History-aware Self-Scheduling

Arun Kejariwal† Alexandru Nicolau† Constantine D. Polychronopoulos§

†Center for Embedded Computer Systems
School of Information and Computer Science
University of California at Irvine
arun_kejariwal@computer.org
nicolau@cecs.uci.edu

§Center of Supercomputing Research and Development
Computer and Systems Research Laboratory
University of Illinois at Urbana-Champaign
cdp@csrd.uiuc.edu

Abstract

Scheduling parallel loops, i.e., the way iterations are mapped on to different processors, plays a critical role in the efficient execution of programs, particularly of supercomputing applications, on multiprocessor systems. In applications where the problem dimension (and hence execution time) is dependent on run-time data, loop iterations also tend to be of variable length – this variability affects both sequential and parallel loops and in particular nested loops and it is quite prevalent in sparse matrix solvers. In this paper, we propose a (execution) history-aware self-scheduling approach of irregular parallel loops on heterogeneous multiprocessor systems. First, the proposed method computes the chunk size, i.e., the amount of work allocated to a processor at each scheduling step, based on the variance in workload distribution across the iteration space. Second, it fine tunes the chunk size based on the execution history of the loop, wherein the workload of an iteration is determined at run-time based on the statistical deviation of workload estimates of previously executed iterations from their corresponding actual workloads. We evaluate our techniques using a set of kernels (extracted from industry-strength SPEC OMPM 2001 benchmark) with uneven workload distributions. The results show that our technique performs 5% – 18% better than the existing schemes.

1 Introduction

Multiprocessor systems are (ubiquitous) platforms of choice for the execution of supercomputing applications used in various scientific and engineering fields (e.g., airfoil design, car crash simulation or the formation and interaction of various proteins). In such systems several parallel modules or tasks of the same application can be executed concurrently. One of the critical problems to be addressed in this context is how to efficiently allocate the parallel tasks amongst the given processors so as to distribute the computational load as evenly as possible, in order to minimize the maximum finishing time [1]. The approaches that have been proposed so far can be broadly classified into two categories: i) Processor allocation, where one or more idle processors are allocated to a task and idle processors are reserved until enough processors become available to satisfy the allotment for that task as determined by a scheduling algorithm. ii) Task allocation, where one or more tasks are allocated to a processor as soon as it becomes idle. The latter approach aims at keeping processors as busy as possible and avoids the deliberate idling of processors, thereby minimizing the maximum finishing time. In this paper, we address the problem of minimizing the maximum finishing time of `DOALL` [2] loops in which the execution time (or workload) of an iteration is known only at run-time. For this class of loops we adopt the second approach; in our case, a task comprises of one or more iterations of a `DOALL` loop. 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.

Static or deterministic scheduling schemes perform task allocation at compile-time; thus, in this case, the task size is "fixed" prior to program execution. Although static scheduling yields optimal schedules when the iterations have equal workload, it does not perform well when the workload of each iteration is different and is not known until run-time.[1] In order to alleviate this problem, dynamic or non-deterministic scheduling schemes perform task allocation "on-the-fly", i.e., one or more iterations are assigned to a processor whenever it becomes available; in this case, the task size is determined dynamically. However, run-time scheduling overhead (incurred due to system calls to the operat-

[1]The workload of individual iterations may differ from each other when there are conditional statements in the loop. The heterogeneity of the multiprocessor system can further introduce variation in iteration workloads. Even otherwise, their workloads may differ due to system variations such as data access latency, operating system and network interference et cetera.

ing system) becomes a critical factor in the context of dynamic scheduling as it can potentially account for a significant portion of the total execution time [3]. 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* [4] exemplifies this philosophy where task size is determined by the processors themselves rather then by the operating system or a global control unit.

