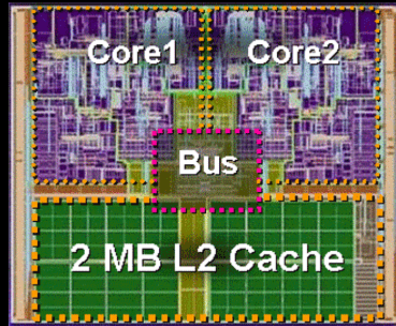
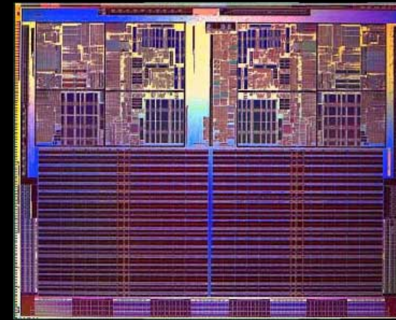
Intel Core Duo





AMD Athlon 64 X2

Multithreading and Parallel Microprocessors

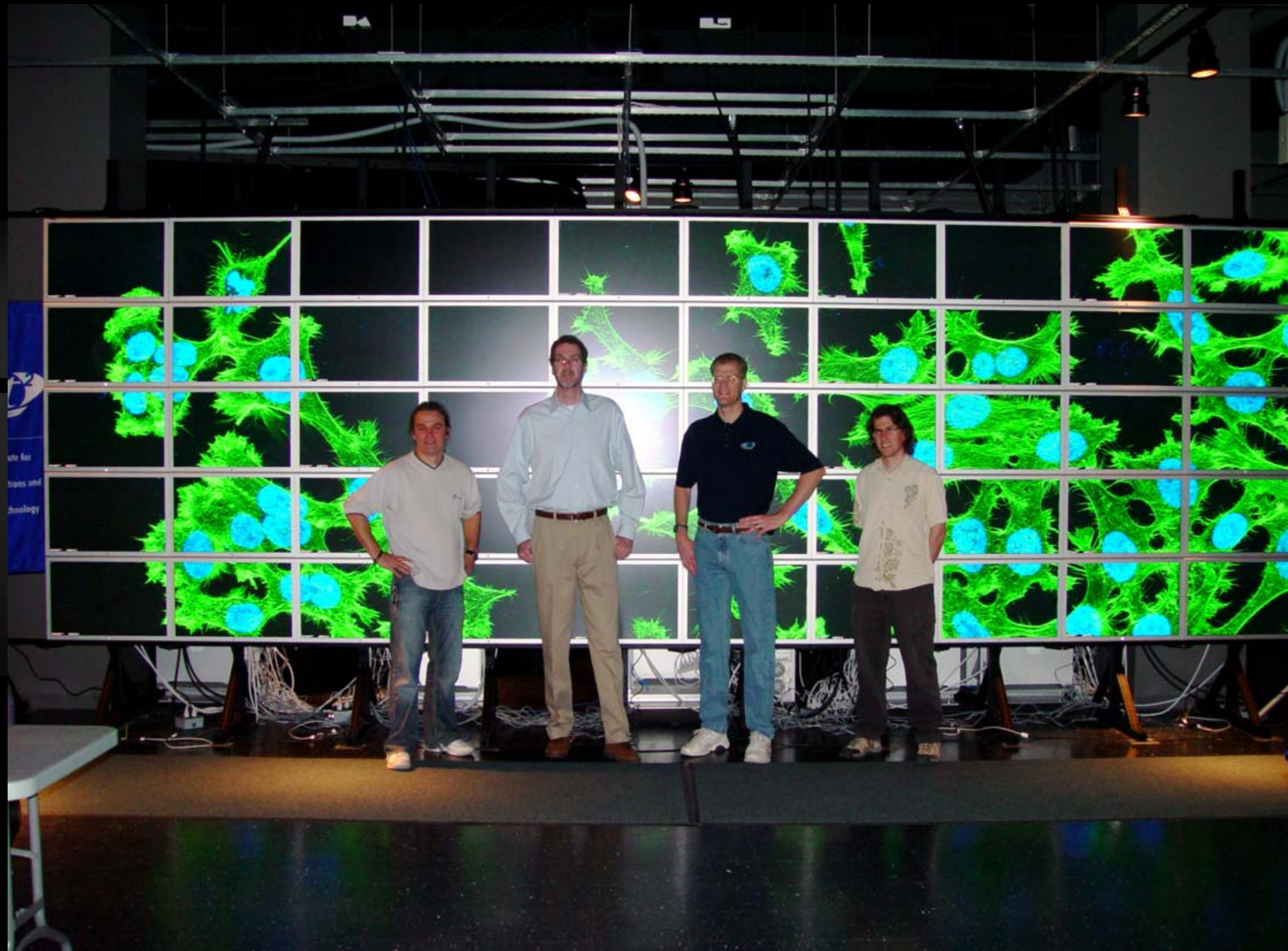
Stephen Jenks
Electrical Engineering and Computer Science
sjenks@uci.edu

