Just in time load balancing

Rosario Cammarota, Alexandru Nicolau, and Alexander V. Veidenbaum

University of California Irvine

Abstract. Leveraging Loop Level Parallelism (LLP) is one of the most attractive techniques for improving program performance on emerging multi-cores. Ordinary programs contain a large amount of parallel and DOALL loops, however emerging multi-core designs feature (a) a rapid increase in the number of on-chip cores and (b) the ways such cores share on-chip resources - such as pipeline and memory hierarchy, leads to an increase in the number of possible high-performance configurations. This trend in emerging multi-core design makes attaining peak performance through the exploitation of LLP an increasingly complex problem.

In this paper, we propose a new iteration scheduling technique to speedup the execution of DOALL loops on complex multi-core systems. Our technique targets the execution of DOALL loops with a variable cost per iteration and exhibiting *either a predictable or an unpredictable behavior* across multiple instances of the loop. In the former case our technique implements a quick run-time pass - to identify chunks of iterations containing the same amount of work - followed by a static assignment of such chunks to cores. If the static parallel execution is not profitable, our technique can decide to run such a loop either sequentially or in parallel, but using dynamic optimization to select an appropriate chunk size to optimize performance.

We implemented our technique in GNU GCC/OpenMP and demonstrate promising results on three important linear algebra kernels - matrix multiply, Gauss-Jordan elimination and adjoint convolution - for which near-optimal speedup against existing scheduling techniques is attained. Furthermore, we demonstrate the impact of our approach on the already parallelized program `470.lbm` from SPEC CPU2006, implementing the Lattice Boltzman Method. On `470.lbm`, our technique attains a speedup up of to 65% on the state-of-the-art 4-cores, 2-way Symmetric Multi-Threading Intel Sandy Bridge architecture.

1 Introduction

Parallel loops are the largest source of parallelism in ordinary programs - the coverage of parallel DOALL loops in SPEC CFP2000 and CFP2006[1, 2] contain inherently parallel or DOALL loops [3]. For parallel loops without dependencies across iterations, e.g., DOALL loops, the execution of such loops in parallel can significantly speedup ordinary programs. Several dynamic [4–9] and static [10] techniques have been proposed for scheduling iterations of parallel loops on parallel machines. However, scheduling DOALL loops on modern multi-cores - with complex memory hierarchy organizations and multiple levels of parallelism, such as instructions level parallelism, vector units, symmetric multi-threading

etc. - may not deliver the peak performance[11]. On one hand, while free of run-time overheads, static techniques are either too simple (e.g., the OpenMP[12] implementations in GNU GCC [13] and Intel ICC [14]) to cope with cases where the cost per iteration is variable, or too complex, e.g., profile-based techniques [10], to be implemented as part of run-time systems. The complexity of such techniques grows with the number of iterations and the loop bounds. Dynamic techniques, on the other hand, may reduce the benefit of the parallel execution when, for example, the run-time synchronization overhead is relatively large compared with the execution time (*cost*) of the serial loop.

This paper proposes a new scheduling technique to speedup the execution of DOALL loops on modern complex multi-core systems. Our technique targets the execution of DOALL loops with a variable cost per iteration - e.g., loops performing triangularization, that constitutes a fundamental step for many linear algebra solvers. The proposed technique relies on the assumption that parallel loops are invoked multiple times during the execution of an ordinary program, as exemplified by the number of instances of hot loops in SPEC OMP2001[15, 16] (> 50 per hot loop) and NAS parallel benchmarks (from 100 to > 1000 times per hot loop). At run-time, our technique first attempts to find a static schedule such that the work is "optimally" distributed among cores. If such a schedule is not profitable and/or unattainable - because the behavior of present instances of the loop is not predictive of the behavior of future instances - our technique can decide to run such a loop either sequentially or in parallel, but using dynamic scheduling techniques with an appropriate selection of the number of iterations to schedule at each time - referred to as a `chunk size`.

Specifically, our technique implements a quick pass at run-time, which attempts to determine chunks of iterations - that are expressed as a percentage of the total number of iterations - such that each chunk contains an equal part of the total cost of the loop - which is expressed in cycles and corresponds to the p^{th} fraction of the total cost of the loop, where p is the number of available cores. Such chunks are subsequently statically assigned to the p cores. Given that the assignment of chunks of iterations to cores is static, at this stage the proposed technique does not have run-time synchronization overheads, which can be a performance bottleneck for dynamic scheduling techniques.

When a parallel loop does not exhibit a variable cost per iteration, our technique outputs chunks of equal size. For example, while scheduling a parallel loop with constant cost per iteration - such as the case of a basic implementation of matrix multiply - on two cores, our technique determines, independently from the matrix size, that 50% of the iterations must be assigned to one core and the remaining 50% must be assigned to another core. On the contrary, in the case of a loop with a decreasing/increasing or exhibiting arbitrary variations in the cost per iteration, our technique will determine the percentage of iterations to assign to each cores such that each core will execute the same amount of work.

When the behavior of previous instances of a parallel loop is not predictive of the behavior of future instances, our technique can decide to execute the instances of such a loop in parallel, but opting for a dynamic scheduling strategy.

