CSEP 524: Parallel Computation
(week 9)

Brad Chamberlain

Tuesdays 6:30 – 9:20

MGH 231

A Note on Final Presentations

- 5 minutes is not a lot of time
  - you won't be able to say everything you've learned
  - pick the most important messages carefully
  - practice & edit a few times to dispel panic about timing

Surveys

- We'll be doing them tonight
  - at the end of class (?)
  - how long do they tend to take?

If we have to vote something off the island…

- We're going to flit around a bit tonight
- If we have to cut a corner, which should be cut?
  - some algorithms (I have many)
  - Software Transactional Memory (STM / atomic sections)
  - HPF/ZPL:
    - Failed languages of the 90's and their influence on Chapel

Amdahl’s Law

Amdahl's Law

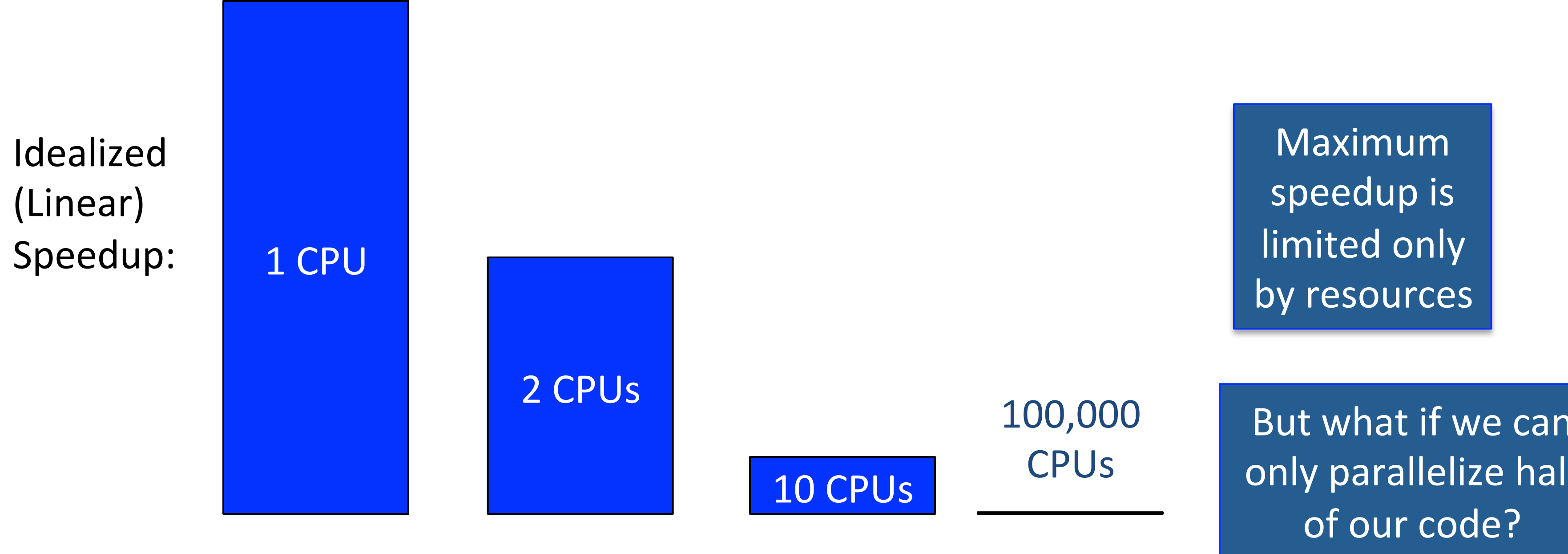
Amdahl's Law: The maximum speedup of a program is limited by the time required by the sequential portions of the code

- i.e., "if you can't parallelize something, eventually it will become the bottleneck."

Amdahl's Law

Amdahl's Law: The maximum speedup of a program is limited by the time required by the sequential portions of the code

– i.e., "if you can't parallelize something, eventually it will become the bottleneck."



Amdahl's Law

Amdahl's Law: The maximum speedup of a program is limited by the time required by the sequential portions of the code

- i.e., "if you can't parallelize something, eventually it will become the bottleneck."

Imagine that 50% of our original code cannot be parallelized…

1 CPU

2 CPUs

10 CPUs

100,000 CPUs

Maximum speedup is limited by sequential code.

Even with infinite CPUs, we could never do better than 2x speedup.

Counterpoint to Amdahl's Law

Reasons not to despair:

- lots of things are parallelizable
  - sometimes they just require a lot of cleverness
- the previous slide was a particularly bad case
  - sequential ops don't often account for bulk of running time
    - particularly as problems scale to massive sizes

- Yet, it is useful to keep in mind
  - to avoid undue frustration when hitting inherent limits
  - to avoid applying more HW than will help

Processor Technology Trends

Coarse Processor Taxonomy

Scalar processors:

- each instruction computes on singleton/scalar values
- this is what we traditionally think in terms of

Vector/SIMD processors:

- each instruction computes on a vector of values
- examples: Cray X1/X2, Nvidia GPUs, desktop CPUs

Multithreaded processors:

- support multiple threads in HW at a time; switch frequently
- examples: Cray MTA/XMT (Simon's talk), Sun Niagara

Coarse Processor Taxonomy

Sc

When available, these represent an additional, and important, source of parallelism within a program
- often targeted automatically by the compiler
- typically can be aided via #pragmas or the like

Vector/SIMD processors:

– each instruction computes on a vector of values

– examples: Cray X1/X2, Nvidia GPUs, desktop CPUs

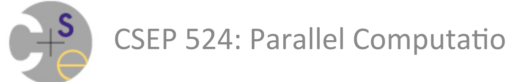
Multithreaded processors:

– support multiple threads in HW at a time; switch frequently

– examples: Cray MTA/XMT (Simon's talk), Sun Niagara

(GP)GPUs: The Promise

- GPUs: Similar in many respects to traditional HPC vector processors
  - each instruction can fire off a ton of operations
  - memory system highly optimized for such instructions
- In addition, has economy of scale going for it
  - many more videogame players than HPC users
- As a result, GPUs have been repurposed
  - "GPUs: they're not just for graphics anymore."
  - GP = General-Purpose
  - (in some circles "accelerators" is the more generic/PC term)

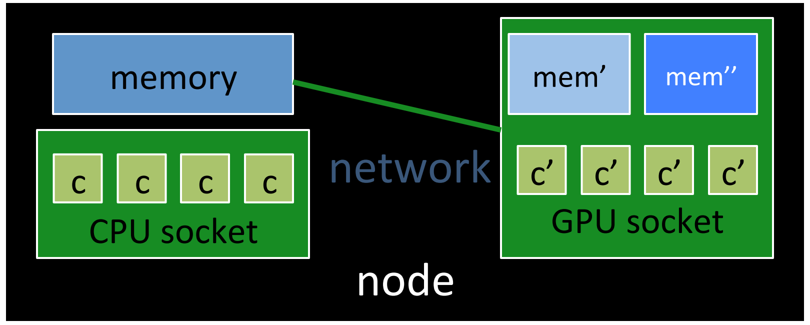


GPGPU: In Pictures

NVIDIA Fermi chip

Image Source: http://news.softpedia.com/newsImage/Nvidia-Kepler-GK104-GPU-Is-Reportedly-Smaller-than-the-HD-7970-2.jpg/

Abstract CPU + GPU Compute Node



GPUs: Limitations

- Tend to have limitations of one form or another:
  - historically:
    - only supported 32-bit floating point
    - not as robust as CPUs ("dropping a pixel for a frame no big deal")
    - programmed by expressing computations via graphics operations
  - more recently:
    - main memory not directly accessible: must copy in and out
    - inability to support function calls and/or recursion
    - esoteric programming models: CUDA, OpenCL
- Over time, things have been improving
  - higher-level programming models: OpenACC, OpenMP
  - yet, arguably there will always be differences/limitations (otherwise, it would simply be a CPU)

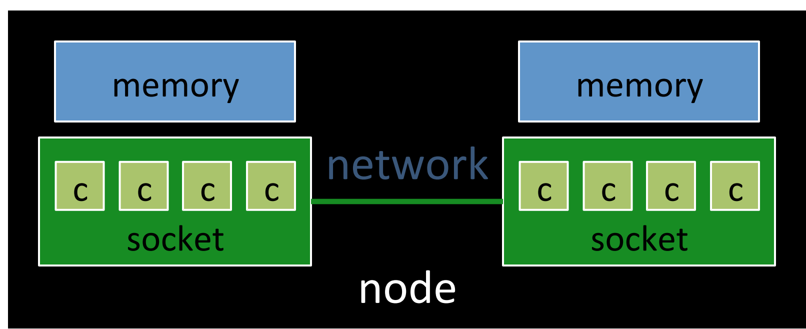
GPU Programming Models

- We'll be hearing about the major GPU programming models in the coming weeks
  - both nights have someone presenting on:
    - CUDA
    - OpenCL
    - OpenACC

NUMA Multicore Nodes

NUMA Multicore Compute Nodes: Multicore chips in which not all memory has uniform access cost

– think "ccNUMA architecture on a board"

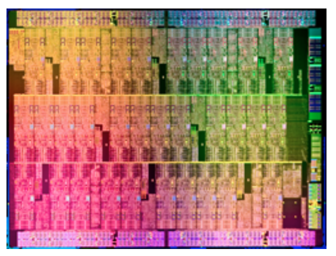


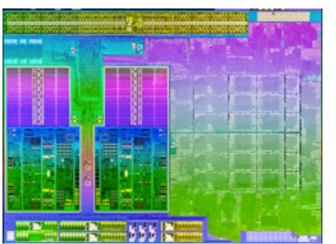
– supports shared memory programming models…
  - still can access all memory via loads/stores
  - still a single OS image per node

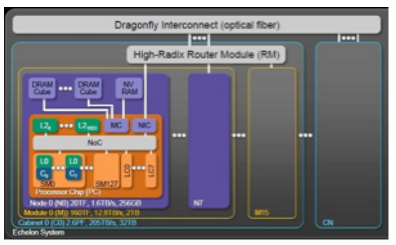
…but to maximize performance, attention to locality required

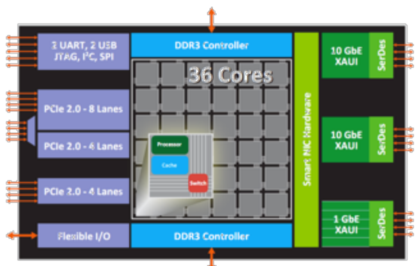
- as in distributed memory, run tasks on cores close to their data

Emerging Compute Nodes: General Characteristics









- Increasing numbers of cores
- Increased hierarchy and/or sensitivity to locality
  - CPU vs. GPU
  - NUMA multicore
- Potentially heterogeneous processor/memory types
  - CPUs vs. GPUs
  - CPU memory vs. GPU memories of various flavors
- Increasingly resemble supercomputers-on-a-chip

⇒ Next-gen programmers will have a lot more to think about at the node level than in the past

HPC Concerns for the coming generation

- System scale is reaching some intimidating limits
  - power budget
  - resilience to (increasingly likely) failures
- Machine model is changing for first time in decades
  - can no longer treat as flat set of homogenous resources
- Diversity in node architectures
  - very different solutions coming from Intel, AMD, Nvidia, …
  - machine model doesn't gloss over differences as in past
- Traditional programming models breaking down

HPC Programming Models & Emerging Node Types

- MPI continues to make sense for inter-node
- but less and less so for intra-node
  – too heavyweight / process-oriented for emerging nodes

Q: So what do we do?

A1: Hybrid programming models?

– e.g., MPI + OpenMP + OpenACC/CUDA/OpenCL?
  - (or maybe simply MPI + OpenMP once it catches up)

A2: A good time for something new?

Chapel: Well-Positioned for Next-Generation

- Multiple styles of parallelism
  - data- vs. task- (*may* and *must*), including nested
  - contrast with task- or process-only

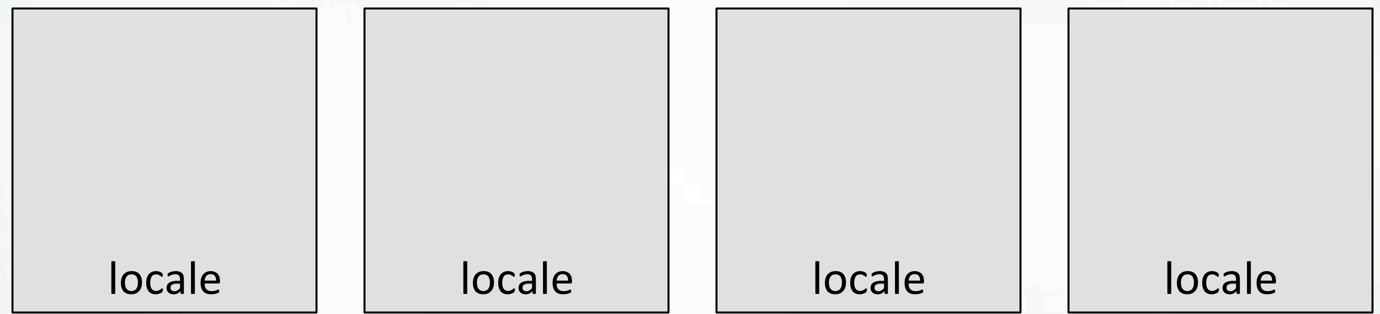
- Distinct concepts for parallelism vs. locality
  - tasks vs. locales
  - contast with:
    - conflation of parallelism and locality (SPMD: MPI, UPC, CAF, …)
    - no real support for locality (OpenMP, Pthreads, …)

Yet additional work remains…

Locales Today

Concept:

- Today, Chapel supports a 1D array of locales
  - users can reshape/slice to suit their computation's needs

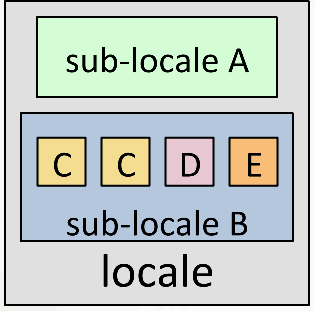


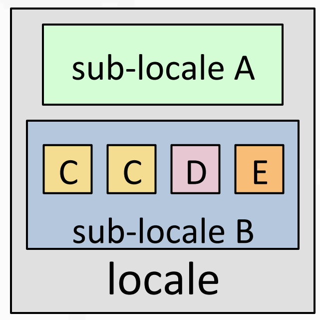
- Apart from locale queries, no further visibility into locale
  - no mechanism to refer to specific NUMA domains, processors, memories, ...
  - assumption: compiler, runtime, OS, HW can handle intra-locale concerns
- Today's locales support *horizontal*, but not *vertical* locality

