An Extension of Charm++ to Optimize Fine-Grained Applications

Alexander Frolov
frolov@dislab.org

Data-Intensive Systems Laboratory (DISLab), NICEVT

14th Annual Workshop on Charm++ and its Applications,
Urbana-Champaign, Illinois, April 19-20, 2016.
Talk Outline

• Introduction
  ■ Fine-grained vs. Coarse-grained Parallelism
  ■ Approaches to Large-scale Graph Processing in Charm++
  ■ Problems of Expressing Vertex-centric Model in Charm++

• uChareLib Programming Model
  ■ uChareLib Programming Model & Library Design
  ■ Comparing uChareLib & Charm++ (on Alltoall)

• Performance Evaluation
  ■ HPCC RandomAccess
  ■ Graphs: Asynchronous Breadth-first Search
  ■ Graphs: PageRank
  ■ Graphs: Single Source Shortest Paths
  ■ Graphs: Connected Components

• Conclusion & Future plans
Fine-grained vs. Coarse-grained Parallelism

**Fine-grained:**
- large number of processes/threads (≫ #CPUs), can be dynamically changed
- small messages (payload up to ∼ 1Kb)
- dynamic partitioning of problem
- load balancing

Applications where fine-grained parallelism can be *naturally* obtained:
- PDE solvers (unstructured, adaptive meshes)
- Graph applications
- Molecular dynamics
- Discrete simulation
- etc.

**Coarse-grained:**
- number of processes/threads equals #CPUs
- large messages (payload from 1Kb)
- static workload assignment
- load balancing is a rare case

Applications where coarse-grained parallelism can be *naturally* obtained:
- PDE solvers (fixed structured meshes)
- Rendering
- etc.

**Common HPC practice:** due to performance reasons to coarsen granularity by aggregating objects/messages and increasing utilization of system resources
Approaches to Large-scale Graph Processing on Charm++

Vertex-centric [= Fine-grained] vs Subgraph-centric [= Coarse/Medium-grained]

- **Vertex-centric**
  - Graph (G) – array of chares distributed across parallel processes (PE)
  - Vertex – char (1:1)
  - Vertices communicate via asynchronous active messages (entry method calls)
  - Program completion detected by CkStartQD

- **Subgraph-centric**
  - Graph (G) – array of chares distributed between parallel processes (PE).
  - Vertex – char (n:1), any local representation possible
  - Algorithms consist of local (sequential) and global parts (parallel, Charm++).
  - Application level optimizations (aggregation, local reductions, etc.)
  - Program completion detected by CkStartQD or manually
HPCC RandomAccess

Table size/PE: $2^{20} \times 8$ bytes, HPC system: [x2 Xeon E5-2630]/IB FDR

RandomAccess, np=8, ppn=8

Performance, GUP/s

charm-randomaccess

tram-randomaccess
uChareLib Programming Model & Design

- uChareLib (micro-Chare Library) – small extension of Charm++, providing opportunity to mitigate overheads of RTS for fine-granular parallelism:
  - `uchare` object is introduced to Charm++ model
  - `entry` method calls are supported for `uchares`
  - `uchare array` is provided to define arrays of uchares (same as chare array)
  - `uchares` are distributed between common `chares`
  - message aggregation is supported inside `uChareLib`
  - new entry method type `reentrant` (only for uchares)

- uChareLib can be downloaded from https://github.com/DISLab/xcharm
Example: Charm++ vs. uChareLib

Charm++ (alltoall.ci):

```cpp
mainmodule charm_alltoall {
  ...
  // Declaration of chare array
  array [1D] Alltoall {
    entry Alltoall();
    entry void ping();
    entry void run();
  };
  ...
};
```

Charm++ (alltoall.C):

```cpp
class Alltoall : public CBase_Alltoall {
private:
  CmiUint8 counter;
public:
  Alltoall() : counter(1) {
    contribute(CkCallback(CkReductionTarget(
        TestDriver, init), driverProxy));
  }
  #entry*/ void run() {
    for (CmiUint8 i = 0; i < N; i++)
      if (i != thisIndex) thisProxy[i].ping();
  }
  #entry*/ void ping() {
    if (++counter == N)
      contribute(CkCallback(CkReductionTarget(
        TestDriver, done), driverProxy));
  }
};
```

uChareLib (alltoall.ci):

```cpp
mainmodule ucharelib_alltoall {
  ...
  // Declaration of uchare array
  uchare array [1D] Alltoall {
    entry Alltoall();
    entry void ping();
    entry void run();
  };
  ...
};
```

uChareLib (alltoall.C):

```cpp
class Alltoall : public CBase_uChare_Alltoall {
private:
  CmiUint8 counter;
public:
  Alltoall(const uChareAttr_Alltoall & attr) :
    counter(1), CBase_uChare_Alltoall(attr) {
    contribute(CkCallback(CkReductionTarget(
        TestDriver, init), driverProxy));
  }
  #entry*/ void start() {
    for (CmiUint8 i = 0; i < N; i++)
      if (i != thisIndex) thisProxy[i]->ping();
  }
  #entry*/ void ping() {
    if (++counter == N)
      contribute(CkCallback(CkReductionTarget(
        TestDriver, done), driverProxy));
  }
};
```
Performance Evaluation

