RC23780 (W0511-044) November 7, 2005
Mathematics

IBM Research Report

Modeling and Analysis of Dynamic Coscheduling in Parallel and Distributed Environments

Mark S. Squillante[1], Yanyong Zhang[2], Anand Sivasubramaniam[3], Natarajan Gautam[3], Hubertus Franke[1], Jose Moreira[1]

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

[2]Rutgers University
Piscataway, NJ 08854

[3]Pennsylvania State University
University Park, PA 16802

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Modeling and Analysis of Dynamic Coscheduling in Parallel and Distributed Environments

Mark S. Squillante,[*] Yanyong Zhang,[†] Anand Sivasubramaniam,[‡]
Natarajan Gautam,[‡] Hubertus Franke,[*] Jose Moreira[*]

Abstract

Scheduling in large parallel systems continues to be an important and challenging research problem. Several key factors have resulted in the emergence of a new class of scheduling strategies: Dynamic coscheduling. Given their large design and performance spaces, it is difficult to fully explore the benefits and limitations of the proposed dynamic coscheduling approaches for large systems solely with the use of simulation or experimentation. We therefore formulate a general mathematical model of this class of scheduling strategies, and derive an exact and approximate matrix-analytic analysis for relatively small and large instances of the model, respectively. The results of numerical experiments with our approximation are in excellent agreement with detailed simulation results. Our analysis is then used to explore fundamental design and performance tradeoffs associated with the class of dynamic coscheduling policies across a broad spectrum of parallel computing environments.

Keywords: stochastic models, parallel systems, parallel scheduling, queueing theory, performance optimization.

1 Introduction

Parallel system scheduling is challenging because of the numerous factors involved in implementing such a scheduler, including the workload, native operating system (OS), node hardware, network interface, network, and communication software. Over the past decade, parallel computing is centered around off-the-shelf clusters for their cost-effectiveness, which further complicate scheduling issues because these platforms usually employ user-level messaging (to reduce latencies and improve bandwidth), wherein the OS is unaware of the task waiting for a message. Multiprocessors have traditionally addressed this problem using a technique called coscheduling [11] or gang scheduling [5], wherein tasks of a job are scheduled on their respective nodes during the same time quantum. However, gang scheduling is not a very attractive nor scalable option for off-the-shelf clusters, since it requires periodic synchronization across the nodes to coordinate the effort. At the same time, on the application side, an ever increasing number of applications, which are rapidly shifting from CPU intensive to I/O intensive or communication intensive, co-exist on the same cluster platform, making gang scheduling unsuitable.

[*]IBM Watson Research Center, Yorktown Heights, NY 10598, USA. {mss,frankeh,jmoreira}@us.ibm.com
[†]Rutgers University, Piscataway, NJ 08854, USA. yyzhang@ece.rutgers.edu
[‡]Pennsylvania State University, University Park, PA 16802, USA. anand@cse.psu.edu; ngautam@psu.edu

As a result, there have been recent efforts [4, 14, 8] to design a new class of scheduling mechanisms – broadly referred to as *dynamic coscheduling* – which approximate coscheduled execution without explicitly synchronizing the nodes. These techniques use local events (such as message arrival) to estimate what is happening at remote nodes, and adjust their local schedules accordingly. They are able to simply adjust task priorities (e.g., this can be implemented using a loadable driver [14, 8]) to achieve this goal, leaving it to the native OS to do the actual scheduling. These scheduling mechanisms have been shown to be easier to implement, incur less overheads, and result in more effective schedules than exact coscheduling in a specific computing environment [19].

The design and performance spaces of dynamic coscheduling mechanisms are quite complex, and a myriad of heuristics are possible to schedule the tasks. Without a unifying performance evaulation framework, it is very difficult to perform an extensive analysis of the benefits and limitations of these different mechanisms and heuristics, especially given so many diverse parameters for the parallel system and workload. As a result, it is difficult to say which approaches perform best, and under what conditions, and thus many important design and performance questions remain open. For instance, it is not clear how the performance of these mechanisms compare as the size and traffic intensity of the system increases. What is the effect of varying the relative fraction of computation (requiring only CPU resources), communication and I/O components in the application, which represents different parallel workloads? Each dynamic coscheduling mechanism has tunable parameters. How can we select values for these parameters to get the best performance? As the underlying hardware or OS changes how do the relative benefits of the different alternatives compare? While one could use experimentation [8] and simulation [19] to study relatively small systems, the suitability of these mechanisms for large systems has not been explored. This is one of the motivating factors for our study as several large clusters, ranging from hundreds to thousands of processors, have been deployed both in production and research environments to handle the high-performance computing needs of both scientific and commercial applications.

Analytical models can be used to address each of the key problems and issues described above, thus complementing previous (and future) experimental and simulation based studies. While some analytical models have been developed for exploring the assignment of tasks to processors (e.g., [12, 15]) and for analyzing certain forms of gang scheduling (e.g., [16]), there has been no prior research to our knowledge that develops analytical models of the behavior of the various dynamic coscheduling strategies recently proposed. The reasons for this center around the inherent complexity of developing such mathematical models and analysis given the dynamic and complex interactions among the parallel processors.

This paper therefore fills a crucial void by developing and exploiting the first general analytical models and analysis (to our knowledge) that accurately capture the execution of dynamically coscheduled parallel systems. We formulate a general mathematical model of this class of scheduling strategies within a unified framework that allows us to investigate a wide range of parallel systems and that supports incor-

porating different scheduling heuristics, workload parameters, system parameters and overheads, policy parameters, and so on. We derive an exact matrix-analytic analysis for relatively small parallel systems, and an approximate matrix-analytic analysis for the large systems motivating our study. A key aspect of our approximate approach consists of probabilistically capturing the various correlations and dynamic behaviors of the parallel system and scheduling policy based on a general stochastic decomposition and fixed-point iteration. Our models and analysis include important aspects of real parallel environments such as the effects of increased contention for system resources, and provides detailed statistics on system and application states (in addition to performance measures) that can be used to explain the overall results, to isolate hotspots (for subsequent optimization), and to gain a better understanding of the benefits and limitations of different dynamic coscheduling approaches. These models and analysis significantly extend and generalize the corresponding results presented in our previous conference paper [18].

Many numerical experiments have been conducted as part of our study. We first consider the accuracy of our approximate approach and find the results to be in excellent agreement with those from detailed simulations of a relatively small parallel system, often being within 5% and always less than 10%. Moreover, the accuracy of our approach increases with the size of the system and our primary interests are in large parallel systems. Using our general models and analysis, we then evaluate in detail for the first time the benefits and limitations of previously proposed dynamic coscheduling mechanisms under a wide range of different workload and system configurations, significantly expanding the results in [18] by exploiting our more general models and analysis. The advantages of such general, flexible and configurable models are demonstrated by conducting studies with different computing environments to explore and understand the fundamental design and performance space issues and tradeoffs associated with the various dynamic coscheduling strategies. While the motivation for our models comes from the suitability of dynamic coscheduling mechanisms for cluster environments, our general models are broadly applicable and can be used to examine the behavior of these mechanisms on a diverse set of parallel and distributed systems subject to a diverse set of workloads.

The next section summarizes the dynamic coscheduling mechanisms. We present our parallel system environment and the corresponding stochastic models in §3. Our general mathematical analysis of these parallel dynamic coscheduling models is provided in §4, and a representative sample of the results from our numerical experiments is presented in §5. We summarize the contributions of this work in §6.

2 Dynamic Coscheduling Policies

In this section we briefly review the three previously proposed dynamic coscheduling mechanisms, namely Spin-Block (SB), Demand-based Coscheduling (DCS) and Periodic Boost (PB). We also describe the system, called Local Scheduling (LS), that does not attempt to do any coscheduling between the nodes, and leaves it to the native OS at each node to time share its processor.

Local Scheduling (LS). The native OS is left to schedule the tasks at each node, with no coordination among the nodes. Most off-the-shelf/commercial OS schedulers use some version of the multi-level feedback queue to implement time sharing. There are a certain number of priority levels, with jobs waiting at each level, and a time quantum associated with that level. In this paper, we associate equal time quanta for all priority levels which is the case in Windows NT. When a task initiates an I/O operation, it relinquishes the CPU, and is put in a blocked state. Upon I/O completion, it typically receives a priority boost and is scheduled again.

Spin Block (SB). Versions of this mechanism have been considered in the context of implicit coscheduling [4, 1] and DCS [14]. In this scheme, a task spins on a message receive for a fixed amount of time (*spin time*) before blocking itself. The rationale here is that if the message arrives in a reasonable amount of time (spin time), the sender task is also currently scheduled and the receiver should hold on to the CPU. Otherwise, it should block so that CPU cycles are not wasted. When the message does arrive, the task is woken up, and consequently gets a boost in priority to get scheduled again. There are costs associated with blocking and waking up (requires an interrupt on message arrival). Our model is flexible enough to allow either fixed or variable (adaptive) spin times and the model derivation allows for incorporating several heuristics that can be used to tune the spin times adaptively.

Demand-based Coscheduling (DCS). DCS [14] uses an incoming message to schedule the task for which it is intended. The underlying rationale is that the receipt of a message denotes the higher likelihood of the sender task being scheduled at the remote workstation at that time. Upon message arrival, if the intended task is not currently scheduled, an interrupt overhead is paid for re-scheduling that task.

