Evaluation of OpenMP Task Scheduling Strategies

Alejandro Duran, Julita Corbalán, and Eduard Ayguadé

Barcelona Supercomputing Center
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona, 1-3, Barcelona, Spain
{`aduran,juli,eduard`}`@ac.upc.edu`

Abstract. OpenMP is in the process of adding a tasking model that allows the programmer to specify independent units of work, called tasks, but does not specify how the scheduling of these tasks should be done (although it imposes some restrictions). We have evaluated different scheduling strategies (schedulers and cut-offs) with several applications and we found that work-first schedules seem to have the best performance but because of the restrictions that OpenMP imposes a breadth-first scheduler is a better choice to have as a default for an OpenMP runtime.

1 Introduction

OpenMP grew out structured around parallel loops and was meant to handle dense numerical applications. The simplicity of its original interface, the use of a shared memory model, and the fact that the parallelism of a program is expressed in directives that are loosely-coupled to the code, all have helped OpenMP become well-accepted today. However, the sophistication of parallel programmers has grown in the last 10 years since OpenMP was introduced, and the complexity of their applications is increasing. Therefore, the forthcoming OpenMP 3.0[13] adds a new tasking model[2] to address this new programming landscape. The new directives allow the user to identify units of independent work, called `tasks`, leaving the scheduling decisions of how and when to execute them to the runtime system.

In this paper we explore different possibilities about the scheduling of these new `tasks`. We have extended our prototype runtime[15] with two scheduling strategies: a breadth-first approach and a work-first approach. We have also implemented several queueing and work-stealing strategies. Then, we have evaluated combinations of the different scheduling components with a set of applications. We also evaluated how these schedulers behave if the application uses `tied tasks`, which have some scheduling restrictions, or `untied` ones, wich have no scheduling restrictions.

The remaining of the paper is structured as follows: Section 2 describes our motivation and the related work, Section 3 describes the different schedulers we have implemented, Section 4 shows the evaluation results and finally Section 5 presents the conclusions of this work.

2 Motivation and Related Work

The Intel *work-queueing* model [14] was an early attempt to add dynamic task generation to OpenMP. This proprietary extension to OpenMP allows hierarchical generation of tasks by nesting `taskq` constructs. The NANOS group proposed `dynamic sections` [4] as an extension to the standard `sections` construct to allow dynamic generation of tasks.

Lately, a committee from different institutions developed a task model[2] for the OpenMP language that seems that it will be finally adopted[13]. One of the things this proposal leaves open is the scheduler of tasks that should be used.

Scheduling of tasks is a very well studied field. There are two main scheduler families: those that use breadth-first schedulers (see for example the work from Narlikar[12] and those that use work-first schedulers with work-stealing techniques (see for example Cilk[7] and Acar et al.[1]). Korch et al.[9] made a very good survey of different task pool implementations and scheduling algorithms and evaluated them with a radiosity application. Many of these works have found that work-first schedulers tend to obtain better performance results.

Several works have studied how to reduce the overhead of task creation by means of using cut-off strategies. They have found that strategies based on controlling the recursion level tend to work very well[10,11]. Another proposal, uses the size of data structures[8] to control task creation but it depends on the compiler understanding complex structures like lists, which is difficult in the C or Fortran languages.

But, it is unclear how all these algorithms will map into the new task model as most of the previous work was in the context of recursive algorithms and where there were no scheduling restrictions at all. But the new task model allows not only non-recursive applications but also applications that mix traditional work-sharing regions with the new task model. Our goal in this work is to evaluate previous techniques in the context of OpenMP and to try to find which ones work best in order to help implementors choose appropriate defaults.

3 Task Scheduling

We have extended our research NANOS runtime[15] with two families of schedulers: breadth-first schedulers and work-first schedulers. These schedulers implement the restrictions about scheduling of `tied tasks` (i.e. tied tasks can only be scheduled on the thread to wich they are tied to). Also, we implemented two cut-off strategies: one based on the level of task recursion and another in the total number of existing tasks.

3.1 Breadth-First Scheduling

Breadth-first scheduling (BF) is a naive scheduler in which every task that is created is placed into the team pool and execution of the parent task continues. So, all tasks in the current recursion level are generated before a thread executes tasks from the next level.