In this paper, we propose a novel approach, referred to as *History-aware Self-Scheduling* (HSS), for dynamic scheduling of irregular parallel loops on heterogeneous multiprocessor systems. In such loops the workload of different iterations vary significantly. Such loops are commonly found in applications such as climate modeling [5], N-body simulations [6], Monte-Carlo method [7], adaptive mesh refinement [8], elastic wave propagation [9], ray tracing, circuit simulation [10], x-ray tomography et cetera. At every scheduling step, HSS computes the amount of workload to be allotted to an idle processor based on the amount of remaining workload and processor speed. Then, for each idle processor, it determines a set of iterations with the above workload. The number of iterations per set is dependent on the workload distribution of the remaining iterations. The key characteristic of our scheme is the dynamic adaptation of the chunk size based on the statistical deviation of the workload estimates of the previously executed iterations from their corresponding actual workloads. In addition, our approach minimizes the synchronization overhead (incurred during self-scheduling as processors use hardware synchronization primitives, such as *fetch & add* [11], to access the shared loop indices) by minimizing the number of synchronization points.

The rest of the paper is organized as follows. In the next section, we introduce the terminology used in the rest of the paper. A motivating example is presented in Section 3. Section 4 discusses our history-aware approach for dynamic scheduling of parallel loops on parallel processor systems. Experimental setup and results are presented in Section 5. Related work is discussed in Section 6. Finally we conclude with directions for future work.

2 Terminology

Our loop model consists of a non-perfectly nested `DOALL` loop [2] with constant loop bounds. Further, our model also supports nested conditionals at each level of the nested loop. The **index variables** of the individual loops are $i_1, i_2, \ldots, i_n$ and they constitute an **index vector** $\mathbf{i} = \langle i_1, i_2, \ldots, i_n \rangle$. An **iteration** is an instance of the index vector $\mathbf{i}$. The set of iterations of a loop nest $\mathbf{L}$ is an **iteration space** $\Gamma = \{\mathbf{i}\}$. Let $\mathbf{N}$ denote the total number of iterations in Γ. Assuming normalized indices, $\mathbf{N}$ is given by:

$$\mathbf{N} = \prod_{k=1}^{n} N_k$$

where, N_k is the upper bound of index variable i_k. An iteration space is said to have *uniform* workload distribution if all the iterations in Γ have equal execution times (or workloads). However, in the presence of conditionals the iterations tend to have different workloads. Such an iteration space is said to have *non-uniform* workload distribution.

Let $\mathcal{W}(\mathbf{i})$ denote the expected workload of an iteration $\mathbf{i}$ and $\mathbf{W}$ denote the total workload. Next we introduce some definitions of some other terms used in the rest of the paper.

Definition 1. *The* mean μ *of a workload distribution of an iteration space is defined as:*

$$\mu = \frac{\sum_{\mathbf{i}\in\Gamma} \mathcal{W}(\mathbf{i})}{\mathbf{N}} \tag{1}$$

Definition 2. *The* variance σ^2 *of a workload distribution of an iteration space is defined as:*

$$\sigma^2 = \frac{\sum_{\mathbf{i}\in\Gamma} (\mathcal{W}(\mathbf{i}) - \mu)^2}{\mathbf{N}} \tag{2}$$

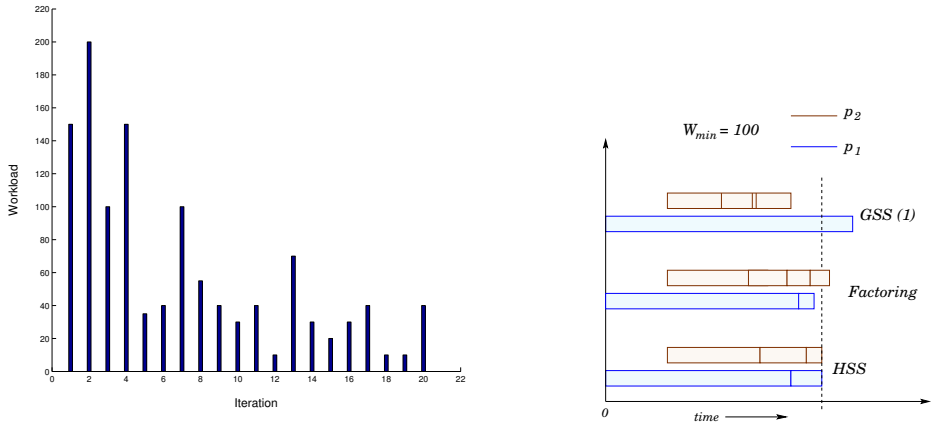
Given a set of m processors $P = \{p_1, \ldots, p_m\}$, let s_i denote the speed of a processor $p_i \in P$.

