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

IPDPS: May 27, 2015

Nikhil Jain, Abhinav Bhavele, 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++ & MPI: Combining the Best of Both Worlds

Nikhil Jain, Parallel Programming Laboratory
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++

User code makes MPI calls which drives the network

Charm++ RTS selects the user code that will execute next
**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

```c
#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();
}
```
#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

(a) Time Division

- **MPI**
- **Charm++**

Time

P(1) P(2) P(N-1) P(N)
Resource sharing: time, space, and hybrid division

(a) Time Division

(b) Space Division

Time

MPI
Charm++

P(1) P(2) P(N-1) P(N)

P(1) P(2) P(N-1) P(N)
Resource sharing: time, space, and hybrid division

(a) Time Division

(b) Space Division

(c) Hybrid

MPI
Charm++

Time

P(1) P(2) P(N-1) P(N)
P(1) P(2) P(N-1) P(N)
P(1) P(2) P(N-1) P(N)
P(1) P(2) P(N-1) P(N)
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

Baseline performance of CHARM on Cray XE6

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

Scaling Bottleneck!
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.)
/* 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 */
...
// 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

```
Weak scaling on Cray XE6

Time (s)

Advance

Multiway-Merge Sort

Charm++ HistSort

Number of cores
```

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

Nikhil Jain, Parallel Programming Laboratory
Strong scaling: 48x speed up on 16k cores of Hopper

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

Nikhil Jain, Parallel Programming Laboratory

Strong scaling on Cray XE6

- Multiway-Merge Sort
- Charm++ HistSort

Time (s)

Number of cores

512 1024 2048 4096 8192 16384
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

Time spent in input on Blue Gene/Q

Sequential reading of Schedule file not done at scale to save CPU hours

Input time (s)

Number of cores

Schedule/Serial
Person/Serial
Schedule/MPI-IO
Person/MPI-IO
Output performance: write to single file even on large #cores

Time spent in simulation + output on Blue Gene/Q

- With Custom Parallel-IO
- With MPI-IO

Custom I/O failed at large core counts

Number of cores:
- 8k
- 16k
- 32k
- 64k
- 128k
- 256k

Total execution time (s):
- 0
- 100
- 200
- 300
- 400
- 500
- 600
- 700
<table>
<thead>
<tr>
<th>Application</th>
<th>Library</th>
<th>Productivity</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARM</td>
<td>HistSort</td>
<td>195 lines removed</td>
<td>48x speed up in sorting</td>
</tr>
<tr>
<td>EpiSimdemics</td>
<td>MPI IO</td>
<td>Writes to a single file</td>
<td>256x faster input</td>
</tr>
<tr>
<td>NAMD</td>
<td>FFTW</td>
<td>280 lines reduction</td>
<td>Similar performance</td>
</tr>
<tr>
<td>Load balancing</td>
<td>ParMetis</td>
<td>Parallel graph partitioning</td>
<td>Faster applications</td>
</tr>
<tr>
<td>framework</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Nikhil Jain, Parallel Programming Laboratory
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