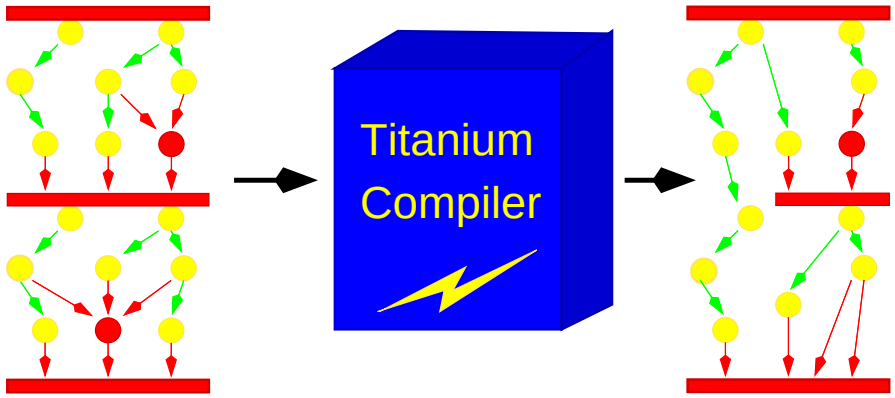
Compiling Explicitly Parallel Programs

Kathy Yelick

in collaboration with Alex Aiken, Susan Graham, and Paul Hilfinger

Computer Science Division, U.C. Berkeley



Parallel Program

Optimized
Parallel Program

Talk Overview

Premise: Most parallel programs are written using explicit parallelism, either

- message passing with a single processor multiple data (SPMD) model
- shared memory with a thread and synchronization library

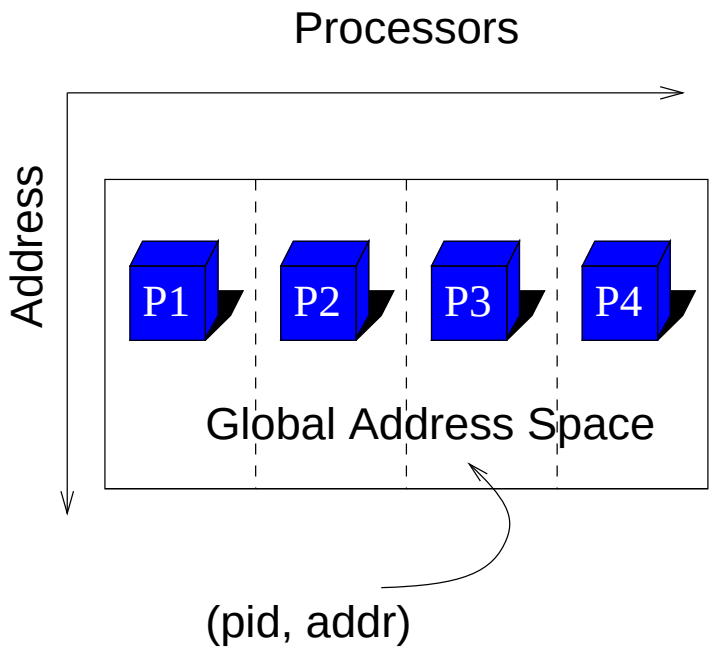
Goal: Show that analyzing and optimizing explicitly parallel programs

- can lead to high payoffs on current and future architectures
- requires new program analyses with interesting technical challenges
- is practical for a growing class of applications

Outline

- A Programming Model: Split-C
- Opportunities for optimization on modern machines (performance)
- The analysis of explicitly parallel code (correctness)
- Performance of a prototype compiler

Split-C



Features:

- SPMD parallelism
- Global address space

  int *global p; (type qualified)
- Non-blocking memory access

  x := *p; (get)

  *p := y; (put)

  sync; (wait for completion)
- One-way communication

  *p :- z; (store)

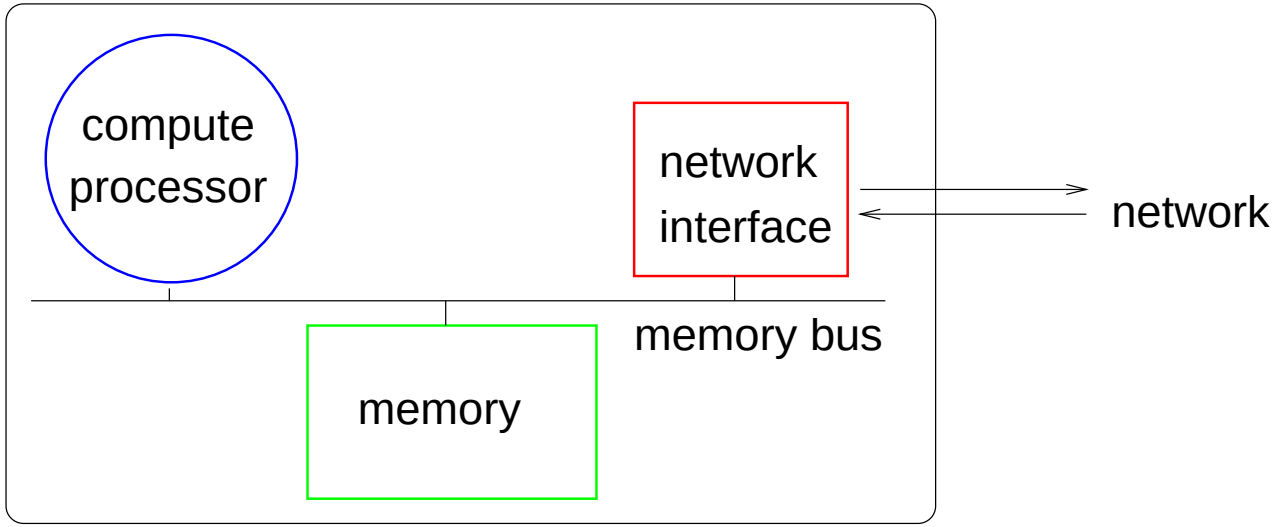
  all_store_sync; (wait globally)

Note: interesting sublanguage is Split-C without non-blocking operations.

