Comparison of Three Popular Parallel Programming Models on the Intel Xeon Phi

Ashkan Tousimojarad and Wim Vanderbauwhede

School of Computing Science
University of Glasgow, Glasgow, UK

Abstract. Systems with large numbers of cores have become commonplace. Accordingly, applications are shifting towards increased parallelism. In a general-purpose system, applications residing in the system compete for shared resources. Thread and task scheduling in such a multithreaded multiprogramming environment is a significant challenge. In this study, we have chosen the Intel Xeon Phi system as a modern platform to explore how popular parallel programming models, namely OpenMP, Intel Cilk Plus and Intel TBB (Threading Building Blocks) scale on manycore architectures. We have used three benchmarks with different features which exercise different aspects of the system performance. Moreover, a multiprogramming scenario is used to compare the behaviours of these models when all three applications reside in the system. Our initial results show that it is to some extent possible to infer multiprogramming performance from single-program cases.

1 Introduction

There are various programming models and runtime libraries that help developers to move from sequential to parallel programming. In this paper, we have chosen three well-known parallel programming approaches to compare their performance on a modern manycore machine. Before going into the details of these models, we would like to introduce the manycore platform chosen for this study:

1.1 Intel Xeon Phi

The Intel Xeon Phi 5110P coprocessor is an SMP (Symmetric Multiprocessor) on-a-chip which is connected to a host Xeon processor via the PCI Express bus interface. The Intel Many Integrated Core (MIC) architecture used by the Intel Xeon Phi coprocessors gives developers the advantage of using standard, existing programming tools and methods. Our Xeon Phi comprises of 60 cores connected by a bidirectional ring interconnect. The Xeon Phi has eight memory controllers supporting 2 GDDR5 memory channels each. The clock speed of the cores is 1.053GHz. According to Jeffers [6], the Xeon Phi provides four hardware threads sharing the same physical core and its cache subsystem in order to hide the latency inherent in in-order execution. As a result, the use of at least two threads per core is almost always beneficial.

Each core has an associated 512KB L2 cache. Data and instruction L1 caches of 32KB are also integrated on each core. Another important feature of the Xeon Phi is that each core includes a SIMD 512-bit wide VPU (Vector Processing Unit). The VPU can be used to process 16 single-precision or 8 double-precision elements per clock cycle. The third benchmark (Sect. 3.3) utilises the VPUs.

1.2 Parallel Programming Models

In order to have a fair comparison, we have chosen three programming models that are all supported by ICC (Intel C/C++ Compiler).

OpenMP. OpenMP, which is the de-facto standard for shared-memory programming, provides an API using the fork-join model. Threads communicate by sharing variables. OpenMP has been historically used for loop-level and regular parallelism through its compiler directives. Since the release of OpenMP 3.0, it also supports task parallelism. Whenever a thread encounters a `task` construct, a new explicit task is generated. An explicit task may be executed in parallel with other tasks by any thread in the current team, and the execution can be immediate or deferred until later [1].

Intel Cilk Plus. Intel Cilk Plus is an extension to C/C++ based on Cilk++[8]. It provides language constructs for both task and data parallelism. Is has become popular because of its simplicity and higher level of abstraction (compared to frameworks such as OpenMP or Intel TBB). Cilk provides the `_cilk_spawn` and `_cilk_sync` keywords to spawn and synchronise tasks; `_cilk_for` loop is a parallel replacement for sequential loops in C/C++. The tasks are executed within a work-stealing framework. The scheduling policy provides load balance close to the optimal [10].

Intel TBB. Intel Threading Building Blocks (TBB) is another well-known approach for expressing parallelism [9]. Intel TBB is an object-oriented C++ runtime library that contains data structures and algorithms to be used in parallel programs. It abstracts the low-level thread interface. However, conversion of legacy code to TBB requires restructuring certain parts of the program to fit the TBB templates. Each worker thread in TBB has a deque (double-ended queue) of tasks. Newly spawned tasks are put at the back of the deque, and each worker thread takes the tasks from the back of its deque to exploit temporal locality. If there is no task in the local deque, the worker steals tasks from the front of the victims' deques [7].

