MESETA: A new scheduling strategy for speculative parallelization of randomized incremental algorithms

Diego R. Llanos*
Departamento de Informática
Universidad de Valladolid, Spain
diego@infor.uva.es

David Orden†
Departamento de Matemáticas
Universidad de Alcalá, Spain
david.orden@uah.es

Belén Palop‡
Departamento de Informática
Universidad de Valladolid, Spain
b.palop@infor.uva.es

Abstract

In this work we address the problem of scheduling loops with dependences in the context of speculative parallelization. We show that scheduling alternatives are highly influenced by the dependence violation pattern presented in the code. We center our analysis in those algorithms where dependences are less likely to appear as the execution proceeds, like incremental randomized algorithms. These algorithms are, in general, hard to parallelize by hand, and represent a challenge for any automatic parallelization scheme. Our analysis led us to the development of MESETA, *a new scheduling strategy that takes into account the probability of a dependence violation to determine the number of iterations being scheduled.* MESETA *is compared, among others, with Fixed-Size Chunking (FSC), the only scheduling alternative used so far in the context of speculative parallelization. Our experimental results show a 3% to 22% speedup improvement over FSC for the same incremental randomized algorithm.*

1. Introduction

Speculative parallelization (also called *thread-level speculation*) is the most promising technique for extracting parallelism of irregular loops. With speculative parallelization, loops that can not be analyzed at compile time are optimistically executed in parallel. Hardware or software mechanisms ensure that all threads access the shared data according to sequential semantics. A *dependence violation* appears when one thread incorrectly consumes a datum that has not been generated yet by a predecessor. In the presence of such a violation, earlier software-only speculative solutions (see, e.g. [12, 17]) interrupt the speculative execution and re-execute the loop serially. More recent approaches [2, 6, 20] squash only the offender thread and its successors, re-starting them with the correct data values.

It is easy to see that frequent squashes adversely affect speculation performance. One way to reduce the cost of a squash is to assign smaller subsets (called *chunks*) of iterations to each thread, reducing the amount of work being discarded in the case of a squash. Unfortunately, smaller chunks also imply more frequent commit operations and a higher scheduling overhead.

The problem of scheduling iterations of parallel loops among different processors in a parallel system has been extensively studied in the literature [9, 10, 15, 22, 23]. However, the proposed solutions only deal with independent iterations and their basic concern is to achieve a good load balancing among processors. Therefore, classic scheduling alternatives are not useful for speculative parallelization. To the best of our knowledge, the only scheduling mechanism used so far in this context is Fixed-Size Chunking (FSC), that schedules chunks of equal number of iterations among processors. This mechanism does not take into account the dependence distribution of the loop to be parallelized.

In this work we study in detail the problem of scheduling loops with dependences in the context of speculative parallelization. We first show that the scheduling alternatives are highly influenced by the dependence violation pattern presented by the code. Then, we propose a new scheduling alternative, MESETA, for those algorithms where dependences are less likely to appear as the execution proceeds. Many incremental algorithms follow this pattern and, among them, *incremental randomized algorithms* have been very well studied and proved to achieve the best performance. They are, in general, hard to parallelize by hand and a challenge for any automatic parallelization scheme. This justifies their choice to test the efficiency of MESETA.

The results obtained using a software-only speculative engine [2], show that MESETA allows a 3% to 22% speedup improvement over the use of FSC for the same incremental

* Partially supported by RII3-CT-2003-506079.
† Partially supported by MCYT BFM2001-1153.
‡ Partially supported by MCYT TIC2003-08933-C02-01.

randomized algorithm, reducing at the same time the cost associated to the squash and re-execution of chunks of iterations.

2. Scheduling alternatives for parallel loops

The problem of scheduling iterations of irregular loops in order to assign them to different processors has been extensively studied in the literature. All existing proposals assume that there are no dependences among iterations, and therefore all the iterations can be executed in parallel in any order. We review in this section some of the solutions that have been proposed in the last years to this problem.

Let N be the total number of iterations, and P the total number of threads (equal to the number of processors in the system). The two simplest scheduling techniques, that will not be considered further due to their poor performance, are the following. *Static scheduling*, divides the iteration space statically into N/P chunks of equal size. This technique does not allow to balance dynamically the workload during the execution of the loops. Hence, the processors may finish at very different times, leading to a poor load balance. On the other hand, *self scheduling* [22] assigns to each thread the next iteration to be executed. This approach minimizes load unbalance, but at the cost of a high increase of the scheduling overhead.