Outline

- A Programming Model: Split-C
- Opportunities for optimization on modern machines (performance)
  - CM5: simple message passing machine
  - T3D: shared address space machine
  - Paragon and Meiko: co-processor machines
- The analysis of explicitly parallel code (correctness)
- Performance of a prototype compiler

CM5: Limited Hardware Support



Split-C memory primitives implemented on top of Active Message abstraction.

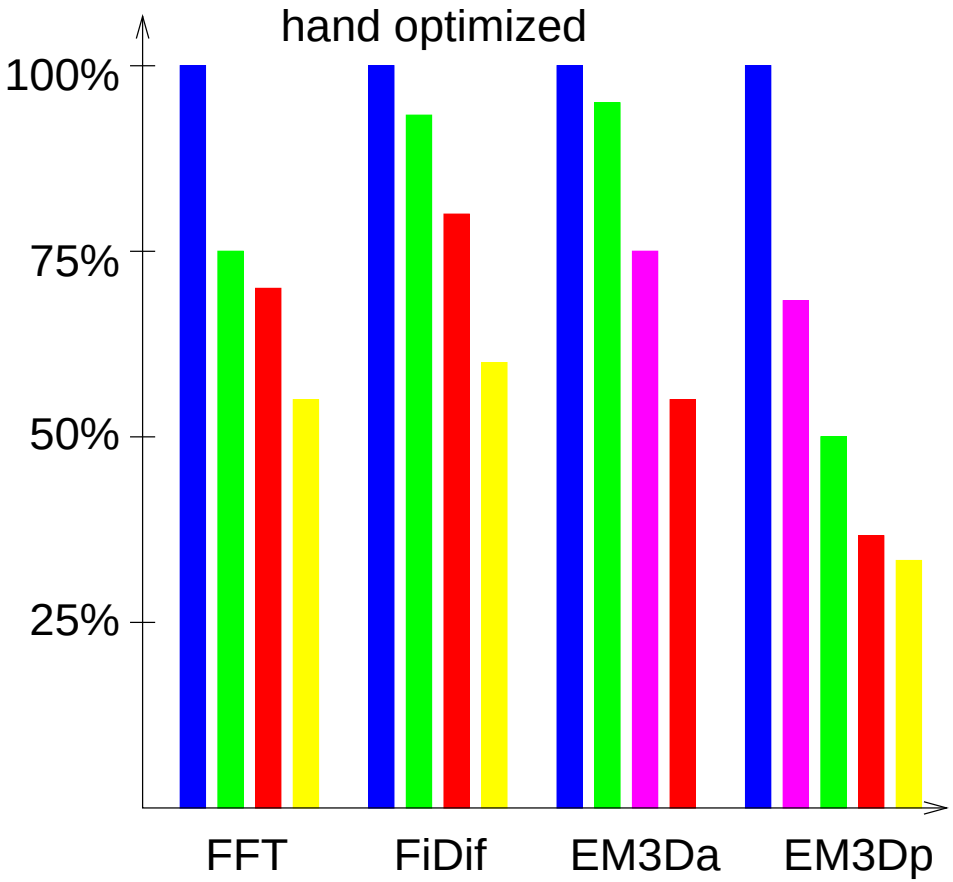
Initiation cost: 2us

Handler cost: 2.5 us

Round trip latency: 12 us

→ non-blocking and one-way operations are useful

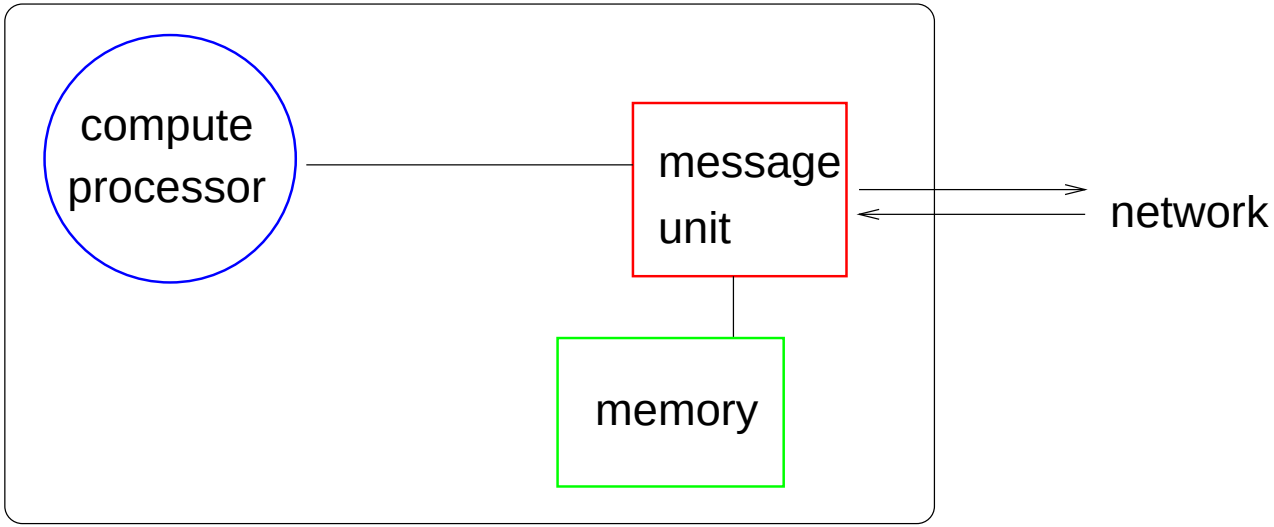
Optimizations on the CM5



Optimizations

- Unoptimized (normalize to 100%)
- Pointer strength reduction (global to local) + scalar opt.
- Caching remote values
- Overlap communication
- Global communication

