ETH Zürich

BSc Thesis

A Parallel Hybrid Particle Mesh Framework

Author:
J. Progsch

Supervisors:
Dr. A. Adelmann & Prof. Dr. R. Hiptmair

July 20, 2012

Abstract

Modern computing platforms pose new challenges to developers by requiring the handling of multiple levels of parallelism and by incorporating heterogeneous compute units in the form of multicore processors and accelerators. This report proposes an approach to take these developments into account and outlines an extensible framework architecture for grid and particle based simulations based on tag dispatching.

1 Introduction

Parallelism has become the key to achieving high performance in modern computing. Parallel computing has become commonplace on all levels of hardware all the way down to mobile platforms. Massively parallel solutions have even found their way into desktop platforms in the form of graphics processing units. In the domain of high performance computing highly integrated massively parallel solutions such as Intel MIC (Many Integrated Core)

architecture [1] will play an important role in the future since they greatly improve the overall compute power and energy efficiency of systems on all scales. To accommodate these changes in computing hardware software and the frameworks and libraries it is based on have to be designed around heterogeneity and parallelism. Existing libraries such as IPPL (Independent Parallel Particle Layer) [2] and POOMA (Parallel Object-Oriented Methods and Applications) [3] are designed to support a single execution model at a time and provide only MPI (Message Passing Interface) or shared memory parallel programming models. It is therefore desirable to develop a new framework that focuses on unification and integration of current and future trends in high performance computing. This report focuses on the design of such a framework for particle and grid based simulation. Applications of such simulations include gravitational N-body simulations, space charge simulations in beam optics as well as finite difference schemes on grids. This report will use a simple stencil operation (the discrete Laplace operator) on a three dimensional grid to demonstrate and explain the concepts of the framework. The framework prototype serves as a proof-of-concept. It is developed in C++ and heavily relies on template meta programming and the Boost [4] libraries.

2 Data Parallelism

The parallelization for the framework is done mostly with a data parallel approach based on domain decomposition. A finite simulation domain is chosen such that it contains all the particles and regions of interest. This domain is then divided into sub domains which are distributed among the compute units (such as MPI nodes, processors, accelerators etc.). The domain decomposition can have multiple levels to reflect the hierarchical structure of the used hardware. Such a system can for example consist of multiple MPI nodes each containing one or multiple multicore processors and accelerators. It is possible to abstract these kind of systems by pure MPI parallelism. However, the hierarchical structure and different compute powers and bandwidths are not taken into account which in turn can prevent good load balancing. The different hierarchies of the parallelization are shown in figure 1. The simulation domain is first divided into as least as many sub domains as there are MPI nodes. Preferably even a multiple of the actual amount of compute units (processor cores). These sub domains are then distributed among MPI

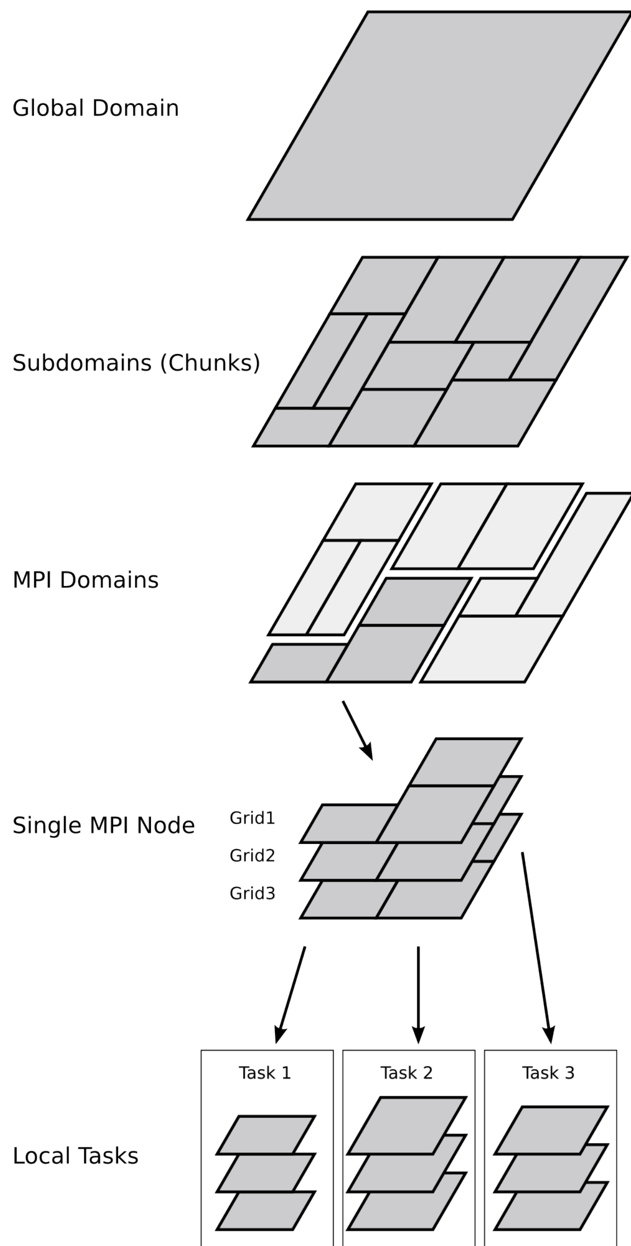


Figure 1: Domain decomposition into MPI and task domains.

