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 ($\gg$ #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:

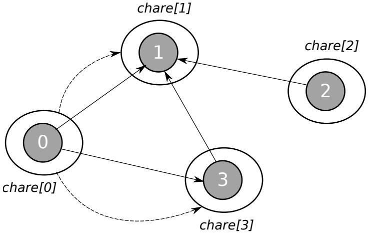
- PDE solvers (fixed structured meshes)
- Rendering
- etc.

Common HPC practice: due to performance reasons to coarsen granularity by aggregating objects/messages and increasing utilization of system resources

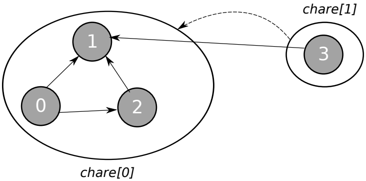
Approaches to Large-scale Graph Processing on Charm++

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

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

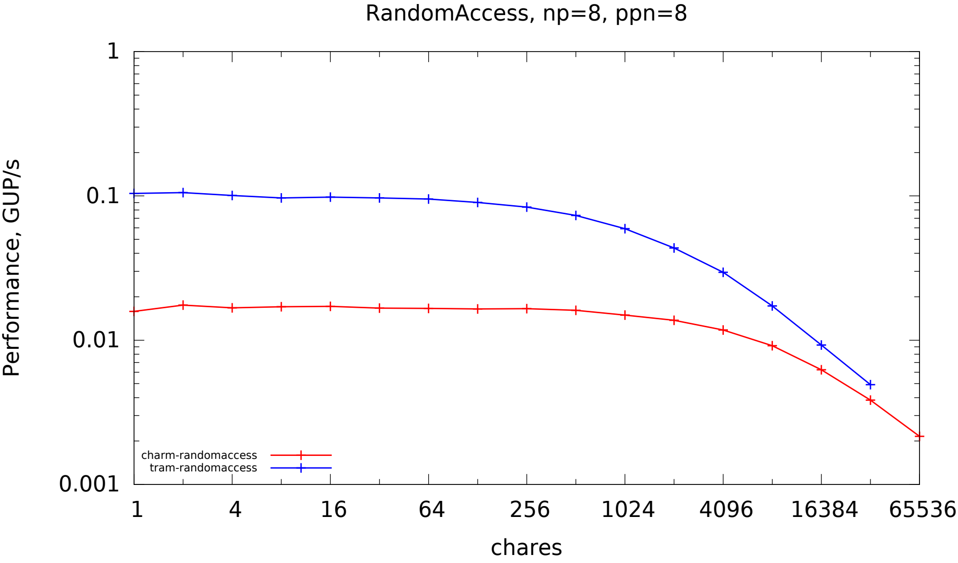


- Subgraph-centric
  - Graph (G) – array of chares distributed between parallel processes (PE).
  - Vertex – chare (n:1), any local representation possible
  - Algorithms consist of local (sequential) and global parts (parallel, Charm++).
  - Application level optimizations (aggregation, local reductions,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



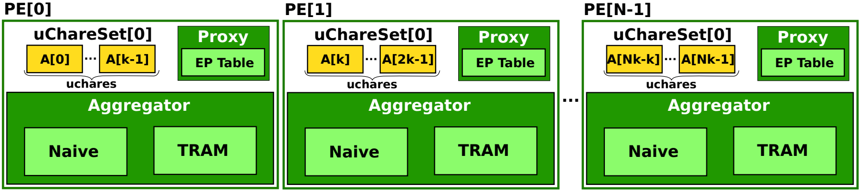
HPCC RandomAccess

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



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):

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

Charm++ (alltoall.C):

```
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):

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

uChareLib (alltoall.C):

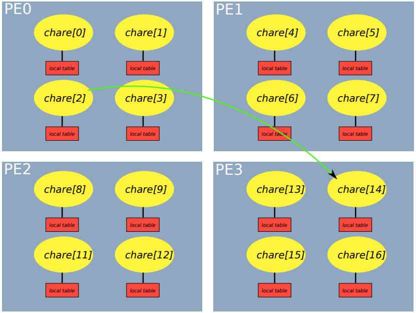
```
class Alltoall : public CBase_uChare_Alltoall {
 private:
  CmiUInt8 counter;
 public:
  Alltoall(const uChareAttr_Alltoall & attr) :
    counter(1), CBase_uChare_Alltoall(attr) {
   contribute(CkCallback(CkReductionTarget(
      TestDriver, init), driverProxy));
  }
  /*entry*/ void start() {
   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
Index ← Pseudo random indices
for all $i \in Index$ do
    $Table[i] \leftarrow Table[i] \oplus i$
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

