An adaptive cut-off for task parallelism

Alejandro Duran, Julita Corbalán, Eduard Ayguadé
Barcelona Supercomputing Center
Departament d’Arquitectura de Computadors
Universitat Politècnica de Catalunya
{aduran, juli, eduard}@ac.upc.edu

Abstract—In task parallel languages, an important factor for achieving a good performance is the use of a cut-off technique to reduce the number of tasks created. Using a cut-off to avoid an excessive number of tasks helps the runtime system to reduce the total overhead associated with task creation, particularly if the tasks are fine grain. Unfortunately, the best cut-off technique is usually dependent on the application structure or even the input data of the application.

We propose a new cut-off technique that, using information from the application collected at runtime, decides which tasks should be pruned to improve the performance of the application. This technique does not rely on the programmer to determine the cut-off technique that is best suited for the application.

We have implemented this cut-off in the context of the new OpenMP tasking model. Our evaluation, with a variety of applications, shows that our adaptive cut-off is able to make good decisions and most of the time matches the optimal cut-off that could be set by hand by a programmer.

I. INTRODUCTION

OpenMP was initially designed for parallel scientific applications executed in shared memory architectures. However, the sophistication of parallel programmers has grown in the 10 years since OpenMP was introduced, and the complexity of their applications is increasing. Therefore, the forthcoming OpenMP 3.0 [1] adds a new tasking model [2] to address this new programming landscape. New OpenMP directives enable programmers to identify units of independent work, called tasks, leaving the scheduling decisions of how and when to execute them to the runtime system.

Some of the factors that determine the speedup of applications using the task paradigm are the number of tasks generated, by means of cut-off strategies that allow to prune task creation, and the scheduling of these tasks. Previous works have shown the importance of cut-off strategies in task parallel languages [11], [16], [17]. Many times the cut-offs are hard-coded into the application by the programmer. This critical decision can be time consuming and difficult to make as it may depend on factors such as the input data for the application.

We propose a runtime mechanism, that we call Adaptive Tasks Cutoff, which dynamically decides the most convenient cut-off for the application. This technique is based on profiling information collected at runtime in order to discover the granularity of the tasks created by the application and prune them appropriately using a cut-off.

We have implemented our proposal in our Nanos OpenMP environment (although the mechanism is not based in any OpenMP dependent feature) and evaluated with a set of diverse applications and benchmarks in an Altix system. Our results show that ATC, in most cases, is able to cut-off tasks adequately to achieve a good speed-up without the intervention of the programmer.

II. MOTIVATION AND RELATED WORK

Scheduling of tasks is a very well studied field. There are two main scheduler families that we use in our work: those that use breadth-first schedulers (see for example the work from Narlikar [3]) and those that use work-first schedulers with work-stealing techniques (see for example Cilk [4] and Acar et al. [5]).

Besides scheduling, task granularity is a key factor that determines how a task parallel program will scale. Dynamic aggregation is a common technique used by many parallel languages to increase the granularity of tasks [4], [6]–[8].

Most languages use some kind of task inlining (or lazy creation) to increase the granularity of the tasks. Inlined tasks retain the potential to spawn a full task if needed. The most common criteria to spawn a new task is based on the load of the processors (i.e., if some processor is idle) [4], [7], [9], [10] although others have explored using level-based or priority-based schemes [11].

Even lazy task creation has problems with very fine grained tasks [9] because there is still some overhead associated to create a task. Instead of inlining lazy tasks, another option that further reduces overhead is to serialize them. The decision to serialize a task is performed by a cut-off function that decides whether to continue creating parallelism or serialize the potential task. Many times this is done manually by the programmer. Figure 1 shows a Cilk program that computes all solutions to the N Queens problem for a board of size n. Lines 21 to 23 perform this task serialization after a given depth in the recursive tree is reached. The serial function nqueens_serial is exactly the same except that it has no Cilk keywords and that it always calls itself.