Periodic Boost (PB). This mechanism has been proposed as an interrupt-less alternative to address the inefficiencies arising from scheduling skews between tasks. Instead of immediately interrupting the host CPU on message arrival as in DCS, the actions are slightly delayed. A kernel activity becomes active periodically to check (in round-robin order) if there is a task with a pending message, and if so it boosts the priority of that task. Even if there is no such task, but the currently scheduled one is busy waiting for a message, the activity boosts another task with useful work to do. Though interrupts are avoided, there is the fear of delaying the actions more than necessary. At the same time, making this too frequent would increase the overheads.

3 System Environment and Models

Our stochastic models of the parallel systems and workloads of interest in this study are based on the broad spectrum of parallel computing environments found in practice; see [16, 9] and the references cited

therein. Similarly, our stochastic models of the above dynamic coscheduling mechanisms are based on actual implementations described in the research literature [14, 8], and the costs used for the various scheduling actions have also been drawn from experimental results. This collection of stochastic models significantly extends and generalizes the corresponding results in our previous paper [18].

3.1 Parallel System

The parallel system consists of P identical processors, each capable of executing any of the parallel tasks comprising an application. Every processor can operate at a maximum multiprogramming level (MPL), which is usually governed by resource availability (e.g., memory/swap space) to provide reasonable overall performance for the executing jobs. Processors are interconnected by a network that has both a latency and a contention component. Upon arrival, jobs specify a certain number of processors that they require (which is equal to the number of tasks in that job). Each of the tasks belonging to the job is then assigned to a processor that is not already operating at its maximum MPL. Even if only one of the tasks cannot be assigned, the job waits in a system queue and the scheduler assigns jobs from this queue in a first-come first-serve (FCFS) order.

Our analysis will primarily focus on the case in which the MPL at every processor allocated to an application is the same. This is often the case in many large parallel systems that are spatially partitioned together with some type of packing scheme to fill holes in the time-sharing slots. It is important to note, however, that our approach is not limited to this case, and in §4 we briefly discuss how our approach can be used to also handle the case in which the set of processors executing an application have different MPLs. Let M denote the maximum MPL at the processor partitions in the system. Another primary focus in this paper is on *large* parallel computer systems which are playing an important role in many scientific and commercial applications.

3.2 Parallel Workload

Parallel jobs are assumed to arrive to the system from an exogenous source according to a Markovian Arrival Process (MAP) having descriptors $(\mathcal{S}_0^A, \mathcal{S}_1^A)$ of order m_A with mean rate λ. An arrival is placed in the FCFS system queue when all M time-sharing slots are filled in each of the processor partitions of the desired size. The time-sharing quantum lengths (QLs) and the context-switch (CS) overheads at each processor are respectively assumed to be independent and identically distributed (i.i.d.) following the PH-type distributions PH$(\underline{\chi}, \mathcal{S}^Q)$ and PH$(\underline{\xi}, \mathcal{S}^O)$ of orders m^Q and m^O having means τ^{-1} and δ^{-1}.

The applications comprising the system workload consist of parallel tasks, each of which alternates among several stages of execution in an iterative manner. As is common in the class of large parallel applications, we assume that the applications tend to be long-running with a relatively large number of iterations. The number of iterations for each application, N_A, is assumed to follow a (shifted) geometric

distribution with parameter p_{N_A}, i.e., $\mathbb{P}[N_A = 1 + n] = (1 - p_{N_A})^n p_{N_A}$, $n \in \mathbb{Z}_+$. We consider a general class of parallel applications in which each iteration consists of a computation stage, an I/O stage, and a communication stage (i.e., sending and receiving messages among tasks). The service times of these per-iteration computation, I/O and communication stages are respectively assumed to be i.i.d. according to the order m^B, m^I and m^C PH-type distributions PH($\underline{\beta}, \mathcal{S}^B$), PH($\underline{\eta}, \mathcal{S}^I$) and PH($\underline{\zeta}, \mathcal{S}^C$) with means μ^{-1}, ν^{-1} and γ^{-1}. As part of the communication stage, a task may also have to wait for the receipt of a message from a peer parallel task. We also consider an all-to-all communication strategy, where a task may have to wait for messages from all other tasks before proceeding. All of the above stochastic sequences are assumed to be mutually independent.

The chosen structure for a parallel job stems from our experiences with numerous parallel applications. Several scientific applications, such as those in the NAS benchmarks [2], and Splash suite [13], exhibit such behavior. For instance, computation of a parallel multidimensional FFT requires a processor to perform a one-dimensional FFT, following which the processors exchange data with each other to implement a transpose, and the sequence repeats iteratively. I/O may be needed to retrieve the data from the disk during the dimensional FFT operation since these are large data sets. Similarly a parallel SOR has each processor compute matrix elements by averaging nearby values, after which the processors exchange their latest values for the next iteration.

The stochastic processes assumed for the mathematical model are important in that they determine both the generality of our solution and the usefulness of our model in practice. We therefore exploit the general class of MAPs and PH-type distributions; refer to [10, 6] for more details including definitions, notation and known results. The use of MAPs and PH-type distributions for all model parameters is of theoretical importance in that we exploit their properties to derive solutions of various instances of our general stochastic scheduling models. It is also of practical importance in that, since the class of PH-type distributions is dense within the set of probability distributions on $[0, \infty)$, and since the class of MAPs provides a general framework for capturing the dependence structure and variability of a process, any stochastic process on this space for the parallel computing environments of interest can in principle be represented arbitrarily closely by a MAP or a PH-type distribution. Moreover, a considerable body of research has examined the fitting of PH-type distributions and MAPs to empirical data, and a number of algorithms have been developed for doing so; e.g., see [7] and the references cited therein. By appropriately setting the parameters of our models, a wide range of parallel application and system environments can be investigated.

4 Mathematical Analysis

In this section we present an overview of our general mathematical analysis of the foregoing parallel scheduling models within a unified framework that allows us to investigate all classes of parallel ap-

plication and system environments of interest using a single formulation. This analysis significantly extends and generalizes the corresponding results presented in [18]. However, due to space limitations, the mathematical details of this analysis are provided in a separate companion paper [17]. Herein we first summarize an approach that makes it possible to obtain an exact solution for relatively small parallel systems using our results, which also illustrates some of the difficulties and complexities of an exact analysis. We then summarize an approximate approach to address the fundamental problems involved in the mathematical modeling and analysis of large instances of this complex parallel system. Our approximate analysis is based on a general form of stochastic decomposition in which we derive distributional characterizations of the dynamic interactions and dependencies among the parallel processors under various dynamic coscheduling policies. We then exploit this distributional characterization to obtain a reduced state-space representation, for which we derive a matrix-analytic analysis based on a fixed-point iteration. Our analysis can be shown to be asymptotically exact in P for certain model instances, and numerous simulation-based experiments generally demonstrate that the accuracy of our approach increases with the value of P and that this approximation is very accurate even for relatively small parallel systems. We also obtain performance measures of interest in terms of our exact and approximate solutions, and then extend both solution approaches to incorporate the impact of resource contention.

4.1 Exact Matrix-Analytic Analysis

The parallel dynamic coscheduling models can be represented by a continuous-time stochastic process $\{X(t)\,;\, t \in \mathbb{R}_+\}$, defined on an infinite, multi-dimensional state space Ω^X where each state $x \in \Omega^X$ records: the total number of parallel jobs in the system; the state of the arrival process; the state of the overall service process for each parallel job at each processor (including scheduling priority ordering); and the state of the QL process (including CS overhead) at each processor. This formulation is exact and it exploits known closure properties of PH-type distributions [10, 6], most notably that the convolution of PH-type distributions is also of PH-type. We therefore can generally capture the various stages of execution (e.g., computation, I/O and communication) for the classes of parallel applications of interest via a single PH-type distribution that represents the appropriate combinations of the PH-type distributions for each of these stages of execution as well as other system behavior (e.g., the impact of a task waiting to receive a message). We refer to this combined process as the *overall service process*.

Let $\boldsymbol{\pi}^X$ be the stationary distribution for the stochastic process $\{X(t)\,;\, t \in \mathbb{R}_+\}$. Assuming this process to be irreducible and positive recurrent, the invariant probability vector is uniquely determined by solving $\boldsymbol{\pi}^X \mathbf{Q}^X = \mathbf{0}$ and $\boldsymbol{\pi}^X \mathbf{e} = 1$, where $\mathbf{Q}^X$ is the generator matrix for the process, and $\mathbf{e}$ denotes the column vector of appropriate dimension containing all ones. The elements of the generator matrix are constructed based on the above formulation and the dynamic transitions among the stages of behavior for the parallel system, which also depend upon the specific dynamic coscheduling strategy.

By exploiting the structure of the generator $\mathbf{Q}^X$ from our formulation, the invariant probability vector $\boldsymbol{\pi}^X$ of the process $\{X(t)\,;\, t \in \mathbb{R}_+\}$ can be obtained from standard matrix-analytic results [10, 6]. While this provides an exact solution, instances of the model even for very small parallel systems can cause the computational complexity to become prohibitive. We therefore establish results to calculate the stationary probability vector $\boldsymbol{\pi}^X$ (and $\boldsymbol{\pi}^Y$ below) that significantly reduce the time and space complexities over numerically computing the solution from standard results; refer to [17] for the mathematical details. This makes it possible for us to compute exact solutions for relatively small instances of the dynamic coscheduling models that are otherwise prohibitively expensive.

