Tuning the Victim Selection Policy of Intel TBB

Alexandru C. Iordan[a,*], Magnus Jahre[a], Lasse Natvig[a]

[a]*Norwegian University of Science and Technology, Trondheim, Norway*

Abstract

The wide adoption of Chip Multiprocessors (CMPs) in almost all ICT segments has triggered a change in the way software needs to be developed. Parallel programming maximizes the performance and energy efficiency of CMPs, but also comes with a new set of challenges. Parallelization overheads can account for sub-linear speedups and can increase the energy consumption of applications. In past experiments we looked at specific operations such as spawning new tasks, dequeuing the task queue and task stealing for Intel TBB. Our results showed that failed steals account for the largest overhead. In this work, we focus on TBB's victim selection policy. We implement a new occupancy-aware policy and we improve the implementation of the pseudo-random policy we proposed in a previous paper. We compare the results of our new policies against an "oracle scheme" as well as against TBB's random victim selection approach. Our results show improvements in execution times and energy-efficiency of up to 11.23% and 14.72% respectively when compared to TBB's default policy.

Keywords: Intel TBB, victim selection, parallelization overheads.

1. Introduction

With Chip Multiprocessors present in almost any computing device today, software developers need to leverage the potential of this hardware and move towards parallel implementations. Parallel programming is a challenge mainly because there is no widely adopted programming model that facilitates easy

*Corresponding author
Email address: iordan@idi.ntnu.no (Alexandru C. Iordan)

parallelization. Parallel software development requires tools and methodologies to reduce time-to-market and maintenance effort. Over the last years, industry and academia have developed several parallel libraries that aim at improving application portability and programming efficiency [1, 2, 3, 4].

The introduction of CMPs almost a decade ago has enabled the mitigation of development constraints like the *power wall* and the *ILP wall* [5]. The performance potential of CMPs lies in exploiting thread level parallelism which means that parallel software is required to fully take advantage of this architecture. Intel's Thread Building Blocks (TBB) [4] is a runtime library designed to encourage software developers to create portable, parallel applications with task parallelism. TBB was developed to dynamically scale on the existing resources and employs task stealing to deal with workload imbalance. It was designed to allow developers to focus on parallelizing their code by providing a runtime system that handles parallelism management.

The cost of TBB's dynamic parallelism management is increased parallelization overhead. Developers may have to harness fine-grained parallelism from their applications in order to fully utilize a CMP's resources and this can incur high parallelization overheads. Understanding and limiting these overheads is a necessary step towards scalable and more efficient runtime parallel libraries. To this end, we investigate the extra instructions added by parallelization management and the energy consumption of these instructions which we refer to as the *energy footprint*. More precisely, the energy footprint is the energy spent for executing the given application or section of code in the context of the test system.

Our paper makes the following important contributions:

- We continue our study of the parallelism management costs of TBB [6, 7] and their impact on a CMP's energy efficiency. To allow for extensive and noninvasive measurements under increasing core counts, we use a performance simulator and a power estimation tool in our study.

- Extending our study into victim selection policies [7], we show that we

can reduce thread contention and improve both execution times and the energy-efficiency of a parallel application when making an informed selection rather than a random one.

- We do a comparative study of several selection policies to show that with increasing core counts, the random victim selection policy employed by TBB is a serious performance bottleneck.

Our experiments show that parallelization overheads can cause sub-linear speedups leading to an increased energy consumption for parallel applications. In this paper, we look into mitigating the impact of these overheads and thereby reducing thread contention for hardware resources. By changing TBB's random victim selection policy to an occupancy-aware or even to a pseudo-random policy we can achieve better performance or improved energy efficiency.

The paper is organized as follows: Section 2 gives a general description of Intel TBB and its mechanisms for parallelizations. Section 3 presents more details about the victim selection policy used in TBB as well as the policies we propose. The simulation tools and the benchmarks used in our experiments are described in Section 4. In Section 5, we present our study of the victim selection policies. Section 6 presents the related work and Section 7 concludes the paper.

2. Intel TBB

The concept of parallel programming is almost as old as the computer itself, yet it is a challenge for most developers. In today's *multi-core era* the overall efficiency of the system suffers if parallel applications are not developed to dynamically scale and take advantage of all the resources that are available to them. Over the years, many parallel languages have been developed and a multitude of research was done in an effort to improve performance and maximize hardware utilization [8]. With the majority of those approaches, one factor was often overlooked: the *composability* of the resulting solution. Composability

