An Efficient Approach for Self-Scheduling Parallel Loops on Multiprogrammed Parallel Computers

Arun Kejariwal[1] Alexandru Nicolau[1] Constantine D. Polychronopoulos[2]

[1] Center for Embedded Computer Systems
University of California at Irvine
Irvine, CA 92697, USA
arun_kejariwal@computer.org, nicolau@cecs.uci.edu
http://www.cecs.uci.edu/

[2] Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
cdp@csrd.uiuc.edu
http://www.csrd.uiuc.edu/

Abstract. Clusters and grids have increasingly become standard platforms for high performance computing as they provide extremely high execution rates with great cost effectiveness. Such systems are designed to support concurrent execution of multiple jobs. It calls for multiprogrammed scheduling of the different jobs for effective system utilization and for keeping average response times low. Although a significant amount of work has been done in scheduling parallel jobs on multiprocessor systems, the problem of scheduling parallel tasks of an individual job on a multiprogrammed parallel system has not been given enough attention so far. In this paper, we present a dynamic scheduling technique for scheduling iterations of a `DOALL` loop (of a single application) to achieve load balance between a given set of processors. Experimental results show the effectiveness of our approach.

1 Introduction

Although multiprogramming allows to better service multiple users, it also greatly complicates the scheduling process. This can be attributed to the space-time sharing of processors by the different jobs and the trade-off between the different performance metrics. Several techniques have been proposed for job scheduling with different objectives such as minimizing average mean response time, minimizing makespan, minimizing the tardiness [1,2]. Similarly, the impact of other parameters such as knowledge of job service demands, variability of job parallelism, preemption of jobs on performance of scheduling policies has also been investigated [3]. However, from the standpoint of performance of an individual job, the impact of the dynamics of a multiprogrammed system on the scheduling of parallel tasks of a single job has not been given enough attention. One of the critical problems to be addressed in this context is how to efficiently allocate the parallel tasks amongst a given set of processors so as to distribute

the computational load as evenly as possible, in order to minimize the maximum completion time.

In this paper, we address the problem of minimizing the maximum completion time of `DOALL` [4] loops. 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. Several static scheduling schemes have been proposed for the above, however, these do not perform well in a multiprogramming environment. Similarly, several dynamic scheduling schemes have been proposed to perform task allocation on the "on-the-fly" wherein one or more iterations are assigned to a processor whenever it becomes available. However, run-time scheduling overhead becomes a critical factor in the context of dynamic scheduling and can potentially account for a significant portion of the total execution time [5]. Thus, the idea is to avoid the use of the operating system in order to minimize the scheduling overhead, by instrumenting the code corresponding to the parallel loop such that the processors perform scheduling by themselves at run-time. *Self-scheduling* [6] exemplifies this philosophy where task size is determined by the processors themselves rather then by the operating system or a global control unit.

Several self-scheduling techniques have been proposed for scheduling parallel loops [7]. The computation of the task size (or chunk size) at any scheduling step (from hereon, we shall use the term *chunk size*, instead of task size, for literary consistency) in each is based on the number of remaining iterations. It assumes the availability of a "fixed" number of processors. However, the latter is not valid in context of multiprogramming. This can potentially give rise to "gaps" in processor availability, i.e., a processor may not be continuously available to the same job. For example, in Figure 1 processor P_2 is not available for $t \in (750, 850)$. The effect of varying number of processors and the presence of *gaps* on self-scheduling is not well understood. In this paper, we propose a novel scheduling technique, referred to as *Gap-Aware Self-Scheduling* (GAS), to capture the effect of presence of *gaps* in processors availability on self-scheduling. At each scheduling step, GAS computes the chunk size based on the number of remaining iterations and *gaps* in processor availability. We show that gap-aware computation of chunk size helps achieve load balance between the different processors.

The rest of the paper is organized as follows. In the next section, we present the motivation behind this work. In Section 3, we present our approach for dynamic scheduling of parallel loops on multiprogrammed parallel processor systems. Experimental setup and results are presented in Section 4.

2 Motivation

Though it is fairly intuitive that gaps reduce the degree of parallel execution, the impact of gaps on load balance between the processors is not obvious. In

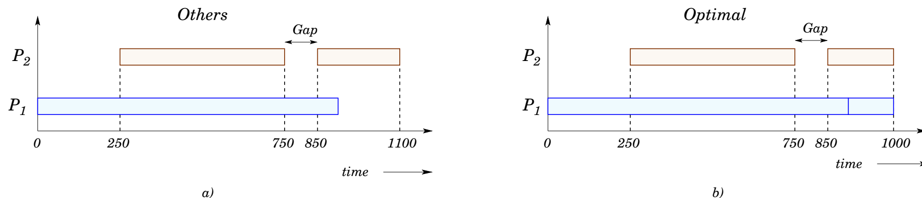


Fig. 1. a) Schedule obtained from the existing techniques; b) Optimal Schedule

this work, we study the latter. From hereon, we shall consider only those gaps which can potentially result in load imbalance. Conditions for the existence of such gaps are discussed in detail in [8]. We argue that it is important to account for such gaps during the scheduling of parallel tasks — iterations of a (nested) parallel loop in our case.

