New Scheduling Strategies for Randomized Incremental Algorithms in the Context of Speculative Parallelization

Diego R. Llanos, *Member*, *IEEE*, David Orden, and Belén Palop

Abstract—In this work, we address 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 the code presents. We center our analysis in those algorithms where dependences are less likely to appear as the execution proceeds. Particularly, we focus on randomized incremental algorithms, widely used as a much more efficient solution to many problems than their deterministic counterparts. These important 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 with existing techniques, including Fixed-Size Chunking (FSC), the only scheduling alternative used so far in the context of speculative parallelization. Our experimental results show a 5.5 percent to 36.25 percent speedup improvement over FSC, leading to a better extraction of the parallelism inherent to randomized incremental algorithms. Moreover, when the cost of dependence violations is too high to obtain speedups, MESETA curves the performance degradation.

Index Terms—Parallelism and concurrency, load balancing and task assignment, scheduling and task partitioning, geometrical problems and computations.

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 cannot 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, for example, [1], [2]) interrupt the speculative execution and reexecute the loop serially. More recent approaches [3], [4], [5] squash only the offender thread and its successors, restarting 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.

- *D.R. Llanos and B. Palop are with the Departamento de Informática, Edif. Tecn. de la Información, Universidad de Valladolid, Campus Miguel Delibes, 47011 Valladolid, Spain. E-mail: {diego, b.palop}@infor.uva.es.*
- *D. Orden is with the Departamento de Matemáticas, Facultad de Ciencias, Universidad de Alcalá, Apdo. de Correos 20, E-28871 Alcalá de Henares (Madrid), Spain. E-mail: david.orden@uah.es.*

Manuscript received 20 Feb. 2006; revised 22 Oct. 2006; accepted 29 Nov. 2006; published online 15 Feb. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0062-0206.
Digital Object Identifier no. 10.1109/TC.2007.1030.

The problem of scheduling iterations of parallel loops among different processors in a parallel system has been extensively studied in the literature during the last 20 years [6], [7], [8], [9], [10]. However, the proposed solutions only deal with independent iterations and their basic concern is to achieve 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), which schedules chunks of an 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 [11], for those algorithms where dependences are less likely to appear as the execution proceeds. Many incremental algorithms follow this pattern and, among them, *randomized incremental algorithms* have been very well studied and proven to achieve the best performance. Randomized incremental algorithms have been deeply studied in areas such as computational geometry and optimization [12], [13], [14], leading to simple, easy-to-code, and efficient algorithms for a variety of problems, including line segment intersection [15], [16], Voronoi diagrams [15], [17], [18], [19], triangulation of simple polygons [20], solving linear programs [21], and many others. Randomized incremental algorithms are, in general, hard to parallelize by hand and a challenge for

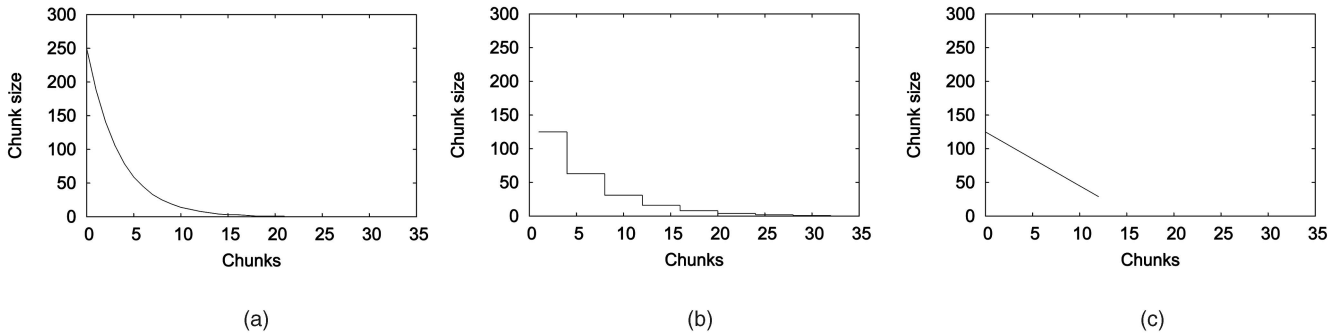


Fig. 1. Variable-sized scheduling alternatives for 1,000 iterations and four processors. (a) GSS with $x = 1$. (b) Factoring with $x = 2$. (c) TSS with $f = 125$, $\ell = 1$.

any automatic parallelization scheme. This justifies their choice to test the efficiency of MESETA.

The results obtained using a software-only speculative engine [3], [22] show that MESETA allows a 5.5 percent to 36.25 percent speedup improvement over the use of FSC for the same randomized incremental algorithm, reducing, at the same time, the cost of the reexecution of chunks of iterations.

The paper is organized as follows: Section 2 reviews some classic scheduling alternatives for loops with no dependences. Section 3 shows how the speculative execution of a loop with dependences differs from the execution of a parallel loop, introducing a cost model to calculate an upper bound of the extra time required by speculative parallelization. Section 4 introduces randomized incremental algorithms, giving some examples and studying how dependences are distributed in the iteration space. Section 5 uses this information to propose MESETA, the new scheduling strategy that allows an efficient speculative execution of this type of algorithms. Section 6 shows the experimental results and Section 7 concludes the paper.

2 Review of 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 of the iterations can be executed in parallel in any order. We review in this section some of the solutions that have been proposed in recent years to this problem.

We will first describe the two simplest techniques to distribute iterations among processors. 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 first one, called *static scheduling*, divides the iteration space statically into N/P chunks of equal size. This system does not allow us to dynamically balance 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* [9] assigns to each thread the next iteration to be executed. This approach minimizes load imbalance, but at the cost of a high increase in the scheduling overhead.

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

Fixed-sized chunking (FSC). In this approach, proposed by Kruskal and Weiss [7], the iteration space is statically divided into chunks of fixed size. Each free thread executes the following chunk. This solution reduces synchronization overhead in comparison with self-scheduling, with better load balancing than 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}:

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

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 [8], 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. When $x = 1$, the GSS function allocates approximately two-thirds of the remaining iterations every P chunks. If the time spent by iterations is uneven, this may result in too large an amount of work scheduled in the first chunks, making it more difficult to balance the work at the end of the loop [6]. This effect can be reduced with bigger values of x.

In order to avoid having many small chunks by the end of the loop, an additional function, $\text{GSS}(K)$, is proposed to bound the chunk size from below by K, specified by either the compiler or the programmer. Fig. 1a shows an example of scheduling with GSS.

Factoring. This mechanism, proposed by Hummel et al. [6], 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 into 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 ith batch group:

$$R_0 = N,$$
$$K_j = \left\lceil \frac{R_i}{xP} \right\rceil = K,\ \forall j \in \{1, \ldots, P\},$$
$$R_{i+1} = R_i - P\ K.$$

By setting $x = 2$, half of the remaining iterations are scheduled in each phase. This setting smooths the scheduling function in comparison with GSS and therefore increases the probability of even finishing times. An additional feature of this solution is that, when all of the iterations take the same time, all of the processors execute an equal portion of the work, N/P. Again, the value of x can be adjusted to schedule more or fewer iterations in the first chunks.

Factoring can be viewed as a generalization of GSS and FSC: GSS is factoring where each batch contains a single chunk, while FSC is factoring with a single batch. Fig. 1b shows an example of scheduling with factoring.

