Using HPX and LibGeoDecomp for Scaling HPC Applications on Heterogeneous Supercomputers

Thomas Heller[1]
thomas.heller@cs.fau.de

Hartmut Kaiser[2]
hkaiser@cct.lsu.edu

Andreas Schäfer[1]
andreas.schaefer@cs.fau.de

Dietmar Fey[1]
dietmar.fey@cs.fau.de

[1]Chair of Computer Science 3, Computer Architectures,
Friedrich-Alexander-University, Erlangen, Germany

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

ABSTRACT

With the general availability of PetaFLOP clusters and the advent of heterogeneous machines equipped with special accelerator cards such as the Xeon Phi[2], computer scientist face the difficult task of improving application scalability beyond what is possible with conventional techniques and programming models today. In addition, the need for highly adaptive runtime algorithms and for applications handling highly inhomogeneous data further impedes our ability to efficiently write code which performs and scales well.

In this paper we present the advantages of using HPX[19, 3, 29], a general purpose parallel runtime system for applications of any scale as a backend for LibGeoDecomp[25] for implementing a three-dimensional N-Body simulation with local interactions. We compare scaling and performance results for this application while using the HPX and MPI backends for LibGeoDecomp. LibGeoDecomp is a Library for Geometric Decomposition codes implementing the idea of a user supplied simulation model, where the library handles the spatial and temporal loops, and the data storage.

The presented results are acquired from various homogeneous and heterogeneous runs including up to 1024 nodes (16384 conventional cores) combined with up to 16 Xeon Phi accelerators (3856 hardware threads) on TACC's Stampede supercomputer[1]. In the configuration using the HPX backend, more than 0.35 PFLOPS have been achieved, which corresponds to a parallel application efficiency of around 79%. Our measurements demonstrate the advantage of using the intrinsically asynchronous and message driven programming model exposed by HPX which enables better latency hiding, fine to medium grain parallelism, and constraint based synchronization. HPX's uniform programming model simplifies writing highly parallel code for heterogeneous resources.

Keywords

High Performance Computing, Parallel Runtime Systems, Application Frameworks

ScalA '13 November 17-21, 2013, Denver CO, USA

ACM 978-1-4503-2508-0/13/11...$15.00.
http://dx.doi.org/10.1145/2530268.2530269.

1. INTRODUCTION

Due to the scale of today's supercomputers, users must be able to exploit multiple levels of parallelism if they hope to achieve decent performance on today's machines. In addition to the theoretical scaling limits described by Amdahl's Law[5] and Gustafson's Law[15] at least four additional factors limit application scalability, also referred to as the **SLOW** factors: a) ***S**tarvation*, i.e. current concurrent work is insufficient to maintain high utilization of all resources, b) ***L**atencies*, i.e. the delay intrinsic to accessing remote resources and services deferring their responses, c) ***O**verheads*, i.e. the work required for the management of parallel actions and resources on the critical execution path which is not necessary in a sequential variant, and d) ***W**aiting for contention resolution*, which is caused by the delays due to oversubscribed shared resources.

We posit that in order to achieve the goal of making even highly dynamic applications scalable, a new programming model is required. This programming model will need to overcome the limitations of how applications are written today and make the full parallelization capabilities of today's and tomorrow's heterogeneous hardware available to the application programmer in an simple and uniform way. The work presented in this paper is based on HPX - a runtime system implementing such a programming model. It is described in detail in Section 2. HPX is based on the set of governing principles of the ParalleX execution model [19, 17, 27] to enable a maximum of application level parallelism, while minimizing the effect of the **SLOW** factors.

In order to efficiently use the proposed programming model today, already existing application frameworks need to be ported to HPX such that those frameworks can benefit from the advanced levels of parallelism provided. As an example, this paper describes the results from porting the parallel simulation backend of LibGeoDecomp (see Section 3) to utilize HPX (see Section 5). Due to the highly modular and careful design of LibGeoDecomp we were able to develop the backend such that the user's simulation code doesn't need to be changed. Nevertheless, the new parallelism provided by the HPX backend is fully utilized. Due to the uniform programming model, we are able to present numbers that outscale and outperform the already existing MPI backend on heterogeneous architectures by a significant margin while maintaining the high productive programmability of LibGeoDecomp (see Section 6).

This paper presents the results obtained from large scale runs on TACC's Stampede resource[1] using LibGeoDecomp's HPX backend, comparing those to equivalent runs performed with LibGeoDecomp's MPI backend. In order to evaluate the achieved performance we used a N-Body application written in LibGeoDecomp (see Section 4). It highlights the capabilities of Stampede's hetero-

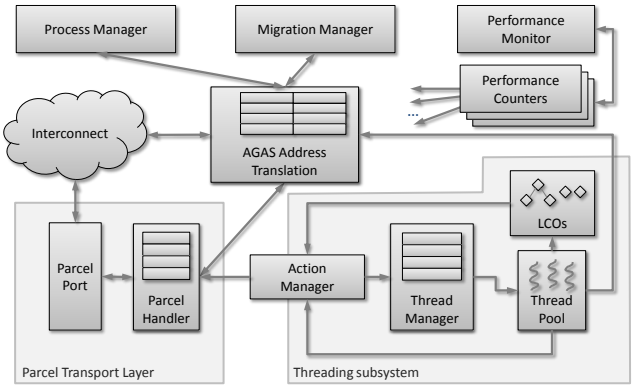

Figure 1: Architecture of the HPX runtime system. An incoming parcel (delivered over the interconnect) is received by the parcel port and dispatched to the parcel handler. The main task of the parcel handler is to buffer incoming parcels for the action manager. The action manager decodes the parcel and creates an HPX-thread based on the encoded information. All HPX-threads are managed by the thread manager, which schedules their execution on one of the cores. Usually HPX creates one OS-thread for each available core. The thread manager implements several scheduling policies, such as a global queue scheduler, where all cores pull their work from a single, global queue, or a local priority scheduler, where each core pulls its work from a separate priority queue. The latter supports work stealing for better load balancing. Local Control Objects (LCOs) are responsible for synchronizing access to shared resources and are tightly integrated with the thread scheduling to resume threads whenever all preconditions for continuing execution of a thread are met.

geneous architecture by fully utilizing its Xeon Phi accelerators [2] combined with all cores of the used host nodes.

