Latency Tolerance

Topics

- Reducing communication cost
- Multithreaded processors
- Simultaneous multiple threading

Reducing Communication Cost

- Reducing effective latency
- Avoiding Latency
- Tolerating Latency

- Communication latency
  vs. synchronization latency
  vs. instruction latency
- Sender initiated
  vs receiver initiated communication

Examples

```
for (i = 0; i <= N; i++) {
   compute A[i];
   write A[i];
   compute other stuff;
   send A[i];
}
```

```
for (i = 0; i <= N; i++) {
   receive myA[i];
   compute myA[i];
   compute other stuff;
}
```

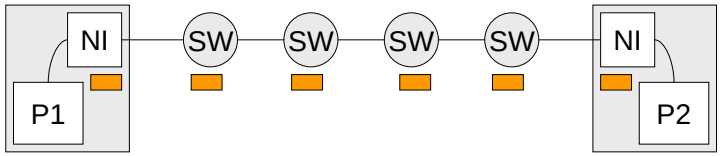
Message passing

```
for (i = 0; i <= N; i++) {
   compute A[i];
   write A[i];
   compute other stuff;
}
```

```
for (i = 0; i <= N; i++) {
   read A[i];
   use A[i];
   compute other stuff;
}
```

Shared address space

Communication Pipeline



- P1 send,
  NI buffer, NI send, SW stage, … , SW stage,NI recv,
  P2 recv
- Send overhead
  vs. time between switches
  vs. receive overhead

Approaches to Latency Tolerance

- Block data transfer
  - Combine multiple transfers into one
  - Why is this helpful?
- Precommunication
  - Generate communication before it is actually needed (asynchronous prefetching)
- Proceeding past an outstanding communication event
  - Continue with independent work in same thread while event outstanding (more asynchronous)
- Multithreading - finding independent work
  - Switch processor to another thread

Another Example

```
for (i = 0; i <= N; i++) {
   compute A[i];
   write A[i];
   send A[i];
   compute B[i];
   write B[i];
   send B[i];
   compute other stuff;
}
```

```
for (i = 0; i <= N; i++) {
   receive myA[i];
   compute myA[i];
   receive myB[i];
   compute myB[i];
   compute other stuff;
}
```

Methods

- Merge multiple sends into one
- Asynchronous send
- Asynchronous receive (provide buffer early)

Fundamental Requirements

- Extra parallelism
- Bandwidth
- Storage
- Sophisticated protocols
  or automatic tools
  or architectural support

Why Multiple Threads?

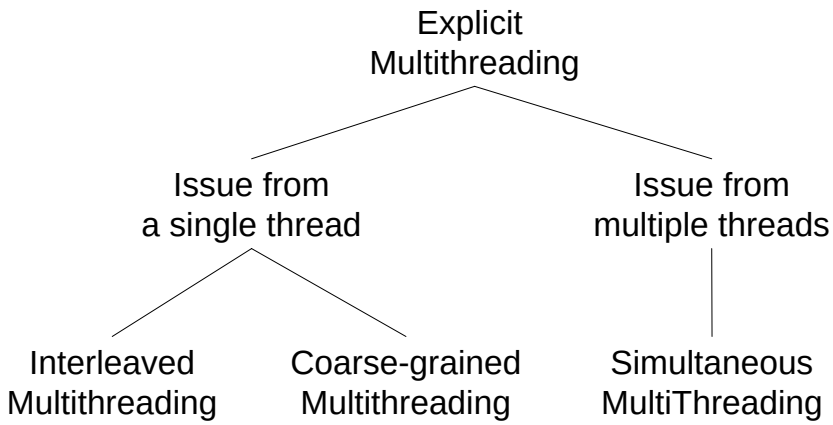
- Power wall
  - Transistors are free, but power is not
  - Many simpler cores with lower clock rates yield better performance/watt
- ILP wall
  - Diminishing return on superscalar
  - Multi-threaded apps get better performance/chip
- Memory wall
  - Gap between CPU and memory access time is increasing exponentially
  - Multiple threads can hide memory latency more effectively than OOO single thread

All “Walls” lead us towards “multiple threads” and multicores

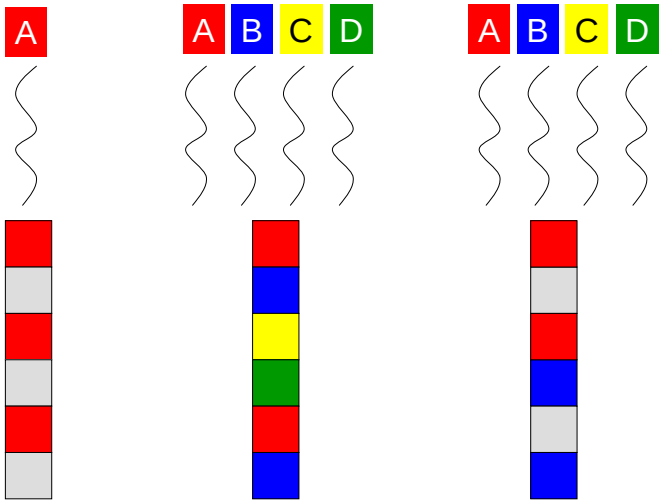
Beyond Simple Multi-core Design

- Latency reduction is important in processor designs
  - Latencies are variable and dependencies are complex
  - Resource contentions
  - L1: 2-4 cycles; L2: ~10 cycles; DRAM: ~200 cycles
- Latencies in shared-memory multiprocessors
  - Remote memory accesses cost much more than local
  - Remote transactions in coherence protocol: 10-100x
- More transistors are available
  - Simple cores may have multiple issues

