Charm++ & MPI: Combining the Best of Both Worlds

IPDPS: May 27, 2015

Nikhil Jain, Abhinav Bhatele, Jae-Seung Yeom, Mark F. Adams, Francesco Miniati, Chao Mei, Laxmikant V. Kale

Motivation: additional capabilities and code reuse

Motivation: additional capabilities and code reuse

- Multi-physics modeling and coupled simulations require sophisticated techniques, but…
- Most applications developed in a single parallel language
  - Limited features
  - No code reuse across languages

Motivation: additional capabilities and code reuse

- Multi-physics modeling and coupled simulations require sophisticated techniques, but…

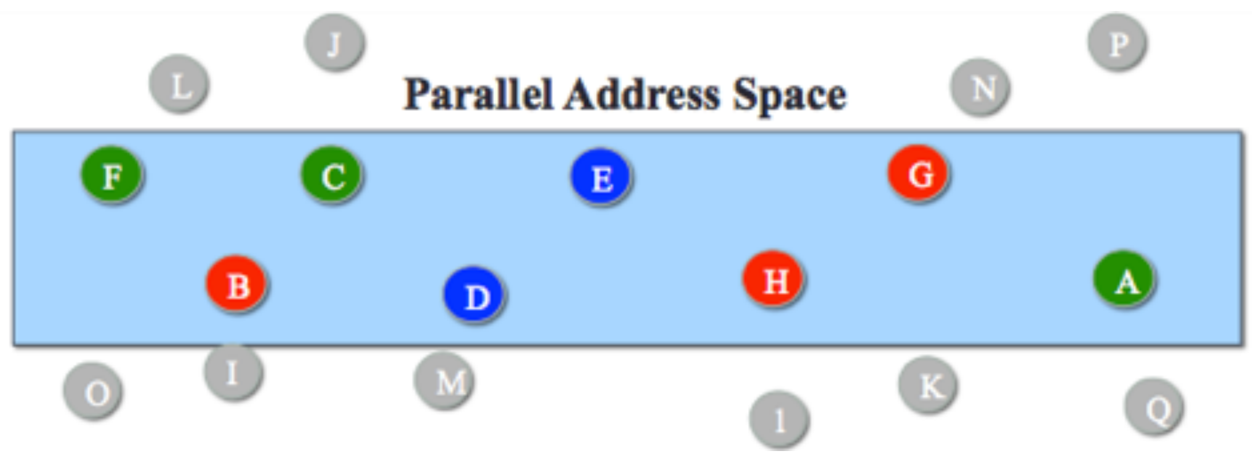
- Most applications developed in a single parallel language
  - Limited features
  - No code reuse across languages

- Interoperation of languages in an application
  - MPI + X, where MPI is across nodes and X is within
  - **MPI + Charm++ : MPI and Charm++ everywhere!**

Charm++: object-based message-driven parallel programming

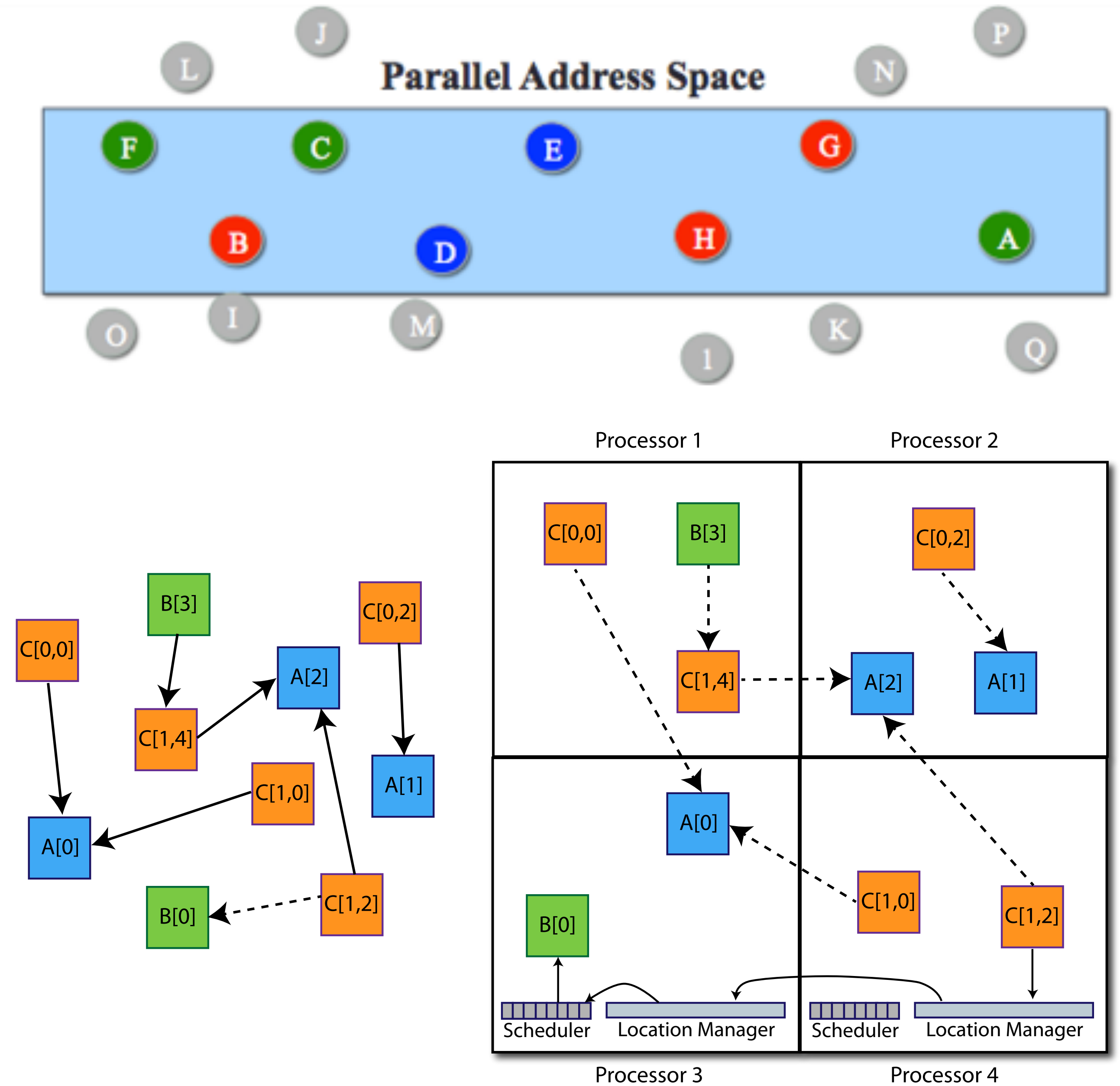
Charm++: object-based message-driven parallel programming

- Fundamental design attributes
  - ➡ Overdecomposition
  - ➡ Asynchronous message driven execution
  - ➡ Migratability
- C++ objects based