Mostly Worked on Clusters

UCIrvine

Also Build Really Big Displays

HIPerWall:
200 Million Pixels
50 Displays
30 Power Mac G5s



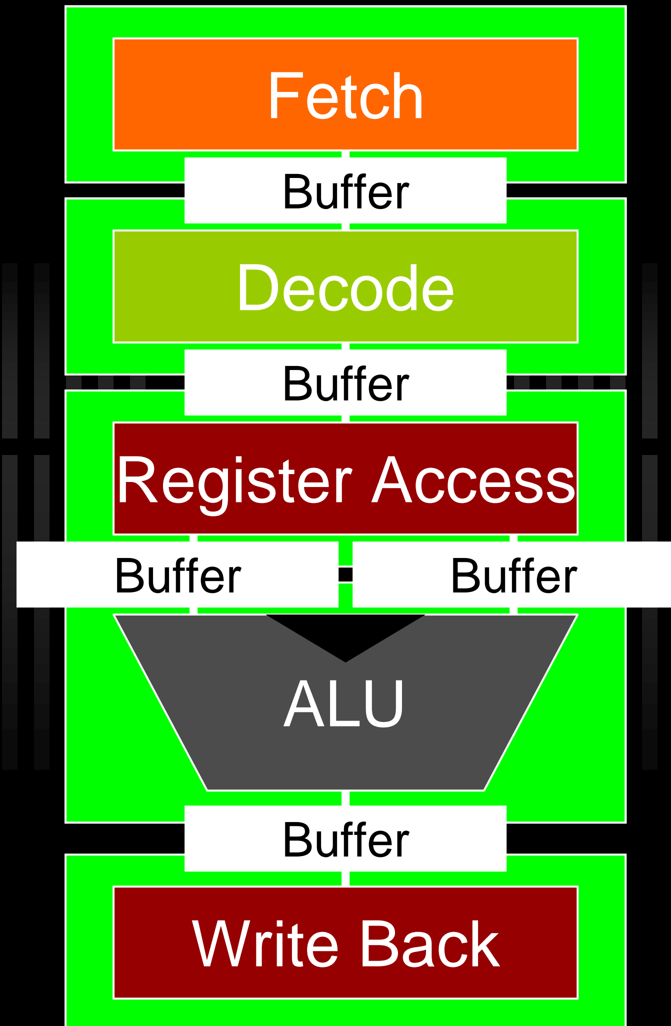
UCIrvine

Outline

- Parallelism in Microprocessors
- Multicore Processor Parallelism
- Parallel Programming for Shared Memory
  - OpenMP
  - POSIX Threads
  - Java Threads
- Parallel Microprocessor Bottlenecks
- Parallel Execution Models to Address Bottlenecks
  - Memory interface
  - Cache-to-cache (coherence) interface
- Current and Future CMP Technology

Parallelism in Microprocessors

- Pipelining is most prevalent
  - Developed in 1960s
  - Used in everything
  - Even microcontrollers
  - Decreases cycle time
  - Allows up to 1 instruction per cycle (IPC)
  - No programming changes
  - Some Pentium 4s have more than 30 stages!



More Microprocessor Parallelism

- Superscalar allows Instruction Level Parallelism (ILP)
  - Replace ALU with multiple functional units
  - Dispatch several instructions at once
- Out of Order Execution
  - Execute based on data availability
  - Requires reorder buffer
- More than 1 IPC
- No program changes

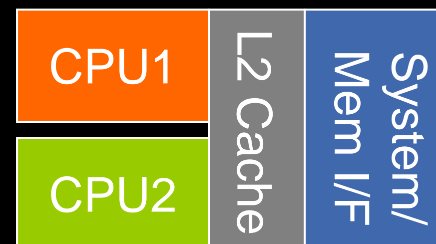
Becomes

UCIrvine

Thread-Level Parallelism

- Simultaneous Multi-threading (SMT)
  - Execute instructions from several threads at same time
  - Intel Hyperthreading, IBM Power 5/6, Cell

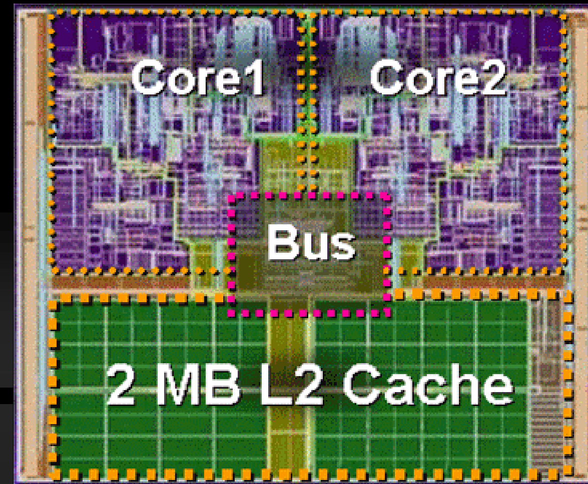
- Chip Multi-processors (CMP)
  - More than 1 CPU per chip
  - AMD Athlon 64 X2, Intel Core Duo, IBM Power 4/5/6, Xenon, Cell



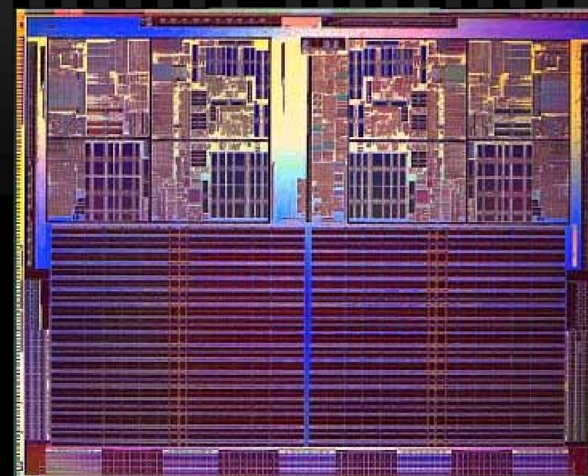
Chip Multiprocessors

- Several CPU Cores
  - Independent execution
  - Symmetric (for now)
- Share Memory Hierarchy
  - Private L1 Caches
  - Shared L2 Cache (Intel Core)
  - Private L2 Caches (AMD) (kept coherent via crossbar)
  - Shared Memory Interface
  - Shared System Interface
- Lower clock speed

Shared Resources Can Help or Hurt!