Figure 2 shows the speed-up obtained (in an Altix machine, see Section VI for more details) for the NQueens when we use the manual cut-off and we remove it (i.e. lines 21-23 are commented). We can observe that while with the manual cut-off the speed-up is close to the ideal, once the serialization is removed the performance drops drastically (it does not scale beyond a few threads). We are not proposing to replace lazy creation but that sometimes is more effective not to create a task even if the task creation mechanism is very efficient.
Another problem of task inlining is that it does not work well with OpenMP tasks with tied semantics [12] because tasks cannot be stolen by other threads [1].

Serializing tasks upon creation is not with its own problems. As noted by Krantz et al. [6], it can lead to load imbalance or deadlock because a wrong decision cannot be undone. Some use the load of a system’s thread as a cut-off [6], [13]. Another proposal uses the size of data structures [14] to control task creation but it depends upon the compiler understanding complex structures such as lists, which is difficult in the C or Fortran languages. Aharoni et al. use the number of elements of the structure at run-time to control granularity [15] but for non-uniform elements they also need compiler help. Rugina and Rinard generate a level based cut-off to serialize tasks in their automatic approach to divide and conquer algorithms [16].

The problem is that the appropriate cut-off depends on the application [12] so to solve the problem we need to incorporate information from the application. To solve this problem Prechelt et al. proposed the REAPAR framework [17] that uses information gathered during previous runs of the application to automatically generate the appropriate cut-off at compile time.

In our work we go one step further by doing the selection at runtime based on profile information while the application is running. The user does not need to modify his source, or even recompile, at all to benefit from our proposal as it is integrated in the OpenMP runtime library. To minimize the impact of the profiler we switch it off when enough information is gathered to make decisions about task creation. Note that this mechanism is orthogonal to lazy creation. Serializing is just another tool for the runtime to reduce the overhead in the appropriate cases.

### III. OpenMP Tasks

The latest specification of OpenMP (3.0) [1] has shifted from a thread–centric to a task–centric execution model, based on the fork-join paradigm where threads are execution vehicles that have access to a shared memory.

Version 3.0 introduces the `task` directive, which allows the programmer to explicitly specify a unit of parallel work called a task. Explicit tasks are useful for expressing unstructured parallelism and for defining dynamically generated units of work. The C/C++ syntax for a task construct is as follows:

```
#pragma omp task [clause[[],clause]] ... structured-block
```

A task construct may be lexically or dynamically nested inside an outer task construct. Tasks created by this construct may be executed immediately by any thread in the team or they may be deferred until a later time. The major difference from the OpenMP task model compared to other existing ones is that tasks are created tied by default. Tied tasks can be executed by any thread but if, after it execution begins the task is suspended, its execution can only be resumed by the thread to which was tied to. So, it is said that the task is tied to that thread. Using work-first semantics, like the Cilk scheduler, with tied tasks does not work well because the parent task that is suspended cannot be stolen by other threads and task generation is stalled in many circumstances leading to poor performance.

The `untied` clause allows the programmer to change this behavior if that does not cause any problem (e.g. use of threadprivate variables within untied tasks is discouraged). Untied tasks have no scheduling restrictions so they can be executed at any time by any thread.

