelism.* An EPIC-based architecture makes ILP explicit in the machine code. It is more effective for the compiler to find the potential ILP than to discover ILP via hardware in the sequential machine code.

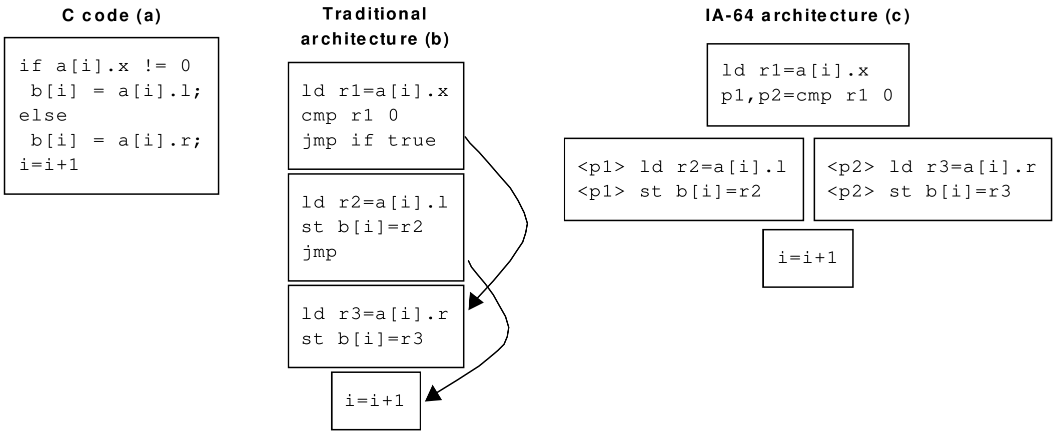

Fig. 4. A classic if-then-else statement in C code (a) becomes four basic blocks in a traditional processor architecture (b). In the IA-64 architecture the "if-then" block and the "else" block execute in parallel (c).

-*ILP features.* The IA-64 provides two features, predication and control speculation, that help the compiler express ILP.
-*Resources for parallel execution.* The IA-64 architecture provides more registers than conventional processors: 128 integer, 128 float, and 64 predicate registers.
The IA-64 uses a *full predication* model, in which compilers can append a predicate to each instruction. Predicates are simple tags that permit a program to execute instructions conditionally, depending on the predicate's value, which in turn depends on the outcome of a conditional statement. An instruction with a predicate value of true executes normally. If the predicate is false, the associated instruction, although issued, does not write its results to register or memory. Predication is an effective technique to remove branches. Figure 4(a) shows the C code for a classic if-then-else statement. In a traditional architecture, the processor loads the data from memory, compares the value of `a[i].x` with zero, and uses the result from the comparison in a conditional branch. Because of the conditional branch, a traditional compiler structures this code into four basic blocks, as shown in Fig. 4(b). The processors must execute the instructions of all four blocks serially. Predication can be used to remove the branch. In the IA-64 architecture, comparison instructions generate two predicates, as shown in Fig. 4(c): a true predicate, which is set if the result of the comparison is true, and a false predicate, which is set if the result of the comparison is false. The resulting code has no branch, so both paths of the conditional statement can execute in parallel. Since the two predicates are mutually exclusive, the two store instructions can execute in parallel, even if they store their results at the same address. The predicate guarantees that only one of the two stores write its result to memory. In this fashion, predication exposes additional ILP by removing branches.

3.3 Memories

Providing a logically shared but physically distributed address space is an interesting and challenging problem. Just as memory hierarchy in conventional architectures allows the data that is bound to an address to be migrated toward the processor that requires it, load instructions in distributed shared memory architectures allow data to migrate toward the local memory of the processor that accesses it. However, migrating and/or replicating data across a distributed shared memory multiprocessor presents a set of challenges because of coherence and remote memory latency issues.

A memory consistency model for a shared address space specifies constraints on the order in which memory operations must appear to be performed (i.e., become visible to the processors) with respect to one another. The most basic consistency model, based on the intuition of a single-processor system, is sequential consistency [14]. Sequential consistency, although an intuitive and straightforward model for the programmer, is difficult to implement inexpensively. Symmetric multiprocessors usually implement a general technique for cache coherence, called snooping, that exploits memory access serialization by the shared bus. An alternative way to implement a coherence protocol is by using a directory, where a central "home" unit keeps track of the memory locations that are shared by multiple caches and of those memory locations that are held exclusively. All memory accesses are handled by the home that, based on its directory, knows which caches need to be updated or invalidated on any given memory access. This approach is sensitive to the interconnection network speed, as all memory operations are handled centrally.