T3D: Special Memory Operation Support



Off-chip circuitry supports remote read/write operations

Read cost: 115 cycles

Write cost: 110 cycles

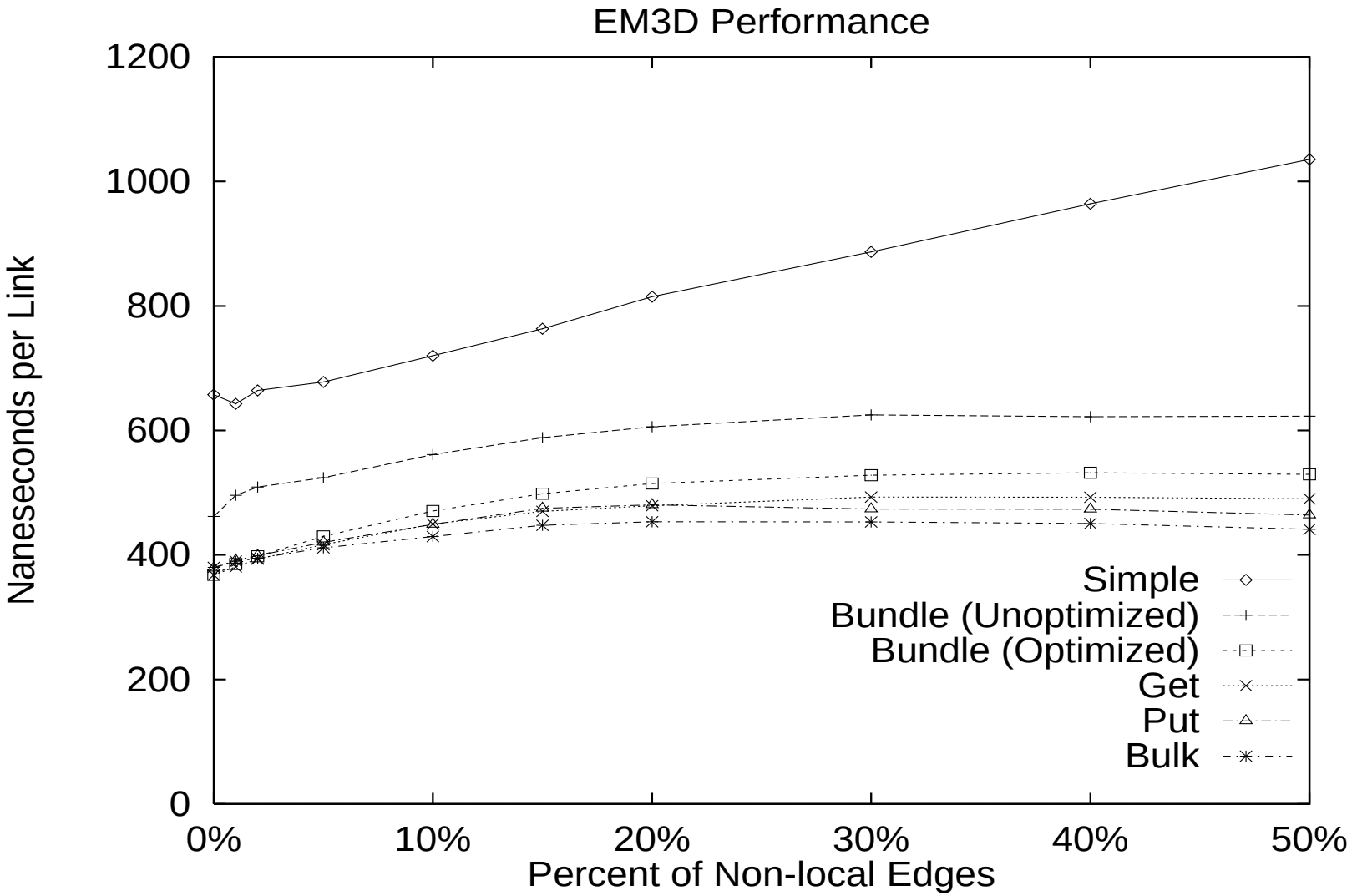
Prefetch mechanism support non-blocking get/put

Prefetch overhead: 50 cycles (prefetch into 16 element queue)

