Evaluation of OpenMP Task Scheduling Algorithms for Large NUMA Architectures

Jérôme Clet-Ortega, Patrick Carribault, and Marc Pérache

CEA, DAM, DIF F-91297, Arpajon, France
{jerome.clet-ortega, patrick.carribault, marc.perache}@cea.fr

Abstract. Current generation of high performance computing platforms tends to hold a large number of cores. Therefore applications have to expose a fine-grain parallelism to be more efficient. Since version 3.0, the OpenMP standard proposes a way to express such parallelism through tasks. Because the task scheduling strategy is implementation defined, each runtime can have a different behavior and efficiency. Notwithstanding, the hierarchical characteristic of current parallel computing systems is rarely considered. This might come down to a loss of performance on large multicore NUMA systems. This paper studies multiple task scheduling algorithms with a configurable scheduler. It relies on a topology-aware tree-based representation of the computing platform to orchestrate the execution and the load-balacing of OpenMP tasks. High-end users can select the task list granularity according to the tree structure and choose the most convenient work-stealing strategy. One of these strategies takes into account data locality with the help of the hierarchical view. It performs well with unbalanced codes, from BOTS benchmarks, in comparison to Intel and GNU OpenMP runtimes on 16-core and 128-core systems.

1 Introduction

Conceiving parallel algorithms is getting more and more intricate in accordance with the evolution of computer architectures. Multi-cores and many-cores systems are widespread in high performance computing landscape. The number of computing units per node massively increase and the future processor design announced by constructors, for example Intel *Many Integrated Core Architecture* [6], continues this upward trend. In order to help the parallel applications programmer in getting the best performance from the hardware, work has been conducted to integrate inside programming models implementations several mechanisms [1–3] that take into account the memory hierarchy of the underlying node. The programming models themselves evolve to offer features fitted with current processors structure. One could cite the adjonction of task parallelism to the OpenMP *de facto* standard [4] (in the 3.0 version) that allow the programmer to express a fine-grained parallelism. Currently, most of the current OpenMP implementations support task programming, like in GNU OpenMP [5] or Intel OpenMP [7]. Thus, each of them work on a particular task management system which directly affect application performance, according to the system architecture. Indeed, C. Terboven and al. [8] point out that the topology needs to be taken into account, especially NUMA architectures. In this article, we propose to draw a list of parameters that control task scheduling and evaluate the different configurations with representative applications over a highly hierarchical system. In this way, we characterize the models of task-based applications and we map each category to the right OpenMP task scheduling configuration. We implemented our work inside the MPC framework [9], which now includes an OpenMP 3.0 implementation.

This paper is structured as follows. The next section presents some related work on task scheduling. Section 3 introduces our customizable OpenMP task runtime which relies on the control of the task list granularity according to the hardware topology and on the selection of a work-stealing policy. Evaluation results of the different combinations provided by our proposal are presented in section 4. The last section sums up these results and deals with some future work.

2 Related Work

In this section we address the description of several task scheduling engines with an eye to draw up the list of mechanisms employed, particularly the type of task list and their consideration about the system topology.

One of the most common impementation is the GNU OpenMP runtime embedded with the GNU compiler collection since 4.2 version. For the task scheduling, it uses a single list per team. Each access to the list implies to acquire a mutex and whenever a thread seeks for tasks to run or needs to store one (deferred task), this global list is accessed. When a large amount of threads execute tasks, this list quickly becomes a memory bottleneck.

On the other side, the other major implementation, Intel OpenMP *Runtime Library*, ties a task list to each OpenMP thread. Each time a thread creates a task that will not be executed immediatly, that task is placed inside the thread task deque (double-ended queue). A random stealing strategy between thread's deques is set up for load balancing purpose.

Open UH is a branch of the Open64 compiler suite providing OpenMP tasking feature [10]. As in Intel runtime, it uses per-thread deques. Inspired by the Cilk scheduler [11], the tasks are created in a breadth-first way and executed in depth-first manner. It also uses a cut-off (on the number of total tasks created and the depth in the task graph) to avoid task overload which could happen with recursive algorithm like Fibonacci sequence computing.