2. HPX: A GENERAL PURPOSE PARALLEL RUNTIME SYSTEM

HPX is a general purpose parallel runtime system exposing a uniform programming model for applications of any scale. It has been developed for conventional architectures, such as SMP nodes, large Non Uniform Memory Access (NUMA) machines and clusters, and heterogeneous systems such as using Xeon Phi accelerators. Strict adherence to Standard C++11 [28] and the utilization of the Boost C++ Libraries [4] makes HPX both portable and highly optimized. It is modular, feature-complete and designed for best possible performance. HPX's design focuses on overcoming conventional limitations such as (implicit and explicit) global barriers, poor latency hiding, static-only resource allocation, and lack of support for medium- to fine-grain parallelism (see Figure 1).

The Active Global Address Space (AGAS): In HPX, AGAS currently is a set of (distributed) services that implement a 128-bit global address space spanning all localities. Those provide two naming layers in HPX. The primary naming service maps 128-bit unique, global identifiers (GIDs) to a tuple of meta-data that can be used to locate an object on a particular locality. The higher-level layer maps hierarchical symbolic names to GIDs. Unlike systems such as X10 [9], Chapel [8], or UPC [30], which are based on PGAS [24], AGAS exposes a dynamic, adaptive address space which evolves over the lifetime of an HPX application. When a globally named object is migrated, the AGAS mapping is updated, however its GID remains the same. This decouples references to those objects from the locality that they are located on.

Parcel Transport Layer: HPX parcels are a form of active messages [32] used for communication between localities. In HPX, parcels encapsulate remote method calls. A parcel contains the global name of an object to act on, a reference to one of the object's methods and the arguments to call the method with. Parcels are used to either migrate work to data by invoking a method on a remote entity, or to bring pieces of data back to the calling locality. Currently, HPX implements parcel communication over TCP/IP, Infiniband, shared memory (for communication between localities running on the same physical resource), and on top of low level MPI functionality (MPI_Isend/MPI_Irecv). The MPI parcelport is used mainly for enabling a smooth transition of existing applications and to ensure HPX can be run on any platforms with an existing MPI transport layer implementation. Each locality has a parcel port which reacts to inbound messages and asynchronously transmits outbound messages. After a parcel port receives and deserializes a message, it passes the parcel to a parcel handler. If the target object of a parcel is local, then the action manager converts the parcel into a HPX-thread, which is scheduled by the HPX thread-manager.

HPX-threads and their management: The HPX thread-manager is responsible for the creation, scheduling, execution and destruction of HPX-threads. In HPX, threading uses an *M:N* or hybrid threading model. In this model, *N* HPX-threads are mapped onto *M* kernel threads (OS-threads), usually one OS-thread per core. This threading model was chosen to enable fine-grained parallelization; HPX-threads can be scheduled without a kernel call, reducing the overhead of their execution and suspension. The thread-manager uses a work-queue based execution strategy with work stealing similar to systems such as Cilk++ [20], Threading Building Blocks (TBB [18]), or the Parallel Patterns Library (PPL [21]). HPX-threads are scheduled cooperatively, that is, they are not preempted by the thread-manager. HPX-threads may voluntarily suspend themselves when they must wait for data that they require to continue execution, I/O operations or synchronization.

Local Control Objects (LCOs): LCOs provide a means of controlling parallelization in HPX. Any object that may create a new HPX-thread or reactivate a suspended HPX-thread exposes the required functionality of an LCO. Support for event-driven HPX-thread creation, protection of shared data structures, and organization of flow control are provided by LCOs. They are designed to allow for HPX-threads to proceed in its execution as far as possible without waiting for a particular blocking operation, such as a data dependency or I/O, to finish. Some of the more prominent LCOs provided by HPX are:

1. *Futures* [7, 14, 16] represent results that are not yet known, possibly because they have not yet been computed. A future synchronizes access to the result value associated with it by suspending HPX-threads requesting the value if the value is not available at the time of the request. When the result becomes available, the future resumes all suspended HPX-threads waiting for the value. These semantics allow execution to proceed unblocked until the actual value is required for computation.
2. *Dataflow objects* [11, 12, 6] provide a powerful mechanism for managing data dependencies without the use of global barriers. A dataflow LCO ensures that a predefined function will be called once a set of values become available. The function is called passing along all of this data.
3. *Traditional concurrency control mechanisms* including various types of mutexes [10], counting semaphores, spinlocks, condition variables and barriers are also exposed as LCOs in HPX. These constructs can be used to cooperatively suspend an HPX-thread while informing the HPX thread-manager that other HPX-threads can be scheduled on the OS-thread.

LCOs are first class objects in HPX, they enable intrinsic overlapping of computation and communication. This not only hides latencies, but also allows many phases of a computation to overlap, exposing greater application parallelism. They can be used to control parallelism across multiple localities. The mechanisms for naming and referencing first class objects such as LCOs is provided by AGAS.

3. LIBGEODECOMP – AN AUTO-PARALLELIZING LIBRARY

The purpose of LibGeoDecomp[25] is to simplify the development of computer simulations. Typical challenges for such codes are the adaptation to new hardware architectures and the scalability on large-scale systems. Simulation models are generally developed by domain scientists, e.g. physicists or material scientists. Their productivity will be greatly increased if they can be relieved from having to worry about the machine architecture.

