Document number: P0234R0
Date: 2015-04-12
Project: Programming Language C++, SG14, SG1, Evolution WG
Authors: Michael Wong, Hartmut Kaiser, Thomas Heller
Reply to: fraggamuffin@gmail.com, hartmut.kaiser@gmail.com, thomas.heller@cs.fau.de

Towards Massive Parallelism (aka Heterogeneous Devices/Accelerator/GPGPU) support in C++ with HPX

1. Introduction

This paper describes a direction for Massive parallelism design that aims to integrate the many domains for C++, rather than only for a few domains, based on the multiple authors' experience with designing massive parallelism within each of those domains and the existing Parallel and

Concurrency TSs. This is based on ongoing telecons within SG14/SG1 where we are considering the latest practice and proposals for Heterogeneous device support.

The world is moving towards massive parallelism with significant impetus from both ends of the usage spectrum. The High Performance Computation Scientists are driving towards Exascale computing with recent mandates from Department of Energy, while the consumer domain is driving towards it with specific requirements to support heterogeneous devices, drones, medical imaging, and embedded devices used in auto vision relied upon by self-driving cars. In between are the increasingly demanding graphics that is encoded in AAA games, as well as the movement towards VR/AR which require processing of massive amount of data in parallel.

For this reason, 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. At the same time, the efficient utilization of the prospective computer architectures 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.

The goal for a future C++ standard should be to enable the seamless exposure and integration of 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. A central focus should be to ensure full portability - in terms of code and performance - on a wide spectrum of heterogeneous computer architectures in an integrated manner for C++.

2. History

By now in 2016, we already have significant experience with designs for GPU computing, Accelerators, or Heterogeneous Computing Designs. We started off in the 90s with the shader languages such as OpenGL, DirectX, and more recently with more company-specific languages such as CUDA. Then at some point, it was realized that a cross-architecture vendor-neutral support at a low-level area based on C was needed and thus OpenCL was born. Boost.Compute tries to extend this support with a thin C++ wrapper on OpenCL but also enable easy integration with OpenCV and VexCL libraries. OpenCL continues to gain success across a broad domain between HPC and consumer/commercial applications but is uniformly considered as low-level, with a C-centric vision.

After that, the need for high-level languages have become apparent and several consortiums such as OpenMP and OpenACC tried to tackle that with a design that is meant for all forms of heterogeneous devices. In reality, despite multi-year efforts, their adoption has remained mostly in HPC domain with some small forays into oil and gas. But the need for a high-level language seem apparent. It was just difficult to collate into a sufficiently broad coverage and acceptance.. Recent efforts include Heterogeneous Systems Architecture (HSA) lead mostly by AMD and

ARM, and the Microsoft-lead C++AMP. Some of these have gained momentum beyond their originating vendors but never wide enough that it can claim to be pervasive across all domains or aimed specifically at C++.

To round out these designs have been additional attempts aimed at C++. They include Khronos SYCL which adds a C++ layer to OpenCL while adding no additional keywords [1] . Another is LSU's HPX that extends the existing C++ Parallelism and Concurrency TS (the futures part) for massive parallelism, with additional extensions that fits well within C++ while acting as a proof-of-concept for the 2 TSs. HPX comes with a HPC domain view-point but is also appropriate for consumer domain as it is modelled after the existing Parallel and Concurrency TSs. This will be the focus of this paper.

This paper will describe the basis of HPX and use it to gain experience from a heterogeneous device programming model that is designed to work with C++ templates, that has been in production compilers for a consumer domain of a wide variety of devices.

3. Problem Statement and Goal

Current massive parallelism design seems to fit each domain well, or its original design purpose, but none seem to be designed from the ground up to specifically support C++, or to broadly cover as many domains as possible including across HPC and consumer devices.

We also must distinguish between GPU, GPGPU, Accelerator, and Heterogeneous Device computing goals. Some designs were meant for one or two of these arenas, but none seem to be inherent for the broadest possible class of devices.

In other words, massive parallelism support for C++ needs to integrate the learnings from many of the existing designs in the various domains while aiming for a high-level language that can service the broadest possible class of devices which we will call Heterogeneous Devices.

Current C++ directions seem positive towards that goal, but it suffers from the potential to be achieved without a coherent integrated approach.We aim to guide it in a direction that enables a coherent and integrated design.

It is also separated between the concept of Parallelism and Concurrency through two different sets of TSs. What we aim to achieve is leverage ideas from both to achieve a marriage of the two pillars of Parallelism and Concurrency.

It is the goal of this paper to consider one of the design in the HPC domain. Other papers focus on each of their domains. Many of these have already been examined through joint SG1/SG14

committee telecon meetings. We aim to add in additional elements that would be ideal for a future C++ specification.