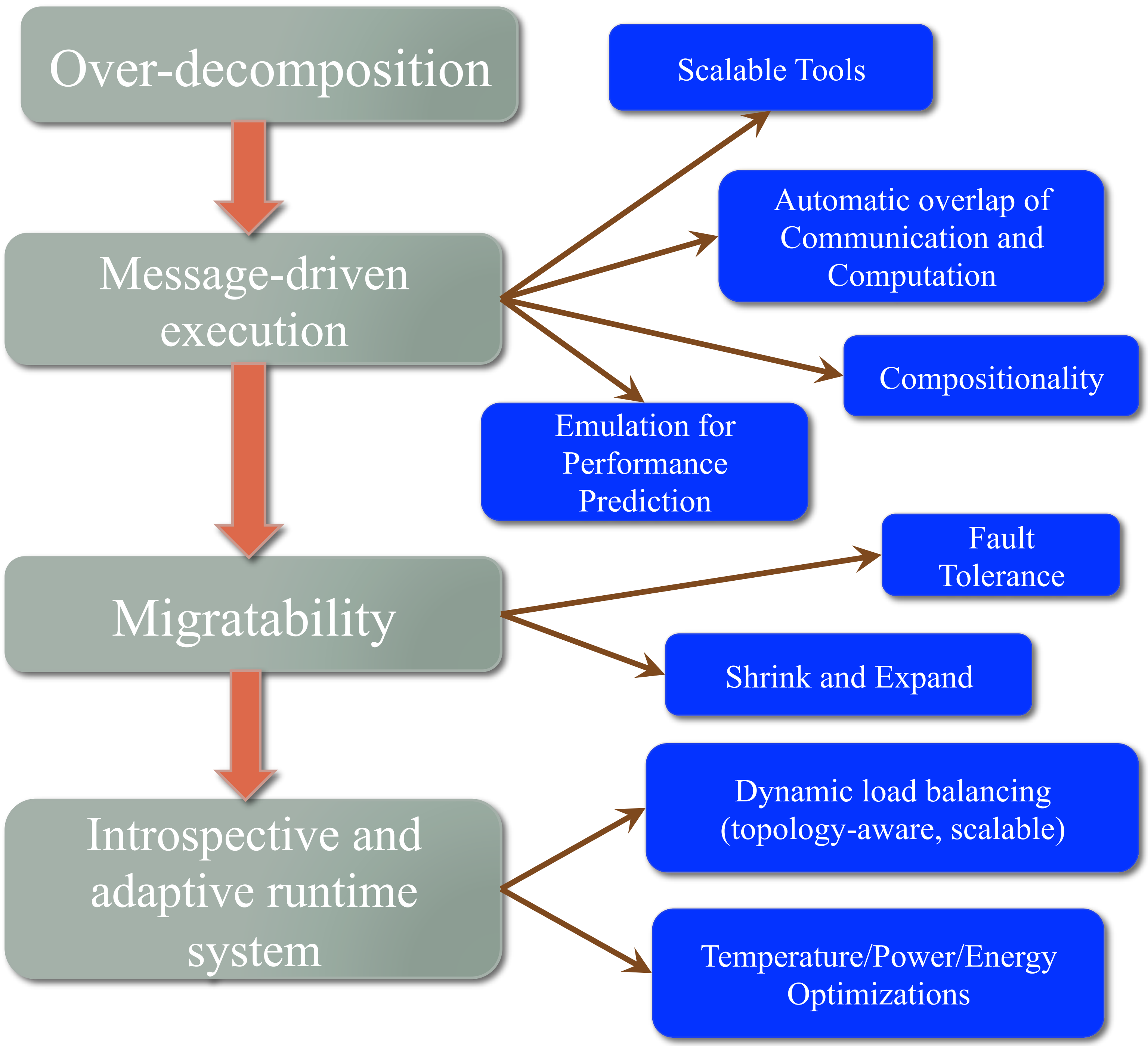
Charm++: object-based message-driven parallel programming

- Fundamental design attributes
  - ➡ Overdecomposition
  - ➡ Asynchronous message driven execution
  - ➡ Migratability
- C++ objects based
- Driven by an adaptive runtime system

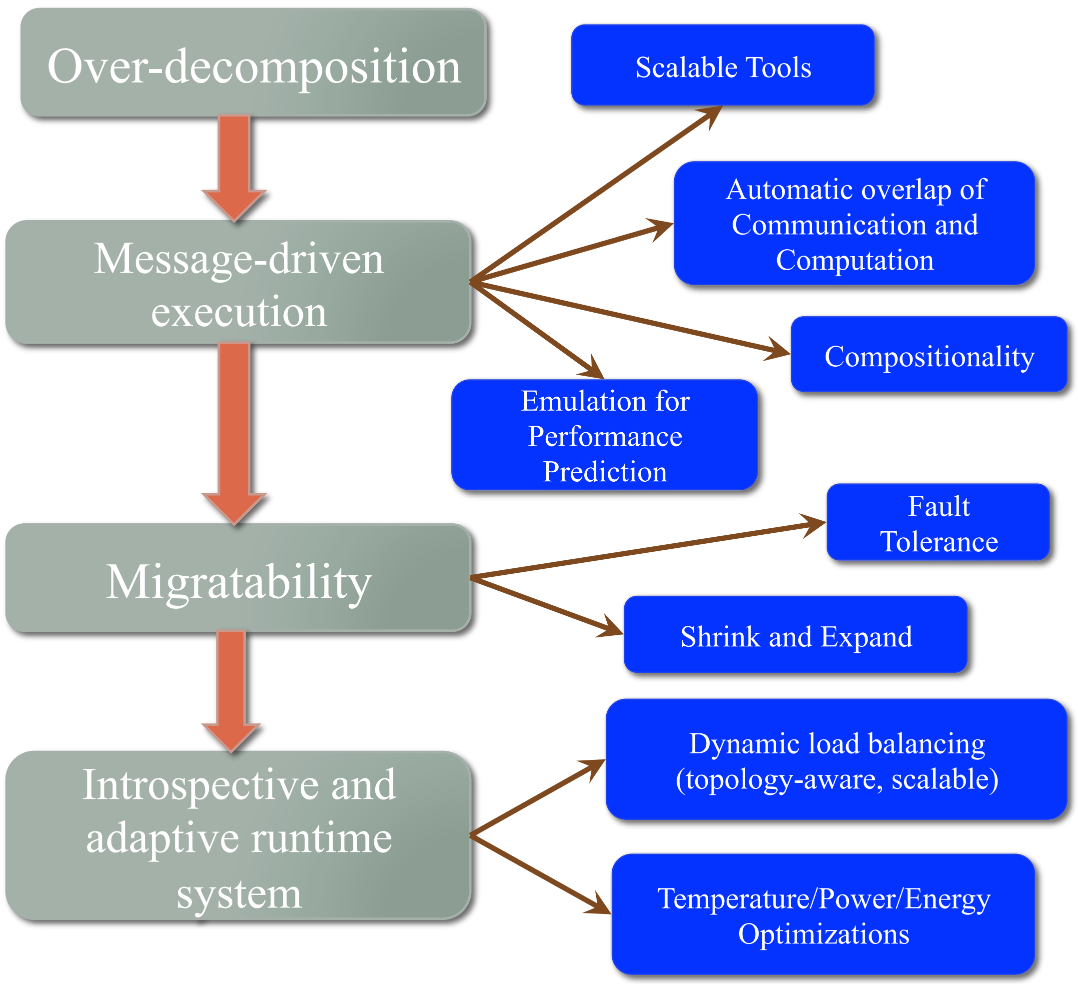


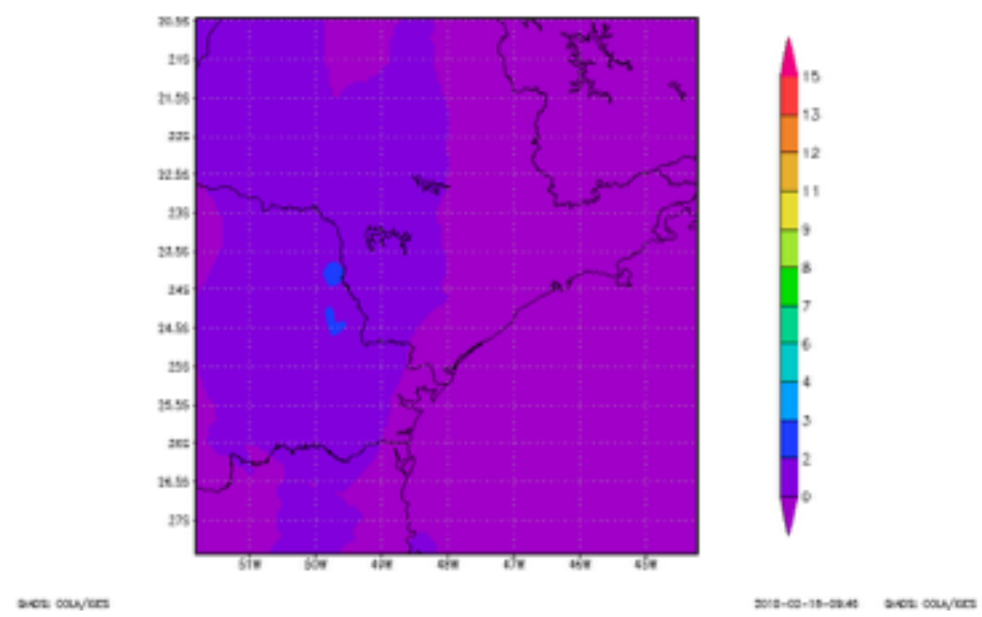
User View and System View

Features: comp-comm overlap, load balancing, introspection…



Features: comp-comm overlap, load balancing, introspection…





Applications: NAMD, ChaNGa, OpenAtom, EpiSimdemics, ClothSim, BRAMS, and many more…