Trapezoidal scheduling (TSS). This technique, proposed by Tzen and Ni [10], uses chunks that decrease in size linearly. This approach is simpler to implement than GSS and, specially, $\mathrm{GSS}(K)$, thus reducing scheduling overhead. Moreover, according to the authors, a big value of K in $\mathrm{GSS}(K)$ leads to a high imbalance, whereas small values lead to too much scheduling overhead. Consequently, an optimum value of K for GSS is difficult to obtain, particularly in unbalanced loops. By decreasing the chunk size linearly, TSS reduces the number of chunks and, hence, the overhead, and simplifies the calculation of the next chunk size, allowing its computation with atomic *Fetch-and-Increment* operations.

TSS is defined as $\mathrm{TSS}(f, \ell, N)$, where f is the size of the first chunk and ℓ is the size of the last one. The maximum value for f is N/P: With this value, the first P chunks will hold 3/4 of the iterations. A more conservative value is to set $f = N/(2P)$ when only 7/16 of the iterations will be executed in the first P chunks.

The area below $\mathrm{TSS}(f, \ell, N)$, called A, should be calculated in order to obtain the decreasing step δ:

$$A = \left\lfloor \frac{2N}{f + \ell} \right\rfloor, \quad \delta = \frac{f - \ell}{A - 1}.$$

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. Fig. 1c shows an example of TSS.

The total number of iterations being scheduled is at least N for all scheduling alternatives described. Only Self-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. Besides this, it is easy to see in Fig. 1 that the use of different scheduling alternatives with the recommended values for free parameters leads to a different number of blocks being scheduled.

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 [23].

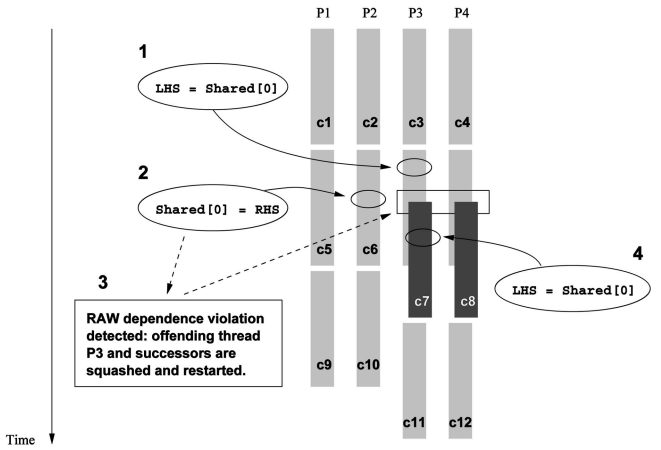


Fig. 2. Example of RAW dependence violation. $P1$ to $P4$ are four different processors executing chunks of consecutive iterations, labeled $c1$ to $c12$. The event sequence is described as follows: 1) A thread reads a shared variable. 2) A predecessor modifies the same variable. 3) A RAW dependence violation is detected. 4) The offending threads are squashed and restarted with the correct value of the variable.

3 Scheduling in the Context of 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 of its successors are squashed and restarted with the correct values. Fig. 2 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 cannot 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 and the time it takes to reexecute 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 of the work done by later threads in this batch will be reexecuted from the beginning and

$$t_p \leq Bt_b + t_b.$$

It is important to note that the 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. It is easy to see that each squash costs, at most, t_b. Hence, for several squashes, we get

$$t_p \leq Bt_b + N_s t_b$$

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

$$t_p \leq t_s/P + C/P\ t_o + N_s(t_s/C + t_o). \quad (1)$$

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 by minimizing t_p, derived 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. However, 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 (1) suggests that a useful strategy for speculative execution is to use big chunks on portions of the loop where fewer 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 Section 4, we will study randomized incremental algorithms and characterize their dependence pattern.

4 Randomized Incremental Algorithms

Randomized incremental algorithms have been deeply studied in areas such as computational geometry and optimization [12], [13], [14]. They have led to simple, easy-to-code, and efficient algorithms for a variety of problems. Among them, we can cite line segment intersection [15], [16], Voronoi diagrams [15], [17], [18], [19], triangulation of simple polygons [20], solving linear programs [21], and many others.

In their general formulation, the input of a randomized incremental 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, independent 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 it 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 a randomized incremental 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 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, the sets of violators and extreme elements are defined as follows:

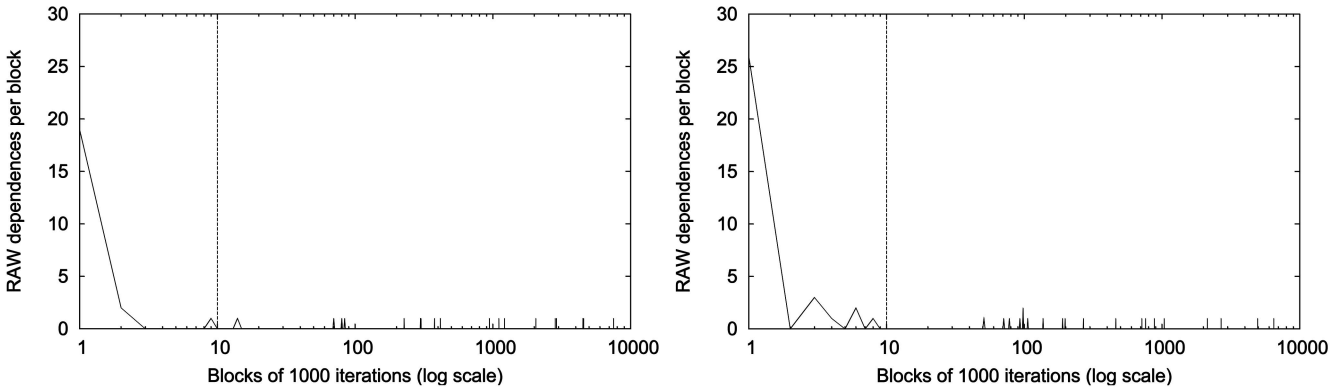
$$V(R_i) := \{s \in S \setminus R_i : \phi(R_i \cup \{s\}) \neq \phi(R_i)\}$$

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 [15], [24] and the Smallest Enclosing Circle [25]. The Convex Hull and the Smallest Enclosing Circle problems consist, respectively, of 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



(a) (b)

Fig. 3. Effective RAW dependences in Welzl's algorithm for the Smallest Enclosing Circle. Note the use of logarithmic scales. (a) Square-shaped input set. (b) Disc-shaped input set.

in $\mathbb{R}^2$. Considering $\phi \equiv$ `ConvexHull`, the violators $V(R_i)$ are those points outside `ConvexHull`(R_i), whereas the extreme points $X(R_i)$ are the vertices of this convex hull. For $\phi \equiv$ `SmallestEnclosingCircle`, 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 following lemma, whose proof can be found in Appendix A, states a relationship between the expected number of extreme elements found before processing element i and the expected number of violators to be found afterward.

Lemma 4.1 (Sampling Lemma). *Let $v_i := E(|V(R_i)|)$ and $x_i := E(|X(R_i)|)$ be the expected numbers of violators and extreme elements for R_i as above. Then, for any $i \in \{1, \ldots, n\}$,*

$$\frac{x_i}{i} = \frac{v_{i-1}}{n-i+1}.$$

That is, the fraction of elements defining the present output over those already processed is expected to equal the fraction of elements forcing the update of the present output over those remaining to be processed.