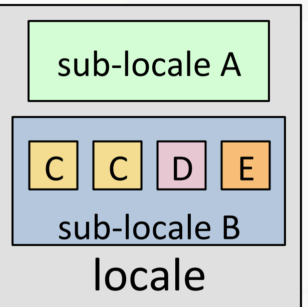
Current Work: Hierarchical Locales

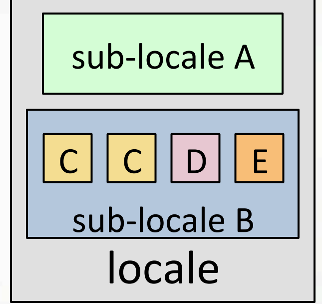
Concept:

- Support locales within locales to describe architectural sub-structures within a node









- As with traditional locales, *on-clauses* and *domain maps* can be used to map tasks and variables to a sub-locale's memory and processors

Goal: Define locale structure as Chapel code

- permits implementation policies to be specified in-language
- introduces a new Chapel role: *architectural modeler*

Evaluating Programming Models

Our Shared Memory Characterizations

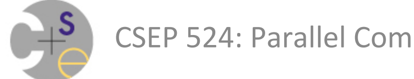
	C+Pthreads	Chapel	OpenMP
degree of voodoo	less voodoo	more voodoo	moderate-to-more voodoo
level of abstraction	more HW-oriented	more problem-oriented	in the middle
verbosity	more verbose	less verbose	in between
control of memory (alignment/ padding)	more control due to C	less control (today)	same as C+Pthreads
HW independenc e	less abstracted from HW	more abstracted…	more abstracted…
portability	quite good	potentially more portable	as portable as C, Fortran, C++

Our Shared Memory Characterizations

	C+Pthreads	Chapel	OpenMP
libraries	lots of existing library support	very little currently* * = extern support for C	can call sequential C
opportunities for error	more opportunities due to C and details of sync primitives	less so	fragility w.r.t. mistyped pragma prefixes (use –Wall); ability to break seq case (reduce/SPMD)
notation	library	language	pragmas
maturity	very mature	much less so	mature, but evolving
“classic” concepts (mutex, condvar, …)	the set of classic concepts	pretty significant departure	lower-level (locks), and higher (critical sections, barriers, reductions, data parallelism)
completeness	confidence that it’s complete	unclear	reasonably complete (no must parallelism)

Our Shared Memory Feature Comparison

	C+Pthreads	Chapel	OpenMP
data parallelism	no	yes	yes
may tasks	yes? (no implicit support)	yes	yes
must tasks	yes	yes	not well
barriers	no	no (not yet)	yes
reductions	no	built-in + user-defined	built-in
scans	no	built-in + user-defined	no?
locks	yes	sync vars	yes (library)
incremental parallelism	so-so	so-so –to- yes	yes
scalability to dist. mem/ locality	no	yes	no



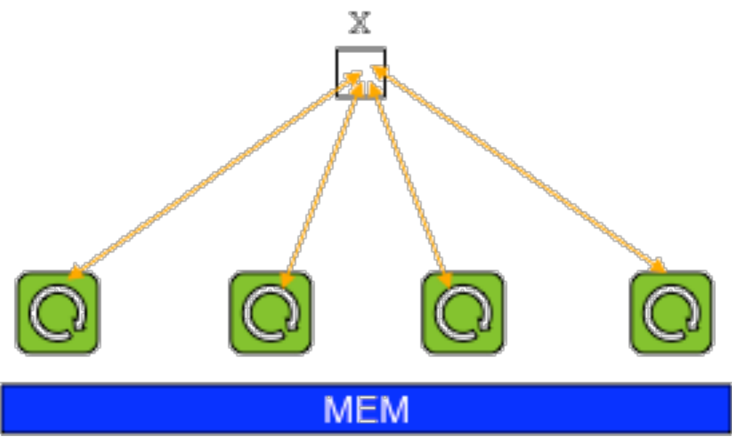
Distributed Memory Characterizations

	MPI	Chapel	CAF/UPC
readability	medium-to-low (but may vary with approach & SW Eng)	good	medium
explicitness of comm.	in your face	syntactically invisible (but semantics/ mechanisms to reason about it)	square brackets / invisible (similar to Chapel)
control over comm. granularity	lots	none-ish (compiler should optimize array slice assignments)	none-ish
distrib. data structures	manually fragmented	global-view (though you could fragment)	syntactically fragmented / global
debuggability	not so good	not so good	???
ease of use	pretty hard	pretty easy	middle

Shared Memory Programming Models

***e.g.*, OpenMP, pthreads**

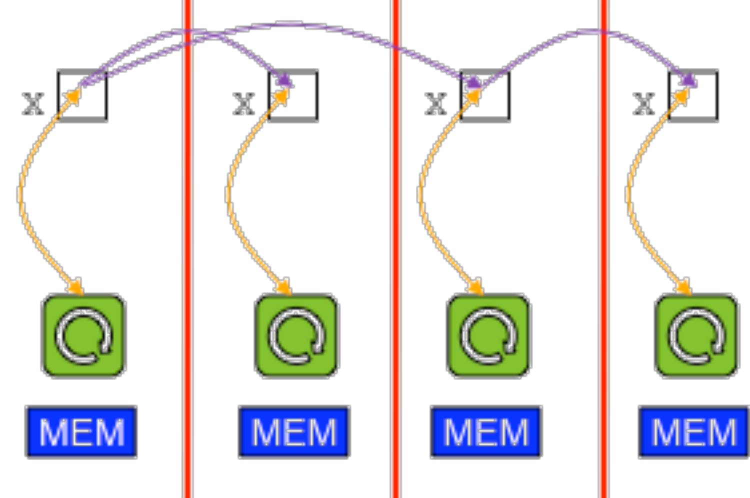
+ support dynamic, fine-grain parallelism
+ considered simpler, more like traditional programming
  - "if you want to access something, simply name it"
– no support for expressing locality/affinity; limits scalability
– bugs can be subtle, difficult to track down (race conditions)
– tend to require complex memory consistency models



Message Passing Programming Models

e.g., MPI

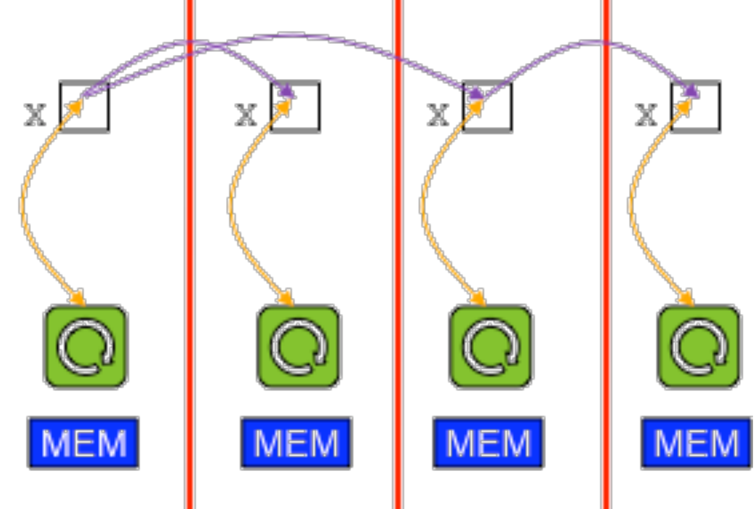
- + a more constrained model; can only access local data
- + runs on most large-scale parallel platforms
  - and for many of them, can achieve near-optimal performance
- + is *relatively* easy to implement
- + can serve as a strong foundation for higher-level models
- + users have been able to get real work done with it



Message Passing Programming Models

e.g., MPI

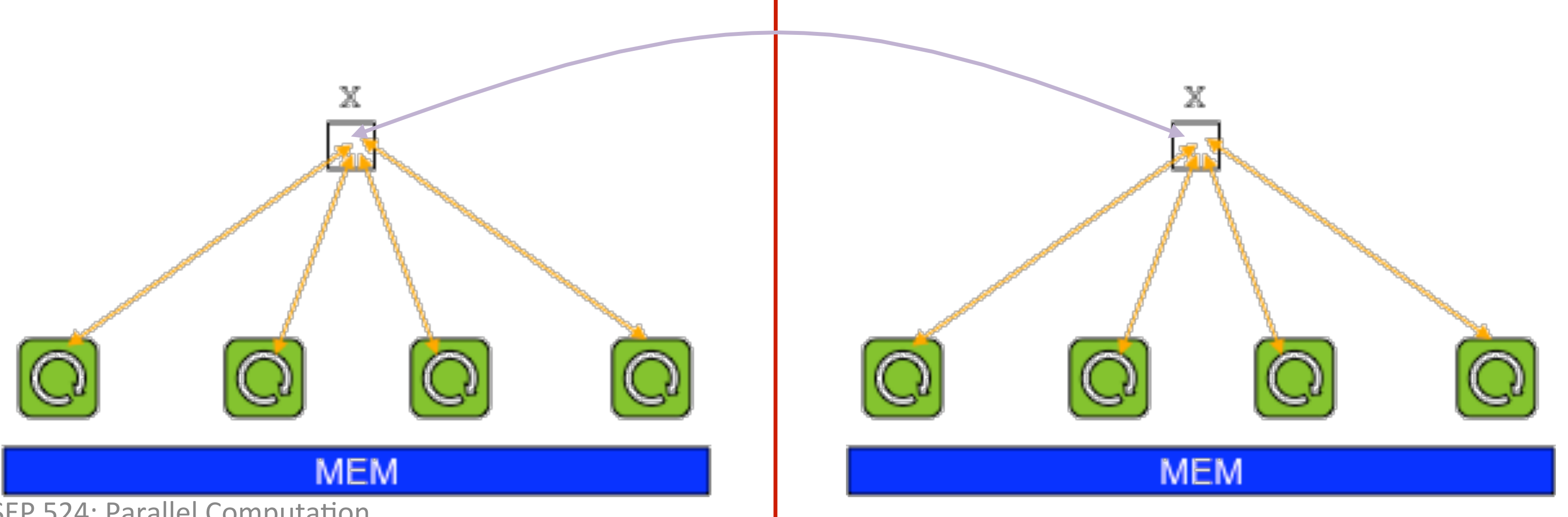
- communication must be used to get copies of remote data
  - tends to reveal too much about *how* to transfer data, not simply *what*
- only supports "cooperating executable"-level parallelism
- couples data transfer and synchronization
- has frustrating classes of bugs of its own
  - e.g., mismatches between sends/recvs, buffer overflows, etc.



Hybrid Programming Models

e.g., MPI+OpenMP/Pthreads/CUDA, UPC+OpenMP, …

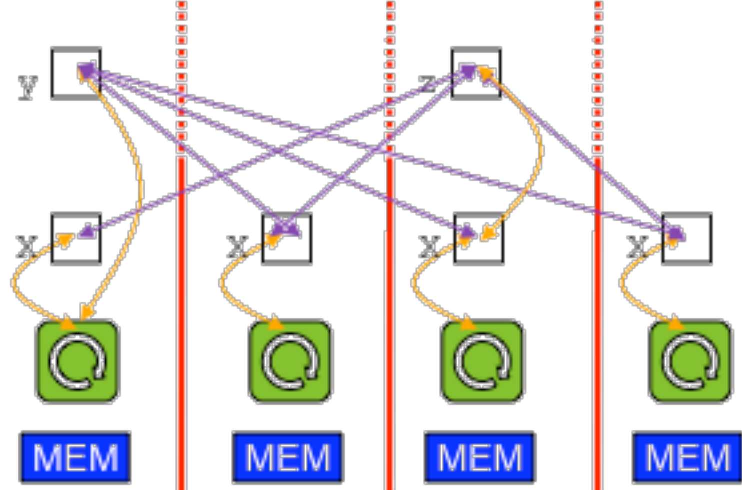
+ supports a division of labor: each handles what it does best
+ permits overheads to be amortized across processor cores, as compared to using MPI alone
– requires multiple notations to express a single logical parallel algorithm, each with its own distinct semantics



Traditional PGAS Models

e.g., Co-Array Fortran (CAF), Unified Parallel C (UPC)

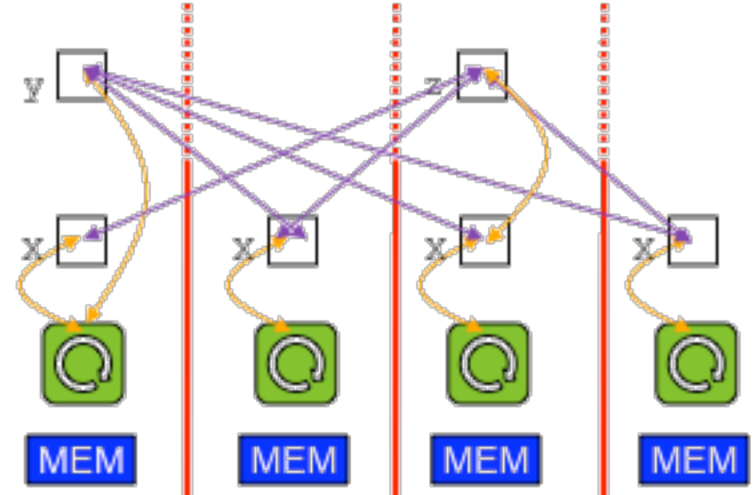
+ support a shared namespace, like shared-memory
+ support a strong sense of ownership and locality
  - each variable is stored in a particular memory segment
  - tasks can access any visible variable, local or remote
  - local variables are cheaper to access than remote ones
+ implicit communication eases user burden; permits compiler to use best mechanisms available



Traditional PGAS Models

e.g., Co-Array Fortran (CAF), Unified Parallel C (UPC)

