STE||AR

stellar.cct.lsu.edu

HPX - Workshop

A GENERAL PURPOSE C++ RUNTIME SYSTEM FOR PARALLEL AND DISTRIBUTED APPLICATIONS OF ANY SCALE

Getting Started

HOW TO INSTALL HPX

Getting Started

How to install

- Prerequisites (recommended, http://stellar-group.github.io/hpx/docs/html/hpx/tutorial/getting_started/prereqs.html)
  - GNU Compiler Collection (g++) V4.6.3 or higher
  - Cmake V2.8.4 or higher
  - Boost V1.49 or higher
  - HWLOC V1.7 or higher

How to build HPX

- Use script: https://github.com/STEllAR-GROUP/hpx/blob/bootstrap_bash_script/tools/hpx_bootstrap.sh

  hpx_booststrap –d <target-dir>
- Needs:
  - Bash (>= V4.0), GAWK (>= V3.8), Wget (>= V1.0), GNU Make (>= V3.5), Git (>= V1.7), Tar (>= V1.23) and Python (>= V2.6)

Download and install HPX manually

- http://stellar-group.github.io/hpx/docs/html/hpx/tutorial/getting_started/unix_installation.html
- Git tag for this workshop: 20140522.workshop (https://github.com/STEllAR-GROUP/hpx/tree/20140522.workshop)

Getting Started

How to build an application

- Using pgk_config:
  http://stellar-group.github.io/hpx/docs/html/hpx/manual/cmake/unix_apps_pkg_config.html
  - Usable with any build system:

    export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$HPX_LOCATION/lib/pkgconfig

    c++ -o hello_world hello_world.cpp `pkg-config --cflags --libs hpx_application` -liostreams -DHPX_APPLICATION_NAME=hello_world

- Using CMake:
  http://stellar-group.github.io/hpx/docs/html/hpx/manual/cmake/existing_apps.html
  - Example with minimal build system can be found at: https://github.com/STEllAR-GROUP/hpx_external_example

Some Simple Examples

A CLOSER LOOK

Hello HPX World

```
#include <hpx/hpx.hpp>
#include <hpx/hpx_init.hpp>
#include <hpx/iostream.hpp>

int hpx_main()
{
    hpx::cout << "Hello HPX World!\n";
    return hpx::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::init(argc, argv);  // pass along command line arguments
}
```

Hello HPX World

HPX applications start with executing `hpx_main`

- Different signatures possible

```
int hpx_main();
int hpx_main(int argc, char* argv[]);
int hpx_main(boost::program_options::variables_map& vm);
```

- By default `hpx_main` is executed on main locality only
  - It is possible to force execution on all localities (easiest: --hpx:run-hpx-main)

HPX applications support many predefined command line options

- All predefined command line options follow scheme: --hpx:<opt> [arg]
- Get a list by invoking: ‘app --hpx:help’ (or simply ‘app -h’)
- See: http://stellar-group.github.io/hpx/docs/html/hpx/manual/init/commandline.html

Hello HPX World

HPX is initialized by invoking `hpx::init()`

- Must be called on all localities
- This will
  - Initialize the runtime system
  - Register every locality with the main locality
  - Schedule `hpx_main()` as the first HPX thread
  - Wait for the whole system to shutdown (triggered by `hpx::finalize()`)

HPX is told to (gracefully) shut down by invoking `hpx::finalize()`

- Can be called on any locality, first invocation triggers exit (must be called at least once)
- This will
  - Wait for all activity (system wide) to cease
  - Releases main threads waiting inside `hpx::init()`

Hello HPX World

```
void say_hello()
{
    hpx::cout << "Hello HPX World from locality: " <<
              << hpx::get_locality_id() << "!\n";
}
HPX_PLAIN_ACTION(say_hello);   // defines say_hello_action

int hpx_main()
{
    say_hello_action sayit;
    for (auto loc: hpx::find_all_localities())
        hpx::apply(sayit, loc);
    return hpx::finalize();
}
```

Hello HPX World

HPX applications can be run in different ways

- Using PBS:

  echo "pbsdsh -u <hpx-app> <app-options> --hpx:nodes=`cat $PBS_NODEFILE`" > ./pbs_script.sh

  qsub <qsub-options> ./pbs_script.sh
- Using SLURM:

  srun <srun-options> <hpx-app> <app-options>
- Using MPI (if MPI parcel-port is enabled)

  mpiexec <mpiexec-options> <hpx-app> <app-options>
- Locally:

  <hpx-app> <app-options> -0 --hpx:localities=2 &

  <hpx-app> <app-options> -1

Hello HPX World

```
int hpx_main()
{
   std::vector<hpx::future<void> > ops;
   say_hello_action sayit;

   for (auto loc: hpx::find_all_localities())
       ops.push_back(hpx::async(sayit, loc));

   hpx::wait_all(ops);
   return hpx::finalize();
}
```

1D Heat Diffusion

A STEP BY STEP CASE STUDY

Parallelization on SMP

Example 1: 1D Heat Equation

Kernel: simple iterative heat diffusion solver, 3 point stencil

```
double heat(double left, double middle, double right)
{
    return middle + (k*dt/dx*dx) * (left - 2*middle + right);
}
```

Time step iteration, periodic boundary conditions:

```
std::vector<std::vector<double>> U(2, std::vector<double>(nx));
for (std::size_t t = 0; t != nt; ++t)
{
    std::vector<double> const& current = U[t % 2];
    std::vector<double>& next = U[(t + 1) % 2];

    next[0] = heat(current[nx-1], current[0], current[1]);
    for (std::size_t i = 1; i != nx-1; ++i)
        next[i] = heat(current[i-1], current[i], current[i+1]);
    next[nx-1] = heat(current[nx-2], current[nx-1], current[0]);
}
```

Example 1: 1D Heat Equation

Reference performance

- System: 12 core Nehalem (2*6 cores, 2 hyper-threads/core, 3GHz), 3 memory channels
- Serial execution for 100 million data points (nx), 45 time steps (nt): ~16s

Example 1: 1D Heat Equation, OpenMP

Adding OpenMP directive:

```
std::vector<std::vector<double>> U(2, std::vector<double>(nx));
for (std::size_t t = 0; t != nt; ++t)
{
    std::vector<double> const& current = U[t % 2];
    std::vector<double>& next = U[(t + 1) % 2];

    next[0] = heat(current[nx-1], current[0], current[1]);

    #pragma omp parallel for
    for (std::size_t i = 1; i != nx-1; ++i)
        next[i] = heat(current[i-1], current[i], current[i+1]);

    next[nx-1] = heat(current[nx-2], current[nx-1], current[0]);
}
```