4. Design Considerations

What is needed by HPC Massive parallelism in C++? OpenMP and OpenACC are the classic HPC designs. Exposing parallelism such as that in the Parallelism TS is not enough for HPC. HPC requires controls over the memory hierarchy, and maximize data locality through the network, disk, RAM, cache and core. OpenMP added thread/task affinity with CPU, as well as improved data locality through first touch, and addressing for High Bandwidth Memory. It further requires reduction of contention through minimizing interaction with false-sharing, locking and synchronization.

What is needed by consumer massive parallelism? TBB, Cilk, TPL, and to some degree OpenCL addresses some of these. Current directions of C++ Std parallelism and concurrency design already serves that direction with Asynchronous tasks (C++11 futures plus C++17 then, when*, is_ready,…), and Parallel Algorithms. It seems certain that in future, there will be some form of Executors, Multi-dim arrays. But what is needed will be support for additional forms such as

- Data: coming in SIMD proposal
- Data Locality: Allocators and Executors
- Task: coming in executors and task blocks
- Embarrassingly parallel: async and threads
- Dataflow: Concurrency TS (.then)

What follows will be an examination of how HPC can serve those goals in an integrated manner.

4.1 Higher-level Parallelism APIs exposed in HPX

Here we present the results of developing higher-level parallelization facilities based on HPX [2], 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 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.

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', etc.)

For each of these properties can be represented and exposed as a C++ type (see Table 1). Existing standardization documents (Parallelism TS [3], Concurrency TS [4]) have already introduced many of those. This experience report lists mainly things which have been introduced in HPX on top of the ongoing standardization efforts with the goal of providing a *uniform* API for various types of parallelism, such as task-based parallelism, asynchronous execution flows, continuation style computation, and explicit fork-join control flow of independent and non-homogeneous code paths.

Note that the types exposing one of the properties listed in the table below can be either one of the predefined types in HPX or a type defined type by the application. This ensures full flexibility, genericity, and extensibility of the parallel facilities implemented by HPX.

Table 1: C++ types representing various parallelization properties

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

4.1.1 Execution Policies

The construct of an `execution_policy` is defined in the Parallelism TS 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` 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 parallel constructs they are used with to return immediately, giving back to the invocation site a future object representing the final outcome of the task. 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 [5] with asynchronous execution flows. 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 block exits. But through task execution policies, asynchronous execution flows and continuation-style programming can be utilized.

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 (similar to N4406 Parallel Algorithms Need Executors [6], see Listing 1 for an example).

4.1.2 Executors

The concept of an executor as implemented in HPX is aligned with what is proposed by N4406. 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.

With a few key exceptions, the API exposed by HPX 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 optionally generated. Minimally, an executor must implement `async_execute`, which returns a future 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.

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 (cores/processing units) 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.

```
// 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: Demonstrate rebinding execution policies with new executors or executor parameters

4.1.3 Executor Parameters

The idea 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 which has proven to be important in HPC workload where sharing or distribution of work by agents gain significant performance. Like any of the other described facilities, execution parameters can be easily specialized by the user, which allows for a wide variety of – possibly application specific – customizations. As an example of how this can be applied, HPX uses this for runtime-decision making, where the executor parameters allows defining how many processing units (cores) to be used for a particular parallel operation depending on various (user defined) runtime parameters.

```
std::vector<double> v = { ... };
parallel::for_each(
      par.with(parallel::static_chunk_size(100)),
      v.begin(), v.end(), [](double) { ... });
```

Listing 2: Demonstrate use of executor parameters to statically combine the given number of iterations into units of parallelized work

```
struct static_chunk_size
{
    explicit static_chunk_size(std::size_t chunk_size)
      : chunk_size_(chunk_size)
    {}

    template <typename Executor, typename F>
    std::size_t get_chunk_size(Executor& exec, F &&, std::size_t num_tasks)
    {
        if (chunk_size_ != 0) return chunk_size_;
        std::size_t const cores = num_processing_units(exec);
        return (num_tasks + cores – 1) / cores;
    }