The basic abstraction within LibGeoDecomp is the simulation cell. Cells are placed in a regular grid and updated once per timestep. During the update they may access their neighbors from the last time step. In other words, in LibGeoDecomp simulations are written as iterative algorithms with spatial discretization. Examples for such models are cellular automata or Lattice Boltzmann Methods. Other models, such as N-body codes, which cannot be directly represented by a regular grid are handled by wrapping the particles into boxes according to their spatial location. The containers then form a regular grid. This procedure works well if the particles are evenly distributed, but efficiency is poor if pronounced hotspots are present. Only local interactions can be represented.

The library is written as a set of C++ class templates. User code describes the behavior and the data stored in a single simulation cell. It is inserted into the library as a template parameter. Interaction of model and library is defined by a two-way callback interface: the library calls a cell to update its state and the cell may call back the library to retrieve the states of itself and its neighbors from the last time step by means of a proxy object – the so called neighborhood.

Within the library the objects which maintain the workflow of the simulation are named `Simulators`. These implement various optimizations such as multi-node and multi-core parallelization, overlapping communication and calculation, parallel IO, etc. The library has support for in-situ visualization and live steering (see Fig. 2).

The key advantage of this approach is that user code and parallelization are segregated. User code may benefit from improvements of the parallelization without the need of modifications. Single investments into the library benefit multiple applications.

4. SIMULATION MODEL

For our evaluation we chose the N-body model presented in [31]. This model represents a larger class of similar models (e.g. gravitating bodies or electrostatically charged particles) and allows comparison of our results with previous publications. The only modification we applied was to introduce a cut-off ratio to the force calculation. This slightly decreases the computational complexity (i.e. it increases the model's demand for memory bandwidth).

In essence, the model is an N-body simulation with short range interaction, which can be described by the following equations:

$$F_i = K \cdot C_i \sum_{j \in H_i} C_j \frac{R_j - R_i}{\|R_j - R_i\| + s}$$

$$H_i = \{j \in \mathbb{N} \,|\, \|R_j - R_i\| \leq D\}$$

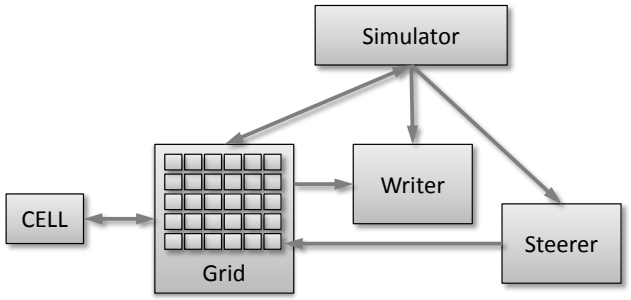

Figure 2: High-level architecture of LibGeoDecomp: the `Simulator` controls the workflow. Data is stored within the `Grid`, whose elements, the `Cells`, are provided by the user. The `Cell` also contains information on the topology and boundary conditions required by the model. A `Writer` performs periodic output, e.g. by dumping the grid in VisIt's BOV format to disk. Conversely, a `Steerer` may modify the grid at runtime. This can be leveraged quickly patch errors within the simulation or to conduct experiments with low turnaround times.

F_i is the force acting on body i, R_i is its location. The force is defined by those particles which interact with the current particle. As said, we do not take into account all particles, but just those within a certain radius D. These particles are found in set H_i. Particles beyond the cutoff are assumed to have only a negligible influence. The direction and magnitude of the forces further depends on the factors C_i which – depending on the physical model – may represent mass or charge of the given body, and the model-dependent constant K. The parameter s is often referred to as a *softening factor* and has the purpose to avoid a division by zero if the interaction of the particle with itself is being calculated – or if two particles should accidentally occupy the same position.

Crucial to an efficient implementation of this model are two aspects: the calculations need to be vectorized and the calculation of the sets H_i should not incur any overhead. The latter can be achieved by placing all particles in parallelepipeds of size D, as shown in Figure 3. These serve as containers. Each particle can only interact with particles from its own container or from those surrounding it. This is a standard technique and has been used in other scalable implementations before (e.g. [13]). As the particles move through space, they may need to switch containers. Checking for such transitions may be a time consuming test, but by choosing the container size slightly too large we can defer such tests, thereby rendering the overhead caused by this transition negligible.

Vectorization requires that particles are not stored as distinct objects, but rather in a *Struct of Arrays* [26] fashion. This means that each member variable (e.g. the three scalars that make up its position) is stored as a vector for all particles within a container. Thanks to the softening factor, our model does not need to avoid self-interaction of the particles. This simplifies the vectorization of the loop. In fact, the pseudocode below can be vectorized in two ways: we can either traverse the particles in `cell.particles` in the innermost loop in a vectorized fashion or traverse the cells `particles` in a scalar way and compute the interactions with multiple `p_1` from `this->particles`. In the first case we would compute all acting forces for exactly one particle `p_1` and all particles `p_2` from `cell`. The latter strategy requires the architecture to perform a scalar load (for retrieving `p_2`) and to broadcast that scalar value to a vector register in an efficient fashion.