Between these two extreme solutions different alternatives have been proposed. A brief description follows.

Fixed-size chunking (FSC): In this approach, proposed by Kruskal and Weiss [10], the iteration space is statically divided into chunks of equal size. Each free thread executes the following chunk. This solution reduces synchronization overhead in comparison with self scheduling, with a better load balancing than the static scheduling. The efficiency of this scheme depends on the choice of an appropriate value for the chunk size, K, a difficult task for both programmers and compilers. Kruskal and Weiss give the following formula for the optimal value of the chunk size, K_{opt}:

$$K_{\text{opt}} = \left(\frac{\sqrt{2}Nh}{\sigma P \sqrt{\log P}} \right)^{2/3},$$

where σ is the variance of the iteration time, h the scheduling overhead, N the number of iterations and P the number of processors. The first three values are unknown at the beginning of the loop, making it difficult to determine the optimal (or at least adequate) chunk size in practice.

Guided self-scheduling (GSS): This technique, proposed by Polychronopoulos and Kuck [15], addresses the problem of uneven start times for each processor. Instead of using a fixed chunk size, they propose decreasing chunk sizes, calculated as a decreasing function of the current iteration number i being executed. As execution proceeds, smaller chunks improve the balance of the workload toward the end of the loop. Let R_i be the remaining iterations at step i. Each chunk size, K_i, is calculated as follows:

$$R_0 = N; \quad K_i = \left\lceil \frac{R_i}{xP} \right\rceil; \quad R_{i+1} = R_i - K_i;$$

where x should be fixed to adjust the amount of work scheduled in each step. In order to avoid having many small chunks by the end of the loop, an additional function GSS(K) is proposed to bound the chunk size from below by K, specified either by the compiler or the programmer.

Factoring: This mechanism, proposed by Hummel et al. [9], is similar in concept to GSS, but the allocation of iterations to processors proceeds in phases. In each phase, a part of the remaining iterations is divided in batches of P equal-size chunks. The optimal number of iterations per batch requires the (*a priori* unknown) mean iteration time μ and, again, the variance σ. The authors argued that, for many common distributions of chunk execution times, no more than half of the remaining iterations should be assigned to each batch. The following equation gives the chunk size K_j for each component of the i-th batch group.

$$R_0 = N; \quad K_j = \left\lceil \frac{R_i}{xP} \right\rceil = K, \; \forall j \in \{1, \ldots, P\};$$

$$R_{i+1} = R_i - P\,K.$$

By setting $x = 2$, half of the remaining iterations are scheduled in each phase.

Trapezoidal scheduling (TSS): This technique, proposed by Tzen and Ni [23], uses chunks that decrease in size linearly. Trapezoidal scheduling is defined as TSS(f, ℓ, N), where f is the size of the first chunk and ℓ the size of the last one. The maximum value for f is N/P: with this value, the first P chunks will hold the first $3/4$ iterations. A more conservative value is to set $f = N/(2P)$, when only the first $7/16$ iterations will be executed in the first P chunks.

The area below TSS(f, ℓ, N), called A, should be calculated in order to obtain the decreasing step, δ:

$$A = \left\lceil \frac{2N}{f + \ell} \right\rceil; \quad \delta = \frac{f - \ell}{A - 1}$$

Again, the values for f and ℓ depend on the execution time, and no heuristics are provided to calculate them. Instead, conservative values $f = N/2P$ and $\ell = 1$ are suggested.

