Universität Stuttgart

From Piz Daint to the Stars: Simulation of Stellar Mergers using High-Level Abstractions

Denver, Colorado

November, 2019

University of Stuttgart, IPVS, SSE

Authors

- **Gregor Daiß**
- **John Biddiscombe**
- Parsa Amini
- Patrick Diehl
- Juhan Frank
- Kevin Huck
- Hartmut Kaiser
- Dominic Marcello
- David Pfander
- Dirk Pflüger

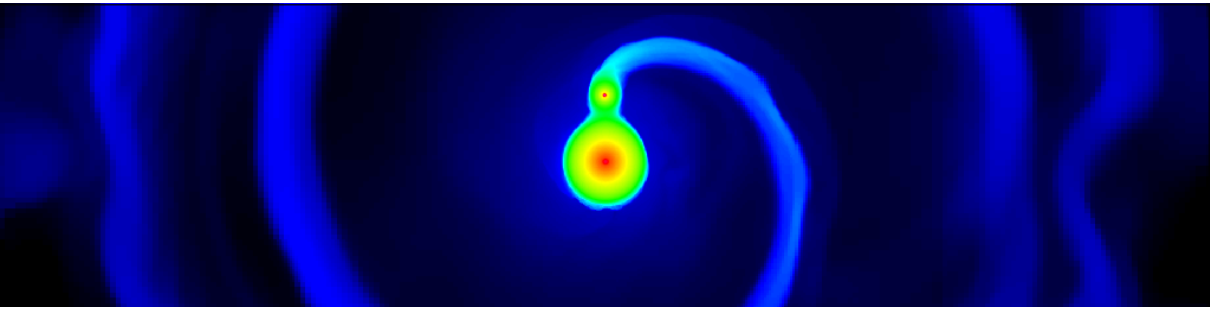
Universität Stuttgart

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

LSU | Center for Computation & Technology
Interdisciplinary | Innovative | Inventive

STE||AR GROUP

Motivation

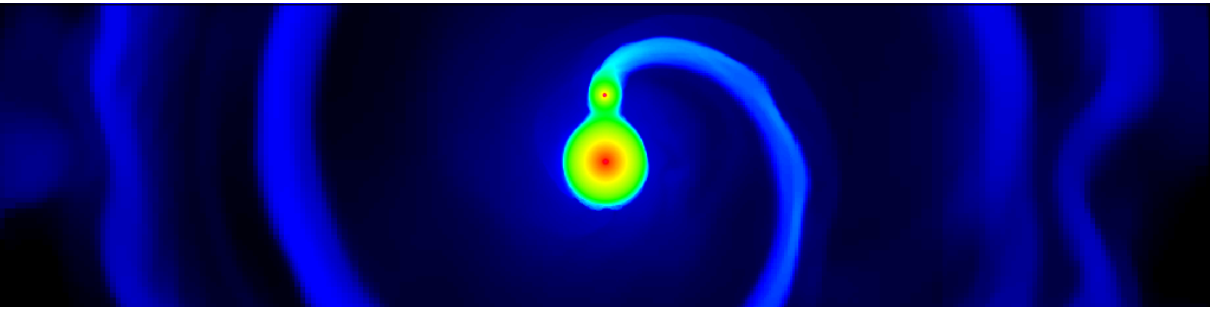


- Simulation of binary star systems and their mergers
- Octo-Tiger models these star systems using self-gravitating fluids on an AMR grid using HPX
- Large-scale runs on Piz Daint use up to 768 million cells.

Contributions:

- Significant speedup replacing MPI with Libfabric without changing any application code
- Integrating small GPU kernels efficiently into an asynchronous many-task runtime system

Motivation



- Simulation of binary star systems and their mergers
- Octo-Tiger models these star systems using self-gravitating fluids on an AMR grid using HPX
- Large-scale runs on Piz Daint use up to 768 million cells.

Contributions:

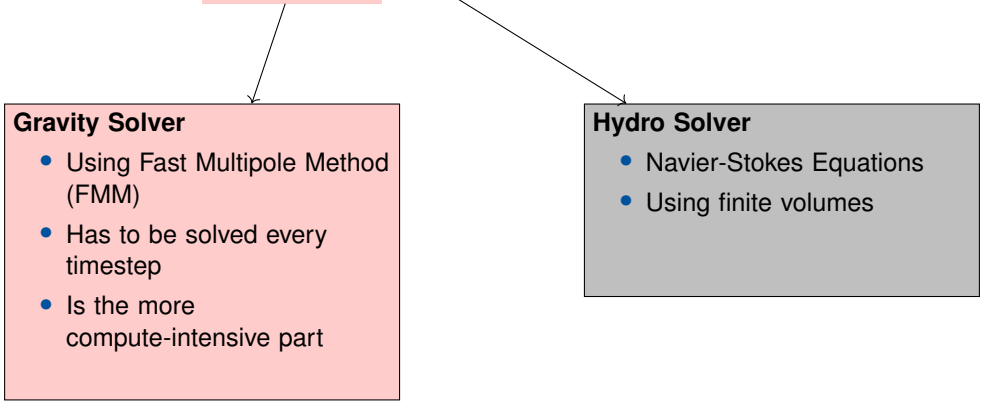
- Significant speedup replacing MPI with Libfabric without changing any application code
- Integrating small GPU kernels efficiently into an asynchronous many-task runtime system

