Parallel Computation Trends and Summary

The ideas we have learned are a significant part of today's computing scene--from the HPC level downward. We identify recent trends as a summary of the ideas learned.

Parallel Computation Trends

- Multi-threading -- Tolerate Latency
  - HEP
  - Alewife
  - Tera
  - SMT
- Multi-core Chips -- Use Si Well
- Cell -- Battle Latency Problems, Use Si Well

Latency -- Just Do Something Else

In theory a memory delay of λ is not a show-stopper; simply switch to other work while waiting for a memory value to be returned

Requires (in theory) P log P threads, but it's actually λP+, and grows however λ grows*

Threads are often abundant, but it is difficult always to have λP threads at each moment, e.g. at decision points

---

* Which is a lot, recently

Hardware Implementations

- The idea of switching to execute instructions from another thread when memory latency stalls processor has been around a long time
  - Early Honeywell machines used related ideas
  - Denelcor HEP (~1982) designed by Burton Smith had 8 threads “in the air” at once
  - The 1990s saw several efforts to build such machines
- We will check out two designs
  - Alewife from Anant Agarwal’s group at MIT
  - The Tera Computer from Burton Smith

The Hardware Solutions

- Focus on keeping 1 processor busy in the presence of long latencies to shared memory, but expect to use many such processors
    - Use multithreading
    - Requires no special software as long as the compiler can produce more threads than processors
    - Handles both predictable and unpredictable situations
    - Handles long latencies even as they grow
    - Doesn’t affect the memory consistency model, i.e. shared variables must be locked or use other mechanism

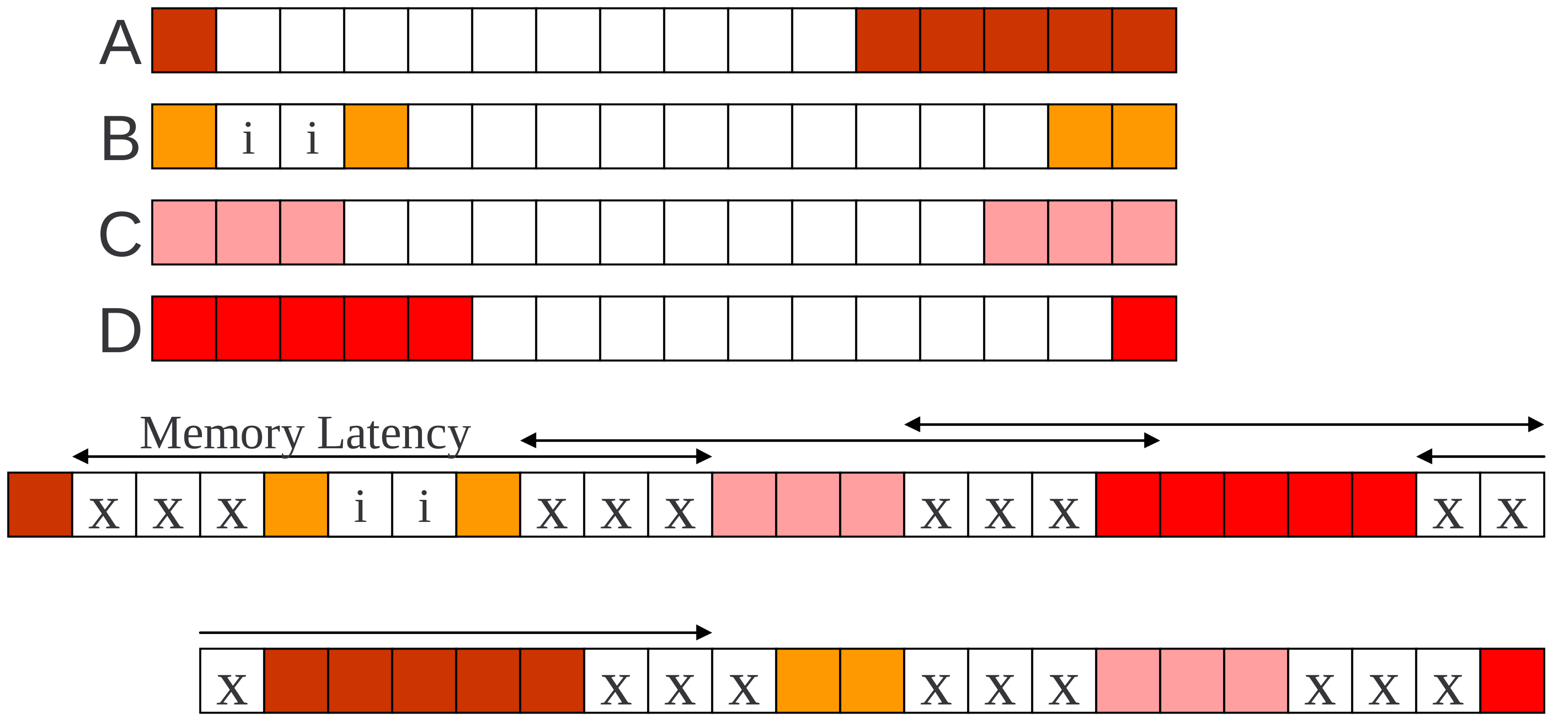
$$\text{utilization} = \frac{\text{work\_time}}{\text{work\_time}+\text{switching}+\text{idle}}$$

Two Techniques for Multi-threading

- Blocked multithreading [Alewife] is like timesharing … continue to execute until the thread is blocked, then switch
  - Has lower hardware impact
  - Good single thread performance
- Interleaved multithreading [Tera] switches execution on threads on each cycle
  - Lower logical switching penalty
  - Greater impact on hardware design
- Keeping multiple contexts is essential

