Hochleistungsrechner - Aktuelle Trends und Entwicklungen
Winter Term 2017/2018

Task-based Programming Models in HPC

Jakob Schneider
Ludwig-Maximilians Universität München

January 19, 2018

Abstract

As programming for large clusters becomes harder and harder, every tool that simplifies the life of an HPC programmer is valuable. Task-based programming (TBP) models are one such tool. This paper describes the basic mechanisms of TBP on shared memory systems and shows how these functionalities are extended to fit distributed memory systems. To this end, a few libraries on shared and distributed memory systems are investigated and their specific features highlighted. To conclude we try to evaluate TBP against other common programming models in terms of performance, efficiency, and scalability.

1 Introduction

With the increasing complexity in high performance computing hardware systems, the software side of things mirrors this intricacy. With threads inside of cores inside of proccesors and the addition of accelerators such as the Xeon Phi co-processor or GPGPUs, application development is getting much harder. Synchronizing all these components is a very complex task (Venkat Subramaniam [15] calls this the *"syncronize and suffer"* model), but thankfully there are advancements that could simplify an HPC programmer's job significantly. Task based programming (TBP) is such a tool.

In TBP, the developer creates tasks with dependencies and the scheduler arranges the task execution, data distribution and synchronization. Tasks can be suspended and resumed on the same or other CPUs, and load balancing is done via work stealing.
TBP can facilitate parallel programming, but it needs to be carefully evaluated, as there are scenarios, in which threads can perform better than tasks. One such scenario may be an application with a lot of blocking (mutexes, waiting for user input). If a task is blocking, the worker thread that this task is assigned to is not able to perform any work while waiting.
The rest of this paper is structured as follows: Section 2 first introduces general features of task based programming models and show specific examples in existing frameworks for shared and distributed memory systems. In Section 3, the different frameworks are evaluated against each other and classic programming models in terms of scalability, performance, and efficiency. Section 4 provides a conclusion about the presented techniques.

2 Task-Based Programming Models

The important features of TBP are the following:

- **Task Spawning:** The programmer can de-

clare parts of code as tasks and these tasks can be spawned, that is, put into a queue of a worker thread and then executed.

- **Dependency Declaration:** Tasks most likely have dependencies, for example results from previous tasks, or the current results are needed by subsequent tasks. These dependencies must be declared, so that the scheduler can arrange the order of execution correctly.
- **Suspension and Resumption of Tasks:** Tasks can be suspended at specific points, for example when waiting for data. Afterwards, the task can be resumed at this point, and can even be transfered to another worker thread for resumption.
- **Load Balancing:** The runtime system schedules the given tasks according to different strategies. If a worker thread has no work, it can steal tasks from other workers.

2.1 Shared Memory

First, we will look at programming models for systems with shared memory to understand what benefits come with task based programming, before looking into how the task model can be implemented in distributed memory systems.

2.1.1 OpenMP 3.0

Task parallelism for OpenMP was proposed in 2006 and made available with the release of OpenMP 3.0 in May 2008. Tasks in OpenMP are defined as follows:
A structured block, executed once for each time the associated task construct is encountered by a thread. The task has private memory associated with it that stays persistent during a single execution. The code in one instance of a task is executed sequentially. [4]
A task is declared by the preprocessor directive
`#pragma omp task [clause]`
followed by a code block. With the clauses the programmer can specify dependencies between tasks, for example stating that a task needs a variable from another task. Furthermore variables can be declared private (initialized inside the task), firstprivate (initialized outside before the task, but then private to each task using it) or shared.
The proposal specifies two important features of tasks:

- **Suspend / resume points:** Tasks can be suspended at these specified points, so that other tasks can be executed first. The suspended tasks will be resumed at the point of interruption. These points are specified with the preprocessor directive

  `#pragma omp taskyield`
- **Thread switching:** If a task is suspended at a suspend point, a different thread can pick it up and continue at the same point.

There are two different types of task barriers in OpenMP 3.0:

- **taskwait** The current task waits for every child task that is generated since the beginning of the current task.
- **taskgroup** The program waits for all tasks in the taskgroup to complete.