Table of Contents

Octo-Tiger in a Nutshell

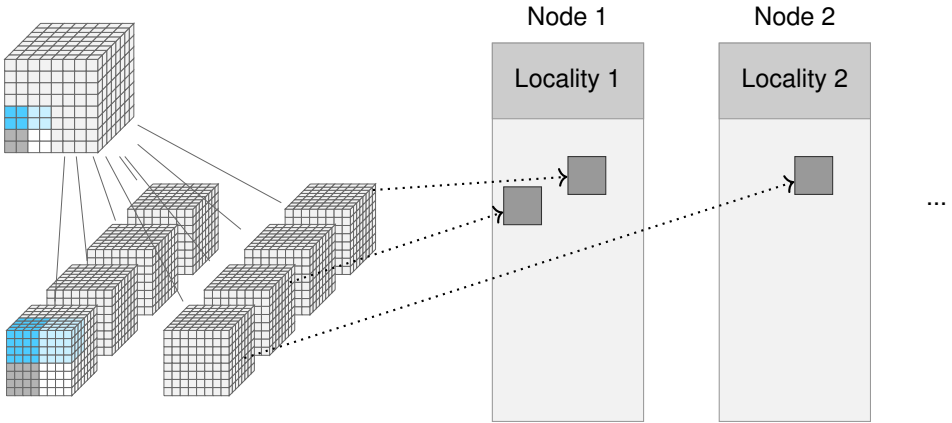
Octo-Tiger in a Nutshell

Octo-Tiger simulates self-gravitating fluids on an AMR grid



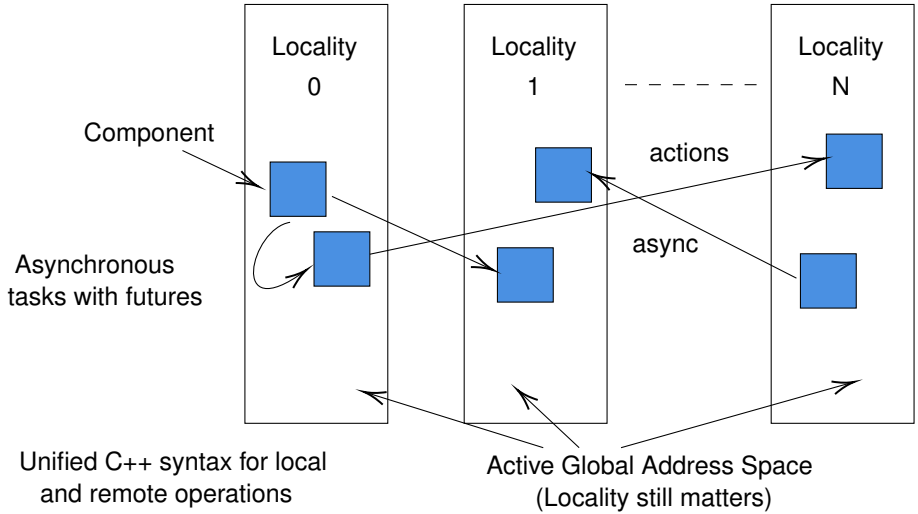
Octo-Tiger in a Nutshell

Octo-Tiger simulates self-gravitating fluids on an AMR grid



HPX and the Libfabric Parcelport in a Nutshell

A Distributed Task-Based Runtime



Standards Driven C++ Tasking for Parallelism and Concurrency

Futures for Synchronization

- Continuation Passing Style (CPS) preferred
- Functional approach to programming
- Task synchronization is also data driven

Runtime

- Lightweight threads
- Suspend on get(), resume when ready
- Work stealing when current task done/suspended

AGAS

- Manages a handle to a component
- Forward work to the locality holding data

Standards Driven C++ Tasking for Parallelism and Concurrency

Futures for Synchronization

- Continuation Passing Style (CPS) preferred
- Functional approach to programming
- Task synchronization is also data driven

Runtime

- Lightweight threads
- Suspend on get(), resume when ready
- Work stealing when current task done/suspended

AGAS

- Manages a handle to a component
- Forward work to the locality holding data

Standards Driven C++ Tasking for Parallelism and Concurrency

Futures for Synchronization

- Continuation Passing Style (CPS) preferred
- Functional approach to programming
- Task synchronization is also data driven

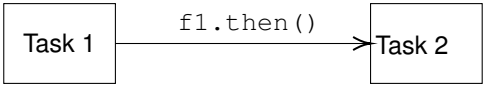
Runtime

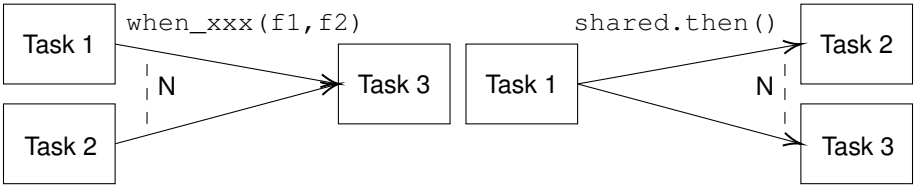
- Lightweight threads
- Suspend on get(), resume when ready
- Work stealing when current task done/suspended