One could cite another OpenMP task scheduling implementation [13], realized upon the ROSE compiler infrastructure [12] and which uses *Qthreads* user-level thread runtime library [14]. This one targets multi-core systems through a hierachical scheduling strategy. They point out a strategy which regroup threads in *sheperds* using one LIFO task queue per shepherd. A work stealing mechanism between shepherds maintain the work balancing. Our proposal is inspired by this structuring however we decide not to fix the granularity of the threads pools, so as to consider it as a parameter.

Based on the runtime system for data-flow parallel applications X-KAAPI [15], LibKOMP [16] implements the OpenMP task model. As previous runtimes, load balancing is realized with a work stealing technic, inspired by Cilk. They also propose several extensions to the standard to deal with task data dependencies now present in OpenMP 4.0. Nevertheless, in this article we do not deal with this part of the standard which add some interesting constraints to task scheduling.

One can also notice the OMPi [17] OpenMP 3.0 infrastructure for C language, including a source-to-source compiler and a runtime library. In order to implement a breadth-first algorithm, it uses a circular deque per thread to manage deferred tasks. Load balancing is managed with a lock-free work-stealing algorithm where each thread thief traverses other thread queues.

The ForestGOMP software [2] also uses work stealing technics according to the memory hierarchy. However this takes place when dealing with nested parallelism and the stealing objects are threads and their data.

Beside OpenMP runtimes, we could talk about some other task schedulers that employ several technics which could be applied to OpenMP task managing. One in particular, StarPU [18] is well-known in HPC domain and is based on data-flow dependencies to schedule tasks. A set of parameters is available to define the kind of task list (FIFO, LIFO, deque) and its granularity (one per thread or a single global one).

From this overview of task schedulers and their internal mechanisms, we decide to build an environment in which we could play with different parameters like the granularity of task lists and the work stealing policy.

3 Task Scheduling Control

OpenMP tasking support implies for a runtime developer to take some significant decisions concerning the implementation, especially the type of datastructure for task management (list, stack, deque, etc.) which bears directly on application performance. Indeed, the task paradigm often leads OpenMP applications to generate a large amount of tasks. That means the runtime has to minimize the overhead of the numerous operations linked to task managing: creation, browsing, sharing and/or stealing, etc. Additionally, the runtime has to control the load balancing, through a work stealing mechanism in most cases.

Considering these issues, we propose a customizable OpenMP task scheduling engine giving us the possibility to evaluate several configuration sets which may be inspired by existing OpenMP task runtimes. This work has been realized inside the MPC framework [9] that provides an unified parallel runtime which conforms to the OpenMP 3.0 standard beside others. The compilation step, which turns OpenMP directives into runtime calls, is performed by a patched version of the GNU C compiler. Actually, we could control at user level the behavior of the scheduler by specifying the number of task list and choosing the work stealing policy.

3.1 Task List Granularity

The omnipresence of multicore architecture in High Performance Computing landscape constrained the parallel applications and runtime developers to take care of the underlying hardware topology. This becomes a necessary step to reach efficiency. Some work on OpenMP task scheduling [2, 13, 8] has shown that the difficulty comes from mapping of task execution scheme with the memory hierarchy of the system. The hwloc software package [19], used in several parallel runtimes and MPI implementations, helps us to discover the entire topological structure of the system from which we build the MPC OpenMP threads tree. This topology tree is a restricted version of from the original one: it ignore the levels that do not bring any structure information (one-to-one links). Thus we hold all the groups of threads defined according to the memory hierarchy, as in the ForestGOMP runtime system with its hierarchical scheduler. For example, on a 16-core architecture (figure 1), there are eight levels but only three are relevant in the hierarchy structure. The first one concerns the processing units, the cores, the L1 and L2 cache memories. The second one regroups the L3 cache and the NUMA node levels. The last one is the whole machine.