3 A Motivating Example

In this section we illustrate the intuitive idea behind our approach (HSS) with the help of an example. For comparison purposes, we consider two well known self-scheduling techniques, viz., *guided self-scheduling* (GSS) [12] and *factoring* [13]. At a given scheduling step, assuming identical processors, GSS assigns $\frac{1}{m}$ of the remaining iterations to an idle processor; factoring assigns iterations to the processors in batches of m chunks, where the batch size is half the number of remaining iterations (a detailed description of the two techniques is presented in Section 6). Consider the workload distribution shown in Figure 1(a). Note that the first ten iterations constitute around 75% of the total workload. Given 2 processors, the chunk sizes for GSS(1), Factoring and HSS are shown in Table 1. The corresponding schedules are shown in Figure 1(b), assuming that processor p_2 is available for time $t \geq 200$.

Scheme	Chunk Sizes	# of Sync. Points
GSS(1)	10 5 3 1 1	5
Factoring	5 5 3 3 2 2	6
HSS	4 6 4 4 2	5

Table 1. Total workload $\mathbf{W} = 1200$ units, $m = 2$



(a) (b)

Figure 1. a) A non-uniform workload distribution; b) Schedules corresponding to different schemes, assuming two processors. ($W_{\min}$ is the the minimum workload of any chunk)

GSS(1) allocates 10 iterations or 900 units of work to processor p_1 at $t = 0$ and the remaining iterations to processor p_2; from Figure 1(b) we observe that it results in heavy load imbalance between the two processors. Factoring achieves better load balance than GSS(1) by limiting the size of the early chunks; however, it incurs additional synchronization overhead due to higher number of synchronization points. Both GSS and Factoring, as well as other existing self-scheduling techniques, compute the chunk sizes independent of the variance in workload distribution across the iteration space. As a result, while some of processors are assigned large amount of workload, the others starve. In addition, chunks with a small amount of workload necessitate frequent scheduling of the remaining iterations which in turn increases the synchronization overhead.

In contrast, HSS computes chunk sizes based on the variance in workload distribution. At each scheduling step, the chunk size is calculated as a function of an estimate of remaining workload, not the number of remaining iterations. The relationship between the chunk size and workload distribution helps minimize load imbalance. For example, the first chunk under HSS consists of only 4 iterations (refer to Table 1) due to heavy workload in the region corresponding to these iterations. In contrast, the second chunk consists of 6 iterations as these iterations have relatively less workload. From Figure 1(b) we observe that restricting the chunk sizes to be progressively decreasing, as proposed by the existing techniques, can potentially result in sub-optimal schedules.

4 The Approach

In this section we present the algorithm for our approach - *History-aware 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 [12] 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 task, once assigned to a processor may not be removed until it has finished execution. The design of our approach is guided by the following factors: a) Non-uniform workload distribution; b) Synchronization overhead between the processors; and c) Selection of $W_{\min}$, i.e., the minimum workload per task. The rest of the section describes the different phases of our scheduling algorithm.

4.1 Determining the workload distribution

First, we determine the expected workload of each iteration on a processor p_i with $s_i = 1$ via loop profiling (done offline) [14]. Note that an iteration may have different execution times for different data sets as it may take a different path in the control flow graph with change in the input data. We employ an estimation-based approach wherein we profile the loop with multiple training sets. Then, we determine the execution probability of each basic block in each iteration. The expected execution time or workload of an iteration is the sum of the execution times of all the basic blocks weighted with their respective execution probabilities for that iteration.

4.2 Chunk Size

Markatos et al. showed that the different availability times of the processors does not affect the performance significantly [15]. Instead, load imbalance is the prime factor governing the efficiency of a schedule. The extent