AGAS

- Manages a handle to a component
- Forward work to the locality holding data

Building DAGs from Futures





Remote Actions use Active Messages

AMR refinement and redistribution:

- Moving a subgrid from one node to another

  calling a (copy) constructor on a remote node

  with the contents of this subgrid as parameter(s)

Halo exchange:

- Execute a put on remote node
  - with a data buffer as parameter
- Execute a get on remote node
  - with a (local) buffer address as parameter

Remote Actions use Active Messages

AMR refinement and redistribution:

- Moving a subgrid from one node to another

  calling a (copy) constructor on a remote node
  with the contents of this subgrid as parameter(s)

Halo exchange:

- Execute a put on remote node
  - with a data buffer as parameter
- Execute a get on remote node
  - with a (local) buffer address as parameter

Active Messages

Syntax

- Instead of a more traditional

```
MPI_Isend(buffer, count, datatype, dest_rank, tag, comm, request)
```

HPX messages take the form of a remote function invocation

```
future = hpx::async(dest_locality, function, arg1, arg2...)
```

where any C++ data `args` can be sent (`vector/set/list/map/custom`)

Implementation

- Data is passed as arguments to a remote function (or `object::function`)
- Remote function parameters are serialized into a parcel - consisting of
  - a function identifier, (including object if complex like a `grid::node`)
  - a list of parameters

Channels

- HPX uses Channel abstraction to simplify send/recv for halo regions

Active Messages

Syntax

- Instead of a more traditional

```
MPI_Isend(buffer, count, datatype, dest_rank, tag, comm, request)
```

  HPX messages take the form of a remote function invocation

```
future = hpx::async(dest_locality, function, arg1, arg2...)
```

  where any C++ data `args` can be sent (`vector/set/list/map/custom`)

Implementation

- Data is passed as arguments to a remote function (or `object::function`)
- Remote function parameters are serialized into a parcel - consisting of
  - a function identifier, (including object if complex like a `grid::node`)
  - a list of parameters

Channels

- HPX uses Channel abstraction to simplify send/recv for halo regions

Active Messages

Syntax

- Instead of a more traditional

```
MPI_Isend(buffer, count, datatype, dest_rank, tag, comm, request)
```

  HPX messages take the form of a remote function invocation

```
future = hpx::async(dest_locality, function, arg1, arg2...)
```

  where any C++ data `args` can be sent (`vector/set/list/map/custom`)

Implementation

- Data is passed as arguments to a remote function (or `object::function`)
- Remote function parameters are serialized into a parcel - consisting of
  - a function identifier, (including object if complex like a `grid::node`)
  - a list of parameters

Channels

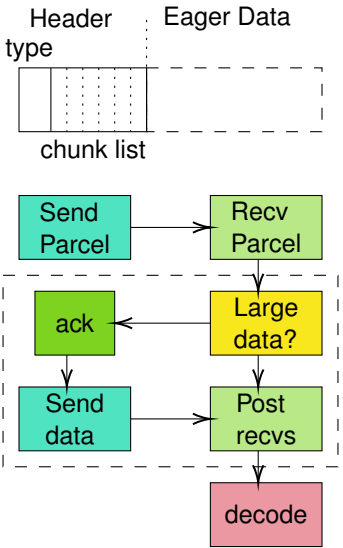
- HPX uses Channel abstraction to simplify send/recv for halo regions

Existing MPI-Based Parcelport Implementation

A parcel is represented by a ‘chunk list’ + data block

- If params are small (eager protocol)
  - Index chunk (size/offset)

  copy into parcel buffer
- For large params (rendezvous)
  - Pointer chunk - separate sends
- Message handling of parcels is currently sub-optimal one sided put/get can/should be used for rendevous items



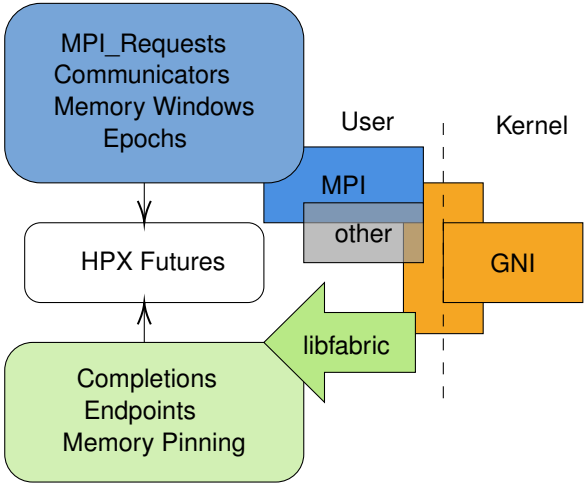
Libfabric as an Alternative to MPI

Downsides of MPI

- MPI_Put/Get not asynchronous API
- Copies completions to MPI_Request handles
- Memory management less flexible

