Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP

Alejandro Duran, Xavier Teruel, Roger Ferrer
Computer Sciences Department
Barcelona Supercomputing Center
Jordi Girona, 31, Barcelona, Spain.
{alex.duran,xavier.teruel,roger.ferrer}@bsc.es

Xavier Martorell, Eduard Ayguadé
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona, 1-3, Barcelona, Spain.
{xavim,eduard}@ac.upc.edu

***Abstract*—Traditional parallel applications have exploited regular parallelism, based on parallel loops. Only a few applications exploit sections parallelism. With the release of the new OpenMP specification (3.0), this programming model supports tasking. Parallel tasks allow the exploitation of irregular parallelism, but there is a lack of benchmarks exploiting tasks in OpenMP.**

With the current (and projected) multicore architectures that offer many more alternatives to execute parallel applications than traditional SMP machines, this kind of parallelism is increasingly important. And so, the need to have some set of benchmarks to evaluate it.

In this paper, we motivate the need of having such a benchmarks suite, for irregular and/or recursive task parallelism. We present our proposal, the Barcelona OpenMP Tasks Suite (*BOTS*), with a set of applications exploiting regular and irregular parallelism, based on tasks.

We present an overall evaluation of the *BOTS* benchmarks in an Altix system and we discuss some of the different experiments that can be done with the different compilation and runtime alternatives of the benchmarks.

***Index Terms*—OpenMP, benchmark suite, task parallelism**

I. Introduction and Motivation

Multicore processors, both in homogeneous and heterogeneous environments, pose new challenges in the evaluation of application performance and programmer productivity. The increased density of processing cores radically changes resource availability, communication costs, data placement and locality management. Such changes allow to execute applications with a much diversity schemes and scheduling options than before.

New architectural features are available in such environments. Their use can be of high complexity. For this reason, most of traditional compilation environments fail to obtain high performance on such environments. By now, exploitation of such features results in complex programming and large time investments from programmers.

Parallel to the development of multicore processors, the latest OpenMP specification (3.0)[1] introduced a new execution model for task parallelism to address the needs to express parallelism in irregular applications, which seems to reduce the complexity of programming multicores.

We think that we are facing a great opportunity to develop new schemes supporting irregular parallelism, and different ways to execute applications. For these reasons, a new set of benchmarks is needed to evaluate all alternatives that programmers will be able to exploit with new advanced features provided by the programming model.

This paper presents a collection of applications, the Barcelona OpenMP Tasks Suite (*BOTS*), that makes use of the new task parallelism in OpenMP. Our aim is to provide a basic set of applications that will allow researchers and vendors alike to evaluate OpenMP implementations, and that can be easily ported to other programming models. And an additional goal is for the OpenMP community to have a set of examples using the tasking model.

II. Related Work

There are a number of OpenMP benchmark suites in the literature, including OpenMP microbenchmarks, kernels and applications, namely EPCC microbenchmarks[2], the LLNL OpenMP Performance Suite[3], the OpenMP Source code Repository (OmpSCR)[4], PARSEC[5], NAS[6], [7], [8], and SPEComp[9] benchmarks.

The EPCC microbenchmarks[2] are designed to measure the overhead of OpenMP directives and clauses on different platforms. There is a microbenchmark for each one of the features of OpenMP, from parallel support and synchronization to loop scheduling. They have been used in a number of publications to evaluate different OpenMP implementations.

A similar approach is taken by The LLNL OpenMP Performance Suite[3], which also includes a set of microbenchmarks to evaluate the overhead of the directives and clauses.

The OmpSCR[4] contains a total of 12 benchmarks, ranging from computing PI and QuickSort, to a molecular dynamics application. The PARSEC[5], [10] benchmark suite includes 4 benchmarks (out of 12) parallelized with OpenMP, including body tracking, simlarity search, and an association rule mining application. The NAS benchmarks[7] are a collection of 7 kernels (EP, IS, MG, CG, DC, FT, UA) and 3 applications (BT, SP, and LU). Most of them are written in Fortran, and include versions in MPI and OpenMP. There is a version written in C with OpenMP from the OMNI Compiler Project[11]. SPEComp[9], distributed by the Standard Performance Evaluation Corporation, includes 9 Fortran applications, and 2 C applications, parallelized with OpenMP constructs.