The cache-only memory architecture of the KSR1: a case study
The KSR1 system, released in 1992 by Kendall Square Research, was a demonstration that scalable shared memory multiprocessors of a thousand processors are feasible. The general KSR architecture is a multiprocessor system composed of a hierarchy of up to three layers of rings, where the processing elements occupy the lowest level, ring:0, and the routing processors occupy the higher levels. A novel feature of the KSR1 was its distributed memory scheme. Each processing element contains a data and an instruction cache, referred to as *subcaches*, and 32MB of memory referred to as *local cache*. All of the local caches, together with the interconnecting rings, make up the ALLCACHE^{TM} distributed memory memory system. A single System Virtual Address (SVA) space, spanning across all of the local caches, is supported by fully associative hardware translation tables. When a processor references an SVA, a *search engine*, which is the collection of processing element interfaces and the routing processor on each ring:0 along with the ring interface, locates the SVA and moves its contents to the local cache of the referencing processor. Variables are not bound to any particular memory, but rather migrate dynamically to

the processors requesting the data. The entire memory is shared and the memory is viewed as a hierarchy of caches, where coherence is implemented by a directory based protocol. Thus the KSR1 can be classified as a Cache–Only Memory Architecture (COMA) [11]. To take advantage of the architecture, programmers are provided with a *poststore* option. When a variable is updated, using poststore will ensure that all copies of the variable in all caches are updated immediately. This will shorten the access time for any future remote read access. A tradeoff exists since using poststore will help the "reader" threads at the expense of the "writer" threads, since the various level in the memory hierarchy have significantly different latencies, ranging from 10ns for the subcache to 850ns for ring:0 and 3μs for ring:1. Both an automatic and a programmer controlled prefetching function together with block data transfers are also available to help hide such long latencies.

In [26] an experimental and modeling study of a single ring:0 KSR1 system to determine under which conditions the use of poststore is an advantage is presented. Controlled experiments, using a simple readers-and-writers workload, are run to observe performance with and without poststore. An analytic model of the system is constructed and validated that can be used for predicting the general behavior of the KSR1 system and the specific behavior of poststore. Sensitivity analysis is conducted w.r.t. the request granularity, the amount of delay between requests, the read access pattern, the number of writers, the amount of data set sharing among readers, and the number of reader/writer threads running concurrently. Small, medium, and large data set sizes are used, in order to test the effect of processing for longer periods of time between consecutive accesses. The performance metric used is the average read time, i.e., access time, per subcache block by an average reader thread. The analytic model illustrates the processing which occurs in the subcaches, local caches, and the ring under the selected workloads. A Generalized Stochastic Petri Net (GSPN) [17,2] was selected to model the system. We report in Fig. 5 the measured (a) and predicted (b) access time per subcache block, for the base case experiment with a single writer thread and a variable number of reader threads for various access granularities, with and without poststore. The modeling and experimental results are in good agreement.

The experimental results indicate where poststore is most effective. They show that as the number of reader threads increases, poststore is more advantageous. However, if the number of reader threads is small, poststore should not be used since the benefit to the readers does not offset the extra overhead incurred by the writers. Thus, the number of reader threads influences the decision of whether or not to use poststore, and this number is a parameter of the programmer's application code. However, as contention increases at the processing elements, poststore is not beneficial and should not be used. The system load is controlled by the operating system scheduler. Therefore, neither allowing a programmer to use poststore without knowledge of the system load,

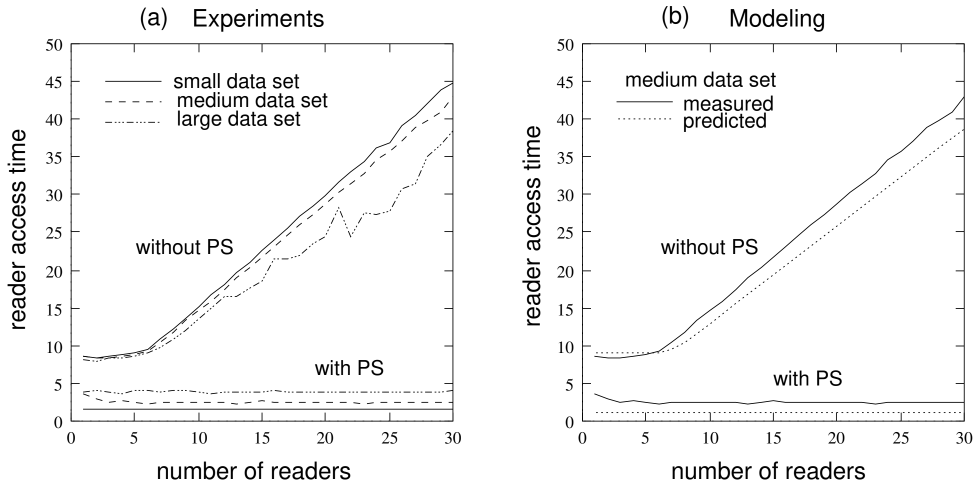

Fig. 5. Experimental (a) and modeling (b) results for the base case of the poststore analysis on the KSR1.

nor allowing the operating system to determine the use of poststore without knowledge of the application code, is advisable.