Algorithm RandomAccess

```
$Index \leftarrow$ Pseudo random indices
for all $i \in Index$ do
    $Table[i] \leftarrow Table[i] \oplus i$
end for
```

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

Charm++ & uChareLib (randomAccess.C):

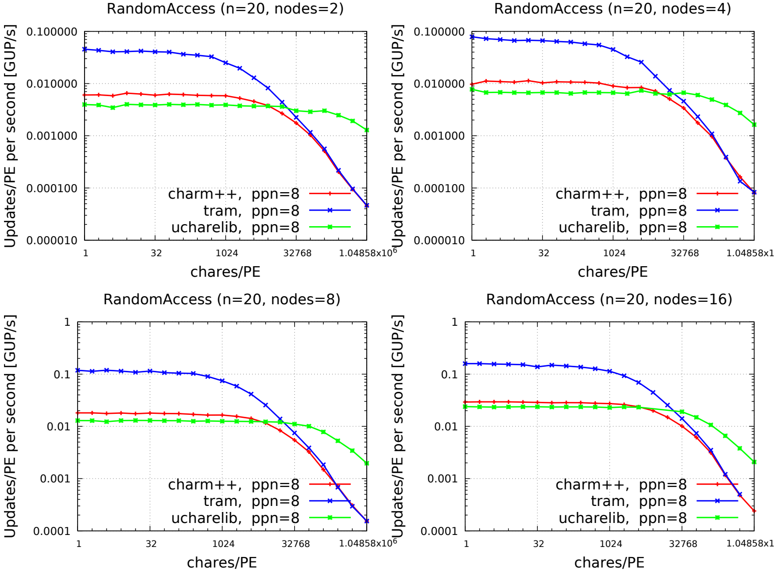
```
void Updater::generateUpdates() {
 int arrayN = N - (int) log2((double) numElementsPerPe
 int numElements = CkNumPes() * numElementsPerPe;
 CmiUInt8 key = HPCC_starts(4 * globalStartmyProc);
 for(CmiInt8 i = 0; i < 4 * localTableSize; i++) {
  key = key << 1 ^ ((CmiInt8) key < 0 ? POLY : 0);
  int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

TRAM (randomAccess.C):