The `taskwait` construct completes the task support. Taskwait suspends execution of the current task until completion of all of its child tasks. Fig. 3 shows a simple tree postorder tree traversal coded with OpenMP tasks.
```c
void traverse(node *p)
{
    if (p->left)
        #pragma omp task
        traverse(p->left);
    if (p->right)
        #pragma omp task
        traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

Fig. 3. Simple tree traversal with OpenMP tasks

The OpenMP specification also allows an implementation to serialize tasks and execute them immediately as part of the parent task (although they need to have their own data environment). This allows the runtime to implement cut-off techniques in order to reduce overheads by dynamically aggregating several tasks into a single one.

For example, the code shown in Fig. 3, with a `task` construct before each procedure call, could generate a task structure like the one shown in Fig. 4. The runtime could decide to aggregate them (the dashed areas in Fig. 4) by not creating some of the user specified tasks and instead executing them serially.

The remaining clauses of the `task` construct are related to the creation of a data environment for the new task (i.e. data scoping) and another that allows dynamic serialization of tasks based on a condition (i.e. `if` clause).

IV. PROFILING TASKS

To adaptively coalesce OpenMP tasks by employing a cut-off to prune excess parallelism, we needed to gather information about the application at runtime. To obtain this information we have implemented a dynamic profiler in our Nanos runtime [18]. The Nanos runtime is a research OpenMP runtime which implements most the major features. Nanos uses user-level threads, called nano-threads [19], on top of POSIX threads which are created when the application starts. For each OpenMP task that is spawned, a nano-thread is eagerly created. But, if a task is executed immediately only a small context is allocated in the nano-thread stack and the same nano-thread that encountered the task executes it.

Consequently, multiple OpenMP tasks can be executed by a single nano-thread.

We are interested in two kinds of information: the amount of work in each user specified task and the amount of work each nano-thread (or real task) has done.

In particular, we want to know from a given node in a recursive tree how much total work there is in the subtree spanning from that node (including the node itself). Note that regular loops with tasks are just a particular case of a recursive tree (i.e. with just one level of depth). This will allow us to predict how much work a future task created at the same level will do.

We keep track of the time each user OpenMP task spends running. As we are interested in the computational load we disable the timers at synchronization points (such as `taskwait`). We have two timer counters associated with each OpenMP task: one for the work load of the task itself and another for the time of all its descendants (i.e. its spanning tree). Then, we have another time counter for the total execution time of a nano-thread.

When an OpenMP task finishes, it processes its profile information. This processing consists of three steps:

1. The task adds its time to the total time of the nano-thread.
2. The task adds its time to the tree time of its OpenMP parent task. Note, that as the OpenMP parent task might be being executed by a different thread this update needs to be protected by mutual exclusion.
3. The task updates a shared depth-level indexed structure where we keep the average computational load of the subtree spanning from a given depth level in the recursive tree. This operation also needs to be performed under mutual exclusion.

As this post-processing can have a large impact in the application we have implemented three different profiling modes:

Full mode In this profiling mode, we collect all the information so we have a very accurate description of the application behavior.

Minimal mode In this profiling mode, we only collect the total time of the nano-thread. Thus, the overhead of the profiling is minimal as there are no updates from other threads in the same memory locations (which require extra synchronization). The problem with this mode is that it does not obtain enough information to feed an algorithm.

Adaptive mode In this profiling mode, the application starts in full profiling mode but it progressively switches to minimal mode to avoid overheads. When enough samples (a runtime parameter currently defined to 100) are collected at a given depth-level profiling in that level is switched to minimum.

The information obtained from this profiling is used to
predict the behavior of future tasks by our adaptive cut-off algorithm.

V. ADAPTIVE TASKS CUT-OFF (ATC)

In this section we present our adaptive tasks cut-off (ATC). The cut-off uses information obtained from the profiler to decide whether or not to allow the creation of a new task. ATC does not need any source code modification from the user. It will be invoked by the runtime whenever an OpenMP task is about to be created to decide whether to prune it or not.