The total number of iterations being scheduled is, at least, N for all scheduling alternatives described. Only Self

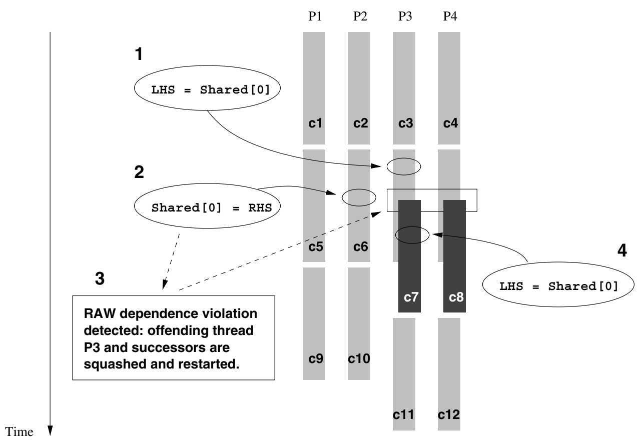

P1
P2
P3
P4
1
LHS = Shared[0]
c1
c2
c3
c4
2
Shared[0] = RHS
c5
c6
4
LHS = Shared[0]
3
RAW dependence violation detected: offending thread P3 and successors are squashed and restarted.
c7
c8
c9
c10
c11
c12
Time

Figure 1. Example of RAW dependence violation. $P1$ to $P4$ are four different processors executing chunks of consecutive iterations, labeled $c1$ to $c12$.

Scheduling always leads to exact correspondence. Consequently, the scheduler should always check whether the upper limit will be exceeded, and order the execution of only the remaining iterations.

Finally, other proposals determine the optimum chunk size at runtime, based on the total available parallelism, the optimal grain size and the statistical variance of execution times for individual tasks. We do not consider adaptive scheduling policies in this work, such as the Tapering algorithm by Lucco [11].

3. Scheduling with speculative parallelization

In the context of speculative parallelization, however, loops may present dependences among iterations. A *Read-after-write* (RAW) dependence violation appears when a thread speculatively reads a value and later a predecessor modifies the same value. If a dependence violation occurs during the parallel execution of the loop, the offending thread and all its successors are squashed and re-started with the correct values. Figure 1 shows the events involved in a RAW dependence violation, where a thread modifies a value that a successor has already consumed. Therefore, the scheduling alternatives described in Section 2 can not be directly applied to speculative parallelization, since they are designed to achieve load balancing and low scheduling overhead on loops composed of independent iterations.

We will now develop a simple model to compute an upper bound of the squash overhead. Suppose that some loop is divided into C equal size chunks and that we have P processors where we can schedule in batch P chunks at a time. The number of batches is then $B = \lceil C/P \rceil$. Let us call t_s the time it takes to execute the loop sequentially, t_c the time to complete a chunk and t_b the time to execute a batch. We assume that the overhead time, t_o, measured as the time it takes to assign each chunk to some processor plus the time to commit or squash the results on the main copy of the shared variables, is similar in all batches. If we assume that all iterations take equal time, then $t_b = t_c = t_s/C + t_o$.

We will now calculate an upper bound for the time needed to complete the loop in parallel, t_p, when only a single squash arises and then extend this formula for several squashes.

We can decompose t_p into two parts: the time it would take to execute the loop if there were no squashes, plus the time it takes to re-execute the work that was already done when the squash was produced. In the worst case, the squash will be produced in the last iteration of some chunk. Therefore, all the work done by later threads in this batch will be re-executed from the beginning and

$$t_p \leq Bt_b + t_b.$$

It is easy to see that each squash costs, at most, t_b, and that dependences between iterations in the same chunk do not lead to squashes. Therefore, the number of squashes, N_s, is in general, smaller than the number of dependences. Hence, now

$$t_p \leq Bt_b + N_s t_b$$

and since $t_b = t_s/C + t_o$ and $B = C/P$, we have

$$t_p \leq t_s/P + C/P\, t_o + N_s(t_s/C + t_o). \tag{1}$$

The term $N_s(t_s/C + t_o)$ indicates that the greater the number of chunks (thus, using smaller chunks), the smaller the time lost on each squash. On the other hand, the term $C/P\ t_o$ indicates that we can obtain better time bounds if we minimize the number of chunks, that is, making chunks bigger.

If we decide to use a single chunk size during the entire execution of the loop, the optimal number of chunks we should use can be determined minimizing t_p deriving as follows:

$$\frac{\partial(t_s/P + C/P\, t_o + N_s(t_s/C + t_o))}{\partial C} = 0$$

and we obtain

$$C^* = \sqrt{\frac{N_s t_s P}{t_o}}$$

Therefore, the optimal number of chunks C^* depends on the number of squashes N_s. But changing the number of chunks varies the number of dependences leading to squashes. That is, only one chunk executing the whole loop would make $N_s = 0$, and chunks of size equal to one iteration would produce one squash for each dependence.