OpenMP applications in OmpSCR, PARSEC, NAS, and SPEComp suites are mostly regular, and parallelism is exploited based on loops, with only a pair of applications exploiting parallelism based on sections (*sort* in OmpSCR, and *galgel* in SPEComp), and none in PARSEC and the NAS benchmarks.

Exploiting tasking in OpenMP has been evaluated in several proposals. Intel Task Queues[12] used a set of 4 benchmarks written using this style of programming: Strassen[13], FFT, Queens and Multisort. The last 3 originated from the Cilk benchmarks[14]. The current task definition has been evaluated with some of these benchmarks, which have been rewritten to adopt the new syntax[15], [16], [17], [18].

Other interesting benchmark suites include SPARK [19] and Lonestar [20].

SPARK contains a number of sparse algorithms based on techniques like finite elements, direct solvers and eigenvalue problems, nonlinear systems of equations, differential algebraic equations, and finite differences. It targets the evaluation of the computing power of a given architecture.

Finally, Lonestar contains a collection of widely–used real–world sequential applications that exhibit irregular behaviour, but contain a significant amount of amorphous data parallelism. They are intended to serve as examples of data–parallel programs to which a programmer might apply various parallelization techniques. Some examples are clustering algorithms, Barnes-Hut N-Body simulations, mesh refinement, and survey propagation.

III. Suite overview

The OpenMP definition of the tasking model leaves a lot of freedom to vendors in how this model should be implemeted. For example, it places few restrictions on task scheduling or it does not specify whether or not task switching should be supported. Our aim was to provide a collection of benchmarks that would allow vendors to test the impact of different implementation decisions in a multicore architecture.

A. Methodology

While a few of the benchmarks are in-house developments, most of them are versions of publicly available benchmarks from either the Cilk project[14], the Application Kernel Matrix project[21] or the Olden suite [22], which we have ported to OpenMP in a coherent benchmark framework.

a) Multiple versions: Because at this point the different trade-offs of the OpenMP tasking model are still not clear and depend very much on the quality of the implementation, we have developed different versions of each benchmark with different characteristics:

```
#pragma omp task if(condition)
    work();
```

Fig. 1. Cut-off implemented with an `if` clause

```
if (condition)
#pragma omp task
    work();
else
    work_sequential();
```

Fig. 2. Manually implemented task cut-off

- All benchmarks come with versions with `tied` and `untied`[1] tasks that allow to experiment how the implementation behaves with both kinds of tasks.
- Many of the benchmarks create a very large number of small tasks. Because of this, we have developed three different versions of those benchmarks in which controlling the amount of parallelism is important:
  - one that does not limit task creation and puts all the burden on the implementation. This would be the ideal from the programmmer perspectite because, potentially, the implementation could limit task creation by itself. It remains to be seen how effective implementions will be doing that.
  - another where the application controls task creation by means of an `if` clause on the `task` directive (see Fig. 1). The exact condition varies from benchmark to benchmark but it usually dependent on the depth in a recursion path.
  - another where the application controls task creation manually by calling a function with task directives or without them based on the same condition as in the previous version (see Fig. 2).
- Some benchmarks allow for either multiple generators (i.e., `tasks` under a `for`/`sections` construct) or a single generator (i.e., `tasks` under a `single` construct). In those cases, versions of the same benchmarks under both approaches have been developed to evaluate the support for both.

b) Handling indeterminism: It is common that task parallelism by its irregular nature presents some kind of indeterminism in its execution (e.g. pruning in search algorithms). Because indeterminism does not fit well with benchmarking, applications with indeterminism are usually avoided. We think it is important to incorporate this kind of applications in our suite as they represent legitimate uses of task parallelism. In these cases, we have tried to keep the indeterminism under control by slightly modifiying the application behavior. Because the approaches are different, we comment each case individually in the next section.

c) Self-verification: Self-verification is another important characteristic in any benchmark as it allows to test whether implementations or specific optimizations implement the correct

[1]`tied` impose certain restrictions on scheduling (e.g. no thread switching), while `untied` have no restrictions.

semantics. As such, all benchmarks come with one of the three following verification methods:

- In those cases where possible, benchmarks apply some validation method to the output.
- In some other, we have included validation data in the input data so the benchmark can validate its output against it.
- When not possible to apply any of the two previous methods, a serial version of the benchmark is also executed when the user requests a validation and the result is compared against that from the parallel execution.

d) Input sets: For each application in the suite we have defined a set of different data inputs to test the applications under different scenarios:

test	The test class is very small. Such input should be used only to quickly check that benchmarks work.
small	The small input data set is designed so that neither the overall memory requirements go over 1 Gb., nor the serial execution time is greater than one minute in our reference platform[2].
medium	The medium data set is designed so that neither the overall memory memory requirements go over 4 Gb., nor the serial execution time is over ten minutes in our reference platform.
large	The large input data set contains the inputs with larger memory requirements (up to 10 Gb.) and larger serial execution times (up to half-hour).

B. Applications

A short description of the benchmarks[3] that form the *Barcelona OpenMP Tasks Suite* follows:

e) Alignment*:* aligns all protein sequences from an input file against every other sequence using the *Myers and Miller*[23] algorithm. The alignments are scored and the best score for each pair is provided as a result. The scoring method is a full dynamic programming algorithm. It uses a weight matrix to score mismatches, and assigns penalties for opening and extending gaps. The output is the best score for each pair of them.

In this application, we parallelized the outer loop with an `omp for` worksharing with tasks created inside this parallel loop. This allows the implementation to *break* the iterations when number of threads is large compared to the number of iterations and when there is imbalance. To be able to use `untied` tasks we moved several global variables in the original version,used as temporal space, to local variables.

f) FFT*:* computes the one-dimensional *Fast Fourier Transform* of a vector of n complex values using the *Cooley-Tukey* [24] algorithm. This is a divide and conquer algorithm that recursively breaks down a *Discrete Fourier Transform* (DFT) into many smaller DFT's. In each of the divisions multiple tasks are generated.

[2]An SGI Altix 4700 system.

[3]This list may grow as we are still exploring new benchmarks.

g) Fibonacci*:* computes the nth fibonacci number using a recursive paralellization. While not representative of an efficient fibonacci computation it is still useful because it is a simple test case of a deep tree composed of very fine grain tasks. It comes with versions that use a cut-off based on the depth of the tree (i.e after a certain level it will not generate more tasks) to avoid the creation of very fine grained tasks.

h) Floorplan*:* kernel computes the optimal floorplan distribution of a number of cells. The algorithm gets an input file with cell's description and it returns the minimum area size which includes all cells. This minimum area is found through a recursive branch and bound search. We hierarchically generate tasks for each branch of the solution space. The state of the algorithm needs to be copied into each newly created task so they can proceed. This implies that additional synchronizations have been introduced in the code to maintain the parent state alive.

The application comes with a pruning mechanism to reduce the search space. This pruning is very irregular and very aggressive and, as a result the tree is heavily unbalanced. The pruning is based on the best result found up to that moment which generates a source of indeterminism. Because all nodes of the tree have roughly the same computational load, we compute the total number of nodes visited to find a solution. With this metric different versions and optimizations can be evaluated as the number of nodes per second should increase if the comptutation is more eficient (e.g., with more threads) even if it takes more time to find a solution due to the indeterminism.

As *Fibonacci*, *Floorplan* comes with versions that have a cut-off based on the depth of the tree to avoid creating fine grain tasks.

i) Health*:* simulates de Columbian Health Care System[25]. It uses multilevel lists where each element in the structure represents a village with a list of potential patients and one hospital. The hospital has several double-linked lists representing the possible status of a patient inside it (waiting, in assessment, in treatment or waiting for reallocation). At each timestep all patients are simulated according with several probabilities (of getting sick, needing a convalescence treatment, or being reallocated to an upper level hospital). A task is created for each village being simulated. Once the lower levels have been simulated synchronization occurs. *Health* comes with a cut-off mechanism based on the village level in the hierarchy.