Intel Core Duo

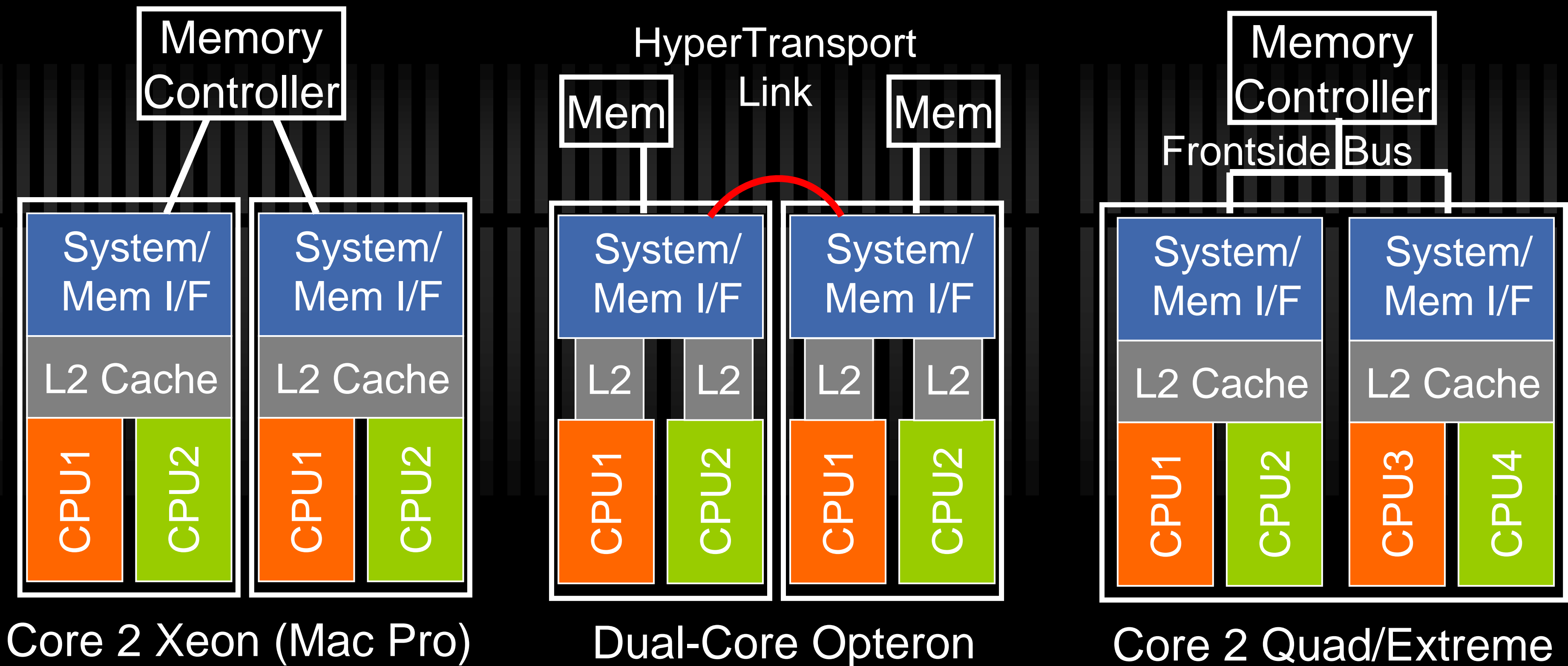


AMD Athlon 64 X2

Images from Intel and AMD

UCIrvine

Quad Cores Today



Shared Memory Parallel Programming

- Could just run multiple programs at once
  - Multiprogramming
  - Good idea, but long tasks still take long
- Need to partition work among processors
  - Implicitly (Get the compiler to do it)
    - Intel C/C++/Fortran compilers do pretty well
    - OpenMP code annotations help
    - Not reasonable for complex code
  - Explicitly (Thread programming)
- Primary needs
  - Scientific computing
  - Media encoding and editing
  - Games

Multithreading

- Definitions
  - Process - a program in execution
    - CPU state (Regs, PC)
    - Resources
    - Address space
  - Thread - lightweight process
    - CPU state
    - Shares resources and address space with other threads in same process
    - Stack
- Thread operations
  - Create / spawn
  - Join / destroy
  - Suspend & resume
- Uses
  - Solve problem together (Divide & Conquer)
  - Do different things
    - Manage game economy
    - NPC actions
    - Manage screen drawing
    - Sound
    - Input handling

UCIrvine

OpenMP Programming Model

- Implicit Parallelism with Source Code Annotations

```
#pragma omp parallel for private (i,k)
   for (i = 0; i < nx; i++)
      for (k = 0; k < nz; k++) {
         ez[i][0][k] = 0.0; ez[i][1][k] = 0.0; …
```

- Compiler reads pragma and parallelizes loop
  - Partitions work among threads (1 per CPU)
  - Vars `i` and `k` are private to each thread
  - Other vars (`ez` array, for example) are shared across all threads
  - Can force parallelization of “unsafe” loops

Thread pitfalls

- Shared data
  - 2 threads perform A = A + 1

Thread 1:	Thread 2:
1) Load A into R1	1) Load A into R1
2) Add 1 to R1	2) Add 1 to R1
3) Store R1 to A	3) Store R1 to A

  - Mutual exclusion preserves correctness
    - Locks/mutexes
    - Semaphores
    - Monitors
    - Java "synchronized"
- False sharing
  - Non-shared data packed into same cache line

```
int thread1data;
int thread2data;
```

  - Cache line ping-pongs between CPUs when threads access their data
- Locks for heap access
  - malloc() is expensive because of mutual exclusion
  - Use private heaps

POSIX Threads

- IEEE 1003.4 (Portable Operating System Interface) Committee
- Lightweight "threads of control"/processes operating within a single address space
  - A Typical "Process" contains a single thread in its address space
  - Threads run concurrently and allow
    - Overlapping I/O and computation
    - Efficient use of multiprocessors
- Also called *pthreads*

Concept of Operation

1. When program starts, main thread is running
2. Main thread **spawns** child threads as needed
3. Main thread and child threads run concurrently
4. Child threads finish and **join** with main thread
5. Main thread terminates when process ends

UCIrvine

Approximate Pi with pthreads

```c
/* the thread control function */
void* PiRunner(void* param)
{
  int threadNum = (int) param;
  int i;
  double h, sum, mypi, x;

  printf("Thread %d starting.\n", threadNum);

  h = 1.0 / (double) iterations;
  sum = 0.0;
  for (i = threadNum + 1; i <= iterations; i += threadCount) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  mypi = h * sum;

  /* now store the result into the result array */
  resultArray[threadNum] = mypi;

  printf("Thread %d exiting.\n", threadNum);
  pthread_exit(0);
}
```

More Pi with pthreads: main()

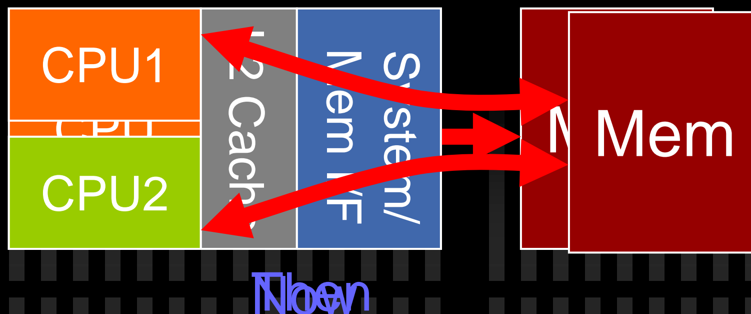
```c
/* get the default attributes and set up for creation */
 for (i = 0; i < threadCount; i++) {
   pthread_attr_init(&attrs[i]);
   /* system-wide contention */
   pthread_attr_setscope(&attrs[i], PTHREAD_SCOPE_SYSTEM);
 }

 /* create the threads */
 for (i = 0; i < threadCount; i++) {
   pthread_create(&tids[i], &attrs[i], PiRunner, (void*)i);
 }

 /* now wait for the threads to exit */
 for (i = 0; i < threadCount; i++)
   pthread_join(tids[i], NULL);

 pi = 0.0;
 for (i = 0; i < threadCount; i++)
   pi += resultArray[i];
```