Procedure 1 Determining the workload distribution

Input : A N-dimensional iteration space Γ and a processor with $s_i = 1$

Output : Workload distribution of Γ

Let $B = \{b_1, b_2, \ldots\}$ be the set of basic blocks

Determine length $\ell(b)$ of each basic block $b \in B$

Profile the loop corresponding to Γ with the given training sets

/* Compute the execution probability of basic blocks */

for all i $\in \Gamma$ **do**
  for all $b \in B$ **do**

$$p(b) = \frac{\text{Number of times } b \text{ is executed}}{\text{Number of training sets}}$$

  end for
end for
for all i $\in \Gamma$ **do**

/* Compute the expected workload $\mathcal{W}(\mathbf{i})$ of **i** */

$$\mathcal{W}(\mathbf{i}) = \sum_{b \in B} p(b) \times \ell(b)$$

end for

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 [2] 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. Let $\mathbf{s} = \sum_{1 \le i \le m} s_i$. The chunk size corresponding to a processor p_i is given by the following expression:

$$\Lambda(p_i) = \max\left(W_{\min}, f\left(W_R, s_i, \mathbf{s}, \right)\right) \tag{3}$$

where W_R represents the remaining workload at a given scheduling step. Akin to guided self-scheduling, the function f in Equation (3) is given by:

$$f = \left\lceil \frac{s_i \; W_R}{\mathbf{s}} \right\rceil$$

However, as discussed in [13], the above may result in allocation of too much work to early chunks; specifically, two-thirds of the work is assigned to first m chunks in case of identical processors. It has been shown that 50% of the total workload is sufficient to even out the finishing times of the processors [13]. Therefore, we introduce a correction factor to "relax" the exponential decay of chunk size. Assuming identical processors, the amount of workload remaining after m allocations can be approximated as $(1 - \frac{1}{\eta m})^m \mathbf{W}$, where η is the correction factor and m is the number of processors. From the above, η must satisfy the following:

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

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

$$f = \left\lceil \frac{s_i \; W_R}{1.5 \; \mathbf{s}} \right\rceil \tag{4}$$

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. HSS minimizes the sensitivity of the schedule to $W_{\min}$ as it avoids scheduling of individual iterations with small workloads.

So far, we computed the workload of the different iterations based on loop profiling. However, estimation-based techniques may not perform well in case of highly irregular workload distributions, i.e., when the workload of an iteration is very sensitive to the input data. In such cases, an estimate of workload of an iteration can potentially have large deviation from its actual value (known only after execution of the iteration). In order to minimize the error, we propose dynamic update of the estimates to better capture the irregular nature of the application. Let $\mathcal{W}_a(\mathbf{i})$ denote the actual workload of an iteration **i** on a processor p_i with $s_i = 1$. The actual workload of an iteration **i** executed on a processor p_i with $s_i > 1$ is weighted with s_i to obtain $\mathcal{W}_a(\mathbf{i})$. The error, denoted by $e(\mathbf{i})$, in the workload estimate is given by:

$$e(\mathbf{i}) = \mathcal{W}_a(\mathbf{i}) - \mathcal{W}(\mathbf{i})$$

The mean value of the error for all $\mathbf{j} < \mathbf{i}$ [16] is defined as:

$$\mu_e = \frac{\sum \lambda_{\mathbf{j}} \; e(\mathbf{j})}{\sum \lambda_{\mathbf{j}}}$$

Similarly, for $\mathbf{j} < \mathbf{i}$, the variance in error is defined as:

$$\sigma_e^2 = \frac{\sum \lambda_{\mathbf{j}} (e(\mathbf{j}) - \mu_e)^2}{\sum \lambda_{\mathbf{j}}}$$

The workload of an iteration **i** is updated as follows [17]:

[2] Multiple processors may be available at the same time.

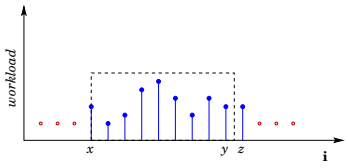


Figure 2. Best-fit index range selection for a given Λ

$$\mathcal{W}(\mathbf{i}) + \mu_e + \sigma_e \sqrt{\frac{\mathbf{n}}{2}}$$

where, $\mathbf{n}$ is the number of iterations corresponding to $\mathbf{j} < \mathbf{i}$ and is referred to as the *history window*. The coefficients (λs) are monotonically increasing in their lexicographic order [18]. Intuitively, we update the workload of an iteration $\mathbf{i}$ based on the deviation of the workload estimates of the previously executed iterations from their actual workloads. The λs are constrained to be in increasing order so as to better capture the sensitivity of the recently executed iterations to input data. The width of the history window is parameterized. A large window width increases the accuracy of the update process, however, it incurs more run-time scheduling overhead. Naturally, the dynamic update of the workload estimates has a trade-off between load balancing and scheduling overhead. However, it has been shown that run-time performance measurement via use of hardware performance counters incurs minimal scheduling overhead [19]. In Section 5.1 we show that better load balance (achieved via dynamic update of iteration workloads) outweighs the associated scheduling overhead.