A. Design objectives

In order to design our cut-off we have tried to achieve the following objectives that in our opinion will maximize the application performance:

1) Obtain profiling information quickly. For the cut-off to be effective, we need to obtain information from the profiler quickly. This means, we need to force threads to do a depth first execution as soon as possible to obtain information about the spanning trees.

2) Generate enough tasks for all threads. Obviously, we do not want threads to be idle unless absolutely necessary (i.e., unless creating tasks for them is too expensive). The goal is to generate enough tasks so the scheduler can avoid load imbalances by giving work to all the threads.

3) Do not allow an unbounded number of tasks. We want to limit the number of task created in order to reduce the resources allocated to the runtime.

4) Avoid fine-grain tasks. Doing this we try to reduce the overheads associated with task creation and only create tasks when their computational load pays off the overhead of creation. By reducing the overhead we improve the performance of the application.

Several of these objectives are contradictory (e.g. objectives #1 and #2). We have tried to balance them adequately in the design of the algorithm.

B. The ATC algorithm

Initially, as the cut-off still does not have information from the profiler we have a decision process that, based in our previous observations [12], in most cases it will maximize the profiler we have a decision process that, based in our design objectives. Tasks are allowed to be created if the following two conditions are met:

1) There are fewer ready tasks than twice the number of threads in a given level. This condition serves two purposes: first, it forces the threads to go deeper into the tree so the profiler get information (objective #1); second, it restricts the number of tasks to a bound number (objective #3) while allowing a minimum number of tasks to ensure that all threads will have work (objective #2).

2) The depth-level is less than a certain limit (defined to be 4 for us). This condition tries to be conservative in allowing the creation of deep tasks (which tend to be fine-grain, objective #4) at this stage of the execution.

Note that by combining both conditions, we only generate the top level tasks that are the most promising however we do not generate too many of them. Note that combining both condition, we are only generating top level tasks which are the most promising, but we do not generate so many of them that we later regret our decision.

Once the profiler has gathered information about a level an estimation of computational load that the task would have is computed. The estimation, currently, is the average computational load of all the sub-trees spanning from the level of the tree that has been profiled (which cannot be obtained with the minimal profile mode). This estimation assumes that all tasks of a given level will have a similar behavior. It is a very simple approach that can be changed in the future to use more powerful prediction techniques.

Fig. 5. Adaptive task cut-off pseudo-code

If the predicted grain size is smaller than a certain value (we use 1 millisecond\(1\)) then the task will not be created (objective #4). Otherwise, it can be created if there are not enough ready tasks for the threads to execute. This, again, ensures some bound on the number of tasks (objective #3) but generates sufficient parallelism for the threads (objective #2).

The overhead of making this decision can be very large in applications with very fine grain tasks. As an optimization, to reduce it, we allow the profiler in the adaptive mode to mark a level as closed. When all the samples of a level have been collected the profiler checks which is the estimated time for that level, and as it will be constant in the future (because no more samples will be collected), if the estimation determines the grain is to small the level is closed. This allows the cut-off to make a decision with just a comparison. Note, that closing a level does not preclude that all tasks in deeper level are cut off.

\(1\)This value was obtained through microbenchmarking of task creation
VI. Evaluation

A. Applications

We have used the following applications for the evaluation of our cut-off proposal:

- Strassen
  Strassen is an algorithm [20] for multiplication of large dense matrices. It uses hierarchical decomposition of a matrix.

- 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 \( n \times n \) chessboard such that none of the queens attacks any other. Most OpenMP tasks in NQueens are very fine grain (below a microsecond).

- Multisort
  Multisort is a variation of the ordinary mergesort, which uses a parallel divide-and-conquer merge-sort and a serial quicksort when the array is too small.

- Alignment