3.4 Interconnection Network

A network is characterized by its topology, routing algorithm and switching strategy. The *topology* is the physical interconnection structure of the network graph such as d-dimensional grid or torus, and logarithmic networks. In logarithmic networks the average distance between two nodes increases logarithmically with the d-root of the number N of nodes, in contrast with grids and tori where the average distance increases linearly with the d-root of N. Examples of logarithmic networks are trees, butterflies, and Omega networks. A distinction is often made between direct and indirect networks. Direct networks have a processors connected to each switch, whereas indirect networks have processors connected only to a specific subset of the switches, which form the edges of the network. Circuit switching and packets switching are the two most popular switching strategies. Packet switching typically allows better utilization of network resources because links and buffers are only occupied while a packet is traversing them.

Desirable properties of communication networks include low latency, high bandwidth, and scalability. Table 1 shows the scalability properties of some network topologies. N is the number of processors in the machine, *links* is the total number of connections between switches, *latency* is the worst case number of switches which must be traversed, and *bisectional bandwidth* is the worst case bandwidth between the two halves of the machine.

The wormhole routed mesh of the Paragon: a case study

Type	Switches	Links	Latency	Bisectional bandwidth
d-dimensional grid	N	dN	$d\sqrt[d]{N}$	$\sqrt[d]{N}$
Arity d Omega net	$N \log N$	$(dN \log N)/2$	$\log N$	N
Fully connected	N^2	N^2	1	N

Table 1
Properties of some network topologies.

The Intel Paragon is a mesh connected wormhole routed multiprocessor that found a considerable commercial success. Wormhole routing is a technique that guarantees high-performance interconnection network and allows for multiprocessor scalability [4]. One of its distinctive features is that communication latency is nearly independent of the distance between the communicating processes. Messages are divided into small pieces of few bytes each called flits (flow control digits). The header flit (or flits) reserves the channels as it moves from source to destination. The remaining flits follow in a pipelined fashion, thus hiding latency due to node distance. The wormhole routing on the Paragon is deterministic [4], meaning that the route established by the header flit is fixed and independent of the network load. Since message routing does not adapt to changes in the network load [1], contention can degrade communication latency in networks with deterministic routing.

In [29] the impact of interconnection network traffic on the scalability of Single Program Multiple Data (SPMD) applications is investigated using a queueing network model and results are then validated experimentally on a 512 node Paragon. Factors that affect performance and variance in workload scalability include the number and geometry of processors assigned to each program (i.e., the partition), the mapping of the communicating processes on the assigned partition, the available bandwidth, the system communication overhead, and the granularity of computation relative to communication. A communication intensive SPMD application, namely matrix transposition, is used to investigate the impact of such factors on program execution time. To model application speedup, a non-product form queueing network model is constructed [15]. The model is solved by simulation. Patterns of channel contention are generated by different process mapping (or data placement) strategies. Various matrix sizes are considered to vary the amount of network traffic and the time between consecutive transmissions. The optimal mapping, regardless of the physical partition geometry, allocates matrix subblocks that must be exchanged onto adjacent nodes so that no link contention occurs. The worst mapping maximizes channel contention. As expected, the best speedup is obtained with the optimal mapping. As the number of assigned processors increases, the difference between the worst and optimal placements becomes more pronounced. However, the gap in the speedup curves of the larger data set case is not as wide as in the smaller data set case. Although the link holding time increases with the matrix size, the gap among the three mapping

schemes decreases because of the higher increase in the ratio of computation to communication time of the workload. The computation part dominates and the workload execution time becomes less sensitive to network contention.
The effect of interleaving computation and communication was also investigated. Transposition is performed in a step-wise fashion. Smaller messages, combined with smaller amounts of computation between two message exchanges, are the key performance factors for this experiment. Interleaving computation and communication allows for the partial overlap of the application computation and communication phases. In general, it is noticed that the mapping schemes do not affect much the variance in workload scalability. As the number of exchanged messages increases, the gap in the speedups of the three mapping schemes decreases. Experimental validation is performed on a 512 node Paragon, organized as a 16×32 mesh. Figure 6 compares predicted versus actual speedups of the base case algorithm for two data sizes. As Fig. 6