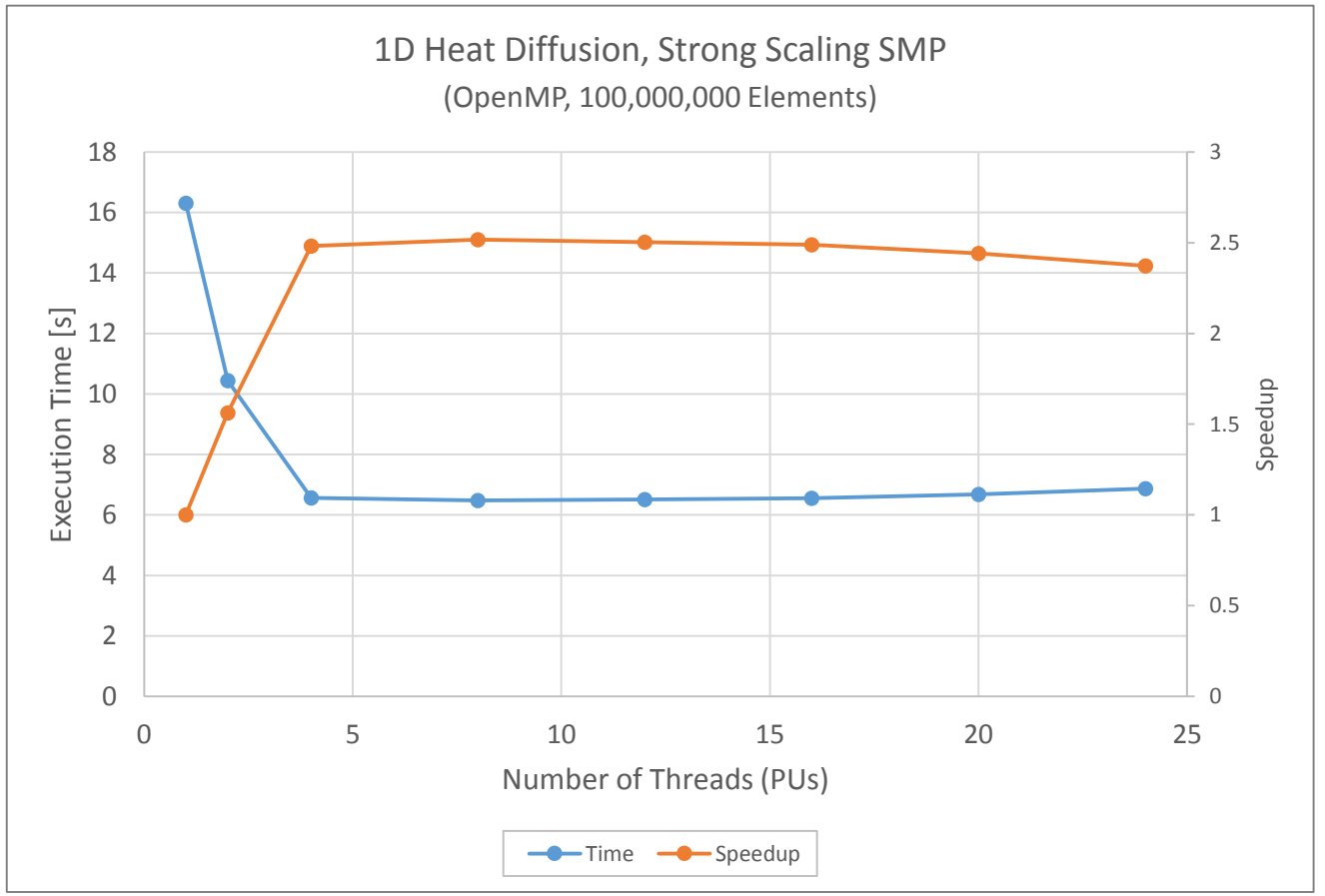
Example 1: 1D Heat Equation, OpenMP

Adding some typedefs:

```
typedef double partition;
typedef std::vector<partition> space;

std::vector<space> U(2, space(nx));
for (std::size_t t = 0; t != nt; ++t)
{
    space const& current = U[t % 2];
    space& next = U[(t + 1) % 2];

    next[0] = heat(current[nx-1], current[0], current[1]);

    #pragma omp parallel for
    for (std::size_t i = 1; i != nx-1; ++i)
        next[i] = heat(current[i-1], current[i], current[i+1]);

    next[nx-1] = heat(current[nx-2], current[nx-1], current[0]);
}
```

Example 1: 1D Heat Equation, OpenMP

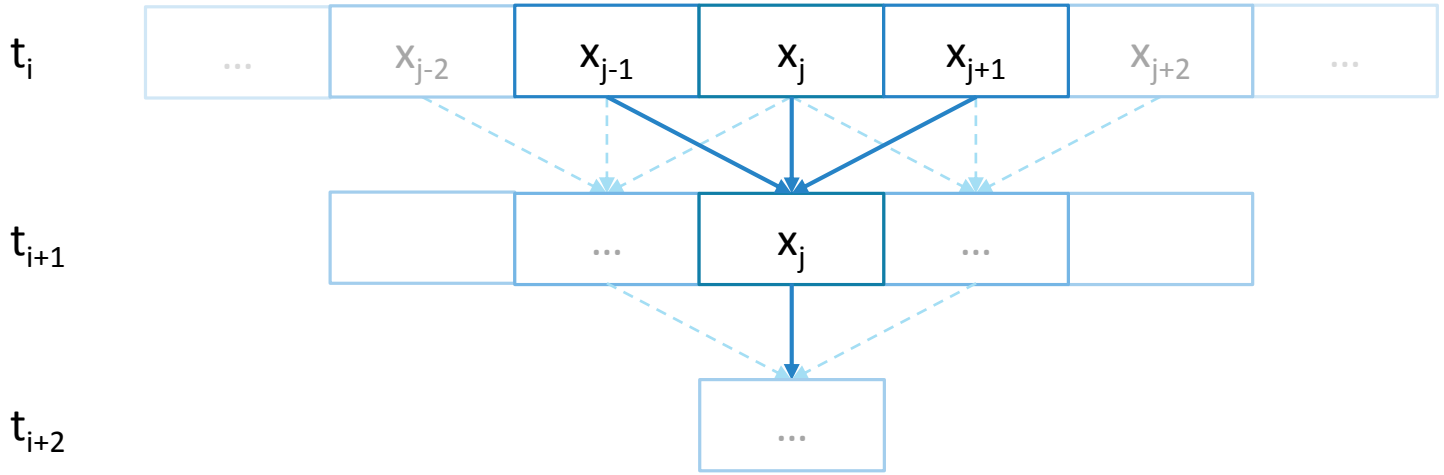
Results:



Example 2: 1D Heat Equation, HPX

Apply futurization

- Each data point is represented by a future
- Stencil is expressed as data dependencies between futures



