Compilation Techniques for Partitioned Global Address Space Languages

Katherine Yelick

U.C. Berkeley and Lawrence Berkeley National Lab

http://titanium.cs.berkeley.edu
http://upc.lbl.gov
Outline

• LCPC is the most important conference
• PGAS Languages: what, why, how
• Open problems in analysis
  • Memory consistency
  • Data races
  • Deadlock analysis
• Open problems in optimization
• Open problems in language design
  • Beyond SPMD
• Applications in PGAS languages
Revolution in Hardware: Multicore

Performance (vs. VAX-11/780)

Power density and ILP limits ➔ software-visible parallelism

Transistor Count and HPC Parallelism

2X growth in 1.96 years!

< 100K transistors per chip until 1980

“Petascale”

Year


BG/L

Intel Paragon

ASCI Red

BG/L

Year


Transistors (MT)

Processors (MP)

1000

100

10

1

0.1

10

1

0.1

0.01

0.01

0.001

0.001

8085 8086 8080 8008 4004

8086

8080

8085

8004

486 386 286

P6

Pentium® proc

1988

1994

1997

2004

2005

2010

Slide source Horst Simon, LBNL
HPC Programming: Where are We?

• IBM SP at NERSC/LBNL has as 6K processors
  • There were 6K transistors in the Intel 8080a implementation

• BG/L at LLNL has 64K processor cores
  • There were 68K transistors in the MC68000

• A BG/Q system with 1.5M processors may have more processors than there are logic gates per processor

• HPC Applications developers today write programs that are as complex as describing where every single bit must move between the 6,000 transistors of the 8080a

• We need to at least get to “assembly language” level

Slide source: Horst Simon and John Shalf, LBNL/NERSC
Petaflop with ~1M Cores Common by 2015?

1 PFlop system in 2008

SUM

#1

#500

LCPC 2006

Slide source Horst Simon, LBNL

Kathy Yelick, 6
Predictions

• Parallelism will explode
  • Number of cores will double every 12-24 months
  • Petaflop (million processor) machines will be common in HPC by 2015 (all top 500 machines will have this)

• Performance will become a software problem
  • Parallelism and locality are key will be concerns for many programmers – not just an HPC problem

• A new programming model will emerge for multicore programming
  • Can one language cover laptop to top500 space?

• Conclusion: LCPC is arguably the most important conference in computing today!
PGAS Languages: 
What, Why, and How
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:
- Object heaps are shared
- Program stacks are private

- **3 Current languages**: UPC, CAF, and Titanium
  - All three use an SPMD execution model
  - Emphasis in this talk on UPC and Titanium (based on Java)
- **3 Emerging languages**: X10, Fortress, and Chapel
**PGAS Language Overview**

- Many common concepts, although specifics differ
  - Consistent with base language, e.g., Titanium is strongly typed
- Both private and shared data
  - `int x[10];` and `shared int y[10];`
- Support for distributed data structures
  - Distributed arrays; local and global pointers/references
- One-sided shared-memory communication
  - Simple assignment statements: `x[i] = y[i];` or `t = *p;`
  - Bulk operations: `memcpy` in UPC, array ops in Titanium and CAF
- Synchronization
  - Global barriers, locks, memory fences
- Collective Communication, IO libraries, etc.
PGAS Language for Multicore

- PGAS languages are a good fit to shared memory machines
  - Global address space implemented as reads/writes
  - Current UPC and Titanium implementation uses threads
  - Working on System V shared memory for UPC
- “Competition” on shared memory is OpenMP
  - PGAS has locality information that may be important when we get to >100 cores per chip
  - Also may be exploited for processor with explicit local store rather than cache, e.g., Cell processor
  - SPMD model in current PGAS languages is both an advantage (for performance) and constraining
PGAS Languages on Clusters: One-Sided vs Two-Sided Communication

- A one-sided put/get message can be handled directly by a network interface with RDMA support
  - Avoid interrupting the CPU or storing data from CPU (preposts)
- A two-sided messages needs to be matched with a receive to identify memory address to put data
  - Offloaded to Network Interface in networks like Quadrics
  - Need to download match tables to interface (from host)
One-Sided vs. Two-Sided: Practice

- InfiniBand: GASNet vapi-conduit and OSU MVAPICH 0.9.5
- Half power point (N ½ ) differs by *one order of magnitude*
- This is not a criticism of the implementation!