The analysis of Eq. (1) suggests that a useful strategy for speculative execution is to use big chunks on portions of the loop where less dependences are expected to be found, and smaller chunks on portions where we expect to find many dependences.

Obviously, it is not a simple task to characterize the distribution of dependences inside a general loop. In the next section we will study incremental randomized algorithms and characterize their dependence pattern.

4. Incremental randomized algorithms

Incremental randomized algorithms have been deeply studied in areas such as Computational Geometry and Optimization [13, 14, 16]. They have led to simple, easy-to-code and efficient algorithms for a variety of problems.

In their general formulation, the input of an incremental randomized algorithm is a set of elements (not necessarily points), for which a certain output needs to be computed. The algorithm proceeds incrementally by adding the input elements one by one and obtaining the intermediate results. The main feature is that the elements are added in random order, determined by the choice of a random permutation at the beginning.

Our main concern is that, independently of the problem, these algorithms are shown to present a common dependence pattern: At the beginning of the execution many iterations depend on values calculated by previous ones. However, as the execution proceeds, fewer and fewer dependences arise between different iterations. This behavior makes possible (and attractive) to develop scheduling strategies for the speculative parallelization of this type of algorithms, as well as all algorithms sharing their dependence pattern, and motivates the proposal of MESETA in Section 5.

4.1. Expected number of dependences

Let S and $\phi(S)$ represent respectively the input and the output of an incremental randomized algorithm. These algorithms start by choosing a random permutation $\{s_1, \ldots, s_n\}$ of the elements in S and then incrementally construct $\phi(R_i)$ for $R_i := \{s_1, \ldots, s_i\}$.

In order to study the expected number of dependences appearing at a given step i, let us introduce the following two notions: We call *violators* those elements of S not processed yet that would cause the present output $\phi(R_i)$ to be updated, that is, the elements leading to a potential RAW dependence. The *extreme* elements will be the ones needed to define the present output $\phi(R_i)$.

More formally:

$$V(R_i) := \{s \in S \setminus R_i : \phi(R_i \cup \{s\}) \neq \phi(R_i)\}$$

is the set of violators of R_i in S, and

$$X(R_i) := \{s \in R_i : \phi(R_i \setminus \{s\}) \neq \phi(R_i)\}$$

is the set of extreme elements of R_i.

For a couple of examples consider the computation of the Convex Hull [5, 21] and the Smallest Enclosing Circle [24]. The Convex Hull and the Smallest Enclosing Circle problems consist, respectively, in obtaining the smallest convex polygon and the smallest circle that enclose a set of points in the plane. Let us choose S to be a subset of points in $\mathbb{R}^2$. Considering $\phi \equiv$Convex Hull, the violators $V(R_i)$ are those points outside $\texttt{ConvexHull}(R_i)$, while the extreme points $X(R_i)$ are the vertices of this convex hull. For $\phi \equiv$Smallest Enclosing Circle, the points outside the smallest enclosing circle of R_i are the violators $V(R_i)$, and the extreme points $X(R_i)$ are the points defining this circle.

As observed above, the number $|V(R_i)|$ of violators equals the number of potential RAW dependences at step i. The Sampling Lemma, whose proof can be found in [7], states a relationship between the expected number of extreme elements found before processing element i and the expected number of violators to be found afterwards. At any step i the probability of causing a potential RAW dependences is $\frac{x_{i+1}}{i+1}$. Incremental randomized algorithms are used when x_i is asymptotically smaller than i and hence this probability decreases as execution proceeds.

4.2. Incremental randomized construction of a Convex Hull

For the randomized construction of the 2-dimensional Convex Hull, the usual input data sets are uniform distributions of points in a k-gon (see [8]). We will not consider degenerate cases, like i cocircular points, in which every iteration is dependent on the previous ones and the problem is inherently non-parallel. It is well known, see [18, 19], that in order to define the convex hull of i points uniformly distributed in a k-gon, only $x_i = O(k \log(i))$ of them are needed, while only $x_i = O(\sqrt[3]{i})$ are needed for the limit situation of points uniformly distributed in a disc.