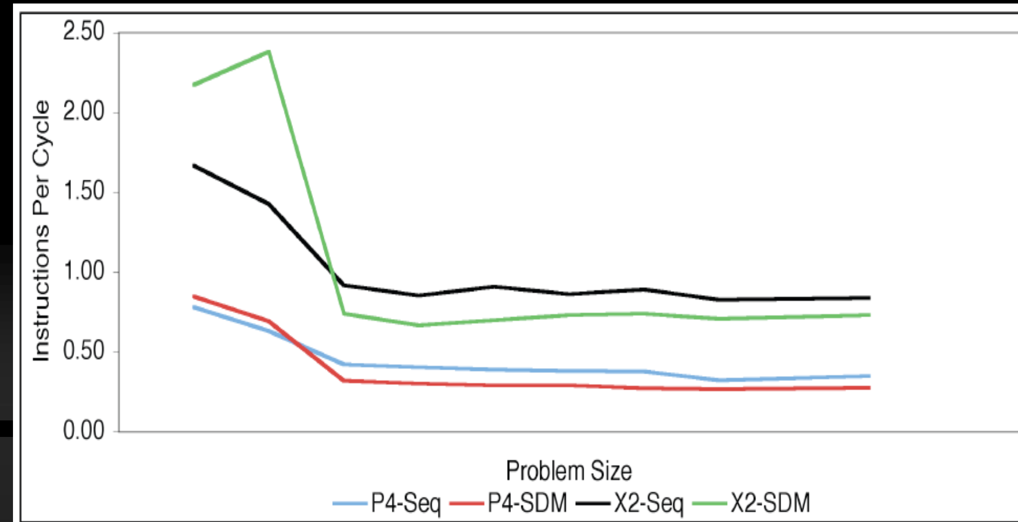
UCIrvine

Java Threads

- Threading and synchronization built in
- An object can have associated thread
  - Subclass Thread or Implement Runnable
  - “run” method is thread body
  - “synchonized” methods provide mutual exclusion
- Main program
  - Calls “start” method of Thread objects to spawn
  - Calls “join” to wait for completion

Parallel Microprocessor Problems

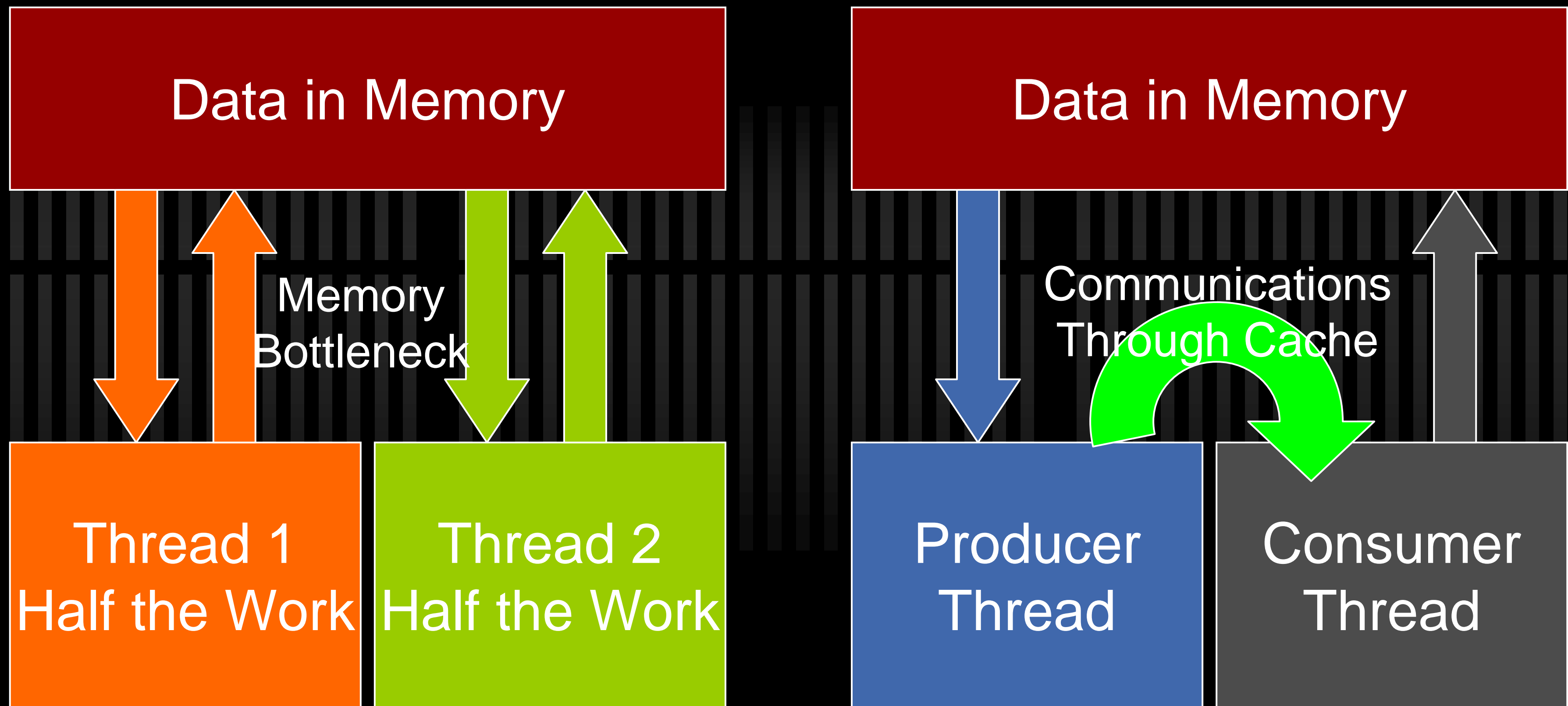




- Memory interface too slow for 1 core/thread
- Now multiple threads access memory simultaneously, overwhelming memory interface
- Parallel programs can run as slowly as sequential ones!

