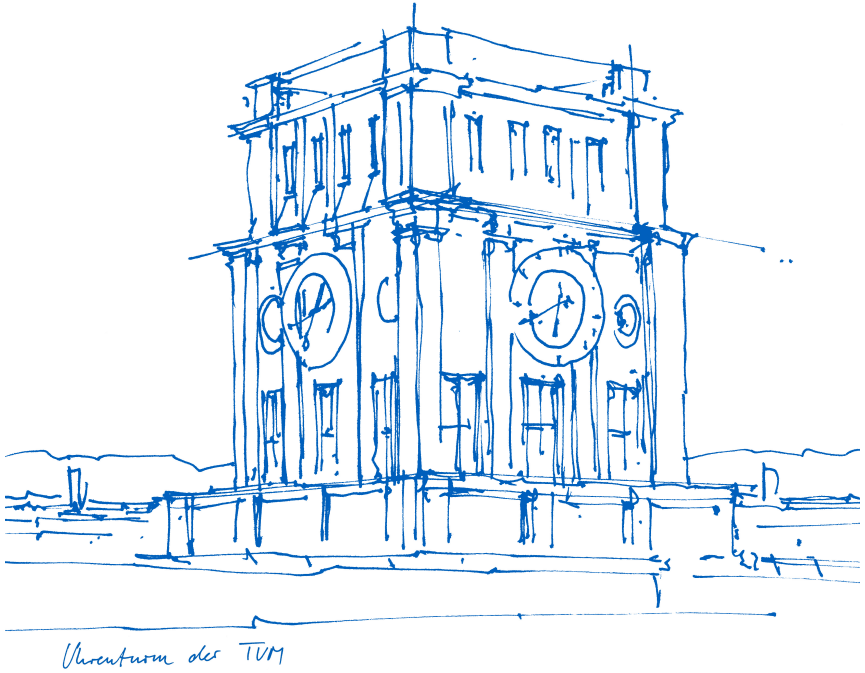
Lecture IN-2147
Parallel Programming

SoSe 2018

Martin Schulz
Exercises: Amir Raoofy
Technische Universität München
Fakultät für Informatik

Lecture 13:
Alternative Parallel Programming Models



Summary From Last Time

Accelerators can help boost per node performance
- Dedicated hardware with specialized features
- Examples: FPGAs, Dataflow engines, GPUs

GPUs have evolved to massively parallel compute engines
- Hierarchical structure
- Inner elements are SIMT

Impact on programmability
- Accelerators rely on a host system
- Specialized language (extension) to cross-compile and control

CUDA targeting NVIDIA GPUs specifically
- Manual specification of data transfers through CUDA runtime
- Manual specification of thread mapping

OpenMP supports accelerators since OpenMP 4.0
- Allows specification of kernels on a device
- Pragmas to create the necessary data environment

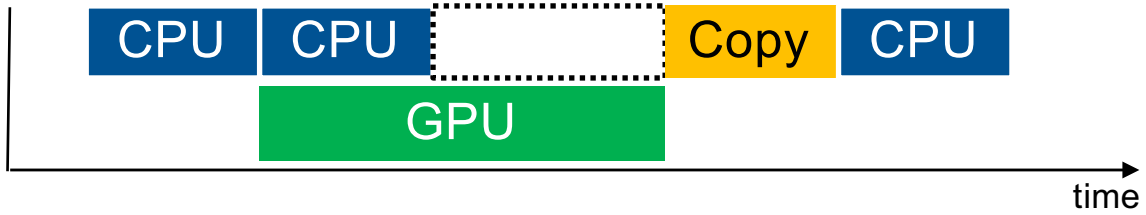
Addendum: Asynchrony in CUDA

```
//Kernel
__global__
void daxpy(int n, double a, double *x, double *y){
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  if (i<n) y[i]=a*x[i]+y[i]
}
```

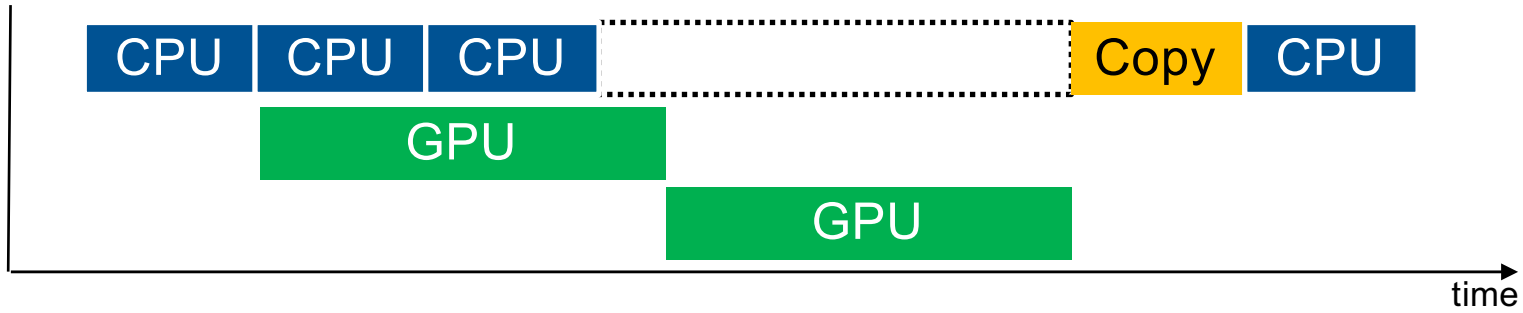
```
//Copy data to device
cudaMalloc(&d_x, nbytes); cudaMalloc(&d_y, nbytes);
cudaMemcpy(d_x,h_x,nbytes,cudaMemocpyHostToDevice);
cudaMemcpy(d_y,h_y,nbytes,cudaMemocpyHostToDevice);

// invoke DAXPY with 256 threads per Thread Block
int nblocks = (n+255)/256;

daxpy<<<nblocks, 256>>>(n,2.0,d_x,d_y);

//Copy data from device
cudaMemocpy(h_y,d_y,nbytes,cudaMemcopyDevicetoHost);
```

Asynchrony Rules in CUDA

All CUDA kernel invocations are asynchronous
- Kernel just scheduled on GPU
- CPU continues executing
- Synchronization through explicit sync routines or synchronizing MemCopy



However, multiple kernel invocations serialize by default
- Each kernel waiting on the completion of the previous one
- Synchronizing MemCopy waiting on all kernels



CUDA Streams

Streams represents queues for kernel execution
- Host CPU code places work into the stream/queue and continues
- GPU schedules kernels from queue when resources are free

Overlapping
- Operations within the same stream are ordered (FIFO) and cannot overlap
- Operations in different streams are unordered and can overlap

Stream Management

```
cudaStream_t stream;
```
Declares a stream handle

```
cudaStreamCreate(&stream);
```
Allocates a stream

```
cudaStreamDestroy(stream);
```
Deallocates a stream

Synchronizes host until work in stream has completed

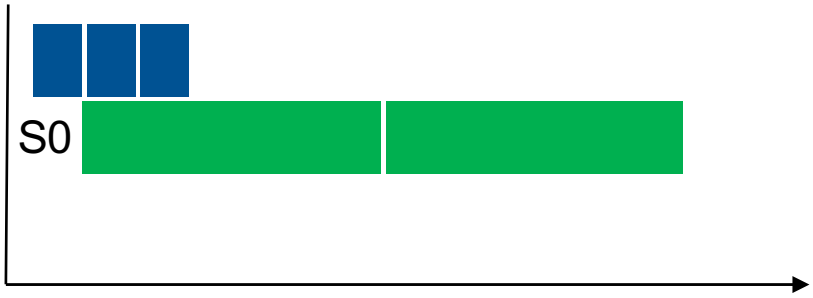
Kernel invocation on particular stream

```
name<<blocks, threads, size, stream>>(<params>)
```

CUDA Stream Example

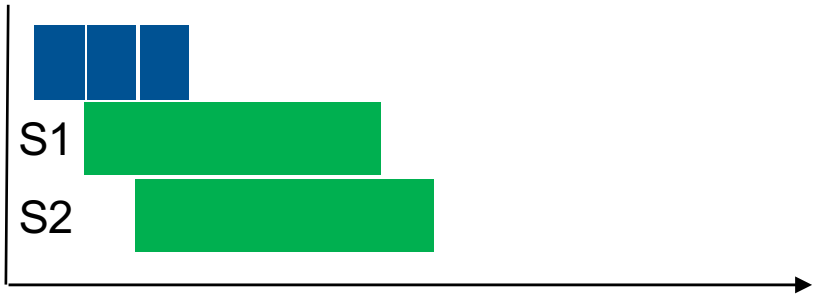
Starting two kernels

```
foo<<blocks,threads>>();
foo<<blocks,threads>>();
```



Starting two kernels on two streams

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

foo<<blocks,threads,0,stream1>>();
foo<<blocks,threads,0,stream2>>();

cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```



Default Stream 0

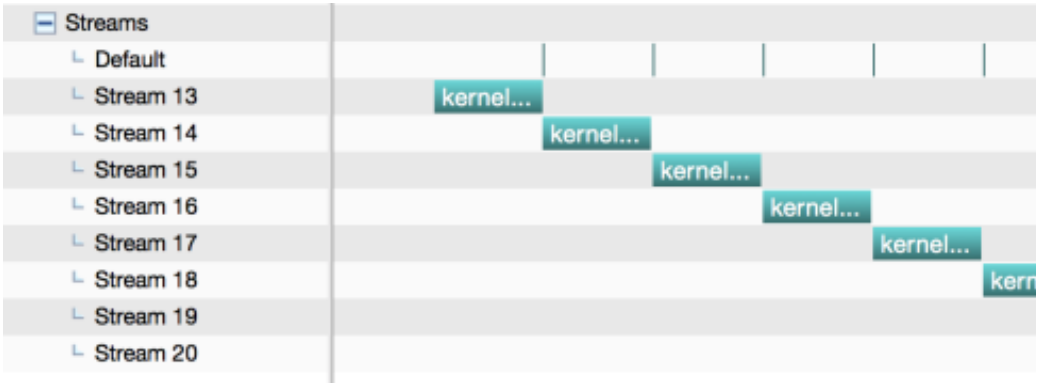
Starting two kernels with one extra stream

```
cudaStream_t stream;
cudaStreamCreate(&stream);

foo<<blocks,threads>>();
foo<<blocks,threads,0,stream>>();

cudaStreamDestroy(stream);
```

Stream 0 is the default stream

Stream is synchronizing to all streams

```
for (int i=0; i<ns; i++)
{
  cudaStreamCreate(&s[i]);
  kernel<<<1,64,0,s[i]>>>();
  kernel<<<1, 1>>>();
}
```



Per Stream Memory Copies

Synchronizing MemCopy (synchronizes to stream 0)

```
cudaMemcpy (void* d, void* s, size_t c, cudaMemcpyKind kind)
```

Async MemCopy

```
cudaMemcpyAsync (void* d, void* s, size_t c,
                 cudaMemcpyKind kind,
                 cudaStream_t stream = 0 )
```

Async copy requires pinned host memory

- Memory not pageable, i.e., can be safely transferred at any time
- Special allocation/deallocation calls

  `cudaMallocHost(...) / cudaHostAlloc(...)`

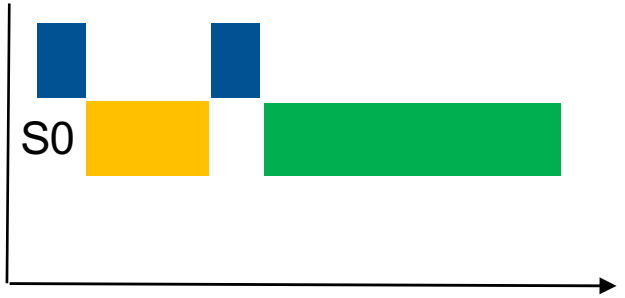
  `cudaFreeHost(...)`
- Calls to pin/unpin memory