4.3 Best-fit Approximation

Note that there may not exist any index range constituting Λ amount of work.[3] In such cases, we select the "best-fitting" range to aid load balancing. For example, consider the example shown in Figure 2 (only a part of the workload distribution is shown). Let the current index be $\mathbf{i} = x$. The dashed box signifies that $\mathcal{W}[x, y] < \Lambda < \mathcal{W}[x, z]$, where $\mathcal{W}[a, b]$ denotes the total workload in the range $a \leq \mathbf{i} \leq b$. In such a case, Algorithm 1 selects the range $[x, y]$ if $|\Lambda - \mathcal{W}[x, y]| < |\mathcal{W}[x, z] - \Lambda|$, else it selects the range $[x, z]$. In contrast, the existing techniques select $[x, z]$ irrespective of the deviation which results in significant load imbalance in case of irregular programs.

4.4 The Algorithm

In this section we present a formal description of the algorithm for HSS. First, Algorithm 1 determines the workload distribution across the iteration space using Procedure 1. Next, it computes the total workload.

[3] Recall that an iteration space is partitioned into tasks at iteration boundaries.

Algorithm 1 History-aware Self-Scheduling

Input : An N-dimensional rectangular iteration space Γ and m processors.

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

Determine the workload distribution using Procedure 1

/* Compute the total workload */

$\mathbf{W} = \sum_{\mathbf{i} \in \Gamma} \mathcal{W}(\mathbf{i})$

/* Generate the schedule (assuming implicit loop * * coalescing [20]) */

Let $P_{\text{idle}} \subseteq P$ be a set of idle processors at any given time instant

repeat

if $|P_{\text{idle}}| \; ! = 0$ **then**

for all $p_i \in P_{\text{idle}}$ **do**

/* Compute the chunk size */

$$\Lambda(p_i) = \max\left(W_{\min}, \left\lceil \frac{s_i \; \mathbf{W}}{1.5 \; \mathbf{s}} \right\rceil\right) \quad (5)$$

Compute index range for each processor using best-fit approximation

Allocate the iterations corresponding to index range to p_i

end for

end if

$\mathbf{W} \leftarrow \mathbf{W} - \sum_{p_i \in P_{\text{idle}}} \Lambda(p_i)$

Update the workload of the remaining iterations

until $\mathbf{W} > 0$

where, $W_{\min}$ is the minimum workload size (prespecified by the user).

Subsequently, at each scheduling step, it allocates $\Lambda(p_i)$ (given by Equation (5)) amount of work to an idle processor p_i. Next, it determines the range of the iterations to be mapped to each processor based on best-fit approximation (refer to Section 4.3) and maps the corresponding iterations onto processor p_i. It then updates the workload of the remaining iterations based on the loop execution history (refer to Section 4.2).

To summarize, our approach provides a simple and practical solution of dynamic scheduling of parallel nested loops on heterogeneous multiprocessor systems. It has dual benefits: achieving near-optimal load bal-

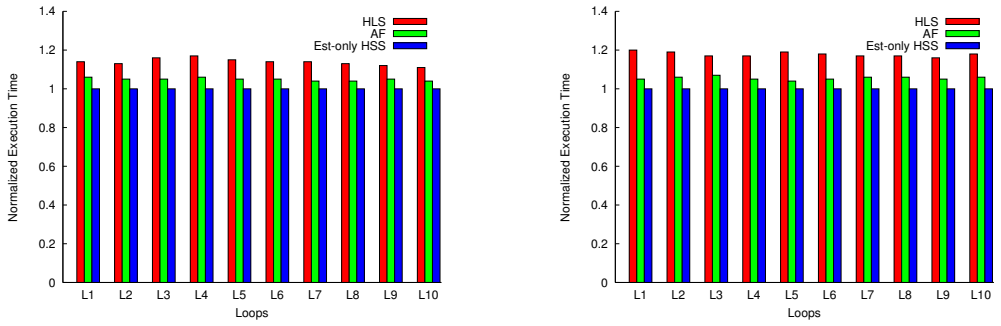


Figure 3. a) Case I; b) Case II. Performance comparison of Est-only HSS with HLS and AF