NERSC Jacquard machine with Opteron processors
GASNet: Portability and High-Performance

GASNet better for latency across machines

Joint work with UPC Group; GASNet design by Dan Bonachea
GASNet: Portability and High-Performance

Flood Bandwidth for 2MB messages

GASNet at least as high (comparable) for large messages

LCPC 2006  Joint work with UPC Group; GASNet design by Dan Bonachea  Kathy Yelick, 15
GASNet excels at mid-range sizes: important for overlap

Joint work with UPC Group; GASNet design by Dan Bonachea

Kathy Yelick, 16LCPC 2006
Communication Strategies for 3D FFT

• Three approaches:
  • **Chunk:**
    • Wait for 2nd dim FFTs to finish
    • Minimize # messages
  • **Slab:**
    • Wait for chunk of rows destined for 1 proc to finish
    • Overlap with computation
  • **Pencil:**
    • Send each row as it completes
    • Maximize overlap and
    • Match natural layout
NAS FT Variants Performance Summary

- Slab is always best for MPI; small message cost too high
- Pencil is always best for UPC; more overlap

Best MFlop rates for all NAS FT Benchmark versions

- Best NAS Fortran/MPI
- Best MPI (always Slabs)
- Best UPC (always Pencils)

(chunk NAS FT with FFTW)

<table>
<thead>
<tr>
<th>#procs</th>
<th>MFlops per Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Myrinet</td>
<td></td>
</tr>
<tr>
<td>Infiniband</td>
<td></td>
</tr>
<tr>
<td>Elan3 256</td>
<td></td>
</tr>
<tr>
<td>Elan3 512</td>
<td></td>
</tr>
<tr>
<td>Elan4 256</td>
<td></td>
</tr>
<tr>
<td>Elan4 512</td>
<td></td>
</tr>
</tbody>
</table>

.5 Tflops
Top Ten PGAS Problems

1. Pointer localization
2. Automatic aggregation of communication
3. Synchronization strength reduction
4. Automatic overlap of communication
5. Collective communication scheduling
6. Data race detection
7. Deadlock detection
8. Memory consistency
9. Global view $\rightarrow$ local view
10. Mixed Task and Data Parallelism
Communication Optimizations for Fine-grained Programs

- Programs written in a shared memory style (individual remote accesses) perform poorly on clusters
- Partial Redundancy Elimination (PRE) on address arithmetic
  - Applies also to UPC shared pointer and array arithmetic
- Split-phase communication
  - Naïve scheme of waiting for remote reads/writes is slow
  - Hide communication latency through overlapping
  - View as get/synch and put/synch pairs, and separate
- Message coalescing
  - Reduce number of messages to save startup overhead
  - Currently within an iteration; across loop (vectorization) underway

Joint work with Wei Chen and Costin Iancu
Split-phase Read (with Merge) Example

shared double *p;
...
upc_barrier;
if (...) {
    ... *p;
    ... *p;
}

H = get(&t, p);
sync(H);
= t;
sync(H);
= t;

def point of *p
merged get

got
get
Coalescing Example

shared struct { double x; double y;} *p;

stack tmp: \textit{double} $t_1$, $t_2$

\begin{align*}
p &= \text{foo}(); \\
h1 &= \text{get}(&t1, &p->x, 8); \\
h2 &= \text{get}(&t2, &p->y, 8); \\
\end{align*}

\begin{align*}
\text{sync}(h1); &\quad \text{sync}(h2); \\
= t_1; &\quad = t_2; \\
\text{sync}(h2); &\quad \text{sync}(h1); \\
= t_2; &\quad = t_1; \\
\end{align*}

stack tmp: \textit{double} $t[2]$

\begin{align*}
p &= \text{foo}(); \\
h &= \text{get}(&t, &p->x, 16); \\
\text{sync}(h); &\quad \text{sync}(h); \\
= t[0]; &\quad = t[1]; \\
\text{sync}(h); &\quad \text{sync}(h); \\
= t[1]; &\quad = t[0]; \\
\end{align*}

\begin{align*}
= p->x; &\quad = p->y; \\
= p->y; &\quad = p->x; \\
\end{align*}

a). Original UPC code \hspace{1cm} b). After split-phase \hspace{1cm} c). After coalescing
Benchmarks