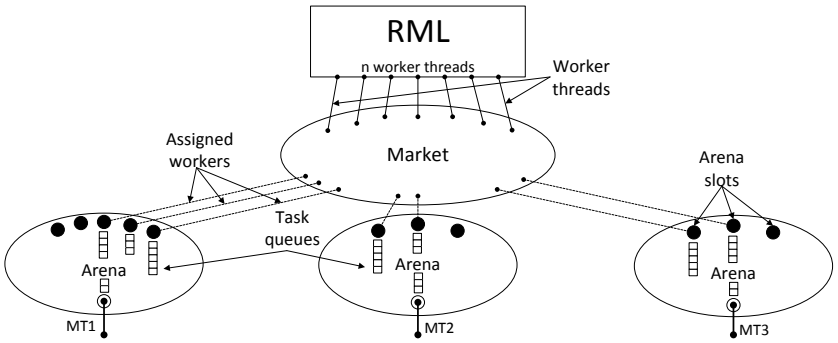

Figure 1: Components of TBB's task scheduler

of an application refers to its ability to run efficiently side by side with other applications and to be able to cope with the fact that it does not have exclusive access to the hardware resources [9]. We see this characteristic as a requirement for efficient exploitation of CMPs. For this reason, we focus on Intel's TBB version 4.1.1, which was designed to provide a high degree of composability.

Figure 1 gives an overview of the structures TBB maintains in order to create and balance its parallel executing threads. The library allows parallelism to be annotated both explicitly and implicitly. Explicit task creation is achieved through the use of methods like *spawn()* which gives the programmer complete control over the work performed by each task. Implicit task creation makes use of some templates like *parallel_for* or *parallel_reduce* which make code writing faster but gives control of the task creation over to the TBB library. Tasks are created and then added to the calling thread's task queue inside the *arena* (see Figure 1). From the arena the task is available for execution by its owner thread or by other workers through stealing. A task can instantiate and spawn other tasks resulting in a hierarchical task tree.

A TBB *master thread* (*MT*) is an application thread that instantiates the *tbb::task_scheduler_init* object. All threads created by TBB to help complete the work of the MT are called *worker threads*. The *Resource Management Layer* (*RML*) is the component that hosts the pool of worker threads and gets

instantiated first (see Figure 1). No worker threads are created at this point, this being postponed until the first task is spawned.

Continuing top-down in Figure 1, the *Market* is instantiated. This component was added in version 3.0 of TBB to ensure the composability of the framework. It separates the workload (the tasks) of one MT from other MTs that may be executing on the same machine. The role of the market is to assign workers to the arenas of each MT. The limit of the total number of workers available is set to 1 less than the maximum of the argument of the *tbb::task_scheduler_init* constructor and the total number of logical CPUs on the executing system.

The last structure to be created is the *Arena* associated with calling MT. An arena encapsulates all the tasks and the execution resources (worker threads) available to a MT. Each arena is assigned a number of slots representing the number of workers that arena requires to complete its parallel tasks. This is defined as 1 less than the minimum of the argument of the *tbb::task_scheduler_init* constructor and the total number of workers available (limit set by the market). Because several MTs can coexist, the total number of workers requested by all arenas can be greater than the number of workers available in the RML's pool. In this situation, the market will allot workers proportionally to each MT's request.

All these components and limits are created once, during the first instance of the *tbb::task_scheduler_init* object in the current execution. If an MT is not the first one to call the task scheduler, it will create a new arena that will comply with the limitation imposed by the market. Upon creation or destruction of an arena, the worker threads can migrate between the active arenas.

After they are created, each worker thread runs a scheduling procedure called *wait_for_all()* consisting of 3 nested loops. The inner loop executes the current task by calling its *execute()* method. TBB is a continuation-passing style library which means that the completion of this task returns a pointer to the next task that needs to be executed. If a new task is not referenced, the inner loop exits. In the middle loop the *get_task()* method tries to dequeue the local task queue in a LIFO order. If successful, the inner loop is called again. If unsuccessful

because the queue is empty, the middle loop exits and the outer loop invokes the stealing mechanism by calling the *receive_or_steal_task()* method.

3. The stealing mechanism

3.1. The TBB implementation

The *receive_or_steal_task()* method is part of the outer loop in the scheduling procedure and it looks for all work available at this level. This includes: tasks mailed via the task-to-thread affinity mechanism, reload offloaded non-priority tasks or reload tasks abandoned by other workers. If none of these calls return a task to execute, a steal is attempted from a randomly selected victim thread in the current arena. If the attempt is successful, the method returns and the scheduler re-enters the inner loop of the scheduling procedure. If unsuccessful, a failure counter is incremented and the execution pauses before looping back to the beginning of *receive_or_steal_task()* method. Also, if the failure counter surpasses a given threshold (default value is 100) and the arena is still empty, the current worker thread is freed and returns to the RML.

When attempting a steal, the thief must first get a lock on the victim's queue using the *lock_task_pool()* method. If that fails, the thief goes through a 5 step exponential backoff. After 5 fails, the current thread yields its resources and waits for its next time slot to try to lock the same victim again. This locking mechanism assures the high composability of TBB we discussed in Section 2. However, the most common situation is when only one thread is running on each hardware core, making the yielding function return immediately. This means that the thief thread will continue trying to lock its victim. In our experiments, we match the simulated number of threads to the simulated number of cores which makes us face this locking issue.

