HPX:
A C++11 Distributed Runtime System

Bryce Adelstein-Lelbach, Hartmut Kaiser, Matthew Anderson
Louisiana State University, Indiana University

stellar.cct.lsu.edu

THE BIGGER PICTURE

Technology Demands New Responses

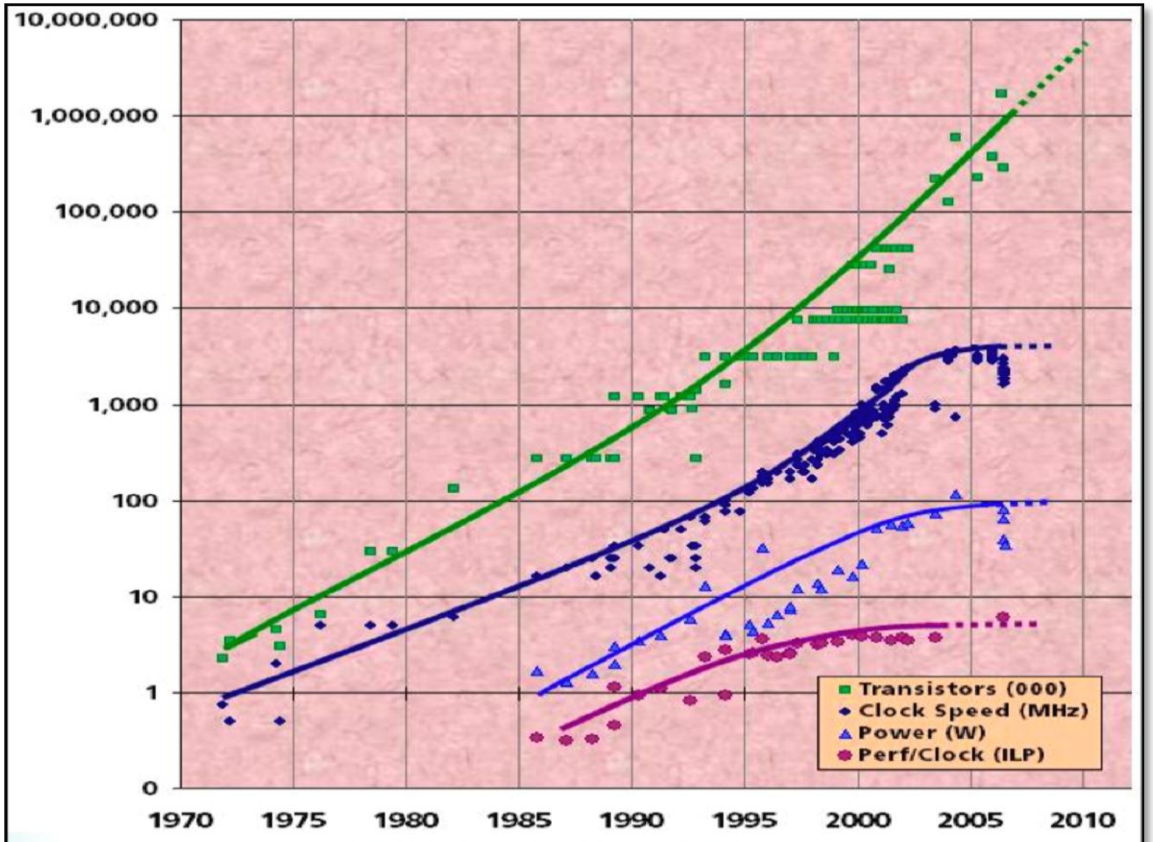


Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith

Technology Demands New Responses

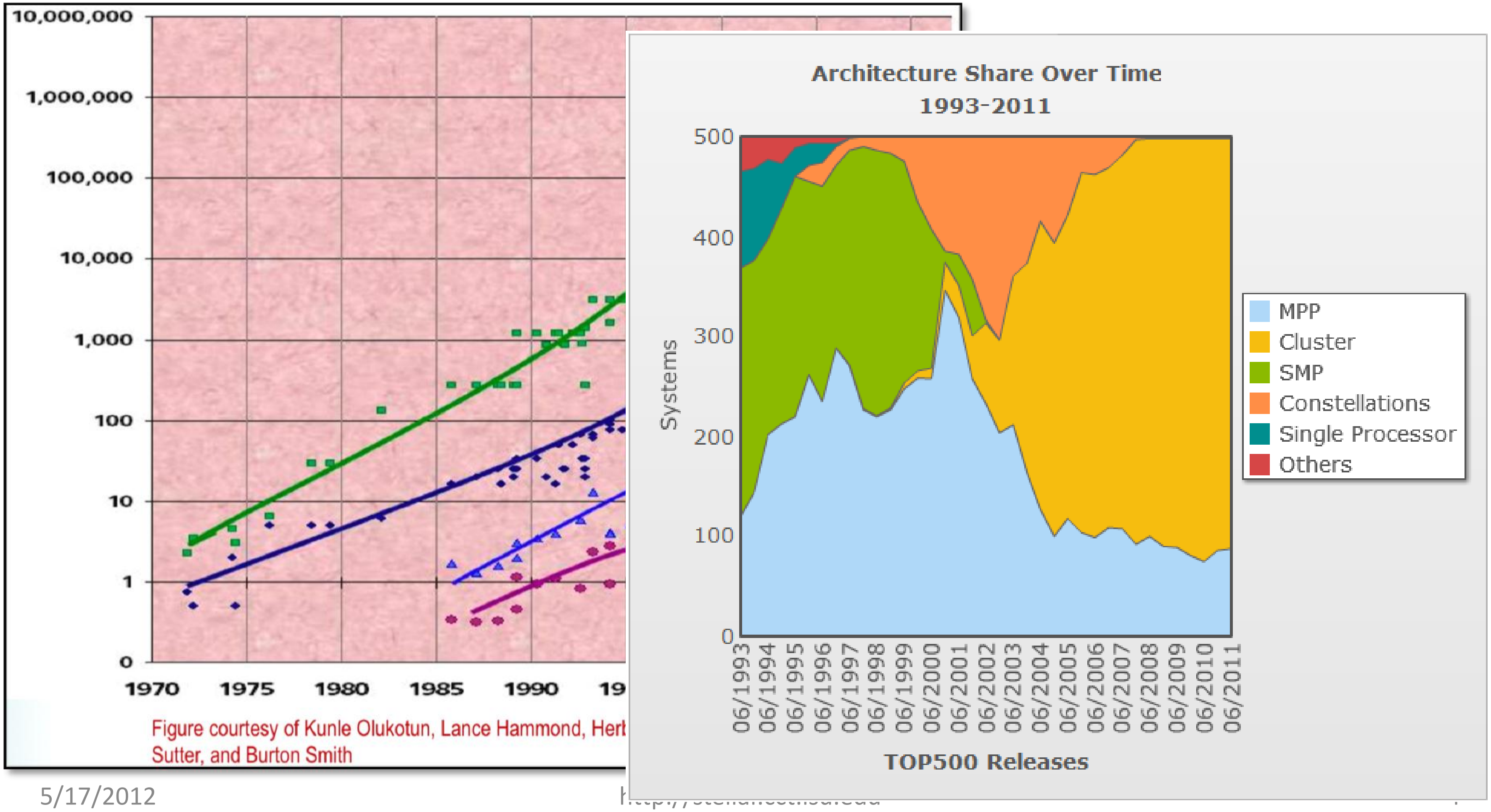


Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith

Technology Demands New Responses

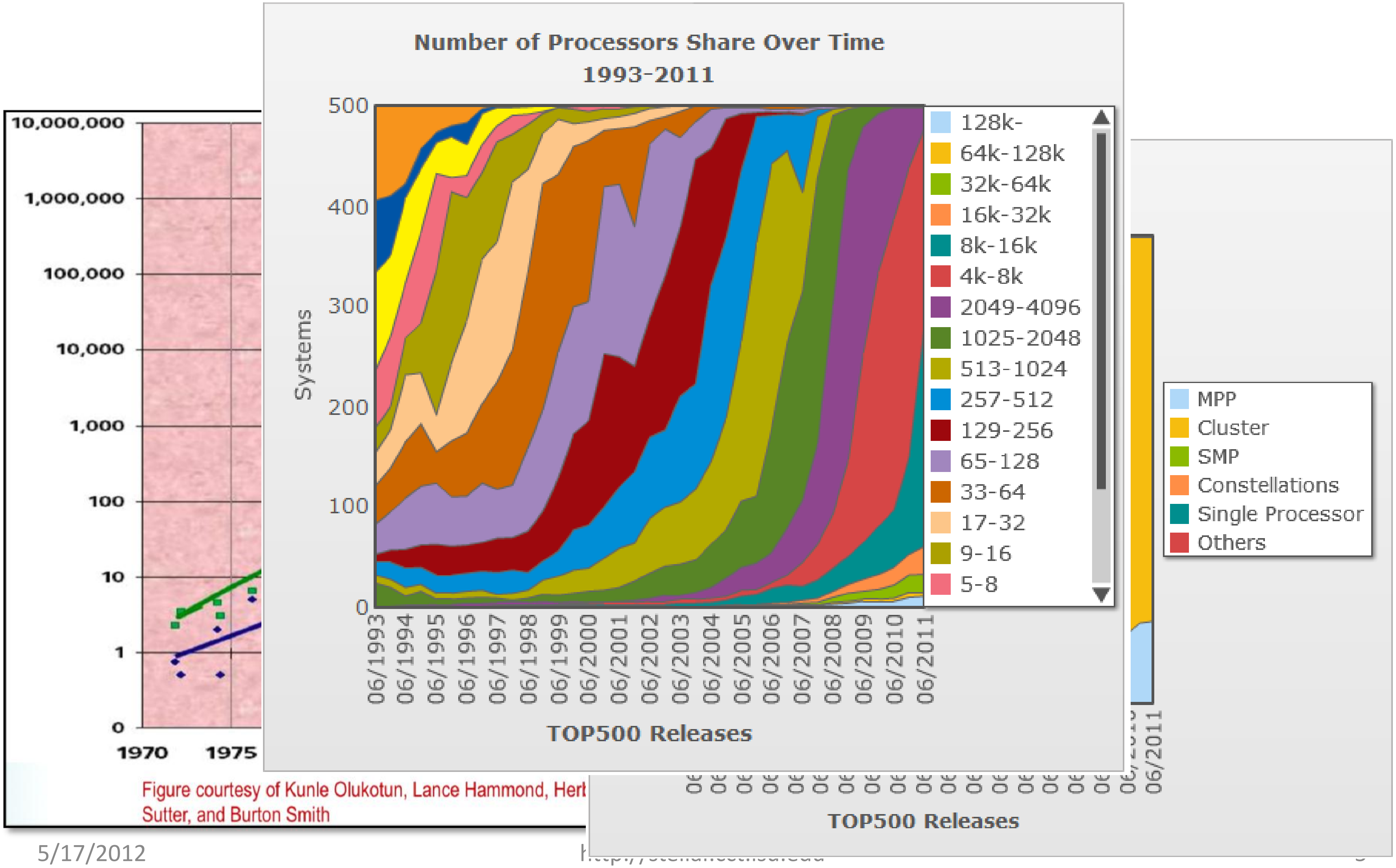


Figure courtesy of Kunle Olukotun, Lance Hammond, Hert
Sutter, and Burton Smith

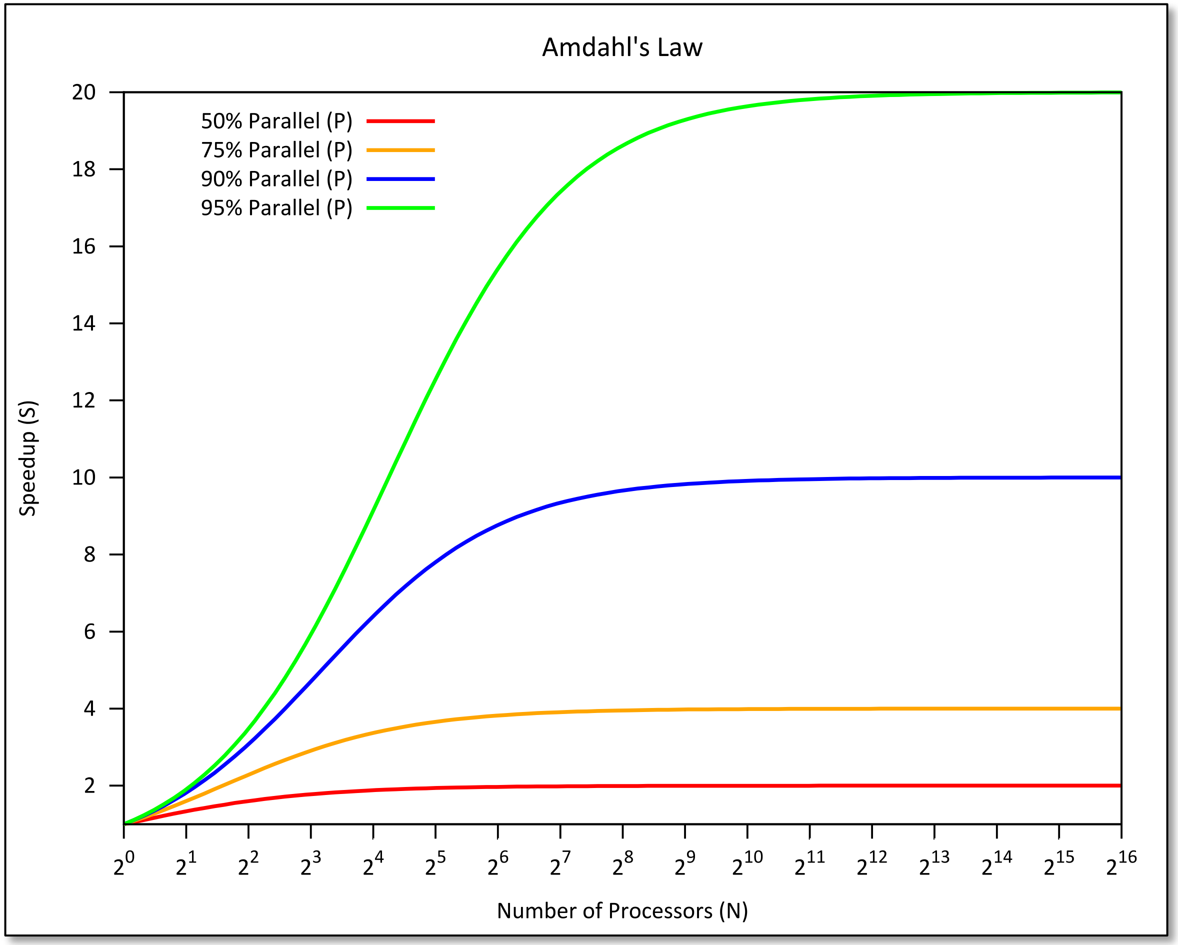
Amdahl’s Law (Strong Scaling)