We have performed several sequential executions on small sets of points in order to accurately determine the constants involved, which tend to be 2.60 for the square and 3.34 for the disc. Hence, the Sampling Lemma for incremental randomized algorithms claims that the probabilities $2.60 \log(i)/i$ (square) and $3.34\sqrt[3]{i}/i$ (disc) of causing a potential RAW dependence are much higher at the first iterations.

Two sample executions for square- and disc-shaped input sets have been made in order to check this result. We have used constant size blocks of $1\,000$ iterations of the main loop with the best incremental randomized algorithm for computing 2-dimensional Convex Hulls, due to Clarkson et al. [4]. As expected, only a portion of the potential RAW dependences turn into real RAW dependences, since some of them appear inside each chunk and will be executed sequentially. Figure 2 shows the effective distribution of RAW dependences for both input sets. The situation can be considered *stable* when the probability ϵ of finding a potential RAW dependence is close enough to 0. Also as expected, this happens earlier in the square than in the disc.

5. MESETA: Scheduling strategy for incremental randomized algorithms

We have seen in the previous section how the number of dependences appearing in an incremental randomized algorithm tends to decrease as the algorithm proceeds. This is why it is rather difficult to find a fixed block size that minimizes the number of squashes. Moreover, the rest of scheduling alternatives proposed lead to poor performance, since they schedule bigger chunk sizes at the beginning of the loop, precisely where we have proved that potential RAW dependences are more likely to be found.

We propose here a new scheduling strategy for the speculative execution of those algorithms in which dependences are less likely to appear as execution proceeds, like incremental randomized algorithms. MESETA (Spanish word for tableland) divides the execution of the loop into three parts (see Fig. 3): At the beginning of the loop, many dependences are likely to be found. We propose to assign small chunks to processors in that part of the execution, progressively increasing their size as the execution proceeds. The benefits are twofold. First, we are preventing dependences between distant iterations to appear since they will not be processed in parallel (and therefore many potential RAW dependences will not turn into real RAW dependences). Second, the amount of work to be redone after each squash is smaller, since we are scheduling small chunks. Except for FSC, any of the scheduling functions reviewed in Section 2 can be mirrored with respect to the y-axis and applied here in order to obtain increasing size chunks.

The probability of finding dependences between iterations will lower as the execution proceeds, reaching some ϵ where the situation can be considered stable. At this point we can use a fixed chunk size to minimize both squash and overhead costs.

Finally, in the last part of the loop we can safely assume that the number of dependences can be neglected. At this point, our main concern is load balancing. To achieve this goal, any of the techniques proposed in Section 2 can again be applied.

Some decisions still have to be made in order to show that MESETA improves the performance of the fixed-block-size basic technique.

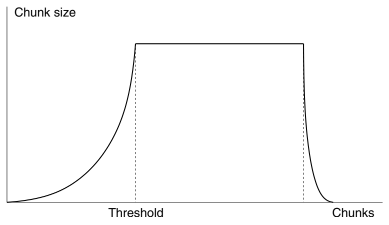


Figure 3. Distribution of chunk sizes in MESETA.

The first problem is how to determine the number of iterations to be executed before the probability of finding dependences is considered low enough. In Section 4.1 we have seen that incremental randomized algorithms follow the Sampling Lemma. Hence, for them, the probability of having a potential RAW dependence can be computed for any value of i. Once a threshold ϵ is chosen, it is easy to derive from the formula the iteration i in which that ϵ is achieved. Even though this threshold can be as close to 0 as the implementer's optimism allows, our experimental results show that conservative enough iteration numbers are obtained in most cases with $\epsilon = 3 \cdot 10^{-4}$.

Let us illustrate this by considering again the Convex Hull problem. For the case of a randomly-distributed square-shaped input set, setting

$$2.60 \log(i)/i = 3 \cdot 10^{-4} \Longrightarrow i \approx 10^5,$$

while for a randomly-distributed disc-shaped input set,

$$3.34 \sqrt[3]{i}/i = 3 \cdot 10^{-4} \Longrightarrow i \approx 10^6.$$

The second important decision is to fix the chunk size for the stable part of the loop. This decision will be postponed to the next section since, unfortunately, it does not only depend on the dependence pattern but also on the overheads produced by squashed threads.