    std::size_t chunk_size_;
};
```

Listing 3: Example implementation of the `static_chunk_size` parameters object used in Listing 2

4.2 Types of Parallelism

The various types of parallelism 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, 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. OpenMP has a similar auto shedule which enables the runtime to decide. 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.

The various types of parallelism conforming to the ideas listed in the previous section 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.

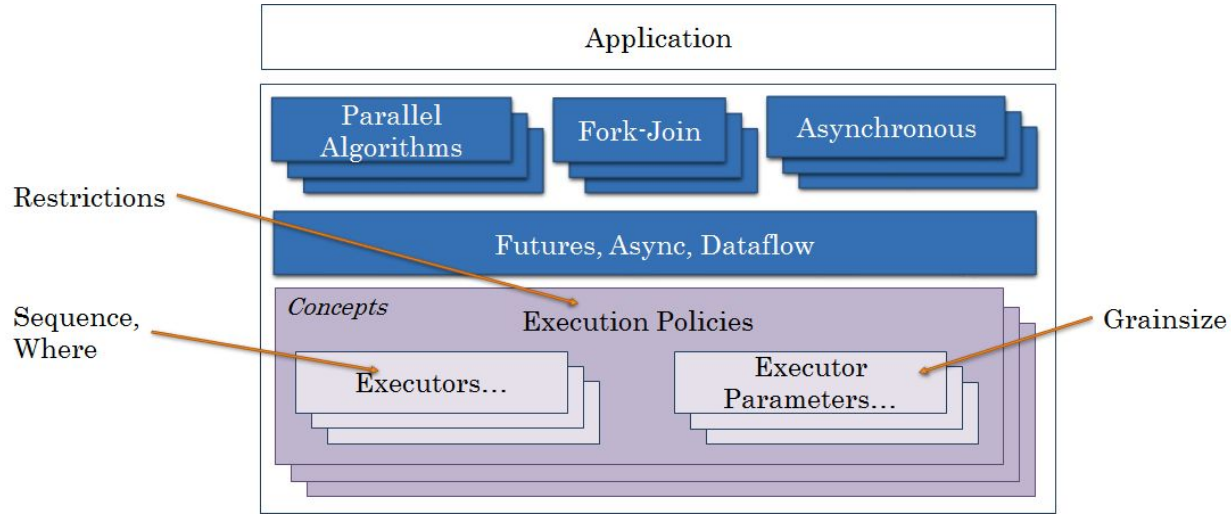

Figure 1: The layer of abstractions of different types of parallelism in HPX. Each layer relies on the layer below where the lowest layer, executors and executor parameters, which rely on functionality exposed from the core HPX runtime modules (not shown). The shown layers are directly accessible to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

4.2.1 Task-based Parallelism

In HPX, the main method of task-based parallelism is the template function `async` which receives a function and optional parameters (very similar to `std::async`). 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 which represents the result of the scheduled function. Using `async`, both local and remote asynchronous function invocation can be performed, ensuring syntactic and semantic equivalence of local and remote operations. Note, that the future returned by `hpx::async` does not block during destruction if it has not become ready yet.

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 Concurrency TS, N4501 [4]).

4.2.2. Asynchronous Flow Control

HPX also exposes facilities that allow composing futures sequentially and in parallel. These facilities are aligned with the C++ Concurrency TS N4501. 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 executed 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. HPX also adds a generalization of `future::wait()`, namely `wait_all()`, `wait_any()`, and `wait_some()`. Those will wait for all, any, or some of the passed 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 5 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.

```
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 4: 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.

```
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(
        [](future<BiIter> r1, future<BiIter> r2)
        {
            return make_pair(r1.get(), r2.get());  // does not suspend!
        },
        f1, f2);
}
```

Listing 5: Example demonstrating asynchronous flow control and execution policies. This code is an asynchronous version of Listing 4. 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.

4.2.3 Loop-based Parallelism

The recently published Parallelism TS 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. 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. Listing 5 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 – 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.

4.2.4 Fork-join Parallelism

The task-block proposal [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 and the TBB libraries. Task Block does not, however, integrate the ideas of execution policies and/or executors into fork-join parallelism. Unfortunately, this imposes unneeded limitations and inconsistencies.

For this reason, HPX extends the facilities as proposed by Task Block and allows passing an execution policy to the task block. This 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 Func>
int traverse(node *n, Func && compute)
{
    int left = 0, right = 0;
    define_task_block(
        par,    // parallel_execution_policy
        [&](auto& tb) {
            if (n->left)
            {
                // use explicitly specified executor to run this task
                tb.run(my_executor(), [&] { left = traverse(n->left, compute); });
            }
            if (n->right)
            {
                // use the executor associated with the par execution policy
                tb.run([&] { right = traverse(n->right, compute); });
            }
        });
    return compute(n) + left + right;
```

```
}
```
Listing 6: Example demonstrating using a execution policy and an executor in conjunction with a task block (see N4411)

4.3 Resumable Functions

In our experience, asynchronous code tends to much better utilize the machine's resources. Replacing synchronous parallel algorithms or task blocks with a corresponding asynchronous version (by using a task based execution policy) usually allows to mitigate the negative effect of the implicit (or explicit) join operation at the end. Often it is possible to execute other (unrelated) tasks before forcing the synchronization with the outcome of the parallel algorithm or task block. In those cases the join operation does not starve the resources as that other work can be executed.

At the same time, writing asynchronous code can become very complex and unwieldy very quickly.