Classification of Multithreading

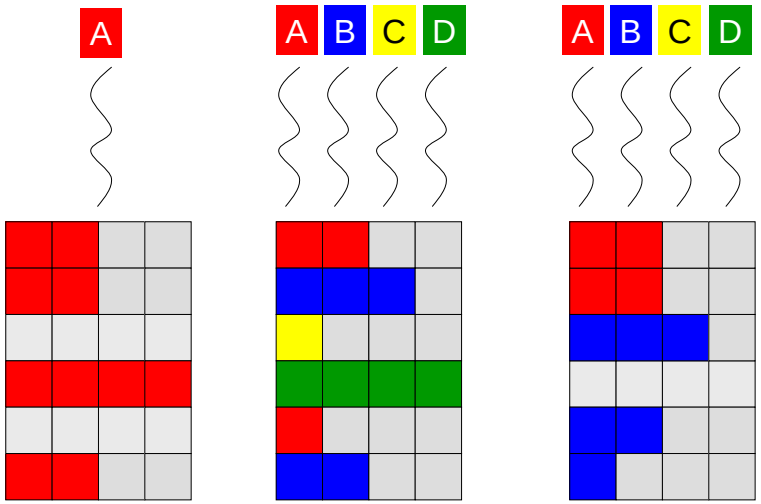


1 thread vs. Interleaved vs. coarse-grained

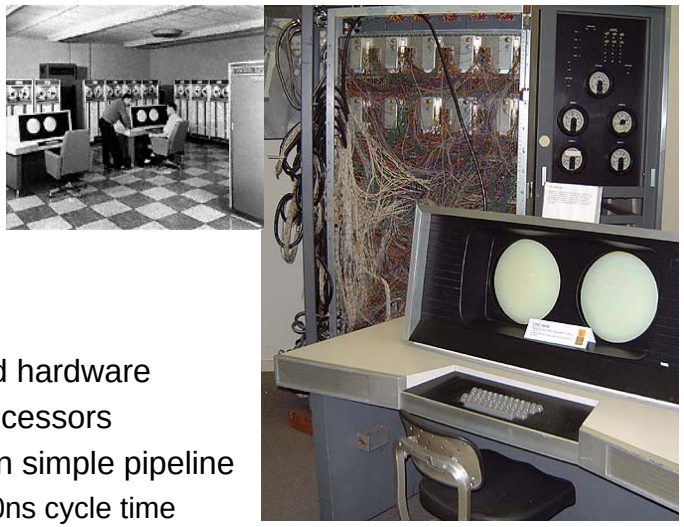


- 1 thread: dependencies limit HW utilization
- Interleaved instruction streams improve HW utilization

Multiple Issue: 1 thread vs. IMT vs. BMT

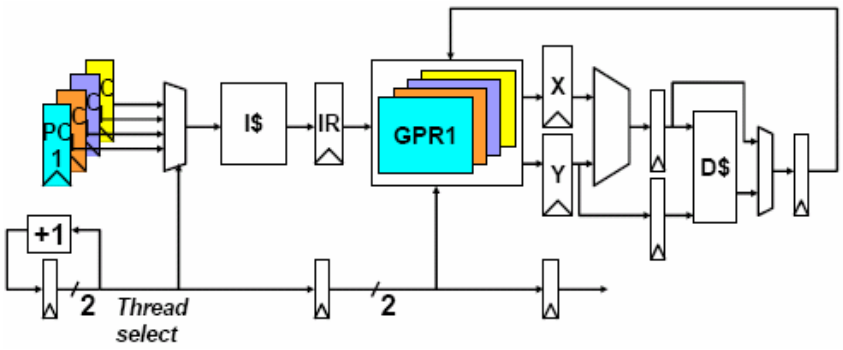


CDC 6600 Peripheral Processors (Cray, 1965)



- First multithreaded hardware
- 10 “virtual” I/O processors
- Fixed interleave on simple pipeline
  - Pipeline has 100ns cycle time
  - Each processor executes one instruction / 10 cycles
  - accumulator-based instruction set to reduce processor state

Simple Multithreaded Pipeline



- Thread select drives the pipeline to ensure correct state bits read/written at each pipe stage
- If there is no ready thread to select, insert a bubble

Multithreading Costs

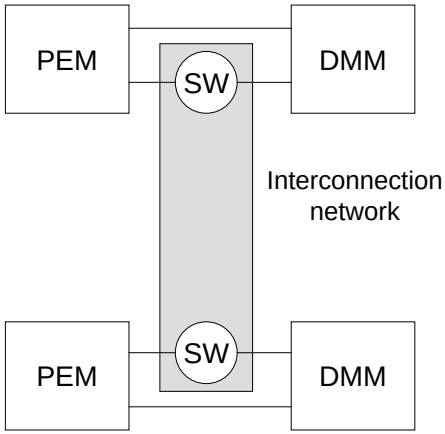
- Appears to software (including OS) as multiple slower processors
- Each thread requires its own user state
  - GPRs
  - PC
- Also, needs own OS control state
  - virtual memory page table base register
  - exception handling registers

HEP (Heterogeneous Element Processor)

- Burdon Smith at Denelcor (1982)
- Parallel machine
  - 16 processors
  - 128 threads per processor
  - Share registers
- Processor pipeline
  - 8 stages
  - Each thread per stage
  - Switch to a different thread every clock cycle
  - If thread queue is empty, schedule the independent instruction from the last thread
  - No need to worry about dependencies among stages

