Higher-level Parallelization for Local and Distributed Asynchronous Task-Based Programming

Hartmut Kaiser[1,3]
hkaiser@cct.lsu.edu

Thomas Heller[2,3]
thomas.heller@cs.fau.de

Daniel Bourgeois[1,3]
dbourgeois@cct.lsu.edu

Dietmar Fey[2]
dietmar.fey@cs.fau.de

[1]Center for Computation and Technology,
Louisiana State University, Louisiana, U.S.A.

[2]Computer Science 3, Computer Architectures,
Friedrich-Alexander-University, Erlangen, Germany

[3]The STE||AR Group (http://www.stellar-group.org)

ABSTRACT

One of the biggest challenges on the way to exascale computing is programmability in the context of performance portability. The efficient utilization of the prospective architectures of exascale supercomputers will be challenging in many ways, very much because of a massive increase of on-node parallelism, and an increase of complexity of memory hierarchies. Parallel programming models need to be able to formulate algorithms that allow exploiting these architectural peculiarities. The recent revival of interest in the industry and wider community for the C++ language has spurred a remarkable amount of standardization proposals and technical specifications. Among those efforts is the development of seamlessly integrating various types of parallelism, such as iterative parallel execution, task-based parallelism, asynchronous execution flows, continuation style computation, and explicit fork-join control flow of independent and non-homogeneous code paths. Those proposals are the foundation of a powerful high-level abstraction that allows C++ codes to deal with an ever increasing architectural complexity in recent hardware developments.

In this paper, we present the results of developing those higher level parallelization facilities in HPX, a general purpose C++ runtime system for applications of any scale. The developed higher-level parallelization APIs have been designed to overcome the limitations of today's prevalently used programming models in C++ codes. HPX exposes a uniform higher-level API which gives the application programmer syntactic and semantic equivalence of various types of on-node and off-node parallelism, all of which are well integrated into the C++ type system. We show that these higher level facilities which are fully aligned with modern C++ programming concepts, are easily extensible, fully generic, and enable highly efficient parallelization on par with or better than existing equivalent applications based on OpenMP and/or MPI.

Keywords

task-based parallelism, distributed asynchronous computing, HPX

ESPM2 2015 November 15 - 20, 2015, Austin TX, USA

ACM 978-1-4503-3996-4/15/11...$15.00.
DOI: http://dx.doi.org/10.1145/2832241.2832244

1. INTRODUCTION

The massive increase in local parallelism is one of the greatest challenges amongst the many issues imposed by today's and tomorrow's peta- and exascale systems. This paper focuses on how to possibly address massive on-node parallelism and data locality, how to improve the programmability of such systems while maintaining performance portability. It is to be expected that exascale ready architectures will be build with up to billion way concurrency and multiple levels of memory hierarchies. The growing complexity of those next generation supercomputing architectures requires for new programming models to arise. Those have to be able to exploit the underlying hardware in a programmable way [8, 22]. We are focusing our efforts on providing higher-level abstractions based on and derived from the current C++ standard.

This paper explores the capabilities of parallel abstractions which go beyond the current state of the art, like loop based parallelism as known from OpenMP [2, 10, 11], or other pragma based approaches. The immediate benefit is that our solution tightly integrates with the C++ type system and can take advantage of modern C++. In addition with the help of the exposed uniform API of HPX, we are able to leverage the same parallelization constructs in a distributed memory environment. This sets our solution apart from the prevalently used programming model for distributed computing, MPI [25], which only exposes fairly low level message passing functionality, augmented by higher level collective operations.

The recent standardization effort to extend the C++ standard library with a comprehensive set of parallel algorithms has partially rectified the situation for C++ codes by defining higher-level loop parallelization facilities (Parallelism TS [7]). The Parallelism TS applies the proven design concepts of the Standard Template Library (STL, [28]) to parallel loop executions. It newly introduces parallel algorithms which are fully generic and well integrated within the C++ type system and maintain the same flexibility and extensibility as the original STL algorithms.

However, the Parallelism TS does not provide a solution that seamlessly integrates iterative execution with other types of parallelization, such as task-based parallelism, asynchronous execution flows, continuation style computation, and explicit fork-join control flow of independent and non-homogeneous code paths. Other standardization documents, such as the Concurrency TS [6] introduce facilities for parallelizing asynchronous, continuation based execution,

and the task-block proposal [5] addresses fork-join parallelism of independent tasks. But these documents are still not integrated well enough with the Parallelism TS, offering only limited interoperability between different types of parallelism.

In this paper we present the results of developing higher-level parallelization facilities based on HPX [20, 21], a general purpose C++ runtime system for applications of any scale. The implemented higher-level parallelization APIs have been designed to overcome limitations of today's prevalently used programming models in C++ codes. The constructs exposed by HPX are well aligned with the existing C++ standard [31, 32] and the ongoing standardization work. However, they go beyond those with the goal of providing a flexible, well integrated, and extensible framework for parallelizing applications using a wide range of types of parallelism. Because HPX is designed for use on both, single and multiple nodes, the described facilities are also available in distributed use cases, which further broadens their usability. HPX exposes a uniform higher-level API which gives the application programmer syntactic and semantic equivalence of various types of on-node and off-node parallelism, all of which are well integrated into the C++ type system. We show that not only are these higher level facilities fully aligned with modern C++ programming concepts, easily extensible and fully generic, but that they also enable highly efficient parallelization on par or better than what existing equivalent applications based on OpenMP and/or MPI can achieve.

2. RELATED WORK

The differences between HPX and other parallel models and runtime systems like X10, Chapel, Charm++, Habanero, OpenMP and MPI have been discussed in detail in [20]. The high-level parallelization APIs discussed in this paper are unique features implemented in the HPX parallel runtime system. One match is the pragma-based construct in OpenMP (`#pragma omp`), which often feel misplaced in a modern C++ application since they operate outside of the C++ type system and are restricted to integer based loop constructs. The other notable solutions are provided by the Intel Threading Building Blocks [18] and Microsoft's Parallel Patterns Library [26] which provide similar APIs as presented in this paper. However, all the three aforementioned solutions fall short when it comes to applications for distributed memory.