Still, a primary difficulty in the above (exact) approach for solving larger instances of the stochastic process $\{X(t)\,;\, t \in \mathbb{R}_+\}$, which records the state of each processor in the system, is the combinatorial explosion of the size and complexity of its state space Ω^X. A similar difficulty exists in analyzing the stochastic process that records the state of every processor in each of the partitions in isolation, particularly given the large parallel computing environments motivating our study. We next present an overview of our approximate solution approach that addresses these issues.

4.2 Stochastic Decomposition

To create a tractable formulation for large instances of our model, we first partition the system into sets of processors that are executing the same collection of parallel applications. We approximate each of these processor partitions by assuming that the distribution of the state of each processor in a given partition is stochastically independent and identical to the distribution of the states of the other processors in the partition. For each processor partition, the corresponding decomposed continuous-time stochastic process $\{Y(t)\,;\, t \in \mathbb{R}_+\}$, representing each individual processor in the partition, then can be solved in isolation by modifying its overall service process to reflect the distributional behavior of the other processors in the partition. Thus, the complex dependencies among the processors of a partition and their complex dynamic interactions are probabilistically captured, in part, through the overall service process. Performance measures for the entire partition can be obtained by analyzing the model of a single processor in this manner via a fixed-point iteration, and performance measures for the entire parallel system are obtained by combining the solutions for each of the partitions in the system.

This formulation of our approach assumes a specific parallel computing environment in which each of the processors allocated to an application has the same MPL, as noted in §3.1. However, our approach also can be used to handle large parallel systems in which the set of processors executing an application has different numbers of applications assigned to them. This is achieved by further partitioning the set of processors allocated to an application based on their MPL. Our approach is then used to solve in isolation the decomposed stochastic process representing each individual processor in each of these subpartitions by modifying its overall service process to reflect the distributional behavior of the other processors in

the subpartition, as well as the processors in the other subpartitions. The fixed-point iteration of our approach is extended in a relatively straightforward manner to handle these subpartitions.

4.2.1 Matrix-Analytic Analysis

Consider a particular processor from a specific processor partition represented by a continuous-time stochastic process $\{Y(t)\,;\, t \in \mathbb{R}_+\}$, defined on a state space Ω^Y where each state $y \in \Omega^Y$ records: the number of parallel jobs at the given processor; the state of the arrival process; the state of the overall service process for each parallel job at the processor (including scheduling priority ordering); and the state of the QL process (including CS overhead) at the processor. Let $\boldsymbol{\pi}^Y$ be the stationary distribution for the stochastic process $\{Y(t)\,;\, t \in \mathbb{R}_+\}$. Assuming this process to be irreducible and positive recurrent, the invariant probability vector is uniquely determined by solving $\boldsymbol{\pi}^Y \mathbf{Q}^Y = \mathbf{0}$ and $\boldsymbol{\pi}^Y \mathbf{e} = 1$, where $\mathbf{Q}^Y$ is the generator matrix for the process $\{Y(t)\,;\, t \in \mathbb{R}_+\}$. We again exploit the structure of the generator $\mathbf{Q}^Y$ from our formulation to efficiently obtain the invariant probability vector $\boldsymbol{\pi}^Y$ of the process $\{Y(t)\,;\, t \in \mathbb{R}_+\}$ using a combination of standard matrix-analytic results [10, 6] and our results established in this study [17]. This makes it possible for us to compute solutions for large instances of the dynamic coscheduling models that are otherwise intractable. Moreover, the algorithm (based on our results) used to compute these solutions is numerically stable across a wide spectrum of model parameters, and we encountered no numerical stability problems throughout the numerous experiments performed as part of our study, some of which are provided in §5.

The elements of the generator matrix $\mathbf{Q}^Y$ are constructed based on the above formulation and the dynamic transitions among the stages of behavior for the parallel system, which also depend upon the specific version of dynamic coscheduling of interest. Our approach includes capturing these dynamic behaviors in the overall service process. The first 3 stages of the overall service process for each application represent the computation, I/O and communication stages of execution. In the following overview of the technical details of our analysis of each processor under all coscheduling strategies, we assume that the communication stage of each task consists of sending a *single* message to one other task followed by receiving a *single* message from one other task. (More general cases of sending/receiving multiple messages are addressed within the context of our approach in §4.2.2.) We further assume that the communication stage is comprised of the total processor demands for the send operation and the portion of the processor demands for the receive operation up to the point of checking if the message from another task has arrived; the processor demands for the remainder of the receive operation are included as part of the computation stage of the next iteration. For this reason, we shall henceforth use "send stage" and "communication stage" interchangeably. Recall that a CS occurs on an I/O operation. At the end of the I/O stage for each iteration, the job completes and departs the system with probability (w.p.) p_{N_A}, and otherwise it proceeds to the send stage. Upon completion of this communication stage, the application

immediately enters the computation stage of the next iteration if the message to be received has already arrived, which occurs w.p. $\widehat{p}_A$. On the other hand, if the message to be received has not arrived, then the sender of the message can be in 1 of 4 (generic) states, namely computation, I/O, send and waiting for a message from yet another processor. The corresponding events that the sender of the message is active in the computation, I/O or send stages occur w.p. $\widehat{p}_B$, $\widehat{p}_I$ and $\widehat{p}_C$, respectively.

Let T_S denote the maximum number of iterations, at any given time, that a waiting task can be ahead of the task which will be sending the corresponding message, under the dynamic coscheduling policy being examined. To elucidate the exposition, suppose that $T_S = 1$ in the following illustrative examples which implies that at any time the 2 communicating tasks are not more than 1 iteration apart. (The general case for T_S is considered in §4.2.2.) We therefore have, under most of the coscheduling policies (except SB), stages 4, 5 and 6 of the overall service process for each (waiting) application represent the computation, I/O and communication stages of execution for the sender. The measures $\widehat{p}_A$, $\widehat{p}_B$, $\widehat{p}_I$, $\widehat{p}_C$ and the PH-type distributions for the waiting stages (as well as other measures involved in the mathematical details) are all calculated from our model solution and form the basis for a fixed-point iteration that is discussed in §4.2.2. More general communication strategies are handled in a similar manner. For example, in the case of an all-to-all communication strategy, we derive a formula for the distribution of time that the task waits for messages from all other tasks (based on distributions conditioned on the state of the system), and then we use a PH-type distribution to estimate (or bound) this waiting time distribution.

We now turn to the specific details of the dynamic system behavior and set of transitions for our model of each processor under each coscheduling strategy of interest, as illustrated in Fig. 1. To further clarify the presentation, the following examples assume an exponential distribution for each stage of execution and use a random communication pattern between tasks, i.e., during each iteration, a task only sends and receives 1 message, and its receiver and sender are randomly chosen. By replacing each exponential state with a set of states representing the desired PH-type distribution, we obtain the more general models that are solved by our matrix-analytic analysis. We also assume the message transmission overhead is 0.

Local Scheduling (LS). The execution of an application task on a processor is managed by the underlying OS. From the application behavior viewpoint, a task goes through 6 execution stages: computation (w.p. $\widehat{p}_B$), I/O (w.p. $\widehat{p}_I$), sending messages (w.p. $\widehat{p}_C$), waiting for messages from tasks doing computation, waiting for messages from tasks doing I/O, and waiting for messages from tasks in the process of sending them. From the underlying OS viewpoint, a task can be in 1 of 3 states: running, ready, or blocked. The combination of these 2 dimensions determines the state of a task at any instant; see Fig. 1(a). Let us next look at the transition rules between these states, first focusing on transitions common to all policies and then turning to LS specific transitions.

