A Quantitative Evaluation of popular Task-Centric Programming Models and Libraries

Artur Podobas¹, Mats Båtsson¹,² and Karl-Filip Faxén²

(podobas,matsbror)@kth.se kff@sics.se

¹KTH, Royal Institute of Technology, Sweden
²SICS, Swedish Institute of Computer Science, Sweden

Abstract—Programmers today face a bewildering array of parallel programming models and tools, making it difficult to choose an appropriate one for each application. The present study focuses on the task-centric approach and compares several popular systems, including Cilk Plus, TBB and various implementations of OpenMP 3.0. We analyse their performance on the BOTS benchmark suite both on a 48 core Magny Cours server and a 64 core TILEPro64 embedded manycore processor.

INTRODUCTION

The task-centric parallel programming model has gained popularity since the mid 2000 due to its simplicity and versatility. Often users are abstracted from all architectural details of the system, and can focus on improving the code. In the task-centric programming model, parallel work is exposed by the user as tasks. A task is any form of computation, and the data it needs, that can be performed in parallel with some other task(s). Created tasks are then scheduled for execution onto cores (typically worker threads) by a runtime system.

Exposing tasks and inserting synchronization where appropriate is much simpler than to orchestrate a program using explicit threading or to partition work and data with message-passing. In addition, the task-programming models have potential to offer support for heterogeneity[24], [5], [4]. Since GPU computing (using the graphics processor for computing) is currently a very active area, this is a major advantage over other programming models. Other researchers have also investigated the use of FPGAs[9] as accelerators for the task-centric model.

Given the importance of the task scheduler in the runtime system, a lot of both development and research effort is devoted to improve on this scheduling in various ways. Examples of this include increasing data locality, decreasing the temperature or reducing the power-consumption. Other efforts focus on extending the case-of-use. Data-driven and implicit synchronization have been successfully employed ([4], [26] in task-centric models to remove the previously-required step of manually inserting synchronization points, and even transactional memory[19] has been proposed.

In the present study we present an evaluation of some of the more popular task-centric implementations in terms of performance and power consumption. We try to explain the results by looking at cache behaviour, resource utilization and overheads for each of the target implementations. We address questions such as: Why do some models suffer performance while other do not? What models are suitable for embedded devices and how do they perform? We present results for two different systems; a large AMD Magny-Cours based NUMA multi processor and an embedded many-core processor (TilePRO64). We believe that our results apply three levels of computing: HPC, General-Purpose and Embedded computing.

RELATED WORK

A micro-benchmark study performed by Dimakopulous et. al [10] is similar to our study but deals exclusively with overheads with nested parallelism in mind. They focused on the OpenMP (2.5+) thread-centric model while we focus on the task-centric model. They evaluated a series of implementations in terms of different scheduling techniques for for-loop parallelism as well as the overheads for reduction, parallel, critical, lock and barrier primitives. The evaluated implementations include Intel's implementation (ICC), Sun's implementation (now Oracle)¹, GNU's implementation (GCC) and OMPI [2].

Another micro-benchmark study was performed by Bull et al. [8] that targeted OpenMP 2.0 constructs on SGI Origin and Sun machines, with similar aspects in mind as [10], in particular, targeting only the thread-centric aspects of OpenMP.

Andersch et al. [3] created a benchmark suite targeting the data-driven type of task-centric parallelism support by e.g. OmpSs [11]. The study includes a series of benchmarks that were annotated with the proposed extensions for implicit task synchronization. Olivier et al. [22] compared different implementation of a benchmark they wrote that aims to be as unbalanced as possible, called UTS. They evaluated the benchmark using different task-centric implementation including OpenMP implementation from Intel, Sun, GCC and the Cilk runtime system. Duran et al. [12] performed an evaluation of different scheduling policies using their runtime system Nanos [28]. They compared the Breadth-First, Work-first, Distributed Breadth-First and Cilk-style scheduler on the BOTS benchmark suite [13]. Olivier et al. [21] compared their scheduler in Qtthreads with Intel's and GCC's OpenMP implementations concerning multi-socket SMPs. They compared different scheduling policies based on work-stealing or work-dealing using a subset of the BOTS [13] benchmark suite. Addison et al. [1] created the OpenMP implementation in the compiler Open64 [17]. They compared their implementation with the Nanos and Cilk implementations on a subset of the benchmarks used in the present study.

We have partly the same aim and purpose with our study as some of the above studies. Our work differs mainly from previous research in that we study a much wider set of both implementations of task-centric run-times as well as in the benchmarks used. Our work include the BOTS benchmark with two additional benchmark added: N-Body simulation and UTS [22]. We have included all previously studied implementations, including the OpenUH's OpenMP implementation and some more.

THE TASK-CENTRIC PROGRAMMING MODEL

In the task-centric programming model the programmer has the responsibility of exposing tasks and synchronization points. A task is any form of computation that can be run

¹SUN is now Oracle but the compiler we use is suncc and thus, we will refer to it as simply SunCC
in parallel, typically either a compound statement, subroutine or function call, or a class method. A synchronization point is a special statement that blocks the execution until certain tasks have finished executing. The models differ in exactly which tasks are in the scope of a particular synchronization point, but in general the scope is the most recently created tasks that have not yet been synchronized. In this case there is an implicit stack of tasks, with a synchronization acting on the topmost ones. The synchronization points are required to ensure that the application respects data dependencies and in general behaves as its sequential counterpart.