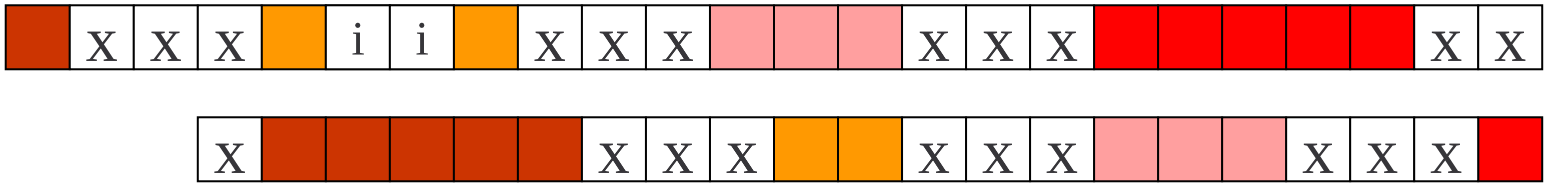
Four Threads Using Blocked Approach

- Threads make memory reference every few instructions -- 3 tick switch penalty



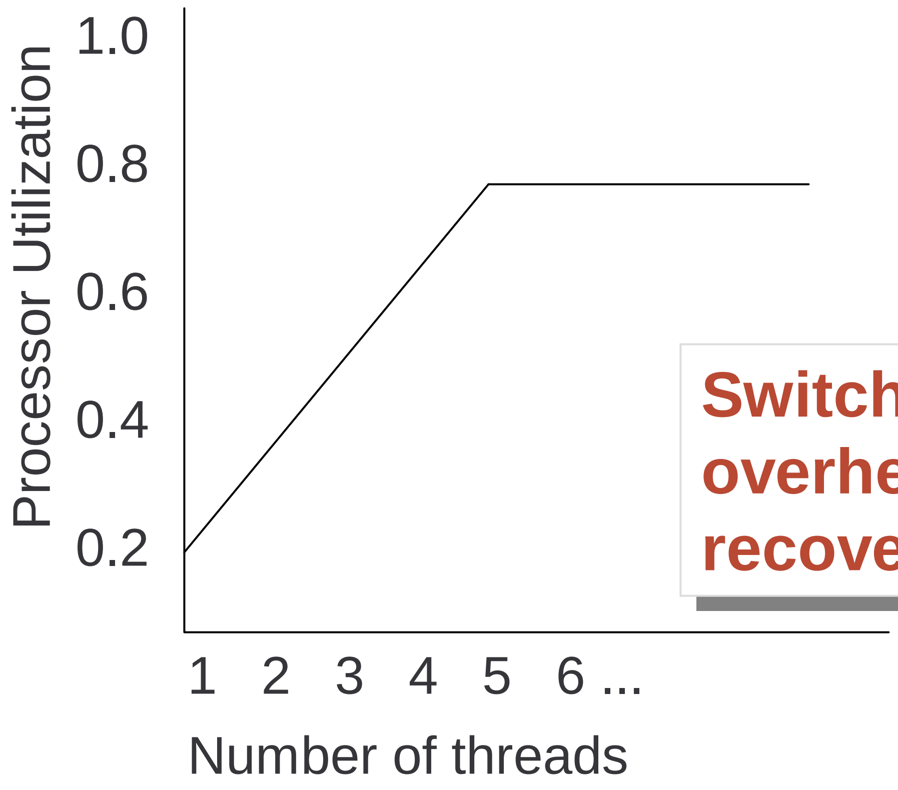
Utilization of Blocked Approach

- Total instruction times: 45
- Total work instructions: 24
- Total switch time: 21
- Total wait time: 0
- Utilization = 24/45 = 53%