The probabilities in the different steps of the simulation represent a source of indeterminism. To avoid it we have used, instead of a single seed for random numbers, one seed for each village. This way all the probabilities inside each village (which are computed by a single task) will be the same across different executions and not affected by other tasks.

j) N Queens*:* 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 attack any other. It uses a backtracking search algorithm with pruning. A task is created for each step of the solution. As, in *Floorplan*,

Application	Origin	Domain	Computation structure	# of task directives	tasks inside omp...	nested tasks	Application cut-off
Alignment	AKM	Dynamic programming	Iterative	1	for	no	none
FFT	Cilk	Spectral method	At leafs	41	single	yes	none
Fib	-	Integer	At each node	2	single	yes	depth-based
Floorplan	AKM	Optimization	At each node	1	single	yes	depth-based
Health	Oden	Simulation	At each node	1	single	yes	depth-based
NQueens	Cilk	Search	At each node	1	single	yes	depth-based
Sort	Cilk	Integer sorting	At leafs	9	single	yes	none
SparseLU	-	Sparse linear algebra	Iterative	4	single/for	no	none
Strassen	Cilk	Dense linear algebra	At each node	8	single	yes	depth-based

TABLE I
BOTS APPLICATIONS SUMMARY

the parent state needs to be copied to the children tasks which introduces additional synchronizations. *NQueens* prunes those branches that will not find a correct answer. This generates some degree of unbalance in the tree. The pruning introduces some indeterminism, but not as much as in Floorplan because it does not depend on any current solution, in the number of nodes to be visited. To avoid it, instead of just finding one solution to the problem, this kernel will find all possible solutions. This guarantees that the application has always the same computational load. To count all the solutions found by different tasks one approach is to surround the accumulation with a critical directive but this would cause a lot of contention. To avoid it, we used `threadprivate` variables. In this way, all threads can acumulate the solutions they find. Each thread reduces the variable, within a `critical` directive, to the global variable at the end of the parallel region.

k) Sort*:* sorts a random permutation of n 32-bit numbers with a fast parallel sorting variation [26] of the ordinary mergesort. First, it divides an array of elements in two halves, sorting each half recursively, and then merging the sorted halves with a parallel divide-and-conquer method rather than the conventional serial merge. Tasks are used for each split and merge. When the array is too small, a serial quicksort is used so increase the task granularity. To avoid the overhead of quicksort, an insertion sort is used for very small arrays (below a threshold of 20 elements).

l) SparseLU*:* computes an LU matrix factorization over sparse matrices. A first level matrix is composed by pointers to small submatrices that may not be allocated. Due to the sparseness of the matrix, a lot of imbalance exists. Matrix size and submatrix size can be set at execution time. While a `dynamic` schedule can reduce the imbalance, a soultion with tasks paralellism seems to obtain better results[17]. In each of the sparseLU phases, a task is created for each block of the matrix that is not empty.

We developed two different versions of the benchmark, one that generates all the tasks from inside a `single` worksharing and another that uses a `omp for` worksharing to allow multiple threads to create the tasks for each phase.

m) Strassen*:* algorithm uses hierarchical decomposition of a matrix for multiplication of large dense matrices[13]. Decomposition is done by dividing each dimension of the matrix into two sections of equal size. For each descomposition a task is created. To avoid the creation of many small tasks, we developed versions with depth based cut-offs.

n) Summary: TABLE I briefly summarizes the applications that we have presented, while TABLE II shows some characteristics of the different benchmarks when executed with the *medium* input class. These numbers were collected from a serial execution in our reference system of a specially profiled version where the compiler added additional code to obtain this information[4]. Most columns are self-explicative but some require some clarification: *Captured environment* refers to the amount of data that is copied from parent tasks to their children upon creation (i.e. `firstprivate` variables or memcpy from the parent memory by a child task). Non-private writes refer to writes that do not reference a task private variables and, thus, can be affected by locality decisions. Table II shows the percentage of writes which are non-private and the average number of arithmetic operations between two of such writes.

From this profiling we can see that the benchmarks have different characteristics. Some applications have a large amount of very fine-grained tasks (e.g., *Fib*, *NQueens*, *Floorplan*) where the challenge is exploiting the available parallelism while reducing the associated overheads. In other cases, there are relatively few coarse tasks (e.g., *Alignment*, *sparseLU*) and the challenge for the implementation is to avoid load balance situations.