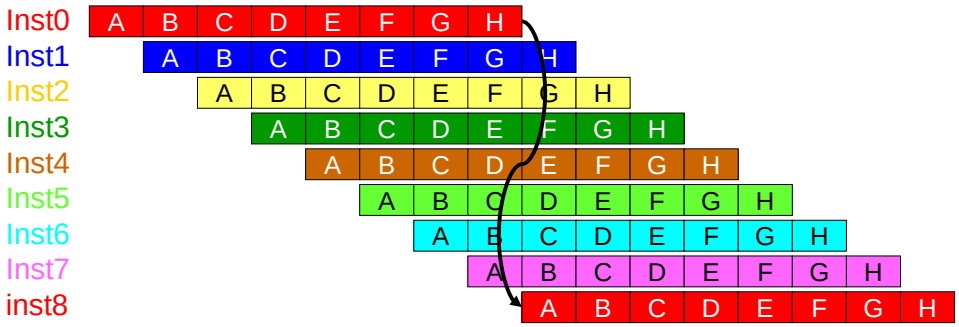
HEP Architecture in more detail

- Basic components
  - PEM: Processing Element Module
  - DMM: Data Memory Module
  - Interconnection network is multi-stage
- How things work
  - Each PEM has 2k registers
  - Each PEM has a DMM
  - Any PEM can access any memory (all physical)
  - Any PEM can access any registers
- Full/empty bit
  - Each word has a F/E bit
  - Empty: no data
  - Full: valid data
  - Read memory w/ empty bit causes a stall or an exception
  - Why is this useful?



Instruction Latency Hiding

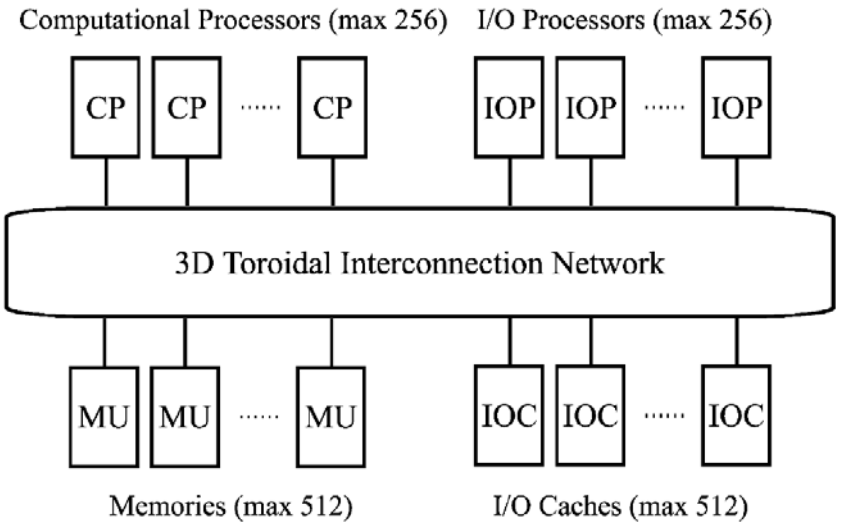
- Every cycle an instruction from a different thread is launched into the pipeline
- Worst case DRAM access might be many cycles (more threads)
- How to balance CPU and Memory?



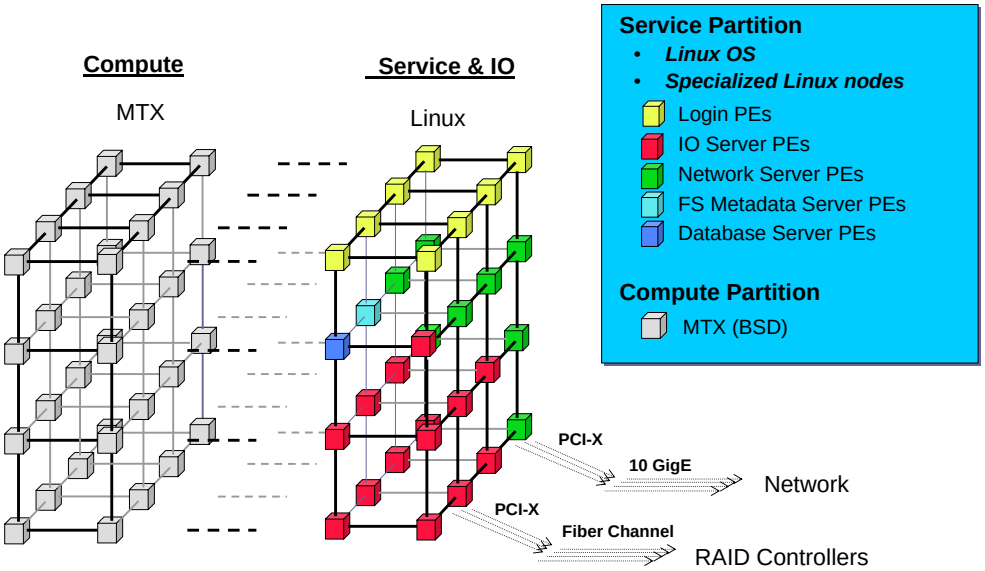
Horizon (Paper design 1988)

- Basic components
  - Up to 256 processors
  - Up to 512 memory modules
  - Internal network 16 x 16 x 6
- Processor
  - Up to 128 active threads per processor
  - 128 register sets
  - Context switch every clock cycle
  - Allow multiple memory accesses outstanding per thread