Example 2: 1D Heat Equation, HPX

```
typedef future<double> partition;
typedef std::vector<partition> space;

std::vector<space> U(2, space(nx));
for (std::size_t t = 0; t != nt; ++t)
{
    space const& current = U[t % 2];
    space& next = U[(t + 1) % 2];

    for (std::size_t i = 0; i != nx; ++i)
    {
        next[i] = dataflow(
                launch::async, unwrapped(&heat),
                current[idx(i-1, nx)], current[i], current[idx(i+1, nx)]
            );
    }
}
```

HPX Dataflow Construct

The HPX dataflow construct is receiving

- A set of futures
- A function to execute once all futures become ready
- A (optional) launch policy which determines how the function will be executed

The function is expected to receive all futures as separate arguments

The hpx::`unwrapped()` facility wraps a function and 'unwraps' all futures which are passed to that function (calls `future::get()`)

- No overheads as we know that at that point all futures have become ready

Example 2: 1D Heat Equation, HPX

Results:

- Very bad! ☹
- About 500 times slower than serial code, very high memory requirements

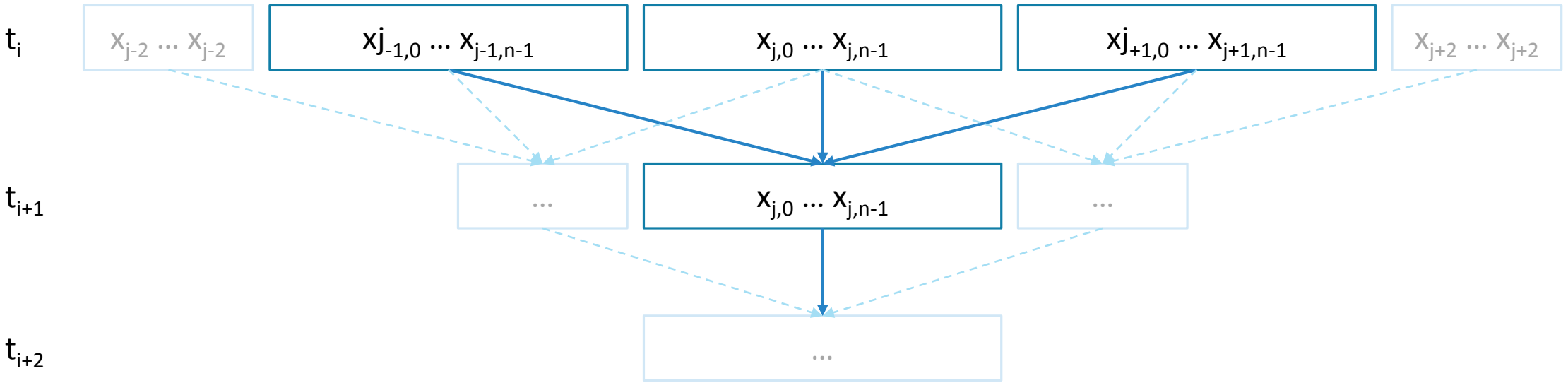
Too large overheads introduced by futurization

- Amount of work performed for each future is comparably small to overheads introduced

Need a way to control amount of work represented by one future (one thread)

Example 3: 1D Heat Equation, Partitioned

Same kernel as before, however partitioning of the overall data array into equally sized chunks



Same data dependencies as before, just representing the results of blocks of data points

Special care must be taken with regard to boundary points

Example 3: 1D Heat Equation, Partitioned

New partition type:

```
struct partition_data
{
    partition_data(std::size_t size = 0)
      : data_(size)
    {}

    double& operator[](std::size_t idx) { return data_[idx]; }
    double operator[](std::size_t idx) const { return data_[idx]; }

    std::size_t size() const { return data_.size(); }

private:
    std::vector<double> data_;
};
```

Example 3: 1D Heat Equation, Partitioned

New operator for calculating heat diffusion based on partitions:

```
partition_data heat_part(partition_data const& left, partition_data const& middle,
    partition_data const& right)
{
    std::size_t size = middle.size();
    partition_data next(size);

    next[0] = heat(left[size-1], middle[0], middle[1]);

    for (std::size_t i = 1; i != size-1; ++i)
        next[i] = heat(middle[i-1], middle[i], middle[i+1]);

    next[size-1] = heat(middle[size-2], middle[size-1], right[0]);

    return next;
}
```

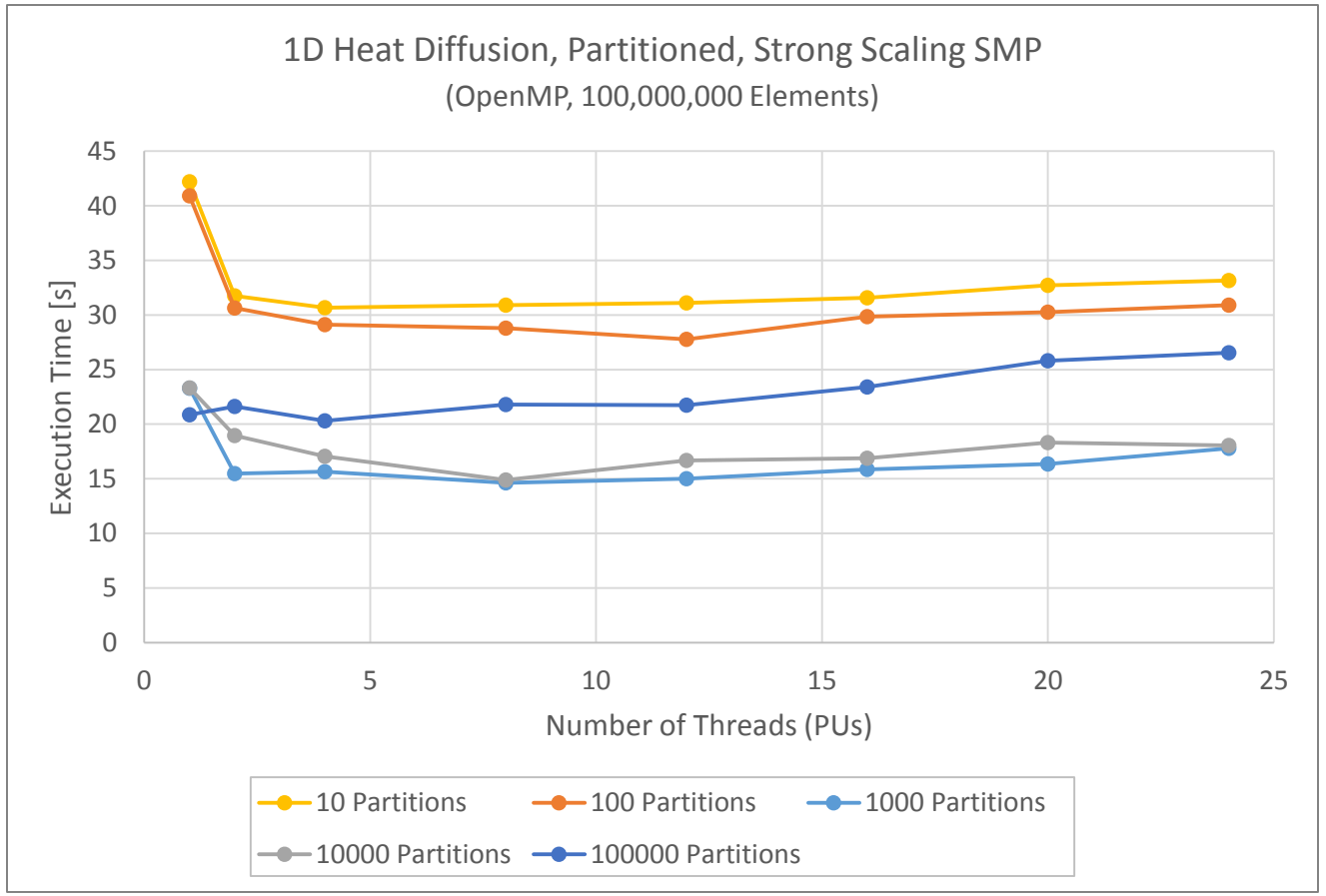
Example 3: 1D Heat Equation, Partitioned