  This application aligns a number of \( N \) 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.

- 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.

- Floorplan
  The Floorplan kernel computes the optimal floorplan distribution of a number of cells. The algorithm is a recursive branch and bound algorithm. We hierarchically generate tasks for each branch of the solution space that is not pruned.

In all applications (except Alignment) we marked all tasks as untied and we removed all manual cut-offs provided by the programmer leaving total scheduling freedom. The Alignment application makes heavy use of threadprivate and, because of that, we could not mark the tasks as untied. For more information on the parallelization of these applications please check our previous work [21].

B. Experimental setup

We evaluated all the benchmarks on an SGI Altix 4700 inside a CPU partition (with its own memory) of 32 processors to avoid interferences with other running applications.

We compiled all applications with our Mercurium compiler [22] 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.

C. Profiler impact

The first question we want to address is how much overhead the profiler introduced. This will allow us to determine if it was worth the additional complexity of enabling and disabling the profiler instead of continuously running the profiler.

We used NQueens for this experiment because it has the finest-grained tasks of all the benchmarks. If the overhead of the profiler is low enough for NQueens it will most likely have a low impact for the others applications with coarser tasks.

![Fig. 6. Overhead of Queens (board of 13x13) with different profiling modes](image)

Fig. 6 shows the overheads of the different profile methods (compared against a non-profiled version) with a chess board of size 13x13. We can see that, while only profiling the nano-thread execution (the minimal profiling mode) time has almost no impact, profiling the whole tree structure (full profiling mode) severely reduces the performance obtained due to the overhead of the profiling. In fact, the overhead scales up with the number of threads due to contention to access the profiler structures. But, limiting the amount of samples we collect to one hundred and then disable the full profiling (adaptive profiling mode) reduces this impact to a minimum barely noticeable (with the overhead being at most around 4%).

Therefore, the adaptive profiling mode is the one we have used to obtain the information for our adaptive cut-off mechanism.

D. Cut-off evaluation

1) Methodology: We wanted to compare our adaptive cut-off with the best cut-off for each of the applications. First we looked for the best schedule and cut-off pair. We executed all the applications with the cross-product of combinations of schedulers and cut-offs (Table I and Table II summarize the schedulers and cut-offs we have used). For those cut-off that worked well we tried several variations with different parameters. We then fixed the schedule to the one obtained in the previous step and we also tried variations of the cut-offs that seemed to work worst to try to find the worst one.

For some applications we did these experiments exploring different input sizes to see how our algorithm performs with variations of the input data.

Table III summarizes the best and worst cut-off and the schedule (which was the one that obtained the best speed-up)
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>work-stealing</td>
<td>Work-first scheduler with work-stealing.</td>
</tr>
<tr>
<td>bff</td>
<td>Breadth-first scheduler (FIFO order).</td>
</tr>
<tr>
<td>bfl</td>
<td>Breadth-first scheduler (LIFO order).</td>
</tr>
</tbody>
</table>

**TABLE I**
SUMMARY OF USED SCHEDULERS

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxlevel=N</td>
<td>Tasks are cut off based on their depth (N is the level where they are cut).</td>
</tr>
<tr>
<td>maxtasks=N</td>
<td>Tasks are cut off based on the total number of tasks in the system (N per num-threads is the number of allowed tasks).</td>
</tr>
<tr>
<td>numready=N</td>
<td>Tasks are cut off based on the number of ready tasks in the system (N per num-threads in the number of allowed ready tasks).</td>
</tr>
<tr>
<td>load based</td>
<td>Tasks are only created if there is an idle thread at creation time.</td>
</tr>
</tbody>
</table>

**TABLE II**
SUMMARY OF USED CUT-OFFS.

used for application and input. Keep in mind that many times there are several cut-offs which are close to the best and worst ones. We consider this values as the bounds to measure the success of our adaptive cut-off: we would like to be as close as possible to the best cut-off and as far away as possible from the worst one.

<table>
<thead>
<tr>
<th>Application</th>
<th>Schedule</th>
<th>Best cut-off</th>
<th>Worst cut-off</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multisort (32 million integers)</td>
<td>work-stealing</td>
<td>max-tasks=8</td>
<td>max-level=3</td>
</tr>
<tr>
<td>Alignment (20 sequences)</td>
<td>bff</td>
<td>max-tasks=8</td>