Therefore, at any step i, the expected number of potential RAW dependences is

$$v_{i-1} = \frac{x_i}{i}(n-i+1).$$

In other words, the probability of finding a dependence at step i is at most $\frac{x_i}{i}$. Randomized incremental algorithms are used when x_i is asymptotically smaller than i and, hence, this probability decreases as execution proceeds.

The situation can be considered *stable* when the probability ϵ of finding a potential RAW dependence is close enough to 0. Using the Sampling Lemma, the probability of having a potential RAW dependence can be computed for any value of i. Reciprocally, once a threshold ϵ is chosen, it is easy to derive from the formula $\frac{x_i}{i} = \epsilon$ the iteration i in which that ϵ is achieved. Now, observe that, given two possible thresholds, the faster the probability of finding a dependence tends to 0 (that is, the smaller the numerator x_i is), the more similar are the iteration values obtained for those thresholds. This implies that, the faster the probability of dependences goes to 0, the less determinant accuracy is in the choice of a threshold ϵ. We have chosen $\epsilon = 3 \cdot 10^{-4}$, for which our experimental results show that the situation is stable regardless of the input set.

4.2 Randomized Incremental Construction of the Smallest Enclosing Circle

For the above example of $\phi \equiv$ `SmallestEnclosingCircle`, it is clear that $x_i \leq 3$ for any i since at most three points are needed in order to define a circle in $\mathbb{R}^2$. Hence, the expected number of points outside the smallest circle enclosing R_i is

$$v_{i-1} \leq \frac{3}{i}(n-i+1).$$

In other words, the probability of finding a dependence at step i is at most $\frac{3}{i}$. The fact that this amount decreases quickly when i gets bigger shows why the probability of finding a dependence is much higher in the first iterations than in the remaining ones.

To check this result, two sample executions for square and disc-shaped input sets have been made. These sets are extreme in the sense that all other convex polygonal shapes (but triangles) happen to be in between these two. Performance for these sets will obviously be closer to one or the other, depending on the number of sides of the polygon. We have used constant-size blocks of 1,000 iterations to run Welzl's Smallest Enclosing Circle algorithm [25], measuring the number of potential dependence violations in each block. 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. Fig. 3 shows the effective distribution of RAW dependences for both input sets.

Applying the Sampling Lemma in the case of the Smallest Enclosing Circle, we obtain

$$\frac{3}{i} = 3 \cdot 10^{-4} \implies i \approx 10^4.$$

This iteration number is represented by vertical lines in Figs. 3a and 3b. As can be seen in the figures, the number of

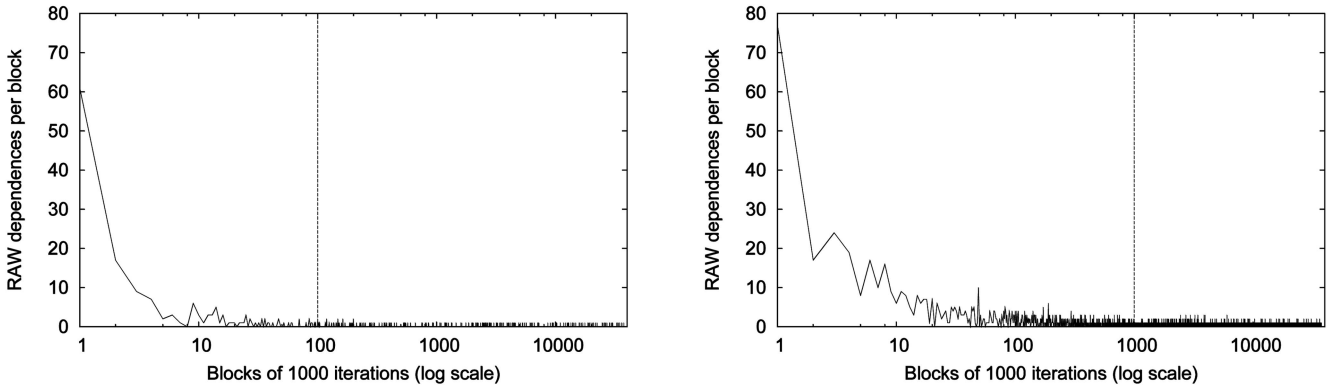


Fig. 4. Effective RAW dependences in Clarkson et al.'s algorithm for the 2D Convex Hull. Note the use of logarithmic scales. (a) Square-shaped input set. (b) Disc-shaped input set.

dependences after this iteration number is low enough to consider the situation stable.

4.3 Randomized Incremental Construction of a Convex Hull

For the randomized construction of the 2D Convex Hull ($\phi \equiv$ `ConvexHull`), the usual input data sets are uniform distributions of points in a k-gon (see [26]). 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 nonparallel. It is well known (see [27] and [28]) 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, whereas 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 randomized incremental 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 randomized incremental algorithm for computing 2D Convex Hulls due to Clarkson et al. [29]. 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. Fig. 4 shows the effective distribution of RAW dependences for both input sets. If we again considered the situation *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; we again apply the Sampling Lemma in order to obtain the iteration number, after which the situation can be considered stable. For a randomly distributed square-shaped input set, we set

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

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

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

Both iteration numbers are represented with vertical lines in Figs. 4a and 4b. It is easy to see that the number of dependences after each of them is low enough to consider the situation stable.

5 MESETA: SCHEDULING STRATEGY FOR RANDOMIZED INCREMENTAL ALGORITHMS

We have seen in the previous section how the number of dependences appearing in a randomized incremental algorithm tends to decrease as the algorithm proceeds. All dependences are, in principle, candidates to produce a squash in the context of speculative parallelization. The cost model described in Section 3 shows that the optimal size of chunks depends on the number of squashes, which is not known until execution time and depends on the size of the chunks. This is why it is rather difficult to find a fixed chunk size that minimizes the number of squashes. Moreover, the rest of the 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 randomized incremental algorithms. MESETA (the Spanish word for tableland) divides the execution of the loop into three parts (see Fig. 5):

- At the beginning of the loop, many dependences are likely to be found. We propose assigning 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

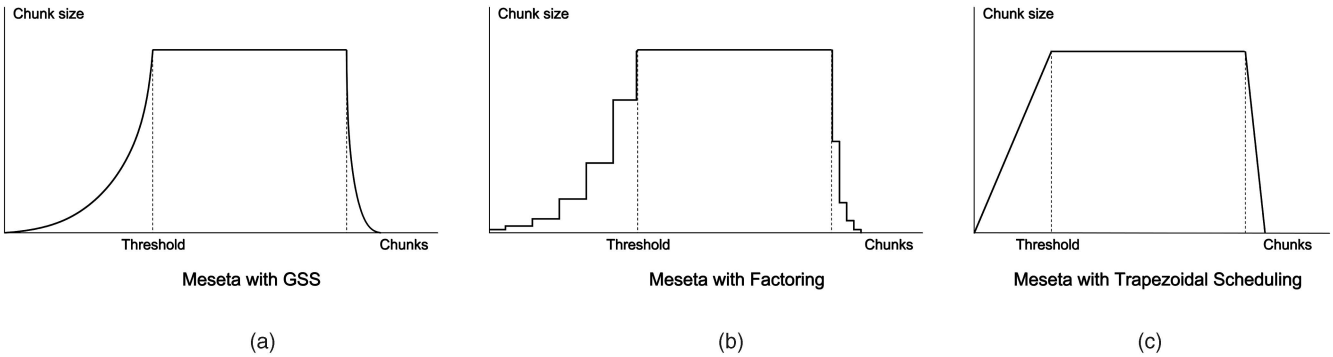


Fig. 5. Distribution of chunk sizes for different versions of MESETA. The area below each plot is the total number N of iterations being scheduled. The area below the ascending and descending phase is the same in all the versions considered.

