Programming the Cray XMT

Schedule – Day 1

Starting time	Topic
9:00	Introductions and Outline
9:15	Overview of the XMT architecture and history
10:00	XMT Applications
10:30	Programming Environment Basics
11:00	Exercise 1
12:00	Lunch
1:15	Programming for Performance 1
2:30	Exercise 2
5:00	End of Day 1

Schedule – Day 2

Starting time	Topic
9:00	Shared Memory Considerations
10:00	Debugging with MDB
10:30	Programming for Performance 2
11:30	Exercise 3
12:00	Lunch
1:00	Using Canal, Traceview and Bprof in Apprentice 2
2:00	Betweenness Centrality – Multi-level Parallelism
2:30	Graph Generation and Snapshot-Restore I/O
3:15	Break
3:30	Exercise 4
4:30	Class Feedback; Final Comments
5:00	End of Day 2

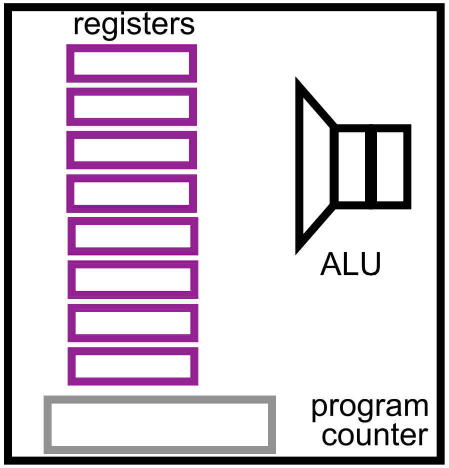
XMT Architecture Overview

Multithreading

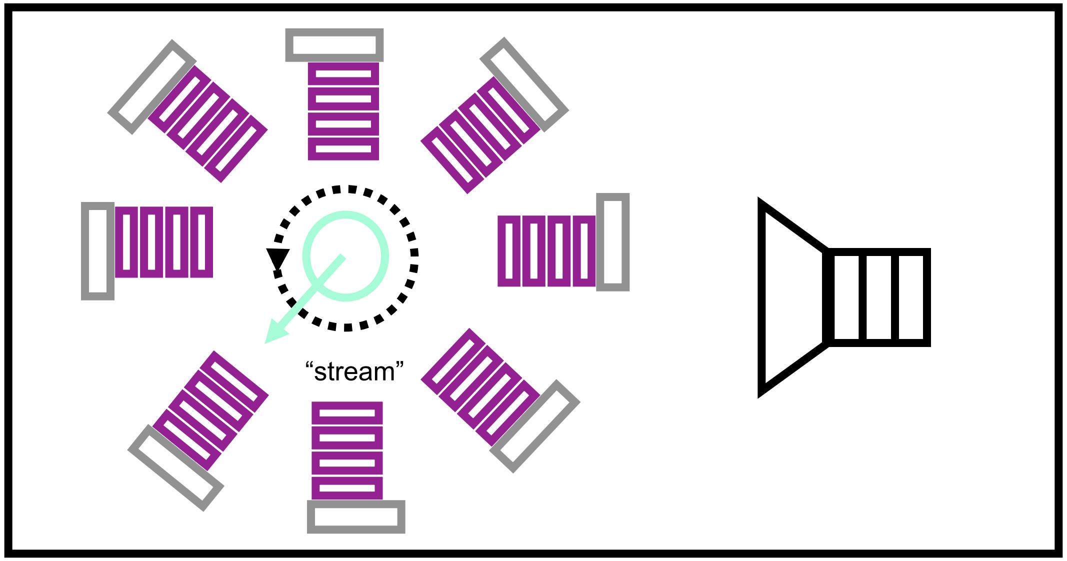
Multithreaded processors are to conventional processors as Gatling guns are to conventional machine guns.

Many threads per processor core; small thread state

Thread-level context switch at every instruction cycle



Commodity processor



Multithreaded

Why Multithreading?

Relative latency to memory continues to increase

- Vector processors *amortize* memory latency
- Cache-based microprocessors *reduce* memory latency
- Multithreaded processors *tolerate* memory latency

Multithreading is most effective when:

- Parallelism is abundant
- Data locality is scarce

Large graph problems perform well on the Cray XMT

- Semantic databases
- Big data

Hiding Memory Latencies

Caches

- Reduce latency by storing some data in fast, nearby memory

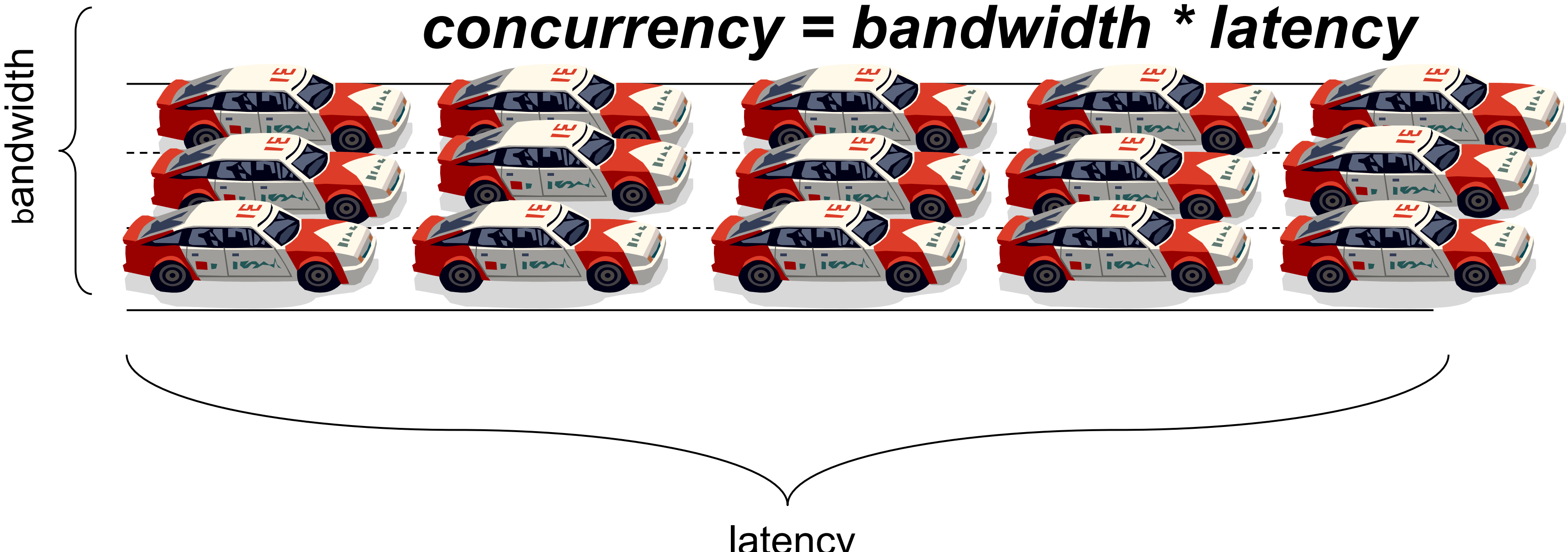
Vectors

- Amortize latency by fetching *N* words at a time

Parallelism

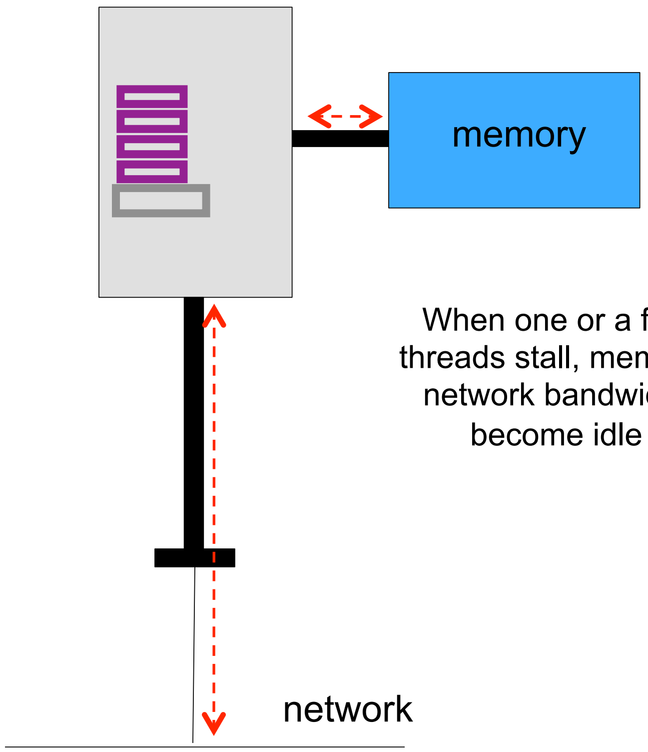
- Hide latency by switching tasks
- Multithreading tries to balance “Little’s Law:”