nodes. When a task is launched for multiple data structures (grids, parti-

cle containers) that share the same domain decomposition, one sub task is launched per sub domain and operates only on the elements inside that sub domain. This way the sub tasks are independent of each other and can be scheduled according to the available compute units. They might be scheduled sequentially on a single core node or in multiple threads on a multicore node. The tasks can even be distributed among heterogeneous compute units (CPUs and accelerators) within the same MPI node. For the purposes of applying the discrete Laplace operator mentioned in the introduction two grids are needed. A source and a destination grid. Those grids have to share the same domain decomposition such that their data chunks match in size and position. Zhe operation does not depend on the outer shape of the chunks. This allows us to choose any decomposition of the grid we deem appropriate for the underlying hardware configuration. A reasonable starting point would be to uniformly subdivide such that the amount of sub domains matches the total amount of compute units/processors. On execution of the task one thread is created for each sub domain. Each of those threads then executes the stencil operation independently on it's assigned source and destination chunks by taking the source chunk as input and writing the result to the destination chunk.

3 Programming Techniques

3.1 Template Meta programming

Through templates C++ provides the means to offload significant amounts of logic to compile time. Furthermore since templates are evaluated statically they don't incur additional run time cost through dynamic dispatching or dynamic casting. Sadly template meta programming is cumbersome and hard to debug. To reduce the burden of writing template code and improve maintainability Boost provides powerful meta programming libraries such as MPL [5] [6], Fusion [7] and Proto [8]. MPL mimics the functionality of the standard template library (STL) by providing container and algorithms for types (as opposed to run time data). It allows templates to store, manipulate and iterate over collections of types. The Fusion library connects pure compile time type constructs with run time data. It does so by providing the essentially same types as type containers as MPL with the difference that Fusion types can be instantiated and can serve as heterogeneous collections

of data while still allowing for iteration and retrieval of type information of the members (see Appendix A.1 for a more detailed explanation and an example). MPL and Fusion are central to the implementation of the discussed design. The third mentioned library (Proto) builds upon MPL and Fusion and provides means to implement expression templates and domain specific languages within C++. It is not used in the current proof of concept implementation of the design, but could be used to great effect to provide expressive interfaces for higher level operations such as stencil operators and particle updates.

3.2 Tag Dispatching

An important feature of the framework has to be extensibility. Preferably extending the functionality, to for example additional execution models like OpenCL [9] or CUDA (Compute Unified Device Architecture) [10], should be doable in a non intrusive fashion and without adding strong dependencies to the core framework. One way to do this is to provide hooks via template specialization for tags. This technique is called tag dispatching and is closely related to "traits" that are for example extensively used in the Boost libraries. It is based on the ability to provide different implementations of a template class based on the template arguments. By providing specialization for new types one can therefore change and extend the functionality of internally used template types in a framework. Tags are empty types that are there solely to provide named types for which such templates can be specialized. The tag can then be passed to the framework which in turn can "pull in" all the functionality associated with that tag. This mechanism simplifies the addition of new functionality by providing well defined entry points for the extensions while also decoupling the extension from internals of the framework.

And example of this technique is provided in Listing 1. In the proposed framework the components can be supplied with sequences of such tags to attach multiple different behaviors to a single component.

4 Structure of a Particle-Mesh library

The framework has to provide two central data structures which are grids and particle containers. Along with these structures it also has to expose

Listing 1 Example for tab based extensions.

```
template<class T, class Tag>
struct PlatformSpecific;

template<class T, class Tag>
class Container {
public:
//...
private:
   PlatformSpecific<T, Tag> representation;
};

class cpu_tag { };

template<class T>
struct PlatformSpecific<T, cpu_tag> {
   std::vector<T> data;
};

class acc_tag { };

template<class T>
struct PlatformSpecific<T, acc_tag> {
   acc_buffer<T> data;
};
```

ways to implement and execute operators that act on them which includes stencil operations, particle updates, particle particle interactions, gather and scatter operations of particle data onto and off the grids. The particle containers and grids are attached to a layout object which holds the information about how the data chunks are distributed among MPI processes. In the proposed approach the layout in turn is attached to a context object that determines which resources are usable by the attached structures. The context contains the MPI communicator used for the process level parallelism as well as tagged `Processor` components that contain the information required for the different execution models such as single threaded execution, thread pool based, OpenCL based etc.. Similarly the data chunks contain tagged `representation` objects that allow the chunk to switch between different representations required for different execution models. The tasks that operate on the grids and particle containers are implicitly defined by a `TaskInfo` object and one or more `TaskImplementation` template specializations. The TaskInfo object is a user defined type that contains the required information about the arguments of the Task. The `TaskImplementation` template gets specialized for any desired TaskInfo/tag combination. By doing this exist-