Initially, tasks are placed in a team pool and any thread of the team can grab tasks from that pool. When a task is suspended (e.g. because a `taskwait`), if it is a `tied task` it will go to a private pool of tasks of the thread that was executing the tasks. Otherwise (i.e an `untied task`), it will be queued into the team pool.

Threads will always try to schedule first a task from their local pool. If it is empty then they will try to get tasks from the team pool.

We implemented two access policies for the task pools: LIFO (i.e., last queued tasks will be executed first) and FIFO (i.e., oldest queued tasks will be executed firsts).

3.2 Work-First Scheduling

Work-first scheduling (WF) tries to follow the serial execution path hoping that if the sequential algorithm was well designed it will lead to better data locality.

The WF scheduler works as follows: whenever a task is created, the creating task (i.e. the parent task) is suspended and the executing thread switches to the newly created task. When a task is suspended (either because it created an new one or because some synchronization) the task is placed in a per thread local pool. Again, this pool can be accessed in a LIFO or FIFO manner.

When looking for tasks to execute, threads will look on their local pool. If it is empty, they will try to steal work from other threads. In order to minimize contention we used a strategy where a thread traverses all other threads starting by the next thread (i.e. thread 0 starts trying to steal from thread 1, thread 1 from thread 2, ... and thread n from thread 0). When stealing from another thread pool, to comply with OpenMP restrictions, a task that has become tied to a thread cannot be stolen (note that a `tied task` that has not been yet executed can be stolen). The access to the victim's pool can also be LIFO or FIFO.

We also implemented a stealing strategy that first tries to steal the parent task of the current task. If the parent task cannot be stolen (i.e. because is either already running or waiting on some synchronization) then the default stealing mechanism is used.

The Cilk scheduler[7] pertains to this family of schedulers. In particular, it is a work-first scheduler where access to the local pool is LIFO, tries to steal the parent task first and otherwise steals from another thread pool in a FIFO manner.

3.3 Cutting Off

In order to reduce the size of the runtime structures and, also, reduce the overheads associated to creating tasks, the runtime can decide to start executing tasks immediately. This is usually referred as cutting off.

This is particularly important with breadth-first scheduling as it tends to generate a large number of tasks before executing them. In work-first scheduling the number of tasks that exist at a given time is not so large but it may grow over time because of tasks being suspended at synchronization points.

It is important to note that tasks that are executed immediately because of a cut-off policy are different than the ones that get executed immediately with the work-first scheduler. When cutting off, the new task does not go through to the whole creation process and in many aspects forms part of the creating tasks (e.g. cannot be suspended on its own).

We have implemented two simple but effective cut-off policies:

Max number of tasks (max-task). The total number of tasks that can exist in the pool is computed as a factor of the number of OpenMP threads (i.e. $k * num\_threads$). Once this number is reached new tasks are executed immediately. When enough tasks finish, tasks will be put into the task pool again. In our implementation, we use a default value for k of 8.

Max task recursion level (max-level). When a new task is created, if it has more ancestors than a fixed limit l then the new task is executed immediately. Otherwise it can be placed in the task pool. In our implementation, we use a default value for l of 4.

4 Evaluation

4.1 Applications

We have used the following applications (for more information on the parallelization please check our previous work[3]) for the evaluation of the schedulers:

Strassen. Strassen is an algorithm[6]for multiplication of large dense matrices. It uses hierarchical decomposition of a matrix. We used a 1280x1280 matrix for our experiments.

NQueens. This program, which uses a backtracking search algorithm, computes all solutions of the n-queens problem, whose objective is to find a placement for n queens on an n x n chessboard such that none of the queens attacks any other. We used a chessboard of size 14 by 14 in our experiments.

FFT. FFT computes the one-dimensional Fast Fourier Transform of a vector of n complex values using the Cooley-Tukey algorithm[5]. We used a vector with 33554432 complex numbers.

Multisort. Multisort is a variation of the ordinary mergesort, which uses a parallel divide-and-conquer mergesort and a serial quicksort when the array is too small. In our experiments we were sorting a random array of 33554432 integer numbers.

Alignment. This application aligns all protein sequences from an input file against every other sequence and compute the best scorings for each pair by means of a full dynamic programming algorithm. In our experiments we used 100 sequences as input for the algorithm.

SparseLU. The sparseLU kernel computes an LU matrix factorization. The matrix is organized in blocks that may not be allocated. Due to the sparseness of the matrix, a lot of imbalance exists. In our experiments, the matrix had 50 blocks each of 100x100 floats.