The most common situation for stealing failure is due to selecting a victim with an empty task queue. Applications with an unbalanced workload distribution face this problem often. The default random selection policy in TBB cannot prevent against this type of failures.

Race contention is also a common situation for failure. When two or more threads are trying to get exclusive access to the same task queue by calling the *lock_task_pool()*, only one can succeed. A thief can return from the *lock_task_pool()* only if it either succeeds or the victim's task queue has been depleted. This means that the thread who did not acquire the lock will wait around until that lock is freed or until the victim queue has been emptied.

A special situation is when a thief thread is competing for access with the owner thread of that task queue. If there is more than one task in the queue, there is no race contention because the thief will steal at one end while the owner will dequeue the other. However, if there is only one task in the queue, the owner thread will have priority and the thief will backoff.

3.2. The oracle selection scheme

In an attempt to see how much performance can be improved by tuning the victim selection, we introduced an "all knowing" scheme we call the oracle selection [7]. This method leverages on the fact that we use a simulator and not a real machine. Thus, we can provide TBB with information that would be otherwise very "expensive" to obtain. Outside the simulated memory space, we created a data structure that stores the occupancy of each task queue in the arena as well as their level of congestion (the number of workers trying to steal from each queue). This structure is updated by the application through specialized instructions called markers that only our simulator recognizes and executes. Since we do all this computation outside the simulated environment, our TBB application sees the victim selection as an extremely fast, zero-overhead procedure. The scheme selects as victim the queue with some available tasks for stealing and with the lowest congestion level. Even though this oracle scheme provides very fast and accurate results, it is not perfect. For our simulator there are still a few situations when updates to our structure do not propagate fast enough and the selected victim ends up creating conflicts.

3.3. The pseudo-random selection scheme

Our second selection method is a pseudo-random scheme inspired by the Wool library [3]. This policy was also introduced in [7], but for this paper we improved its implementation and tuned its performance. For the first stealing attempt, we randomly select a task queue. If stealing from this victim fails, we then start a loop and sequentially scan the other active task queues, excluding the one of the current thread. In this way we will first try to steal from all possible queues before looping back in the *receive_or_steal_task()* and selecting a new random victim. There are two major benefits to this approach. First, all the stealing attempts during the sequential scan are very cheap in terms of number of instructions, reducing the overheads. Second, we can conclude much earlier than the TBB implementation that an arena is out of work and we can put a worker thread to sleep sooner. To tune our implementation even further, we removed the call to the yielding function from the *lock_task_pool()*. This forces the method to return after the 5 steps exponential backoff and eliminates the conflicts caused by the immediate return of the yielding function. However, this makes the stealing mechanism a bit more aggressive since it allows it to select new victims faster.

3.4. The occupancy-aware selection scheme

This method is inspired by the oracle scheme and tries to find the task queue with the most work available to steal from. In contrast to the oracle scheme, we now select our victim based solely on the level of occupancy of the task queues. Also, in contrast to our "all knowing" policy, this scheme is implemented fully in the TBB library and can be used outside of our simulated environment. We use a 2-dimensional array to store the occupancy level of the queues, with each thread logging separately information about tasks that it spawned, tasks that it stole or tasks that it executed. In this way we eliminate the possibility of races on writing and the need for a locking mechanism. To increase selection speed, we also do the scanning of the array with no locks. All these ensure that this approach is fast enough to work with TBB. However it also means that a snapshot of

Table 1: Main characteristics of modeled processor

	Parameter	**Value**	
Core	Clock frequency	2.66 GHz	
	Instruction set	x86-64	
	Dispatch width	4	
	Window size	128	
	Core count	1,2,4,8,16,32 cores	
Cache		Size	Assoc.
	L1 iCache/dCache	#cores x 32KB	4/8
	L2 Cache	#cores x 256KB	8
	L3 Cache	2/4/8/16/32/64 MB	16
Main memory	Size	2/4/8/16/32/64 GB	

the occupancy array will not always be accurate. Since the congestion level of the task queues are not monitored (like the oracle policy does), a queue can be selected as victim by several thieves at the same time. To make sure the thieves will first deplete the tasks of this victim before attempting a new selection, we used the default TBB approach for the *lock_task_pool()* function. A thief will not return from this function unless it either acquired the lock or the task queue is empty. With this selection scheme, just like with the pseudo-random one, we can find out faster than the default TBB approach that an arena is out of work.

4. Methodology

4.1. Simulation tools

We performed our experiments using a parallel, x86 computer architecture simulator called Sniper [10]. Sniper uses the interval core model [11] and Graphite simulation infrastructure [12] to provide fast and accurate simulations. Our model is a Nehalem-based Xeon 5500-series multi-core CPU (code name Gainestown) with a clock frequency of 2.66 GHz and 3 levels of cache.

The simulations do not include an operating system and no mechanism for frequency and/or voltage control is used. Table 1 lists the main characteristics of the modeled processor.