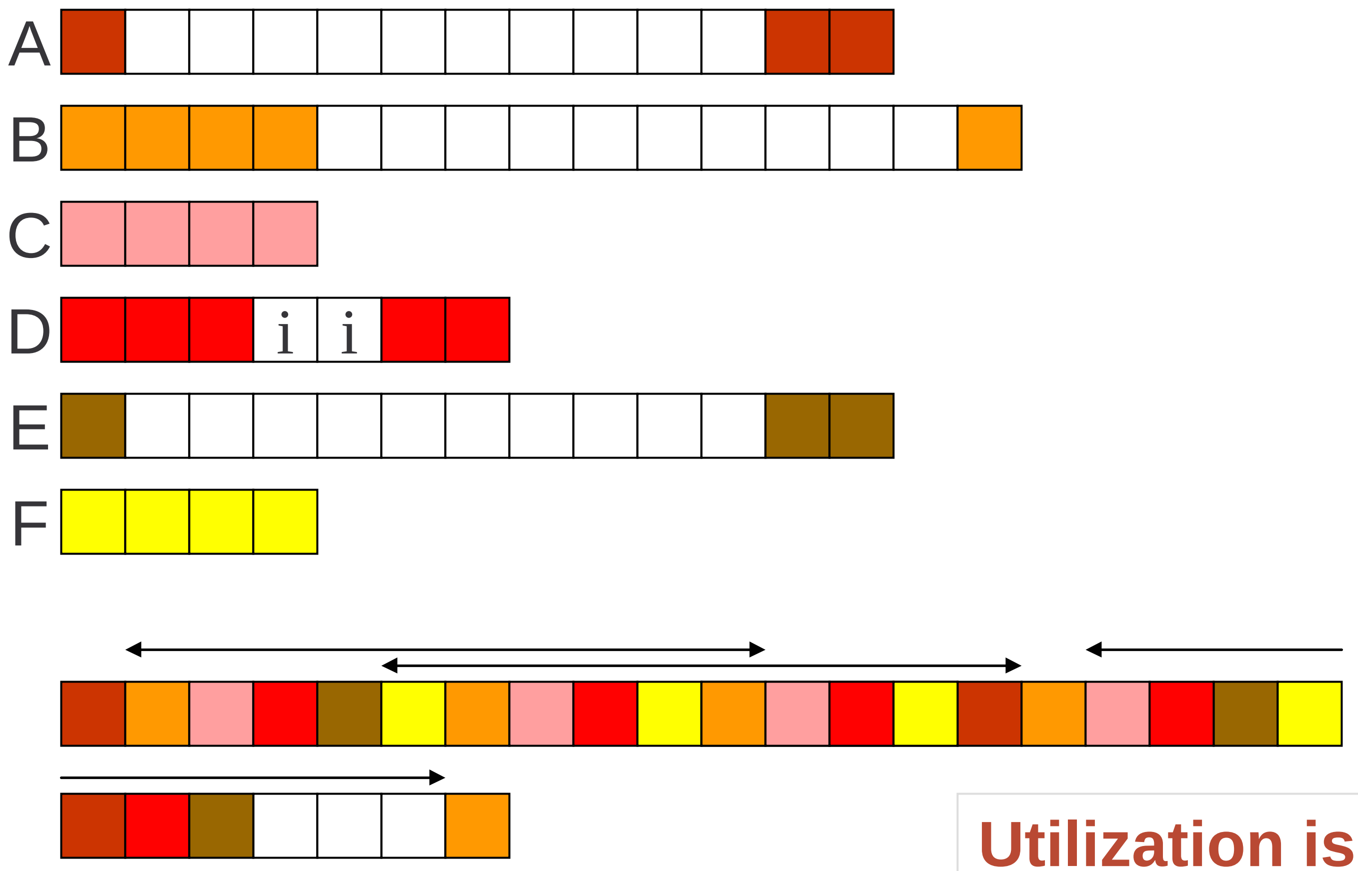
Benefits of Available Threads

- For the blocked approach the availability of ready threads improves utilization



