Probablistic Self-Scheduling

Milind Girkar[1] Arun Kejariwal[2] Xinmin Tian[1] Hideki Saito[1]
Alexandru Nicolau[2] Alexander Veidenbaum[2] Constantine Polychronopoulos[3]

[1] Intel Corporation
3600, Juliette Lane
Santa Clara, CA, 95050, USA
[2] Center for Embedded Computer Systems
University of California at Irvine
Irvine, CA 92697, USA
[3] Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

Abstract. *Scheduling for large parallel systems such as clusters and grids presents new challenges due to multiprogramming/polyprocessing [1]. In such systems, several jobs (each consisting of a number of parallel tasks) of multiple users may run at the same time. Processors are allocated to the different jobs either statically or dynamically; further, a processor may be taken away from a task of one job and be reassigned to a task of another job. Thus, the number of processors available to a job varies with time. Although several approaches have been proposed in the past for scheduling tasks on multiprocessors, they assume a dedicated availability of processors. Consequently, the existing scheduling approaches are not suitable for multiprogrammed systems. In this paper, we present a novel probabilistic approach for scheduling parallel tasks on multiprogrammed parallel systems. The key characteristic of the proposed scheme is its self-adaptive nature, i.e., it is responsive to systemic parameters such as number of processors available. Self-adaptation helps achieve better load balance between the different processors and helps reduce the synchronization overhead (number of allocation points). Experimental results show the effectiveness of our technique.*

1 Introduction

Scheduling for parallel systems is done at two levels. At the first level, jobs (of different users) are scheduled such that each job receives a fair share of the resources. On the other hand, tasks of a job are scheduled on different processors such that the overall completion time (also known as *makespan*) is minimized. In context of dynamic scheduling schemes, this also involves minimizing the runtime scheduling overhead. Processors may be allocated (to a job) either statically or dynamically. Further, a processor may be taken away from a task of one job and be reassigned to a task of another job. As a consequence, the number of processors available to a job varies with time. To validate this, we recorded the number of user-level processes on a real multiprogrammed system for an hour (see Figure 1). Clearly, the number of idle processors varies with time, by as much as 10%, which has a direct effect on the performance of a scheduling

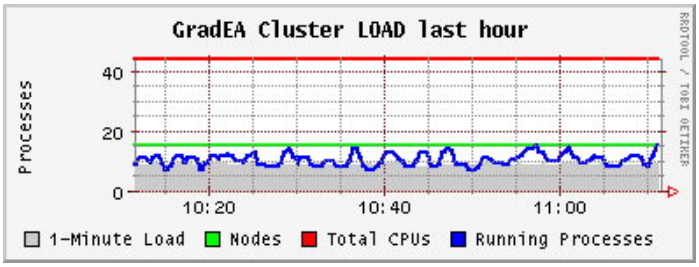


Fig. 1. Variation in the number of processes over an hour on a real multiprogramming system with 15 nodes (44 CPUs) (http://www.gradea.uci.edu)

policy. Therefore, a scheduling policy should be designed such that it is aware of such systemic variations.

In this paper, we address the problem of scheduling parallel tasks of a given job. Without loss of any generality, we focus on scheduling iterations of a `DOALL` loop [2]; note that the proposed technique is general in nature, e.g., it can also be used for scheduling coarse-grain (function-level) parallel independent tasks. We model the problem as a task allocation problem wherein at any scheduling step, given a set of idle processors, one or more iterations are allocated to each processor. The key consideration in task allocation is the selection of the task size, i.e., the number of iterations constituting a task. While a small task size incurs significant scheduling overhead, a large task size results in load imbalance. Thus, the task allocation problem naturally reduces to determining the optimal task size in order to minimize the total execution time. For this, several self-scheduling techniques have been proposed for scheduling parallel loops [3]. However, none of the existing techniques account for the variation in the number of available processors with time. For this, we propose a novel approach, referred to as *Probabilistic Self-Scheduling* (PSS), for scheduling of (nested) parallel loops on multiprogrammed parallel systems. At any scheduling step, the number of iterations allocated to an idle processor is determined based on the number of remaining iterations and the number of processors expected to be available in future. The latter is determined based on the the number of processors available in the past. The proposed approach is compatible with the environment established by auto-scheduling compilers [4].