We can also see that many are memory-bound applications (i.e., low Operations per write) but that in many of the benchmarks most memory accesses are to the private memory of the task (low % of writes to non-private memory). This indicates that careful allocation of the private memory with respect to where the task is executed (including data migration if the task migrates from one thread to another) may yield important improvements (see for example *Alignment* for the difference between Operations per write and Operations per non-private write).

Another important characteristic is that is profiled is the amount of data that is communicated from the parent to its child tasks at creation. We can see that except in one

[4]Note that this information is not obtained from performance counters, but from actual operations which are independent of the architecture.

					Average per task						
Application	Input	Serial time	Memory size	Number of potential tasks	Arithmetic operations	Taskwaits	Captured environment size (in bytes)	# of Writes to captured environment	% of writes to non-private data	Operations per write	Arithmetic operations per non-private write
Alignment	100 proteins	44.4 s	4.7GB	4950	$\simeq$ 14 M	0.00	16	0.00	0.03%	1.88	7 K
FFT	128M floats	98.73 s	3 GB	$\simeq$ 10 M	$\simeq$ 2K	0.18	37.22	0.00	3.49%	1.40	40.11
Fib	50	140 s	3.2 MB	$\simeq$ 40 G	2.50	0.50	4	0.00	100 %	2.50	2.50
Floorplan	20 shapes	37.38s	3 MB	$\simeq$ 67 M	90.78	0.15	$\simeq$ 5 Kb	5.00	74.10%	1.53	2.06
Health	4 levels with 38 cities each	137 s	4 GB	$\simeq$ 17 M	293.72	0.03	8.00	0.00	12.33%	1.74	14.13
NQueens	14x14 board	73 s	3 MB	$\simeq$ 377 M	463.70	0.07	42.32	1.07	0%	4.75	-
Sort	128M integers	39.17 s	2 GB	$\simeq$ 2 M	$\simeq$ 8 K	0.45	39.91	0.00	25.13%	1.30	5.18
SparseLU	7500x7500 sparse matrix of 100x100 blocks	770 s	120 MB	39480	$\simeq$ 11 M	0.00	11.71	0.00	49.46%	5.95	12.03
Strassen	8192x8192 matrix	486.94 s	4GB	$\simeq$ 1 M	$\simeq$ 800 K	0.14	37.71	0.00	8.36%	2.63	31.49

TABLE II
APPLICATION CHARACTERISTICS WITH THE MEDIUM INPUT SETS

case (i.e., *Floorplan*)) the amount of communication is rather small (i.e., under 45 bytes on average). This seems to suggest that implementations that pre-allocate small memory areas associated with tasks descriptors might avoid to allocate in most case any data related to `firstprivate` and thus reducing the creation overheads.

Finally, we can see that in some applications (e.g., *Fib*, *Floorplan* and *SparseLU* shared access dominate the memory operations. Not all of them are necessarily shared with multiple task. For example, in *Fib* all shared access are writes to the parent task stack (in OpenMP tasks results are returned through `shared` variables). Trying to allocate in parent and child tasks in the same processor (a common technique) should provide benefits in this cases. In other cases, being able to improve shared data reuse between different task (e.g., task A writes some shared data that will be used by task B) remains a challenge because the runtime does not have enough information.

IV. EVALUATION EXAMPLE

In this section, we show the kind of evaluation and experiments that we think can be conducted with the suite that we have presented. Because of space limitations we have chosen a small subset of aspects that can be analyzed through the suite.

All the benchmarks were executed on a SGI Altix 4700 with 128 processors running on a cpuset of 32 processors to avoid interferences with other applications. The compiler used is the *Intel C Compiler* version 11.0. In all the cases we have used the optimization $-O3$ level. We have executed all the different versions of each application with the medium input set previously described in TABLE II. We computed all the *speed-up*s using the serial time as the baseline except for the *Floorplan* application where the *speed-up* represents the improvement in nodes executed per second instead of execution time[5].

[5]Even so, we have observed that the execution time scales very similarly.

In the following sections, we show some examples of possible evaluation with *BOTS*. First, a general evaluation of the benchmarks. Then, a study of different cut-off mechanisms and of the differences among the use of `tied` and `untied` tasks. We show the results obtained and we discuss how such aspects can impact the OpenMP programming model implementation. Other interesting aspects to study with our benchmark suite are finally discussed in Section IV-D.

