Speeding Up OpenMP Tasking*

Spiros N. Agathos**, Nikolaos D. Kallimanis***, and Vassilios V. Dimakopoulos

Department of Computer Science, University of Ioannina
P.O. Box 1186, Ioannina, Greece, GR-45110
{sagathos,nkallima,dimako}@cs.uoi.gr

Abstract. In this work we present a highly efficient implementation of OpenMP tasks. It is based on a runtime infrastructure architected for data locality, a crucial prerequisite for exploiting the NUMA nature of modern multicore multiprocessors. In addition, we employ fast work-stealing structures, based on a novel, efficient and fair blocking algorithm. Synthetic benchmarks show up to a 6-fold increase in throughput (tasks completed per second), while for a task-based OpenMP application suite we measured up to 87% reduction in execution times, as compared to other OpenMP implementations.

1 Introduction

Parallel computing is quickly becoming synonymous with mainstream computing. Multicore processors have conquered not only the desktop but also the hand-held devices market (e.g. smartphones) while many-core systems are well under way. Still, although highly advanced and sophisticated hardware is at the disposal of everybody, programming it efficiently is a prerequisite to achieving actual performance improvements.

OpenMP [13] is nowadays one of the most widely used paradigms for harnessing multicore hardware. Its popularity stems from the fact that it is a directive-based system which does not change the base language (C/C++/Fortran), making it quite accessible to mainstream programmers. Its simple and intuitive structure facilitates incremental parallelization of sequential applications, while at the same time producing actual speedups with relatively small effort.

The power and expressiveness of OpenMP has increased substantially with the recent addition of tasking facilities. In particular V3.0 of the specifications include directives that allow the creation of a task out of a given code block. Upon creation, tasks include a snapshot of their data environment, since their execution may be deferred for a later time or when task synchronization/scheduling directives are met. Tasking is already supported by many commercial and non commercial compilers (e.g. [2,4,16]).

Most of these implementations rely on sophisticated runtime libraries that provide each participating thread with private and/or shared queues to store tasks pending for

* This work has been supported in part by the General Secretariat for Research and Technology and the European Commission (ERDF) through the Artemisia SMECY project (grant 100230).

** S.N. Agathos is supported by the Greek State Scholarships Foundation (IKY).

*** N.D. Kallimanis is supported by the Empirikion Foundation.

C. Kaklamanis et al. (Eds.): Euro-Par 2012, LNCS 7484, pp. 650–661, 2012.

execution. Work-stealing [3] is usually employed for task scheduling, whereby idle threads, with no local tasks to execute, try to "steal" tasks from other thread queues. Work-stealing is a widely studied and deployed scheduling strategy, well known for its load balancing capabilities. Efficient implementation of the work-stealing algorithm and its related data structures is hence crucial for the performance of an OpenMP tasking system. The associated overheads for enqueing, dequeuing and stealing tasks can easily become performance bottlenecks limiting system's scalability as the number of cores keeps increasing.

In this work we present a high-performance tasking infrastructure built in the runtime system of the OMPi OpenMP/C compiler [6]. Support for tasking was recently added to OMPi [1], including an initial functional, albeit non-optimized, general tasking layer in its runtime library. Here we present a complete redesign of OMPi's tasking system, engineered to take advantage of modern multicore multiprocessors. The deep cache hierarchies and private memory channels of recent multicore CPUs make such systems behave with pronounced non-uniform memory access (NUMA) characteristics. To exploit these architectures our runtime system is organized in such a way as to maximize local operations and minimize remote accesses which may have detrimental performance effects. This organization is coupled with a work-stealing system which is based on an efficient blocking algorithm that emphasizes operation combining and thread cooperation in order to reduce synchronization overheads.

We have tested our system exhaustively. Using a synthetic benchmark we reveal a very significant— up to 6x—increase in attainable throughput (tasks completed per second), as compared to other OpenMP compilers, thus enjoying scalability under high task loads. At the same time applications from the BOTS tasking suite [8] experience reduced execution times (up to 87%), again in comparison to the rest of the available OpenMP systems.

The rest of the paper is organized as follows: in Section 2 we present OMPi and the way it handles tasking. The organization of its optimized runtime system is presented in detail. A key part, namely the work-stealing subsystem, is discussed separately in Section 3. Section 4 is devoted to the experiments we performed in order to assess the performance of our implementation and finally Section 5 concludes this work.