Main loop looks (almost) the same as example 1:

```
typedef std::vector<partition_data> space;

// np: number of partitions.
std::vector<space> U(2, space(np));
for (std::size_t t = 0; t != nt; ++t)
{
    space const& current = U[t % 2];
    space& next = U[(t + 1) % 2];

    for (std::size_t i = 0; i != np; ++i)
        next[i] = heat_part(current[idx(i-1, np)], current[i], current[idx(i+1, np)]);
}
```

Example 3: OpenMP, Partitioned

Results:



Example 3: 1D Heat Equation, Partitioned

Results:

- Serial execution for 100 thousand data points (nx), 1000 partitions (np), 45 time steps (nt):

  ~24s (vs. example 1: ~16s)

- Additional overheads
- No significant advantage of using OpenMP in this case

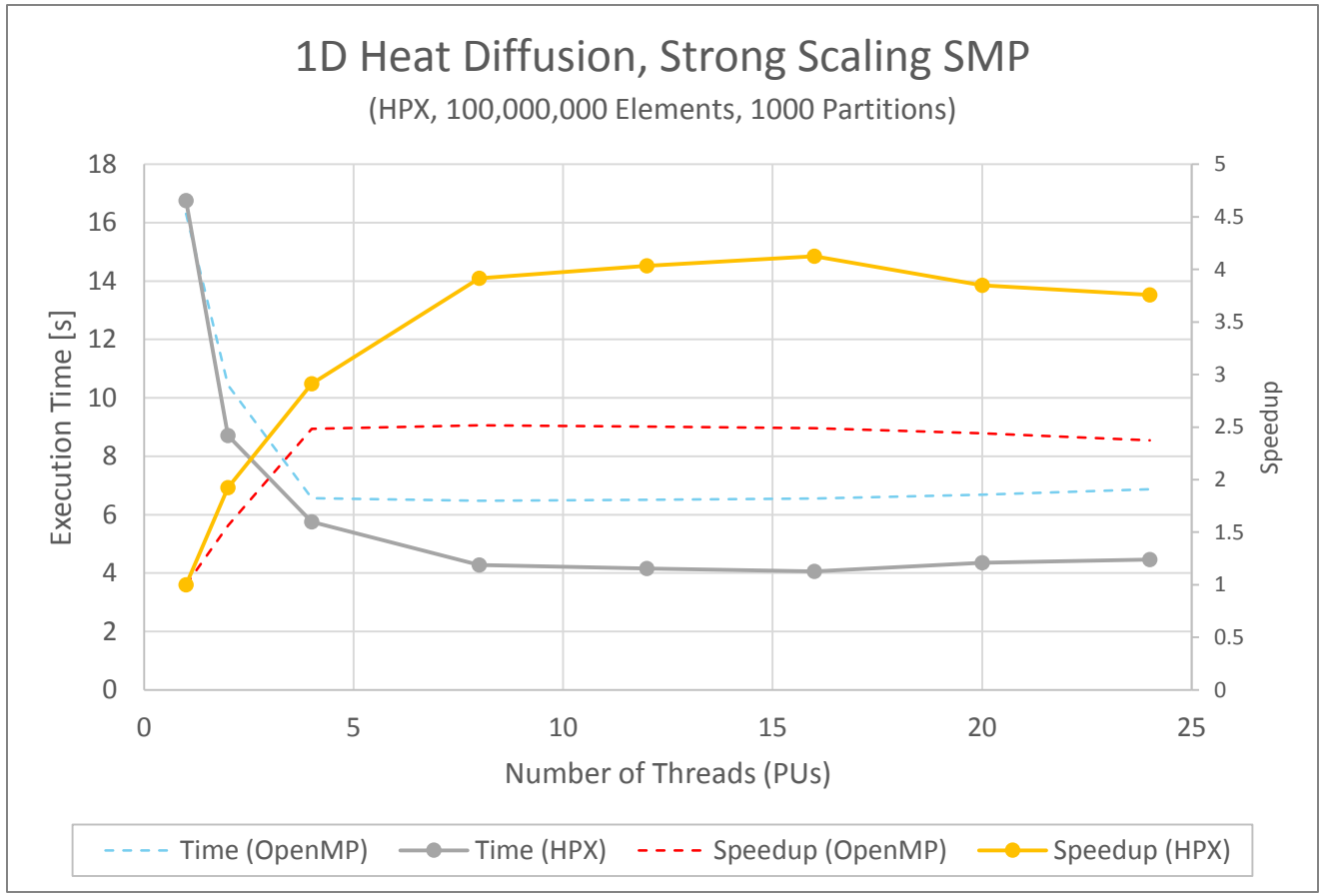
Example 4: Partitioned, HPX

Same `heat_part()` kernel as example 3, same futurization technique as in example 2:

```
typedef shared_future<partition_data> partition;
typedef std::vector<partition> space;

std::vector<space> U(2, space(np));
for (std::size_t t = 0; t != nt; ++t)
{
    space const& current = U[t % 2];
    space& next = U[(t + 1) % 2];

    for (std::size_t i = 0; i != np; ++i)
    {
        next[i] = dataflow(
                launch::async, unwrapped(&heat_part),
                current[idx(i-1, np)], current[i], current[idx(i+1, np)]
            );
    }
}
```