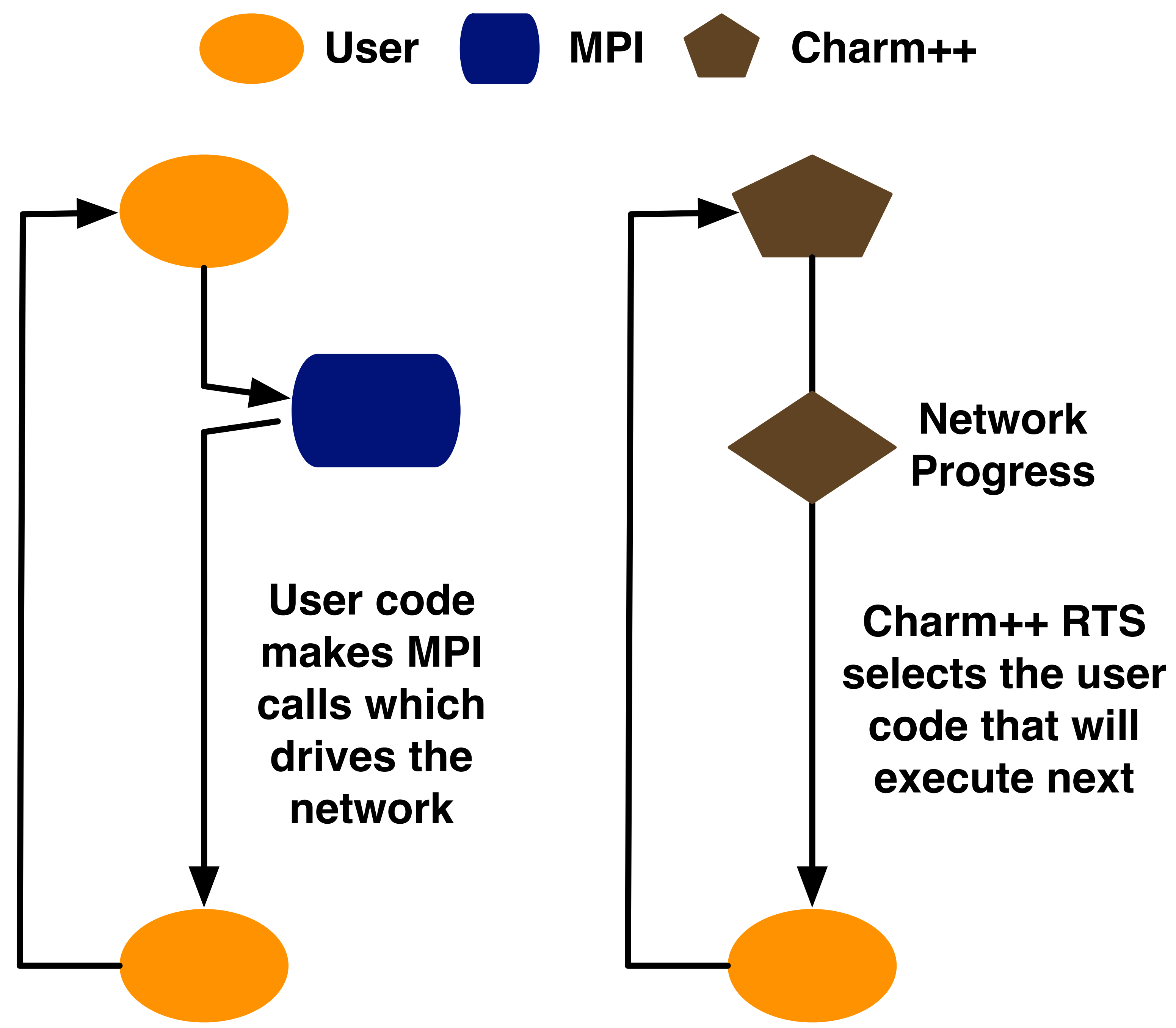
Related Work

- Harper et al. : PVM in Legion environment
- MetaChaos : HPF + Chaos + pC++
- Kale et al. : MPI, PVM, and Charm++ on Converse
- OpenMP + MPI
- Dinan et al. : MPI + UPC
- Zhao et al. : Active messages in MPI

Novelty: control flow, code reuse, and performance studies

- The control flow styles for MPI and Charm++ are different
  - MPI is user-driven, while Charm++ is system-driven
- Minimal (re)implementation of languages
- Focus on reuse of existing code with minor changes!
- In contrast to interoperation via reimplementing MPI on Converse, this scheme works with any MPI
- Demonstration via performance studies at scale

Control flow management in MPI vs Charm++



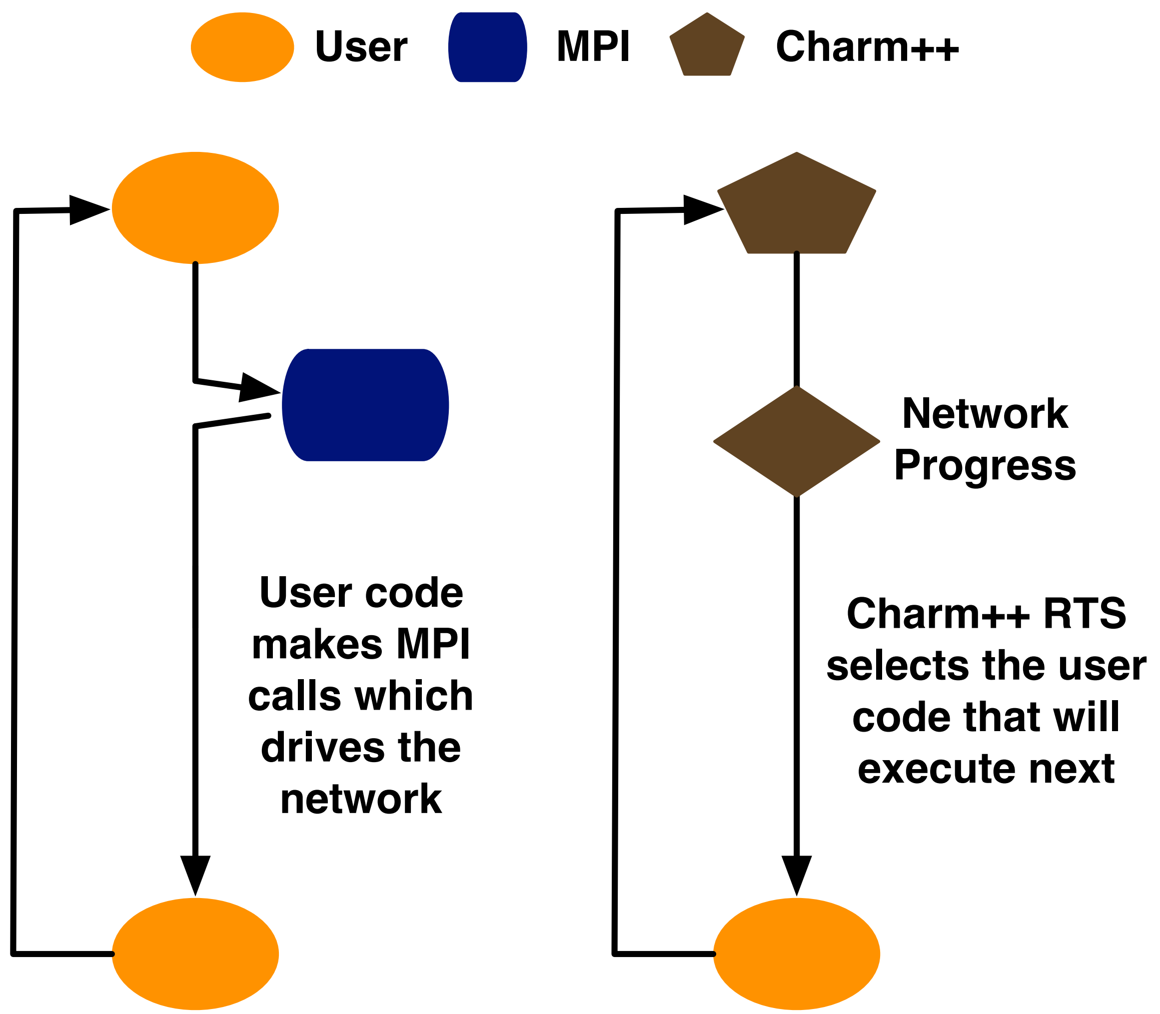
Flow management solution I: concurrent threads