The performance results from Sniper are fed into a power estimation tool called McPAT [13]. An important characteristic of McPAT is its ability to model dynamic, static and short-circuit power. Dynamic power refers to the power required by a circuit to switch from one logical state to the other. For each system component, dynamic power is defined as: $power_{dynamic} \sim AF \cdot C \cdot V_{dd}^2 \cdot f$, where AF is the activity factor, C is the total load capacitance, V_{dd} is the supply voltage and f is the clock frequency [14]. Switching circuits also dissipate short-circuit power which McPAT modeles analytically. Static power is caused by current leakage during periods of non-activity. McPAT estimates leakage current using models of real-world CMOS circuits.

A recent study shows that McPAT's area and power models can have significant errors [15]. The authors assess McPAT's estimations against measurements of an IBM POWER7 CMP. They note that read/write port overestimates caused by high issue width and modeling of simultaneous multithreading (SMT) are two of the major sources of error they observed. For this reason, our measurements are only marginally affected by these errors since our modeled CPU has a relative low issue width (4 compared to 8 for the POWER7) and no SMT enabled.

To account for both active and idle core time, we use the dynamic power and static power (totaling subthreshold and gate leakage) outputted by McPAT for each core. In estimating the energy footprint, we multiply these by the active runtime and the idle time respectively of the cores to get a measure of the energy they use. Adding them all together gives us the CPU energy usage.

4.2. Benchmarks

For our experiments, we used the default TBB implementations of *Blackscholes*, *Bodytrack*, *Fluidanimate*, *Streamcluster* and *Swaptions* benchmarks with the simlarge input set from the PARSEC suite [16]. All of them were built us-

ing the 4.1.1 version of TBB. Collectively, these benchmarks express parallelism both explicitly as well as implicitly and employ some special TBB constructs like cache affinity partitioners and cache allocators. They provide a wide test base for our study.

Blackscholes uses the Black-Scholes partial differential equation to analytically calculate the prices for a portfolio of European options. The differential equation is implemented numerically and *parallel_for* templates are employed to divide the work among worker threads. In order to improve cache affinity, a TBB affinity partitioner is used.

Bodytrack is a computer vision application which tracks a human body with multiple cameras. It uses pipeline parallelism and *parallel_for* templates to express parallelism.

Fluidanimate simulates an incompressible fluid for interactive animation purposes. It uses an extension of the Smoothed Particle Hydrodynamics method to describe the fluid. Parallelism is annotated explicitly through *spawn(task_list)*.

Streamcluster is a mining application that tries to solve the online clustering problem. Parallelism is annotated explicitly through *spawn(task_list)* as well as using *parallel_for* and *parallel_reduce* templates. TBB's cache allocators are also used to optimize access to shared data.

Swaptions uses the Heath-Jarrow-Morton framework to price a portfolio of swaptions. Price computation is achieved through the Monte Carlo simulation. Swaptions uses *parallel_for* templates and cache allocators to express parallelism and optimize access to shared data.

Our experiments are meant to study the impact of the victim selection policy on the overall performance of the parallel execution. To that end, we want to minimize all possible interference on our test policies and quantify their impact as accurately as possible. To eliminate context switching on the simulated cores, we always match their number with the number of parallel threads. Also, we simulated only one benchmark at a time. It will be very difficult (if not impossible) to account for the effects of thread interleaving when two or more applications are executed at the same time.

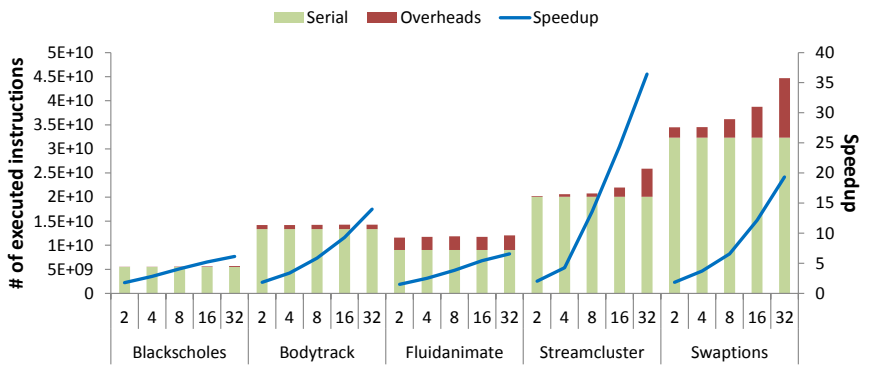

Figure 2: Overheads and speedups for the default random selection policy

To account for the non-deterministic simulation of Sniper, we performed 10 simulations of each benchmark for every core count. We averaged the performance results (μ) and used these to estimate the power requirements. We also computed the standard deviation (σ) of the execution time for each of the 10 simulation set. In none of our experiments we found any outliers, where an outlier is a value beyond $3\sigma \pm \mu$. For Blackscholes, Bodytrack, Fluidanimate and Streamcluster, our results show a σ/μ in the 0.012% - 1.83% range. Swaptions, due to its use of the Monte Carlo simulation has a higher variability between simulation, with σ/μ in the 1.18% - 14.73% range.