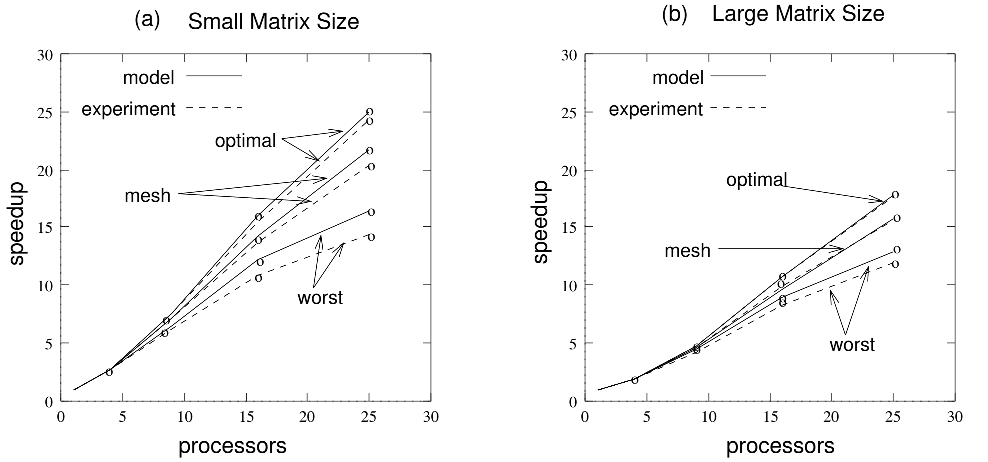

Fig. 6. Model predicted and experimentally measured speedup curves for three mapping schemes for a matrix of size 600×600 and a matrix of size 1200×1200.

indicates, the model predicts the execution time of the application with the optimal and the mesh data placements more accurately than with the worst data placement. The same observation can be made with respect to the matrix size. The model prediction is tighter with larger matrices as a result of the high computation to communication ratio.

3.5 Input/Output Subsystem

Complementary to high-speed processors, high-speed memories, and communication links, is high performance input/output (I/O). For many parallel applications that manipulate vast amounts of data, the broadening disparity in the performance of I/O devices and the performance of processors and communication links is the primary obstacle to achieving high performance [20].

As the nature of the problems to be solved requires larger and larger memories, and with I/O resources belonging to the slowest level of memory hierarchy, and as the speed of disk controllers remains considerably slow in comparison with other hardware components, parallelism is the only avenue to achieve the required I/O performance. Parallelism in I/O is achieved by using a pool of disks in parallel by striping data across all devices that are organized in a parallel disk array fashion (e.g., consider the various types of RAIDs), and by developing parallel file systems and their companion architecture and software support [31,30,13].
Performance evaluation studies are instrumental for the evaluation and assessment of different design alternatives in such systems as well as for the development of parallel I/O libraries and parallel I/O APIs. Prerequisite to the design of high performance parallel I/O systems is detailed workload characterization of I/O intensive applications. Such characterization can provide a better understanding of the interactions of computation and I/O requirements and offers valuable insights to both application programmers and system designers. For the application programmer, detailed workload characterization can become a source of suggestions for code restructuring by identifying system bottlenecks and consequently hinting ways to remove them. For the system designer, the characterization is instrumental for the design and tuning of file system policies [16] that respond well to diverse workload requirements and, in a broader sense, for the design and evaluation of general resource management policies (e.g., processor scheduling) in parallel systems [24].

Parallel I/O characterization: a case study
Here, we present a summary of a workload characterization study that aims to offer a better understanding of the interactions between application I/O request patterns and parallel I/O systems. To identify "common" behavior of scientific applications with respect to I/O requirements, we considered scientific codes from the Scalable I/O Initiative (SIO) suite [20]. We present the behavior of one selected code that effectively captures the I/O behavior and requirements that have been identified as characteristic of scientific applications in recent characterization studies [28].

The selected application is NWChem [12], a FORTRAN implementation of the Hartree-Fock self consistent field (SCF) iterative method that calculates the electron density around a molecule by considering each electron in the molecule in the collective field of the others. The measurements were conducted on the 512-node Intel Paragon XP/S with a 2 GB RAID-3 disk array composed of 12 disks, at the Caltech Center for Advanced Computing Research. Performance measures are collected using Pablo, a performance analysis environment that provides trace data for all I/O and CPU requests of the parallel applications [21].

NWChem is very I/O intensive, with I/O operations consuming almost 78%

of the overall execution time of the application. Pablo reported 499,912 I/O operations, dominated mainly by reads. During execution, all processors repeatedly read integrals from secondary storage to construct the Fock matrix and then solve the SCF equations. Because this integral database is computed and written to storage once, but then reread many times, the NWChem code is heavily read limited. Overall, the ratios of operation costs are analogous to operation counts, i.e., 93.05% of all I/O operations are read operations that consume 72% of the execution time, while only 6.48% are write operations that consume 5.39% of the execution time.