Concurrent Threads: execute each module/language in its own *home* thread

Pros: Easy to understand and implement

Cons:

- Thread scheduling overhead
- Sub-optimal scheduling
- Adaptive scheduling requires significant code changes



Flow management solution II: user controlled transfer

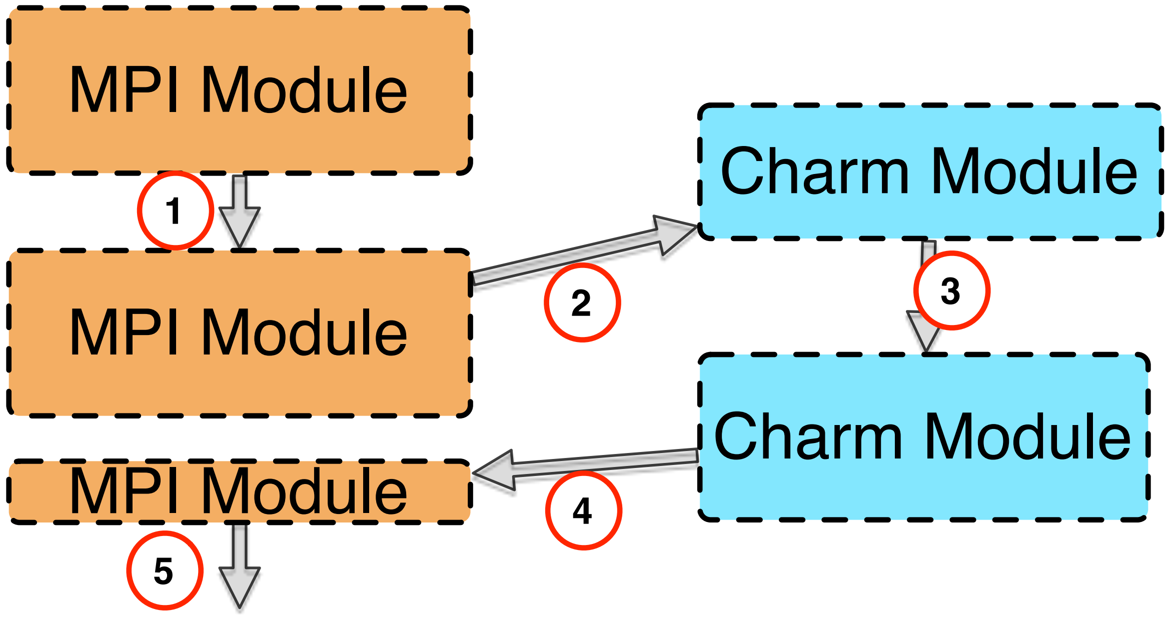
Exposing the Charm++ scheduler at a coarse granularity

Pros:

- Eliminates the thread overheads
- Reuse of existing code is easy

Cons:

- Switching decisions by user *(or is it a disadvantage?)*
- Inter-module overlap is absent



Language APIs: additions to enable interoperation

Language APIs: additions to enable interoperation

- **Initialize:** set up to create a module/language instance
  - ➡ MPI_Init/Comm_create, CharmLibInit

Language APIs: additions to enable interoperation

- **Initialize:** set up to create a module/language instance
  - ➡ MPI_Init/Comm_create, CharmLibInit
- **Execute:** make progress
  - ➡ Implicit in MPI, StartCharmScheduler

Language APIs: additions to enable interoperation

- **Initialize:** set up to create a module/language instance
  - ➡ MPI_Init/Comm_create, CharmLibInit
- **Execute:** make progress
  - ➡ Implicit in MPI, StartCharmScheduler
- **Transfer:** stop execution
  - ➡ Implicit in MPI, StopCharmScheduler/CkExit

Language APIs: additions to enable interoperation

- **Initialize:** set up to create a module/language instance
  - ➡ MPI_Init/Comm_create, CharmLibInit
- **Execute:** make progress
  - ➡ Implicit in MPI, StartCharmScheduler
- **Transfer:** stop execution
  - ➡ Implicit in MPI, StopCharmScheduler/CkExit
- **Clean up:** destroy the instance
  - ➡ MPI_Comm_free, CharmLibExit

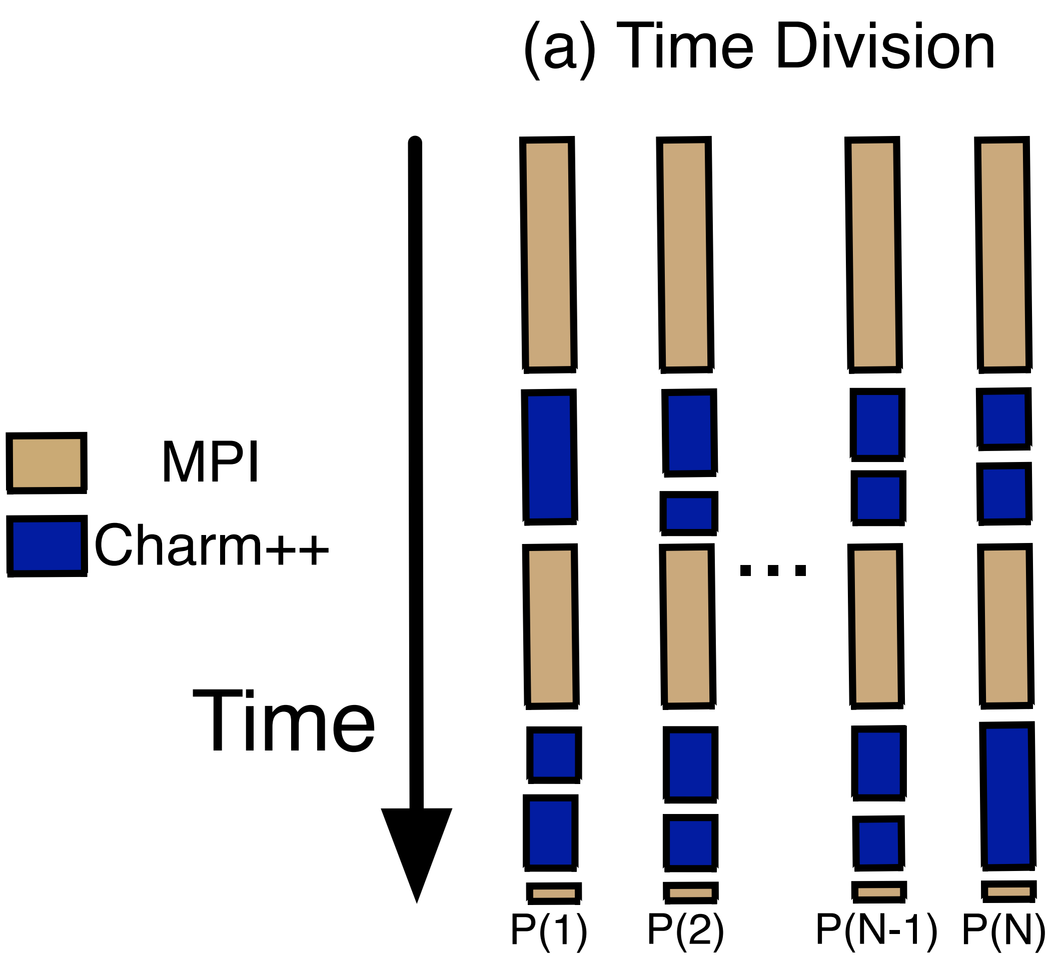
MPI code example: create language instances and execute

```
#include "mpi-interoperate.h"

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_split(MPI_COMM_WORLD, myrank%2, myrank, &newComm);
  if(myrank % 2) {
    // Create Charm++ instance on subset of processes
    CharmLibInit(newComm, argc, argv);
    StartCharm(16); // Call Charm++ library
    CharmLibExit(); // Destroy Charm++ instance
  } else {
    // MPI work on rest of the processes
  }
  MPI_Finalize();
}
```