3. HPX – A GENERAL PURPOSE PARALLEL C++ RUNTIME SYSTEM

HPX is a general purpose C++ runtime system for parallel and distributed applications of any scale. It has been described in detail in other publications [15, 16, 20, 21]. We will highlight its main characteristics in this section.

HPX represents an innovative mixture of a global system-wide address space (AGAS - Active Global 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. As such, HPX is a novel combination of well-known ideas with new and unique overarching concepts. It aims to resolve the problems related to scalability, resiliency, power efficiency, and runtime adaptive resource management that will be of growing importance as HPC architectures evolve from Peta- to Exa-scale. HPX departs from today's prevalent parallel programming models with the goal of mitigating traditional limitations, such as implicit and explicit (global and local) barriers, coarse grain parallelism, and lack of easily achievable overlap between computation and communication.

HPX exposes a coherent programming model unifying all different types of parallelism available in HPC systems. By modelling the API after the interfaces defined by the C++ standards, programmers are enabled to write fully asynchronous code using hundreds of millions of thread. This ease of programming extends to both parallel and distributed applications. In addition, the implementation adheres to the programming guidelines and code quality requirements defined by the Boost collection of C++ libraries [1].

HPX is the first open source runtime system implementing the concepts of the ParalleX execution model [19, 34] on conventional systems including Linux clusters, Windows, Macintosh, Android, Xeon/Phi, and the Bluegene/Q. It is published under a liberal open-source license and has an open, active, and thriving developer and user community. HPX is built using existing ideas and concepts (such as static and dynamic dataflow, fine grain futures-based synchronization, and continuation style programming), some of which have been known for decades, however the combination of these ideas and their strict application forms overarching design principles that make HPX unique [20].

External to HPX, libraries have been developed which provide additional functionality and extend the HPX paradigm beyond CPU computing. Notable of these libraries are HPXCL [30] and APEX [17]. The first of these libraries, HPXCL, allows programmers to seamlessly incorporate GPUs into their HPX applications. Users write an OpenCL kernel and pass it to HPXCL which manages the data offloading and synchronization of the results with the parallel execution flow on the main CPUs. APEX, a policy engine, takes advantage of the HPX performance counter framework to gather arbitrary knowledge about the system and uses the information to make runtime-adaptive decisions based on user defined policies.

4. CONCEPTS OF PARALLELISM

Any type of parallelism can be seen as a stream of small, independent or dependent work items which can be executed in a certain, perhaps concurrent, order. Given a set of work items, the purpose of a particular implementation of any parallel programming model is to, at a minimum, allow control over the following execution properties:

- The *execution restrictions* or the guarantees on thread safety that are applicable for the work items (i.e 'can be run concurrently,' or 'has to be run sequentially', etc.)
- In what *sequence* the work items have to be executed (i.e. 'this work item depends on the availability of a result', 'no restrictions apply', etc.)
- *Where* the work items should be executed (i.e. 'on this core', 'on that node', 'on this NUMA domain', or 'wherever this data item is located', etc.)
- The *grain size* control for tasks which should be run together on the same thread of execution. (i.e. 'each thread of execution should run the same number of work items')

For each of these properties we define a concept (see Table 1). A C++ type conforms to a given concept if it implements the set of operations defined by that concept and adheres to the same syntax and semantics. In particular, for many parallel operations exposed

by the HPX API, a C++ type conforming to a specific concept is required. Note that the types conforming to one of the concepts listed in Table 1 can be either one of the predefined types in HPX or a type defined by the application. This ensures full flexibility, genericity, and extensibility of the parallel facilities implemented by HPX.

Property	C++ concept name
Execution *restrictions*	`execution_policy`
Sequence of execution	`executor`
Where execution happens	`executor`
Grain size of work items	`executor_parameter`

Table 1: Concept names defined by HPX to represent the exposed execution properties.

4.1 Execution Policies

The concept of an `execution_policy` is defined in the Parallelism TS [6] as "an object that expresses the requirements on the ordering of functions invoked as a consequence of the invocation of a standard algorithm". For example, calling an algorithm with a sequential execution policy would require the algorithm to be run sequentially. Whereas calling an algorithm with a parallel execution policy would permit, but not require, the algorithm to be run in parallel. The `execution_policy` concept and types in HPX syntactically and semantically conform to the Parallelism TS. In addition to parallel algorithms, execution policies in HPX are used as a general means of specifying the execution restrictions of the work items for all different types of parallelism supported by HPX. The Parallelism TS specified three execution policies, but explicitly allows implementations to add their own (see Table 2 for all execution policies specified and additionally implemented by HPX).

The addition of `par(task)` and `seq(task)`, or task execution policies, in HPX is an important extension of the Parallelism TS. In HPX, task execution policies instruct any of the algorithms to return immediately, giving back to the invocation site a future object representing the final outcome of the algorithm. For example, calling the `all_of` algorithm with `par` computes and then returns a boolean result. Whereas calling the `all_of` algorithm with `par(task)` immediately returns a future object representing a forthcoming boolean. Task execution policies also enable the integration of parallel algorithms and fork-join task blocks with asynchronous execution flow. Parallel algorithms and fork-join task blocks are conventionally fully synchronous in the sense that all iterations of a loop or tasks of the task block have to finish before the algorithm or task exits. But through task execution policies, asynchronous execution flows and continuation-style programming can be utilitized(see Section 5.2).