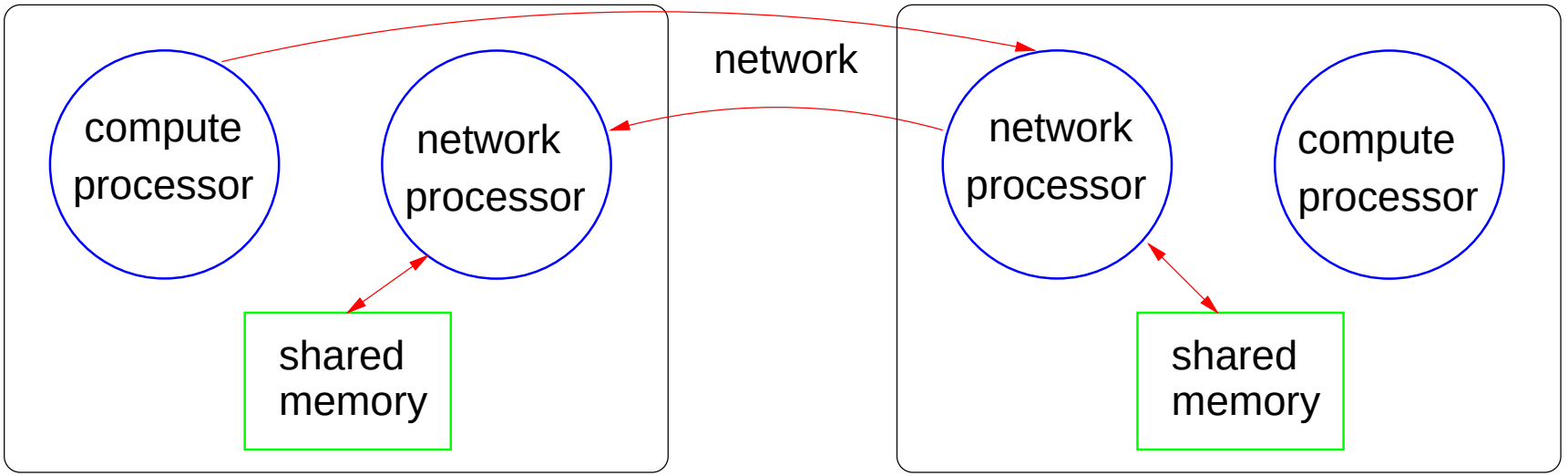
Compiler support required to manage address mapping and queue

Hardware acknowledgements => no advantage to one-way store

Optimizations on the T3D



Paragon & Meiko: Network Co-Processor



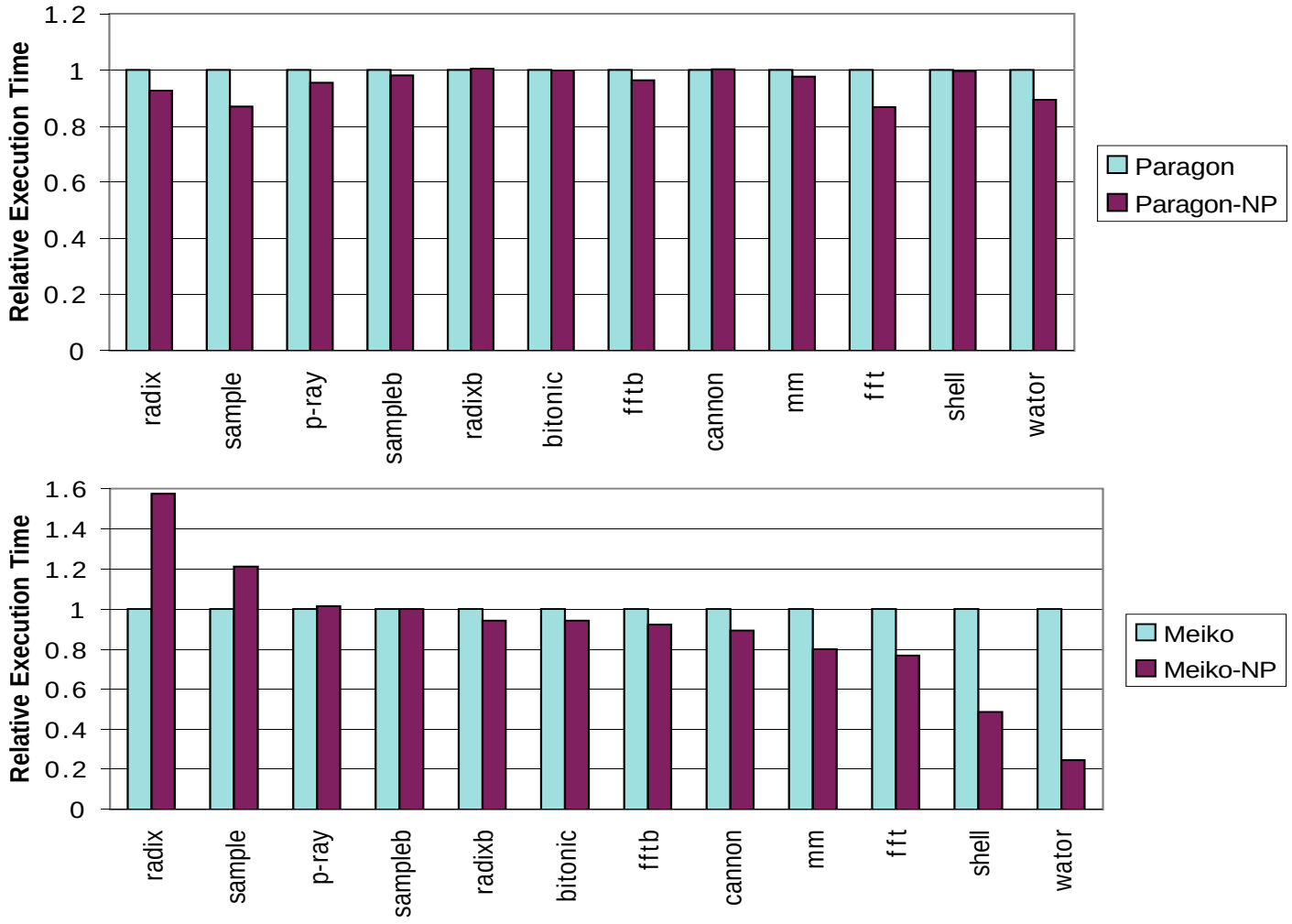
Many designs possible, since both processor can talk to network.