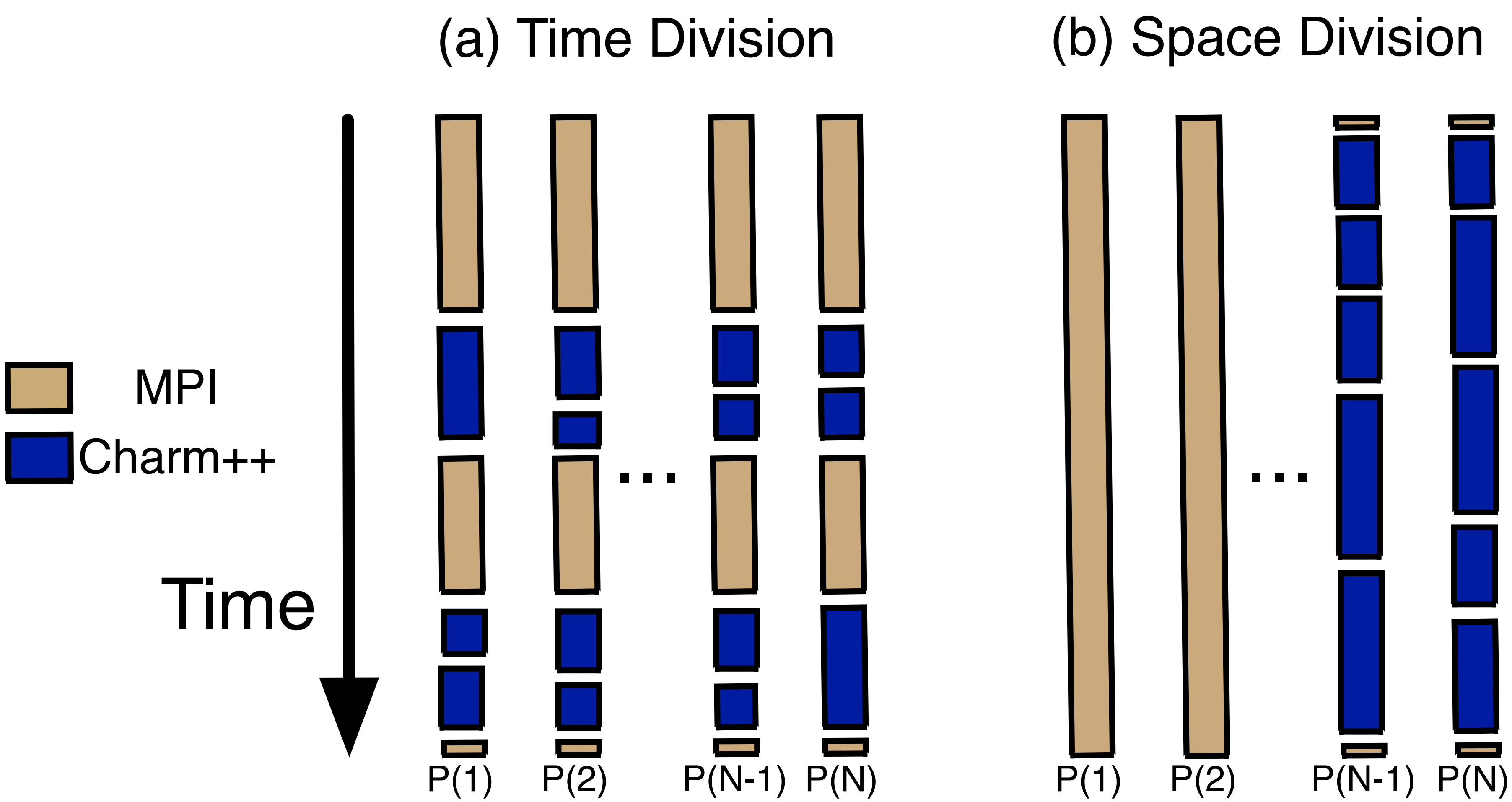
Charm++ code example: interface function

```
#include "mpi-interoperate.h"

// invoked from MPI, marks the beginning of Charm++
void StartCharm(int elems) {
  if(CkMyPe() == 0) {
    workerProxy.StartWork(elems);
  }
  StartCharmScheduler();
}

// Charm++ function that deactivates scheduler
void Worker::StartWork(int elems) {
  // Charm++ work on a subset of processes
  CkExit();
}
```

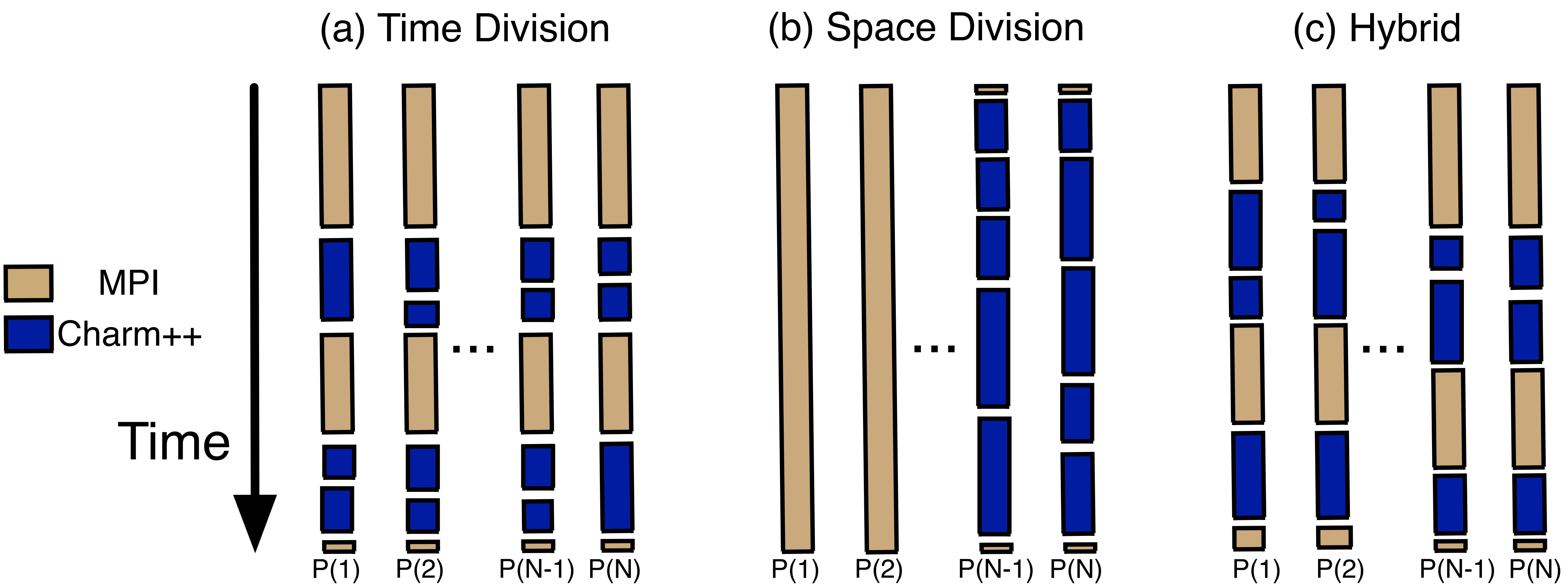
Resource sharing: time, space, and hybrid division