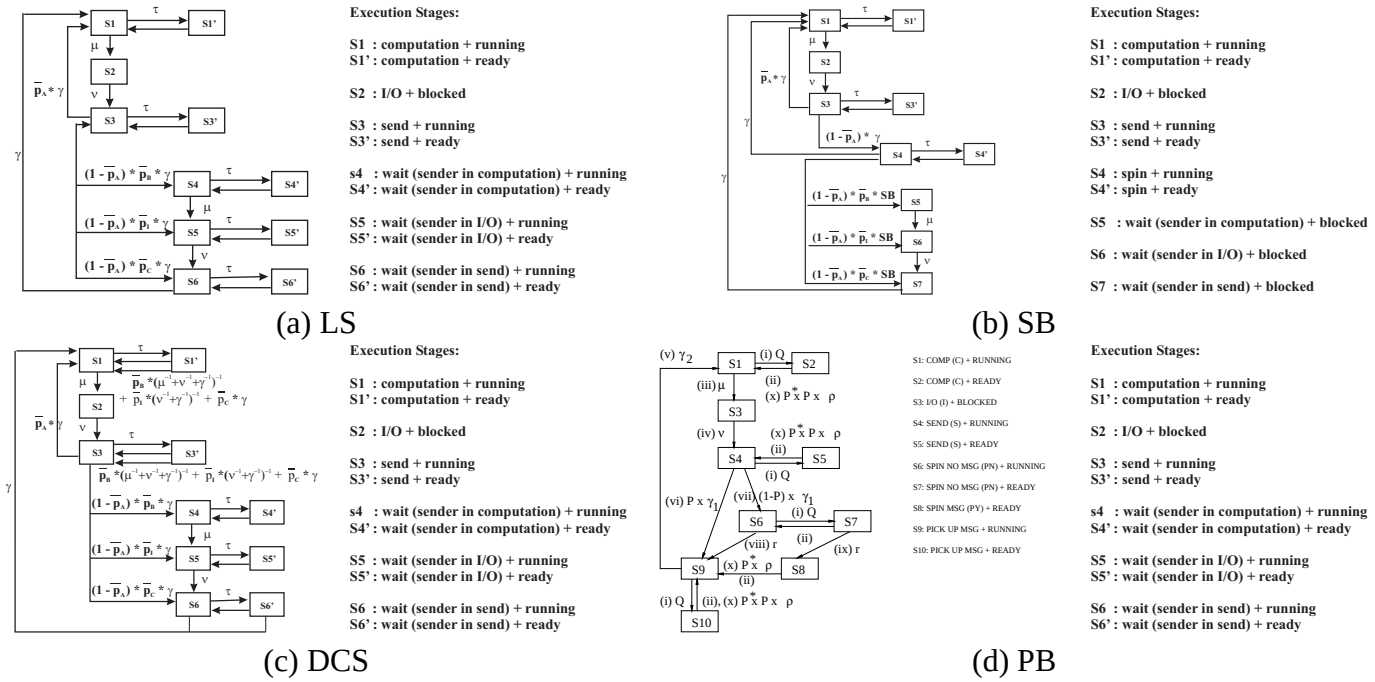


Figure 1: The state transitions for the 4 coscheduling policies.

Scheme-Independent Transitions: In general, a task moves to a ready state from running whenever it is context switched out (shown as transitions $S_i \rightarrow S_i'$ in Fig. 1(a)) at the rate τ of quantum expiration. The reverse transition to the running state (from ready) depends on the number and states of other tasks at that CPU (shown as transitions $S_i' \rightarrow S_i$). (Note that DCS and PB will transit a task from state S_i' to S_i for other reasons as well.) The transition rates for $S1 \rightarrow S2$ and $S2 \rightarrow S3$ are simply a function of the computation and I/O service rates (μ and ν), resp. These transitions remain the same across the 4 dynamic coscheduling schemes considered in this paper.

Scheme-specific Transitions: Once a task leaves state S_3 (finishes sending messages), it either moves to state S_1 (starts next iteration) if the message it will receive is available as determined by the probability $\widehat{p}_A$, or it starts busy-waiting for the message in 1 of the states S_4, S_5 or S_6 depending on the state of the sender task w.p. $(1-\widehat{p}_A) \times \widehat{p}_B$, $(1-\widehat{p}_A) \times \widehat{p}_I$, or $(1-\widehat{p}_A) \times \widehat{p}_C$, resp. The value of $\widehat{p}_A$ can be calculated as discussed in §4.2.2. The transitions $S_4 \rightarrow S_5$ and $S_5 \rightarrow S_6$ are the same as transitions $S_1 \rightarrow S_2$ and $S_2 \rightarrow S_3$, resp. Once the sender task is in the sending state, the considered task will leave state S_6 at rate γ, and move to state S_1 (start next iteration).

Spin Block (SB). A task goes through computation (S_1 and S_1'), I/O (S_2), sending messages (S_3 and S_3'), and waiting for messages. The stage of waiting for messages is further broken down into 5 states: spinning (S_4 and S_4'), waiting (while blocked) for messages from a task that is doing computation (state S_5), from a sender that is doing I/O (state S_6), and from a sender that is sending messages (state S_7); see Fig. 1(b). In addition to the transitions for LS, 2 more transitions are noteworthy for SB. In state S_4, the task spins for time SB^{-1} if the message has not arrived within that time, then it will block itself

and transit to 1 of the 3 states: S_5, S_6 and S_7 depending on the state of the sending task. A subsequent message arrival will transition the task out of the blocked state as shown by the transition $S_7 \rightarrow S_1$.

Demand-based Co-Scheduling (DCS). The main differences between DCS (Fig. 1(c)) and LS (Fig. 1(a)) are in that a message arrival when the task is waiting for a message immediately takes it to the next iteration, represented by the transition $S'_6 \rightarrow S_1$. For the very same reason, the priority boost can again affect the rates at which the task transitions from some of the ready states back to the corresponding running states, i.e., from states S'_1 and S'_3 to $S1$ and $S3$, resp. The rate of message arrival before the considered task starts waiting for it can be calculated as $\widehat{p}_B \times \frac{1}{\mu^{-1}+\nu^{-1}+\gamma^{-1}} + \widehat{p}_I \times \frac{1}{\nu^{-1}+\gamma^{-1}} + \widehat{p}_C \times \gamma$.

Periodic Boost (PB). The states remain the same as in LS, with the only differences in transition rates due to the periodic activity which can boost priorities. These differences will affect the transitions from ready to running states when the task is in computation, sending and waiting stages (from states S'_1, S'_3 and S'_6 to states S_1, S_3 and S_1, resp.) as was the case for DCS; see Fig. 1(d). This rate is defined as $\widehat{p}_A \times \widehat{P}^* \times \rho$, where $\widehat{p}_A$ is the probability of this task having a pending message (calculated as discussed in §4.2.2) when the periodic activity takes place, $\widehat{P}^*$ is the probability that all the other tasks at that CPU which have higher priority than this task do not have a pending message (i.e., $(1 - \widehat{p}_A)^n$ if there are n such tasks), and PB denotes the frequency for this periodic activity.

4.2.2 Fixed-Point Iteration

The analysis of §4.2.1 forms the solution to our decomposed stochastic model, in terms of $\boldsymbol{\pi}^Y$, provided that certain aspects of the parallel system behavior are known. These unknown measures consist of $\widehat{p}_A$, $\widehat{p}_B$, $\widehat{p}_I$, $\widehat{p}_C$ and other measures involved in the mathematical details. Estimates of these measures of the dynamic system behavior are calculated in terms of the decomposed model solution, and a fixed-point iteration is used to obtain the final solution for the processor partition. We now briefly present an overview of our general approach for LS, initially focusing on the specific example of the previous section and then turning to the more general case. The corresponding analysis for each of the other coscheduling strategies is derived in an analogous manner; refer to [17].

To calculate $\widehat{p}_B$, $\widehat{p}_I$ and $\widehat{p}_C$ in terms of the decomposed model solution, recall that these measures represent the set of probabilities that the sender of the message being waited for (by the receiving task being modeled) is active in the computation, I/O and send stages, respectively. Due to the assumptions of stochastically independent and identical processors as part of our approximate matrix-analytic solution, these probability vectors then can be expressed in closed matrix form in terms of $\boldsymbol{\pi}^Y$ and a separate probabilistic analysis of the stochastic process $\{Y(t)\,;\, t \in \mathbb{R}_+\}$. A related derivation yields closed-form expressions for the initial probability vectors $\underline{\beta}'_k$, $\underline{\eta}'_k$ and $\underline{\zeta}'_k$ (following the notation of §3) of the stage

4, 5 and 6 PH-type distributions. The measure $\widehat{p}_A$ represents the probabilities that the modeled task immediately enters the computation stage of the next iteration upon completion of the communication stage because the message to be received has already arrived. Since $T_S = 1$, the sender of this message must either be in stages 4, 5 or 6, and thus we can similarly express $\widehat{p}_A$ in closed matrix form. Finally, the PH-type distributions for stage 5 represent the times that the sender (of the message being waited for by the receiving task being modeled) spends in the I/O stage up until entering the send stage and gaining access to the processor. We construct these distributions from the decomposed model solution by analyzing first passage times in a new Markov process $\{Y'(t)\,;\, t \in \mathbb{R}_+\}$ derived from the original process $\{Y(t)\,;\, t \in \mathbb{R}_+\}$. The stage 5 distributions PH$(\underline{\eta}'_k, \mathcal{S}_k^{I'})$ then can be obtained either directly from this first passage time analysis or by constructing a more compact PH-type distribution to match as many moments (and/or density function) of the first passage times in this process $\{Y'(t)\,;\, t \in \mathbb{R}_+\}$ as are of interest, using any of the best known methods for doing so; e.g., see [7] and the references therein.