In this case, an estimation of the number of iterations to schedule at once, i.e., the chunk size is attempted. The determination of the chunk size uses a heuristic based on the average cost per iteration and the cost of synchronization overhead per iteration. While the average cost per iteration is determined at run-time by our technique, the cost of synchronization overhead is estimated offline using micro-benchmarking [17] on the architecture in use. The chunk size is determined using the cost of synchronization overhead over the average cost per iteration - refer to Section 2.2. Alternatively, if dynamic scheduling is not profitable, our technique opts to run the loop sequentially.

We implemented our technique in GNU GCC OpenMP and show promising results on three important linear algebra kernels - matrix multiply, Gauss-Jordan elimination and adjoint convolution implemented as in [5] - and `470.lbm` from SPEC CPU2006 on 4-cores, 2-way Symmetric Multi-Threading (SMT) cores Intel Sandy Bridge architecture. Specifically, our technique attains nearly optimal speedup for the three kernels above and up to 65% performance improvement for `470.lbm` against the use of prior scheduling techniques.

The rest of the paper is organized as follows: our technique is detailed in Section 2; Experimental results are presented in Section 3; Prior and related work are discussed in Section 4; The conclusion is presented in Section 5.

2 Just in time load balancing

The technique proposed in this paper is motivated by the observation that many instances of a parallel DOALL loop are usually executed in ordinary programs [16]. Therefore, a few of these instances can be leveraged to learn properties of the parallel loop and prepare a schedule which optimizes program performance. Our initial goal is to prepare a schedule which distributes uniform chunks of iterations to the available cores. When the threads involved in the parallel execution start nearly at the same time (this situation typically occur for program executing parallel loops according to a fork-join execution model) such threads are also likely to complete their executions nearly at the same time and overall the number of elapsed cycles of the parallel execution is minimized.

The typical scenario considered in this paper is shown in Listing 1.1, where a sequence of many instances of a parallel loop - the inner loop - are executed. The assumption made on Listing 1.1 about the body of the parallel loop are: (a) the body of the loop must contain re-entrant/thread-safe code; (b) the statements in the body of the loop are at most a function of the indexes of the loops surrounding it - in the example the body of the loop depends on the indexes `xx` and `tt`; (c) the loop bounds and the stride are constant (a relaxation of such an assumption is discussed later on in this section). [1]

The basic idea of our technique is to transform - at compile-time - the loop of Listing 1.1 into that of Listing 1.2. The outermost serial loop is "distributed" in three consecutive passes, as shown in Listing 1.2.[2] The first two instances of

[1] Note that, the assumptions made on the body of the loop admit the presence of nested (parallel/serial) perfect or multi-way loops with conditional, functions calls, indirect references, etc.

[2] The pseudo-code in Listing 1.2 illustrates a principle implementation.

the parallel loop are executed sequentially whereas the subsequent instances are executed with a schedule optimizing performance.

Listing 1.1. Loop model

```
/* Serial loop iterating over time steps */
for (tt=0; tt<time_max; tt++)
{
  /* Parallel loop automatically parallelized with OpenMP */
  #pragma omp parallel for
  for (xx=start; xx<end; x+=stride)
  {
    /* Body of the loop */
    Body(xx, tt);
  }
}
```

Listing 1.2. Transformed, adaptive instrumented loop

```
...
/* parameters of the scheduling algorithm
   where p is the number of threads      */
int *parts;
int ii;
long long d, C;
...

/* Serial loop iterating over time steps */

/* Pass 1 - Compute the total cost per iteration  */
start_probe_cost(&C);
for (xx=start; xx<end; x+=stride)
{
  Body(xx, 0);
}
get_probe_cost(&C);

/* Pass 2.a - Compute the schedule  */
start_diff_probe_cost(&d, &ii, p, C, parts);
for (xx=start; xx<end; x+=stride)
{
  Body(xx, 1);
  get_diff_probe_cost(&d, &ii, parts);
}

/* Pass 2.b - Configure the scheduler */
omp_set_scheduler(p,parts);

/* Pass 3 - execute the remaining instances of the
   parallel loop in parallel */
for (tt=2; tt<time_max; tt++)
{
  /* Parallel loop automatically parallelized with OpenMP */
  #pragma omp parallel for
  for (xx=start; xx<end; x+=stride)
  {
    /* Body of the loop */
    Body(xx, tt);
  }
}
```

The first instance of the parallel loop is instrumented to compute the overall cost of the parallel loop - expressed in terms of the total number of elapsed cycles. The function `start_probe_cost(·)` initializes the variable C, which will contain the total cost of executing the serial loop, whereas the function `get_probe_cost(·)` reads the elapsed cycles after the execution of the loop and assigns the number

of cycles to C. This pass also counts the total number of iterations - referred to as `# iterations` and derives the average cost per iteration as the total cost divided by the number of iterations.