2 Experimental Setup

All the parallel benchmarks are implemented as C++ programs. They are executed natively on the MIC. For that purpose, the executables are copied to the

Xeon Phi, and we connect to the device from the host using `ssh`. For the OpenMP applications, the `libiomp5.so` library is required. The `libcilkrts.so.5` is needed for Cilk Plus applications and the `libtbb.so.2` library is required for the TBB programs. The path to these libraries should be set before the execution, e.g. `export LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH`. The TBB programs should be compiled with the `-ltbb` flag. The OpenMP programs need `-openmp` flag. The Intel compiler `icpc (ICC) 14.0.2` is used with `-O2 -mmic -no-offload` flags for compiling the benchmarks for native execution on the Xeon Phi. All speedup ratios are computed against the running time of the sequential code implemented in C++.

3 Single-programming benchmarks

Three different benchmarks have been used for the purposes of this study. They are intentionally simple, because we want to be able to reason about the observed differences in performance between the selected models. We first compare the results for each single program.

3.1 Fibonacci

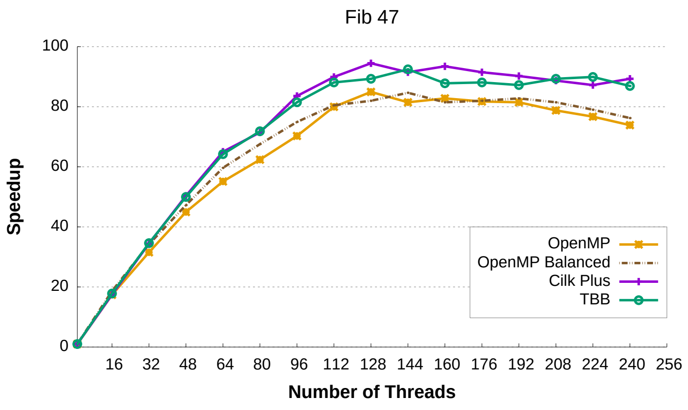
We consider a parallel Fibonacci benchmark as the first testcase. The Fibonacci benchmark has traditionally been used as a basic example of parallel computing. Although it is not an efficient way of computing Fibonacci numbers, the simple recursive pattern can easily be parallelised and is a good example of creating unbalanced tasks, resulting in load imbalance. In order to achieve desirable performance, a suitable cut-off for the recursion is crucial. Otherwise, too many fine-grained tasks would impose an unacceptable overhead to the system. The cutoff limits the tree_depth in the recursive algorithm, which results in generating 2^{tree_depth} tasks.

Figure 1 shows all the results taken from running this benchmark with different programming models. Figure 1(a) shows the speedup chart for the integer number 47 with 2048 unbalanced tasks at the last level of the Fibonacci heap. Cilk Plus and TBB show similar results. Increasing the number of threads causes visible performance degradation for OpenMP. Setting `KMP_AFFINITY=balanced` results in a negligible improvement of the OpenMP performance.

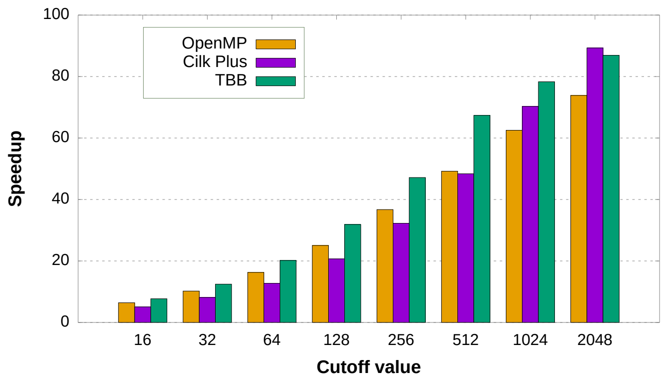
Figure 1(b) shows the importance of a proper cutoff on the performance of this unbalanced problem. Having more tasks (as long as they are not too fine-grained) gives enough opportunities for load balancing.