Our approach to handling general instances of the stochastic parallel LS model depends in part upon the specific value of T_S for the system of interest. When T_S is relatively small, we simply expand our approach above to capture the cases in which a series of up to T_S additional senders are waiting for messages from other processors by repeating T_S times the execution stages 4, 5 and 6. For example, the ℓ^{th} set of waiting stages, consisting of execution stages $4 + 3\ell$, $5 + 3\ell$ and $6 + 3\ell$, are entered according to a set of rules analogous to those governing transitions from stage 3 to stages 4, 5 and 6 where each transition rule is multiplied by the probability $(1 - \widehat{p}_A)^\ell$, which represents the probability that the sender of the message to be received by the task being modeled is itself waiting on a series of ℓ processors to send their corresponding messages, $\ell = 0, \ldots, T_S$. A set of transition rules analogous to those governing transitions from stage 6 to stage 1 are also appropriately constructed for each of these sets of waiting stages to make the transition to the next set of waiting stages in the series. On the other hand, when T_S is relatively large, we can simply use the execution stages 4, 5 and 6 of our original approach together with a form of the geometric distribution. In particular, upon completing stage 6, the system returns to stages 4, 5 and 6 w.p. $(1 - \widehat{p}_A)$ (following appropriately modified versions of the set of rules governing transitions from stage 3 to stages 4, 5 and 6) and otherwise enters stage 1 according to appropriately modified versions of the set of rules governing transitions from stage 6 to stage 1. Of course, both approaches can be used in combination. Lastly, the sending and receiving of multiple messages is accommodated as follows. The sending of multiple messages is simply incorporated in the corresponding PH-type distribution(s) of the model. Our approach is extended to handle the case of receiving messages from multiple tasks by replacing the expressions and arguments provided above with versions of these expressions and arguments based on the appropriate order statistics. As previously noted, in the case of all-to-all communication, we derive an expression for the distribution of time that the task waits for messages from all other tasks and use a PH-type distribution to estimate this waiting

time distribution.

Let $\kappa = (\widehat{p}_A, \widehat{p}_B, \widehat{p}_I, \widehat{p}_C, \underline{\beta}', \underline{\eta}', \underline{\zeta}')$. Note that κ is expressed in terms of the decomposed model solution, and thus we use a fixed-point iteration to solve the stochastic process. Initial values are chosen for κ and the components of the stationary probability vector $\boldsymbol{\pi}^Y$ are obtained using our matrix-analytic analysis. This solution yields new values for κ via the above equations and the model is solved again with these new values. This iterative procedure continues until the differences between the values of an iteration and those of the previous iteration are arbitrarily small. Numerous experiments were performed with this fixed-point iteration. We note that the fixed-point iteration always converged quite rapidly, and that in all of the cases tested, changing the initial values had no effect on the calculated fixed-point, i.e., the model solution was insensitive to the initial values chosen for κ.

4.3 Performance Measures

Various performance measures of interest can be obtained from the components of the stationary probability vector $\boldsymbol{\pi}$. In particular, the mean number of parallel jobs in the system, or partition, and the mean response time of these jobs can be expressed in closed matrix form in terms of the stationary distribution $\boldsymbol{\pi}^X$ or $\boldsymbol{\pi}^Y$. Another set of performance measures of interest is the long-run proportion of time that a processor spends performing computation, I/O, communication, CS, and some form of waiting. These measures also can be expressed in closed matrix form in terms of $\boldsymbol{\pi}^X$ or $\boldsymbol{\pi}^Y$. Refer to [18].

4.4 Resource Contention

The above analysis solves our parallel dynamic coscheduling models in the case where there is no contention for system resources among the processors in a partition or across the entire system. However, the processing capacity of the entire parallel system can degrade significantly when there is interference among the processors executing parallel applications due to various factors; e.g., increased contention for the communication network. We focus here on network contention, noting that our approach can be extended to consider other sources of contention and their impact on performance. The degradation in system performance due to network contention is complex and depends upon many aspects of the parallel system being considered. Of particular importance are the characteristics of the network, the speed of the processors, and the execution patterns of the parallel applications. We now present an overview of a simple, yet effective, general approach to incorporate these effects in our analysis, where the specific parallel architecture details are implicitly included in the resource contention model parameters.

Let $f_d(k)$ denote the performance degradation factor for the processor partition of interest (comprised of K processors), or the entire system ($K = P$), when k of K processors are performing communication operations, $f_d(0) = 0$ and $f_d(k) < 1$; i.e., the corresponding network communication capacity of the system is given by $(1 - f_d(k))$. Note that values for the function $f_d(k)$ can be estimated using a separate

model (e.g., see [3]) or obtained via measurement on existing computer systems. In our exact analysis, we directly use $f_d(k)$ in all states where k processors are performing communication operations by scaling the PH-type distribution PH$(\underline{\zeta}, \mathcal{S}^C)$ for the per-iteration communication demands of the parallel applications in the analysis of §4.1 according to $\gamma' = \gamma(1 - f_d(k))$ such that $(\gamma')^{-1} = -\underline{\zeta}(\mathcal{S}^C)^{-1}\mathbf{e}$.

In our approximate analysis, due to our stochastic homogeneity assumptions, the long-run proportion of time that k of the K processors are performing communication operations, denoted by $q_K(k)$, is binomially distributed with parameters K and $\widetilde{p}^C$. Using the functions $f_d(k)$ and $q_K(k)$, the average service rate of per-iteration communication stages can thus be expressed as $\gamma' = \gamma/(1-q_K(0)) \sum_{k=1}^{K} q_K(k)(1-f_d(k))$. The normalization factor in this expression accounts for the fact that at least 1 processor must be performing communication operations when the natural service rate of the modeled processor is γ. Hence, the potential system performance degradation due to network contention is incorporated in our approximate analysis by scaling the PH-type distribution PH$(\underline{\zeta}, \mathcal{S}^C)$ in the analysis of §4.2 according to this equation for γ' such that $(\gamma')^{-1} = -\underline{\zeta}(\mathcal{S}^C)^{-1}\mathbf{e}$. The final solution of the resulting stochastic process then can be obtained via appropriately modified versions of the fixed-point iteration.

5 Results

We now exploit our mathematical analysis of §4 to study fundamental properties of the dynamic coscheduling strategies defined in §2 across a wide range of parallel computing environments as represented by the parallel system and workload models of §3.

We first present some results that validate our approximate analysis against detailed simulations, and then we turn to examine in detail the large design and performance spaces of the four dynamic coscheduling mechanisms. In each case, we provide a representative sample of the results from our numerical experiments. The intensity of computation, communication and I/O in a workload is designated by $\frac{\mu^{-1}}{\mu^{-1}+\nu^{-1}+\gamma^{-1}}$, $\frac{\gamma^{-1}}{\mu^{-1}+\nu^{-1}+\gamma^{-1}}$, and $\frac{\nu^{-1}}{\mu^{-1}+\nu^{-1}+\gamma^{-1}}$ respectively, where μ^{-1}, ν^{-1} and γ^{-1} are the mean computation, I/O and communication latencies incurred by a task in each iteration. The default parameter settings for our numerical and simulation experiments are: CS cost 200us, interrupt cost 50us, QL 20ms, γ^{-1} 185.48us, priority change cost 3us, message queue check cost 2us, PB interval 1ms, fixed maximum spin time 200us, message size 4096 bytes. However, these parameters may take on other values when explicitly specified. Many of these values have been drawn from actual system platforms. Most of the results presented in this section are based on nearest neighbor communication pattern.

5.1 Validation

Given that our analysis derived in §4.2 yields an approximate solution, we first must validate the results of this analysis against detailed simulations to demonstrate the accuracy of our approach. Fig. 2 presents

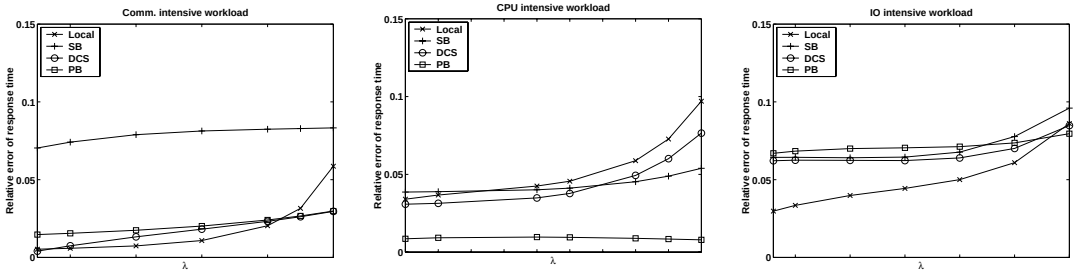
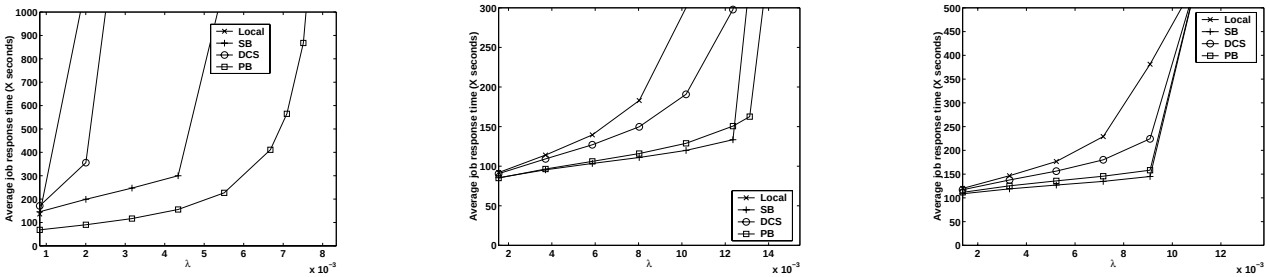


Figure 2: Validation Results

the relative errors of our models and analysis as a function of the arrival rate λ for a 32-processor system. A representative sample of the results are provided for all four dynamic coscheduling strategies under computation-intensive, communication-intensive and I/O-intensive workloads. In each case, our model is in excellent agreement with detailed simulations of a relatively small parallel system, often being within 5% and always less than 10%. Moreover, the accuracy of our approach increases with the size of the system where our primary interests are in large parallel systems consisting of many computing nodes.