- Written by people outside of our group
- Most are fine-grained, except FT and IS

- Gups: Random access (read/modify/write) to distributed array
- Mcop: Parallel dynamic programming algorithm
- Sobel: Image filter
- Psearch: Dynamic load balancing/work stealing
- Barnes Hut: Shared memory style code from SPLASH2
- NAS IS: Integer Sort using bulk communication
- NAS FT: 3d FFT using bulk communication

Joint work with Wei Chen and Costin Iancu
Optimization Breakdown

- Compares add-only, split-phase+add, and everything
- Effectiveness of optimization depends on the application
- Effectiveness also depends on networks

Relative Running Time (unoptimized = 1)

- Quadrics
- Myrinet
- Infiniband

Joint work with Wei Chen and Costin Iancu
Performance Improvement for Bulk Code

Joint work with Wei Chen and Costin Iancu
Optimizations in Titanium

• The same optimizations can be done in Titanium

• Analysis is easier:
  • Strong typing helps with alias analysis
  • Single analysis identifies global execution points that all threads will reach “together” (in same synch phase)
    • I.e., a barrier would be legal here

• Allows global optimizations
  • Convert remote reads to remote writes by other side
  • Perform global runtime analysis (inspector-executor)
  • Especially useful for sparse matrix code with indirection:
    \[ y[i] = \ldots a[b[i]] \]
Global Communication Optimizations

Sparse Matrix-Vector Multiply on Itanium/Myrinet
Speedup of Titanium over Aztec Library

- Titanium code is written with fine-grained remote accesses
- Compile identifies legal “inspector” points
- Runtime selects (pack, bounding box) per machine / matrix / thread pair

Joint work with Jimmy Su

Kathy Yelick, 27
**Local vs Global Pointers**

- *Local* keyword in Titanium ensures that compiler statically knows that data is local:

  ```c
  double [3d] myData = (double [3d] local) data[myBlockPos];
  ```

- This allows the compiler to use more efficient native pointers to reference the array:
  - Avoid runtime check for local/remote, which inhibits other optimizations
  - Use more compact pointer representation

- Titanium optimizer can often automatically propagate locality info using *Local Qualifier Inference (LQI)*:
  - Especially important because global is the default
  - Similar optimization also done in our UPC compiler on forall loops

- **Locals are important to performance:**
  - NAS CG: 58% improvement
  - NAS MG: 77% improvement

Class B on 8 procs in Titanium
Parallel Program Analysis

- To perform optimizations, new analyses are needed for parallel languages
- In a data parallel or serial (auto-parallelized) language, the semantics are serial
  - Analysis is “easier” but more critical to performance
- Parallel semantics requires
  - Concurrency analysis: which code sequences may run concurrently
  - Parallel alias analysis: which accesses could conflict between threads
- Analysis is used to detect races, identify localizable pointers, and ensure memory consistency semantics (if desired)
Concurreny Analysis in Titanium

- Relies on Titanium’s *textual barriers* and *single-valued* expressions
- Titanium has *textual barriers*: all threads must execute the same *textual* sequence of barriers (this is illegal)
  
  ```
  if (Ti.thisProc() % 2 == 0)
    Ti.barrier(); // even ID threads
  else
    Ti.barrier(); // odd ID threads
  ```
- *Single-valued* expressions used to enforce textual barriers while permitting useful programs
  
  ```
  single boolean allGo = broadcast go from 0;
  if (allGo) Ti.barrier();
  ```
- May also be used in loops to ensure same number of iterations
Concurrency Analysis