We evaluated eight different task-centric implementations. Five of these are based on or inspired by the OpenMP 3.0 methodology: Intel’s OpenMP, SUN’s OpenMP, GCC’s OpenMP (libomp), Nanos++[11] and OpenUH [1]. Of these five, OpenUH and Nanos++ could be considered research implementations while the other three are widely used production systems. Two of the other selected run-time implementations are Wool [14] and Intel Cilk Plus. These share the property of being minimalistic and, unlike the OpenMP inspired implementation, consist of just a handful of keywords to manage the parallelism. The final implementation evaluated is Intel’s Threading Building Blocks (TBB), which is a task-centric C++ library. Our implementation of Intel Cilk Plus and TBB is the one that comes embedded in the Intel CIC++ compiler except for the TilePRO64 study where we built (our port of) TBB with GCC.

Enabling parallelism using OpenMP is performed by annotating the source code with #pragma statements. A #pragma statement is usually transparent to the pre-processor, and handled by the compiler after this phase. Some of the implementations we studied (Nanos++ and OpenUH) use a source-to-source compilation step before invoking the back-end compiler. This makes it possible to use different back-end compilers. For other models (ICC, SUN, and GCC) the OpenMP parsing is integrated with the ordinary parser. In our work, this transformation is done by the respective compilers (ICC, SunCC and GCC). Open64 for the OpenUH implementation, and Mercurium [7] for the Nanos++ implementation. A simple example of an application annotated with OpenMP directives can be seen in figure 1. Here the #pragma omp parallel will start a number of threads that will all execute the compound statement, while the #pragma omp single chooses just one of the threads to execute the compound statement. However, Nanos++ starts all threads at the beginning of execution and therefore, the parallel and single pragmas are not needed.

Exposing a task is done with the #pragma omp task directive, which will create a task out of the following compound statement. The #pragma omp task accepts a number of clauses, all of which can be found in the OpenMP specification manual by the curious reader. In figure 1 we create two tasks where each task sort a chunk the array a (a → left and a → right). Following the creation of the tasks, we insert a #pragma omp taskwait statement which acts as the synchronization point. The directive will ensure that both arrays have been sorted before moving on to the next step. As we merge the two arrays another task is spawned to merge the arrays, and one more synchronization point is inserted before finally finishing the program.

Intel Cilk Plus is based on Cilk++ (by Cilkarts) which itself is based upon research from MIT on the Cilk-5 runtime system [15]. The application code for sorting two arrays and then merging them is shown in figure 2. The Cilk Plus implementation supports two basic constructs: _Cilk_spawn and _Cilk_sync. Although there are more constructs for advanced usage such as task based _Cilk_for loops, we refer the reader to the Cilk Plus reference manual. The source code in figure 2 works in a similar way as the OpenMP example except that

```c
int main (int argc, char *argv[]) {
...
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    sort (a->left);
    #pragma omp task
    sort (a->right);
    #pragma omp taskwait
    #pragma omp taskwait

    _Cilk_spawn merge (a->left, a->right);
    _Cilk_sync;
    }...
Fig. 1. Example source code annotated with OpenMP 3.0 pragmas
```

here the _Cilk_spawn creates a task and _Cilk_sync works as a synchronization point. Similar to Nanos++, Cilk Plus starts all of its threads at application start time and they are not explicit to the programmer.

```c
int main (int argc, char *argv[]) {
...
_Cilk_spawn sort (a->left);
_Cilk_sync;
_Cilk_spawn merge (a->left, a->right);
_Cilk_sync;
...
}
Fig. 2. Example source code using the Cilk Plus unique C keywords for parallel execution
```

Wool [14], being inspired by the simplicity of Cilk-5, provides similar keywords. The keywords are pre-defined macros which transform the task code during pre-processing. Due to Wool’s implementation, a SYNC statement acts as a synchronization point for the most recently spawned tasks. This can be seen in the sorting example in figure 3. Here we are required to use a synchronization point for each of the spawned tasks. Wool does also contain keywords for task-based parallel FOR loops.

Intel Threading Building blocks is a task-centric C++ runtime library. The sample code using TBB primitives is shown in figure 4. In TBB, a task is created by allocating a placeholder of the type task, and using the spawn function to create it. The set_ref_count function sets the amount of tasks spawned in the current function scope. The wait_for_all() statement acts as a synchronization point and can be combined with a spawn (spawn_and_wait_for_all()). Not shown in the example, a task can return a pointer to a task, which is immediately executed by the current thread. Should a task create many other tasks, support exists for creating a list where task are spawned, which is later submitted to the underlying run-time system.

In addition to OpenMP 3.0, Wool, TBB and Cilk Plus, there

2From this point, we will refer the OpenMP run-time libraries based on the compiler they ship with. For example, ICC’s OpenMP becomes ICC and Sun’s OpenMP becomes SUN. This is done to save space and relieve the reader of the abundant redundancy of these terms.
are several other task-centric programming models not covered in this evaluation. Java 7 has its fork/join interface. C# (and other .NET-based languages) can use the Task-Parallel Library (TPL) from Microsoft and Apple has built in a task-centric API in its MacOS (and various *BSD distributions), known as the grand-central dispatch (GCD).

**Introduction to evaluated run-time systems**

All tasks that are exposed by the programmer usually ends up in the scheduler of the run-time system. The scheduler is responsible for 'what task goes where and when'. Find below a description for each of the run-time systems we used:

**a) GCC's libgomp:** GNU's C Compiler's OpenMP run-time system is called libgomp. Libgomp currently supports a majority of OpenMP's clauses with the exception of the `untied` clause which allows for preemption and resumed execution of a task by a different thread than that which started it. The internal structure of libgomp is a centralized queue which in libgomp's case is a linked list. Threads continuously sweep this centralized queue, pushing or popping work from it. Each task also contains a pointer to information relevant to its children. A `#pragma omp taskwait` clause, when encountered by a thread, will ensure that all children of the task are finished before resuming parent execution. Libgomp also have a messaging system implemented using fast-userspace mutexes. In essence, these mutexes control the threads, and activate threads when tasks are spawned or to implement `#pragma omp barrier` synchronization. Libgomp has a in-built cutoff of 64 × Num_Threads amount of tasks; exceeding this limit will enforce tasks to be serially called. The centralized queue is protected by a global lock.