  `cudaHostRegister(...) / cudaHostUnregister(...)`

Streams Can Also Help Transfer Concurrency

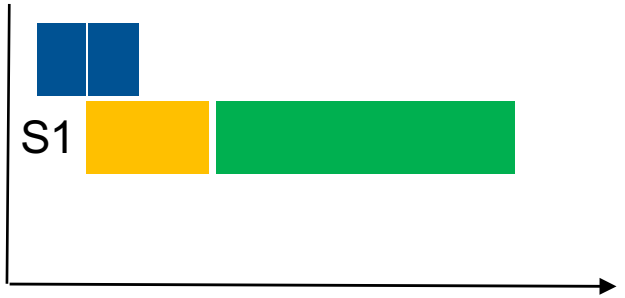
Synchronous

```
cudaMemcpy(...);
foo<<...>>();
```



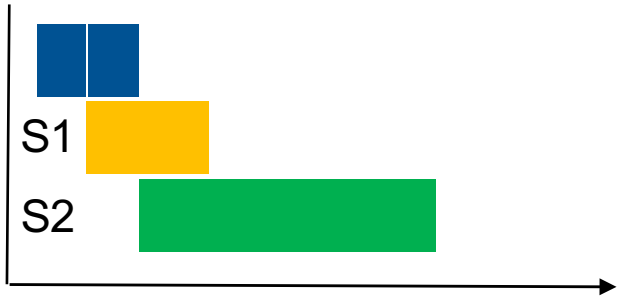
Asynchronous Same Stream

```
cudaMemcpyAsync(...,stream1);
foo<<...,stream1>>();
```



Asynchronous Different Streams

```
cudaMemcpyAsync(...,stream1);
foo<<...,stream2>>();
```



Advanced CUDA Synchronization Topics

Explicit Synchronization
- Synchronize everything: `cudaDeviceSynchronize()`
- Synchronize one stream: `cudaStreamSynchronize(stream)`
- Synchronize using events -> look it up

Stream Priorities
- High priority streams will preempt lower priority streams

  `cudaDeviceGetStreamPriorityRange(&low, &high)`
  - Lower number is higher priority
- Create using special API

  `cudaStreamCreateWithPriority(&stream, flags, priority)`

Stream Callbacks
- Registration of host side callbacks with respect to a stream
- Invocation when tasks in the stream are complete

More information
- http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf
- https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

Programming Models/Approaches So Far

Several on-node models

- Pthreads
  - Explicit access to hardware threads and synchronization
- OpenMP
  - Loop and task parallelism with more structured synchronization
- CUDA
  - NVIDIA GPU programming
- Focus on shared address spaces (even CUDA to some degree)

Only cross-node model: MPI

- Explicit message passing
- Point-to-Point communication as well as collectives
- Newer versions provide one-sided communication
  - Limited version of a shared address space

Question:

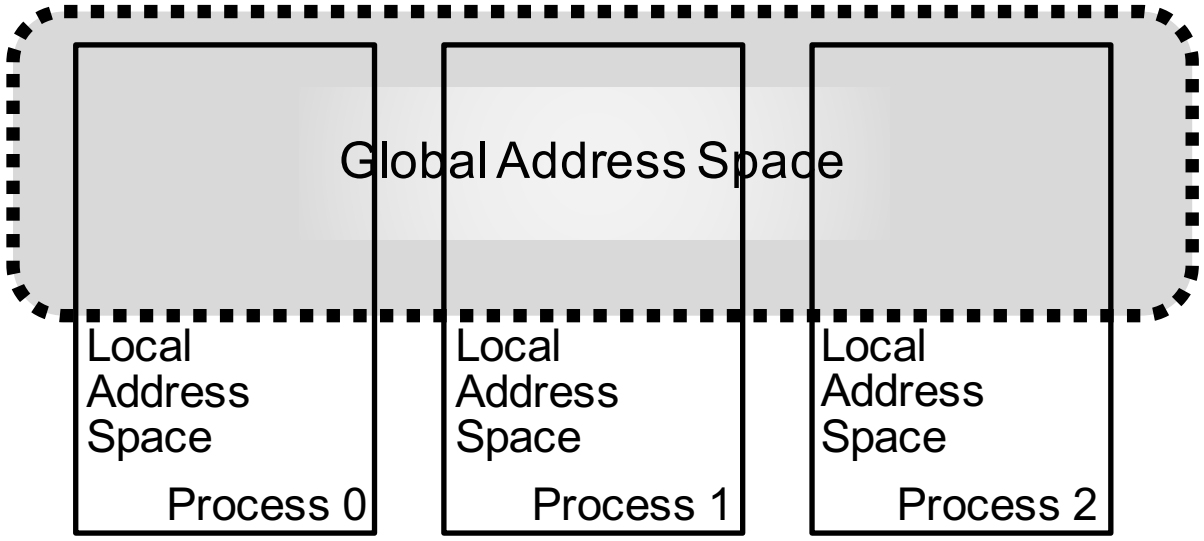
Can a global address space also work across nodes?

Partitioned Global Address Space

Global address space
- Any thread/process may directly read/write data allocated by another

Partitioned:
- Data is designated as local or global

By default:
- Heaps are Global
- Stacks are Local
- Can be adjusted



Examples
- SPMD languages: UPC, CAF, and Titanium
- All three use an SPMD execution model

Dynamic languages: X10, Fortress, Chapel and Charm++

Example: UPC

Unified Parallel C [LBL, UC Berkeley, GWU]

Both private and shared data

```
int x[10];          vs.  shared int y[10];
```

One-sided shared-memory communication through simple assignment:

```
x[i] = y[i];        or   t = *p;
```

Support for distributed data structures

- Distributed arrays; local and global pointers/references

Synchronization

- Global barriers, locks, memory fences

Collective Communication, IO libraries, etc.

https://upc-lang.org/ and http://upc.lbl.gov/ and http://upc.gwu.edu/

UPC Example

```
#define SIZE 4*THREADS
shared int A[SIZE];

int main()
{
        int i;
        upc_forall(i=0;  i < SIZE; i++; &A[i])
        {
                A[i] = MYTHREAD;
        }
        upc_barrier;
        if (MYTHREAD == 0)
        {
                for (i=0; i < SIZE; i++)
                        { printf("%d ", A[i]); }
                printf("\n");
        }
        return 0;
}
```

Affinity Expression

Local & Remote Reads from A

PGAS Pros & Cons

Advantages of PGAS
- Low latency memory access operations instead of two sided messaging
- Direct mapping of global data structures

Disadvantages of PGAS
- All the baggage of shared memory, incl. synchronization and possible races
- Harder to optimize
- Global memory accesses have larger latencies
- Only two hierarchy levels

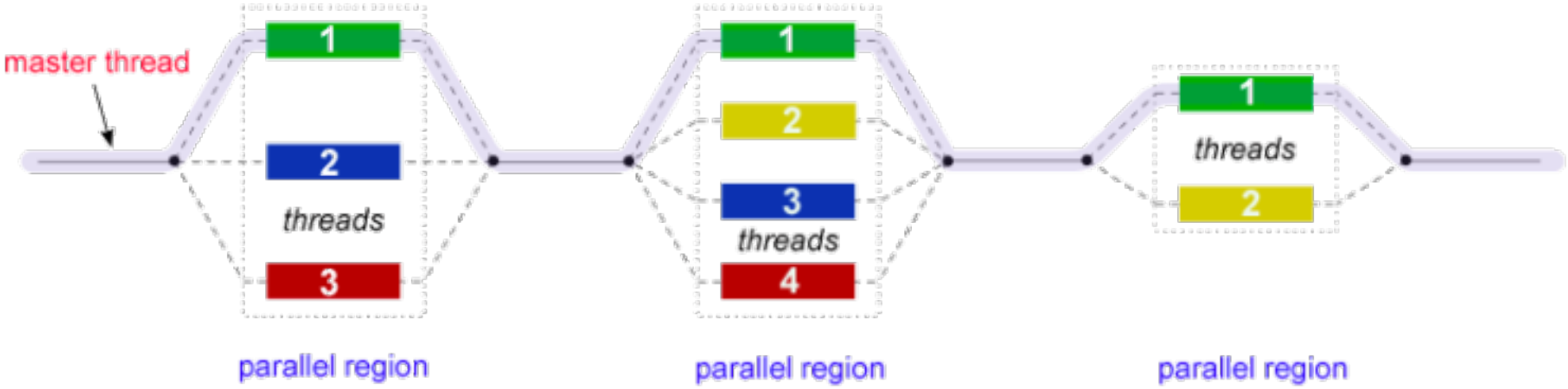
Common with the other approaches (MPI, OpenMP, Pthreads, CUDA, …)
- Low-level and close to hardware
- Good abstractions for performance oriented programming
- Not well suited for user-level and application-level abstractions

Question:

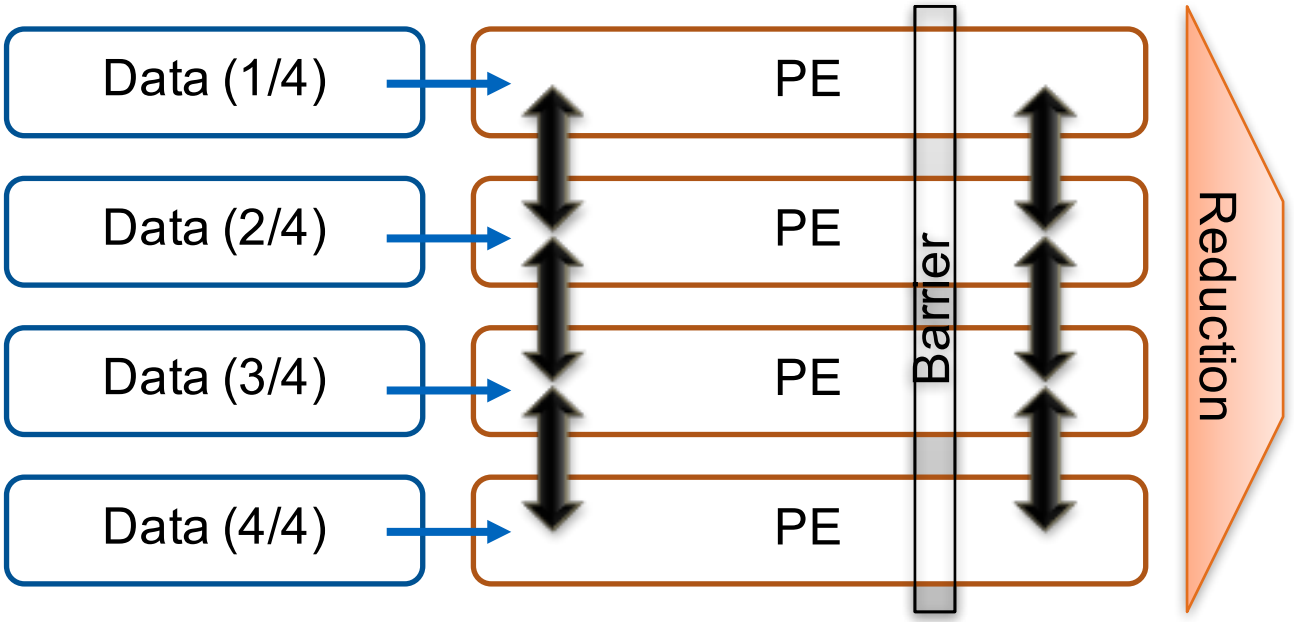
Can we find abstractions that are closer to the applications and its requirements?
How to get away from restrictions of SPMD and/or Fork/Join?