HPCC RandomAccess

Algorithm RandomAccess

1: Index ← Pseudo random indices
2: for all i ∈ Index do
3: Table[i] ← Table[i] ⊕ i
4: end for

- Original TRAM implementation is used (from Charm++ trunk)
- Charm++ & uChareLib implementations are simple conversions from TRAM based RandomAccess code

NB: update function does not contain calls to other chares => no nested calls (insertData/entry method) for TRAM and uChareLib
Performance Evaluation
HPCC RandomAccess

- Original TRAM implementation is used (from Charm++ trunk)
- Charm++ & uChareLib implementations are simple conversions from TRAM based RandomAccess code

Algorithm RandomAccess

1. $Index \leftarrow \text{Pseudo random indices}$
2. for all $i \in Index$ do
3. $Table[i] \leftarrow Table[i] \oplus i$
4. end for

Charm++ & uChareLib (randomAccess.C):

```c
void Updater::generateUpdates() {
    int arrayN = N - (int) log2((double) numElementsPerPe);
    int numElements = CkNumPes() * numElementsPerPe;
    CmiUInt8 key = HPCC_starts(4 * globalStartmyProc);
    for(CmiInt8 i = 0; i < 4 * localTableSize; i++) {
        key = key << 1 ^ ((CmiInt8) key < 0 ? POLY : 0);
        int destinationIndex = key >> arrayN & numElements - 1;
        thisProxy[destinationIndex].update(key);
    }
}
```

TRAM (randomAccess.C):

```c
void Updater::generateUpdates() {
    int arrayN = N - (int) log2((double) numElementsPerPe);
    int numElements = CkNumPes() * numElementsPerPe;
    CmiUInt8 key = HPCC_starts(4 * globalStartmyProc);
    for(CmiInt8 i = 0; i < 4 * localTableSize; i++) {
        key = key << 1 ^ ((CmiInt8) key < 0 ? POLY : 0);
        int destinationIndex = key >> arrayN & numElements - 1;
        localAggregator->insertData(key, destinationIndex);
    }
    localAggregator->done();
}
```
Performance Evaluation
HPCC RandomAccess (\(N=2^{20}/\text{PE}\)), HPC system: [x2 Xeon E5-2630]/IB FDR

![Graphs showing performance evaluation results](image-url)
Performance Evaluation

HPCC RandomAccess (N=2^{22}/PE), HPC system: [x2 Xeon E5-2630]/IB FDR

RandomAccess (n=22, nodes=2)

RandomAccess (n=22, nodes=4)

RandomAccess (n=22, nodes=8)

RandomAccess (n=22, nodes=16)
Performance Evaluation

PageRank

- **Problem description:**
  - Iteratively compute ranks for all $v \in G$
  
  $$PR_v^{i+1} = \left(1-d\right)/N + d \times \sum_{u \in \text{Adj}(v)} PR_u^i / Lu$$

- **Implementations:**
  - Charm++, naive
  - Charm++, with incoming msg counting
  - TRAM, naive
  - uChareLib, naive

---


NB: update function does not contain calls to other chares $\Rightarrow$ no nested calls (insertData/entry method) for TRAM and uChareLib
Performance Evaluation

PageRank, naive algorithm

Algorithm Naive PageRank

1: function PageRankVertex:: doPageRankStep_init
2:   \[ PR_{\text{old}} \leftarrow PR_{\text{new}} \]
3:   \[ PR_{\text{new}} \leftarrow \frac{(1.0 - d)}{N} \]
4: end function
5: function PageRankVertex::doPageRankStep_update
6:   for \( u \in \text{AdjList} \) do
7:     thisProxy[u].update(\( PR_{\text{old}}/L \))
8:   end for
9: end function
10: function PageRankVertex::update(r)
11:   \[ PR_{\text{new}} \leftarrow d \times r \]
12: end function
13: function TestDriver::doPageRank
14:   for \( i = 0; i < \text{N\_iters}; i \leftarrow i + 1 \) do
15:     g.doPageRankStep_init()
16:     CkStartQD(CkCallbackResumeThread())
17:     g.doPageRankStep_update()
18:     CkStartQD(CkCallbackResumeThread())
19:   end for
20: end function
Performance Evaluation
PageRank, with counting incoming messages