*concurrency = bandwidth * latency*



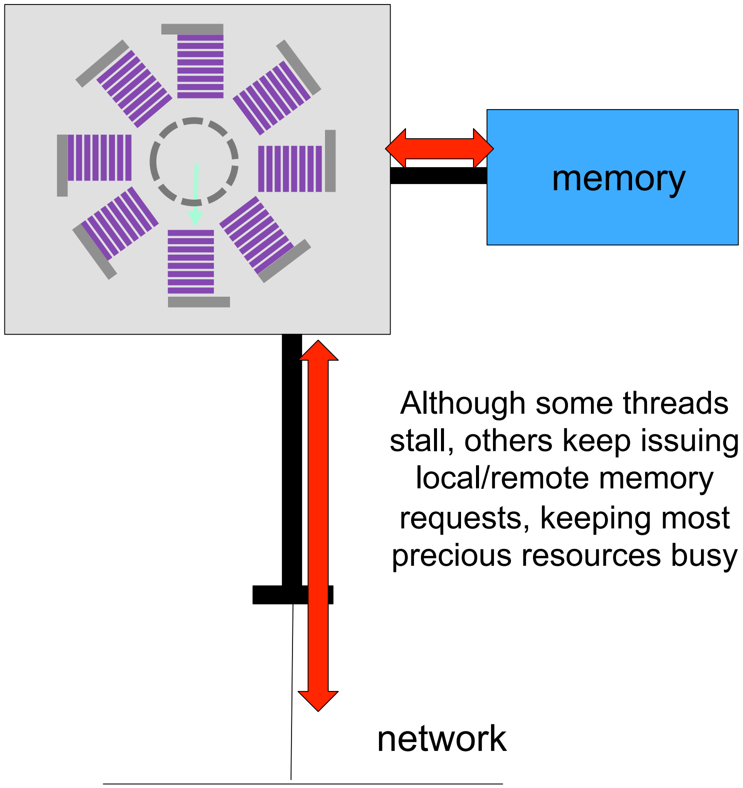
Keeping the Bottlenecks Saturated

Conventional processor



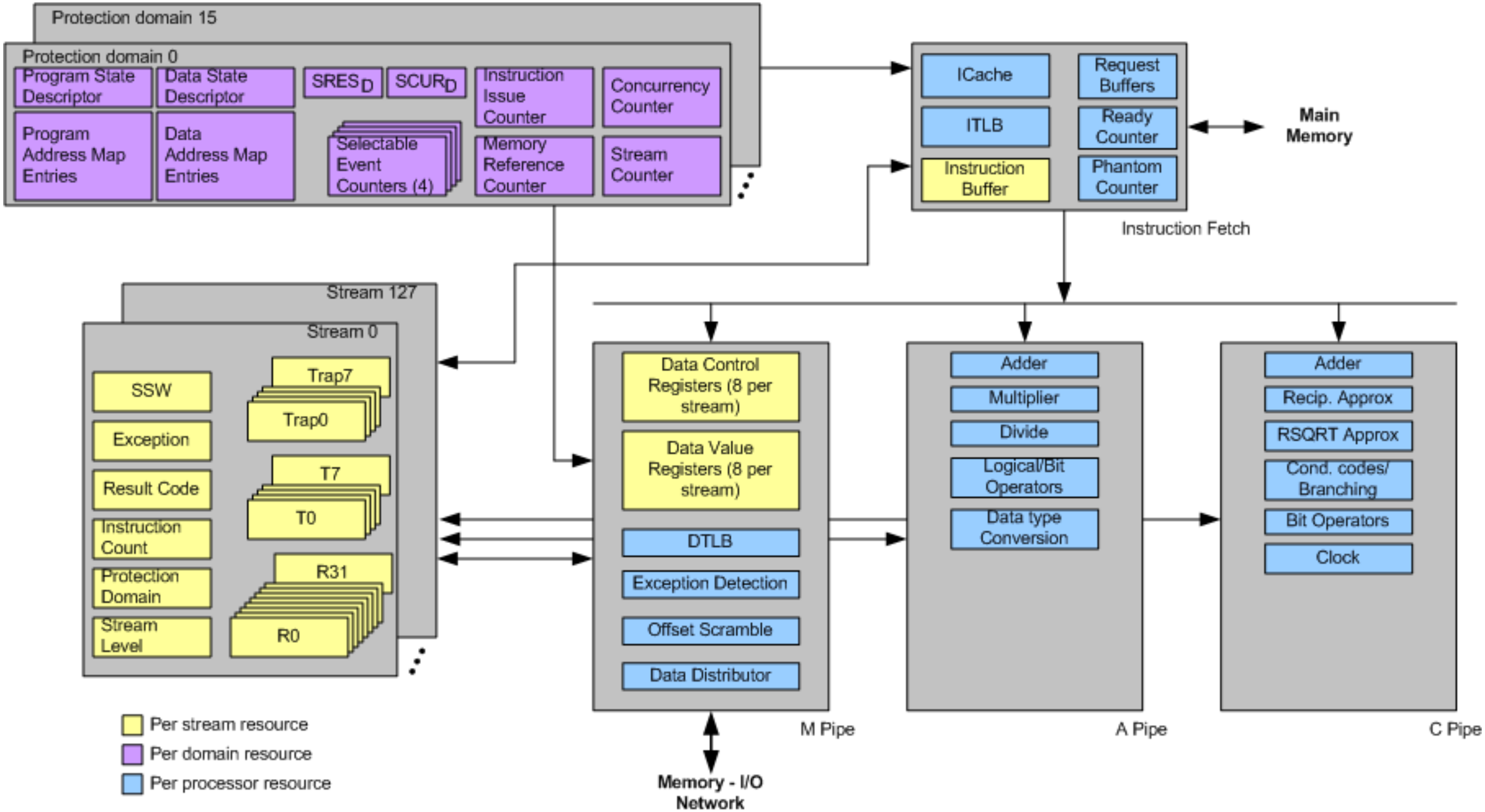
When one or a few threads stall, memory/ network bandwidth become idle

Multithreaded processor



Although some threads stall, others keep issuing local/remote memory requests, keeping most precious resources busy

XMT's "Threadstorm" CPU Architecture



Threads and Streams

A thread is a software object

- A program counter and a set of registers
- Very lightweight
  - Not pthreads
  - No OS state

A stream is a hardware object

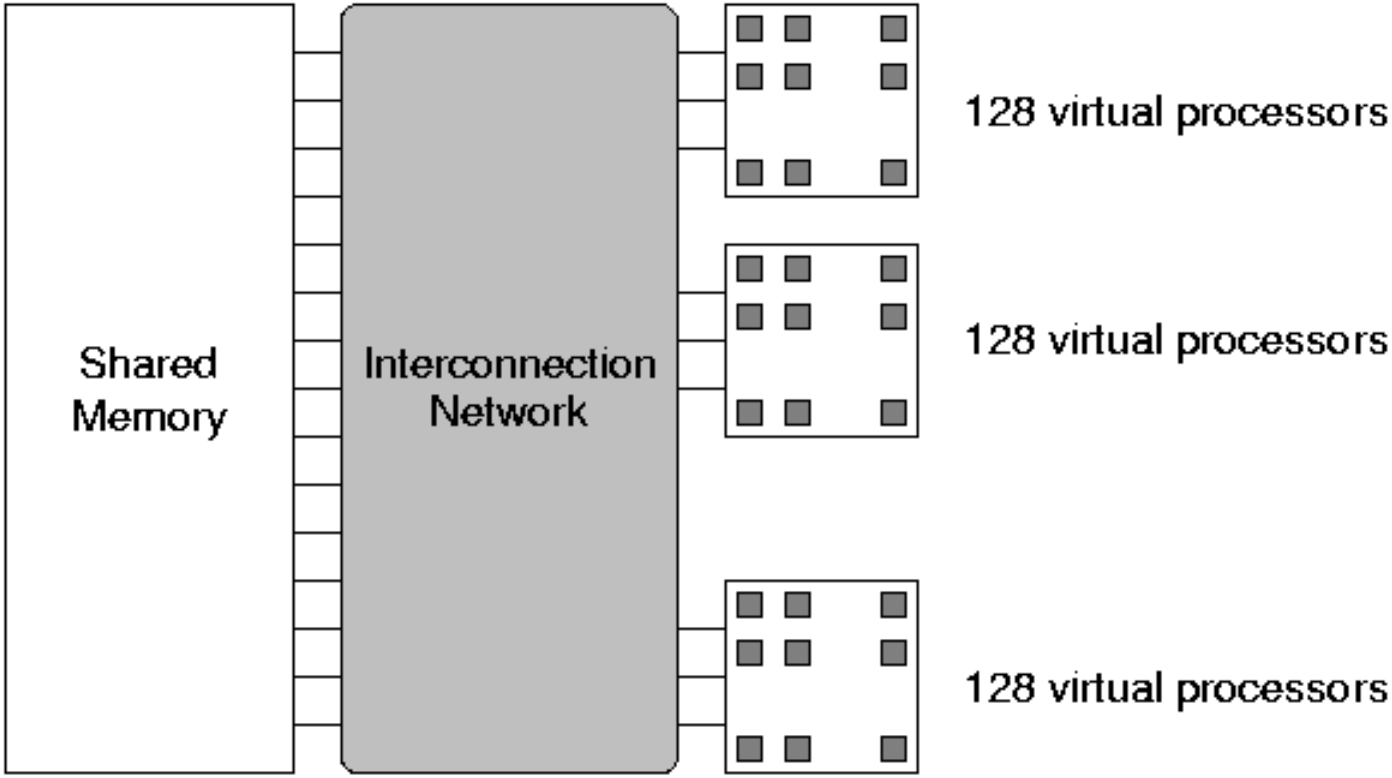
- Stores and manipulates a thread’s state
- Very lightweight stream creation
  - A single instruction executed from user space