Policy	Description	Implemented by
`seq`	sequential execution	Parallelism TS, HPX
`par`	parallel execution	Parallelism TS, HPX
`par_vec`	parallel and vectorized execution	Parallelism TS
`seq(task)`	sequential and asynchronous execution	HPX
`par(task)`	parallel and asynchronous execution	HPX

Table 2: The execution policies defined by the Parallelism TS and implemented in HPX (HPX does not implement the `par_vec` execution policy as this requires compiler support).

Besides the option to create task execution policies, every HPX execution policy also has associated default `executor` and `executor_parameter` instances which are accessible through the exposed `execution_policy` interface. Additionally, execution policies can be rebound to another (possibly user defined) executor or executor parameters object (see Listing 1 for a corresponding example).

4.2 Executors

The concept of an `executor` as implemented in HPX is aligned with what is proposed by the C++ standardization document N4406 [4]. The document defines an executor as "an object responsible for creating execution agents on which work is performed, thus abstracting the (potentially platform-specific) mechanisms for launching work". While N4406 limits the use of executors to parallel algorithms, in HPX this concept is applied wherever the API requires specifying a means of executing work items (see Section 5 for more details).

With a few key exceptions, the API exposed by executors is similar to what is proposed by N4406. Both HPX and N4406 rely on `executor_traits` to make use of executors. This is done so that any implemented executor type only has to expose a subset of the functions provided by `executor_traits` and the rest are generated. Minimally, an executor must implement `async_execute`, which returns a future (see Section 5.1) that represents the result of an asynchronous function invocation. From that instance, the synchronous `execute` call can be synthesized by `executor_traits` or, if implemented, forwarded to the executor. `executor_traits` also provides overloads to `bulk_execute` and `async_bulk_execute` that calls a function multiple times for each element in a shape parameter. In the bulk overload case, N4406 specifies not to keep the results that the function invocations may provide, whereas HPX does return the results. This design decision better supports asynchronous flow control (see Section 5.2).

In addition, HPX extends N4406 by adding the optional possibility for an executor to provide timed scheduling services ('run at a certain point in time' and 'run after a certain time duration'). HPX also optionally exposes the number of underlying compute resources

```
// uses default executor: par
std::vector<double> d = { ... };
parallel::fill(par
  , begin(d), end(d), 0.0);

// rebind par to user-defined executor
my_executor my_exec = ...;
parallel::fill(par.on(my_exec)
  , begin(d), end(d), 0.0);

// rebind par to user-defined executor and
// user defined executor parameters
my_params my_par = ...
parallel::fill(par.on(my_exec).with(my_par)
  , begin(d), end(d), 0.0);
```

Listing 1: Example demonstrating rebinding execution policies to a new executor using `.on()` and a new executor parameters object using `.with()`. Both rebind operations return a new execution policy representing the original one combined with the passed argument.

(cores) the executor uses to schedule work items. The mechanism by which additional executor services are provided in HPX is via additional traits, such as `timed_executor_traits` and `executor_information_traits`.

HPX exposes a multitude of different predefined executor types to the user. In addition to generic sequential and parallel executors which are used as defaults for the `seq` and `par` execution policies, HPX implements executors encapsulating special schedulers, like NUMA-aware schedulers, LIFO or FIFO scheduling policy schedulers, or executors to manage distributed work.

4.3 Executor Parameters

The concept of `executor_parameters` has been added to HPX to allow controlling the grain-size of work, i.e. which/how many work items should be executed by the same execution agent (thread). This is very similar to the OpenMP `static` or `guided` scheduling directives. However, unlike OpenMP scheduling directives that are not a part of the C++ type system, types that belong to the executor parameters concept can make decisions at runtime. As a case of runtime-decision making, the executor parameters concept also allows defining how many processing units (cores) to be used for a particular parallel operation. Like any of the other described concepts, execution parameters can be easily specialized by the user, which allows for a wide variety of – possibly application specific – customizations.

5. TYPES OF PARALLELISM

The various types of parallelism conforming to the listed concepts are exposed through the HPX API and can be understood as layers of abstractions that are implemented on top of each other. Figure 1 gives an overview of this implementation layering. Applications have direct access to all types of parallelism by calling functionality exposed by the shown layers. Moreover, since all concepts implemented by HPX are well defined and documented, an application can provide its own implementation of any particular concept. These application specific types seamlessly integrate with the existing infrastructure and customize the default behavior of the HPX runtime system. Additionally, this design has the advantage of being able to cater to runtime-adaptive implementations of the various concepts. One example for this is a predefined executor parameters type, `auto_chunk_size`, which makes runtime-adaptive decisions about how many iterations of a parallel algorithm to run on the same thread. This opens the door for a wide range of possibly application-specific customizations where the application provides a small amount of code to adapt any of the predefined parallel facilities in HPX.

5.1 Task-based Parallelism

Task-based Parallelism is the lowest level API for orchestrating parallel execution. In HPX, the main method of task-based parallelism is the template function `async` which receives a function and optional parameters. Designed to schedule the parallel execution of tasks of various lengths and times, `async` schedules the given function 'as if on a new thread' (as mandated by the C++ standard) and immediately returns a `future` object [9, 13] which represents the result of the scheduled function. Using `async`, both local and remote asynchronous function invocation can be formed, ensuring syntactic and semantic equivalence of local and remote operations. Other publications have shown that HPX's implementation of `async` enables writing highly performant and scalable applications [14, 16, 20].