UCIrvine

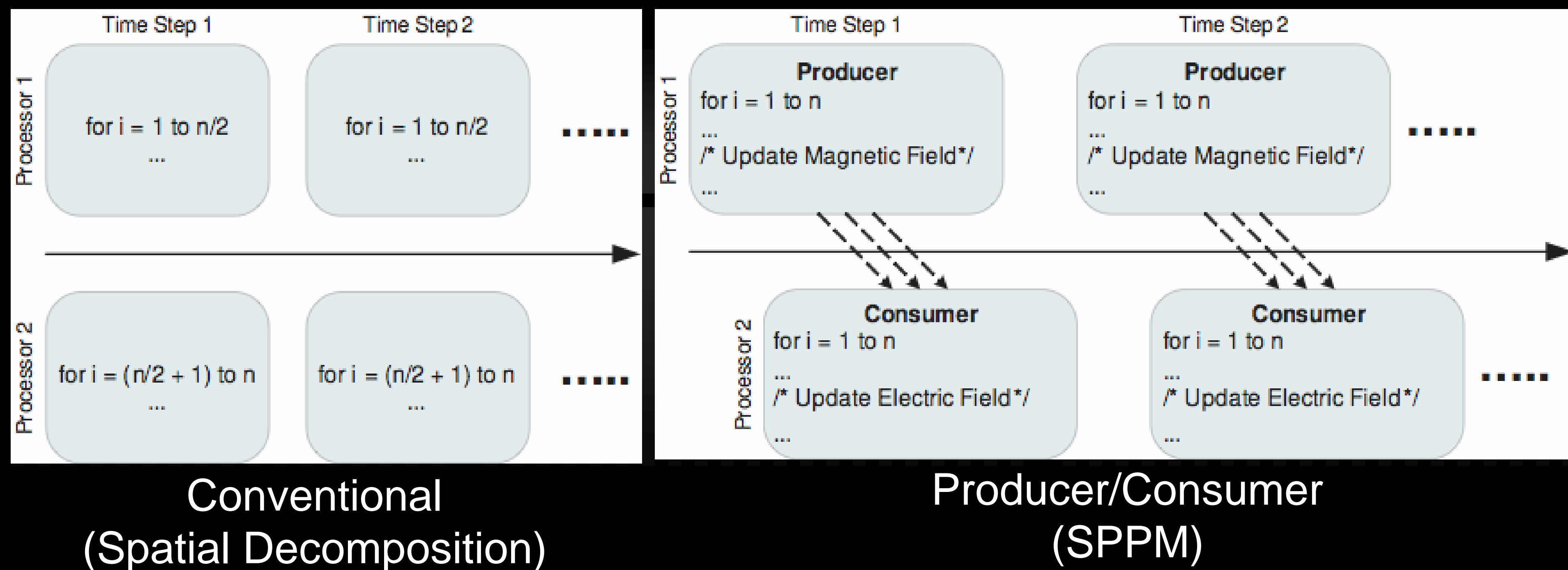
Our Solution: Producer/Consumer Parallelism Using The Cache



Converting to Producer/Consumer

```
for (i = 1; i < nx - 1; i++){
  for (j = 1; j < ny - 1; j++){
    /* Update Magnetic Field */
    for (k = 1; k < nz - 1; k++){
      double invmu = 1.0/mu[i][j][k];
      double tmpx = rx*invmu; double tmpy = ry*invmu; double tmpz = rz*invmu;
      hx[i][j][k] += tmpz * (ey[i][j][k+1] - ey[i][j][k])
        - tmpy * (ez[i][j+1][k] - ez[i][j][k]);
      hy[i][j][k] += tmpx * (ez[i+1][j][k] - ez[i][j][k])
        - tmpz * (ex[i][j][k+1] - ex[i][j][k]);
      hz[i][j][k] += tmpy * (ex[i][j+1][k] - ex[i][j][k])
        - tmpx * (ey[i+1][j][k] - ey[i][j][k]);
    }
    /* Update Electric Field */
    for (k = 1; k < nz - 1; k++){
      double invep = 1.0/ep[i][j][k];
      double tmpx = rx*invep; double tmpy = ry*invep; double tmpz = rz*invep;
      ex[i][j][k] += tmpy * (hz[i][j][k] - hz[i][j-1][k])
         - tmpz * (hy[i][j][k] - hy[i][j][k-1]);
      ey[i][j][k] += tmpz * (hx[i][j][k] - hx[i][j][k-1])
         - tmpx * (hz[i][j][k] - hz[i-1][j][k]);
      ez[i][j][k] += tmpx * (hy[i][j][k] - hy[i-1][j][k])
         - tmpy * (hx[i][j][k] - hx[i][j-1][k]); } } }
```

Synchronized Pipelined Parallelism Model (SPPM)



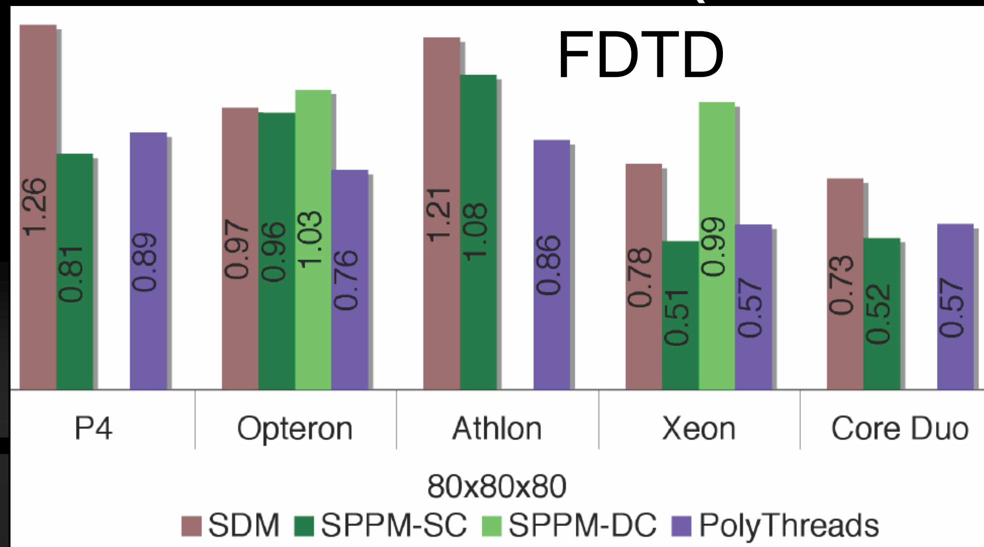
Conventional (Spatial Decomposition)

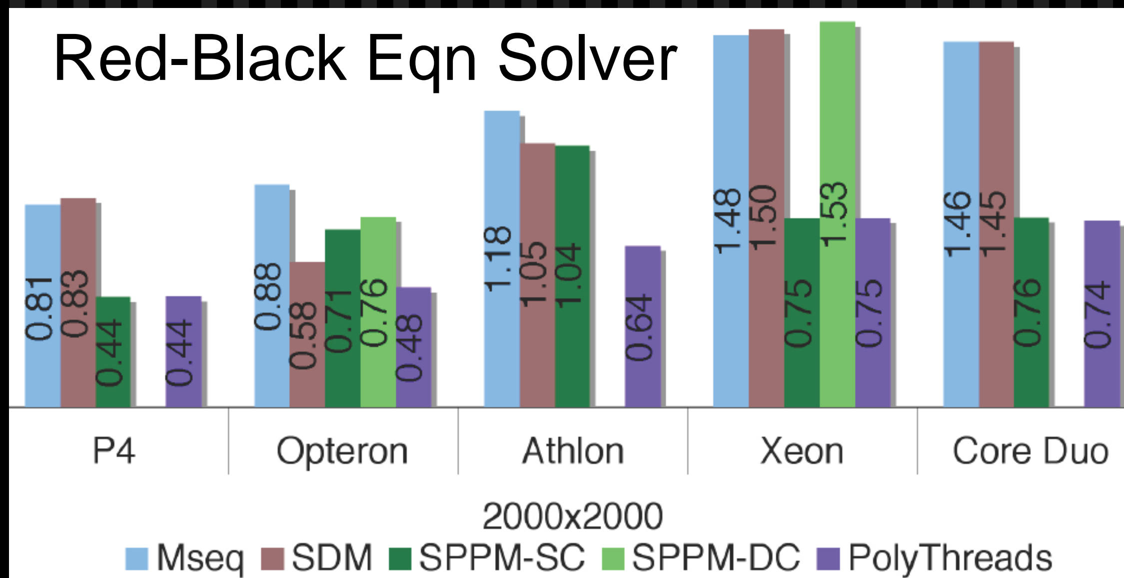
Producer/Consumer (SPPM)