TABLE 1
Characteristics of the Algorithms and Input Sizes Used

Algorithm	Input set description	Spec data size in KB	Iterations per invocation	% of dependence violations	Cost of squashing iter. i	Max chunk size used (iterations)
2D-Hull	Square, 10M points	13	9 999 997	2.00	$O(\log i)$	5 000
2D-Hull	Square, 40M points	15	39 999 997	0.64	$O(\log i)$	5 000
2D-Hull	Disc, 10M points	86	9 999 997	15.48	$O(\sqrt[3]{i})$	2 500
2D-Hull	Disc, 40M points	137	39 999 997	7.35	$O(\sqrt[3]{i})$	2 500
2D-SEC	Disc, 10M points	< 0.1	10 000 000	< 0.01	$O(i)$	1 000
2D-SEC	Disc, 40M points	< 0.1	10 000 000	< 0.01	$O(i)$	1 000

from appearing 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 probability value $\epsilon \in [0, 1]$ where the situation can be considered stable. At this point, we can use a fixed chunk size to minimize both squash and overhead costs.
- 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.

The first problem is how to determine the number of iterations to be executed before the probability of finding dependences is considered low enough. Fortunately, accuracy at this point is not determinant to the success of the scheduling mechanism, which performs well regardless of the particular transition point chosen. In Section 4.1, we have seen that randomized incremental algorithms follow the Sampling Lemma and, using this lemma, we have shown that conservative-enough iteration numbers are obtained in all of the studied cases with $\epsilon = 3 \cdot 10^{-4}$.

The second important decision is fixing the chunk size for the stable part of the loop. This decision will be postponed to Section 6 since it not only depends on the dependence pattern but also on the overheads produced by squashed threads.

We finally reach the descending part of the MESETA. We can now simply rely on the 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 [3], [22] was used to execute in parallel the two randomized incremental algorithms considered above: Clarkson et al.'s algorithm for the 2D Convex Hull problem (`2D-Hull`) [29] and Welzl's algorithm for the 2D Smallest Enclosing Circle problem (`2D-SEC`) [25]. In both cases, we executed the outer loop in parallel, which accounts for 100 percent of the algorithm sequential execution time. Table 1 summarizes the characteristics of both algorithms.

To test these applications, we have used four different standard input sets: two composed by points randomly distributed inside a square (10 and 40 million points) and two with the same numbers of random points distributed inside a disc. All input sets have been generated using the random points generator of the Computational Geometry Algorithms Library (CGAL) 2.4 [30] 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.

6.1 Environment Setup

The experiments performed were done on a Sun Fire 15K symmetric multiprocessor (SMP), equipped with 900 MHz UltraSparc III processors, each one with a private 64 KByte, 4-way set-associative L1 cache, a private 8 MByte, direct-mapped L2 cache, and 1 GByte of shared memory per processor. The system runs on Sun OS 5.8. The application was compiled with the Forte Developer 7 Fortran 95 compiler using the highest optimization settings for our execution environment: `-O3 -xchip = ultra3 -xarch = v8plusb -cache = 64/32/4 : 8192/64/1`. 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 or the smallest enclosing circle 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.

6.2 Dependences in the Applications Considered

In the case of Clarkson et al.'s algorithm, the number of violations between executions is bounded by the number of points lying outside the convex hull computed up to their insertion. As we saw in Section 4, the number of dependences changes according to how quickly the growing convex hull tends to the final one and the usual input data sets are uniform distributions of points. The asymptotic cost of a dependence violation is $O(\log i)$ for the square input set and $O(\sqrt[3]{i})$ for the disc input set (Appendix B shows the details). These costs are small compared with the $O(n \log n)$ expected complexity of the algorithm and, therefore, its speculative execution leads to good speedups despite the comparatively high number of dependence violations that may appear (see Table 1).

The situation is very different for Welzl's algorithm. The randomized incremental construction of the smallest enclosing circle has an expected complexity of $O(n)$ and it is easy to see that a dependence violation in iteration i will have a cost that is also in $O(i)$ (see Appendix B for the details). This fact severely affects the speculative performance of the algorithm since a single violation implies the recalculation of the solution for all of the points considered so far. As we will see in the following sections, the traditional scheduling mechanisms fail in the extraction of any parallelism for this algorithm, whereas MESETA effectively curves the performance degradation.

6.3 Choosing the Maximum Chunk Size

Recall that, in Section 5, we fixed the iteration number after which the loop can be considered stable for both the square and disc-shaped input sets. The next point is to obtain the best value for the fixed chunk size used in the middle part of the loop execution. As in the case of FSC, this value can only be obtained by experimentation. We will use the chunk sizes shown in Table 1. Fortunately, an incorrect choice for this value will not affect the obtained speedup significantly since it is only used when the number of expected dependences is considered low enough. An incorrect choice is much more serious if we use a fixed chunk size for the execution of the entire loop, as shown in [31].

6.4 Comparing Different MESETA Shapes

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

Three different scheduling functions will be used to distribute iterations in both the beginning and end of the speculative execution: GSS, Factoring, and TSS (see Fig. 5).

Fig. 6 shows the execution time breakdown for the processing of the 10-million point sets with the applications considered. As can be seen from the figure, the relative performance of `2D-Hull` 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 (more details on the behavior of the speculative engine can be found in [22]). In the case of the `2D-SEC` algorithm, the GSS version of MESETA leads to a much better performance than the other versions. The reason is that GSS distributes smaller chunks in the ascending part of the execution, thus leading to better management of the dependences found. It is also noticeable that most of the execution time is spent in spin waits. The reason is that the thread that detects a dependence violation should recalculate the new solution from scratch, again processing all of the points processed so far while the remaining threads wait for the new solution in order to continue. This effect makes it extremely difficult to obtain any speedup, despite the number of processors being used.

Fig. 7 shows the number of dependence violations triggered by each of our scheduling strategy versions together with the number of corresponding squashed threads. The plots show that the behavior of all MESETA versions is comparable in terms of dependence violations and squashes triggered. It is interesting to note that the poor performance of the Trapezoidal and Factoring versions of MESETA in the execution of `2D-SEC` is not reflected in these plots since the difference among these versions is not the number of dependences but the time spent in the reexecution of chunks, which is proportional to their size.

6.5 Performance Evaluation of MESETA

The last part of our study compares the performance of MESETA with respect to FSC, the only scheduling mechanism used so far in the field of speculative parallelization. The version of MESETA compared here is the one that uses GSS for both the increasing and decreasing parts of the loop execution: As we saw above, this function leads to a better performance than the other two alternatives. To give a global perspective of the scheduling problem in this context, we also compare both scheduling mechanisms with GSS. To better balance the work at the end of the loop in GSS, we have set $x = 2$ (see Section 2).