More threads than streams

Threads multiplexed onto the processor’s streams

XMT Programming Model

To the programmer, a multiple processor XMT looks like a single processor, except that the number of threads is increased.



Memory

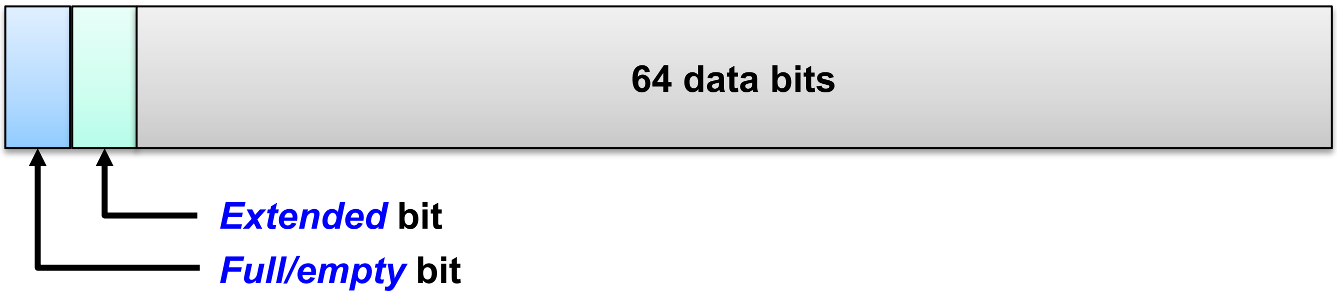
When the system is booted, the memory of each compute node is divided into:

- Local memory
  - Local or “nearby” memory for the user application
  - Used for runtime data structures and I/O
- Global memory
  - The MTK RAMFS is loaded here
  - Shared memory for the user application
  - Addresses for global memory are hashed to distribute addresses throughout the global memory
    - Addresses ranges are not blocked to a node
    - This is done to reduce memory and network contention

Memory word

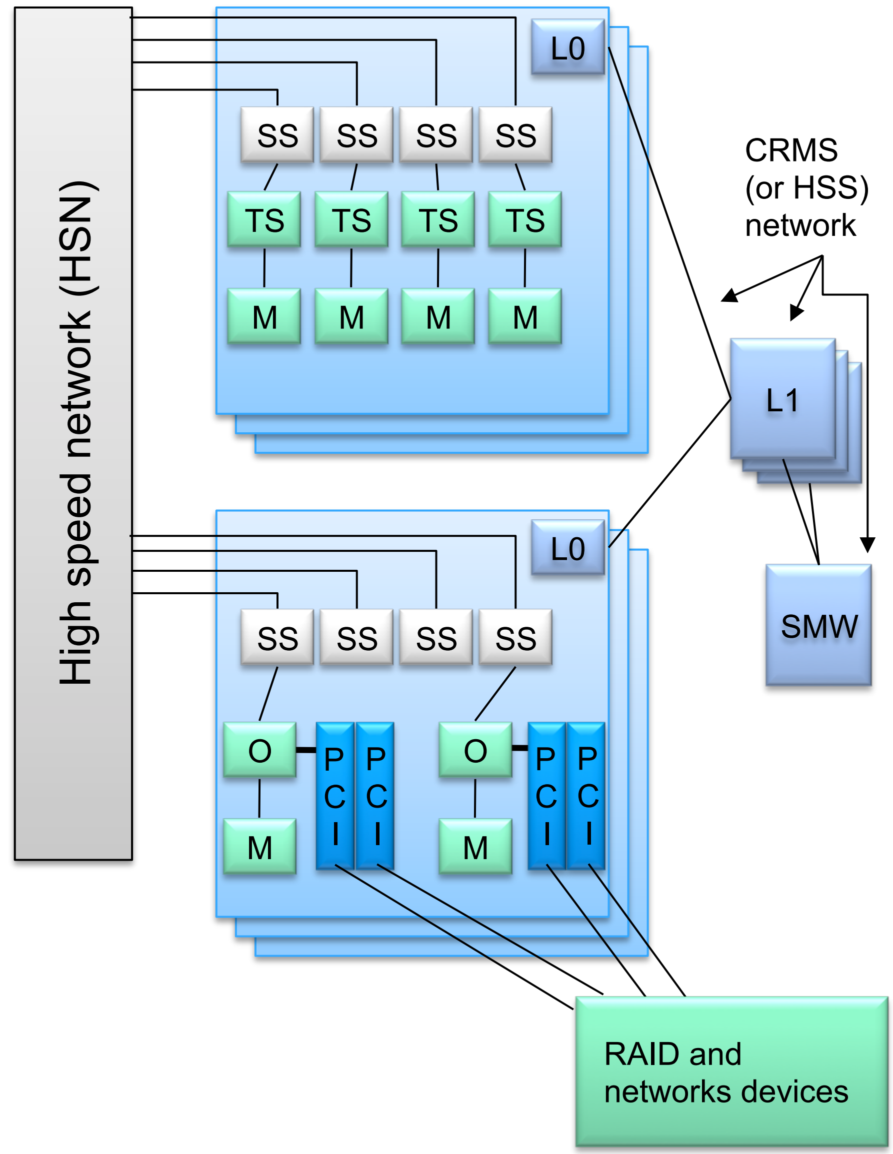
The XMT memory word has 66 bits

- **64 bits of data, byte addressable**
  - Data is stored big-endian
- **2 tag bits**
  - The *full/empty* bit
    - Used for synchronization
  - The *extended* bit
    - Set by the hardware, for example when there is a trap