2 Tasking in the OMPi Compiler

OMPi [6] is an experimental, lightweight OpenMP V3.0 infrastructure for C. It consists of a source-to-source compiler and a runtime library. The compiler takes as input C code with OpenMP pragmas and outputs multithreaded C code augmented with calls to its runtime library, ready to be compiled by any standard C compiler.

Upon encountering an OpenMP `task` construct, the compiler uses *outlining* to move the code residing within the `task` region to a new function. Because each task is a block of code that may be executed asynchronously at a later time, its data environment must be captured at the time of task *creation*. Thus the compiler inserts code which allocates the required memory space, copies the relevant (firstprivate) variables and places a call to the runtime system to create the task using the outlined function and the captured data environment.

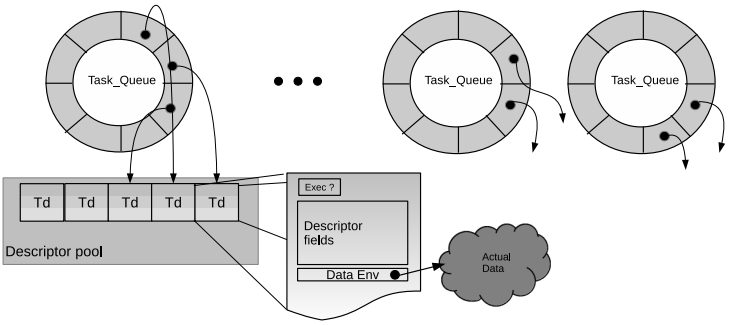


Fig. 1. Task queues organization. Each thread owns a circular queue (TASK_QUEUE) where pointers to task descriptors (Td) are inserted. Each Td carries bookkeeping information, a special flag (Exec) and a pointer to the task data.

If there exists an `if` clause whose condition evaluates to false or the runtime system cannot (or selects not to) create the task, then the task must be executed immediately. To optimize this case, the compiler produces a second copy of the task code (called *fast path*), this time *inlined*. Local variables are declared to capture directly the data environment and are used within the task code. In this manner, the task is executed with almost no overheads. Depending on the runtime conditions, either the normal (outlined) or the fast (inlined) path is executed.

2.1 Optimized Runtime

Our runtime organization is based on distributed task queues, one for each OpenMP thread as shown in Fig. 1. These are circular queues (TASK_QUEUEs) of fixed size which is user-controlled as one of OMPi's environment variables. When a thread meets a new task region then it has the choice of executing it immediately or submitting it for deferred execution. OMPi follows the second approach, that is our runtime uses a *breadth-first* task creation policy, and the new task is stored in the thread's local TASK_QUEUE. Whenever a thread is idle and decides to execute a task, then it dequeues a deferred task from its TASK_QUEUE. If a thread's TASK_QUEUE is empty then this thread becomes a thief and traverses other threads queues in order to steal tasks. The manipulation of a TASK_QUEUE is a crucial synchronization point in OMPi, since multiple threads may concurrently access it. OMPi utilizes a highly efficient work-stealing algorithm described in the next section.

If a thread tries to store a new task in its queue and there is no space, the thread enters *throttling mode*. In throttling mode newly created tasks are executed immediately and hence the task creation policy changes to *depth-first*. In addition, as described above, throttled threads utilize the fast execution path. While in throttling mode all descendant tasks are executed immediately in the context of parent task, favoring data locality. Notice that a suspended parent task never enters the TASK_QUEUE hence it can never be stolen by any other thread. This is to say that in OMPi all tasks are *tied*.

A thread's entrance in throttling mode is one of the runtime objectives. However, a thread operating in throttling mode does not produce deferred tasks, which results in a

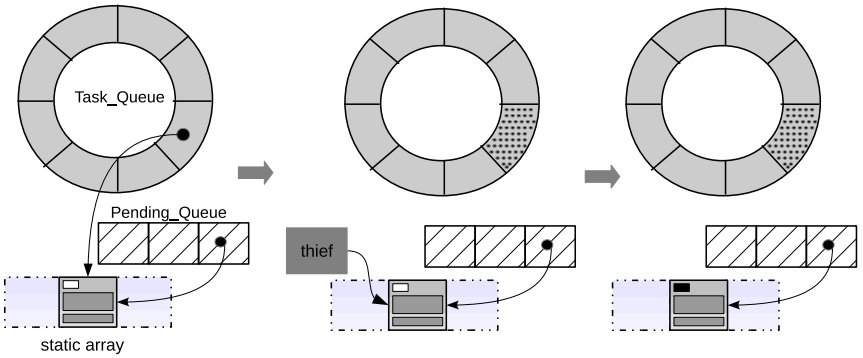


Fig. 2. Pending, executing (stolen) and finished task. When a task is pending for execution then corresponding entries in TASK_QUEUE and PENDING_QUEUE point to its Td. Upon dequeing, only the link in TASK_QUEUE is removed, freeing one slot. When the task is finished, the executing thread sets the Exec flag to announce that the descriptor can be recycled.