The rest of the paper is organized as follows. In the next section, we present a motivating example. Section 3 presents our approach PSS. Experimental setup and results are presented in Section 4. Previous work is discussed in Section 5. Finally, in Section 6, we conclude with directions for future research.

2 A Motivating Example

In this section we illustrate the intuitive idea behind our approach (PSS) with the help of an example. For comparison purposes, we consider two well known self-scheduling techniques, viz., *guided self-scheduling* GSS(1) [5] and *factoring* [6]. Assuming identical processors, at a given scheduling step GSS(1) assigns $\frac{1}{\mathbf{P}}$ of the remaining iterations to an idle processor, where $\mathbf{P}$ is the total number of processors; factoring assigns iterations to the processors in batches of $\mathbf{P}$ chunks,

where the batch size is half the number of remaining iterations, for example, given 100 iterations and 4 processors, the initial batch size is 50 (= 100/2) and the chunk size of the first four chunks is 13 (= $\lceil 50/4 \rceil$). Consider a multiprogrammed system consisting of 4 processors. Let processors P_1 and P_2 be available for time $t \geq 0$ and $t \geq 22$ respectively and let processors P_3 and P_4 be busy serving other jobs in the system. For a `DOALL` loop with 100 iterations (for simplicity of exposition, we assume that each iteration has a workload of 1 unit), the chunk sizes for GSS(1) and factoring are shown in Table 1. From the table we note that GSS(1) and factoring incur large synchronization overhead due to large number of allocation points. This can be attributed to the fact that GSS(1) and factoring are oblivious of the number of processors available. In contrast, PSS assigns $\frac{1}{E[P]}$ of the remaining iterations to an idle processor, where $E[P]$ is the average number of processors available to the job under consideration. In the current context, $E[P] = 2$, as processors P_3 and P_4 are never available for scheduling. The chunk sizes for PSS is shown in Table 1. From the table we see that PSS reduces the synchronization overhead by 50% w.r.t. GSS(1) and by 65% w.r.t. factoring. Clearly, PSS yields better performance than GSS(1) and factoring as it incurs far less synchronization overhead.

Scheme	Chunk Sizes	# of Allocation Points
GSS(1)	25 19 14 11 8 6 5 3 2 2 2 1 1 1	14
Factoring	13 13 13 13 6 6 6 6 3 3 3 3 2 2 2 2 1 1 1 1	20
PSS	50 25 13 6 3 2 1	7

Table 1. Total number of iterations = 100, **P** = 4

3 The Approach

In this section we present the algorithm for our approach - *Probabilistic Self-Scheduling*. Although several models have been proposed, viz., global, local and hybrid, for work queues in context of self-scheduling, we adopt the model proposed by Polychronopoulos and Kuck in [5] owing to its simplicity. Note that model selection is orthogonal to the concerns we address in this paper. The algorithm is designed for non-preemptive scheduling, whereby a task once assigned to a processor may not be removed until it has finished execution. The rest of the section describes the different phases of our scheduling algorithm.

3.1 Expected processor availability

As discussed in the previous section, the presence of other jobs in a multiprogramming environment has direct impact on the performance of a self-schedule. In order to address the above, at a given scheduling step t, PSS computes the chunk size (discussed further in subsection 3.2) based on the number of remaining iterations and the expected value of the number of processors available after step t assuming that it would be the same as the average number of processors available in the past. Before discussing how to compute the above, we defines some terms of probability (for a detailed discussion, the reader is referred to the book by Meyer [7]).

