A Compilation and Run-Time Framework for Maximizing Performance of Self-scheduling Algorithms

Yizhuo Wang, Laleh Aghababaie Beni, Alexandru Nicolau, Alexander V. Veidenbaum, Rosario Cammarota

A Compilation and Run-Time Framework for Maximizing Performance of Self-Scheduling Algorithms *

Yizhuo Wang[1], Laleh Aghababaie Beni[2], Alexandru Nicolau[2], Alexander V. Veidenbaum[2], and Rosario Cammarota[3]

[1] Beijing Institute of Technology, Beijing 100081, P.R.China
frankwyz@bit.edu.cn
[2] University of California, Irvine CA 92697, USA
[3] Qualcomm Research, San Diego CA 92121, USA

Abstract. Ordinary programs contain many parallel loops which account for a significant portion of these programs' completion time. The parallel executions of such loops can significantly speedup performance of modern multi-core systems. We propose a new framework - Locality Aware Self-scheduling (LASS) - for scheduling parallel loops to multi-core systems and boost up performance of known self-scheduling algorithms in diverse execution conditions. LASS enforces data locality, by forcing the execution of consecutive chunks of iterations to the same core, and favours load balancing with the introduction of a work-stealing mechanism. LASS is evaluated on a set of kernels on a multi-core system with 16 cores. Two execution scenarios are considered. In the first scenario our application runs alone on top of the operating system. In the second scenario our application runs in conjunction with an interfering parallel job. The average speedup achieved by LASS for first execution scenario is 11% and for the second one is 31%.

Keywords: loop scheduling, self-scheduling, random forest.

1 Introduction

Multi-core, multi-socket systems offer a great potential for improving performance of ordinary programs, which are composed of many parallel loops and/or loops that can be auto-parallelized by the compiler or by the user. However, an effective exploitation of such a parallelism requires care in adapting chunks of parallel loops and allocating such chunks to the available cores - in order to balancing the load across cores and minimizing synchronization costs.

Loop scheduling algorithms and in particular self-scheduling algorithms (SS) addresses finding the correct trade-off between load balancing and synchronization costs to minimize the completion time of a parallel loop. However, the load

* This work was partially supported by the National Natural Science Foundation of China under grant NSFC-61300011.

imbalance which rises in modern multi-core systems - due to a deep and complex memory hierarchy organization and shared access to the main memory by multiple threads and processes, is such that self-scheduling algorithms deliver inconsistent performance across different parallel loops and in diverse execution conditions.

In this work we propose a new framework for scheduling parallel loops to multi-core systems - Locality Aware Self-scheduling (`LASS`). `LASS` has two main components: (a) a compilation environment which partitions the iterations of a parallel loops in batches and assigns each batch statically to one core. Each batch of iterations is subsequently partitioned in chunks of iterations according to one out of four widely adopted self-scheduling algorithms, a.k.a. `SS` [1], `GSS` [2], `FSS` [3], `TSS` [4] - these algorithms are customarily implemented in the GNU GCC compiler, the IBM XLC compiler and the Intel ICC compiler; (b) a runtime environment, which first selects the type of self-scheduling algorithm that is the most likely to speedup performance of a given parallel loop and second deploys `LASS` with the selected self-scheduling algorithm. A machine learning aided heuristic to select the self-scheduling algorithm and the number of cores to use is constructed offline. Experimental results show that LASS boosts up performance of known self-scheduling algorithms in diverse execution conditions.

The rest of the paper is organized as follows. Section 2 describes `LASS`. Experimental results are presented in Section 3. Section 4 provides a breakdown of prior work on self-scheduling and iteration scheduling in the presence of shared levels of memory hierarchy. Our conclusion is presented in Section 5.

2 Technique

In this section we present the LASS technique. To improve affinity, LASS assumes that each worker thread is assigned to a core, so the number of workers never exceeds the number of cores available on the system underneath.

2.1 Locality aware self-scheduler

The Master thread spawns P Workers and pins each Worker to a core. Next, the Master produces a list of chunk sizes, $\mathbf{C}$, according to a given self-scheduling algorithm. In addition to the above, the LASS scheduler partitions the parallel loop in P batches and assigns one batch to each Worker. Subsequently, each Worker executes the Algorithm 1 during the execution of a parallel loop.