5. Results

As described by Amdahl's law, the maximum expected speedup of parallelization is limited by the sequential fraction of the program. When managing overheads are taken into consideration, this theoretical maximum becomes even harder to achieve. As we showed in our previous study, these overheads become larger as we scale the core count [7]. Even though with parallel executions the work gets done faster, the energy required to complete it is often equal or greater than the sequential execution.

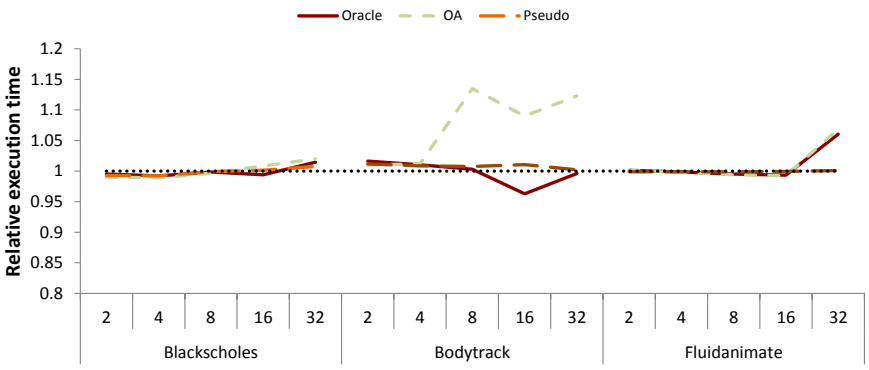

Figure 3: Total execution times relative to the random selection policy

5.1. Parallelization overheads

As mentioned in Section 2, each worker thread runs a scheduling procedure containing an infinite nested loop. This loop tries to execute tasks from its own queue or to obtain some work through the *receive_or_steal_task()* method. By default, the *receive_or_steal_task()* method loops a maximum of 100 times in an attempt to obtain a task before reporting that the arena is empty and returning the thread to RML. This means that each time a steal fails, the *receive_or_steal_task()* will loop to the beginning adding overheads and delay to the execution.

A very simple way to see what trend parallelization overheads form as you scale up the number of cores is to look at the execution statistics reported by TBB. There you can see how many times each parallel thread successfully stole a task, how many times it failed, how many times out of those fails was due to conflicts with other threads and many other. Looking at these statistics for the default TBB implementation, it becomes apparent that random victim selection policy is a serious bottleneck for high core counts. For applications with high numbers of parallel tasks like Swaptions, failed tasks range from an average of 40000 for 2-cores executions to almost 18 millions for 32-cores ones. That translates into 38.18% increase in instruction count when compared to the serial execution (see Figure 2). A detailed analysis of the results presented in

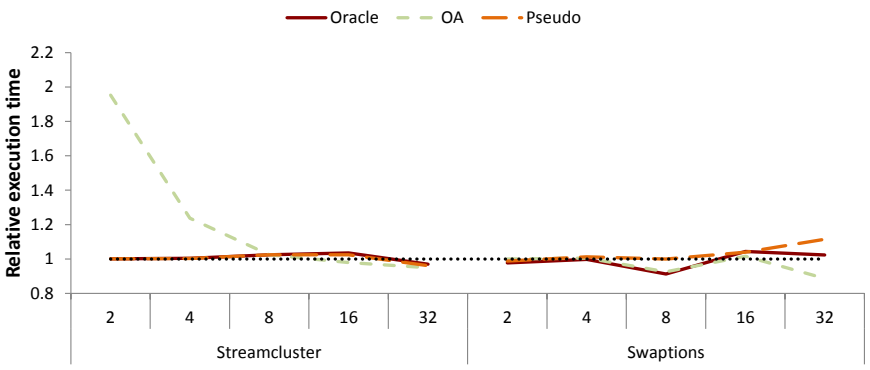

Figure 4: Total execution times relative to the random selection policy

Figure 2, including a breakdown of the overheads and a discussion on speedups, can be found in [7].

With the occupancy-aware selection scheme we wanted to reduce the overheads by removing all (or as many as possible) failed tasks caused by conflicts. Although we managed to do that, the overall overheads are generally higher than those of the random selection experiments. This is due to the fact that we scan the occupancy array for each steal attempt and this adds up fast. For all our low core counts (2 or 4) results there is not enough contention among threads in order to balance-out the added number of instructions of the scanning operation. In addition, some benchmarks like Blackscholes, Bodytrack and Fluidanimate have low numbers of total tasks to execute which again makes it hard to make up for the overhead of the scanning operation.