Total CPU Time

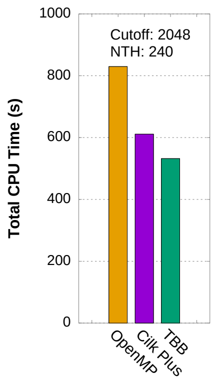
This is a lower-is-better metric that shows the total CPU times consumed in the system from the start until the accomplishment of the job(s). This metric and the detailed breakdown of CPU times are obtained using Intel's VTune Amplifier XE 2013 performance analyser [5]. Figures 1(d) to 1(f) are screenshots taken from the VTune Amplifier when running Fib 47 with cutoff 2048 natively on the



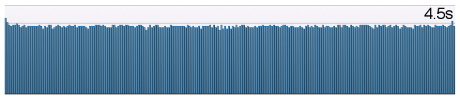
(a) Speedup, cutoff 2048, varying numbers of threads



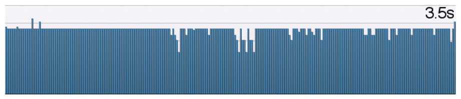
(b) Speedup, 240 threads, varying cutoffs



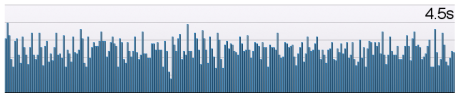
(c) Total CPU Time



(d) OpenMP, CPU balance



(e) Cilk Plus, CPU balance



(f) TBB, CPU balance

Fig. 1. Parallel Fibonacci benchmark for the integer number 47.
The best performance can be obtained by using Cilk Plus or TBB.
Choosing a proper cutoff value is key to good performance. If there are enough tasks in the system, the load balancing techniques become effective and yield better speedup.
A detailed breakdown of overall CPU time for the case with 240 threads and cutoff value 2048 is illustrated for each approach in the charts (d) to (f). TBB consumes less CPU time in total while providing good performance, and Cilk Plus has the best performance. The y-axis on the (d) to (f) charts is the time per logical core, from 0 to the maximum number specified in seconds.

Xeon Phi. The x-axis shows the logical cores of the Xeon Phi (240 cores), and the y-axis is the CPU time for each core.[1]

For the Fibonacci benchmark, OpenMP consumes the most CPU time, and its performance is bad, the worst amongst the three approaches.

3.2 MergeSort

This benchmark sorts an array of 80 million integers using a merge sort algorithm. The ith element of the array is initialised with the number i$*$((i%2)+2). The cutoff value determines the point after which the operation should be performed sequentially. For example, cutoff 2048 means that chunks of 1/2048 of the 80M array should be sorted sequentially, in parallel, and afterwards the results will be merged two by two, in parallel to produce the final sorted array.

For the MergeSort benchmark, tasks are not homogeneous, i.e. there are children and parent tasks. The same scenario existed in the previous Fibonacci benchmark, but the parent tasks were integer additions that did not impose overhead to the system. Here, the parent tasks are heavyweight merge operations, and this is what makes this benchmark distinct from the previous one.

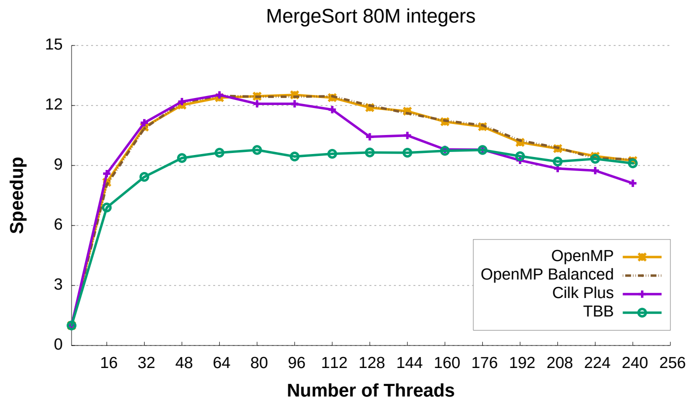
As shown in Fig. 2(a) with larger numbers of threads, there is either no noticeable change (in the case of TBB), or a slowdown (in the case of OpenMP and Cilk Plus). Using thread affinity for OpenMP in this case does not make an appreciable difference.