5.2 Impact of Load



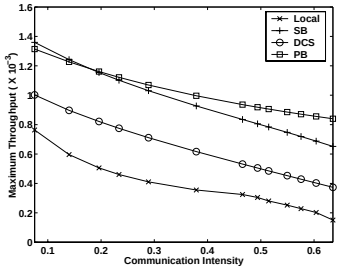
(a) Comm. Intensive workload (μ^{-1} = 0.14ms, γ^{-1} = 0.19ms, ν^{-1} = 0.024ms)

(b) CPU Intensive workload (μ^{-1} = 36ms, γ^{-1} = 0.19ms, ν^{-1} = 2ms)

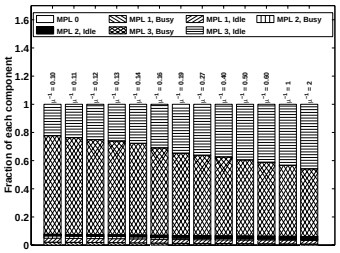
(c) I/O Intensive workload (μ^{-1} = 10ms, γ^{-1} = 0.19ms, ν^{-1} = 5ms)

Figure 3: Impact of Load on Response Time, $\frac{1}{P_{N_A}}(\mu^{-1} + \gamma^{-1} + \nu^{-1}) = 38.19$ second.

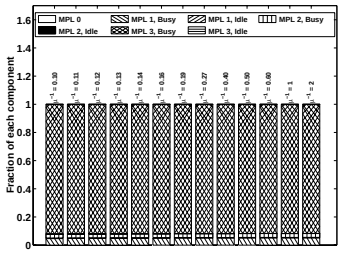
Fig. 3 provides a representative sample of the results for investigating the effects of the arrival rate on the mean job response time under three types of workloads. In general, the differences between the schemes are less significant with lower communication in the workload, as is to be expected. Even in the CPU and I/O intensive workloads, the LS and DCS mechanisms saturate earlier than the other two, and this becomes more apparent in the communication intensive workload. Overall, we can say that LS and DCS are not very desirable. Note that, until now, no one has been able to study these mechanisms for dynamic job arrivals with such a broad spectrum of arrival rates.



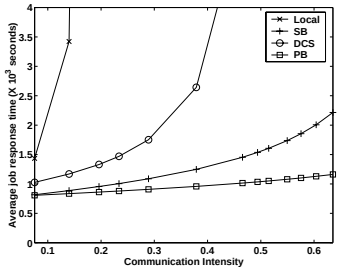
(a) Maximum Throughput



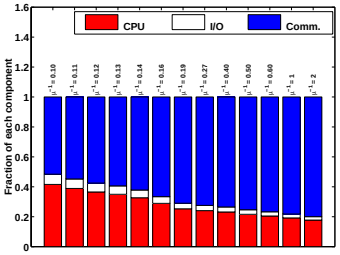
(b) System Profile at Maximum Utilization for SB (time unit is millisecond)



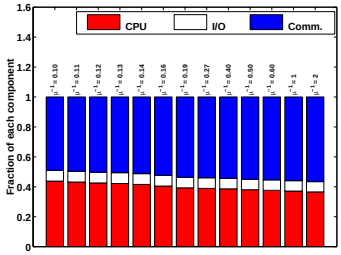
(c) System Profile at Maximum Utilization for PB (time unit is millisecond)



(d) Average Response Time (λ=0.006)

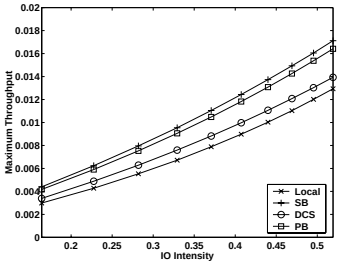


(e) Application Profile for SB (time unit is millisecond, λ=0.006)

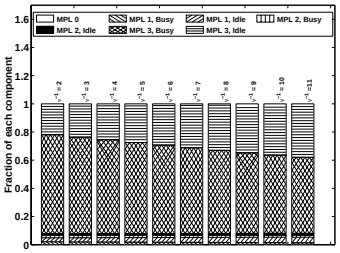


(f) Application Profile for PB (time unit is millisecond, λ=0.006)

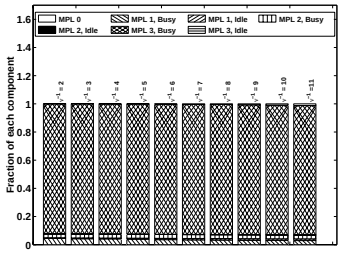
Figure 4: Impact of Communication Intensity on Response Time, ν^{-1}/μ^{-1} = 0.17, $\gamma^{-1} = 0.19$ms, $\frac{1}{P_{N_A}}(\mu^{-1} + \gamma^{-1} + \nu^{-1}) = 38.19$ second.



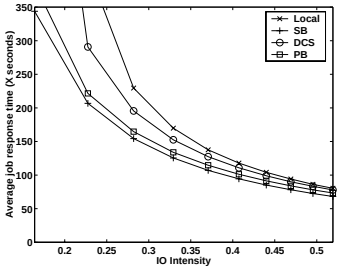
(a) Maximum Throughput



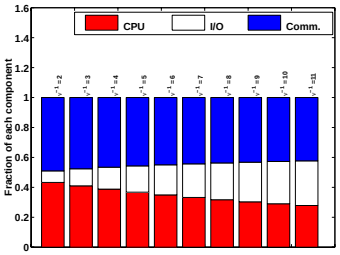
(b) System Profile at Maximum Utilization for SB (time unit is millisecond)



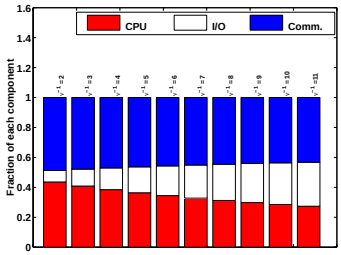
(c) System Profile at Maximum Utilization for PB (time unit is millisecond.)



(d) Average Response Time (λ=0.008)



(e) Application Profile for SB (time unit is millisecond, λ=0.008)



(f) Application Profile for PB (time unit is millisecond, λ=0.008)

Figure 5: Impact of I/O Intensity, μ^{-1} = 10ms, γ^{-1} = 0.19ms, $\frac{1}{P_{N_A}}(\mu^{-1} + \gamma^{-1} + \nu^{-1}) = 38.19$ second.

5.3 Impact of Workload Characteristics

Since communication intensive and I/O intensive workloads are most interesting, we focus on these two workloads in Figs. 4 and 5. In both of these figures: (a) shows the maximum achievable throughput for the given configuration; (b) shows the profile of the system utilization with SB at the maximum

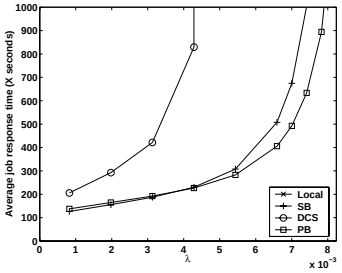
throughput point in terms of what fraction of the time three tasks are present at a processor (which we denote as MPL3, with this in turn broken down into that spent in executing application code, and that when it is idle due to all these three tasks being in a blocked or I/O state), MPL2, MPL1 and having no task to execute (MPL0); and (c) conveys the same information for PB. The performance is presented from the application's perspective in (d) which shows the mean response time for a given arrival rate, and the profile of the time spent by the application in the three operations (at this arrival rate) is given for SB and PB in (e) and (f), respectively.

These figures reconfirm the results in the previous subsection for varying system load, with LS and DCS performing poorly. For the communication intensive workload, PB does better than SB in terms of both throughput and response time. The benefits of PB are accentuated as the communication intensity increases. Clearly, we can see that the fraction of the time spent in communication increases with the intensity in Figs. 4(e) – (f), but the increase is more gradual for PB than for SB. This can be explained based on the behavior in (b) and (c), where we can see MPL3-idle is significantly higher in SB than in PB. At such high communication intensities, blocking to relinquish the processor incurs CS and interrupt overheads with little benefits since there is no other work to do (everyone is blocked). PB which does not switch under those conditions does not experience the CS costs, and thus yields better performance.

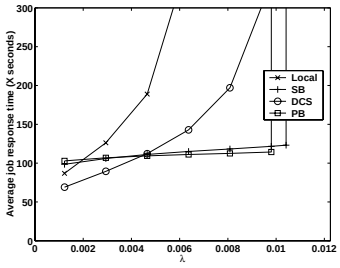
For the I/O intensive workload, the differences among the schemes are less noticeable since the performance is dictated more by the I/O in the application than by the schemes themselves (which do not behave differently for I/O). In fact, as suggested by the curves in Fig. 5(d), our model confirms that the response times under all of the scheduling strategies converge at large I/O intensities. The reason why throughput increases and response time decreases for this case (which is in contrast to the communication intensive figures), is because I/O can be performed concurrently by tasks at a node (while computation/communication cannot) in this instance of our model. The profile graphs show similar behavior to that of the communication intensive workload. At this point, we can also explain why SB does marginally better than PB for I/O intensive (and, incidentally, CPU intensive) workloads. With large computation or I/O fractions, the skewness of the work to be done among the tasks of an application also increases. This can cause tasks to spin more than the message latencies in PB, while SB can limit the effect of such skewness. For the communication intensive workload, this skewness gets smaller, and PB realizes the full benefits of spinning.