In all applications (except *Alignment*) we marked all tasks as *untied* and we removed any kind of manual cut-off that was there from the programmer to leave total freedom to the scheduler. The *Aligment* application makes heavy use of `threadprivate` and, because of that, we could not mark the tasks as `untied`.

4.2 Methodology

We evaluated all the benchmarks on an SGI Altix 4700 with 128 processors, although they were run on a cpuset comprising a subset of the machine to avoid interferences with other running applications.

We compiled all applications with our Mercurium compiler[4] using gcc with option -O3 as the backend. The serial version of the application was compiled with gcc -O3 as well. The speed-ups were computed using the serial execution time as the baseline and using the average execution time of 5 executions.

We have executed all applications with different combinations of schedulers. Table 1 summarizes the different schedules we have used in the evaluation, their properties (see Section 3 for details) and the name we will be using to refer to them in the next sections.

Table 1. Summary of schedules used in the evaluation

Scheduler Name	Scheduler Type	Pool Access	Steal Access	Steal Parent
bff	breadth-first	FIFO	-	-
bfl	breadth-first	LIFO	-	-
wfff	work-first	FIFO	FIFO	No
wffl	work-first	FIFO	LIFO	No
wflf	work-first	LIFO	FIFO	No
wfll	work-first	LIFO	LIFO	No
cilk	work-first	LIFO	FIFO	Yes

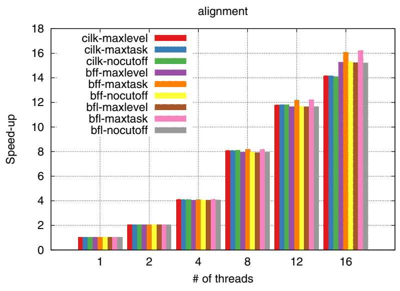
For each schedule we have run the applications using no cut-off and then using the cut-offs we had implemented:the *max-task* and the *max-level*.

Then, we wanted to know how the restrictions of `untied tasks` affected the performance that can be obtained with the different schedulers. So, we have also tried for those combinations that were best from each application but with all tasks `tied` (we control this via an environment variable that the runtime checks).

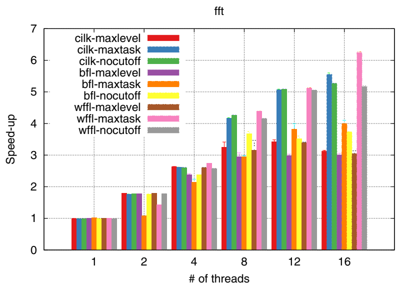
4.3 Results

In this section we present several lessons we have learned about task scheduling from this evaluation. Because of space considerations we are only showing part of the evaluation.

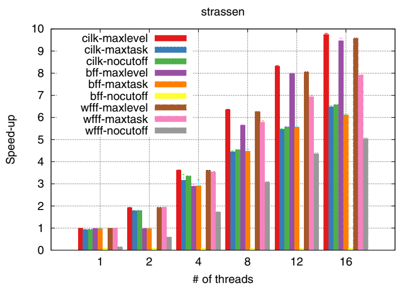
Lesson 1: Cutting Off: Yes, But How?. Figure 1 shows the speed-ups of three of the applications (*Alignment*,*FFT* and *Strassen*) and different schedulers. For each of them, three versions are shown: one that uses no cutoff, another that uses the max-level cutoff mechanism and the last that uses the max-task mechanism.



(a) for Alignment



(b) for FFT



(c) for Strassen

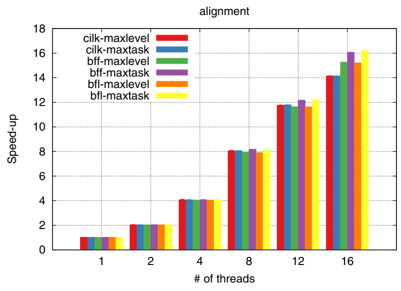
Fig. 1. Performance of difference cutoffs

Except for *Alignment*, if a cut-off strategy is not used there is a degradation in the obtained performance. The amount of degradation depends on the scheduler and the application but as a general rule we can see that breath-first schedulers suffer more (see for example *Strassen*) from the lack of a cut-off while work-first schedulers seem to withstand better the lack of a cut-off.