Fig. 8 shows the speedup of all three mechanisms in the execution of the applications considered. From the figure, we can draw the following observations:

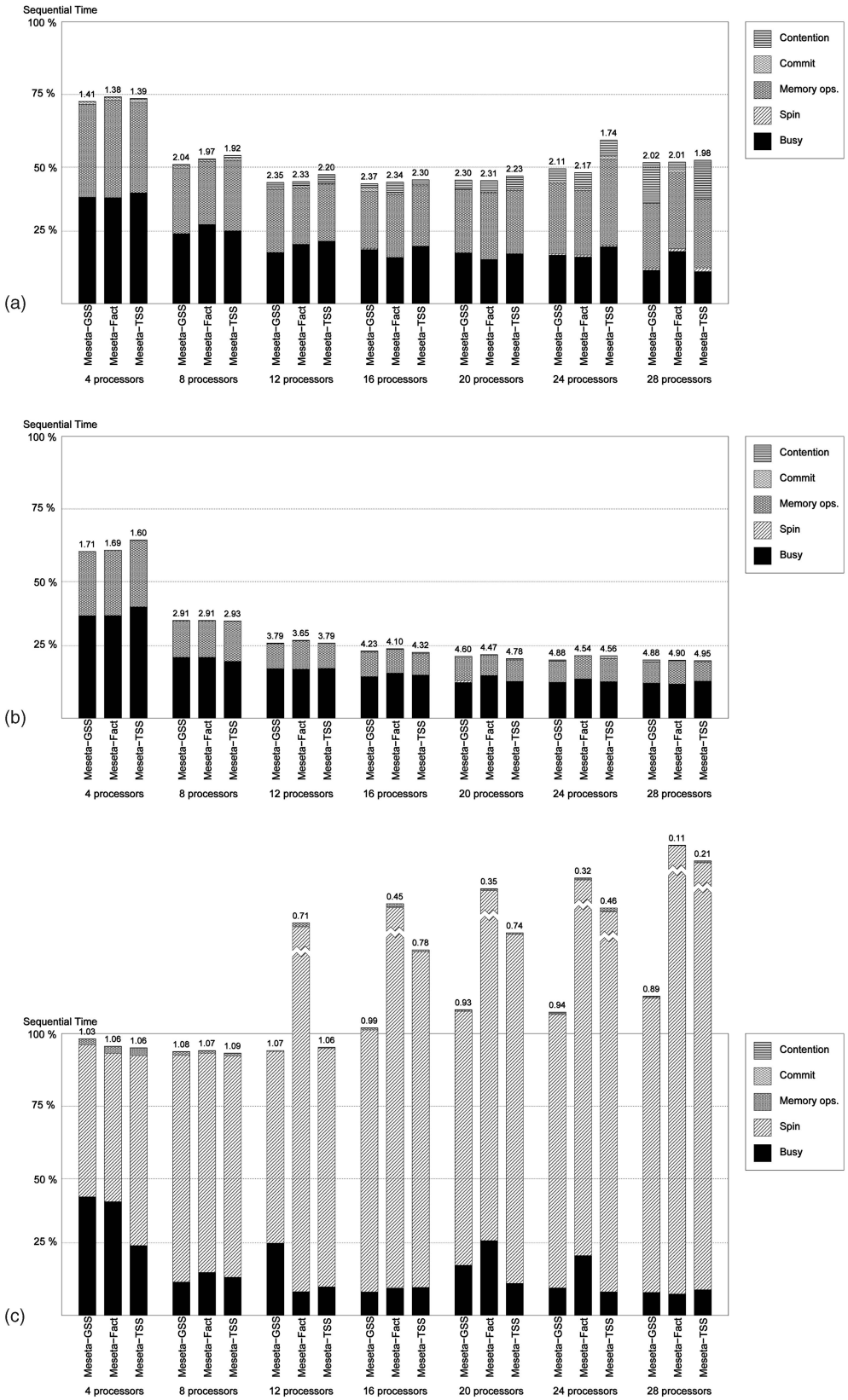


Fig. 6. Execution time breakdowns using different versions of MESETA. (a) `2D-Hull`, 10 million points, disc-shaped input set. (b) `2D-Hull`, 10 million points, square-shaped input set. (c) `2D-SEC`, 10 million points, disc-shaped input set.

- MESETA-GSS overcomes the FSC approach in all cases. Both mechanisms lead to very similar results when the number of dependence violations and their time costs are very small, such as `2D-Hull` processing the square input sets. Our scheduling strategy gives better results when the cost of a dependence violation grows: With the 10-million-point disc input set, the use of MESETA-GSS leads to a 5.5 percent to

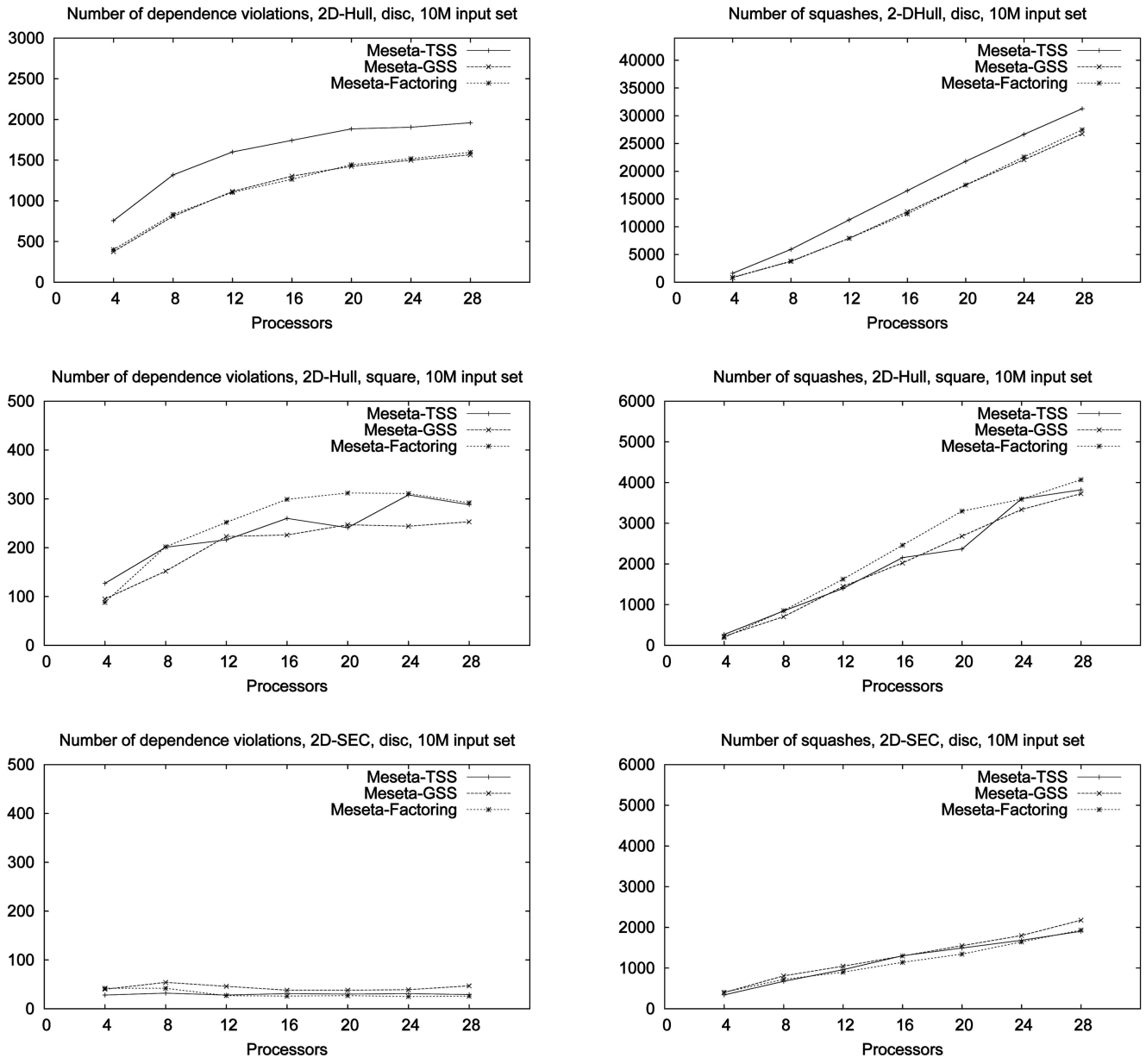


Fig. 7. Number of dependence violations and number of squashed threads using different versions of MESETA for three of the six combinations of applications and input sets listed in Table 1.