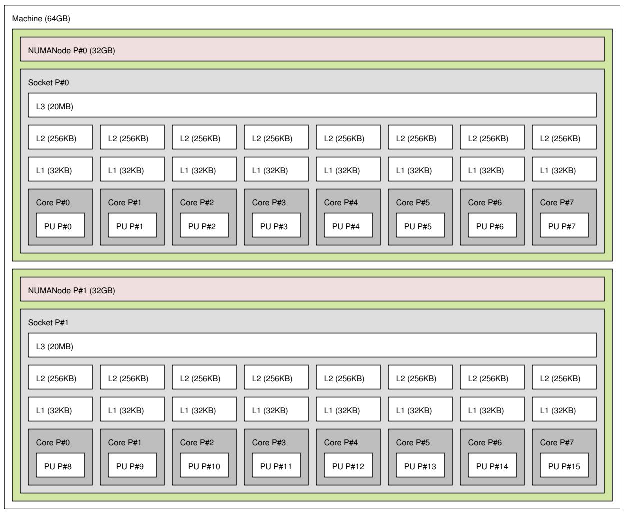

Fig. 1. Architecture topology of an eight-core Intel Sandy Bridge EP dual processor

With this representation, the user can decide at which level the task lists are allocated and accessed by the OpenMP threads. This parameter allows him to play with the impact of access contention to the list(s) and data locality, noticed as a main challenge [8]. Thus we extend the *shepherds* concept of S.L. Olivier and al. [13] to the whole hierarchical levels of a computing node.

3.2 Stealing Strategies

Since *Cilk* [11], the work stealing algorithm is the most studied one for dynamic load balancing purpose. It provides pretty good performance on average and is implemented inside a large majority of task schedulers. When it comes to starvation for a thread, it becomes a thief looking for work inside other threads task pools. Most of the time the victim is randomly chosen, which is a pretty good strategy: generally it avoids contention for multiple thieves at the same time and is a quick decision algorithm which matters at such critical point. However, this strategy does not take into account the memory position of the stolen data according to the binding of the thief thread. On a SMT system or a small scale system, that does not really matter. When executing this algorithm on a large multiprocessor and multicore machine, the impact over the performance may not be negligible anymore. So we designed several policies for the selection of the victim, whose one considers the memory hierarchy.
There are two kinds of stealing strategies. The first one looks for a task to execute inside all available lists, which is a pretty agressive policy. The second one tries to steal inside only one list among those available. After the task search, the thread which initiates the stealing process performs a yield call.

For the *Hierarchical* strategy, the search starts from the closest list in the hierarchical order determined with the physical architecture structure. For example, in case of one task list per thread, a thread whose list is empty will start to steal a task from the lists of threads running on the cores of the same processor, before looking further. The *Random* strategy looks for a randomly-chosen victim and the *Random Order* generates a random-ordered sequence of all task lists to look for. Regarding the *Round Robin*, it browses the lists for a task to steal according to a static and global ordering. A thread which needs to steal a task would look inside the first neighbor of its own list, then the second, and so on. Finally, the *Producer* algorithm selects the list which contains the largest number of tasks enqueued and the *Producer Order* strategy builds a sequence of task lists according to this indicator.

One could notice that only the *Random* and the *Producer* policies are single trial ones. Moreover if we consider that the second one (*Producer*) implies a lookup inside an attribute (number of element) of each list, *Random* is the only real *single trial* policy.

4 Evaluation

This section reports the results of our experiments on the BOTS benchmarks suite to evaluate our different strategies in comparison to two other OpenMP implementations: the Intel OpenMP *Runtime Library* coming with version 13.1.3 of the Intel C compiler and the GNU OpenMP library with the version 4.7 of the GNU C compiler.