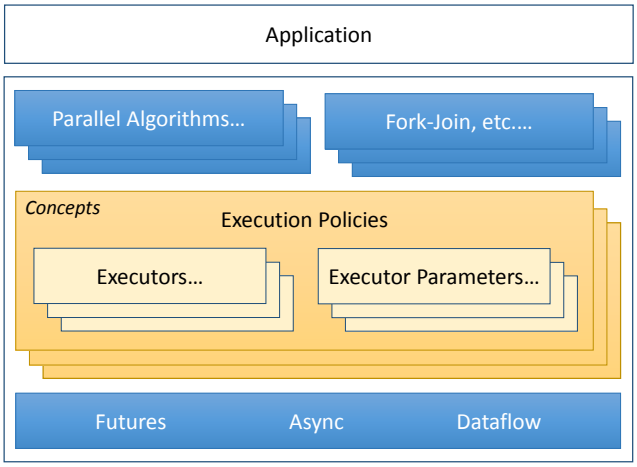

Figure 1: The layer of abstractions of different concepts and types of parallelism in HPX. Each layer relies on the layer below where the lowest layer, 'Futures, Async, Dataflow', relies on functionality exposed from the core HPX runtime modules (not shown). The shown layers are directly accessible to applications and are designed with the possibility of extension.

```
template <typename BiIter, typename Pred>
pair<BiIter, BiIter> gather(BiIter f,
    BiIter l, BiIter p, Pred pred)
{
    BiIter r1 = stable_partition(f, p,
        not1(pred));
    BiIter r2 = stable_partition(p, l,
        pred);
    return make_pair(r1, r2);
}
```

Listing 2: Example showing a conventional implementation of the `gather` algorithm; it is called by passing the begin and end iterators of a sequence `f` and `l`, the insertion point `p`, and the binary predicate `pred` identifying the items to gather at the insertion point. It returns a the range locating the moved items of the original sequence.

The future object returned by `async` naturally establishes an explicit data dependency as any subsequent operations that depend on the result represented by the future object can be attached as an asynchronous continuation, which is guaranteed to be executed only after the future has become ready (see Section 5.2).

5.2 Asynchronous Flow Control

HPX also exposes facilities that allow composing futures sequentially and in parallel. These facilities are aligned with the C++ Concurrency TS [6]. *Sequential composition* is achieved by calling a future's member function `f.then(g)` which attaches a given function `g` to the future object `f`. Here, this member function returns a new future object representing the result of the attached continuation function `g`. The function will be asynchronously invoked whenever the future `f` becomes ready. Sequential composition is the main mechanism for sequentially executing several tasks that can still run in parallel with other tasks. *Parallel composition* is implemented using the utility function `when_all(f1, f2, ...)` which also returns a future object. The returned future object becomes ready whenever all argument futures `f1`, `f2`, etc. have become ready. Parallel composition is the main building block for fork-join style task execution, where several tasks are ex-

ecuted in parallel and all of them must finish running before other tasks can be scheduled. Other facilities complement the API, like `when_any` or `when_some` wait for one or a given number of futures to become ready.

The `dataflow` function combines sequential and parallel composition. It is a special version of `async` which delays the execution of the passed function until after all of the arguments which are futures have become ready. Listing 3 demonstrates the use of `dataflow` to ensure the overall result of `gather_async` will be calculated only after the two partitioning steps have completed.

Any future object in HPX generated by an asynchronous operation, regardless whether this operation is local or remote, is usable with the described facilities. In fact, the future objects returned from local operations are indistinguishable from those returned from remote operations. This way, the described facilities intrinsically support overlapping computation with communication on the API level without requiring additional effort from the developer.

5.3 Loop-based Parallelism

The recently published Parallelism TS [7] is the basis for many of the extensions expected to come to support parallelism in C++. It specifies a comprehensive set of parallel algorithms for inclusion into the C++ standard library. The specified parallel algorithms are similar to the well-known STL algorithms except that the first argument should be an execution policy. HPX implements almost all of the specified algorithms. The HPX parallel algorithms have, however, been extended to support the full spectrum of execution policies (including the asynchronous ones), combined with the full set of executors and executor parameters as described in Section 4. The ability for the application to provide specific implementation of one or more of the parallelism concepts ensures full flexibility and genericity of the provided API. In Section 6.1 we present an example for this by utilizing a custom made NUMA-aware executor for the presented benchmark. Listing 3 gives a demonstration of invoking one of the parallel algorithms, `stable_partition`, with an asynchronous execution policy.

The HPX parallel algorithms will generally execute the loop body (the iterations) at the locality where the corresponding data item is located. For purely local data structures (e.g. `std::vector`) all of the execution is local, for distributed data structures (e.g. `hpx::partitioned_vector`) every particular iteration is executed close to the data item, i.e. on the correct locality. Special executors, based on HPX's distribution policies – a concept abstracting data layout and distribution over different localities – are available and enable more fine control over this mapping process, such as enabling SPMD style execution, where each locality works on the local portion of the distributed data only.

5.4 Fork-join Parallelism

The task-block proposal N4411 [5] specifies new facilities for easy implementation of fork-join parallelism in applications. It proposes the `define_task_block` function which is based on the task_group concept that is a part of the common subset of the PPL [29] and the TBB [18] libraries. N4411 does not, however, integrate the ideas of execution policies and/or executors into fork-join parallelism. Unfortunatelt, this imposes unneeded limitations and inconsistencies.

For this reason, HPX extends the facilities as proposed by N4411 and allows passing an execution policy to the task block. This again includes full support for executors and – where applicable – to executor parameters. For task-blocks, the use of asynchronous execution policies is a powerful addition as it easily allows to integrate those with asynchronous execution flows.

```
template <typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(
    BiIter f, BiIter l, BiIter p, Pred pred)
{
    future<BiIter> f1 = parallel::
        stable_partition(par(task), f, p,
        not1(pred));
    future<BiIter> f2 = parallel::
        stable_partition(par(task), p, l,
        pred);
    return dataflow(
        unwrapped([](BiIter r1, BiIter r2)
            { return make_pair(r1, r2); }),
        f1, f2);
}
```

Listing 3: Example demonstrating asynchronous flow control and execution policies. The `gather_async` algorithm is called by passing the begin and end iterators of a sequence `f` and `l`, the insertion point `p`, and the binary predicate `pred`. It returns a future representing the range locating the moved items of the original sequence.