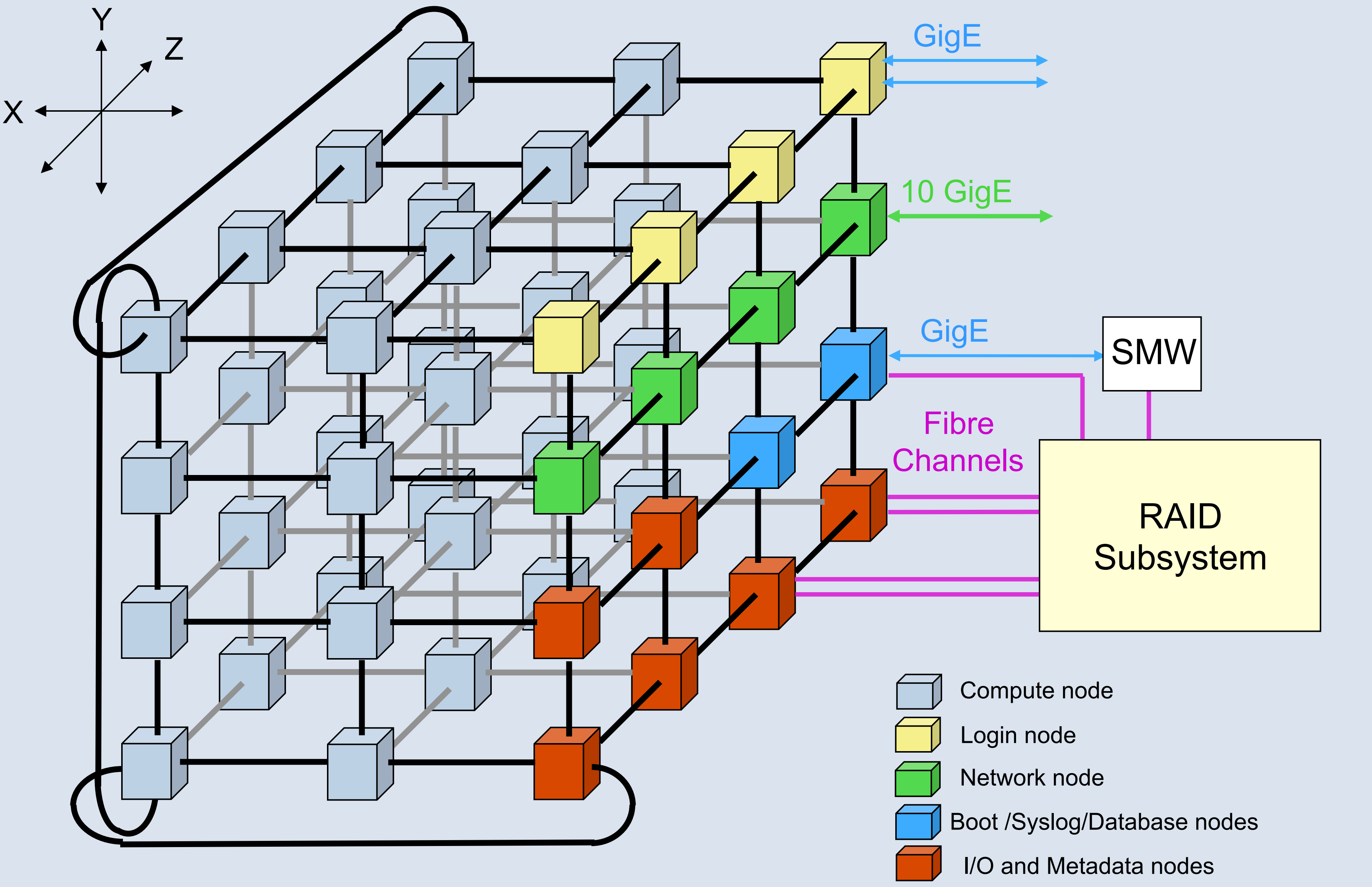


XMT System Architecture

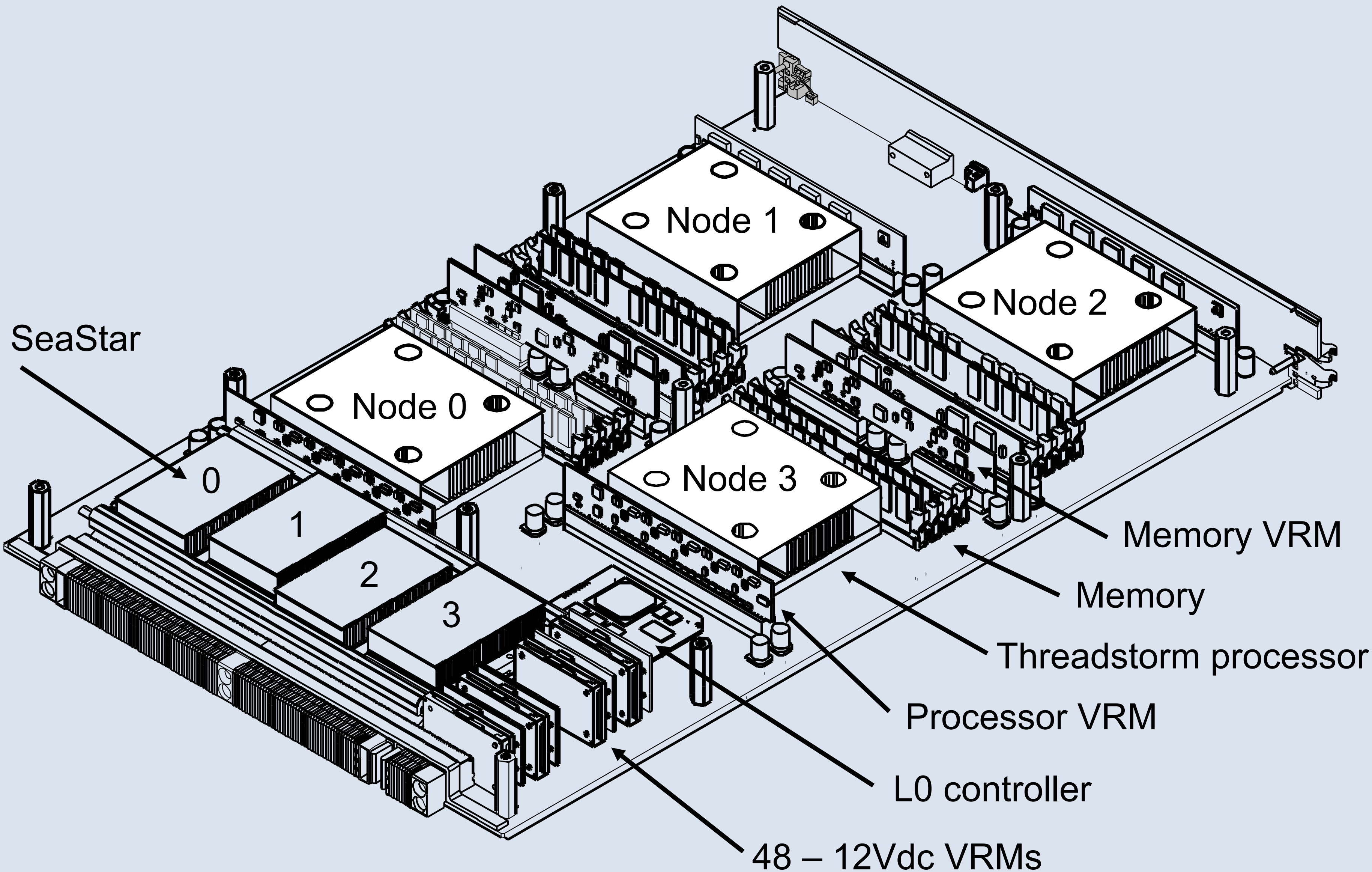
- **Compute nodes are based on Cray Threadstorm processor**
  - **Executes MTA instruction set; compatible with previous MTA systems**
- **Service nodes are based on AMD Opteron processor**
  - **Run a full version of SUSE Linux with additional Cray and third-party software**
  - **I/O uses PCI-X/PCIe interfaces associated with service nodes**
    - **Fibre Channel HBAs to RAIDs**
    - **1- and 10-Gb Ethernet cards for network connections**