reduction of available parallelism. To strike a balance, before a throttled thread executes a new task, it checks its TASK_QUEUE free space. If the queue has become at least 30% empty then throttling is disabled and task creation policy returns to breadth-first.

As shown in Fig. 1 each entry in the TASK_QUEUE is a pointer to a task descriptor (Td), which stores all the runtime information related to the task execution as well as the task data environment. The descriptor is obtained out of the thread's descriptor pool. This pool contains an array of pre-allocated descriptors (in order to speed up the allocation process) and a dynamic overflow list for the case the array becomes empty. Whenever a task finishes its execution, the corresponding Td is returned to a descriptor pool, recycled for future reuse. A task created by a thread might be stolen and executed by another thread in its team. When the task finishes and the descriptor must be recycled, a decision has to be made as to which pool the descriptor should return to. If it enters the pool of the thread that executed the task, severe memory consumption is possible in cases where only few threads create a big number of tasks while the rest execute them. On the other hand, this option is a local operation, enjoying lack of contention. Memory consumption is reduced if the descriptor is put back to the task creator's pool, and this is what OMPi does. Notice though that synchronization needs arise since threads that stole tasks from the same thread may try to store to its descriptor pool concurrently.

In order to avoid the aforementioned synchronization overheads, we have used a garbage-collecting strategy, shown in Fig. 2. Each thread t maintains a private set of pointers (PENDING_QUEUE) to the task descriptors it has created and are either stored for deferred execution or are currently executing. When a task is dequeued for execution (e.g. because a thief stole it), the Td pointer is removed form TASK_QUEUE but remains intact in PENDING_QUEUE. The descriptor contains a special flag ('Exec' in Fig. 1). When the task completes its execution, the executing thread sets this flag to announce that the descriptor can now be recycled. On specific occasions thread t traverses its PENDING_QUEUE to find Tds that represent executed tasks and returns them to its pool for future use.

The PENDING_QUEUE plays a central role in the implementation of the `taskwait` and `barrier` constructs, too. Whenever a task meets a `taskwait`, it must wait until the completion of all tasks it created (it is actually then that the task execution and stealing mechanism is triggered). This completion condition is fulfilled simply when all the descriptors in the thread's PENDING_QUEUE have been flagged as executed. Upon meeting a `barrier`, a thread must wait until: (i) all its siblings reach the barrier and (ii) all team-generated tasks are executed. For the first condition an atomic counter is employed, getting increased by every thread reaching the barrier. For the second condition each thread contiguously executes/steals pending tasks from all TASK_QUEUE's within its team until all team PENDING_QUEUEs become empty.

Our runtime design aims at using as little shared data as possible, so as to reduce atomic operations and minimize thread synchronization. It is worth noting that in our tasking system thread synchronization occurs only in two cases. The first is during the unavoidable barrier construct and the second is during the work-stealing operations, as described in the next section, for which a very fast algorithm is employed. All data structures (e.g. Td's in the descriptors pool) are cache line-size aligned so as to eliminate false sharing phenomena and avoid triggering coherency protocol actions, which deteriorate the performance, especially in NUMA platforms.

While in OMPi each thread owns a public task queue where it stores newly created tasks, other compilers use different organizations. In IBM XL compilers [16], a shared task pool is associated with each parallel region where new tasks are put in the end of the queue and threads pick up tasks from the front. In contrast, in OpenUH [4] each OpenMP thread retains two task queues. The first queue is private and used for keeping tied tasks, while the second is public and used to store newly created and untied tasks. In Nanos [17], two types of queues are used. Here, a team of threads has a shared queue for newly created and untied tasks. Furthermore, each thread owns a private local queue used for tied tasks. A detailed comparison of many other queue organization alternatives has been performed by Korch and Rauber [11].