SPPM Features

- Benefits
  - Memory bandwidth same as sequential version
  - Performance improvement (usually)
  - Easy in concept
- Drawbacks
  - Complex programming
  - Some synchronization overhead
  - Not always faster than SDM (or sequential)

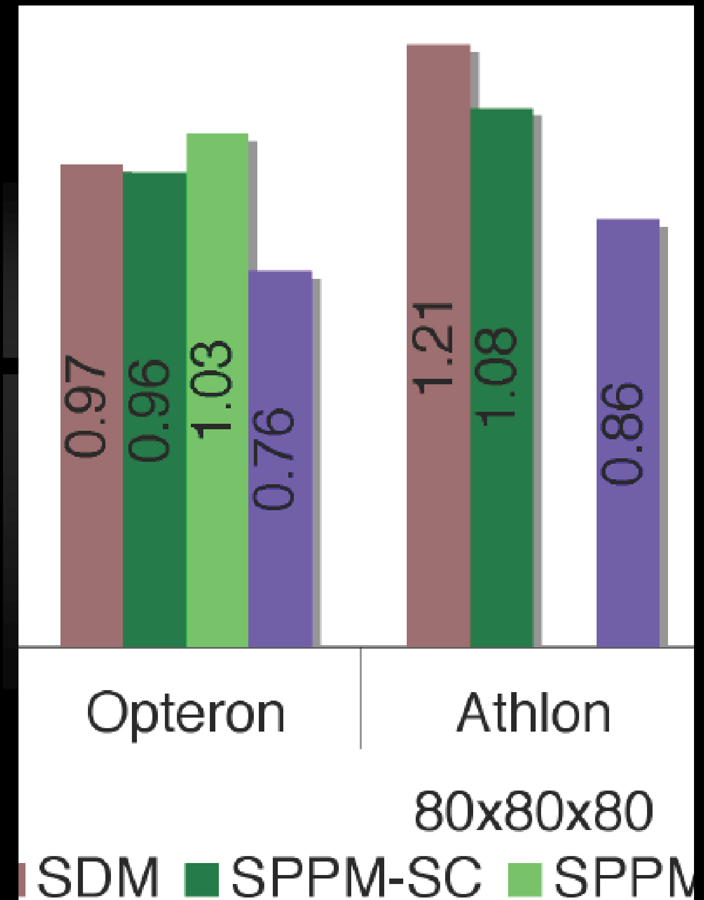
SPPM Performance (Normalized)



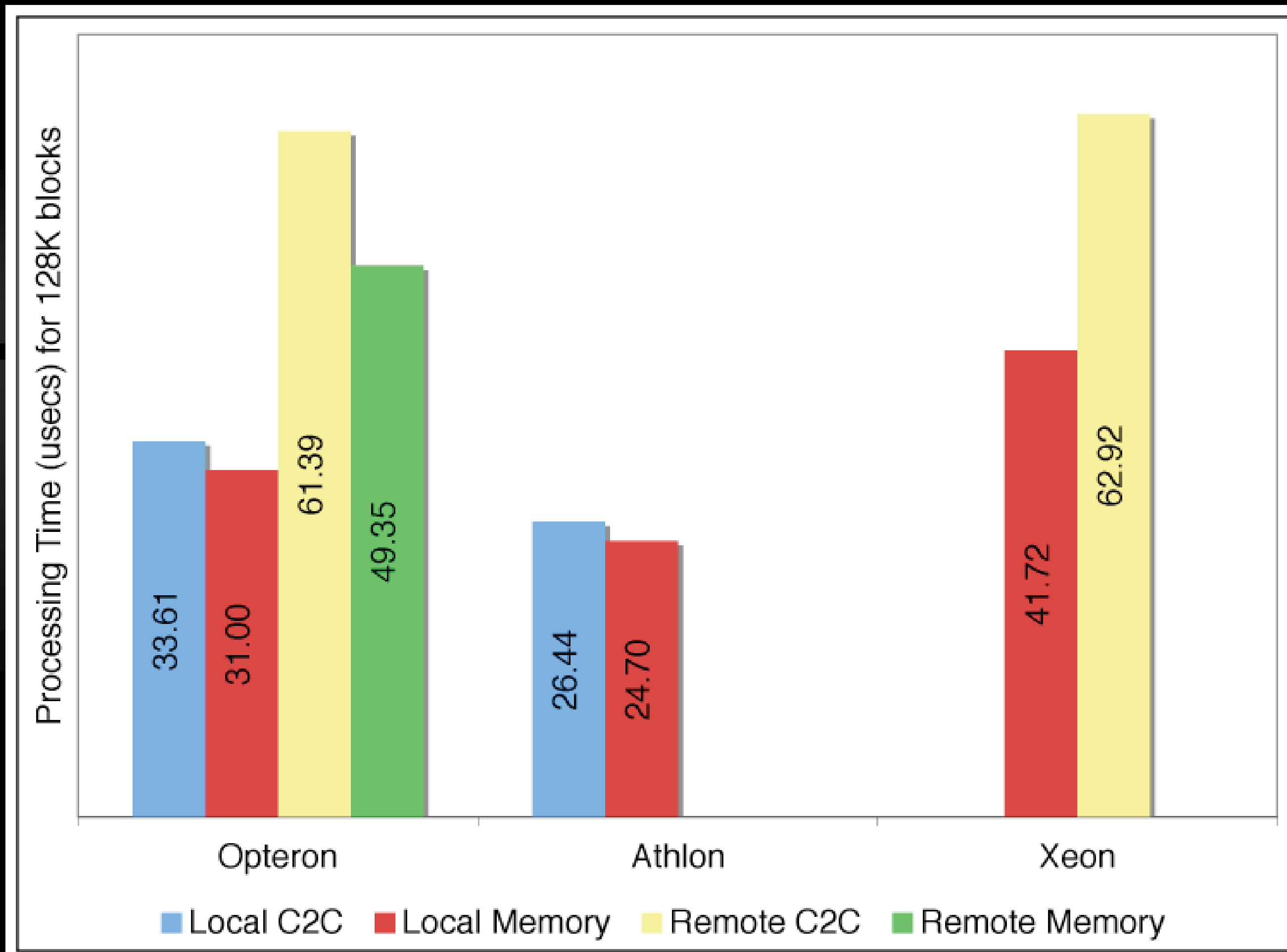


So What's Up With AMD CPUs?