When the Worker T_i completes the execution of its current chunk, it first attempts to fetch the next available chunk size C_j in the list $\mathbf{C}$ and then attempts to fetch C_j iterations from its batch B_i. If C_j iterations are available in the batch B_i, then T_i fetches C_j iterations from B_i starting from the iteration # n_i. Toward the end of the batch, however, the number of iterations available in B_i may be less than C_j. In this circumstance, the chunk C_j is split in two parts at run-time. The iterations from n_i until u_i are fetched by T_i, whereas a new chunk $C' = C_j - (u_i - n_i)$ is inserted in the queue $\mathbf{C}$. Eventually, if no more

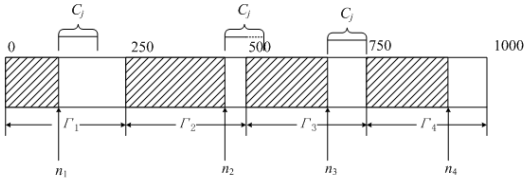


Fig. 1. LASS operations

iterations are available in the batch B_i, T_i can help other Workers completing their batches. In this case, multiple Workers will contend the access to the same batch of iterations, hence synchronization is required. This is the only scenario in which LASS requires synchronization. Indeed, with the exception of the last case mentioned above, a Worker can fetch C_j iterations from $\mathbf{C}$ without explicitly gain exclusive access to the queue of iterations. Once a Worker fetches a chunk size number from $\mathbf{C}$, it moves the index of $\mathbf{C}$ to the next position. Because the index of $\mathbf{C}$ is shared by all the Workers, two or more Workers can access the same chunk size sometimes. Even if this happens, the algorithm can still run correctly because the termination of the loop is not detected by checking $\mathbf{C}$ and $\mathbf{C}$ just provides chunk sizes but not real chunks.

For clarity, we present the example in Figure 1. Let us assume the iterations space being composed of 1000 iterations, that is $\Gamma = \{I_1, I_2, \cdots, I_{1000}\}$, and that these iterations need to be scheduled to run on $P = 4$ cores. The iteration space is partitioned in four batches composed of 250 iterations, $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \cup \Gamma_4$. Four Workers are spawn, T_1, T_2, T_3 and T_4. Each Worker is assigned to a different core, so that any time the Worker T_j processes a chunk, it will always run on the core P_j.

The Worker T_j is the owner of the batch Γ_j. When the parallel execution starts, the Worker T_j has exclusive access to its own batch. Before the Workers start, a self-scheduling algorithms is used to create a list of chunks, named $\mathbf{C}$. Iterations are scheduled in chunks as indicated in $\mathbf{C}$. Let n_i be the iterations index in Γ_i. At a scheduling step in Figure 1, $n_1 = 100$, $n_2 = 450$, and the upper bound for Γ_3 is $n_3 = 750$. There are three distinct possible scenarios:

- The Worker T_1 attempts to fetch C_j iterations from Γ_1. If $n_1 + C_j < u_1$, C_j consecutive iterations can be fetched from Γ_1. Next, the Worker T_1 fetches C_j consecutive iterations from Γ_1 starting from n_1 and executes them.
- The Worker T_2 attempts to fetch C_j from Γ_2. If $n_2 + C_j > u_2$, only $u_2 - n_2$ are fetched from Γ_2, and $C' = C_j - (u_2 - n_2)$ is a new chunk size which is appended to the list of chunk sizes.
- The Worker T_3 attempts to fetch C_j from Γ_3. If $n_3 + C_j = u_3$, all iterations in Γ_3 have already been processed. In this case, n_3 points to n_4. If the iterations in Γ_4 have also been consumed, both n_3 and n_4 point to n_1.

Algorithm 1 - Locality aware self-scheduling

P : number of cores;
$\mathbf{T} = \{T_1, T_2, \cdots, T_P\}$: worker threads;
$\mathbf{C} = \{C_1, C_2, \cdots, C_{\#chunks}, 0\}$: list of chunk sizes;
$\Gamma = \{I_1, I_2, \cdots, I_N\}$: queue of iterations;
n_i : current index of the i^{th} partition;
u_i : upper bound index of the i^{th} partition;
f_i : set to 1 if P_i shares its partition;