```
void Updater::generateUpdates() {
...
 ArrayMeshStreamer<dtype, int, Updater, SimpleMeshRout
  * localAggregator = aggregator.ckLocalBranch();
 for(CmiInt8 i = 0; i < 4 * localTableSize; i++) {
  key = key << 1 ^ ((CmiInt8) key < 0 ? POLY : 0);
  int destinationIndex = key >> arrayN & numElements -
  localAggregator->insertData(key, destinationIndex);
 }
 localAggregator->done();
 }
}
```

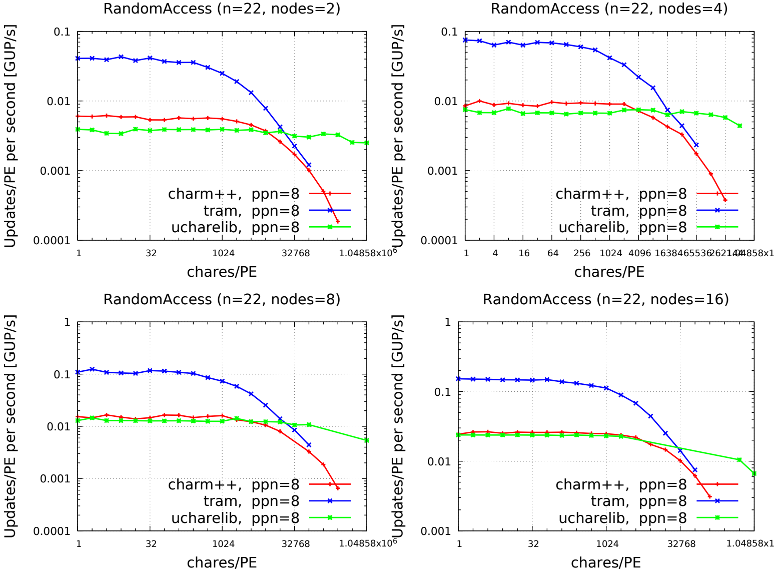
Performance Evaluation

HPCC RandomAccess (N=2^{20}/PE), HPC system: [x2 Xeon E5-2630]/IB FDR



Performance Evaluation

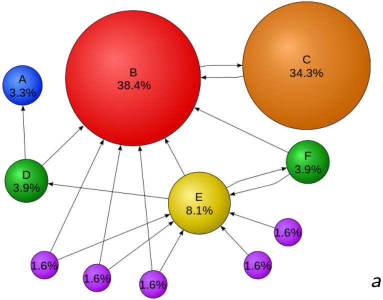
HPCC RandomAccess (N=2^{22}/PE), HPC system: [x2 Xeon E5-2630]/IB FDR



Performance Evaluation

PageRank

- Problem description:
  - Iteratively compute ranks for all $v \in G$
    $PR_v^{i+1} = (1-d)/N + d \times \sum_{u \in Adj(v)} PR_u^i / L_u$
- Implementations:
  - Charm++, naive
  - Charm++, with incoming msg counting
  - TRAM, naive
  - uChareLib, naive



[a] source: Wikipedia

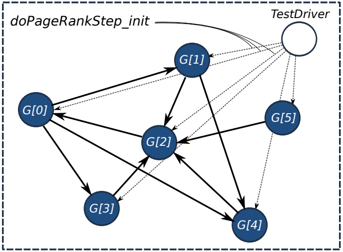
NB: `update` function does not contain calls to other chares => no nested calls (insertData/entry method) for TRAM and uChareLib

Performance Evaluation

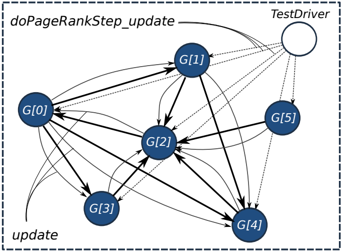
PageRank, naive algorithm

Algorithm Naive PageRank