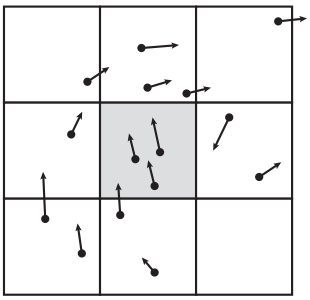

Figure 3: Illustration of simulation model. Particles are placed in a container according to their physical location. All particles of the shaded cell may interact with the particles of their own cell and those cells neighboring it.

```
void addForce(Particle& p_1, const Particle
    & p_2)
{
  vec3 delta = p_2.pos - p_1.pos;
  p_1.force += p_2.c * delta / (norm(delta)
       + s);
}

for(Particle p_1: this->particles) {
  for(Cell cell: neighborhood) {
    for(Particle p_2: cell.particles) {
      add_force(p_1, p_2);
    }
  }
}
```

5. IMPLEMENTATION

This section details the port of LibGeoDecomp to HPX and the implemented mechanisms that allow the efficient use of heterogeneous resources such as the Stampede supercomputer. As described in Section 3, LibGeoDecomp consists of several high-level components. The HPX backend to LibGeoDecomp merely touches the Simulator module without the need to change the already existing API and semantics of existing backends, such as the MPI backend. As a matter of fact, the HPX backend is closely modeled after the MPI implementation to allow the reuse of the already existing infrastructure for domain decomposition and synchronization.

5.1 Overview

The simulator features a layered design, which separates the domain decomposition from the synchronization of the node domains and the intra-node threading. The design needs to satisfy two goals: reduce the impact of network latency and equalize the load on CPUs and accelerators. Latency can be hidden by overlapping communication and calculation. This in turn requires a low-overhead communication infrastructure and the ability to make asynchronous progress. Load equalization mandates a tunable domain decomposition scheme. In LibGeoDecomp a domain decomposition (or partition) is a function which maps the nodes to sets of coordinates. These coordinate sets represent the domains of the nodes. Their sizes can be tuned via a weight vector. The design can be summarized by the following modules:

`Simulator`: The `Simulator` acts as the main interface to the user. It acts as the glue code to set up all the following classes. It will set up as many `UpdateGroups` as requested at runtime.

`UpdateGroup`: An `UpdateGroup` represents the entity to create the simulation `Stepper` and the neighborhood communications defined by the `PatchAcceptor` and `PatchProvider`. Additionally, the partitioning is done within an `UpdateGroup` which is implemented in a `PartitionManager`.

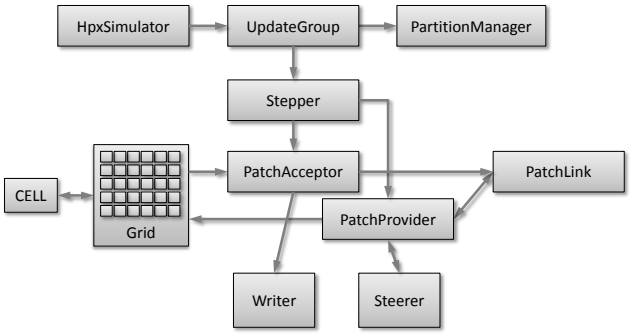

Figure 4: Main objects and their interactions when using LibGeoDecomp's HPX backend. The idea is to reduce complexity by decomposing the codebase into specialized classes. The `Simulator` manages the workflow of the simulation. It will contain one `UpdateGroup` per NUMA. Each `UpdateGroup` is responsible for a sub-domain of the grid. Which domain exactly is determined by the `PartitionManager`. The `Stepper` contains the temporal and spatial loops. It will call back the user code within the `CELL` class. Halos are synchronized by the `PatchLinks`. A `Writer` is an output plugin, while a `Steerer` can be used for live-steering.

`PartitionManager`: As the discrete domain of computation needs to be decomposed, or partitioned, in order to be parallelized efficiently, a `PartitionManager` is needed to implement various partitioning strategies and determine the simulation domain as well as the ghostzones of a certain `UpdateGroup`.

`Stepper`: The `Stepper` class represents the main simulation control flow implementation. The `Cell`'s update function as well as the ghost zone exchange as provided by the `PatchAcceptors` and `PatchProviders` is done here. This class is mainly responsible for the scalability of LibGeoDecomp and is the class in which the parallelization has to happen. A more detailed discussion of the algorithm implemented and the parallelization strategy can be found in Subsection 5.2.

`PatchAcceptor`: A `PatchAcceptor` provides an abstraction for the `Stepper` which is used to retrieve the state of the Grid in the current time step. It used to either notify a `Writer` to write the grid elements (see Section 3) or to update it's neighboring `UpdateGroups` ghost zones through a `PatchLink`. The setting of a neighboring ghost zone can be completely behind the computation through the `Stepper`.

`PatchProvider`: Similar to the `PatchAcceptor`, the `PatchProvider` is providing an abstraction to set the portions of the Grid at the current timestep. It is used to either notify a `Steerer` to set a new state of the current's grid elements (see Section 3) or to set the ghost zone retrieved from a neighboring `UpdateGroup` through a `PatchLink`. This is the only place in LibGeoDecomp's parallelization backend where a synchronization between the different `UpdateGroups` happens as it is guaranteed that the timesteps match between `UpdateGroups`.

As noted above, both the Scalable MPI backend (implemented in the class `HiParSimulator`) and the HPX backend (implemented in the class `HpxSimulator`) make use of this generic structure while maintaining as much API compatibility as possible. The only notable difference in the interface lies within the creation of an `Simulator` object. Where with the `HiParSimulator`,

the number of UpdateGroups are equal to the number MPI Ranks created, the HPXSimulator allows the user to decide how many UpdateGroups are created per node by inputting an additional parameter. The details and benefits to this approach are discussed in the following subsection.