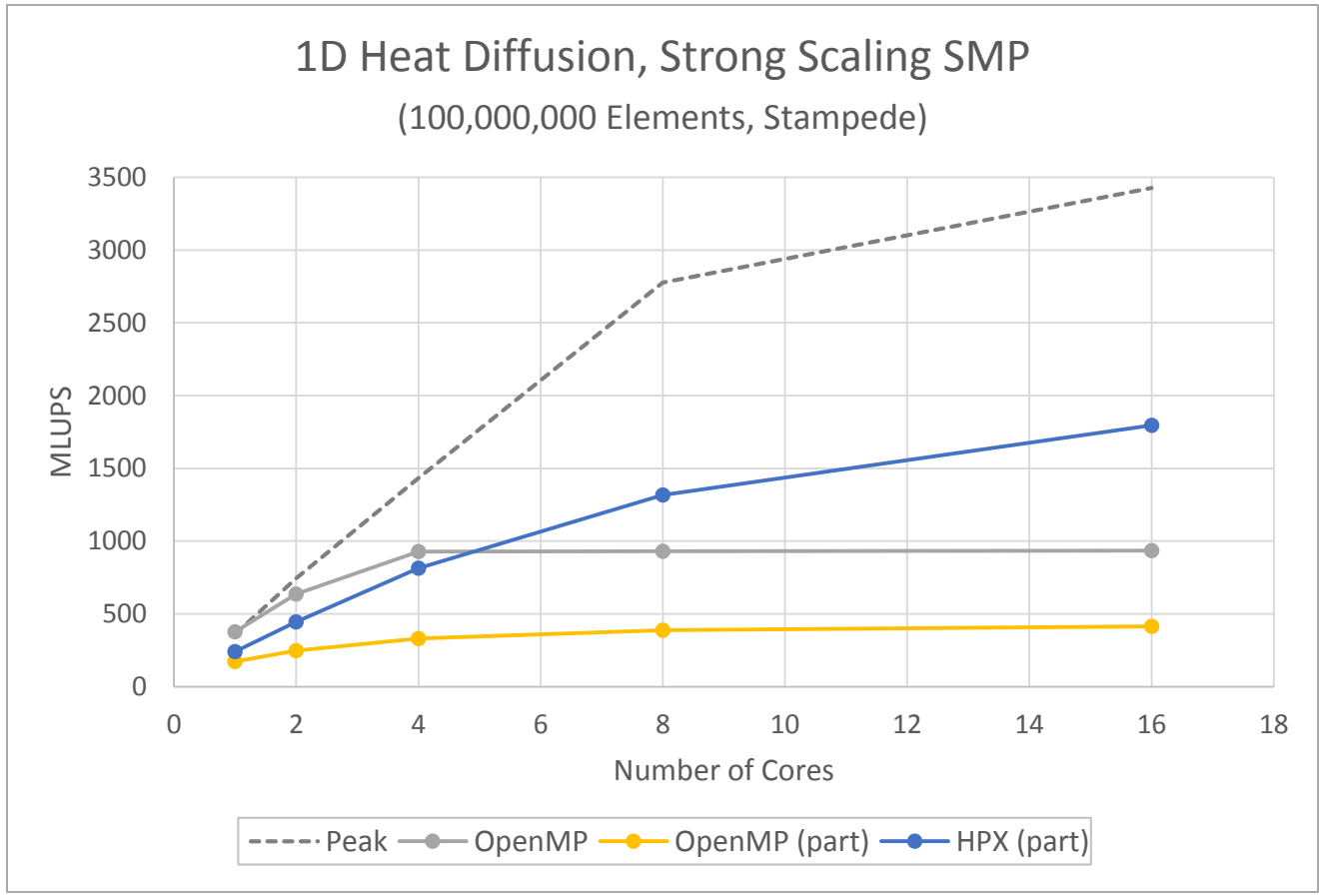
Example 4: Partitioned, HPX

Results:



Example 4: Partitioned, HPX

Efficiency:



Parallelization on Cluster

Remote Function Invocation

HPX allows to invoke any (C/C++) function remotely

- No explicit data transfer functionality
  - Data is passed on as arguments to (remote) functions
  - Data is returned as return values from (remote) functions
- Remote functions are encapsulated as HPX actions
  - Function objects encapsulating the function invocation
- Actions are invoked on a target
  - Target is a global id referencing the destination
  - If target is local → new thread is created
  - If target is remote → parcel is sent (message) which creates a thread on the locality hosting the target

AGAS – Active Global Address Space

HPX spans a global address space encompassing all localities which constitute the application
- Non-cache coherent, no consistency is enforced, relies on C++11 memory model
- Special operations allow atomic access to globally exposed local memory regions

Addresses are global identifiers (GIDs) represented by `hpx::id_type`
- Think: 128 bit void*

GIDs are assigned dynamically at runtime, but they never change, even if the target is moved (migrated) to a different locality
- Programmer does not need to worry whether a target is local or not

GIDs can be passed to other localities

GIDs are globally garbage collected
- Programmer does not have to worry about freeing objects which are referenced by GIDs

Currently the programmer decides which objects are globally visible, i.e. have GID assigned

Plain Actions

Plain actions wrap global (or class-static) functions

- Target is a GID referencing a locality:

```
void say_hello()
{
    hpx::cout << "Hello HPX World from locality: " <<
              << hpx::get_locality_id() << "!\n";
}
HPX_PLAIN_ACTION(say_hello);   // defines say_hello_action

int hpx_main()
{
    say_hello_action sayit;
    for (hpx::id_type loc: hpx::find_all_localities())
        hpx::apply(sayit, loc);
    return hpx::finalize();
}
```

Fibonacci Number Sequence

```
int fibonacci(int n);

HPX_PLAIN_ACTION(fibonacci);   // defines fibonacci_action
fibonacci_action fib;

int fibonacci(int n)
{
    if (n < 2) return n;

    hpx::id_type loc = hpx::find_here();
    hpx::future<int> f = hpx::async(fib, loc, n-1);

    int r = fib(loc, n-2);
    return f.get() + r;
}
```

Component Actions

HPX components are C++ objects which are remotely accessible

HPX component actions wrap member functions

- Target is a GID referencing a C++ object instance (component instance)

HPX components need to follow certain implementation rules:

```
struct hello_world_server : hpx::components::simple_component_base<hello_world_server>
{
    void print() const { cout << "hello world\n" << flush; }

    HPX_DEFINE_COMPONENT_CONST_ACTION(hello_world_server, print); // defines print_action
};

HPX_REGISTER_MINIMAL_COMPONENT_FACTORY(
    hpx::components::simple_component<hello_world_server>, hello_world_server);

HPX_REGISTER_ACTION(hello_world_server::print_action);
```

Component Actions

Create a component instance:

```
hpx::future<hpx::id_type> id = hpx::new_<hello_world_server>(hpx::find_here());
```

Access a component instance:

```
print_action act;
hpx::future<void> f = hpx::async(act, id.get());
```

Delete a component instance:

```
// do nothing :-P
```

Using Client Objects for Components

Disadvantage of hpx::id_type is that is looses type information (it's essentially a void*)