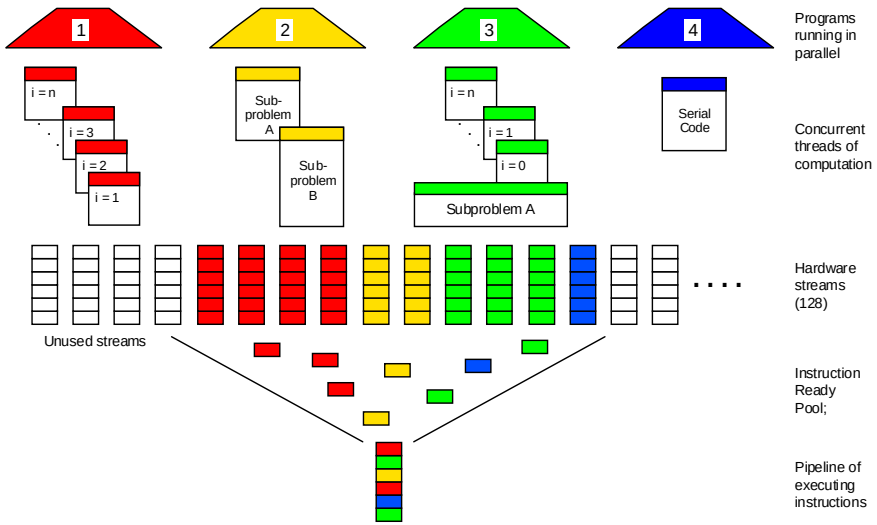
Tera/Cray MTA (1990-)



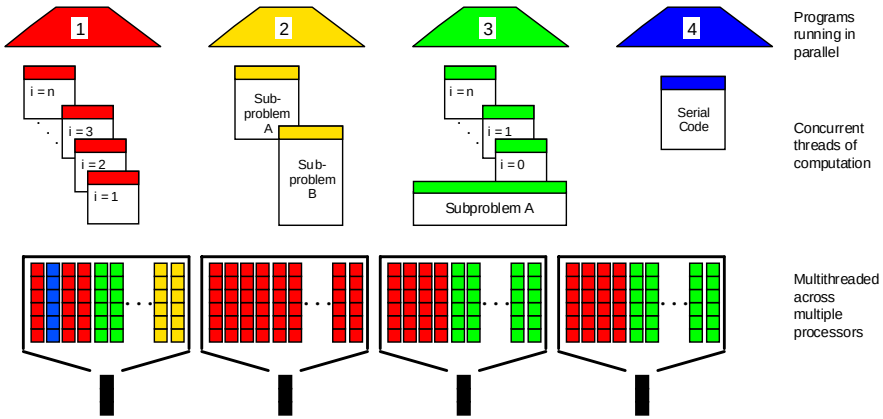
MTX (evolved from MTA) System Architecture



MTA/MTX Processor (from Cray)



MTA/MTX System (from Cray)

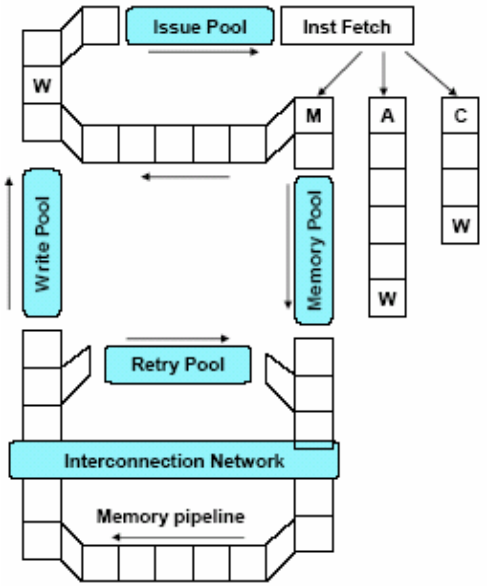


MTA/MXT Processor

- Each processor supports 128 active threads
  - 1 x 128 status word registers
  - 8 x 128 branch-target registers
  - 32 x 128 GP registers
- Each 64-bit instruction does 3 operations
  - Memory (M), arithmetic (A), arithmetic or branch (C)
  - 3-bit lookahead field indicating # of independent subsequent instructions
- 21 pipeline stages
  - Each stage does a context-switch
- 8 outstanding memory requests per thread

MTA Pipeline

- Every cycle, an instruction of an active thread is issued
- Memory operation incurs about 150 cycles
- Assuming
  - A thread issues 1 instruction/21 cycles
  - 220 Mhz clock
- What's the performance?

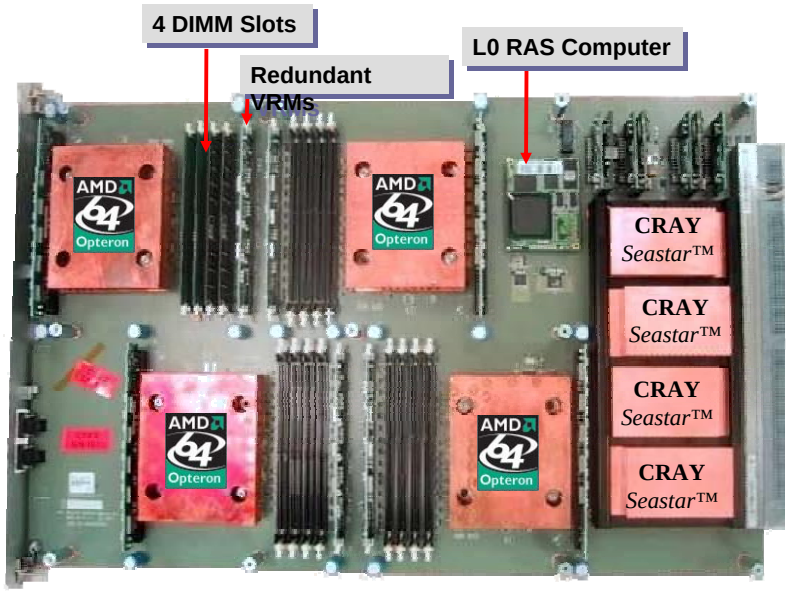


MTA-2 / MXT Comparisons (from Cray)