Traditional Parallelization Patterns

Fork/Join



SPMD



Task Graphs

Idea composition of the problem into basic elements

- Individual tasks
- Dependencies between tasks

Only have to fulfill dependencies

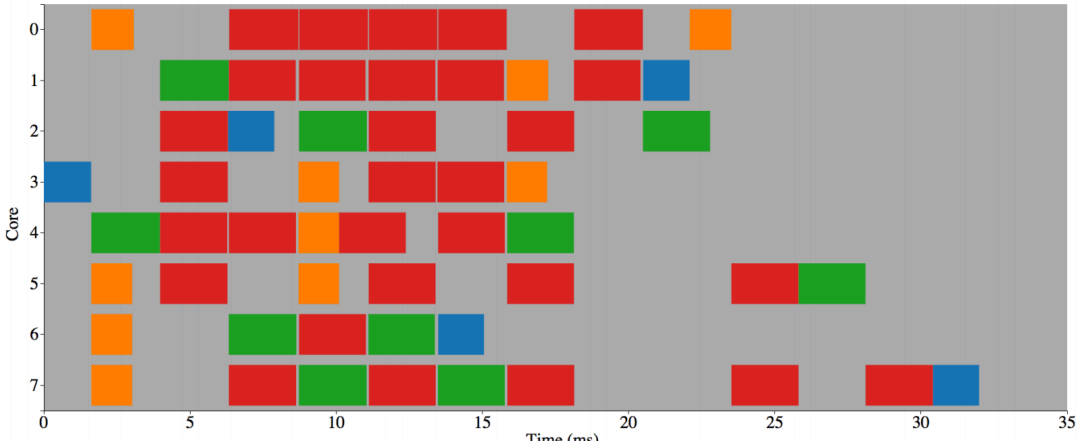
- No extra synchronization
- Scheduling freedom

Difficulties

- Granularity
- Overhead

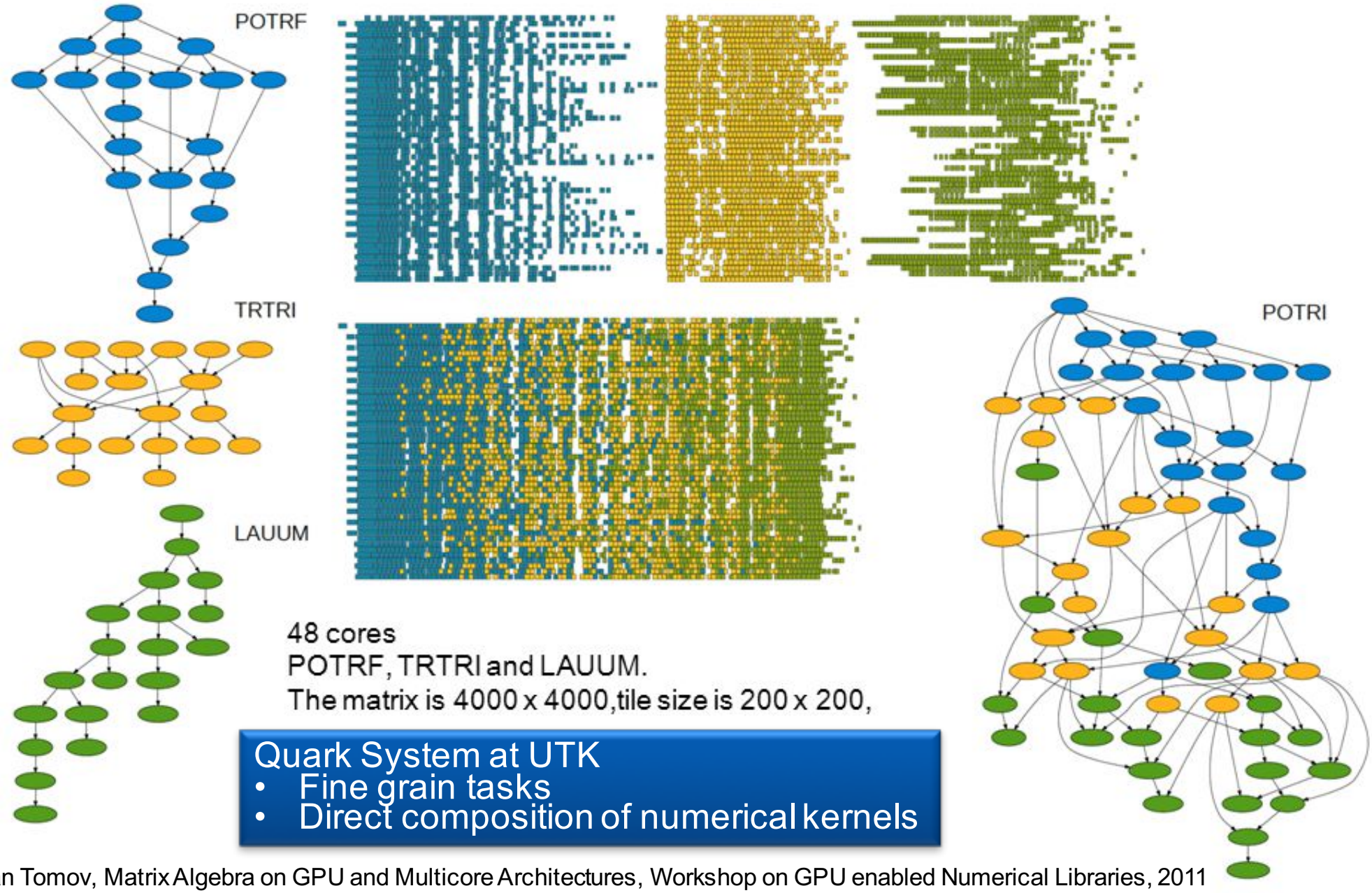
Challenge: schedule tasks

- Efficient mapping of tasks
- NUMA awareness
- Fast handover



Pictures from Haugen et al., VPA 2015

Pipelining: Cholesky Inversion



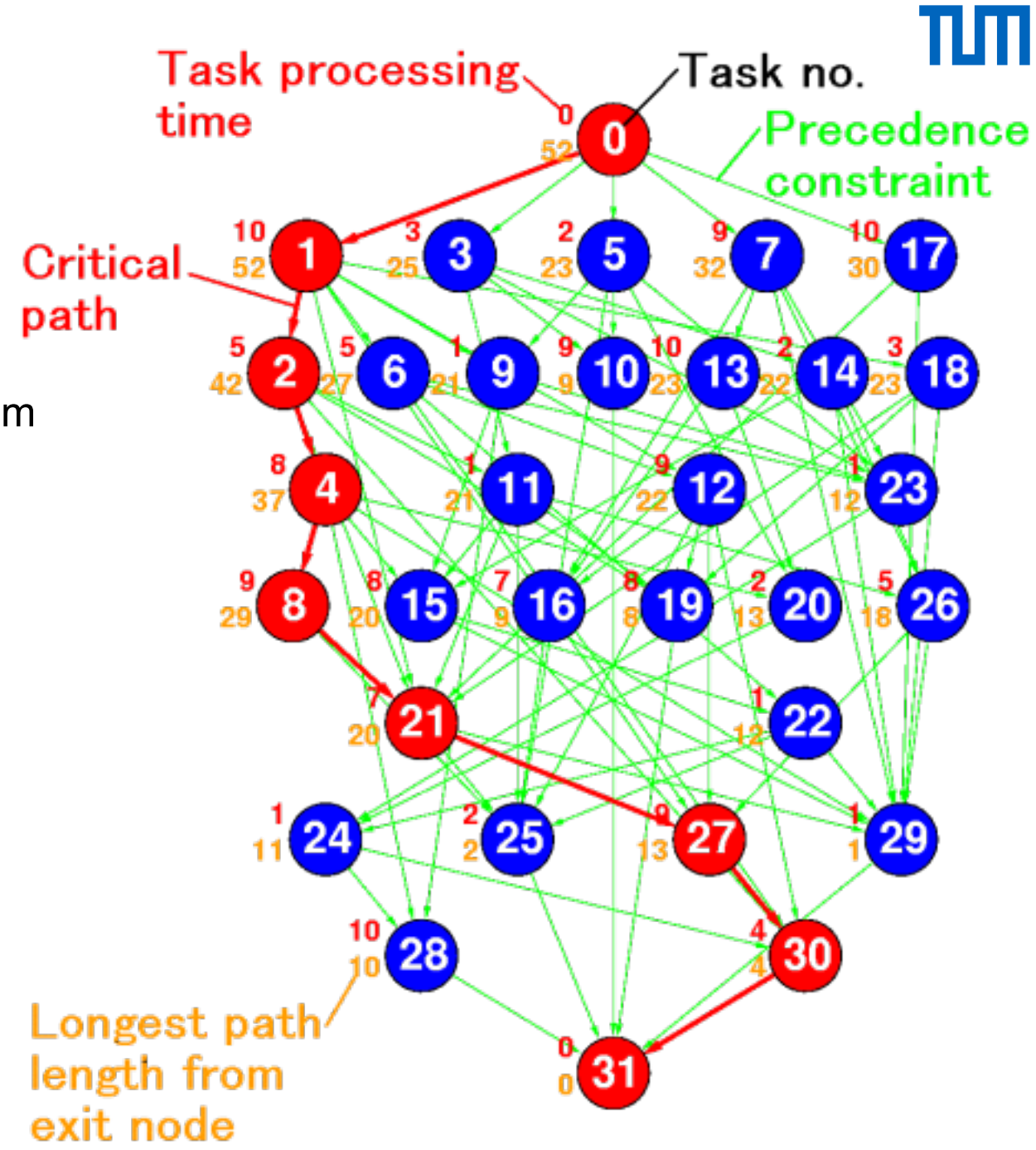
Quark System at UTK
- Fine grain tasks
- Direct composition of numerical kernels

Stan Tomov, Matrix Algebra on GPU and Multicore Architectures, Workshop on GPU enabled Numerical Libraries, 2011

Critical Paths

Only relevant metric
- Length of critical path
- Longest dependency list between tasks in system

Consequences
- Dictates performance
- Focus performance optimization on this path



From: STG http://www.kasahara.elec.waseda.ac.jp/schedule/index.html

Two Task-Based Examples

Charm++

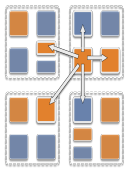
- Actor-Based programming language
- Developed at UIUC
- Slides by Prof. Dr. Laxmikant (Sanjay) Kale

Legion

- Data centric tasking runtime and programming model
- Developed at Stanford together with LANL and NVIDIA
- Slides by Prof. Dr. Alex Aiken

What is Charm++?