Resource sharing: time, space, and hybrid division



Resource sharing: time, space, and hybrid division



Data Sharing and Rank Mapping

- Data Sharing
  - ➡ Shared memory pointer-based
  - ➡ Data repository
- Rank Mapping - Dinan et al. for MPI + UPC
  - ➡ One to one
  - ➡ Many to one
  - ➡ One to none

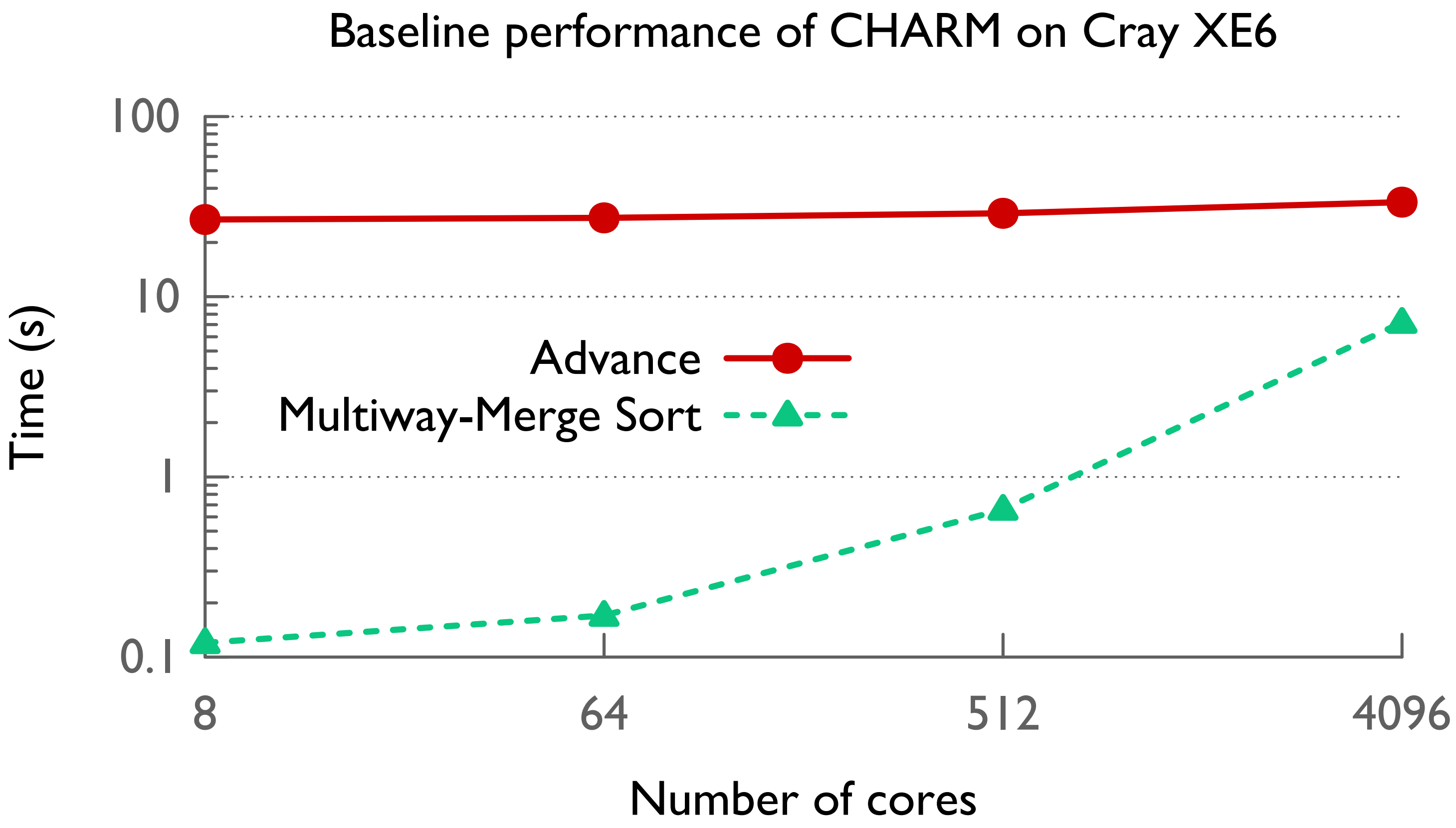
Application Studies

CHARM: scaling bottleneck caused by global sorting

- CHARM is a cosmology code based on Chombo (MPI)
  - Non-uniform particle distribution
  - Load balancing and locality requires global sorting every step

CHARM: scaling bottleneck caused by global sorting

- CHARM is a cosmology code based on Chombo (MPI)
  - Non-uniform particle distribution
  - Load balancing and locality requires global sorting every step



Amount of time spent in sorting increases, while time spent in computation is constant

Scaling Bottleneck!

Eliminating bottleneck via a high performance sorting library

Eliminating bottleneck via a high performance sorting library

- What does efficient sorting need?
  - ➡ Asynchrony and non-blocking communication
  - ➡ Overlap of local sorting with communication

Eliminating bottleneck via a high performance sorting library

- What does efficient sorting need?
  - ➡ Asynchrony and non-blocking communication
  - ➡ Overlap of local sorting with communication
- Option 1: Implement a new MPI based code and optimize it!

Eliminating bottleneck via a high performance sorting library

- What does efficient sorting need?
  - ➡ Asynchrony and non-blocking communication
  - ➡ Overlap of local sorting with communication
- Option 1: Implement a new MPI based code and optimize it!
- Option 2: Reuse an existing sorting library
  - ➡ HistSort - Highly scalable sorting library in Charm++ (Solomonik et al.)

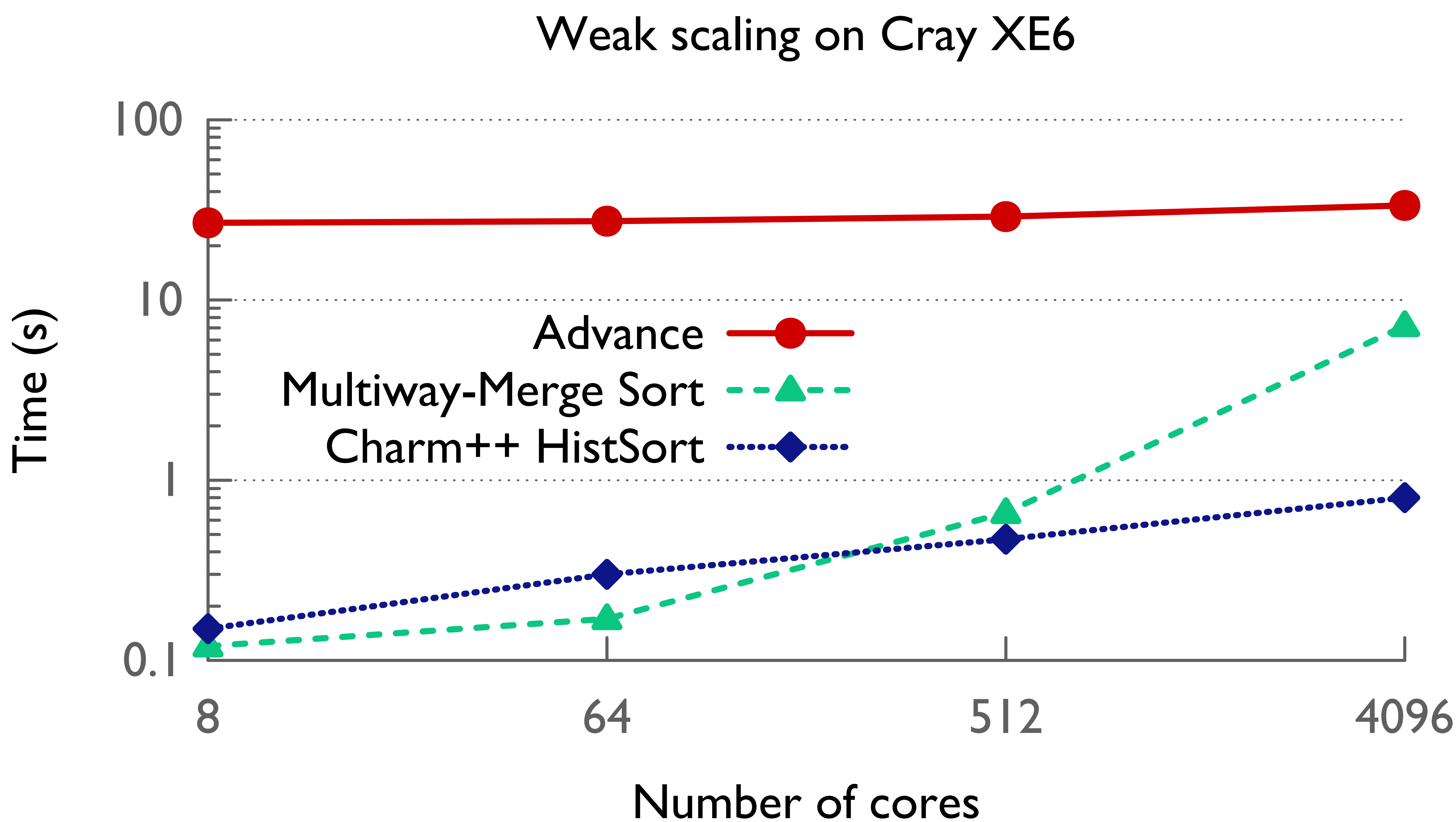
Using HistSort in CHARM: time sharing MPI and Charm++

```
/* CHARM code that prepares the input */
...
@195 lines of Multi-way Merge sort in MPI@
/* Computation code in CHARM */
...

----------------------------------------

/* CHARM code that prepares the input */
...
// call to HistSort
HistSorting<key_type, std::pair<partType,
      char[MAX_PART_SZ]>>(loc_s_len, dataIn,
      &loc_r_len, &dataOut);
/* Computation code in CHARM */
...
```