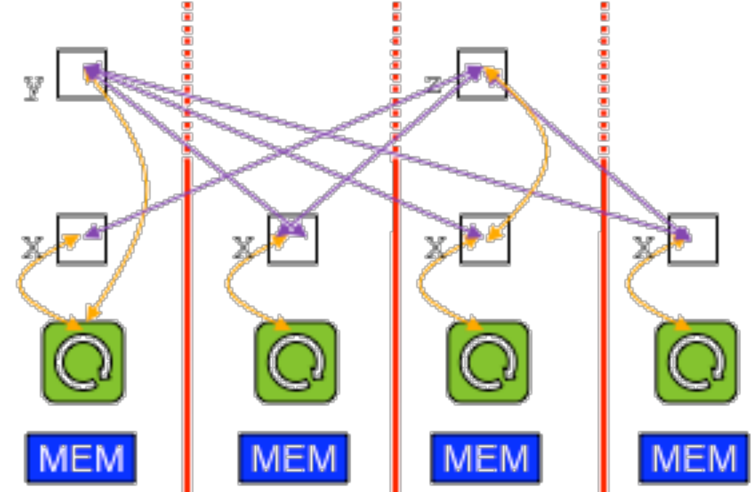
- restricted to SPMD programming and execution models
- data structures not as flexible/rich as one might like
- retain many of the downsides of shared-memory
  - error cases, memory consistency models



Next-Generation PGAS Models

e.g., Chapel (possibly X10, Fortress)

- \+ breaks out of SPMD mold via global multithreading
- \+ richer set of distributed data structures
- – retains many of the downsides of shared-memory
  - error cases, memory consistency models



Categorizing Based on Features/Capabilities

	MPI	Chapel	UPC	CAF
data parallelism	no (SPMD only)	yes (forall, whole-array ops)	yes (upc_forall)	no (SPMD only)
may tasks	no	yes	no	no
must tasks	no	yes	no	no
SPMD	yes	optionally	yes	yes
barriers	yes	no (not yet)	yes	yes
reductions	yes	yes	yes (library)	yes
scans	yes	yes	?	?
locks	no	sync vars	yes (library)	yes
incr. par.	no	so-so –to- yes	no	no
incr. dist.	no	yes	no	no
dist. mem.	yes	yes	yes	yes
comm. visible?	yes	no	no	yes
data-race-free?	yes	no	no	no

Categorizing Based on Features/Capabilities

	MPI	Chapel	UPC	CAF
data parallelism	no (SPMD only)	yes (forall, whole-array ops)	yes (upc_forall)	no (SPMD only)
may tasks	no	yes	no	no
must tasks	no	yes	no	no
SPMD	yes	optionally	yes	yes
barriers	yes	no (not yet)	yes	yes
reductions	yes	yes	yes (library)	yes
scans	yes	yes	?	?
locks	no	sync vars	yes (library)	yes
incr. par.	no	so-so –to- yes	no	no
incr. dist.	no	yes	no	no
dist. mem.	yes	yes	yes	yes
comm. visible?	yes	no	no	yes
data-race-free?	yes	no	no	no

Global-View vs. SPMD (Chapel vs. MPI/UPC/CAF)

Incremental Distribution and Convenience…

- “Let me change this shared-memory program to distributed-”
  - analogous to incremental parallelism in OpenMP/Chapel
- compare: “Let’s write out this array” in Chapel vs. MPI

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
         D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  [(i,j) in D] Temp[i,j] = (A[i-1,j] + A[i+1,j]
                          + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