t_exit=**FALSE**
while (**TRUE**) **do**
  if (f_i=**TRUE**) **then**
    lock(Γ)
  end if
  get C_j iterations from Γ;
  $k = n_i + C_j$;
  if ($k < u_i$) **then**
    $lb = n_i$; $ub = n_i + C_j$; $n_i = n_i + C_j$;
  else
    if ($k - u_i > 0$) **then**
      Split the partition of the current chunk
      $C' = k - u_i$
      append C' to **C**
    end if
    $lb = n_i$; $ub = u_i$; $n_i = u_i$; $k = i$;
    repeat
      $n_k = n_{(k+1)\%P}$; $u_k = u_{(k+1)\%P}$; $k = k + 1$;
      if ($i = k$) **then**
        t_exit=**TRUE**; break;
      end if
    until ($n_k \neq u_k$)
    f_k=**TRUE**
  end if
  if (f_i=**TRUE**) **then**
    unlock(Γ)
  end if
  for $k = lb \rightarrow ub$ **do**
    Body of the parallel loop
  end for
  if (*t_exit*=**TRUE**) **then**
    exit
  end if
end while

2.2 Selection of the iteration scheduling algorithm and the number of Workers

LASS can work in combination with any self-scheduling algorithm and because there is no self-scheduling algorithm that enables optimal performance for any

parallel loop, we propose a simple heuristic to the problem of selecting the most suitable self-scheduling algorithm, given a characterization of a parallel loop. Likewise, we propose a heuristic to select the number of Workers delivering best performance.

Note that the selection of a self-scheduling strategy and the number of threads to maximize performance depends on many factors on a real system, such as the dynamic availability of cores, their instant load, etc. Thus, accurate analytical models cannot be derived, and in any case, building such models is out of the scope of this paper.

The heuristic proposed in this section is based on classification trees [5]. We characterize the behavior of a parallel loop based on the features of its loop body, such as uniform vs. non-uniform loop body. Non-uniform loop bodies are further characterized in terms of the source of non-uniformity, such as multi-way loop, non-perfectly nested loop, presence of conditionals and nested conditionals, etc. To such features we associate - as a label - the most profitable self-scheduling algorithm which maximizes performance of these loops, e.g., G for GSS, F for FSS and T for TSS.

We build a predictor based on classification tree which learns from examples such as $\mathbf{f} \rightarrow \{G, F, T\}$, where $\mathbf{f}$ indicates the description of the loop. Given an unseen vector of features, our predictor is in charge to predict the most suitable self-scheduling algorithm to minimize the execution time of a parallel loop. Such a prediction, as we will see in the next section, can be performed independently from the number of Workers allocated for its execution.

Following the same principle, we build a second classification tree using as features as combination of loop's feature, the self-scheduling algorithm previously selected and the input size - which is expressed as the total number of instructions retired. The output of this second classifier is the number of Workers to use in order to maximize performance. Our predictor learns from examples such as $(\mathbf{f}, s, I) \rightarrow p$, where $s \in \{G, F, T\}$, I is the number of instructions retired, and p indicates the execution time (performance) of the parallel loop.

3 Experiments

In order to evaluate our locality aware self-scheduling technique, we selected three popular self-scheduling algorithms to run in combination with our technique. These algorithms are guided self-scheduling (GSS)[2], factoring self-scheduling (FSS) [3] and trapezoid self-scheduling (TSS) [4].

3.1 Experimental setup

We extracted several kernels from the benchmark suites SPEC CPU2000/2006, SPEC OMP2001 and MiBenchII. The description of these kernels is provided in Table 1. We compiled and executed our kernels on the system configuration summarized in Table 2. Intel X7350 is a quad-core processor, which consists of

Table 1. List of kernels

Kernel	Benchmark suite	Benchmark	File, line
*L*1	SPEC CPU2000	**179.art**	scanner.c, 317
*L*2	MiBench	JPEG	jcdctmgr.c, 195
*L*3	SPEC CPU2000	**183.equake**	quake.c, 447
*L*4	SPEC CPU2006	**470.lbm**	lbm.c, 186
*L*5		matrix multiplication	mm.c
*L*6	MiBench	susan	susan.c, 738
*L*7	SPEC CPU2006	**433.milc**	quark_stuff.c, 1523
*L*8	SPEC CPU2006	**462.libquantum**	gates.c, 89
*L*9	SPEC CPU2006	**462.libquantum**	gates.c, 61
*L*10	SPEC CPU2006	**464.h264ref**	mv-search.c, 394
*L*11	SPEC CPU2006	**482.sphnix3**	vector.c, 512
*L*12	SPEC OMP2001	**172.mgrid**	mgrid.f, 189
*L*13		matrix transposition	mt.c

Table 2. System configurations

Processors	4 x Intel Xeon X7350 (Tigertown) @ 2.93GHz
L2	2 x 4MB
Main memory [GB]	8
Compilation	gcc4.5 -O3 -lpthread -lrt -lm
Thread library	NPTL 2.7
Operating system	Linux 2.6.22

two dual-core. This configuration accounts for a total of 16 cores. Each dual-core shares 4MB of shared L2 cache. We compiled the kernels listed in in Table 1 using GNU GCC v4.5 and the optimization level $-O3$ enabled.