Algorithm PageRank /w msg counting

1: function PageRankVertex::doPageRankStep
2: \( PR_{old} \leftarrow (n_{iter} \% 2)?rank0 : rank1 \)
3: for \( u \in \text{AdjList} \) do
4: \( \text{thisProxy}[u].update(PR_{old}/L) \)
5: end for
6: end function
7: function PageRankVertex::update(r)
8: \( PR_{new} \leftarrow (n_{iter} \% 2)?rank1 : rank0 \)
9: \( PR_{new} \leftarrow d \times r \)
10: \( n_{msg} \leftarrow n_{msg} - 1 \)
11: if \( n_{msg} = 0 \) then
12: \( n_{msg} \leftarrow D_{in} \)
13: \( n_{iter} \leftarrow n_{iter} + 1 \)
14: \( PR_{new} \leftarrow (n_{iter} \% 2)?rank1 : rank0 \)
15: \( PR_{new} \leftarrow (1.0 - d)/N \)
16: end if
17: end function
18: function TestDriver::doPageRank
19: for \( i = 0; i < N_{iters}; i \leftarrow i + 1 \) do
20: \( g.doPageRankStep() \)
21: \( CkStartQD(CkCallbackResumeThread()) \)
22: end for
23: end function
Performance Evaluation

PageRank, Kronecker/Graph500, HPC system: [x2 Xeon E5-2630]/IB FDR

**pagerank (n=20, strong scaling)**

**pagerank (n=22, strong scaling)**

**pagerank (n=16, weak scaling)**

**pagerank (n=18, weak scaling)**
Performance Evaluation
Asynchronous Breadth-first Search (AsyncBFS)

Problem description:
- Find all reachable vertices from root
  \((NB: \text{levels are not detected})\)

Implementations:
- Charm++, naive
- TRAM, naive
- uChareLib, naive
- uChareLib, radix

NB: update function have calls to other chares ⇒ nested calls in TRAM and uChareLib can lead to stack overflow
Performance Evaluation
Asynchronous Breadth-first Search, naive

Algorithm Async BFS

1: function BFSVertex::Update
2:     if visited ≠ true then
3:         visited ← true
4:     for u ∈ AdjList do
5:         thisProxy[u].update()
6:     end for
7: end if
8: end function
Performance Evaluation
Asynchronous Breadth-first Search, radix

Algorithm Async BFS /w Radix

1: function BFSVertex::Update(r)
2:     if state = White then
3:         if r > 0 then
4:             state ← Black
5:         for u ∈ AdjList do
6:             thisProxy[u].update(r - 1)
7:         end for
8:     else
9:         state ← Gray
10:     end if
11: end function
12: function BFSVertex::Resume(r)
13:     if state = Gray then
14:         state ← Black
15:         for u ∈ AdjList do
16:             thisProxy[u].update(r - 1)
17:         end for
18:     end if
19: end function
Performance Evaluation
Asynchronous Breadth-first Search, Kronecker/Graph500, HPC system: [x2 Xeon E5-2630]/IB FDR

bfs (n=20, strong scaling)

bfs (n=22, strong scaling)

bfs (n=16, weak scaling)
Performance Evaluation
Single Source Shortest Path (SSSP)

- Problem description:
  - Find minimum paths from root to other vertices
- Implementations (all are based on Bellman-Ford algorithm):
  - Charm++: naive
  - TRAM: naive
  - TRAM: naive, radix
  - uChareLib: naive, radix

NB: update function have calls to other chares ⇒ nested calls in TRAM and uChareLib can lead to stack overflow
Performance Evaluation
Single Source Shortest Path (SSSP)

Algorithm  Naive SSSP

1: function SSSPVertex::make_root
2:     weight ← 0
3:     parent ← thisIndex
4:   for e ∈ AdjList do
5:       thisProxy[e.u].
6:       update(thisIndex, w + e.w)
7:   end for
8: function SSSPVertex::update(v, w)
9:   if w < weight then
10:     parent ← v
11:     weight ← w
12:   for e ∈ AdjList do
13:     thisProxy[e.u].
14:     update(thisIndex, w + e.w)
15:   end for
16: end if
17: end function
Performance Evaluation

Single Source Shortest Path (SSSP), radix (for TRAM)

**Algorithm**  Radix SSSP

```
1: function SSSPVertex::update(v, w, r)
2:     if w < weight then
3:         parent ← v
4:         weight ← w
5:     for e ∈ AdjList do
6:         if r > 0 then
7:             localAggregator.insertData(dtype(thisIndex, w + e.w, r - 1), e.u)
8:         else
9:             thisProxy[e.u].update(thisIndex, w + e.w, r - 1)
10:     end if
11: end for
12: end if
13: end function
```
Performance Evaluation
Single Source Shortest Path (SSSP), radix