4.1 Experiments platforms

Our experiments were conducted on a 128-core node of the *Curie* supercomputer (GENCI) and composed of 16 eight-core Intel *Nehalem-EX* processors at 2.27 GHz and associated to 512 GB of memory (32 GB per NUMA node). This structure comes from the association of 4 motherboards inter-connected through a Bull network. Each processor exposes three levels of cache memory: 32 KB of L1 and 256 KB of L2 cache owned by each core and 24 MB of L3 shared by eight cores.
For MPC, the hierarchical representation of the system, used for OpenMP thread scheduling, is a four-level tree. The root (level 0) corresponds to the whole computational node, the next level (1) to the motherboard, the next one (level 2) to the socket and L3 cache memory and the leafs (level 3) represent the cores with their L1 and L2 cache memories. That means the granularity of the task lists could be defined among four values: a single one for whole system; 4 lists, one per each motherboard; 16, one per socket; and 128, one per core.

4.2 Results

The results we present in this section come from the execution of the Barcelona OpenMP Tasks Suite (BOTS). These benchmarks, inspired from real-life applications, evaluate the performance of OpenMP tasks runtimes. Several versions of each benchmark are available and described in [20]. The figure 4.2 presents the main characteristics of those benchmarks we took interested in. Thus, some kernels use a single thread to produce all the tasks (with a *single* construct) whereas for others, all threads generate a certain amount of tasks. Moreover we wanted to consider the number of tasks produced which directly impacts the performance of the runtime system.

Application	**Creation pattern**	**Task type**	**Load-balancing**	**#Tasks**
Alignment	Single & Multiple	Final	Regular	$\approx 50K$
FFT	Multiple	Nested	Regular	$\approx 10M$
Fibonacci	Multiple	Nested	Regular	$\approx 860M$
Sort	Single	Nested	Irregular	$\approx 2M$
Sparse LU	Single & Multiple	Final	Irregular	$\approx 40K$

One has to specify that the modified version of GCC used for MPC is 4.4 and the version used for comparisons is 4.7. This explains the execution time differences between GCC and MPC for sequential runs. The next figures show the differences for two of the BOTS applications. The MPC version is around 30-35 % less effective than GCC .

	GOMP	**MPC**
T_{serial}	210.56	286.44
T_1 (*for*)	203.14	264.87
T_1 (*single*)	204.34	265.09

Fig. 2. *Alignment* results for serial execution

	GOMP	**MPC**
T_{serial}	4.05	5.51
T_1	4.10	5.49

Fig. 3. *Strassen* results for serial execution

Globally, we observed that, for all benchmarks but the *SparseLU* in multiple producer version, using one tasks list per thread provides the best performance. Each thread own its list and generate a pool of tasks locally.

4.3 Alignment

Alignment is an application where the data locality really matters. Indeed, the quantity of write operations to non-private memory is very low as presented in [20]. The great majority of writes are to the private memory of the task. Considering a 128-core node, task scheduling with a single global list, as for GCC , is the worst solution whereas giving one list per thread, like ICC , seems to be more efficient. In figures 4 and 5, the different execution times presenting the runs for MPC confirm this assessment.

Regarding the work-stealing mechanisms, all strategies deliver more or less the same performance. Gaps between strategies execution time are just a bit less perceptible with the *single* construct version.

Of interest, for both single and multiple producers versions, MPC performs worse than ICC and even than GCC with sequential and with 8, 16, 32, 64 threads. When it reaches the number of 128 threads, MPC distinguishes itself and outperforms GCC . For a well balanced code, like the multiple producer version, ICC still get better performance. However with the *single* construct, the benchmark is unbalanced and MPC outperforms ICC . The best results comes from the *hierarchical* stealing policy, always with a task list per thread.

4.4 FFT

The *FFT* benchmark computes the *Fast Fourier Transform* of a vector of n complex values. ICC gave the best results in this benchmarks even if the speedup is'nt really significant. For its own part, GCC does not scale and performs really