<td>load-based</td>
</tr>
<tr>
<td>Alignment (100 sequences)</td>
<td>bff</td>
<td>max-tasks=8</td>
<td>num-ready=4</td>
</tr>
<tr>
<td>Strassen (1280x1280 matrix)</td>
<td>work-stealing</td>
<td>max-level=4</td>
<td>num-ready=4</td>
</tr>
<tr>
<td>Floorplan (15 cells)</td>
<td>bff</td>
<td>max-level=4</td>
<td>num-ready=1</td>
</tr>
<tr>
<td>Floorplan (20 cells)</td>
<td>bff</td>
<td>max-level=5</td>
<td>num-ready=1</td>
</tr>
<tr>
<td>Queens (13x13 board)</td>
<td>bff</td>
<td>max-level=3</td>
<td>max-tasks=8</td>
</tr>
<tr>
<td>Queens (14x14 board)</td>
<td>bff</td>
<td>max-level=3</td>
<td>max-tasks=8</td>
</tr>
<tr>
<td>SparseLU (50 blocks of 100x100 elements)</td>
<td>bff</td>
<td>num-ready=4</td>
<td>load-based</td>
</tr>
</tbody>
</table>

**TABLE III**
SUMMARY OF BEST AND WORST CUT-OFFS

2) *Results:* In the following results, we present for all the experiments the speed-up obtained without using any cut-off, with the best cut-off, with the worst cut-off, with our adaptive cut-off (labeled ATC) and with a work-stealing scheduler (with no cut-off and no lazy creation) for comparison purposes. For those applications (*multisort*, *nqueens* and *sparseLU*) that we have the corresponding Cilk code we also evaluated them. For each application we show the speed-up obtained and also the number of tasks created at each depth level with the different cut-offs plus the potential number of tasks at each level (i.e. the number of OpenMP tasks defined by the user). This allows us to observe the differences in behavior among the different cut-offs.

<table>
<thead>
<tr>
<th>Label</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nocutoff</td>
<td>Best schedule with no cut-off</td>
</tr>
<tr>
<td>worst</td>
<td>Best schedule with worst cut-off</td>
</tr>
<tr>
<td>ATC</td>
<td>Best schedule with ATC</td>
</tr>
<tr>
<td>best</td>
<td>Best schedule with best cut-off</td>
</tr>
<tr>
<td>work-stealing</td>
<td>Work-first schedule with no cut-off</td>
</tr>
<tr>
<td>cilk</td>
<td>Cilk version</td>
</tr>
</tbody>
</table>

**TABLE IV**
SUMMARY OF LABELS USED IN THE EVALUATION

Fig. 8 and Fig. 7 show the results for the *Multisort* and *SparseLU* applications respectively. We can see that ATC does a good job and there is almost no difference in speed-up with the best cut-off as the number of tasks created at each level are roughly the same. Both applications have coarse grain tasks and benefit the most by applying some bound to the number of tasks created.

Fig. 9 presents the performance obtained for the *Strassen* application. While ATC does not do a bad job, it opens up one more level (the 5th level) than the best cut-off and that reduces the achieved speed-up. We have checked the profile information and the size of the tasks at depth 5 are coarse enough to be executed. So, probably there are other factors we are not accounting for (e.g. amount of synchronization) that affect the identification of the right cut-off point. ATC is much better than some of the other cut-offs so we still see this as a positive result because is much closer to the best one than to the worst one.

In figures 10 and 11 we show the results for the *Alignment* benchmark with two data sets: a small one of 20 sequences and a much bigger one of 100 sequences. In both cases, ATC makes good decisions and controls the number of tasks created better than the best cut-offs. The size of the tasks with the small data set is much smaller but ATC does a good job dealing with them. Note also that the gap between the best and worst cut-offs is very narrow for this application. In fact, because *alignment* uses tied tasks, unlike in other applications, the critical factor is the schedule being used and not as much the cut-off. For example, note how, with both inputs, that when no cut-off is used both work-stealing and the best schedule (*bff* in this case) have the same amount of tasks to schedule. But, because tied task do not interact well with work-stealing the performance obtained is much lower.
The results for the Floorplan benchmark are shown in Fig. 12 and 13. We have also tried two input sets: one with 15 cells and another with 20 cells. The size of the problem increases exponentially with the size of the input and the number of potential tasks at some levels is over 40 million which causes severe performance problems for all the versions that do not have a cut-off (note that even the worst cut-off reduces drastically the number of created tasks) and we run out of CPU time in many executions. That is why there are no results for the nocutoff or work-stealing versions in many cases. Floorplan is a branch and bound algorithm and, as such, it is highly irregular in the size and number of tasks created at each level (see in Fig. 13(b) for the potential number of tasks at each level). Because of this, with one of the inputs ATC cuts the creation of tasks too soon which unbalances the application resulting in a less than optimal speedup. The scenario where there are many top level branches but a few of them accumulate most of the work load is, probably, the worst for our simple prediction technique. Even so, ATC manages to perform much better than the worst cut-offs (including several level-based cut-offs) and even the worst cut-offs reduce the amount of created tasks by a large amount. This allows the executions to finish which does not always happen with no cut-off.