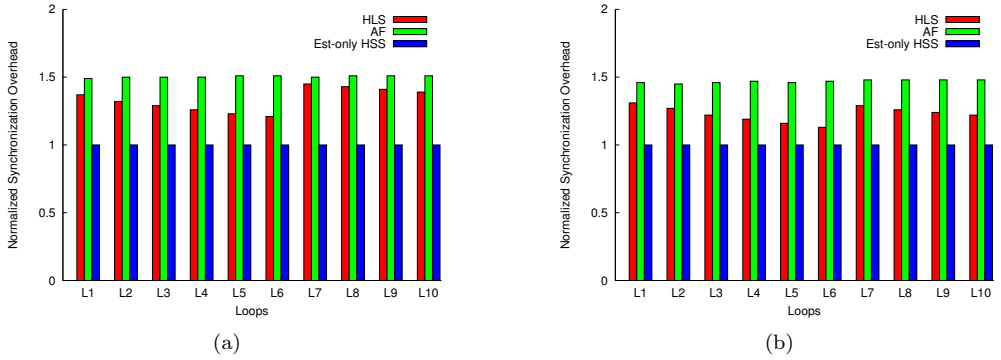


Figure 4. a) Case I; b) Case II. Comparison of synchronization overhead of Est-only HSS, HLS, AF

ance and minimizing the number of synchronization points. Like guided self-scheduling, HSS is also insensitive to uneven start times of the processors. Furthermore, HSS is also applicable in context of multi-way loops.[4] However, a detailed discussion of this is beyond the scope of this paper.

5 Experiments

We implemented a simulator [21] to compare the performance of HSS with *adaptive self-tuning scheduling* [22] (referred to as HLS in the rest of the paper) and *adaptive factoring* (AF) [23]. For consistency purposes (w.r.t. the task granularity), we only consider the "upper algorithm" of HLS which does scheduling is done 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. AF employs a probabilistic model to compute the chunk size. First, we evaluate our approach in absence of (execution) history, referred to as *Est-only HSS*. Subsequently, we evaluate our approach in presence of (execution) history.

[4] A loop is *multi-way nested* if there are two or more loops at the same level [20].

5.1 Results

We conducted experiments to evaluate and compare our approach with the existing techniques for different number of processors. Due to space considerations, we present results for the following two processor configurations: I) $m = 1000$ and II) $m = 2000$. However, we obtain similar speedups for other processor configurations, i.e., for fewer and larger number of processors. In other words, the applicability of our approach is not restricted to any particular processor configuration. We conducted experiments for the two cases for a set of 10 computation-intensive loops $(L_1, L_2, \ldots, L_{10})$ (with non-uniform workload distribution) extracted from the industry-strength SPEC OMPM 2001 benchmark [24]. The kernels differ in both, the workload distribution and the total number of iterations $\mathbf{N}$. The actual workload $(\mathcal{W}_a)$ of the iterations of these kernels correspond to run using the reference data set. The start times of the processors were chosen arbitrarily; however, it does not affect the results significantly as shown by Markatos and LeBlanc in [15]. The dynamic processor availability was simulated via the aforementioned random number generator. The reduction in synchronization overhead achieved (presented later in this section) using our approach corresponds to the reduction in the number of synchronization points.