2.1.2 Intel Cilk Plus

Intel Cilk Plus adds simple language extensions to C and C++ to allow the programmer to use task and data parallelism. It adds three keywords (`cilk_for`, `cilk_spawn` and `cilk_reduce`), as well as so-called hyperobjects, that simplify shared states between tasks without race conditions or locks.
The keywords are pretty self-explanatory:

- `cilk_for` parallizes loops,
- `cilk_spawn` spawns a task,
- `cilk_sync` waits for all spawned tasks in function before the program continues.

The most common hyperopbject is a *reducer*. The following example computes $\sum_0^N foo(i)$ in parallel:

```
cilk::reducer_opadd<float> result(0);
cilk_for (int i = 0; i < N; i++)
    result += foo(i);
```

Each worker thread has a deque instead of a classic queue. It pushes and pops tasks to and from the tail (and works thus in a LIFO order), which tends to improve locality. Additionally, if a worker tries to steal work from another worker, it does so from the head of the other worker's deque. The tasks at the head of the deque are more likely to be larger and spawn a lot of child tasks, and thus may lead to less overhead created by stealing.
Intel Cilk Plus will be deprecated with the release of the Intel Software Development Tools in 2018 and remain in deprecation mode for two years. Intel recommends migrating to OpenMP or Intel Threading Builing Blocks.

2.1.3 Intel Threading Building Blocks

Intel Threading Building Blocks (or TBB) is a C++ template library that again allows the division of the program in tasks. In contrast to Cilk Plus, TBB offers no keywords, but algorithms (e.g. `parallel_for`), containers (e.g. `concurrent_vector`), constructs for memory allocation (e.g. `scalable_malloc`), synchronization (e.g. `mutex`) and atomic operations (e.g. `fetch_and_add`):
It also provides the Intel Flow Graph to express dependencies and data flow:

```
//create the graph
tbb::flow::graph g;

//create a node
tbb::flow::continue_node
  <tbb::flow::continue_msg>
  h (g, [](const continue_msg &)
    {std::cout << "Hello ";});


//create a second node
tbb::flow::continue_node
  <tbb::flow::continue_msg>
  w (g, [](const continue_msg &)
    {std::cout << "World!";});

//link the nodes
tbb::flow::make_edge(h,w);

//start the first node
h.try_put(continue_msg());

//wait for the whole graph to finish
g.wait_for_all();
```

These nodes fulfill the same purpose as tasks.
There is also a visual graph design tool, called Flow Graph Designer, which lets you create, arrange and connect nodes. The Flow Graph Designer then generates the C++ code and you just have to fill in the content of the nodes.

2.1.4 Task Parallel Library

The Task Parallel Library (TPL) is a library for .NET. It provides types and functions that facilitates exposing parallelism in a program. It makes heavy use of generics and delegates. Leijen et al. [13] offer an overview of the design and implementation of the TPL.
Similar to Cilk Plus, it introduces tools to parallelize loops (`Parallel.For()`) and create tasks or futures. It also introduces the `Interlocked` class, that provides function for thread-safe incrementing, decrementing or adding a value to a variable. Fork-join parallelism is realized through the `Parallel.Do()` method.
The task queues, which are double ended like in Cilk Plus, are mostly lock free. Only if there are not enough tasks in a queue for popping and possible work-stealing, the local thread takes a lock. The freedom from locks is bought by adding states (`init`, `running` and `done`) to the tasks, which are set with atomic compare-and-swap operations, to ensure that each task is only run once.

2.1.5 Ordered Read Write Locks

The "Ordered Read-Write Locks"(ORWL)-Model builds upon the concept of read-write locks. Clauss et al. [6] explain, that while read-write locks fulfill their purpose, the programmer often has to take further actions to prevent deadlocks or starvation. The solution proposed by the authors is a synchronization overlay coupled with queues for the locks.

Additionally, handles are introduced, by which the locks are accessed, so a task can request a lock and may continue other work until the lock is granted.

The synchronization overlay is an abstraction of the data dependencies between the tasks. It is generated and optimized at startup and ensures a lock free execution.

Gustedt et al. [9] enrich the ORWL model with an automated system to improve data locality in systems with distributed memory. It utilizes hwloc, a framework presented by Broquedis et al. [5], to obtain information on the hardware topology and generate an allocation strategy, that aims to reduce communication between the nodes and optimise the shared caches in the nodes.