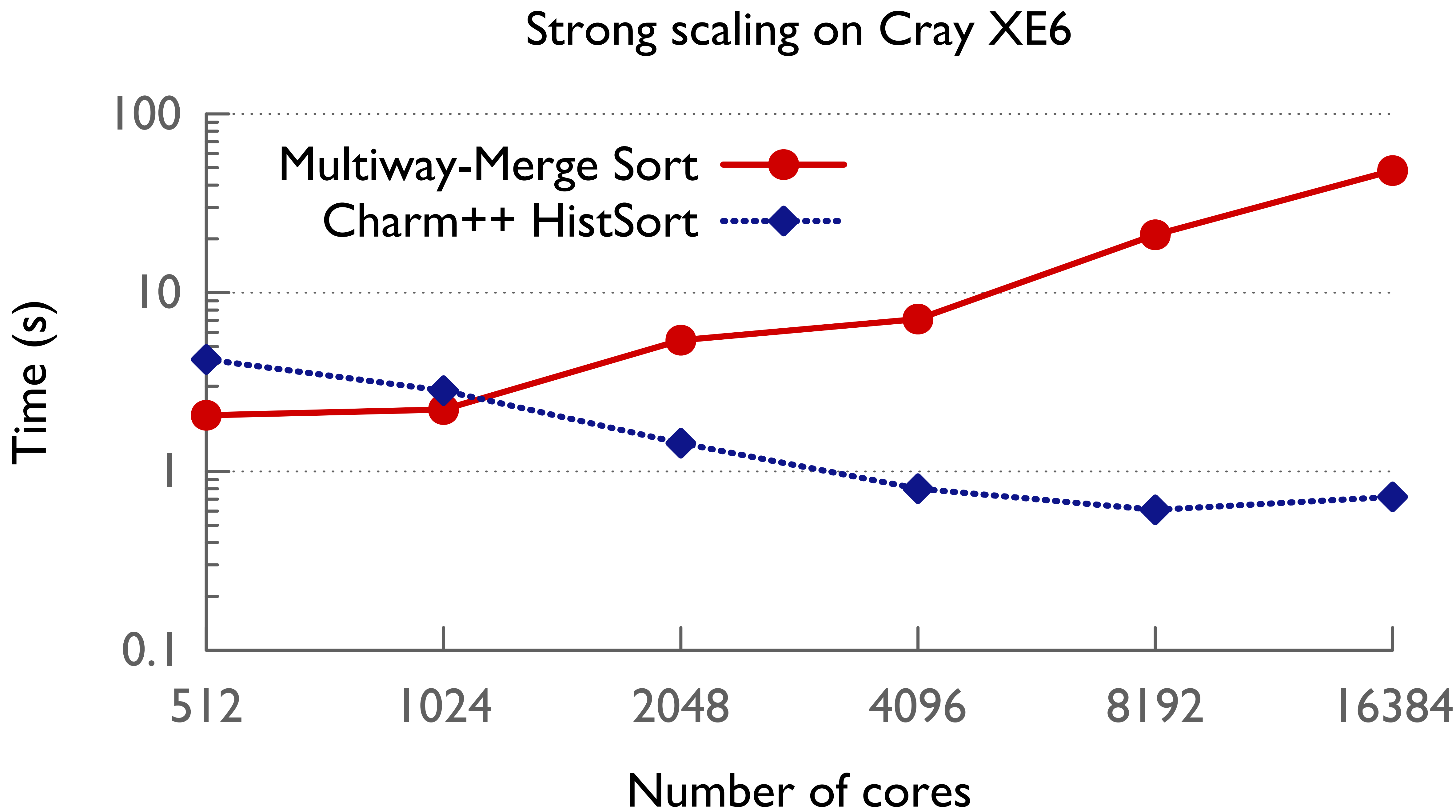
Interoperable HistSort library: minor changes lead to reuse

```
// interface function for HistSort
template <class key, class value>
void HistSorting(int input_elems_, kv_pair<key, value>* dataIn_, int *
output_elems_, kv_pair<key, value>** dataOut_) {
  // store parameters to global locations
  dataIn = (void*)dataIn_;
  dataOut = (void**)dataOut_;
  in_elems = input_elems_;
  out_elems = output_elems_;
  // initiate message to main object
  if(CkMyPe() == 0) {
    static CProxy_Main<key,value> mainProxy =
                              CProxy_Main<key,value>::ckNew(CkNumPes());
    mainProxy.DataReady();
  }
  StartCharmScheduler();
}
```

Weak scaling: time spent in sorting increases slowly



Strong scaling: 48x speed up on 16k cores of Hopper



EpiSimdemics: IO leads to performance and productivity loss

EpiSimdemics: IO leads to performance and productivity loss

- Agent-based simulator used to study spread of contagious diseases over social networks, implemented in Charm++

EpiSimdemics: IO leads to performance and productivity loss

- Agent-based simulator used to study spread of contagious diseases over social networks, implemented in Charm++
- Requires reading many large input files: an hour long startup!
  - Cause: sequential input