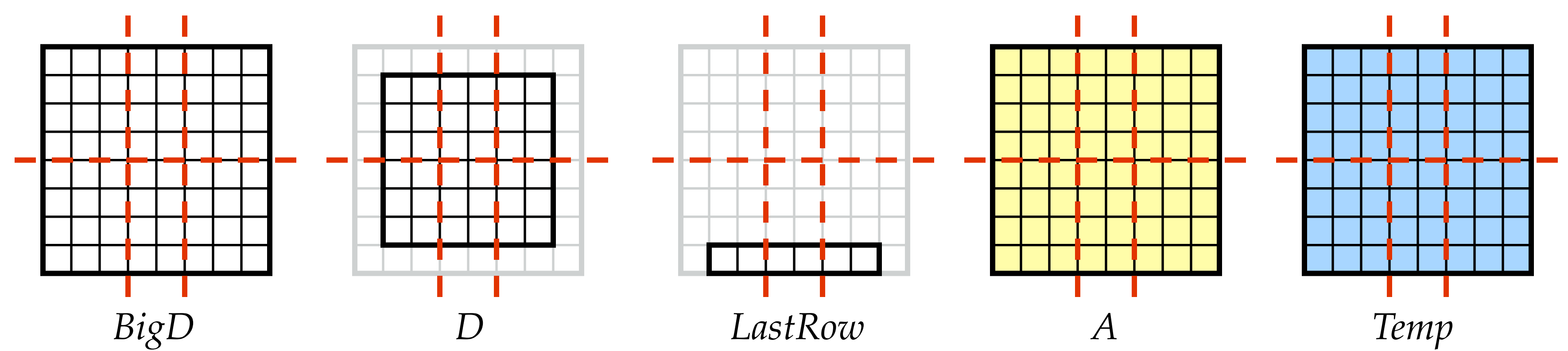
Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1} dmapped Block(…),
         D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
```

With this change, same code runs in a distributed manner

Domain distribution maps indices to *locales*

- ⇒ decomposition of arrays & default mapping of iterations to locales
- Subdomains inherit parent domain's distribution



Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1} dmapped Block(…),
         D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  [(i,j) in D] Temp[i,j] = (A[i-1,j] + A[i+1,j]
                          + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Global-View vs. SPMD (Chapel vs. MPI/CAF)

Incremental Distribution and Convenience…

- “Let me change this shared-memory program to distributed-”
  - analogous to incremental parallelism in OpenMP/Chapel
- compare: “Let’s write out this array” in Chapel vs. MPI

… vs. Incremental Performance Tuning and Data Races

- Smith-Waterman: “Why is my performance bad? … Oh, all accesses to my sequences go to locale 0”
- “Oops, did I access that before you were ready for me to?”

Recap: Why did we use Chapel?

A: “Because Brad made us”

- Well, yes, but…

Recap: Why did we use Chapel?

- Because it's the one language that naturally supports all the concepts we wanted to study
  - data vs. task (may + must) vs. wavefront vs. nested vs. SPMD
  - shared memory vs. PGAS vs. message passing
  - desktop vs. cluster vs. large-scale
  - synchronization, deadlock, livelock
  - memory consistency model, data races
  - reductions, scans, stencils, …
  - embarrassingly parallel, searches, histogram, bounded buffer, collective and global-view reductions, full scans, atomic operations, 9-point stencil, Mandelbrot, Smith-Waterman, …

Chapel and Education

- If I were teaching parallel programming, I'd want to cover:
  - data parallelism
  - task parallelism
  - concurrency
  - synchronization
  - locality/affinity
  - deadlock, livelock, and other pitfalls
  - performance tuning
  - …
- I don't think there's been a good language out there…
  - for teaching *all* of these things
  - for teaching some of these things well at all
  - ***until now:*** We believe Chapel can potentially play a crucial role here

(see http://chapel.cray.com/education.html for more information)

The Parallel Programmer's Toolbox

Parallel Algorithms

- You can't learn them all
  - though studying a number of common ones is useful
- Instead, focus on what to reason about:
  - **parallelism:**
    - what is amenable to parallelization?
    - what type? data? task? (may vs. must?) pipelined? multiple types?
    - how much parallelism is appropriate?
  - **locality:**
    - how should I distribute my data?
      - goal: minimize communication, maximize locality
    - how should I store data locally?
      - goal: minimize interference of intra-node parallelism (e.g., false sharing)

Parallel Algorithm Building Blocks

- Embarrassingly Parallel
- Broadcasts
- Reductions
- Scans
- Stencils
- Wavefronts/Pipelining
- All-to-alls / All-to-many
  - permutations/redistributions
  - scatters/gathers

Implementing All-to-all Communications

- Lots of potential techniques
  - compute everything that goes to each processor?
  - fill buckets per processor and send when full?
  - send off an element at a time?

- Best solution depends a lot on network, algorithm
  - a nice thing to leave to the language/library if you can

Introduction to FFT

Given: m-element vector z of complex numbers (where $m = 2^n$)

Compute: 1D Discrete Fourier Transform of z

Pictorially (using a radix-4 algorithm):

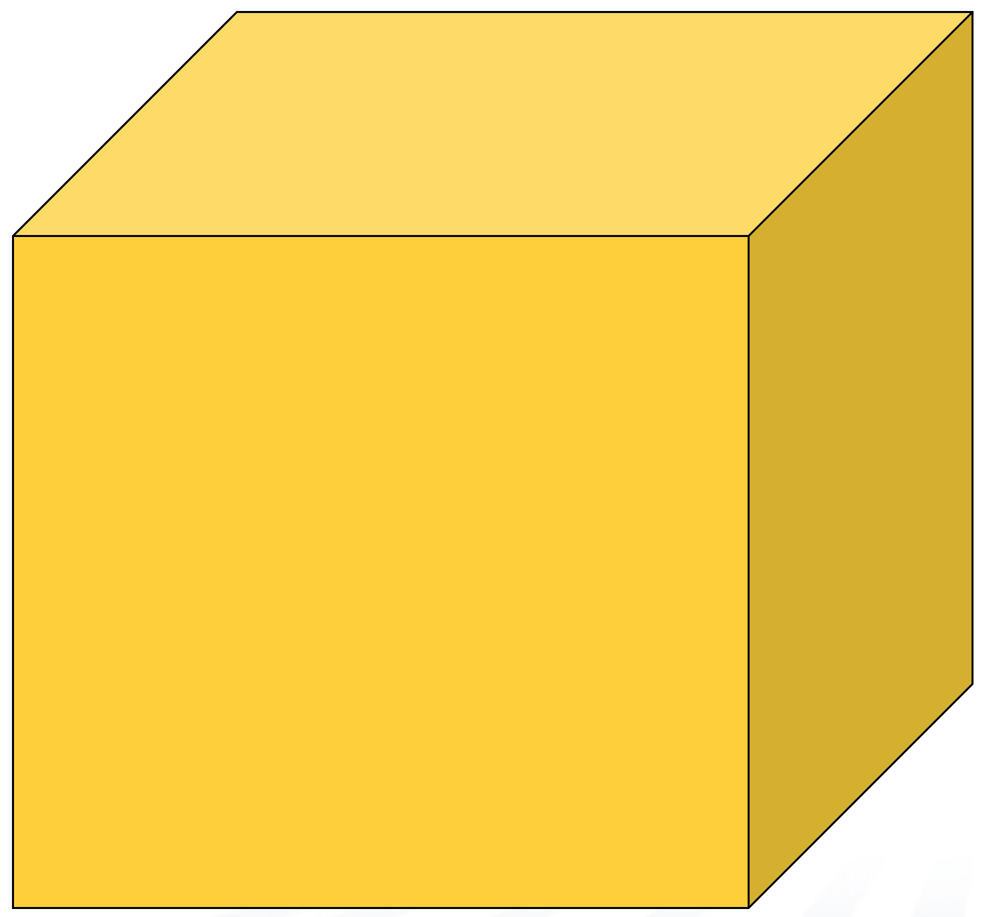
FFT Using Distributed Memory

How to best distribute the data?

- Initially, Block is ideal
- Then (~halfway?), Cyclic is ideal (if using 2^k processes/locales)
- So… compute half, redistribute everything, then compute next half

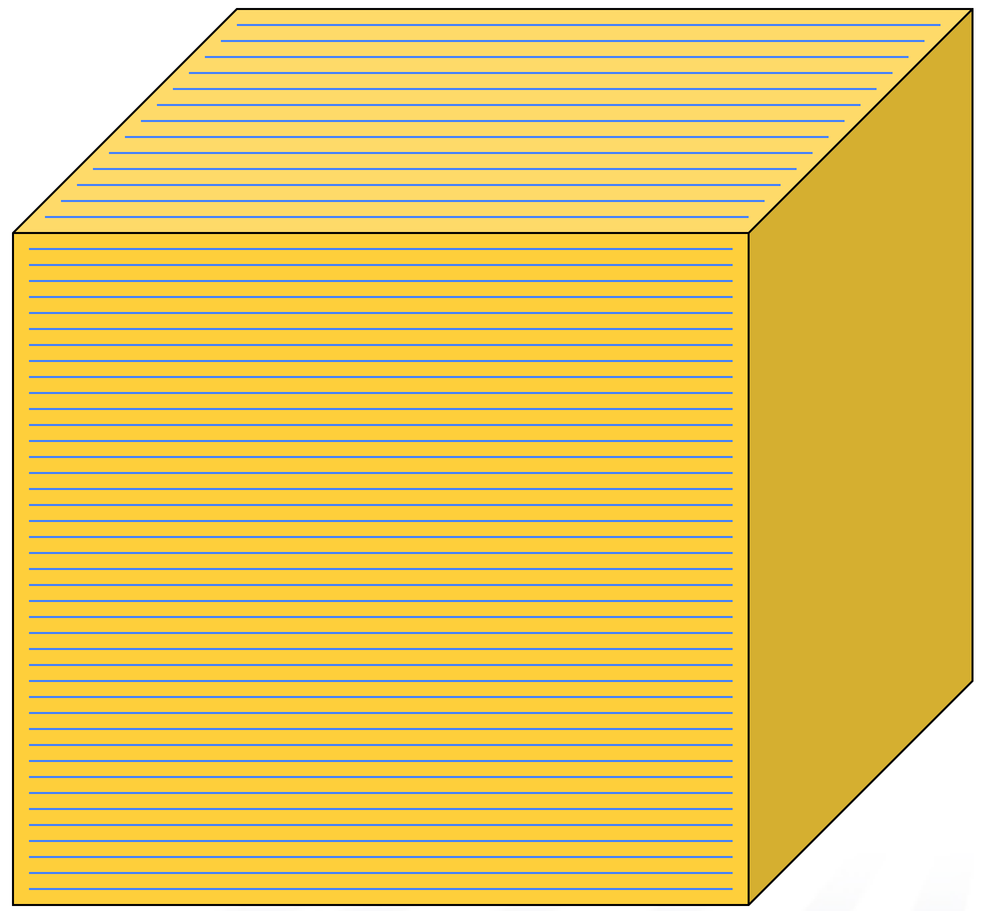
3D FFT

Far more important than a single 1D FFT in practice



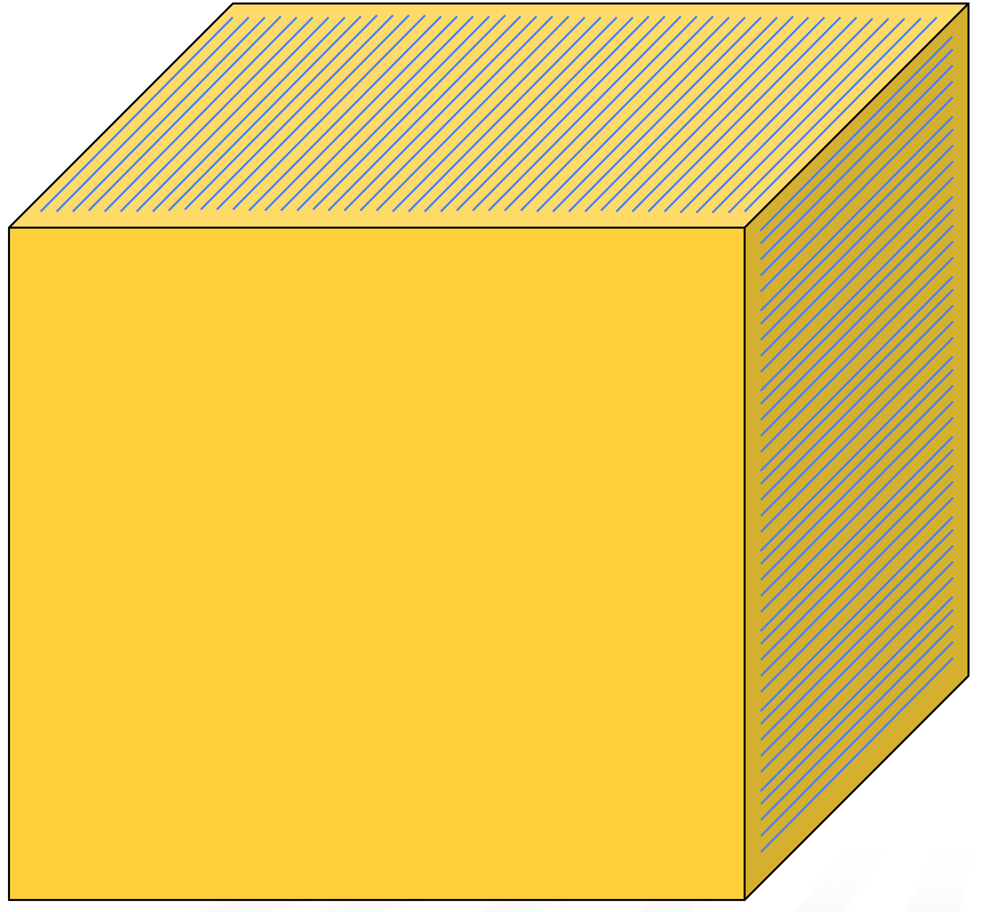
3D FFT

First compute 1D FFTs along one axis



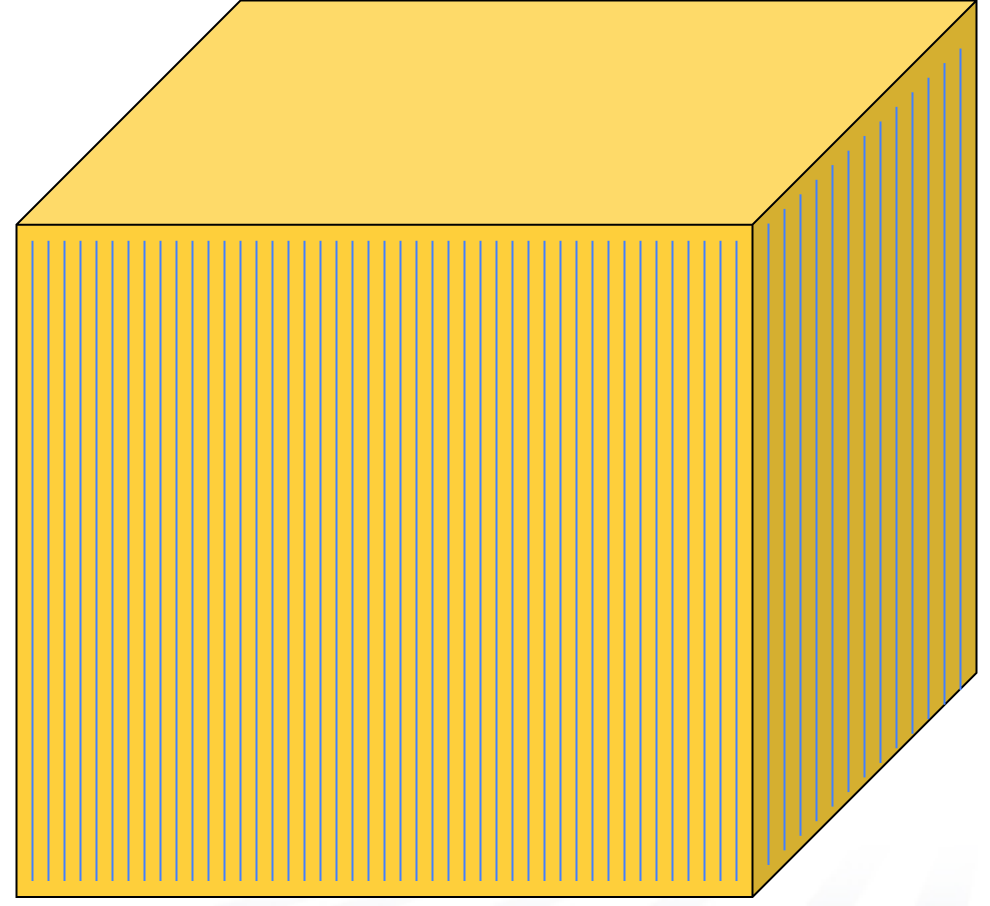
3D FFT

…then the second…



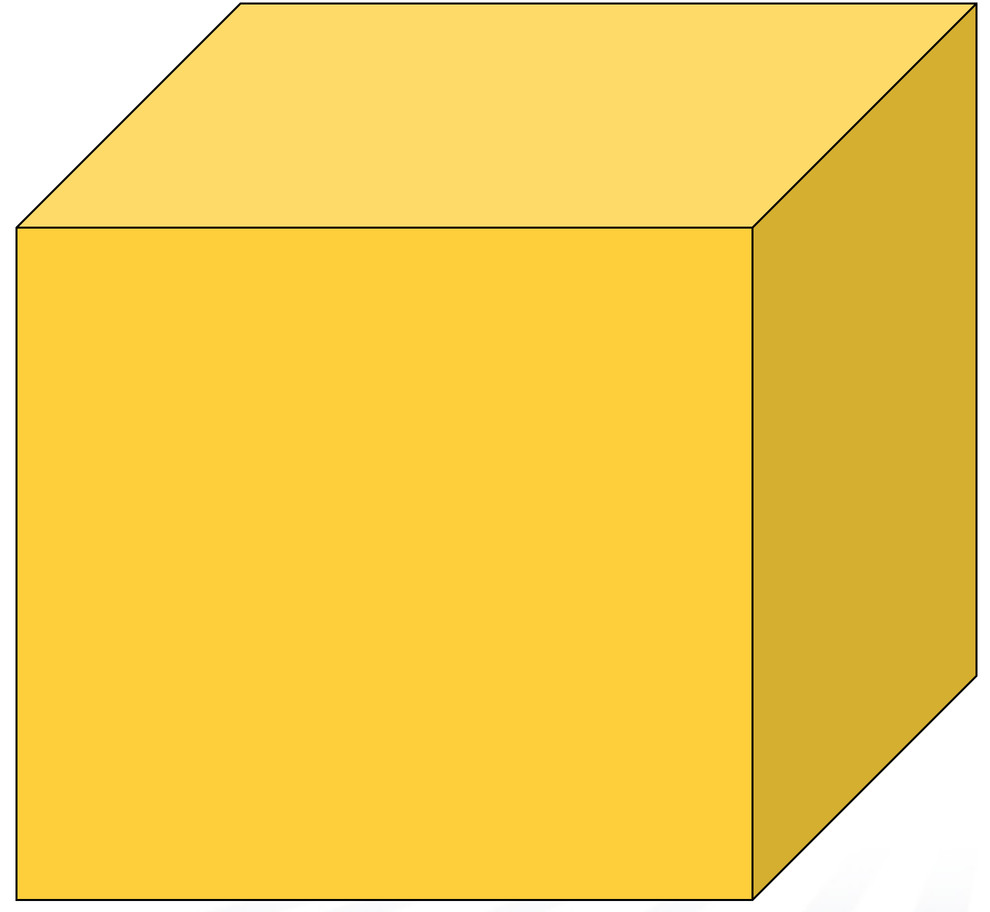
3D FFT

…then the third.



3D FFT

Q: How to distribute the data?



3D FFT

A: use a 1D block distribution:

3D FFT

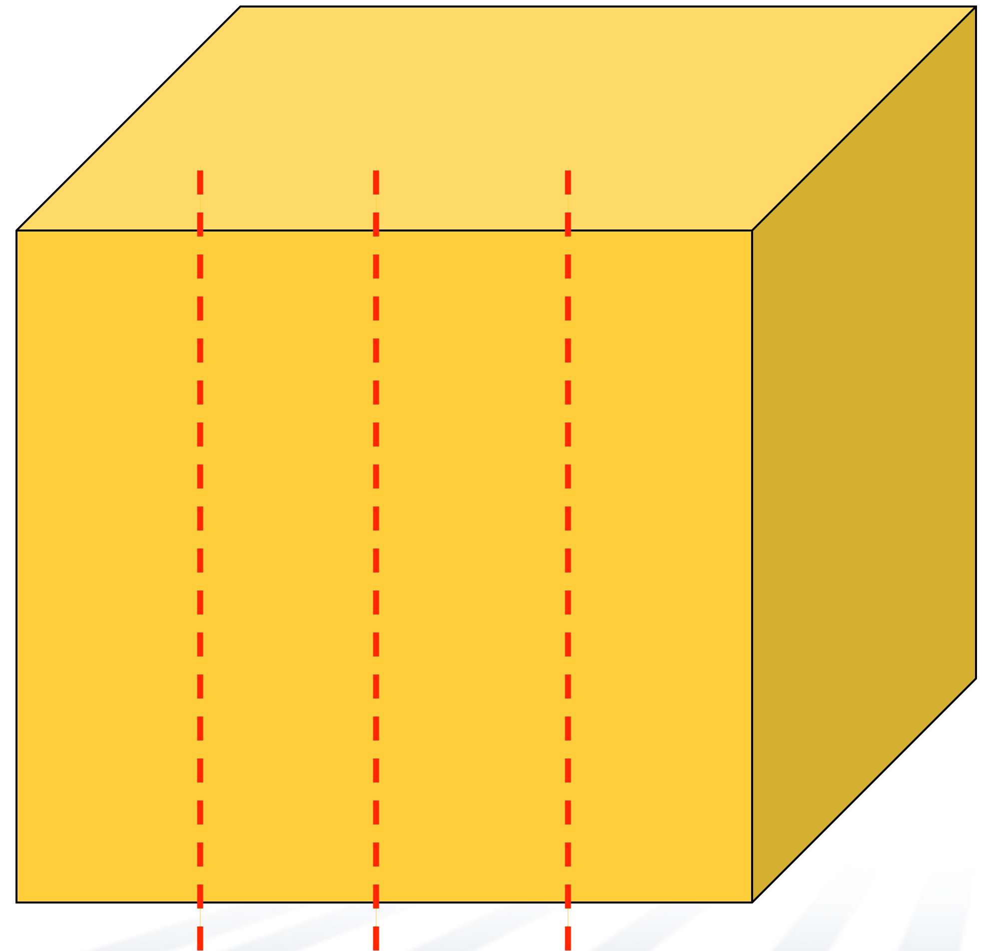
- compute 2 dimensions of 1D local FFTs

3D FFT

- compute 2 dimensions of 1D local FFTs

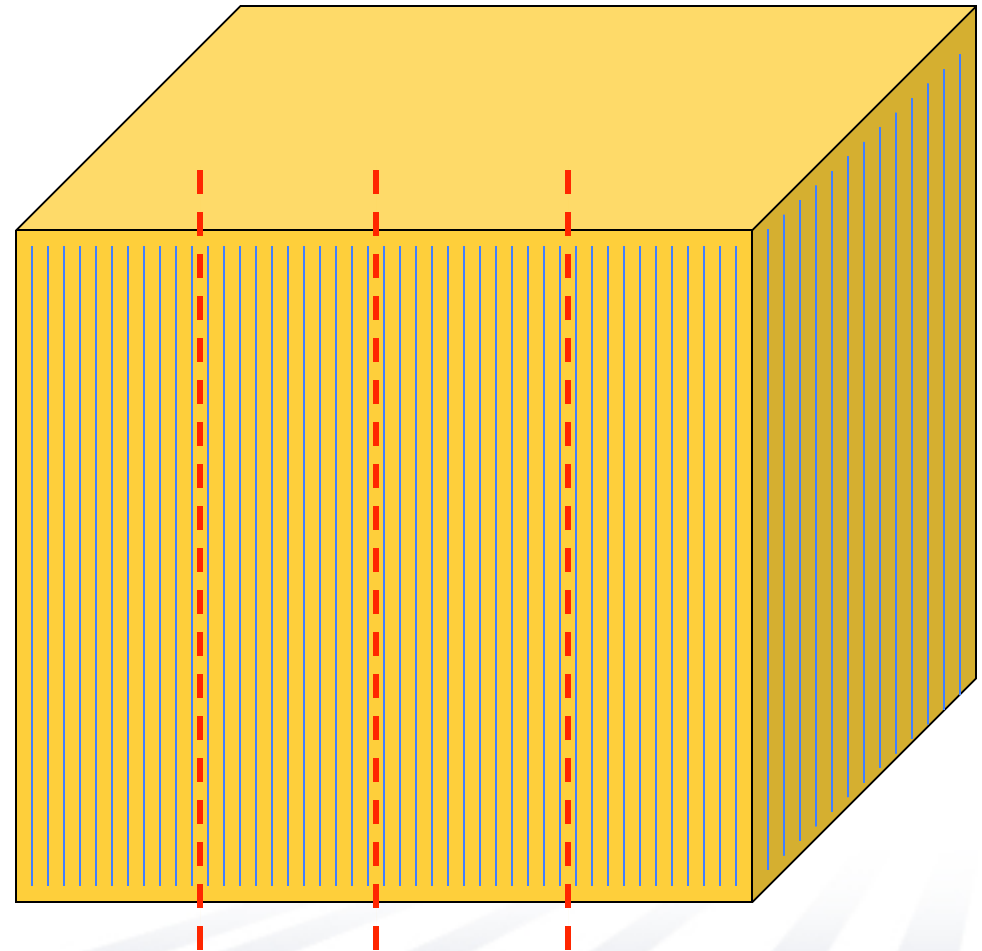
3D FFT

- then do a global transpose/redistribution



3D FFT

- and do the final dimension locally



Software Transactional Memory (STM)

Atomic

An *easier-to-use* and *harder-to-implement* primitive