Hand-off to network processor reduces overhead but increases latency
Running handlers on network processor improves responsiveness & latency

Paragon: Overhead: 3us
Round-trip latency: 13us (about 40% improvement)

"Best" design depends on relative speed of 2 processors and application

Optimizations on the Paragon and Meiko



Other Sources of Reordering

Split-C implementation allows for reordering on put/get/store operations through the use of asynchronous communication in the runtime layer.

Other sources of hardware reordering include:

* out-of-order instruction execution by the processor
* write buffers with read by-pass
* non-FIFO write buffers
* inconsistent caches in SMP or cluster (software shared memory)
* in general, weak memory models

Other sources of software reordering

* register allocation
* code motion
* code elimination

System provide enforcement primitives: memory fence, volatile, etc.

Local Memory Hierarchy Optimizations: EM3d Application

1) "unoptimized" version (has main loop unrolled, and bulk communication)

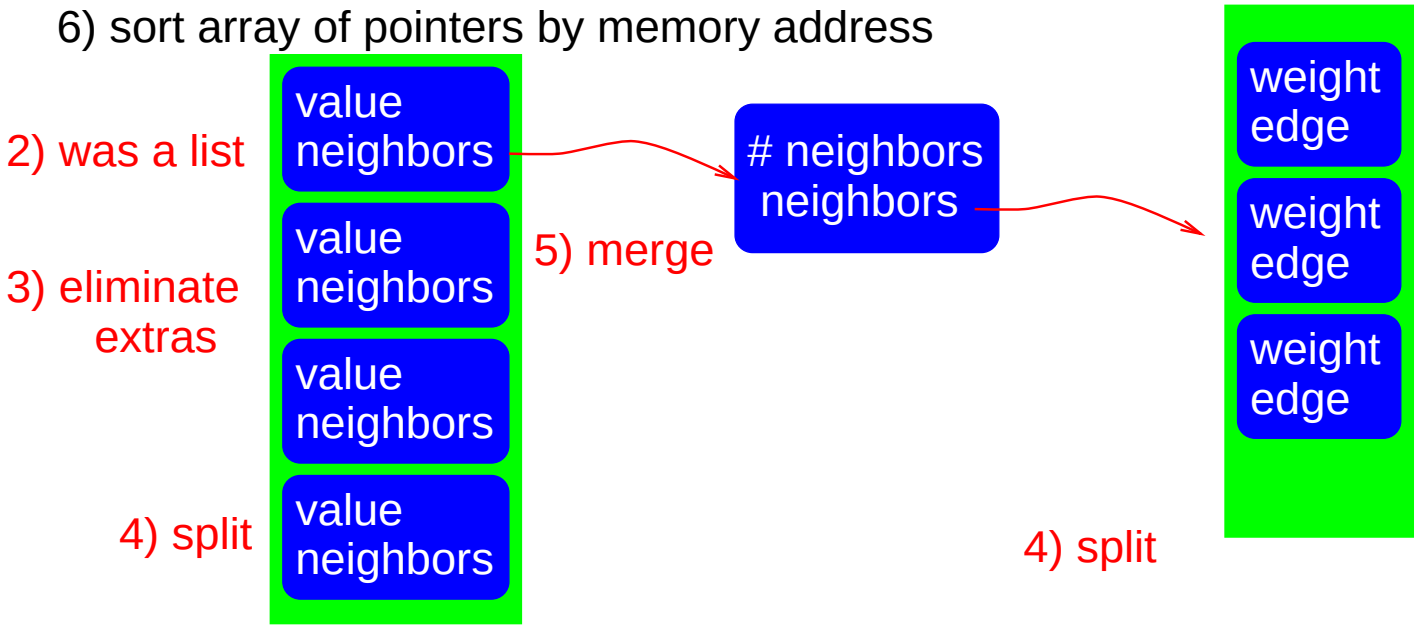
2) turn linked list into an array (graph is static)