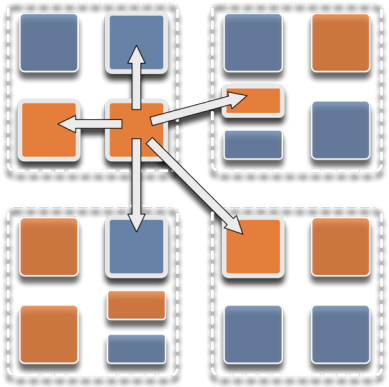
- Charm++ is a generalized approach to writing parallel programs
  - An alternative to the likes of MPI, UPC, GA etc.
  - But not to sequential languages such as C, C++, and Fortran
- Represents:
  - The style of writing parallel programs
  - The runtime system
  - And the entire ecosystem that surrounds it
- Three design principles:
  - Overdecomposition, Migratability, Asynchrony

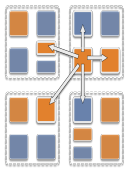


PPL
UIUC

Overdecomposition

- Decompose the work units & data units into many more pieces than execution units
  - Cores/Nodes/…
- Not so hard: we do decomposition anyway

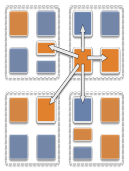




PPL UIUC

Migratability

- Allow these work and data units to be migratable at runtime
  - i.e. the programmer or runtime can move them
- Consequences for the application developer
  - Communication must now be addressed to logical units with global names, not to physical processors
  - But this is a good thing
- Consequences for RTS
  - Must keep track of where each unit is
  - Naming and location management

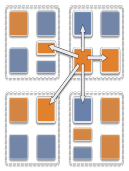


PPL UIUC

Asynchrony: Message-Driven Execution

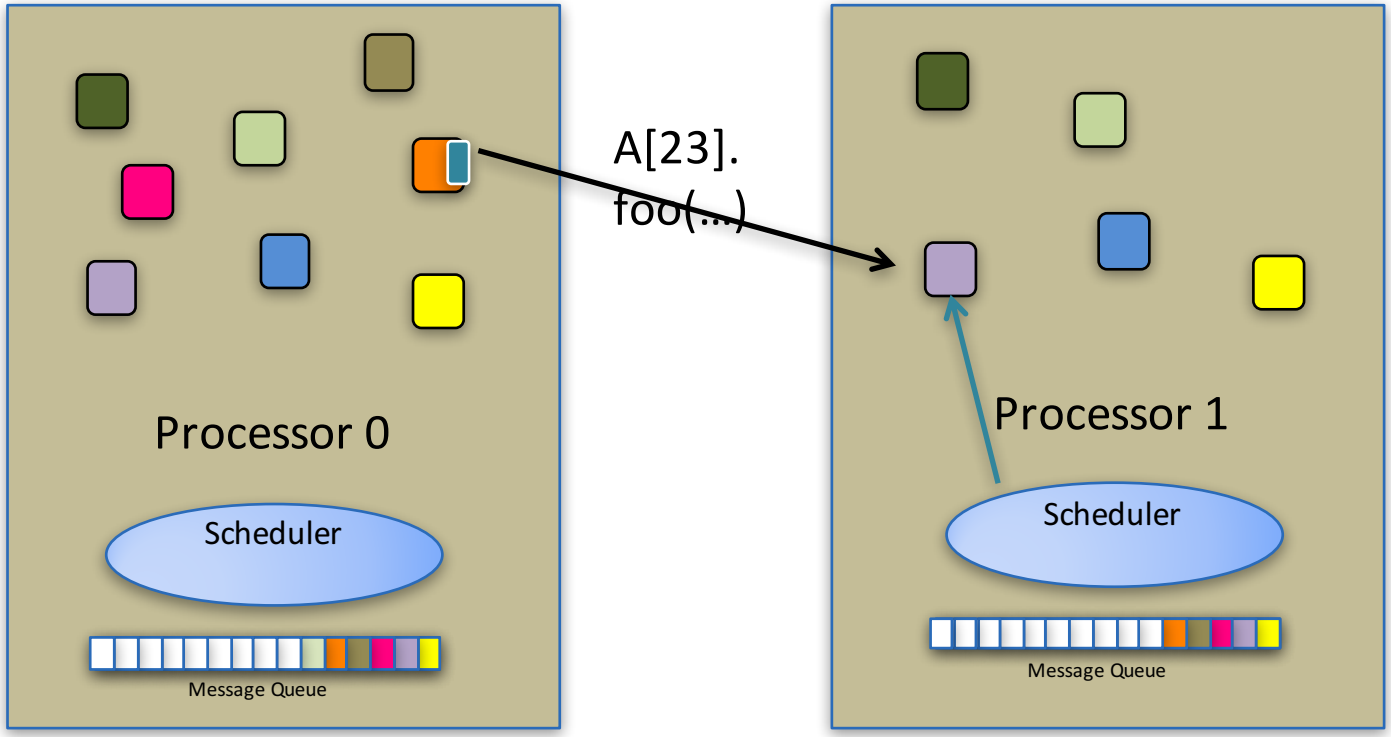
- With over-decomposition and migratability:
  - You have multiple units on each processor
  - They address each other via logical names

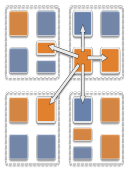
- Need for scheduling:
  - What sequence should the work units execute in?
  - One answer: let the programmer sequence them
    - Seen in current codes, e.g. some AMR frameworks
  - Message-driven execution:
    - Let the work-unit that happens to have data ("message") available for it execute next
    - Let the RTS select among ready work units
    - Programmer should not specify what executes next, but can influence it via priorities



PPL UIUC

Message-driven Execution

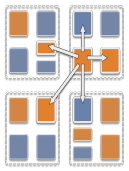




PPL
UIUC

Grainsize

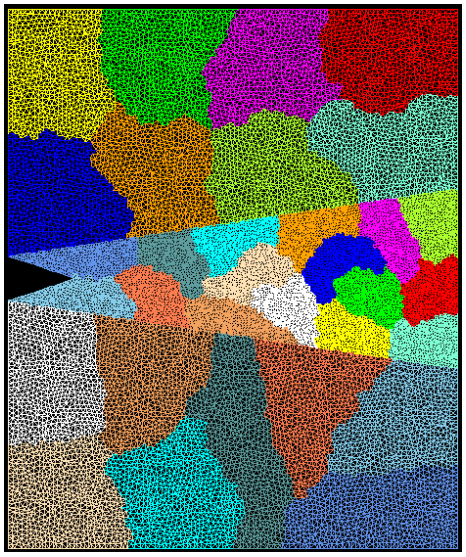
- Charm++ philosophy:
  - Let the programmer decompose their work and data into coarse-grained entities
- It is important to understand what I mean by coarse-grained entities
  - You don't write sequential programs that some system will auto-decompose
  - You don't write programs when there is one object for each *float*
  - You consciously choose a grainsize, **but** choose it independently of the number of processors
    - Or parameterize it, so you can tune later

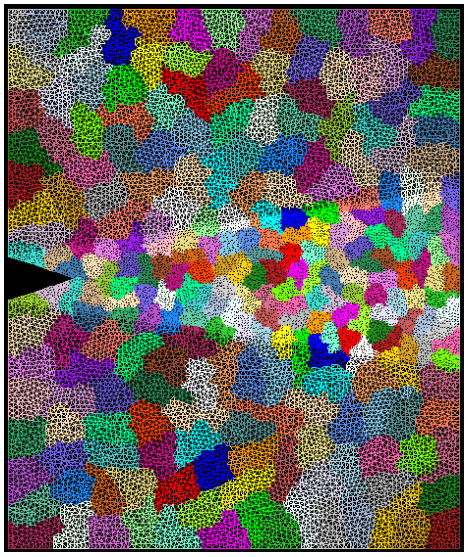


PPL UIUC

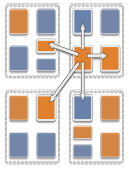
Crack Propagation

This is 2D, circa 2002...
but shows overdecomposition for unstructured meshes





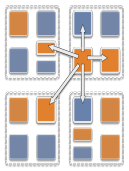
Decomposition into 16 chunks (left) and 128 chunks, 8 for each PE (right). The middle area contains cohesive elements. Both decompositions obtained using Metis. Pictures: S. Breitenfeld, and P. Geubelle



PPL
UIUC

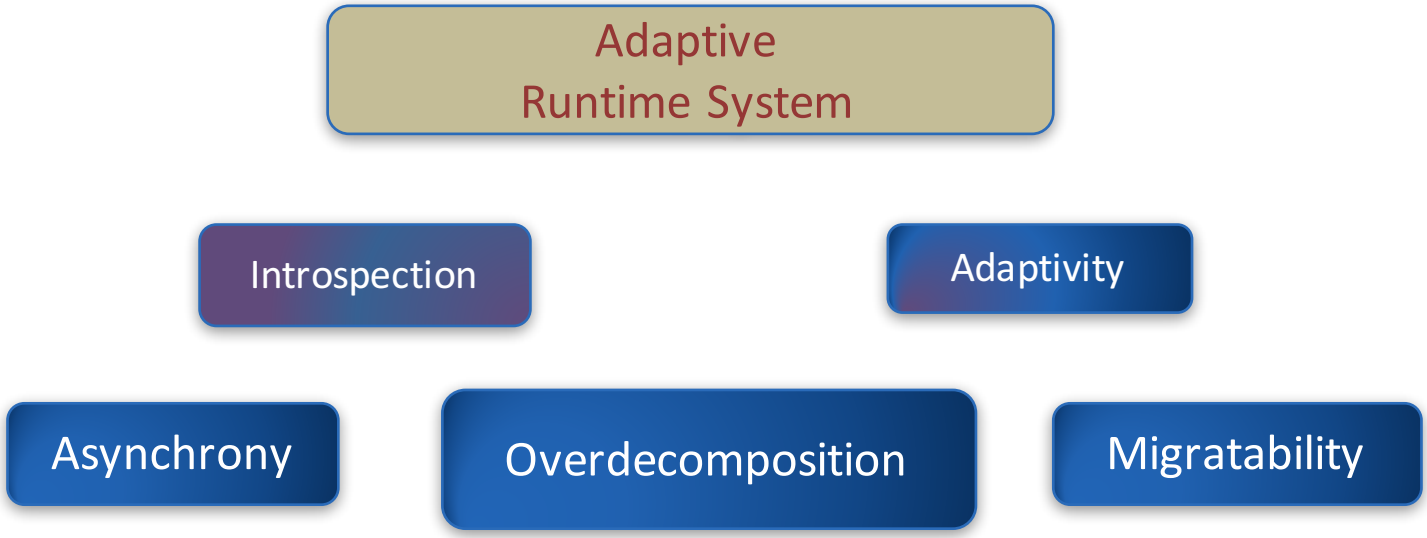
Realization of This Model in Charm++

- Overdecomposed entities: chares
  - Chares are C++ objects
  - With methods designated as "entry" methods
    - Which can be invoked asynchronously by remote chares
  - Chares are organized into indexed collections
    - Each collection may have its own indexing scheme
      - 1D, ..., 6D
      - Sparse
      - Bitvector or string as an index
  - Chares communicate via asynchronous method invocations
    - `A[i].foo(…);`
      - `A` is the name of a collection, `i` is the index of the particular chare.