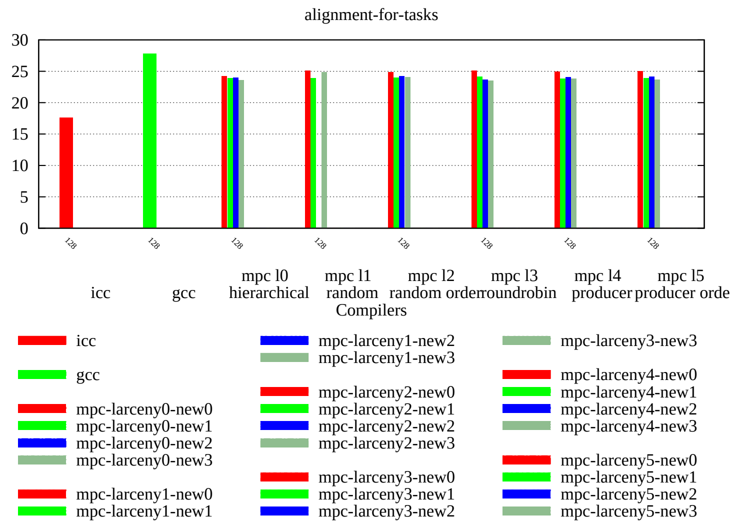

Fig. 4. *Alignment for* results on a 128-core node

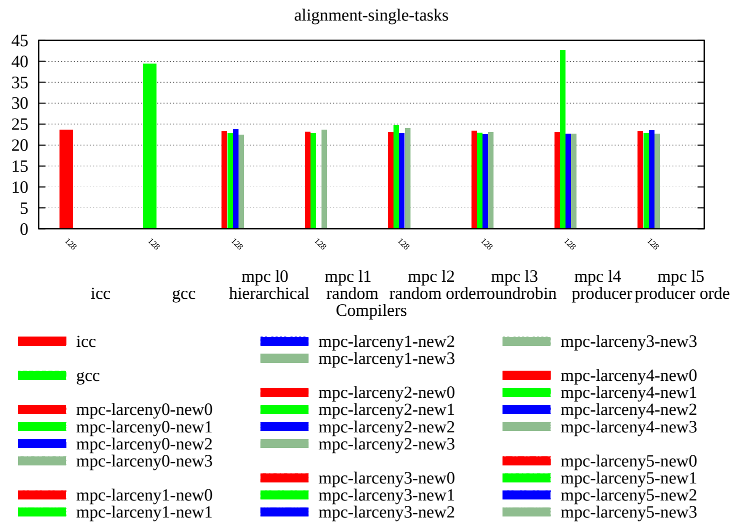

Fig. 5. *Alignment single* results on a 128-core node

	Intel	**GOMP**	**MPC H.**	**MPC R.**	**MPC R.O.**	**MPC R.R.**	**MPC P.**	**MPC P.O.**
T_{serial}	284.78	301.03	306.15	306.15	306.15	306.15	306.15	306.15
T_{1list}	27.41	1467.32	973.54	1205.36	3113.86	1474.11	1249.48	1277.9
T_{4lists}			1127.05	912.43	569.06	n/a	1392.54	1216.01
$T_{16lists}$			1219.73	n/a	1164.68	n/a	324.14	364.04
$T_{128lists}$			204.06	1028.93	146.76	205.95	175.79	1186.2

Fig. 6. *FFT* execution time (in seconds) on a 128-core node

worst than ICC . Among several policies given by MPC , the ones using a single list per core show the best performance. Moreover all others policies deliver a consequent deterioration of the execution time.
Concerning the differences between work-stealing strategies, only *hierarchical* , *random order* , *roundrobin* , *producer* versions performs better than the sequential execution of the benchmark.
There are a really large number of tasks to manage for the runtime. In order to limit the impact of the overhead for managing so many elements, LibGOMP use a threshold on the number of tasks generated at one time. We also choose this solution. However the parallelism of the application is limited by this threshold. This explain the best performance come from the Intel runtime and the worst ones from GCC and MPC .

