How runtime systems can support resource awareness in HPC: the HPX case

Seminar "Future Trends in High Performance Computing"

Tommaso Bianucci
Fakultät für Informatik
Technische Universität München
Email: bianucci@in.tum.de

***Abstract*— Resource awareness is the ability of a program to dynamically read the state of the resources it consumes and to adapt its execution accordingly. This awareness can be exploited to achieve a more efficient resource usage, reduce overheads, improve performance and optimize energy consumption. As HPC systems grow in size and complexity, resource awareness can also offer a way to reduce the effort required for tuning and portability. As applications use runtime systems for parallelization, there is the need for runtime systems to support and expose resource awareness in an efficient and integrated way. HPX is a modern C++ runtime system for high-performance task-based parallelization which is, to some degree, resource aware and adaptive by design. It supports load-aware task scheduling and a powerful performance monitoring framework which can be used for runtime introspection and for building dynamic resource management heuristics. Recent research efforts also show how HPX can support dynamic task grain size control and how HPX parallel algorithms can be dynamically tuned by using a mix of compile-time and runtime information for better performance. This work is a survey of existing literature providing an introduction to resource awareness and to HPX architecture and main features, providing also a perspective on how and to what degree HPX can support resource awareness and to its limitations.**

***Index Terms*— Resource awareness, HPX, task-based parallelism, HPC**

I. Introduction

Current supercomputers are approaching the long awaited exascale and are characterized by millions of cores, distributed not only over nodes but also over several different architectures as many-core CPUs, GPUs, gpGPUs[1] and FPGAs[2]. The number of processing units and their heterogeneity makes it extremely complex to program them using static and manual resource allocation paradigms.

Furthermore, many important application classes are characterized by highly unbalanced execution trees (e.g. AMR[3] strategies): in these cases the resource requirements and the load are inherently unbalanced and difficult to mitigate. This impacts on parallel performance and causes a sub-optimal resource usage, where most of the processes sit waiting for the few more expensive ones to finish their respective computations.

Solving this problem and the ability to effectively and efficiently scale computations up to the current supercomputer capabilities requires the possibility to change resource allocation and requirements dynamically.

This is where *resource aware computing* comes into play: as a general concept, we would like applications and systems to be able to react to different load distributions and resource availability directly at runtime. Resource awareness can be seen at all levels of a system: starting from hardware and operating system, through the runtime environment of choice and the application. But resource awareness can also be facilitated and supported by the choice of appropriate algorithms and programming models.

The current predominant programming model in HPC is to use a *fork-join* model for shared memory parallelization together with a *Communicating Sequential Processes (CSP)* [1] model for distributed memory parallelization. This is done in practice by leveraging respectively OpenMP and MPI. This hybrid model involves many synchronization barriers at the boundaries of parallel regions, which are often global or involve a high number of parallel processes.

Such barriers quickly become bottlenecks when the number of processes increase and in case of any load unbalance, since all threads will have to wait for the slowest one to complete its tasks (Figure 1). As foreseen by the Exascale Study Group in 2008: "Somewhere between Petascale and Exascale computing [...] the MPI model may find its limit". [2]

In order to solve this problem, alternative runtime systems based on the Asynchronous Many Tasks (AMT) model are being proposed [3]. This approach is based on splitting the computation in many fine-grained *tasks* and in defining their precise dependencies on each other. It is then the job of the runtime system to make sure each dependency is satisfied by using the appropriate synchronization. In this way we obtain an execution graph where each task will only wait for the completion of the tasks it depends on before starting. We thus lose the concept of a group of threads executing in parallel between global barriers and, most importantly, we lose the global barriers themselves.

[1] General Purpose GPUs.
[2] Field Programmable Gate Arrays.
[3] Adaptive Mesh Refinement

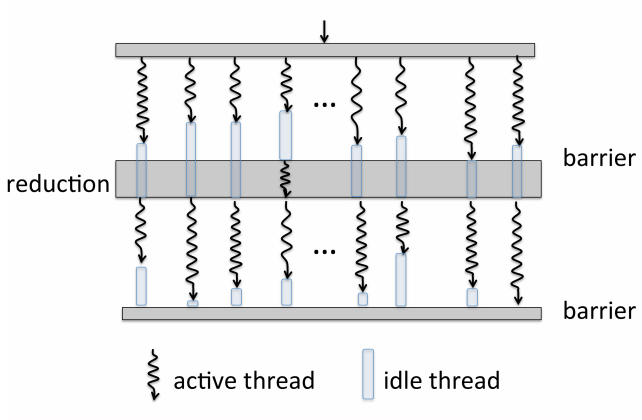

Figure 1. Global barriers and thread idle time [8]. This is a representation of how global barriers cause idling on threads with unbalanced workload. We can also see how, during a reduction, just one thread actually operates it while all the others are idle.