```
function PageRankVertex:: doPageRankStep_init
    $PR_{old} \leftarrow PR_{new}$
    $PR_{new} \leftarrow (1.0 - d)/N$
end function
function PageRankVertex::doPageRankStep_update
    for $u \in AdjList$ do
        $thisProxy[u].update(PR_{old}/L)$
    end for
end function
function PageRankVertex::update(r)
    $PR_{new} \leftarrow d \times r$
end function
function TestDriver::doPageRank
    for $i = 0; i < N_{iters}; i \leftarrow i + 1$ do
        $g.doPageRankStep\_init()$
        $CkStartQD(CkCallbackResumeThread())$
        $g.doPageRankStep\_update()$
        $CkStartQD(CkCallbackResumeThread())$
    end for
end function
```



Quiescence Detection



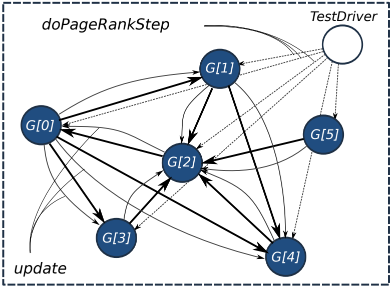
Quiescence Detection

Performance Evaluation

PageRank, with counting incoming messages

Algorithm PageRank /w msg counting

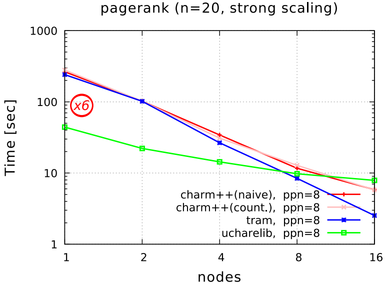
```
function PageRankVertex::doPageRankStep
    $PR_{old} \mapsfrom (n_{iter}\%2)?rank0 : rank1$
    for $u \in AdjList$ do
        $thisProxy[u].update(PR_{old}/L)$
    end for
end function
function PageRankVertex::update(r)
    $PR_{new} \mapsfrom (n_{iter}\%2)?rank1 : rank0$
    $PR_{new} \leftarrow d \times r$
    $n_{msg} \leftarrow n_{msg} - 1$
    if $n_{msg} = 0$ then
        $n_{msg} \leftarrow D_{in}$
        $n_{iter} \leftarrow n_{iter} + 1$
        $PR_{new} \mapsfrom (n_{iter}\%2)?rank1 : rank0$
        $PR_{new} \leftarrow (1.0 - d)/N$
    end if
end function
function TestDriver::doPageRank
    for $i = 0; i < N_{iters}; i \leftarrow i + 1$ do
        $g.doPageRankStep()$
        $CkStartQD(CkCallbackResumeThread())$
    end for
end function
```

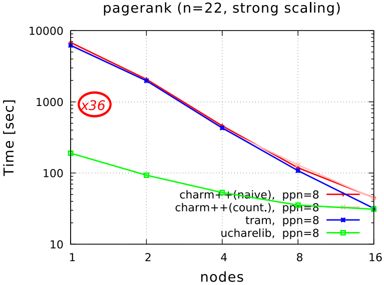


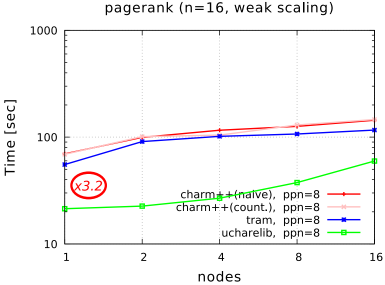
Quiescence Detection

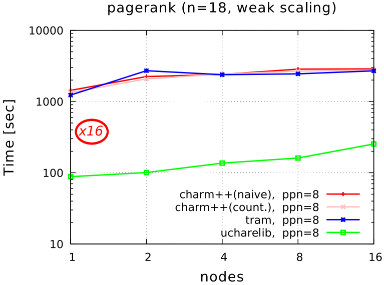
Performance Evaluation

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









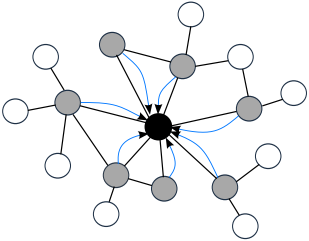
Performance Evaluation

Asynchronous Breadth-first Search (AsyncBFS)

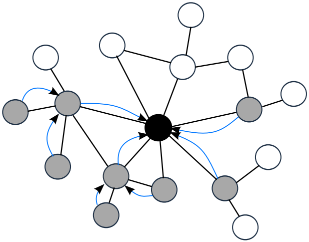
- Problem description:
  - Find all reachable vertices from root (*NB:* levels are not detected)
- Implementations:
  - Charm++, naive
  - TRAM, naive
  - uChareLib, naive
  - uChareLib, radix

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

Level-synchronous BFS:



Asynchronous BFS:

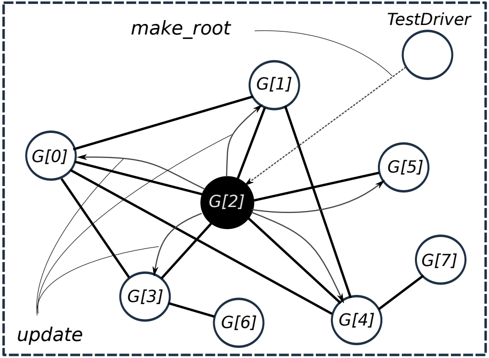


Performance Evaluation

Asynchronous Breadth-first Search, naive

Algorithm Async BFS

```
function BFSVertex::Update
    if $visited \neq true$ then
        $visited \leftarrow true$
        for $u \in AdjList$ do
            $thisProxy[u].update()$
        end for
    end if
end function
```



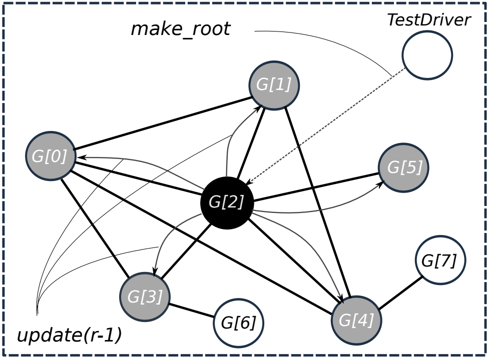
Quiescence Detection

Performance Evaluation

Asynchronous Breadth-first Search, radix

Algorithm Async BFS /w Radix