A. Overall evaluation

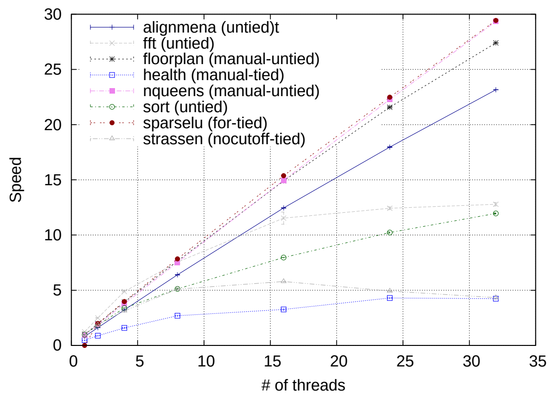

Fig. 3. Benchmark suite results as base code.

Fig. 3 shows the *speed-up* of the best version for each of the applications (in parenthesis, we indicate which is the best version). These results give an idea of the performance behavior for each application. We have applications (*NQueens* or *SparseLU*) which have an almost linear speed-up and other applications (*Strassen*, *Health* or *FFT*) which quickly reach a saturation phase.

B. Cut-off mechanism comparison

Due to the recursive nature of some benchmarks (see Section III-B) we can group cut-off mechanisms into two groups: first, we include cut-off mechanisms which are based on the task depth (i.e. the recursion level). Such kind of cut-off is usually implemented in the application itself. Our benchmark suite implements, when possible, these cut-off mechanisms. In the second group, we can find cut-off mechanisms based on the total number of tasks already created, the number of tasks ready to be executed, etc. Such pruning mechanisms can be easily implemented in the OpenMP runtime itself.

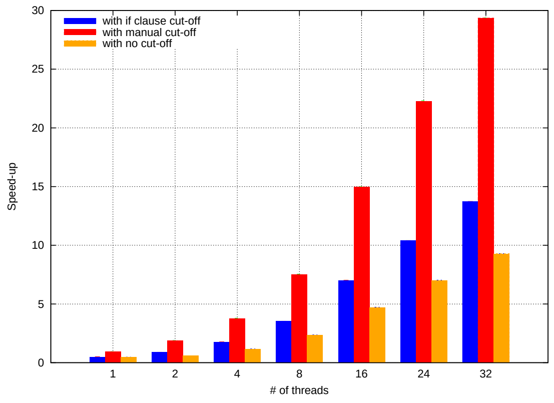

Fig. 4. Queens benchmark using different cut-off mechanisms.

Fig. 4 shows the *speed-ups* obtained using these different cut-offs for the *NQueens* benchmark:

- *manual cut-off*: prunes the generation of tasks in the application code itself. Compiler and runtime are not aware of the possibility of creating a task or not.
- *pragma if cut-off*: uses the OpenMP clause *if*, as a part of the task creation directive *task*. When the condition evaluates to false the task will not be created. But, the runtime still has to do some management in order to keep consistency (e.g. task hierarchy and dependence in order to execute properly a `taskwait`).
- *no-cutoff*: the application does not provide a cut-off and only the one implemented by the runtime (if any) is in use. The Intel Compiler uses a cut-off based on the number of tasks.

We can see in the results that, with the Intel Compiler, programming a manual cut-off is more effective than using an `if` clause, or relying on their runtime cut-off. Being a very new compiler these results were expected. Hopefully, as the task implementations mature these differences will disappear, thus reducing the burden on the programmer.

C. Tied vs. untied tasks

The OpenMP programming model specifies that tasks can be labeled with the `untied` clause, establishing two different kinds of tasks: tied and untied. A `tied` task is a task that, when it is suspended, can be resumed only by the same thread that suspended it, whereas `untied` tasks can be resumed by any thread. Tiedness of a task does not only imply which thread can resume a task but it also implies some task scheduling constraints which can also impact on the application performance.