For example, consider the schedules shown in Figure 1, where the length of a block represents the size of a chunk allocated to a processor at a given scheduling step. For simplicity, we assume that each iteration takes a unit amount of time. In Figure 1(a), we note that 250 iterations are allocated to processor P_2 at $t = 850$ regardless of the gap for $t \in (750, 850)$. Clearly, this results in uneven finishing times. On the other hand, the optimal schedule is shown in Figure 1(b) where the 250 iterations are distributed amongst the two processors to yield even finishing times. From above, we learn that it is critical to modulate the chunk size in presence of gaps in order to achieve better load balance.

3 The Approach

In this section we present the algorithm for our approach — *Gap-Aware Self-Scheduling* (GAS). Although several models have been proposed for work queues, viz., global, local and hybrid, in context of self-scheduling, we adopt the model proposed by Polychronopoulos and Kuck in [9] owing to its simplicity. Note that model selection per se is orthogonal to the concerns we address in this paper. The algorithm is designed for non-preemptive scheduling, whereby a chunk, once assigned to a processor may not be removed until it has finished execution. The design of our approach is guided by the following: a) how to capture the effects of gaps in processor availability; b) how to select W_{min}, i.e., the minimum workload per chunk; and c) how to minimize the synchronization overhead between the processors. The rest of the section describes the different phases of our scheduling algorithm.

3.1 Determining the Gap Factor

As illustrated in Section 2, gaps in processor availability play a critical role in load balancing and directly relate to the efficiency of a dynamic scheduling scheme. In order to capture the effects of gaps on the performance of a self-schedule, we define a *displacement factor*, denoted by α, for online modulation of the chunk size. Let t_{last} denote the finishing time of the most recently completed chunk (on any processor) and let t_{first} denote the earliest finishing time of any chunk

under execution (on any processor). At a given time instant t, the *displacement factor* is computed as follows:

$$\alpha(t) = \begin{cases} \frac{t - t_{\text{last}}}{t_{\text{first}} - t_{\text{last}}} & \exists \text{ a gap at time } t \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Intuitively, the *displacement factor* is a measure of the length of a gap w.r.t. the earliest finishing time of all the currently active processors. Arguably, one could potentially use α as the modulation factor. However, from Equation 1 we observe that when $t = t_{\text{last}}$, $\alpha = 0$. Thus, in this case the chunk size is reduced to zero. Consequently, α in itself cannot be used for chunk size modulation. In order to alleviate the problem, we define a *gap factor* as a function of α, denoted by $\beta(\alpha)$, and is computed as follows:

$$\beta(\alpha) = a\alpha^2 + b\alpha + c \tag{2}$$

Let us now revisit Equation 1 to study the behavior of α as $t \to t_{\text{first}}$ (by definition, $t \neq t_{\text{first}}$) as it is required to derive the boundary conditions for β.

$$\lim_{t \to t_{\text{first}}} \frac{t - t_{\text{last}}}{t_{\text{first}} - t_{\text{last}}} = 1 \quad \Rightarrow \alpha(t \to t_{\text{first}}) = 1 \tag{3}$$

From Equation 2, we deduce that when $\alpha = 0$, $\beta = 1$ and when $\alpha \to 1$ (refer to Equation 3), $\beta \to 1$. Note that the above conditions are compliant with the existence conditions of a gap [8] which form the very basis of online chunk size modulation. To summarize,

$$\beta = 1, \text{ when } \alpha = 0 \text{ and } \alpha = 1$$

Further, we assume that $\beta = 0.5$ when $\alpha = 0.5$. Solving for a, b and c using the above conditions yields the following:

$$\beta = 2\alpha(\alpha - 1) + 1 \tag{4}$$

3.2 Determining the Chunk Size

Markatos and LeBlanc showed that load imbalance is the prime factor governing the efficiency of a self-schedule [10]. The extent of load imbalance introduced depends on the amount of workload allocated relative to the amount of remaining workload. At any point in time, the amount of workload assigned to each processor[3] should be chosen such that the remaining workload is "sufficient" to balance the workload evenly, i.e., the difference in finishing times of the processors (at the end of the schedule) is minimal. With the above goal, we now derive the expression for the chunk size, denoted by Λ. In general, at any given time instant t in a self-schedule, the chunk size is defined as a multivariate function f, as given below:

$$\Lambda(t) = f(W_R(t), P, (t_{\text{first}} - t), \beta, W_{\min}) \tag{5}$$

[3] Recall that multiple processors may be available at the same time.