2.1.6 StarPU-MPI

StarPU is a software tool that simplifies the creation of highly parallel applications for heterogeneous multicore architectures. It is presented by Augonnet et al. [3].

It features a runtime support library, which handles the scheduling of its tasks. StarPU tasks consist of a *codelet* (computational kernel for a worker, can be a CPU, CUDA or OpenCL device), a data set on which the codelet works and information on its data dependencies. Task dependencies in StarPU are deduced by data dependencies by default, but can also be specifically set by the programmer.

As the dependencies can be specified, so can the scheduling. The programmer can choose to implement her own algorithms or choose from various designs offered by StarPU. Available queue designs span FIFOs, priority FIFOs, stacks and deques, which can be adapted at will. Data transfers are automatically handled by the StarPU runtime system.

Augonnet et al. [2] augment StarPU with MPI, which allows its use on distributed systems, but shows, that the network poses a severe bottleneck, but that could change as newer CUDA drivers enable the direct communication between the GPU and and the network cards.

2.2 Distributed Memory

In this section, we will take a look at frameworks that are specifically designed for systems with distributed memory and use the task based programming model.

2.2.1 Charm++

Charm++ is a programming language based on C++ that provides high level abstractions to simplify the development of parallel programs. Programs written in Charm++ can be executed unchanged on systems with shared and distributed memory. Kale et al. [12] presents the design principles behind Charm++:

Portability: Charm++ lays a heavy focus on portability, and it achieves it through tasks, but they are called `Chares`. Chares have the ability to create other chares and communicate with them.

Latency Tolerance: Instead of blocking-receive-based communication, Charm++ uses message driven execution, meaning all computations are initiated by received messages. Futures are supported as well, and while some chare may need to wait on data, others can be executed.

Object Orientation: Charm++ consists of the following classes of objects:

- **Sequential Objects** give the programmer the choice of having non-parallelized parts of the program.
- **Concurrent Objects** are the chares.
- **Shared Objects** contain data of any type, and can be distributed data structures. There are different classes of shared objects for example read-only, accumulator objects or distributed tables.
- **Communicaton Objects** describe the message entities.

Several load balancing and memory management strategies are available: *random*, *share with neighbours*, *central manager/scheduler* and *prioritized task creation*.

Zheng et al. [17] introduce FTC-Charm++, which extends Charm++ with a fault tolerance protocol based on checkpoints. The reason why it is based on checkpoints rather than on logging mechanisms is explained by a look to the evolution of applications that need fault tolerance at all. In the past these applications were mission-critical programs that under no circumstances should fail. Efficiency loss or overhead imposed by the fault tolerance techniques were neglectable, as long as the application didn't crash.
In most of today's high performance computing, a strong focus is set on efficiency. A computation, that runs for hours or days, fails and has to be restarted again, costs a lot of time and with it energy. Logging techniques entail a lot of overhead on communication, which again makes the application less efficient.
Zheng et al set the following requirements for a fault tolerance system in high performance computing:

- No reliance on fault-free storage
- Low impact on fault-free run time
- Low recovery time after a crash
- High execution efficiency after a restart (and maybe a loss of a processor)

The proposed solution is based on *double in-memory checkpointing* or *double in-disk checkpointing*. The checkpoints are saved in the local memory (or on a local disk) of a process instead of a central storage location. Additionally, every processor gets a *buddy processor*, and both their checkpoints are saved in both their memories or disks. That way, the system is not dependent on fault-free storage. In-memory checkpointing, opposed to in-disk checkpointing, ensures the low impact on fault-free run time.
To maintain a high efficiency after a fault, the task based model of Charm++ plays an important role, as the work of lost processors can be dynamically load balanced onto the remaining ones. Figure 1 shows the performance of LeanMD (a molecular dynamics simulation program) after a crash with and without load balancing.