```
function BFSVertex::Update(r)
    if $state$ = $White$ then
        if $r > 0$ then
            $state \leftarrow Black$
            for $u \in AdjList$ do
                $thisProxy[u].update(r-1)$
            end for
        else
            $state \leftarrow Gray$
        end if
    end if
end function
function BFSVertex::Resume(r)
    if $state$ = $Gray$ then
        $state \leftarrow Black$
        for $u \in AdjList$ do
            $thisProxy[u].update(r-1)$
        end for
    end if
end function
```

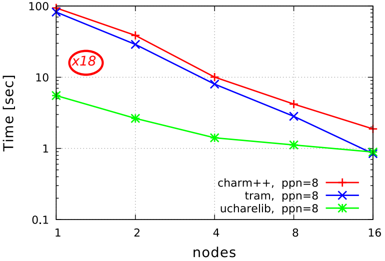


Quiescence Detection

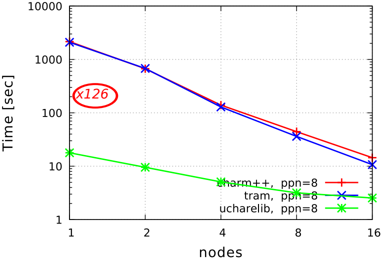
Performance Evaluation

Asynchronous Breadth-first Search, Kronecker/Graph500, HPC system: [x2 Xeon E5-2630]/IB FDR

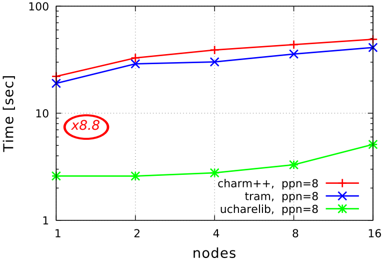
bfs (n=20, strong scaling)



bfs (n=22, strong scaling)



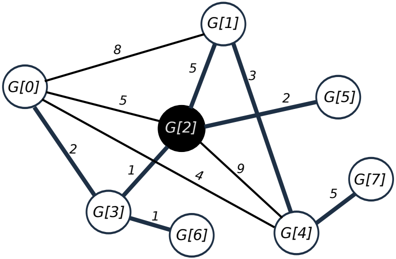
bfs (n=16, weak scaling)



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



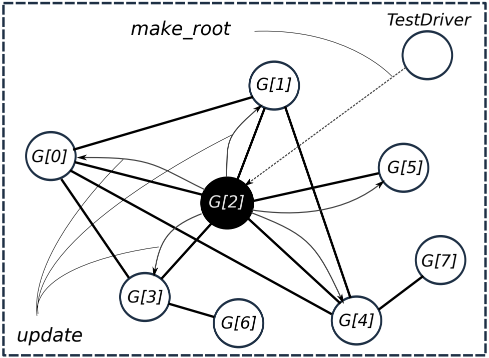
NB: `update` function have calls to other chares ⇒ nested calls in TRAM and uChareLib can lead to stack overflow

Performance Evaluation

Single Source Shortest Path (SSSP)

Algorithm Naive SSSP

```
function SSSPVertex::make_root
    $weight \leftarrow 0$
    $parent \leftarrow thisIndex$
    for $e \in AdjList$ do
        $thisProxy[e.u].$
        $update(thisIndex, w + e.w)$
    end for
end function
function SSSPVertex::update(v, w)
    if $w < weight$ then
        $parent \leftarrow v$
        $weight \leftarrow w$
        for $e \in AdjList$ do
            $thisProxy[e.u].$
            $update(thisIndex, w + e.w)$
        end for
    end if
end function
```



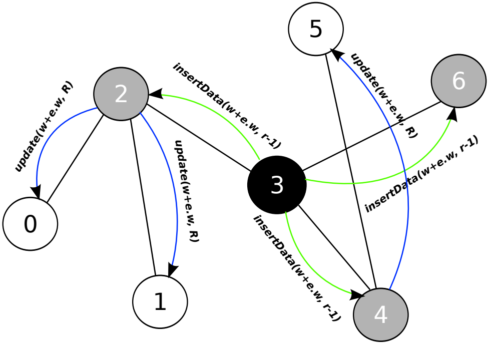
Quescence Detection

Performance Evaluation

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

Algorithm Radix SSSP