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

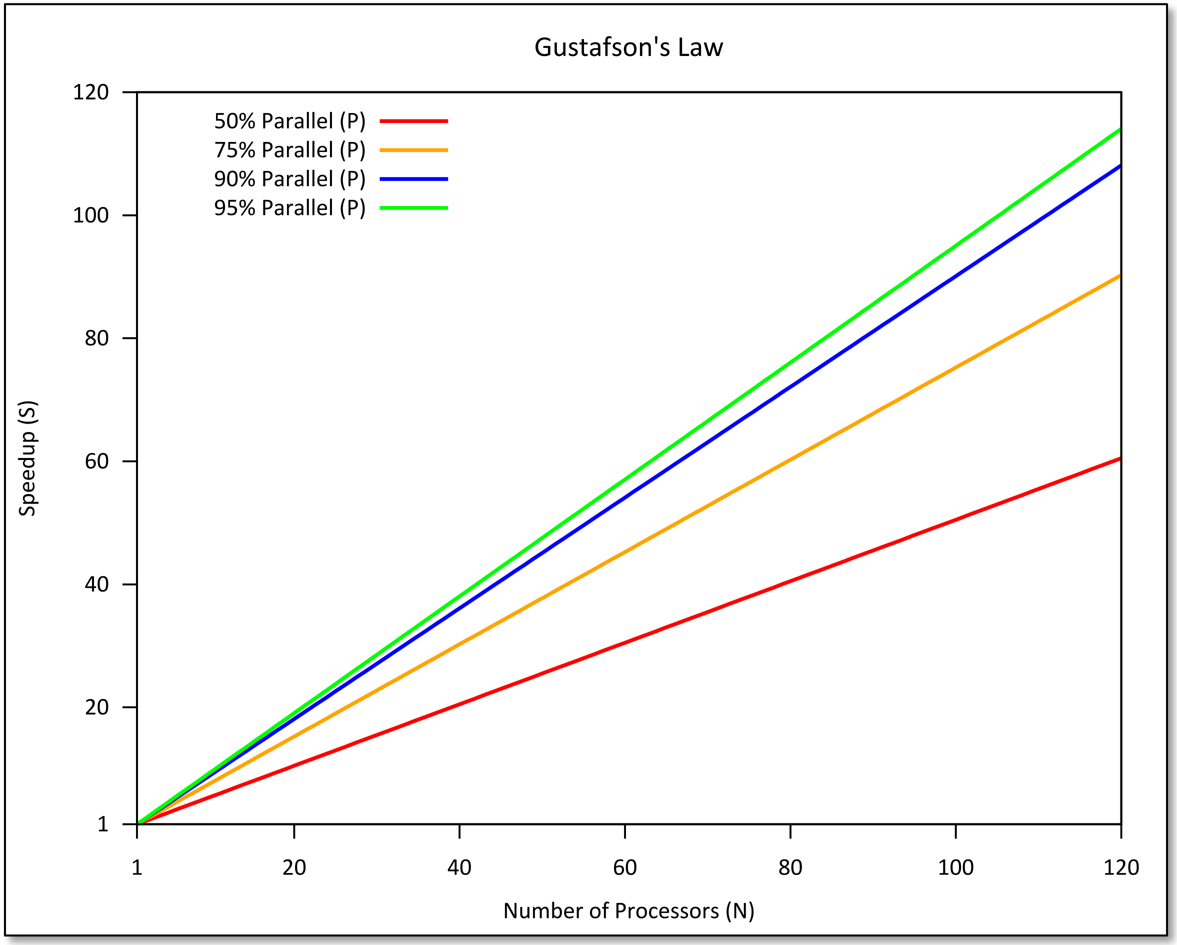
S: Speedup

P: Proportion of parallel code

N: Number of processors



Gustafson's Law (Weak Scaling)



$$S = NP - P + 1$$

S: Speedup

P: Proportion of parallel code

N: Number of processors

The 4 Horsemen of the Apocalypse: SLOW

- Starvation
  - Insufficient concurrent work to maintain high utilization of resources.
- Latencies
  - Time-distance delay of remote resource access and services.
- Overheads
  - Work for management of parallel actions and resources on critical path which are not necessary in sequential variant.
- Waiting for Contention Resolution
  - Delays due to lack of availability of oversubscribed shared resource.

courtesy of www.albrecht-durer.org

The 4 Horsemen of the Apocalypse: SLOW

- Starvation
  - Insufficient concurrent work to maintain high utilization of resourc
- Latencies
  - Time-distance delay of r resource access and
- Overheads
  - Work fo acti al path w n sequential va
- Waiting ention Resolution
  - Delays e to lack of availability of oversubscribed shared resource.

Impose upper bound on both weak and strong scaling

courtesy of www.albrecht-durer.org

Main Runtime System Tasks

- Manage parallel execution for the application — Starvation
  - Exposing parallelism, runtime adaptive management of parallelism and resources.
  - Synchronizing parallel tasks.
  - Thread (task) scheduling, load balancing, object migration.
- Mitigate latencies for the application — Latency
  - Latency hiding through overlap of computation and communication.
  - Latency avoidance through locality management.
- Reduce overhead for the application — Overhead
  - Synchronization, scheduling, load balancing, communication, context switching, memory management, address translation.
- Resolve contention for the application — Contention
  - Adaptive routing, resource scheduling, load balancing.
  - Localized request buffering for logical resources.

What’s HPX?

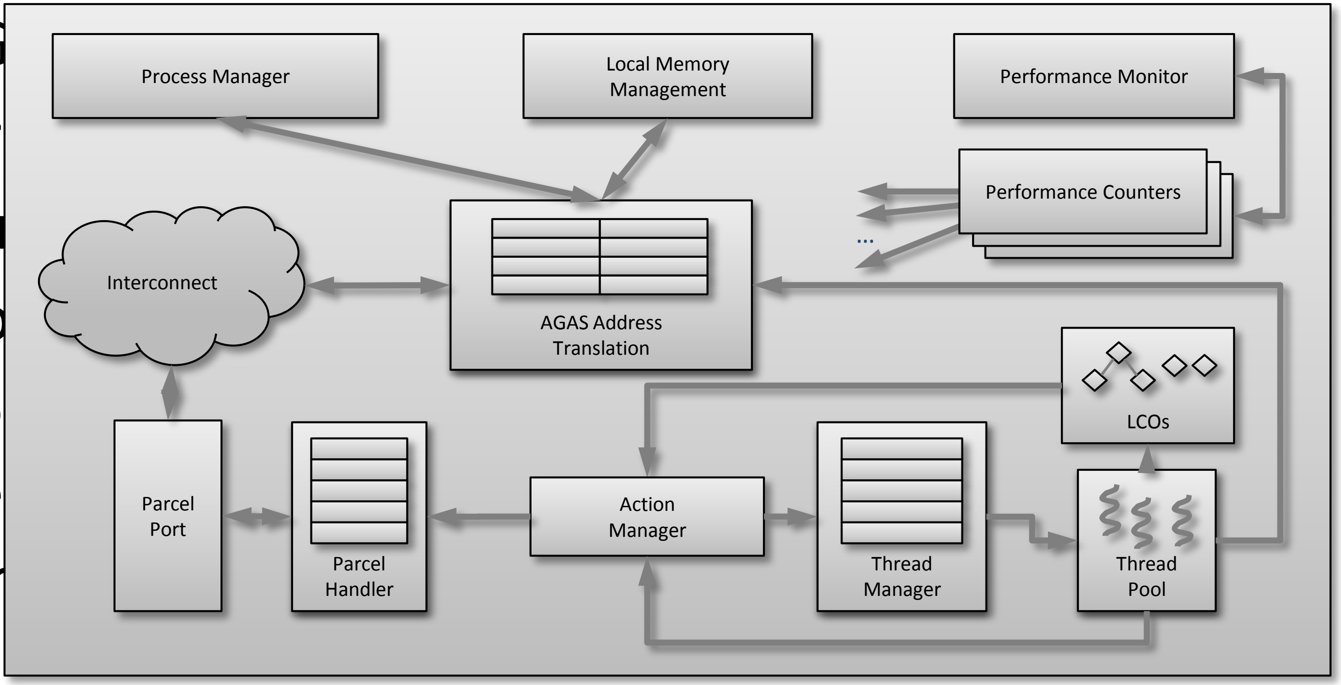
- Active global address space (AGAS) instead of PGAS.
- Message driven instead of message passing.
- Lightweight control objects instead of global barriers.
- Latency hiding instead of latency avoidance.
- Adaptive locality control instead of static data distribution.
- Moving work to data instead of moving data to work.
- Fine grained parallelism of lightweight threads instead of Communicating Sequential Processes (CSP/MPI).
- Open source – Boost Software License.

HPX Runtime System Design

- Current version of HPX provides the following infrastructure on conventional systems as defined by the ParalleX execution model.
  - Active Global Address Space (AGAS).
  - HPX Threads and Thread Management.
  - Parcel Transport and Parcel Management.
  - Local Control Objects (LCOs).
  - HPX Processes (distributed objects).
    - Namespace and policies management, locality control.
  - Monitoring subsystem.

HPX Runtime System Design

- Current version of HPX provides the following infrastructure on conventional systems as defined by the ParalleX execution model.
  - Active G
  - HPX Thr
  - Parcel T
  - Local Co
  - HPX Pro
    - Name
  - Monitor

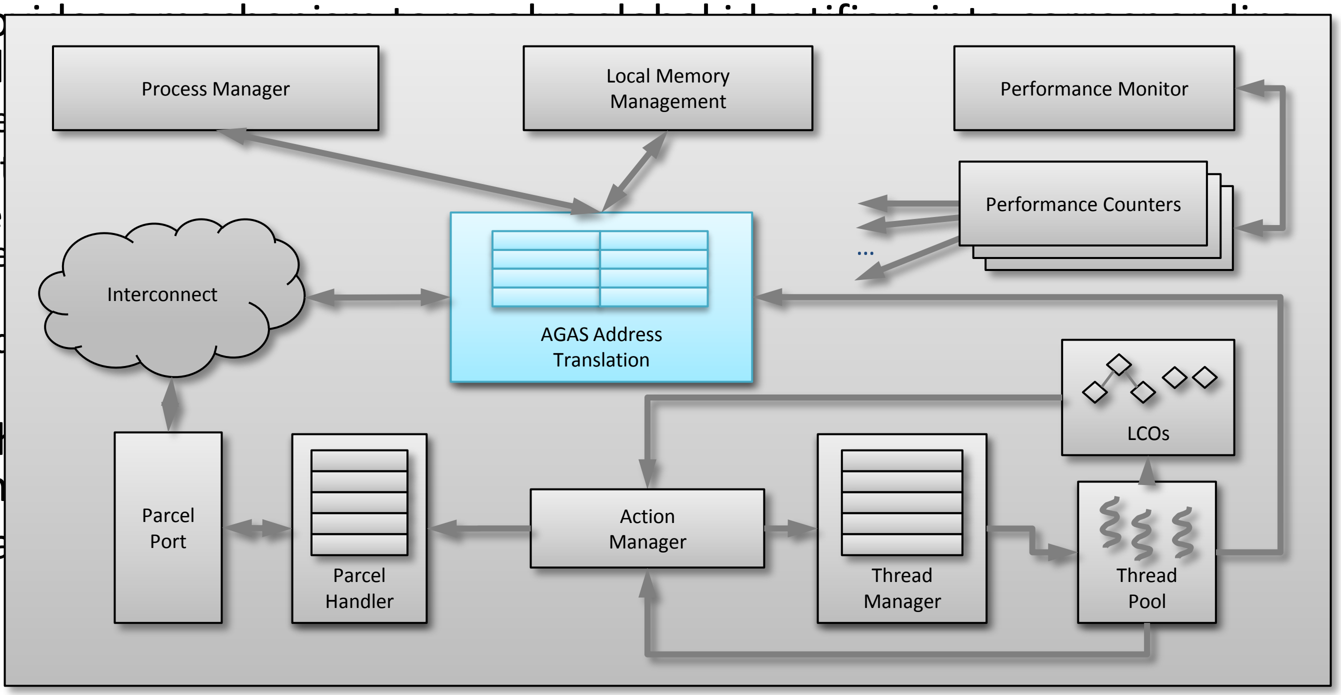


Active Global Address Space

- Global Address Space throughout the system.
  - Removes dependency on static data distribution.
  - Enables dynamic load balancing of application and system data.
- AGAS assigns global names (identifiers, unstructured 128 bit integers) to all entities managed by HPX.
- Unlike PGAS provides a mechanism to resolve global identifiers into corresponding local virtual addresses (LVA).
  - LVAs comprise – Locality ID, type of entity being referenced and its local memory address.
  - Moving an entity to a different locality updates this mapping.
  - Current implementation is based on centralized database storing the mappings which are accessible over the local area network.
  - Local caching policies have been implemented to prevent bottlenecks and minimize the number of required round-trips.
- Current implementation allows autonomous creation of globally unique ids in the locality where the entity is initially located and supports memory pooling of similar objects to minimize overhead.
- Implemented garbage collection scheme of HPX objects.

Active Global Address Space

- Global Address Space throughout the system.
  - Removes dependency on static data distribution.
  - Enables dynamic load balancing of application and system data.
- AGAS assigns global names (identifiers, unstructured 128 bit integers) to all entities managed by HPX.
- Unlike PGAS pro
  local virtual add
  - LVAs comprise
  - Moving an ent
  - Current imple
    accessible ove
  - Local caching
    number of req
- Current implem
  locality where th
  objects to minim
- Implemented ga

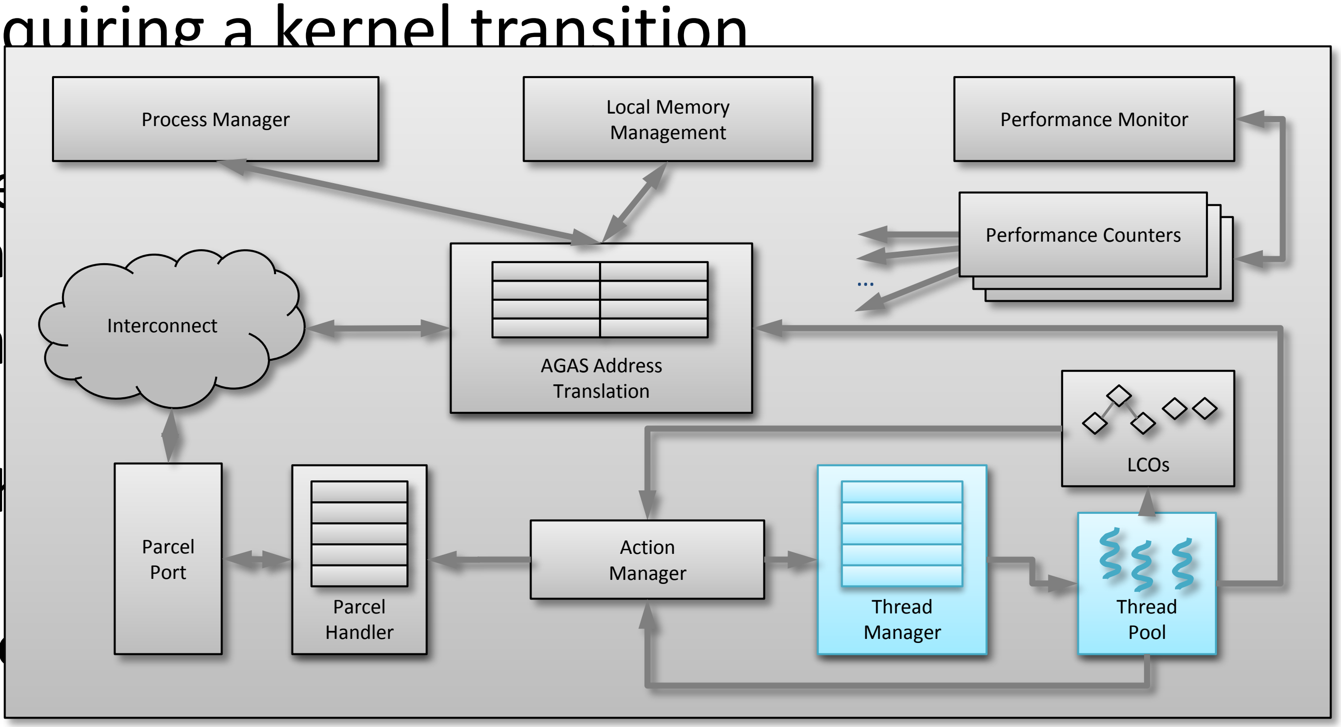


Thread Management

- Thread manager is modular and implements a work-queue based task management
- Threads are cooperatively scheduled at user level without requiring a kernel transition
- Specially designed synchronization primitives such as semaphores, mutexes etc. allow synchronization of HPX- threads in the same way as conventional threads
- Thread management currently supports several key modes
  - Global Thread Queue
  - Local Queue (work stealing)
  - Local Priority Queue (work stealing)

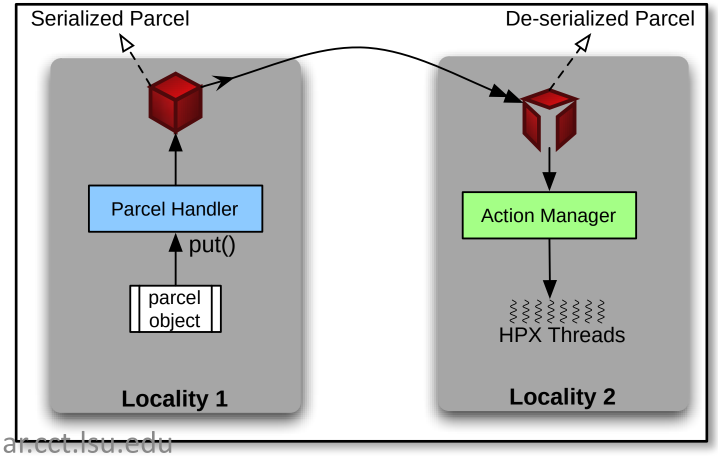
Thread Management

- Thread manager is modular and implements a work-queue based task management
- Threads are cooperatively scheduled at user level without requiring a kernel transition
- Specially d semaphore HPX- threa
- Thread ma modes
  - Global Th
  - Local Qu
  - Local Pri