The second instance of the loop is instrumented to profile the cost per iteration and to determine the percentages of iterations - taken in lexicographic order - that contain the $\frac{1}{p}$ part of the total cost of the loop, where p is the number of available cores. In particular, the function `start_diff_probe_cost(·,·,·,···)` initializes the following counters: (a) `d` - which, for each iteration, will contain the cost per iteration; (b) the iteration number `ii`; and (c) the integer vector `parts` - which contains as many entries as the number of threads/cores. For example, in the case of four threads, the vector parts will be initialized as `parts={0, 0, 0, 0}`. During the execution of the second instance of the loop, the cost per iteration `d` is computed by subtracting the current cycles count from the count of cycles annotated at the previous iteration - such cycles counts are taken from the beginning of the loop. For example, let c_{ii-1} be the elapsed cycles from the beginning of the loop until the iteration $ii-1$ and c_{ii} be the cost accumulated from the beginning of the loop until iteration ii. The value of the counter `d` is defined according to Equation 1.

$$\mathtt{d} = c_{ii} - c_{ii-1} \tag{1}$$

The function `get_diff_probe_cost(·,·,·)` implements the steps in Equations 2 and 3. In particular, for each $s = 1, 2, \cdots, p$ such a function finds the percentage of the iteration space containing the p^{st} part of the total elapsed cycles and assign such percentage to the position s of the array `parts` (to compensate for the ceiling operation, for $s = p$, $\mathtt{parts[p]} = 100 - \sum_{s=1}^{p-1} \mathtt{parts[s]}$):

$$\forall s = 1, 2, \cdots, p \;\; \mathtt{find\ ii_s} \;:\; \sum_{ii=ii_{s-1}}^{ii_s} \mathtt{d_{ii}} < s \times \frac{C}{p} < \sum_{ii=ii_{s-1}}^{ii_s} \mathtt{d_{ii}} + \mathtt{d_{ii_s+1}} \tag{2}$$

$$\mathtt{parts[s]} = \lceil \frac{\mathtt{ii_s}}{\#iterations} \rceil \times 100 \tag{3}$$

When a parallel loop exhibits a variable cost per iteration, the fraction of iterations containing a certain percentage $x\%$ of the overall cost of the loop is no longer proportional to x, as it would be in the case of a loop with uniform cost per iteration. Furthermore, in the case of a loop with an equal cost per iteration, the passes in Equations 2 and 3 provide `parts` to contain the equal elements. For example, let $p = 2$ and let us assume that the parallel loop have nearly-equal cost per iteration, then the array `parts` will be equal to `parts={50, 50}`. Likewise, for $p = 4$, `parts={25,25,25,25}` etc.

The next step in our run-time technique is the deployment of the schedule. If the number of operating threads are allocated to individual cores and nearly start at the same time, the run-time partitioning technique described above results in the minimum completion time - such as when iterations of a DO ALL loop are scheduled using the OpenMP construct `parallel for`, assuming that there is no significant relative change from run to run.

2.1 Instrumentation and profiling overhead

The definition of `d` given in Equation 1 is useful in practice. Such a definition allows (a) measuring the cost per iteration accurately and (b) estimating the accuracy with which the cost per iteration is measured - refer to Equation 4.

The sum of the measured costs per iteration, measured using the procedure `get_diff_probe_cost(·,·,·)`, is an estimator of the whole cost of the serial loop. Therefore, at run-time the total count of elapsed cycles `C` can be compared with the quantity $\sum_{ii=1}^{\#iterations} \mathtt{d_{ii}}$ to estimate the accuracy of our profiling technique. Likewise, for such a comparison, the total number of instructions executed (retired in the case of out-of-order cores such can Intel Xeon cores) can be used. We define the accuracy of the profiling as in Equation 4. The lower is $\epsilon\%$, the lower is the contribution of the instrumentation overhead to the run-time behavior of the parallel loop and the more the partitions of the iteration space convey the same amount of work to each core.

$$\epsilon\% = \frac{\left|\mathtt{C} - \sum_{ii=1}^{\#iterations} \mathtt{d_{ii}}\right|}{\mathtt{C}} \times 100 \tag{4}$$

2.2 Extension to more variable and non profitable cases

There exist cases that violate the assumptions that we made at the beginning of this section. For example, the bounds and the stride of the parallel loop can be functions of the outermost serial loop and/or functions of the input data. A typical illustration is provided by multi-grid kernels, that improve the resolution of raster images by varying the size of the image grid. In such a case, we relax our assumptions of constant bounds and strides and admit the possibility that the parallel loop bounds and/or stride can vary. However, for our scheme to work well, it requires that such a variation be slow - that is, given a variation of the loop bounds and/or stride is of interest for (i.e., is the same or similar) several subsequent instances of the parallel loop - we refer to such instances with similar behavior as a phase within which the serial loop is executing and we allow that instances of the parallel loop can be in different phases. Such phases can either occur systematically, such as in the case of multi-grid kernels - where phase changes are triggered by the outermost loop, or in an unpredictable way - e.g., the behavior of the body of the loop depends on the input data. In the former case, our technique can be programmed to be triggered by phase changes, as the occurrence of a phase change can be predicted. In the latter case our technique can be re-invoked periodically and attempt to optimize performance.

Either way, admitting a slow variation of the bounds and/or stride of the parallel loop implies that the elapsed time of the parallel loop vary across phases. This means that there can be instances of the parallel loop where the parallel execution is not profitable (e.g., when the body of the parallel loop is too small compared with the parallelization overhead) and instances of the parallel loop with a large number of iterations and a small uniform cost per iteration, where a dynamic iteration scheduling technique may perform better than a static technique when the chunk size is selected properly. Furthermore, dynamic

scheduling techniques are important as they can cope with adversary conditions of the system underneath - multi-programming conflicts or over/bad utilization of architectural resource, including under utilization of the memory hierarchy.