Figures 2(c) to 2(f) are again based on the results obtained by the VTune Amplifier when running the benchmark with 240 threads and cutoff 2048. Since all merges in a branch of the task tree can run on the same core as their children, there would be no need to have balanced load for good performance. In other words, the unbalanced distribution in Fig. 2(f) does not imply a poor behaviour of the TBB runtime system.

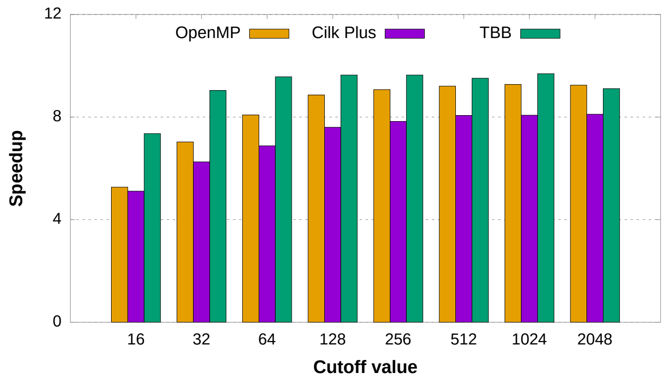
3.3 MatMul

This benchmark performs a naive matrix multiplication by a triple nested loop with ikj loop ordering for caching benefits on square matrices of N$\times$N double-precision floating point numbers. This is a completely data parallel problem which fits very well to OpenMP and its `for` worksharing construct. There is a concept similar to the cutoff in the loop parallelism context to control chunking. It specifies the size of chunk for each thread in a data parallel worksharing scenario. If the cutoff value is assumed as the number of chunks, the chunk (grain) size can be specified for the OpenMP `for` as follows: `#pragma omp for schedule(dynamic, N/cutoff)`. The `dynamic` keyword can be replaced by `static` as well. Grain size in the Cilk Plus is similarly specified via a pragma:

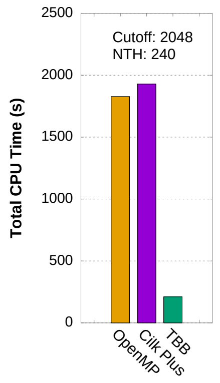
[1] It should be noted that for all experiments, results from the benchmark's kernel are considered in the figures (a) and (b), while in the other results taken from the VTune Amplifier, all information from the start of the application, including its initial phase and the CPU time consumed by the shared libraries is taken into account.



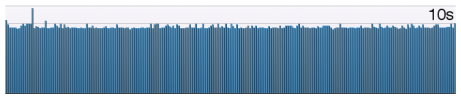
(a) Speedup, cutoff 2048, varying numbers of threads



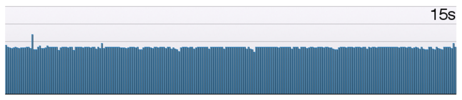
(b) Speedup, 240 threads, varying cutoffs



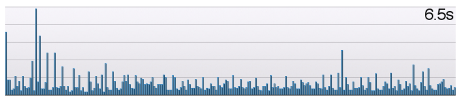
(c) Total CPU Time



(d) OpenMP, CPU balance



(e) Cilk Plus, CPU balance



(f) TBB, CPU balance

Fig. 2. Parallel MergeSort benchmark for an array of 80 million integers.
This benchmark does not scale well. The best performance, however, can be obtained by using OpenMP or Cilk Plus.
For this memory-intensive benchmark, cutoff values greater than 64 are enough to lead to good performance with as many threads as the number of cores.
TBB consumes significantly less Total CPU Time. With small number of threads, OpenMP and Cilk Plus yield better performance, but finally (with 240 threads) OpenMP and TBB provide slightly better performance.