Task-based semantic has been integrated in a variety of runtimes libraries, such as Intel TBB [4], Charm++ [5], Qthreads [6] and HPX [3]. Also OpenMP has added extensions for task-based parallelism starting from its 3.0 release [7]. In order to achieve true scalability, a runtime supporting task-based parallelism must be able to support it in a massive fashion. This requires the ability to schedule tasks without allocating a new OS thread for each of them, which would be infeasible, but it also requires adaptive resource management and task scheduling in order to ensure the required performance.

In this work I review the general concepts of resource awareness (II), then I review the *High Performance paralleX (HPX)* runtime system, its design and its features (III). Then I discuss how HPX can support resource awareness and its limitations in this (IV). In section V I also present a basic example of HPX code and I briefly compare its performance against PThreads and OpenMP parallelizations on a shared memory machine.

II. Resource Awareness

Resource aware computing (RAC), also referred to as *runtime adaptivity* or *elasticity*, comprises a broad spectrum of techniques aimed at achieving runtime adaptivity in resource allocation and usage. The definition of resource is very broad and comprises both (i) hardware entities such as computational units, memory, bus or network bandwidth, I/O and (ii) software entities, such as task queues, message buffers, etc.

The key point of resource awareness is to make the application and the system aware, at runtime, of what resources are available and in what amount and how much of these resources is currently in use.

This is opposed to the standard approach in which software is optimized to a certain degree for specific constraints and amount of resources before execution, either at compile time or through passing tuned parameters. Also, in the current standard approach the program is statically assigned a certain amount of resources (e.g. processors) at the beginning of its run and this amount stays assigned to the program also in phases when it does not need them. This causes the resource to be unavailable to other processes while it is, in fact, idle.

Resource awareness can be seen at several levels and in different contexts.

In embedded computing and in the context of *Multi-Processor System on Chip (MPSoC)* the aim is to "deal with increasing imperfections such as process variation, fault rates, aging effects, and power as well as thermal problems" [9]. In this field interesting research efforts are going in the direction of the *Invasive Computing* [10] paradigm: in this model a program can dynamically explore and claim resources in its neighborhood in order to increase its parallelism, in a phase called *invasion*; then, when a lower degree of parallelism is needed, it can autonomously release these resources through an opposite process called *retreat*, making the resources available to other applications; all this without the need for a centralized mechanism managing the allocation of resources.

In the context of a runtime system or a single parallel application we can see resource awareness as the ability to steer the execution in a way to dynamically fit current resource availability. In this regard, task scheduling takes a major role within runtime systems, as we will later see for HPX in section IV. The ability to query the state of the resources at runtime can also lead to resource awareness in applications: it can be used for runtime tuning of execution parameters in order to achieve, e.g. a better time to solution or a higher energy efficiency. One example for this is automatic tuning of task grain size [8].

At the supercomputing facility level, resource awareness can instead be used for more efficient scheduling of jobs, in order to get a better utilization of resources and more predictable power requirements. One interesting example in this direction comes again from invasive computing: here an extension of MPI supporting invasive operations has been developed in order to allow varying resource utilization at runtime [11]. The research effort is currently into developing a process manager capable of leveraging this elasticity for a better machine utilization and, ultimately, an improved throughput.

III. HPX

HPX is a C++ library and runtime system for task-based parallelization. It treats both intra- and inter-node parallelization within an homogeneous interface and it adheres to the C++11/14 standard, which introduced basic support for task-based parallelism.

It features a global address space, the ability to migrate work remotely in the proximity of data, and it supports task dependencies and continuations. Its task scheduler is designed to introduce minimal overhead and supports very fine-grained tasks down to ~1μs [12] and work-stealing for automatic load balancing. It also delivers a powerful performance monitoring system which allows a program to query various performance metrics at runtime and to react accordingly.