Switching time is overhead that is not recovered

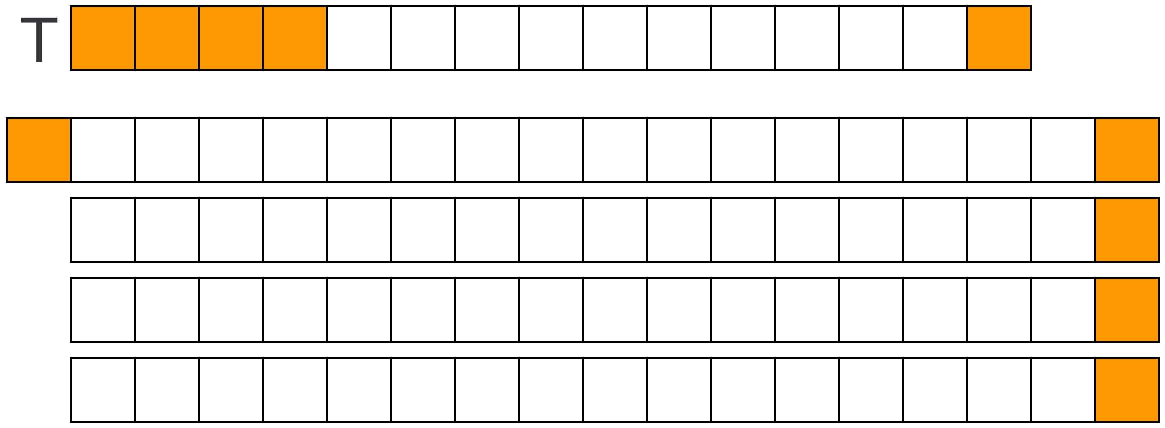
Six Threads With Interleaved Approach



Utilization is 24/27 or 89%

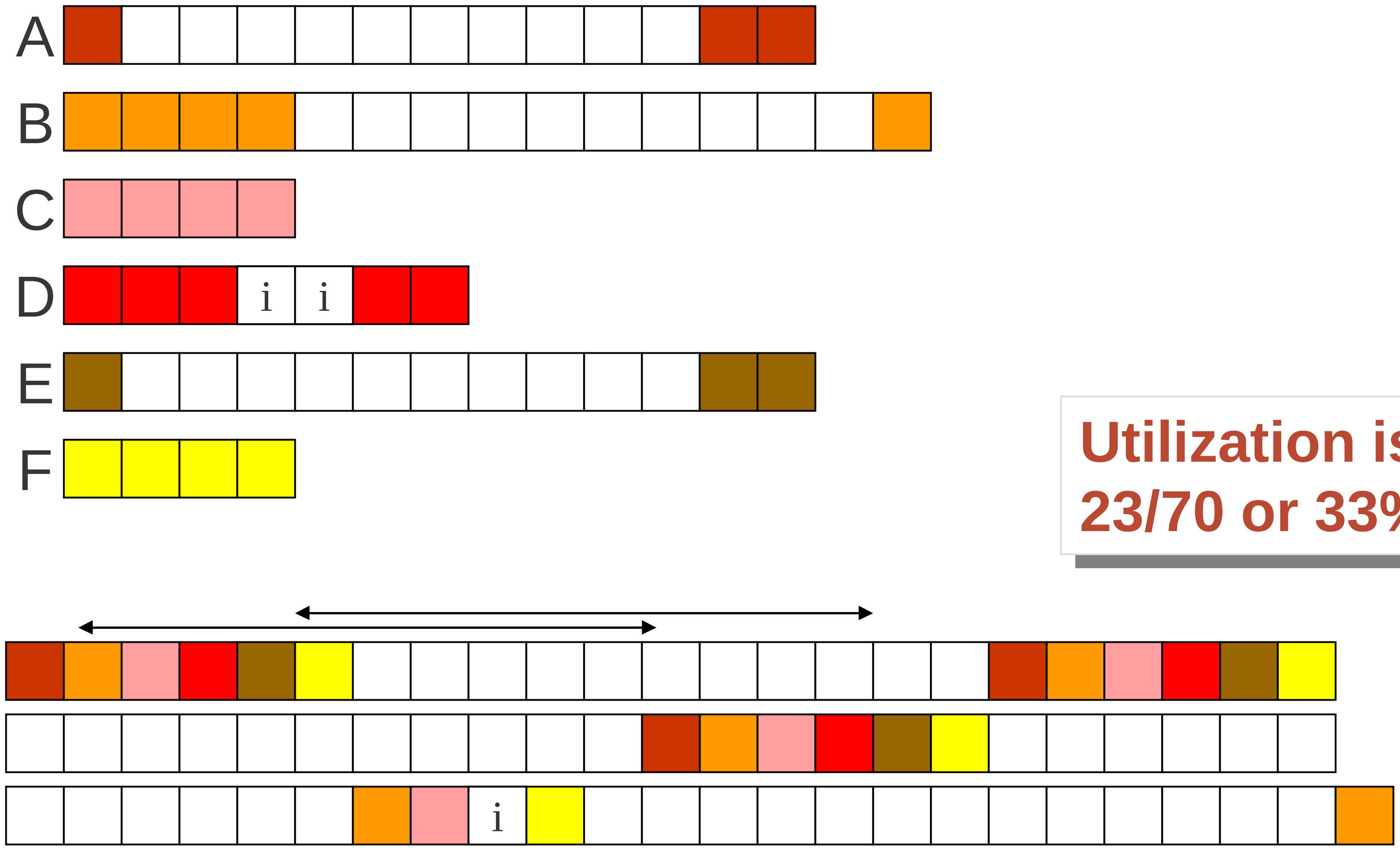
Basics of Tera Design

- Instructions are [arithmetic, control, memory] or [arithmetic, arithmetic, memory]
- Ready instructions issue on each tick, but there is a 16 tick minimum issue delay for consecutive instructions from a thread



Converts latency from 15 ticks to 69 … U = 7%

Six Threads Revisited (Tera Design)



Utilization is 23/70 or 33%

More On Tera

- Since there is a 16 instruction minimum issue delay, it takes 16 threads to execute sequentially *without* latency hiding
- Each (memory) instruction has a 3 bit tag telling how many instructions forward are independent of this memory reference (in this thread)
- Average memory latency without contention is 70 cycles

Still More On Tera

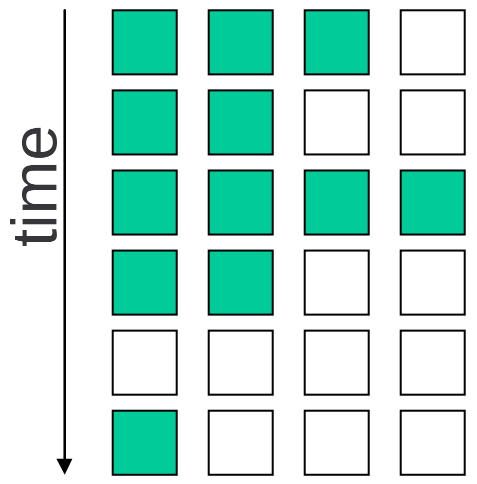
- Each processor has 128 full contexts
- Synchronization latency can even be covered
- When everything works, the Tera should approximate a PRAM

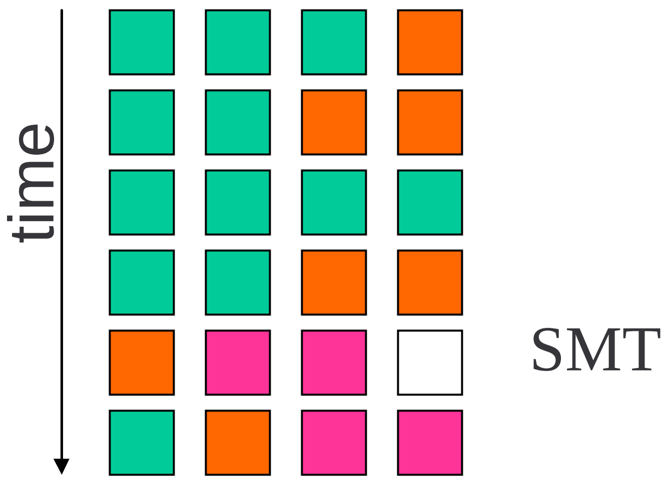
Think of the Bulk Synchronous model with completely decentralized supersteps

SMT (Simultaneous Multithreading)

Parallel computation applied to serial processor

Superscalar machines can execute n instructions per tick--but many slots not used