```
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 make_pair(await r1, await r2);  // does not suspend!
}
```
Listing 7: Example demonstrating asynchronous flow control and execution policies. This code is semantically equivalent to the version in Listing 4.

In short, the proposed Resumable Function [7] facilities will blend very well with any of the asynchronous facilities described above. Using `await` truly enables writing asynchronous code in a way very similar to straight synchronous code.

5. Code Examples of C++11, OpenMP and HPX

Having seen how HPX works, here are code examples [8] that compares C++11, OpenMP and HPX.

```
int fibonacci(int n)
{
```

```
    if (n < 2)
        return n;
    if (n < threshold)
        return fibonacci(n-1) + fibonacci(n-2);
    std::future<int> n1 = std::async(fibonacci, n-1);
    std::future<int> n2 = std::async(fibonacci, n-2);

    return n1.get() + n2.get();
}
```

Listing 8. Shows the Fibonacci code using C++11

```
int fibonacci(int n)
{
    if (n < 2)
        return n;
    if (n < threshold)
        return fibonacci(n-1) + fibonacci(n-2);

    int n1, n2;
    #pragma omp task shared(n1)
        n1 = fibonacci(n-1);
    #pragma omp task shared(n2)
        n2 = fibonacci(n-2);
    #pragma omp taskwait
    return n1 + n2;
}
```

Listing 9. Shows the Fibonacci code using OpenMP tasks.

```
hpx::future<int> fibonacci(int n)
{
    if (n < 2)
        return hpx::make_ready_future(n);
    if (n < threshold)
        return hpx::make_ready_future(fib_serial(n));

    hpx::future<int> n1 = hpx::async(fibonacci, n-1);
    hpx::future<int> n2 = fibonacci(n-2);

    return dataflow( hpx::util::unwrapped([](int lhs, int rhs) {
        return lhs + rhs;}, std::move(n1), std::move(n2)
    );
}
```

Listing 10. Shows the Fibonacci code using HPX with the dataflow to do computation scheduled once futures are ready and then unwrapped values are used

```
hpx::future<int> fibonacci(int n)
{
    if (n < 2)
        return hpx::make_ready_future(n);
    if (n < threshold)
        return hpx::make_ready_future(fib_serial(n));

    hpx::future<int> n1 = hpx::async(fibonacci, n-1);
    hpx::future<int> n2 = fibonacci(n-2);

    return await n1 + await n2;
}
```

Listing 11. Shows the Fibonacci code using HPX and `await` to do computation scheduled once futures are ready and then unwrapped values are used. Note that this is semantically equivalent to the code in Listing 10.

6. Conclusions

In conclusion, the full integration of the parallelism concepts 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.

Using `future` objects as the main facility for synchronizing any type of asynchrony is one of the main lessons we have learnt from designing HPX. The `future` facility does not only represent a result which might not have been computed yet, but it also:

- Allows for composition of several asynchronous operations
- Enables transparent synchronization with producer
- Allows to make data dependencies explicit, which in turn enables auto-parallelization
- Makes asynchrony manageable

We hope to work with SG1/SG14 to integrate these features into the heterogeneous device support proposal and make C++ parallelism/concurrency support unified. This will be possible through the addition of the following:

- single and multi-nodes
- execution properties of parallelism
- returning the results of a bulk_execute and async_bulk_execute

- executors encapsulating special schedulers, timed schedulers, like NUMA-aware schedulers, LIFO or FIFO scheduling policy schedulers, or executors to manage distributed work.
- additional types of parallelism including task-based, loop-based, asynchronous, for-join
- Resumable functions

7. References

[1] P0236R0, M. Wong, A. Richards, M. Rovatsou, R. Reyes, Khronos's OpenCL SYCL to support Heterogeneous Devices for C++
[2] HPX, https://github.com/STEllAR-GROUP/hpx
[3] N4507, Technical Specification for C++ Extensions for Parallelism, (ed) J. Hoberock, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507.pdf
[4] N4501, Working Draft, Technical Specification for C++ Extensions for Concurrency, (ed) A. Laksberg, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4501.html
[5] P0015R0, P. Halpern et al, Task Block R5
[6] N4406 Integrating Executors with Parallel Algorithm Execution, J. Hoberock et al, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4406.pdf
[7] N4402, G. Nishanov, Resumable Functions
[8] D. Eachempati, J, Kemp, Futures in OpenMP 5