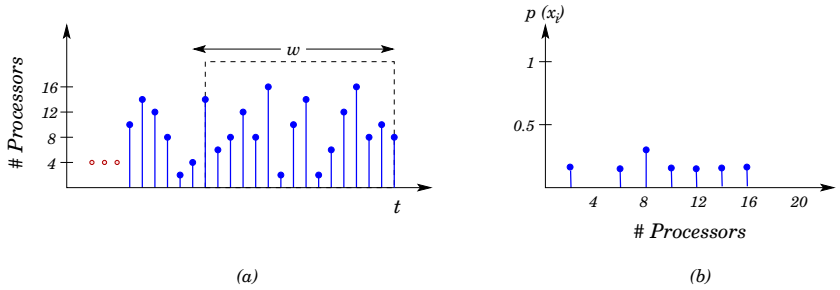


Fig. 2. An illustration of how to determine the probability distribution from the processor availability record. a) For a given time t, processor availability in the past; b) probability distribution of processor availability during the window w. (Total number of processors in the multiprogrammed system $\mathbf{P} = 20$)

Preliminaries

Let X be a discrete random variable and its range space, denoted by R_X, consist of a countably infinite number of values, $x_1, x_2, \ldots$. With each possible outcome x_i, we associate a number $p(x_i) = P(X = x_i)$, called the probability of x_i. The numbers $p(x_i), i = 1, 2, \ldots$ must satisfy the following:

$$p(x_i) \geq 0, \quad \forall i$$

$$\sum_{i=1}^{\infty} p(x_i) = 1.$$

The function p defined above is called the *probability function* of the random variable X. The collection of pairs $(x_i, p(x_i))$, for $i = 1, 2, \ldots$ is called the *probability distribution* of X.

Definition 1. *The expected value of a discrete random variable X, denoted by $E(X)$, is defined as:*

$$E(X) = \sum_{i=1}^{\infty} x_i p(x_i) \tag{1}$$

if the series $\sum_{i=1}^{\infty} x_i p(x_i)$ converges absolutely, i.e., if $\sum_{i=1}^{\infty} |x_i| p(x_i) < \infty$. $E(X)$ is also referred to as the mean value *of X.*

Processor availability

We model the number of processor available at each scheduling step as a discrete random variable P. At each scheduling step t, we record the number of available processors. Also, we determine the expected value of the number of processors available subsequently. For this, we define a window of width w to compute the above. The processor availability during this window can be represented as a histogram, as illustrated in Figure 2(a). From this histogram, the probability distribution of processor availability is computed [7]. For example, for the window

shown in Figure 2(a), $p(x_i = 8) = 4/16 = 0.25$, as shown in Figure 2(b). Finally, the expected value is computed using Equation 1.

The window width is parameterized. A larger width increases the accuracy of the update process, however, it incurs more overhead. It has been shown that run-time performance measurement via use of hardware performance counters incurs minimal scheduling overhead [8]. Note that the expected value computed above is not fixed. This is due to the fact that the processor availability profile in two different windows need not be the same, as evident from Figure 1. Also, the expected value cannot be determined statically as the processor availability profile in a given window is non-deterministic. Hence, under PSS, the expected value is "updated" at every scheduling step.

3.2 Chunk Size

Under GSS, at a given scheduling step, the chunk size (denoted by Λ) is determined as follows:

$$\Lambda = \left\lceil \frac{W_R}{\mathbf{P}} \right\rceil \tag{2}$$

where, W_R is the number of remaining iterations. However, as discussed in [6], the above may result in allocation of too much work to early chunks; specifically, two-thirds of the work is assigned to first $\mathbf{P}$ chunks in case of identical processors. It has been shown that 50% of the total number of iterations is sufficient to even out the finishing times of the processors [6]. Therefore, we introduce a correction factor to "relax" the exponential decay of chunk size. Assuming identical processors, the number of iterations remaining after $\mathbf{P}$ allocations can be approximated as $(1 - \frac{1}{\eta \mathbf{P}})^{\mathbf{P}} \mathbf{W_R}$, where η is the correction factor and $\mathbf{P}$ is the number of processors. From the above, η must satisfy the following:

$$\lim_{\mathbf{P} \to \infty} \left(1 - \frac{1}{\eta \mathbf{P}}\right)^{\mathbf{P}} = 0.5$$