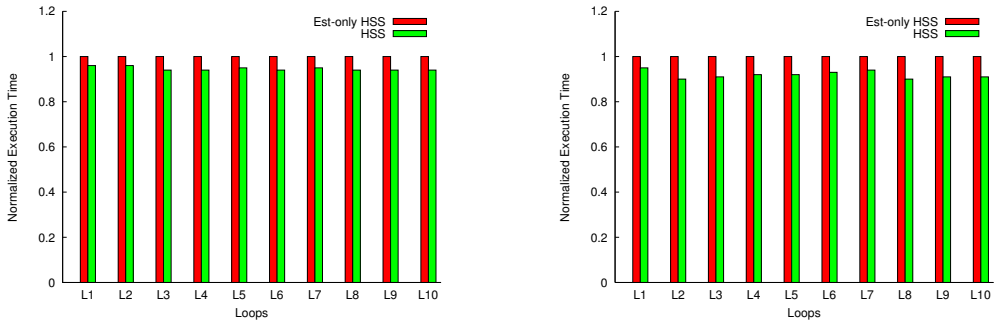


Figure 5. a) Case I; b) Case II. Performance comparison of Est-only HSS and HSS

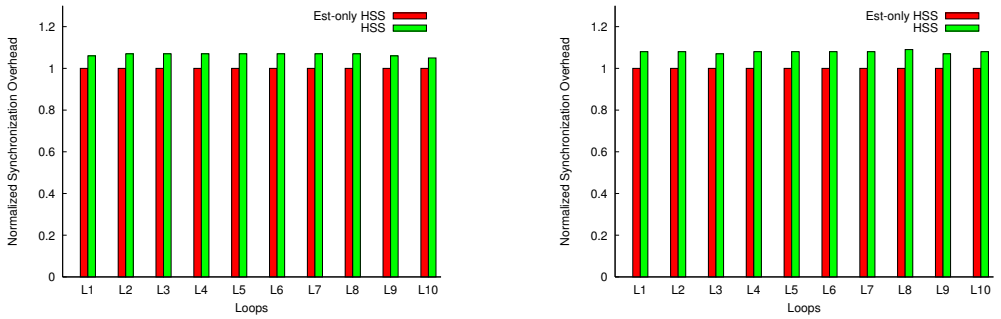


Figure 6. a) Case I; b) Case II. Comparison of synchronization overhead in Est-only HSS and HSS

Figures 3(a) and 3(b) present a performance comparison of Est-only HSS with HLS and AF for case I and case II respectively (the execution times have been normalized w.r.t. Est-only HSS). Given a loop, the workload of the each iteration was determined using the training data set. Note that the workload distribution across the iteration space of the same loop may vary from one instance to another. This is due to the fact that different instances of the same loop have different input data sets. In order to minimize the effect of uneven start times of the processors, the execution times were computed as an average of execution times of 10 simulation runs. We observe that Est-only HSS performs 15% better (on an average) than HLS in case I and 18% in case II. Note that the performance gain increases with increase in m. This can be attributed to the cumulative effect of better load balance across a larger number of processors. Est-only HSS performs 5% better (on an average) than AF in both cases. The performance gain, in general, can be attributed to the dynamic adaptation of chunk size by HSS in accordance with the workload distribution. Figures 4(a) and 4(b) show the synchronization overhead (normalized w.r.t. Est-only HSS) incurred by the three scheduling schemes for case I and II respectively. We observe that both HLS and AF incur a high synchronization overhead as compared to Est-only HSS. Note that for the purposes of worst-case analysis, we assumed a very conservative estimate of the synchronization overhead (per task assignment). However, in practice it can be potentially much larger. In such cases, HSS will outperform the existing techniques by much larger margins as HSS incurs far less synchronization overhead than other techniques, as evidenced by Figure 4.

As discussed in Section 4, scheduling iteration spaces with highly unpredictable and uneven workload distributions using an estimation-based approach may not yield high performance. In order to alleviate the above problem, we employ a (execution) history-aware scheme. Figures 5(a) and 5(b) present a performance comparison of HSS with Est-only HSS for case I and II respectively (the execution times have been normalized w.r.t. Est-only HSS). We note that HSS performs 5% better on an average than Est-only HSS in case I and 9% better in case II. As explained earlier, the higher performance gain in case II can be attributed to better load balance across a larger number of processors. Figures 6(a) and 6(b) compare the synchronization overhead (normalized w.r.t. Est-only HSS) incurred by the two Est-only HSS and HSS. We observe that the latter incurs 7% (on an average) additional overhead. This can be attributed to the overhead incurred in dynamic update of the expected workload of the remaining iterations.

Thus, HSS provides a simple yet powerful approach for dynamic scheduling of parallel loops on multiprogrammed parallel systems.

