

# Scalable Shared Memory Programing

Marc Snir

PARALLEL@ILLINOIS

[www.parallel.illinois.edu](http://www.parallel.illinois.edu)

# What is (my definition of) Shared Memory

- Global name space (global references)
- Implicit data movement
- Caching: User gets good memory performance by taking care of locality
  - Temporal locality
  - Spatial locality
  - Processor locality

Locality is a property of memory accesses, not of memory layout! (good location is derived from good access pattern)

# Why Do We Like Shared Memory

- HPC applications are designed in terms of global (partitioned) data structures; one would like to avoid (up-front) manual translation from global name space to local name spaces
- One would like users to manage locality, rather than manage communication
- Good (load balanced, communication efficient) computation partition is derived from a good data partition only for very simplistic codes – need control parallelism, not data parallelism
  - Determine data location from place it is accessed, not vice-versa

# What's Bad About (hardware supported) Shared Memory

- Coherence protocols do not scale
- Need more prefetching, to cope with latency; prefetch predictors are not sufficient
  - True even at small scale
- Need to move data in bigger chunks (than a cache line) to cope with communication overhead; but large, fixed chunks result in false sharing (e.g., DSVM)
- Data races cause insidious bugs

# Does PGAS Provide Shared Memory?

- ✓ Global name space
- ✓ Implicit data movement (but copying needed for performance)
- ✗ Caching
  - Good temporal locality does not necessarily translate into reduced communication
- ✗ Limited ability to aggregate communication
- ✗ Limited ability to prefetch
  - Both depend on compiler analysis

# PGAS Productivity Issues (1)

- UPC, CAF partitions are unlikely to match application needs; need an additional level of mapping (e.g., mapping mesh patches unto UPC partitions)
  - Need not be part of core language
  - Need generic programming facilities to handle well
- UPC++? Fortran 2008?

# PGAS Productivity Issues (2)

- Good performance in UPC/CAF/... requires explicit coding of communication (aggregating, prefetching and caching) – i.e., requires coding in a distributed memory style
  - PGAS languages do not provide mechanisms for users to help with aggregation, prefetching and caching, except via explicit messaging

# Is PGAS Useful?

- Not as a productivity enhancer, but as a performance enhancer: compiled communications
  - Compiler & run-time can optimize communications (aggregation and prefetch)
  - Communications can be mapped more directly to the iron (vendors may be willing to expose to a run-time a layer they do not want to expose to end-users)
  - Second opportunity may be the most significant, but is not leveraged enough

# Beyond PGAS?

## Goals:

- Support shared memory (global name space, implicit communication, caching) at scale
- Enable user to provide information about communication – doing so in an orthogonal manner (preferably, impacting performance, not semantics)
  - Support for prefetch & aggregation
- Avoid races

# What Gives?

## Assumptions:

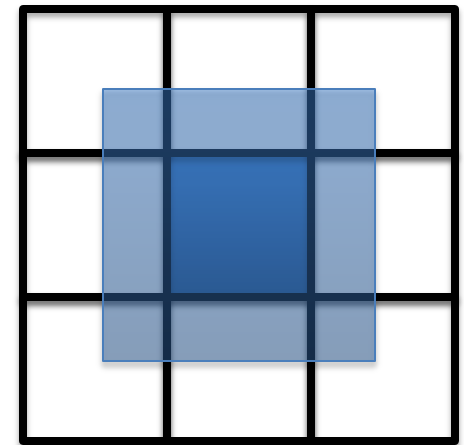
- code is “bulk-synchronous” (or nested bulk synchronous) – “ownership” of data does not change too frequently
- Communications can be coarse grained
- Changes of ownership are synchronized (no races)
  - Synchronization is collective

# Let's Think of Hardware Supported Shared Memory

- *Cache line*: unit of transfer
- *Home*: default location for line
- *Directory*: mechanism for tracking locations and state of line
- *Coherence protocol*: protocol to ensure that states of line in different caches and information in directory are consistent

# Local Memory Copy of “Cache Line”

- Data object
  - Dynamically allocated + reference swizzling
  - Or additional level of indirection
- Partitions of array
  - Dynamically allocated array chunks; UPC-style global to local address translation
- Halo
  - Extend address mapping of surrounded cell