One could ask how do the results change with an alternative communication pattern to the nearest-neighbor communication strategy used above. To address this question, Fig. 6 shows the impact of load on the schemes with the *all-to-all communication pattern* that is common in some applications (such as FFT). Overall, we find similar trends as in the nearest-neighbor communication. The only point to note is that the differences between PB and SB become less significant for the communication intensive workload. This communication pattern tends to keep processors automatically more or less

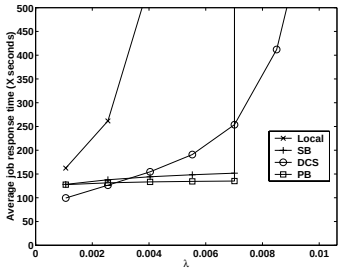
synchronized, thus lessening the effect of scheduling skews.



(a) Comm. Intensive workload (μ^{-1} = 0.14ms, γ^{-1} = 0.19ms, ν^{-1} = 0.024ms)



(b) CPU Intensive workload (μ^{-1} = 36ms, γ^{-1} = 0.19ms, ν^{-1} = 2ms)



(c) I/O Intensive workload (μ^{-1} = 10ms, γ^{-1} = 0.19ms, ν^{-1} = 5ms)

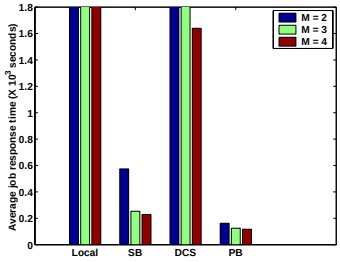
Figure 6: Impact of Load on Response Time, $\frac{1}{P_{N_A}}(\mu^{-1}+\gamma^{-1}+\nu^{-1}) = 38.19$s, All-to-all communication

5.4 Impact of Maximum MPL (M)

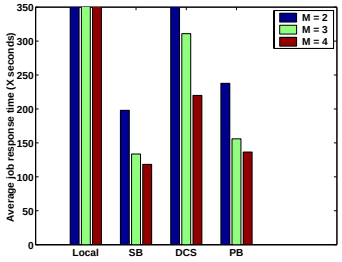
Fig. 7 considers the impact of the maximum MPL (M) on the dynamic coscheduling strategies. Increasing M can help to reduce the processor idling (during I/O or when waiting for a message) by providing more opportunities to switch to another task and execute useful work. As a result, performance improvements due to higher values of M are more noticeable in the I/O intensive workload, or in schemes where the waiting time is high (DCS). This is also the reason why SB needs a higher value of M to become as competitive as PB for the communication intensive workload, but it should be noted that too high a value for M is not appropriate due to practical resource limitations.

5.5 Optimum QL (τ^{-1})

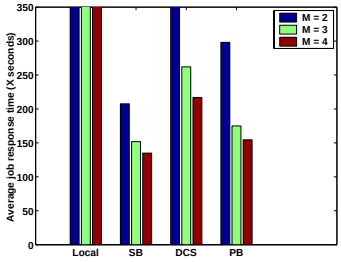
For any scheme, a small time quantum increases the impact of CS overheads, while at the same time a small quantum can mitigate the effects of scheduling skews across processors (to avoid large wait times). These two contrasting factors play a role in determining a good operating point for the time quantum. Fig. 8 captures the effect of these factors for the four schemes on the three workload classes. In general, for the LS and DCS schemes (which are more susceptible to scheduling skews as was shown in earlier results), the second factor that can mitigate scheduling skews is more important, causing the good operating points for LS and DCS in Fig. 8 to be more to the left than those for SB or PB. In fact, SB and PB would prefer a long time quantum, since they do not really rely on the native OS scheduler and perform the task switches whenever needed. As for the effect of the workload itself, CPU and I/O intensive workloads should prefer longer time quanta (because the first factor concerning CS overheads are more important) than the communication intensive workload.



(a) Comm. Intensive workload (μ^{-1} = 0.14ms, γ^{-1} = 0.19ms, ν^{-1} = 0.024ms, λ = 0.003)

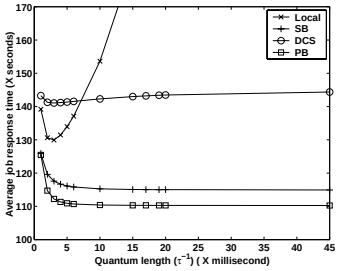


(b) CPU Intensive workload (μ^{-1} = 36ms, γ^{-1} = 0.19ms, ν^{-1} = 2ms, λ = 0.0123)

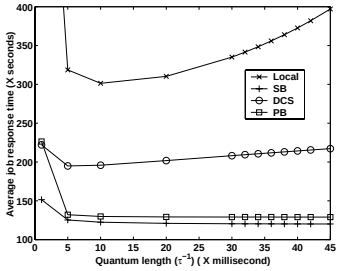


(c) I/O Intensive workload (μ^{-1} = 10ms, γ^{-1} = 0.19ms, ν^{-1} = 5ms, λ = 0.01)

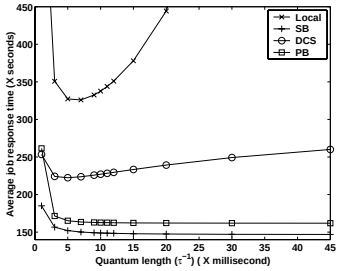
Figure 7: Impact of M on response time, $\frac{1}{P_{N_A}}(\mu^{-1} + \gamma^{-1} + \nu^{-1}) = 38.19$ seconds.



(a) Comm. Intensive workload (μ^{-1} = 2ms, γ^{-1} = 0.19ms, ν^{-1} = 0.34ms, λ = 0.004)



(b) CPU Intensive workload (μ^{-1} = 36ms, γ^{-1} = 0.19ms, ν^{-1} = 2ms, λ = 0.0103)

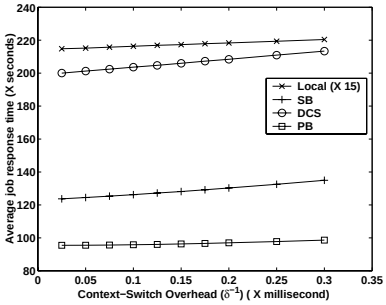


(c) I/O Intensive workload (μ^{-1} = 10ms, γ^{-1} = 0.19ms, ν^{-1} = 5ms, λ = 0.0095)

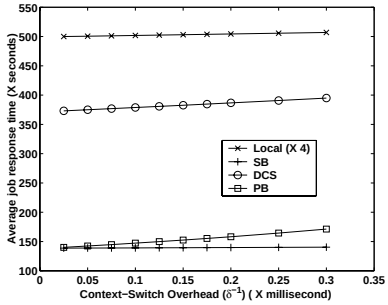
Figure 8: Impact of QL (τ^{-1}) on Response Time, $\frac{1}{P_{N_A}}(\mu^{-1} + \gamma^{-1} + \nu^{-1}) = 38.19$ seconds.

5.6 Impact of System Overheads

Fig. 9(a) studies the impact of the mean CS costs (δ^{-1}) on the relative performance of the schemes for the communication intensive workload. LS and PB are less influenced by CS costs in comparison with SB or DCS, which switch more often depending on message arrivals (rather than on the basis of who can proceed). For the compute intensive workload in Fig. 9(b), PB and DCS are more influenced by the CS overheads since there is the possibility of switching to another task (when a message arrives for it) that was pre-empted while performing useful computation. In such situations, SB will switch to that task only when the current quantum expires or when the currently scheduled task blocks. The reader should



(a) Communication Intensive workload (λ = 0.030, μ^{-1} = 0.14ms, γ^{-1} = 0.19ms, ν^{-1} = 0.002ms)



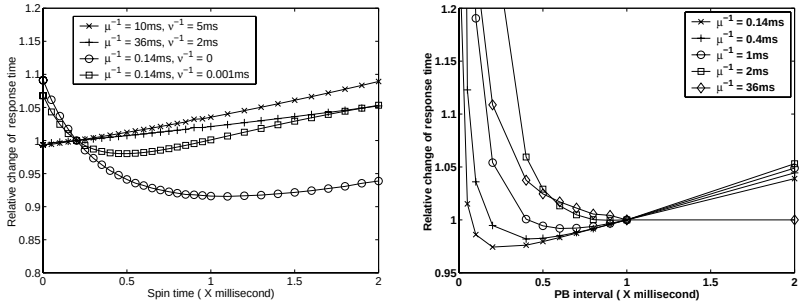
(b) CPU Intensive workload (λ = 0.013, μ^{-1} = 36ms, γ^{-1} = 0.19ms, ν^{-1} = 2ms)

Figure 9: Impact of CS Overheads (δ^{-1}) on Response Time, $\frac{1}{P_{N_A}}(\mu^{-1} + \gamma^{-1} + \nu^{-1}) = 38.19$ seconds.

note that the scales for LS are different (its values have been scaled down).