The more variable and the unprofitable cases mentioned above are taken into account in the following extension of our technique, where the source code of Listing 1.1 is transformed in that of Listing 1.3. Such code extends our technique with a set of conditions to enable/disable its passes in order to trigger the exploration of static and dynamic iteration scheduling techniques. The code in Listing 1.3 has the ability to adapt to different phases in which instances of the parallel loop can execute. Such an extension allows for the possibility to switch from the proposed static iteration scheduling schema to a dynamic scheduling schema, where the determination of the chunk size is fundamental to optimize performance.

Listing 1.3. Transformed, instrumented loop

```
/* Serial loop iterating over time steps */
for (tt=0; tt<time_max; tt++)
{
  if (condition_pass1(tt) or switch_scheduling) {
    /* Pass 1 - Compute the total and the average
       cost per iteration */
    ...
  }
  if (condition_pass2(tt) or switch_scheduling) {
    /* Pass 2.a - Compute the schedule  */
    ...
    /* Evaluate when to switch or to keep the schedule to:
       sequential, parallel static or dynamic */
    ...
    /* Pass 2.b - Compute chunk size and configure the scheduler */
    ...
  }
  if (condition_pass3(tt) or switch_scheduling) {
    /* Pass 3 - execute the remaining instances of the
       parallel loop in parallel */
    ...
  }
}
```

In this work, the determination of the chunk size is performed by profiling several types of parallelization and synchronization overheads using micro-benchmarking [17] and having run-time overhead costs factored into our technique.[3] In the cases that dynamic scheduling is adopted, the chunk size is determined such that the average cost per iteration (which is computed in pass 1 of Listing 1.3) times the chunk size is greater than the synchronization overhead (such overhead can be estimated using micro-benchmarking). Such a rule for selecting the chunk size guarantees that the resulting synchronization overhead is lower than the cost of the serial loop. Therefore the parallel execution on a number of cores $p \geq 2$ is profitable. Indeed, let n be the number of iterations and $a = \frac{C}{n}$ the average cost per iteration. Let δ be the cost of synchronization per iteration. The total cost of executing the parallel version of the loop on a single core, with a certain `chunk_size` will be equal to $C + \delta \times \frac{n}{\texttt{chunk\_size}}$ - as the threads will be executed in a serial fashion. Imposing $\delta \times \frac{n}{\texttt{chunk\_size}} < C$,

[3] Note that, run-time overheads are tightly coupled to the architecture underneath.

Model	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
L1 I/D cache [KB]	32
L2 cache [kB]	256
LLC [MB]	8
Memory [GB]	8
Compilers/linker, options	GNU GCC 4.6.2, -O3 -fopenmp
Operating system	Linux kernel 3.0.0

Table 1. System level setup

provides the total cost of execution on one core being less than $2C$. Therefore, the total cost of execution on p cores will be less than $\frac{2}{p} \times C$, which guarantees a speedup larger than one when $p > 2$. Eventually, $\delta \times \frac{n}{\texttt{chunk\_size}} < C$ is equivalent to $\frac{C}{n} \times \texttt{chunk\_size} > \delta$, which is equivalent to $a \times \texttt{chunk\_size} > \delta$, or otherwise $\texttt{chunk\_size} > \frac{\delta}{a}$.

Finally, during a phase when the total cost of the loop is small compared to the parallelization overhead - for example because the number of iterations and the cost of the body of the parallel loop are relatively small, our extended technique can decide to run the serial version of the loop - refer to the pass 1 in Listing 1.2.

3 Experiments

We implemented our technique in the GNU GCC compiler as an extension of its OpenMP implementation [12, 18] - referred to as GOMP. GOMP includes a static scheduler that distributes equal chunks of iterations to the available cores, and two dynamic schedulers in each of which an idle core gains exclusive access to the queue of iterations and fetches the next available chunk of iterations. `Fixed chunk size` scheduling and `Guided` [4] are the two dynamic scheduling strategies implemented in GOMP. In addition, we implemented two other popular dynamic scheduling strategies: `Factoring` [5] and `Trapezoid` [7].

We use PAPI [19] to access the hardware performance counters to count elapsed cycles. Our experiments are conducted on the state-of-the art Intel Sandy Bridge architecture - the system level configuration illustrated in Table 1. The dynamic frequency scaling was disabled to provide dependable time and counters measurements.

As benchmarks, we use OpenMP implementations in C[20] of three linear algebra kernels: Matrix Multiply; Gauss-Jordan elimination; and Adjoint Convolution[4] - and the program `470.lbm` from SPEC CPU2006[2], which implements the Lattice Boltzman Method as illustrated in [21] and gives us a more "full application" sample program. Fifty instances of matrix multiply and adjoint convolution were executed for each experiment. The number of times the parallel loop in Gauss-Jordan elimination is iterated is a function of its outermost serial loop [5]. The first few - up to 5 - instances are leveraged by our technique to find a profitable schedule.

We selected the three kernels above to verify that the profiling method included in our technique effectively provides accurate estimates of the cost per

[4] Our implementation resembles the parallel form of such kernels as illustrated in [5].

iteration and therefore our technique can identify partitions of iterations with equal costs during the training phase. A more in depth description of the runtime properties per iteration of these kernels is discussed below. Such kernels are also utilized to verify basic scalability properties in the performance results attained with our technique against the use of prior scheduling techniques, when the number of threads increases. Eventually, the program `470.lbm` is used as a real world benchmark for our technique.

	Matrix Multiply	Gauss-Jordan elimination	Adjoint convolution
Average # cycles per iteration	26786087	1936474	9346
Standard deviation	22683	860399	5337

Table 2. Average cost per iteration and standard deviation.