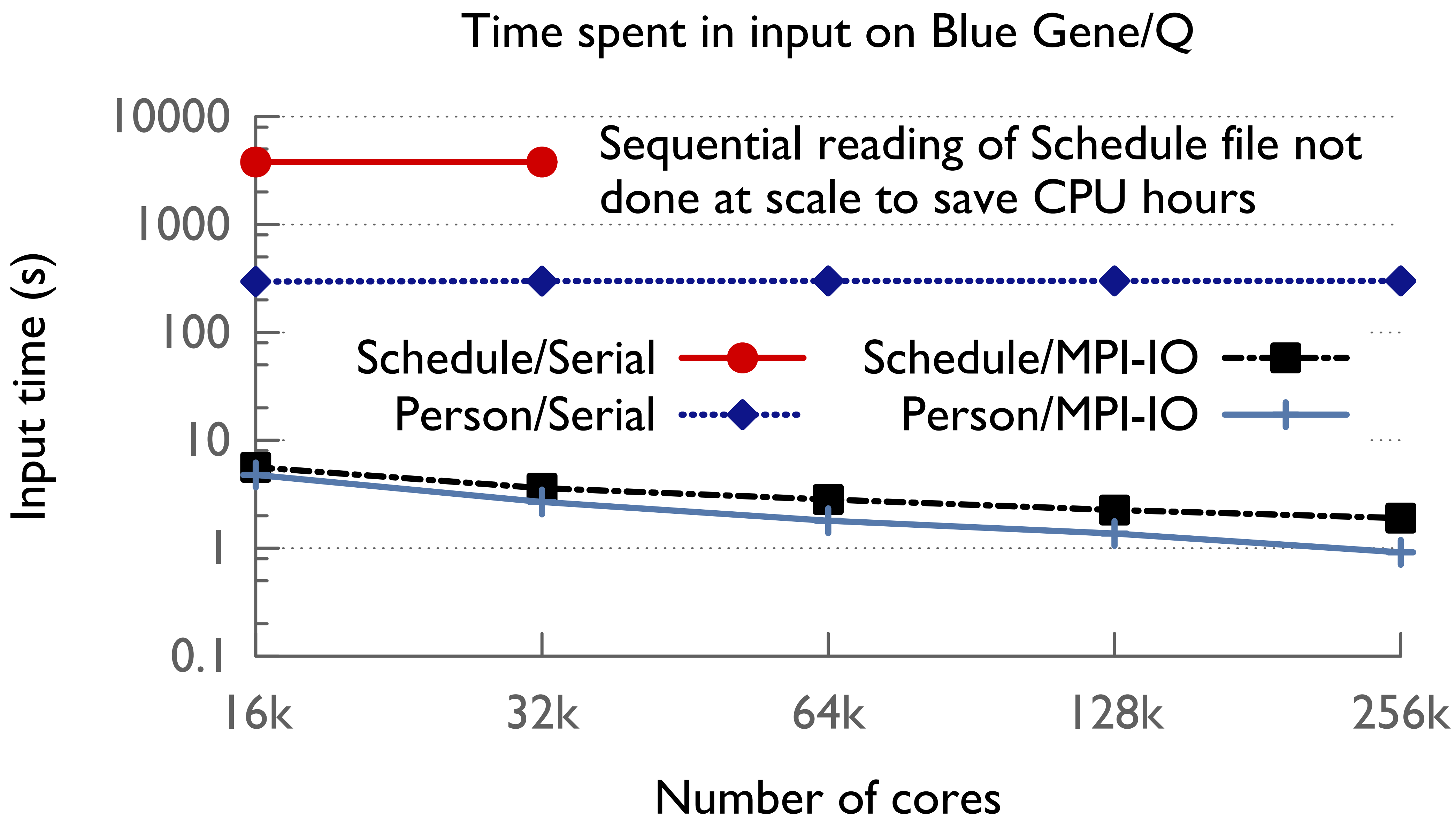
EpiSimdemics: IO leads to performance and productivity loss

- Agent-based simulator used to study spread of contagious diseases over social networks, implemented in Charm++
- Requires reading many large input files: an hour long startup!
  - Cause: sequential input
- Many large output files, written periodically
  - Writes to multiple files, aggregates later
  - Limited number of allowed open file descriptors prevents execution

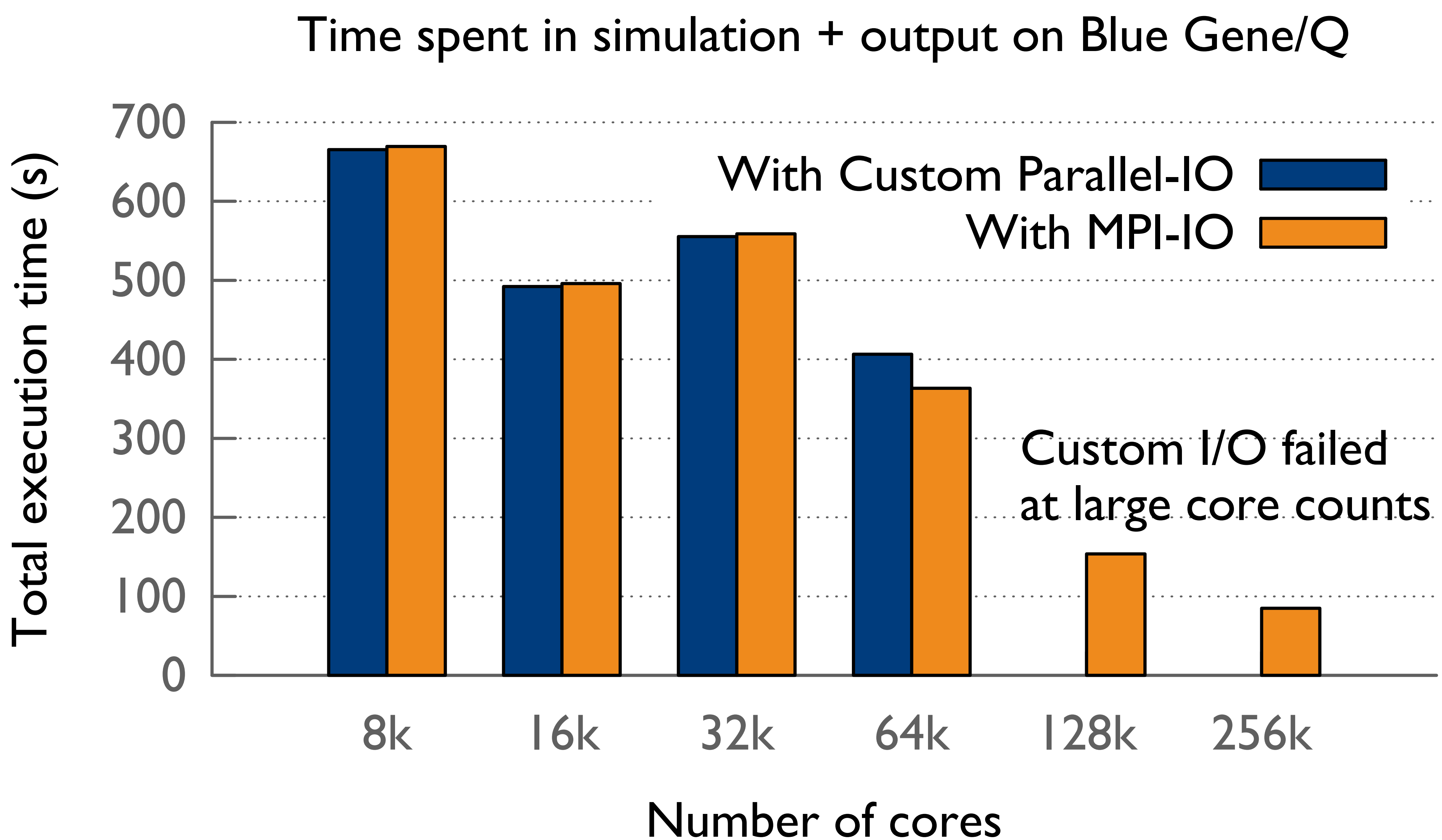
MPI IO with EpiSimdemics

- MPI IO - portable, often vendor-implemented
- Use of MPI collectives to aggregate IO meta-data
- IO module executed in a hybrid manner with rest of the code

Input performance: input time reduced to less than 10s



Output performance: write to single file even on large #cores



Application	Library	Productivity	Performance
CHARM	HistSort	195 lines removed	48x speed up in sorting
EpiSimdemics	MPI IO	Writes to a single file	256x faster input
NAMD	FFTW	280 lines reduction	Similar performance
Load balancing framework	ParMetis	Parallel graph paratitioning	Faster applications

Conclusion

- Interoperating Charm++ and MPI is easy
- Leads to several benefits
- Available in production version of Charm++ along with any MPI implementation:
- http://charmplusplus.org
- http://charm.cs.illinois.edu/manuals/html/charm++/25.html

Questions