**b) Wool:** In Wool, each thread has a private queue, into which it spawns task. Load-balancing is done by allowing threads to steal work from each other. Each queue acts as a stack so that synchronization will always occur with the most recently created task. Should the last-spawned task be stolen by another core, the synchronizing core will attempt to steal one of that task's children, also known as leap-frogging. Task queues are not protected by locks but rather use atomic instructions to directly manipulate the contents of the queue, potentially leading to better performance.

**c) Cilk Plus:** Intel Cilk Plus, based on Cilkarts Cilk++ which itself is based on MIT Cilk-5[15], uses a depth first parent stealing scheduling algorithm. Parent stealing differs from child stealing (used by for instance Wool and TBB) in that the spawned code is executed immediately, rather than being put in the task queue. Instead, the continuation of the call (the computation that is to be executed when the call returns) is made available for stealing. The benefits of parent stealing is that the single processor execution becomes identical to an execution of a corresponding sequential program, formed by erasing all `#Cilk_spawn` and `#Cilk_sync` keywords from the parallel program. Like in other depth first schedulers, locality is preserved; especially in a divide-and-conquer benchmark where the leafs contain the bulk of the application work. Threads have their own private queues, and when a thread finds no work in its private queue it will proceed to steal the parent (which was usually left as a continuation) of the last task executed. Should that fail, it will randomly select a victim to steal from.

**d) SunCC / Oracle:** We failed to discover any source of information concerning the internal mechanisms of SunCC.

**e) Intel TBB:** Intel TBB is a work-stealing C++ based API aimed towards objective-oriented programming. In its basic form it consists of three stages. The inner-stage is the one that executes local tasks or private tasks. If there are no local tasks, the scheduler proceeds to steal a task from another thread and continue to execute the local tasks within the stolen task. Stealing tasks will continue as long as the root node for the application exists, i.e. as long as the first task have not had its children completed.

**f) OpenUH:** OpenUH is the run-time system for the Open64[17] compiler. In OpenUH queues are distributed and implemented as linked-lists, protected by locks. Since both `tied` and `untied` clauses are supported, each worker contains besides the exposed distributed queue also a private queue. The private queue is used to store `tied` tasks that have started execution. Due to the distributed nature of the queues, work-stealing is employed to load-balance the system; work-stealing is initiated when a worker’s queues are empty. OpenUH implements a cutoff of a type called `depmod` and `queuemod`. Depmod is when the current depth of the task (with respect to the root) is tested against a modulus operation of two to decide when the execute a task serially. The `queuemod` is a overflow check on a current queue to decide when to spawn a task.

**g) OpenMP:** Intel's OpenMP library uses a distributed queue approach, with work-stealing as method to load-balance tasks across workers [16]. The work-stealing is performed randomly. However, thieves are recorded and become the first objective of an idle worker. Task-cutoff is implemented when the thread-local queue is full at which point tasks are executed immediately. The queues are protected by locks.

**h) Nanos++:** Nanos++ is the run-time system for most of the *Ss task-centric models, such as OmpSs, StarSs and SmpSs ([11], [6], [18]). As such, it is not limited to supporting OpenMP-only clauses, but can also support a different style of task synchronization such as the Hierarchical (Implicit) Synchronization node [26]. Being a research vehicle, it is

```c
int main (int argc, char *argv[]) {
    ...
    SPAWN (sort, a->left);
    SPAWN (sort, a->left);
    SYNC (sort);
    SYNC (sort);
    SPAWN (merge, a->left, a->right);
    SYNC (merge);
    ...
}
```

Fig. 3. Example source code using the Wool macros for parallel execution

```c
int main (int argc, char *argv[]) {
    ...
    tbb::task *t1 = new (allocate_child())
        sort (a->left);
    tbb::task *t2 = new (allocate_child())
        sort (a->left);
    set_ref_count(3);
    spawn (*t1);
    spawn_and_wait_for_all (*t2);
    tbb::task *t3 = new (allocate_child())
        merge (a->left, a->right);
    set_ref_count(2);
    spawn_and_wait_for_all (*t3);
    ...
}
```

Fig. 4. Example source code using the TBB methods for parallel execution
highly parametrized with options including different scheduling policies, variable cutoffs (in both depth and num) and back-off mechanisms with concern to locks. We chose to include Nanos++ as a representation for two different kinds of schedulers; the Breadth-first and the Depth-first scheduler [28]. The Breadth-first scheduler works similar to gcc’s libgomp in that it is a centralized queue where all work is contained. The Depth-first is the complete opposite having distributed queues with work-stealing between workers. This enables us to study two widely used approaches and compare their performance since they are using the same underlying mechanism. Queues are protected by locks.

BENCHMARKS

We evaluated all the parallel libraries using the Barcelona OpenMP Tasking Suite [13], BOTS, a benchmark suite specifically tailored to exploring OpenMP task-level parallelism. We ported the entire BOTS suite to the other task-centric frameworks we studied. The BOTS benchmark suite contains nine benchmarks, listed below. We choose to exclude Health from the benchmarks as it is written in a unoptimized and generally poor way, as remarked by [29]. The benchmarks were compiled using the flags shown in Table 1.