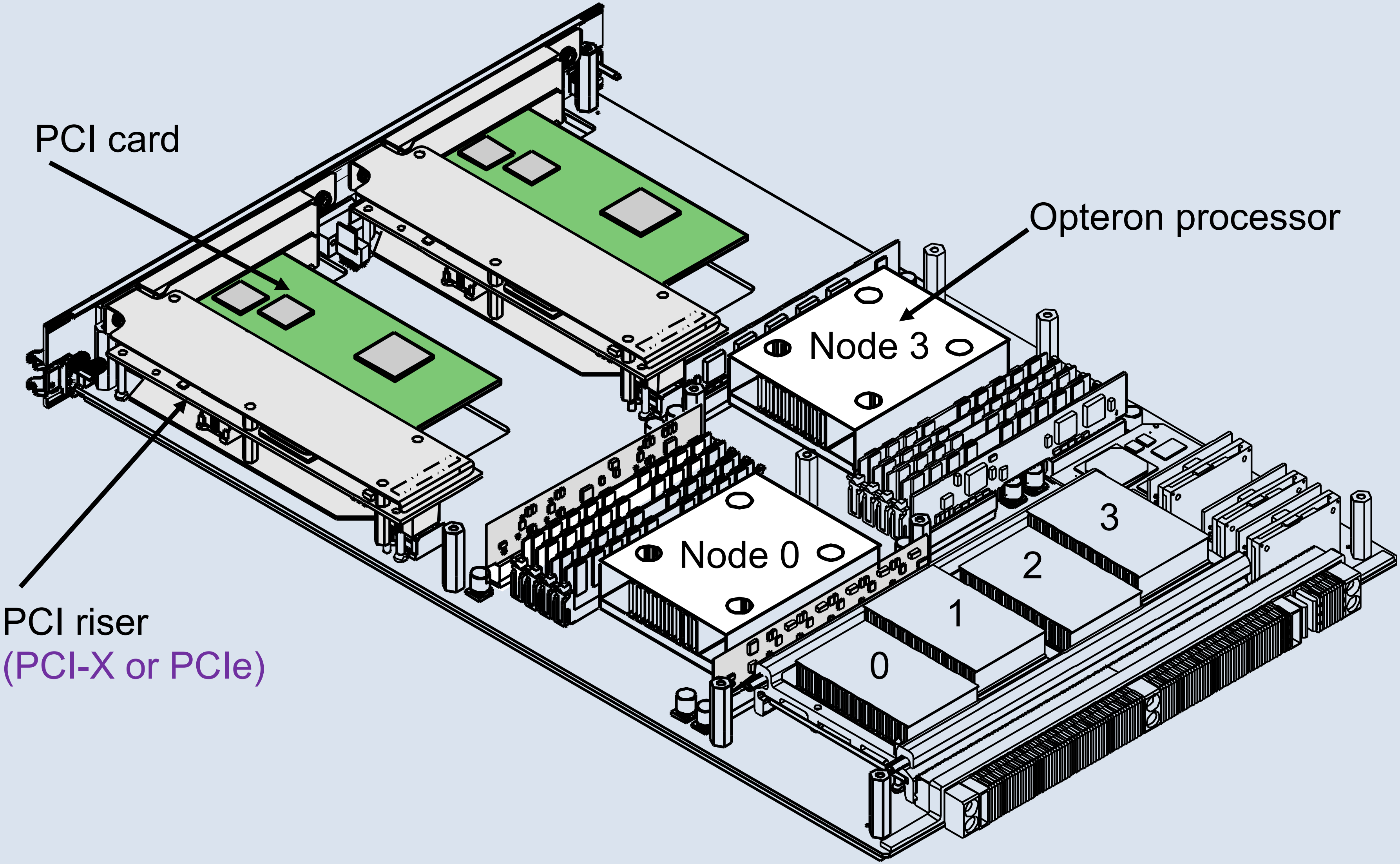
Cray XMT System Connections



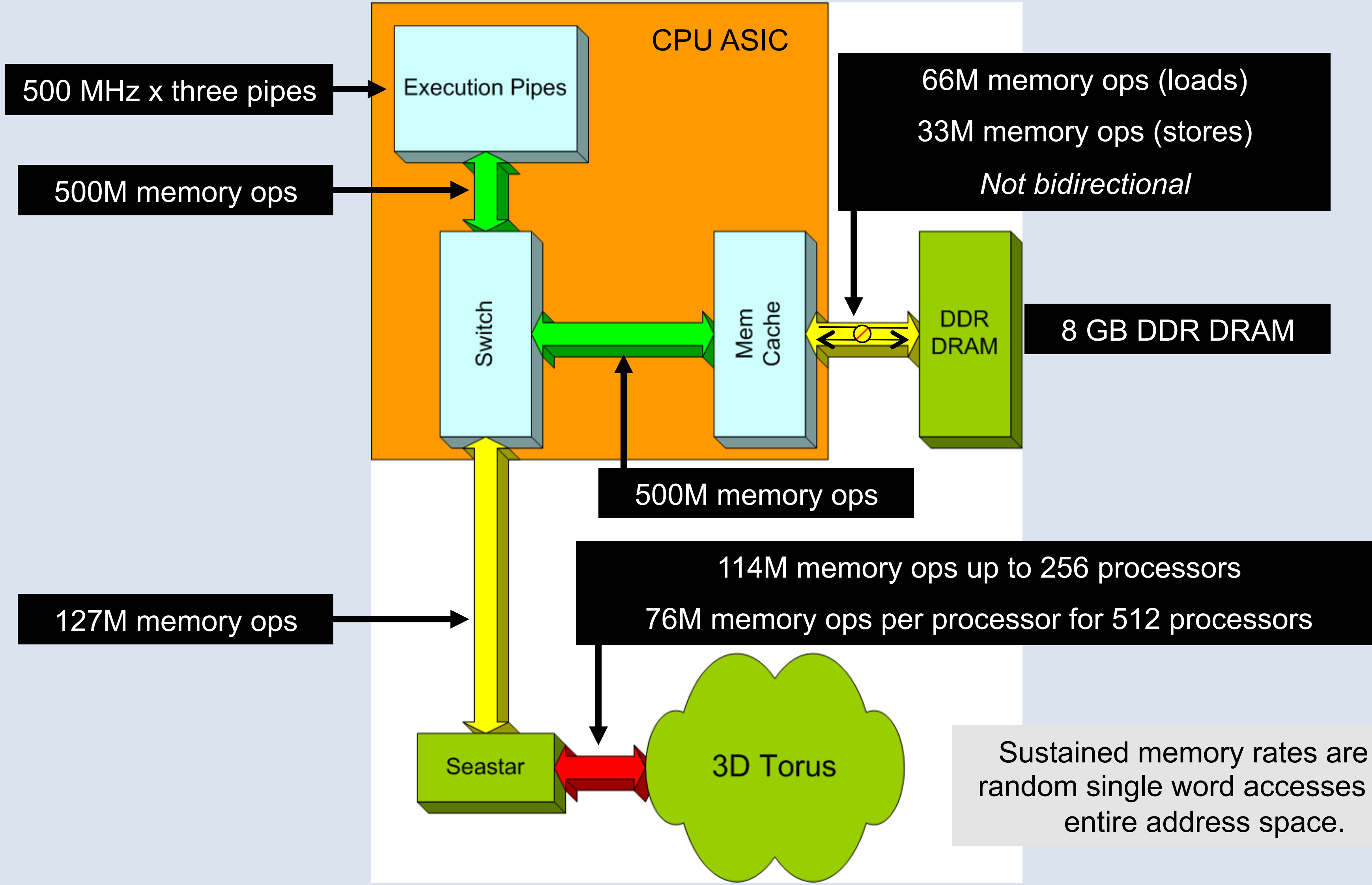
Cray XMT Compute Blade



Cray XMT (XT) Service Blade



Speeds and Feeds



Sustained memory rates are for random single word accesses over entire address space.

Potential Architectural Bottlenecks

- **Processor throughput? Never observed to be the bottleneck.**
  - **Typical processor utilization ~ 30%**

- **Network bandwidth**
  - **Especially at larger scale**
    - **Tunable HW settings in the network, based on the amount of concurrency needed to saturate the bisection BW, in place to avoid over-saturation of the network**
    - **128P sized systems and smaller limited to 180 outstanding memory operations per processor**
    - **For 512P system, this drops to 144 outstanding operations**

- **Memory bandwidth**
  - **Minimizing trips to memory is important**

Additional Performance Considerations

- **Sequential code murders performance.**
  - **21 cycles per instruction issue**
- **XMT memory references are hashed**
  - **Granularity of 8-word cache lines**
- **All jobs use all memories in the system**
  - **Example: “betweenness centrality” graph computation on 16 processors of a 128-processor system: 25% faster than on 16 processors of a 16-processor system**
- **Exclusive protection domains, but no exclusive ownership of physical hardware resources**

Cray XMT Architecture – Summary

- Heavily multi-threaded processor: 128 hardware threads multiplexed between OS and all applications
  - 16 protection domains (address maps) per processor
  - Multi-threaded architecture tolerates memory latency
  - Data locality not critical for performance
- Scrambled and distributed shared memory to avoid contention
- Lightweight synchronization using full/empty bits on all memory
- Interconnect bisection bandwidth scales with the number of processors
- No hardware interrupts
  - Hardware threads allocated by user via instruction, not OS
- Exceptions and traps do not cause a privilege change
  - They are handled at the privilege level in which they occur

History of the XMT Architecture

- MTA-1 (Multi Threaded Architecture) launched in 1998
  - 18 GaAs chips per processor blade, with custom memory
- Cray MTA-2 launched in 2002
  - 5 CMOS chips per processor on 1 large PC board with custom DIMMS
- Cray XMT launched 2008
  - Processor reduced to single CMOS chip in Opteron socket
  - 4 processors per PC board, standard DIMMS
  - Cray XT network, packaging, cooling and RAS features
- First Next Generation XMT delivered to CSCS in 2011