In the case of the pseudo-random selection policy, things are almost the opposite of occupancy-aware scheme: we generally have more failed steal attempts, but overall the overheads are lower. This is explained by the fact that the pseudo-random policy is far more aggressive in trying to find new work, but due to our sequential scanning implementation each attempt is cheaper. Also, the *receive_or_steal_task()* method returns much faster reducing the overheads even further.

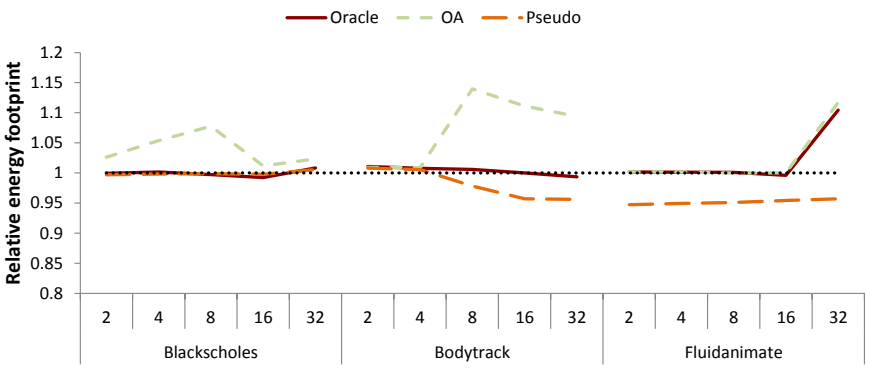

Figure 5: Energy footprint relative to the random selection policy

5.2. Victim selection policies - comparative study

The main issue faced by the random selection policy is its inability to scale. For high core counts or when we are dealing with very fine parallelism which forces the worker threads to steal often, random selection causes overheads to grow exponentially. We developed the occupancy-aware and the pseudo-random schemes to address this limitation, by adding some information gathering in the selection process. By doing this we increased the work the threads need to do, so the added performance has to pay for this as well. Because of its very simple nature, the random victim selection policy remains hard to outperform in situations when race contention among threads are rare (see Figure 3). Our occupancy-aware policy proves to be great in theory but difficult in practice. Our results show that it manages to significantly reduce the conflicts among threads. However, our implementation relies on scanning the occupancy array for each steal attempt which proves to be very costly. In addition, we implemented some guards against conflicts with the main thread which proved to have unexpected effects in some situations (see the 2-core results for Streamline in Figure 4). What becomes apparent when looking at the results in Figure 3 and 5 is that we can't always afford the added complexity. However, when there is enough congestion for this policy to make a difference, it can reduce execution time with up to 11.23% and the energy footprint with up to 7.83% (see Figure 4

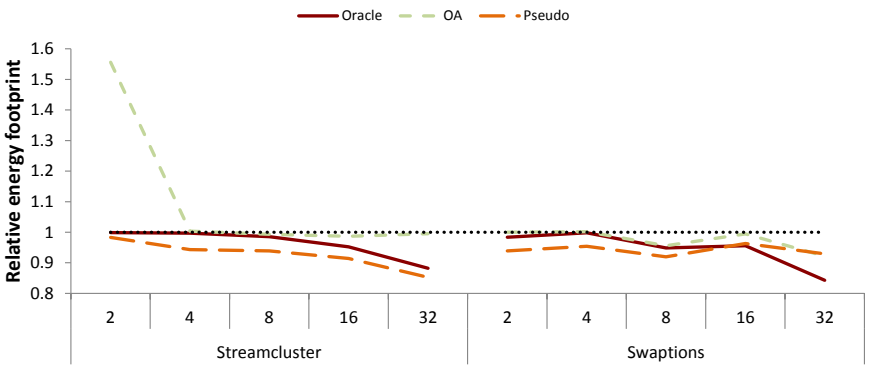

Figure 6: Energy footprint relative to the random selection policy

and 6).

The pseudo-random selection is much lighter in terms of extra-work compared to the occupancy-aware policy, but is also more aggressive. Our results with this scheme show that it can only be marginally faster than the default random selection, but it constantly does better in terms of energy-footprint (see Figure 3, 4, 5 and 6). This happens because with this policy it is very easy to identify the situations when there is no work to be done by the worker threads. By putting them to sleep sooner, we save energy. In this way it manages to reduce the energy footprint with up to 14.72%.

6. Related Work

The energy efficiency of parallel systems and the overheads parallelization brings have been the subject of many studies. Reducing the power requirements of multi-core CPUs, improving the energy efficiency of big parallel systems or reducing the overheads of parallel implementations have been explored by many researchers and plenty of solutions have been found.

Li and Martinez studied the power-performance implications of running parallel applications on CMPs [17]. Using both an analytical model and detailed simulations, the authors show that parallel computing can bring signif-

icant power savings through judiciously selections of the granularity and voltage/frequency levels.

Contreras and Martonosi study and characterize some of the overheads of Intel's TBB [18]. They concluded that task management operation can have a detrimental effect on the performance of parallel execution. The authors also note that random stealing fails to scale with increasing core counts and that alternative policies can improve performance.