4.5 Fibonacci

	Intel	**GOMP**	**MPC H.**	**MPC R.**	**MPC R.O.**	**MPC R.R.**	**MPC P.**	**MPC P.O.**
T_{serial}	71.12	115.23	118.34	118.34	118.34	118.34	118.34	118.34
T_{1list}	5.06	n/a	186.1	101.43	13.41	8.67	134.57	129.83
T_{4lists}			440.49	151.71	1544.99	7.94	149.52	14.01
$T_{16lists}$			140.66	n/a	165.41	4.74	29.04	135.41
$T_{128lists}$			107.99	2.68	118.76	2.75	3.46	732.26

Fig. 7. *Fibonacci* execution time (in seconds) on a 128-core node

This application benchmark generate a huge number of fine grained OpenMP task to compute the nth Fibonacci number thanks to a recursive algorithm. For our run we use the parameter $n = 42$.
Among all strategies, only the ones which do not require a long time process to select a victim, like *random* , *roundrobin* and *producer* , deliver a good speedup, even better than ICC . Indeed, in this *Fibonacci* algorithm, the duration of executing a task code is so tiny that the steal decision step became more critical. A way to counterbalance with these overheads is to steal more than only one task and to determine the quantity of tasks to steal, like in libKOMP or in the OpenMP runtime developed over the Rose compiler. As for GCC , there are so many accesses to the global list during the whole run that it didn't finished in a reasonnable duration.

4.6 Sort

The *Sort* benchmark sorts a random permutation of n numbers with a fast parallel sorting variation of the classical mergesort. As seen in the figure 8, the best execution times for MPC correspond to the *one list per thread* strategy. The one using the *random* work-stealing policy is the most performant.

	Intel	GOMP	MPC H.	MPC R.	MPC R.O.	MPC R.R.	MPC P.	MPC P.O.
T_{serial}	4.4	4.28	4.97	4.97	4.97	4.97	4.97	4.97
T_{1list}	0.89	9.32	3.76	4.93	3.16	3.75	5.19	3.87
T_{4lists}			2.79	1.94	1.91	2.38	3.47	2.49
$T_{16lists}$			1.79	n/a	8.65	2.08	1.42	1.91
$T_{128lists}$			1.64	1.07	1.34	1.59	1.27	2.5

Fig. 8. *Sort* execution time (in seconds) on a 128-core node

	Intel	GOMP	MPC H.	MPC R.
T_{serial}	2993.16	1005.52	985.45	985.45
T_{1list}			52.87	131.51
T_{4lists}			32.78	130.94
$T_{16lists}$			52.23	n/a
$T_{128lists}$			33.0	49.72

Fig. 9. *SparseLU for*
execution time (in seconds) on a 128-core node

4.7 SparseLU

The Sparse LU application computes a LU matrix factorization over sparse matrixes. We were interested into the multiple producers version. A group of submatrices is assigned to each thread, some of them may not be allocated which explains the unbalance of the algorithm. Due to a large percentage of writes to shared data, the locality must be a priority for the task scheduler. Moreover, unlike the previous algorithms that's not nested data, thus there is less likely for an OpenMP thread to get tasks that will work on the same data. On the figure 9 we could see that a hierarchical approach for task stealing gets better results than the classical random one.

To sum up, we observed that, even if it delivers good performance in most cases, the approach of random stealing with one tasklist per thread does not suit to every cases. For some algorithms, the data locality must be taken into consideration and a topology-aware task scheduler can benefit from the NUMA characteristic of some actual systems.

5 Conclusion

Computing platforms design is getting embarassingly parallel for application programmer. The latters need to split up their algorithm structure to offer sufficient parallel work to the massive number of cores forming the current and future architecture. The evolution of parallel programming models, like OpenMP with the task support, allow them to express their problem in a finer-grained parallelism. Nevertheless, the hierarchical structure of the underlying system is rarely considered in the OpenMP task schedulers. On large NUMA node it can be really penalizing due to the overhead of distant memory accesses over local ones. We proposed in this paper an evaluation of several task scheduling technics over such computing system. We designed a configurable task scheduler which allows us to control the granularity of task list, according to the hardware topology, and to select a specific work-stealing strategy. Thus, we have compared many technics combinations through the execution of *Barcelona OpenMP Task Suite*. We notice that the strategy adopted by LIBGOMP cannot scale on a large NUMA node and for several applications. Moreover we also noticed that a hierarchical work stealing strategy can outperform the efficient INTEL OPENMP RUNTIME LIBRARY. We are actually working on the support of INTEL C compiler in order to benefits from its optimizations and compare only the performance of both runtime.