# Home

## Current focus of PGAS languages

- Static of dynamic?
  - E.g., particles in cells: if a particle migrates, does it change “name” (memcpy) or does it change home?
- Virtualization (for load balancing and fault tolerance) requires support for home relocation
  - Global and (hopefully) infrequent operation

# Coherence Protocol (1)

- If code is race-free then all threads that need to participate in a “coherence transaction” execute a (collective) synchronization
  - Conflicting accesses to shared variables (read on write, write on write) must be ordered by an explicit synchronization; we assume it is a collective operation

# Coherence Protocol (2)

- Problem: identifying which “cache lines” are involved in the synchronization  
(doing better than cache flush/invalidate)
- Possible solution: have explicit transfer of ownership (collective state change for a “cache line” or set of “cache lines”)
  - Less painful than message passing
- Early prototypes (PPL1, MSA) show good expressiveness; not yet good performance
- If code is race-free, then no directory is needed!

# Performance Enhancers

- Aggregation: Large enough cache lines
- Prefetch:
  - memget(“cache line”)
  - memput(“cache line”, locale, state)
- Changes caching state, does not change name
- Need to be properly synchronized (as any other access), to avoid races

# Extensions

- Reductions (histograms) are handled as added states in coherence protocol:
  - Add-reduce – line can be accessed concurrently by multiple threads; only access allowed is increment.
- Still need collectives...
- Did not discuss control; this is an orthogonal issue

# Can We Ensure that Code Has No Concurrency Bugs?

Run-time checks:

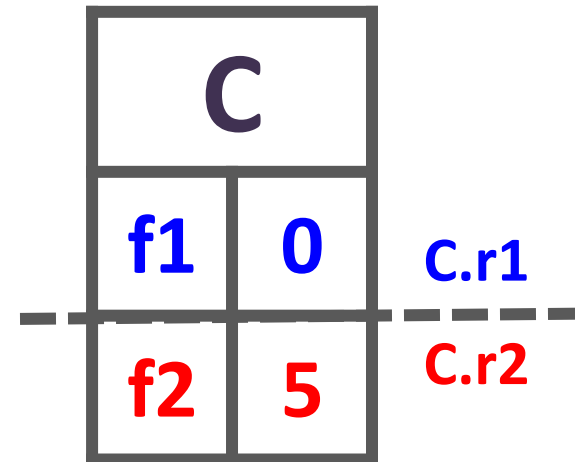
- Add global directory, and local state tag for each line
- Make sure that thread accesses only cache lines it is allowed to access, and performs only allowable accesses
  - “out of line bound” checks and guards on loads and stores
- Make sure that coherence protocol is correct
  - Check directory change is valid

# Can We Do Checks at Compile Time? (Concurrency Safety)

- Deterministic Parallel Java (Vikram Adve, Rob Bohino)
- Use Type and Effect notation
  - Heap is partitioned into regions, each with a different type
  - Methods are annotated with effect notation that specifies which type is read and which is written
  - Compile time checks ensures that concurrent accesses are consistent
  - Types can be recursive or parameterized

# Examples: Regions and Effects

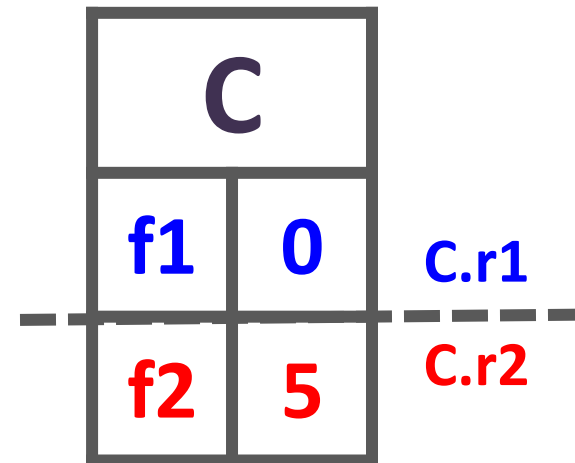
```
class C {  
  region r1, r2;  
  int f1 in r1;  
  int f2 in r2;  
  void m1(int x) writes r1 {f1 = x;}  
  void m2(int y) writes r2 {f2 = y;}  
  void m3(int x, int y){  
    cobegin {  
      m1(x);  
      m2(y);  
    }  
  }  
}
```