```
void deposit(int x){
synchronized(this){
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

```
void deposit(int x){
atomic {
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

lock acquire/release

(behave as if)
no interleaved computation

Proposed Atomic Transactions Concept in Chapel
(joint work with U. Notre Dame)

- Syntax

```
atomic-statement:
  atomic stmt
```

- Semantics
  - Executes stmt so it appears as a single operation
  - No other task sees a partial result
- Examples

```
// safe increment
atomic A[i] += 1;
```

```
// doubly linked list insertion
atomic {
  newNode.next = node;
  newNode.prev = node.prev;
  node.prev.next = newNode;
  node.prev = newNode;
}
```

Code evolution

```
void deposit(…)  { synchronized(this) { … }}
void withdraw(…) { synchronized(this) { … }}
int  balance(…)  { synchronized(this) { … }}
```

Code evolution

```
void deposit(…)  { synchronized(this) { … }}
void withdraw(…) { synchronized(this) { … }}
int  balance(…)  { synchronized(this) { … }}

void transfer(Acct from, int amt) {

   if(from.balance()>=amt && amt < maxXfer) {
     from.withdraw(amt);
     this.deposit(amt);
   }

}
```

Code evolution

```
void deposit(…)  { synchronized(this) { … }}
void withdraw(…) { synchronized(this) { … }}
int  balance(…)  { synchronized(this) { … }}

void transfer(Acct from, int amt) {
  synchronized(this) {
   //race
   if(from.balance()>=amt && amt < maxXfer) {
     from.withdraw(amt);
     this.deposit(amt);
   }
  }
}
```

Code evolution

```
void deposit(…)  { synchronized(this) { … }}
void withdraw(…) { synchronized(this) { … }}
int  balance(…)  { synchronized(this) { … }}

void transfer(Acct from, int amt) {
  synchronized(this) {
  synchronized(from) { //deadlock
   if(from.balance()>=amt && amt < maxXfer) {
     from.withdraw(amt);
     this.deposit(amt);
   }
  }}
}
```

Code evolution

```
void deposit(…)  { atomic { … }}
void withdraw(…) { atomic { … }}
int  balance(…)  { atomic { … }}
void transfer(Acct from, int amt) {
  atomic {
    //correct and parallelism-preserving!
   if(from.balance()>=amt && amt < maxXfer) {
     from.withdraw(amt);
     this.deposit(amt);
   }
  }
}
```

Track what you touch

High-level ideas:

- Optimistic: proceed assuming conflicts unlikely
  - contrast with pessimistic locking: take lock "just in case"
- Maintain transaction's *read set*
  - so you can *abort* if another thread writes to it before you *commit* (detect *conflicts*)
- Maintain transaction's *write set*
  - again for *conflicts*
  - also to *commit* or *abort* correctly

Writing

- Two approaches to writes
  - *Eager update*
    - update in place, "own until commit" to prevent access by others
    - log previous value; undo update if abort
    - if owned by another thread, abort to prevent *deadlock* (livelock is possible)
  - *Lazy update*
    - write to private buffer
    - reads must check buffer
    - abort is trivial
    - commit is fancy to ensure "all at once"

Reading

- Reads
  - May read an *inconsistent value*
    - detect with version numbers and such
    - inconsistent read requires an abort

```
 initially x=0, y=0
atomic {            atomic {
  while(x!=y)         ++x;
     ;                ++y;
}                    }
```

Other Implementation Challenges

- I/O
- memory allocation/freeing
- Exceptions
- …more advanced STM concepts…

General strategy: track enough state that you can "undo" things…

Advantages

So `atomic` "sure feels better than locks"

But the crisp reasons I've seen are all (great) examples

- Account transfer from Flanagan et al
  - See also Java's **`StringBuffer append`**
- Double-ended queue from Herlihy

Double-ended queue

Operations

```
void enqueue_left(Object)
void enqueue_right(Object)
obj  dequeue_left()
obj  dequeue_right()
```

Correctness

- Behave like a queue, even when ≤ 2 elements
- Dequeuers wait if necessary, but can't "get lost"

Parallelism

- Access both ends in parallel, except when ≤ 1 elements (because ends overlap)

Good luck with that…

- One lock?
  - No parallelism
- Locks at each end?
  - Deadlock potential
  - Gets very complicated, etc.
- Waking blocked dequeuers?
  - Harder than it looks

Actual Solution

- A clean solution to this apparent "homework problem" would be a publishable result?
  - In fact it was: [Michael & Scott, PODC 96]

- So locks and condition variables are not a "natural methodology" for this problem

- Implementation with transactions is trivial
  - Wrap 4 operations written sequentially in `atomic`
    - With `retry` for dequeuing from empty queue
  - Correct and parallel

STM vs. HTM

- Because of my (Brad's) software-oriented bias/ background, I've focused on atomic statements from a software perspective

- HTM is a related line of research in which the hardware supports transactional memory concepts

- Hybrid approaches are also pursued which combined SW- and HW-based approaches

For More Information

- "Director's Cut" version of these slides:

  http://homes.cs.washington.edu/~djg/slides/grossman_bangalore08.ppt

- Repository of TM-related publications/work:

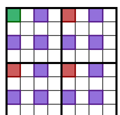
  http://www.cs.wisc.edu/trans-memory

- STM work for Chapel (*key challenge* = distributed memory):
  - *A Scalable Implementation of Language-Based Software Transactional Memory for Distributed Memory Systems*
    http://ft.ornl.gov/pubs-archive/chplstm1-2011-tr.pdf
  - *Software Transactional Memory for Large-Scale Clusters*
    http://www.cs.cmu.edu/~rbocchin/Publications_files/Bocchino-PPoPP-2008.pdf

So… Where are my atomics?

- Has not yet made it from research to production
- Challenges to adoption:
  - semantic questions/challenges
  - performance relative to locks
  - complete, production-grade implementation
- Two prevailing, but opposing, views:
  - STM is like GC in the 80's… en route, just be patient
  - STM is unlikely to ever be adoptable

Back to the Stencil Ramp



MG's Guts (*mg3P*)

rprj3

rprj3

rprj3

psinv

interp

resid

psinv

interp

resid

psinv

interp

resid

psinv

Continuing the Stencil Ramp: the Fast Multipole Method

(switch slide decks)

Wrap-up

From the Course Description…

- **styles of parallelism**
  - data-parallel
  - task-parallel
  - concurrency
  - pipelined parallelism
  - nested parallelism
- **abstract programming models**
  - shared memory
  - Single Program, Multiple Data (SPMD)
  - message passing
  - Partitioned Global Address Space (PGAS)
- **architectural implications**
  - shared vs. distributed memory
  - multicore processors and accelerators
  - networks
  - caches and memory
- **programming issues and hazards**
  - synchronization
  - memory consistency
  - race conditions
  - deadlock and livelock
- **performance tuning**
  - scalability
  - locality
  - communication
  - scalar concerns
- **programming languages and notations**
  - OpenMP
  - MPI
  - UPC
  - Chapel
  - CUDA/OpenCL/OpenACC (?)
- **algorithms and patterns**
  - reductions and scans
  - stencils
  - graph algorithms
  - ...

Overall Goals

- Expose you to as much information about parallel computing as possible within the allotted time
  - foundations
  - best practices
  - recent trends
- Teach you principles of parallel programming
- Give you practical parallel programming experience
  - using adopted programming models
    - Pthreads, OpenMP, MPI, ~~UPC~~
  - using Chapel as an idealized parallel language

Course Content

Backbone: follow a progression of architectures and programming models from shared memory to distributed memory

Along the way: cover common parallel algorithms/ patterns, hazards, grab-bag topics, …

Thank you!

For being a particularly attentive and inquisitive class

For your consistent punctuality

For lots of good discussions, in class, after class, online

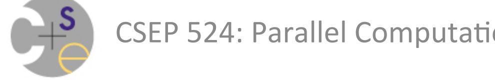
Surveys

- I value your feedback heavily (kudos & criticisms)
- please put time & thought into it

Survey: Extra Questions (for back of Scantron)

1. Value of Pthreads programming within this class
2. Enjoyment of programming in Pthreads in general
3. Value of OpenMP programming within this class
4. Enjoyment of programming in OpenMP in general
5. Value of MPI programming within this class
6. Enjoyment of programming in MPI in general
7. Value of Chapel programming within this class
8. Enjoyment of programming in Chapel in general

A	B	C	D	E	F	G
Excellent			eh…			Very Poor



Bonus Slides

Unstructured Stencils:
the Finite Element Method (FEM)

Finite Element Meshes

Image source: http://public.cranfield.ac.uk/ab051154/FEMEC/femec.html

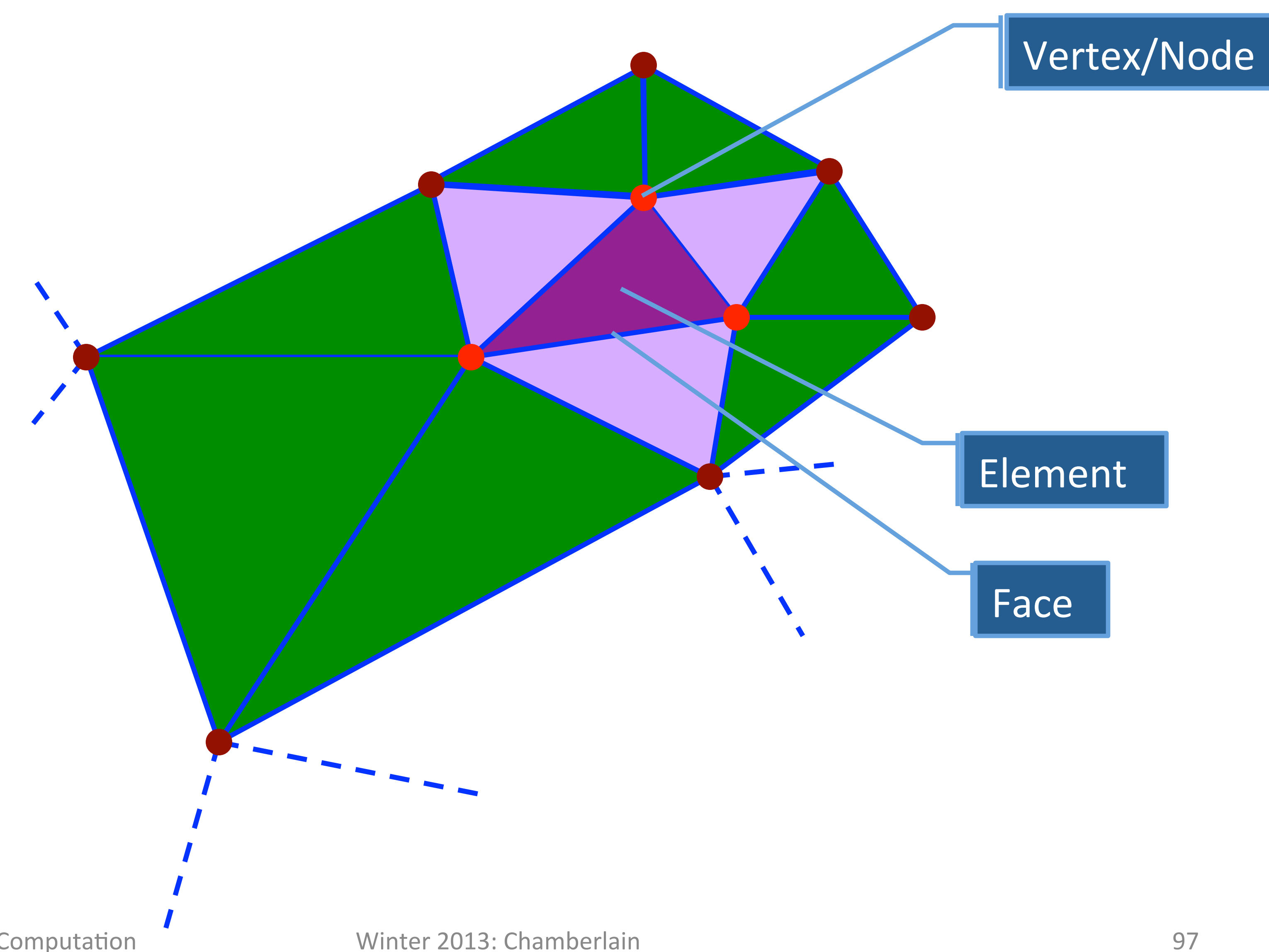
Finite Element Meshes

Image source: http://geuz.org/gmsh/gallery/bike.png

Finite Element Meshes

Image source: http://ceeserver.cee.cornell.edu/pk285/cee4720-6720_2010.html

Finite Element Mesh Terminology



FEM Declarations

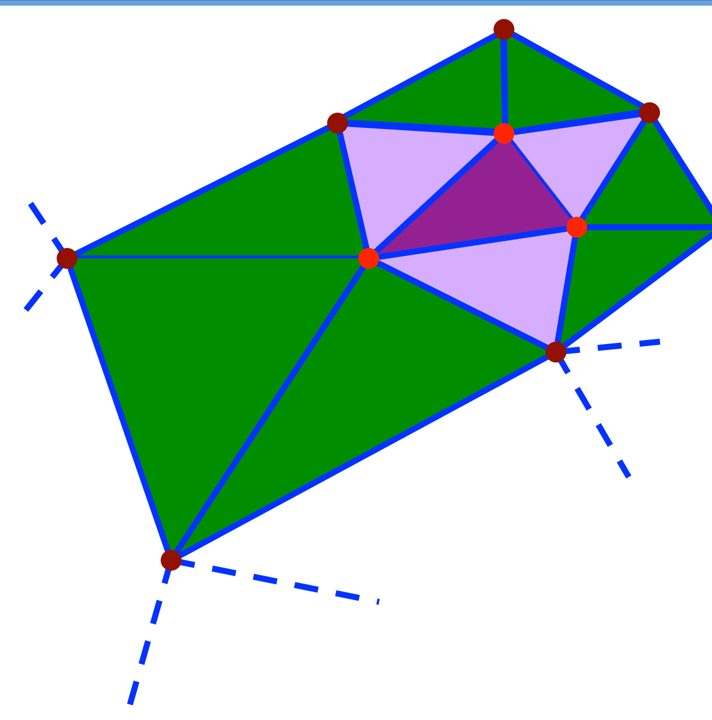
```
config param numdims = 2;
const facesPerElem = numdims+1,
      vertsPerFace = 3,
      vertsPerElem = numdims+1;

var Elements: domain(…) = …,
    Faces: domain(…) = …,
    Vertices: domain(…) = …;

type element = index(Elements),
     face = index(Faces),
     vertex = index(Vertices);

var elementFaces: [Elements] [1..facesPerElem] face,
    elemVertices: [Elements] [1..vertsPerElem] vertex,
    faceVertices: [Faces] [1..vertsPerFace] vertex;
```

FEM Computation

- Sample Idioms:

```
var a: [Vertices] atomic real;
var b, c, f: [Vertices] real;
var p: [1..2] [Vertices] real;

proc PoissonComputeA() {
  forall e in Elements {
    const c = 0.10 * volume(e);
    for v1 in elemVertices[e] {
      a[v1].add(c*f[v1]);
      for v2 in elemVertices[e] do
        if (v1 != v2) then
          a[v2].add(0.5*c*f[v2]);
} } }

proc computePressure(pressure: [Vertices] real) {
  pressure = (a - b) / c;
}
```

This loop nest is effectively a stencil on an unstructured grid

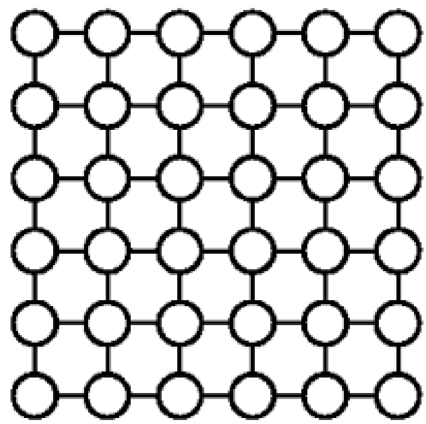


Distributing Unstructured Meshes

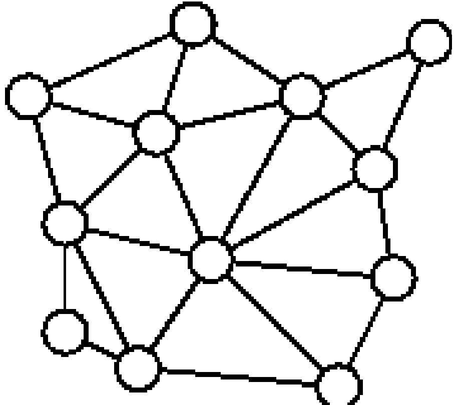
Workload Graphs

Workload graphs: a scheme for representing work

- Each vertex represents a unit of data
- Each edge represents a data dependency



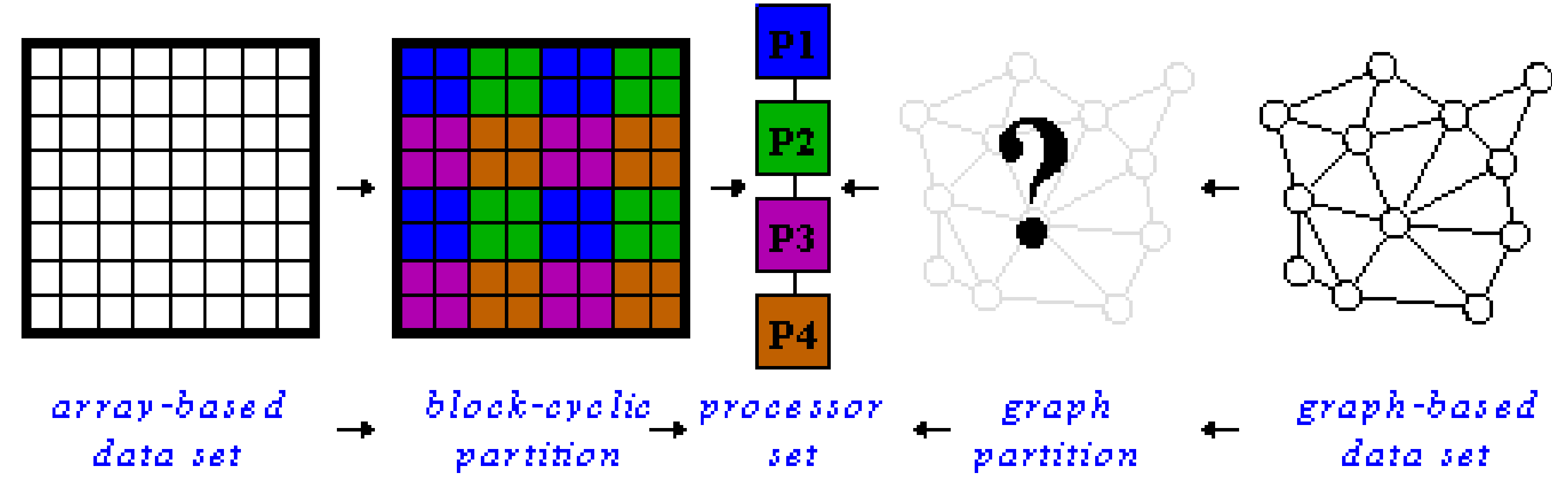
structured
workload graph



unstructured
workload graph

Workload Graph Distribution

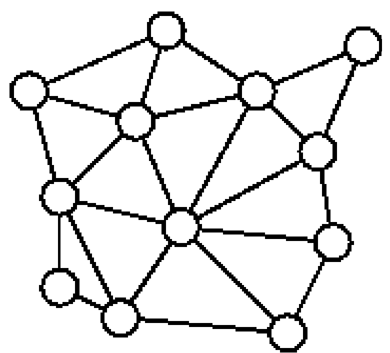
- Structured graphs have obvious distributions (*e.g., blocked, cyclic, block-cyclic*)
- Unstructured graphs do not



Graph Partitioning Problem

Given: Input graph, $G=(V,E)$;

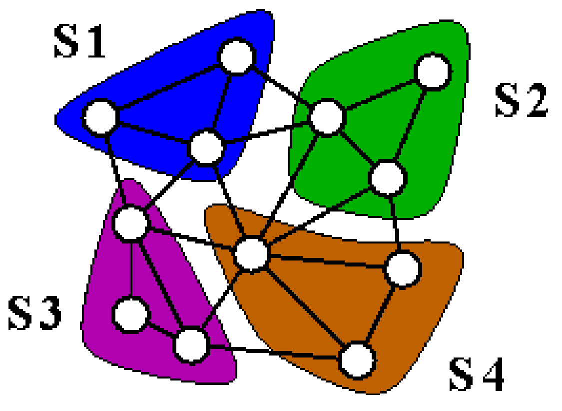
Number of partitions, p.

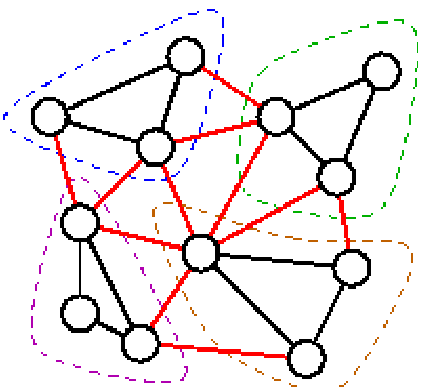


p = 4

Find: Disjoint vertex subsets, $S_1, S_2, \ldots, S_p$, that:

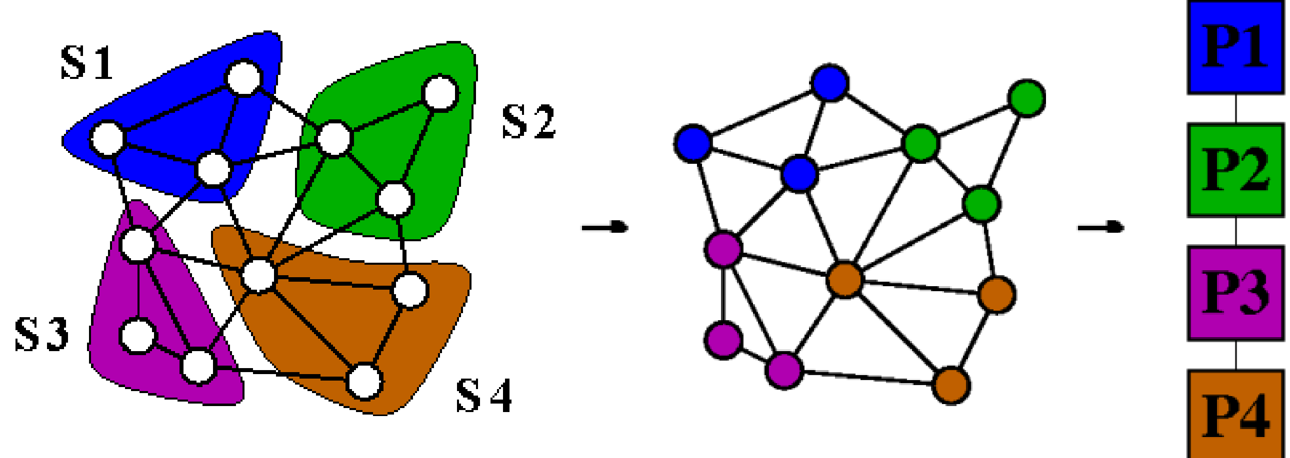
1) have equal numbers of vertices
2) minimize the number of cut edges





Mapping Workload Distribution to Graph Partitioning

- Good workload distributions:
  - balance computation between processors
  - minimize interprocessor communication
- Similar goals as graph partitioning, so…
- Apply graph partitioning to workload graphs



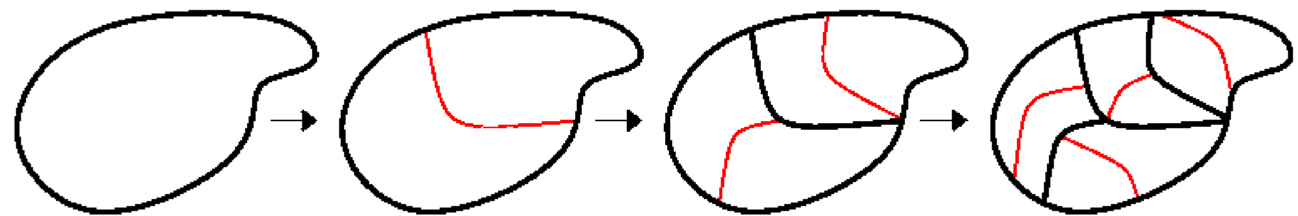
The Bad News

- Graph partitioning is NP-Complete
- Therefore all known algorithms are heuristics that approximate the optimal solution

(That said, there are some pretty good heuristics)

Recursive Bisection

- An approach to *p*-way partitioning that:
  - computes a 2-way partition for *G* (bisection)
  - recursively considers the resulting subgraphs



- Optimal graph bisection is still NP-Complete.
- However, its simplicity makes it widely used.

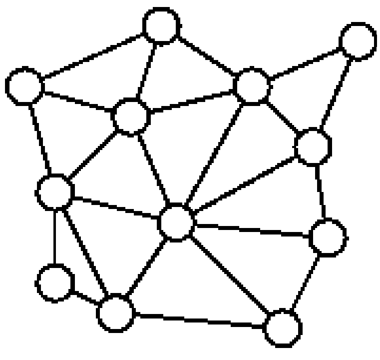
Geometric Algorithms

Geometric partitioning algorithms…

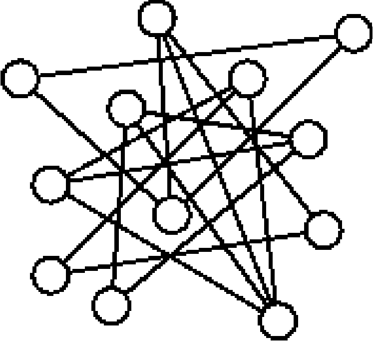
…utilize the vertices' geometric coordinates

…are generally very fast

…assume that *spatial proximity* implies *graph locality*



well-behaved



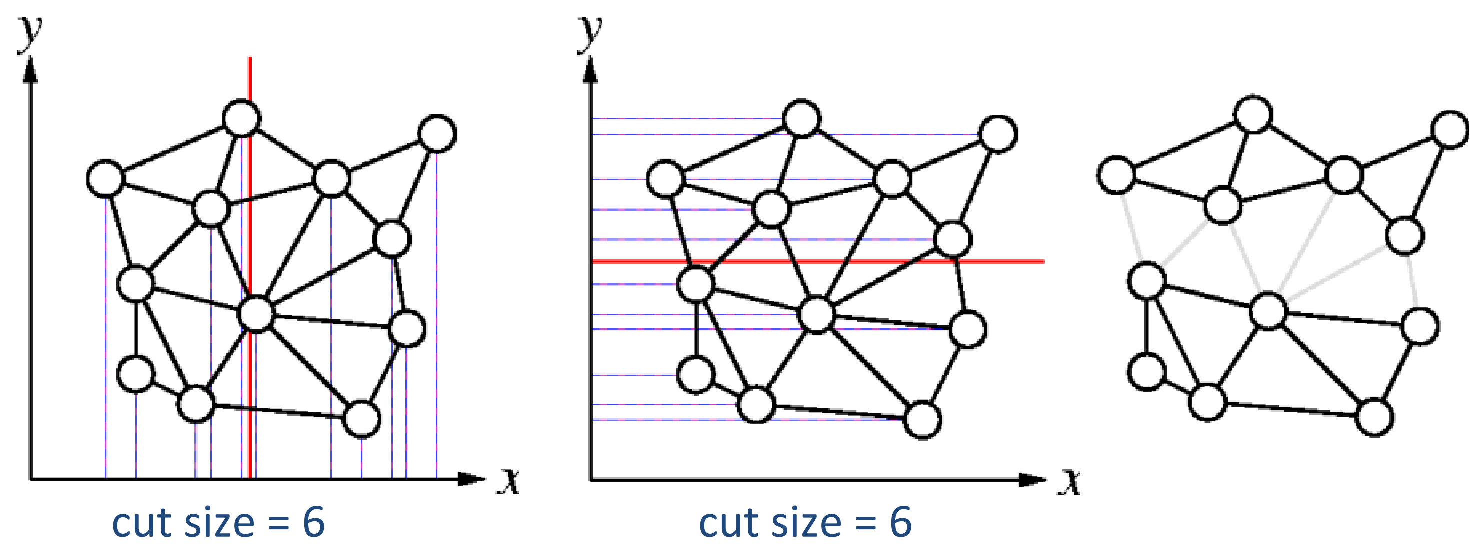
poorly-behaved

Note that many important graphs have no geometric coordinates – e.g., social networks!

A Simple Geometric Algorithm

Coordinate Bisection:

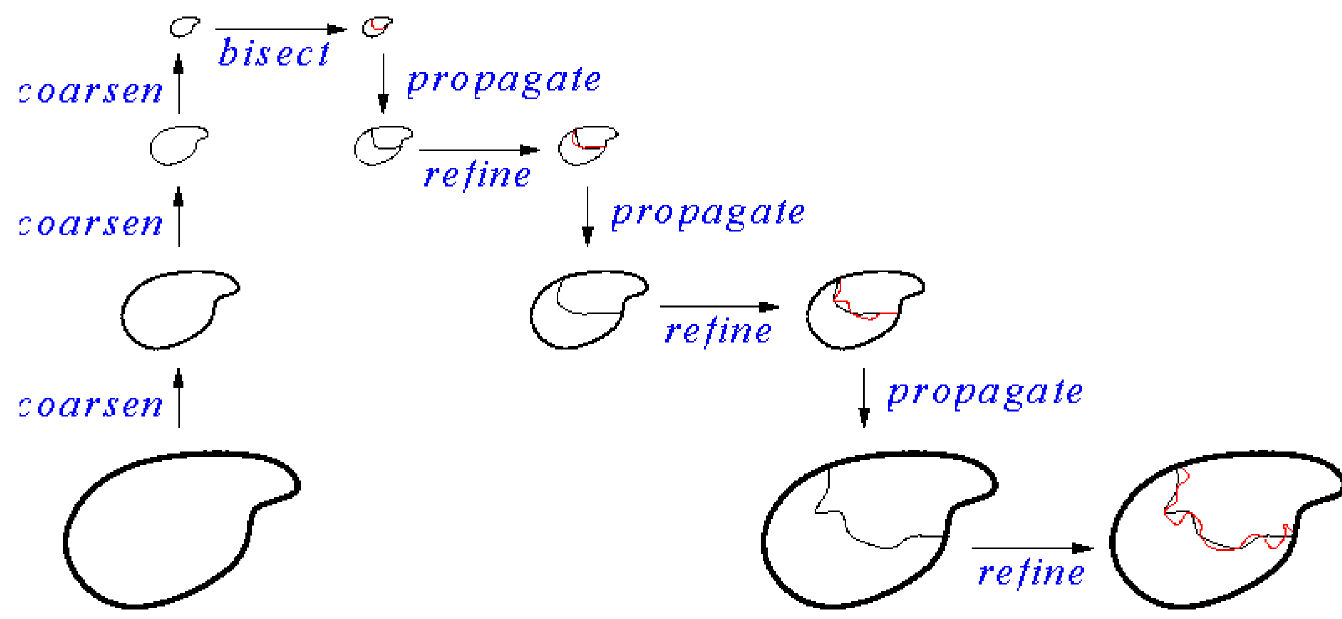
- find the median coordinate for each dimension
- construct a hyperplane at this location
- bisect using the hyperplane that cuts fewer edges



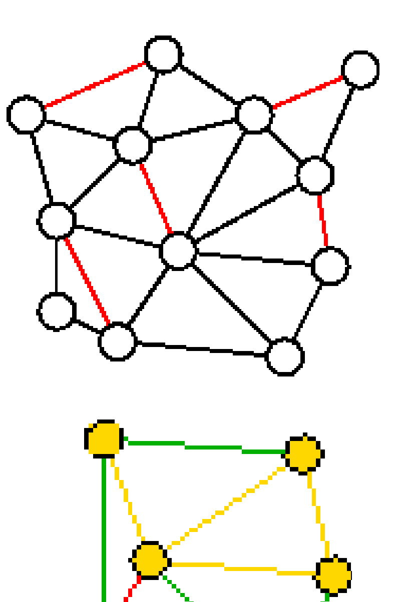
Multilevel Partitioning

Multilevel Framework:

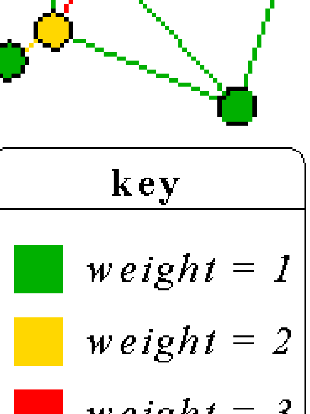
- Coarsen original graph
- Bisect coarsest graph
- Convert coarse bisection into bisection for original



Coarsening Step



- Compute a *maximal matching* for the graph.
- Collapse matched vertices into *multinodes;* combine incident edges.
- Use weights to retain information about the original graph:
  - vertex weights to represent vertex count
  - edge weights to represent combined edges

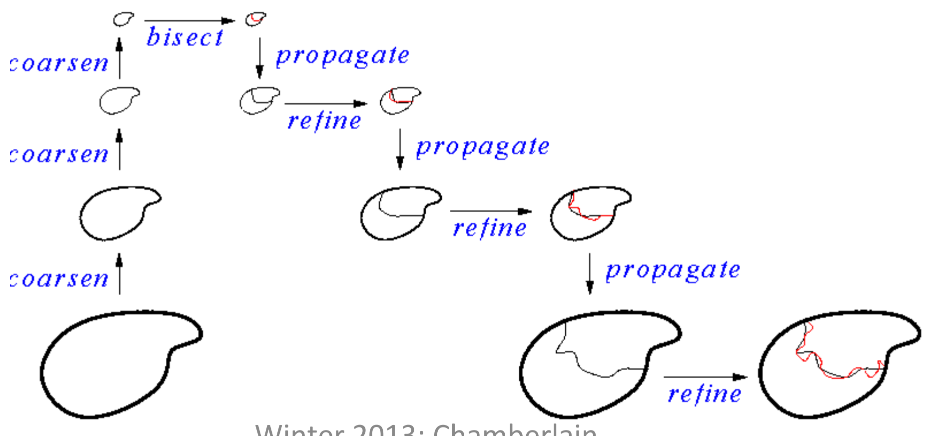


Other Multilevel Steps

Bisection Step: use any bisection algorithm that can handle weighted graphs (most can)

Propagation Step: expand multinodes and collapsed edges

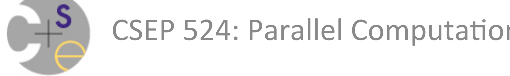
Refinement Step: use a local refinement algorithm such as Kernighan-Lin



Graph Partitioning Software

- METIS/ParMETIS:
  - http://glaros.dtc.umn.edu/gkhome/views/metis
- Chaco/Zoltan:
  - http://www.cs.sandia.gov/~bahendr/chaco.html
  - http://www.cs.sandia.gov/Zoltan/
- Jostle
  - http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle/

…and several others



HPF and ZPL:
Learn from the mistakes of your elders

High Performance Fortran (HPF)

HPF: an array-based data-parallel language

Developed by: HPF Forum (virtually everyone in HPC?)

Timeframe: 1990's

Target machines: 1990's HPC parallel platforms

Main concepts:

- directive-based extension to Fortran 90/95
- virtual processor grid
- distribution of arrays using standard set of distributions; alignment
- assertion of loop-level parallelism

Jacobi Iteration in HPF