A. The ParalleX execution model

HPX implements the ParalleX [13] execution model, which leverages medium- and fine-grained task parallelism and aims at optimizing both parallel efficiency and programmability of parallel code.

The key highlights of this model are:

Asynchronous task execution Functions are meant to be called asynchronously and to yield a proxy for the actual return value. Such a proxy is called a *future* (figure 2). The program will need to synchronize only when the actual return value is needed, e.g. for a later computation. If the task was able to complete between the asynchronous call and the synchronization step, then no waiting time is needed.

Lightweight synchronization Not only we can use futures but we can also make an asynchronous call depend on one or several futures: this enables the runtime system to keep track of the actual dependencies and removes the need for expensive synchronization mechanisms, such as global barriers.

Active Global Address Space (AGAS) ParalleX features a global address space abstraction service. This address space spans all the available hardware entities, called *localities*. The peculiarity of AGAS is that it is not a *partitioned* global address space (PGAS): in PGAS the global address space is statically partitioned across the available localities and moving an object to a different locality requires a change of its address; in AGAS the address space is dynamically and adaptively mapped to the underlying localities, allowing transparent migration of objects across localities while they can still retain their *global identifier (GID)*.

Message-driven queue-based scheduling When a task execution is requested, which may be on any remote locality, an active message, called *parcel*, is sent to the target locality. This triggers the creation of a *PX-Thread*[4], which will be queued and then scheduled for execution on the OS thread(s) managed by the target locality. This form of message passing is therefore not limited to data and it does not require explicit receive operations to be invoked on the target side. The queuing and scheduling is designed in a way to allow for idling processors or cores to *steal* work from the queues of other ones: this allows for efficient load balancing and prevents starvation .

B. HPX high-level architecture

HPX is a runtime system implementing the ParalleX model as a C++ library and this is done by strictly adhering to C++11/14 standard interfaces for task-based parallelism. It aims to address the four main factors that prevent scaling in scaling-impaired applications, which are referred to as *SLOW* [14]:

[4]From now on PX-Threads will be referred to just as *threads*.

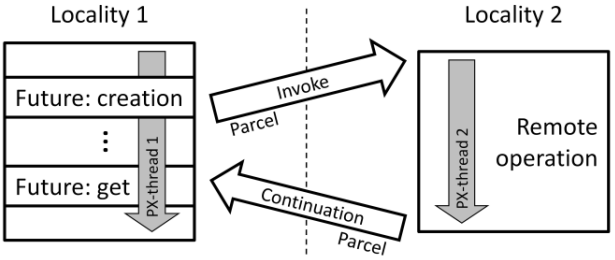

Figure 2. Schematic flow diagram of a future [13]. An asynchronous function invocation immediately returns a future, while the function is executed on another thread and potentially on another locality. On the caller thread the execution can proceed further until the .get() method is called on the future: this is when the actual synchronization with the remote operation takes place.

Starvation Not all resources are fully utilized because of a lack of enough concurrent work to execute.

Latencies Intrinsic delays in accessing remote resources.

Overheads The overheads of parallelization, i.e. the work required for management of the parallel computation and any extra work which would not be necessary in a sequential version.

Waiting for contention resolution Any delay caused by oversubscription of shared resources.

HPX tries to deal with the above problems by embracing the following design principles [14]:

- focus on latency hiding instead of latency avoidance,
- embrace fine-grained parallelism instead of heavyweight threads,
- rediscover constraint based synchronization to replace global barriers,
- use adaptive locality control instead of static data distribution,
- prefer moving work to data over moving data to work and
- favor message driven computation over message passing.

These principles are well-known and some of them are currently used also by other parallelization-oriented languages and libraries. However HPX is the first case in which all of them are coherently embraced and exposed in its programming model [14].