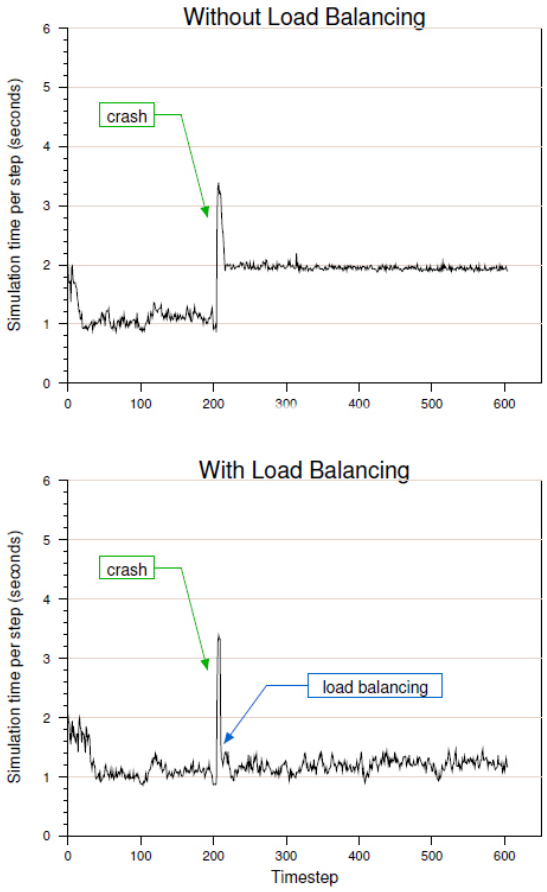


Figure 1: Performance with and without load balancing after a crash [16].

2.2.2 High Performance ParalleX

Kaiser et al. [11] introduce High Performance ParalleX (HPX) as a general purpose C++ parallel runtime system. The paper describes HPX as follows:

HPX represents an innovative mixture of a global system-wide address space, fine grain parallelism, and lightweight synchronization combined with implicit, work queue based, message driven computation, full semantic equivalence of local and remote execution, and explicit support for hardware accelerators through percolation.

Design Principles In this paper, four main factors are identified, that prevent scaling. They are appropriately named the **SLOW** factors: **S**tarvation, **L**atencies, **O**verheads and **W**aiting for contention resolution. To reduce these factors, the following design principles are established:

- **Latency Hiding instead of Latency Avoidance:** It is pointed out, that latency is tightly related to an operation waiting on a resource. Instead of focusing on minimizing this waiting time, it could be employed to do useful work.

- **Fine-grained Parallelism instead of Heavyweight Threads:** This is closely related to Latency Hiding, as even very small latencies can be used to do work, but only if context switching is fast enough to gain performance.

- **Avoid Global Barriers:** The paper mentions, that many common operations, such as loops parallelized with OpenMP, entail global barriers. But usually there is no need to wait on all iterations to finish before doing other work.

- **Adaptive Locality Control:** Static data distribution may lead to inhomogenous workloads across the system, therefore an dynamic data placement system is required. HPX implements a global, uniform adress space and a decision engine that is able to distribute data at runtime.

- **Move Work instead of Data:** As operations are commonly much smaller than the data required to perform them, the work should be moved to the data and not the other way around.

- **Avoid Message Passing:** When two threads communicate, they have to synchronize, in other words wait for each other, therefore message driven computation should be prefered.

Subsystems: HPX's architecture consists of five subsystems, each of which encapsulates a specific functionality.

- **The Parcel Subsystem:** The parcel subsystem handles network communication. Parcels are a form of active messages, which means that they are capable of performing processing on their own. In HPX, a parcel contains the global address of its destination object, a reference to one of the object's methods, the arguments for the method and optionally a continuation. The communicating entities are called *localities*. A locality transmits and receives parcels asynchronously.

- **The Active Global Address Space:** The AGAS constructs a global address space for all localities the application currently runs on. Opposed to partitioned global address space, the AGAS is dynamic and adaptive, it evolves over the lifetime of an application.

- **The Threading Subsystem:** When a locality receives a parcel, it is scheduled, run and recycled by the threading subsystem. Context switching between HPX threads does not require a kernel call, so that millions of threads can be executed per second on each core. There is usually one OS-thread per core, that the HPX threads are mapped onto. The threading subsystem uses work-stealing for intra-locality load balancing and threads are never interrupted by the scheduler, but can suspend themselves when waiting for data.

- **Local Control Objects:** LCOs handle parallelization and synchronization, they provide means of shared resource protection and execution flow organization. Examples for LCOs are `futures`, `dataflow objects`, `mutexes` and `barriers`.