- How can SPPM be slower than Seq?
  - Fetching from other core's cache is slower than fetching from memory!
  - Makes consumer slower than producer!

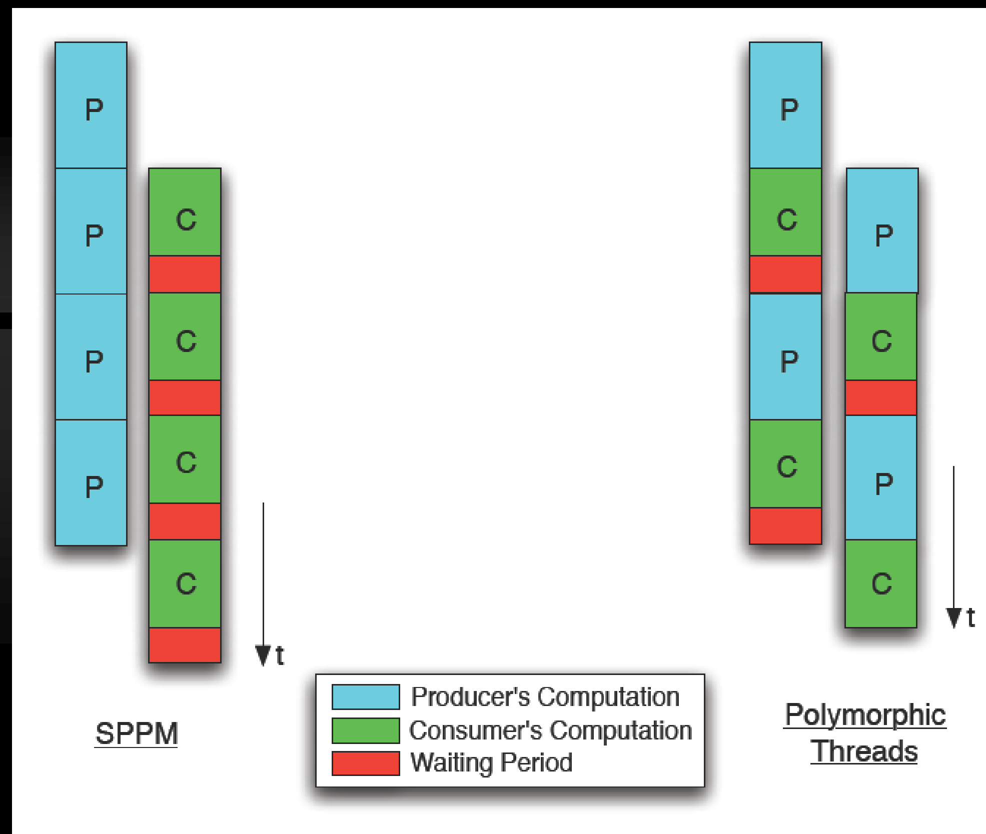


CMP Cache Coherence Stinks!



Private Cache Solution: Polymorphic Threads

- Cache-to-Cache too slow
- Therefore can't move much data between cores
- Polymorphic Threads
  - Thread morphs between producer and consumer for each block
  - Sync data passed between caches
  - But more complex program
  - Good on private caches!
  - Not faster on shared caches



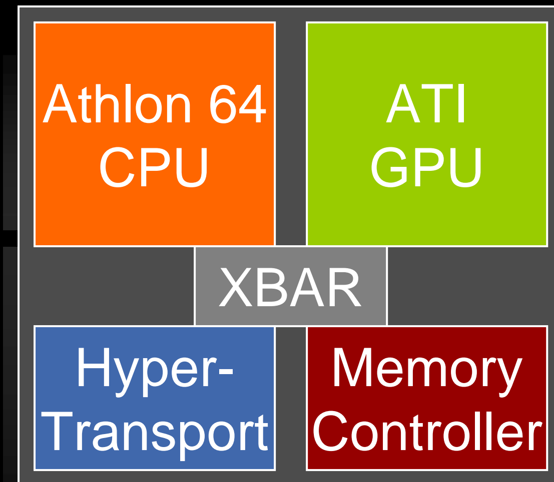
Ongoing Research

- C++ Runtime to make SPPM & Polymorphic Threads programming easier
- Exploration of problem space
  - Media encoding
  - Data-stream handling (gzip)
  - Fine-grain concurrency in applications (protocol processing, I/O, etc.)
- Hardware architecture improvements
  - Better communications between cores

UCIrvine

Future CMP Technology

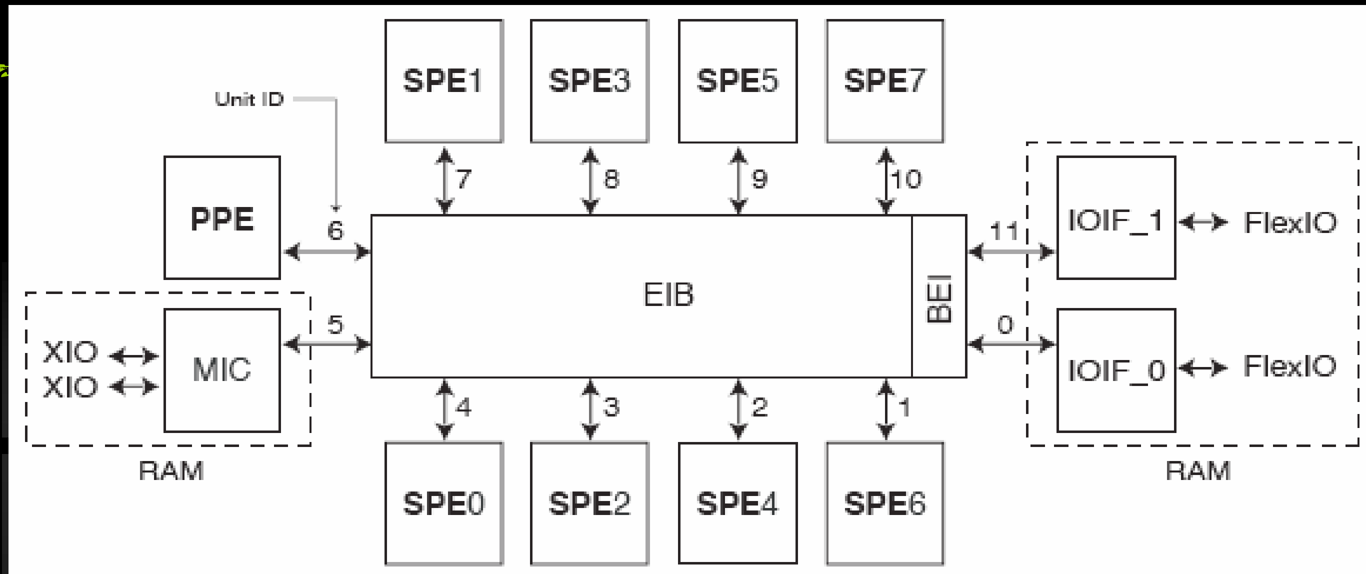
- ✓ 8 cores soon
- ✓ Room for improvement
  - ✓Multi-way caches expensive
  - ✓Coherence protocols perform poorly
- ✓ Stream programming
  - ✓GPU or multi-core
  - ✓GPGPU.org for details