1) **Protein Alignment** aligns a sequence of proteins according to the Myers and Miller algorithm.
2) **Fast-Fourier Transform** calculates the fast-fourier transform based on the Cooley-Tukey algorithm. Its recursive nature break the input DFT into smaller DFTs that are exposed as tasks.
3) **Fibonacci’s Series** calculates the fibonacci series for a certain in-parameter. It is recursive, and is probably the most common application to introduce newcomers to task-centric parallelization.
4) **Floorplan** is an application that takes a number of blocks, and tries to place them onto a floor so that they occupy as little area as possible.
5) **n-Queens** is a benchmark that tries to solve the n-Queens puzzle; given a NxN chessboard, and N queens, in how many different ways can the N queens be placed onto the chessboard so that no queen threatens another. The benchmark places queens recursively in different positions on the chessboard, where each position is evaluated by a task.
6) **Merge-Sort** is a parallel version of the by Von Neumann discovered Merge-Sort that recursively divides an unsorted array until there is only one element in each sub-arrays, and then merge the sub-arrays to get the final sorted array.
7) **SparseLU** is a Sparse Matrix LU factorization over several sub-matrixes.
8) **Strassen Multiplication** is an improved matrix multiplication. The initial arrays are sub-divided into tasks that each multiplies segments of the arras in parallel.
9) **Unbalanced Tree Sort** (UTS) is a benchmark from Olivier et al. [23] designed to study load-balancing issues. It recursively spawns tasks with a certain probability to spawn further tasks. The benchmark itself is deterministic in terms of how many tasks are spawned and executed, but is highly irregular and unbalanced, hence the name.
10) **N-Body simulation** calculates the forces amongst celestial bodies based on classical newtonian physics. The algorithm is $O(n^2)$ in complexity (not Barnes-Hut) and sweep through all bodies, calculating and updating the new position for each.

EXPERIMENTAL PLATFORM

We performed the evaluation of the targeted run-time system on two different platforms; a large SMP multiprocessor server and an embedded platform. The SMP multiprocessor system is a 48-processor ‘Magny-Cours’ system by AMD, containing 4 NUMA sockets, where each socket contain 8 GB RAM and 2x6 x86-64 processors (Opteron 6172). The strengths of using a large SMP-NUMA system is that we can draw conclusions on a) how a run-time system would scale on a general-purpose processor (using 1 socket), b) how a run-time system performs on a large SMP machine, and c) How an HPC machine would be affected by the intra-node performance of a run-time system since it is common to utilize MPI for extra-node communication and e.g. OpenMP intra-node.

Our embedded platform was the TilePRO64 processor from Tiler. The TilePRO64 is a processor with 64 general-purpose, three-wide issue VLIW CPUs. They are inter-connected with an 8x8 mesh based network-on-chip (NoC) and share 4 memory controllers, which are address interleaved. The TilePRO64 processor does not contain any hardware floating-point units, and because of this, floating-point operations are comparatively slow. TilePRO64 operates on a constant frequency of 700 MHz.

EXPERIMENTAL METHODOLOGY

We have performed a series of micro-benchmarks to study certain isolated aspects of each task-centric model implementation and then application on the benchmarks mentioned above.

Micro-benchmarks

We performed a series of micro-benchmark tests to reason about the performance of the different task-centric programming libraries that we explored. The micro-benchmarks were created to stress and measure different parts of the API exposed to the programmer. We used the gettimeofday() system call and the hardware timestamp counter to measure the timing within the micro-benchmarks.

Run-Time library overheads: Figure 5 shows the time to expose a single task (a), and the time to synchronize with the said task (b). We notice that time to expose a task is several orders of magnitude faster in Wool and Cilk Plus compared to the rest of the models. The slowest implementations are Nanos++, GCC and SunCC. Although there are several ways to improve this metric, such as inlining a task (done in Nanos++ and gcc), these measurements are without taking that into consideration since inlining a task will serialize the
operation. Similarly, we see that while synchronizing with a task is roughly equal across the libraries, Ns++ takes many times longer to synchronize with a task compared to the others.

Figure 6 shows the results of two other micro-benchmarks. The first one (a) is a flat benchmark that contains a certain work that is divided into tasks. We start by dividing the work into very many small-grained tasks and progressively decrease the task count (and thus increasing the granularity). We record the time it takes for each of the implementation to execute this benchmark. We see that Wool handles fine-grained tasks very well, and the executing time roughly remains constant as we increase the granularity. The other implementations perform typically an order of magnitude slower than Wool for small granularities and converge at different granularities indicating variance in the resilience to fine-grained parallelism.

The one exception is Ns++ (DBF) which, when the slow task exposure rate is combined with having distributed queues making it perform even worse. Note that Ns++(BF), being a centralized queue approach performs much better than Ns++(DBF) for flat applications. Figure 6 b) shows the same application but using a divide and conquer methodology. Here, Cilk Plus together with Wool and TBB contend for being the fastest, while the Ns++(BF) scheduler performs the worst. This is a fine example of the differences between a distributed (Ns++[DBF]) and a centralized (Ns++[BF]) queuing strategy. Note that all implementations converge to the same point as the parallelism is reduced to only one task.

Load imbalance: Load-balance is the property of equalizing the load across the resources to maximize the amount of computation occurring concurrently and is probably the single most impacting factor when it comes to the execution performance and scalability of applications, assuming that the application has the potential to scale.