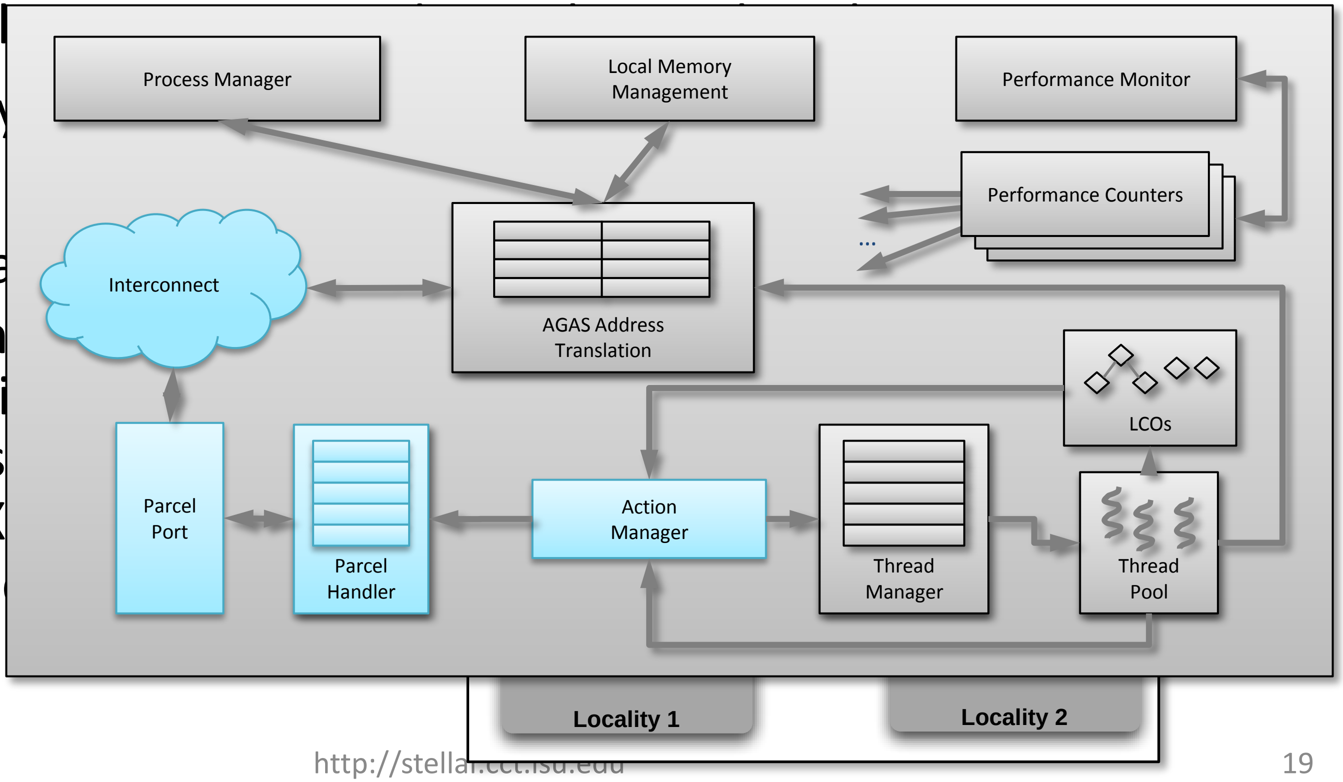
Parcel Management

- Active messages (parcels)
- Destination address, function to execute, parameters, continuation
- Any inter-locality messaging is based on Parcels
- In HPX parcels are represented as polymorphic objects
- An HPX entity on creating a parcel object hands it to the parcel handler.
- The parcel handler serializes the parcel where all dependent data is bundled along with the parcel
- At the receiving locality the parcel is de-serialized and causes a HPX thread to be created based on its content

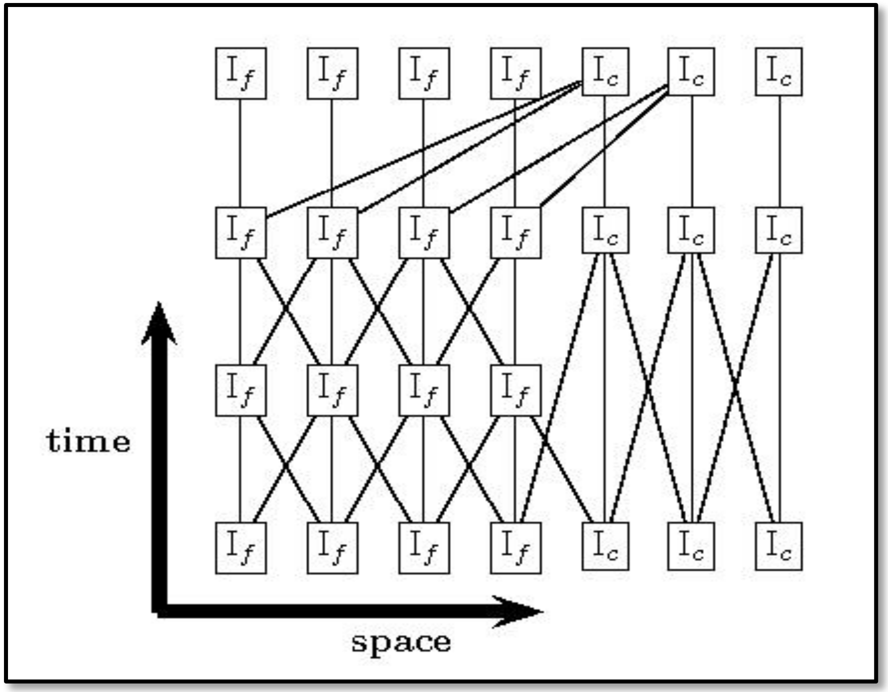


Parcel Management

- Active messages (parcels)
- Destination address, function to execute, parameters, continuation
- Any inter-locality messaging is based on Parcels
- In HPX parcel
- An HPX entity
  handler.
- The parcel ha
  bundled alon
- At the receivi
  parcel is de-s
  causes a HPX
  created base



Constraint-based Synchronization



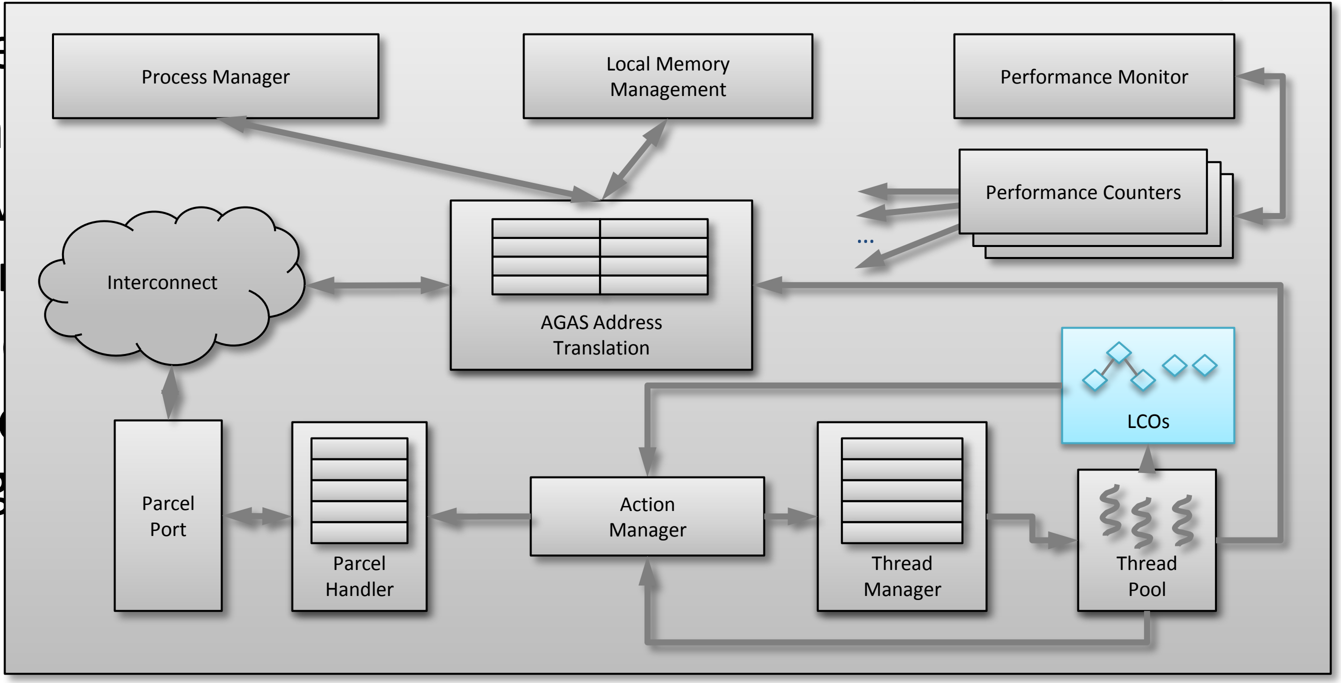
- Compute dependencies at task instantiation time.
- No global barriers, uses constraint based synchronization.
- Computation flows at its own pace.
- Message driven.
- Symmetry between local and remote task creation/execution.
- Possibility to control grain size.

LCOs (Local Control Objects)

- LCOs provide a means of controlling parallelization and synchronization of HPX-threads.
- Enable event-driven thread creation and can support in-place data structure protection and on-the-fly scheduling.
- Preferably embedded in the data structures they protect.
- Abstraction of a multitude of different functionalities for.
  - Event driven PX-thread creation.
  - Protection of data structures from race conditions.
  - Automatic on-the-fly scheduling of work.
- LCO may create (or reactivate) a HPX-thread as a result of “being triggered”.

LCOs (Local Control Objects)

- LCOs provide a means of controlling parallelization and synchronization of HPX-threads.
- Enable event-driven thread creation and can support in-place data structure protection and on-the-fly scheduling.
- Preferably e
- Abstraction
  - Event driv
  - Protectio
  - Automati
- LCO may cr
  “being trigg



Hello HPX World

- Simplest HPX program:

```
#include <hpx/hpx.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);
}
```

Hello HPX World

- HPX program with remote actions:

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

HPX_REGISTER_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 program with synchronized parallel remote actions:

```
int hpx_main()
{
    std::vector<hpx::lcos::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();
}
```

HPX – The API

- Fully asynchronous
  - All possibly remote operations are asynchronous by default.
    - Fire and forget semantics (result is not available).
    - Pure asynchronous semantics (result is available via hpx::future).
  - Composition of asynchronous operations.
    - hpx::wait_all, hpx::wait_any, hpx::wait_n
    - hpx::future::when(f)
  - Can be used synchronously, but doesn’t block.
    - HPX-Thread is suspended while waiting for result.

HPX – The API

- As close as possible to C++11 standard library, where appropriate, for instance
  - std::thread → hpx::thread
  - std::mutex → hpx::mutex
  - std::future → hpx::future
  - std::async → hpx::async
  - std::bind → hpx::bind
  - std::function → hpx::function
  - std::cout → hpx::cout

HPX – The API

- Fully move enabled (using Boost.Move)
  - hpx::bind, hpx::function, hpx::tuple
- Fully type safe
  - Extends the notion of a callabl' to remote case (actions).
  - Everything you can do with functions is possible with actions as well.
- Usable in remote contexts.
  - Can be sent over the wire (hpx::bind, hpx::function).
  - Can be used with actions (hpx::async, hpx::bind, hpx::function).

Fibonacci Number Sequence

- The pathologic corner case:

```
int fibonacci(int n);




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

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


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

Fibonacci Number Sequence

- The pathologic corner case:

```
int fibonacci(int n);

HPX_REGISTER_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;
}
```

SOME IMPLEMENTATION DETAILS

How do we craft parallel algorithms with HPX?

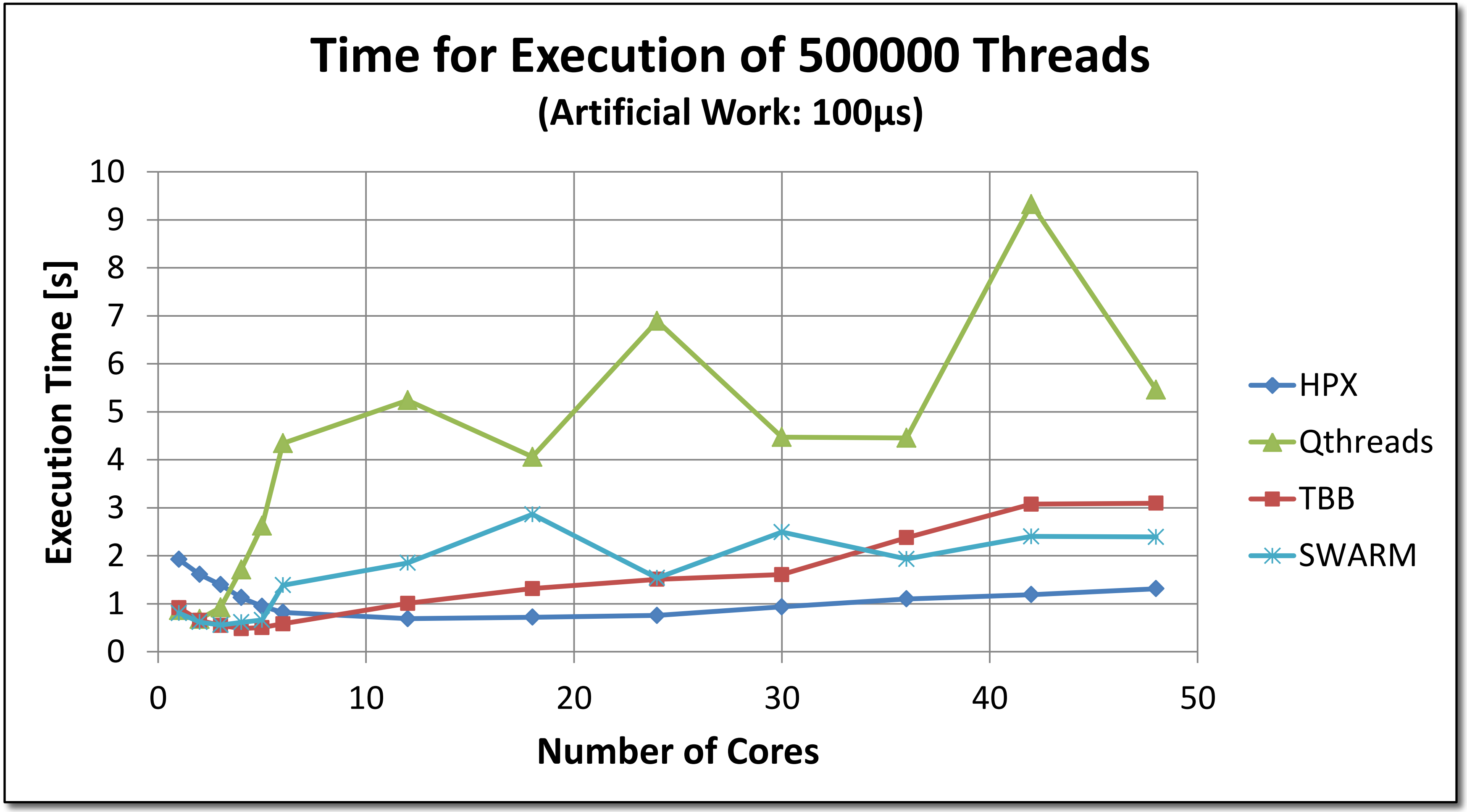
By coding functionally!

Why functional?

Think Functionally!

- Coding functionally implicitly breaks algorithms down into finer grained atoms (functions) which have clearly defined inputs (arguments) and outputs (return values).
  - Finer grained units of execution are easier to dynamically load balance.
  - Modern hardware facilitates very fine grain threading.
- The dependencies between these atoms can be identified in a straightforward fashion.

Think Functionally!



Overhead: HPX-threads

2 µs

of overhead to create and use an HPX-thread

(hardware: 8 hexa-core AMD Opteron with 96G DDR2)

50 ns

to perform the context switch between HPX-threads

(hardware: 1 quad-core Intel Sandy Bridge with 16G DDR3)

2^{18} threads per GB of RAM

can run concurrently with HPX with the minimum stack size

(hardware: x86-64)

Global Identifiers (GIDs)

- Global identifiers (GIDs, `hpx::id_type`) provide a way to refer to objects in a computing environment that spans shared-memory boundaries.
- `hpx::find_here()` – Returns the GID that refers to the object that provides runtime services on this locality.
  - This is effectively the GID of a locality.
- `hpx::find_all_localities()` – Return an `std::vector<hpx::id_type>` which contains the GIDs of all available localities.

Actions

- Actions – The building blocks of asynchronous execution in HPX.
  - Actions wrap C++ Functions into remotable Function Objects (Function Objects which can be transported to and invoked on other localities).
  - The return type of an action must be serializable.
  - HPX utilizes Boost.Serialization for serializing functions.

Actions

- HPX has two forms of actions.
  - Plain actions wrap global functions.
  - Component actions wrap member functions.
- How do actions fit in with other ParalleX constructs in HPX?
  - A parcel's payload is an action.
  - Each action is executed in its own HPX-thread.

Actions Example