We reach finally the descending part of the MESETA. We can now simply rely on known scheduling solutions presented in Section 2, since squashes are less and less likely to happen and we only have to care about load balancing. For the different alternatives, we will apply the parameters proposed by the authors in each case, starting at the height of the tableland. This will determine the iteration number in which the descending part of MESETA starts.

6. Experimental results

A state-of-the-art, software-only speculative parallelization engine [2] was used to execute in parallel the Clarkson's incremental randomized algorithm for the Convex Hull problem [4]. We executed in parallel the outer loop, that accounts for 100% of the algorithm sequential execution time.

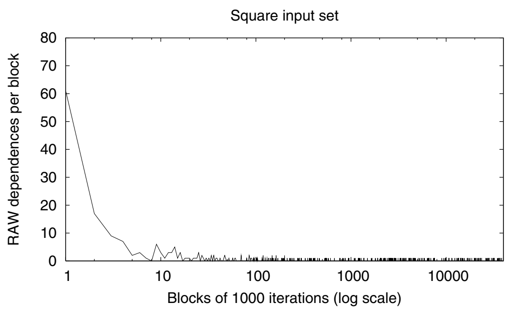


Figure 2. Effective RAW dependences in Clarkson's algorithm for the 2-D Convex Hull (log scales).

The number of violations between executions is bounded by the number of points lying outside the convex hull computed up to their insertion. Thus, we have used three different standard input sets: one composed by 40 million points randomly distributed inside a square, and another two with 10 and 40 million random points inside a disc. The input sets have been generated using the random points generator of CGAL 2.4 [1] and have been randomly ordered using its `shuffle` function. The expected performance for these sets is the same as for any other with the same shape and size. The following table resumes the characteristics of the algorithm and the input sets used.

Input description	Spec data size in KB	Iterations per invocation	% of violations
Square set, 40M points	15	39,999,997	0.64
Disc set, 10M points	86	9,999,997	15.48
Disc set, 40M points	137	39,999,997	7.35

The experiments performed were done on a Sun Fire 15K symmetric multiprocessor (SMP), equipped with 900MHz UltraSparc-III processors. The system runs SunOS 5.8. The application was compiled with the Forte Developer 7 Fortran 95 compiler using the highest optimization settings for our execution environment. Times shown in the following sections represent the time spent in the execution of the main loop of the application. The time needed to read the input set and the time needed to output the convex hull have not been taken into account. The application had exclusive use of the processors during the entire execution and we use wall-clock time in our measurements.

6.1. Comparing different MESETA shapes

We will now measure the performance of our scheduling proposal for incremental randomized algorithms with respect to existent solutions. Recall that we divide the scheduling profile into three parts, scheduling increasing chunk sizes at the beginning (to avoid dependence violations), a fixed block size for the stable part of the loop, and decreasing chunk sizes at the end in order to achieve a good load balancing.

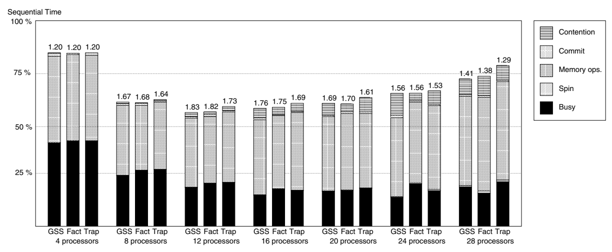


Figure 4. Execution time breakdown using different versions of MESETA to process the disc-shaped input set with 10 million points.

Three different scheduling functions will be used to distribute iterations in both the beginning and end of the speculative execution: GSS, Factoring, and Trapezoidal scheduling. We compare them in the execution of the Convex Hull algorithm for a disc-shaped, 10 million points input set. For this input set, almost 16% of the iterations lead to dependences. Therefore, a correct choice of the scheduling mechanism is very important. Figure 4 shows their execution time breakdown. As can be seen from the figure, the relative performance is similar for all three mechanisms, with a slight slowdown for the Trapezoidal version of the MESETA scheduling. The execution time breakdown shows that most of the time is consumed by speculative memory operations, and that contention and commit times are not significant for this problem (see [2] for more details on the behavior of the speculative engine).