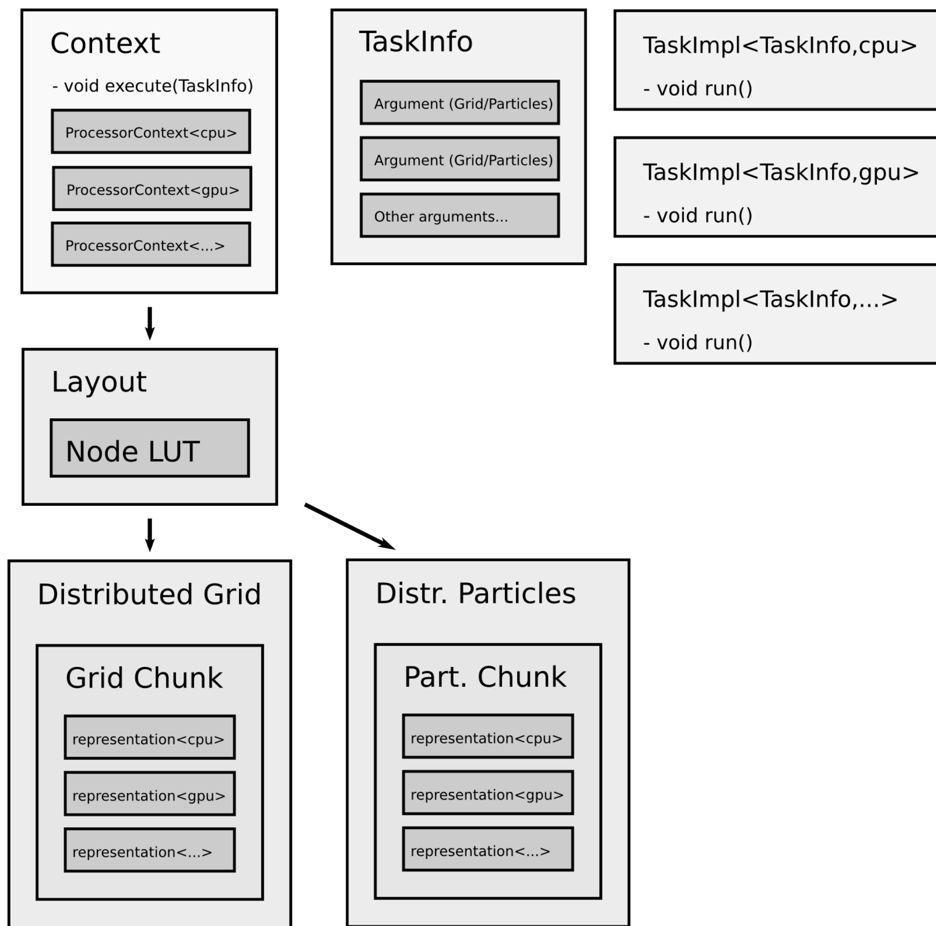


Figure 2: Diagram of principal components

ing `TaskInfo` objects can be extended to additional execution models with relative ease. On execution the context tries to instantiate a fitting implementation template and on success enqueues one instance of the task for each data chunk of the arguments on their respective processors. A component diagram of the proposed framework is presented in Figure 2.

4.1 Context

The `Context` object takes one template argument which has to be a MPL container containing one or more tags for the execution models available through that context. In addition the constructor takes a Boost.MPI communicator

that it stores and exposes to the attached objects. The tag container is then used to instantiate a Boost.Fusion map that provides a mapping of `tag` to `ProcessorContext<tag>` where the `ProcessorContext` template is specialized to hold the specific resources required for execution of tasks on the computing resources associated with `tag`. So the `thread_pool_tag` for example would instantiate the actual thread pool object in the `ProcessorContext` and a `opencl_tag` would create the required OpenCL contexts, queues etc..

4.2 Layout

The layout has the sole purpose to associate domains of the grids/particle containers to chunks and their respective "address" within the MPI nodes. It does so by storing domain/rank-pairs in a node look up table (LUT). In addition it forwards the `Context` it takes in the constructor to the attached objects. In the prototype framework the rank to domain mapping is provided by a multipmap. The inverse mapping is done by searching the value set of the multimap which could be improved on by using a tree data structure as it is done in IPPL [2].

4.3 DistributedGrid and DistributedParticles

The DistributedGrid/-Particle objects take a `Layout` object and create data chunks for each domain in the layout on the MPI rank associated with that domain. The `Chunk` objects acquire the tag list from the `Context` via the `Layout` and similarly to the map of `ProcessorContext` objects in the context the chunks contain maps of `ChunkRepresentation` objects. These representation objects allow the chunk to be converted into execution model specific representations such as for example device side buffers for accelerator cards.

