Performance Portability via Object Mesh

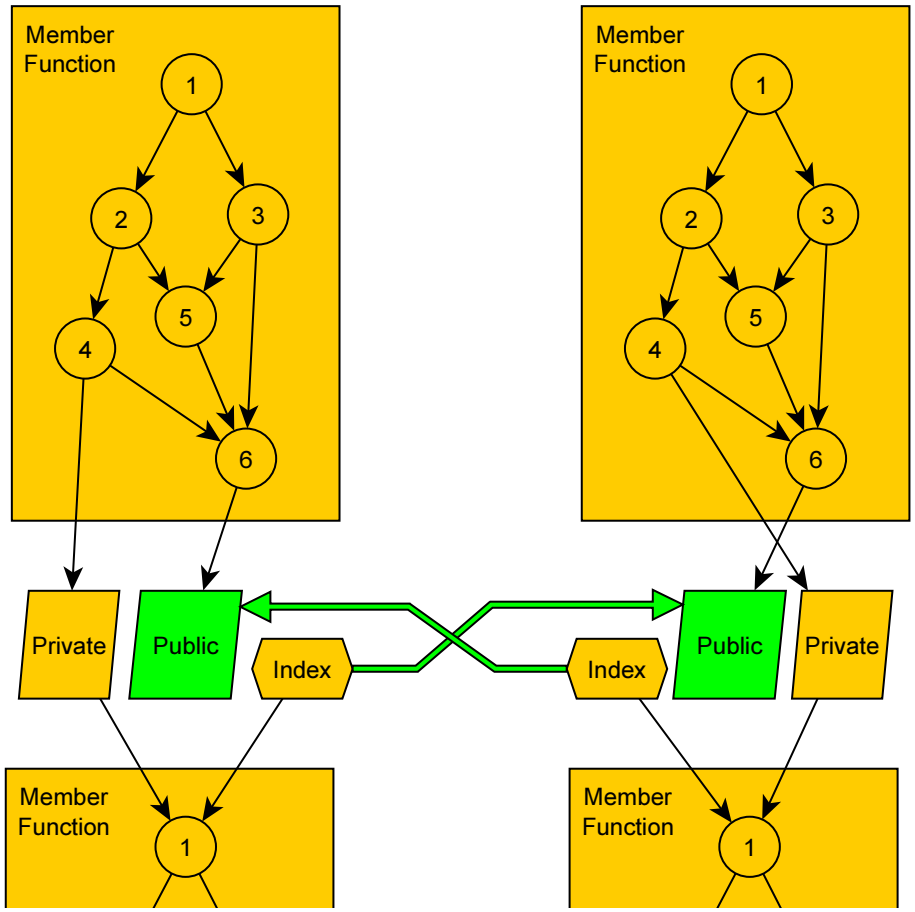
Changhoan Kim, James Kozloski
IBM

Performance Portability

- Key concerns for high performance
  - Parallelism: hardware resources(e.g. FPU), synchronization, load balancing
  - Locality: Binding data and function (mem hierarchy, SIMD)
- Why hard for compiler?
  - code is hard to analyze ➔ new code
  - best mapping/scheduling is hard to find ➔ help from domain experts
- New programming framework
  - Easily analyzable code: Acode ← easily analyzable to human as well
    - Borrow OO concept: encapsulation(**data+function**), attributed variables, inheritance(?)
    - connectivity, parallelism(parallel_for, reduce..), workflow
  - Programmability of automatic compiler+runtime scheduling: Scode
    - partitioning/mapping dependency graph ← portability from *quick/automatic* adaptation of Scode

Acode concept

- class description ← ‘fiber’
  - private variable : internal state
  - public variable : communicate with other instances(producer-consumer)
  - neighbor list : explicit connectivity between instances
  - member function: bound to member variables
- Parallelism ➔ array/vector of ‘fibers’
- Connectivity description
  - setting the neighbor list
  - ‘fabric’ specification: woven by neighbor list
- Workflow description
- Dependency graph is factorized explicitly
  - {fiber dependency graph} × {mesh}
- Specify algorithm in an architecture independent way