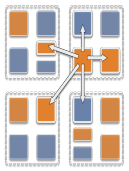


PPL UIUC

Empowering the RTS



- The Adaptive RTS can:
  - Dynamically balance loads
  - Optimize communication:
    - Spread over time, async collectives
  - Automatic latency tolerance
  - Prefetch data with almost perfect predictability



PPL
UIUC

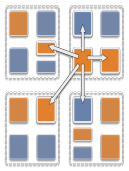
Hello World Example

- hello.ci file

```
mainmodule hello {
  mainchare Main {
    entry Main(CkArgMsg *m);
  };
};
```

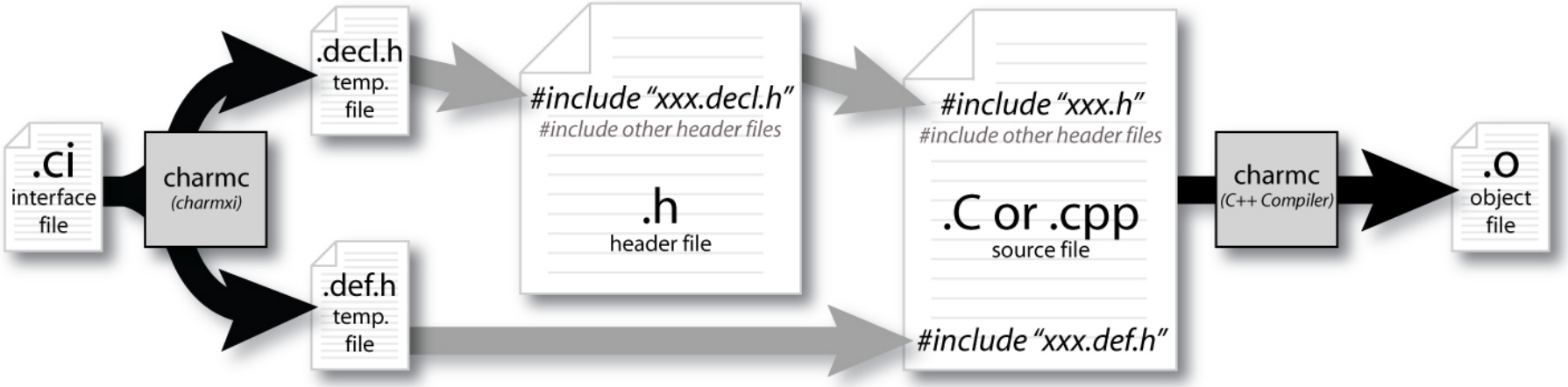
- hello.cpp file

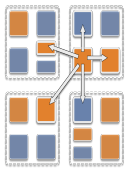
```
#include <stdio.h>
#include "hello.decl.h"

class Main : public CBase_Main {
  public: Main(CkArgMsg* m) {
    ckout << "Hello World!" << endl;
    CkExit();
  };
};

#include "hello.def.h"
```



PPL UIUC

Compiling a Charm++ Program

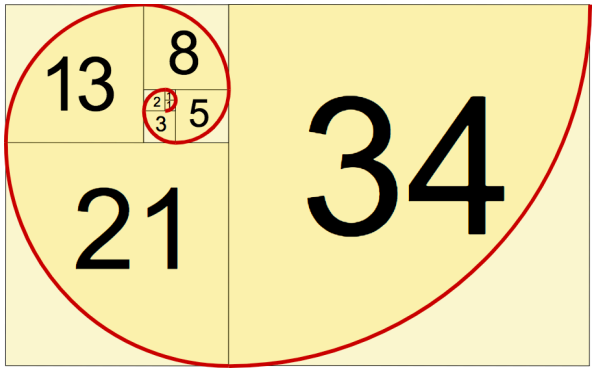




PPL UIUC

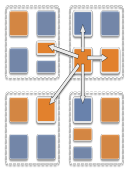
Example: Fibonacci (.ci file)

```
mainmodule fib {
  readonly int n;

  mainchare Main {
    entry Main(CkArgMsg* m);
    entry void done();
  };

  chare Fib {
    entry Fib(int n, bool is_root, CProxy_Fib parent);
    entry void calc(int n) {
      if (n < THRESHOLD)
        serial { respond(seqFib(n)); }
      else {
        serial {
          CProxy_Fib::ckNew(n - 1, false, thisProxy);
          CProxy_Fib::ckNew(n - 2, false, thisProxy);
        }
        when response(int val)
          when response(int val2)
            serial { respond(val + val2); }
      }
    };
    entry void response(int val);
  };
};
```



From: mathisfun.com

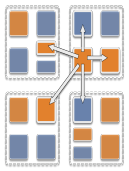
F(n)=
F(n-1) +
F(n-2)



PPL UIUC

Example: Fibonacci (.cpp file, part 1)

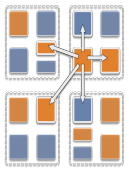
```
#include "fib.decl.h"

#define THRESHOLD 3 // calculate sequentially if below

/* readonly */ int n; // fibonacci number
/* readonly */ CProxy_Main mainProxy;

class Main : public CBase_Main {
  double start_time;

  public:
  Main(CkArgMsg* m) {
    // determine fibonacci number
    n = 20;
    if (m->argc == 2)
      n = atoi(m->argv[1]);

    CkPrintf("\n[Exercise 1. Fibonacci Sequence]\n");

    // create chares and start computation
    start_time = CkWallTimer();
    CProxy_Fib::ckNew(n, true, CProxy_Fib());
  }

  void done() {
    CkPrintf("Elapsed time: %lf seconds\n\n",CkWallTimer()-start_time);
    CkExit();
  }
```



PPL UIUC

Example: Fibonacci (.cpp file, part 2)

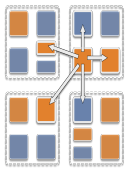
```
class Fib : public CBase_Fib {
  Fib_SDAG_CODE
  CProxy_Fib parent;
  bool is_root;

  public:
  Fib(int n, bool is_root_, CProxy_Fib parent_)
    : parent(parent_), is_root(is_root_) {
    thisProxy.calc(n);
  }

  int seqFib(int n) {
    return (n < 2) ? n : seqFib(n - 1) + seqFib(n - 2);
  }

  void respond(int val) {
    if (!is_root) {
      parent.response(val);
      delete this;
    }
    else {
      CkPrintf("Fibonacci number is: %d\n", val);
      mainProxy.done();
    }
  }
};

#include "fib.def.h"
```



PPL UIUC

Summary: What is Charm++?

- Charm++ is a way of parallel programming
- It is based on:
  - Objects
  - Overdecomposition
  - Asynchrony
    - Asynchronous method invocations
  - Migratability
  - Adaptive runtime system
- It has been co-developed synergistically with multiple CSE applications



PPL UIUC

Two Task-Based Examples

Charm++

- Actor-Based programming language
- Developed at UIUC
- Slides by Prof. Dr. Laxmikant (Sanjay) Kale

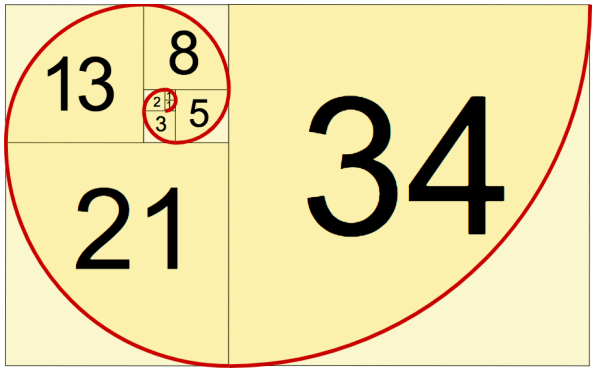
Legion

- Data centric tasking runtime and programming model
- Developed at Stanford together with LANL and NVIDIA
- Slides by Prof. Dr. Alex Aiken

Legion Approach

- Asynchronous tasking

Fibonacci (definitions)



From: mathisfun.com

F(n)=
F(n-1) +
F(n-2)

```
#include <cstdio>
#include <cassert>
#include <cstdlib>
#include "legion.h"

using namespace Legion;

enum TaskIDs
{
        TOP_LEVEL_TASK_ID,
        FIBONACCI_TASK_ID,
        SUM_TASK_ID,
};
```

Fibonacci (main)

```
int main(int argc, char **argv)
{
  Runtime::set_top_level_task_id(TOP_LEVEL_TASK_ID);
  {
    TaskVariantRegistrar registrar(TOP_LEVEL_TASK_ID, "top_level");
    registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
    Runtime::preregister_task_variant<top_level_task>(registrar, "top_level");
  }
  {
    TaskVariantRegistrar registrar(FIBONACCI_TASK_ID, "fibonacci");
    registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
    Runtime::preregister_task_variant<int, fibonacci_task>(registrar,"fibonacci");
  }
  {
    TaskVariantRegistrar registrar(SUM_TASK_ID, "sum");
    registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
    registrar.set_leaf(true);
    Runtime::preregister_task_variant<int, sum_task>(registrar, "sum",
                      AUTO_GENERATE_ID);
  }

  return Runtime::start(argc, argv);
}
```

Fibonacci (top-level task)

```
void top_level_task(const Task *task,
                    const std::vector<PhysicalRegion> &regions,
                    Context ctx, Runtime *runtime)
{
   int num_fibonacci = 7; // Default value
   const InputArgs &command_args = Runtime::get_input_args();

   for (int i = 1; i < command_args.argc; i++)
   {
      // Skip any legion runtime configuration parameters
      if (command_args.argv[i][0] == '-') { i++; continue; }
      num_fibonacci = atoi(command_args.argv[i]);
      break;
   }
   printf("Computing the first %d Fibonacci numbers...\n", num_fibonacci);
   std::vector<Future> fib_results;
   for (int i = 0; i < num_fibonacci; i++)
   {
      TaskLauncher launcher(FIBONACCI_TASK_ID, TaskArgument(&i,sizeof(i)));
      fib_results.push_back(runtime->execute_task(ctx, launcher));
   }
   for (int i = 0; i < num_fibonacci; i++)
   {
      int result = fib_results[i].get_result<int>();
      printf("Fibonacci(%d) = %d\n", i, result);
   }
   fib_results.clear();
}
```

Fibonacci (fib task)

```
int fibonacci_task(const Task *task,
                   const std::vector<PhysicalRegion> &regions,
                   Context ctx, Runtime *runtime)
{
   assert(task->arglen == sizeof(int));
   int fib_num = *(const int*)task->args;

   if (fib_num == 0) return 0;
   if (fib_num == 1) return 1;

   // Launch fib-1
   const int fib1 = fib_num-1;
   TaskLauncher t1(FIBONACCI_TASK_ID, TaskArgument(&fib1,sizeof(fib1)));
   Future f1 = runtime->execute_task(ctx, t1);

   // Launch fib-2
   const int fib2 = fib_num-2;
   TaskLauncher t2(FIBONACCI_TASK_ID, TaskArgument(&fib2,sizeof(fib2)));
   Future f2 = runtime->execute_task(ctx, t2);

   TaskLauncher sum(SUM_TASK_ID, TaskArgument(NULL, 0));
   sum.add_future(f1);
   sum.add_future(f2);

   Future result = runtime->execute_task(ctx, sum);
   return result.get_result<int>();
}
```

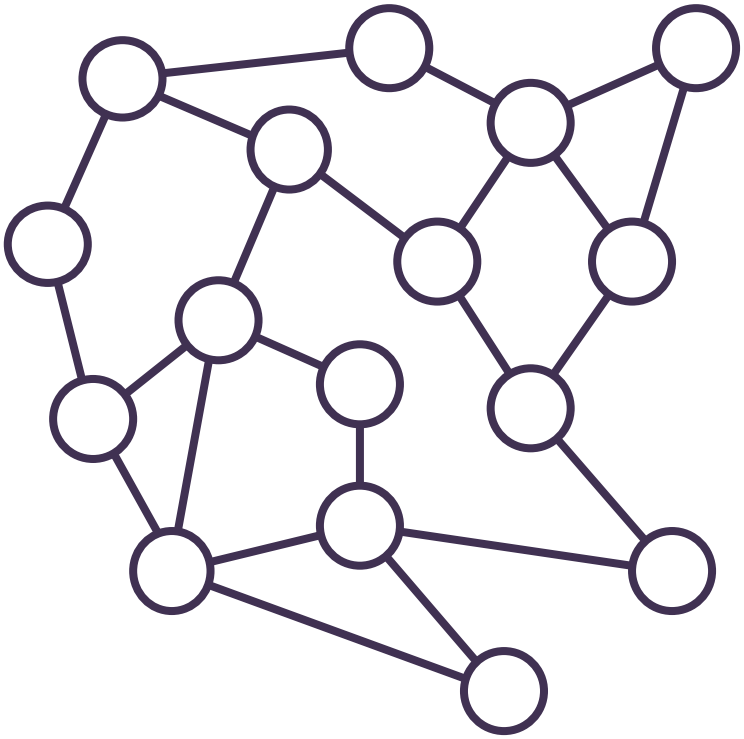
Fibonacci (sum task)

```
int sum_task(const Task *task,
             const std::vector<PhysicalRegion> &regions,
             Context ctx, Runtime *runtime)
{
   assert(task->futures.size() == 2);

   Future f1 = task->futures[0];
   int r1 = f1.get_result<int>();

   Future f2 = task->futures[1];
   int r2 = f2.get_result<int>();

   return (r1 + r2);
}
```