4.4 Tasks

Tasks that can be executed on the grids and particle containers are specified by a `TaskInfo` object which can be any type that exposes the associated arguments through a fusion sequence. The actual implementation of the task is done inside a specialization of the `TaskImplementation<TaskInfo, Tag>` template which has a `run()` member function that takes the arguments specified in the info object in their tag specific representation as a fusion sequence. For the purpose of our discrete Laplace operator the task information will

be supplied by a user defined template struct `LaplaceStencil`. The class simply stores references to the source and destination grids and exposes them as fusion sequence in its `argument` member. The `TaskImplementation` then gets specialized for LaplaceStencil in the first template argument and the tag of the target execution model in the second template argument. Possible implementations are shown in Listing 2.

5 Parallel Execution Models

In the proposed approach process parallelism through MPI is implicitly supported by the Context and Layout objects. Thread parallelism and other parallel execution models such as OpenCL and CUDA is provided by instancing the context with additional tags to pull in the respective Processor objects and data representations. For pure MPI parallel execution one would only pass a tag for single threaded execution to the context. For a mixed approach with multiple threads per process and OpenCL one would instantiate a context object with a thread pool and a OpenCL tag to pull in the respective processor objects and data representations. Thread pool parallelism can be implemented by means of C++11, Boost.Threads or Intel Treading Building Blocks. Thanks to the tag dispatching mechanism these implementations can be made interchangeable.

6 Conclusion

An extensible design for a particle mesh framework has been developed and a prototype has been implemented as a proof of concept. The prototype shows that the proposed method of tag based dispatch to different compute units and data representations can be efficiently implemented in terms of template meta programming within libraries provided by Boost. In addition different approaches to thread level parallelism and GPGPU execution models in the form of OpenCL have been studied to aid with the design of the architecture. Future development towards a production level framework will include the integration of expression template [11] [12] [13] functionality to lift the burden of explicit implementation of tasks from the user (most likely by means of Boost.Proto). Further challenges arise from the need to load balance the different execution models that can be present in the framework

Listing 2 Implementation of a `TaskInfo` and `TaskImplementation` objects for the discrete Laplace operator. L and T are template arguments of the `DistributedGrid` templates. L supplies the Layout type and T is the element type of the grids.

```
template<class L, class T>
struct LaplaceStencil {
   // constructor taking source and destination grid
   LaplaceStencil(DistributedGrid<L, T> &source,
                  DistributedGrid<L, T> &destination)
    : arguments(source, destination) { }

   // the arguments are stored in a fusion sequence type
   typedef fusion::vector<
      DistributedGrid<L, T>&, DistributedGrid<L, T>&
   > Arguments_t;

   Arguments_t arguments;
};

// implementation for execution in a thread pool (thread_pool_tag)
template<class L, class T>
class TaskImplementation<LaplaceStencil<L,T>, thread_pool_tag> {
public:
   template<class A>
   void run(A arguments) const
   {
      // extract the arguments from the fusion sequence
      auto &source = fusion::at_c<0>(arguments);
      auto &destination = fusion::at_c<1>(arguments);

      // find local domain
      NRange<index_t,3> local = source.chunk.bounds();

      // perform the stencil operation
      for(index_t i = local[0].begin;i<local[0].end;++i)
         for(index_t j = local[1].begin;j<local[1].end;++j)
            for(index_t k = local[2].begin;k<local[2].end;++k)
            {
               destination.chunk(i,j,k) = -6*source.chunk(i,j,k)
               + source.chunk(i+1,j,k) + source.chunk(i-1,j,k)
               + source.chunk(i,j+1,k) + source.chunk(i,j-1,k)
               + source.chunk(i,j,k+1) + source.chunk(i,j,k-1);
            }
   }
};
```

and from assuring efficient latency hiding between the many asynchronous communication layers.

References

[1] Intel MIC. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[2] A. Adelmann. The IPPL (Independent Parallel Particle Layer) Framework . Technical Report PSI-PR-09-05, Paul Scherrer Institut, 2009.

[3] POOMA. http://acts.nersc.gov/formertools/pooma/index.html.

[4] Boost. http://www.boost.org/.

[5] Boost.MPL. http://www.boost.org/doc/libs/1_50_0/libs/mpl/doc/index.html.

[6] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[7] Boost.Fusion. http://www.boost.org/doc/libs/1_50_0/libs/fusion/doc/html/index.html.

[8] Boost.Proto. http://www.boost.org/doc/libs/1_50_0/doc/html/proto.html.

[9] OpenCL. http://www.khronos.org/opencl/.

[10] CUDA. http://www.nvidia.com/content/cuda/cuda-developer-resources.html.

[11] Todd Veldhuizen. Expression templates, c++ report 7. 1995.

[12] C. Pflaum and Z. Rahimi. Parallelization of staggered grid codes with expression templates.