Partitioning the heap

# Examples: Regions and Effects

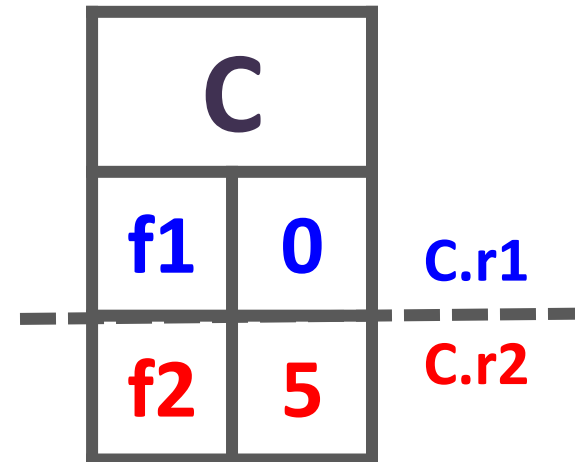
```
class C {  
  region r1, r2;  
  int f1 in r1;  
  int f2 in r2;  
  void m1(int x) writes r1 {f1 = x;}  
  void m2(int y) writes r2 {f2 = y;}  
  void m3(int x, int y){  
    cobegin {  
      m1(x);  
      m2(y);  
    }  
  }  
}
```



Summarizing method effects

# Examples: Regions and Effects

```
class C {  
  region r1, r2;  
  int f1 in r1;  
  int f2 in r2;  
  void m1(int x) writes r1 {f1 = x;}  
  void m2(int y) writes r2 {f2 = y;}  
  void m3(int x, int y){  
    cobegin {  
      m1(x); // Effect = writes r1  
      m2(y); // Effect = writes r2  
    }  
  }  
}
```



Expressing parallelism

**No run-time checks are needed!**

# More Powerful Type Expressions

- Recursive types (recursively nested regions)
  - Handles tree computations, divide and conquer, etc.
- Parameterized types
  - Handles parallel access to arrays

No run-time checks needed in either case

# Recursive Types

```
class Tree Node<region P> {  
  region Links, L, R, M, F;  
  double mass in P: M;  
  double force in P: F;  
  Tree Node<L> left in Links;  
  Tree Node<R> right in Links;  
  Tree Node<*> link in Links;  
  void compForce( )  
    reads Links, *:M writes P:F {  
    cobegin {  
      this.force = this.mass*link.mass;  
      if (left != null) left.compForce( );  
      if (right != null) right.compForce( );  
    }  
  }  
}
```

compForce reads mass fields  
from nodes reachable from  
Links and writes the local force  
field