References

1. Teng Ma and Bosilca, G. and Bouteiller, A. and Goglin, B. and Squyres, J.M. and Dongarra, J.J.: Kernel Assisted Collective Intra-node MPI Communication among Multi-Core and Many-Core CPUs. International Conference on Parallel Processing (ICPP), 532–541 (2011)
2. Broquedis, F. and Furmento, N. and Goglin, B. and Wacrenier, P-A and Namyst, R.: ForestGOMP: an efficient OpenMP environment for NUMA architectures. International Journal on Parallel Programming, 418–439 (2010)
3. Jin, H. and Jespersen, D. and Mehrotra, P. and Biswas, R. and Huang, L. and Chapman, B.: High performance computing using MPI and OpenMP on multi-core parallel systems. Parallel Computing, 562–575, (2011)
4. The OpenMP API specification for parallel programming. `http://www.openmp.org`
5. An OpenMP implementation for GCC. `http://gcc.gnu.org/projects/gomp`
6. Intel Xeon Phi Coprocessor - The Architecture. `http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner`
7. Intel OpenMP Runtime Library. `https://www.openmprtl.org`
8. Terboven, C. and Schmidl, D. and Cramer, T. and Mey, D.: Assessing OpenMP Tasking Implementations on NUMA Architectures. IWOMP'12 Proceedings of the 8th international conference on OpenMP in a Heterogeneous World, 182–195 (2012)
9. Pérache, M. Jourdren, H. and Namyst, R.: MPC: a unified parallel runtime for clusters of NUMA machines. Proceedings of the 14th international EURO-PAR conference, 78–88 (2008)
10. Addison, C., LaGrone, J., Huang, L., Chapman, B.: OpenMP 3.0 tasking implementation in OpenUH. Open64 Workshop at CGO (2009)
11. Blumofe, R. D. and Joerg, C. F. and Kuszmaul, B. C. and Leiserson, C. E. and Randall, K. H. and Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. Journal of Parallel and Distributed Computing, 207–216 (1995)
12. Liao, C. and Quinlan, D. J. and Panas, T. and de Supinski, B. R.: A ROSE-Based OpenMP 3.0 research compiler supporting multiple runtime libraries. IWOMP'10 Proceedings of the 6th international conference on Beyond Loop Level Parallelism in OpenMP: accelerators, Tasking and more, 15–28 (2010)
13. Olivier, S. and Porterfield, A. and Wheeler, K. B. and Spiegel, M. and Prins, J. F.: OpenMP task scheduling strategies for multicore NUMA systems. International Journal of High Performance Computing Applications, 110–124 (2012)

14. Wheeler, K.B. and Murphy, R.C. and Thain, D.: Qthreads: An API for programming with millions of lightweight threads. IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, 1-8 (2008)
15. Gautier, T. and Ferreira Lima, J. V. and Maillard, N. and Raffin, B.: XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. IEEE International Parallel and Distributed Processing Symposium (IPDPS), 1299–1308 (2013)
16. Broquedis, F. and Gautier, T. and Danjean, V.: libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms. IWOMP'12 Proceedings of the 8th international conference on OpenMP in a Heterogeneous World, 102–115 (2012)
17. Agathos, S. N. and Kallimanis N. D. and Dimakopoulos V. V.: Speeding up OpenMP tasking. Euro-Par'12 Proceedings of the 18th international conference on Parallel Processing, 650–661 (2012)
18. Augonnet, C. and Thibault, S. and Namyst, R. and Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, 23:187-198 (2011)
19. Broquedis, F. and Clet-Ortega, J. and Moreaud, S. and Furmento, N. and Goglin, B. and Mercier, G. and Thibault, S. and Namyst, R.: hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, PDP (2010)
20. Duran, A. and Teruel, X. and Ferrer, R. and Martorell, X. and Ayguade, E. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. Proceedings of the 2009 International Conference on Parallel Processing, 124–131 (2009)