We measured the load-balancing features using two methods. The first method involves recording the amount of time each thread performs application work; if a large deviation is found across threads, this would indicate a load-balance issue. Similarly, we also recorded the amount of time a thread spends not performing application work; this takes the library overheads and the run-time system into account. The second method we used was to calculate the average number of threads that are executing application work simultaneously;

---

**TABLE I**

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compilation Flags for the Different Task-Centric Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC libgomp</td>
<td>gcc -O3 -m64 -lopemmp gcc 4.7</td>
</tr>
<tr>
<td>ICC OpenMP</td>
<td>icc -O3 -m64 -openmp icc 12.0.1</td>
</tr>
<tr>
<td>Cilk Plus</td>
<td>icc -O3 -m64 icc 12.0.1</td>
</tr>
<tr>
<td>TBB</td>
<td>icc -O3 -m64 icc 12.0.1</td>
</tr>
<tr>
<td>SunCC (Oracle) OpenMP</td>
<td>suncc -O3 -m64 -openmp=parallel suncc 5.11</td>
</tr>
<tr>
<td>Ns++</td>
<td>micc -O3 -m64 -openmp nns++ 0.7/a</td>
</tr>
<tr>
<td>Wool</td>
<td>gcc -O3 -m64 wool.o -lpthread gcc 4.7</td>
</tr>
<tr>
<td>Open/H</td>
<td>micc -m64 -o -lpthread</td>
</tr>
<tr>
<td>TBB</td>
<td>micc -g++ -O2</td>
</tr>
<tr>
<td>Ns++</td>
<td>micc(tic-cic) -O3 -m64 -openmp nns++ 0.6a</td>
</tr>
<tr>
<td>Wool</td>
<td>gcc -O2 wool.o -lpthread gcc 4.4.3</td>
</tr>
</tbody>
</table>

---

The graph shows the overhead cost for flat (a) and recursive (b) type of benchmark with varying granularity. All times are in nano-seconds.

In overall, there is not much difference between the benchmarks compiled with different compilers. The most prominent difference is in the n-body and n-queen benchmark. The n-body benchmark experience a speedup of nearly 80% when compiled with the Intel C compiler, something that is also reflected in the speedup of ICC having a superlinear speedup of up to 70x on a 48-core machine. N-queens receive a 25% speedup when compiled with Open64, something that will appear some of the experimental results. Sun’s C compiler often generates particular bad code, especially in the N-body and Alignment case. We will refer to this section when commenting on the compiler difference’s impact on the application speed-up.
Benchmark metrics

Speed-up performance measurements: The speed-up study is intended to give an overview of how the different models behave in terms of raw performance, which is what the end-user cares about. However, we have used it to find any discrepancies between the benchmarks and programming models. We executed all the benchmark:mode:thread_count configurations ten times, and took the median to represent the execution time that would normally be seen by the user. The measured time is taken when the application enters the parallel region of the benchmark. The parallel region is where the fork to parallel is done for the first time. The time was measured using gettimeofday(). To calculate the speed-up of the application we normalized the parallel execution for each of the combinations against the serial version of the same benchmark/combination without parallel runtime system calls. The serial benchmark was compiled with GCC 4.7 with aggressive optimization flags (-m64 -O3). The compilation flags for each of the different implementations can be seen in table 1.

Cache Effects: Our approach to measure the caching and resource performance of tasks was to profile only the work within a task rather than to include run-time system overheads. This allows us to compare how the behaviour of the total amount of factual work changes with different parallel runtime system. These measurements include the effect of task-placement and cache-thrashing due to run-time activity. We used the AMD Opteron hardware performance counters as exposed by PAPI [20] to sample and profile the work within tasks. We annotated the application source code to record when application code is executed.

Special care needs to be taken at places when the run-time system can be invoked as they might lead to a task switch. We used the L1 DCA and L1 DCM counters to reason about the L1 data-cache miss performance, and the L2 DCA and L2 DCM counters to reason about the L2 data-cache miss performance. The cache miss ratios was for the un-stalled application time; that is (tot_cycle – resource_cycle) which remain constant for the same benchmark on different run-time systems.

Similarly, we used the RES_STY performance counter to record the aggregated amount of stall cycles the entire application experience as a result of the run-time system. The RES_STY performance counter record the amount of cycles that a processor is stalled due to any resource activity. Again, we would like to remind the reader that the profiling was done on a application level; since the amount of instructions is the same (as the code is the same) for the application, this will provide an apple to apple comparison on how the runtime system affects the code executed.

Tied versus untied: Tied or untied is a property of a task under the OpenMP framework. A untied task can start executing on one thread, and finish executing on another thread. A tied task, once started by a thread, is required to be finished by that thread and that thread alone. For our experiment, we executed both the tied- and the untied-version and used the results for the best performing one. This was usually the untied-one. The untied/tied discussion does not concern other run-time implementations, as they only use one or the other. For instance, tasks in Cilk Plus are untied (otherwise parent stealing would be impossible) while in Wool and TBB (which are child stealers) they are tied.

Embedded Power measurements: To perform power-analysis, we attached a National Instruments data acquisition device (NI USB-7140) to the power rails of the TilePRO64 as per the documentation regarding power measurements. Similar studies has been performed by Själander et a [27]. The power consumption sampling does not interfere with the program execution in any way. Furthermore, we use a metric we called speed-up power cost which is the amount of speed-up that is experienced per added Watt; that is, how much more speed-up do you get for the amount of power used?

EXPERIMENTAL RESULTS AND DISCUSSION

AMD Opteron performance

Alignment: The first benchmark we will look at is the Protein Alignment benchmark. It contains no recursion, and is flat where only the root node contains children. Here we see that most of the libraries scale fairly well (figure 8 a) except for SunCC and TBB. Generally with flat parallel applications, unless the scheduler is locality aware, typically there is no advantage of using a distributed compared to a centralized queue; in fact, it can even hurt performance. The two known approaches that uses a centralized queue is the Nanos++ (BF) and GCC run-time systems; both of which scale very well on this application. Intel TBB is most poor performing of all the run-time system and this is because of the slowdown of the application itself. In fact, the scaling of protein alignment on TBB ($\alpha=44.61 @ 48$ threads) is linear when compare to the one threaded TBB version. SunCC($\alpha=44.88 @ 48$ threads) suffers from degraded program code due to compiler differences seen in 8:b, and both SunCC and ICC($\alpha=38.46 @ 48$ threads) spends far more time in idle mode (relative to Wool, see 8 b).