In conclusion, the full integration of the parallelism concepts described in Section 4 with HPX's API enables an uniform handling of local and remote execution exploiting various forms of parallelism. The resulting API is extensible by the application, fully generic in the sense that it can be applied to any data types, but still ensures best possible performance as shown in Section 6.

6. BENCHMARKS

In this section we present the results of two benchmarks with two aims in mind. First, to demonstrate that the facilities provided by HPX reach the same performance as would be achieved with the well established OpenMP and MPI paradigms. Second, to show that achievements in performance do not come at a hidden cost to implement – namely, that the facilities provided by HPX are in fact practical and easy to use. The first benchmark shown assesses purely local performance on a single node by comparing HPX's parallel algorithm implementations with the results of running the STREAM benchmark [23, 24]. The second presented benchmark shows results from a real mini-application taken from the Intel Parallel Research Kernels ([12, 3], git hash af1d35b088), matrix transposition, which relies on various types of parallelism.

6.1 The STREAM Benchmark

The STREAM benchmark [23, 24] has been widely used for years to assess the memory bandwidth of a computer system. In this section we compare the results measured by running the standard STREAM benchmark (OpenMP version), with results of running an equivalent version developed using the described higher-level parallelization facilities in HPX. The code for the STREAM benchmark ported to HPX is available from the HPX repository ([33], git hash 4ee915c3b6). The results of the TRIAD part of this comparison are shown in Figure 2. The STREAM results were collected on a node from NERSC's Edison cluster [27]. The test machine has two Intel Xeon E5-2695 processors, each with 12 cores clocked at 2.4GHz, and 64GB of DDR3-1866. We have not used the system's

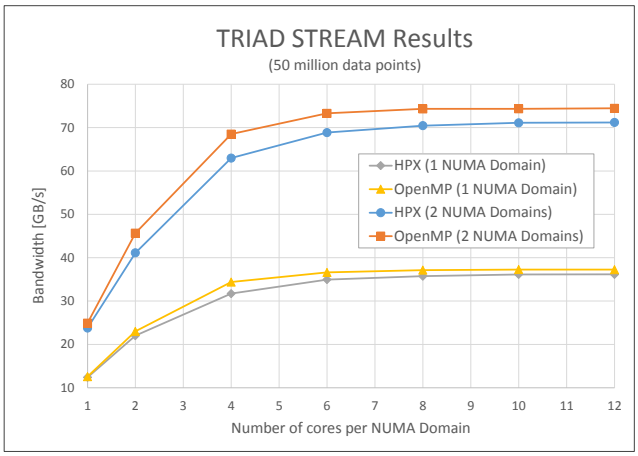

Figure 2: Strong scaling results of running the STREAM benchmark with 50 million data points comparing the original standard version and the version ported to HPX. The HPX version is based on the described higher-level parallelism facilities. The speedup of running on two NUMA-domains over one NUMA-domain at 12 cores is close to a perfect factor of two (1.95 for OpenMP, 1.97 for HPX).

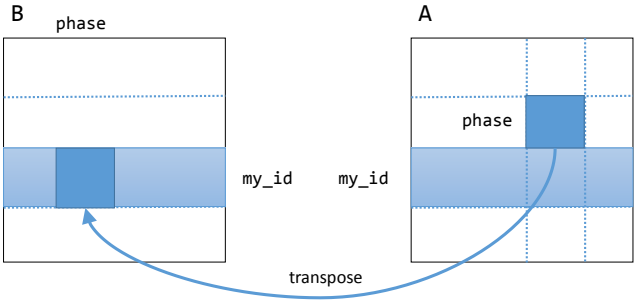

Figure 3: Block-wise matrix transposition. The matrix is (optionally) distributed over several localities, each holding a row-block of the overall matrix (light blue), the current locality is marked as `my_id`. The matrix is transposed by invoking parallel transposition operations for each of the blocks of the corresponding column-block in matrix `A`, while placing the transposed result (dark blue) into the correct position in matrix `B`'s row-block.

hyper-threading capabilities for any of the benchmarks. The shown results are the averaged results of 10 iterations of the benchmark. All code on Edison was compiled using the Intel C++ compiler V15.0.1 20141023.

In order to be able to measure the practically available memory bandwidth of a system, the original STREAM benchmark is written such that each of the created OpenMP threads accesses only data located in a memory bank associated with the NUMA domain the thread is running on. This is achieved by utilizing a NUMA placement policy where after the memory allocation (for instance, using `malloc()`) the thread which first writes to the newly allocated memory determines the locality domain where the physical address mapped behind the allocated virtual address will be placed (first touch policy). If the loops that initialize the data array are parallelized in exactly the same way and use the same access patterns as the loops that use the data later, cross-NUMA domain data transfer can be minimized [35]. A precondition for this first touch initialization to work is that all threads maintain their affinity to the same core throughout their lifetime. All of this reduces the operating system noise and maximizes the data transfer between memory and cores by avoiding cross NUMA-domain traffic.

In the case of OpenMP, the consistency of the access patterns during initialization and the actual measurement is implicitly ensured by using the `#pragma omp parallel static` directive for both loops. No direct, more sophisticated control over the parallel access pattern is available in OpenMP. Also, pinning the threads has to be performed by using special external OS tools (such as `numactl` or `likwid-pin`) as OpenMP itself does not provide any such facilities.

The original STREAM benchmark consists of 4 subsequent parallel loops over 3 separate, equally sized data arrays (`a`, `b`, and `c`): a copy step (`c = a`), a scale step (`b = k * c`), a step which adds two of the arrays (`c = a + b`), and a triad step (`a = b + k * c`). The HPX version of the STREAM benchmark replaces each of the parallel OpenMP loops (`#pragma` directives) with a corresponding parallel algorithm which performs the exact same operations. For this, we have used the `copy`, and the unary and binary `transform` parallel algorithms (see Listing 4).