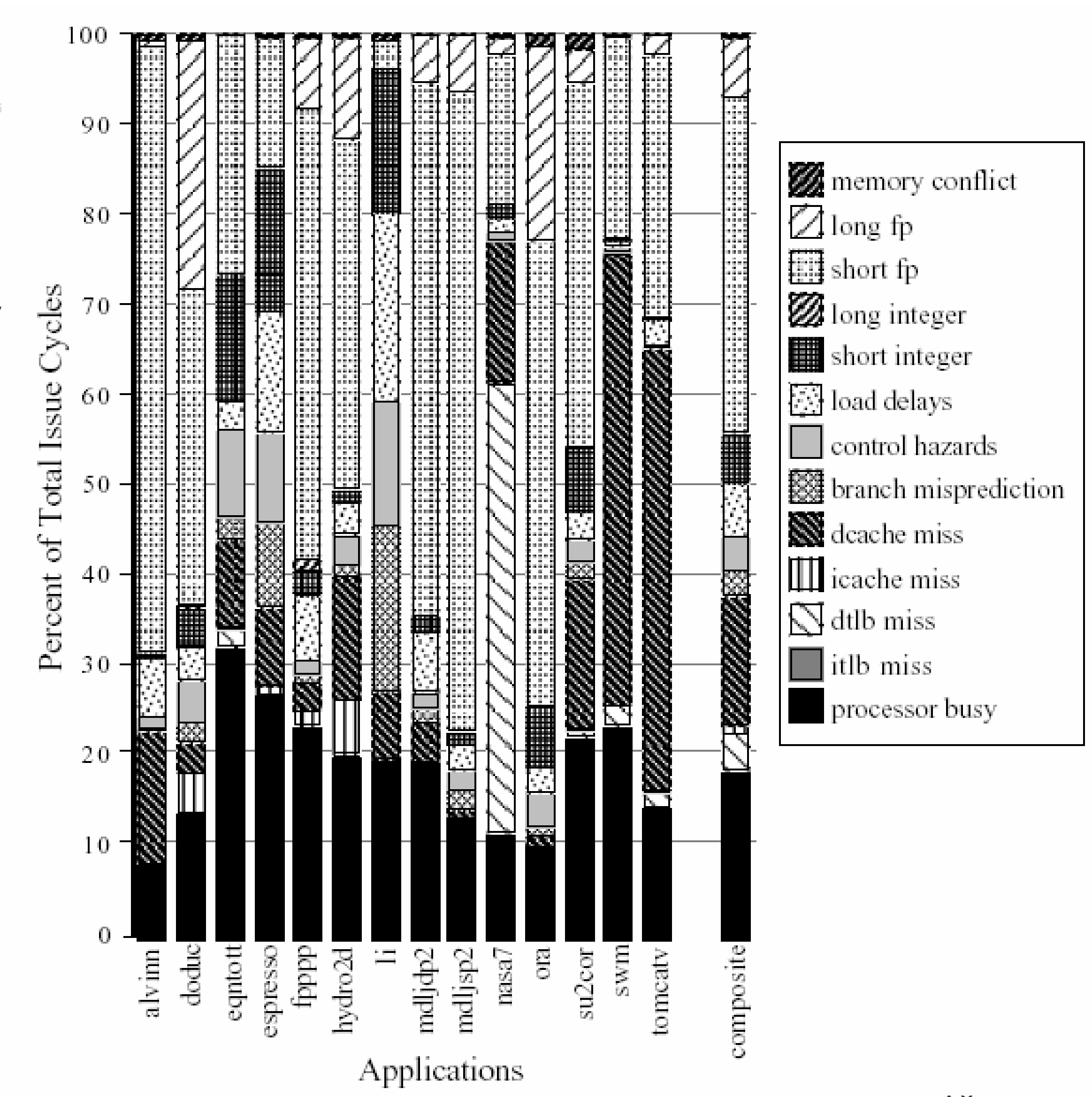




🟩 Instruction execution slot filled by green task

SMT Potential

- Consider 8-issue superscalar … only the black area is useful work
- Processor designs apply SMT



Multi-Core Processor Chips

- Why multi-core?
  - Cynic says, how else can the Si be used?
  - Marketing says, “multi-core advantages include a better ratio of performance to power usage, less heat dissipation, and a smaller physical footprint. One prime market for multi-core SoCs appears to be networking equipment such as firewalls that deeply inspect packets or perform compute-intensive spam filtering.”
- New multi-cores are SMP-on-a-chip
- Gang Broadcom (up to 8) chips together to get ccNUMA through HyperTransport

Cell Computers

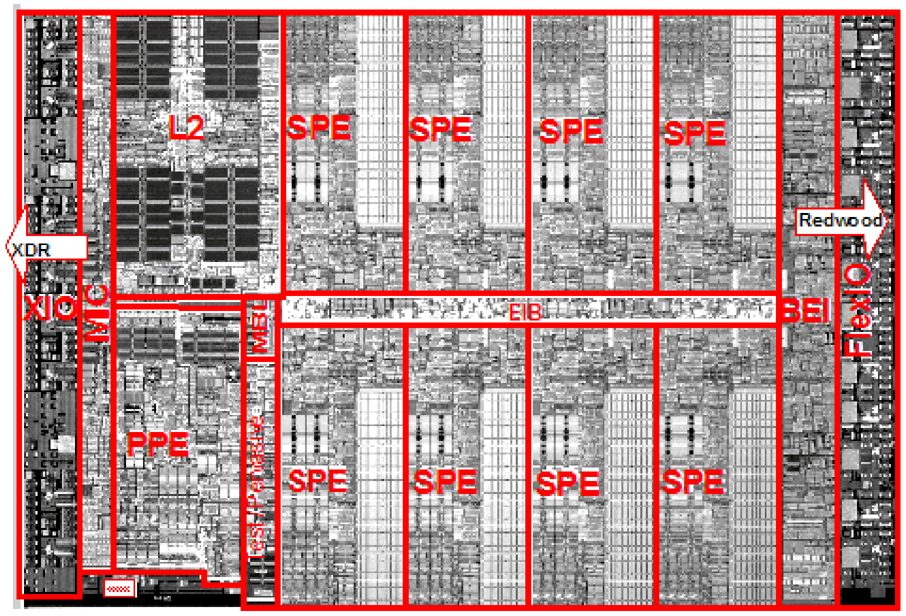
Cell computers were announced in the October '04 to much fanfare, and were hailed as a major advance in performance

