Parallelizing Irregular Applications through the YAPPA Compilation Framework

Silvia Lovergine, Antonino Tumeo, Oreste Villa, Fabrizio Ferrandi

Motivation

- Modern High Performance Computing (HPC) clusters composed of hundred of nodes integrating multicore processors with advanced cache hierarchies.
- They reach several petaflops of peak performance, but they are optimized for floating point intensive applications, and regular, localizable data structures.
- Network interconnection optimized for bulk, synchronous transfers.
- Many scientific applications are irregular:
  - Dynamic, linked data structures (e.g., graphs, unbalanced trees, unstructured grids), irregular control flow, unpredictable and fine-grained communication
  - Inherently parallel, but difficult to partition in a balanced way
  - Almost no locality
- High synchronization intensity

Complex design challenges and significant programming effort

Our approach

YAPPA (Yet Another Parallel Programming Approach): a compilation framework for the **automatic parallelization** of **irregular applications** on modern HPC systems, based on **LLVM**

- YAPPA builds on the top of GMT (Global Memory and Threading library): a runtime library for irregular applications on distributed memory HPC systems
- YAPPA analyzes a C/C++ application to automatically produce its parallel version. YAPPA requires the programmer to manage synchronization by means of atomic operations

GMT: run-time library for Irregular applications on distributed memory HPC systems

- Built around three main concepts:PGAS data model, latency tolerance through fine-grained software multithreading, and aggregation.
  - Partitioned Global Address Space (PGAS) removes need to partition datasets
  - Fine grained software multithreading tolerates network latencies
  - Data aggregation reduces overheads of fine grained network accesses
  - Fork/join control model with simple parallel for constructs, eases application development

Example: Breadth First Search (BFS) Algorithm

Sequential version with synchronization

```
1:  while (QN ! = 0) do
2:    for (vId = 0; vId < QN ; vId + +) do
3:      uint64_t vertex = gQ[vId]
4:      uint64_t curIdx = gIdxs[vertex]
5:      uint64_t nextIdx = gIdxs[vertex + 1]
6:      for ( uint64_t i = curIdx; i < nextIdx; i + +) do
7:        uint64_t neighbor = gEdges[i];
8:        if ( gmt_atomicCAS(gM arked, neighbor *
              sizeof (unit64_t), 0, 1, sizeof (uint64_t)) ) then
9:          gM arked[neighbor] = 1;
10:         uint64_t Qvalue = gmt_atomicAdd(gQnextN , 0, 1, sizeof (uint64_t));
11:         gQnext[Qvalue] = neighbor;
12:       end if
13:     end for
14:   end for
15:   gQ=gQnext
16:   QN = gQnextN
17:   gQnext = 0
18: end while
```