```
REAL u(0:nx,0:ny), unew(0:nx,0:ny), f(0:nx,0:ny)

!HPF$ DISTRIBUTE u(BLOCK,*)
!HPF$ ALIGN WITH u(:,:) :: unew(:,:), f(:,:)

dx = 1.0/nx; dy = 1.0/ny; err = tol * 1e6

FORALL ( i=0:nx, j=0:ny )
  f(i,j) = -2*(dx*i)**2+2*dx*i-2*(dy*j)**2+2*dy*j
END FORALL

u = 0.0; unew = 0.0

DO WHILE (err > tol)
  FORALL ( i=1:nx-1, j=1:ny-1 ) &
    unew(i,j) = (u(i-1,j)+u(i+1,j)+u(i,j-1)+ &
                 u(i,j+1)+f(i,j))/4
  err = MAXVAL( ABS(unew-u) )
  u = unew
END DO
```

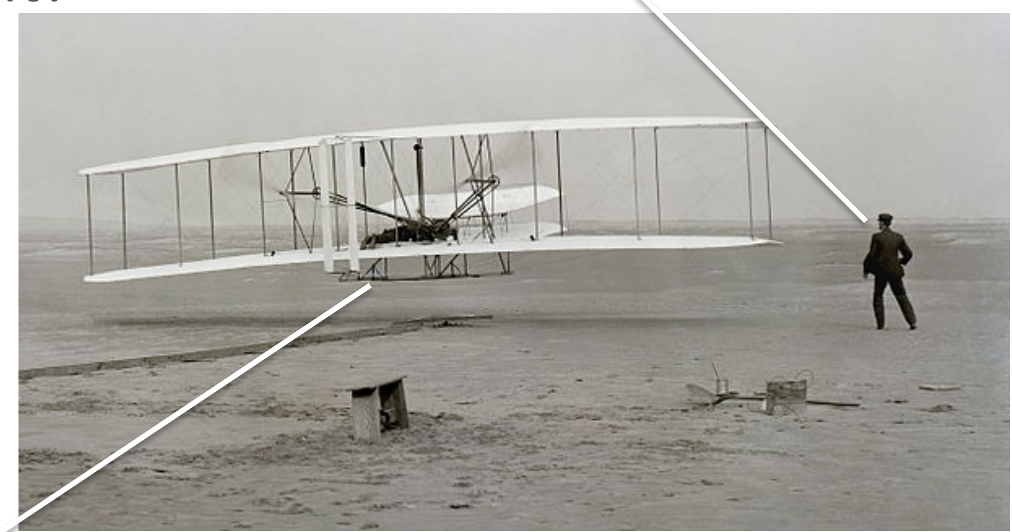
Jacobi Stencil in D-HPF (Rice)

```
      program jacobi
        integer N, m, t
        PARAMETER (N=1024)
        double precision a(N, N), b(N, N)

CHPF$   processors p(4,8)
CHPF$   template t(N,N)
CHPF$   align a(i,j) with t(i,j)
CHPF$   align b(i,j) with t(i,j)
CHPF$   distribute t(block,block) onto p

CHPF$    INDEPENDENT
       do j = 2, 1024 - 1
CHPF$    INDEPENDENT
          do i = 2, 1024 - 1
            a(i, j) = 0.25 * (b(i - 1, j) + b(i + 1, j) + b(i, j - 1) +
b(i, j + 1))
          enddo
        enddo
      end
```

A Common Question about Chapel

Q: Didn't we try this before with HPF?

Q': Orville, didn't Percy Pilcher die in *his* prototype powered aircraft?



A': No Wilbur, he died in a glider; and even if it had been in his prototype, that doesn't mean we're doomed to fail.

Q: How Can Chapel Succeed When HPF Failed?

A: Chapel has had the chance to learn from HPF's mistakes (and other languages' successes and failures)

- Why did HPF fail?
  - lack of sufficient performance soon enough
  - vagueness in execution/implementation model
  - only supported a single level of data parallelism, no task/nested
  - inability to drop to lower levels of control
  - fixed set of limited distributions on dense arrays
  - lacked richer data parallel abstractions
  - lacked an open source implementation
  - too Fortran-based for modern programmers
  - …?
- The failure of one language---even a well-funded, US-backed one---does not dictate the failure of all future languages

(For more on this topic see https://www.ieeetcsc.org/activities/blog/myths_about_scalable_parallel_programming_languages_part2)

ZPL

ZPL: a contemporary of HPF
- similar goals, but a very different approach

Developed by: University of Washington

Timeframe: 1991 – 2003 (can still download today)

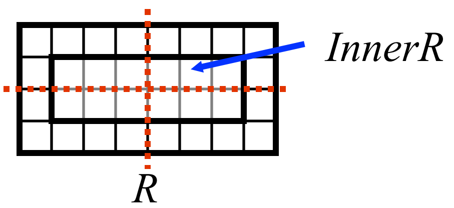
Target machines: 1990's HPC parallel platforms
- clusters of commodity processors
- clusters of SMPs
- custom parallel architectures
  - Cray T3E, KSR, SGI Origin, IBM SP2, Sun Enterprise, …

Main concepts:
- abstract machine model: CTA
- regions: first-class index sets
- WYSIWYG performance model

ZPL Concepts: Regions

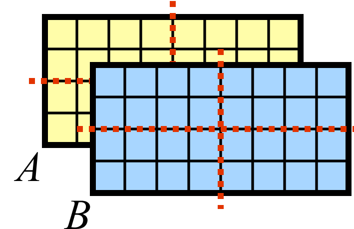
regions: distributed index sets…

```
region R      = [1..m, 1..n];
       InnerR = [2..m-1, 2..n-1];
```



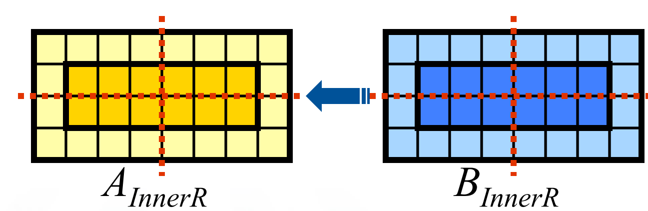
…used to declare distributed arrays…