Possible Hybrid AMD Multi-Core Design

What About The Cell Processor?

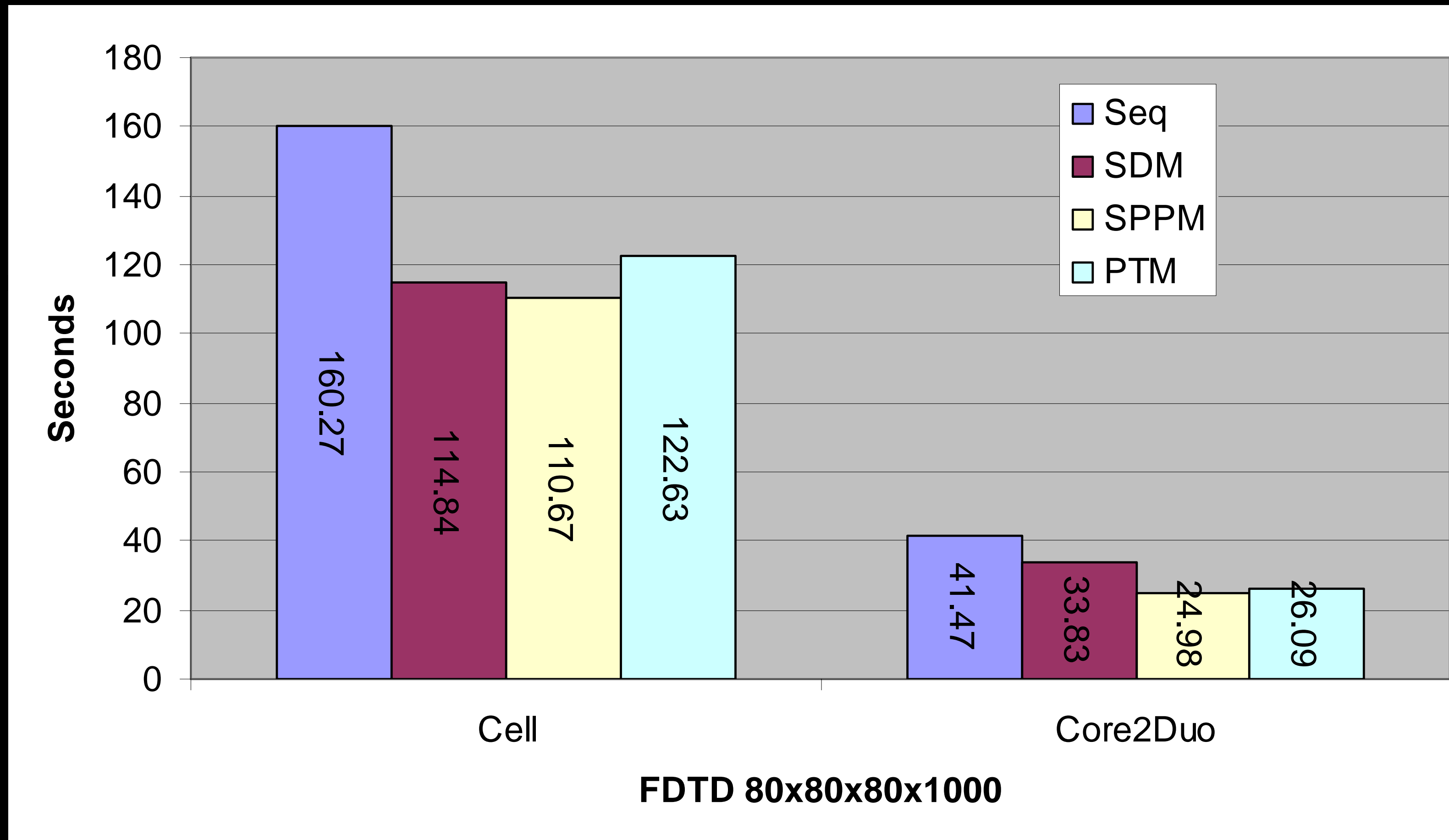
From *IBM Cell Broadband Engine Programmer Handbook*, 10 May 2006



- PowerPC Processing Element with SMT
- 8 Synergistic Processing Elements
  - Optimized for SIMD
  - 256KB Local Storage - no cache
- 4x16-byte-wide rings @ 96 bytes per clock cycle

UCIrvine

CBE PowerPC Performance



Summary

- Parallel microprocessors provide tremendous peak performance
- Need proper programming to achieve it
- Thread programming is not hard, but requires more care
- Architecture & implementation bottlenecks require additional work for good performance
  - Performance is architecture-dependent
  - Non-uniform cache interconnects will become more common

UCIrvine

Additional slides

Spawning Threads

- Initialize Attributes (pthread_attr_init)
  - Default attributes OK
- Put thread in system-wide scheduling contention
  - `pthread_attr_setscope(&attrs, PTHREAD_SCOPE_SYSTEM);`
- Spawn thread (pthread_create)
  - Creates a thread identifier
  - Need attribute structure for thread
  - Needs a function where thread starts
  - One 32-bit parameter can be passed (void *)

Thread Spawning Issues

- How does a thread know which thread it is? Does it matter?
  - Yes, it matters if threads are to work together
  - Could pass some identifier in through parameter
  - Could contend for a shared counter in a critical section
  - `pthread_self()` returns the thread ID, but doesn't help.
- How big is a thread's stack?
  - By default, not very big. (What are the ramifications?)
  - `pthread_attr_setstacksize()` changes stack size

Join Issues

- Main thread must join with child threads (`pthread_join`)
  - Why?
  - Ans: So it knows when they are done.
- `pthread_join` can pass back a 32-bit value
  - Can be used as a pointer to pass back a result
  - What kind of variable can be passed back that way? Local? Static? Global? Heap?