• Graph generated from program as follows:
  • Node for each code segment between barriers and single conditionals
  • Edges added to represent control flow between segments
  • Barrier edges removed

• Two accesses can run concurrently if:
  • They are in the same node, or
  • One access’s node is reachable from the other access’s node

```c
// segment 1
if ([single])
  // segment 2
else
  // segment 3
// segment 4
ti.barrier()
// segment 5
```
Alias Analysis

- Allocation sites correspond to *abstract locations (a-locs)*
  - Abstract locations (a-locs) are typed
- All explicit and implicit program variables have *points-to sets*
  - Each field of an object has a separate set
  - Arrays have a single points-to set for all elements
- Thread aware: Two kinds of abstract locations: local and remote
  - Local locations reside in local thread’s memory
  - Remote locations reside on another thread
  - Generalizes to multiple levels (thread, node, cluster)

Joint work with Amir Kamil
**Alias Analysis: Allocation**

- Creates new local abstract location
  - Result of allocation must reside in local memory

```java
class Foo {
    Object z;
}

static void bar() {
    L1: Foo a = new Foo();
    Foo b = broadcast a from 0;
    Foo c = a;
    L2: a.z = new Object();
}
```

<table>
<thead>
<tr>
<th>A-locs</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points-to Sets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alias Analysis: Assignment

- Copies source abstract locations into points-to set of target

```java
class Foo {
    Object z;
}

static void bar() {
    L1: Foo a = new Foo();
    Foo b = broadcast a from 0;
    Foo c = a;
    L2: a.z = new Object();
}
```

<table>
<thead>
<tr>
<th>A-locs</th>
<th>1, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points-to Sets</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>1.z</td>
<td>2</td>
</tr>
</tbody>
</table>
Alias Analysis: Broadcast

- Produces both local and remote versions of source abstract location
  - Remote a-loc points to remote analog of what local a-loc points to

```java
class Foo {
    Object z;
}

static void bar() {
    L1: Foo a = new Foo();
    Foo b = broadcast a from 0;
    Foo c = a;
    L2: a.z = new Object();
}
```

<table>
<thead>
<tr>
<th>A-locs</th>
<th>1, 2, 1&lt;sub&gt;r&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points-to Sets</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>1, 1&lt;sub&gt;r&lt;/sub&gt;</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>1.z</td>
<td>2</td>
</tr>
<tr>
<td>1&lt;sub&gt;r&lt;/sub&gt;.z</td>
<td>2&lt;sub&gt;r&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Points-to Sets

Joint work with Amir Kamil
Aliasing Results

• Two variables $A$ and $B$ may alias if:
  \[ \exists x \in \text{pointsTo}(A). \quad x \in \text{pointsTo}(B) \]

• Two variables $A$ and $B$ may alias across threads if:
  \[ \exists x \in \text{pointsTo}(A). \quad R(x) \in \text{pointsTo}(B), \]
  (where $R(x)$ is the remote counterpart of $x$)

<table>
<thead>
<tr>
<th>Points-to Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alias [Across Threads]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines(^1)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>56</td>
<td>Monte Carlo integration</td>
</tr>
<tr>
<td>demv</td>
<td>122</td>
<td>Dense matrix-vector multiply</td>
</tr>
<tr>
<td>sample-sort</td>
<td>321</td>
<td>Parallel sort</td>
</tr>
<tr>
<td>lu-fact</td>
<td>420</td>
<td>Dense linear algebra</td>
</tr>
<tr>
<td>3d-fft</td>
<td>614</td>
<td>Fourier transform</td>
</tr>
<tr>
<td>gsrerb</td>
<td>1090</td>
<td>Computational fluid dynamics kernel</td>
</tr>
<tr>
<td>spmv</td>
<td>1493</td>
<td>Sparse matrix-vector multiply</td>
</tr>
<tr>
<td>amr-gas</td>
<td>8841</td>
<td>Hyperbolic AMR solver for gas dynamics</td>
</tr>
<tr>
<td>amr-poisson</td>
<td>4700</td>
<td>AMR Poisson (elliptic) solver</td>
</tr>
</tbody>
</table>