5.2 Parallelization and scalability considerations

As discussed previously, the heavy lifting of the parallelization efforts of LibGeoDecomp lie within the responsibilities of the UpdateGroup, Stepper, and PatchLink.
An UpdateGroup creates a partition for itself based on the selected partitioning scheme, which not only determines which partition belongs to the current UpdateGroup in question, but also determined which UpdateGroups have corresponding neighboring regions. The size of a partition is determined by an initial weight vector, which might consist of equal weights for homogeneous runs, or consist of different weights, for heterogeneous runs. This weight factor is determined by the user and defined in the Cell implementation, which takes different computation speeds of the various processing units involved in the simulation into account.
In addition, two PatchLinks are created for each corresponding neighboring Region; They are responsible for (a) receiving a ghostzone fragment (handled by a PatchProvider) or (b) sending a ghostzone fragment (handled by a PatchAcceptor). As described in the previous subsection, the Patchlink of a PatchProvider, is the only place where a single UpdateGroup is synchronized with its neighbors. It is important to note that no collective operations are used here; therefore, the implementation is scalable by design. The HpxSimulator and HiParSimulator share this important design decision, however, the implementation specifics are worthy to note as they highlight the advantages of the HPX programming model over MPI. While within the HiParSimulator one UpdateGroup per rank is created and the communication within the PatchLink is implemented via the two-sided MPI asynchronous communication primitives, the HPX Backend is able to leverage the AGAS (see Sec. 2). It creates a varying number of UpdateGroup components per node based on the compute requirements of a specific node. Additionally, the PatchLinks do not need to communicate via low level primitives such as MPI_Isend() and MPI_Irecv() but can rely on invoking a (possibly remote) set function of the neighboring UpdateGroup component taking full advantage of the unified program model provided by HPX. This mechanism is not only implemented in a truly Object Oriented fashion, but it is also inherently asynchronous. By moving the UpdateGroups into the AGAS, and thereby only needing one HPX locality per node, we gain considerable advantages over the MPI programming model which requires one MPI process per CPU core. This avoids, possibly expensive, inter-process communications which leads to an increased scalability on a single node.

The techniques described above amount to a complete port of LibGeoDecomp to HPX. However, only a small portion of the HPX parallel runtime system is used. To fully exploit the potentials of the emerging technology, the Stepper class will need to be ported to HPX as well. However, for now, the Stepper used by MPI backend can be used without any further modifications. The conventional VanillaStepper is outlined in Fig. 5: For each timestep, the inner region is updated, once done, we notify the PatchAccepters in order to retrieve the new ghostzones. Afterwards, our inner ghostzone can be updated and, once finished, sent to the neighboring UpdateGroups.

```
for(Region r: innerRegion) {
  update(r, oldGrid, newGrid, step);
}
swap(oldGrid, newGrid);
++step;
for(Region r: outerGhostZoneRegion) {
  notifyPatchProviders(r, oldGrid);
}

for(Region r: outerGhostZoneRegion) {
  update(r, oldGrid, newGrid, step);
}
for(Region r: innerGhostZoneRegion) {
  notifyPatchAccepters(r, oldGrid);
}
```

Figure 5: `VanillaStepper`: Algorithm sketch of a Stepper for LibGeoDecomp. This is a basic outline for how the simulation stepping inside LibGeoDecomp works. The implementation is fully serial.

Due to the serial nature of a single MPI process, the outlined algorithm is as good as it can get. However, the advanced parallelization techniques provided by HPX open the doors to further improve and take full advantage of the parallel capabilities of a single CPU. The described code can be fully futurized. Futurization is a technique which allows users to turn otherwise serial code into a chain of asynchronously executed functions. The serial control flow is transformed into a sequence of depending continuations to previous calculations. A simple loop without dependencies can be simply formulated as a loop where every loop body is executed in parallel. A possibly depending calculation can simply be chained by passing a continuation function which will be executed whenever every chunk of the loop has finished (as described in [22] and [23]). This will lead us to the futurized version of the VanillaStepper, the HpxStepper (see Fig. 6). The algorithm works in the same way as the one presented in Fig. 5. The distinctions between the two codes are that different independent regions are computed and notification of the PatchAccepters and PatchProviders are performed in parallel. In addition we break up each step into a sequence of continuations. This results in taking a very coarse grained function and reducing it into multiple fine grained functions whose parts are executed in parallel.

By applying the described techniques, we gain a powerful backend which is able to make use of all parallel resources. On the node level, we benefit from futurization and the ability to have only one process per node. When running the application in distributed, we are profiting from the unified programming model which gives us increased asynchronity which in turn leads to better latency hiding by being able to properly hide communications behind useful computation. All this was achieved while being 100% API compatible for existing LibGeoDecomp applications, which means they can immediately benefit from the HPX backend.

6. BENCHMARKS

In order to evaluate our developed approaches, and test the scalability of our the newly developed library we used the simulation model as described in Section 4. The computing resource used is TACC's Stampede [1]. It consist of a total of 6400 nodes with two Intel Xeon E5 processors and one Intel Xeon Phi coprocessor (see Table 1). The compute nodes are interconnected with Mellanox FDR InfiniBand technology (56 Gb/s) in a 2-level fat-tree topology. The complete system is a 10 PFLOPS cluster. In our weak scaling experiments, we scale the problem size with the number of cores. For each core we assigned 1000 grid elements. The dis-

```
typedef vector<hpx::future<void>>
    future_vector;
future_vector updateFutures;
for(Region r: innerRegion) {
  updateFutures.push_back(
    hpx::async(update, r, oldGrid, newGrid,
        step));
}
hpx::when_all(updateFutures,
[](const future_vector&) {
  swap(oldGrid, newGrid);
  ++step;
  future_vector notifyFutures;
  for(Region r: outerGhostZoneRegion) {
    notifyFutures.push_back(
      hpx::async(notifyPatchProviders, r,
          oldGrid));
  }
  hpx::when_all(notifyFutures,
  [](const future_vector&) {
    future_vector updateFutures;
    for(Region r: outerGhostZoneRegion) {
      updateFutures.push_back(
        hpx::async(update, r, oldGrid,
            newGrid, step));
    }
    hpx::when_all(updateFutures,
    [](const future_vector&) {
      future_vector notifyFutures;
      for(Region r: innerGhostZoneRegion) {
        notifyFutures.push_back(
          hpx::async(notifyPatchAccepters,
              r, oldGrid));
      }
    });
  });
});
```

Figure 6: Futurized `VanillaStepper`, the `HpxStepper`. Due to the Advanced parallelization strategies provided by HPX, this Stepper implementation is now fully parallel and scalable due to it's highly asynchronous nature.

tributed HPX applications use the MPI parcelport since it currently provides the best performance.