```
int add(int x, int y) { return x + y; }

// Plain actions wrap global functions.
// This macro defines the action type add_action.
HPX_REGISTER_PLAIN_ACTION(add);

void foo() {
    add_action f;

    // Actions are function objects.
    f(hpx::find_here(), 2, 2);

    // std::bind can be used to bind actions, but std::bind isn’t
    // serializable – hpx::bind is.
    hpx::bind(f, hpx::find_here(), _1, 5)(7);
}
```

Components

- Components - Classes that are globally named, meaning they can be referenced from any locality.
- Components expose methods that can be called remotely (component actions).
- Writing components is easy. A class just needs to inherit from a component base class and implement actions.

Components Example

```
struct counter : hpx::components::simple_component_base<counter> {
    int value;
  public:
    int increment() { return ++value; }

    // Component actions wrap member functions.
    // This macro defines the action type increment_action.
    HPX_DEFINE_COMPONENT_ACTION(counter, increment, increment_action);
};

// This macro must be called explicitly for component actions only
// because it must go in the global namespace.
HPX_REGISTER_ACTION((counter)(increment_action));
```

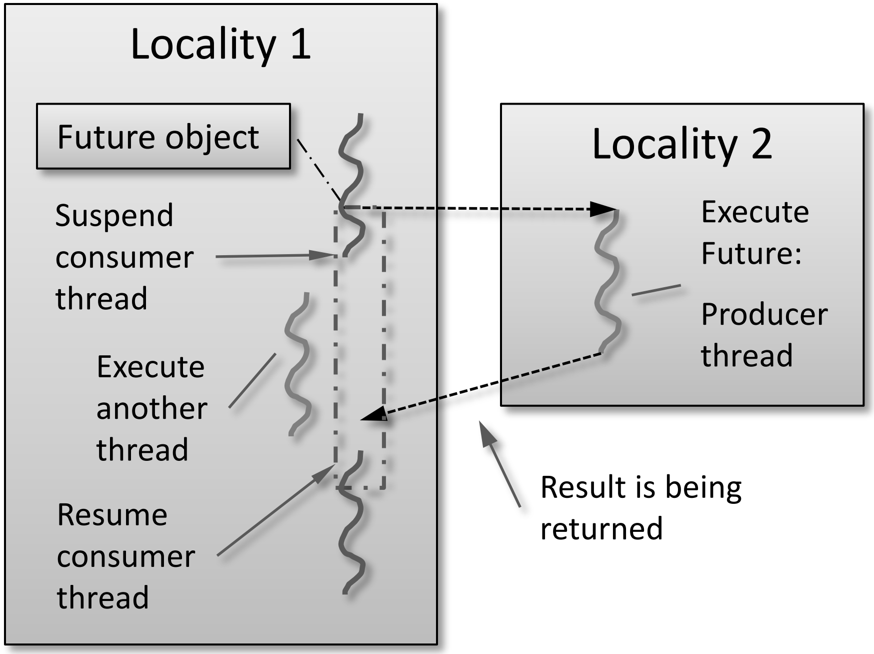
LCOs (Local Control Objects)

- LCOs are concurrency control primitives.
  - LCOs can be used to control and coordinate the execution of multiple HPX-threads.

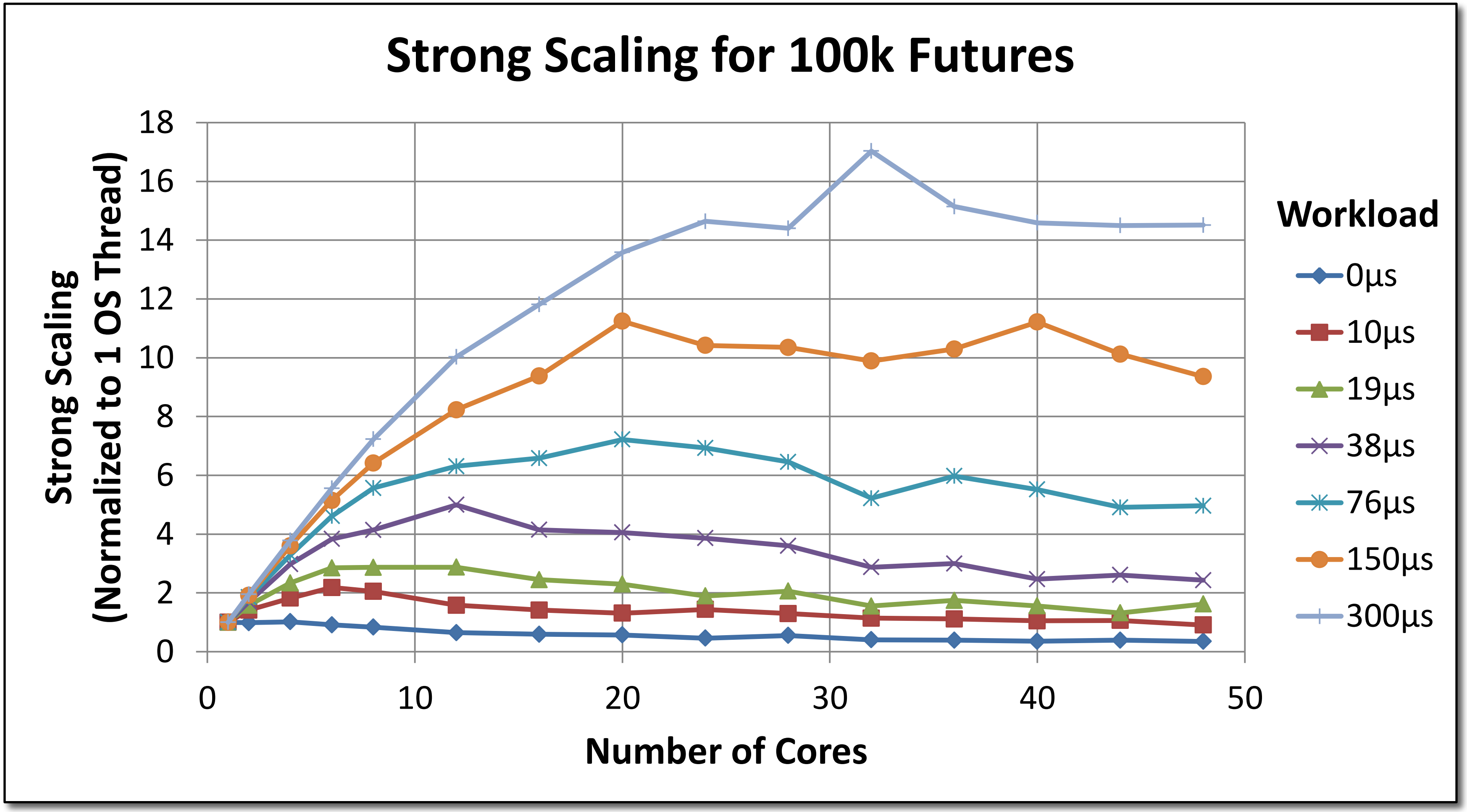
Synchrony Primitives	**Asynchrony Primitives**	**Dependency-driven Primitives**
• Mutexes • Spinlocks • Barriers • Condition Variables	• Futures • Promises	• Dataflows • Queues

Future LCOs

- Future LCOs are essential to HPX because they are the primary method of controlling asynchrony in HPX.
- Compatible with `std::future<>`.
- Futures act as proxies for values that are being computed asynchronously by actions.



Future LCOs


Strong Scaling for 100k Futures
Strong Scaling
(Normalized to 1 OS Thread)
0
2
4
6
8
10
12
14
16
18
0
10
20
30
40
50
Number of Cores
Workload
0µs
10µs
19µs
38µs
76µs
150µs
300µs

Overhead: Future LCOs

17 µs

of overhead to create and use an `hpx::future<>`

(hardware: 8 hexa-core AMD Opteron with 96G DDR2)

Controlling Asynchrony

- Three ways to invoke functions:
  - Synchronously – Wait for the function to execute.
  - Asynchronously – Don’t wait for the function to execute and make the result of the function available through polling (checking if the result is ready) or callback functions (functions invoked when the result is ready).
  - Fire and Forget – Don’t wait for the function to execute and disregard its result.

Controlling Asynchrony

`R f(a...)`	**Synchronous (returns R)**	**Asynchronous (returns hpx::future<R>)**	**Fire and Forget (returns void)**
Functions (Direct)	`f(a...)`	`async(f, a...)`	`apply(f, a...)`
Functions (Lazy)	`bind(f, a...)`	`async(bind(f, a...))`	`apply(bind(f, a...))`
Actions (Direct)	`f(id, a...)`	`async(f, id, a...)`	`apply(f, id, a...)`
Actions (Lazy)	`bind(f, id, a...)`	`async(bind(f, id, a...))`	`apply(bind(f, id, a...))`

- `id` is a global identifier (GID).

Naïve Futurized Fibonacci

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

    // Asynchronously launch the creation of one of the sub-terms of the
    // execution graph.
    hpx::future<int> f = hpx::async(fibonacci, n - 1);
    int r = fibonacci(n - 2);

    // Wait for f to finish, then add it to r.
    return f.get() + r;
}
```

Fibonacci with Continuations

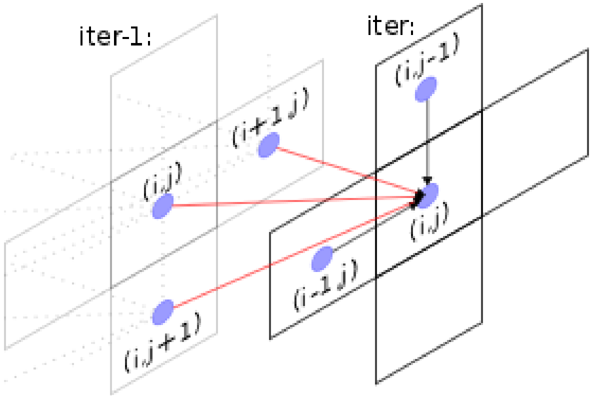
```
hpx::future<int> fibonacci(hpx::future<int> const& n) {
    if (n.get() < 2) return n;
    return hpx::async(fibonacci, n.get() - 1).when(fibonacci_continuation(n));
}

hpx::future<int> fibonacci(int n) {
    if (n < 2) return hpx::create_future_value(n);
    return fibonacci(hpx::create_future_value(n));
}

struct fibonacci_continuation {
    typedef int result_type;
    hpx::future<int> n_;
    fibonacci_continuation(hpx::future<int> n) : n_(n) {}

    result_type operator()(hpx::future<int> res) const {
        return fibonacci(n_.get() - 2).get() + res.get();
    }
};
```

Dataflow LCOs



- Dataflow LCOs are an extension of futures that enable dependency-driven asynchrony.
- Compatible with `std::future<>`.
- Computation of the action associated with a dataflow does not begin until all the arguments of the dataflow are ready.
  - Non-LCO arguments are always ready.
  - LCO arguments, such as futures, might not be ready.

Dataflow Interest Calculator

```
double calc(double principal, double rate) { return principal * rate; }

double add(double principal, double interest) { return principal + interest; }

double interest(double principal, double rate, int time) {
    hpx::dataflow_value<double> principal = hpx::create_dataflow_value(p);
    hpx::dataflow_value<double> rate = hpx::create_dataflow_value(i_rate);

    for (int i = 0; i < time; ++i) {
        hpx::dataflow_value<double> interest
            = hpx::dataflow(calc, principal, rate);
        principal = hpx::dataflow(add, principal, interest);
    }

    return principal.get();
}
```

Data Distribution

- We often want to ask HPX to distribute components across all available localities for us.
- Factories – Components which create and distributed objects according to a certain policy.
- Policies could distribute objects according to the number of cores on each locality, the amount of local memory on each locality, the number of GPGPUs on each locality, etc.

Nonintrusive Components

- Components are fine, but implementing them is intrusive.
- `hpx::object<>` – Non-intrusively adapts a type into a component.
- `hpx::new_<>()` – Creates an hpx::object<> instance on a specific locality.
- Remotable Function Objects (actions and serializable Function Objects) can be applied to types wrapped in this fashion.

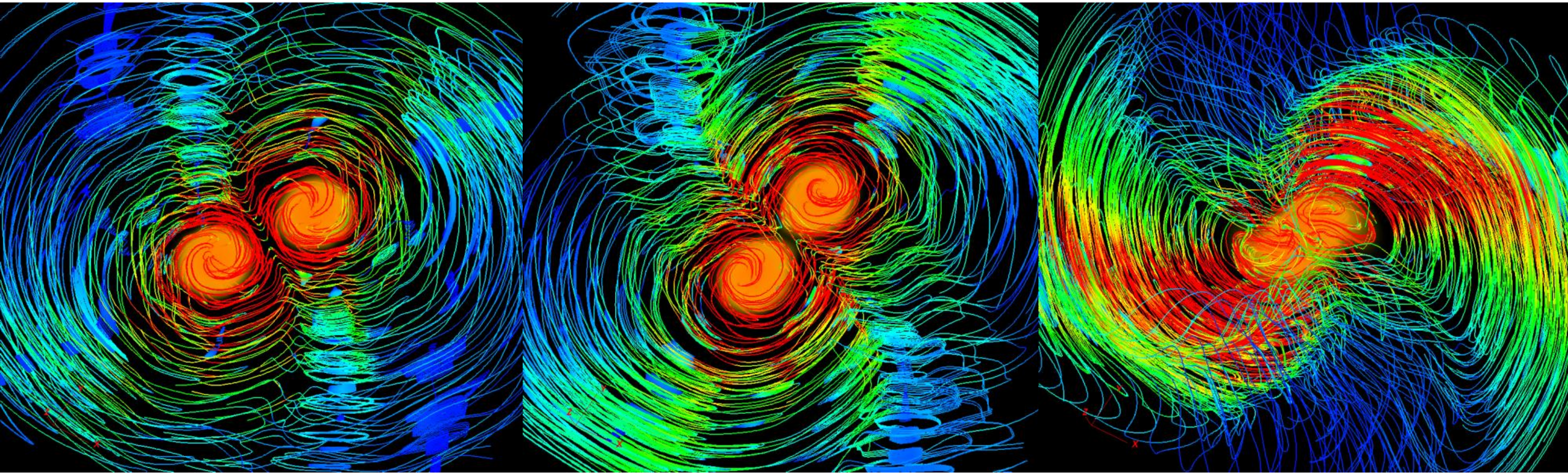
Nonintrusive Components Example

```
struct A {
    A(int i = 0) : i_(i) {}
    int i_;
};

void output(A const& a) { hpx::cout << a.i_ << "\n" << hpx::flush; }
HPX_PLAIN_ACTION(output, output_action)

void bar(hpx::id_type const& locality) {
    hpx::future<hpx::object<A> > a = hpx::new_<A>(locality, 17);

    // Monadic syntax.
    (o <= output_action()).get(); // output_action() is an unnamed temporary.
}
```