Algorithm Radix SSSP

1: function SSSPVertex::Update(v,w,r)
2:   if w < weight then
3:     parent ← v
4:     weight ← w
5:   if r > 0 then
6:     for e ∈ AdjList do
7:       thisProxy[e.u].update(thisIndex, w + e.w, r - 1)
8:   end for
9: else
10:   state ← Gray
11:   driverProxy.doResume()
12: end if
13: end if
14: end function
15: function SSSPVertex::Resume(r)
16:   if state = Gray then
17:     state ← White
18:     for u ∈ AdjList do
19:       thisProxy[u].update(r - 1)
20:   end for
21: end if
22: end function

NB: same approach as for Asynchronous BFS
Performance Evaluation

Single Source Shortest Path (SSSP), Kronecker/Graph500, HPC system: [x2 Xeon E5-2630]/IB FDR

![Graph showing performance evaluation results](image-url)
Performance Evaluation
Connected Components (CC)

- Problem description:
  - Find all connected components in the graph

- Implementations (based on Asynchronous BFS):
  - Charm++: naive
  - TRAM: naive, radix
  - uChareLib: naive, radix

NB: update function have calls to other chares ⇒ nested calls in TRAM and uChareLib can lead to stack overflow

Before CC execution:

After CC execution:
Performance Evaluation
Connected Components (CC), naive algorithm

Algorithm Naive CC

```plaintext
1: function CCVertex::start
2:   for e ∈ AdjList do
3:       thisProxy[e.u].update(C_id)
4:   end for
5: end function

6: function CCVertex::Update(c)
7:   if c < C_id then
8:     C_id ← c
9:     for e ∈ AdjList do
10:        thisProxy[e.u].update(C_id)
11:    end for
12: end if
13: end function
```

start_cc

TestDriver

Quescence Detection

Performance Evaluation
Connected Components (CC), radix (for TRAM and uChareLib)

Algorithm  Radix CC (TRAM)

1: function CCVertex::update(v, w, r)
2:   if w < weight then
3:     parent ← v
4:     weight ← w
5:   for e ∈ AdjList do
6:     if r > 0 then
7:       localAggregator.insertData
        (dtype(thisIndex, w + e.w, r − 1), e.u)
8:     else
9:       thisProxy[e.u].update(thisIndex, w + e.w, r − 1)
10:   end if
11: end for
12: end function

Algorithm  Radix CC

1: function CCVertex::Update(c, r)
2:   if c < Cid then
3:     Cid ← c
4:     if r > 0 then
5:       for e ∈ AdjList do
6:         thisProxy[e.u].update(Cid, r − 1)
7:       end for
8:     else
9:       state ← Gray
10:      driverProxy.doResume()
11:   end if
12: end if
13: end function
14: function CCVertex::Resume(r)
15:   if state = Gray then
16:     state ← White
17:   for u ∈ AdjList do
18:     thisProxy[u].update(r − 1)
19:   end for
20: end if
21: end function
Performance Evaluation
Contected Components (CC), Kronecker/Graph500, HPC system: [x2 Xeon E5-2630]/IB FDR

cc (n=20, strong scaling)

cc (n=22, strong scaling)

cc (n=14, weak scaling)

cc (n=16, weak scaling)
Limitations of uChareLib

- only single distribution mechanism is available (block [1D] distribution);
- PUPer class is not supported (but message can be used with custom pack/unpack/size methods);
- currently only one uchare array can be created;
- it is not clear how to implement uchare migration/checkpoint.
Conclusion & Future Plans

• uChareLib, an extension of Charm++ to increase performance of highly fine-grained applications, is proposed.

• uChareLib allows to use Vertex-centric approach for development of parallel graph applications in Charm++.

• A set of benchmarks for estimating performance of uChareLib as well as Charm++ and TRAM has been developed.

• Performance evaluation showed that uChareLib has significant performance improvement over Charm++ and TRAM when the number of shares per PE is large, in other cases its performance is close to Charm++.

• Directions for future work:
  (1) comparing of Charm++ tools (pure, TRAM, uChareLib) with other runtimes (AM++, HPX, and Grappa) on developed benchmarks;
  (2) designing more complex graph applications (MST search, community detection, betweenness centrality etc.);
  (3) supporting more features of Charm++ in uChareLib (distributions, PUPer, etc.)
  (4) adding new features to uChareLib (for example, dynamic domain synchronization/collectives, Charm++ RTS integration).
  (5) development/porting of domain-specific language for graph applications adapted to Charm++/uChareLib programming model.

This work is partially supported by Russian Foundation for Basic Research (RFBR) under Contract 15-07-09368.
Thank you! Questions?