As a first measure we conduct weak scaling tests on a single node. The theoretical peak performance of the two Xeon E5 processors is 691.2 GFLOPS, or 896 GFLOPS with the Intel Turbo Boost technology. This performance can be reached by simultaneously scheduling 8 multiply and 8 addition operations per cycle. Our computational kernel introduced in Section 4 has a ratio of multiply/add to other floating instructions of 4:6, as such the theoretical peak performance of our algorithm is 552.9 GFLOPS on the Xeon E5 processor (assuming the CPU is almost always running in its turbo mode).

Fig. 7 shows the results obtained from running on a single node. While the MPI backend needs to perform inter-process communication in order to exchange the ghostzones, the HPX is able to overcome this limitation by efficiently parallelizing the simulation steps (see Section 5.2). We are able to reach ∼98% peak performance with the `HpxSimulator` while MPI backend is only able to achieve ∼63%. The sub-optimal performance of the MPI backend is mainly due to the extra overheads introduced by inter-process communication and the need to update the ghostzones separately. Even though one would think that using high level parallelization APIs, like the C++ API exposed by HPX and high level application frameworks like LibGeoDecomp, would create a significant overhead, the benchmark shows that they impose an overhead of only ∼8%.

	Intel Xeon E5	Intel Xeon Phi
Clock Frequency	2.7 (3.5 Turbo) GHz	1.1 GHz
Number of Cores	16 (2x8)	61
SMT	2-way (deactivated)	4-way
NUMA Domains	2	1
RAM	32 GB	8 GB
SIMD	AVX (256 bit)	MIC (512 bit)
GFLOPS	691.2 (896.0 Turbo)	2147.2
Microarchitecture	Sandy Bridge	Knights Corner

Table 1: Overview of the processors built into one compute node of the Stampede supercomputer. GLFOPS are presented in single precision

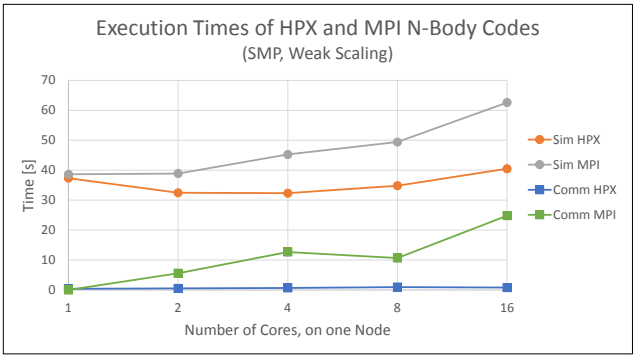

Figure 7: Weak scaling execution times for the HPX and MPI N-Body codes collected on a single SMP node while varying the number of cores used from 1 to 16. The figure shows the times for the overall *Sim* and the overheads introduced by *Comm*. Additionally demonstrating the benefit of the futurization technique, as no extra communication is needed for the HPX backend while the MPI backend needs to perform extra actions for the halo exchange. By that, the HPX backend only has 8% parallel overhead, while the MPI backend shows ∼27% overhead. Which accounts for a sustained performance of ∼546 GFLOPS using the HPX backend.

In addition to the scaling experiments on the host CPU, we are interested in how well our code works on the Intel Xeon Phi architecture. Due to the lack of OpenMP support in LibGeoDecomp, we only ran experiments with the HPX backend. To use the Xeon Phi, we compiled native binaries for the coprocessor. Similar to the host CPU, the Xeon Phi coprocessor is able to schedule one, so called, fused multiply add (fma) instruction, which is able to perform 16 single precision floating point operations in a single cycle. As such, our algorithm has a theoretical peak of 1717.7 GLFOPS. We are able to sustain the same performance characteristics we see on the host processor when running on the Xeon Phi (see Fig. 8). The futurization approach is able to scale even on this architecture. However, due to the architectural specifics of its in-order nature, the usage of more than 2 cores per thread only leads to a minimal advantage. Overall the HPX backend is able to achieve 89% of the peak performance on a single Xeon Phi. Achieving an overhead of only 11% is remarkable as no further modifications to the runtime system or the application framework were necessary besides the intrinsics that were used in the computational kernel.

To further test the scalability of our approach, we ran our benchmark on multiple nodes. We were able to scale up to 1024 nodes, using 16384 Xeon E5 cores. Furthermore, heterogeneous runs using up to 16 nodes utilizing both, the Xeon E5 host CPUs and one Xeon Phi coprocessor on each node, using a total of 4160 hardware

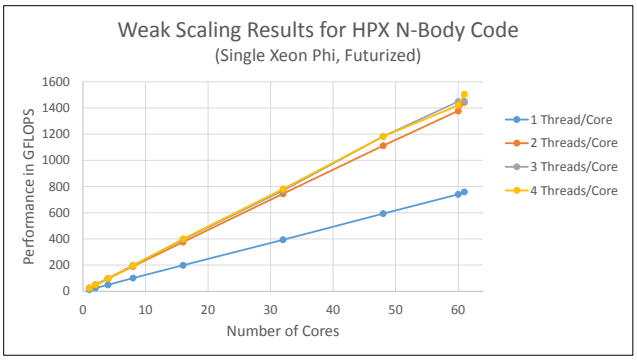

Figure 8: Weak scaling results for the HPX N-Body code collected on a single Xeon Phi coprocessor while varying both the number of cores used and the number of threads per core. The figure shows excellent weak scaling behavior of the HPX backend. The overall performance doesn't increase much more after two threads per core are used. We are able to sustain a peak performance of 1504.7 GFLOPS using all 244 available hardware threads which is equivalent to a parallel overhead of ∼11% and ∼89% of the theoretical achievable peak performance.