- Cooperative design by IBM, Sony and Toshiba
- Details remain sketchy, but basics are clear
  - The design has enormous floating-point power, making it ideal for games, DSP, scientific and HPC computing
  - The processor will become a building block for HPC machines
  - Programming to exploit power will be tough

Performance Is Tied To Interconnect

Cell Chip

DRAM



System Interface

Package has 1236 contacts: 506 are signals

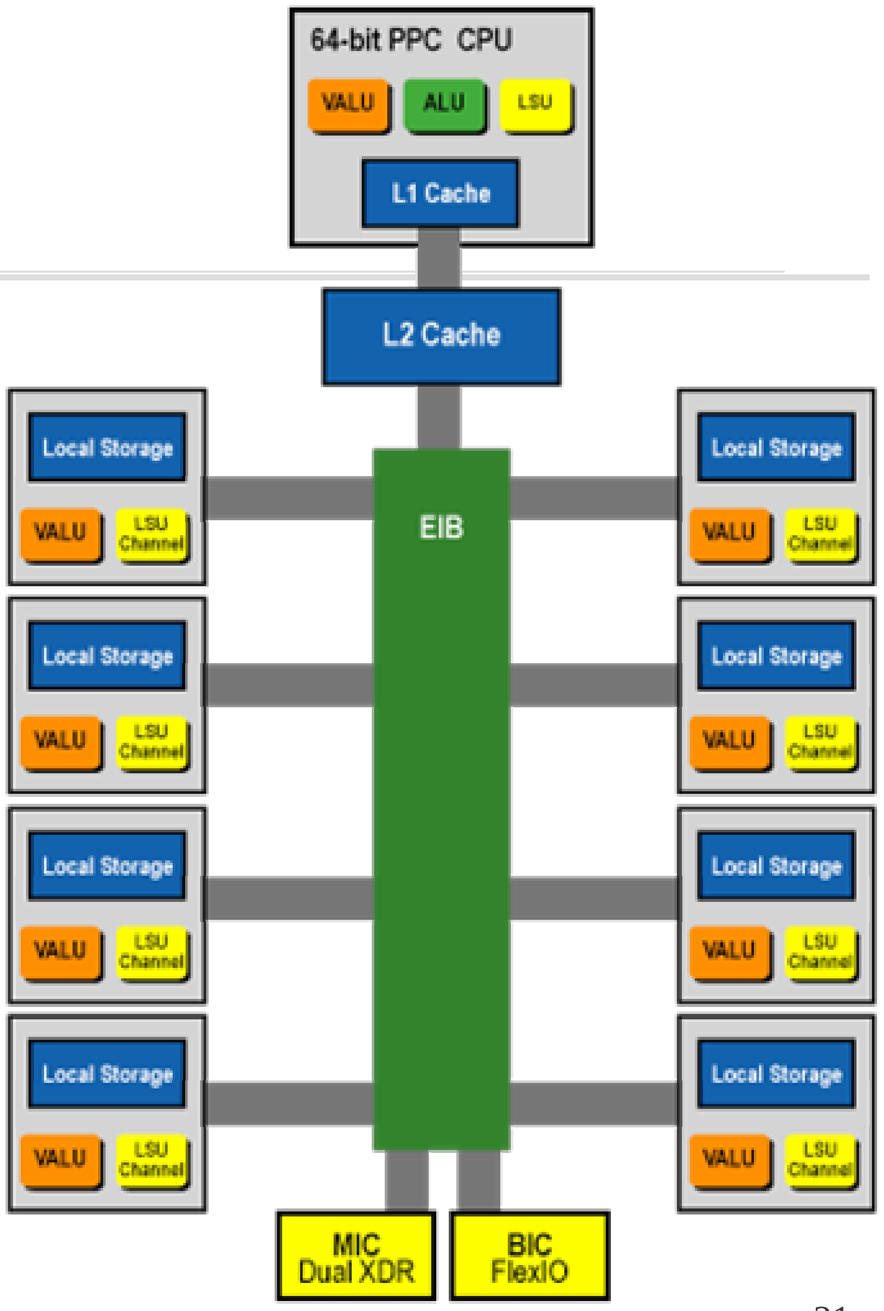
44.8 GB/s Out, 32GB/s In, 25 GB/s Memory

Specs

- Clock speed over 4GHz.
- 100 GBytes per second aggregate Memory & I/O speed:
- Dual XDR controller gives 25.6 GBytes per second.
- Dual configurable interfaces give 76.8 GBytes per second.
- Memory currently limited to 256 MB per Cell (this applies to direct connections only, additional memory can be accessed via I/O) .
- 8 X "SPEs", 128 bit vector engines, 128 registers each.• 2 instructions issued per cycle per SPE.
- Peak = 256 GigaFlops (SP), Double precision math operations supported.
- 256KBytes "Local Store" per SPE.
- Internal communication is via 4 X 128 bit rings, up to 96 Bytes per cycle.
- PPE can handle 2 threads
- PPE includes VMX.
- PPE includes 512 KBytes Cache.
- 234 million transistors• 90nm SOI, Low K, 8 layers of metal & Copper interconnect