[13] J. Progsch, Yves Ineichen, and Andreas Adelmann. A new vectorization technique for expression templates in c++. *American Journal of Undergraduate Research*, Volume 10, Number 4, 2012.

A Devlog Entries

The following subsections are entries of a development weblog that was written during development of the prototype framework. The entries treat various implementation aspects in more detail.

A.1 Fun with Fusion

Lets have a look at Fusion. Fusion is a part of Boost and provides facilities to handle fusion sequences. These sequences bridge the gap between compile time type containers such as Alexandrescus typelists or the classes present in Boost's MPL and run time types. As opposed to a pure type containers fusion sequences can be instantiated. A fusion sequences is anything that fulfills the necessary concepts (similar to STL containers). A lot of types that are commonly used already are fusion sequences or can be adapted as such. Examples are `std::pair`, tuples, arrays and even structs can be adapted as fusion sequences. Fusion also has it's own types that we can use like for example `fusion::vector`

```
1  typedef fusion::vector<int,float,char> sequence_t;
2  sequence_t sequence(42, 3.14f, 'c');
3
4  cout << fusion::size(sequence) << endl;
5  cout << fusion::at_c<1>(sequence) << endl;
6
7  //we can also get compile time information
8  int array[fusion::result_of::size<sequence_t>::value];
9
10 remove_reference<
11    fusion::result_of::at_c<sequence_t, 1>::type
12 >::type foo; // foo is a float
```

Fusion is generally very useful for meta programming and is used by many of the other Boost libraries. A specific usage I see for my bachelor thesis project is in the definition of particle formats. Since Im developing a particle framework I need a way to tell the framework what a Particle actually is. From a programmers point of view a particle is a collection of data such as position, velocity and charge. But it might also contain non physical data such as a Id to identify particles globally or flags. So the obvious thing would be to have the library user define the particle in terms of a struct. The problem with that is, that the library actually doesnt know what is in that struct. But if the particle is defined as a fusion sequence the framework knows all the static types and can access the data in the particle. Luckily

we dont even have to choose between fusion sequences and structs because we can adapt a struct as a fusion sequence. So we can do this:

```
typedef fusion::vector<
   Vector3d, // position
   Vector3d, // velocity
   double,   // charge
   size_t   // ID
> particle_t;
```

or this:

```
struct particle_t {
   Vector3d position;
   Vector3d velocity;
   double charge;
   size_t Id;
};
// tell fusion about the struct
BOOST_FUSION_ADAPT_STRUCT(
   particle_t,
   (Vector3d, position)
   (Vector3d, velocity)
   (double, charge)
   (size_t, Id)
)
```

An example where being able to query the properties of the particle type would be useful is when we want to be able to decide the storage order of the particles in the library. For graphics applications we may want the data to be laid out as an array of structs, in other contexts a struct of arrays can be preferable. Here is how this could look like:

```
struct vectorify {
   template<class> struct result;

   template<class F, class T>
   struct result<F(T)> {
      typedef std::vector<T> type;
   };
};

struct ArrayOfStructs { };
struct StructOfArrays { };

// generic "broken on purpose" version
template <class P, class storage_order, class Enable=void>
class ParticleContainer {
private:
   ParticleContainer() { }
};

// StructOfArrays specialization
template<class P>
class ParticleContainer<
```

```
23       P, StructOfArrays,
24       typename boost::enable_if<
25          boost::fusion::traits::is_sequence<P>
26       >::type>
27 {
28 public:
29    typedef typename boost::fusion::result_of::as_vector<
30       typename boost::fusion::result_of::transform<
31          P, vectorify
32       >::type
33    >::type storage_type;
34    typedef P particle_type;
35
36 private:
37    storage_type data_;
38 };
39
40 // ArrayOfStructs specialization
41 template<class P>
42 class ParticleContainer<
43       P, ArrayOfStructs,
44       typename boost::enable_if<
45          boost::fusion::traits::is_sequence<P>
46       >::type>
47 {
48 public:
49    typedef std::vector<P> storage_type;
50    typedef P particle_type;
51
52 private:
53    storage_type data_;
54 };
```

The `enable_if` constructs on lines 22 and 41 make sure that the container can only be instantiated with fusion sequences. Lines 27-31 and 46 are where the storage types are defined. In the array of structs case we simply create a `std::vector` of the particle type. In the struct of arrays we have to do some template/fusion magic. The vectorify functor is a metafunction that we can use with the metafunction algorithm `fusion::transform` analogously to what `std::transform` does. So it transforms a sequence like `fusion::vector<int, float, char>` into `fusion::vector<std::vector<int>, std::vector<float>, std::vector<char> >`, which is exactly what we wanted.