Arguments struct for the loop body function

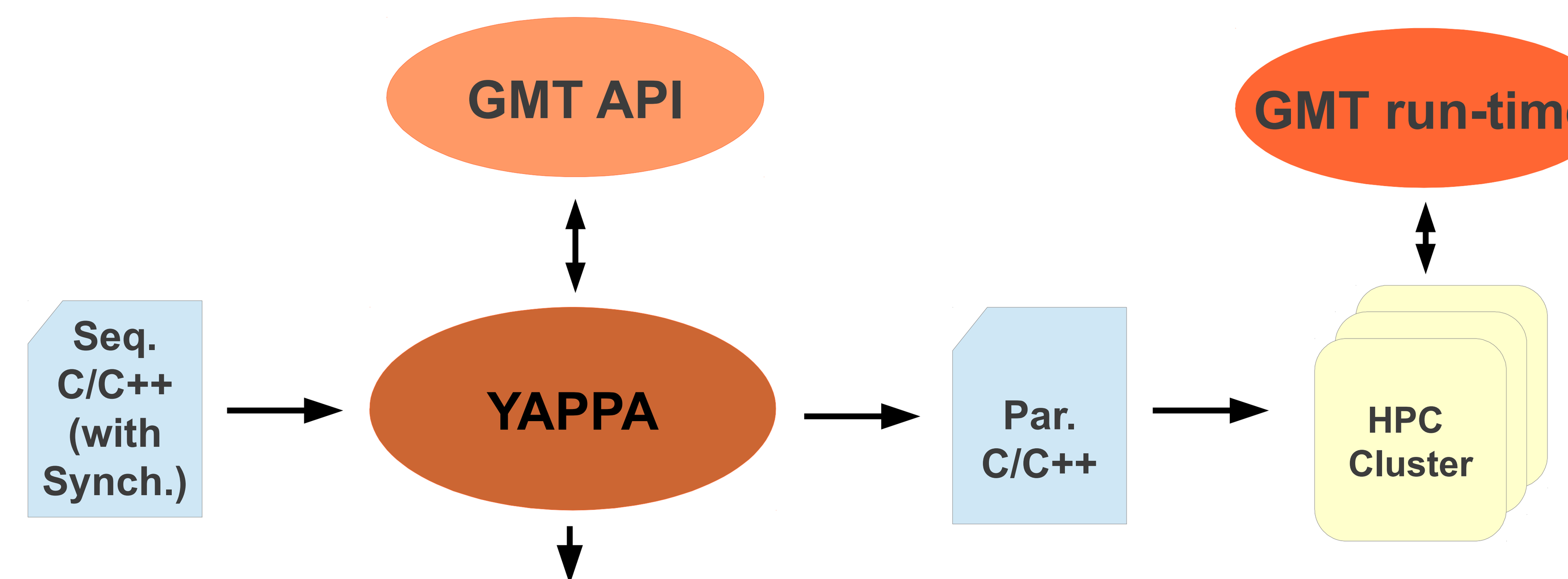
```
1: typedef struct Args_t {
2: gmt_data_t gM arked;
3: gmt_data_t gIdx;
4: gmt_data_t gEdges;
5: gmt_data_t gQ
6: gmt_data_t gQnext;
7: gmt_data_t gQnextN;
8: } args_t;
```

Parallel loop body function unoptimized

```
1:  void F (uint64_t iterId, void * Args) {
2:    args_t* args = (args_t*)Args;
3:    uint64_t vertex, curIdx, nextIdx;
4:    gmt_get(args→gQ, iterid * sizeof (uint64t ), &vertex, sizeof (uint64_t));
5:    gmt_get(args→gIdxs, iterid * sizeof (uint64t ), &curIdx, sizeof (uint64_t));
6:    gmt_get(args→gIdxs, (iterid+1) * sizeof (uint64t ), &nextIdx, sizeof (uint64_t));
7:    for ( uint64_t i = curIdx; i < nextIdx; i + +) do
8:      gmt_get(args→gEdges, i, neighbor, sizeof (uint64_t));
9:      if ( gmt_atomicCAS(args→gM arked, neighbor *
            sizeof (unit64_t), 0, 1, sizeof (uint64_t)) ) then
10:       gMarked[neighbor] = 1;
11:       uint64_t Qvalue = gmt_atomicAdd(args→QnextN , 0, 1, sizeof (uint64_t));
12:       gmt_put(args→Qnext, Qvalue*sizeof (uint64_t), &neighbor, sizeof (uint64_t));
13:     end if
14:   end for
15: }
```

Parallel main function unoptimized

```
1:  uint64t ∗ gIdxs = gmt_alloc(vertex + 1 ∗ sizeof (uint64_t);
2:  ...
3:  while (QN ! = 0) do
4:    args_t args;
5:    args.gM arked = gM arked;
6:    args.gIdx = gEdges;
7:    args.gIdxs = gIdxs;
8:    args.gQt = gQ;
9:    args.gQnext = gQnext;
10:   args.gQnextN = gQnextN ;
11:   uint64_t nT hr = QN /chunkSize;
12:   if (nThr * chunkSize < QN ) then
13:     nT hr + +;
14:   end if
15:   gmt_parFor(nT hr, chunkSize, F, &args, sizeof (args));
16:   gQnext = gQ;
17:   get(gQnextN , 0, &QN , sizeof (uint64_t));
18:   put(gQnextN , 0, 0, sizeof (uint64_t));
19: end while
```



Transformations

Data management:
1. Shared data identification
2. Allocation of global data structures
3. Transformation of the access to global data in global memory accesses

Loop Parallelization:
1. Parallel loop bodies extraction to generates task functions
2. Alias analysis for arguments identification
3. Variables localization

Optimizations

Unblocking of memory accesses:
Move at the beginning of the loop as many independent memory operations as possible, substituting blocking memory operation primitives with their equivalent non-blocking versions

Block-Hoisting:
If the loop only reads scalar values from the array, sequentially, one iteration after the other, and the loop is not parallelized, global get only generates a sequence of very fine grained operations inside the same task. In such cases, YAPPA hoists the scalar read operations from the loop and aggregates them in a single get operation, writing to an entire local subarray.