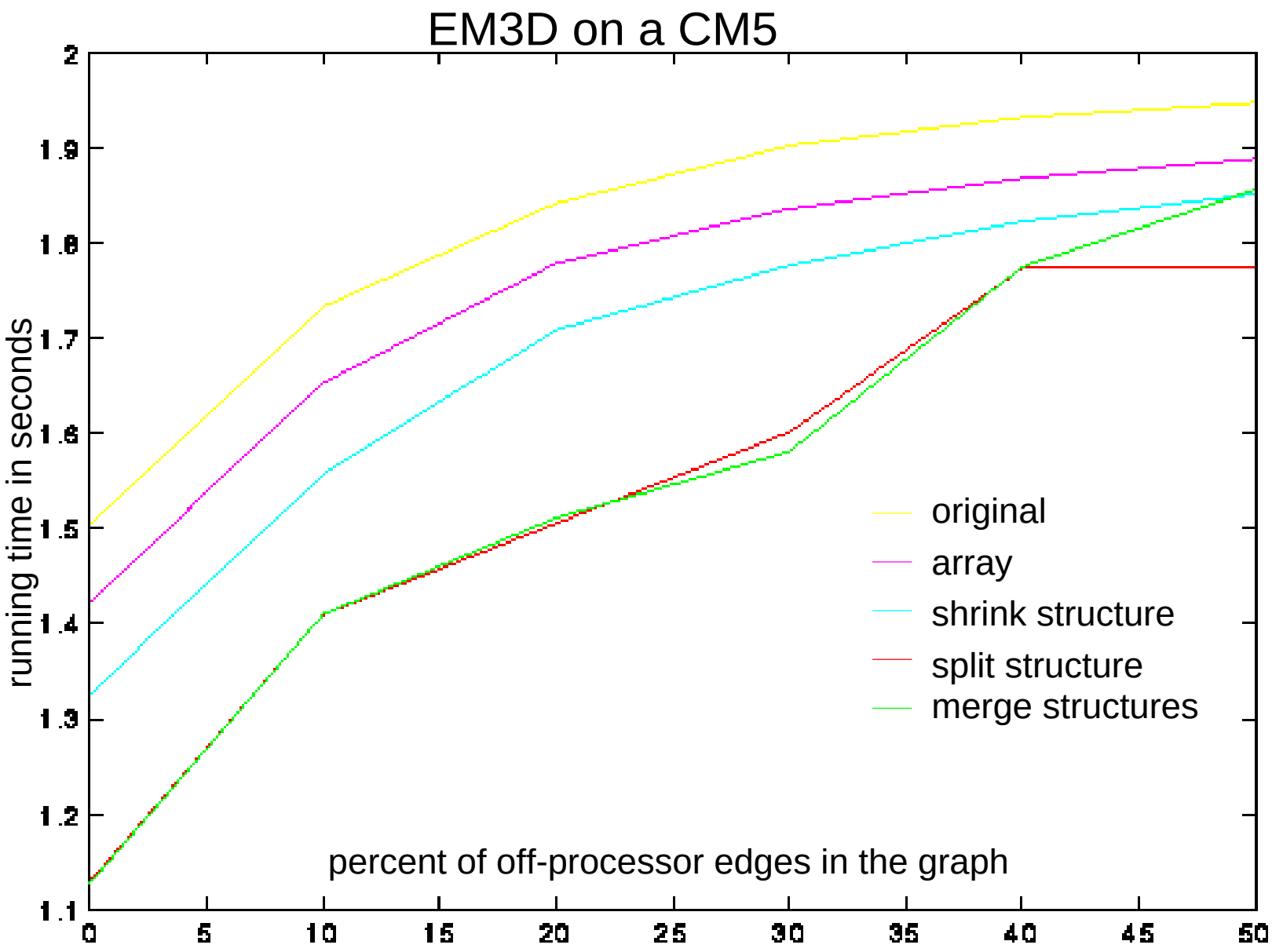
3) compress structures to remove values that are unused in compute phase

4) split structures into two to get better cache utilization during inner loop

5) merge structures linked by pointers into one

6) sort array of pointers by memory address




EM3D on a CM5
running time in seconds
percent of off-processor edges in the graph
original
array
shrink structure
split structure
merge structures
2
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
0
5
10
15
20
25
30
35
40
45
50

Summary of Performance Studies

- Exploiting machine particulars is important to high performance
- Programmers do not want to see these differences
- Even the non-blocking memory operations in Split-C are for experts only
- Moving communication operations enable other scalar optimizations

Question:

If the programmer writes a shared memory program, when is it legal for the compiler to reorder memory operations or to allow the hardware to reorder memory operations?

Outline

- A Programming Model: Split-C
- Opportunities for optimization on modern machines (performance)
- The analysis of explicitly parallel code (correctness)
  - Sequential Consistency: correctness condition
  - Cycle Detection: parallel dependence analysis
  - Synchronization Analysis: parallel flow analysis
- Performance of a prototype compiler

Sequential Consistency: Correctness Condition

When compiling sequential programs:

```
x = expr1;          ------------>    y = expr2;
y = expr2;                          x = expr1;
```

Transformation is valid if y not in exp1 and x not in exp2.

When compiling explicitly parallel code this is not a sufficient test:

initially Flag = Data = 0

Thread A	Thread B
Data = exp1	Read Flag
Flag = 1	Read Data

Neither the writes in Thread A nor reads in Thread B can be reordered.

Sequential consistency: parallel execution behaves as if memory operations from different processors are interleaved -- program order must be preserved.

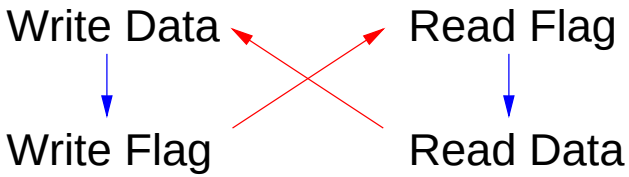
Cycle Detection: Parallel Dependence Analysis

Processors define a "program order" on accesses from the same thread.

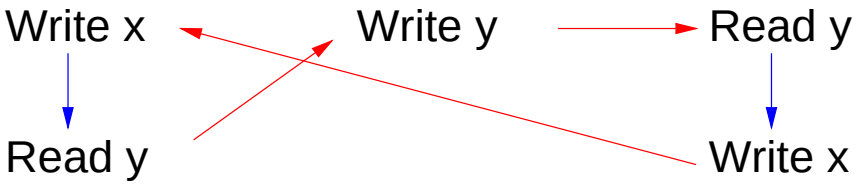
→ P is the union of these total orders

Memory system defines and "access order" on accesses to the same variable.

→ A is the access order on variables (read/write and write/write pairs)



Any violation of sequential consistency is indicated by a cycle in P + A.