```
function SSSPVertex::update(v, w, r)
    if $w < weight$ then
        $parent \leftarrow v$
        $weight \leftarrow w$
        for $e \in AdjList$ do
            if $r > 0$ then
                $localAggregator.insertData(dtype(thisIndex, w + e.w, r - 1), e.u)$
            else
                $thisProxy[e.u].update(thisIndex, w + e.w, r - 1)$
            end if
        end for
    end if
end function
```

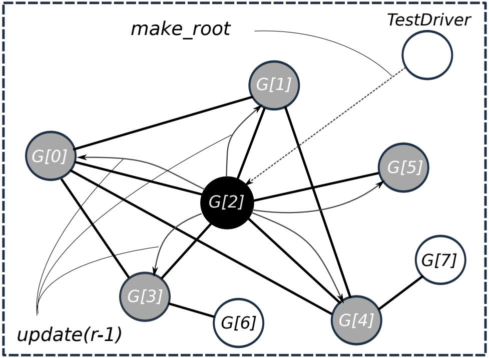


Performance Evaluation

Single Source Shortest Path (SSSP), radix

Algorithm Radix SSSP

```
function SSSPVertex::Update(v,w,r)
    if $w < weight$ then
        $parent \leftarrow v$
        $weight \leftarrow w$
        if $r > 0$ then
            for $e \in AdjList$ do
                $thisProxy[e.u].update(thisIndex, w + e.w, r - 1)$
            end for
        else
            $state \leftarrow Gray$
            $driverProxy.doResume()$
        end if
    end if
end function
function SSSPVertex::Resume(r)
    if $state = Gray$ then
        $state \leftarrow White$
        for $u \in AdjList$ do
            $thisProxy[u].update(r - 1)$
        end for
    end if
end function
```

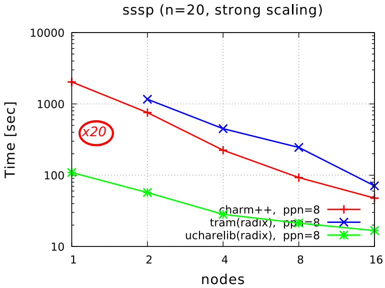


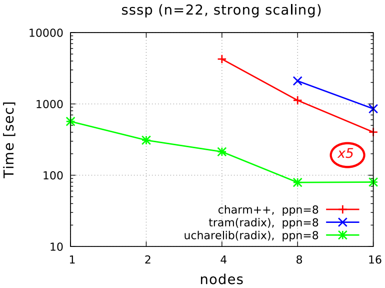
Quiescence Detection

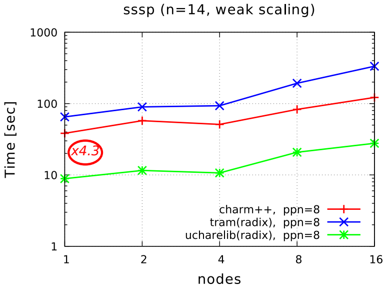
NB: same approach as for Asynchronous BFS

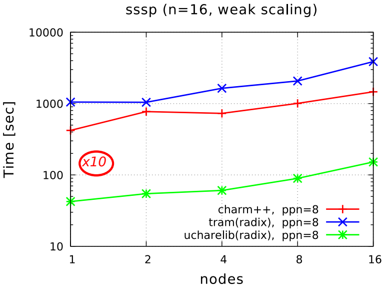
Performance Evaluation

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









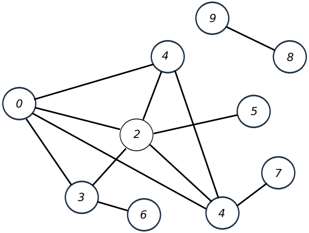
Performance Evaluation

Contected Components (CC)

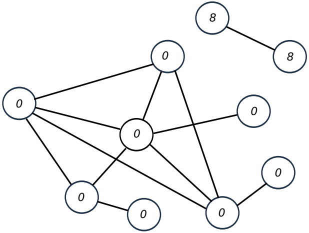
- Problem description:
  - Find all connected components in the graph
- Implementations (based on Asynchronous BFS):
  - Charm++: naive
  - TRAM: naive, radix
  - uChareLib: naive, radix

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

Before CC execution:



After CC exectution:

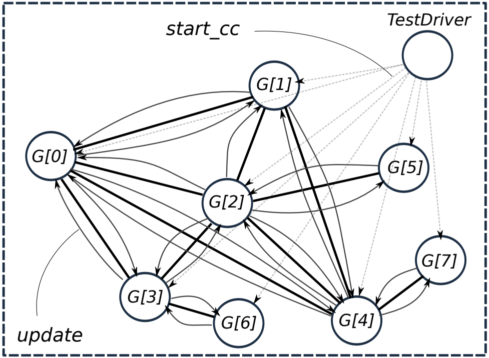


Performance Evaluation

Connected Components (CC), naive algorithm

Algorithm Naive CC