Based on these principles, HPX was designed with a modular architecture composed of five high-level subsystems, each exposing a specific set of functionalities:

a) The Threading Subsystem: When a new thread is created, HPX queues it at an appropriate locality and it then schedules it according to configurable policies. The thread scheduling is cooperative, i.e. non-preemptive, since preemption and the overhead associated to content switches would not make sense in the context of a single application. Threads are scheduled onto a pool of OS threads, which are usually one per core, without requiring any kernel transition and thus removing all the overhead associated to the creation of OS threads.

b) The Parcel Subsystem: Parcels are the HPX implementation of active messages, i.e. messages which can not only deliver data but also trigger execution of methods on remote localities. Parcels carry the GID of the destination, a remote action, arguments for the action (data) and, if required, a continuation.

c) Local Control Objects: "Local Control Objects (LCOs) control parallelization and synchronization of HPX applications. An LCO is any object that may create, activate, or reactivate an HPX thread." [8]

The most prominent LCOs delivered by HPX are *futures* and *dataflow* objects. Futures are proxies for values which might not have been computed yet and include a synchronization when the actual value is requested. Dataflow objects are instead LCOs which depend on a set of futures as input and they return themselves a future for the result of their continuation.

Futures and dataflow LCOs allow to express the true data dependencies within an application and to translate them into the associated execution tree and necessary synchronizations.

d) Active Global Address Space: As already mentioned in III-A, one of ParalleX main features is the AGAS: HPX implements an AGAS service which delivers those functionalities.

e) The Performance Monitoring Framework: HPX implements a performance monitoring framework based on a variety of *performance counters*, which are objects providing metrics and statistics on the performance of (i) hardware, (ii) application, (iii) HPX runtime and (iv) OS. Performance counters are first class objects, they are therefore addressable by their GID from any locality and are available to both the application and the HPX runtime for performing introspection at runtime on how well the system is performing. [14] Performance counters expose a predefined interface and HPX exposes specific API functions allowing to create, manage and release counters, as well as to read their data in a structured way. Performance counters are not only available from within the application. By passing standard command line flags to an HPX application we can conveniently have the performance counters of interest periodically logged to screen or to a file during execution. Performance counter can also be accessed in real-time by other utilities by connecting to a running HPX application through its parcel transport layer [14]. They are useful tools for performance analysis and for identifying bottlenecks, and they are even more useful as they provide the necessary infrastructure for building resource awareness into an HPX application [12].

IV. Resource awareness in HPX

HPX supports resource awareness by design and, up to a certain degree, automatically. Its execution model is based on the assumption that threading, synchronization and data distribution must not be exposed to the programmer and must be handled automatically behind the scenes. HPX provides an abstraction for parallelism which does not require the programmer to think about localities, send/receive instructions, threads or barriers, which are referred to by Hartmut Kaiser, the HPX project lead, as the "GOTOs of parallel computing" [15]. HPX also encourages the programmer to define fine grained tasks, while the runtime takes care of the actual scheduling and synchronization. This has the positive side effect of making the runtime adaptive, by default, in terms of load balancing.

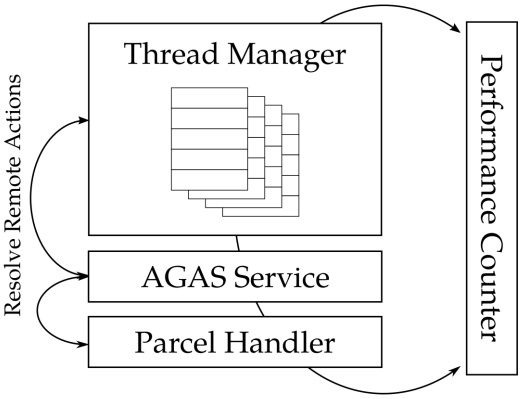

Figure 3. Overview of the HPX runtime system components [3].

HPX manages hardware resources and exposes them through an abstraction process according to its execution model. With the exception of percolation, which is described later in this section, in HPX we can consider as resources all these abstractions provided by the runtime system. For example worker threads, localities and the parcel subsystem are respectively abstractions of processing units, compute nodes and the network infrastructure. Task queues are instead a software entity type of resource which does not correspond to any hardware resource but represents the load associated with a processing unit.

It is also important to consider that the performance monitoring framework — i.e. the infrastructure allowing an application to retrieve performance statistics about the underlying resources and which enables adaptivity on the application side — can be extended by new, user-defined, performance counters, thus potentially broadening the set of resources the application can be aware of.

No comprehensive analysis of resource awareness in HPX has been published so far. Therefore I below summarize the capabilities of HPX — as reported in scientific literature — in support for resource awareness together with the factors which currently limit the scope of its adaptivity.