Benefits of libfabric

- API asynchronous (inc. Put/Get – enqueue many)
- Maps driver/GNI completion queues (without copy)
- Robustly threadsafe
- Vectorized sends : fi_sendv
- Flexible memory pinning



Fine Tuning of RDMA-Based Parcelport

Impedance match between HPX and libfabric API

- Identical asynchronous GNI/driver level completions for send/recv/get/(put)
- Trigger futures directly from completion handler

Memory Management

- C++ allocator for pinned memory blocks
- Flow control – we explicitly manage queues (=buffers)
- FI_sendv allows reduced memory copies
- Multi-Parcels when send buffers filling up (FI_sendv)
- RDMA<T> types integrated into our parcelport (channels ongoing)

Threading

- Robust threadsafe libfabric API
- FI_CONTEXT allows us to be 100% lock free in our HPX layer
  - map completions directly to objects (c.f. communicators)

Fine Tuning of RDMA-Based Parcelport

Impedance match between HPX and libfabric API

- Identical asynchronous GNI/driver level completions for send/recv/get/(put)
- Trigger futures directly from completion handler

Memory Management

- C++ allocator for pinned memory blocks
- Flow control – we explicitly manage queues (=buffers)
- FI_sendv allows reduced memory copies
- Multi-Parcels when send buffers filling up (FI_sendv)
- RDMA<T> types integrated into our parcelport (channels ongoing)

Threading

- Robust threadsafe libfabric API
- FI_CONTEXT allows us to be 100% lock free in our HPX layer
  - map completions directly to objects (c.f. communicators)

Fine Tuning of RDMA-Based Parcelport

Impedance match between HPX and libfabric API

- Identical asynchronous GNI/driver level completions for send/recv/get/(put)
- Trigger futures directly from completion handler

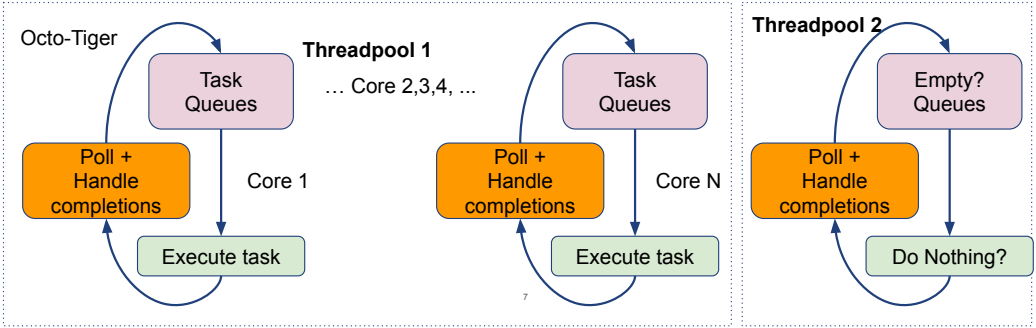
Memory Management

- C++ allocator for pinned memory blocks
- Flow control – we explicitly manage queues (=buffers)
- FI_sendv allows reduced memory copies
- Multi-Parcels when send buffers filling up (FI_sendv)
- RDMA<T> types integrated into our parcelport (channels ongoing)

Threading

- Robust threadsafe libfabric API
- FI_CONTEXT allows us to be 100% lock free in our HPX layer
  - map completions directly to objects (c.f. communicators)

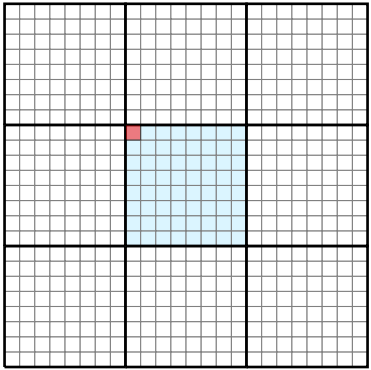
Performance/Integration of Libfabric in the Runtime



- Every core can poll for completion events during background processing
- Polling can be moved to another thread pool, with or without tasks
- Every microsecond saved in polling/handling = 1MFlop on a 1TFlop GPU

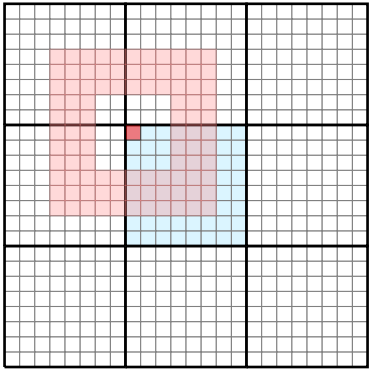
Asynchronous Many Tasks with GPUs

Example FMM Kernels from Octo-Tiger



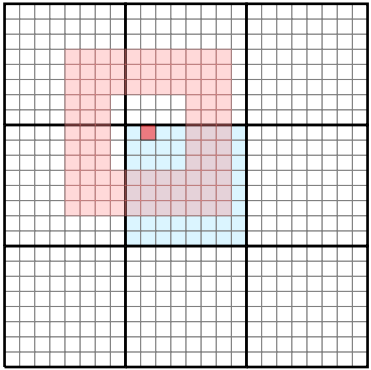
- Calculation of the gravity interactions between neighboring cells on the same oct-tree level
- Stencil code