REAL APPLICATIONS

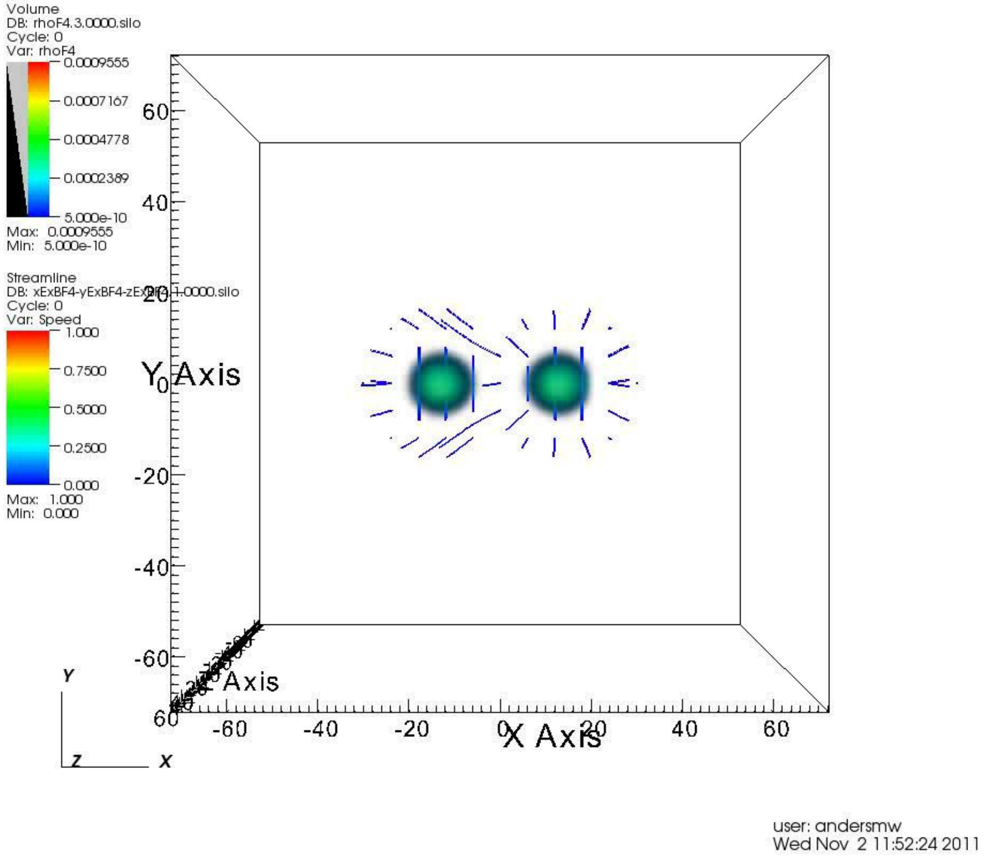
Overview

- Advanced global address space parallel methods to enable neutron star simulations with a tabulated equation of state
- Dynamic load balancing via message-driven work-queue execution for Adaptive Mesh Refinement (AMR) applications
- Other applications: Particle-In-Cell, Symmetric Contact, N-body

Global Address Space Models

- One controversial issue is the relative value of global address space models and management versus more conventional distributed memory structure.
- Finite temperature Equations of State for Neutron star simulations provides a nice venue for exploring this

Neutron Star Simulations



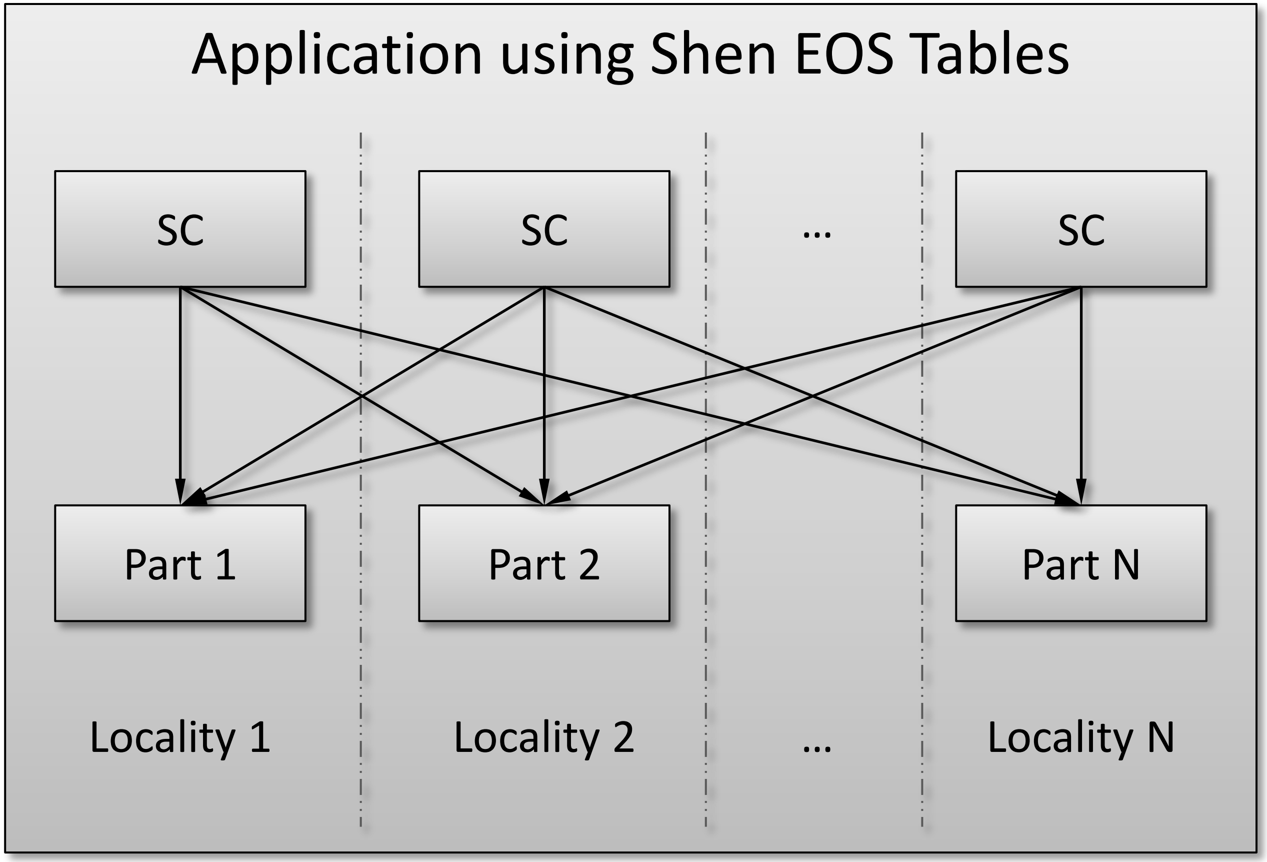
A little about Equations of State

- Realistic equations of state generally cannot be computed “in place”: they must be precomputed and placed in tables
- These tables tend to be too large (many GB’s) for application using conventional practices
- Conventional practice is to read in the table for each core
- Current generation tables are reaching out-of-core sizes

Equation of State Tables

- A 2 x 2 x 2 cube of double-precision floating numbers must be accessed for trilinear interpolation
- The aggregate size of accessed data volume for neutron star simulations significantly exceeds the combined size of L3 processor caches.
- Performance of equation of state interpolation is memory bound

Table Access Based on Futures



Each locality has a client side object allowing transparent access to all of the table data. The table itself is partitioned into chunks.

SheneOS Example

```
std::size_t num_partitions = 32;
char const* shen_table = "sheneos.h5";
char const* shen_symbolic_name = "/neutron_star/sheneos";

// Create a distributed interpolation object.
sheneos::interpolator shen;
shen.create(shen_table, shen_symbolic_name,num_partitions);
```

ShenEOS Example

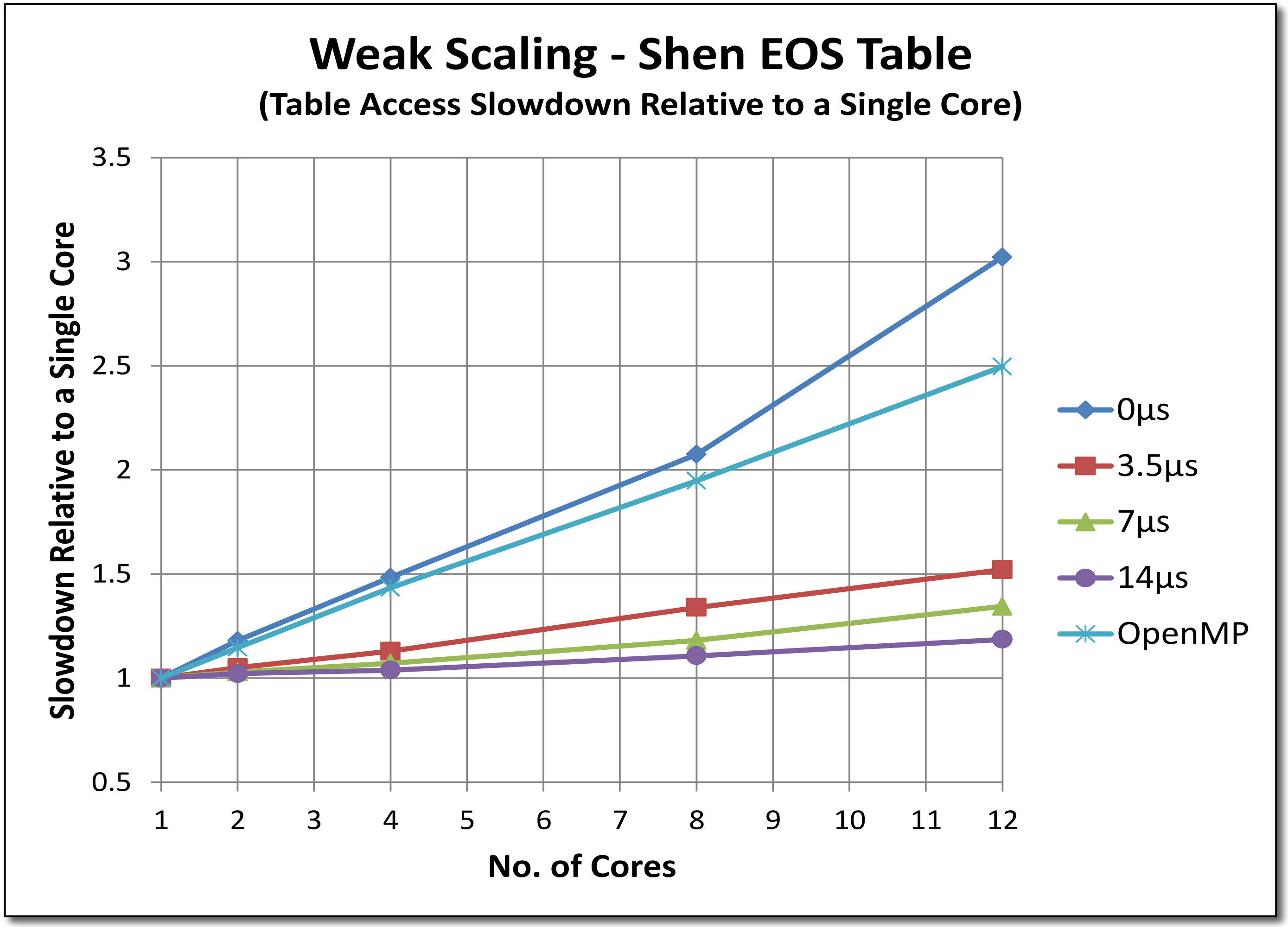
```
// Connect to our distributed interpolation object.
sheneos::interpolator shen;
shen.connect("/neutron_star/sheneos");

std::vector<hpx::future<std::vector<double> > eosaccess;
for (std::size_t k; k < value.ksize(); ++k) {
    for (std::size_t j; j < value.jsize(); ++j) {
        for (std::size_t i; i < value.isize(); ++i) {
            auto ye = xye(i, j, k);
            auto temp = xtemp(i, j, k);
            auto rho = xrho(i, j, k);
            eosaccess.push_back(shen.interpolate_async(ye, temp, rho));
        }
    }
}

auto callback = boost::bind(fill_in_primitives, _1, _2, boost::ref(value));
hpx::wait(eosaccess, callback);
```

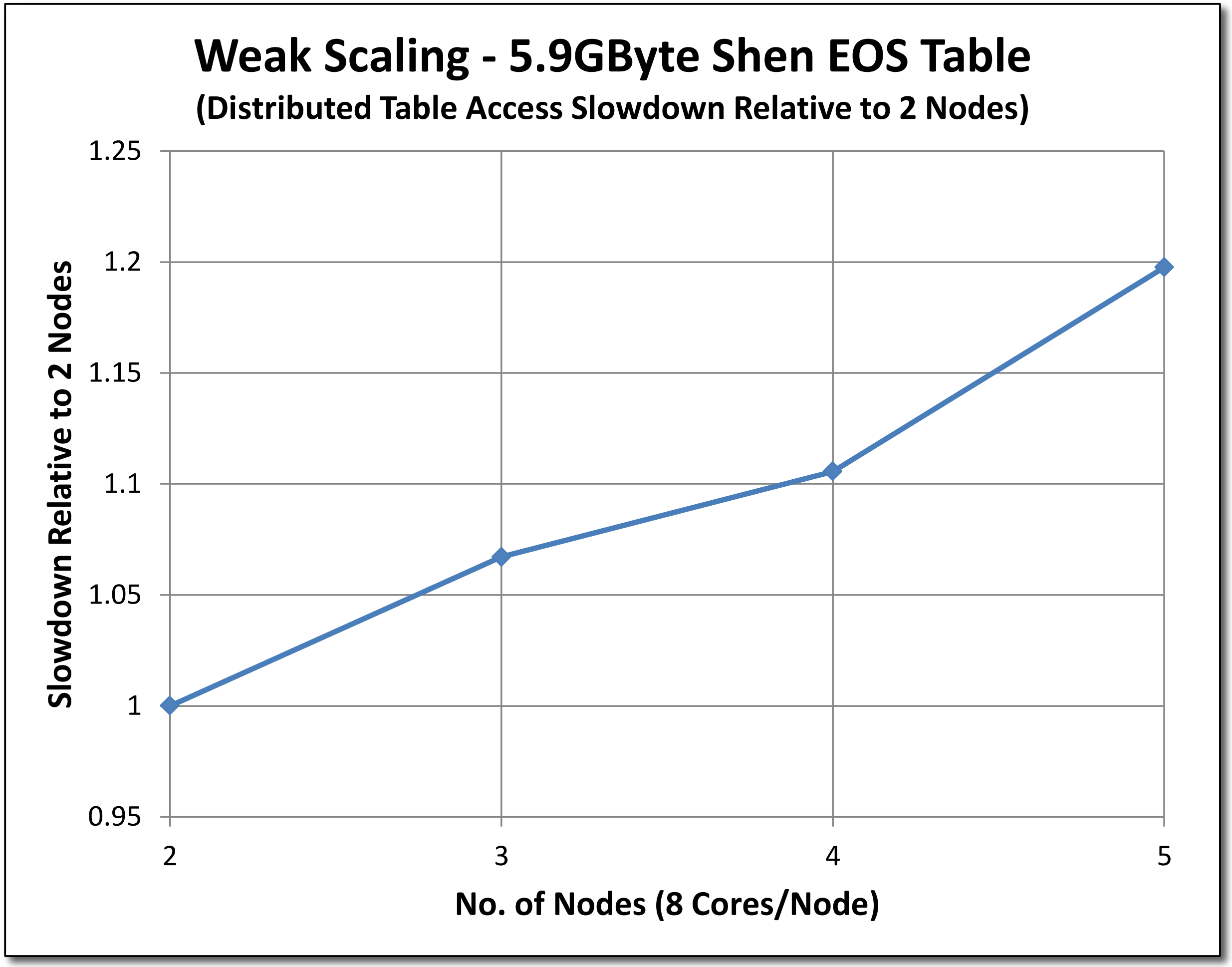
Weak Scaling for Table Access

We compare Futures with OpenMP table access. When no work is overlapped, both OpenMP and Futures show significant slowdown in concurrent table access. But by overlapping usable workload with the table access using Futures, the table access slowdown becomes negligible.