These algorithms are called with executor objects which are configured to run as many threads as there are cores in a single NUMA-domain on the target machine. Those threads have their affinity defined such that each thread is confined to a separate core. The HPX STREAM benchmark creates as many of such executor objects as there are NUMA-domains on the target system and the algorithms are run in parallel on each NUMA-domain, passing the appropriate executor instances. Each of the algorithm instances works on part of the data arrays. The executor threads work exclusively with data located on the NUMA-domain the thread is running on. For instance, the system used to collect the shown benchmark results has two NUMA-domains, with twelve cores each. This results in two executor objects being created, one for each NUMA-domain. Each executor object manages 12 kernel threads, one for each core, pinned to one of the cores of its NUMA-domain. Each thread is then used to run the algorithms.

For the data arrays for this benchmark we used 3 instances of a `std::vector<double, allocator>`, where `allocator` is a special HPX NUMA-aware allocator type. This allocator performs the data allocation and uses the same executor instances as described above to first touch the allocated memory ranges. This ensures proper memory locality minimizing cross NUMA-domain data transfer during the execution of the parallel algorithms.

The HPX executors applied here customize the execution pattern of the threads which perform the data initialization and benchmark runs. This customization is fully generic and encapsulated; any of the parallel algorithms may be adapted this way. As shown, while the parallelization APIs are fairly high level, it is still possible to fine tune and adapt various parameters of the actual execution to a wide variety of use cases without a significant loss of performance. In this case, a maximum of a 3% loss of performance compared to the relatively low level, inflexible, albeit highly tuned OpenMP solution (see Figure 2).

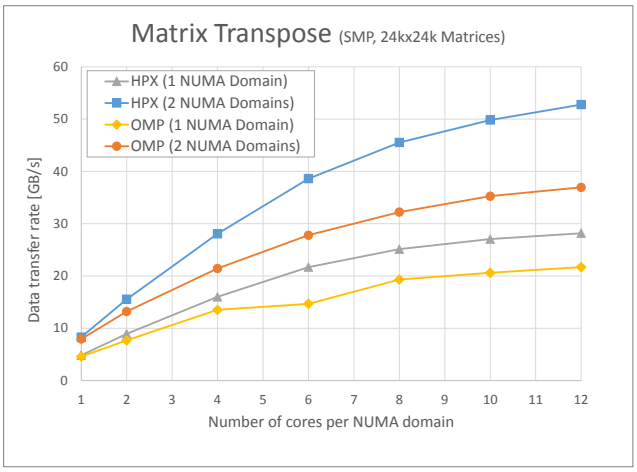

Figure 4: Strong scaling results of running the OpenMP version of the Matrix Transpose benchmark from the Intel Parallel Research Kernels on a SMP system compared with the same code ported to HPX. The HPX version is based on the described higher-level parallelism facilities. Note that the (blue) HPX results are the same as shown in Figure 5.

```
// copy step: c = a
hpx::parallel::copy(policy,
    a_begin, a_end, c_begin);

// scale step: b = k * c
hpx::parallel::transform(policy,
    c_begin, c_end, b_begin,
    [k](double val) { return k * val; });

// add two arrays: c = a + b
hpx::parallel::transform(policy,
    a_begin, a_end,
    b_begin, b_end, c_begin,
    [](double val1, double val2)
        { return val1 + val2; });

// triad step: a = b + k * c
hpx::parallel::transform(policy,
    b_begin, b_end,
    c_begin, c_end, a_begin,
    [k](double val1, double val2)
        { return val1 + k * val2; });
```

Listing 4: The four steps of the STREAM benchmark implemented using the HPX parallel algorithms `copy` and `transform`. Here: `policy` is the execution policy used, e.g., `a_begin` and `a_end` are iterators referring to the begin and the end of the data array `a`.

6.2 Matrix Transposition

In this section we present the results of comparing the performance of one of the applications of the Intel Parallel Reseach Kernels [12] – Matrix Transpose – with a version of this application ported to HPX. We compare three original versions of this kernel, the pure OpenMP, the MPI version (running one MPI rank per core), and the combined MPI + OpenMP implementations. We also show the results of running on different architectures, namely conventional Intel Xeon systems and an Intel Xeon Phi coprocessor. The HPX code used for those comparisons is the same for all three base cases and is available from the HPX repository [33]. The HPX version was written using the higher-level parallelization facilities described above such as asynchronous, continuation based constructs,

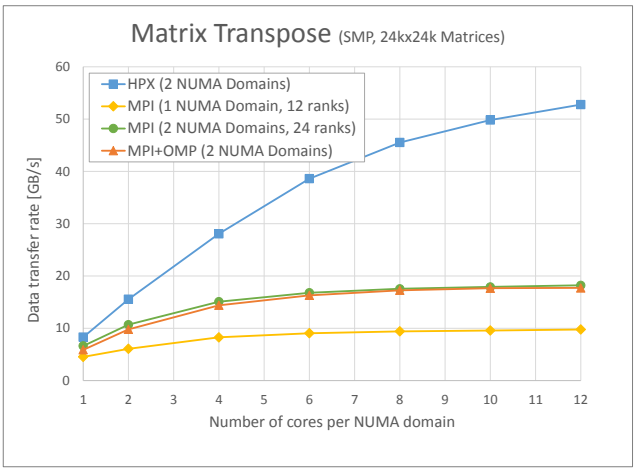

Figure 5: Strong scaling results of running the MPI and MPI + OpenMP versions of the Matrix Transpose benchmark from the Intel Parallel Research Kernels on a SMP system compared with the same code ported to HPX. The HPX version is based on the described higher-level parallelism facilities. Note that the (blue) HPX results are the same as shown in Figure 4.

execution policies and executors, and parallel algorithms.