A. Capabilities

a) Task scheduling: The HPX thread manager and the way it schedules queues for each worker thread are configurable

with three main pre-defined policies [3], [16]:

Static The thread manager maintains a work queue for each worker thread and tasks are distributed to queues in a round robin fashion. In this policy there is no way for tasks to change their allocation to queues.

Local This is the default scheduling policy. In this policy the thread manager maintains a work queue for each worker thread plus extra special queues for high and low priority tasks. Work is distributed to queues as in the static policy, but as soon as a load imbalance is detected on the worker threads, work is stolen from higher loaded threads and distributed to lower loaded ones.

Hierarchical In this policy the thread manager maintains a tree of work queues where the leaves correspond to the worker threads. New tasks are always enqueued at the root of the tree and then trickle down the hierarchy. In this policy work can be stolen at any level of the tree, from queues belonging to the same level.

From the above policies we can see one very important behaviour of the task scheduler which allows for resource awareness when enabled: *work stealing*.

As mentioned, work stealing occurs as soon as a load imbalance is detected among different worker threads' queues. This is optionally done in a NUMA-aware way, i.e. the work is redistributed preferentially within the same NUMA domain in order to preserve memory access performance.

Work stealing therefore operates an automatic load balancing action at runtime and is therefore a means to get adaptivity out of the box with HPX.

An important factor that has to be taken into account for effective task scheduling is data — both temporal and spatial — locality. Performing task scheduling blind to data locality can lead to substantial performance degradation [17], while locality aware scheduling can effectively reduce the number of data load and transfer operations and their associated overhead.

HPX can include data locality awareness in task scheduling and its task dependency semantics allows hierarchical grouping of tasks, therefore naturally exposing different levels of data locality [16].

b) AGAS: Although AGAS does not a priori imply resource awareness, it is an important feature within HPX in support for adaptivity. It "is a dynamic and adaptive address space which evolves over the lifetime of an application" and "is the foundation for dynamic locality control as objects can be moved to other localities without changing their global address". [14] This allows implementing mechanisms to relocate objects to different localities in response to performance metrics and resource constraints [16] and still being able to access them from the rest of the application in a transparent way. The migration decision has to be based on performance metrics and be aware of resources also for what concerns the target locality, in order to ensure a better performance as outcome. This requires a performance monitoring infrastructure allowing access to metrics for remote localities, which in HPX is fulfilled by performance counters. The object relocation logic can be therefore completely decoupled from the code accessing the objects and this clearly allows a much cleaner and modular code and it fulfills the underlying design goal of having parallelization and data distribution internalities as much hidden from the programmer as possible.

c) Percolation: Percolation is a special use of parcels which allows directly targeting hardware resources, instead of logical data objects, as destination. Percolation provides therefore a way to address specialized hardware — such as gpGPUs and FPGAs — directly through the parcel layer, enabling to provide work to these accelerators as well as getting back the results in a clean way. The percolation mechanism in HPX is aware of the characteristics of specialized hardware resources and can therefore route relevant tasks to these resources in a efficient way [16].

d) Performance counters: Performance counters are a central feature of HPX and play a key role in its resource awareness capabilities.

An efficient performance measurement framework is a requirement for any runtime system targeting fine grained parallelism and exascale computing: it needs to be able to gather and manage a massive amount of information, while at the same time being as unobtrusive as possible, in order to minimize its perturbations to the application performance. Furthermore, such a measurement framework needs to be able to scale as required by applications and available resources [16].

Traditionally performance measurements are used for post-run analysis, for either debugging or optimization of parameters. However as the degree of parallelism increases — and so do performance-affecting parameters — it becomes more and more crucial to have well-integrated dynamic performance measurement capabilities and to leverage the measured data for dynamic tuning of application parameters.

HPX performance monitoring framework was designed having runtime adaptivity in mind [14] and they can indeed support it in a convenient way.

An example of how an application could consume performance counters to enhance its resource awareness is given in [8]: "[...] an application that transfers data over the network could consume counter data from a network switch to determine how much data to transfer without competing for network bandwidth with other network traffic. The application could use the counter data to adjust its transfer rate as the bandwidth usage from other network traffic increases or decreases."