- **Instrumentation and Adaptivity:** HPX implements ways to analyse performance measured by hardware, the OS, HPX runtime services and the application. It can use this data

to adapt the resource management system and therefore increase performance.

To determine if an optimal task size for specific applications exit on the HPX runtime system, Grubel et al. Grubel et al. [8] used an one-dimensional stencil operation, the heat distribution benchmark HPX-Stencil, with varying task sizes. In this benchmark, the calculations for each grid point are wrapped in futures. Additionally, the grid points are partitioned, and these partitions are represented as futures, too. This way, the number of calculations per future can be fine-tuned.
Measurements were taken by the previously mentioned instrumentation. Over the course of the experiments, the total number of grid points was always 100 million, but the size of the partitions ranged from 160 to 100 million points. Taking into account the idle-rate (computation to thread management ratio), thread management overhead and wait time per thread, a clear optimal range for the task size can be determined. Figure 2 shows the different measurements exemplarily on a Haswell 8 core CPU. Wait time in this figure can be negative, because of the way it is calculated: Wait time is defined as the difference between the measured average task duration and the task duration of the same experiment on one core. The time on one core can be higher, because of behaviours such as caching effects [8].
This knowledge can easily be used to statically and potentially even dynamically assign optimal task sizes.

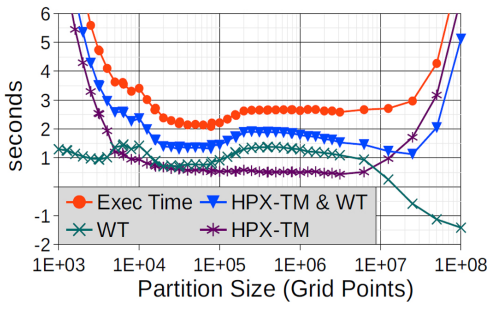


Figure 2: Execution Time (Exec Time), combined thread management and wait time (HPX-TM & WT), wait time(WT), and thread management (HPX-TM) on a Haswell 8 core CPU.

3 Evaluation

In this chapter we will look into several benchmarks on different kinds of systems to compare scalability, performance and efficiency.

3.1 Scalability

Scalability is very important for applications, otherwise the application needs to be adjusted to every system, or maybe even completely rewritten.

The first interesting benchmarking result is from the work of Kaiser et al. [11]. They used the Homogeneous-Timed-Task-Spawn benchmark on a Intel Ivybridge system with two sockets, and evaluated HPX against Qthreads and TBB. The result show a clear performance drop at the socket boundary for Qthreads and TBB (see Figure 3). 2.5 million tasks were user for each core, and these tasks did no work and returned immediately, simulating very fine grained tasks.

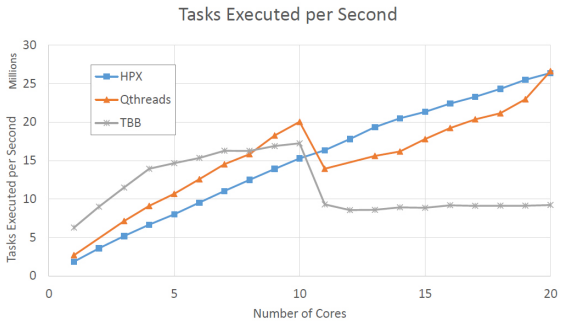


Figure 3: Tasks executed per second on a two-socket Intel Ivybridge system [11].

Another interesting benchmark was conducted by Bryce Adelstein-Lelbach from the Center of Computation and Technology (CTT) at Louisiana State

University. He spawned 500000 tasks on a HP DL785 G6 node with 48 cores on eight sockets[1]. The tasks did not perform any communication and had artificial workloads of 0, 100 and 1000 μs, which exposes the task overheads of each framework. Figure 4 shows, that TBB and HPX are almost tied when tasks are fine grained, but with growing task sizes, HPX is more stable in scaling than TBB.
Note that the links provided to the SWARM library no longer works and no information on this library could be found.