11.48 percent performance gain over FSC. The results are even better for the 40-million-point disc input set, leading to a performance gain between 15.63 percent and 36.25 percent. The gain is proportionally higher for the disc because, as expected, this input set generates many more dependences than the square-shaped one and therefore benefits more from our scheduling strategy.

- The high cost of a single dependence violation in the `2D-SEC` algorithm, discussed in Section 6.2, makes it impossible for FSC to extract any speedup at all. However, MESETA-GSS manages to curve the performance degradation, that is, smaller for the bigger input set, since the obtained solution is better amortized during the processing of the additional points.
- It is worth noting the poor performance of GSS in these experiments. This behavior is easy to explain, considering that, as we saw in Section 3, the only concern of GSS is to achieve good load balancing at the end of the loop. Therefore, GSS schedules the biggest chunks first, precisely where more dependences are found. The same considerations apply to TSS and Factoring as well.
- Finally, no performance loss due to the higher scheduling cost of MESETA-GSS in comparison with FSC 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 on those algorithms where dependences are less likely to appear as the

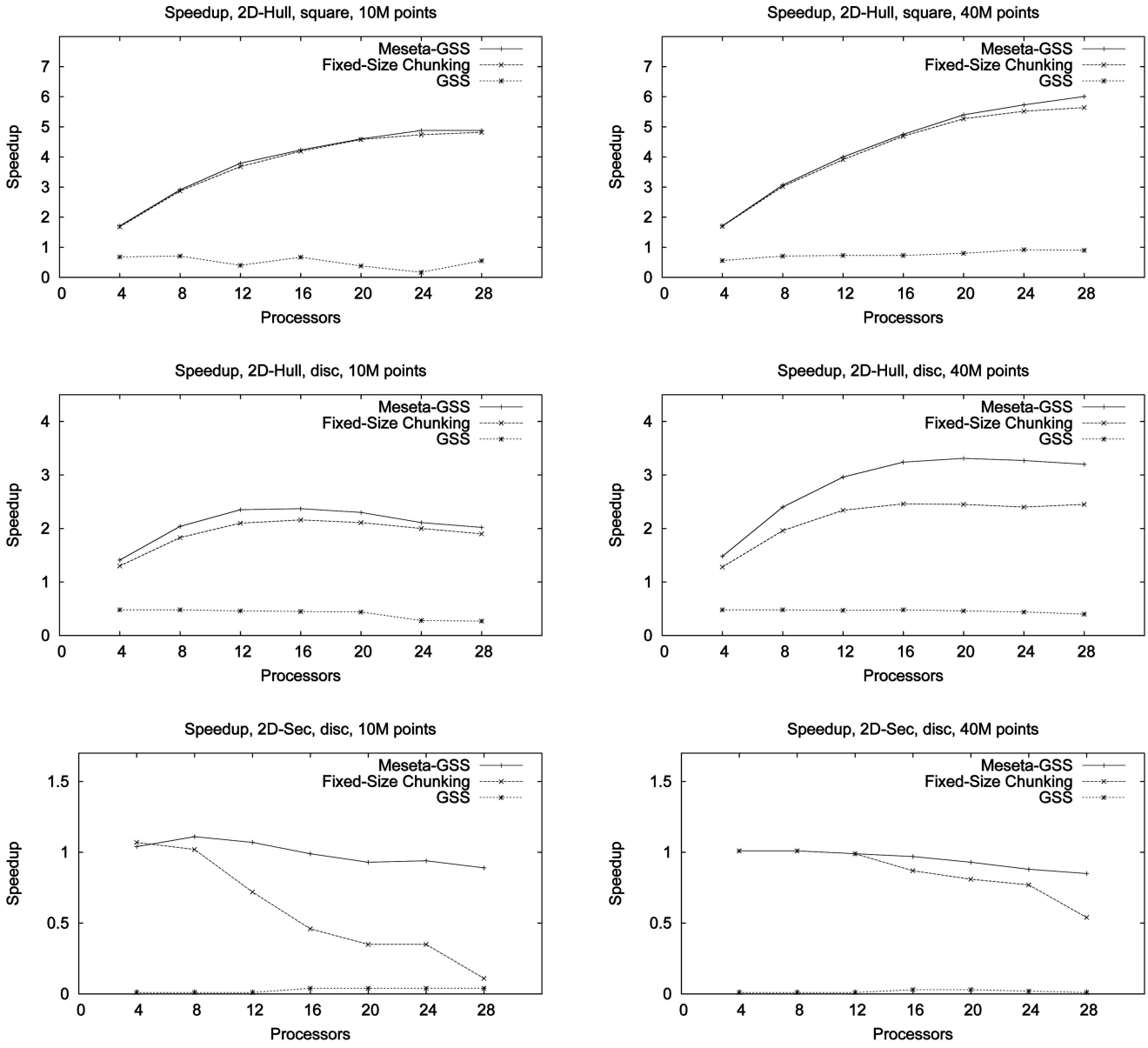


Fig. 8. Speedups during the execution of the applications considered with two different input set sizes.

execution proceeds, like randomized incremental 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 5.5 percent to 36.25 percent speedup improvement of MESETA over FSC, leading to a better extraction of the inherent parallelism of randomized incremental algorithms. Finally, in cases where the reexecution cost of a dependence violation is high, MESETA effectively curves the performance degradation compared to the rest of the scheduling strategies available.

APPENDIX A
PROOF OF THE SAMPLING LEMMA

In this section, we give the proof of the Sampling Lemma used in Section 4.1 to obtain the expected number of RAW dependences at each step of a randomized incremental algorithm. More details can be found in [32].

Proof. Consider the set $\binom{S}{r}$ of those subsets of S having cardinality r. Analogously, consider $\binom{S}{r+1}$ the subsets of

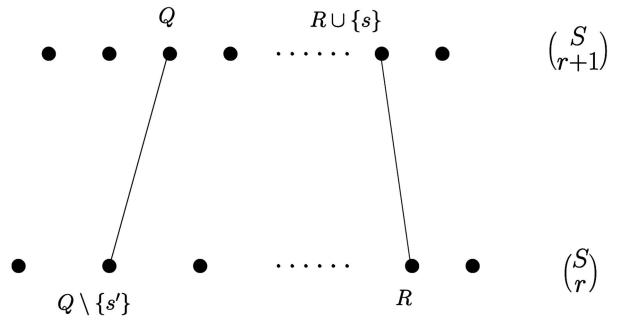


Fig. 9. Graph for the proof of the Sampling Lemma.