Therefore, $\eta = 1.5$. The modified formula for the function Λ is as follows:

$$\Lambda = \left\lceil \frac{W_R}{1.5\ \mathbf{P}} \right\rceil \tag{3}$$

Based on our discussion in the previous subsection, we adapt Equation 3 for multiprogramming systems as follows:

$$\Lambda = \left\lceil \frac{W_R}{1.5\ E[P]} \right\rceil \tag{4}$$

So far, the chunk size is computed oblivious of the variation in the number workload (execution time) of the different iterations. To account for this, Equation 4 can be further refined as proposed in [9]. A detailed discussion of an integrated approach is beyond the scope of the paper.

Algorithm 1 Probabilistic Self-Scheduling

Input : A `DOALL` loop with **N** iterations and **P** processors.

Output : A near-optimal dynamic schedule w.r.t. load balance amongst the different processors and schedule length

$\mathbf{W_R} \leftarrow \mathbf{N}$

/* Generate the schedule (assuming implicit loop coalescing [10][1]) */

Let $P_{\text{idle}} \subseteq \mathbf{P}$ be a set of idle processors at any given scheduling step

repeat
  if $|P_{\text{idle}}| \; ! = 0$ **then**
    Determine $E[P]$
    for all $p_i \in P_{\text{idle}}$ **do**
      /* Compute the chunk size */

$$\Lambda = \max\left(W_{\min}, \left\lceil \frac{\mathbf{W_R}}{1.5E[P]} \right\rceil\right) \tag{5}$$

      Compute index range for each processor
      Allocate the iterations corresponding to index range to p_i
    end for
  end if
  $\mathbf{W_R} \leftarrow \mathbf{W_R} - |P_{\text{idle}}| \times \Lambda$
until $\mathbf{W_R} > 0$

where, $W_{\min}$ is the minimum chunk size (pre-specified by the user).

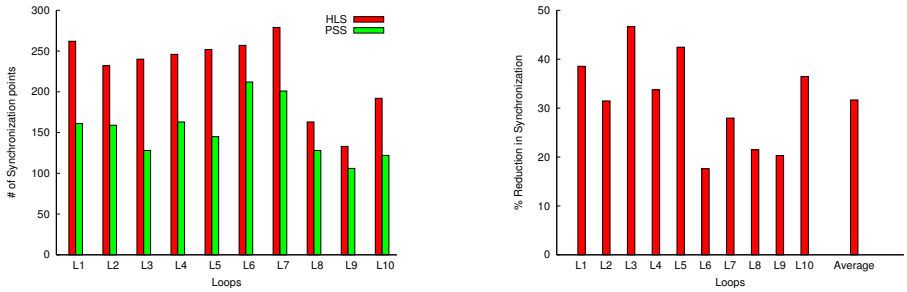
3.3 The Algorithm

In this section we present a formal description of the algorithm for PSS. At each scheduling step, Algorithm 1 first determines the expected number of available processors (refer to subsection 3.1). Subsequently, it determines the chunk size Λ (given by Equation 5), i.e., the number of iterations to be allocated to an idle processor p_i. Next, it determines the range of the iterations to be mapped to each processor and maps the corresponding iterations on to processor p_i. Note that PSS is an online algorithm as the chunk size is determined at run-time based on $\mathbf{W_R}$ and $E[P]$.

4 Experiments

We obtained traces of processor availability on a real multiprogramming system with 15 nodes (44 CPUs) (`http://www.gradea.uci.edu`). Also, we extracted several kernels (`DOALL` loops $L_1, L_2, \ldots, L_{10}$) from SPEC OMP 2001M [11] and other scientific applications such as LAMMPS [12] and DAKOTA [13]. We used the above two as inputs to our simulator [14] to compare the performance of PSS with *adaptive self-tuning scheduling* [15] (referred to as HLS in the rest

[1] Loop coalescing transforms multiply nested `DOALL` loops into singly nested loops.