Furthermore, as shown in a proof of concept by Grubel [8], the HPX performance monitoring framework can be used, together with an additional library and additional changes in the application code, to achieve automatic tuning of task grain size. This kind of feedback loop allows the application to react to real-time performance metrics such as task scheduling overheads to dynamically adapt its behaviour to achieve better performance.

Another interesting work which involves adaptivity mediated by performance counters is about *smart executors* in HPX. Khatami et al [18] showed how loop parallelization efficiency can be tuned, at runtime, using machine learning on a combination of static information gathered at compile time and dynamic information captured at runtime. Their tests on HPX parallel algorithms[5] showed a speedup improvement of 12% ~ 35% for the Matrix Multiplication, Stream and 2D Stencil benchmarks with respect to the standard HPX implementation.

B. Limitations

a) (An)elasticity of HPX: Although we have seen that an application can be elastic within HPX, HPX itself does not currently support any variation of the number of worker threads or localities at runtime. As already mentioned in section II, this capability, coupled to a process manager capable of leveraging it, can improve resource usage and throughput of a supercomputer.

Projects as Elastic MPI — an MPI extension implementing invasive computing semantics [19] — and Adaptive MPI — which implements MPI ranks as Charm++ user level threads [20], [21] — are examples of already ongoing research in this direction.

b) Power management: Although HPX can lead to a more efficient power consumption due to its load balancing capabilities, it does not explicitly support any power management facility.

The current trend in supercomputing is to pay more and more attention to energy efficiency, as current top tier supercomputers already have energy requirements in the order of 10MW. Research efforts are currently going into the development of more energy efficient hardware, as well as integrating energy management capabilities into the software [22].

Energy awareness requires the ability to read metrics as temperature, clock frequency and power consumption from the underlying hardware and to react with strategies as Dynamic Voltage and Frequency Scaling (DVFS).

c) Fault tolerance: Increasing the parallelism of a system implies an increase in the number of hardware devices involved in a computation. This increases the chances of incurring in a component failure. Current and future HPC systems must therefore include means to recover the intermediate state of a computation prior to the failure and to resume the task from that point.

HPX does not currently support any facility for checkpointing or for recovery from partial data losses.

[5]HPX implements parallel algorithms as defined in C++ Standard.

V. Examples

A. Mandelbrot set

Drawing the Mandelbrot set is a classical example of an *embarrassingly parallel* problem, i.e. a problem which can be parallelized with little or no dependency between the parallel components. In this case there is no dependency at all, since each pixel can be computed independently from the others.

However, pixels belonging to the set and pixels at different distances from the boundary require different computational efforts to be drawn[6], causing the parallel tasks to be potentially unbalanced.

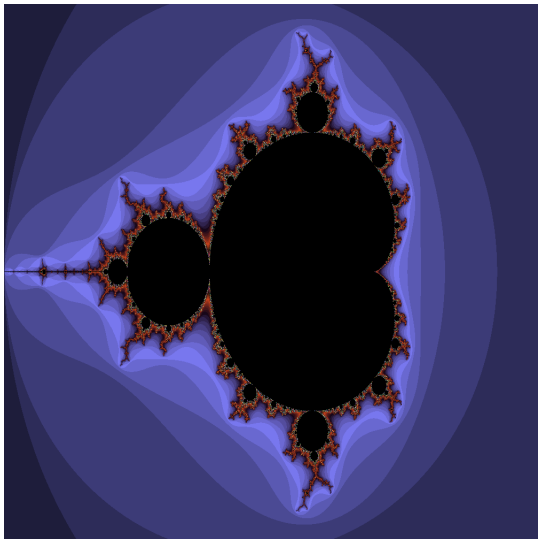

Figure 4. An image of the Mandelbrot set generated by the code in this example.

Usually the image is split into "slices" containing a certain number of rows or columns of the image and each of these slices is assigned to a thread. Within a given slice the computation is performed pixel-by-pixel, sequentially. The traditional way to ensure load balancing in a PThreads implementation is to split the image into small slices and to assign one of these to each thread; as soon as a thread finishes processing its slice, it gets a new one and proceeds further. This requires additional code for managing how threads can get new work to do and it also requires synchronization, to ensure that no two threads get the same slice to process.