Bhattacharjee and Martonosi propose a hardware thread criticality predictor which they build using already-accessible on-chip information like memory statistics [19]. The authors test this predictor in two different scenarios. First, they use it to assist TBB's task scheduler and show that task stealing can be improved over the original random approach. Second, they use the predictor to guide DVFS and to reduce dynamic energy in barrier-based applications. The authors conclude that the thread criticality predictor offers good accuracy at very low hardware overhead.

Podobas et al. do a performance comparative study of several parallelization libraries, including TBB [20]. They use both micro-benchmarks and a subset of the BOTS suite to characterize application performance and the costs for task creation and stealing. The study concludes that Wool has the lowest overhead for task spawning and task stealing. However, our previous study showed Wool to be far more aggressive when stealing than TBB which means that as we scale up the core number, Wool will perform worse [6].

The *direct task stack* is a TBP algorithm for extremely fine grained parallel applications [21]. Its implementation in the Wool library shows very low overheads for task creation and task stealing. The experimental results show that Wool significantly outperforms other implementations like Cilk++, TBB or OpenMP for extremely fine grained parallel applications (tens of cycles/task).

Vandierendonck et al. advocate the use of TBP models with nested task spawning for writing general-purpose programs [22]. The authors developed a Cilk-like language to express parallel pipelines and extended a Cilk-like scheduler to recognize and enforce argument dependency types on task spawns. This

programming model enhances the ease of programming parallel pipelines.

Chen et al. do a study to evaluate TBB's scalability against Pthreads implementations and to measure some of TBB's overheads [23]. Their results show possible bottlenecks that limit the scalability of TBB. They also show that TBB runtime overheads increase with core counts and in the current implementation will become the main performance bottleneck when scaling to tens of cores.

Ami Marowka introduces TBBench, a micro-benchmark suite designed for Intel's TBB [24]. TBBench is designed to measure the overheads associated with *parallel_for* and *parallel_reduce* constructs and mutual exclusion mechanisms like *Mutex*, *Spin_mutex* and *Queuing_mutex*. The experimental results show that TBB's mutual exclusion mechanisms and scheduler exhibit less overheads than the equivalent OpenMP constructs.

7. Conclusion

Intel's TBB is a runtime library designed to encourage programmers to create portable, parallel applications using task parallelism. TBB was developed to dynamically scale on the existing resources, employing task stealing to deal with workload imbalance. However, as CPU's core counts are ever-increasing, TBB proves to have a performance bottleneck in its use of a random victim selection policy.

Continuing our previous study [7], we propose two alternatives for the victim selection process. Based on the "all knowing" oracle scheme, we developed an occupancy-aware policy to reduce the number of failed steals. However, our implementation proved to be too complex and in many situation we recorded an overall increase in overheads. Nevertheless, for applications with very high thread contention, this scheme proved to be very beneficial, reducing the execution time and the energy footprint with up to 11.23% and 7.83% respectively. We think that our implementation can be improved and we will pursue this in future work.

The pseudo-random victim selection is the second policy we experimented

with. The implementation in this paper is a refinement of the one in [7] and it showed better energy footprints across the board when compared to the default TBB scheme. Even though it copes better in situations with many races between threads than the random one, the pseudo-random selection's performance is still affected in such scenarios.

With this work we showed that TBB can be improved for both performance and energy efficiency, even though not always at the same time. The results of our occupancy-aware scheme can be improved and we plan to do this in future work. Also, seeing how the pseudo-random approach performs well under low core counts, we are also considering a combined selection policy. The idea is to use each scheme for the core counts that they perform best. Based on our experiments so far, for core counts of 2 to 8 pseudo-random could be used and occupancy-aware for anything above. However, a more extensive testing needs to be done on a larger number of benchmarks before confirming this threshold.

References

[1] C. Leiserson, The Cilk++ concurrency platform, The Journal of Supercomputing 51 (3) (2010) 244–257. doi:10.1007/s11227-010-0405-3.
URL http://dx.doi.org/10.1007/s11227-010-0405-3

[2] D. Leijen, W. Schulte, S. Burckhardt, The Design of a Task Parallel Library, in: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09, ACM, New York, NY, USA, 2009, pp. 227–242. doi:10.1145/1640089.1640106.
URL http://doi.acm.org/10.1145/1640089.1640106

[3] K.-F. Faxén, Wool - A Work Stealing Library, SIGARCH Comput. Archit. News 36 (5) (2009) 93–100. doi:10.1145/1556444.1556457.
URL http://doi.acm.org/10.1145/1556444.1556457

[4] C. Pheatt, Intel Threading Building Blocks, J. Comput. Sci. Coll. 23 (4)

(2008) 298–298.
URL http://dl.acm.org/citation.cfm?id=1352079.1352134

[5] S. Fuller, L. Millett, Computing Performance: Game Over or Next Level?, Computer 44 (1) (2011) 31–38. doi:http://doi.ieeecomputersociety.org/10.1109/MC.2011.15.