A summary of the variability of the cost per iteration for the the kernels matrix multiply, Gauss-Jordan elimination and adjoint convolution is shown in Table 2 - each column reports the average cost per iteration of one instance of each kernel and the corresponding standard deviation.

For matrix multiply and Gauss-Jordan elimination, we adopted matrices of type `double` whose size is 1024×1024 - a single matrix has the same size as the last level of cache in our architecture. Matrix multiply exhibits a constant cost per iteration. The small variability in the cost per iteration is due to the variable latency in the accesses to memory. Gauss-Jordan elimination exhibits a cost per iteration that is slightly variable because of a conditional in the body of the parallel loop. This kernel executes multiple instance of its innermost parallel loop. The cost per iteration within each instance of the parallel loop has a trapezoidal shape, which vary slowly - from nearly rectangular to nearly triangular - across subsequent instances of the parallel loop. For adjoint convolution, we adopted vectors of type `double` whose size of 102400. The cost per iteration decreases with a constant rate and falls within a large range. More importantly, iterations with larger cost are not uniformly distributed across the iteration space. Indeed, most of the whole cost of the loop is concentrated within the first few iterations. As we will see in this section, such a biased distribution of the cost per iteration is a limiting factor for dynamic schedulers, such as guided self-scheduling.

The program `470.lbm`, using the reference dataset in SPEC CPU2006, calls frequently (> 100 times) the function `LBM_performStreamCollide`, that accounts for most of the execution time of the program. Such a function includes a singly nested parallel loop with conditionals - the loop is hand-optimized as illustrated in [21]. The cost per iteration is constant although very small (each iteration executes in ≈ 3 `ns`). The number of iterations in the corresponding reference input data set [2] is very large (it amounts to $26,000,000$ iterations).

3.1 Profiling accuracy and micro-benchmarking summary

The iteration cost profiling method proposed for the pass 2 of our technique (refer to Listing 1.2) provides accurate estimates of the cost per iteration - refer to Table 3. Indeed, for each benchmark $\epsilon\%$ is very small - negligible in the case of matrix multiply and adjoint convolution. The worse case estimation happens

in the case of the kernel adjoint convolution, because of the presence of a few iteration with a large cost followed by plenty of iterations with a small cost. The profiling the former type of iterations is more accurate than that of the latter type of iterations, nevertheless $\epsilon\% < 2\%$.

	Matrix Multiply	Gauss-Jordan elimination	Adjoint convolution
$\epsilon\%$	0.04	0.03	1.9

Table 3. Relative estimation error - refer to Equation 4.

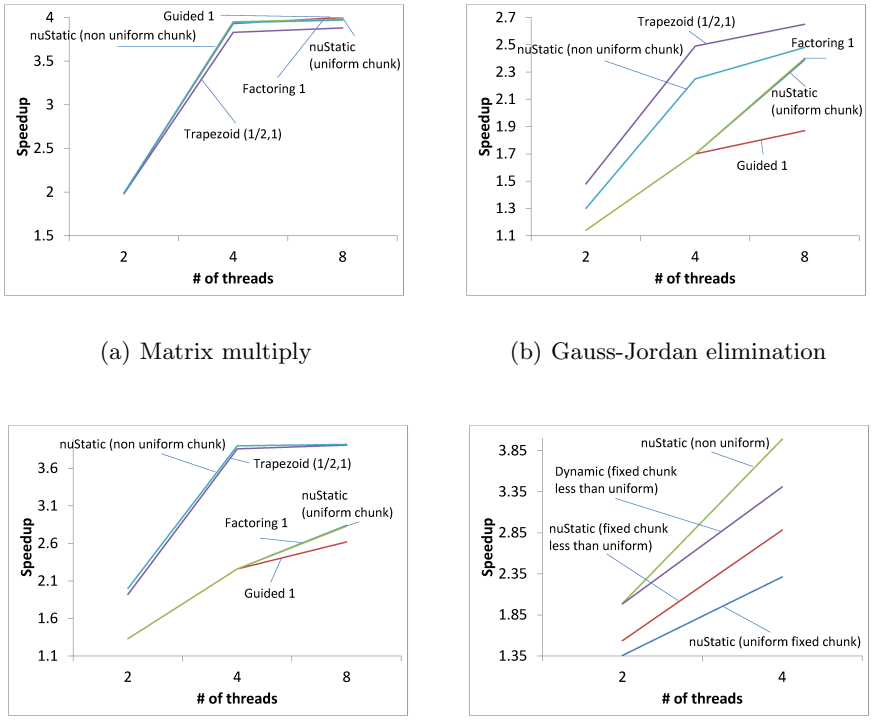
On the architecture in use - refer to Table 1 - we characterize the overhead costs involved with the use of the OpenMP constructs for scheduling and synchronization using the suite of micro-benchmarks EPCC [17]. In the micro-benchmarking method proposed in [17], the overhead involved in the parallel execution of a parallel loop on p cores is empirically defined as $O(p) = T_p - \frac{T_s}{p}$, where T_s is the number of cycles needed for the sequential execution and T_p is the number of cycles need for the parallel execution. Our experimental results - conducted for $p = 2, 4, 6, 8$ can be summarized as follows: (a) the overhead involved with static scheduling is $\approx 0.25\mu s$ for up to four threads and bumps to $\approx 3\mu s$ fore more than four threads - because of the presence of two hardware threads sharing the same core in a symmetric multi-threading fashion. Such a overhead is independent from the number and the sizes of the chunks; (b) the overhead involved with dynamic fixed chunk scheduling increases with the number of threads - it raises from ≈ 0.25 to $17\mu s$. However the overhead drops significantly when the chunk size increases - the trend is that the overhead decreases as $\approx \frac{1}{\texttt{chunk\_size}}$; (c) the overhead involved with guided scheduling increases with the number of threads - it raises from ≈ 0.25 to $4.5\mu s$, and decreases nearly linearly when the chunk sizes increases.