(a) (b)

Fig. 3. a) Number of synchronization points for the different kernels; b) % Reduction in synchronization

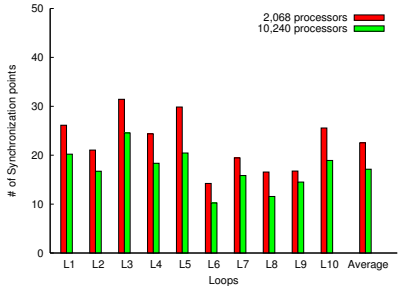


Fig. 4. % Reduction in synchronization (w.r.t. HLS) on systems with 2,068 and 10,240 processors

of the paper). For consistency purposes (w.r.t. the task granularity), we only consider the "upper algorithm" of HLS which does scheduling at the iteration level. HLS samples the performance of a number of self-scheduling techniques, such as guided self-scheduling, factoring, trapezoidal self-scheduling et cetera, at runtime to determine the best scheme for each loop in a given application program. Thus, HLS is in essence the best of all the self-scheduling techniques proposed so far. Due to this, we demonstrate the effectiveness of our approach over HLS only.

4.1 Results

We conducted two sets of experiments: (i) First, we evaluated the effectiveness of our approach and compare it with HLS for a small multiprogramming system (http://www.gradea.uci.edu) with 15 nodes (44 CPUs); (ii) Second, assuming random processor availability, we evaluated the effectiveness of PSS for number of processors — 2,068 as in the Bigben [16] and 10,240 processors such as in the Columbia supercomputer [17]. Note that the applicability of our approach is not restricted to any particular processor configuration.

Figure 3 presents a performance (number of synchronization points) comparison of PSS with HLS. In order to minimize the effect of uneven start times of the processors, the number of synchronization points required was computed as an average of 10 simulation runs. We observe that PSS reduces the synchronization

overhead by a maximum of 46.67% and by 31.67% on an average. The decrease in synchronization directly increases performance at the application level. The better allocation of the processors will also tend to increase the performance at the system level. The latter can be attributed to the reduced contention for accessing the interconnection network which yields higher throughput.

Next, we evaluated the performance of PSS for large parallel systems, such as the Bigben [16] and the Columbia supercomputer [17]. Since we did not have processors availability traces for such systems, we simulated the same using a random number generator, as in [5]. Figure 4 presents the results for the performance (% reduction in synchronization) of PSS w.r.t. HLS for $2,068$ and $10,240$ processors. From the figure, we see that PSS reduces synchronization overhead by a maximum of 29.86% and 22.55% for a system consisting of $2,068$ and $10,240$ processors respectively and by 22.54% and 17.14% on an average respectively. In case of heavy workloads (i.e., when there are a large number of jobs of other users running on the system) PSS can potentially yield higher reduction in the synchronization overhead. This can be explained as follows: in such cases the expected number of available processors is small which results in large chunk sizes, see Equation 4. This is in turn leads to reduction in the synchronization overhead.

5 Previous Work

Early work on scheduling for multiprogrammed parallel systems addressed problems such as the effect of program concurrency on the throughput of batch processing systems [18,19]. Later, Ousterhout proposed *co-scheduling*, where groups of cooperating processes are assigned processors at the same time [20]. In order to facilitate sharing of the multiprocessor system amongst several groups of processors, a group of cooperating processes would execute on the processors in time-multiplexed fashion. Rommel et al. analyzed the processor sharing discipline in context of parallel jobs running on uniprocessor systems [21]. Approaches for multiprogramming distributed memory systems are discussed in [22,23]. Policies for processor allocation in multiprogrammed environments are discussed in [24,25]. Program characterization and performance evaluation of scheduling algorithms in multiprogrammed systems is discussed in [26,27,28].