Distributed Table Access

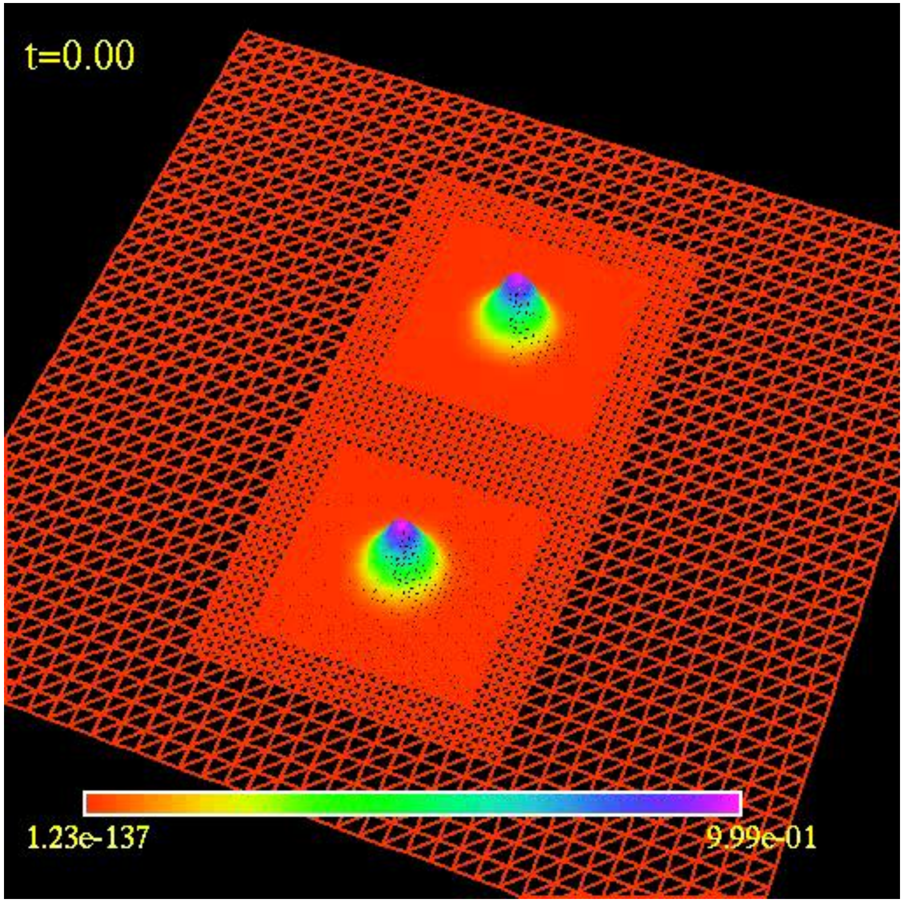
Out-of-core sized tables are increasingly common. The Futures approach to table access Works just as well in distributed memory settings as in shared memory settings.



Key Points from this Example

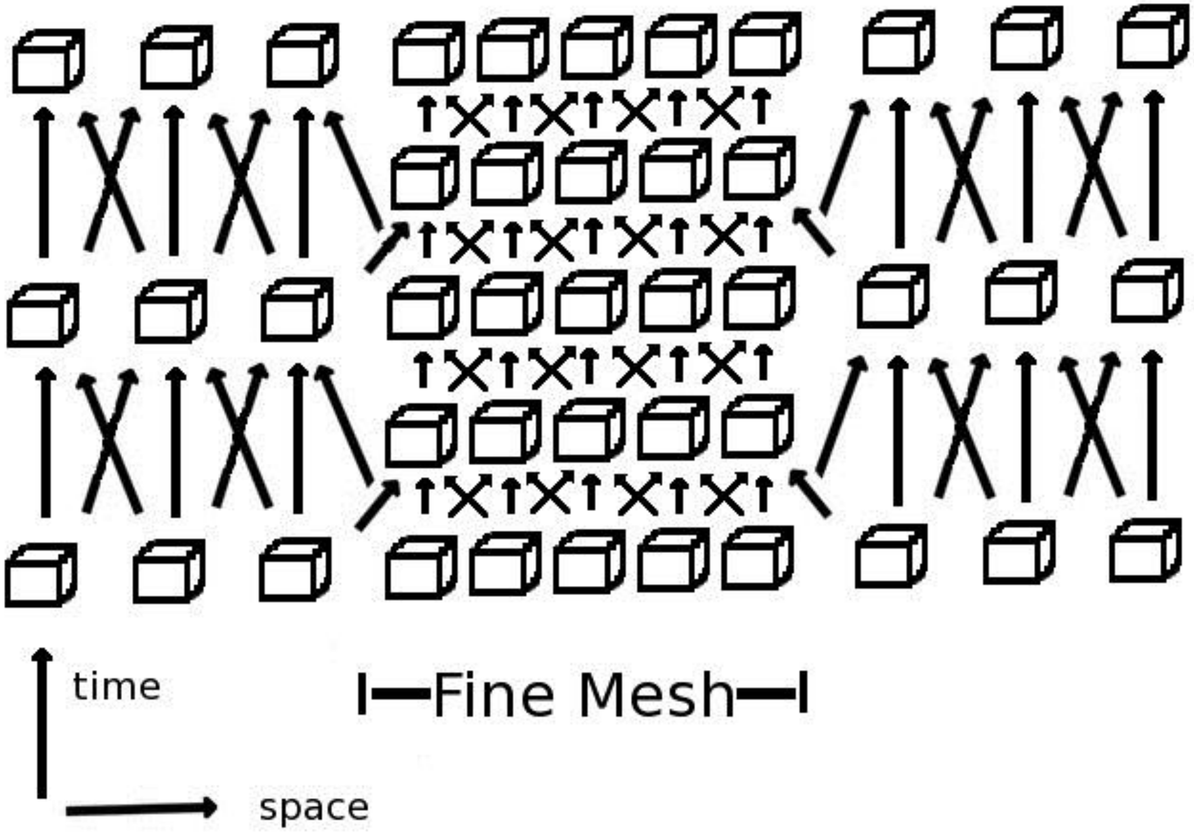
- AGAS models together with futures significantly simplify hiding network latency and amortizing contention.
- The conventional distributed memory structure isn’t a viable option for high volume, random coordinate stream table access applications.

Dynamic Load Balancing via Message-driven Work-queue Execution for Adaptive Mesh Refinement (AMR)

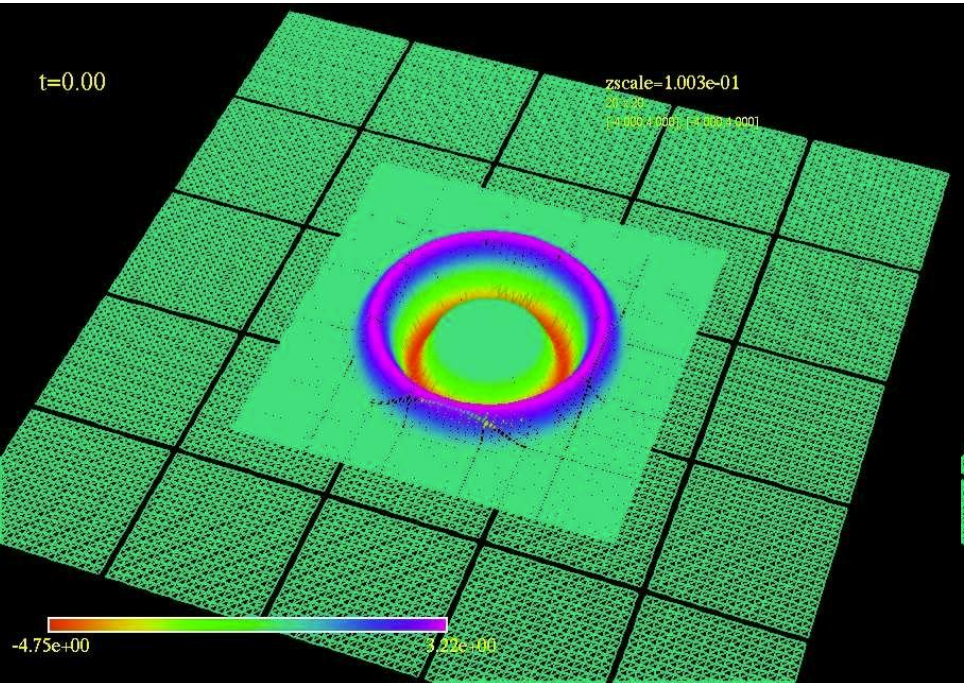


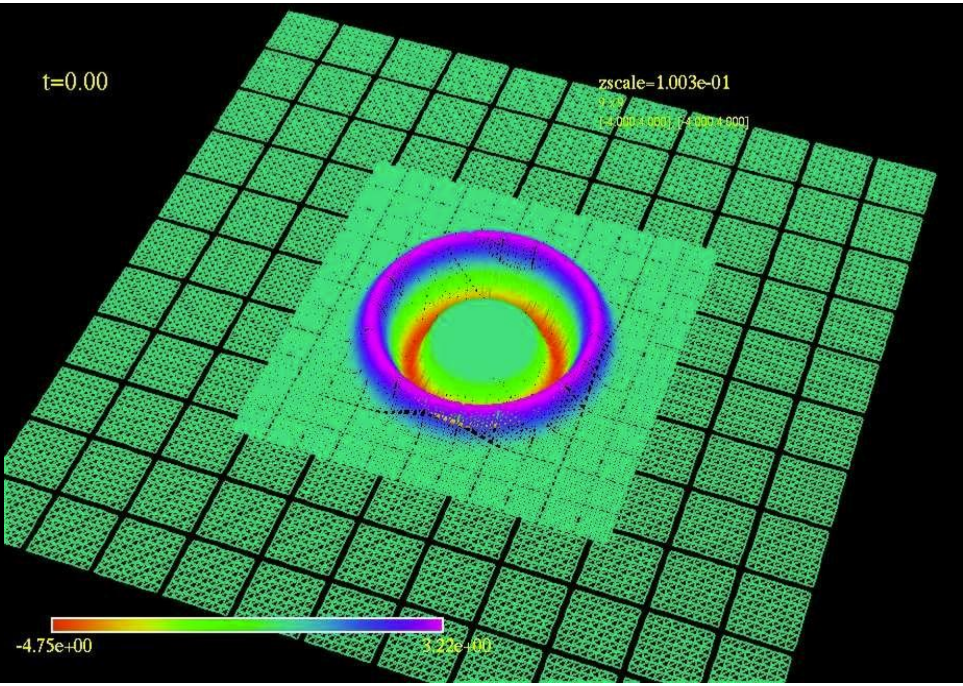
Removing Global Barriers

Remove all global barriers using dataflow; make the grain size of computation adjustable at runtime