A.2 Making OpenCL slightly less awkward

One of the goals of my bachelor thesis is to design the simulation framework in such a way that it can also use GPUs. I have used both OpenCL and CUDA in the past and figured I would concentrate on OpenCL for now since it allows to generate kernels at run time. Sadly OpenCLs API is very

Cish (12 argument functions, void pointers etc.) and verbose. So naturally the first thing to do is wrapping it in a more C++ way. Since I already wrote wrappers around a lot of OpenGL functionality (see here on github) and OpenCL resembles OpenGL in a lot of ways I am going to orient my self after that. So far only the very bare bones functionality is implemented which is context creation, buffers and kernels. The current version is available at `http://github.com/progschj/clp`. And finally here is a code sample that shows the usage:

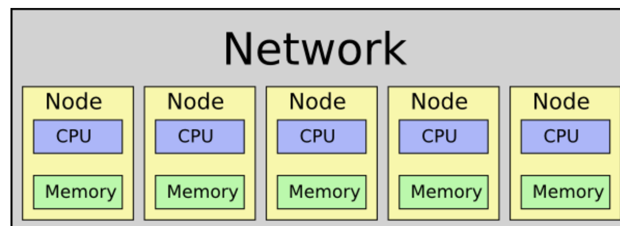
```
1  #include <iostream>
2  #include <algorithm>
3
4  #include "include/CLUtility.h"
5  #include "include/CLEvent.h"
6  #include "include/CLContext.h"
7  #include "include/CLBuffer.h"
8  #include "include/CLProgram.h"
9
10 int main()
11 {
12    // create a context for the second GPU
13    clp::Context context(CL_DEVICE_TYPE_GPU, 1, 1);
14
15    // create and build a program
16    clp::Program program(context);
17    program.setSource(
18    "kernel void saxpy(\n"
19    "   global float *x, global float *y, float a\n"
20    ")\n"
21    "{\n"
22    "   const uint index = get_global_id(0);\n"
23    "   x[index] += a*y[index];\n"
24    "}\n"
25    );
26    program.build();
27
28    // obtain a kernel object
29    clp::Kernel<void(float*,float*,float)> saxpy =
30       program.getKernel<void(float*,float*,float)>("saxpy");
31
32    // create device buffers
33    clp::Buffer<float> x(context, 1024);
34    clp::Buffer<float> y(context, 1024);
35
36    // map the buffers
37    clp::Event xevent = x.map();
38    clp::Event yevent = y.map();
39
40    // fill them with data and unmap
41    xevent.wait();
42    std::fill(x.begin(), x.end(), 45);
43    x.unmap();
44
45    yevent.wait();
```

```
46     std::fill(y.begin(), y.end(), 3);
47     y.unmap();
48 
49     // execute kernel
50     saxpy(clp::Worksize(1024,256), x, y, 13);
51 
52     return 0;
53 }
```

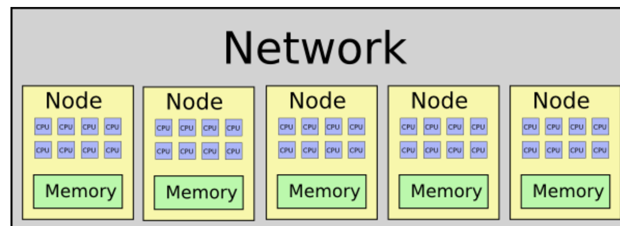
A.3 Why I think pure MPI doesnt cut it anymore

The reason is simple: Because it doesnt represent the underlying hardware very well.

MPI sees a cluster as a bunch of processors each with their own memory and a big fat network connecting them. And even tough MPI itself doesnt assume this, most programs are written under the assumption that all the Nodes are equal in terms of computing power etc. What MPI thinks is happening

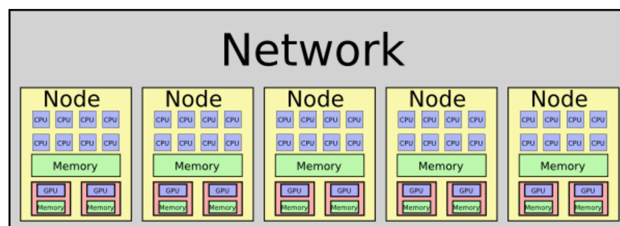


In reality modern clusters use multicore processors. They are often even using multiple multicore processors on a node, all sharing the same memory. We are looking at 12 to 48 cores per node for current systems and that number is most likely going to rise further. What actually happens



And what makes it even worse is that GPUs make their way into cluster architectures. So we even end up with heterogenity inside a single node

(not to mention that GPUs themselves are again massively parallel systems with hundreds or even thousands of cores and local memory). What actually actually happens



So pure MPI clearly doesnt do a good job at representing the actual architecture. It still works as an abstraction. But an abstraction that encourages inefficient use of the underlying hardware. For one thing we might up ending up duplicating shared data across MPI-nodes even tough the actual processes are sitting on the same physical node and could be using the same data. Also the ownership of the GPUs isnt enteirly clear. If there are two GPUs but sixteen cores which cores own the GPUs? can other cores execute code on the GPUs? And if so, how? By sending the required data via MPI to the core owning the CPU, from there down to the GPU and later all the way back?