```
var A, B: [R] real;
```



…and computation over distributed arrays

```
[InnerR] A = B;
```



ZPL Concepts: Array Operators

array operators: describe nontrivial array indexing

translation via *at operator* (@)

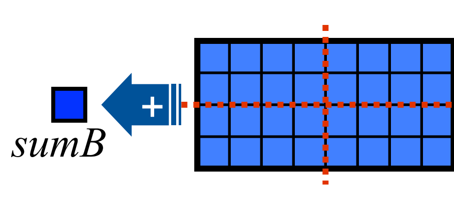
```
[InnerR] A = B@[0,1];
```

replication via *flood operator* (>>)

```
[R] A = >>[1, 1..n] B;
```

reduction via *reduction operator* (*op*<<)

```
[R] sumB = +<< B;
```

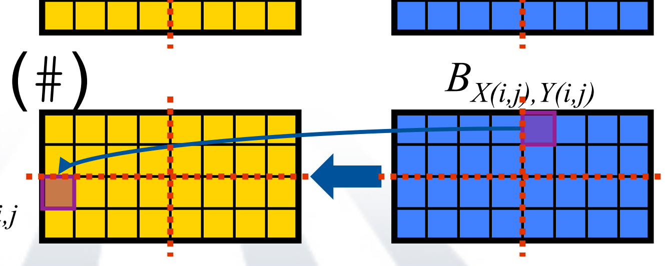


parallel prefix via *scan operator* (*op*||)

```
[R] A = +|| B;
```

arbitrary indexing via *remap operator* (#)

```
[R] A = B#[X,Y];
```



ZPL Concepts: Syntactic Performance Model

```
[InnerR] A = B;
```

No Array Operators ⇒ No Communication

```
[InnerR] A = B@[0,1];
```

At Operator ⇒ Point-to-Point Communication

```
[R] A = >>[1, 1..n] B;
```

Flood Operator ⇒ Broadcast (log-tree) Communication

```
[R] sumB = +<< B;
```

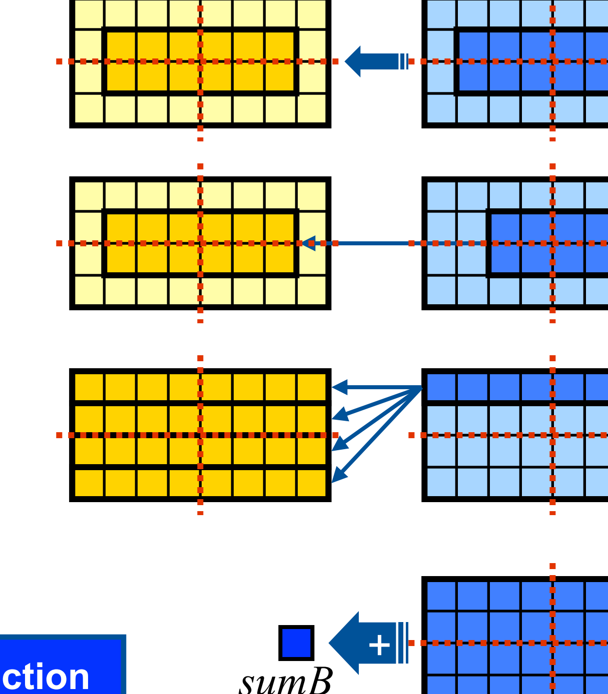
Reduce Operator ⇒ Reduction (log-tree) Communication

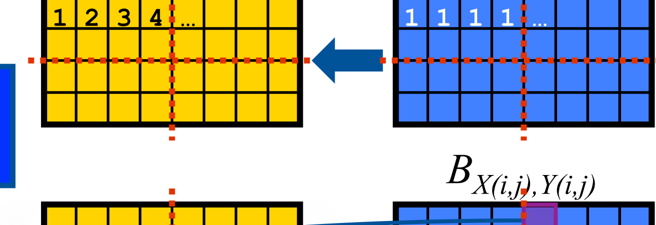
```
[R] A = +|| B;
```

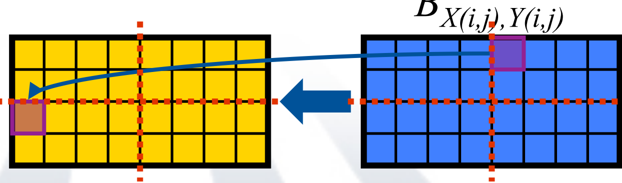
Scan Operator ⇒ Parallel-Prefix (log-tree) Communication

```
[R] A = B#[X,Y];
```

Remap Operator ⇒ Arbitrary (all-to-all) Communication







ZPL’s Lesson: Compact High-Level Code…

EP

Lines of Code (0–80) by Language

Language	computation	declarations	communication
F+MPI	54	17	3
ZPL	36	25	

CG

Lines of Code (0–300) by Language

Language	computation	declarations	communication
F+MPI	82	79	89
ZPL	38	37	

FT

Lines of Code (0–800) by Language

Language	computation	declarations	communication
F+MPI	249	332	135
ZPL	204	128	

MG

Lines of Code (0–1200) by Language

Language	computation	declarations	communication
F+MPI	242	202	566
ZPL	70	87	

IS

Lines of Code (0–300) by Language

Language	computation	declarations	communication
C+MPI	152	72	22
ZPL	111	80	

communication
declarations
computation

…need not perform poorly

EP

Speedup over best 2-processor time (2167.78 seconds in F+MPI)

128, 96, 64, 32, 0

0, 32, 64, 128, 256

Processors

- linear speedup
- F+MPI
- ZPL Grid
- ZPL Free

CG

Speedup over best 8-processor time (2620.58 seconds in A-ZPL)

44, 33, 22, 11, 0

0, 32, 64, 128, 256

Processors

- linear speedup
- F+MPI
- A-ZPL

FT

Speedup over best 32-processor time (369.26 seconds in ZPL Grid)

8, 6, 4, 2, 0

0, 32, 64, 128, 256

Processors

- linear speedup
- ZPL Grid
- F+MPI
- ZPL

MG

Speedup over best 16-processor time (114.607 seconds in A-ZPL)

16, 12, 8, 4, 0

0, 32, 64, 128, 256

Processors

- linear speedup
- A-ZPL
- ZPL
- F+MPI

IS

Speedup over best 32-processor time (34.97 seconds in C+MPI)

8, 6, 4, 2, 0

0, 32, 64, 128, 256

Processors

- linear speedup
- C+MPI
- ZPL Free
- ZPL Cut
- ZPL Pin

C/Fortran + MPI

ZPL versions

What were ZPL's shortcomings?

- Only supports a single level of data parallelism
  - imposed by execution model: single-threaded SPMD
  - not well-suited for task parallelism, dynamic parallelism
  - no support for nested parallelism
- Distinct types & operators for distributed and local arrays
  - supports ZPL's WYSIWYG syntactic model
  - impedes code reuse (and has potential for bad cross-products)
  - annoying
- Only supports a small set of built-in distributions for arrays
  - e.g., Block, Cut (irregular block), …
  - if you need something else, you're stuck

ZPL's Successes

- First-class concept for representing index sets
  - ⇒ makes clouds of scalars in array declarations and loops concrete
  - ⇒ supports global-view of data and control; improved productivity
  - ⇒ useful abstraction for user and compiler

  The Design and Implementation of a Region-Based Parallel Language. Bradford L. Chamberlain. PhD thesis, University of Washington, November 2001

- Semantics constraining alignment of interacting arrays
  - ⇒ communication requirements visible to user and compiler in syntax

  ZPL's WYSIWYG performance model. Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998.

- Implementation-neutral expression of communication
  - ⇒ supports implementation on each architecture using best paradigm

  A compiler abstraction for machine independent parallel communication generation. Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1997.

- A good start on supporting distributions, task parallelism

  Steven J. Deitz. ***High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations.*** PhD thesis, University of Washington, February 2005.

For more information on HPF and ZPL

HoPL: 3rd ACM Conference on History of Programming Languages

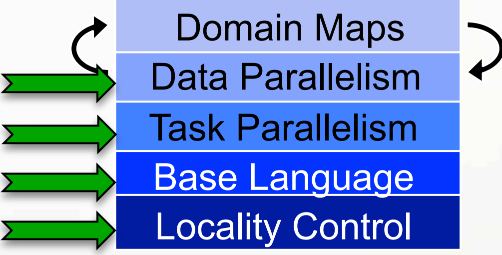
- Good retrospective summaries of languages
  - and the groups that developed them
- Papers are generally very readable, enjoyable
- Videos of talks also available on-line
- http://dl.acm.org/citation.cfm?id=1238844&picked=prox&CFID=288610646&CFTOKEN=99080594

Lessons Chapel Learned from HPF and ZPL

- Single-level of data parallelism insufficient (both)
- Practical programmers like imperative semantics (HPF)
  - not merely hints / “trust the compiler”
- Ability to reason about locality crucial (HPF)
- Syntactic performance too restrictive (ZPL)
  - semantic model with runtime queries is better
- Users need ability to specify parallel features (both)
  - data distributions
  - parallel loop schedules
  - …

Chapel's Domain Map Philosophy

1. Chapel provides a library of standard domain maps
   - to support common array implementations effortlessly
2. Advanced users can write their own domain maps in Chapel
   - to cope with shortcomings in the standard library

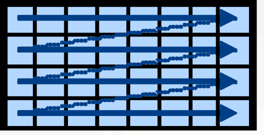
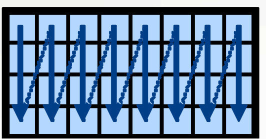
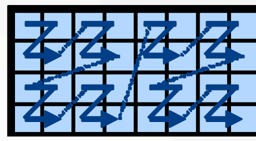
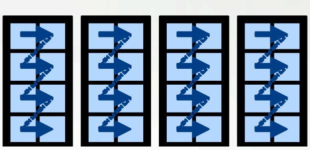


3. Chapel's standard domain maps are written using the same end-user framework
   - to avoid a performance cliff between "built-in" and user-defined cases

Data Parallelism: Implementation Qs

Q1: How are arrays laid out in memory?

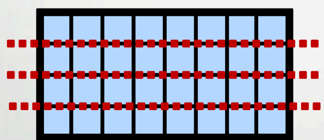
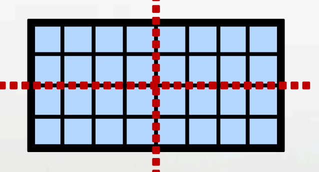
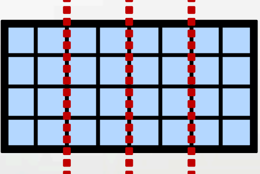
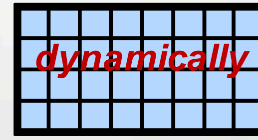
- Are regular arrays laid out in row- or column-major order? Or…?

    …?

- What data structure is used to store sparse arrays? (COO, CSR, …?)

Q2: How are data parallel operators implemented?

- How many tasks?
- How is the iteration space divided between the tasks?

…?

Data Parallelism: Implementation Qs

Q3: How are arrays distributed between locales?

- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?

Q4: What architectural features will be used?

- Can/Will the computation be executed using CPUs? GPUs? both?
- What memory type(s) is the array stored in? CPU? GPU? texture? …?

A1: In Chapel, any of these could be the correct answer

A2: Chapel's *domain maps* are designed to give the user full control over such decisions

For More Information on Domain Maps

HotPAR'10: *User-Defined Distributions and Layouts in Chapel: Philosophy and Framework,* Chamberlain, Deitz, Iten, Choi; June 2010

CUG 2011: *Authoring User-Defined Domain Maps in Chapel,* Chamberlain, Choi, Deitz, Iten, Litvinov; May 2011

Chapel release:

- Technical notes detailing domain map interface for programmers:
  $CHPL_HOME/doc/technotes/README.dsi
- Current domain maps:
  $CHPL_HOME/modules/dists/*.chpl
  layouts/*.chpl
  internal/Default*.chpl

Forall Loops: Implementation Questions

```
forall a in A do
  writeln("Here is an element of A: ", a);
```

- How many tasks will be used?
- How are iterations mapped to the tasks?

```
forall (a, i) in zip(A, 1..n) do
  a = i / 10.0;
```

Forall-loops may be zippered, like for-loops

- Corresponding iterations must match up
- But how does this work?

Leader-Follower Iterators: Definition

- Chapel defines all zippered forall loops in terms of leader-follower iterators:
  - *leader iterators:* create parallelism, assign iterations to tasks
  - *follower iterators:* serially execute work generated by leader

- Given…

```
forall (a,b,c) in zip(A,B,C) do
  a = b + alpha * c;
```

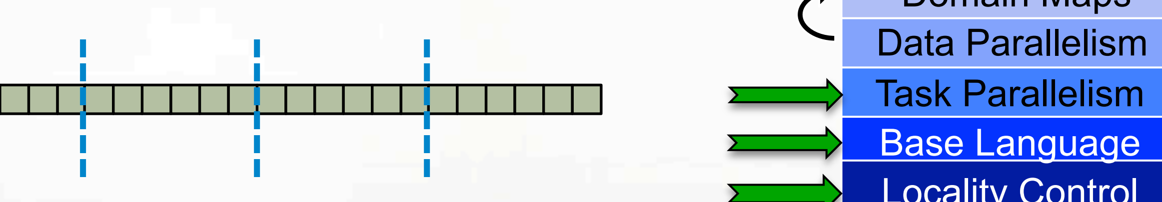
…A is defined to be the *leader*

…*A, B,* and *C* are all defined to be *followers*

Writing Leaders and Followers

Leader iterators are defined using task parallelism:

```
iter BlockArr.lead() {
  const numTasks = here.numCores();
  coforall tid in numTasks do
    yield computeMyChunk(tid, numTasks);
}
```



Follower iterators simply use serial features:

```
iter BlockArr.follow(work) {
  for i in work do
    yield accessElement(i);
}
```

For More Information on Leader-Follower Iterators

PGAS 2011: *User-Defined Parallel Zippered Iterators in Chapel,* Chamberlain, Choi, Deitz, Navarro; October 2011

Chapel release:

- $CHPL_HOME/examples/primers/leaderfollower.chpl
- See the *AdvancedIters* module, described in the "Standard Modules" section of the language specification for some interesting leader-follower iterators:
  - OpenMP-style dynamic schedules
  - work-stealing iterators