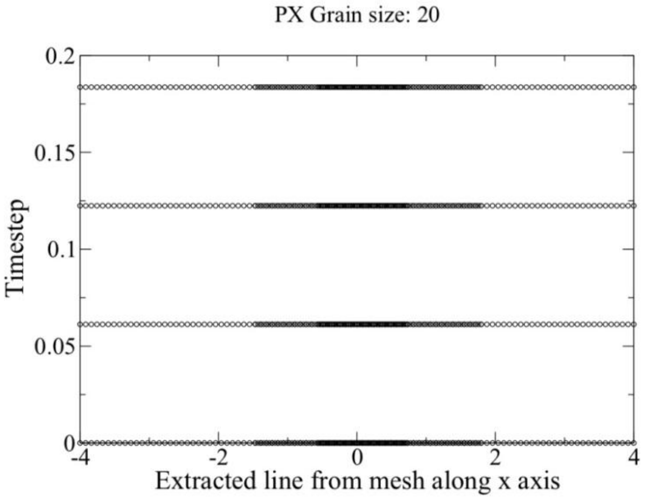


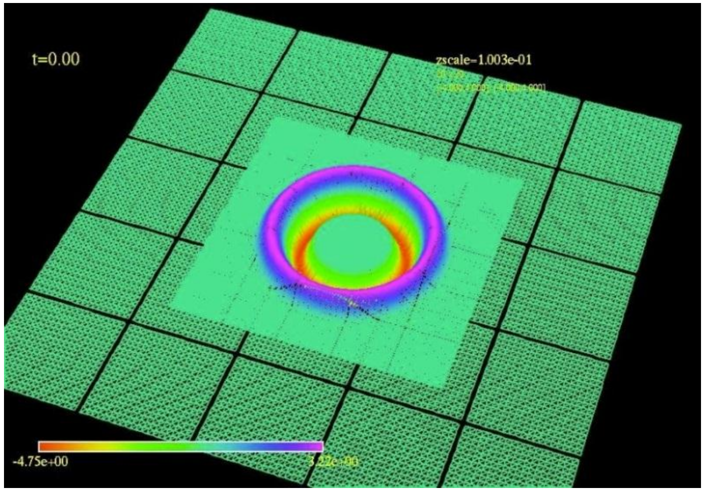
The Impact of Granularity

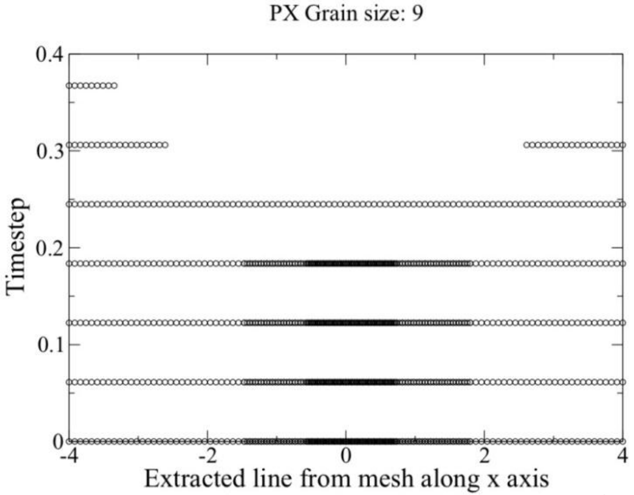


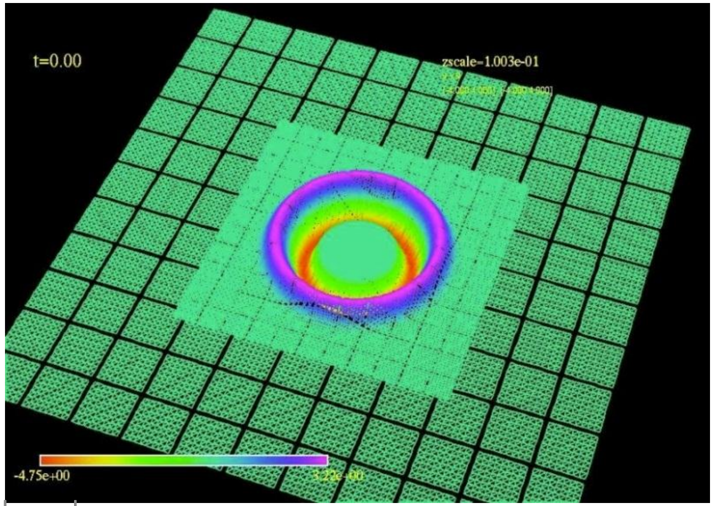


The Impact of Granularity

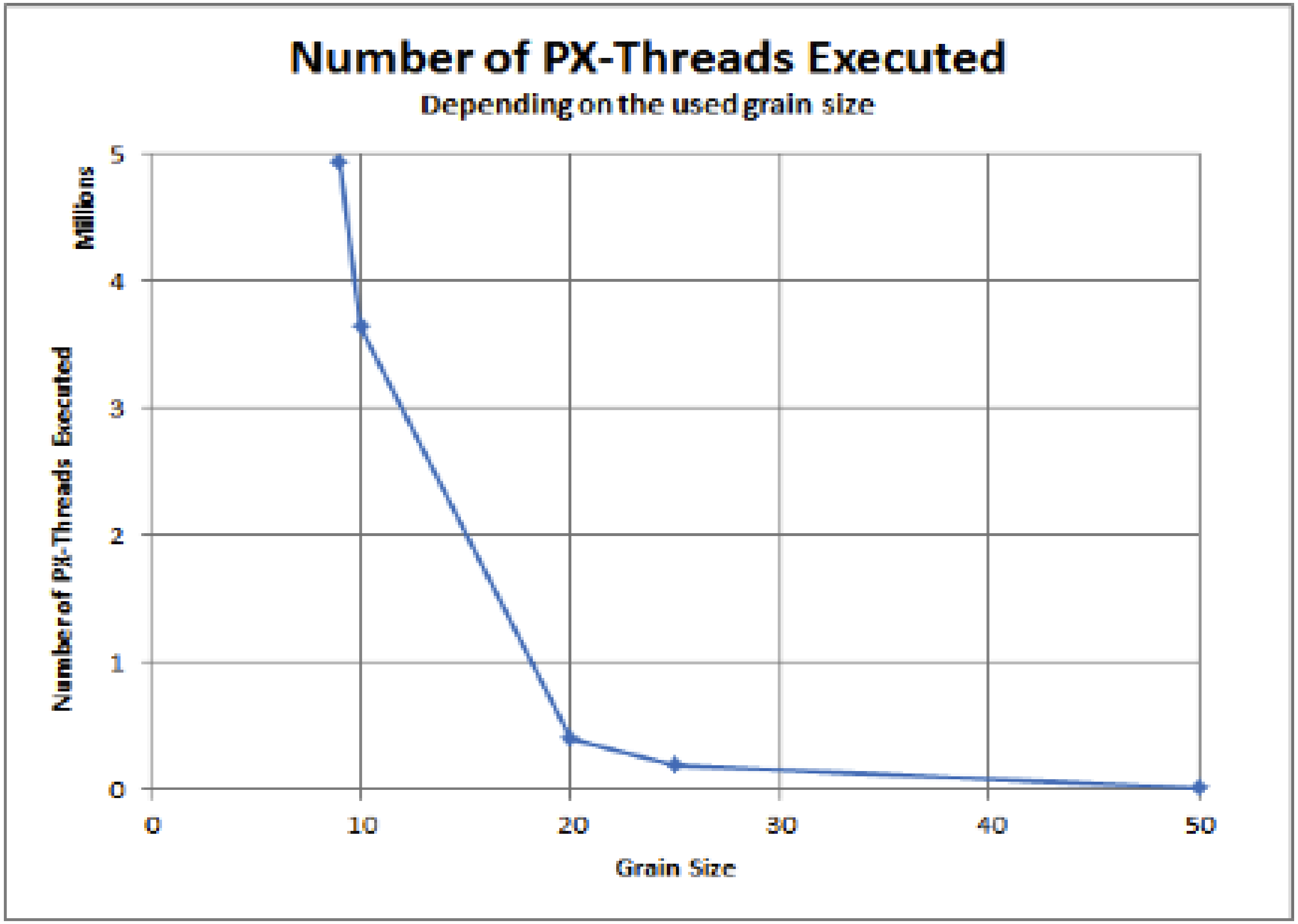




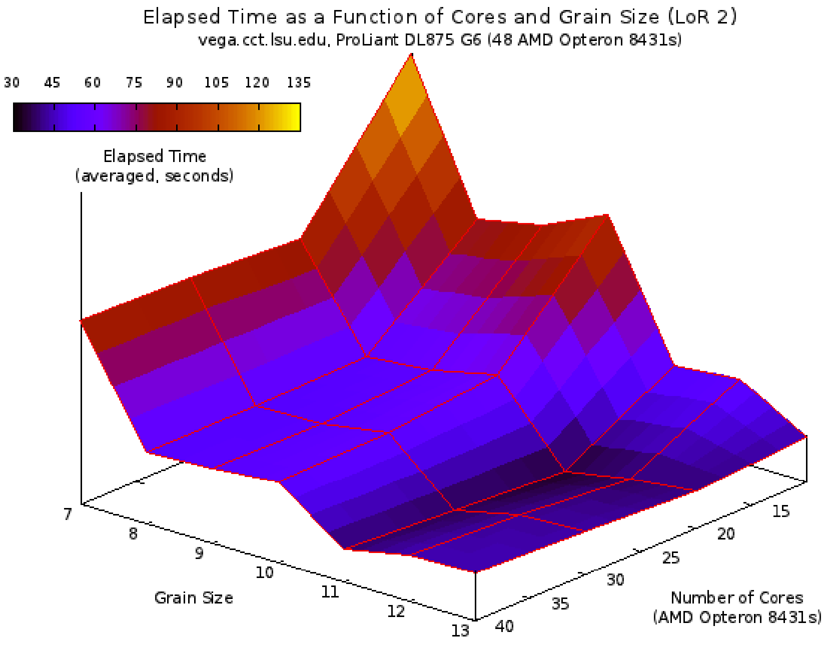


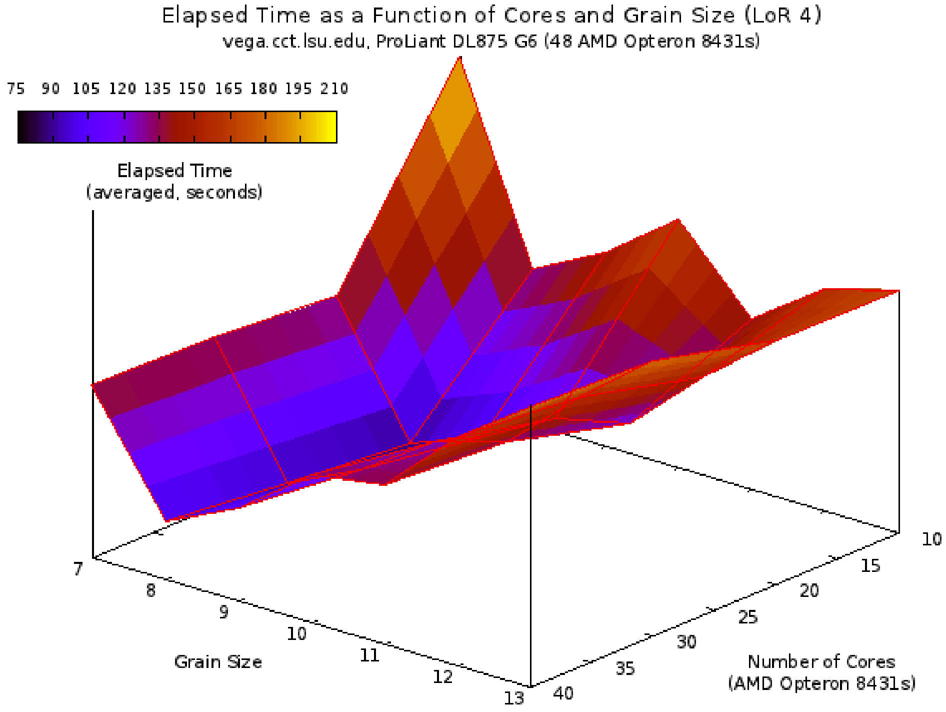


Competing Effects

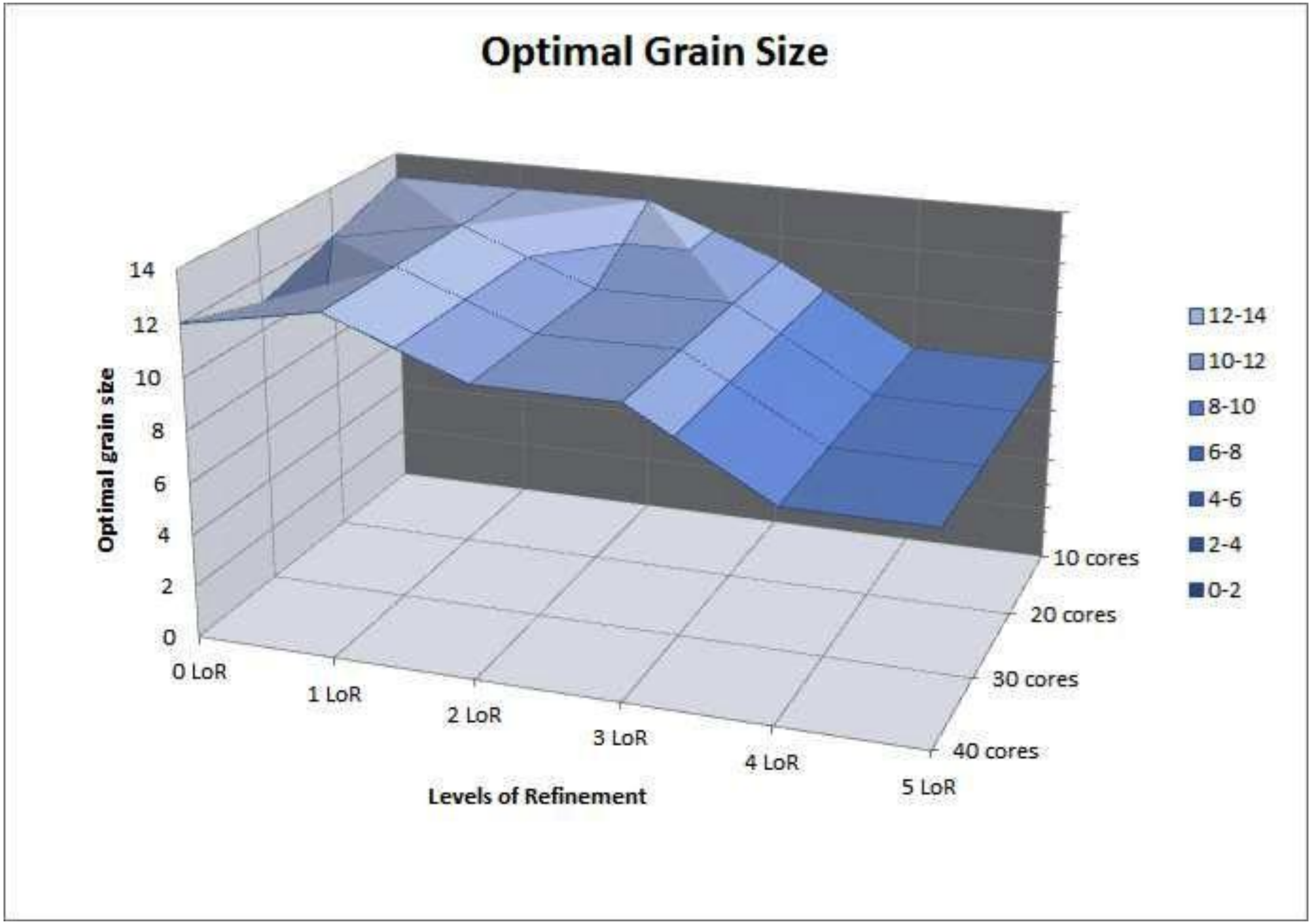


Granularity and Performance

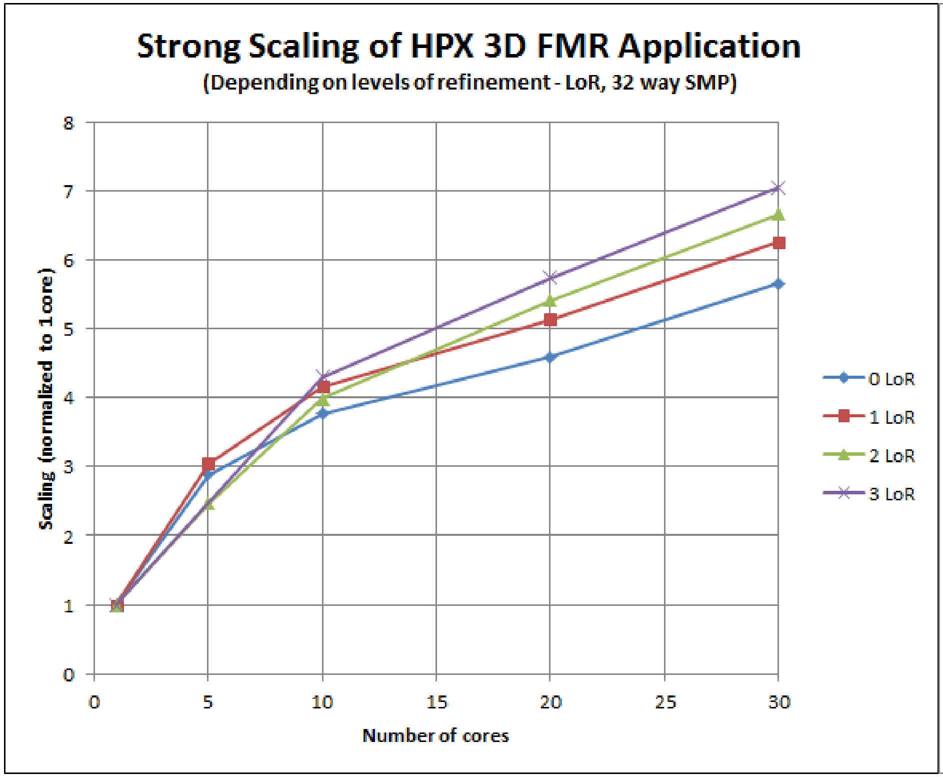


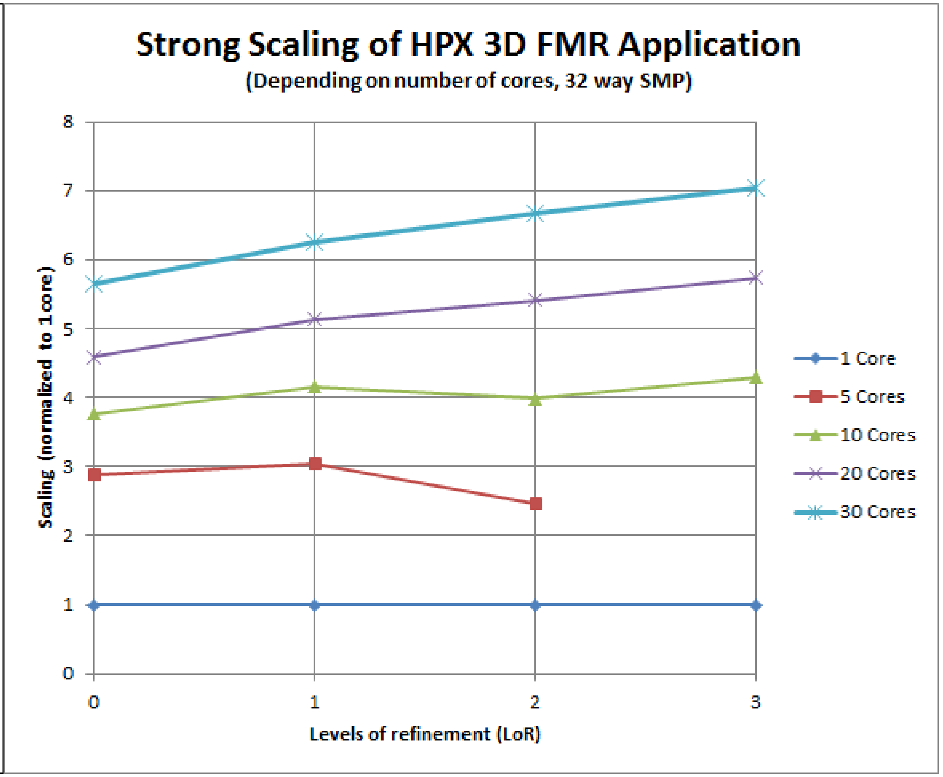


Optimal Grain Size

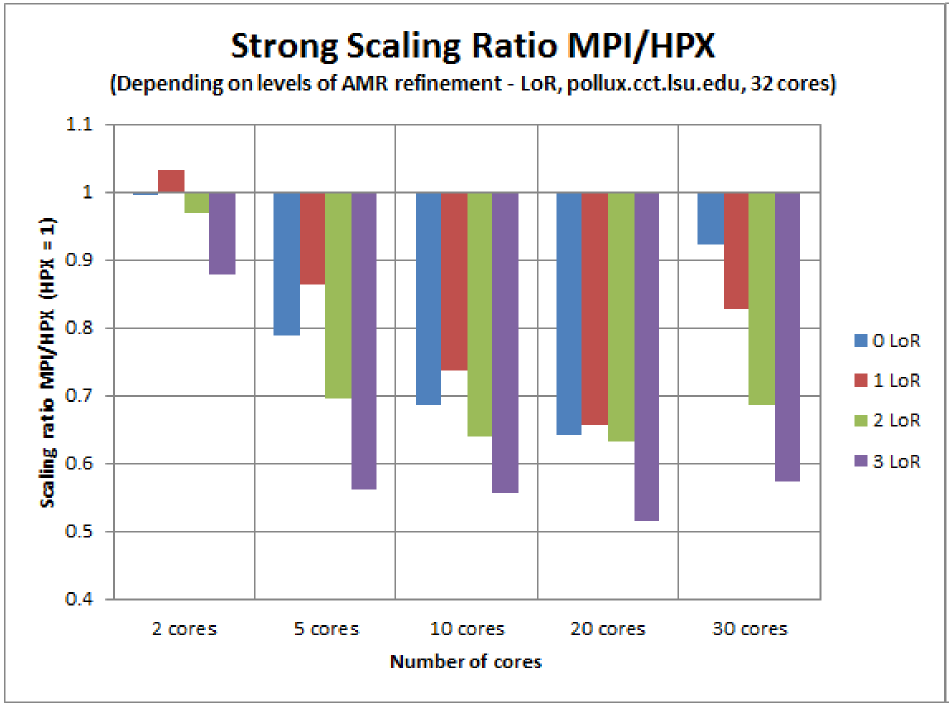


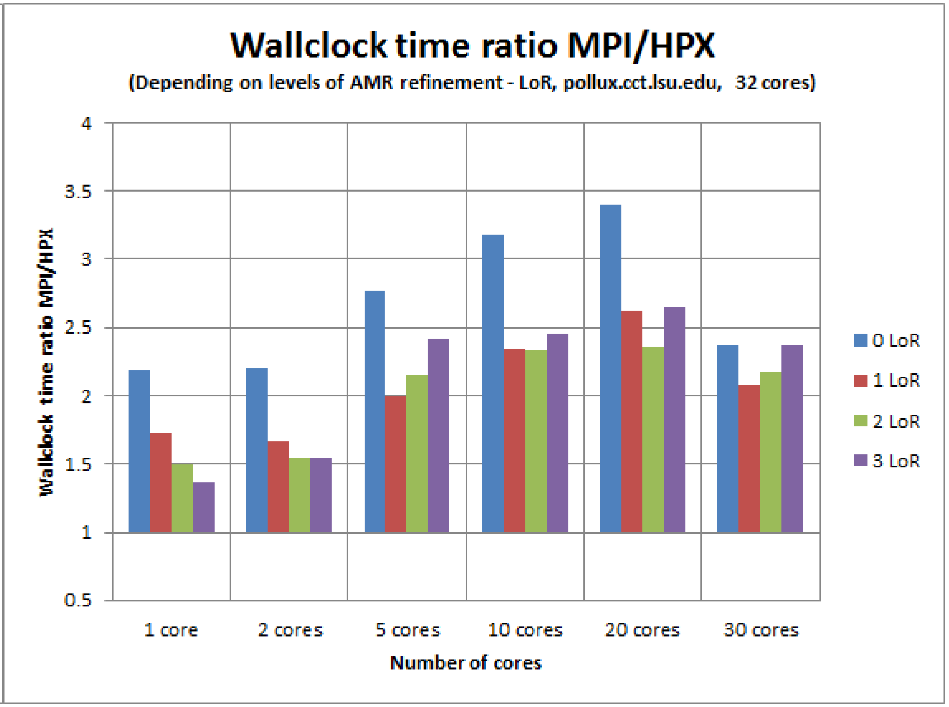
3-D Results


Strong Scaling of HPX 3D FMR Application
(Depending on levels of refinement - LoR, 32 way SMP)
Scaling (normalized to 1 core)
0
1
2
3
4
5
6
7
8
0
5
10
15
20
25
30
Number of cores
0 LoR
1 LoR
2 LoR
3 LoR


Strong Scaling of HPX 3D FMR Application
(Depending on number of cores, 32 way SMP)
Scaling (normalized to 1 core)
0
1
2
3
4
5
6
7
8
0
1
2
3
Levels of refinement (LoR)
1 Core
5 Cores
10 Cores
20 Cores
30 Cores

3-D Results: MPI/HPX

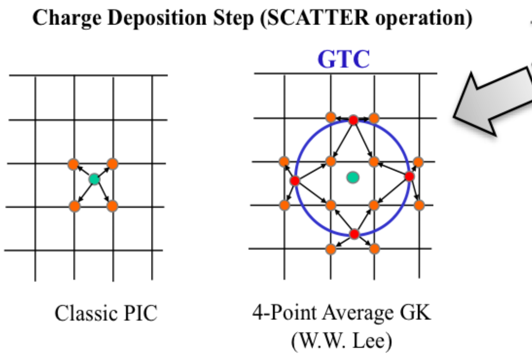




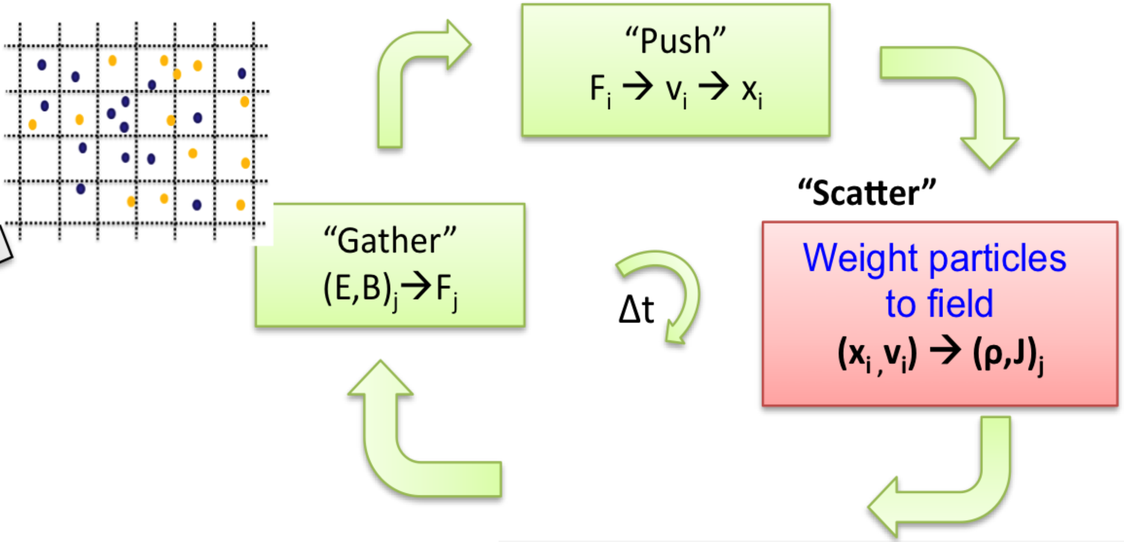
Key Points from this Example

- Message-driven work-queue execution performance can depend significantly on the grain size of the problem. An optimal grain size exists.