Solution is to rewrap hpx::id_type into a type safe client side object

- Client_base wraps a future<id_type>

```
struct hello_world : hpx::components::client_base<hello_world, hello_world_server>
{
    typedef hpx::components::client_base<hello_world, hello_world_server> base_type;
    hello_world(hpx::future<hpx::id_type> && id) : base_type(std::move(id)) {}

    void print() { hello_world_server::print_action()(this->get_gid()); }
};
```

Enabling this:

```
hello_world hw = hello_world::create(find_here());
hw.print();
```

Example 5: Partitioned, Distributed, HPX

Let’s start with the simplest possible solution: leave execution flow on main locality

Introduce component representing partition_data:

```
struct partition_server : hpx::components::simple_component_base<partition_server>
{
    // construct new instances
    partition_server() {}
    partition_server(partition_data const& data) : data_(data) {}

    // access data
    partition_data get_data() const { return data_; }
    HPX_DEFINE_COMPONENT_CONST_DIRECT_ACTION(partition_server, get_data, get_data_action);

private:
    partition_data data_;
};
```

Example 5: Partitioned, Distributed, HPX

And a partition client object:

```
struct partition : hpx::components::client_base<partition, partition_server>
{
    typedef hpx::components::client_base<partition, partition_server> base_type;

    partition() {}
    partition(hpx::id_type where, partition_data && data)
      : base_type(hpx::new_colocated<partition_server>(where, std::move(data)))
    {}
    // other constructors ...

    hpx::future<partition_data> get_data() const
    {
        return hpx::async(get_data_action(), get_gid());
    }
};
```

Example 5: Partitioned, Distributed, HPX

Function heat_part should be executed on the locality where the middle partition lives:

```
partition heat_part_remote(partition const& left, partition const& middle,
    partition const& right)
{
    return lcos::local::dataflow(
        util::unwrapped(
            [middle](partition_data const& l, partition_data const& m, partition_data const& r)
            {
                // The new partition_data will be allocated on the same locality as 'middle'.
                return partition(middle.get_gid(), heat_part(l, m, r));
            }
        ),
        left.get_data(),
        middle.get_data(),
        right.get_data());
}

HPX_PLAIN_ACTION(heat_part_remote, heat_part_action);   // Wrap into an action
```

Example 4: Partitioned, HPX

Same `heat_part()` kernel as example 3, same futurization technique as in example 2:

```
typedef shared_future<partition_data> partition;
typedef std::vector<partition> space;

std::vector<space> U(2, space(np));
for (std::size_t t = 0; t != nt; ++t)
{
    space const& current = U[t % 2];
    space& next = U[(t + 1) % 2];

    for (std::size_t i = 0; i != np; ++i)
    {
        next[i] = dataflow(
                launch::async, util::unwrapped(&heat_part),
                current[idx(i-1, np)], current[i], current[idx(i+1, np)]
            );
    }
}
```

Example 5: Partitioned, Distributed, HPX

Same `heat_part()` kernel as example 4

```
typedef std::vector<partition> space;
auto Op = util::bind(heat_part_action(), loc, _1, _2, _3);

std::vector<space> U(2, space(np));
for (std::size_t t = 0; t != nt; ++t)
{
    space const& current = U[t % 2];
    space& next = U[(t + 1) % 2];

    for (std::size_t i = 0; i != np; ++i)
    {
        next[i] = dataflow(
                launch::async, Op,
                current[idx(i-1, np)], current[i], current[idx(i+1, np)]
            );
    }
}
```

Example 8: Partitioned, Distributed, HPX

Execute hpx_main() on all localities

- Simplest is to pass --hpx:run-hpx-main on command line
- Alternative programmatic solution:

```
int main(int argc, char* argv[])
{
    // Initialize and run HPX, this example requires to run hpx_main on all localities
    std::vector<std::string> cfg;
    cfg.push_back("hpx.run_hpx_main!=1");

    return hpx::init(argc, argv, cfg);
}
```

Example 8: Partitioned, Distributed, HPX

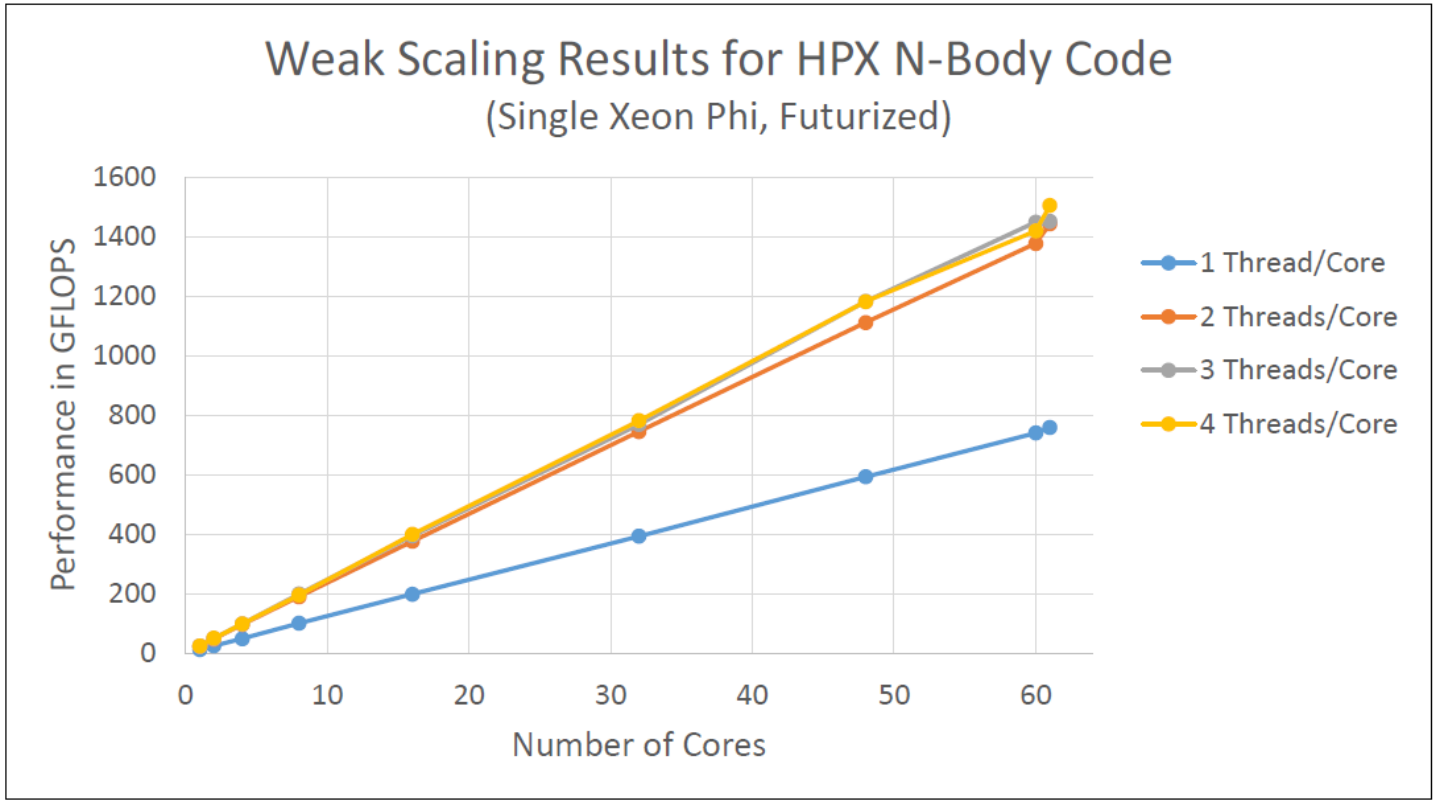
What's left is to distribute the execution flow