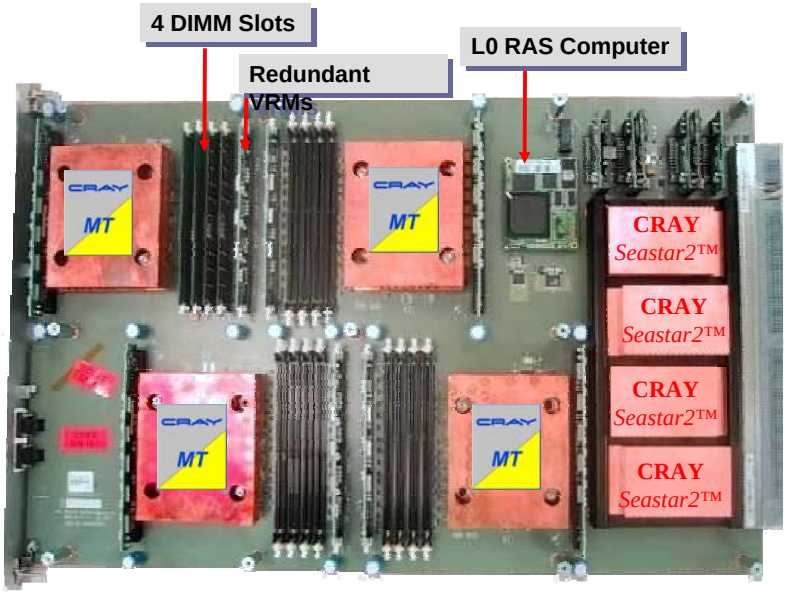
	MTA-2	MXT
CPU clock speed	**220 MHz**	**500 MHz**
Max system size	**256 P**	**8192 P**
Max memory capacity	**1 TB (4 GB/P)**	**128 TB (16 GB/P)**
TLB reach	**128 GB**	**128 TB**
Network topology	**Modified Cayley graph**	**3D torus**
Network bisection bandwidth	**3.5 * P GB/s**	**15.36 * $P^{2/3}$ GB/s**
Network injection rate	**220 MW/s per processor**	**Variable (next slide)**

How many threads can the largest MXT support?

Red Storm Compute Board (from Cray)

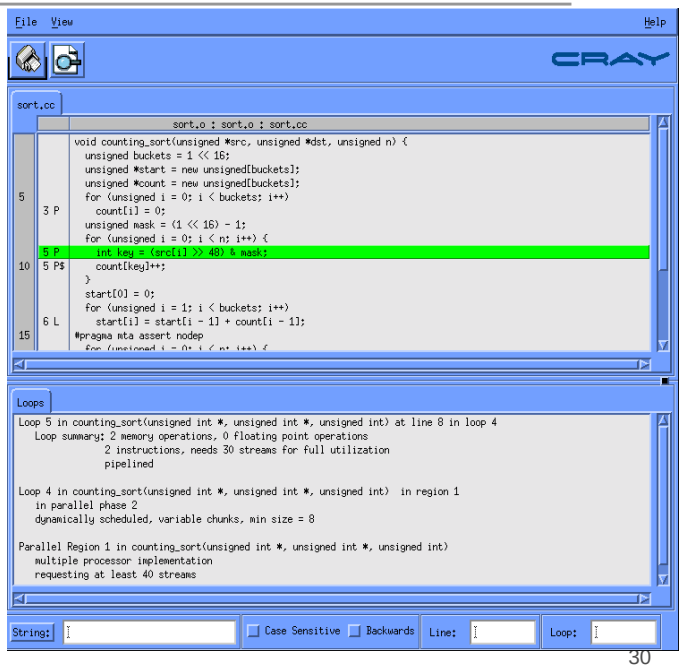


MTX Compute Board (from Cray)