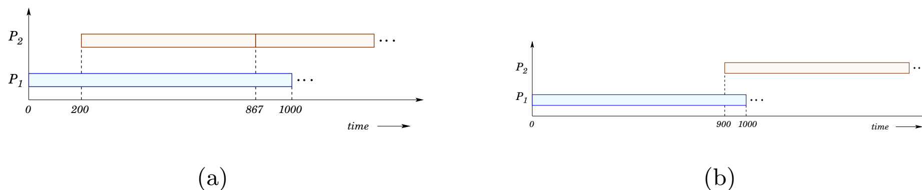
where $W_R(t)$ denotes the number of remaining iterations at time t and $W_{\min}$ denotes the minimum chunk size. In the rest of this subsection, we follow a step-by-step approach to derive the expression for chunk size.

A modified form of Λ (w.r.t. the one proposed in *guided self-scheduling* by Polychronopoulos and Kuck [9]) is given by:[4]

$$\Lambda(t) = \left\lceil \frac{W_R(t)}{1.5P} \right\rceil \tag{6}$$

However, the chunk size as defined above can potentially increase the scheduling overhead as illustrated by the following example.

Example 1 *Consider a (coalesced) parallel loop with 3000 iterations with identical workloads and a system of two processors P_1 and P_2, where P_1 is available at $t = 0$ and P_2 is available at $t = 200$ in case a) and at $t = 900$ in case b). For simplicity of exposition, we assume that there do not exist gaps in processor availability.*



(a) (b)

Fig. 2. Example partial schedules

Consider the partial self-schedule shown in Figure 2(a). From the figure, we observe that at $t = 200$, P_2 is assigned only 667 iterations. This implies that P_2 would finish before P_1 finishes and would result in "early" rescheduling of P_2. Clearly, this incurs additional scheduling overhead without any increase in parallel execution. In order to alleviate the above, we propose to "delay" the rescheduling of P_2 by allocating 800 iterations at time $t = 200$. In such cases, we argue to allocate $(t_{first} - t)$ number of iterations. This minimizes the number of allocation points without loss in parallel execution.

However, as $t \rightarrow t_{first}$ the above strategy may result in allocation of small chunks which adversely affects the performance of a self-schedule [9]. For example, consider the partial schedule shown in Figure 2(b), where P_2 is available at time $t = 900$. In this case, we assign 667 iterations to P_2 instead of 100 $(= t_{first} - t)$ iterations so as to minimize the number of allocations, thereby reducing the scheduling overhead.

Based on the discussion of Example 1 we refine the expression for computation of chunk size (given in Equation 6) to balance the trade-off between

[4] For derivation of Equation 6, the reader is referred to [8].

maximizing parallel execution and minimizing scheduling overhead. For this, we introduce a new parammeter called *lag*, as defined below:

$$lag(t) = \begin{cases} t_{\text{first}} - t & \exists \text{ a gap at time } t \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

The modified expression for chunk size is given as follows:

$$\Lambda(t) = \max\left(lag(t), \left\lceil \frac{W_R(t)}{1.5P} \right\rceil\right) \tag{8}$$

Equation 8 implicitly assumes that for all t in a self-schedule $t_{\text{first}} - t < W_R$. It is easy to see that the same is valid for $P \geq 2$ as at any scheduling step less than half of the remaining number of iterations are allocated. Next, we incorporate the effect of existence of gaps in processor availability in the expression for chunk size.

$$\Lambda(t) = \max\left(lag(t), \left\lceil \frac{\beta\ W_R(t)}{1.5P} \right\rceil\right) \tag{9}$$

The exponential decrease of chunk size during self-scheduling results in scheduling of individual iterations towards the end of the schedule. The latter incurs high scheduling overhead. In order to alleviate this the chunk size is restricted to a pre-defined quantum, denoted by $W_{\min}$ (for further details the reader is referred to [9]). We further refine the expression of chunk size to capture $W_{\min}$ and is given as follows:

$$\boxed{\Lambda(t) = \max\left(W_{\min}, \max\left(lag(t), \left\lceil \frac{\beta\ W_R(t)}{1.5P} \right\rceil\right)\right)} \tag{10}$$

The parameter $W_{\min}$ is application and input data dependent. The selection of an appropriate value for $W_{\min}$ is critical for the existing self-scheduling schemes. While a small value of $W_{\min}$ may result in scheduling of individual iterations (irrespective of their workload) at the end which may incur significant synchronization overhead, whereas a large value of $W_{\min}$ may lead to load imbalance. A formal description of the algorithm for GAS is presented as Algorithm 1.

The discussion in this subsection so far has been based on the assumption that iterations have equal workloads (or execution times). However, the workload of individual iterations may differ from each other when there are conditional statements in the loop; even otherwise, their workloads may differ due to system variations such as data access latency, network interference and operating system. Even in such cases, our gap-driven chunk size modulation approach is still applicable. A detailed discussion of this beyond the scope of this paper.