Fig. 14 and 15 show the results for the NQueens benchmark with two inputs: a 13x13 board and a 14x14 board. Again, the size of the problem increases exponentially with the size of the input. Also, being a backtracking algorithm it is very irregular. However, as the pruning increases in a regular way as we progress down the task tree ATC does a good job in detecting it. And we can see it obtains the same performance than the best cut-off as they both cut task creation at level 3. The performance of the cut-offs not based on the depth level was so poor we were not able to obtain a result with more than a few CPUs because we ran out of CPU time (1 hour maximum) for a baseline run of a few minutes. The number of tasks generated for those cases are in the order of millions like in Floorplan but with even smaller granularity. Because the tasks are so fine, this benchmark allows us to verify that, with the optimizations we performed, the cut-off overhead can
be kept to a minimum.

Overall, ATC makes very good decisions. In all but two cases ATC does find an near-optimal cut-off and, in those two cases, its decisions are much better than some of the worst decision that a user can make. Note that in many scenarios when the level-based cut-offs obtain good results the ones based on number of tasks do not (and vice versa) while ATC works well overall. We think this results prove that is a good cut-off technique to be used to save time to the average user that does not have (or does not want) the time to explore all cut-off possibilities to find the optimal.

VII. CONCLUSIONS AND FUTURE WORK

We have presented an adaptive technique for task parallel languages, that we call Adaptive Tasks Cut-off (ATC), to reduce the number of created tasks. Tasks that are cut-off have no chance to be spawned (even lazily) thus reducing the overhead. This is particularly in the case of very fine grain tasks where we have seen that even lazy creation might be costly (e.g., N Queens). ATC uses dynamic profiling of the application to estimate the granularity of the tasks that are being created. The profiling is progressively switched off dynamically to reduce the overhead it causes.

Our evaluation, with a set of applications with very different properties, shows that ATC, in most cases, correctly discovers the granularity of the tasks and it decides the cut-off them appropriately. In all cases it behaves much better than other possible cut-offs that can be selected by inexperienced (or careless) users. This suggests ATC is a good option for both a naïve user and as a default in a parallel runtime.

Although we have implemented the ATC cut-off in the context of OpenMP, we think that the technique is general enough, and by no means tailored to OpenMP, so it can be useful in other languages with support for task parallelism (particularly with fine grain tasks).

The profiler and the estimation is very naïve and needs to be extended, in the future, in several ways. First, it should allow to maintain the characterization of different tasks separately so different decisions can be made in function if is one kind of task or another. The address of the task code can be used to identify the different kinds of tasks. Second, in some
applications the same kind of task may change its behavior multiple times through the lifetime of the application. Once the detailed sampling is disabled, the profiler can use the information obtained in the minimal mode to verify that the behavior is consistent with the prediction. If an inconsistency is found then the detailed profiling can be enabled again to collect samples again. Last, the prediction technique should be extended to deal better with unbalanced codes like Floorplan.

Other lines of future work would try to extend the evaluation with more applications. It would also be interesting to study how the quality of the prediction relates to the number of sample. Another direction we would like to investigate is the resource usage by the different cut-offs because although the performance may be similar in some cases, the number of concurrent tasks (and thus the system resources usage) can be quite different. This might be important in environment where resources are scarce.

ACKNOWLEDGMENTS

This work has been supported by the Ministry of Education of Spain under contract TIN2007-60625, the European Commission in the context of the SARC integrated project #27648 (FP6), the HiPEAC Network of Excellence (IST-004408) and the MareIncognito BSC-IBM agreement.

REFERENCES

Fig. 13. Results for Floorplan (20 cells)

Fig. 14. Results for N Queens (13x13 board)


Fig. 15. Results for N Queens (14x14 board)