3.2 Performance

Kaiser et al. [11] show that HPX can maintain 98% of the theoretical peak performance of a system on a single node (16 cores) and 87% with 1024 nodes. It outperforms the MPI implementation by a factor of 1.4 (see Figure 5). On a Xeon Phi co-processor, HPX is able to reach 89% of the theoretical peak performance.

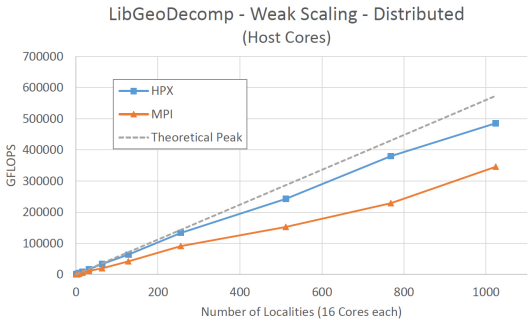


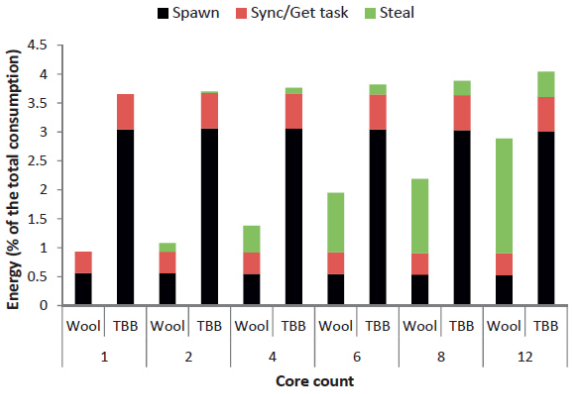
Figure 5: HPX vs MPI in terms of efficiency [11].

Interestingly, Perez et al. [14] mention, that an application using their task based programming model can rival highly tuned libraries in terms of performance, without much tuning effort.
Most other benchmarks evaluate the different task-based programming models among themselves, which is not relevant for this paper, as it tries to measure task-based to classic apporaches.

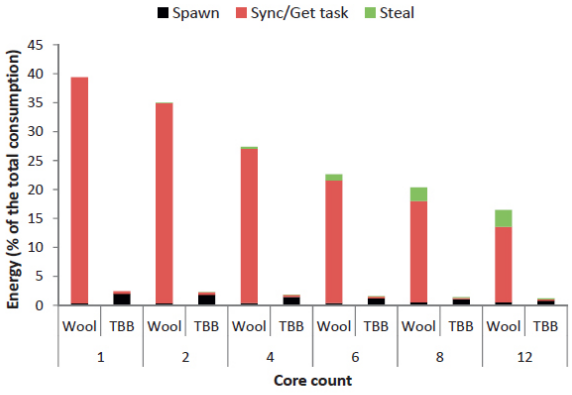
[1] http://stellar.cct.lsu.edu/2012/03/benchmarking-user-level-threads/

3.3 Efficiency

As power consumption has become an increasingly important factor in HPC, it is mandatory to take a look at the efficiency of the applications.
Iordan et al. [10] compare the overheads for different task operations, such as spawning new tasks, stealing and synchronizing. They use TBB and Wool ([7]), an experimental library with the goal of extremely low overhead task creation and synchronization.



(a) Stress



(b) MemStress

Figure 6: Percentage of total energy consumption of different operations in CPU and memory bound situations [10].