Floorplan: On Floorplan (figure 9) there is a rather large spread concerning the performance using a tree-depth of 6, while when increasing the tree-depth to 10 the run-time systems converge into two groups; one that scales and one that doesn’t. In general, the best performing run-time systems are Wool, Cilk Plus and ICC. OpenUH performs very well for a tree-depth of 6 which can be attributed to the compiler since, as we increase the tree-depth to 10, OpenUH is on par (even with the benefits of the better compiler) to Wool, Cilk Plus and ICC.

During the transition from a tree-depth of 6 to a tree-depth of 10, OpenUH changes it’s behaviour from an $\alpha=22.27$ at 48 threads. Similarity, Nanos++ (BF)($\alpha=14.9$), Nanos++ (DBF)($\alpha=39.44$) and GCC($\alpha=27.48$) at a tree-depth of 6 are degraded to $\alpha=12.54$ and 1.21 respectively when...
increasing the tree-depth. SunCC does not perform at all beyond 8 threads, with lower-than-serial slowdown experienced (speed-up only occurs with much lower tree-depth).

As with Protein Alignment, TBB experience a slow down when using one thread compared to the serial version. The scaling of TBB, compared to it’s own 1 threaded version would is around 25x at 48 threads for both tree-depths and the runtime is very stable in terms of load-balancing for both tree-depths.

**UTS**: UTS is a benchmark designed to be poorly balanced and to stress the runtime system in ways concerning load-balancing and fine-granularity. We see that nearly all the runtime system (figure 10) perform sub-serially except for TBB, ICC and Wool. It may be noted, that it seems that the resilience to fine-grained tasks together with the load-balancing feature of Wool enables it to scale almost linearly with the number of threads.

**N-Queens**: N-Queens (figure 11) is another benchmark where, as we increase the amount of parallelism (and thus making it more fine-grained), the performance of some of the libraries decreases. This presents us with a very clear grouping; those capable of handling fine-grained parallelism and load-balance it properly and those suffering from either load-balancing related issues or, simply speaking, have large amount of overheads.

The only three run-time systems capable of handling n-Queens with a tree-depth of 7 (figure 11 b) are Wool, Cilk Plus and Intel TBB. Even OpenUH, which performed remarkably well at a tree-depth of 3 (figure 11 a), thanks to the n-Queen having better code generation on Open64, performs ‘sub‘ optimal when exposing a fair amount of fine-grained parallelism.

**Strassen**: For the Strassen’s matrix multiplication (figure 12), all library implementations performs reasonably well at a sub-matrix granularity of 128x128 (a). However, as we reduce this to 32x32 (b) we see a general decrease in scalability for some of the libraries in particular GCC(α=3.38 @ 48 threads), Nanos++(DBF)(α=19.9 @ 48 threads) and Nanos++(BF)(α=18.21 @ 48 threads). If we look closer at the L2 miss ratio for the entire application code, we see that the Nanos++(BF) implementation suffers from increased L2 miss ratio compared to the other libraries (figure 12 c).

There is also a large deviation in the amount of work each thread performs; both GCC and Nanos++(BF) show problems with load-balancing compared to open64(α=27.5 @ 48 threads) and Nanos++(DBF) (figure 12 d). Note that the the amount of work performed is different between libraries due to the resource stalls experienced by each model. Similarly, for Nanos++(BF) and GCC, each thread’s time spent not in application mode is an order of magnitude higher than for both Nanos++(DBF) and Open64, leading to poor scalability. Also the time the entire application code experiences resource stalls is much higher for Nanos++(BF) than for the other implementations, leading to performance degradation (figure 12 e).

**FFT**: The FFT benchmark scales well for very few of libraries (figure 13 a). Wool performs the best, followed by TBB, Cilk Plus and ICC. If we look at why some of the libraries perform badly, we see that compared to Wool (α=18.21 @ 48 threads), the other libraries have experience a degraded L2 cache miss ratio(figure 13 b). The L2 cache miss ratio is also reflected in the amount of time the applications experience resource stalls, Nanos++(BF)(α=7.4 @ 48 threads) experiences almost 9x times as many resource stall cycles compared to Wool, and 7x as many compared to its cousin, Nanos++(DBF)(α=14.36 @ 48 threads). Similarly, the corresponding stall cycles for GCC(α=4.27 @ 48 threads) and for SunCC (α=13.02 @ 48 threads) is slightly higher than for Wool. More interesting is the load-balance graph,
shown in figure 13 c. Here we see the threads on the x-axis, and the amount of application work performed on the y-axis. As with Strassen, both GCC and Nanos++(BF) have a poor load-balance while Nanos++(DBF), suncc and especially Wool balances more equally. This can also be seen in the idle-time graph (figure 13:d), where both GCC and Nanos++(BF) spend much more time in the run-time library, rather than in the application as Nanos++(DBF), suncc and Wool is doing.

SparseLU: The SparseLU benchmark performs generally very well on all run-time implementation except Nanos++(DBF) and Nanos++(BF) (figure 14 a). We focus on explaining the difference between Wool (α=37.45 @ 48 threads, performing well), Nanos++(DBF) (α=29.29 @ 48 threads, performing poorly) and SunCC(α=33.00 @ 48 threads, inbetween).

