Interoperating MPI and Charm++ for Productivity and Performance

Nikhil Jain[1], Abhinav Bhatele[2], Jae-Seung Yeom[2], Mark F. Adams[3], Francesco Miniati[4], Chao Mei[5], and Laxmikant Kale[1]

[1]University of Illinois at Urbana-Champaign, [2]Lawrence Livermore National Laboratory, [3]Lawrence Berkeley National Laboratory, [4]ETH Zurich, [5]Google Inc.

Introduction

MOTIVATION

Developing multi-physics, coupled applications using one parallel language can be challenging:

a) For optimal performance, different modules may require features provided by distinct parallel programming languages.
b) A good match between module requirements and language features may increase programmer's productivity if the module is implemented in the given language.
c) Various modules may require existing software implemented in other different languages.

INTEROPERATION

- Coordinated use of multiple parallel programming languages in an application, termed interoperation, is a realistic solution to challenges described above.
- We explore interoperation between these two languages:
  - User-driven **MPI** where the programmer explicitly defines the control flow.
  - System-driven**Charm++** where a runtime drives the execution based on the availability of data.

CHALLENGES

- Given the difference in control flow between Charm++ and MPI, how is control transferred and managed between them?
- How are resources (such as cores and network) and data shared between the two programming languages?
- How easy-to-use and scalable is the interoperation methodology for production applications?

Framework

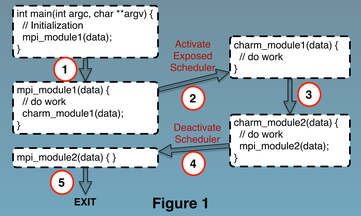


Figure 1

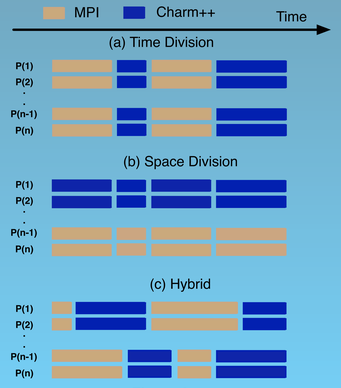


Figure 2

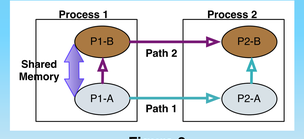


Figure 3

CONTROL FLOW

- Expose Charm++ scheduler to the programmer: new API to activate and deactivate the scheduler.
- Explicit control transfer by the programmer: begin in MPI, switch to Charm++ by invoking its scheduler, switch back to MPI by stopping the scheduler, and so on.
- Switching control infrequently is advisable for ease of programming and good performance.

RESOURCE SHARING

✓ Automatic shared use of low-level network resources via distinct communication domains, e.g. create clients using PAMI_Client_create on Blue Gene/Q.
✓ Three schemes available for sharing cores:
1. *Time Division*: execute either Charm++ or MPI on all cores at a given time during execution.
2. *Space Division*: statically divide cores into subsets that execute either Charm++ or MPI.
3. *Hybrid Division*: divide cores into subsets that alternate between MPI and Charm++ together.
✓ API for division of cores based on MPI communicators.

DATA SHARING

✓ Method 1: A generic *data transfer repository* to which each language module deposits (or queries) the data. Under the hood, communication performed by the repository.
✓ Method 2: *Pointer-based* data sharing via reserved memory within nodes assisted by explicit communication by programmer code.

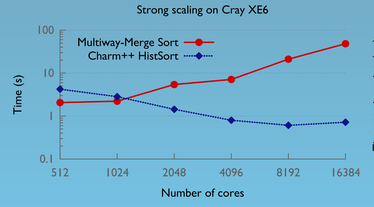
Examples and Results

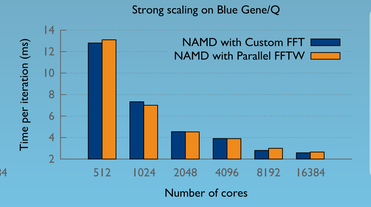
```
#include "mpi-interoperate.h"

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);
    HiStart(16); // Call Charm++ library
    CharmLibExit(); // Destroy Charm++ instance
  } else {
    // MPI work on rest of the processes
  }
  MPI_Finalize();
}
```

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

CHARM code flow with Multi-way Merge Sort
-----------------------------------------
/* CHARM code that prepares the input */
...
// call to HistSort
HistSorting<key_type, std::pair<partType,
    char[MAX_FART_SZ]>>(loc_s_len, dataIn,
    &loc_r_len, &dataOut);
/* Computation code in CHARM*/
...

CHARM code flow with Charm++'s HistSort
```

```
// 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();
}
```

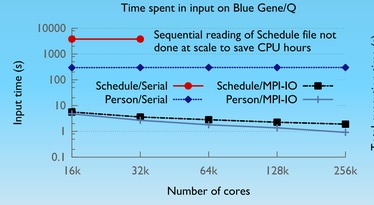
Left figure: basic MPI example to interoperate with Charm++; lines in red are the only additional code required.
Centre and right figures: all changes required for calling HistSorting library in Charm++ from a production MPI code.s

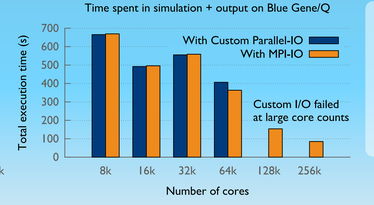




Charm++-based NAMD using MPI-based FFTW with *space-division*: similar performance obtained without the need to maintain a custom FFT library in Charm++.

MPI-based Chombo using Charm++-based HistSorting with *time-division*: the performance of global sorting, which is the scaling bottleneck, is improved by 48x with minimal changes to the code (as shown in the figures at the top)





Charm++-based EpiSimdemics using MPI-IO with *hybrid-division*: scaling bottleneck due to file reading is eliminated, while file output is enabled at scale.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-POST-662677).