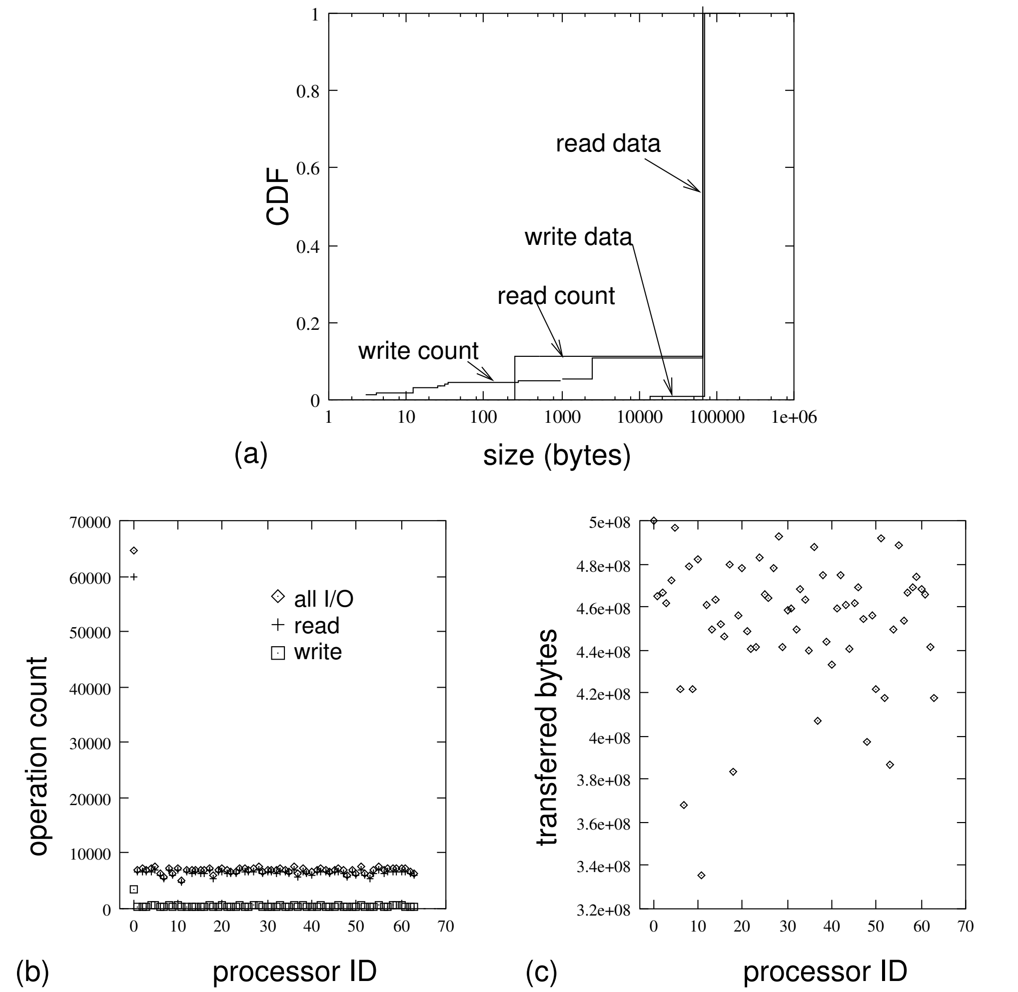

Fig. 7. CDF of read/write request sizes (a), number of I/O operations per processor (b), and data transfers per processor (c).

From a workload characterization perspective, the distribution and sizes of I/O operations are also very important and can suggest file system optimizations. Figure 7(a) shows the cumulative distribution function of the percentage of reads and writes as well as the volume of transferred data. The code does not exhibit significant variability in the read and write sizes. Actually, almost 90% of reads and writes are equal to 64 KB with the majority of data transferred

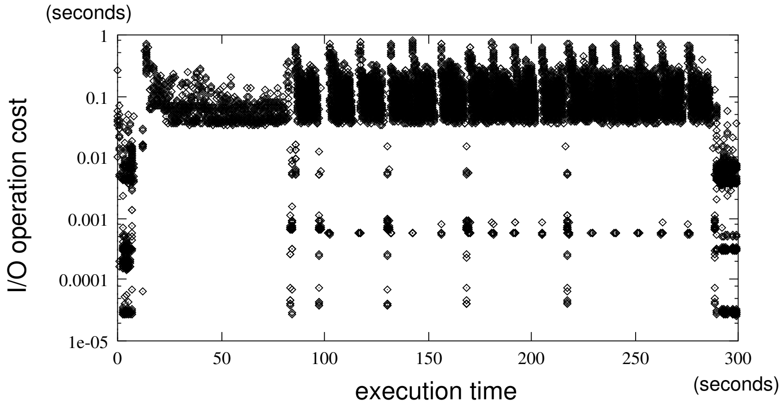

Fig. 8. Operation durations for NWChem as a function of the execution time.

through these operations. Figures 7(b) and (c) also illustrate the variations of I/O across processors. In this respect also, NWChem exhibits a typical behavior seen in many parallel applications. Logical node (processor) 0 executes more operations than the remaining of the processors. This is a direct effect of the developers' choice to use one node as coordinator (i.e., to collect data from others and then write them on the disks or equivalently to read data from the file system and then broadcast them to the remaining of the nodes).