Next Generation Cray XMT

- Next Generation builds on successful Cray XMT

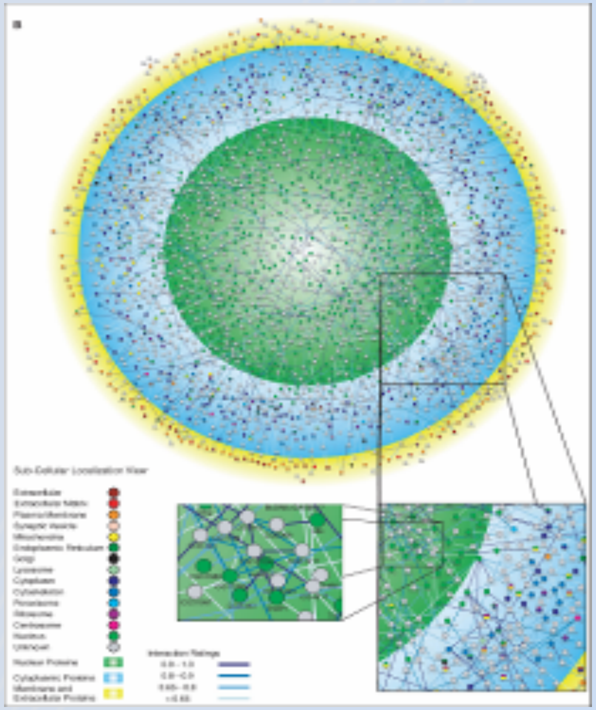
- Memory system improved significantly
  - Large improvement in bandwidth
  - Very large improvement in capacity

- Hot Spot Avoidance
  - Productivity—simple implementation performs best
  - Reliability—difficult programs cannot interrupt system services
  - Performance—use network more efficiently

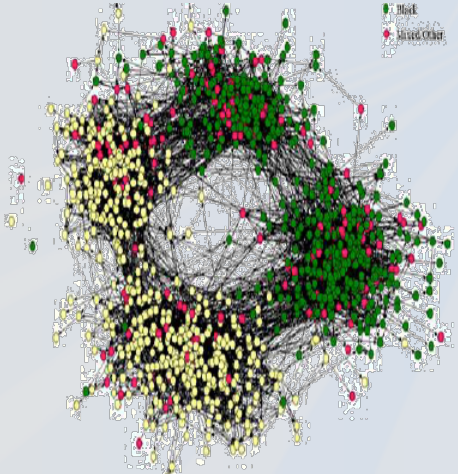
XMT Applications

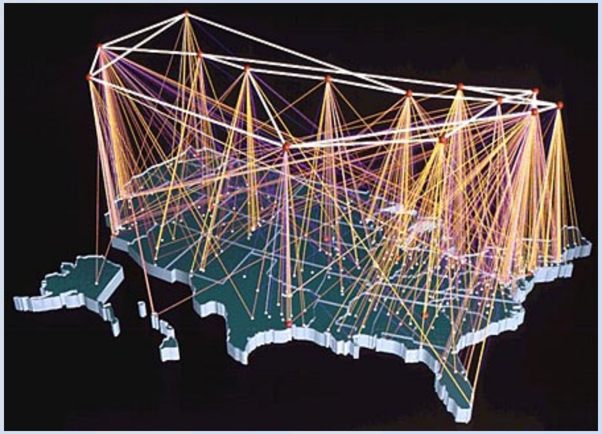
Cray XMT's Application Sweet Spot

- Any application that involves ...
  - Random or indirect memory accesses
  - Dynamic or unbalanced subcomputations
  - Unstructured, dynamic, and/or sparse data structures
  - Linked data structures (lists, graphs, trees)
  - Sorting or searching
  - … on HUGE data sets



- Applications that need to access large amounts of memory (terabytes) and in an unpredictable manner.
  - Graph Analysis (intelligence, protein folding, bioinformatics)
  - Data mining
  - "Graph mining"
  - Business intelligence
  - Pattern matching
  - Power grid analysis





Cray XMT-Based Applications & Solutions

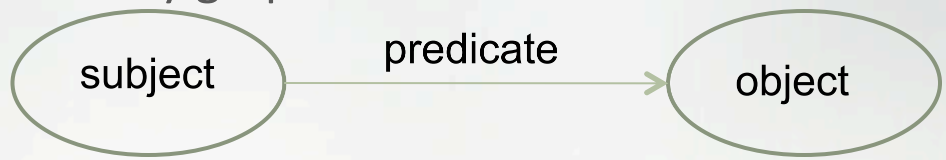
- Cyber Security
  - Dynamic Network Analysis for Network Intrusion
  - Anomaly Detection – PDTree
  - String Matching
- Informatics
  - Semantic Database
  - Interactive Analytics
  - Visualization for Large-scale Graphs
- Bioinformatics
  - Large-scale Sequence Alignment
  - Histopathological Images Analysis
  - Epidemiology: simulating individual-based models of epidemics in networks
  - Dynamic Biological Network Analysis
- Video Analytics
  - Unstructured Data Analysis using Sparse Graph Network-of-Networks Algorithms
- Agent-Based Parallel Discrete Event Simulation
  - Organizational Business Process Simulation
- Electric Grid
  - Contingency Analysis
  - Smart Grid

Active (funded) Research Areas

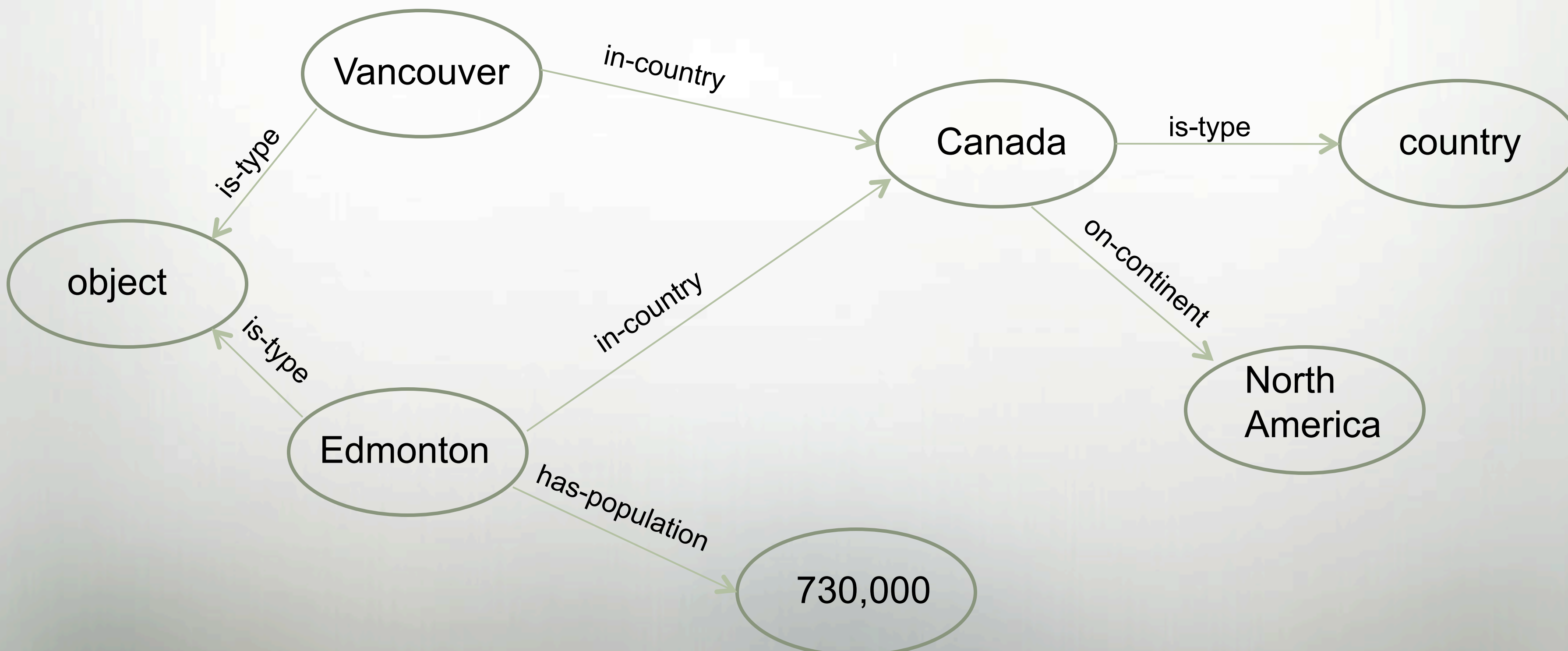
Semantic Database of RDF Triples

- RDF triples databases are inherently graphical

(subject predicate object) =



- Some researchers call semantic databases “semantic graph databases”