Probabilistic scheduling approaches have been proposed in several different fields of research. In [29], Chandy and Reynolds proposed an approach for scheduling partially ordered tasks with probabilistic execution times. Bruno and Downey studied the probabilistic bounds on list scheduling in [30]. Tongsima et al. [31] proposed confidence-based probabilistic scheduling of data-flow graphs. In [32], Som et al. presented a probabilistic event scheduling policy for optimistic parallel discrete event simulation. Burns et al. [33] proposed a scheduling policy based on probabilistic guarantees for fault-tolerant real-time systems. In [34], Fujita and Zhou proposed a multiprocessor scheduling problem with probabilistic execution costs. Li and Pan presented a probabilistic analysis of scheduling precedence constrained parallel tasks on multicomputers with contiguous proces-

sor allocation [35]. Moulin [36] proposed a probabilistic approach for split-proof[2] scheduling of parallel jobs to ensure fairness between the different users. Özsoy [37] investigated the effect of coordinated splitting by several users and proposed that the *uniform rule* — given n jobs, choose each ordering of the n jobs (for scheduling) with an equal probability of $1/n!$ — is the only rule immnue to coordinated splitting. Recently, Glatard et al. [38] proposed a probabilistic approach for job partitioning and scheduling on a grid infrastructures. The problem addressed in each of the aforementioned works is orthogonal to the problem addressed in this paper (load balancing between the different processors). Furthermore, the techniques proposed in prior work assume that a "fixed" number of processors are available for scheduling each job. This assumption is not representative of the multiprogrammed systems thereby restricting their applicability.

6 Conclusion

In this paper we presented an algorithm for self-scheduling of parallel tasks in multiprogrammed systems. The key characteristic of our approach is the dynamic adaptation of the chunk size based on the variation in the number of available processors. The approach achieves dual objectives: (i) it achieves load balance between different processors; and (ii) reduces the synchronization overhead by reducing the number of allocations points. As future work, we would like to extend our approach to address other issues such as minimizing maximum tardiness.

References

1. C. D. Polychronopoulos. Multiprocessing vs multiprogramming. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages II–223–II–230, August 1989.
2. S. Lundstrom and G. Barnes. A controllable MIMD architectures. In *Proceedings of the 1980 International Conference on Parallel Processing*, St. Charles, IL, August 1980.
3. A. Kejariwal and A. Nicolau. Reading list of self-scheduling of parallel loops. http://www.ics.uci.edu/~akejariw/SelfScheduleReadingList.pdf.
4. C. D. Polychronopoulos. Towards autoscheduling compilers. *Journal of Supercomputing*, 2(3):297–330, 1988.
5. C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, 1987.
6. S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: a method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, 1992.
7. P. L. Meyer. *Introductory Probability and Statistical Applications*. Reading, MA, 1970.
8. A. B. Downey and D. G. Feitelson. The elusive goal of workload characterization. *SIGMETRICS Performance Evaluation Review*, 26(4):14–29, 1999.
9. A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos. Feedback-based guided self-scheduling. In *Proceedings of the 12th SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 2006.
10. C. Polychronopoulos. Loop coalescing: A compiler transformation for parallel machines. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 235–242, August 1987.
11. SPEC OMP. http://www.spec.org/omp.
12. LAMMPS. http://www.cs.sandia.gov/~sjplimp/lammps.html.
13. DAKOTA. http://endo.sandia.gov/DAKOTA/software.html.

[2] Under *splitting*, a user breaks down his job into multiple smaller jobs under different aliases. This can potentially reduce the expected wait time if the shortest jobs are served first.