3 A Fast Work-Stealing Algorithm

The work-stealing mechanism is a crucial component of an OpenMP runtime and should thus be designed in a way to be efficient and scalable in cases of high contention. A number of workstealing algorithms with various characteristics has been proposed, such as Intel TBB's AP/SP and Lazy Binary Spliting [14,15] which are targeting taks generated by do-all loops. Cilk's workstealing infrastructure [3] is another well-known example; however Cilk's runtime is not directly applicable to OpenMP since OpenMP allows barriers among team threads. The initial implementation of OMPi tasks [1] utilized a lock-free workstealing algorithm based on [5].

In many applications task creation is unbalanced and it is a very common phenomenon few threads to produce many tasks and all other threads to consume them. In such cases contention could be lowered if threads cooperated instead of competed for obtaining the next tasks to execute. In our OpenMP tasking runtime each thread maintains (owns) a TASK_QUEUE, as explained above. A TASK_QUEUE is a shared object similar to the shared queue [12] supporting two operations: `OwnerEnqueue` and

`Dequeue` for inserting and removing tasks, correspondingly. `OwnerEnqueue`(q, t) inserts a new task t in queue q in case there is enough free space and returns true; otherwise, `OwnerEnqueue` fails and returns false. In contrast to `Enqueue` of a conventional shared queue, `OwnerEnqueue` is executed only by the thread that owns q. `Dequeue` is executed by any thread and removes the most early inserted task of q. Recently, Fatourou and Kallimanis [10] presented `CC-Synch`, an object which is able to implement (simulate) any shared object very efficiently. For example, to implement a shared queue, it is enough to use one instance of `CC-Synch` and to supply the sequential code for the `Enqueue` and `Dequeue` operations. `CC-Synch` supports only one operation called `ApplyOp`(*sfunc*, *arg*, *th_id*); *sfunc* is the serial code of the operation, *arg* is the argument of the operation and *th_id* is the id of the thread that executes the operation.

In [10], it is shown that `CC-Synch` significantly outperforms the state-of-the-art synchronization techniques. This is a result of the efficient implementation of the *combining* technique whereby, one thread (the *combiner*) holds a coarse lock, and additionally to the application of its own operation, serves the operations of all other active threads. Whenever a thread executes an operation using a conventional synchronization technique (such as spin-locks), it causes cache misses by fetching part of a shared object's state to the local processor cache in order to apply its operation. In the combining technique, only the combiner fetches parts of object's state and applies the operations of all active threads. Therefore, a lot of cache misses are avoided and the communication overheads among processors are much lower.

Using `CC-Synch` to implement an operation that is executed only by a single thread in any point of time is rather expensive. Thus, in our work-stealing queue implementation, we designed `OwnerEnqueue` (which is is executed only by the owner of the work-stealing queue) in a way that it does not make calls to `ApplyOp`. Thus, we avoid making the expensive calls of `CC-Synch`, wherever possible. It is noticeable that `CC-Synch` is better suited for cache-coherent NUMA machines, which constitute the majority of modern multicore multiprocessors.

We now give more details for our work-stealing implementation. Our work-stealing task queue (Fig. 3) consists of (i) a shared array of pointers to `TASK` structs, which is called `TASK_QUEUE`, (ii) a shared integer *Top* which points to the topmost element of the queue, (iii) a shared integer *Bottom* which points to the bottommost element of the queue, and (iv) an instance of `CC-Synch`. Since the `OwnerEnqueue` operation is executed only by the owner of the queue, its design is simplified. Whenever a thread p executes an `OwnerEnqueue` operation, it firstly executes a `read` on *Bottom* and after that a `read` on *Top*. If there exists free space, p inserts the new task and increases *Top* by one; otherwise, `OwnerEnqueue` returns false. Since p is the owner of the work-stealing queue and `OwnerEnqueue` is executed only by the owner, p is the only thread that modifies the shared variable *Top*. Therefore, no special care is needed while modifying *Top*. Whenever p wants to execute a `Dequeue` operation, it first checks if at least one element exists in the queue and in that case increases *Bottom* by one. Many threads may access *Bottom* simultaneously, since any thread is able to execute `Dequeue` in any `TASK_QUEUE`. We implement `Dequeue` using an instance of the `CC-Synch`

```
typedef struct WSQueue {
   int Bottom, Top;
   TASK *QArray[m];
   an instance of CC-Synch synchronization technique;
} WSQueue;

bool OwnerEnqueue(WSQueue *l, TASK *arg, int pid) {
   int top = l->top, bottom = l->bottom;
   int new_top = (top + 1) % TASKQUEUESIZE;

   if (new_top == bottom) return false;
   else {
      l->QArray[top] = arg;
      l->top = new_top;
      return true;
   }
}

TASK *Dequeue(WSQueue *l, int pid) { // Serial code for Dequeue, the concurrent
   void *ret;                        // version is implemented using CC-Synch.

   if (l->bottom == l->top) ret = NULL;
   else {
      ret = l->QArray[bottom]
      l->bottom = (l->bottom + 1) % TASKQUEUESIZE;
   }
   return ret;
}
```