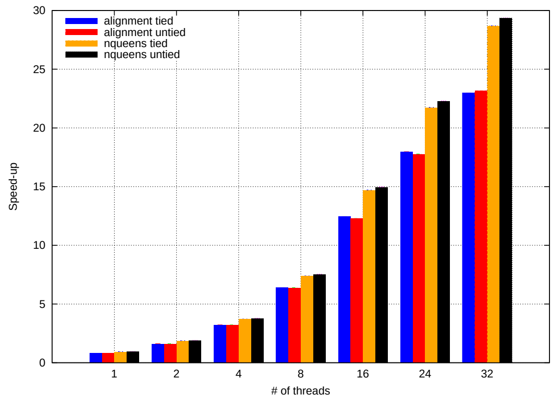

Fig. 5. Benchmark suite results using tied and untied tasks.

The suite comes with versions for all applications with `tied` and `untied` tasks to compare their behavior. Fig. 5 shows the results obtained using `tied` and `untied` tasks with the *Alignment* and *NQueens* benchmarks. Results are similar with both versions. Although a deeper analysis will be needed, the results suggest two main hypothesis:

- The Intel Compiler does not implement *thread switching* and thus untied tasks cannot benefit from this feature which should avoid imbalances. This is particularly evident in the *Alignment* benchmark which has been reported to scale nicely[27].
- Task scheduling constraints do not seem to impact significantly the performance results (at most there is a 4% difference between the versions). The other applications show a similar behavior.

D. Other opportunities for analysis

The Intel Compiler does not implement mechanisms that allow the user choose among different task scheduling policies but other OpenMP compilers exist[28], [16] that have such capabilities. One interesting study is to find how task scheduling policies (and how they can mantain locality across tasks) can affect the performance results of the benchmarks of the suite.

In previous sections, we have discussed how implementing a cut-off mechanism can affect application performance but we have not discussed, due to space limitations, how the different cut-off values (i.e., at which point in the recursion we cut) relate with the creation of parallelism and the overall performance. Choosing a low cut-off value can restrict parallelism opportunities but choosing a high cut-off value can saturate the system with a large amount of tasks which have no thread available to execute them. The right choice depends many times of the input data set. Comparing the application behaviour using different cut-off values or testing runtime features which allow to modify dynamically the cut-off mechanism[27] can also be interesting analyses.

The quality of implementations for different task generation schemes (e.g., in the *SparseLU* benchmark, which can use a single or multiple generator scheme), taskwait constructs, or other task related implementation details could also be analyzed with our benchmark suite proposal.

V. Conclusions and Future work

We have presented *BOTS* (*Barcelona OpenMP Task Suite*), built with the double motivation of coping with the great characteristics of the multicore processors, and offer a set of benchmarks to evaluate OpenMP tasking. We think that *BOTS* will help implementors and programmers to have a better understanding of the OpenMP tasking model, and its performance implications.

Each of these benchmarks comes also with different versions to test different aspects of the tasking model. For example they can be used to evaluate task scheduling alternatives, tiedness…Also, a number of input sets are provided, so that benchmarks can be used as tests, or really stress the processors and memory system in your machine.

It is interesting to note that we have tried to select benchmarks with diverse characteristics. In this paper, we have highlighted the differences, and we have shown their evaluation on an SGI Altix machine, with up to 32 processors and we report some of their characteristics per task (e.g., operations, memory writes…). Their evaluation also shows that there is plenty of work to do at all levels (architecture, compiler, runtime system, programming model) to improve certain benchmarks given that their current scalability is very limited. This suite can be used to obtain useful data of the strenghts and weaknesses of an OpenMP implementation, that can help developers to improve it.

Currently, we are working to add new benchmarks to the suite to cover more problem domains and scenarios. We are, as well, planning to do a full cross-vendor evaluation to find which is the current state of the OpenMP tasking implementations.

Acknowledgments

This research was supported by the Spanish Ministry of Science and Innovation (contracts no. TIN2007-60625 and CSD2007-00050), the European Commission in the context of the SARC project (contract no. 27648), the HiPEAC Network of Excellence (contract no. IST-004408), the IBM CAS Program and the Mare Incognito project under the BSC-IBM collaboration agreement.

References

[1] O. ARB, "OpenMP Application Program Interface, v. 3.0," May 2008.

[2] J. M. Bull, "Measuring Synchronization and Scheduling Overheads in OpenMP," in *First European Workshop on OpenMP*, September 1999.