Finally, we turn to the spacing of I/O requests as a function of the application execution time (see Fig. 8). The I/O intensive nature of the application is illustrated by the figure as well as the cyclic and bursty behavior of I/O requests. The first phase (up to the 80th second of execution) is dominated by small writes, all administered through logical node 0. During the second, read-intensive phase of the code, all nodes participate in reading the quadrature data from the disk, performing the SCF calculations, and writing the data to disk. The repetitive and bursty nature of I/O activity depicted can be effectively captured by Eq. 5 so that code scalability is effectively predicted as a function of both processor and I/O resources. In general, apart from predicting application scalability, observations about the application I/O characteristics can suggest application optimizations by suggesting code restructuring (e.g., by matching the application requests to the file system stripe size, by explicitly aggregating I/O requests, by overlapping computation and I/O via the use of asynchronous I/O requests), file system optimizations via implicit overlapping of computation and I/O transfers, I/O request aggregation, collective I/O, and efficient file pointer consistency protocols.

4 Operating System Components

In this section we briefly address the performance issues related to the software components of parallel systems at system level. Performance analysis is critical to the development of operating systems, for tuning, optimization, comparison and evaluation of different algorithms. In addition to the traditional services offered in single processor machines, operating systems of multiprocessors must coordinate the allocation and use by concurrent threads of multiple resources such as the processing elements, the parallel file system, memories and caches, and the interconnection network. We have already considered some of these issues in the previous section as they depend upon hardware components, e.g., memory and cache management or I/O and disks. Several services, such as communication libraries and thread and data allocation primitives, are provided by highly portable, robust and transparent runtime environments that allow to support different programming paradigms [19]. Such environments hide the architecture details from the end-users and contribute to the portability and robustness of parallel codes across a variety of hardware substrates.

Performance evaluation studies investigate the impact of resource management strategies at various levels on the entire system as a function of the workload. An important function provided by the operating system is multiprogramming that allows to share resources among several users, also known as processor partitioning or scheduling. The problem of scheduling in parallel systems is the problem of deciding *where* and *when* an application should execute, that is on which processors the application processes (or threads) will run (i.e., the processor allocation or job scheduling problem) and in what order (i.e., the process dispatching problem). Processors are assigned to competing applications in order to optimize two conflicting performance metrics: minimize the individual job response time, thus operating in the interest of the user, and maximize the system throughput or utilization, thus operating in the interests of the system as it allows to amortize the cost of parallel supercomputers. Coordinated allocation of dual resources, such as processors and memories or processors and disks, is the necessary extension of existing results for the processor-only case [18,24].

Processor scheduling: a case study
A large body of results have been generated by research in application scheduling for parallel systems (see [8] for a detailed survey). Different assumptions about the type of parallel workload, the goals of scheduling, and the type of parallel system have been investigated. Analytical, simulation, and experimental studies have been conducted. An example of a three-pronged approach that combines all the three methodologies for the evaluation of non-preemptive adaptive processor saving scheduling policies is given in [25]. Non-preemptive adaptive policies allow for processor redistribution only at application scheduling time, when the number of allocated processors is computed, so that each program runs exclusively on its assigned set of processors until it completes. Processor saving policies may keep processors unassigned in the presence of unfinished work, for unexpected future events, such as sudden bursty arrivals, unusually long execution time, irregular arrival behavior. Performance trade-

offs are investigated using Markovian and simulation models and are experimentally validated on a multiprocessor system using an actual workload. It is shown that non-preemptive adaptive processor saving policies may counterintuitively yield better performance in terms of average response time than their greedy, or work-conserving, counterparts, i.e., policies that assign all available processors as soon as possible.