```
function CCVertex::start
    for $e \in AdjList$ do
        $thisProxy[e.u].update(C_{id})$
    end for
end function
function CCVertex::Update(c)
    if $c < C_{id}$ then
        $C_{id} \leftarrow c$
        for $e \in AdjList$ do
            $thisProxy[e.u].update(C_{id})$
        end for
    end if
end function
```



Quescence Detection

Performance Evaluation

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

Algorithm Radix CC (TRAM)

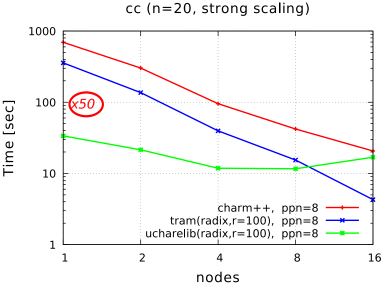
```
function CCVertex::update(v, w, r)
    if $w < weight$ then
        $parent \leftarrow v$
        $weight \leftarrow w$
        for $e \in AdjList$ do
            if $r > 0$ then
                $localAggregator.insertData(dtype(thisIndex, w + e.w, r - 1), e.u)$
            else
                $thisProxy[e.u].update(thisIndex, w + e.w, r - 1)$
            end if
        end for
    end if
end function
```

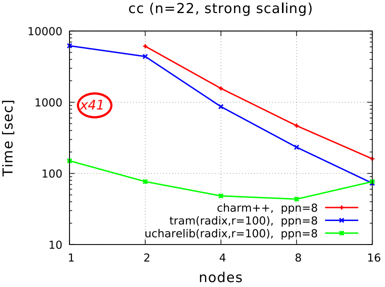
Algorithm Radix CC

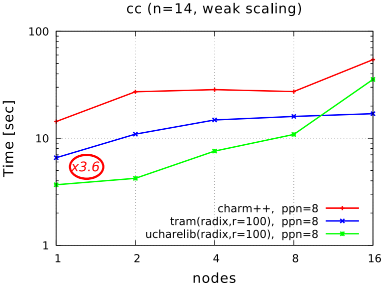
```
function CCVertex::Update(c, r)
    if $c < C_{id}$ then
        $C_{id} \leftarrow c$
        if $r > 0$ then
            for $e \in AdjList$ do
                $thisProxy[e.u].update(C_{id}, r - 1)$
            end for
        else
            $state \leftarrow Gray$
            $driverProxy.doResume()$
        end if
    end if
end function
function CCVertex::Resume(r)
    if $state = Gray$ then
        $state \leftarrow White$
        for $u \in AdjList$ do
            $thisProxy[u].update(r - 1)$
        end for
    end if
end function
```

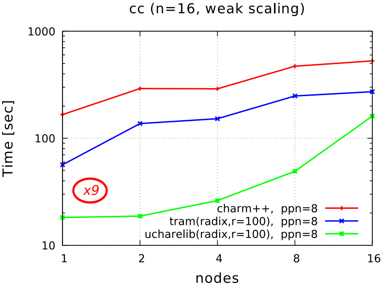
Performance Evaluation

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 chares 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 domen 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?