Each performance result is the average of one hundred execution of each kernel to ensure dependability of the results. During each run we collect hardware performance counters using Perfmon2 [6].

3.2 Experimental results

We implemented the Algorithm 1 presented in section 2. To produce the list of chunks we refer to three widely used self-scheduling strategies: GSS, FSS and TSS. These three self-scheduling algorithms differ in terms of their chunking strategy, thereby their synchronization costs are different [3].

In the presentation of the experimental results, we refer as LASS-G when LASS is applied in combination with GSS. Mutatis mutandis, we use the nomenclatures LASS-F and LASS-T to indicate that LASS is applied in combination with FSS and TSS respectively.

For each kernel, we compare completion time obtained with a given self-scheduling strategy with the completion time of LASS, say LASS-{G,F,T}. As indicator of performance we use the speedup as defined in equation 1. Such a speedup is relative to the completion time the parallel execution of a kernel subject to a given self-scheduling algorithm.

$$S^{\#Workers}_{\{G,F,T\}} = \frac{Completion\ time_{G,F,T}}{Completion\ time_{LASS-G,F,T}} \tag{1}$$

We conducted the experiments in two execution environments. In the first execution environment, named **free system**, our applications run alone, one by one, on the system. In the second execution environment, named **full system** our applications run in conjunction with an interfering parallel job influencing the load of multiple cores at random. For each execution environment we conducted our experiments for a variable number of Worker threads from 2 to 16.

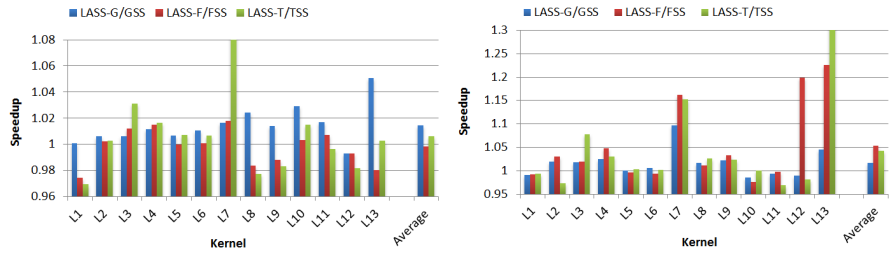
Analysis of performance and locality Results for the *free system* are reported in Figure 2. LASS improved performance in most cases. Our performance results are supported by the counters collected. In multi-cores, the cache miss count is the main reflection of the locality exploitation. Figure 3 shows the miss rate in the case of four threads running on the *free system*. This case is relevant given our hardware configuration. Figure 3 shows that L1 cache misses decreases, whereas L2 cache misses vary slightly or remains constant. The reduction of L1 cache misses is a direct effect of the adoption of LASS and does contribute to ameliorate performance. The slight variation in L2 misses is an artifact of the system we are running on.

For more than two workers, only couples of Workers share the second level of cache - because of the topology of the memory hierarchy on our system, limiting the benefit deriving from the enforcement of locality. Indeed, the kernels L10 and L11, whose working set size fits inside the last level of cache slightly benefit from the parallel execution with 2 Workers and their performance is severely compromised with the adoption a larger number of Workers.

On the other end, performance still improves because of the behavior of LASS toward the end of the parallel execution. Toward the completion of the parallel execution LASS creates additional chunks by splitting the last few chunks available. The availability of additional chunks increases the number of tasks to execute in parallel, the parallel execution still results profitable, thereby improves performance despite the obstacle imposed by our system configuration.

Moreover, kernels L7, L12 and L13 achieve the best speedups in most cases. Most likely reason is that the data in these kernels is much denser than other kernels. Therefore, LASS gains more benefits from the improvement of the data locality. However, it is hard to break down performance improvements attributed to various factors in a real machine.