[6] A. C. Iordan, M. Jahre, L. Natvig, On the Energy Footprint of Task Based Parallel Applications, in: High Performance Computing and Simulation (HPCS), 2013 International Conference on, 2013, pp. 164–171. doi:10.1109/HPCSim.2013.6641409.

[7] A. C. Iordan, M. Jahre, L. Natvig, Victim Selection Policies for Intel TBB: Overheads and Energy Footprint, in: E. Maehle, K. Rmer, W. Karl, E. Tovar (Eds.), Architecture of Computing Systems - ARCS 2014, Vol. 8350 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 13–24. doi:10.1007/978-3-319-04891-8_2.
URL http://dx.doi.org/10.1007/978-3-319-04891-8_2

[8] D. Patterson, The Trouble With Multicore, IEEE Spectrum 47 (7) (2010) 28–32, 53. doi:10.1109/MSPEC.2010.5491011.

[9] H. Pan, B. Hindman, K. Asanović, Composing Parallel Software Efficiently with Lithe, in: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, ACM, New York, NY, USA, 2010, pp. 376–387. doi:10.1145/1806596.1806639.
URL http://doi.acm.org/10.1145/1806596.1806639

[10] T. E. Carlson, W. Heirman, L. Eeckhout, Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, ACM, New York, NY, USA, 2011, pp. 52:1–52:12. doi:10.1145/2063384.2063454.
URL http://doi.acm.org/10.1145/2063384.2063454

[11] D. Genbrugge, S. Eyerman, L. Eeckhout, Interval simulation: Raising the level of abstraction in architectural simulation, in: High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on, 2010, pp. 1–12. doi:10.1109/HPCA.2010.5416636.

[12] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, A. Agarwal, Graphite: A distributed parallel simulator for multicores, in: High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on, 2010, pp. 1–12. doi:10.1109/HPCA.2010.5416635.

[13] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, N. P. Jouppi, McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures, in: Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, ACM, New York, NY, USA, 2009, pp. 469–480. doi:10.1145/1669112.1669172.
URL http://doi.acm.org/10.1145/1669112.1669172

[14] S. Kaxiras, M. Martonosi, Computer Architecture Techniques for Power-Efficiency, 1st Edition, Morgan and Claypool Publishers, 2008.

[15] S. L. Xi, H. Jacobson, P. Bose, G.-Y. Wei, D. Brooks, Quantifying sources of error in McPAT and potential impacts on architectural studies, in: High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on, 2015, pp. 577–589. doi:10.1109/HPCA.2015.7056064.

[16] C. Bienia, Benchmarking Modern Multiprocessors, Ph.D. thesis, Princeton, NJ, USA, aAI3445564 (2011).

[17] J. Li, J. F. Martínez, Power-performance Considerations of Parallel Computing on Chip Multiprocessors, ACM Trans. Archit. Code Optim. 2 (4) (2005) 397–422. doi:10.1145/1113841.1113844.
URL http://doi.acm.org/10.1145/1113841.1113844

[18] G. Contreras, M. Martonosi, Characterizing and improving the performance of Intel Threading Building Blocks, in: Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on, 2008, pp. 57–66. doi:10.1109/IISWC.2008.4636091.

[19] A. Bhattacharjee, M. Martonosi, Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors, in: Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA '09, ACM, New York, NY, USA, 2009, pp. 290–301. doi:10.1145/1555754.1555792.
URL http://doi.acm.org/10.1145/1555754.1555792

[20] A. Podobas, M. Brorsson, K.-F. Faxén, A Comparison of Some Recent Task-based Parallel Programming Models, in: Third Workshop on Programmability Issues for Multi-Core Computers, 2009.

[21] K.-F. Faxén, Efficient Work Stealing for Fine Grained Parallelism, in: Parallel Processing (ICPP), 2010 39th International Conference on, 2010, pp. 313–322. doi:10.1109/ICPP.2010.39.

[22] H. Vandierendonck, P. Pratikakis, D. S. Nikolopoulos, Parallel Programming of General-purpose Programs Using Task-based Programming Models, in: Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism, HotPar'11, USENIX Association, Berkeley, CA, USA, 2011, pp. 13–13.
URL http://dl.acm.org/citation.cfm?id=2001252.2001265

[23] X. Chen, W. Chen, J. Li, Z. Zheng, L. Shen, Z. Wang, Characterizing Fine-Grain Parallelism on Modern Multicore Platform, in: Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on, 2011, pp. 941–946. doi:10.1109/ICPADS.2011.41.

[24] A. Marowka, TBBench: A Micro-Benchmark Suite for Intel Threading Building Blocks., JIPS 8 (2) (2012) 331–346.

URL http://dblp.uni-trier.de/db/journals/jips/jips8.html#Marowka12