Overall Cell Design

- PPC=PowerPC
- PPC CPU=PPE
- LSU=load/store unit
- SPE=SIMD or synergistic proc elem
- EIB=element inter-connect bus



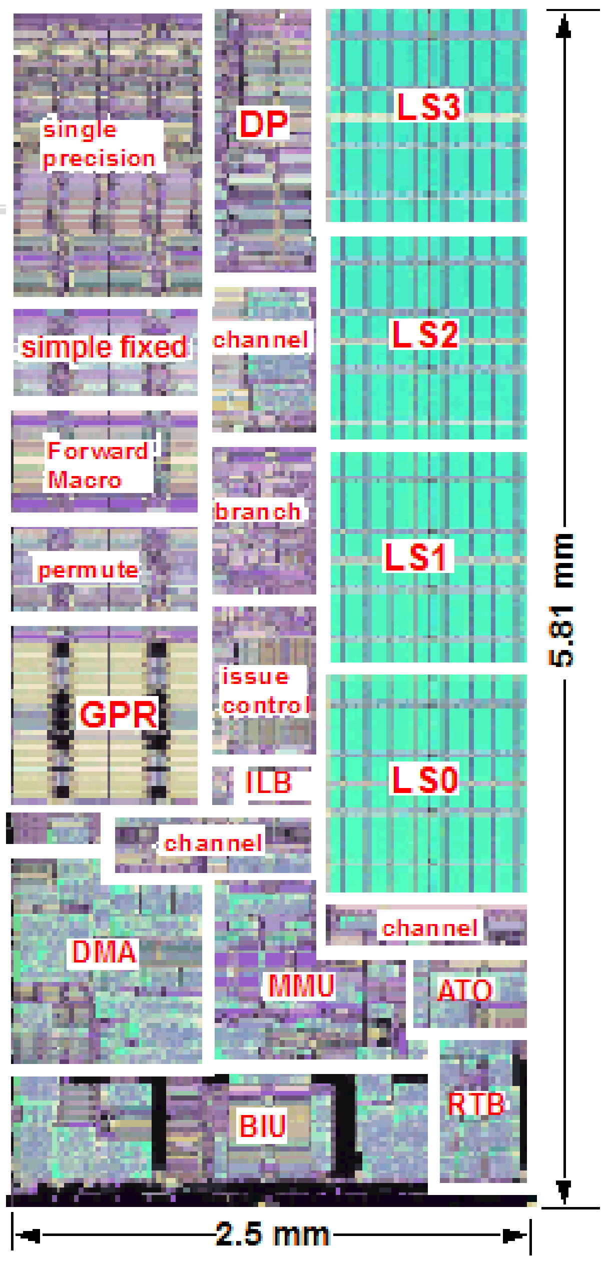
The CELL Architecture

General Purpose Processing

- The PPE is a simplified PowerPC with the usual L1 cache, VM, FPU, etc.
    - Dual-issue in order SMT--probably one issue/thread
    - Very simplified issue logic and pipeline
    - Primary task is to orchestrate the SPEs

SIMD Processing Elements

- 128-bit FP pipelined processor
- 128 GPRs
- Four private memory units with 256KB capacity (total)
  - Separate from system memory
  - Not Coherent
  - 6 cycle latency, SRAM
- Smallest addressable unit is 1024 bits
- 18 cycle branch prediction latency
- VMX ISA (nonstandard)
- Each instruction sources 3 operands and produces 1 result



Summarizing CSE524

- But, first, a break

Our Original Goal

Goal: To give a good idea of parallelism

- Concepts -- look at problems with “parallel eyes”
- Algorithms -- different resources to work with
- Programming -- describe the computation without saying it sequentially
- Languages -- reduce control flow; increase independence
- Architecture -- HW support to share memory not?
- Hardware -- the challenge is communication, not instruction execution

Start with HW, and review (pop) to Concepts ...

Routing

- Chaos routing is effective because “non-minimal” adaptivity can by-pass congestion
  - Light traffic, randomize routes over a regular, symmetric, consistent networks, avoids creating hot spots; no point where packets can get “stuck”
  - Moderate traffic, wait in a node for a route to clear, this is better than “hot potato” which *must* route
  - Heavy traffic and faults, deroute in the wrong direction to move around the problem

Higher throughput, lower latency, higher load-carrying capacity than other routers

Chaos Routing (continued)

- Deadlock is not possible because of packet exchange protocol
- Probabilistically livelock-free,
  - As good or better than deterministically livelock-free in practice
  - Solves difficult (but rare) problem for adaptive routers by randomizing, and gambling
- Chaos is not perfect; not good with wormhole
  - Inefficient for long messages; use two nets or pick a variable length packet with large-ish maximum

Networks

- Full cross-bar is not practical
- Direct and Indirect Networks are alternatives
  - Indirect, e.g. Ω-network
    - Has only "long" paths of O(log P), no nearest neighbor
    - Multiple references to a location can collide, so try combining at the switches
    - In fact, exploit combining with Fetch&Add -- it's better for shared memory than Test&Set because it "schedules"
    - Both Fetch&Adds and Load/Stores can be combined
    - Combining requires "smart" switches that slow net
    - Analysis shows combining opportunities are rare; hot spots due to colliding references to different locations is the problem

F&A + combine is smart but flawed

Networks (continued)

- Direct Networks
  - Short, nearest neighbor paths are available
  - Adaptive routing techniques are available
  - Much more asynchronous; NIC is extra processor
  - Different load properties for different architectures
    - Non-shared memory, network carries little overhead
    - Shared memory, network carries coherency protocol messages, which can be “proportional to the sharing”
  - n-ary, d-cubes are realistic topologies
    - Torus is better because of symmetry
    - Fat trees also work; hypercube has “log P node degree”

Direct (regular) networks are only choice

Architecture

Main architecture decision: hardware support for shared memory or not?