- Let every locality work on the partitions located there, do boundary exchange with neighboring localities

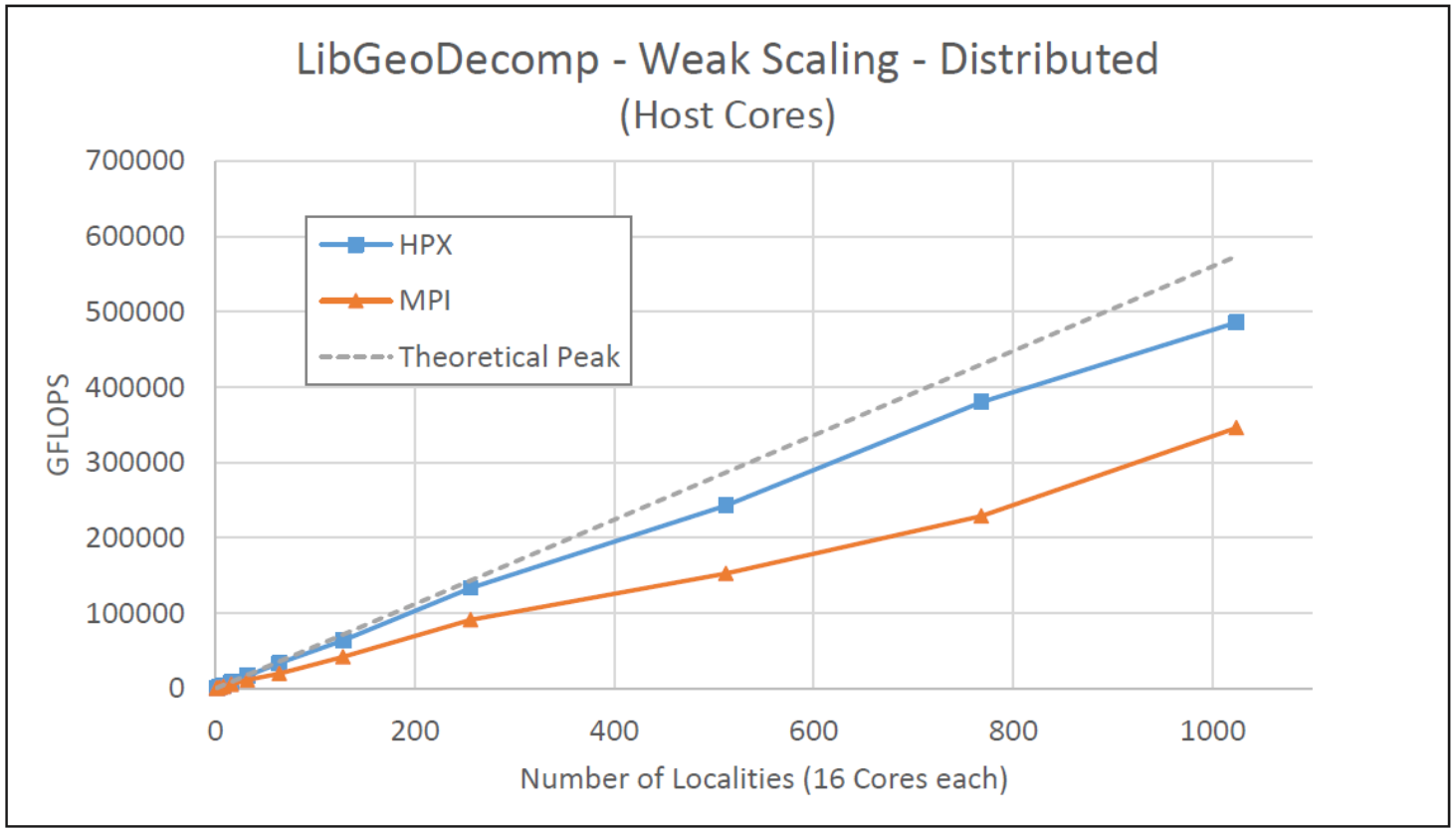
```
for (std::size_t t = 0; t != nt; ++t)
{
    space const& current = U_[t % 2];
    space& next = U_[(t + 1) % 2];

    next[0] = dataflow(hpx::launch::async, &heat_part,
        receive_left(t), current[0], current[1]);
    send_right(t, next[0]);

    for (std::size_t i = 1; i != local_np-1; ++i) {
        next[i] = dataflow(hpx::launch::async, &heat_part,
            current[i-1], current[i], current[i+1]);
    }

    next[local_np-1] = dataflow(hpx::launch::async, &heat_part,
        current[local_np-2], current[local_np-1], receive_right(t));
    send_left(t, next[local_np-1]);
}
```