6 Related Work

There is a large body of work in the area of self-scheduling. Due to space limitations, we shall only discuss the early work. The reader is referred to [25] for an extensive overview of the related work. Self-scheduling of parallel processors, where successive iterations are allocated and executed on to different processors one by one, was first used on Denelcor HEP multiprocessors [26]. Kruskal and Weiss proposed *static chunking* of iterations during the scheduling process [3]. They model the execution times of the iterations as independent identically distributed (i.i.d.) random variables and possessing a moment generating function. However, their model is restricted to IFR distributions [27] such as uniform, normal and exponential. For the class of distributions mentioned above, they showed that the expected completion time is given by $N\mu/p + \sigma\sqrt{2N \ln p/p}$, for $N \gg p \log p$. Although static chunking reduces the synchronization overhead, it has a greater potential for load imbalance than self-scheduling as processors finish within K iterations of each other in the worst case, where K is the chunk size. Arguably, one can randomly assign chunks of iterations to the processors; however, Lucco showed that the random assignment is more efficient than dynamic methods only when $\sigma \ll \mu$, i.e., for a uniform workload distribution or if the scheduling overhead is much greater than μ. In [28], Tang and Yew proposed a scheme for self-scheduling of multiple nested parallel loops. Fang et al. proposed an approach for self-scheduling general parallel nested loops in [29].

7 Conclusion

In this paper we presented an algorithm for dynamic scheduling of nested parallel loops on heterogeneous multiprocessor systems. The key characteristic of our approach is the dynamic adaptation of the chunk size based on the statistical deviation of the workload estimates of the previously executed iterations from their corresponding actual workloads. As future work, we would like to extend our approach to dynamic scheduling of affine loops.

References

[1] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of scheduling*. Addison-Wesley, Reading, MA, 1967.

[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] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, 1985.

[4] 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.

[5] Climate modeling groups. http://stommel.tamu.edu/~baum/climate_modeling.html.

[6] J. Barnes and P. Hut. A hierarchical $o(n \log n)$ force calculation algorithm. *Nature*, 324:446–449, 1986.

[7] S. Ulam and N. Metropolis. The Monte Carlo method. *Journal of the American Statistical Association*, 44:335–341, 1949.

[8] O. Hanson and A. Mayer. Heuristic search as evidential reasoning. In *Proceedings of the Fifth Workshop on Uncertainty in AI*, August 1989.

[9] V. Pereyra, E. Richardson, and S. E. Zarantonello. Large scale calculations of 3d elastic wave propagation in a complex geology. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 301–309, Minneapolis, Minnesota, 1992.

[10] B. Ackland, S. Lucco, T. London, and E. DeBenedictis. CEMU - A parallel circuit simulator. In *Proceedings of the International Conference on Computer Design*, October 1986.

[11] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large number of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, 1983.

[12] 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.

[13] 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.

[14] V. Sarkar. Determining average program execution times and their variance. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 298–312, July 1989.

[15] 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.

[16] U. Banerjee. *Loop Transformation for Restructuring Compilers*. Kluwer Academic Publishers, Boston, MA, 1993.

[17] E. J. Gumbel. The maxima of the mean of the largest value of the range. *The Annals of Mathematical Statistics*, 25(1):76–84, March 1954.

[18] D. Knuth. *The Art of Computer Programming: Fundamental Algorithms (Vol 1)*. Addison-Wesley, 1973.

[19] A. B. Downey and D. G. Feitelson. The elusive goal of workload characterization. *SIGMETRICS Performance Evaluation Review*, 26(4):14–29, 1999.

[20] 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.

[21] 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.

[22] 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.

[23] I. Banicescu and V. Velusamy. Balancing highly irregular computations with the adaptive factoring. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 87–98, 2002.

[24] SPEC OMP. http://www.spec.org/omp.

[25] 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.

[26] E. L. Lusk and R. A. Overbeek. Implementation of monitors with macros: A programming aid for the HEP and other parallel processors. TR ANL-83-97, Argonne National Laboratory, December 1983.

[27] R. E. Barlow and F. Proschan. *Statistical Theory of Reliability and Life Testing*. Holt Rinehart & Winston Inc., 1975.

[28] P. Tang and P. C. Yew. Processor self-scheduling for multiple nested parallel loops. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 528–535, August 1986.

[29] Z. Fang, P. Tang, P.-C. Yew, and C.-Q. Zhu. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Transactions on Computers*, 39(7):919–929, 1990.