The above results from micro-benchmarking help us in building a heuristic for our technique to estimate a chunk size for dynamic scheduling to optimize performance.

3.2 Experimental results

We present a first set of results aimed to compare static and dynamic iteration scheduling schemes, including the one proposed as a part of our technique - which we refer as `nuStatic`. `nuStatic` schedules statically chunks of iterations with equal cost - in terms of elapsed cycles - to cores. We recall that in `nuStatic` chunks contains the same number of iterations in the case of loops with constant cost per iterations, whereas the chunks contain different numbers of iterations when the cost per iteration vary. In the case of constant cost per iteration `nuStatic` is equivalent to the classic scheduler implemented in conventional OpenMP implementations. We refer to the case of classic static scheduling as `nuStatic uniform chunk`, whereas we use the term `nuStatic non uniform chunk` to refer the case of static scheduling when non uniform chunks are determined at run-time by our technique.

For matrix multiply all the iteration scheduling techniques attain nearly the same performance, as the cost per iteration is constant and much larger than the scheduling overhead. Performance is shown in Figure 1(a). `nuStatic uniform`



(a) Matrix multiply

(b) Gauss-Jordan elimination

(c) Adjoint convolution

(d) Adjoint convolution analysis

Fig. 1. Performance evaluation/analysis of individual scheduling techniques for multiple threads.

`chunk` and `nuStatic non uniform chunk` determine the same vectors of `parts` and the parts are equal. For example, $p = 2$ implies that `parts=`$\{$`50, 50`$\}$.

For Gauss-Jordan elimination - refer to Figure 1(b), we re-execute the determination of the vector `parts` three times - at the beginning, at $\frac{1}{3}$ and at $\frac{2}{3}$ of the iterations of the outer-most loop. For example, when $p = 4$ and during each training phase, the following vectors of parts are learned by our technique: first `parts` $= \{25, 25, 25, 25\}$; subsequently `parts` $= \{15, 20, 25, 40\}$; and finally `parts` $= \{10, 15, 25, 45\}$. While performance attained by `nuStatic non uniform chunk` is slightly lower than that attained with `Trapezoid`, the former significantly outperforms the other schedulers, because of its relatively low synchronization overhead. Conceivably, the performance of `nuStatic non uniform chunk size` can conceivably approach the performance of `Trapezoid` by providing the former with more re-invocations of the pass 1 in our technique.

Among the three kernels considered in this paper, the most interesting is adjoint convolution. For adjoint convolution, none among static and dynamic scheduling strategies are able to deliver good performance - refer to Figure 1(c)

for the performance results. The issue with dynamic scheduling is either that synchronization costs are large and/or severe load imbalance occurs because iterations with large cost fall in large initial chunks for both `Factoring` and `Guided`. Unfortunately, an increase of the chunk size does not help reduce the size of the first few chunks for both `Factoring` and `Guided`. On the contrary, the fact that the first few chunk sizes are smaller in `Trapezoid` determines its success when compared with the other techniques. Nevertheless, to learn at runtime the parameters of `Trapezoid` for any kind of loop would be expensive. On the contrary, `nuStatic` provides equal or better performance than `Trapezoid` and the training phase of `nuStatic` is simple and efficient. Note that on an alternative implementation of adjoint convolution, when the cost per iteration increases, `Trapezoid` would have been unable to deliver best performance.

For adjoint convolution, an analysis of the capabilities of `nuStatic` compared with classic static and dynamic scheduling is presented in Figure 1(d). In the case of static scheduling, when equal chunks of iterations are distributed among the cores, performance is reduced because of load imbalance. For example, when $p = 2$, the thread assigned with the first chunk of iterations executes much more work than the other thread - because 50% of the whole cost of the loop is contained in the first 30% of the iterations. Likewise, when a smaller chunk size is used, threads are assigned multiple chunks of iterations in a round robin fashion. For example, the first threads will execute 60% of the iterations in two chunks, whereas the second thread will execute 40% in two chunks. However, independently from the cost associated with re-scheduling chunks on threads, 60% of the iterations containing much more than 50% of the total cost of the loop. The issue with dynamic fixed chunk scheduling is that the synchronization overhead becomes large when the number of threads increases - refer to Figure 1(d). Indeed, while performance of dynamic fixed chunk scheduling is comparable with that of `nuStatic` on two threads - giving the appearance that dynamic scheduling could cope with all the cases `nuStatic` can, `nuStatic` significantly outperforms dynamic on four threads.

Overall, Figure 1(c) shows that `nuStatic` performs as well as `Trapezoid` and outperforms other iteration schedulers. For different number of threads `nuStatic` determines the following `parts` of the iterations space: $\{30, 70\}$ when $p = 2$; $\{13, 16, 21, 50\}$ when $p = 4$ and $\{6, 7, 8, 9, 11, 15, 36\}$ when $p = 8$.