Fig. 3. Pseudocode for the work-stealing queue implementation

synchronization queue. Since `CC-Synch` is a synchronization technique that serves operations with FIFO order, threads that execute `Dequeue` operations are also served with a FIFO order. Thus, our implementation satisfies strong fairness properties.

4 Performance Evaluation

In this section we evaluate the efficiency of our OpenMP tasking implementation. A synthetic producer/consumer benchmark was used to measure the task creation and the task execution throughput. Furthermore, the Barcelona OpenMP Tasks suite (BOTS) [8] was utilized in order to test our system in a broad range of task applications. All experiments were run on a 16-core machine equipped with two 8-core AMD Opteron 6128 CPUs running at 2.0GHz and with a total of 16GB RAM. The system runs Debian Squeeze based on Linux kernel 2.6.32.5. We compare the performance of our compiler with GNU GCC (version 4.4.5-8), Intel ICC (version 12.1.0) and Oracle SunStudio SUNCC (version 12.2). For reference the initial unoptimized implementation of OMPi in [1] is also included, labeled as 'OLD'.

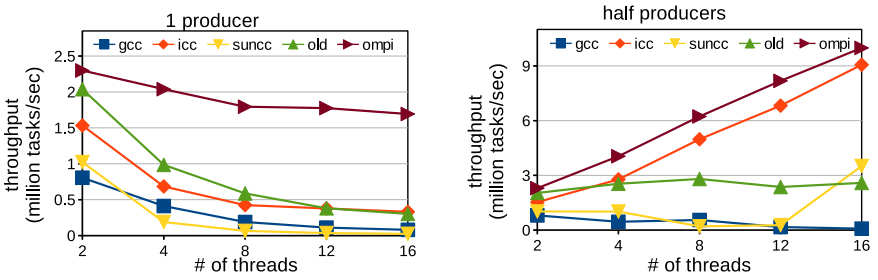
In [7], it is shown that choosing the appropriate limits to enable and disable task cut-off is not an easy task. When dealing with task cut-off, it is required to have good knowledge of application's behavior for a specified input size, and of the runtime 's tasking implementation. We thus chose to deactivate all manual cut-off techniques in all our benchmarks and let the OpenMP implementation operate under its default settings. As far as OMPi and OLD compilers are concerned, we used the default values for the size of TASK_QUEUEs which is 24.

```
main()
{
  #pragma omp parallel num_threads(nthr)
    if(omp_get_thread_num() < nprod) {
      for (int i=0;i<16E6/nprod);i++)
        #pragma omp task
        do_random_work();
    }
}
```

```
do_random_work()
{
  volatile long i;

  for (i=0;i<RandomRange(0,maxload);i++)
    ;
}
```

Fig. 4. Code for synthetic microbenchmark



(a) (b)

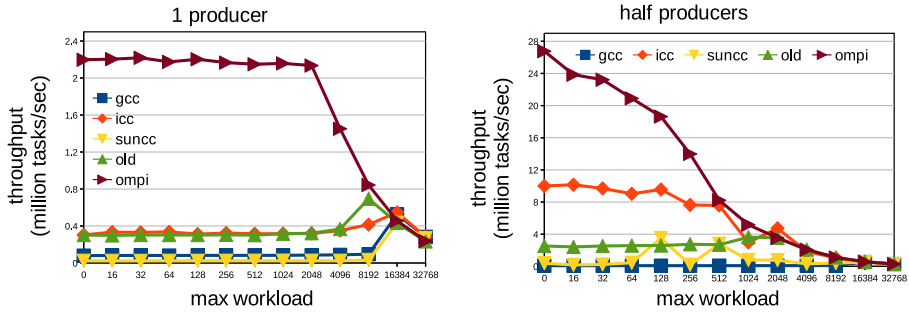
Fig. 5. Synthetic benchmark, `maxload`=128

We used GNU GCC with the "-O3" flag as a back-end compiler for OMPi. The corresponding flags for GCC, ICC and SUNCC were "-O3 -fopenmp", "-fast -openmp" and "-fast -xopenmp=parallel". We experimented with a lot of other flag combinations for all compilers but we didn't notice significant performance differences. All experiments were executed twelve times each, then the best and worst runs were discarded; from the ten remaining executions average values were calculated and reported.