`#pragma cilk grainsize = N/cutoff`. Intel TBB has a template function called `parallel_for`, which can be called with `simple_partitioner()` to control the grain size.

Before going into details of the results, we would like to focus on some technical considerations:

In order to achieve automatic vectorization on the Xeon Phi, the Intel TBB and OpenMP codes have to be compiled with the `-ansi-alias` flag.

The `schedule` clause used with OpenMP `for` specifies how iterations of the associated loops are divided (statically/dynamically) into contiguous chunks, and how these chunks are distributed amongst threads of the team. In order to have a better understanding of the relations between the cutoff value (number of the chunks), number of threads, and the thread affinity on the Xeon Phi, consider the following example. Suppose that for the MatMul benchmark, the OpenMP `for` construct with static schedule is used, which means that iterations are divided statically between the execution threads in a round-robin fashion:

Example

```
#pragma omp for schedule(static, N/cutoff).
```

Runtime of the case(a) on the Xeon Phi is $\approx 3\times$ better than that of the case(b).

a) `omp_set_num_threads(32)`, `cutoff=32`, `KMP_AFFINITY=balanced`
The threads will be spread across 32 physical cores. With the balanced affinity, they have to be distributed as evenly as possible across the chip, which is one thread per physical core. As a result, every chunk will be run on a separate physical core.

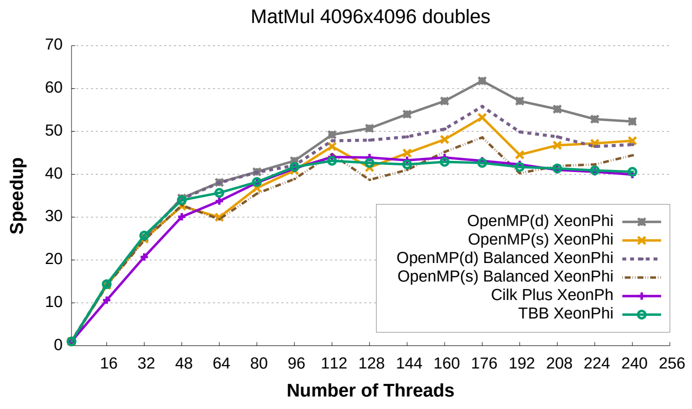
b) `omp_set_num_threads(240)`, `cutoff=32`, `KMP_AFFINITY=balanced`
The threads will be spread across all 60 physical cores. But the work will be distributed between 8 physical cores, which are the first 32 hardware threads. The reason is that with 240 threads, there will be one thread per logical core, and with cutoff 32, every thread with the thread id from 0 to 31 gets a chunk of size N/32.

With these considerations, we are ready to run the MatMul benchmark and compare the programming models in a data parallel scenario. The results can be found in Fig 3.

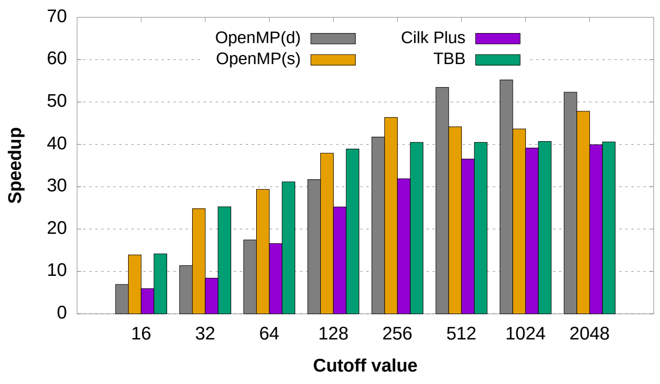
4 Discussion

One way to reason about the differences between these parallel programming models is to compare the amount of the Total CPU Time consumed by their runtime libraries. We have therefore summarised the results as the percentage of time spent on the shared libraries in each case.