Algorithm 1 Gap-Aware Self-Scheduling

Input : A N-dimensional iteration space Γ and P processors. Note that at any given time instant, all the processors may be available.

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

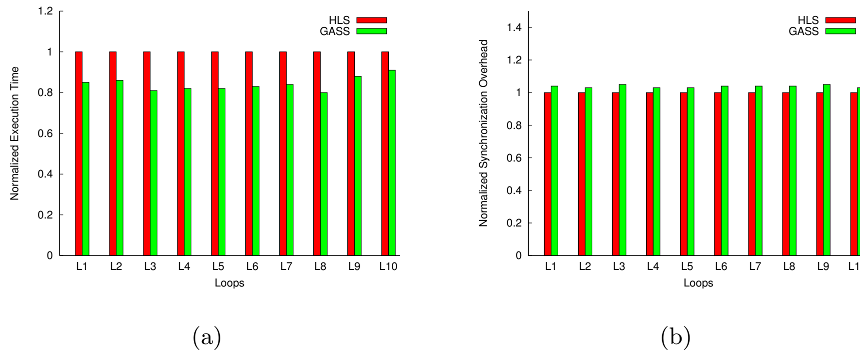
repeat
  /* Self-schedule the remaining iterations at time t */
  $p_f \leftarrow 0$
  Compute $lag(t)$ using Equation 7
  Compute the gap factor $\beta(t)$ using Equation 4
  Compute the chunk size $\Lambda(t)$ using Equation 10
  for each available processor **do**
    Compute index range for each processor
    Assign the corresponding iterations to the processor
    $p_f \leftarrow p_f + 1$
  end for

  /* Update the remaining workload */
  $\mathbf{W} \leftarrow \mathbf{W} - \Lambda(t) \times p_f$
until $\mathbf{W} > 0$

4 Experiments

We implemented a simulator to compare the performance of GAS with the "upper algorithm" of the *adaptive self-tuning scheduling* scheme [11] (referred to as HLS in the rest of the paper). For our experiments, we extracted kernels (parallel nested loops) from LAMMPS [12] (a classical molecular dynamics code designed to simulate systems at the atomic and molecular level) and DAKOTA [13] (a design analysis kit for optimization and terascale applications). The execution time of the loops was determined via profiling. A random generator was used for dynamic processor allocation; random numbers are generated using a uniform distribution. The simulator supports uneven start times of the processors. Further, it also accounts for the synchronization overhead. Processors are assumed to access the shared variables, in our case loop indexes, using appropriate synchronization primitives. A maximum of 2000 processors was assumed as commonly found in clusters and grids. Recall that at any given scheduling step, the number of processors available is *not* fixed. The loops were dynamically scheduled using Algorithm 1.

Figure 3(a) presents a performance comparison between HLS and GAS. The execution times were computed as an average of execution times of 10 simulation runs with different processor availability configurations. From the figure, we note that our approach achieves a speedup of 10–15%. As explained earlier, the speed



(a) (b)

Fig. 3. *a) Performance comparison; b) Synchronization overhead*

up can be attributed to the better load balance between the different processors which facilitates higher degree of parallel execution. In addition, it enables better processor utilization.

From Figure 3(b) we observe that the GAS incurs 3% (on an average) synchronization overhead. It can be attributed to the overhead incurred in online update of the expected workload of the remaining iterations. However, the performance gain obtained by online chunk size modulation outweighs the synchronization overhead.

References

1. R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of scheduling*. Addison-Wesley, Reading, MA, 1967.
2. D. G. Feitelson. A survey of scheduling in multiprogrammed parallel systems. Technical Report RC 19790(87657), IBM T. J. Watson Research Center, February 1995.
3. 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.
4. S. Lundstrom and G. Barnes. A controllable MIMD architectures. In *Proceedings of the 1980 International Conference on Parallel Processing*, St. Charles, IL, August 1980.
5. C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, 1985.
6. B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Proceedings of SPIE - Real-Time Signal Processing IV*, pages 241–248, 1981.
7. A. Kejariwal and A. Nicolau. Reading list of self-scheduling of parallel loops. http://www.ics.uci.edu/~akejariw/SelfScheduleReadingList.pdf.
8. A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos. Accounting for "Gaps" in processor availability during self-scheduling of parallel loops on multiprogrammed parallel computers. Technical Report TR-05-14, School of Information and Computer Science, University of California at Irvine, October 2005.
9. 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.
10. E. Markatos and T. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.
11. 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.
12. LAMMPS. http://www.cs.sandia.gov/~sjplimp/lammps.html.
13. DAKOTA. http://endo.sandia.gov/DAKOTA/software.html.