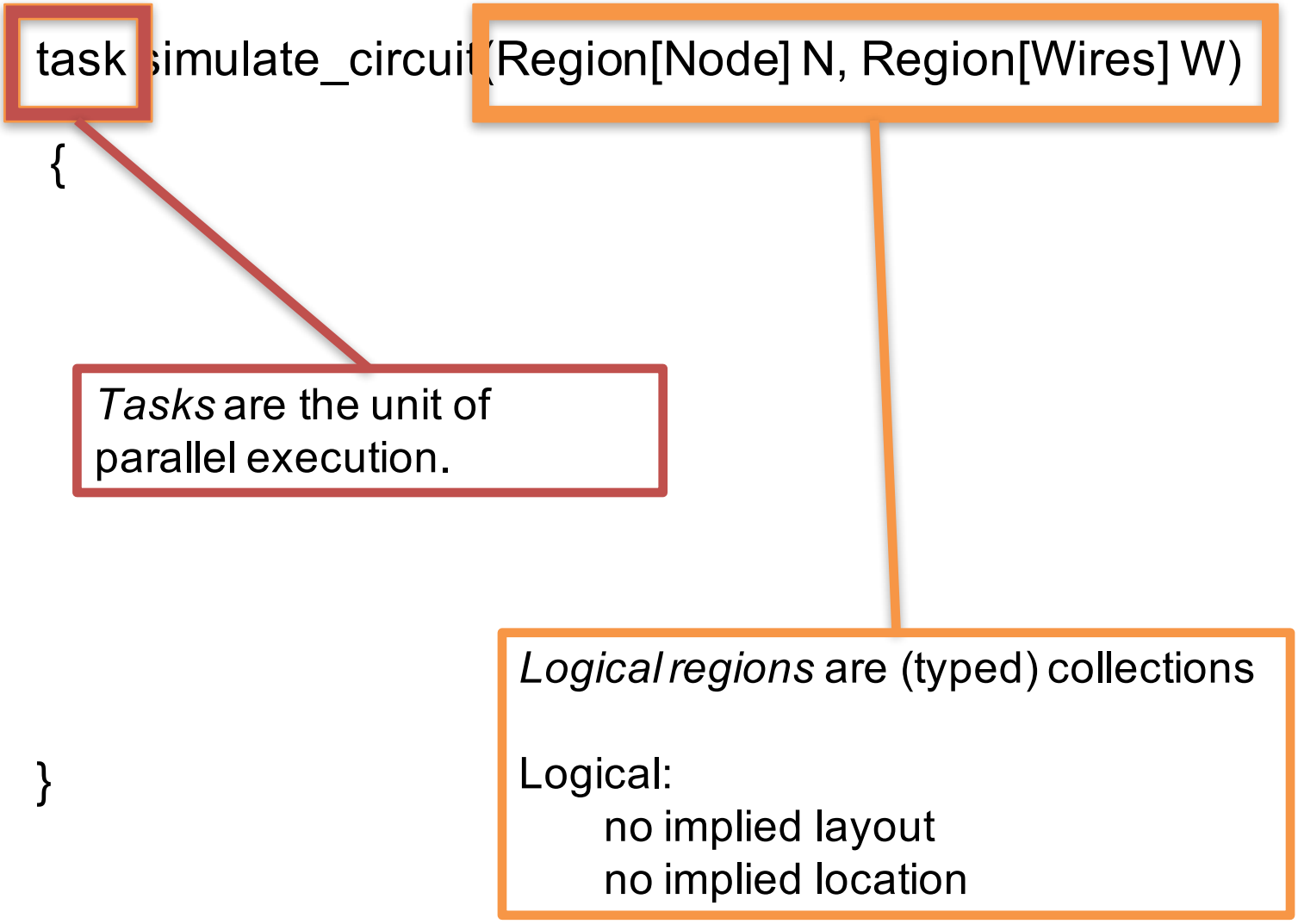
Legion Approach

- Asynchronous tasking
- Capture the structure of program data
- Decouple specification from *mapping*
- Automate
  - data movement
  - parallelism discovery
  - synchronization
  - hiding long latency operations

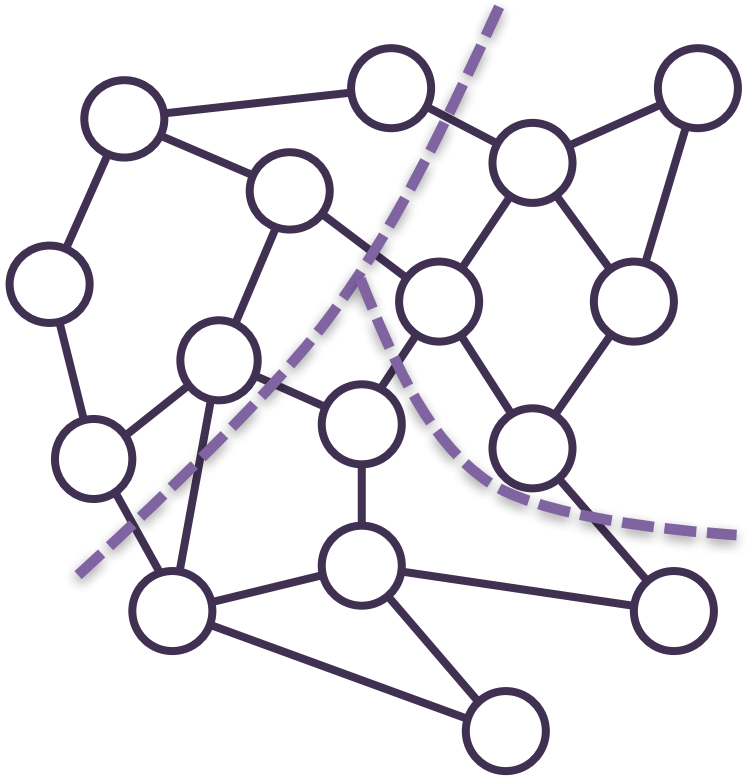
Example: Circuit Simulation



Example: Circuit Simulation

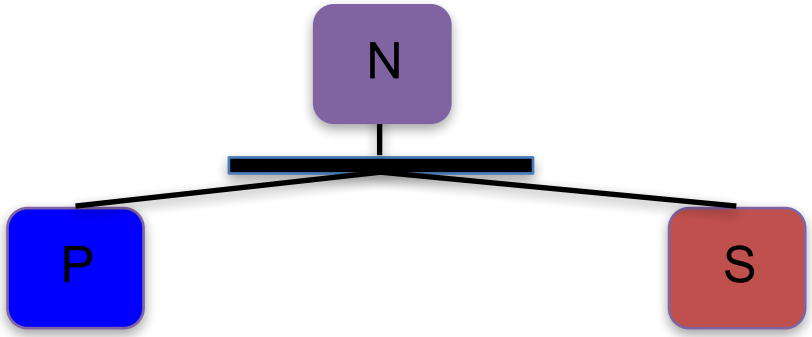


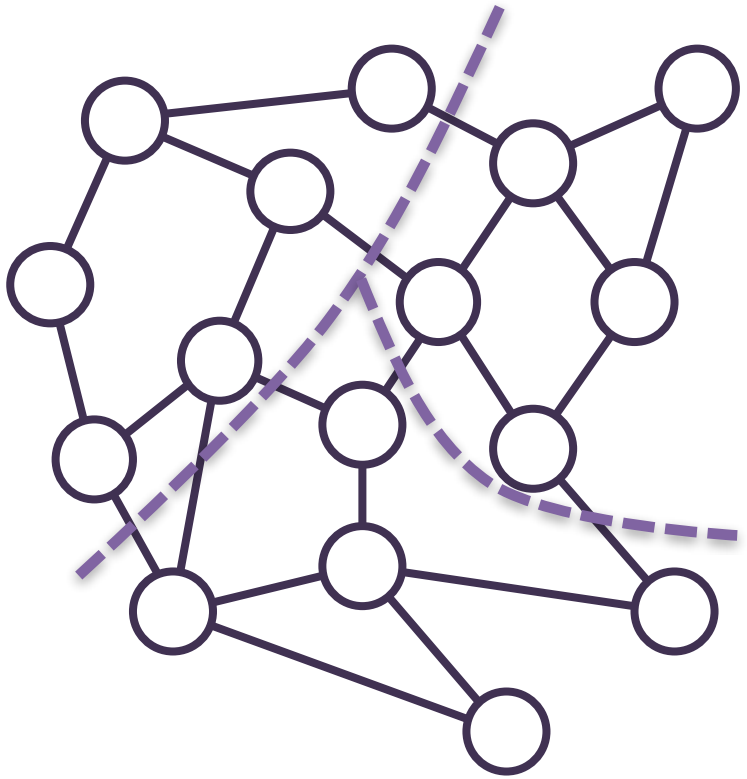
Partitioning



Partitioning

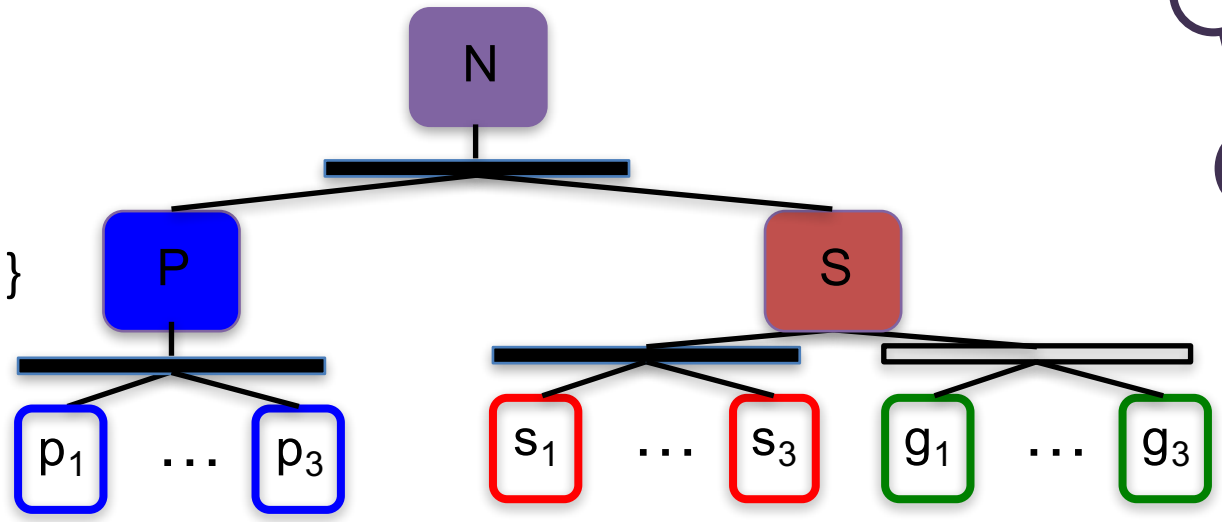
```
task simulate_circuit(Region[Node] N, Region[Wires] W)

{

    [ P, S ] = partition(ps_map, N)

}
```