\(^1\) Line counts do not include the reachable portion of the 37,000 line Titanium/Java 1.0 libraries

Joint work with Amir Kamil
## Analysis Levels

- Analyses of varying levels of precision

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>naïve</td>
<td>All heap accesses</td>
</tr>
<tr>
<td>old LQI/SQI/Sharing</td>
<td>Previous constraint-based type analysis by Aiken, Gay, and Liblit (different versions for each client)</td>
</tr>
<tr>
<td>concur</td>
<td>Concurrency analysis + type-based alias analysis</td>
</tr>
<tr>
<td>concur-pointer</td>
<td>Concurrency analysis + thread-aware alias analysis</td>
</tr>
<tr>
<td>concur-multi-level-pointer</td>
<td>Concurrency analysis + hierarchical (on and off node) thread-aware alias analysis</td>
</tr>
</tbody>
</table>
**Declarations Identified at “Local”**

Local pointers are both faster and smaller

- Old Constraint-Based (LQI)
- Thread-Aware Pointer Analysis
- Hierarchical Pointer Analysis

Joint work with Amir Kamil
Declarations Identified as Private

Private data may be cached and is known not to be in a race

Joint work with Amir Kamil
An ongoing debate is about the memory consistency model for PGAS (and OpenMP, and threads, …)

**Sequential consistency (SC):** all threads appear to execute operations in order
- a software/hardware reordering is illegal if it can be observed by another thread

**Relaxed consistency:** reordering may be observed, but local dependencies and synchronization is preserved (roughly)
- Comes in many varieties (release, total store, alpha, ppc,…)
- Trend in hardware is away from this due to complexity

**Titanium, Java, & UPC are not sequentially consistent (SC)**
- *Perceived* cost of enforcing it is too high
- For Titanium and UPC, network latency (wait for completion) is the cost
- For Java shared memory fences and code transformations are the cost

**Notes:**
- A language may be SC even if the hardware is not (insert fences)
- A language may be non-SC even though hardware is (compiler)
What Problems Arise?

- **Reordering by the compiler**
  - Reordering T1’s writes is legal by dependence analysis.

- **Reordering by hardware**
  - Same as above, if a write buffer or network may reorder.
  - Can be prevented using some form of *fence* instruction.

- **Tearing**
  - E.g., if hardware writes only 32-bit words atomically, a 64-bit value may be half-written.

```plaintext
T1
- data = 1
- flag = 1

T2
- f = flag
- d = data
```

Reordering T1’s writes is legal by dependence analysis.
Implementing Sequential Consistency

- Can sequential consistency be practical?
- How to have an SC language while allowing memory ops to be overlapped and reordered:
  - Ensure that all memory operations are ordered with respect to synchronization (fences around barriers, etc.)
  - Find all *race conditions*: concurrent accesses (at least 1 being a write) from two threads to a common variable
    - Concurrency analysis: find potentially-concurrent blocks of code
    - Alias analysis: find multiple accesses to the same variable
    - Note: both of these may be done conservatively
  - Insert fences (prevent compiler and hardware reordering) around all accesses that are part of a potential race
Implementing Sequential Consistency with Fences and Analysis

% of Static Fences Removed

- fft
- amr-gas
- gsrb
- lu-fact
- pi
- pps
- pps
- sort
- demv
- spmv
- amr-poisson

Joint work with Amir Kamil
Implementing Sequential Consistency with Fences and Analysis

% of Dynamic Fences Removed

- fft
- amr-gas
- gsrb
- lu-fact
- pi
- pi
- pps
- sort
- demv
- spmv
- amr-poisson
Making PGAS Real: Applications and Portability
AMR in Titanium

C++/Fortran/MPI AMR
- Chombo package from LBNL
- Bulk-synchronous comm:
  - Pack boundary data between procs

Titanium AMR
- Entirely in Titanium
- Finer-grained communication
  - No explicit pack/unpack code
  - Automated in runtime system

<table>
<thead>
<tr>
<th>Code Size in Lines</th>
<th>C++/F/MPI</th>
<th>Titanium</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMR data Structures</td>
<td>35000</td>
<td>2000</td>
</tr>
<tr>
<td>AMR operations</td>
<td>6500</td>
<td>1200</td>
</tr>
<tr>
<td>Elliptic PDE solver</td>
<td>4200*</td>
<td>1500</td>
</tr>
</tbody>
</table>

10X reduction in lines of code!