Finally, we present an analysis of the program `470.lbm` - Figure 2. We focus our attention on the performance of the parallel section in the function `LBM_performStreamCollide` - within which the program spends a significant percentage of its execution cycles ($> 90\%$). Our technique selects dynamic scheduling with a chunk size of 40, which is determined as follows. From the micro-benchmarking experiments, the minimum synchronization overhead is attained for 8 threads - this overhead is $\approx 0.12\mu s$. $3 \times 10^{-9} \times$ `chunk_size` $= 0.12 \times 10^{-6}$, which corresponds to a `chunk_size=40`. Such a chunk size represents a conservative choice for a lower number of threads. In the case of two threads, `Guided 40` is selected, whereas when the number of threads increases to four and eight, our technique switches to `Fixed chunk`, with a chunk size

equal to 40, and attains significant speedups over the baseline. Note that, in the case of `Fixed chunk`, a chunk size equal to 40 means that idle threads attempt to fetch 40 iterations at a time. In the case of `Guided`, a chunk size equal to 40 indicates the minimum number of iterations that can be fetched. Figure 2 shows performance improvements up to 65% for `470.lbm`. In Figure 2, the baseline is the sequential execution time. When $p = 2$, `Guided 40` is selected by our technique as it outperforms both `Dynamic 40` and `nuStatic` [5]. For $p > 2$, `Dynamic 40` is selected by our technique. For $p > 2$, `Dynamic 40` significantly outperforms the other techniques.

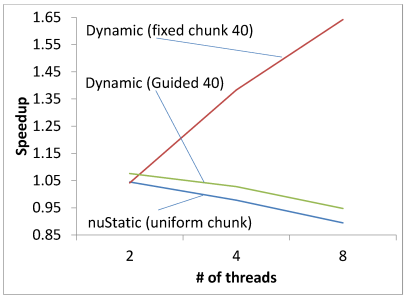


Fig. 2. Speedup `470.lbm`

3.3 Future work on the determination of the number of threads

The determination of the number of threads is also a fundamental point for improving the performance of a DO ALL loop and must be combined to the problem addressed in this paper. That is, given a number of threads, find a schedule that given the number of iterations, optimizes performance of the DO ALL loop. While the determination of the number of threads could be attempted using corollaries of Amdahl's law, e.g., [22], such corollaries do not account for the complex system/software interactions happening on real systems - an example is provided by the case of `470.lbm` explained above, where a combination of the number of threads, the scheduling technique and an appropriate selection of the chunk size are both necessary to attain a significant speedup. We plan to extend the technique proposed in this paper to select at run-time both the number of threads and the iteration schedule for speeding up the performance of parallel loops.

4 Related work

Parallel loops and in particular DOALL loops are pervasive in ordinary programs, e.g., $\approx 90\%$ [10] of the loops present in SPEC CPU2000[1] are parallel loops or can be restructured to expose such a parallelism [23, 24]. As implemented in the OpenMP pass of modern compilers, scheduling techniques for executing DOALL loops can be roughly divided in two categories: static - where the scheduler is responsible to assign chunks of iterations to the available cores; dynamic

[5] In this particular case, where the cost per iteration is constant, we remark that `nuStatic` is equivalent to classic static scheduling.

(fixed chunk or self-scheduling) - where idle cores first use synchronization (mutex/locks) to earn an exclusive access to the queue of iterations and second fetch a chunk of iteration to execute [4, 5, 7]. The time spent by idle threads to acquire/release the queue of iterations multiplied by the number of exclusive accesses and times the average number of cores attempting to fetch iterations concurrently constitutes the synchronization overhead for dynamic scheduling strategies. Such an overhead can either reduce significantly or annihilate the benefits of a parallel execution. Several dynamic scheduling techniques allow the cores to fetch chunks with a progressively small chunk size [4, 5, 7, 25] to reduce the overhead. Polychronopoulos and Kuck in [4] proposed `Guided`, a technique whose synchronization overhead is proportional to the number of cores times the natural logarithm of the number of iterations. Hummel and Flynn in [5] proposed `Factoring`, whose synchronization overhead is proportional to the natural logarithm of the number of iterations. Tzen and Ni in [7] propose `Trapezoid`, whose synchronization overhead is proportional to the number of cores. Yue and Liljia in [8] proposed the generation of chunking heuristics using genetic algorithms and show performance improvements against self-scheduling techniques. However, performance attained using dynamic and in particular self-scheduling techniques is shown to be negatively impacted by the presence of a relatively large standard deviation in the cost per iterations [8]. Under the assumption that iterations with large cost are distributed according to well known statistical distributions, e.g., normally, there exist dynamic scheduling techniques [6, 9] aimed to improve performance of parallel loops. The technique proposed in this paper addresses the case of variability of the cost per iteration independently from the distribution of such costs. This is especially important for such cases when iterations with large cost occur in bursts.