Acode concept

explicit list of neighbor

state

logically single var

```
class Node {

 neighbor:
   index* nb_list;

 public:
   double phi;

 private:
   double pv_phi,tmp2,tmp3;

 shared:

 void update() { phi = pv_phi; }

 void calc()
 void calc2()
 void calc3()
}
```

updating public indicates producing

```
void Node::calc()
{
  pv_phi = 0;
  foreach(index x,nb_list)
  {
    pv_phi += Node[x].phi;
  }
}
```

accessing other instance: consumer

work flow

À la 'Chapel'

parallelism bundle

setting up connectivity

```
input const int Nx;
input const int Ny;

int main()
{
 Node myNode[Nx*Ny];

 set_neighbors(myNode,.....);

 for(int i=0;i<100;i++)
 {
  foreach(myNode.calc2());
  foreach(myNode.calc3());
  foreach(myNode.calc());
  foreach(myNode.update());
 }

 double ans=reduce(plus,myNode.phi);
 cout << "answer=" << ans << endl;
}
```

Scode concept

- help to find Mapping & Scheduling
  - Partition of the factorized dependency graph
    - Factorization {dependency graph of an object} × {object mesh} → horizontal/vertical partitioning
    - conforming to memory hierarchy : 'cutting fabric in proper size' (footprint or # of works)
    - L1 cache, double buffering, heterogeneous system
  - scheduling/synchronization/communication plan: conversion of producer-consumer model
    - node-to-node, cpu-to-gpu, socket-to-socket, core-to-core(data layout)
- Architecture dependent / Application specific
  - Domain knowledge is transferred to the compiler
  - default but slow version is possible
- Portability is achieved by 'close to automatic' adaptation of Scode only
  - Domain specific knowledge is represented by the application specific optimization parameter/algorithm set

Real world example

- IBM neuro simulator
- Acode is composed of two parts
  - ‘MDL’ : fiber description
  - ‘GSL’ : fiber connectivity, execution flow, initial conditions, parallelism
- Single code for shared/distributed memory
- GPU support is on going
- Default scheduling
  - Simple partitioning + MPI AlltoAllv
  - No Scode interface yet

Game of life I

- Shared
  - shared among all LifeNode
  - similar to static variable
- <<
  - explicit producer
- >>
  - explicit consumer
- member functions
  - initialize, update, copy
  - grouped
    - initialize() ∈ InitPhase
    - update,copy ∈ RuntimePhase
  - uses only member data of *this instance
    ➔ easy for compiler to analyze

```
Node LifeNode Implements ValueProducer {
  int value;
  int publicValue;
  int* [] neighbors;

  Shared {
    int tooSparse;
    int tooCrowded;
  }

  InitPhase initialize();
  RuntimePhase update();
  RuntimePhase copy(publicValue);

  ValueProducer.value << &publicValue;

  Connection Pre Node () Expects ValueProducer {
    ValueProducer.value >> neighbors;
  }
}
```

Game of Life II

- declaration of instances
- initial condition
- connectivity
- task flow
- iteration
- stopping condition
- synchronization is explicit

```
#include "../std/std.gsl"
InitPhases = { initialize };
RuntimePhases = { dataCollect, update, copy, lastPhase };
FinalPhases = { finalize };

NodeType LifeNode(< tooSparse=1, tooCrowded=4 >);

Grid World
{
  Dimension(1000,1000);
  Layer(nodes, LifeNode, UniformLayout(1), < nodekind="Nodes" >);
  InitNodes ( .[250:750, 250:750].Layer(nodes), Same( Pset<LifeNode, NodeInit> ( <value = 1> ) ) );
  InitNodes ( .[400:600, 400:600].Layer(nodes), Same( Pset<LifeNode, NodeInit> ( <value = 0> ) ) );
  NodeSet all(.[].Layer(nodes));
  connectNodeSets(all, all, EachDst(RadialSampler(1.5)), outAttrDef, inAttrDef);
};

World world;

// DCA directives here

VariableType LifeDataCollector;
LifeDataCollector collector<fileName="LifeOutput.txt">;

polyConnect(world[].Layer(nodes), collector, <>, <>);

Trigger UnsignedTrigger(string description, Service svc, string operator, int criterion, int delay);
Trigger CompositeTrigger(string description, Trigger triggerA, int critA, string operator, Trigger triggerB, int critB, int
delay);

UnsignedTrigger iterTrig("Iteration Trigger : >= 1 ",
                ::Iteration, ">=", 1, 0, dataCollect);

UnsignedTrigger endTrig("Iteration Trigger to end or stop",
                ::Iteration, "==", 1000, 0, lastPhase);
collector.dataCollection() on iterTrig;
Stop on endTrig;
```

Discussions

- Can it handle more than one fiber classes?
- Dynamic creation of class? → lambda capture?
- Connectivity structure change?
  - it can be detected but what to do
- Ambiguity on synchronization?
  - when to produce if public variable updated many times?
- Relations to other framework?
  - openMP, MPI, CUDA, Legion, PGAS, RAJA, Kokos
  - Agent based modeling