The results of the measurements on a single SMP system are shown in Figure 4 (comparing HPX with the pure OpenMP version), and Figure 5 (comparing HPX with the MPI and MPI + OpenMP versions). The results collected from runs on the Intel Xeon Phi are shown in Figure 6 and the results for a corresponding distributed application are shown in Figure 7. The SMP Matrix Transpose strong scaling results were collected on the same system as the STREAM benchmark above and additionally on a Intel Xeon Phi 7120P coprocessor (16GB RAM, clocked at 1.238 GHz, 61 core). The weak scaling distributed results were collected on up to 8 nodes from LSU's Hermione cluster. The test machines had each two Intel Xeon E5 2450 processors (each with 8 cores clocked at 2.1GHz), and 48GB of DDR3-1600 memory accessible through 3 independent memory buses. All code on Hermione was compiled using gcc V5.2, and the Intel C++ compiler V15.0.2 20150121 on the Intel Xeon Phi.

Figure 3 depicts the matrix transposition algorithm implemented by the HPX benchmark application. Both matrices (the source `A` and the destination `B`) are sub-divided into blocks (dark-blue in Figure 3), each of this blocks is completely allocated on one of the NUMA-domains. The HPX version uses the same NUMA-aware allocator as described in Section 6.1 which gives a very precise control over where the memory is placed. The OpenMP version relies on a rather coincidental first-touch distribution derived from the parallel access pattern which is used for the data initialization. OpenMP does not expose any more sophisticated means of controlling the memory locality. All of the shown SMP measurements are strong scaling results of transposing a matrix of $24k \times 24k$ elements, using a block-size of 64×64 (384×384 blocks). The size of the blocks was selected such that one block fit into the L1 cache of one of the cores. The shown results are the averaged results of 10 matrix transposition iterations.

The HPX version uses executors for both the first-touch data placement (through the NUMA-aware allocator) and for running the ac-

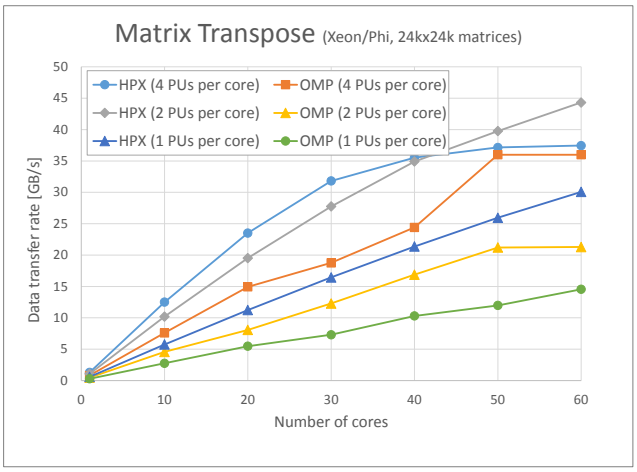

Figure 6: Strong scaling results of running the OpenMP version of the Matrix Transpose benchmark from the Intel Parallel Research Kernels on a Intel Xeon/Phi coprocessor system compared with the same code ported to HPX. The HPX version is based on the described higher-level parallelism facilities.

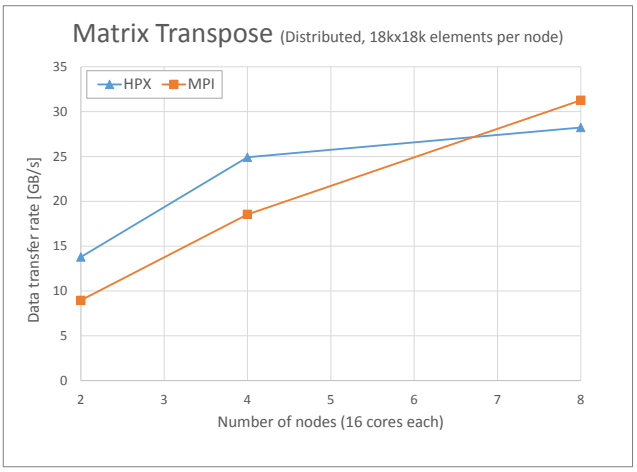

Figure 7: Weak scaling results of running the OpenMP/MPI version of the Matrix Transpose benchmark from the Intel Parallel Research Kernels on a cluster on up to 8 nodes (128 cores) compared with the same code ported to HPX. The HPX version is based on the described higher-level parallelism facilities.

tual transposition operation on a particular data block. Both operations use the same threads for the same data block, thus minimizing cross-NUMA-domain memory traffic, and improving data and cache locality. The HPX version is written such that each block-transposition operation is scheduled to run on the NUMA-domain where the destination block is placed. The source block can be both, local or remote. In fact, our measurements show that the overall ratio of local NUMA-domain memory accesses to remote NUMA-domain memory accesses is close to one.

The results show that the HPX versions significantly outperform all of the original benchmarks run on a single node (pure OpenMP, MPI, and MPI + OpenMP) and on the Intel Xeon Phi. The reasons for this large difference are a) while HPX uses higher-level parallelization facilities, it still allows for much more precise control over the data locality of the matrix blocks, and b) the HPX executor task stealing capabilities allow for better (automatic) load-balancing amongst the cores of every NUMA domain and for automatic overlap of communication (memory traffic) and computation which hides the variances in latencies from accessing memory from the different NUMA domains.

The distributed results show some advantage of using the blocked matrix transposition with HPX for smaller node numbers compared to the original MPI + OpenMP benchmark taken from the Parallel Research Kernels. However, the more nodes that are used for the weak scaling test, the larger the number of generated network messages which impedes the overall performance. More work needs to be done to tune the HPX networking solutions to alleviate this. Though it is clear that the degraded performance is not caused by the higher-level abstractions used to implement the benchmarks.