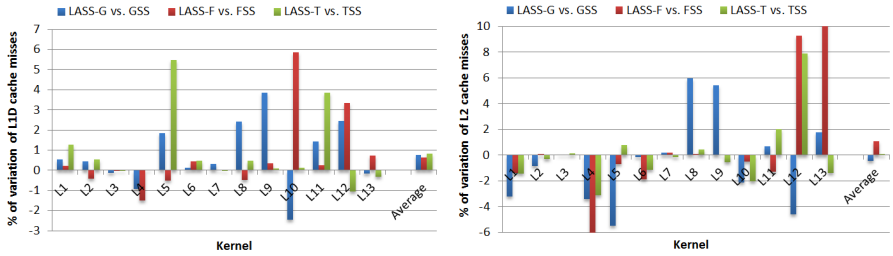
Next, we considered another execution environment, the *full system*. In this execution scenario cores are not available for our applications at the same time. Nevertheless, LASS still enhances performance of classical self-scheduling strategies, as it is shown in Figure 4. Experimental results show that the average speedup is significantly higher when compared to those of system free. These results highlight that performance achieved because of the adaptivity of self-scheduling strategy is effectively amplified by LASS. Furthermore, these results show that there is opportunity to achieve higher speedups if, when applying a self-scheduling strategy in both free and full systems, we were able to select ad hoc the number of working threads.



(a) 2 Workers (b) 4 Workers

(c) 8 Workers (d) 16 Workers

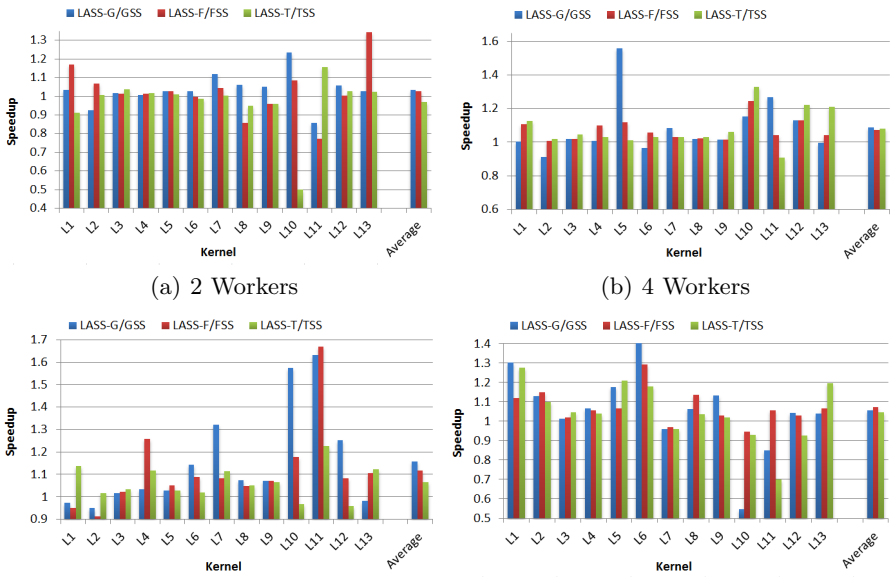
Fig. 2. Speedup w.r.t. self-scheduling algorithm on *free system*



(a) L1 (b) L2

Fig. 3. L1 and L2 miss rates improvement for four threads on *free system*

Analysis of synchronization operations Figure 5 shows the number of synchronization operations required to run LASS is significantly lower than the number of synchronization operations required by other non LASS self-scheduling strategies. This is a trend across the three GSS, FSS and TSS. The relative reduction of synchronization costs influences performance of each self-scheduling algorithms in a different way. For example, let us consider experiments using 16 worker threads. FSS is the self-scheduling strategy suffering from the highest synchronization costs because of the chunk sizes' distribution. When the threads involved in the computation start and progress simultaneously, the probability of having concurrent accesses is higher for FSS than GSS and TSS. Arguably, FSS is the self-scheduling strategy gaining the highest benefit from the elimination of the synchronization operations. Our experiments show an 3.42% average



(a) 2 Workers (b) 4 Workers

(c) 8 Workers (d) 16 Workers

Fig. 4. Speedup w.r.t. self-scheduling algorithm on *full system*

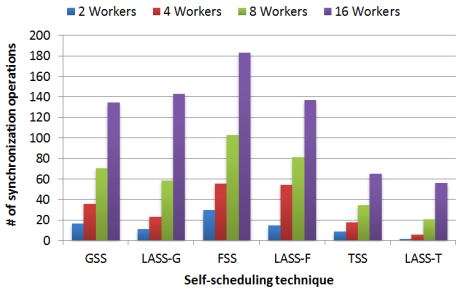


Fig. 5. Number of synchronization operations

reduction in execution time. Also TSS shows an 1.83% average reduction in execution time and this number is 3.34% for GSS algorithm. The influence that the reduction of synchronization operations has on performance of a self-scheduling algorithm depends on the distribution of the chunk sizes.