The implementation with HPX is much simpler: for each row of the image an `async` call of the compute kernel is performed and the resulting future is stored into a vector. When all the asynchronous tasks have been created, `hpx::wait_all` is called on the vector of futures, making sure all tasks are completed before exiting.

I have tested strong scaling[7] on the HPX- and PThread-based versions, with different problem sizes. I have also implemented

[6]Deciding whether a complex number belongs to the Mandelbrot set is done by checking if an associated sequence converges: where the sequence diverges faster, the point is quickly marked as external, while where the divergence is slower or where the sequence converges, it takes longer before a decision is reached.

[7]How runtime improves by increasing the number of threads for a fixed problem size.

and tested, for comparison purpose, a simple OpenMP-based version, which just parallelizes the outer loop (the row-loop). It uses dynamic scheduling for load balancing and a stride of 1 for consistency with the task grain size used with HPX.

The machine where I performed the tests has two sockets with 10 cores each, with each core exposing 2 hyperthreads. Unfortunately the machine was shared with other jobs, meaning that tests performed with a high number of threads have surely experienced preemption by the OS, artificially increasing their runtime. However, this test is meant to roughly compare the three implementations and not to give an accurate measure on HPX's scaling capabilities. Keeping this and the aforementioned caveat in mind, we can see (Figure 5) how the performance of the HPX-based implementation is on par with the optimized PThreads-based one. On the other end we can also see how following the same approach with OpenMP does not yield the same performance: one possible explanation is that the relatively small task grain size chosen causes a relevant overhead in OpenMP.

This is clearly a toy example but it shows how HPX allows achieving good performance with simple and clean code and without any explicit optimization effort. Programmability is an important factor for making parallelism more accessible and this is surely one of the strengths of HPX.

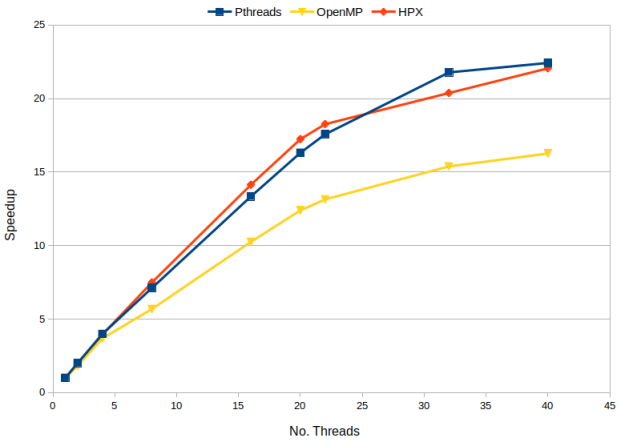

Figure 5. Speedup comparison for PThreads, OpenMP and HPX implementations on a 10^4x10^4 image. Both the OpenMP and HPX versions use the basic features of the respective runtime system, meaning a parallelization on the row-loop (with dynamic scheduling) and an asynchronous execution for each row of the image respectively. The PThreads version instead includes a more complex logic to ensure load balancing between the threads. The results show how the HPX implementation, using basic syntax and without any specific tuning, is able to perform on par with a specifically optimized PThreads implementation. All three implementations use the same task grain size consisting of one image row.

B. *Reading performance counters from the command line*

Although it is possible to access performance counter data from within the application, using the HPX API, it is also possible to make the HPX runtime print data from any performance counter to the command line. This can be useful for debugging or for manual tuning of an HPX application.

This is achieved by passing extra flags to an HPX application and is completely handled by the runtime system, thus not requiring any change in the application code.

Available counters can be listed using the `--hpx:list-counters` flag and, if HPX was built with PAPI[8] support, available PAPI events can be listed with `--hpx:papi-event-info=all`.

A counter can be printed by passing the `--hpx:print-counter='<counter_name>'` argument and several counters may be specified at the same time by passing this argument multiple times. This makes the HPX runtime system print the specified counters at the end of the execution. It is also possible to have the counters printed at regular time intervals during execution by adding the `--hpx:print-counter-interval=<interval_ms>` argument, where the interval is expressed in milliseconds.

VI. Conclusions

As the generalized trend in computing is to move towards manycore architectures and more etherogeneous machines, resource awareness is becoming more and more important in order to efficiently use the extremely high number of resources involved in a computation. Highly parallel environments are very sensitive to bottlenecks and inefficiencies and a dynamically adaptive and resource aware execution can help in avoiding performance degradation.