Example FMM Kernels from Octo-Tiger



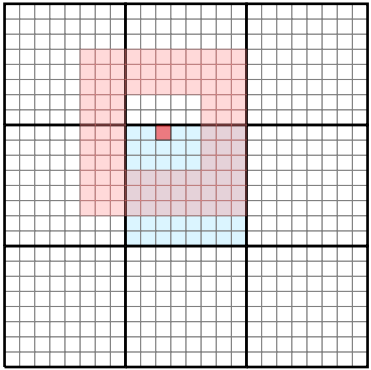
- Calculation of the gravity interactions between neighboring cells on the same oct-tree level
- Stencil code

Example FMM Kernels from Octo-Tiger



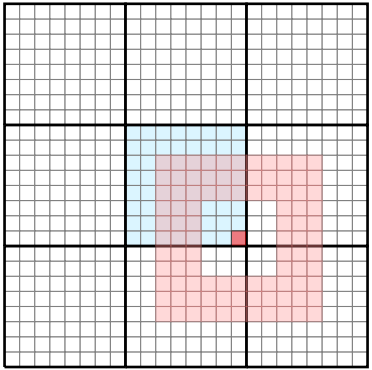
- Calculation of the gravity interactions between neighboring cells on the same oct-tree level
- Stencil code

Example FMM Kernels from Octo-Tiger



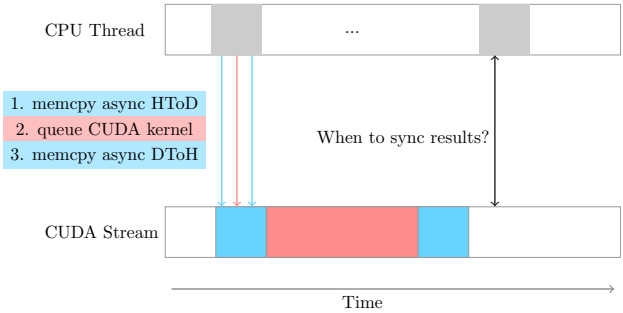
- Calculation of the gravity interactions between neighboring cells on the same oct-tree level
- Stencil code

Example FMM Kernels from Octo-Tiger



- Calculation of the gravity interactions between neighboring cells on the same oct-tree level
- Stencil code
- (3D) Stencil has 1074 elements
- Stencil gets applied for all the 512 cells per subgrid

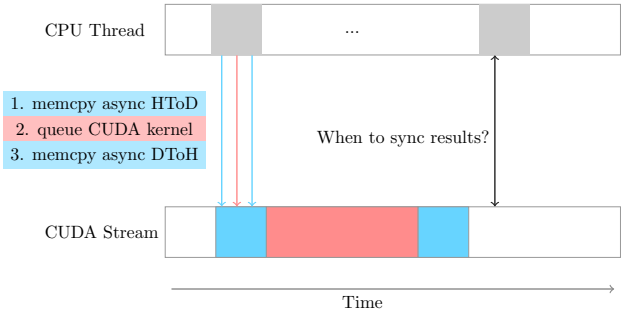
Running the CUDA Kernel



Goals:

- Interleave GPU kernel with arbitrary CPU kernels and communication
- Non-blocking synchronization

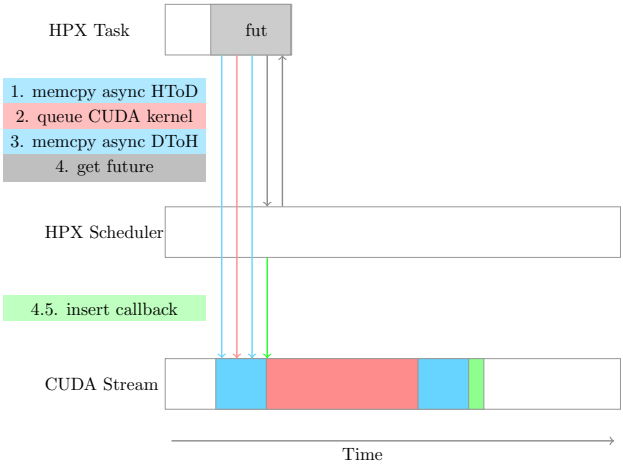
Running the CUDA Kernel



Goals:

- Interleave GPU kernel with arbitrary CPU kernels and communication
- Non-blocking synchronization

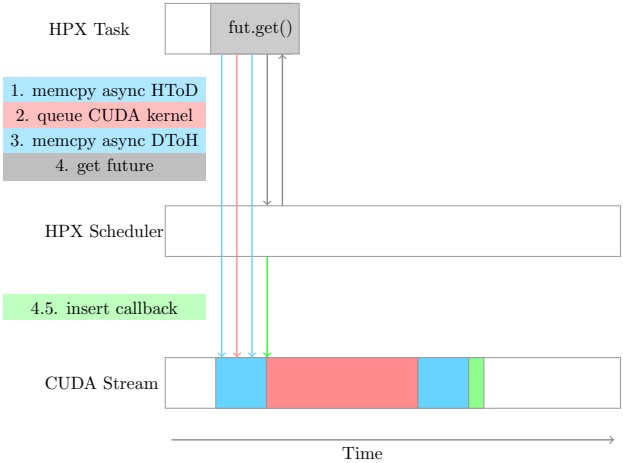
Integrating with HPX



Solution:

- Use HPX tasks instead
- Insert callback into CUDA stream
- Scheduler can return a future that becomes ready once this callback get executed

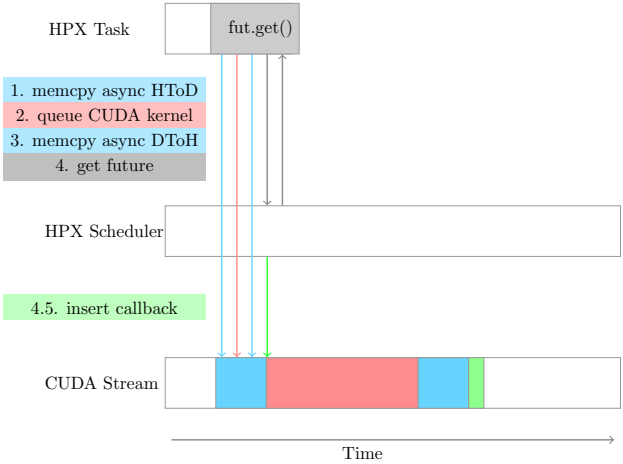
Integrating with HPX



Solution:

- Use HPX tasks instead
- Insert callback into CUDA stream
- Scheduler can return a future that becomes ready once this callback get executed
- HPX task gets suspended

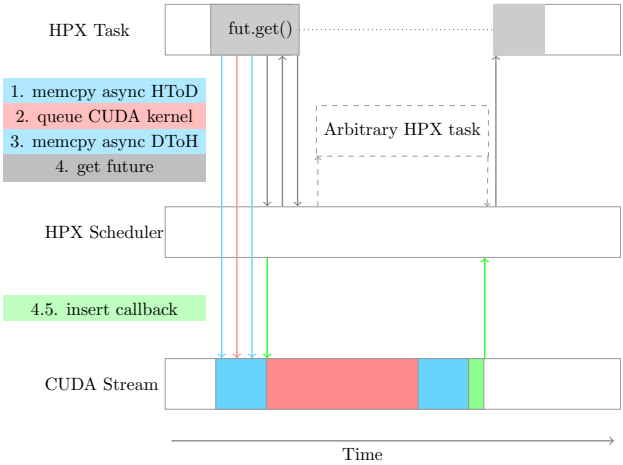
Integrating with HPX



Solution:

- Use HPX tasks instead
- Insert callback into CUDA stream
- Scheduler can return a future that becomes ready once this callback get executed
- HPX task gets suspended

Integrating with HPX



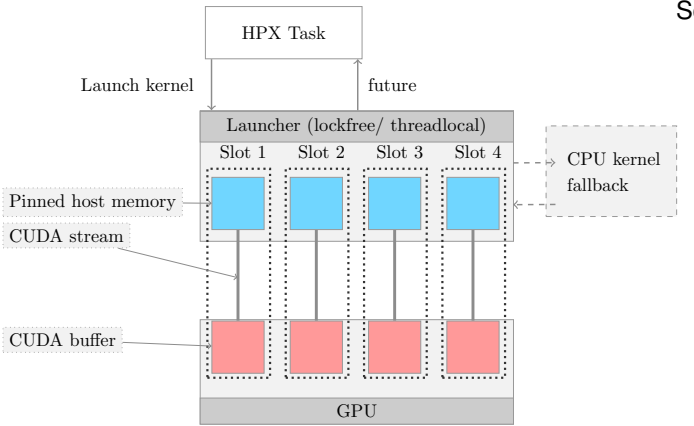
Solution:

- Use HPX tasks instead
- Insert callback into CUDA stream
- Scheduler can return a future that becomes ready once this callback get executed
- HPX task gets suspended
- HPX thread can work on other tasks/communication
- Task will be resumed once the GPU kernel has finished

Filling the GPU?

- One kernel calculates 512 * 1074 cell interactions
  - Depending on the type, 12 to 455 floating point operations
  - Still not enough work to utilize even one GPU
- Leverage CUDA streams for implicit work aggregation
- Avoid on-the-fly allocations

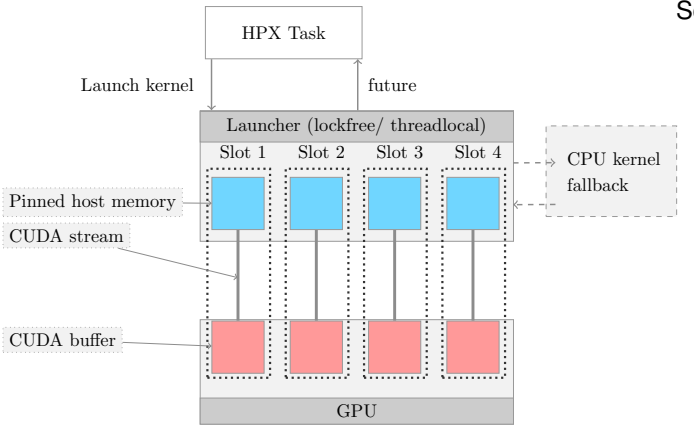
Filling the GPU!