14. A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos. An efficient approach for self-scheduling parallel loops on multiprogrammed parallel computers. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, Hawthorne, NY, October 2005.
15. Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss. An adaptive OpenMP loop scheduler for hyperthreaded SMPs. In *Proceedings of the 17th International Conference for Parallel and Distributed Computing Systems*, San Francisco, CA, 2004.
16. Bigben: Pittsburgh Supercomputing Center. http://www.psc.edu/machines/cray/xt3/bigben.html.
17. SGI Altix: Columbia Supercomputer. http://www.nas.nasa.gov/Resources/Systems/columbia.html.
18. J. C. Browne, K. M. Chandy, J. Hogarth, and C. Lee. The effect on throughput in multiprocessing in a multi-programming environment. *IEEE Transactions on Computers*, C-22(8):728–735, August 1973.
19. C. H. Sauer and K. M. Chandy. The impact of distributions and disciplines on multiple processor systems. *Communications of the ACM*, 22(1):25–34, 1979.
20. J. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the Conference on Distributed Computing Systems*, pages 22–30, 1982.
21. C. G. Rommel, D. Towsley, and J. A. Stankovic. Analysis of fork-join jobs using processor-sharing. Technical Report UM-CS-1987-052, University of Massachusetts, 1987.
22. M. R. Leuze, L. W. Dowdy, and K. H. Park. Multiprogramming a distributed-memory multiprocessor. *Concurrency: Practice and Experience*, 1(1):19–33, 1989.
23. S. K. Setia, M. S. Squillante, and S. K. Tripathi. Processor scheduling on multiprogrammed, distributed memory parallel computers. *SIGMETRICS Performance Evaluation Review*, 21(1):158–170, 1993.
24. C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor aladdress policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.
25. K. C. Sevcik. Application scheduling and processor aladdress in multiprogrammed parallel processing systems. *Performance Evaluation*, 19(2-3):107–140, 1994.
26. A. R. Miller. *Nonpreemptive run-time scheduling issues on a multitasked, multiprogrammed multiprocessor with dependencies, bidimensional tasks, folding and dynamic graphs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1987.
27. S. Majumdar, D. L. Eager, and R. B. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of the 1988 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 104–113, Santa Fe, NM, 1988.
28. S. T. Leutenegger and M. K. Vernon. The performance of multiprogrammed multiprocessor scheduling algorithms. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 226–236, Boulder, CO, 1990.
29. K. M. Chandy and P. F. Reynolds. Scheduling partially ordered tasks with probabilistic execution times. In *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 169–177, Austin, TX, 1975.
30. J. Bruno and P. Downey. Probabilistic bounds on the performance of list scheduling. *SIAM Journal of Computing*, 15(2):409–417, 1986.
31. S. Tongsima, C. Chantrapornchai, E. H.-M Sha, and N. L. Passos. Scheduling with confidence for probabilistic data-flow graphs. In *Proceedings of 7th Great Lakes Symposium on VLSI*, pages 150–155, Urbana, IL, 1997.
32. T. K. Som and R. G. Sargent. A probabilistic event scheduling policy for optimistic parallel discrete event simulation. In *Proceedings of 12th Workshop on Parallel and Distributed Simulation*, pages 56–63, Banff, Alberta, Canada, May 1998.
33. A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Proceedings of Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, pages 361–378, San Jose, CA, January 1999.
34. S. Fujita and H. Zhou. Multiprocessor scheduling problem with probabilistic execution costs. In *Proceedings International Symposium on Parallel Architectures, Algorithms and Networks*, pages 121–126, Dallas/Richardson, TX, December 2000.
35. K. Li and Y. Pan. Probabilistic analysis of scheduling precedence constrained parallel tasks on multicomputers with contiguous processor allocation. *IEEE Transactions on Computers*, 49(10):1021–1030, 2000.
36. H. Moulin. Split-proof probabilistic scheduling. In *New Trends in Co-operative Game Theory*, January 2005.
37. H. Özsoy. Coordinated splitting in probabilistic scheduling. In *Public Economic Theory*, Marseille, France, 2005.
38. T. Glatard, , J. Montagnat, and X. Pennec. Probabilistic and dynamic optimization of job partitioning on a grid infrastructure. In *Proceedings of the 14th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Montbilard-Sochaux, France, February 2006.