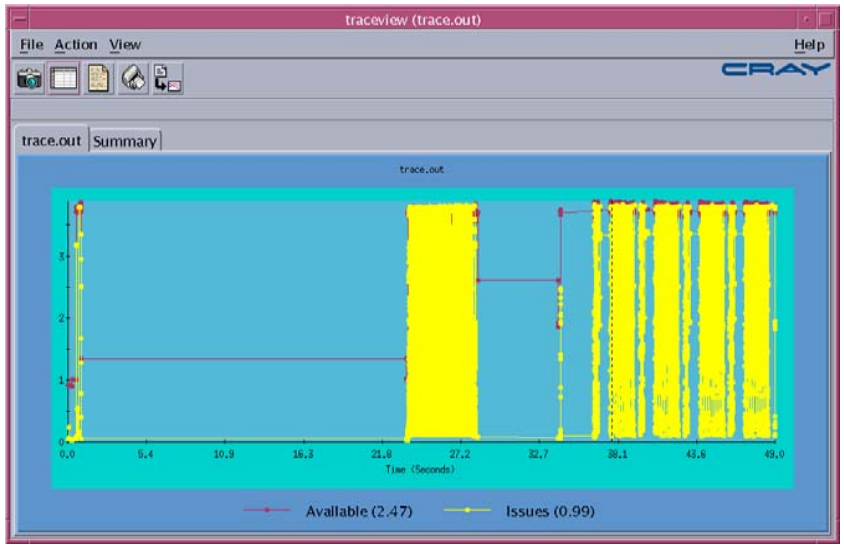


CANAL

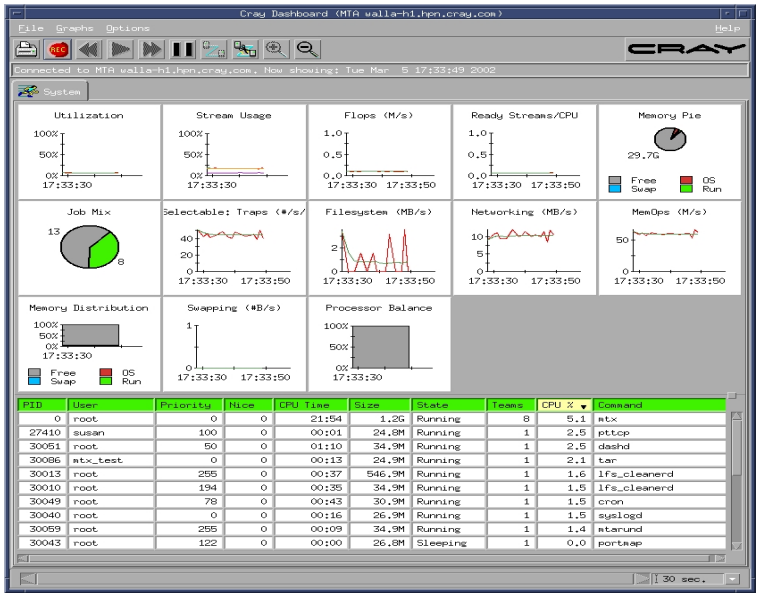
- Compiler ANALysis
- Static tool
- Shows how the code is compiled and why



Traceview



Dashboard



Sparse Matrix – Vector Multiply

- **C** $_{n \times 1}$ = **A** $_{n \times m}$ * **B** $_{m \times 1}$
- Store **A** in packed row form
  - **A**[*nz*], where *nz* is the number of non-zeros
  - **cols**[*nz*] stores the column index of the non-zeros
  - **rows**[*n*] stores the start index of each row in **A**

```
#pragma mta use 100 streams
#pragma mta assert no dependence
for (i = 0; i < n; i++) {
    int j;
    double sum = 0.0;
    for (j = rows[i]; j < rows[i+1]; j++)
        sum += A[j] * B[cols[j]];
    C[i] = sum;
}
```


Canal Report

```
       | #pragma mta use 100 streams
       | #pragma mta assert no dependence
       | for (i = 0; i < n; i++) {
       |     int j;
 3 P   |     double sum = 0.0;
 4 P-  |     for (j = rows[i]; j < rows[i+1]; j++)
       |         sum += A[j] * B[cols[j]];
 3 P   |       C[i] = sum;
       |   }
```

```
Parallel region   2 in SpMVM
       Multiple processor implementation
       Requesting at least 100 streams

Loop   3 in SpMVM at line 33 in region 2
       In parallel phase 1
       Dynamically scheduled

Loop   4 in SpMVM at line 34 in loop 3
       Loop summary: 3 memory operations, 2 floating point operations
           3 instructions, needs 30 streams for full utilization,
pipelined
```


Performance

- N = M = 1,000,000
- Non-zeros 0 to 1000 per row, uniform distribution
  - Nz = 499,902,410

P	T	Sp
1	7.11	1.0
2	3.59	1.98
4	1.83	3.88
8	0.94	7.56

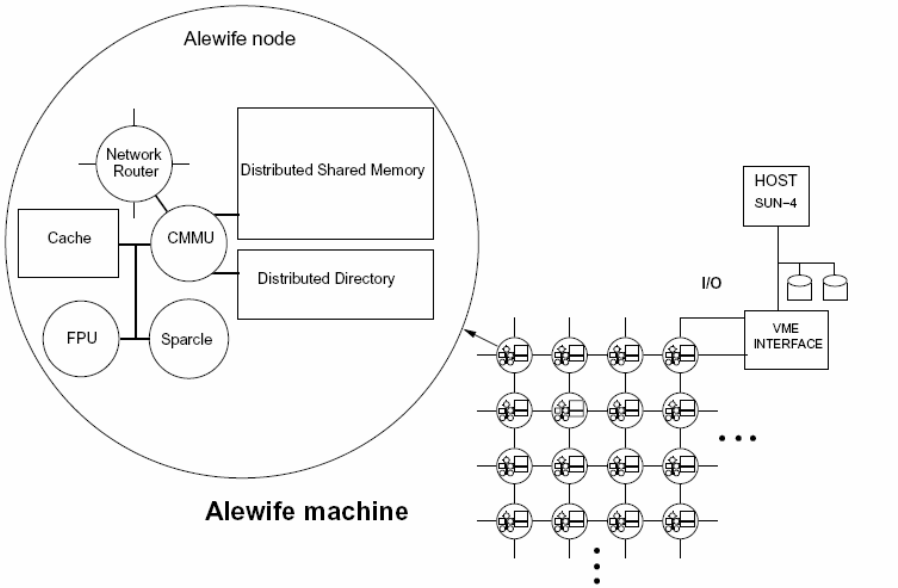
Time = (3 cycles * 499902410 iterations) / 220000000 cycles/sec = 6.82 sec

96% utilization

MTX's Sweet Spot (Cray's claim)