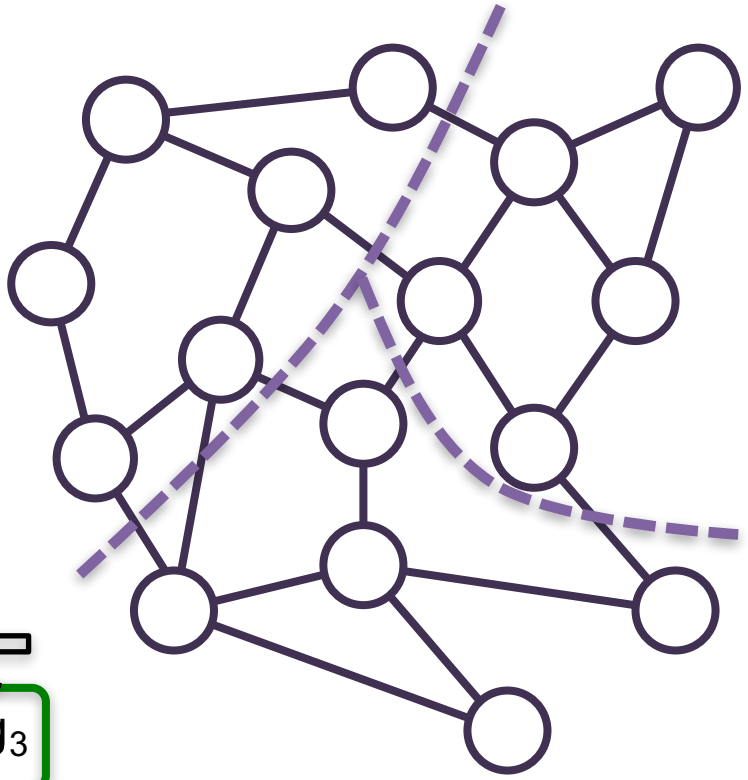




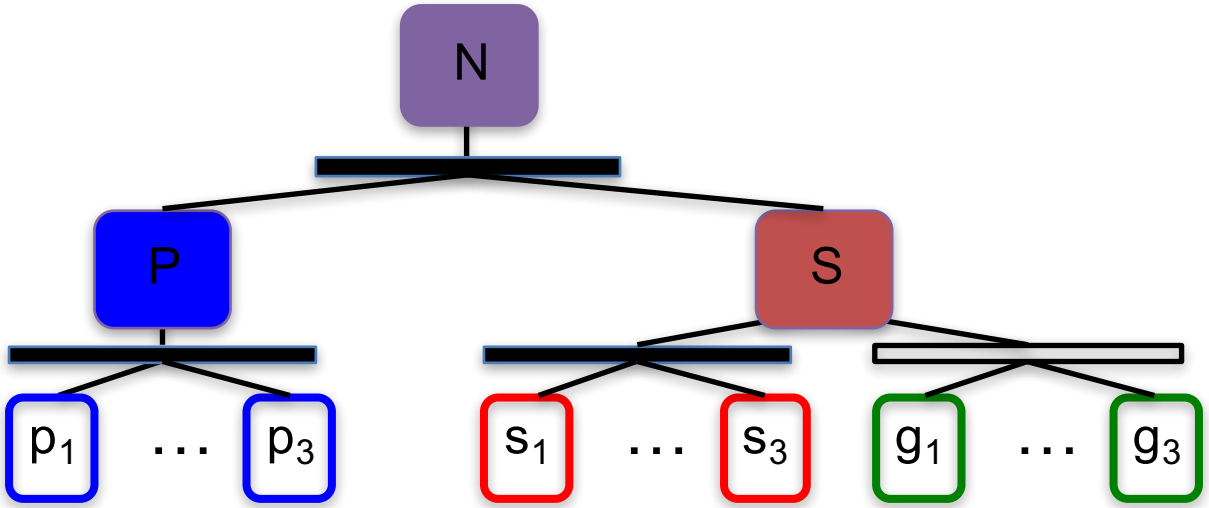
Partitioning

```
task simulate_circuit(Region[Node] N, Region[Wires] W)

{
    Array[Region[Node]] private, shared, ghost;
    …
    [ P, S ] = partition(ps_map, N)
    private = partition(private_map, P)
    shared = partition(shared_map, S)
    ghost = partition(ghost_map, S)
}
```