The most obvious solution is not spawn an MPI process per core. Instead we spawn an MPI process per node and handle the concurrency inside the node via threads. To account for heterogenity we can further use a threadpool instead of locking specific tasks/work items to specific threads/cores inside the node. This approach gives us additional wiggleroom for latency hiding and load balancing. Also the memory consumption should be lower than in a pure MPI configuration since data can be shared between work items and MPI buffers etc. only have to be present once per node instead of once per core.

We get all those benefits for the small cost of handling parallelism on two levels (the MPI and the thread level). How hard that exactely is, is something Im planning to find out in the coming weeks.

A.4 Boosts.MPI and Serialization

Since my project is going to involve MPI parallelization and there is a Boost.MPI library Ill of course have a look at it and most likely also use

it (I really like Boost). One of the main advantages of using Boost.MPI over MPIs C interface is that it is typesafe, more expressive and generally nice to use. Instead of packaging data by hand into buffers before sending them we can have Boost.Serialization handle this task for us.

Lets say we have the following structs that represent one and multidimensional Ranges:

```
template<class T>
struct Range {
   T begin, end;
};
template<class T, int Dim>
struct NRange {
   Range<T>& operator[](int i)
   { return ranges[i]; }
   const Range<T>& operator[](int i) const
   { return ranges[i]; }
private:
   Range<T> ranges[Dim];
};
```

To make them serializable we could put a serialize function as required by Boost.Serialization into their definitions. But since they are already in their own headers etc. that dont depend on Boost serialization we might want to make them serializable in a non intrusive way, which is luckily possible:

```
#include "Range.h"

namespace boost {
namespace serialization {

template<class Archive, class T>
void serialize(Archive & ar, Range<T> & r, const unsigned int version)
{
   ar & r.begin;
   ar & r.end;
}

template<class Archive, class T, int Dim>
void serialize(Archive & ar, NRange<T,Dim> & r, const unsigned int version)
{
   for(int i = 0;i<Dim;++i)
      ar & r[i];
}

} // namespace serialization
} // namespace boost

//...
```

Notice how we dont even have to include any Boost headers since serialize only takes template parameters, so we actually dont even introduce any Boost

dependency here. So since the structs are now serializable we can just send them off with MPI:

```
//...

#include <iostream>
#include <boost/mpi.hpp>

int main(int argc, char* argv[])
{
   mpi::environment env(argc, argv);
   mpi::communicator world;

   if (world.rank() == 0)
   {
      NRange<int,3> range;
      range[0].begin = 23;
      range[0].end = 42;
      world.send(1, 0, range);
   }
   else
   {
      NRange<int,3> range;
      world.recv(0, 0, range);
      std::cout << range[0].begin << ' ' << range[0].end;
   }
   return 0;
}
```

One of the core features of the framework Im working are distributed grids. So naturally we probably want a way to send chunks of the grid to other nodes via MPI. A very simple implementation of a chunk class could look like this:

```
#include <vector>
#include <boost/serialization/vector.hpp>

//...

template<class T>
class Chunk {
public:
    Chunk() { }

    Chunk(const NRange<size_t, 3> &b)
    : bounds_(b), data_(b.volume())
    {
    }

    // subscript operators etc...

private:
    friend class boost::serialization::access;

    template<class Archive>
```

```
22     void serialize(Archive & ar, const unsigned int version)
23     {
24         ar & bounds_;
25         ar & data_;
26     }
27
28     NRange<size_t, 3> bounds_;
29     std::vector<T> data_;
30 };
```

Here we see the intrusive version of putting the serialize function right into class itself. The additional header we included along with vector provides serialization for std::vector so we dont have to do that ourselves. There are a few additional traits we can define which for example allow bitwise copy in some cases or turn of the versioning of the serialization library, but essentially we can now send around Chunks without worrying about their packaging etc.

A.5 A Thread Pool with Boost.Threads and Boost.Asio

After spending some time being frustrated with the C++11 async/future stuff, I dug up some old code and found this nice way of doing a thread pool with Boost.Threads (obviously) and Boost.Asio. Since the code is actually pretty short for what it does Ill just dump it here:

```
1  #include <boost/thread/thread.hpp>
2  #include <boost/asio.hpp>
3
4  class ThreadPool;
5
6  // our worker thread objects
7  class Worker {
8  public:
9      Worker(ThreadPool &s) : pool(s) { }
10     void operator()();
11 private:
12     ThreadPool &pool;
13 };
14
15 // the actual thread pool
16 class ThreadPool {
17 public:
18     ThreadPool(size_t);
19     template<class F>
20     void enqueue(F f);
21     ~ThreadPool();
22 private:
23     // need to keep track of threads so we can join them
24     std::vector< std::unique_ptr<boost::thread> > workers;
25
26     // the io_service we are wrapping
27     boost::asio::io_service service;
```

```
28     boost::asio::io_service::work working;
29     friend class Worker;
30 };
31
32 // all the workers do is execute the io_service
33 void Worker::operator()() { pool.service.run(); }
34
35 // the constructor just launches some amount of workers
36 ThreadPool::ThreadPool(size_t threads) : working(service)
37 {
38     for(size_t i = 0;i<threads;++i)
39         workers.push_back(
40             std::unique_ptr<boost::thread>(
41                 new boost::thread(Worker(*this))
42             )
43         );
44 }
45
46 // add new work item to the pool
47 template<class F>
48 void ThreadPool::enqueue(F f)
49 {
50     service.post(f);
51 }
52
53 // the destructor joins all threads
54 ThreadPool::~ThreadPool()
55 {
56     service.stop();
57     for(size_t i = 0;i<workers.size();++i)
58         workers[i]->join();
59 }
```