4.1 Synthetic Benchmark

In order to evaluate the performance of OMPi, a synthetic benchmark with a controllable number of task producers and task consumers was used, as shown in Fig. 4. In this benchmark, a parallel region is created and a specified number of threads (equal to `nthr`) is created. Only `nprod` threads become producers and are allowed to create tasks. The rest of threads simply reach the end of parallel region and become consumers (executors) of the created tasks. Each run of the specified benchmark creates 16×10^6 tasks, the creation of which is equally assigned to producer threads. Each task consists of a dummy loop used to simulate workload that a task may have to execute in a way similar to [12,9,10]. The number of iterations is a random number between 0 and `maxload`, a variable controlling the task granularity. Iterator variable i is annotated as volatile in order to avoid compiler code elimination optimizations. This benchmark aims to stress the runtime's ability to create, steal and execute tasks.

We run several tests for different values of `nthr`, `nprod` and `maxload`. In Figs. 5–6 we present each implementation's throughput, measured as the number of tasks completed per second. For Fig. 5(a) we employed one producer and `nthr`-1 consumers.



(a) (b)

Fig. 6. Synthetic benchmark, `nthr`=16

In this experiment `maxload` was chosen to be equal to 128, representing fine-grain work. Some lock-free shared objects show unrealistic high performance when choosing a `maxload` value equal to 0, thus it is a common benchmarking strategy [12,9,10] to choose a small value for `maxload`, but not equal to 0. In this experiment, as more threads try to steal from the task queue of the producer, task throughput decreases. This is the result of extra synchronization overhead added, since more threads compete to get shared access to the same TASK_QUEUE. Due to the combining technique in our work-stealing implementation, OMPi outperforms all other compilers even in cases with very high contention and has the best scalability among them. Specifically, OMPi exhibits up to 5 times higher task throughput (at 16 threads) compared to ICC which is ranked as second best. The original OMPi implementation performs well only when 2 threads are used but its throughput quickly decreases. In Fig. 5(b), we study the behavior for different `nthr` values when `nprod`=`nthr`/2, while `maxload` is still equal to 128. The results are similar, confirming OMPi's superiority.

In Fig. 6(a), the performance results for different values of `maxload` and for a total of 16 threads (one of which produces tasks) are displayed. In this benchmark, our runtime exhibits higher throughput when compared to all other compilers for almost any `maxload` value. For values of 8192 or less, the work that each task executes is quite small and is overwhelmed by the contention that the work-stealing part induces. Since OMPi exploits the combining technique in its work-stealing queue, the synchronization overheads between threads are vastly minimized and the performance advances a lot. We achieved a little more that 6 times better performance compared to ICC and even better compared to GCC and SUNCC when application produces fine-grain tasks. When the task's granularity becomes coarser (`maxload` values greater than 8192), synchronization overheads between threads are not a bottleneck anymore and all compilers tend to exhibit similar behavior. Similar observations can be made with the results in Fig. 6(b), where 8 out of the 16 threads produce tasks for different values of `maxload`. For `maxload` values between 0 and 256 our new runtime achieves from 2.6 to 1.8 times higher throughput than the second best (ICC).

4.2 Performance of the BOTS Application Suite

The Barcelona OpenMP Tasks Suite (BOTS) v.1.1.1 was used for evaluating our tasking environment's efficiency in a wide range of tasking scenarios. Due to space limitations we present detailed results for the Fib, NQueens and Floorplan applications, while a brief discussion is made for Alignment, FFT, Health, Sort, SparseLU and Strassen. In order for every compiler to have full scheduling opportunities, we run both the tied and the untied task versions of the applications (while OMPi always utilizes tied tasks). We report the best execution times observed, although there were no significant performance differences as noted also in [8].