S with $r+1$ elements. We construct a graph G whose vertices correspond to the elements of $\binom{S}{r} \cup \binom{S}{r+1}$, depicted in two horizontal rows: the bottom one for the elements of $\binom{S}{r}$ and the top one for those of $\binom{S}{r+1}$. See Fig. 9.

As for the edges of G, a vertex $R \in \binom{S}{r}$ is joined to a vertex $R \cup \{s\}$ precisely if $\phi(R \cup \{s\}) \neq \phi(R)$ (that is, s is a violator of R). Note that this is a *bipartite graph*, that is, every edge is incident to exactly one vertex on each row. Let us count the number of edges incident to each of the two rows:

- By the above definition, the number of edges incident to an $R \in \binom{S}{r}$ on the bottom row equals the number $|V(R)|$ of violators.
- For a $Q \in \binom{S}{r+1}$ in the top row, there is an edge incident to it precisely if there is an element $s' \in Q$ such that $\phi(Q \setminus \{s'\}) \neq \phi(Q)$. Therefore, the number of edges incident to a $Q \in \binom{S}{r+1}$ on the top row equals the number $|X(Q)|$ of extreme points.

On the one hand, the graph being bipartite implies that the number of edges incident to the top row equals the number of edges incident to the bottom one. Therefore,

$$\sum_{R \in \binom{S}{r}} |V(R)| = \sum_{Q \in \binom{S}{r+1}} |X(Q)|.$$

On the other hand, the expected number of violators of a random $R \in \binom{S}{r}$ and the expected number of extreme points of a random $Q \in \binom{S}{r+1}$ are, respectively,

$$v_r = \frac{\sum_{R \in \binom{S}{r}} |V(R)|}{\binom{n}{r}}$$

and

$$x_{r+1} = \frac{\sum_{Q \in \binom{S}{r+1}} |X(Q)|}{\binom{n}{r+1}}.$$

Using these two facts, one gets that

$$v_r = x_{r+1} \frac{\binom{n}{r+1}}{\binom{n}{r}} = x_{r+1} \frac{n-r}{r+1},$$

which, in particular, gives the desired result for the R_i with i elements considered in the statement. □

APPENDIX B
COST OF RAW DEPENDENCES

We show here how to analyze the cost of a RAW dependence at step i in the algorithms considered.

```
SmallestEnclosingCircle(P)
 Ensure: B ← ∅
 for i = 1 to n
   if p_i ∉ EncBall(B) then
     B ← {p_i}
     for j = 1 to i − 1
       if p_j ∉ EncBall(B) then
         B ← {p_i, p_j}
         for k = 1 to j − 1
           if p_k ∉ EncBall(B) then
             B ← {p_i, p_j, p_k}
           end if
         end for
       end if
     end for
   end if
 end for
 return B
```

Fig. 10. Welzl's algorithm for the Smallest Enclosing Circle problem.

B.1 Dependence Cost for the Smallest Enclosing Circle

For the computation of the Smallest Enclosing Circle, we have used the iterative version of the original recursive algorithm by Welzl [25]. The algorithm is shown in Fig. 10. Even though this algorithm runs in expected $O(n)$ time, it is interesting now to take a deeper look at the constant involved. If we assume that conditional and assignment sentences cost one time unit, we can give a more precise upper bound for the time it takes to run the algorithm.

Let $T(n)$ be the time needed for the algorithm to run for an input of n points. The following observations are needed:

- For each step i of the outer loop, one containment query is performed for sure. If the condition holds, then an assignment is done and the second loop is executed. When the point p_i is outside the current candidate solution, then a new enclosing circle with p_i on the boundary has to be computed. As follows from the Sampling Lemma, this happens with probability $\frac{3}{i}$. Calling $T_1(i)$ the time needed to run the second loop, we obtain

$$T(n) = \sum_{i=1}^{n} \left(1 + \frac{3}{i} T_1(i)\right).$$

- For each step j of the next loop, again, one containment query is sure. The probability of running the assignment and entering the innermost loop is $\frac{2}{j}$. Therefore, calling $T_2(j)$ the time needed to run the innermost loop, we have

$$T_1(i) = 1 + \sum_{j=1}^{i-1} \left(1 + \frac{2}{j} T_2(j)\right).$$

- Finally, exactly one containment query is made at each step k of the innermost loop that will be

followed by an assignment with probability $\frac{1}{k}$, making

$$T_2(j) = 1 + \sum_{k=1}^{j-1} \left(1 + \frac{1}{k}\right).$$

Since $\frac{1}{k} \leq 1$, we have that

$$T_2(j) \leq 1 + 2(j-1) \leq 2j,$$

so

$$T_1(i) \leq 1 + 5(i-1) \leq 5i$$

and

$$T(n) \leq 16n.$$

Therefore, $T(n) \in O(n)$ and $T_1(i) \in O(i)$: Since $T_1(i)$ is the cost of point i being a violator, we conclude that both the expected complexity of the algorithm and the expected cost of a dependence violation are linear. This situation adversely affects the speculative parallelization of the algorithm, even in the presence of few dependences.

B.2 Dependence Cost for the Convex Hull

Clarkson et al.'s algorithm keeps the incrementally computed solutions in a data structure, from which point location queries will benefit. Whenever a point happens to be outside of the current solution, two tasks have to be performed. First, the two tangents to the candidate convex hull are computed. Second, the edges between the tangency points are updated in the data structure.

Finding the tangents from one point to a polygon with k edges has a cost of $O(\log k)$ using binary search. On the other hand, at most $k-1$ edges will have to be updated (in constant time) for the underlying structure in the algorithm. The cost of inserting a point that lies out of the current convex hull is therefore $O(k)$. As we said in Section 4.3, the number of convex hull edges depends on the distribution of the point set.

If the input points are uniformly distributed in a square, the expected number of edges after insertion of the ith point is $O(\log i)$. Hence, if point $i+1$ lies out of the hull, its insertion has a cost of $O(\log i)$.

On the other hand, if points are distributed uniformly in a disc, the complexity of the hull at step i is $O(\sqrt[3]{i})$, which coincides with the time bound of the insertion of point $i+1$ given that it lies out of the hull.

As above, from these costs for a violator $i+1$, we can conclude that the expected cost of a dependence violation for both input sets is much lower than the expected cost for the algorithm itself, which is $O(n \log n)$. Therefore, the algorithm is expected to have good behavior under speculative parallelization.

ACKNOWLEDGMENTS

This work was supported in part by Junta de Castilla y León under Grant VA031B06 and by Comunidad Autónoma de Madrid under Grant CAM S-0505/DPI/000235. Diego R. Llanos is partially supported by the European Commission under Contract RII3-CT-2003-506079. David Orden is partially supported by Grant MEC MTM2005-08618-C02-02. Belén Palop is partially supported by MCYT TIC2003-08933-C02-01. Part of this work was carried out while David Orden was visiting the Departamento de Informática, Universidad de Valladolid, with the support of the Universidad de Alcalá. The authors would like to thank the anonymous referees for their valuable suggestions. 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 Edinburgh Parallel Computing Center (EPCC) for the main computing resources used in this work and its support staff, in particular, Chris Johnson and Catherine Inglis.