A hierarchy of processor saving policies is constructed based on the amount of information regarding the past system behavior included in the policy. The work-conserving version of the policy is the baseline for the entire hierarchy. System behavior is modeled with a continuous time Markov chain, which can be solved analytically only when the number of possible partitions is restricted. If no restriction applies, a simulation model is used. Performance trade-offs with respect to the baseline work conserving counterpart of the class of policies considered are investigated under a range of scalability characteristics of the workload given by relative speedup varying from 0% to 100% and of distributions of the arrival and service processes when possible. Therefore, the ratio of the average response time under a given processor saving policy to the average response time under the baseline work-conserving policy is considered. The impact of different workloads is investigated by changing the speedup concavity and the job arrival process. Experimentation is used to validate the performance results under single class workloads and investigate the impact of multiclass workloads on the performance of a 512 node Intel Paragon. Experiments with 32, 64, and 128 processors were conducted and results are presented for the 64 processor case. The application used as test workload is an LU decomposition kernel (see Fig. 1) executed on matrices of different size, yielding corresponding relative speedups as those used in the simulations. Two multiclass workloads are obtained with different workload mixes, i.e., a 2-class and a 4-class workload. Figure 9 summarizes the analytical, simulation, and experimental results for the base case under exponential interarrival and service times for various workloads. The horizontal performance line at 1 represents the performance of the baseline policy. Performance above the horizontal line at 1 indicates a loss with respect to the greedy counterpart, while performance below 1 indicates a gain from using a processor saving policy.

As expected, on poorly parallel workloads and on highly variable workloads (the two multiclass cases) the processor saving policies outperform the greedy counterpart. Relative performance rankings from poorly parallel to highly parallel workloads are preserved in all the three cases. In general, both losses and gains are less pronounced in the analytical case because of the restriction on the number of possible partitions. The wider variety of partition sizes of the simulated and experimental system considered here accounts for the observed differences. The intersection point of each curve with the reference horizontal line moves leftward as the system size increases and the workload speedup decreases. With larger systems where processor scheduling policies allow for a

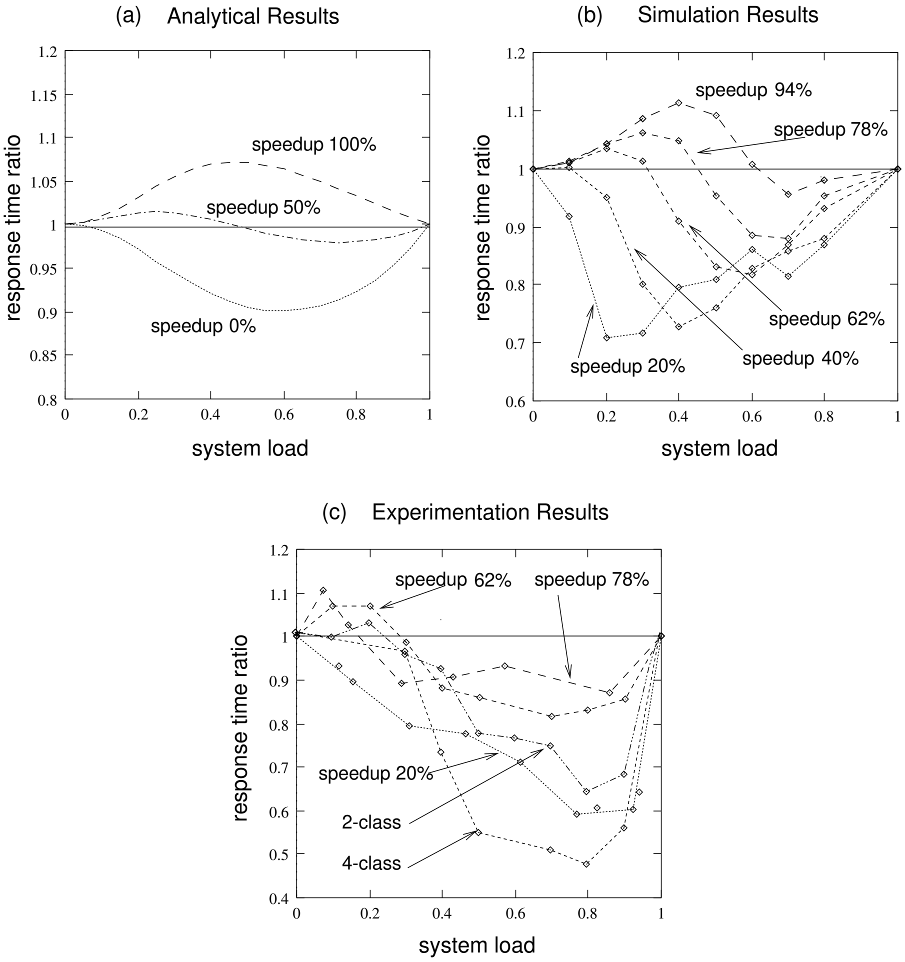

Fig. 9. Analytical (a), simulation (b), and experimentation (c) results for a processor saving policy with respect to the baseline work-conserving counterpart (horizontal line) for various single class and multiclass workloads under exponential interarrival and execution times as a function of system load.

wider variety of partition sizes and higher multiprogramming levels, the disadvantages of processor saving policies (i.e., potential waste of idle processors) is minimized.