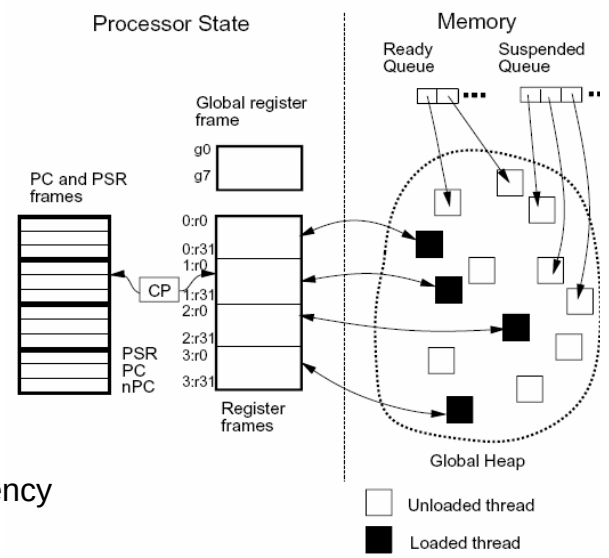
- Any cache-unfriendly parallel application
- Any application whose performance depends upon ...
  - Random access tables (GUPS, hash tables)
  - Linked data structures (binary trees, relational graphs)
  - Highly unstructured, sparse methods
  - Sorting
- Some candidate application areas:
  - Adaptive meshes
  - Graph problems (intelligence, protein folding, bioinformatics)
  - Optimization problems (branch-and-bound, linear programming)
  - Computational geometry (graphics, scene recognition and tracking)

Alewife Prototype (MIT, 1994)



Sparcle Processor (Coarse-Grained)

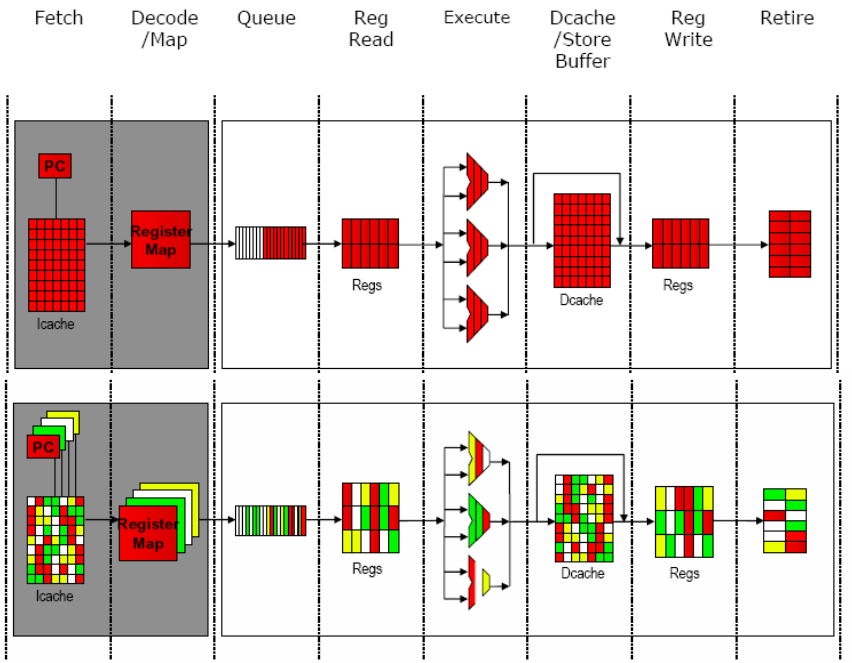
- Leverage Sparc
- Use each reg window as frames
- Loaded threads are bound to frames
- Every memory word has a full/empty bit
  - J-structure: Raise exception
  - L-structure: Block / nonblock
- Only switch on long latency
  - Coherence
  - Access empty data



Simultaneous Multithreading
(Tullsen, Eggers, Levy, 1995)

- Main idea
  - dynamic and flexible sharing of functional unites among threads
- Main observation
  - Increase utilization ⇒ increase throughput
- Change OOO pipeline
  - Multiple context and fetch engines
  - Utilize wide OOO superscalar processor issue
  - Resources can satisfy superscalar or multiple threads

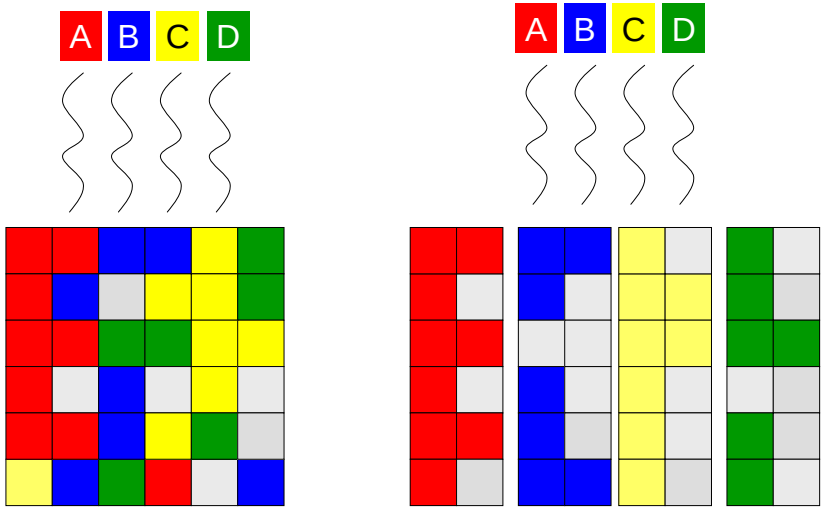
OOO Superscalar vs. SMT Pipeline



SMT Processors

- Alpha EV8 (cancelled)
  - 8-wide superscalar with 4-way SMT support
  - SMT mode is like 4-CPU with shared caches and TLBs
  - Replicated PCs and registers
  - Shared inst queue, caches, TLB, branch predictors
- Pentium4 HT (2 threads)
  - Logical CPUs share caches, FUs, predictors
  - Separate context, registers, etc.
  - No synchronization support (such as full/empty bit)
  - Accessing the same cache line will trigger an expensive event
- IBM Power5
- Sun Niagara I and Niagara II (Kunle's talk)