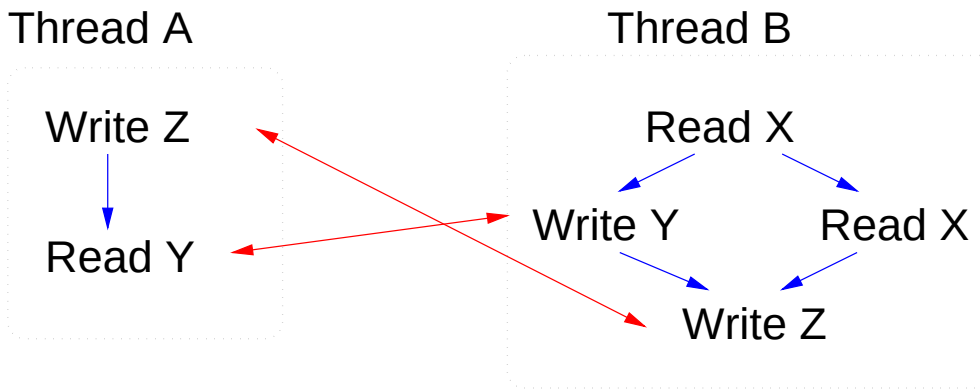
Generalizes to multiple processors and more than two variables.

Making Cycle Detection Practical

Shasha & Snir: if a cycle in P+A appears at runtime, the reordering was illegal.

For static analysis:

Approximate P by the control flow graph
Approximate A undirected "dependence" edges between threads



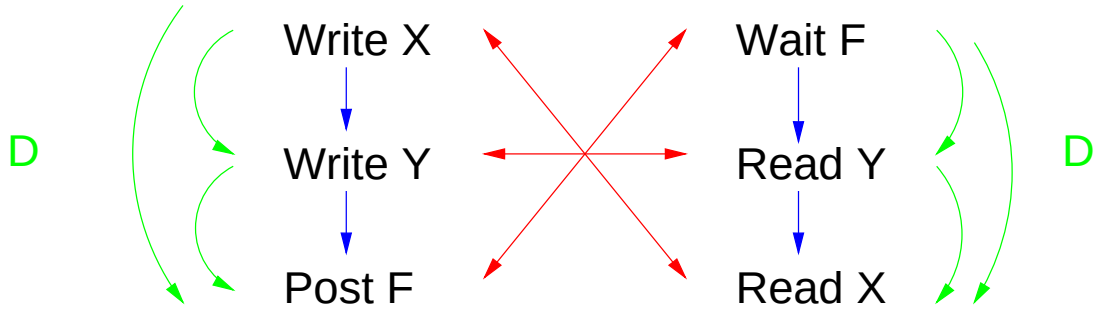
Let the "delay set" D be all edges from P that are part of a cycle.

The execution order of D edges must be preserved; other P edges may be reordered.

Inadequacies of Previous Approach

It is correct to treat synchronization constructs as reads and writes.

However, this results in undirected edges in A and spurious delay edges.



Delay edges between X and Y operations is unnecessary.

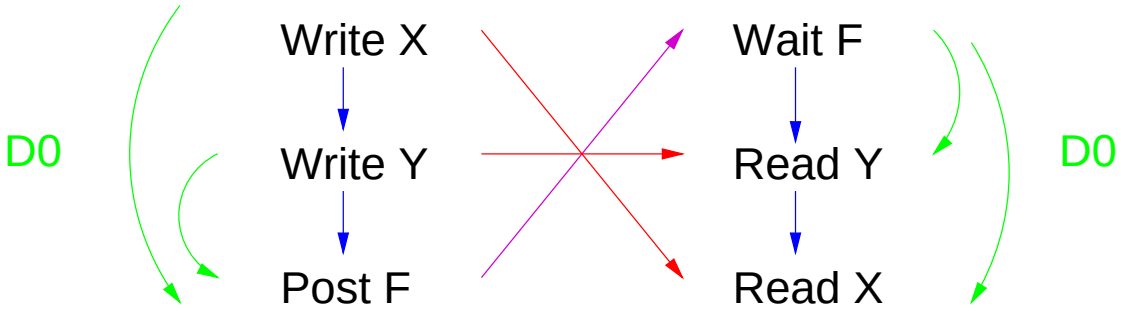
Strategy:

Treat delay edges involving synchronization constructs as more important.

Use synchronization information to improve the analysis of other edges.

Conflict Edge Orientation

Combine delay edges involving synchronization constructs, D0, with precedence to order conflict edges