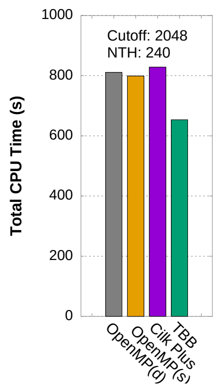
Table 1 gives a better understanding of where the CPU times have been consumed. For instance, for the OpenMP runtime library, the wasted CPU time generally falls into two categories: I) A master thread is executing a serial region,



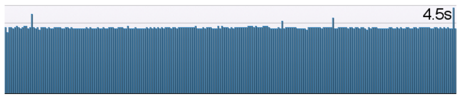
(a) Speedup, cutoff 2048, varying numbers of threads



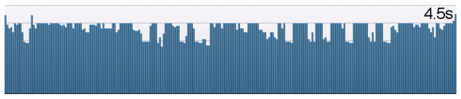
(b) Speedup, 240 threads, varying cutoffs



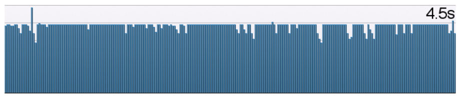
(c) Total CPU Time



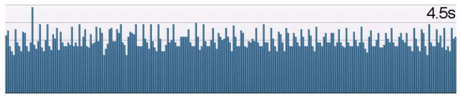
(d) OpenMP (dynamic), CPU balance



(e) OpenMP (static), CPU balance



(f) Cilk Plus, CPU balance



(g) TBB, CPU balance

Fig. 3. Parallel MatMul benchmark on a 4096×4096 matrix of double numbers.
The best results can be obtained by using OpenMP approaches.
For the cutoff values greater than 256, OpenMP with dynamic scheduling has the best scaling amongst all.
Again the Total CPU Time of TBB is the least amongst all. There is an evident distinction between the distribution of CPU times in the charts (d) and (e) that shows how OpenMP load balancing, when using dynamic scheduling leads to better performance.

Table 1. Percentage of the Total CPU Time consumed by the runtime libraries

Benchmark	**OpenMP** (`libiomp5.so`)	**Cilk Plus** (`libcilkrts.so.5`)	**TBB** (`libtbb.so.2`)
Fibonacci	50%	16%	5%
MergeSort	78%	81%	3%
MatMul	22% *(Dynamic)* 20% *(Static)*	6%	1%

and the slave threads are spinning. II) A thread has finished a parallel region, and is spinning in the barrier waiting for all other threads to reach that synchronisation point. Although sometimes in solo execution of the programs, these extra CPU cycles have negligible influence on the running time (wall time), we will show in the next section, how they will affect other programs under multiprogrammed execution.

5 Multiprogramming

In this section, we consider a multiprogramming scenario to see how these models behave in a multiprogramming environment. The metric used for the comparison is the user-oriented metric Turnaround Time [3], which is the time between submitting a job and its completion in a multiprogram system.

The three benchmarks have the same input sizes as the single-program cases with the cutoff value 2048 and the default number of threads 240 (the same as the number of logical cores in the Xeon Phi). We do not start all of them at the same time. Rather, we want the parallel phases to start almost simultaneously, such that all of the applications' threads compete for the resources. For that purpose, the MergeSort benchmark enters the system first. Two seconds later the MatMul benchmark enters the system, and half a second after that, the Fib benchmark starts[2].