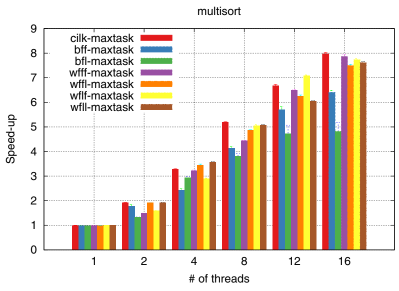
Another observation from these results is that choosing the wrong cut-off can be worse performance-wise than having no cut-off (see for example *FFT* where the max-level cut-off has less speed-up than no cutoff). But, we can also see that for different applications the right cut-off is different (for example compare *FFT* versus *Strassen*).

So, while it seems clear that is important to use a cut-off technique the decision of which to use remains unclear because it depends on the scheduler and also on the exact application.

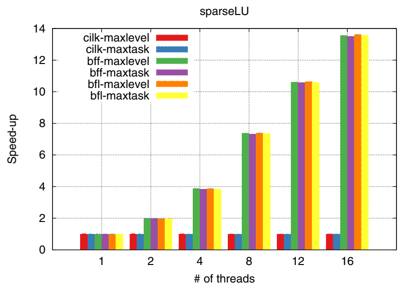
Lesson 2: Work-First Schedulers Work Best. Figure 2 shows the speed-up obtained with different schedulers (in general we show the most efficient schedulers, but also some others that might be interesting).



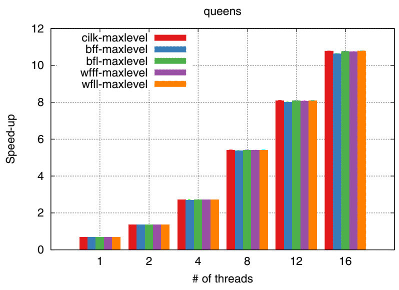
(a) Alignment



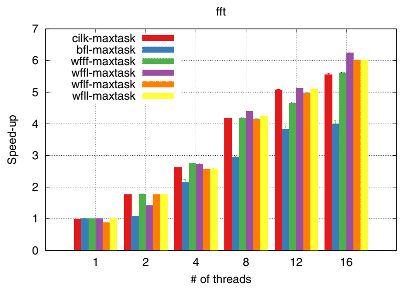
(b) Multisort



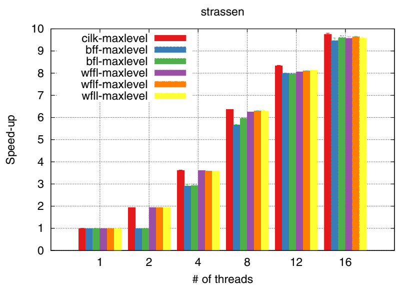
(c) SparseLU



(d) N Queens



(e) FFT



(f) Strassen

Fig. 2. Speed-ups with different schedulers

We can see that in most applications work-first schedulers get the best speed-up as they tend to exploit data locality better. The exceptions are *Alignment*, where tasks are `tied`, and *SparseLU*, where also a `tied` task limits the performance that can be obtained (see next sections for more details). Among work-first schedulers it seems that the Cilk scheduler is the best except for *FFT* where a *wffl* scheduler gets the best speed-up.

Also, we can observe again the difference in performance depending on the cut-off. In *Alignment* and *SparseLU* there is only a small difference in performance from using one cut-off or the other but in all other applications only schedulers using a particular cut-off are among the top: for *Multisort* and *FFT* cutting by number of tasks works better and for *N Queens* and *Strassen* cutting by the depth level works better. *Alignment* and *SparseLU* are non-recursive applications and this may be the reason why the cutting-off is not so important.

Lesson 3: Beware the Single!. In the *sparseLU* performance results from Figure 2(c) the work-first schedulers do not scale even a bit. The structure of the *sparseLU* application is similar to the one shown in Figure 3. There are a number of `untied tasks` inside a nest of loops inside a `single`.

```
#pragma omp parallel
#pragma omp single
#pragma omp task default(shared) untied
{
for ( k = 0; k < N ; k++ )
     // bunch of untied tasks
}
```

Fig. 3. Structure of a false untied application

All explicit tasks are `untied` but then the `single` region forms part of an implicit task. As such, it is always `tied`. As all tasks (unlike in other applications with a recursive structure) are generated from the `single` region, but the region cannot threadswitch, a work-first scheduling becomes a serial execution.