Kejariwal et al. in [10] proposed profile-based iteration space partitioning techniques. Such profile-based techniques are computationally expensive. Indeed, partitioning the iteration space requires storing the sequence of cost per iterations - which can be arbitrarily large depending on the input size, subsequently interpolating such a sequence and finally performing the partitioning using numerical integration. The technique proposed in this work is fairly simple to implement and execute at run-time. Furthermore, expressing the partitions in terms of percentages of the iteration space releases the partitions from the particular instance of a parallel loop. This is particularly useful in the case of parallel loops with a fixed geometry of the iteration space, such as the case of the kernel adjoint convolution. Indeed, once the number of threads is assigned, the partitioning found for a given input size can be applied for any input size and still provides optimal work sharing. Similarly to Just-in-Time compilation [26], our technique attempts to optimize code execution from past observations. Differently from such techniques, past observations are utilized to determine a schedule able to optimize performance of DOALL loops. Rauchwerger et al. [27] proposed a technique for finding an optimal schedule to execute partially parallel loops. That is, loops whose parallel execution requires synchronization to ensure the correct execution order of the iterations of the loop. Our technique is a run-time technique that focuses on speeding up the execution of DOALL loops.

While iterations of DOALL loops can be executed in any order and yet produce correct results, different schedules can attain significantly better performance than others on modern multi-core systems.

Thus, in this work, we acknowledge the importance of both static and dynamic scheduling techniques, and propose a new run-time technique that (a) accurately profiles the iteration space; (b) partitions the iteration space in non uniform chunks of iterations containing equal portions of the execution time; (c) attempts to schedule such chunks of iterations or to find the dynamic schedule that is more suitable for the particular instances of the loop on a given architecture.

5 Conclusion

We proposed a new scheduling technique to speedup DOALL loops in ordinary programs on modern complex multi-core systems. Our technique targets the execution of DOALL loops with variable cost per iterations and exhibiting *either a predictable or an unpredictable* behavior across multiple instances of the loop. In the former case our technique implements a quick run-time pass to determine chunks of iterations containing the same amount of work to cores, which is followed by a static assignment of such chunks to core. At run-time, the performance of such a static schedule is compared with the performance of both the sequential execution and the parallel execution using dynamic scheduling techniques - with a nearly optimal selection of the chunk size to optimize performance. The best scheduling technique is subsequently used for executing subsequent instances of the parallel loop.

We implemented our technique in GNU GCC OpenMP and show promising results on the linear algebra kernel adjoint convolution and the program `470.lbm` from SPEC CPU2006 - implementing the Lattice Boltzman Method - on the state-of-the-art 4, 2-way SMT cores Intel Sandy Bridge architecture. Specifically, our technique attains nearly optimal speedup for the adjoint convolution and up to 65% performance improvement for `470.lbm`.

Acknowledgments

This work was partially supported by the NSF grant number CCF-0811882 and the NSF Variability Expedition Grant number CCF-1029783.

References

1. John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *IEEE Computer*, 33(7):28–35, 2000.
2. J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
3. S. F. Lundstrom and G. H. Barnes. Advanced computer architecture. chapter A controllable MIMD architecture. IEEE Computer Society Press, Los Alamitos, CA, USA, 1986.
4. C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, 1987.
5. S. Hummel, E. Schonberg, and L. E. Flynn. Factoring: a method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, 1992.

6. S. Lucco. A dynamic scheduling technique for irregular parallel programs. pages 200–211, 1992.
7. T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Trans. Parallel Distrib. Syst.*, 4(1):87–98, 1993.
8. K. K. Yue and D. J. Lilja. Parameter estimation for a generalized parallel loop scheduling algorithm. In *HICSS*, pages 187–, 1995.
9. D. J. Hancock, R. W. Ford, T. L. Freeman, and J. M. Bull. An investigation of feedback guided dynamic scheduling of nested loops. In *Proceedings of the International Workshop on Parallel Processing*, 2000.
10. A. Kejariwal, A. Nicolau, U. Banerjee, A. V. Veidenbaum, and C. D. Polychronopoulos. Cache-aware partitioning of multi-dimensional iteration spaces. In *Proceedings of SYSTOR*, 2009.
11. S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4), 2009.
12. Openmp. http://www.openmp.org.
13. Gnu gcc v4.6. http://gcc.gnu.org/gcc-4.6/.
14. Intel compilers. http://software.intel.com/en-us/articles/intel-compilers/.
15. V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In *Proceedings of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, 2001.
16. Y. Zhang and M. Voss. Runtime empirical selection of loop schedulers on hyperthreaded smps. In *19th International Parallel and Distributed Processing Symposium*, 2005.
17. J. M. Bull and D. O'Neill. A microbenchmark suite for openmp 2.0. *SIGARCH Comput. Archit. News*, 29:41–48, December 2001.
18. D. Novillo. Openmp and automatic parallelization in gcc. In *GCC Developers Summit*, 2006.
19. P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.
20. B. W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.
21. T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rüde. Optimization and profiling of the cache performance of parallel lattice boltzmann codes. *Parallel Processing Letters*, 13(4), 2003.
22. H. P. Flatt and K. Kennedy. Performance of parallel processors. *Parallel Computing*, 12(1):1 – 20, 1989.
23. L. Lamport. The hyperplane method for an array computer. In *Sagamore Computer Conference*, 1974.
24. U. Banerjee. *Loop transformations for restructuring compilers - the foundations*. Kluwer, 1993.
25. C. P. Kruskal and Alan Weiss. Allocating independent subtasks on parallel processors. *IEEE Trans. Softw. Eng.*, 11(10), 1985.
26. J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003.
27. L. Rauchwerger, N. M. Amato, and D. A. Padua. A scalable method for run-time loop parallelization. *International Journal of Parallel Programming*, 23(6), 1995.