Selection of the self-scheduling algorithm and of the number of Workers The analysis of the vectors of counters collected and the types of parallel loop adopted in our experiments suggest the adoption of two simple heuristics, based on decision trees [7], to cope with the following problems: (a) Selecting the most beneficial self-scheduling algorithm for a given loop. (b) Selecting the number of Workers to achieve best performance from the parallel execution.

We classify our loops using the rules as follows: We refer as *uniform* such parallel loops which have constant cost per iteration. In this category fall loops with constant bounds and stride, containing inner loops with constant bounds and uniform strides, and containing function calls. We refer as *non uniform* such parallel loops containing conditionals, indirect references, variable bounds and/or strides. As first classification step we separate uniform from non uniform loops. Uniform loops containing other nested loops are labeled with an F, indicating FSS as the best candidate for this type of loops. Other uniform loops are labeled with G, which stands for GSS. Non uniform loops containing branches are labeled with F, which stands for FSS, whereas non uniform loops with indirect references or non constant loop body are labeled with T, which stands for TSS. This heuristic applied on our kernels is illustrated in Figure 6. Experimental results show that for both the execution environments, the *free system* and the *full system*, the selection of self-scheduling algorithm to apply can be performed visiting the decision tree in Figure 6 using the description of the parallel loop. This pass is done offline. We provide another offline heuristic which, given the features of a parallel loop and a self-scheduling algorithm, predicts the number of working threads needed to minimize its execution time. This second heuristic is based on the size of the input, represented by the number of instructions retired. This second heuristic is illustrated as an example in Figure 7.

The results of the experiments conducted using the heuristics described above are summarized in Table 3. Experimental results show an average speedup of 11% in the *free system*, and an average speedup of 31% in the *full system*.

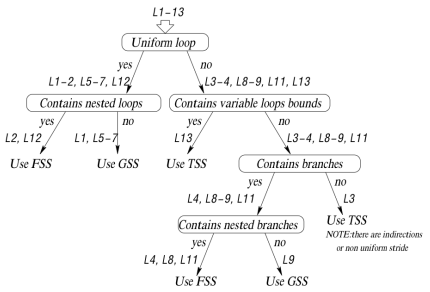


Fig. 6. Selection of self-scheduling per loop

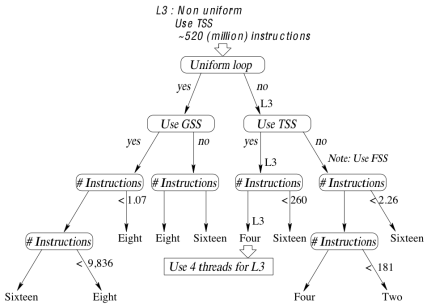


Fig. 7. Selection of the number of threads

4 Related work

Many iteration scheduling algorithms have been proposed in the literature. These algorithms leverage the presence of parallelism in a architecture to reduce execution time of ordinary programs. On one extreme, there is static scheduling which

Table 3. Selection of self-scheduling and # Workers on *free* and *full system*

Kernel	**Free system**			**Full system**		
	LASS	# Workers	*Speedup*	LASS	# Workers	*Speedup*
L1	G	8	1.02	G	16	1.30
L2	F	16	1.04	F	16	1.15
L3	T	4	1.08	T	16	1.05
L4	F	4	1.05	F	8	1.26
L5	G	8	1.06	G	2	1.56
L6	G	4	1.01	G	16	1.41
L7	G or F	16	1.16	G	8	1.32
L8	F	16	1.10	F	16	1.14
L9	G	16	1.10	G	16	1.13
L10	G	2	1.03	G	8	1.57
L11	F	2	1.01	F	8	1.67
L12	F	8	1.49	G	8	1.25
L13	T	16	1.24	T	8	1.21
Average			*1.11*			*1.31*