This can be easily solved by inserting an extra `untied task` after the single construct as shown in Figure 4. Now, the generator code is part of an `untied task` instead of the implicit task so it can be threadswitched.

We have implemented a second version of the *sparseLU* benchmark with this minor modification. We can see, from the results in Figure 5, that the work-first schedulers now achieve a better than the best breadth-first schedule. So, we can see that the effect of this, rather obscure, performance mistake can actually make a good scheduler look bad.

Lesson 4: Deep-First Schedulers Should Be the Default. The *sparseLU* problem is already an indication that the work-first schedulers may have problems when there are no `untied tasks`. Figure 6 shows how the same schedulers perform when task are `untied` versus when they are `tied`.

We can see that in all cases if the tasks are `tied` performance of the work-first schedulers is severely degraded to the point that no speed-up is obtained.

But, for the breadth-first schedulers the difference is barely noticeable. Moreover, in some cases (*multisort* and *FFT*) the speed-up obtained is better than when tasks are allowed to threadswitch.

Figure 7 shows the average speed-up from all the applications for all combinations of schedulers and cut-offs both for when `untied tasks` and `tied tasks`

```
#pragma omp parallel
#pragma omp single
#pragma omp task default(shared) untied
{
for ( k = 0; k < N ; k++ )
    // bunch of untied tasks
}
```

Fig. 4. Solution to the false untied problem

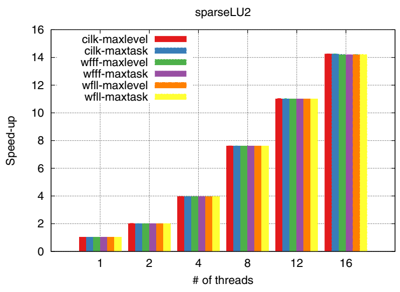
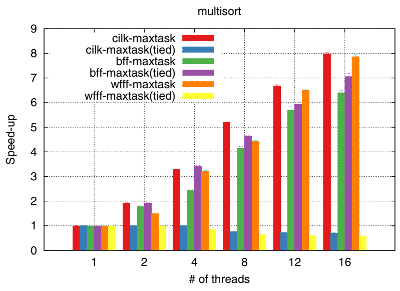
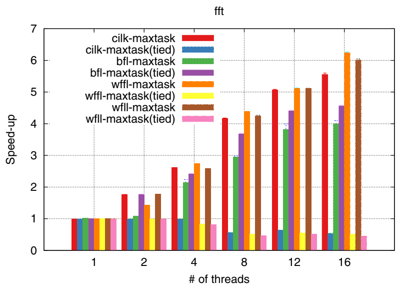


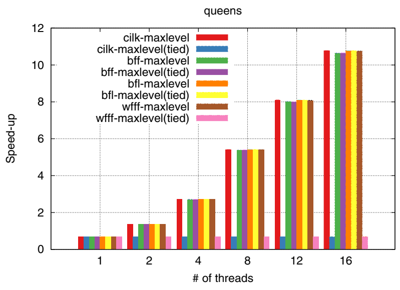
Fig. 5. Speed-ups for sparseLU with an extra untied task



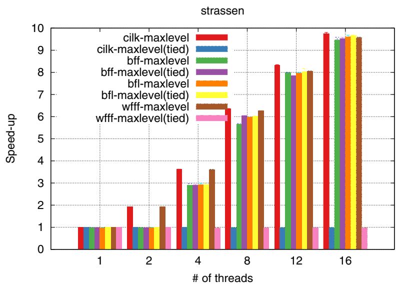
(a) Multisort



(b) FFT

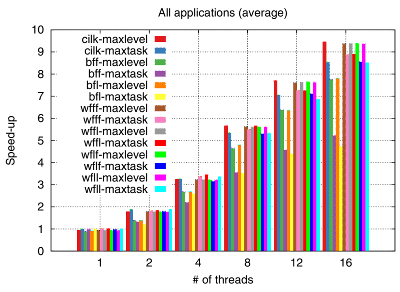


(c) N Queens

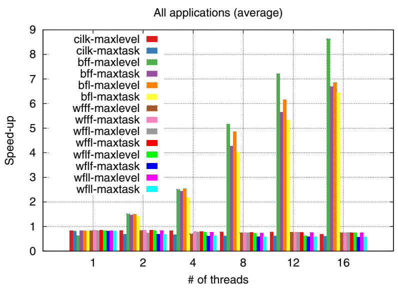


(d) Strassen

Fig. 6. Untied vs tied tasks



(a) with untied tasks



(b) with tide tasks

Fig. 7. Average speed-ups for all schedulers

are used. These graphs stress our last lesson, when using `untied tasks` work-first schedulers tend to obtain better speed-up but they drop flat when `tied tasks` are used. In that case, breadth-first schedulers perform about the same as they did with `untied task` thus outperforming work-first schedulers.