The Fib application computes the nth Fibonacci number using a recursive paralellization producing a very large number of fine-grain tasks. In Fig. 7, execution time results for the the 40th Fibonacci number are shown. Since it was a very common phenomenon for OMPi to outperform some compilers by a factor of ten or more, a logarithmic scale is used in y-axis. OMPi appears to be from 4 to 8 times faster than ICC and 20 to 80 times faster than the original (OLD) implementation. Since Fib exploits nested task parallelization which creates a deep tree of small tasks, it is a common phenomenon some threads to fill their queues. OMPi has a significant performance advantage by leveraging the new work-stealing implementation and the fast execution path produced by the compiler; task load is quickly balanced between threads, and the application delves into throttling mode. Moreover, OMPi, along with ICC, scales up with the number of threads.

NQueens calculates all the solutions of the n-queens chessboard problem. It uses a backtracking search algorithm with pruning that creates unbalanced tasks. Similarly to Fib, Nqueens exploits nested task parallelization which creates a deep tree of tasks. In the NQueens benchmark displayed in Fig. 8, for an input of 14 queens we get similar results to Fib and OMPi gives the best times. OMPi is up to 2 times faster than OLD and up to 3 times faster than ICC(not shown clearly in the logarithmic scale).

Floorplan calculates the optimal floor plan distribution of a number of cells. Tasks are hierarchically generated for each branch of the solution space. This application induces many data synchronizations and comes with a very irregular and aggressive pruning mechanism, which results in a heavily unbalanced task tree. Fig. 9 displays results of the application when the input.20 file is used; ICC is not included here because the application could not compile properly with this compiler. OMPi achieves the fastest times and our original implementation follows. Since Floorplan generates deep nested tasks, OMPi performs well due to the the work-stealing implementation along with the efficient fast path execution. SUNCC cannot exhibit speed-up, while GCC experiences significant slow-down when more threads are used.

Results from the rest of BOTS applications are given in Table 1, for the case of 16 threads. In this table we included results from OMPi when using ICC as back-end compiler, which in many situations produces faster code for the sequential part of the application. In FFT, SparseLU, Strassen and Alignment applications OMPi with ICC as backend proves to be faster, while performing second best only in two applications with very small margins (3% in Sort and 0.2% in Health). ICC has the best behavior in Health application, while our OLD system is the fastest as far as the Sort application is concerned. Thus, OMPi proves to perform consistently well in many different application

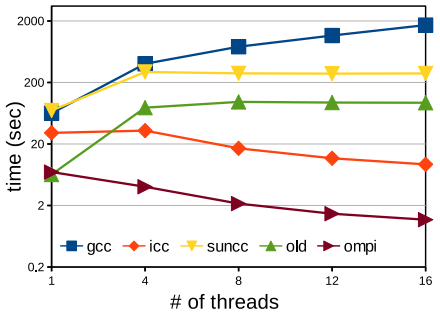


Fig. 7. Fibonacci

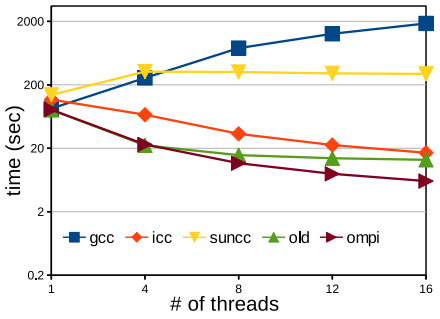


Fig. 8. Nqueens

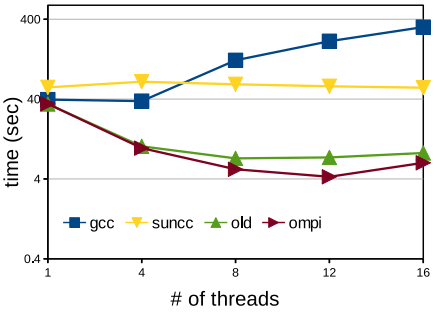


Fig. 9. Floorplan

Table 1. Execution time (sec) of BOTS using 16 threads

Compiler	FFT	Health	Sort	SpLU	Str.	Align.
GCC	17.571	141.85	2.007	1.679	24.602	1.576
ICC	2.086	**4.778**	0.621	1.676	20.641	1.338
SUNCC	2.473	15.694	0.652	1.835	21.619	1.218
OLD	2.086	7.114	**0.591**	1.766	21.589	1.587
OMPi	1.918	5.327	0.610	1.668	22.368	1.604
OMPi_ICC	**1.889**	4.787	0.621	**1.667**	**20.524**	**0.957**

scenarios, and especially when it uses an efficient back-end compiler, giving it a serious performance advantage. In general, ICC and SUNCC perform quite well with few exceptions. The version of GCC we had available does not perform up to par.