7. RESULTS

The results of this work demonstrates that the design and development of uniform, versatile, and generic higher-level parallelization abstractions in C++ is not only possible, but also necessary to fully utilize the capabilities of modern computer systems. As shown in HPX's STREAM benchmark, higher-level parallelization facilities can closely match results of conventional fork-join and loop-based parallelization techniques. These abstractions are also capable of outperforming well established programming models by implementing new algorithms which were previously difficult or impossible to implement, as demonstrated by the matrix block transpose test – which used identical HPX code for all benchmarks on all test platforms. This underlines the high portablility in terms of performance *and* code achieved by the described interfaces and their implementations across different architectures. The presented higher-level parallelization abstractions are enabled by using modern C++ facilities on top of a versatile runtime system supporting a fine-grain, task-based scheduling and synchronization. While the same higher-level abstractions show great promise for the distributed use case, the underlying networking implementation requires more future work to match on larger scales the excellent results seen for single node benchmarks.

8. ACKNOWLEDGMENTS

This work was supported by the Bavarian Research Foundation (Bayerische Forschungsstiftung) funding the project AZ-987-11 (ZERPA), by the NSF awards 1240655 (STAR), 1447831 (PXFS), and 1339782 (STORM), and the DoE award DE-SC0008714 (XPRESS).

9. REFERENCES

[1] Boost: a collection of free peer-reviewed portable C++ source libraries, 1998-2015. http://www.boost.org/.

[2] OpenMP V4.0 Specification, 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[3] Intel Parallel research Kernels, 2015. https://github.com/ParRes/Kernels.

[4] N4406: Parallel Algorithms Need Executors. Technical report, , 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4406.pdf.

[5] N4411: Task Block (formerly Task Region) R4. Technical report, , 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf.

[6] N4501: Working Draft: Technical Specification for C++ Extensions for Concurrency. Technical report, , 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4501.html.

[7] N4505: Working Draft: Technical Specification for C++ Extensions for Parallelism. Technical report, , 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4505.pdf.

[8] ASCR. The Challenges of Exascale, 2013. hhttp://science.energy.gov/ascr/research/scidac/exascale-challenges/.

[9] H. C. Baker and C. Hewitt. The incremental garbage collection of processes. In *SIGART Bull.*, pages 55–59, New York, NY, USA, August 1977. ACM.

[10] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[11] L. Dagum and R. Menon. OpenMP: An Industry- Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[12] R. F. V. der Wijngaart and T. G. Mattson. The parallel research kernels. In *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014*, pages 1–6, 2014.

[13] D. P. Friedman and D. S. Wise. CONS Should Not Evaluate its Arguments. In *ICALP*, pages 257–284, 1976.

[14] P. Grubel, H. Kaiser, J. Cook, and A. Serio. The Performance Implication of Task Size for Applications on the HPX Runtime System. In *Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications (HPCMASPA 2015), IEEE Cluster 2015*, September 2015.

[15] T. Heller, H. Kaiser, and K. Iglberger. Application of the ParalleX Execution Model to Stencil-based Problems. In *Proceedings of the International Supercomputing Conference ISC'12, Hamburg, Germany*, 2012.

[16] T. Heller, H. Kaiser, A. Schäfer, and D. Fey. Using HPX and LibGeoDecomp for Scaling HPC Applications on Heterogeneous Supercomputers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '13, pages 1:1–1:8, New York, NY, USA, 2013. ACM.

[17] K. Huck, S. Shende, A. Malony, H. Kaiser, A. jh, R. Fowler, and R. Brightwell. An early prototype of an autonomic performance environment for exascale. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '13, pages 8:1–8:8, New York, NY, USA, 2013. ACM.

[18] Intel. Intel Thread Building Blocks 3.0, 2010. http://www.threadingbuildingblocks.org.

[19] H. Kaiser, M. Brodowicz, and T. Sterling. ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications. In *Parallel Processing Workshops*, pages 394–401, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[20] 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*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[21] H. Kaiser, T. Heller, A. Berge, and B. Adelstein-Lelbach. HPX V0.9.10: A general purpose C++ runtime system for parallel and distributed applications of any scale, 2015. http://github.com/STEllAR-GROUP/hpx.

[22] P. Kogge et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical Report TR-2008-13, University of Notre Dame, Notre Dame, IN, 2008.

[23] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[24] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[25] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. High Performance Computing Center Stuttgart (HLRS), Stuttgart, Germany, September 2009.

[26] Microsoft. Microsoft Parallel Pattern Library, 2010. http://msdn.microsoft.com/en-us/library/dd492418.aspx.

[27] NESRSC. Edison, 2015. https://www.nersc.gov/users/computational-systems/edison/.

[28] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[29] PPL. PPL - Parallel Programming Laboratory, 2011. http://charm.cs.uiuc.edu/.

[30] M. Stumpf. Distributed GPGPU Computing with HPXCL, 2014. Talk at LA-SiGMA TESC Meeting, LSU, Baton Rouge, Louisiana, September 25, 2014, http://stellar.cct.lsu.edu/pubs/martin_stumpf_gputalk.pdf.

[31] The C++ Standards Committee. ISO International Standard ISO/IEC 14882:2011, Programming Language C++. Technical report, Geneva, Switzerland: International Organization for Standardization (ISO)., 2011. http://www.open-std.org/jtc1/sc22/wg21.

[32] The C++ Standards Committee. ISO International Standard ISO/IEC 14882:2014, Programming Language C++. Technical report, Geneva, Switzerland: International Organization for Standardization (ISO)., 2014. http://www.open-std.org/jtc1/sc22/wg21.

[33] The STEllAR Group, Louisiana State University. HPX source code repository, 2007-2015. http://github.com/STEllAR-GROUP/hpx, Available under the Boost Software License (a BSD-style open source license).

[34] Thomas Sterling. ParalleX Execution Model V3.1, 2013. https://www.crest.iu.edu/projects/xpress/_media/public/parallex_v3-1_03182013.doc.

[35] M. Wittmann and G. Hager. Optimizing ccNUMA locality for task-parallel execution under OpenMP and TBB on multicore-based systems. *CoRR*, abs/1101.0093, 2011.