Recent Results

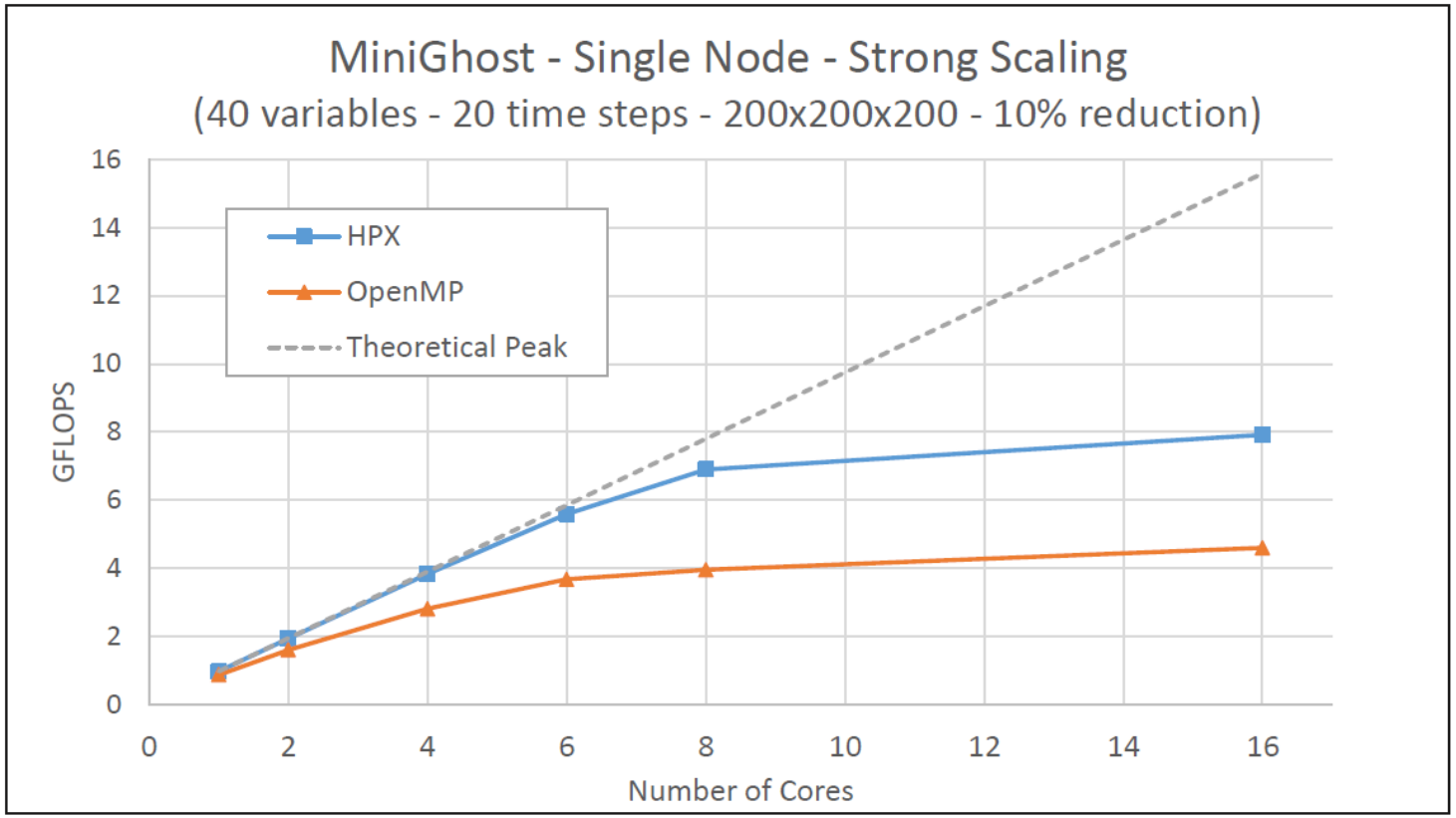
N-Body Code based on LibGeoDecomp



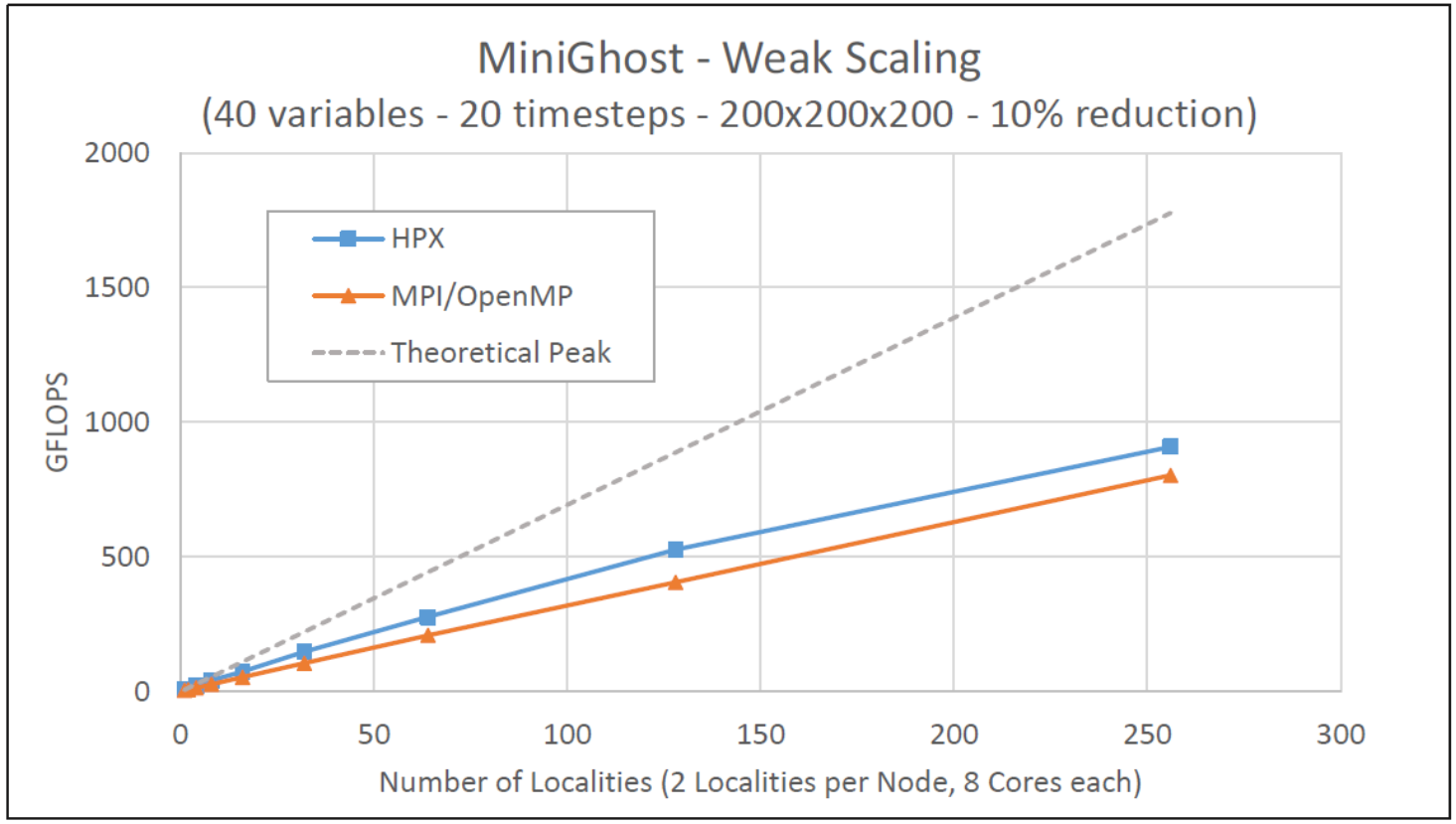
N-Body Code based on LibGeoDecomp



Mini-Ghost (SMP)



Mini-Ghost (distributed runs)



LSU CENTER FOR COMPUTATION & TECHNOLOGY