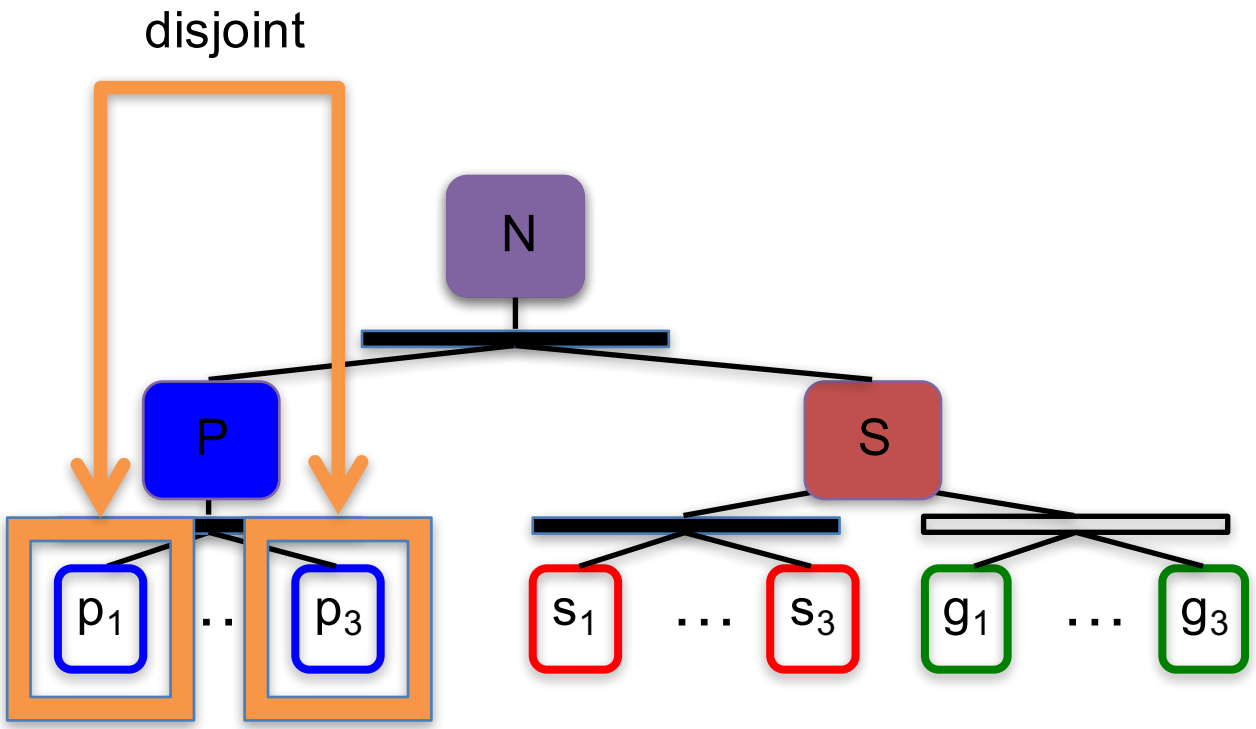




Region Trees

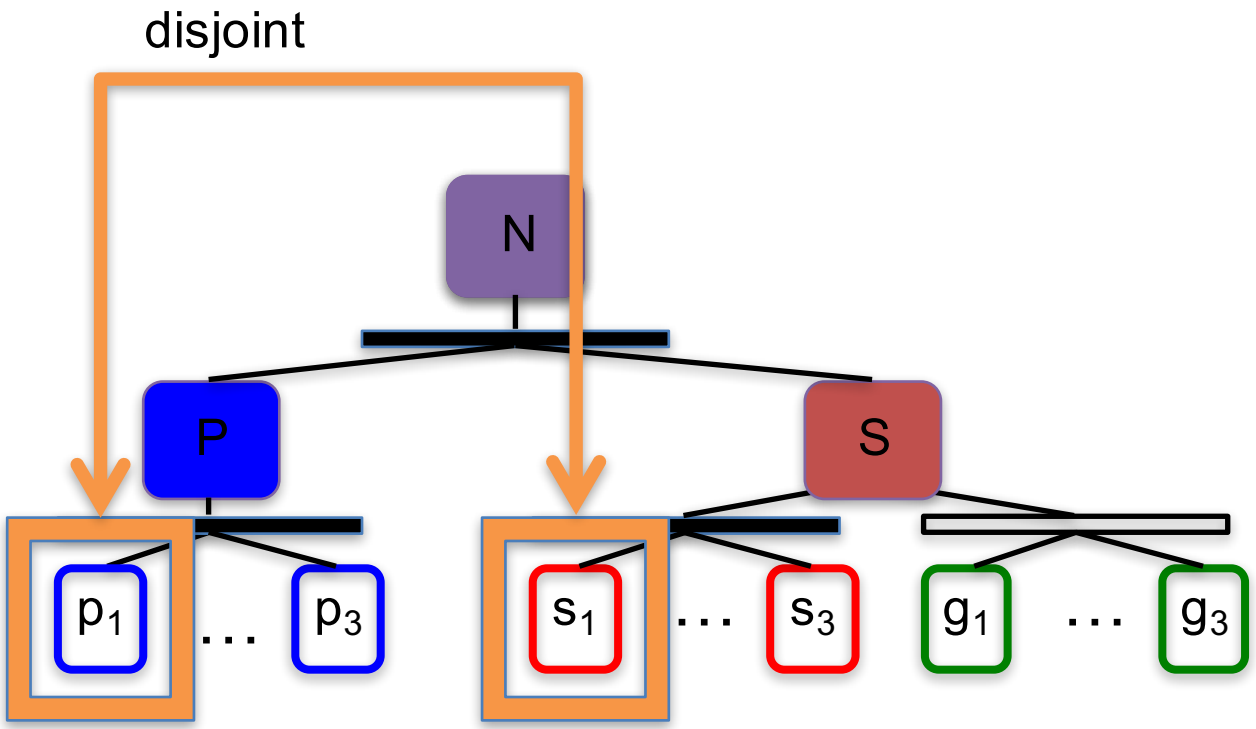


Region Trees



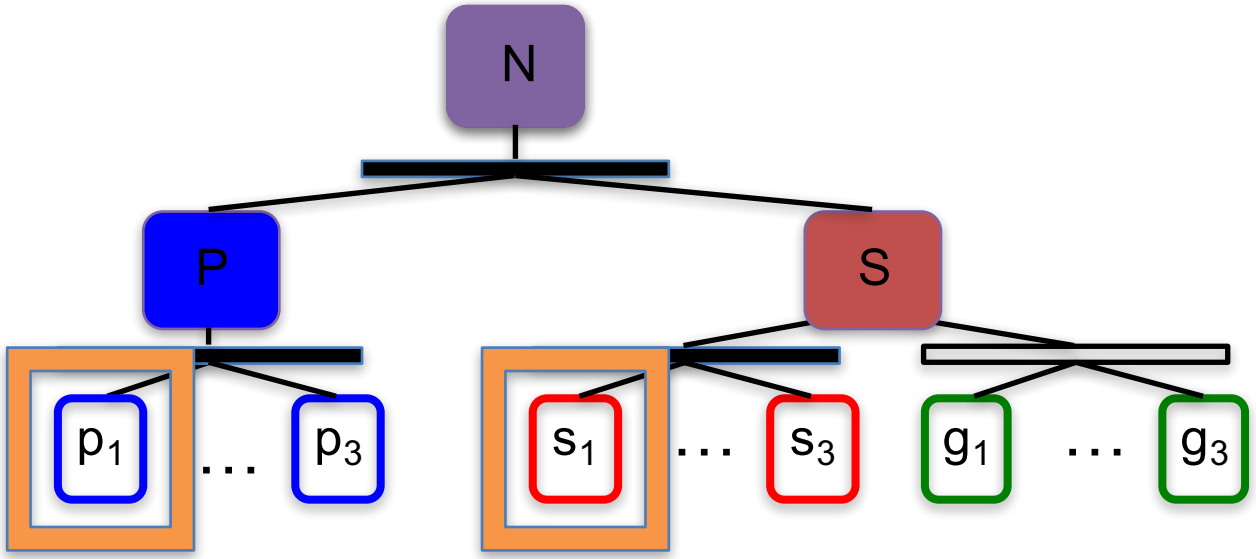
Locality
Independence

Region Trees



Locality
Independence

Region Trees

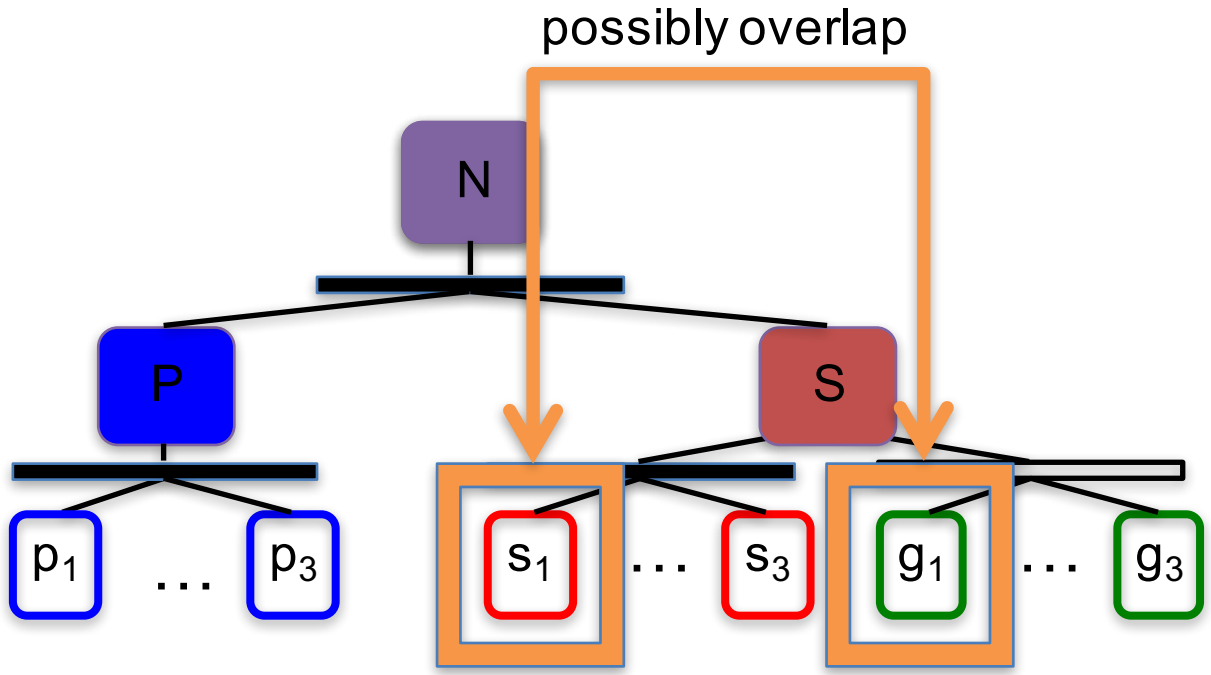


Locality

Independence

Aliasing

Region Trees



Locality

Independence

Aliasing

Legion Tasks & Privileges

```
task simulate_circuit(Region[Node] N, Region[Wires] W) :
      ReadWrite(N, W)
{
    …
    calc_currents(piece[0], p0 , s0 , g0 );
    calc_currents(piece[1], p1 , s1 , g1 );
    distribute_charge(piece[0], p0 , s0 , g0 );
    distribute_charge(piece[1], p1 , s1 , g1 );
    …
}
```

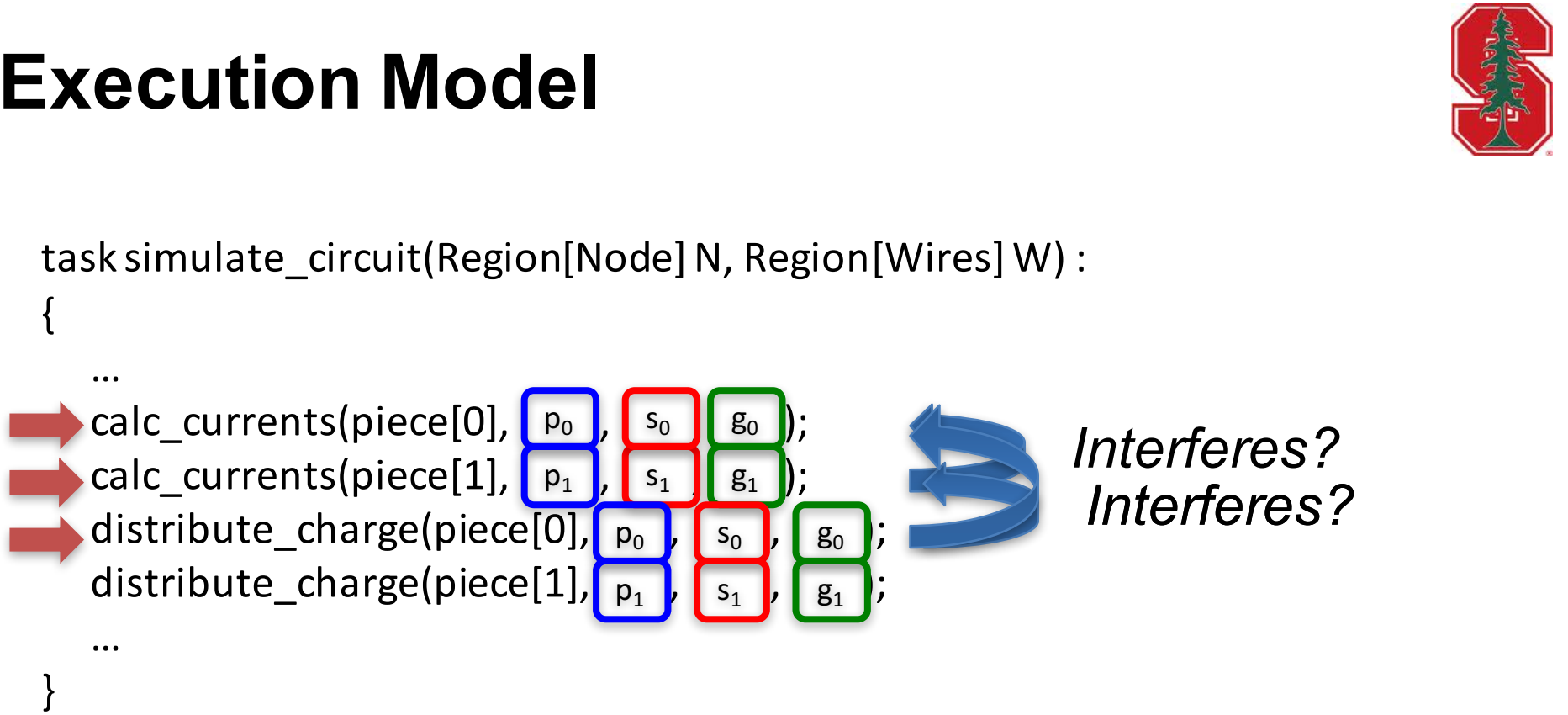
A task must declare the regions it will use & how.

```
task calc_currents(Piece  p) :
      Read(p.private, p.shared, p.ghost),
      ReadWrite(p.wires)

task distribute_charge(Piece  p) :
      Reduce(p.private, p.shared, p.ghost)
      Read(p.wires)
```

Subtask containment: A subtask can only use (sub)regions accessible to its parent task.

Tasks appear to execute in program order.

Execution Model



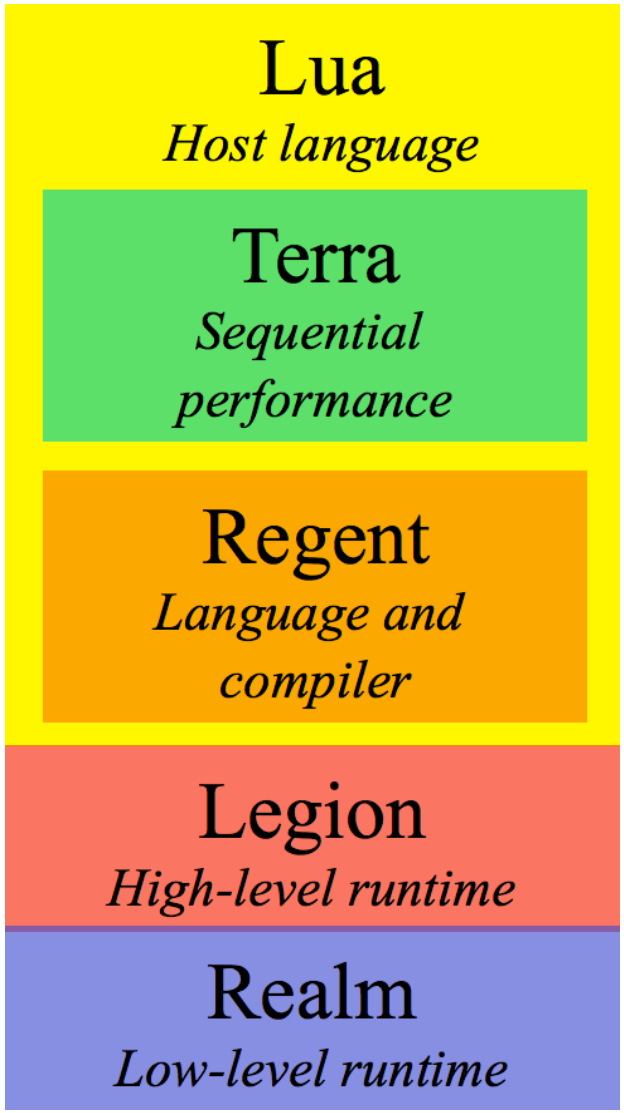
Tasks are issued in program order.

Legion Summary

- Logical regions: a relational data model
  - Support partitioning and slicing
  - Convey locality, independence/aliasing
- Implicit task parallelism
  - Task may have arbitrary sub-tasks
  - Tasks declare region usage including privileges and fields
- Tasks appear to execute in program order
  - Execute in parallel when non-interference established
- Portability by separating mapping from function

http://legion.stanford.edu

Legion as a Runtime for other Languages



From: Legion Bootcamp 2017

Summary: Alternative Systems

PGAS: Partitioned Global Address Space (Example: UPC)
- Alternative to MPI
- Process address space split into local and global memory
- User still responsible for synchronization

Task-based models
- Only focus on actual dependencies in the code
- Critical path becomes ultimate performance metric

Two instantiations of this concept
- Charm++: actor model
- Legion: data centric / separation of concerns of specification and mapping

Such models allow for more asynchrony
- Suitable for modern heterogeneous architectures with high variability
- Challenges: grain size, scheduling, and ultimately standardization

Task-based models will likely grow in importance

Lecture Wrap-Up

Basic introduction into parallelism
- The need and purpose of using parallelism
- Architectures and application areas
- Basic patterns from master/worker and SPMD to tasking
- Metrics (Speed-Up, Efficiency)

Parallel programming APIs
- Message Passing Interface (MPI), OpenMP, Pthreads
- Alternative models: CUDA, Task-based programming, …

Optimization and tuning
- Tuning of sequential code
- Scaling parallel code

Effective Parallelization requires rethinking of the problem
- Possibly different algorithms, data structures, …
- Every few orders of magnitude in scale once again changes the picture

In Case You Have Further Interest in the Topic

Follow-on lectures and Praktika

- IN2106: Efficient Programming of Multicore Processors and Supercomputers
- IN2310: Parallel Program Engineering (tools, optimization)
- IN????: Parallel Programming Systems (runtime system implementations)

Watch out for courses and events at LRZ

- In particular focusing on the new SuperMUC-NG

Possibility for Master thesis, guided research, …

- Prototyping proposals for the MPI and the OpenMP standard
- Implementation of parallel algorithms and applications
- Performance tuning and optimization
- Debugging and performance tools
- Fault tolerance
- Architecture and system evaluations
- ...

The World is Parallel – Think Parallel