Solution

- Launch many small kernels in different streams
- One launcher for each HPX thread
- One kernel launch per slot
- If all slots are busy, execute the kernel on the CPU
- Most kernels executed on the GPU (99.5%)
- Arbitrary number of slots on multiple GPUs

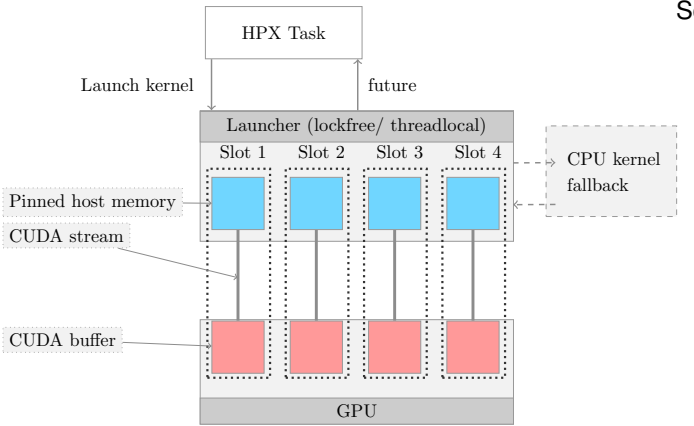
Filling the GPU!



Solution

- Launch many small kernels in different streams
- One launcher for each HPX thread
- One kernel launch per slot
- If all slots are busy, execute the kernel on the CPU
- Most kernels executed on the GPU (99.5%)
- Arbitrary number of slots on multiple GPUs

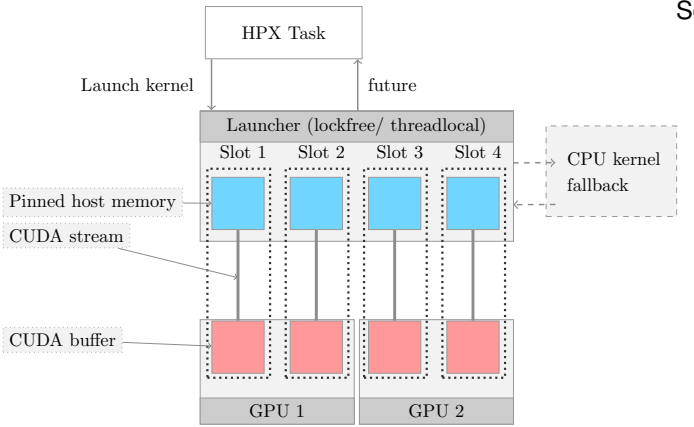
Filling the GPU!



Solution

- Launch many small kernels in different streams
- One launcher for each HPX thread
- One kernel launch per slot
- If all slots are busy, execute the kernel on the CPU
- Most kernels executed on the GPU (99.5%)
- Arbitrary number of slots on multiple GPUs

Work Size



Solution

- Launch many small kernels in different streams
- One launcher for each HPX thread
- One kernel launch per slot
- If all slots are busy, execute the kernel on the CPU
- Most kernels are being executed on the GPU (99.5207%)
- Arbitrary number of slots on multiple GPUs

Asynchronous Many Tasks with GPUs

Advantages:

- Optimize GPU launch
  - CUDA streams
  - HPX futures
  - non-blocking launcher
- Overlapping
  - CPU/GPU tasks;
  - computation and communication

Results

FMM Node-Level Results on Piz Daint

- Ported most important FMM kernels to the GPU
- Scenario: V1309 contact binary merger
- Setup: Single node, 12 HPX-Threads, 128 launch slots (CUDA streams)
- 10928 sub-grids resulting in 5595136 cells

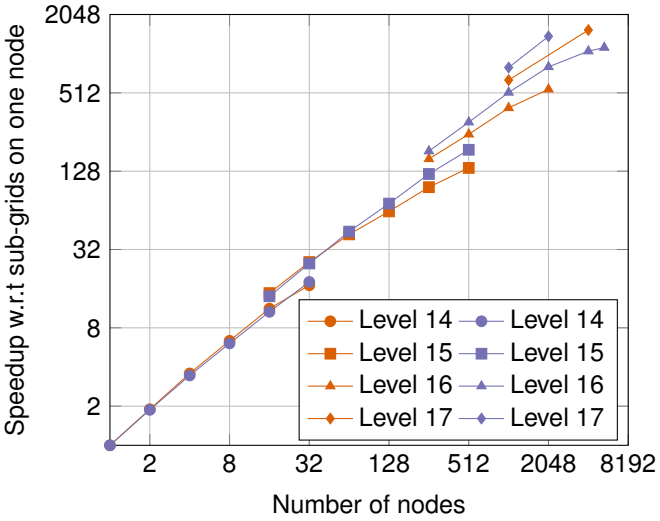
Utilized Hardware	FMM			Total scenario runtime (FMM + Hydro + Others)
	runtime	GFLOP/s	fraction of peak	
One Piz Daint Node				
Intel Xeon E5-2690v3	980s	157 GFLOP/s	31% [CPU]	2415s
with 1x NVIDIA P100 (PCI-E)	158s	973 GFLOP/s	21% [GPU]	1592s