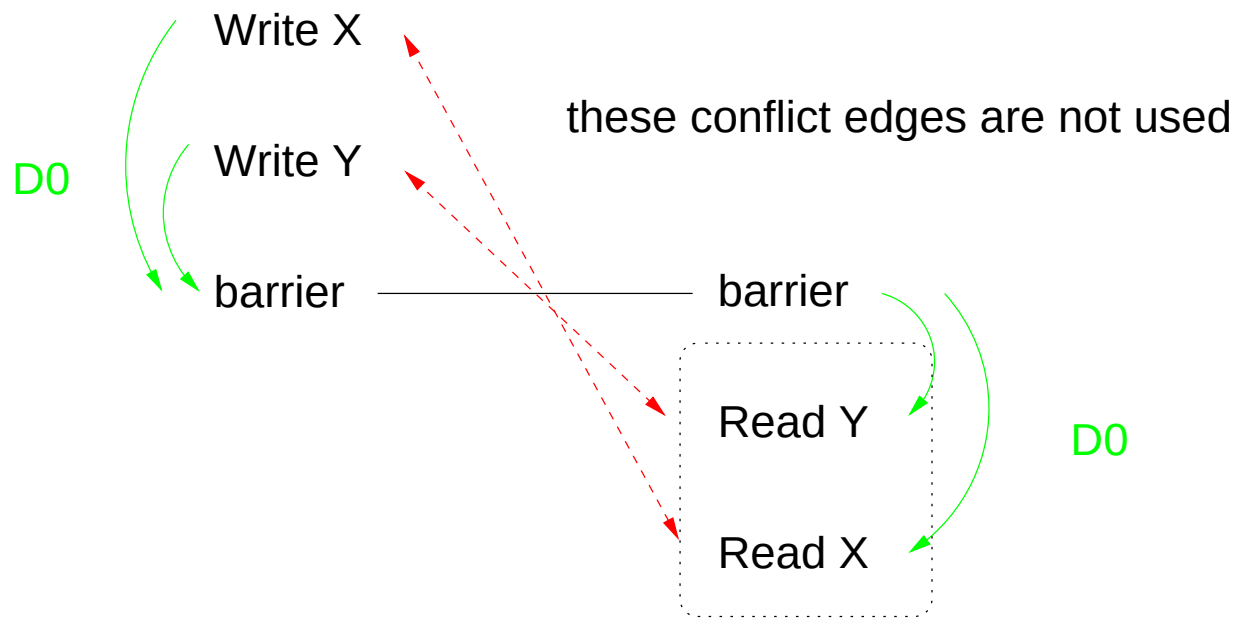


Conflict edges on X and Y become ordered by transitivity.

Final delay set D is simply D0 in this example.

Edge Deletion

When computing cycles on one code block, remove others that cannot be concurrent.



Delays between operations on X and Y are not necessary.

Summary of Parallel Program Analysis

Identify synchronization operations

- Compute precedence information

Compute delay edges for synchronization operations.

- Add these to initial delay set D0

Compute delay edges on other memory operations.

- Perform edge orientation.
- Perform edge deletion.
- If a cycle still remains, add to delay set.

Code Generation

After computing local dependencies and delay set

Split read/write into put/get:

read X → get X
sync

write Y → put Y
sync

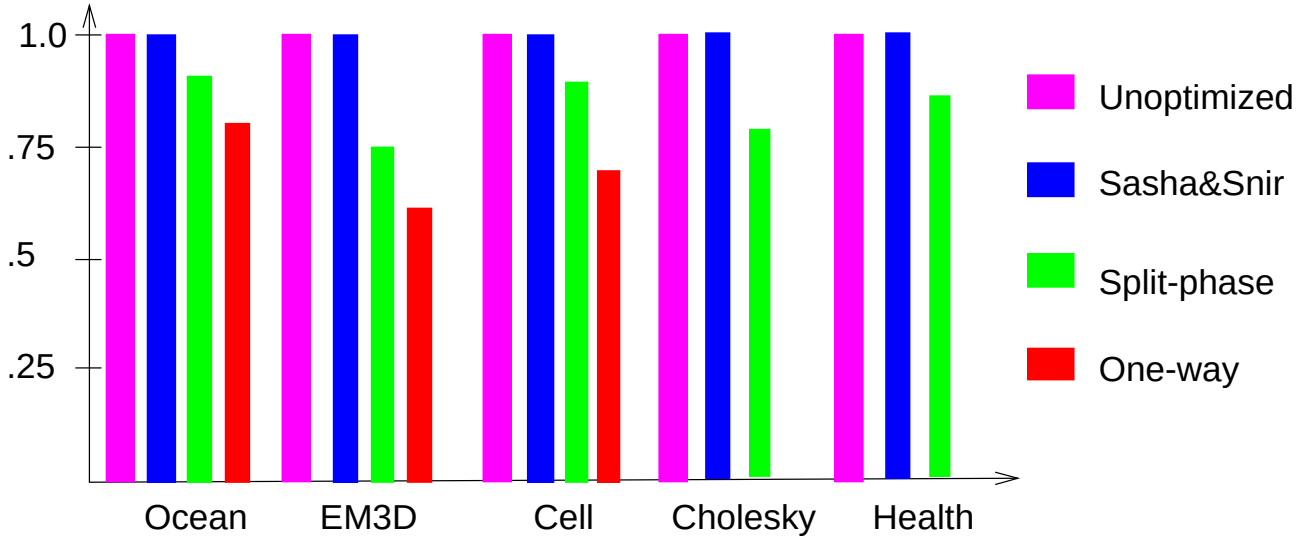
some architectures allow
finer control on sync

Move put/get and sync away from each other to enable overlap

Combine write/sync/barrier into one-way store

Combine multiple gets/puts to same location

Titanium Performance Results



The prototype compiler analyzes simple pointer-free kernels

- extends dependence analysis for shared variables
- handles common synchronization patterns (conservative)

Introduces split-phase operations and one-way communication

Ongoing and Future Work in Titanium

Joint with Aiken, Graham, Hilfinger and many students

Improved Language Design

- C++ based to allow safe arrays without going outside the language
- Synchronization primitives
- Expressibility, e.g., floating point control, out-of-core support

Foundational work

- Generalize synchronization (e.g., barrier) analysis
- Mapping to new memory models

Implementation

- Complete system
- Integrate with local memory hierarchy optimizations

Related Work

Cycle detection

- foundational work by Shasha and Snir
- extended to array analysis by Midkiff et al

Synchronization analysis

- Grunwald and Srinivasan: Data flow analysis
- Jeremiassen and Eggers: False sharing transformations
- Tseng: eliminating barriers

Weak consistency models

- Release, Weak, Entry, Processor, DAG, Location, ...

Conclusions

Obtaining high performance on modern architectures requires detailed machine model.

- increasingly true on SMPs as well as MPPs and clusters

New analysis is needed for explicitly parallel code.

- cycle detection
- synchronization analysis

Hiding the semantics of weak memory models is both possible and desirable.