5 Conclusion

We present a highly optimized implementation of OpenMP tasking in the context of the OMPi compiler. The implementation is based on a carefully designed runtime system that emphasizes locality and operation combining while minimizing remote accesses which have detrimental performance effects in modern NUMA multicore multiprocessors. As a result, our system exhibits excellent scalability for high task loads and impressive improvement in actual application execution times, where OMPi was shown to offer competitive performance in comparison to other OpenMP implementations.

Currently we are working on analyzing the performance impact of the different portions of our runtime system and optimizing OMPi even more for some corner cases. We also work on supporting the recently released V3.1 of the OpenMP specifications [13] which offer even more opportunities for fast execution through the new `mergeable` and `final` clauses. Our preliminary experiences confirm the performance potential.

References

1. Agathos, S.N., Hadjidoukas, P.E., Dimakopoulos, V.V.: Design and Implementation of OpenMP Tasks in the OMPi Compiler. In: Proc. PCI 2011, 15th Panhellenic Conference on Informatics, pp. 265–269. IEEE, Kastoria (2011)
2. Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., Teruel, X.: An Experimental Evaluation of the New OpenMP Tasking Model. In: Adve, V., Garzarán, M.J., Petersen, P. (eds.) LCPC 2007. LNCS, vol. 5234, pp. 63–77. Springer, Heidelberg (2008)
3. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. J. Parallel Distrib. Comput. 37(1), 55–69 (1996)
4. Addison, C., LaGrone, J., Huang, L., Chapman, B.: OpenMP 3.0 tasking implementation in OpenUH. In: Proc. Open64 Workshop in Conjunction with the Int'l Symposium on Code Generation and Optimization, Seattle, USA (March 2009)
5. Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In: Proc. SPAA 2005, 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 21–28. ACM, Las Vegas (2005)
6. Dimakopoulos, V.V., Leontiadis, E., Tzoumas, G.: A portable C compiler for OpenMP V.2.0. In: Proc. EWOMP 2003, 5th European Workshop on OpenMP, Aachen, Germany, pp. 5–11 (September 2003)
7. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of OpenMP Task Scheduling Strategies. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 100–110. Springer, Heidelberg (2008)
8. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguadé, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In: Proc. ICPP 2009, 38th Int'l Conference on Parallel Processing, Vienna, Austria, pp. 124–131 (September 2009)
9. Fatourou, P., Kallimanis, N.D.: A highly-efficient wait-free universal construction. In: Proc. SPAA 2011, Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, pp. 325–334. ACM, San Jose (2011)
10. Fatourou, P., Kallimanis, N.D.: Revisiting the combining synchronization technique. In: Proc. PPoPP 2012, 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 257–266. ACM, New Orleans (2012)
11. Korch, M., Rauber, T.: A comparison of task pools for dynamic load balancing of irregular algorithms: Research Articles. Concurr. Comput.: Pract. Exper. 16(1), 1–47 (2003)
12. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. PODC 1996, 15th Annual ACM Symposium on Principles of Distributed Computing, pp. 267–275. ACM, Philadelphia (1996)
13. OpenMP ARB: OpenMP Application Program Interface V3.1 (July 2011)
14. Reinders, J.: Intel threading building blocks, 1st edn. O'Reilly & Associates, Inc., Sebastopol (2007)
15. Tzannes, A., Caragea, G.C., Barua, R., Vishkin, U.: Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In: Proc. PPoPP 2010, 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 179–190. ACM, Bangalore (2010)
16. Teruel, X., Unnikrishnan, P., Martorell, X., Ayguade, E., Silvera, R., Zhang, G., Tiotto, E.: OpenMP tasks in IBM XL compilers. In: Proc. CASCON 2008, 2008 Conference of the Center for Advanced Studies on Collaborative Research, Ontario, Canada, pp. 207–221 (October 2008)
17. Teruel, X., Martorell, X., Duran, A., Ferrer, R., Ayguadé, E.: Support for OpenMP tasks in Nanos v4. In: CASCON, pp. 256–259 (2007)