threads, have been conducted. For the scaling experiment without the coprocessor, we compared the HPX backend with the MPI backend. Our results show that both backends are able to scale well to up to 1024 nodes (see Fig. 9), reaching a parallel efficiency of ∼80% (HPX) and ∼66% (MPI). The HPX backend is able to outperform the MPI backend at scale by ∼8% and is able to reach a sustained performance of ∼0.35 PFLOPS. The main reason for this performance gain can be attributed to the inherently asynchronous nature of the HPX runtime system which leads to better latency hiding at scale (see Fig. 10) which allows efficient overlapping of computation and communication and furthermore reduces the overheads introduced by communication. The symmetric benchmark

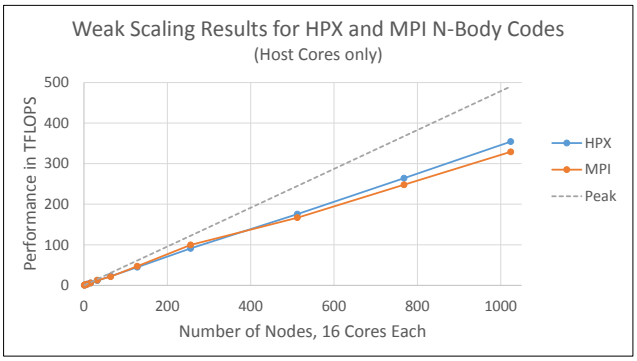

Figure 9: Weak scaling performance results for the HPX and MPI N-Body codes collected for runs on the host's cores only (16 cores per node) while increasing the number of nodes from 1 to 1024 (16 to 16384 cores). On 1024 nodes, the HPX code outperforms the equivalent MPI code by ∼8% and reaches a performance of ∼0.35 PFLOPS.

runs (using both, the host processors and the coprocessor of a node) were only conducted with the HPX backend for the same reason as for the single Xeon Phi runs. Additionally, we needed to take the different speeds of the now heterogeneous processors into account. As the Xeon Phi is about 2.75 times faster then the Xeon E5 processors, we chose the partitioning in such a way, that the Xeon Phi gets 2.75 times more grid elements.

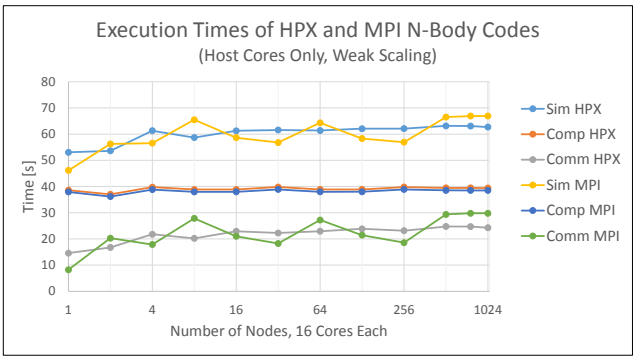

Figure 10: Weak scaling execution times for the HPX and MPI N-Body codes collected on runs for host's cores only (16 cores per node) while increasing the number of nodes from 1 to 1024). The figure shows the times for the overall *Sim* and the overheads introduced by *Comm*. It can be seen that both backend implementations are able to consequently overlap computation with computation, with a slight advantage to the HPX backend at scale which is caused by the advanced asynchronous capabilities of the runtime system.

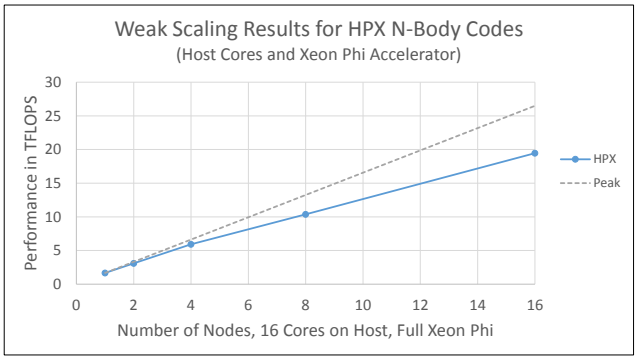

Figure 11: Weak scaling performance results for the HPX N-Body code collected for symmetric runs on the host's cores (16 cores per node) combined with all cores of the associated Xeon Phi accelerators (61 cores, 4 hardware threads each) while increasing the number of hosts from 1 to 16. On 16 nodes, the HPX application reaches a performance of ∼19.5 TFLOPS.

The results (see Fig. 11) show a sustained performance of ∼19.5 TFLOPS when using a total of 4160 hardware threads (256 Xeon E5 cores and 976 Xeon Phi cores, 4 hardware threads each). The parallel efficiency is ∼73% at the achieved scale. The reduced efficiency in comparison to the runs on the hosts only can be explained by an additional overhead introduced by communicating with the coprocessor; the communication always requires an extra hop over the PCI Express bus of the host system. In addition, we were not able to scale beyond 16 nodes at this time as there are still major problems with the underlying MPI software stack provided by Intel which could not be solved in time.