Parallel loop body function: Block-Hoisting

```
1:  void F (uint64_t iterId, void * Args){
2:    args_t* args = (args_t*)Args;
3:    uint64_t vertex, curIdx, nextIdx;
4:    gmt_get(args→gQ, iterid * sizeof (uint64t ), &vertex,
              sizeof (uint64_t));
5:    gmt_get(args→gIdxs, iterid * sizeof (uint64t ), &curIdx,
              sizeof (uint64_t));
6:    gmt_get(args→gIdxs, (iterid+1) * sizeof (uint64t ),
              &nextIdx, sizeof (uint64_t));
7:    uint64_t *neighbors = malloc((nextIdx − curIdx) *
                                 sizeof(uint64_t));
8:    gmt_get(args→gEdges, cur[0] * sizeof (uint64_t),
               neighbors, (nextIdx − curIdx) ∗
               sizeof(uint64_t));
9:    for ( uint64_t i = curIdx; i < nextIdx; i + +) do
10:     if (gmt_atomicCAS(args→gM arked, neighbors[i] *
            sizeof (unit64_t), 0, 1, sizeof (uint64_t))) then
11:       gM arked[neighbors[i]] = 1;
12:       uint64_t Qvalue = gmt_atomicAdd(args→QnextN ,
                                0, 1, sizeof (uint64_t));
13:       gmt_put(args→Qnext, Qvalue∗sizeof (uint64_t),
                  &neighbors[i], sizeof (uint64_t));
14:     end if
15:   end for
16: }
```

Parallel loop body function: unblocking of memory accesses

```
1:  void F (uint64_t iterId, void * Args) {
2:    args_t* args = (args_t*)Args;
3:    uint64_t vertex, curIdx, nextIdx;
4:    gmt_get(args→gQ, iterid * sizeof (uint64t ), &vertex,
              sizeof (uint64_t));
5:    gmt_get(args→gIdxs, iterid * sizeof (uint64t ), &curIdx,
              sizeof (uint64_t));
6:    gmt_get(args→gIdxs, (iterid+1)*sizeof (uint64t ),
              &nextIdx, sizeof (uint64_t));
7:    for ( uint64_t i = curIdx; i < nextIdx; i + +) do
8:      gmt_get(args→gEdges, i, neighbor, sizeof (uint64_t));
9:      if ( gmt_atomicCAS(args→gM arked, neighbor *
            sizeof(unit64_t), 0, 1, sizeof (uint64_t)) ) then
10:       gMarked[neighbor] = 1;
11:       uint64_t Qvalue = gmt_atomicAdd(args→QnextN , 0,
                                           1, sizeof (uint64_t));
12:       gmt_putNB(args→Qnext, Qvalue∗sizeof(uint64_t),
                    &neighbor, sizeof (uint64_t));
13:     end if
14:   end for
15: }
```

Experimental Evaluation

#V-#E	Serial	GMT	Sp. GMT	GMT-BH	Sp. GMT-BH
1,000,000-100	1.35	0.52	2.59	0.49	1.06
1,000,000-1,000	9.83	2.53	3.88	2.39	1.06
1,000,000-10,000	95.86	22.72	4.23	21.27	1.07

Silvia Lovergine, Fabrizio Ferrandi
Politecnico di Milano, DEIB
lovergine@elet.polimi.it ferrandi@elet.polimi.it

Silvia Lovergine, Antonino Tumeo
Pacific Northwest National Laboratory, HPC
silvia.lovergine@pnnl.gov antonino.tumeo@pnnl.gov

Oreste Villa
NVIDIA
ovilla@nvidia.com