5 Conclusions

Performance evaluation as a discipline has repeatedly proved to be critical for design and successful use of parallel systems. At the early stage of design, performance models can be used to project the system scalability and evaluate design alternatives. At the production stage, performance evaluation methodologies can be used to detect bottlenecks and subsequently suggest ways to alleviate them. Queueing and Petri Nets models, simulation, experimental measurements, and hybrid modeling have been successfully used for the evaluation of system components. Via the form of self-contained case studies we illustrated the use of the above techniques for the modeling and evaluation of memory, interconnection network, input/output, and schedulers for processor allocation.

References

[1] V.S. Adve and M.K. Vernon. Performance analysis of mesh interconnection neworks with deterministic routing. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):225–246, March 1994.

[2] M. Ajmone-Marsan, G. Balbo, and G. Conte. *Performance models of multiprocessor systems*. The MIT Press, Cambridge, Massachusetts, 1986.

[3] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of the AFIPS Conference*, pages 483–485. AFIPS, 1967.

[4] W.J. Dally and C.L. Seitz. The torus routing chip. *Journal of Distributed Computing*, 1(3):187–196, March 1986.

[5] J.J. Dongarra, T. Hey, and E. Strohmaier. Selected results from the Parkbench benchmark. In *Proceedings of the Second International Euro-Par '96 conference*, volume 1124 of *Lecture Notes in Computer Science*, page 251, Lyon, Fr, August 1996. Springer Verlag.

[6] C. Dulong. The IA-64 architecture at work. *IEEE Computer*, 31(7):24–32, July 1998.

[7] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[8] D.G. Feitelson. A survey of scheduling in multiprogrammed parallel systems. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, August 1997.

[9] D. Ghosal, G. Serazzi, and S.T. Tripathi. Processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.

[10] L. Gwennap. Intel, HP make EPIC disclosure. Microprocessor report, October 1997.

[11] E. Hagersten, A. Landin, and S. Haridi. DDM: a cache-only memory architecture. *IEEE Computer*, 25(9):45–54, September 1992.

[12] High Performance Computational Chemistry Group, Pacific Northwest National Laboratory, Richland, Washington, 99352, USA. *NWChem, A computational chemistry package for parallel computers, Version 1.1*, 1995.

[13] Intel Corporation, Beaverton, OR. *Paragon system user's guide*, 1995.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[15] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative system performance: Computer systems analysis using queueing network models*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1984.

[16] T. Madhyastha and D.A. Reed. Intelligent, adaptive file system policy selection. In *Proceedings of Frontiers '96*, 1996.

[17] M.K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Transactions on Computers*, C-31(9):913–917, September 1982.

[18] E.W. Parsons and K.C. Sevcik. Coordinated allocation of memory and processors in multiprocessors. In *Proceedings of the 1996 ACM SIGMETRICS Conference*, pages 57–67, Philadelphia, PA, June 1996.

[19] S. Pelagatti. *Structured development of parallel programs*. Taylor & Francis, 1997.

[20] J.T. Poole. Scalable I/O initiative. California Institute of Technology, Available at `http://www.ccsf.caltech.edu/SIO/`, 1996.

[21] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B.W. Schwartz, and L.F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.

[22] J. Rolia and K.C. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, August 1995.

[23] E. Rosti and G. Serazzi. Workload characterization for performance engineering of parallel applications. In *Proc. of the 2nd Euromicro Workshop on Parallel and Distributed Processing*, pages 457–462. IEEE Computer Society Press, January 1994.

[24] E. Rosti, G. Serazzi, E. Smirni, and M.S. Squillante. The impact of I/O on program behavior and parallel scheduling. In *Proceedings of the 1998 ACM SIGMETRICS Conference and Performance '98*, pages 57–65, Madison, WI, June 1998.

[25] E. Rosti, E. Smirni, L.W. Dowdy, G. Serazzi, and K.C. Sevcik. Processor saving scheduling policies for multiprocessor systems. *IEEE Transactions on Computers*, 47(2):178–189, February 1998.

[26] E. Rosti, E. Smirni, T.D. Wagner, A. Apon, and L.W. Dowdy. The KSR1: experimentation and modeling of poststore. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, pages 74–85, Santa Clara, CA, USA, May 1993.

[27] J.P. Singh, W.D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, January.

[28] E. Smirni and D.A. Reed. Lessons from charaterizing the input/output behavior of parallel scientific applications. *Performance Evaluation*, 33(1):27–44, June 1998.

[29] E. Smirni and E. Rosti. Modeling speedup of SPMD applications on the Intel Paragon: a case study. In *Proceedings of the High-Performance Computing and Networking 95 Int'l Conference*, volume 919 of *Lecture Notes in Computer Science*, pages 94–101, Milan, Italy, May 1995. Springer Verlag.

[30] The MPI-IO Committee. MPI-IO: a parallel file I/O interface for MPI, April 1996. Version 0.5.

[31] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.