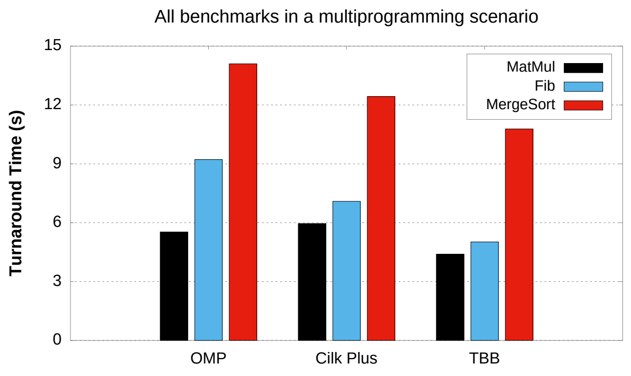
Based on the single-program results, we expect TBB to perform best because it has the least Total CPU Time in all three benchmarks. It might not affect the runtime of a single program significantly, but when there are multiple programs competing for the resources, the wasted CPU time can play an important role. In other words, CPU time wasted by each program can influence the performance of other programs reside in the system.

The results are shown and discussed in Fig. 4

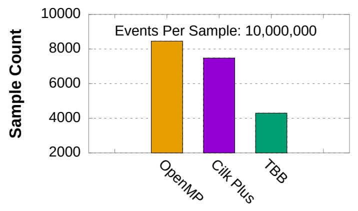
5.1 Related Work

Saule and Catalyurek [10] have compared the same three programming models on the Intel Xeon Phi. They have focused on the scalability of graph algorithms,

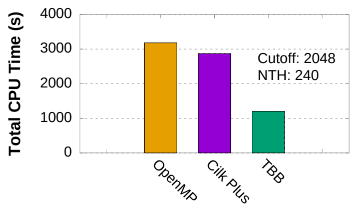
[2] The sequential phase of the MergeSort benchmark with the input size 80 million is around 2 seconds, and the initial phase of the MatMul benchmark with the input size 4096×4096 is about half a second.



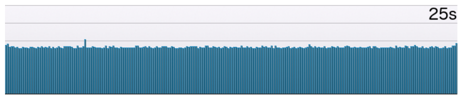
(a) Turnaround times



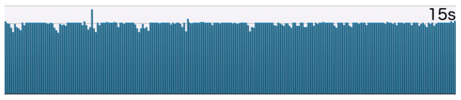
(b) Instructions Executed



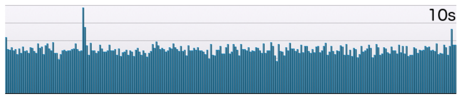
(c) Total CPU Time



(d) OpenMP, CPU balance



(e) Cilk Plus, CPU balance



(f) TBB, CPU balance

Fig. 4. A multiprogramming scenario with the three benchmarks
This is what happens when the three benchmarks compete for the resources: (a) shows that the best turnaround times are obtained with TBB. The hardware event, number of Instructions Executed, sampled by the VTune Amplifier in (b), implies a significant difference between TBB and the other two competitors. Results from the Total CPU Time in chart (c) is similar to those in chart (b) and they both show why TBB performs better than OpenMP and Cilk Plus. A detailed breakdown of overall CPU time in the (d) to (f) charts illustrates how OpenMP consumes more CPU time in total, and therefore has the worst performance.

while we have highlighted more differences between these programming models by adding the Total CPU Time as another performance aspect, and targeted the case of multiprogramming as well.

We have shown that the overhead of the runtime libraries play an important role in the parallel computing world, particularly in multiprogrammed systems. Besides the extra energy dissipation they impose on the system, they have noticeable influence on the performance of multiprogram workloads.

Emani et al. in [2] have used predictive modelling techniques to determine an optimal mapping of a program in the presence of external workload. Harris et al. have introduced Callisto [4] as a user-mode shared library for co-scheduling multiple parallel runtime systems (mainly for OpenMP programs). However, their current version does not support OpenMP tasks.

Varisteas et al. [14] have proposed an adaptive space-sharing scheduler for the Barrelfish operating system to overcome the resource contention between multiple applications running simultaneously in a multiprogrammed system.

In [12], a thread mapping method based on the system's load information is developed for OpenMP programs. Performance of the multiprogram workloads in Linux can be improved by sharing the load information and using it for thread placement. However, for this method to be effective, the optimal number of threads for each single program should be known to the programmer. Most of time, though, programs are run with the default number of threads, similar to what we did in this work.