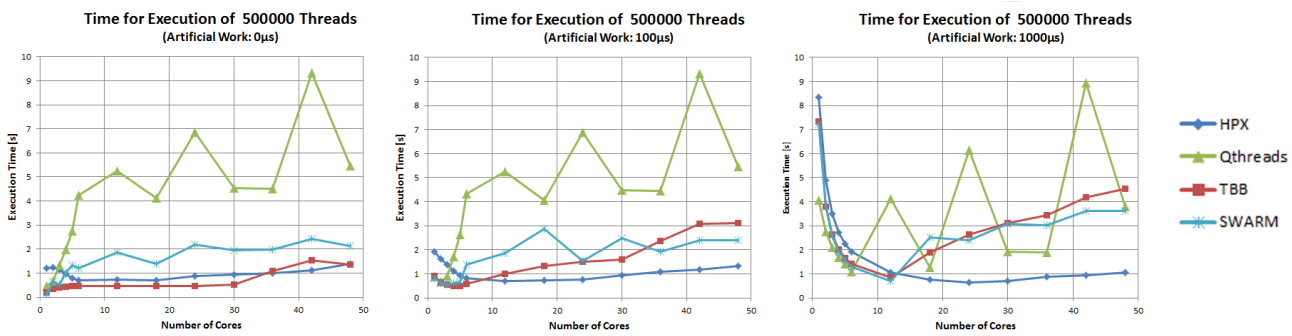


Figure 4: Task overheads for HPX, Qthreads, TBB and SWARM

The results (see Figure 6) show, that Wool surpasses TBB in spawning new tasks, but as soon as the application is bound through memory access, Wool has huge overheads in synchronizing (many workers).
In a third test with matrix multiplications, where tasks are relatively big and few, TBB surpasses Wool in energy efficiency, as Wool has a very aggressive work stealing algorithm, whereas TBB puts workers, that fail to steal tasks several times to sleep and wakes them up later.
Iordan et al. developed FASTA ([10]), a tool that speeds up simulation times by only calculating specified samples. This tool can not be used for all applications, but there are some for which a 12x speedup can be reached with an accuracy error of only 2.6%.
But in the end, Iordan et al. [1] point out, that as the number of cores grows, more power gets consumed by synchronization and management overhead.

4 Conclusion

This paper provides an overview over task-based programming models, for both shared memory and distributed memory systems. First it introduced the common concepts and illustrates the specific implementations of frameworks for shared memory systems, two of which later received support for distributed memory systems. Then it introduced three libraries that were specifically built for distributed memory and emphasizes the differences.
The evaluation showed that, compared to the usual programming models, TBP shows great promise, especially in the field of scalability and ease of use. With minimal optimization effort, performance can rival the common techniques, and as workload distribution is handled by the scheduler, adjustment for different hardware setups may be non-existent.
The presented advantages (scalability, ease of use, and efficiency) show that task-based programming models are a promising tool worth looking into.

References

[1] Iordan Alexandru, Artur Podobas, Lasse Natvig, and Mats Brorsson. Investigating the potential of energy-savings using a fine-grained task based programming model on multi-cores. 2011. QC 20120223.

[2] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. In Siegfried Benkner Jesper Larsson Träff and Jack Dongarra, editors, *The 19th European MPI Users' Group Meeting (EuroMPI 2012)*, volume 7490 of *LNCS*, Vienna, Austria, September 2012. Springer.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Ernesto Su, Priya Unnikrishnan, and Guansong Zhang. A proposal for task parallelism in openmp. In *Proceedings of the 3rd International Workshop on OpenMP: A Practical Programming Model for the Multi-Core Era*, IWOMP '07, pages 1–12, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186, Feb 2010.

[6] Pierre-Nicolas Clauss and Jens Gustedt. Iterative Computations with Ordered Read-Write Locks. *Journal of Parallel and Distributed Computing*, 70(5):496–504, 2010.

[7] Karl-Filip Faxen. Wool-a work stealing library. 36:93–100, 01 2008.

[8] P. Grubel, H. Kaiser, J. Cook, and A. Serio. The performance implication of task size for applications on the hpx runtime system. In *2015 IEEE International Conference on Cluster Computing*, pages 682–689, Sept 2015.

[9] J. Gustedt, E. Jeannot, and F. Mansouri. Optimizing locality by topology-aware placement for a task based programming model. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 164–165, Sept 2016.

[10] A. C. Iordan, M. Jahre, and L. Natvig. On the energy footprint of task based parallel applications. In *2013 International Conference on High Performance Computing Simulation (HPCS)*, pages 164–171, July 2013.

[11] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[12] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, October 1993.

[13] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. Atlanta, FL, September 2009. ACM SIGPLAN.

[14] Josep M Perez, Rosa M Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151. IEEE, 2008.

[15] Venkat Subramaniam. *Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors.* Pragmatic Bookshelf, Dallas, Tex., 2011.

[16] Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *SIGOPS Oper. Syst. Rev.*, 40(2):90–99, April 2006.

[17] Gengbin Zheng, Lixia Shi, and L. V. Kale. Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, pages 93–103, Sept 2004.