two recursive calls can  
execute in parallel

# Parameterized Types

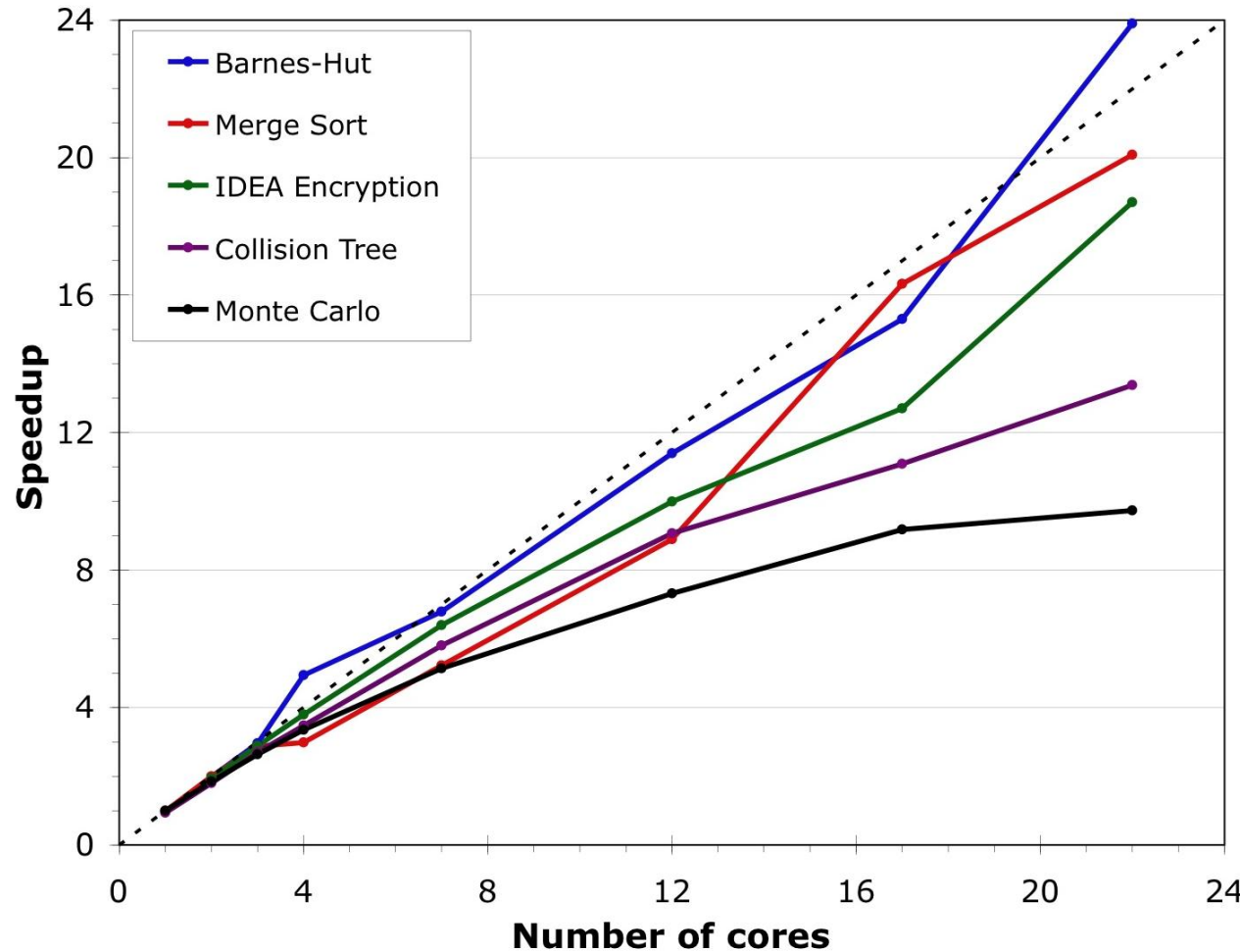
```
class Body<region P> {  
  region Link, M, F;  
  double mass in P:M;  
  double force in P:F;  
  Body<*> link in Link;  
  void compForce( ) reads Link, *:M writes P:F {  
    force = mass*link.mass;  
  }  
}  
Body< >[]< > bodies = new Body< >[N]< >;  
  foreach (int i in 0,N) {  
    bodies[i] = new Body<[i]>( );  
  }  
  foreach (int i in 0,N) {  
    bodies[i].compForce( );  
  }  
}
```

compForce reads mass fields  
from nodes reachable from Links  
and writes the local force field

all invocations can occur  
in parallel

# Expressiveness & Performance

- Can express non-trivial parallel algorithms
- Can achieve reasonable scalability
- Base performance very close to raw Java performance



# DPJizer: Porting Java to DPJ

An interactive Eclipse plug-in

Input: Java program + region declarations + foreach /  
cobegin

Output: Same program + effect summaries on  
methods

Features:

- Fully automatic effect inference
- Supports all major features of DPJ
  - Region parameters; regions for recursive data; array regions
- Interactive GUI using Eclipse mechanisms

Next step: Infer regions automatically as well

# Summary (1)

- PGAS languages essentially support same programming model as MPI
  - Fixed number of threads
  - Data location is static
  - Data movement is via explicit copying (by necessity, with MPI, for performance, with UPC)
- Not necessarily bad – MPI code can be converted with limited effort, in order to achieve better communication performance

# Summary (2)

- A “true” scalable shared memory programming model, while not extant, seems possible
  - In order to achieve performance, need coarser granularity (larger “cache lines”, less frequent state change); doable with many scientific codes
  - Scalable, fine grain shared memory requires HW support that has not proven viable in the last 30 years