SMT vs. Multi-Issue CMP



Challenges to Use SMT Better

- Shared resources
  - Shared execution unit (Niagara II has two)
  - Shared cache
- Thread coordination
  - Spinning consume resources
- False sharing of cache lines
  - May trigger expensive events
  - Pentium4 HT calls it Memory Order Machine Clear or MOMC event

SMT Architectural Support

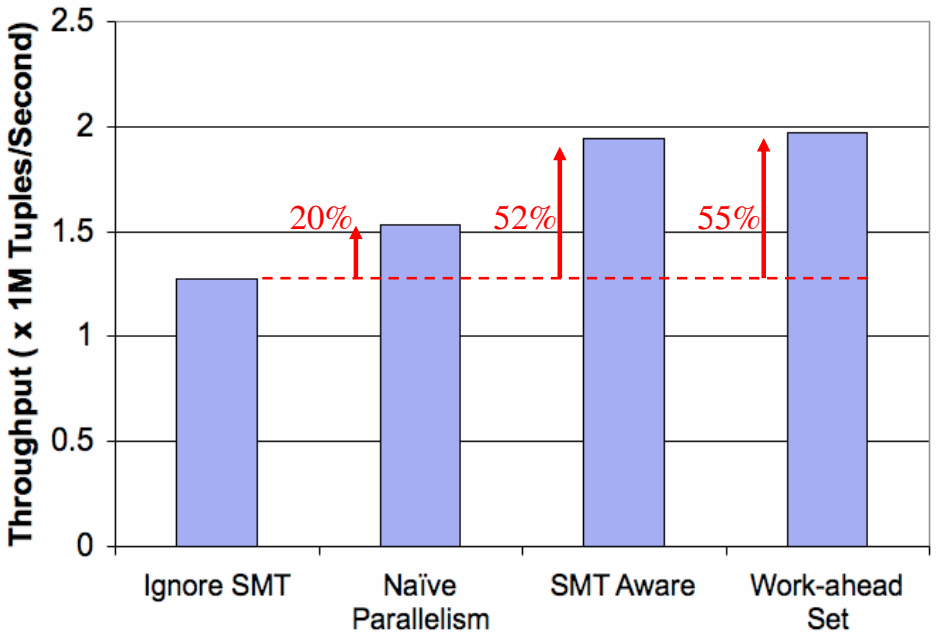
- Which thread to schedule?
  - Thread with min "ICOUNT" counting # instructions in the pipeline of a thread
- What happens if a thread is spining?
  - Use "quiescing" instruction to allow a thread to "sleep" until memory changes its state

```
Loop:
        ARM r1, 0(r2) //load and watch 0(r2)
        BEQ r1, got_it
        QUIESCE       //not schedule until 0(r2)changes
        BR loop
got_it:
```

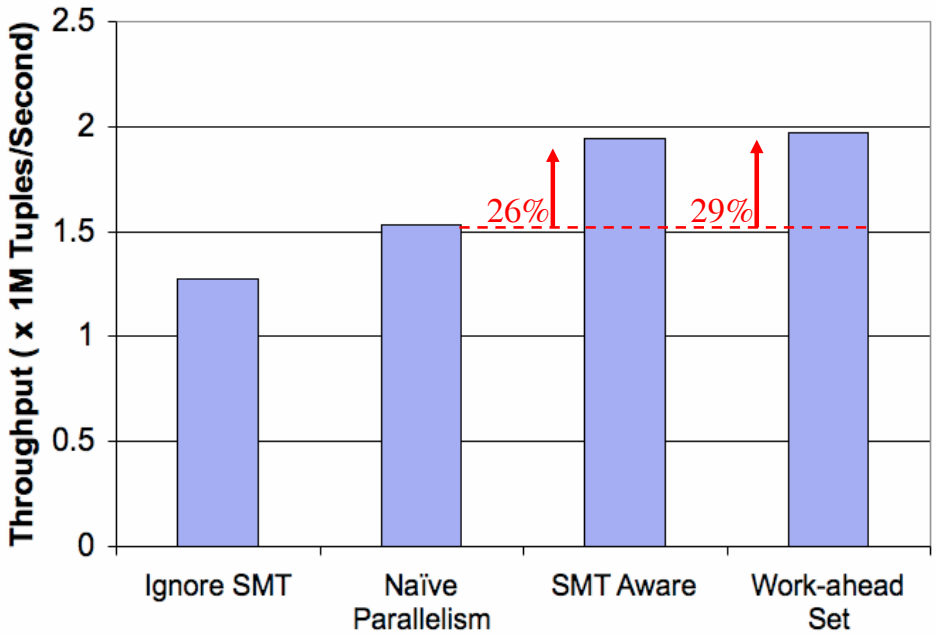
SMT-Aware Programming

- Divide input and use a separate thread to process each part
  - E.g., one thread for even tuples, one for odd tuples.
  - Explicit partitioning step not required.
- Avoid false sharing
  - Partition output and use separate places
  - Merge the final result
- Use shared cache better
  - Schedule threads for cache locality
- Use a helper thread
  - Preload data into the cache
  - Cannot be too fast or slow (especially on P4 HT)

Parallel Operator Performance
(from Zhou, Cieslewicz, Ross, Shah, 2005)



Parallel Operator Performance
(from Zhou, Cieslewicz, Ross, Shah, 2005)



Summary

- Reducing communication cost
  - Reducing overhead
  - Overlapping computation with communication
- Multithreading
  - Improve HW utilization with multiple threads
  - Key is to create many threads (e.g. MTX supports 1M threads)
- Simultaneous Multiple Threading (SMT)
  - Combine multithreads with superscalar
  - Combine with multiple cores
  - Need work to use SMT well