As `tied` is the OpenMP default, it seems that a wise choice for a compiler (or runtime) is a breadth-first scheduler unless it can safely be guaranteed that all tasks will be `untied`.

5 Conclusions and Future Work

In this work, we have explored several scheduling strategies of OpenMP tasks. We found that while work-first schedulers, in general, obtain better performance that breadth-first schedulers they are not appropriate to be used as the default for OpenMP programs. This is because `tied` and implicit tasks (which may be difficult to spot for novice users) severely restrict the performance that can be obtained. And in those circumstances, which are the default for OpenMP , breadth-first schedulers outperform work-first schedulers.

We have found that while it is a good idea to have a cut-off mechanism, it is not clear which one to use as it may affect negatively the performance of the application and more research is needed in that direction.

As future work, it would be interesting to explore a hybrid cut-off strategy (that takes into account the maximum number of tasks and the depth level) as well as some other more complex cut-off strategies that try to estimate the granularity of the work of a task. Also, it would be interesting to develop a scheduler that detects at runtime the structure of the application (whether it is recursive, whether it uses `tied tasks` or not, . . .) and it chooses one scheduler or the other appropriately.

Acknowledgments

This work has been supported by the Ministry of Education of Spain under contract TIN2007-60625, and the European Commission in the context of the SARC project #27648 and the HiPEAC Network of Excellence IST-004408.

References

1. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: SPAA 2000: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures, pp. 1–12. ACM, New York (2000)
2. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Su, E., Unnikrishnan, P., Zhang, G.: A proposal for task parallelism in OpenMP. In: Proceedings of the 3rd International Workshop on OpenMP, Beijing, China (June 2007)
3. Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., Teruel, X.: An Experimental Evaluation of the New OpenMP Tasking Model. In: Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (October 2007)
4. Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E., Labarta, J.: Nanos Mercurium: a Research Compiler for OpenMP. In: Proceedings of the European Workshop on OpenMP 2004 (October 2004)
5. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Mathematics of Computation 19, 297–301 (1965)
6. Fischer, P.C., Probert, R.L.: Efficient procedures for using matrix algorithms. In: Proceedings of the 2nd Colloquium on Automata, Languages and Programming, London, UK, pp. 413–427. Springer, Heidelberg (1974)
7. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: PLDI 1998: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, pp. 212–223. ACM Press, New York (1998)
8. Huelsbergen, L., Larus, J.R., Aiken, A.: Using the run-time sizes of data structures to guide parallel-thread creation. In: LFP 1994: Proceedings of the 1994 ACM conference on LISP and functional programming, pp. 79–90. ACM, New York (1994)
9. Korch, M., Rauber, T.: A comparison of task pools for dynamic load balancing of irregular algorithms: Research articles. Concurr. Comput. Pract. Exper. 16(1), 1–47 (2004)
10. Loidl, H.-W., Hammond, K.: On the Granularity of Divide-and-Conquer Parallelism. In: Glasgow Workshop on Functional Programming, Ullapool, Scotland, July 8–10, 1995, Springer, Heidelberg (1995)
11. Mohr, J. E., Kranz, D.A., Halstead, R.H.: Lazy task creation: a technique for increasing the granularity of parallel programs. In: LFP 1990: Proceedings of the 1990 ACM conference on LISP and functional programming, pp. 185–197. ACM, New York (1990)
12. Narlikar, G.J.: Scheduling threads for low space requirement and good locality. In: SPAA 1999: Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures, pp. 83–95. ACM, New York (1999)
13. OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 3.0 (Draft) (October 2007)
14. Shah, S., Haab, G., Petersen, P., Throop, J.: Flexible control structures for parallelism in OpenMP. In: 1st European Workshop on OpenMP (September 1999)
15. Teruel, X., Martorell, X., Duran, A., Ferrer, R., Ayguadé, E.: Support for OpenMP tasks in Nanos v4. In: CAS Conference 2007 (October 2007)