FMM Node-Level Results on Piz Daint

- Ported most important FMM kernels to the GPU
- Scenario: V1309 contact binary merger
- Setup: Single node, 12 HPX-Threads, 128 launch slots (CUDA streams)
- 10928 sub-grids resulting in 5595136 cells

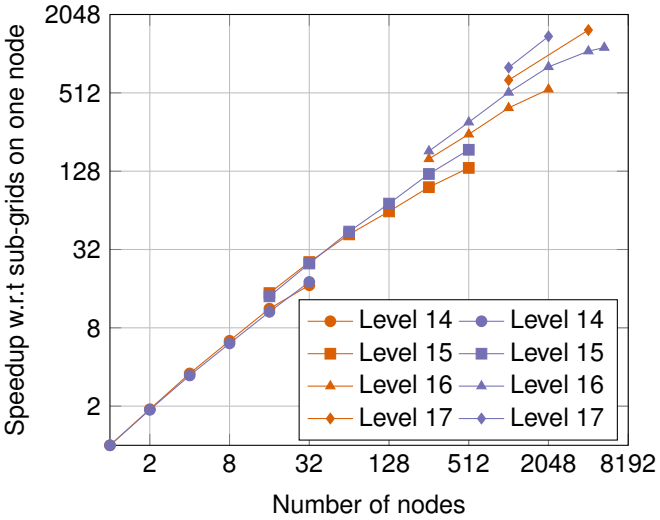
Utilized Hardware	FMM			Total scenario runtime (FMM + Hydro + Others)
	runtime	GFLOP/s	fraction of peak	
One Piz Daint Node				
Intel Xeon E5-2690v3	980s	157 GFLOP/s	31% [CPU]	2415s
with 1x NVIDIA P100 (PCI-E)	158s	973 GFLOP/s	21% [GPU]	1592s

Distributed Results



- The red lines show the results using HPX's MPI parcelport and the blue lines using HPX's Libfabric parcelport, respectively
- Number of subgrids ranges from 10928 (level 14) to 1.5 million subgrids (level 17) depending on the refinement levels
- Achieved a weak scaling of 68.1% with 2048 nodes on Piz Daint on level 17
- At 4096 nodes 2.7 speedup using Libfabric

Distributed Results

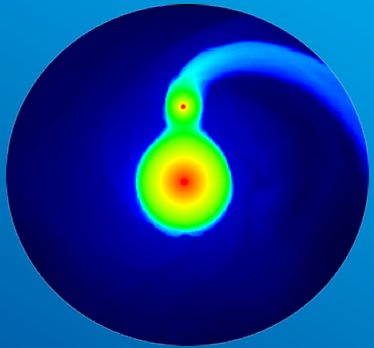


- The red lines show the results using HPX's MPI parcelport and the blue lines using HPX's Libfabric parcelport, respectively
- Number of subgrids ranges from 10928 (level 14) to 1.5 million subgrids (level 17) depending on the refinement levels
- Achieved a weak scaling of 68.1% with 2048 nodes on Piz Daint on level 17
- At 4096 nodes 2.7 speedup using Libfabric

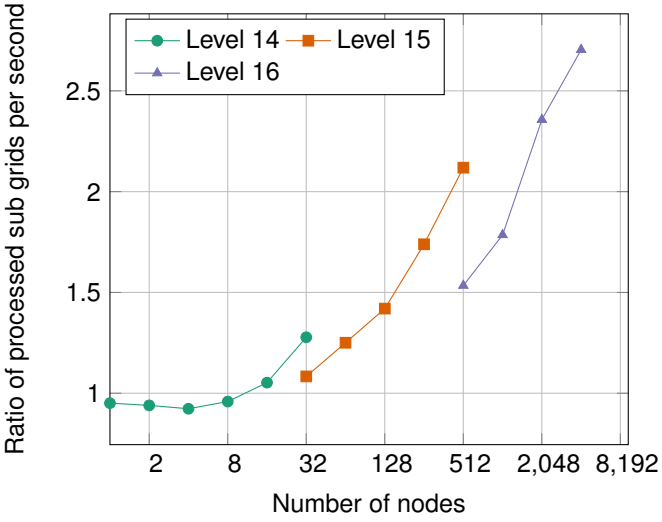
Summary

- HPX programming model
  exposes easy synchronization with futures for
  - Networking
  - GPU / CUDA
  - CPU
- Reduced overhead to maximize throughput

Thank you for your attention!



Distributed Results



- Ratio of processed sub grids per second between HPXs Libfabric and MPI parcelport on Piz Daint
- Switch to Libfabric did not require any changes within Octo-Tiger