To better compare these scheduling functions, Fig. 5 shows the number of dependence violations triggered by each one of our scheduling strategy versions, together with the number of corresponding squashed threads. We can conclude that the behavior of both GSS and Factoring versions is very similar, not only in terms of speedup, but also in terms of squashes and violations. The Trapezoidal version, on the other hand, leads to a bigger number of dependence violations and squashes. This cost might have been also reflected in the speedup results. However, the speedup results are not as bad as we might expect, because Trapezoidal scheduling divides the iteration space to be scheduled in fewer blocks than the other alternatives. Therefore, the scheduling and commit costs are smaller, mitigating the slowdown produced by a higher number of squashes.

6.2. Performance evaluation of MESETA

The last part of our study is to compare the performance of MESETA with respect to Fixed-Size Chunking (the only scheduling mechanism used so far in the field of speculative parallelization) and GSS (a mechanism widely used in the scheduling of loops with no dependences). Fixed-Size Chunking will be used with the chunk size that leads to the maximum speedup for this particular problem [3]: 1 024 iterations for the disc and 4 096 for the square. GSS will be used with $x = 1$. The version of MESETA that will be considered is the one that uses GSS for both the increasing and decreasing part of the loop execution: As we saw above, this function leads to slightly better speedups than the other alternatives. The optimum block size for the stable part of the loop has been experimentally obtained [3], turning out to be around 2 500 for the disc and 5 000 for the square, independently of the number of processors.

Figure 6 shows the relative speedup of both approaches in the execution of the Convex Hull for a 40-million-points input set, both disc- and square-shaped. From the figure we can draw the following observations: First, MESETA overcomes the Fixed-Size Chunking approach in all cases, with a performance gain of 12% to 22% for the disc-shaped input set, and 3% to 11% for the square-shaped input set, with the exception of the four-processors run in the latter, where the speedups are equal for both systems. The gain is proportionally higher for the disc because, as expected, this input set generates much more dependences than the square-shaped one, and therefore benefits more from our scheduling strategy.

Second, as we saw in Section 3, GSS is not suitable for speculative parallelization of loops with dependences, and this observation is confirmed by the experimental results. The same considerations can apply to TSS and Factoring as well.

Finally, no performance loss due to the higher scheduling cost of MESETA in comparison with Fixed-Size Chunking has been observed in any experiment.

7. Conclusions

In this work we study in detail the problem of scheduling loops with dependences in the context of speculative parallelization. We show that the scheduling alternatives are highly influenced by the dependence violation pattern presented by the code. We center our analysis in those algorithms where dependences are less likely to appear as the execution proceeds, like incremental randomized algorithms. We propose MESETA, a new scheduling strategy that schedules variable-size chunks of iterations according to the probability of a dependence violation for each part of the loop. Our results show a 3% to 22% speedup improvement of MESETA over Fixed-Size Chunking for the same incremental randomized algorithm, leading to a better extraction of its inherent parallelism.

Acknowledgments

Part of this work was carried out while David Orden visited the Departamento de Informática, Universidad de Valladolid, with support of the Universidad de Alcalá. Diego R. Llanos and Belén Palop would like to thank Manuel Abellanas and the Departamento de Matemática Aplicada, Universidad Politécnica de Madrid, where part of this research was performed. The authors would also like to thank the EPCC (Edinburgh Parallel Computing Center) for the main computing resources used in this work and its support staff, in particular, Chris Johnson and Catherine Inglis.

References

[1] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org/.

[2] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2003.

[3] M. Cintra, D. R. Llanos, and B. Palop. Speculative parallelization of a randomized incremental convex hull algorithm. In *Proc. of the Comput. Geom. and Applications (CGA), LNCS 3045*, pages 188–197, May 2004.

[4] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comput. Geom. Theory Appl.*, 3(4):185–212, 1993.

[5] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete and Computational Geometry*, 4(1):387–421, 1989.

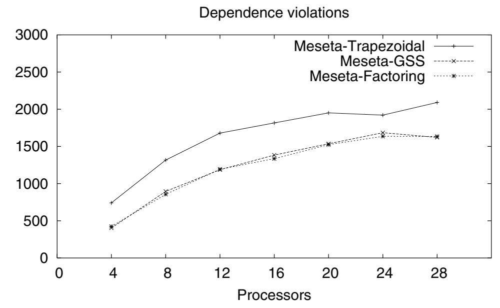
[6] F. Dang, H. Yu, and L. Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Proc. of the 16th International Parallel and Distributed Processing Symposium (IPDPS '02)*, April 2002.