7. CONCLUSION

The work presented in this paper shows how the unified programming model of the HPX runtime system can be efficiently used to implement C++ application frameworks. We successfully ported the parallelization backend of LibGeoDecomp to use HPX. We used a three dimensional N-Body Simulation written in LibGeoDecomp to compare the HPX backend with an already existing

MPI backend. We are able to show perfect scaling at a single node level by reaching ∼98% peak performance of a single node of the Stampede supercomputer. In addition, due to advanced parallelization techniques, we were able to show ∼89% peak performance when using the many-core Xeon Phi coprocessor. Furthermore, the scalability of the HPX backend could be proofed by scaling the code to up to 1024 nodes using only the host CPUs of the Stampede supercomputer (overall 16384 cores) by reaching a parallel efficiency of ∼79% and a sustained performance of ∼0.35 PFLOPS, outperforming and outscaling the MPI backend by ∼8%. Running this code on up to 16 nodes while utilizing the host and the coprocessor, the HPX backend was able to sustain a performance of ∼19.5 TFLOPS while reaching a parallel efficiency of ∼73%.

Our results show that HPX can be efficiently used for homogeneous large scale applications as well as scaling in heterogeneous environments. However, to fully utilize future and current PetaFLOP scale supercomputers, we need to advance further. The ability to make use of migration within AGAS has to be refined in order to dynamically balance load, as we were facing limitations with static load balancing in our heterogeneous benchmarks. Additionally, migration will make it easier to write work-imbalanced applications and improve overall fault tolerance.

8. ACKNOWLEDGMENTS

This work was supported by the Bavarian Research Foundation (Bayerische Forschungsstfitung) funding the project "ZERPA - Zerstörungsfreie Prüfung auf heterogenen Parallelrechner- Architekturen (AZ-987-11)", the XSEDE allocation ASC130032, and by the NSF award 1240655.

9. REFERENCES

[1] Texas Advanced Computing Center - Stampede. http://www.tacc.utexas.edu/resources/hpc/stampede.

[2] The Intel Xeon Phi Product Family. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[3] HPX Source Code Repository, 2007-2013. Available under the Boost Software License. Contact hpx-users@stellar.cct.lsu.edu for repository access.

[4] Boost: a collection of free peer-reviewed portable C++ source libraries, 2013. http://www.boost.org/.

[5] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[6] Arvind and R. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE '87, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*. Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 259.

[7] 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.

[8] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21:291–312, 2007.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[10] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[11] J. B. Dennis. First version of a data flow procedure language. In *Symposium on Programming*, pages 362–376, 1974.

[12] J. B. Dennis and D. Misunas. A preliminary architecture for a basic data-flow processor. In *25 Years ISCA: Retrospectives and Reprints*, pages 125–131, 1998.

[13] B. Fitch, A. Rayshubskiy, M. Eleftheriou, T. Ward, M. Giampapa, Y. Zhestkov, M. Pitman, F. Suits, A. Grossfield, J. Pitera, W. Swope, R. Zhou, S. Feller, and R. Germain. Blue matter: Strong scaling of molecular dynamics on blue gene/l. In V. Alexandrov, G. Albada, P. Sloot, and J. Dongarra, editors, *Computational Science âĂŞ ICCS 2006*, volume 3992 of *Lecture Notes in Computer Science*, pages 846–854. Springer Berlin Heidelberg, 2006.

[14] D. P. Friedman and D. S. Wise. Cons should not evaluate its arguments. In *ICALP*, pages 257–284, 1976.

[15] J. L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, 1988.

[16] R. H. Halstead, Jr. MULTILISP: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, October 1985.

[17] T. Heller, H. Kaiser, and K. Iglberger. Application of the ParalleX Execution Model to Stencil-based Problems. *Computer Science - Research and Development*, pages 1–9, 2012.

[18] Intel. Intel Thread Building Blocks 4.1, 2013. http://www.threadingbuildingblocks.org/.

[19] H. Kaiser, M. Brodowicz, and T. Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICPPW '09, pages 394–401, Washington, DC, USA, 2009. IEEE Computer Society.

[20] C. E. Leiserson. The Cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA, 2009. ACM.

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

[22] Niklas Gustafsson and Artur Laksberg and Herb Sutter and Sana Mithani. Improvements to `std::future<T>` and Related APIs. Technical report, 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3634.pdf.

[23] Niklas Gustafsson and Deon Brewis and Herb Sutter and Sana Mithani. Resumable Functions. Technical report, 2013. http://isocpp.org/files/papers/N3722.pdf.

[24] PGAS. PGAS - Partitioned Global Address Space, 2013. http://www.pgas.org.

[25] A. Schäfer and D. Fey. LibGeoDecomp: A Grid-Enabled Library for Geometric Decomposition Codes. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 285–294, Berlin, Heidelberg, 2008. Springer.

[26] A. Schafer and D. Fey. Zero-overhead interfaces for high-performance computing libraries and kernels. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1139–1146, 2012.

[27] A. Tabbal, M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling. Preliminary design examination of the ParalleX system from a software and hardware perspective. *SIGMETRICS Performance Evaluation Review*, 38:4, Mar 2011.

[28] The C++ Standards Committee. ISO/IEC 14882:2011, Standard for Programming Language C++. Technical report, 2011. http://www.open-std.org/jtc1/sc22/wg21.

[29] The STE||AR Group. Systems Technologies, Emerging Parallelism, and Algorithms Reseach, 2011-2013. http://stellar.cct.lsu.edu.

[30] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[31] A. Vladimirov and V. Karpusenko. Test-driving Intel Xeon Phi coprocessors with a basic N-body simulation. http://goparallel.sourceforge.net/wp-content/uploads/2013/01/Colfax_Nbody_Xeon_Phi.pdf.

[32] D. W. Wall. Messages as active agents. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 34–39, New York, NY, USA, 1982. ACM.