REFERENCES

[1] M. Gupta and R. Nim, "Techniques for Run-Time Parallelization of Loops," *Proc. Supercomputing,* Nov. 1998.

[2] L. Rauchwerger and D.A. Padua, "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '95),* pp. 218-232, June 1995.

[3] M. Cintra and D.R. Llanos, "Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors," *Proc. SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '03),* June 2003.

[4] F. Dang, H. Yu, and L. Rauchwerger, "The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops," *Proc. 16th Int'l Parallel and Distributed Processing Symp. (IPDPS '02),* Apr. 2002.

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

[6] S.F. Hummel, E. Schonberg, and L.E. Flynn, "Factoring: A Method for Scheduling Parallel Loops," *Comm. ACM,* vol. 35, no. 2, pp. 90-100, Aug. 1992.

[7] C.P. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE Trans. Software Eng.,* vol. 11, no. 10, pp. 1001-1016, Oct. 1990.

[8] C.D. Polychronopoulos and D.J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. Computers,* vol. 36, no. 12, pp. 1425-1439, Dec. 1987.

[9] P. Tang and P.-C. Yew, "Processor Self-Scheduling for Multiple Nested Parallel Loops," *Proc. IEEE Int'l Conf. Parallel Processing (ICPP '86),* pp. 528-535, Aug. 1986.

[10] T.H. Tzen and L.M. Ni, "Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers," *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 1, pp. 87-98, Jan. 1993.

[11] D.R. Llanos, D. Orden, and B. Palop, "Meseta: A New Scheduling Strategy for Speculative Parallelization of Randomized Incremental Algorithms," *Proc. 34th Int'l Conf. Parallel Processing (ICPP '05) Workshops, Seventh Workshop High Performance Scientific and Eng. Computing (HPSEC '05),* pp. 121-128, June 2005.

[12] K. Mulmuley, "Randomized Algorithms in Computational Geometry," *Handbook of Computational Geometry,* J.-R. Sack and J. Urrutia, eds., chapter 16, pp. 703-724, North-Holland Publishing, 2000.

[13] K. Mulmuley and O. Schwarzkopf, "Randomized Algorithms," *Handbook of Discrete and Computational Geometry,* J.E. Goodman and J. O'Rourke, eds., chapter 34, pp. 633-652, CRC Press, 1997.

[14] *Handbook of Randomized Computing: Volumes I and II,* S. Rajasekaran, P.M. Paradalos, J.H. Reif, and J.D. Rolim, eds. Kluwer Academic, 2001.

[15] K.L. Clarkson and P.W. Shor, "Applications of Random Sampling in Computational Geometry, II," *Discrete and Computational Geometry,* vol. 4, no. 1, pp. 387-421, 1989.

[16] K. Mulmuley, "A Fast Planar Partition Algorithm, I," *Proc. 29th Ann. Symp. Foundations of Computer Science (FOCS '88),* pp. 580-589, 1988.

[17] O. Devillers, "Randomization Yields Simple $O(n \log^* n)$ Algorithms for Difficult $\omega(n)$ Problems," *Int'l J. Computational Geometry and Applications,* vol. 2, no. 1, pp. 97-111, 1992.
[18] L.J. Guibas, D.E. Knuth, and M. Sharir, "Randomized Incremental Construction of Delaunay and Voronoi Diagrams," *Algorithmica,* vol. 7, pp. 381-413, 1992.
[19] K. Mehlhorn, S. Meiser, and C. Ò'Dunlaing, "On the Construction of Abstract Voronoi Diagrams," *Discrete and Computational Geometry,* vol. 6, pp. 211-224, 1991.
[20] R. Seidel, "A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons," *Computational Geometry: Theory and Applications,* vol. 1, pp. 51-64, 1991.
[21] J. Matoušek, M. Sharir, and E. Welzl, "A Subexponential Bound for Linear Programming," *Algorithmica,* vol. 16, pp. 498-516, 1996.
[22] M. Cintra and D.R. Llanos, "Design Space Exploration of a Software Speculative Parallelization Scheme," *IEEE Trans. Parallel and Distributed Systems,* vol. 16, no. 6, pp. 562-576, June 2005.
[23] S. Lucco, "A Dynamic Scheduling Method for Irregular Parallel Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '92),* pp. 200-211, 1992.
[24] R. Seidel, "Linear Programming and Convex Hulls Made Easy," *Proc. Sixth Ann. ACM Symp. Computational Geometry,* pp. 211-215, 1990.
[25] E. Welzl, "Smallest Enclosing Disks (Balls and Ellipsoids)," *New Results and New Trends in Computer Science,* H. Maurer, ed., pp. 359-370, Springer, 1991.
[26] S. Har-Peled, "On the Expected Complexity of Random Convex Hulls," Technical Report 330, School of Math. Sciences, Tel-Aviv Univ., 1998.
[27] H. Raynaud, "Sur l'Enveloppe Convexe des Nuages de Points Aléatoires dans $\mathbb{R}^n$," *J. Applied Probability,* vol. 7, pp. 35-48, 1970.
[28] A. Renyi and R. Sulanke, "Über die Jkonvexe Hülle von n Zufällig Gerwählten Punkten II," *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete,* vol. 3, pp. 138-147, 1964.
[29] K.L. Clarkson, K. Mehlhorn, and R. Seidel, "Four Results on Randomized Incremental Constructions," *Computational Geometry: Theory and Applications,* vol. 3, no. 4, pp. 185-212, 1993.
[30] CGAL, Computational Geometry Algorithms Library, http://www.cgal.org/, May 2002.
[31] M. Cintra, D.R. Llanos, and B. Palop, "Speculative Parallelization of a Randomized Incremental Convex Hull Algorithm," *Proc. Fourth Int'l Workshop Computational Geometry and Applications (CGA '04),* pp. 188-197, May 2004.
[32] B. Gärtner and E. Welzl, "A Simple Sampling Lemma: Analysis and Applications in Geometric Optimization," *Discrete and Computational Geometry,* vol. 25, no. 4, pp. 569-590, 2001.

Diego R. Llanos received the MS and PhD degrees in computer science from the University of Valladolid, Spain, in 1996 and 2000, respectively. He is an associate professor of computer architecture at the University of Valladolid and his research interests include parallel and distributed computation, computer system performance evaluation, and automatic parallelization of sequential code. He is a member of the IEEE and the IEEE Computer Society. He is a recipient of the Spanish government's national award for academic excellence. More information about his current research activities can be found at http://www.infor.uva.es/~diego.

David Orden received the MS and PhD degrees in mathematics from the University of Cantabria in 1999 and 2003, respectively. He is an assistant professor of applied mathematics at the University of Alcala, Spain. He was formerly a visiting professor of geometry at the University of Alicante and a granted researcher at the University of Cantabria. His research interests are focused on discrete and computational geometry from geometric graphs, including triangulations, pseudotriangulations, rigidity, and crossings to geometric algorithms like incremental randomized ones. More information about his work is available at http://www2.uah.es/ordend.

Belén Palop received the PhD degree in computer science from the Universitat Politècnica de Catalunya in 2003. Since 2002, she has been an associate professor in the Department of Computer Science at the University of Valladolid, Spain. She worked previously for other universities in Spain (in Madrid and Barcelona). Her research interests include computational geometry and automatic parallelization for geometric algorithms. More information about her current research activities can be found at http://www.infor.uva.es/~b.palop.