[7] B. Gärtner and E. Welzl. A simple sampling lemma: Analysis and applications in geometric optimization. *Discrete and Computational Geometry*, 25(4):569–590, 2001.

[8] S. Har-Peled. On the expected complexity of random convex hulls. Technical Report 330, School of Mathematical Sciences, Tel-Aviv University, 1998.

[9] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Communications of the ACM*, 35(2):90–100, August 1992.

[10] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, SE-11(10):1001–1016, 1990.



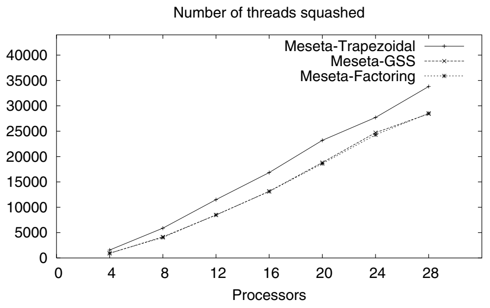
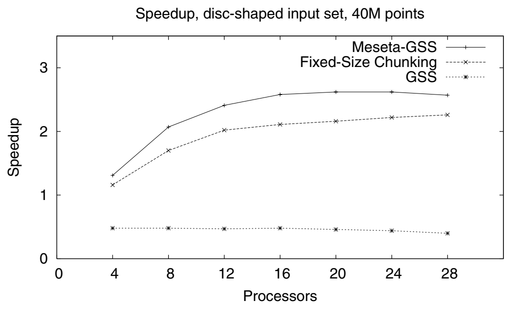


Figure 5. Dependence violations and number of threads squashed during the execution of the disc-shaped input set with 10 million points.



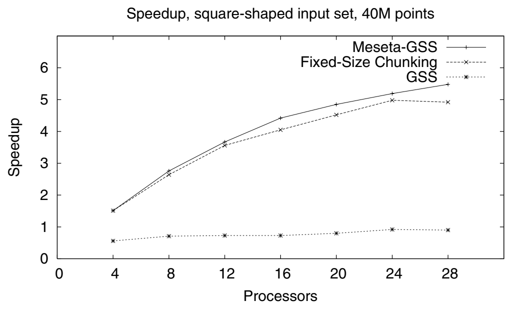


Figure 6. Speedups during the execution of the Convex Hull with different number of processors.

[11] S. Lucco. A dynamic scheduling method for irregular parallel programs. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 200–211. ACM Press, 1992.

[12] M. Gupta and R. Nim. Techniques for run-time parallelization of loops. *Supercomputing*, November 1998.

[13] K. Mulmuley. Randomized algorithms in Computational Geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 16, pages 703–724. North-Holland Publishing Co., 2000.

[14] K. Mulmuley and O. Schwarzkopf. Randomized algorithms. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 34, pages 633–652. CRC Press, New York, 1997.

[15] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[16] S. Rajasekaran, P. M. Paradalos, J. H. Reif, and J. D. Rolim, editors. *Handbook of Randomized Computing: Volumes I and II*, volume 9 of *Combinatorial Optimization*. Kluwer Academic Publishers, 2001.

[17] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Conf. on Programming Languages Design and Implementation*, pages 218–232, June 1995.

[18] H. Raynaud. Sur l'enveloppe convexe des nuages de points aléatoires dans $\mathbb{R}^n$. *Journal of Applied Probability*, 7:35–48, 1970.

[19] A. Renyi and R. Sulanke. Über die konvexe hülle von n zufällig gerwähten punkten II. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, 3:138–147, 1964.

[20] P. Rundberg and P. Stenström. Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors. In *Workshop on Scalable Shared Memory Multiprocessors*, June 2000.

[21] R. Seidel. Linear programming and convex hulls made easy. In *Proceedings of the 6th Annual ACM Symposium on Computational Geometry*, pages 211–215, 1990.

[22] P. Tang and P.-C. Yew. Processor self-scheduling for multiple nested parallel loops. In *IEEE Intl. Conf. on Parallel Processing*, pages 528–535, August 1986.

[23] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A pratical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, 1993.

[24] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and new Trends in Computer Science*, number 555 in LNCS, pages 359–370. Springer, 1991.