* Somewhat more functionality in PDE part of Chombo code
**Performance of Titanium AMR**

![Graph showing speedup comparison between Titanium and Chombo](image)

- **Serial:** Titanium is within a few % of C++/F; sometimes faster!
- **Parallel:** Titanium scaling is comparable with generic optimizations
  - optimizations (SMP-aware) that are not in MPI code
  - additional optimizations (namely overlap) not yet implemented

**Comparable parallel performance**

Joint work with Tong Wen, Jimmy Su, Phil Colella
Particle/Mesh Method: Heart Simulation

• Elastic structures in an incompressible fluid.
  • Blood flow, clotting, inner ear, embryo growth, …

• Complicated parallelization
  • Particle/Mesh method, but “Particles” connected into materials (1D or 2D structures)
  • Communication patterns irregular between particles (structures) and mesh (fluid)

Joint work with Ed Givelberg, Armando Solar-Lezama, Charlie Peskin, Dave McQueen

<table>
<thead>
<tr>
<th>Code Size in Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
</tr>
<tr>
<td>8000</td>
</tr>
</tbody>
</table>

Note: Fortran code is not parallel
Immersed Boundary Method Performance

Hand-Optimized (planes, 2004)
- 256^3 on Power3/Colony
- 512^3 on Power3/Colony
- 512^2x256 on Pent4/Myrinet

Automatically Optimized (sphere, 2006)
- 128^3 on Power4/Federation
- 256^3 on Power4/Federation

Joint work with Ed Givelberg, Armando Solar-Lezama, Charlie Peskin, Dave McQueen  Kathy Yelick, 50
Beyond the SPMD Model

• UPC factorization uses a highly multithreaded style
  • Used to mask latency and to mask dependence delays
  • Three levels of threads:
    • UPC threads (data layout, each runs an event scheduling loop)
    • Multithreaded BLAS (boost efficiency)
    • User level (non-preemptive) threads with explicit yield
  • No dynamic load balancing, but lots of remote invocation
  • Layout is fixed (blocked/cyclic) and tuned for block size

• Same framework being used for sparse Cholesky

• Hard problems
  • Block size tuning (tedious) for both locality and granularity
  • Task prioritization (ensure critical path performance)
  • Resource management can deadlock memory allocator if not careful
Dense and Sparse Matrix Factorization

Panel factorizations involve communication for pivoting.

Completed part of U

Completed part of L

Trailing matrix to be updated

Panel being factored

Blocks 2D block-cyclic distributed

Matrix-matrix multiplication used here. Can be coalesced

Matrix-matrix multiplication used here. Can be coalesced

Completed part of U

A(i,j)  A(i,k)

A(j,i)  A(j,k)
Unified Parallel C (UPC) HP Linpack Performance

- Comparable to MPI HPL (numbers from HPCC database)
- Faster than ScaLAPACK due to less synchronization
- Large scaling of UPC code on Itanium/Quadrics (Thunder)
  - 2.2 TFlops on 512p and 4.4 TFlops on 1024p

Joint work with Parry Husbands

LCPC 2006

Kathy Yelick, 54
Conclusion and Open Questions

• Best time ever for a new parallel language
  • Community is looking for parallel programming solutions
  • Not just an HPC problem

• PGAS Languages
  • Good fit for shared and distributed memory
  • Control over locality and (for better or worse) SPMD

• New role for compiler community
  • Analyzing and optimizing parallel code
  • Analysis is harder than sequential code!

• Some open language questions
  • Can sequential consistency be made practical?
    • May be different answer for shared/distributed settings
  • Will non-SPMD languages have sufficient resource control?
    • Can we get the best of task/data/SPMD models in one?