assigns even partitions of a loop iterations to multiple cores. Compared to other schemes, it has the lowest scheduling overhead but it may incur in the worst load balancing when scheduling irregular parallel loops. On the other extreme, there is the first self-scheduling [1]. It assigns one iteration to an idle core each time, to achieve best load balancing, but has the highest execution and synchronization overheads. For having a trade-off between execution overhead and load balancing, the adoption of fixed chunks was proposed by other authors [8]. However, The selection of the chunk size is challenging. In fact, small chunk sizes allow the exploitation of more parallelism, whereas larger chunk sizes reduce the run-time overhead. Rather than the use of fixed chunk sizes, Kruscal and Weiss in [9] proposed the adoption of chunk sizes with a decreasing profile down to chunks containing only one iteration. In the beginning, threads are allowed to fetch larger chunks, thus achieving low parallel execution overhead. Toward the end of the parallel loop the presence of smaller chunks allows to achieve better load balancing. Among the self-scheduling algorithms proposed in the literature, GSS [2], FSS [3] and TSS [4] are widely used and implemented in open source and commercial compilers.

In the other self-scheduling strategies technique in the literature [10, 11], adjusted chunk sizes at run time or processor affinity is exploited. Markatos and LeBlanc in [12] propose affinity scheduling, which is locality aware, but it suffers of load balancing when dealing with irregular loops.

In the work stealing literature [13], the scheduling algorithms are all locality aware because of the use of per-processor work queues. Work stealing schedulers aim to tasks which are independent units of works that can be executed in parallel. In Cilk [14] and Intel TBB [15] which are popular frameworks using work stealing, a parallel loop is partitioned to fixed chunks. Then each chunk is viewed as a task. To the best of our knowledge, LASS technique combining self-scheduling with work stealing capabilities.

5 Conclusion

In this paper we proposed a new iteration scheduling technique - locality aware self-scheduling - which, in combination with any self-scheduling algorithm, systematically reduces the number of synchronization operations required to assign cores to chunks, enforces both spatial and temporal locality, enforces affinity and adapts the mapping of chunks onto iterations at run-time, therefore improves on load balancing and performance. As a part of our technique we propose a machine learning based heuristic, which is based on decision trees, to select the most suitable iteration scheduling algorithm and number of threads to minimize the completion time of a parallel loop.

References

1. B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. Real Time Signal Processing IV, vol. 298, pp. 241-298, 1981.
2. C. D. Polychronopoulos, D. J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. IEEE Trans on Computers 36(12):1425-1439, 1987.
3. S. Flynn-Hummel, E. Schonberg, and L. E. Flynn. Factoring: A method for scheduling parallel loops. Communications of the ACM 35(8):90-101, 1992.
4. T. H. Tzen, L. M. Ni. Trapezoid self-scheduling: a practical scheduling scheme for parallel computers. IEEE Trans on Parallel and Distributed Systems 4(1):87-98, 1993.
5. L. Breiman, J. Friedman, et al. Classification and Regression Trees. Chapman & Hall/CRC, 1984.
6. S. Jarp, R. Jurga, and A. Nowak. Perfmon2: a leap forward in performance monitoring. J. Phys.: Conf. Ser., 119:042017, 2008.
7. V. Podgorelec, P. Kokol, et al. Decision trees: An overview and their use in medicine, J. Med. Syst. 26, pp. 445-463, 2002.
8. P. Tang, P. C. Yew. Processor self-scheduling for multiple nested parallel loops. In ICPP, pp. 528-535, 1986.
9. C. P. Kruskal, A. Weiss, Allocating independent subtasks on parallel processors, IEEE Trans. Softw. Eng., SE-1 1(10): 1001- 1016, 1985.
10. R. L. Cariño, I. Banicescu. Dynamic load balancing with adaptive factoring methods in scientific applications. J. Supercomput 44(1):41-63, 2008.
11. T. Tabirca, L. Freeman, et al. Feedback guided dynamic loop scheduling: convergence of the continuous case. J. Supercomput 30(2): 151-178, 2004.
12. E. P. Markatos, T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. 5(4):379-400, 1994.
13. R. D. Blumofe, C. E. Leiserson. Scheduling multithreaded computations by work stealing. J. ACM, 46(5): 720C748, 1999.
14. R. D. Blumofe, C. F. Joerg, et al. Cilk: An efficient multithreaded runtime system. In PPoPP, pp. 207-216, 1995.
15. Intel(R) Threading Building Blocks, Intel Corporation.