The L1 cache figures for all three run-time implementations are roughly the same; however, we see in the L2 cache miss ratio (figure 14 b) for the application code that, although
the miss ratio is the same across run-time implementation, the Nano++(DBF) experiences nearly twice as the L2 cache miss ratio when using 32 threads. This is the major reason behind the performance dip at 32 threads for Nano++(DBF), seen in figure 14 (a). This was further verified by examining the resource stalls which showed an 30% increase in resource stall cycles compared to the said library at both 24 and 40 threads. There is also a large difference in the idle time of each run-time systems. SunCC spends 1.65x more time per thread being idle compared to Wool, while Nano++(DBF) spends almost twice the time compared to Wool being idle.

**N-Body:** The performance results for the N-Body benchmarks can be seen in figure 15. It is clear that the speed-up for the Intel compiler based libraries (Cilk Plus and ICC, but not TBB) is due to increase compiler performance, as reported under the compiler differences section. Worth saying is that the Nano++(BF) scheduler rarely completes this benchmark, for reasons unknown to us. We scrutinized the N-Body execution for Cilk Plus ($\alpha=43.71 @ 48$ threads, performing well), TBB ($\alpha=46.91 @ 48$ threads), SunCC ($\alpha=46.91 @ 48$ threads) and Nano++(DBF) ($\alpha=45.66 @ 48$ threads); notice how the high $\alpha$ indicates that the said library implementations do indeed execute work concurrently, often near the thread limit (48). The L1 cache effects showed little difference in between the said libraries, while the L2 cache miss ratio differed by 1.2% between the models. Oddly, TBB performs vastly worse than the other libraries. This was verified by looking at the resource stall cycles, where TBB experiences almost 9x more stall cycles than Cilk Plus, 4x more than Nano++(DBF) and 2.4x more than SunCC. We failed to identify the source of this performance degradation.

**Fibonacci:** For the Fibonacci benchmark, traditionally used in teaching the basics of task-centric programming, we used two tree-depths: 15 and 20. For a tree-depth of 15, most run-time systems perform well save for Nano++(BF) and GCC, which require the task granularities to be even more coarse-grained to scale linearly. Increasing the cutoff to 20 only four run-time systems manage to maintain scaling throughout all the threads; Wool, ICC, Cilk Plus and TBB. The other perform sub-optimally due to load-balancing and, more importantly, library overheads.

**Multisort:** The Multisort benchmark speed-up results are seen in figure 17 (a) and (b). This is a benchmark that is known to scale poorly. For a relatively large cutoff with sub-matrix sizes at 65k (a) we see that most run-time systems performs similar to each other and the limiting factor is actually the amount of parallelism exposed by the application itself. Decreasing the granularity to 1024 (b), both GCC ($\alpha=1.6 @ 48$ threads) and Nano++(BF)($\alpha=1.3 @ 48$ threads) does not scale while Nano++(DBF)($\alpha=5.5 @ 48$ threads) and SunCC($\alpha=5.1 @ 48$ threads) halved their performance compared to (a). Looking at the differences between the said poor-performing run-time systems and TBB($\alpha=8.1 @ 48$ threads) (which performs rather well) we see that the L1 cache miss ratio for when the cutoff is set to 1024 sub-array elements, SunCC and GCC both have a pretty stable miss ratio of roughly 0.6% throughout the thread count; Nano++(BF) experience a slightly higher miss ratio (still stable across threads) of 1.7% which TBB performs the best with a stable L1 miss ratio of 0.14%.

Interestingly, Nano++(BF) starts off somewhere between Nano++(DBF) and GCC (1.0%) but decreases the L1 miss ratio to the same level as TBB at 48 threads (0.19%). However, while the L2 cache miss ratio stays the same throughout the number of cores for SunCC,GCC, Nano++(DBF) and TBB, the Nano++(BF) scheduler experience a rather large L2 miss ratio degradation and, at 48 threads, is as high as 48%; this compare to the other said run-time systems that are around 10%. This effect is also reflect in the amount of stall cycles (figure 17 c) the Sort application experience when executing using Nano++(BF). In fact, Nano++(BF) experience nearly two orders of magnitude more stall cycles
than TBB. In terms of load-balancing, TBB spends half the time of Nanos++(DBF), a quarter of the time of SunCC and nearly two orders of magnitude less time than Nanos++(BF) and GCC idling in the run-time system.

**TilePRO64 performance results**

We performed analysis on a subset of the benchmarks\(^4\) as well as a subset of the runtime libraries that were available to us for the TilePRO64 embedded platform. The libraries we chose were GCC’s libgomp, Intel TBB (ported by ourselves), Wool and Nanos++ (also ported by ourselves). We evaluated the following benchmarks: Alignment, SparseLU, Strassen, FFT and N-queens. For all experiments we measured the exact power consumed by the application. The power recording starts when the benchmark sends a message to our measurement system when reaching the first parallel construct. The power ends when all parallel constructs (the application) finishes. There is a minimal error in the power measurements which is the time it takes for the start and end messages to be sent across PCI; this error is marginal.

**Strassen:** The speed-up result for the Strassen benchmarks on the TilePRO64 can be seen in figure 18 (a). In general, Wool outperforms the other programming models by almost 2.5x at 48 threads. The other models share similar execution performance. In terms of energy, Wool has the best a speed-up power-cost (and the only one to scale to 56 threads) of 1.35x/Watt (@ 56 threads); this compared with GCC (0.93x/Watt @ 40 threads), TBB (0.78x/Watt @ 32 threads), Nanos++(bf)(1.03x/Watt @ 48 threads) and Nanos++(dbf)(0.96x/Watt @ 40 threads) making Wool both the most energy-efficient of the programming models. The power-traces for the execution at each run-time systems peak speed-up power-cost can be seen in figure 18 (b). The power-trace verifies the results; Wool computes much faster (at a higher power-consumption rate) than the other models while TBB takes longer.