- Load balancing can be accomplished implicitly using the work-queue execution approach and thereby substantially improve efficiency compared with the conventional approach

GTC: 3-D Gyrokinetic Toroidal Code



Charge Deposition Step (SCATTER operation)

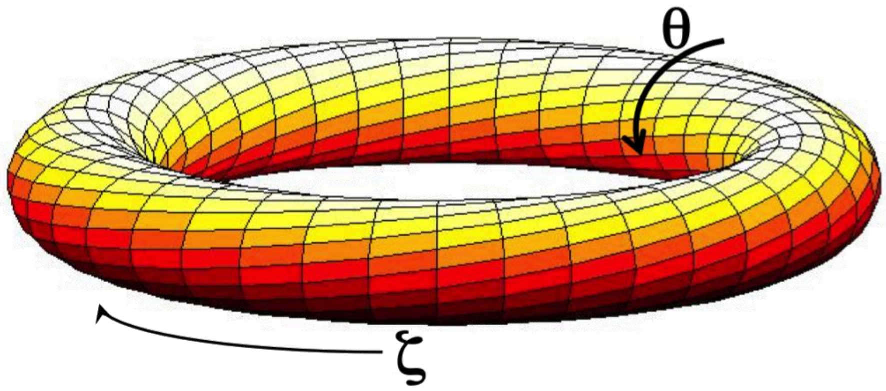


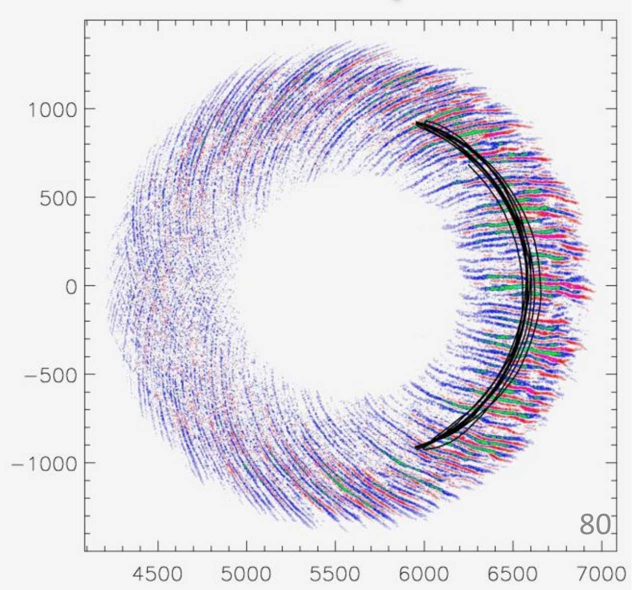
“Push”
$F_i \rightarrow v_i \rightarrow x_i$

“Scatter”
Weight particles to field
$(x_i, v_i) \rightarrow (\rho, J)_j$

“Gather”
$(E,B)_j \rightarrow F_j$

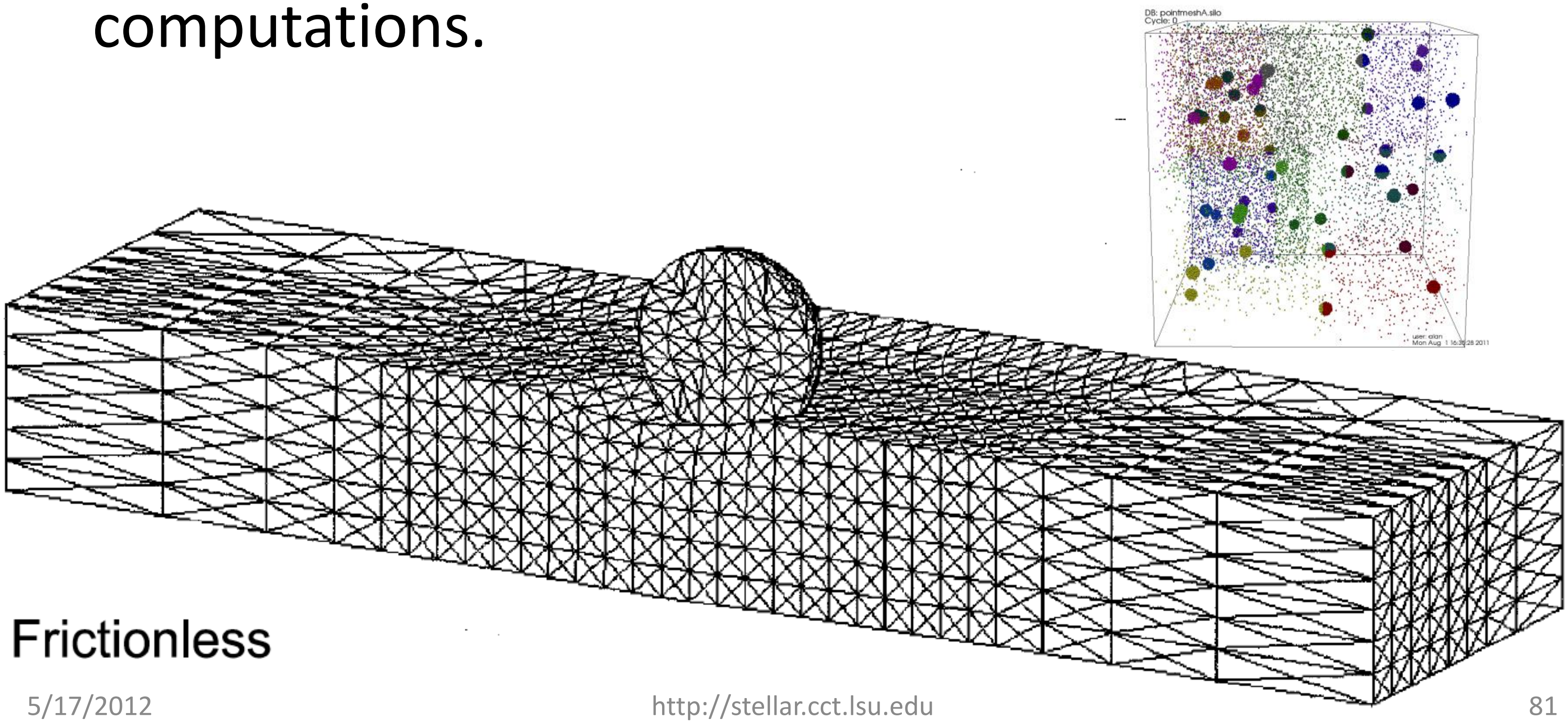
Δt



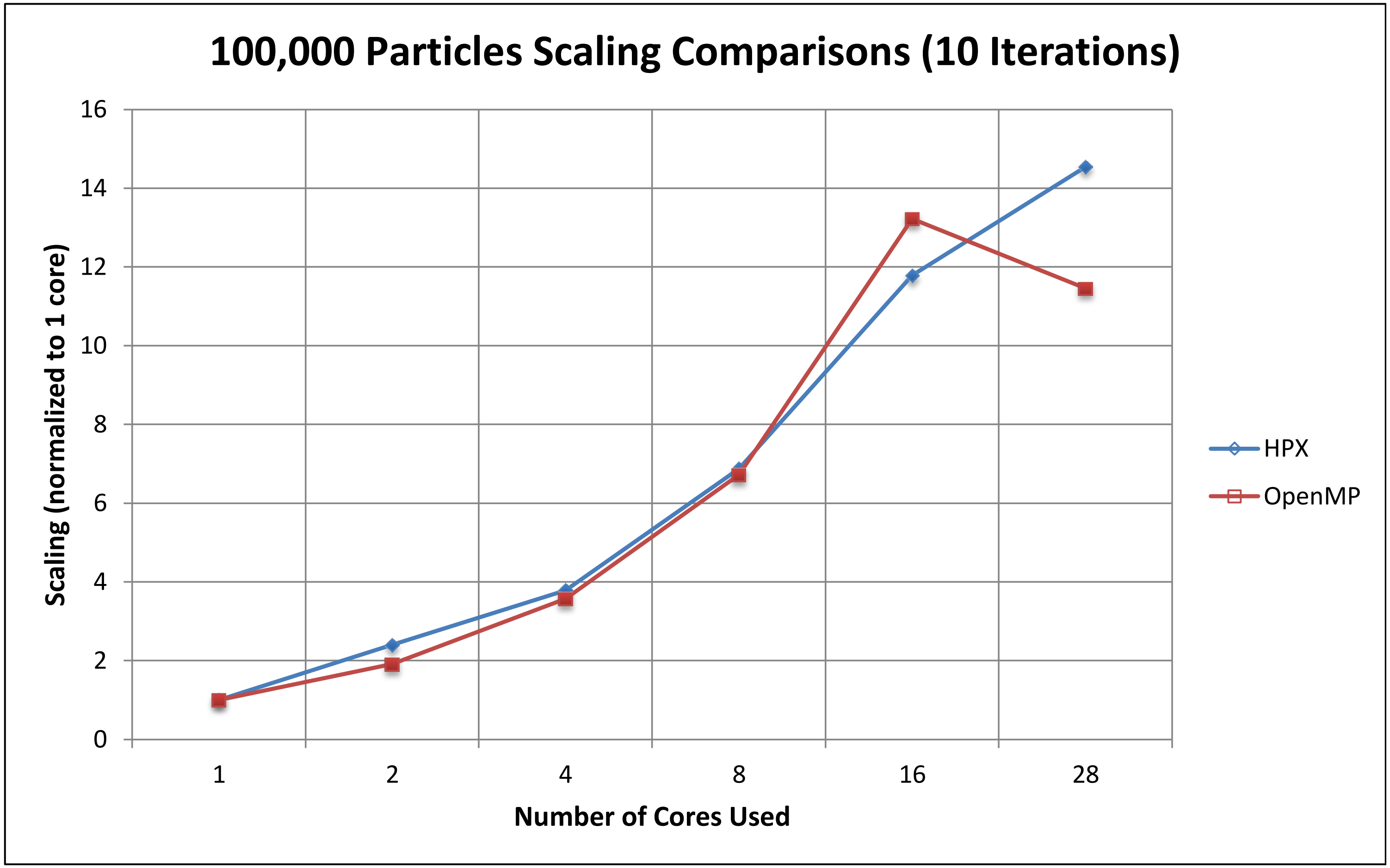


Symmetric Contact for Deformation

- Uses Boost.Geometry for contact iteration in conjunction with futures and AGAS.
- Applications include impulsive loading computations.



N-Body



See Int. J. High Perform C, 11 Apr 2012

CONCLUSIONS

Conclusions

- Message driven, multithreaded approaches can significantly improve performance in scaling constrained applications
- HPX overheads are generally larger than those in OpenMP and MPI
- HPX outperforms conventional approaches when it hides latencies, amortizes contention, and implicitly load balances
- AGAS enables simulation capability not presently available elsewhere
- HPX enables medium and fine grained computation in both SMP and distributed settings

stellar.cct.lsu.edu

For SVN access, contact gopx@cct.lsu.edu