We are currently developing a methodology inside our research framework, called Glasgow Parallel Reduction Machine (GPRM) [11] which allows the applications to use default numbers of threads (i.e. as many as the number of cores), and the same time improves the turnaround time by sharing some information globally. The main focus of GPRM is on tasks rather than threads to decrease the overhead of the runtime system. We have shown its potential, particularly in comparison with OpenMP [13]. We plan to add GPRM to the comparison with these three programming models. We aim to show that having a low-overhead runtime system is crucial in multiprogrammed systems.

6 Conclusion

We have compared some of the performance aspects (in particular speed-up, CPU balance, and the Total CPU Time) of three well-known parallel programming approaches, OpenMP, Cilk and TBB, on the Xeon Phi coprocessor. We used three different parallel benchmarks, Fibonacci, Merge Sort and Matrix Multiplication. Each benchmark has different characteristics which highlight some pros and cons of the studied approaches. Our multiprogramming scenario is to run all three benchmarks together on the system and observe how the different programming models react to this situation.

Based on the results obtained from the single program scenarios, particularly the Total CPU Time, we predicted that the Intel TBB approach would be more suited to a multiprogramming environment, and our experiment confirmed this.

Based on our learnings from these preliminary experiments, we plan to extend the work with more testbenches as well as more programming models.

In addition, since the way Linux deals with multithreaded multiprogramming is sub-optimal, we conclude that there is a need to share additional information on thread placement between the applications present in the system in order to get better performance. We are currently developing this idea inside our novel experimental framework.

References

1. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The design of openmp tasks. Parallel and Distributed Systems, IEEE Transactions on 20(3), 404–418 (2009)
2. Emani, M.K., Wang, Z., O'Boyle, M.F.: Smart, adaptive mapping of parallelism in the presence of external workload. In: Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on. pp. 1–10. IEEE (2013)
3. Eyerman, S., Eeckhout, L.: System-level performance metrics for multiprogram workloads. Micro, IEEE 28(3), 42–53 (2008)
4. Harris, T., Maas, M., Marathe, V.J.: Callisto: co-scheduling parallel runtime systems. In: Proceedings of the Ninth European Conference on Computer Systems. p. 24. ACM (2014)
5. Intel: Software development tools: Intel Vtune Amplifier XE 2013 (2013), `https://software.intel.com/en-us/intel-vtune-amplifier-xe`
6. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High Performance Programming. Newnes (2013)
7. Kim, W., Voss, M.: Multicore desktop programming with intel threading building blocks. IEEE software 28(1), 23–31 (2011)
8. Leiserson, C.E.: The cilk++ concurrency platform. The Journal of Supercomputing 51(3), 244–257 (2010)
9. Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Inc. (2007)
10. Saule, E., Catalyurek, U.V.: An early evaluation of the scalability of graph algorithms on the intel mic architecture. In: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International. pp. 1629–1639. IEEE (2012)
11. Tousimojarad, A., Vanderbauwhede, W.: The Glasgow Parallel Reduction Machine: Programming shared-memory many-core systems using parallel task composition. EPTCS 137, 79–94 (2013)
12. Tousimojarad, A., Vanderbauwhede, W.: An efficient thread mapping strategy for multiprogramming on manycore processors. In: Parallel Computing: Accelerating Computational Science and Engineering (CSE), Advances in Parallel Computing, vol. 25, pp. 63–71. IOS Press (2014)
13. Tousimojarad, A., Vanderbauwhede, W.: A parallel task-based approach to linear algebra. In: Parallel and Distributed Computing (ISPDC), 2014 IEEE 13th International Symposium on. IEEE (2014)
14. Varisteas, G., Brorsson, M., Faxen, K.F.: Resource management for task-based parallel programs over a multi-kernel.: Bias: Barrelfish inter-core adaptive scheduling. In: Proceedings of the 2012 workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE12). pp. 32–36 (2012)