- Non-shared memory architectures are successful
  - Simpler designs, means faster designs
  - Leave memory management to software/programmer
  - A single address space is easy and useful
  - “Proper HW support for shared memory” is still unknown and getting less realistic as technology improves
  - Avoid message passing and its copy/marshal overhead
  - One-sided communication (shmem) is very efficient because it reduces communication’s synchronization
  - Shmem allows “strided communication” with pipelining

Single address space, 1-sided communication is best

Architecture (continued)

- Shared memory
  - Technically very difficult -- and therefore slow -- to keep memory coherent
  - DSM can be implemented by a directory scheme
    - Record sharers/dirty features for each cache line
    - Directory can double the memory requirements of machines, though some simplifications are possible
    - Follow distributed version of coherency protocol because no bus for defining “timing sequence”; use mem location
    - Invalidations, acknowledgements increase with sharing

DSM best when not used, i.e. manage mem yourself

Architecture (continued)

- Symmetric-multiprocessors (SMPs) are an effective way to share memory on a small scale
  - Cache controllers snoop memory bus
  - The bus becomes the “time sequencing” point of the system, where modification order is defined
  - Various protocols speed performance with greater complexity
  - Bus is *serially* used, limiting generalization to small #s

SMP is a standard architecture

Languages

- Shared memory is difficult to use (races, synch); not efficiently implemented
- Message passing with sequential language (C | Fortran) + (MPI | PVM) is current standard
  - Least common denominator -- runs everywhere
  - A huge amount of work (6x code explosion)
  - Only the API is standard; semantics vary, making any program implementation-specific; limit portability
  - Message passing is costly on architectures with “good” communication, e.g. 1-sided, SMP

Use msg passing if you must; but there’s a better way

Languages (continued)

- ZPL 1st (and still only) parallel language with performance model (WYSIWYG)
  - Designed from first principles to help programmers
  - No explicit concurrency, communication, or synch.
  - Programmer is insulated from details, but it is possible to write efficient solutions with WYSIWYG
  - Compiler is heavily optimized, both seq. and para.
  - The communication abstraction is Ironman -- four procedures that mark sender's/receiver's active regions -- uses native communication of machine

ZPL is convenient and efficient

Languages (continued)

- ZPL’s performance model
  - Allows programmers “to keep their distance” from the implementing hardware -- portable!
  - Relies on abstract machine -- CTA
    - CTA gives key structural information, memory reference time, processors, characteristics of interconnection net
    - CTA gives parallel costs; vN defines sequential costs
    - Give ZPL’s runtime model, work & data allocation
    - Describe costs of ZPL’s constructs in CTA terms
  - No absolute performance possible, but relative is good enough for quality programming -- performs!

The most significant idea of this class

Programming

Everyone thinks shared memory is the natural parallel extension of sequential computing: "Ignore memory reference time like vN model, and let HW give the flat memory illusion"

- Memory reference time is key to good algs:
  - Find maximum is the example
    - Best ignore-memory-time (PRAM) is Valiant O(log log P)
    - Best consider-memory-time (CTA) is tournament O(log P)
    - (Actual?) implementation of Valiant's alg O(log P loglog P)
    - Actual implementation of tournament O(log P)

PRAM hides a critical cost => it's hard to get results

Programming (continued)

- The CTA replaces the PRAM as a realistic, but still abstract model of parallel computation
  - CTA models all existing hardware, but is “far enough away” to be independent of all
  - CTA is concerned with a few features, processors, non-local memory reference time, λ, interconnect, which has unspecified topology, low degree
  - Practicing programmers writing message passing code are in effect using the CTA
  - CTA is key to expressing costs of HLL like vN

A machine model separates SW & HW development

Concepts

- The powerful parallel computation ideas are:
  - Pipelining, perform some operations and then pass the task along for completion by other units
  - Overlap, perform communication & computation simultaneously since they need separate resources
  - Partition, form independent (as possible) tasks and assign separate processors to each
  - Most parallel algorithms use a combination of these
    - Languages should support these concepts
    - ZPL does overlap and partitioning for all computations up to available resources, and has abstraction for pipelining

More abstractly: Decompose into independent parts

Concepts (continued)

- Matrix multiplication -- the most studied parallel algorithm
  - Many solutions; van de Geijn,Watts SUMMA best
    - Uses broadcast communication of rows/columns
    - Restructures the problem to “use data completely”
    - Efficiently uses temporary space
    - Most natural and convenient (and efficient) ZPL program
    - Other algs show ‘problem space promotion’ technique

Problem space promotion is a parallel programming technique in which a problem with d dimensional data d is *logically* solved in a higher dimension, usually d+1

Avoid iteration

Concepts (continued)

- Summary for successful parallel computation
  - Rather than using a shared memory abstraction, use the CTA model; it reflects costs accurately
  - Use ZPL for programming to get convenience, speed and portability; use MP as last resort
  - Be suspicious of claims like the “problems” with shared memory have been solved by new machine
  - When choosing architecture, prefer support for global addressing, 1-sided communication, point-to-point network, (randomizing) non-minimal adaptive routing, SMP nodes

The perfect parallel machine has yet to be built

Summary’s Summary

- This has been a very enjoyable class to teach
- Good luck with the remainder of your MS degree