However, in order to exploit the potential of resource awareness there is also the need for a flexible and dynamic programming model: task-based parallelism has proven to be a powerful approach for increasing the efficiency of parallel applications and is available in several popular runtime environments.

HPX is a modern C++ runtime system for high-performance task-based parallelization. It offers an homogeneous interface for both shared memory and distributed memory parallelization and it can outperform both OpenMP and MPI in their respective fields of application. It has efficient and adaptive thread scheduling capabilities and it incorporates a performance monitoring framework which allows building resource-aware heuristics into applications.

HPX also implements parallel algorithms according to the C++ standard, making high-performance high-level parallelism available also to non specialists.

Thanks to its performance, to its solid theoretical foundations and to its flexibility, HPX has the potential to become one of the major next generation parallelization paradigms. Its current limitations, reported in IV-B, indicate a path for possible further development and resolving these limitations would yield an even more versatile and powerful tool for the HPC community.

As MPI and OpenMP start showing their architectural limits, HPX should be considered as a valid alternative, not only for high performance computing but also for general purpose parallelism.

[8]Performance Application Programming Interface, which allows accessing hardware counters.

REFERENCES

[1] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[2] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008.

[3] T. Heller, P. Diehl, Z. Byerly, J. Biddiscombe, and H. Kaiser, "Hpx–an open source c++ standard library for parallelism and concurrency," 2017.

[4] G. Contreras and M. Martonosi, "Characterizing and improving the performance of intel threading building blocks," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008, pp. 57–66.

[5] "Charm++ - parallel programming framework." [Online]. Available: http://charmplusplus.org/

[6] "The qthread library." [Online]. Available: http://www.cs.sandia.gov/qthreads/

[7] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," May 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[8] P. A. Grubel, *Dynamic adaptation in hpx-a task-based parallel runtime system*. New Mexico State University, 2016.

[9] F. Hannig, S. Roloff, G. Snelting, J. Teich, and A. Zwinkau, "Resource-aware programming and simulation of mpsoc architectures through extension of x10," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*. ACM, 2011, pp. 48–55.

[10] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting, "Invasive computing: An overview," in *Multiprocessor System-on-Chip*. Springer, 2011, pp. 241–268.

[11] I. A. C. Ureña, M. Riepen, M. Konow, and M. Gerndt, "Invasive mpi on intel's single-chip cloud computer," in *International Conference on Architecture of Computing Systems*. Springer, 2012, pp. 74–85.

[12] P. Grubel, H. Kaiser, K. Huck, and J. Cook, "Using intrinsic performance counters to assess efficiency in task-based parallel applications," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 1692–1701.

[13] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*. IEEE, 2009, pp. 394–401.

[14] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. 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*. ACM, 2014, p. 6.

[15] H. Kaiser. (2014) Plain threads are the GOTO of todays computing. Meeting C++. [Online]. Available: https://youtu.be/4OCUEgSNIAY

[16] V. C. Amatya, "Parallel processes in hpx: Designing an infrastructure for adaptive resource management," 2014.

[17] C. Connelly and C. S. Ellis, "Workload characterization and locality management for coarse grain multiprocessors," Technical Report CS-1994-30, Duke University, Tech. Rep., 1994.

[18] Z. Khatami, L. Troska, H. Kaiser, J. Ramanujam, and A. Serio, "Hpx smart executors," in *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware*. ACM, 2017, p. 3.

[19] C. Ureña and I. Alberto, "Resource-elasticity support for distributed memory hpc applications," Ph.D. dissertation, Technische Universität München, 2017.

[20] A. Gupta, B. Acun, O. Sarood, and L. V. Kalé, "Towards realizing the potential of malleable jobs," in *High Performance Computing (HiPC), 2014 21st International Conference on*. IEEE, 2014, pp. 1–10.

[21] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. V. Kale, "A batch system with efficient adaptive scheduling for malleable and evolving applications," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 429–438.

[22] A. Surve, A. Khomane, and S. Cheke, "Energy awareness in hpc: A survey," *International Journal of Computer Science and Mobile Computing*, vol. 2, no. 3, pp. 46–51, 2013.