**Alignment:** Protein alignment, where the speed-up can be seen in figure 19 (a), is more equal in terms of execution performance and speed-up power-cost. Wool (0.83x/Watt @ 56 threads) and Nanos++(bf)(0.26x/Watt @ 56 threads) both perform better and are more power-efficient than the other three models. The only run-time system not to increase its speed-up power-cost up to 56 threads is GCC (0.48x/Watt @ 48 threads) where adding those 8 extra threads degrades the speed-up. In figure 19:b we see the power-trace for the optimal speed-up power-cost. Notice how the Nanos++(DF) scheduling algorithm bounces between being 20 Watts and 12 Watt indicating a poorly utilized system. Also here we see that Wool finishes the application faster followed by Nanos++(BF).

**SparseLU:** On the SparseLU benchmark (figure 20 a), the Nanos++ run-time system does not perform well with any of its schedulers and, even though the speed-up power-cost does increase with the the number of threads (up to 56 threads), it reaches only as high as 0.37x/Watt (that is, 3 Watts are needed to double the performance at best). Wool performs the best with a speed-up power-cost scaling up to 48 threads with a maximum of 1.26x/Watt. TBB (0.9x/Watt @ 32 threads) slightly outperforms GCC(0.83x/Watt @ 56 threads) in both aspects of the performance. Figure 20 (b) shows the power-trace for the SparseLU benchmark at the optimal speed-up power-cost of each run-time system, where we clearly see the reduced execution time (often at expense at higher power-consumption) of the run-time systems. Also note how poorly the Nanos++ run-time system utilizes it’s resources, under utilizing them (indicated by the ‘oscillation’ in figure 20 b).

---

\(^4\)Mainly due to space-limitations except for floorplan, which for some reason does not work well with the Nanos++ runtime system on the TilePRO64.
**FFT:** Due to the extreme slow-down of FFT (figure 21 a) when using the Nanos++ run-time system we chose to exclude them from the graphs. Interestingly, Wool (2.54x/Watt @ 56 threads) scales linearly with the amount of cores; something it did not do on our x86-64 platform. The explanation is that the TilePro64 both has worse floating point performance and faster inter core communication, effectively increasing the granularity of the computation as compared to the x86-64 server. As on the x86-64 FFT, TBB (1.46x/Watt @ 48 threads) outperforms GCC (1.07x/Watt @ 48 threads) in both performance and energy consumption. The power-consumption trace (figure 21 b) also shows this where Wool uses 176 Joule to perform the work while TBB and GCC consumes 306 Joules and 417 Joules respectively.

**N-Queens:** In general, N-Queens (figure 22 a) is a gentle application with generous amounts of parallelism and is also not too unbalanced. It thus comes as a surprise that only Wool and Nanos++(BF) scales to more than 30 threads; this is when we tweaked the tree-depth for different models to achieve the best performance. This is also reflected in the speed-up power-cost where Wool (2.731x/Watt) tops the power performance followed by Nanos++(BF) (1.98x/Watt @ 48 threads). TBB (1.44x/Watt) and Nanos++(DBF) (1.30x/Watt @ 48 threads) share a similar speed-up power-cost while GCC is down to
Fig. 23. Intentionally limited parallelism of the Strassen application to notice the resource utilization of run-time systems when given 56 threads.

0.91x/Watt @ 48 threads; all of this is reflected in the speed-up figures. The power-trace for the optimal speed-up power-cost is seen in figure 22 (b) where we see the familiar burst in power consumption of Wool followed by Nanos++(BF). GCC takes the longest time and is also the least energy efficient.

Utilization: To conclude the power measurements, we want to briefly mention another set of experiments. They involve changing the benchmarks to intentionally to briefly mention another set of experiments. They involve Nanos++(BF) and TBB(217 Joules, 12.98 Watts) and energy consumption (20.94 Watts) and energy consumption (295.68 Watts) of spinning on queues shows a dramatic increase in both power. Wool which, as shown in this paper, performed quite well in reducing the scalability to eight threads but still allocating 56 parallelism to see what the run-time system does (in terms of energy consumption) when threads have no work to perform. Figure 23 shows one such example, where Strassen matrix multiplication was executed with 512x512 sub-matrix sizes, reducing the scalability to eight threads but still allocating 56 threads to the run-time system. Here we see a drawback of Wool which, as shown in this paper, performed quite well in most experiments. Even though the benchmarks only allows for eight threads to be used effectively, Wool’s aggressive form of spinning on queues shows a dramatic increase in both power consumption (20.94 Watts) and energy consumption (295.68 Joules) compared to e.g. GCC (186 Joules, 12.98 Watts), Nanos++(196 Joules, 13.10 Watts) and TBB(217 Joules, 12.98 Watts) where the later also took 2 seconds longer to execute.

CONCLUSIONS

We have evaluated a set of task-centric libraries and their implementations. We ported a set of benchmarks that target the task-centric programming approach to several of the popular (commercial and non-commercial) task-centric implementations available and evaluated their performance on an architecture that contains both NUMA properties and several processing cores, as well as an embedded NUCA manycore processor. We have analyzed the results of each of the task-centric implementations with respect to several aspects. Experimental results shows that future task-centric run-time system implementations should focus on library overheads and load-balance (which indirectly includes cache performance), which seem to contribute the most towards non-linearity in speedup; especially when the application is parallelized using very fine-grained tasks.

Our results and conclusion apply to both HPC, general-purpose computing as well as embedded devices. Judging from the results, it would be wise to consider alternative solutions for HPC, for example MPI+IWool or MPI+TBB rather than the MPI+OpenMP approach that is common today.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Community’s Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (www.encore-project.eu), grant agreement nr. 248647. The authors are members of the HiPEAC European network of Excellence (http://www.hipeac.net). Furthermore we would like to thank Magnus Sjölander for his assistance in the setup of the power measurement tool.

REFERENCES