5.7 Optimum Spin Time

For SB, the choice of the spin time is a crucial issue. Fig. 10(a) shows the effect of the spin time on four different workloads. The curves are normalized with respect to the performance for the default spin time. Fig. 10(a) shows that the ideal spin time (giving the lowest response time) is very sensitive to the workload. This makes the selection of a good spin time on a real system very difficult, but it further highlights the importance of the use of our models and analysis. It also should be noted that in the results presented in previous sections, the chosen spin times are reasonably close to their ideal values (no more than 5–10% off).



(a) Spin Time for SB (λ = 0.003) (b) PB Interval (λ=0.003, ν^{-1}/μ^{-1} = 0.17)

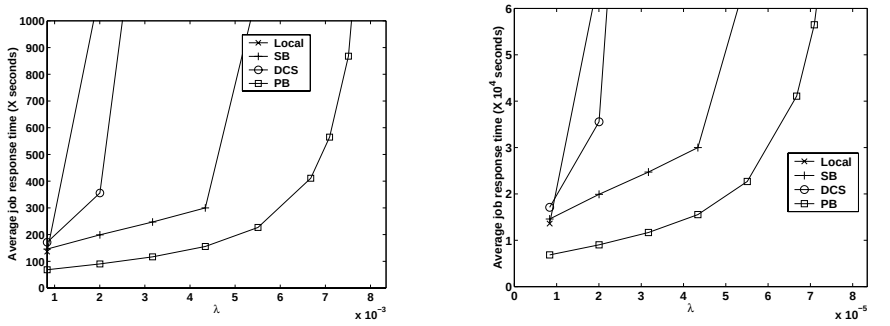
Figure 10: Design Choices for SB and PB, $\frac{1}{P_{N_A}}(\mu^{-1} + \gamma^{-1} + \nu^{-1}) = 38.19$ seconds.

5.8 Optimum PB Frequency

One of the important design considerations for PB is selecting the frequency of the kernel activity. Across the spectrum of workloads ranging from very high to very low communication intensities, we find that the ideal frequency of invocation lies between 0.3ms to 1ms in Fig. 10(b) (the lines are normalized with respect to a 1ms frequency). These results suggest that an invocation frequency of between 0.5 to 1ms would provide good performance across the entire workload spectrum for PB.

5.9 Impact of Job Duration

Fig. 11 compares the response times of two job durations (one of which is 100 times that of the other) as a function of the load for the four schemes. While the absolute values of the mean response times increase with the job duration and the saturation points are reached at a lower arrival rate, we find that the overall trends and differences between the schemes are quite similar. Hence, PB and SB continue to appear to be good choices, regardless of the job duration.



(a) $\frac{1}{P_{N_A}}(\mu^{-1} + \gamma^{-1} + \nu^{-1}) = 38.19$ seconds

(b) $\frac{1}{P_{N_A}}(\mu^{-1} + \gamma^{-1} + \nu^{-1}) = 3819$ seconds

Figure 11: Impact of Job Duration (Communication intensive, μ^{-1} = 0.14ms, γ^{-1} = 0.19ms, ν^{-1} = 0.024ms)

5.10 Impact of Resource Contention

We consider the impact of network contention on parallel system performance under the 4 dynamic coscheduling strategies via the analysis of §4.4. The corresponding set of results are not included due to space limitations. We note, however, that these results demonstrate the very same general trends as have been observed above, although with an overall degradation in performance caused by network contention which exacerbates the effects described above. This degradation in performance is most significant at moderate to fairly high traffic intensities, and especially for communication intensive workloads.

6 Concluding Remarks

The increasing deployment of loosely coupled off-the-shelf clusters of workstations has led to the development of a new class of mechanisms, called dynamic coscheduling, that attempt to coschedule communicating tasks whenever needed without explicit synchronization. Our study has addressed a critical void in scheduling for these parallel systems by developing and validating a general mathematical model within a unified framework in order to study the design and performance spaces of dynamic coscheduling mechanisms across a wide range of system and workload parameters. We derive an exact matrix-analytic analysis for relatively small parallel systems, and an approximate matrix-analytic analysis based on a general stochastic decomposition and fixed-point iteration. Results from numerical experiments with our model are in excellent agreement with those from detailed simulations of fairly small systems, often within 5% and always less than 10%. Given the complex dynamic interactions among the different aspects of both the parallel computing environment and the dynamic coscheduling strategy, these results provide further evidence of the benefits of our general approach especially for large parallel systems.

Our numerical experiments show that it is not advisable to allow the native OSs at each node to switch between tasks at their disposition. Some amount of coordination is definitely needed. Of the three previously proposed dynamic coscheduling mechanisms, DCS does not fare as well as the others

across a broad spectrum of workload and system parameters. SB and PB have their relative merits, with the latter faring better whenever the workload is more communication intensive, or when the overheads for interrupts are higher. PB is also preferable whenever nodes are not operating at the full multiprogramming level. Our model and analysis can be used as a design tool to fine tune the parameters for these mechanisms (spin time in SB and periodic frequency in PB) to derive good operating points. With SB, the choice of the spin time is important for determining performance, while a frequency of once every 0.5–1 millisecond provides good performance for PB across the entire workload spectrum considered. Both of these mechanisms are relatively immune to the native OS switching activity, by taking over whenever the communication events call for coscheduling and becoming less intrusive otherwise. These mechanisms are good choices regardless of whether the system is subject to short running interactive jobs or long running parallel applications. All of these mechanisms have been evaluated based on a user-level communication mechanism. It should be noted that the results presented here for SB are with a fixed spin time, though the model itself allows adaptive tuning of this value (which more recently has been shown to be a better alternative).

Our model and analysis is able to answer several important issues such as the optimal frequency for invoking the periodic boost mechanism, the optimal frequency of context switching, a direct way of calculating the optimal fixed spin time for SB, and the impact of workload characteristics on these issues. We would also like to point out that while the results and experiments have explored a wide and representative range of workload and system parameters, there still are other considerations that one could use our model to explore, such as more communication patterns.

References

[1] A. C. Arpaci-Dusseau, D. E. Culler, A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proc. ACM SIGMETRICS Conf. Meas. and Model. Comp. Sys.*, 1998.

[2] D. Bailey et al. The NAS Parallel Benchmarks. *Intl. J. Supercomputer Appl.*, 5(3):63–73, 1991.

[3] D. Culler, J. P. Singh. *Parallel Computer Architecture: A Hardware-Software Approach*. Morgan Kauffman, 1998.

[4] A. C. Dusseau, R. H. Arpaci, D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proc. ACM SIGMETRICS Conf. Meas. and Model. Comp. Sys.*, pp. 25–36, 1996.

[5] D. G. Feitelson, L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *J. Par. and Dist. Comp.*, 16(4):306–318, Dec. 1992.

[6] G. Latouche, V. Ramaswami. *Introduction to Matrix Analytic Methods in Stochastic Modeling*. ASA-SIAM, 1999.

[7] G. Latouche, P. Taylor. *Advances in Algorithmic Methods for Stochastic Models*. Notable, 2000.

[8] S. Nagar, A. Banerjee, A. Sivasubramaniam, C. R. Das. A Closer Look at Coscheduling Approaches for a Network of Work stations. In *Proc. ACM Symp. Par. Alg. and Arch.*, pp. 96–105, Jun. 1999.

[9] V. K. Naik, S. K. Setia, M. S. Squillante. Processor allocation in multiprogrammed, distributed-memory parallel computer systems. *J. Par. and Dist. Comp.*, 46(1):28–47, Oct. 1997.

[10] M. F. Neuts. *Matrix-Geometric Solutions in Stochastic Models: An Algorithmic Approach*. The Johns Hopkins Univ. Press, 1981.

[11] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proc. Intl. Conf. Dist. Comp. Sys.*, pp. 22–30, May 1982.

[12] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Perf. Eval.*, 19:107–140, 1994.

[13] J. P. Singh, W.-D. Weber, A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Tech. Rep. CSL-TR-91-469, Computer Systems Lab, Stanford Univ., 1991.

[14] P. G. Sobalvarro. *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors*. PhD thesis, Dept. EECS, MIT, Cambridge, MA, Jan. 1997.

[15] M. S. Squillante. Matrix-analytic methods in stochastic parallel-server scheduling models. In *Advances in Matrix-Analytic Methods for Stochastic Models*. Notable, 1998.

[16] M. S. Squillante, F. Wang, M. Papaefthymiou. Stochastic analysis of gang scheduling in parallel and distributed systems. *Perf. Eval.*, 27&28:273–296, Oct. 1996.

[17] M. S. Squillante, Y. Zhang, A. Sivasubramaniam, N. Gautam. Generalized parallel-server fork-join queues with dynamic scheduling of iterative multi-stage tasks. Tech. Rep., IBM Res. Div., 2005.

[18] M. S. Squillante, Y. Zhang, A. Sivasubramaniam, N. Gautam, H. Franke, J. Moreira. Modeling and analysis of dynamic coscheduling in parallel and distributed environments. In *Proc. ACM SIGMETRICS Conf. Meas. and Model. Comp. Sys.*, pp. 43–54, Jun. 2002.

[19] Y. Zhang, A. Sivasubramaniam, J. Moreira, H. Franke. A Simulation-based Study of Scheduling Mechanisms for a Dynamic Cluster Environment. In *Proc. ACM Intl. Conf. Supercomputing*, pp. 100–109, May 2000.