Its essentially a wrapper around a `io_service`. The usage then looks something like this:

```
1 // create a thread pool of 4 worker threads
2 ThreadPool pool(4);
3
4 // queue a bunch of "work items"
5 for(int i = 0;i<8;++i)
6 {
7     pool.enqueue([i]
8     {
9         std::cout << "hello " << i << std::endl;
10        boost::this_thread::sleep(
11            boost::posix_time::milliseconds(1000)
12        );
13        std::cout << "world " << i << std::endl;
14    });
15 }
```

which produces a funny mixture of garbled output clearly showing that the lambdas are executed in parallel.

A.6 A Thread Pool with C++11

After showing a simple thread pool with Boost.Asio in the last post im going to have a look at doing the same thing with the threading facilities in C++11. The biggest difference is that we dont have the Asio library so we have to reproduce the relevant functionality ourselves. The declarations remain mostly the same except that the ThreadPool class doesnt have the `io_service` members anymore but instead has a deque and synchronization primitives that we will use instead:

```
#include <thread>
#include <mutex>
#include <condition_variable>

class ThreadPool;

// our worker thread objects
class Worker {
public:
    Worker(ThreadPool &s) : pool(s) { }
    void operator()();
private:
    ThreadPool &pool;
};

// the actual thread pool
class ThreadPool {
public:
    ThreadPool(size_t);
    template<class F>
    void enqueue(F f);
    ~ThreadPool();
private:
    friend class Worker;

    // need to keep track of threads so we can join them
    std::vector< std::thread > workers;

    // the task queue
    std::deque< std::function<void()> > tasks;

    // synchronization
    std::mutex queue_mutex;
    std::condition_variable condition;
    bool stop;
};
```

Previously the Worker threads simply ran the `io_service`. Now they are where most of the magic happens. The most important part here is the `condition_variable` which is used to make the thread sleep when there are no jobs and wake it up when there are new jobs added to the queue. When calling `condition_variable::wait` with a lock the lock is released

and the thread is suspended. When `condition_variable::notify_one` or `condition_variable::notify_all` is called one or all waiting threads are woken up and reacquire the lock.

```
void Worker::operator()()
{
    std::function<void()> task;
    while(true)
    {
        {   // acquire lock
            std::unique_lock<std::mutex>
                lock(pool.queue_mutex);

            // look for a work item
            while(!pool.stop && pool.tasks.empty())
            { // if there are none wait for notification
                pool.condition.wait(lock);
            }

            if(pool.stop) // exit if the pool is stopped
                return;

            // get the task from the queue
            task = pool.tasks.front();
            pool.tasks.pop_front();

        }   // release lock

        // execute the task
        task();
    }
}
```

Constructor and destructor mostly remain the same. The destructor now uses `notify_all` to make sure any suspended threads see that the stop flag is set.

```
// the constructor just launches some amount of workers
ThreadPool::ThreadPool(size_t threads)
    :   stop(false)
{
    for(size_t i = 0;i<threads;++i)
        workers.push_back(std::thread(Worker(*this)));
}

// the destructor joins all threads
ThreadPool::~ThreadPool()
{
    // stop all threads
    stop = true;
    condition.notify_all();

    // join them
    for(size_t i = 0;i<workers.size();++i)
        workers[i].join();
```

```
19 }
```

Finally the enqueue function just locks the queue, adds a task to it and wakes up one thread in case any thread was suspended.

```
1  // add new work item to the pool
2  template<class F>
3  void ThreadPool::enqueue(F f)
4  {
5      { // acquire lock
6          std::unique_lock<std::mutex> lock(queue_mutex);
7
8          // add the task
9          tasks.push_back(std::function<void()>(f));
10     } // release lock
11
12     // wake up one thread
13     condition.notify_one();
14 }
```

The interface of the ThreadPool is unchanged, so the usage example from the last blog post still works. This version of the ThreadPool is slightly longer than the version with Boost.Asio but actually still relatively short for what it does and reduces the Boost dependencies since we now dont have to link Boost libraries anymore.