[3] "LLNL OpenMP Performance Suite Description," 2001. [Online]. Available: https://computation.llnl.gov/casc/RTS_Report/openmp_perf.html

[4] A. Dorta, C. Rodriguez, F. de Sande, and A. Gonzalez, "The OpenMP Source Code Repository," *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, vol. 0, pp. 244–250, 2005.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 72–81.

[6] H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance," NASA Ames Research Center, Technical Report NAS-99-011, 1999. [Online]. Available: citeseer.ist.psu.edu/408248.html

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *The International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, Fall 1991. [Online]. Available: citeseer.nj.nec.com/bailey95nas.html

[8] H. Jin and R. F. V. der Wijngaart, "Performance Characteristics of the Multi-zone NAS Parallel Benchmarks," *J. Parallel Distrib. Comput.*, vol. 66, no. 5, pp. 674–685, 2006.

[9] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance," *Lecture Notes in Computer Science*, vol. 2104, pp. 1 – 10, 2001. [Online]. Available: citeseer.nj.nec.com/aslot01specomp.html

[10] C. Bienia, S. Kumar, and K. Li, "PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors," *IEEE International Symposium on Workload Characterization 2008*, pp. 47–56, 2008.

[11] K. Kusano, S. Satoh, and M. Sato, "Performance Evaluation of the Omni OpenMP Compiler," in *Prooceedings of the Third International Symposium on High Performance Computing*, 2000, pp. 403–414.

[12] S. Shah, G. Haab, P. Petersen, and J. Throop, "Flexible Control Structures for Parallellism in OpenMP," in *1st European Workshop on OpenMP*, September 1999.

[13] P. C. Fischer and R. L. Probert, "Efficient Procedures for Using Matrix Algorithms," in *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1974, pp. 413–427.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation*, 1998, pp. 212–223.

[15] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang, "A Proposal for Task Parallelism in OpenMP," in *Proceedings of the 3rd International Workshop on OpenMP*, Beijing, China, June 2007.

[16] X. Teruel, X. Martorell, A. Duran, R. Ferrer, and E. Ayguadé, "Support for OpenMP Tasks in Nanos v4," in *CAS Conference 2007*, October 2007.

[17] E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel, "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.

[18] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP Task Scheduling Strategies," in *Proceedings of the 4th International Workshop on OpenMP*, 2008.

[19] H. L. van der Spek, E. M. Bakker, and H. A. Wijshoff, "Characterizing the performance penalties induced by irregular code using pointer structures and indirection arrays on the intel core 2 architecture," in *Computing Frontiers 2009*, May 2009.

[20] M. Burtscher, P. Carribault, M. Kulkarni, K. Pingali, C. Cascaval, and C. von Praun, "Lonestar benchmark suite," http://iss.ices.utexas.edu/lonestar/, 2009.

[21] B. Chamberlain, J. Feo, J. Lewis, and D. Mizell, "An Application Kernel Matrix for Studying the Productivity of Parallel Programming Languages," in *W3S Workshop - 26th International Conference on Software Engineering*, May 2004, pp. 37–41.

[22] M. C. and A. Rogers, "Software Caching and Computation Migration in Olden," 1995.

[23] G. Myers and S. Selznick and Z. Zhang and W. Miller, "Progressive Multiple Alignment with Constraints," in *RECOMB '97: Proceedings*

of the first annual international conference on Computational molecular biology, New York, NY, USA, 1997, pp. 220–225.

[24] J. Cooley and J. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, vol. 19, pp. 297–301, 1965.

[25] S. R. Das and R. M. Fujimoto, "A Performance Study of the Cancelback Protocol for Time Warp," *SIGSIM Simul. Dig.*, vol. 23, no. 1, pp. 135–142, 1993.

[26] S. G. Akl and N. Santoro, "Optimal Parallel Merging and Sorting Without Memory Conflicts," *IEEE Transactions on Computers*, vol. 36, no. 11, pp. 1367–1369, 1987.

[27] A. Duran, J. Corbalán, and E. Ayguadé, "An Adaptive Cut-off for Task Parallelism," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008.

[28] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos Mercurium: a Research Compiler for OpenMP," in *Proceedings of the European Workshop on OpenMP 2004*, October 2004.