Comparing Relational Databases to Semantic Network Databases

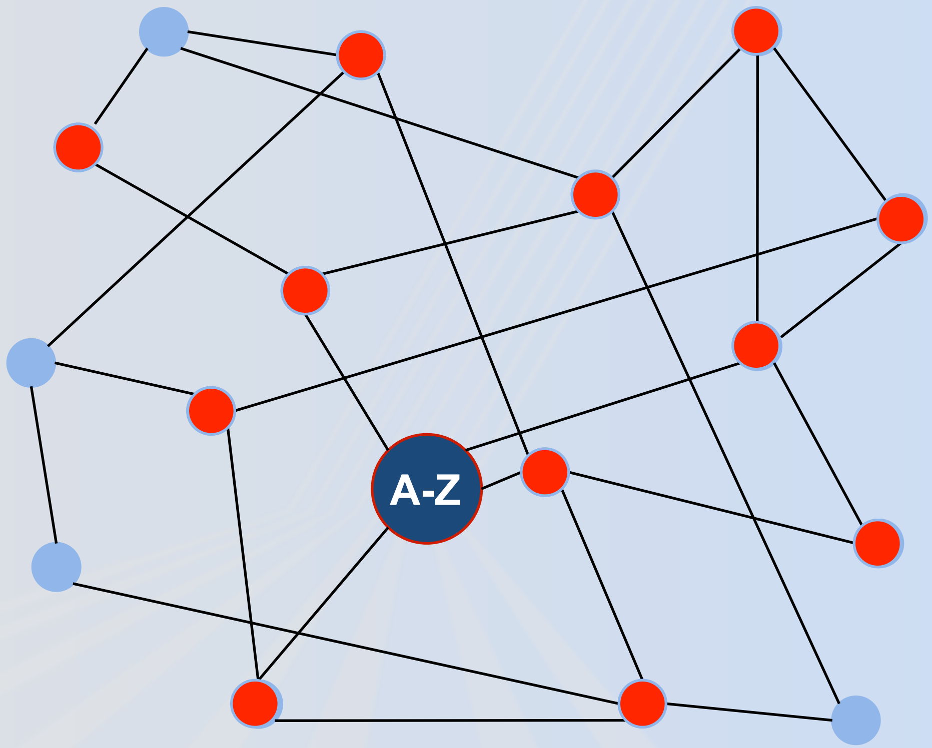
This type of query is easy for either:

"Show all company employees who are age 45 or older"

Smith	27
Jones	36
Johnson	29
Wilson	51
Peterson	48
Ordonez	34
Quigley	61
Roberts	53

This type of query is very hard and slow for relational, fast for semantic network database:

"Show all people who have met with Al-Zawahiri or have met with someone who met with Al-Zawahiri"

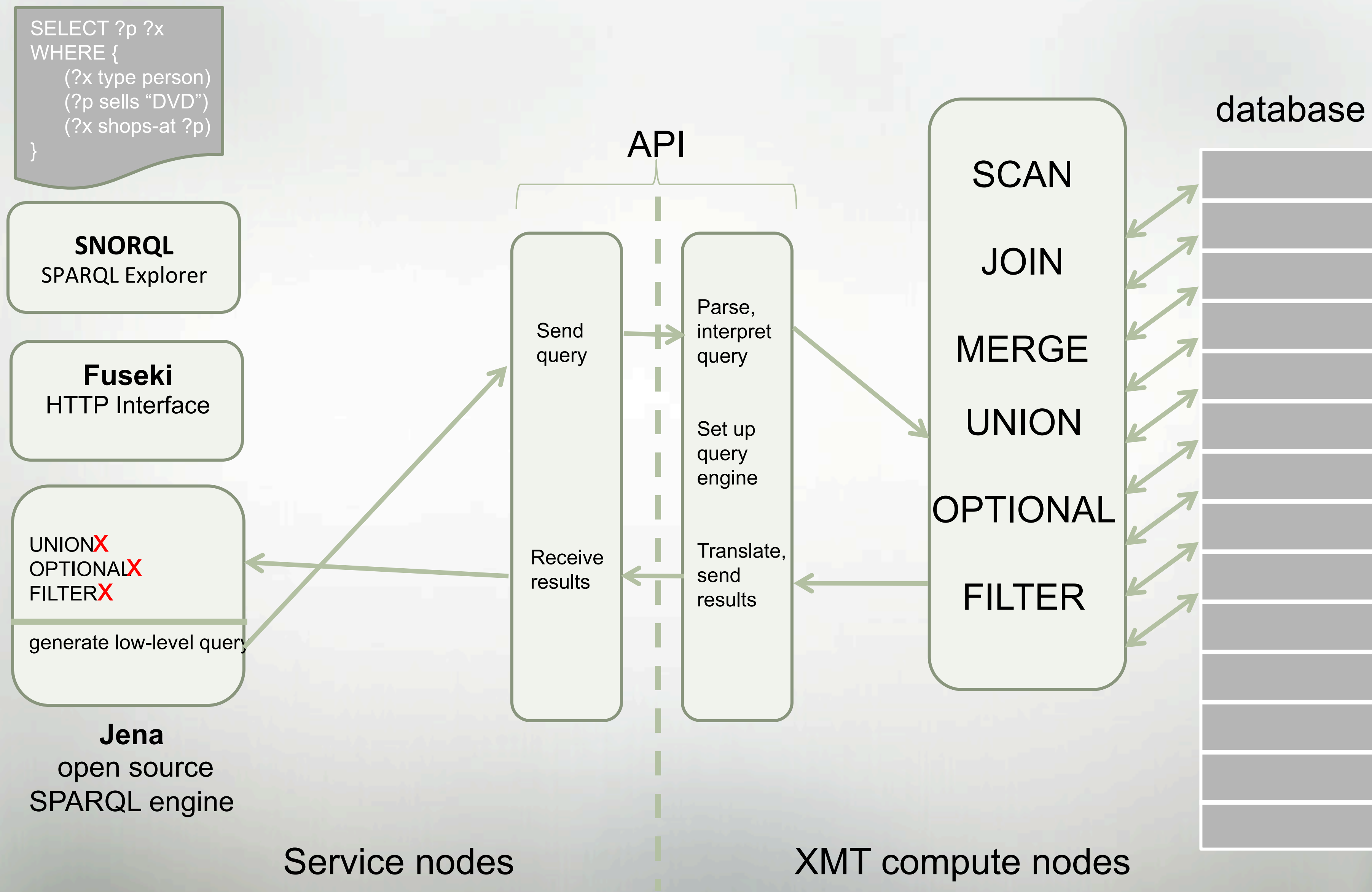


Semantic networks also support *reasoning*:

X attended meeting M &

Y attended meeting M ⟶ X met with Y

Semantic Database Prototype



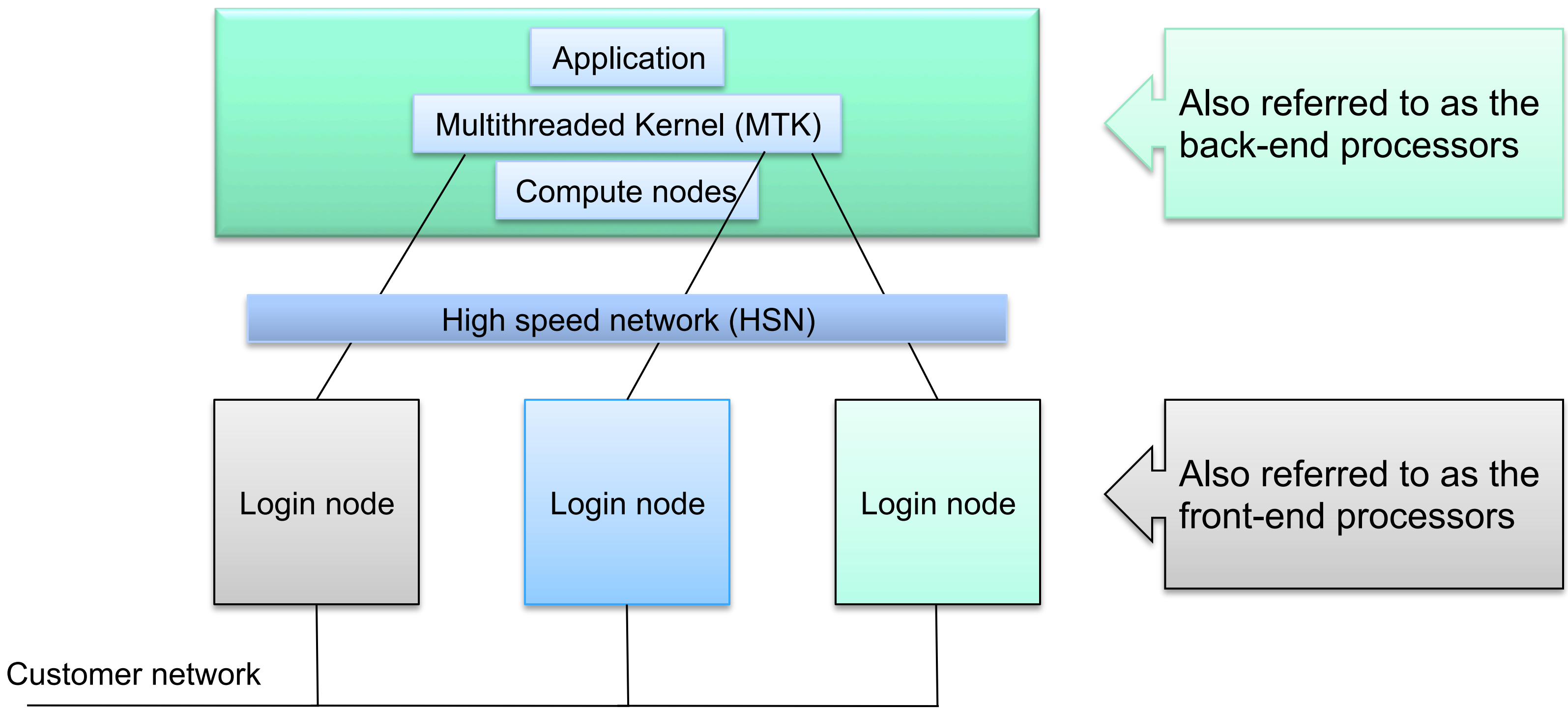
Programming Environment Basics

Topics

- Accessing a Cray XMT System
- Cray XMT Programming Environment
- Interactive Program Launch
- Monitoring
- Batch Program Launch

Accessing a Cray XT System

- `ssh` is normally used to connect to the system
  - User account information is maintained through an LDAP or Kerberos server
  - Passwordless `ssh` can be set up to access a system
    - Set up a pass phrase for a more secure session



Cray XT Programming Environment

- A cross-compiler environment
  - Compiler runs on Linux login node
  - The executable runs on the compute nodes
  - Provides automatic parallelism if proper options are included
  - Recognizes C and C++ directives and language constructs
- Modules utility
  - Consists of the module command and module files
  - Initializes the environment for a specific compiler
  - Allows easy swapping of compilers and compiler versions
- Cray written compiler driver scripts for C (cc) and C++ (CC)
  - MTA compiler options, system libraries, and header files
  - Compiler specific programming environment libraries
  - Compilers support shared libraries (non-static linking)
    - No dynamic libraries

PE Summary

Compilers	C and C++
Compiler tools	Canal
Libraries	libc++.a, libm.a, libprand.a, libluc.a, libsnapshot.a
Runtime libraries	librt.a, libc.a, libs.a
Debuggers	mdb
Performance tools	Tview, Bprof, and Apprentice2

Module Commands

Cray uses modules to control the user environment, use the commands:

module	
list	to list the modules in your environment
avail	to list available modules
load	to load a module
show	to see what a module loads
swap	to swap two modules For example: to swap `mta-pe` version 6.0.0 with 6.2.1 `% module swap mta-pe/6.0.0 mta-pe/6.2.0`

Standard Modules

```
%> module list
Currently Loaded Modulefiles:
  1) modules/3.1.6      3) mta-man/6.5.0
  2) xmt-tools/3.7.2    4) mta-pe/6.5.0
%>
```

- `mta-pe` module includes
  - cc, c++, mdb, nm, dis, etc..
- `xmt-tools` module includes
  - `mtarun` and `mtatop`
- `mta-man` module includes the man pages

Compiling an Application

Two modes for compiling: *skinny* and *fat*

- *skinny* or whole-program mode
  - The preferred mode on the XMT
  - Information about the entire program (every file) is stored in a single program library file (`.pl` suffix)
  - A knowledge of the entire program enables the compiler to perform optimizations
  - `.o` files are still produced, but they are merely timestamps
- *fat* or separate module mode
  - Like traditional gcc
  - Information for each module is stored in a separate `.o` file
  - Useful for porting code and reusing *Makefile*s

Compiling Skinny

To compile *skinny*, either compile and link everything as one event or specify a *program library* file using `-pl`.

- **Skinny compilation as one event:**

```
cc -o myapp foo.c bar.c
```

- **Skinny compilation with `-pl`:**

```
cc -pl myapp.pl -c foo.c
cc -pl myapp.pl -c bar.c
cc -pl myapp.pl -o myapp foo.o bar.o
```

  – **Each successive compilation inserts more information into the original program library**

Compiling Fat

If you compile and link separately and do not specify a `.pl` file (the `-pl` option), the compiler defaults to *fat* mode

```
cc -c foo.c
cc -c bar.c
cc -o myapp foo.o bar.o
```

- **Use this mode only if you must; for example, porting code with a complicated build system.**

Compiler Flags

- **Without any options, the compiler disables implicit parallelism and loop restructuring and observes (enables) parallelization directives**
  - **Available flags are listed below; if multiple flags are provided, the order of precedence is highest to lowest**

`-nopar`	Do not parallelize; ignore everything
`-serial`	Enable automatic loop restructuring and obey parallelization directives
`-par1`	Enable automatic parallelization, but limit execution to a single processor
`-parfuture`	Future-based parallelization (very dynamic scheduling)
`-par`	Normal parallelization (default)

Compilation Failures

- Error in user code
  - May also appear as “warning” or “remark”
  - Does not halt compilation; is only a notification
- Syntax or other error in user’s code:

```
"test.cc", line 4: error: expected a ")"
    int x = strtol("6", 0, 0;
                            ^
```

- Link error – user’s reference is undefined; may need to link a library `-w` or `-l`:

```
resolve: undefined symbol foo(int).data
from a.out.pl(test.cc)
resolve: undefined function foo(int)
from a.out.pl(test.cc)
```

Compiler version mismatch

An attempt to mix compiler versions will cause a mismatch error

- For example, compile `foo.c` and `bar.c` with 6.0.2 and link with 6.0.3

```
Error: Compiler version mismatch on
file: a.out.pl
  Expected: 6.0.3, Actual: 6.0.2
```

Libraries

- The runtime library (`librt`) supports:
  - Future variables
  - Synchronization
  - Scheduling
  - Event logging
  - Compiler generated parallelism
  - Debugging
- Lightweight user communication (LUC) interface
  - Use LUC to build a client/server interface between the front-end and back-end
    - Back-end processors do not have direct access to the Lustre file system; service processors do not have direct access to compute processor memory
    - LUC is a C++ interface
    - Symmetric; RPC-style interface in either direction

Vocabulary Review

- **Task:** The complete program
- **Team:** Resources and data structures that are associated with a single processor
- **Stream:** A set of hardware registers that are used for instruction issue
- **Thread:** A register state; threads (software) run on streams

Launching a Compute Node Program

Use the `mtarun` command

- During execution, the `mtarun` command connects to the `mtarund` daemon on the back-end
  - `mtarun` sends the path to the binary, the users environment, and the command line arguments to the back-end
  - Performs simple permissions checks and setup, then `forks` and `execs` the application
  - Path to user binary must exist on the back-end
    - User home directories are typically available over NFS
  - The PID on the front-end should be same as on the back-end
    - Signals to front-end `mtarun` process are propagated to the back-end process (through `mtarund`)
      - Signal names and numbers are slightly different for MTK

`mtarun` Control

- Two `mtarun` options control job execution:
  - The `-t` *`num_procs`* option specifies the number of teams that are initially assigned to the application
    - A team is a *protection domain*. Normally, an application is allowed only one protection domain for each Threadstorm processor .
  - The `-m` *`max_procs`* option limits the number of processors (teams) that the application is permitted to use
- The user runtime program reads the `MTA_PARAMS` environment variable. Some useful options are:
  - `echo`: prints the parameters (toggled on/off)
  - `stream_limit` *`n`*: specifies the max number of streams per processor
  - `num_procs n`: specifies the maximum number of processors that the application can use
  - `no_prereserve`: prevents the reservation of 3 streams for the debugger (for “benchmarking” runs)

Monitoring and Control

`mtatop`	Similar to the `top` command, but connects to a daemon (`dashd`) that runs on the back-end. Provides additional information such as the number of processors and streams.
`dash`	The Cray XMT performance monitoring GUI (The window displays the GUI name as Dashboard2.)

mtatop

Threadstorm processors

Total memory for Threadstorm processors

```
XMT: nid00033 UP: 4d+21:44:59
[09:50:54]
Total: 128 cpus  @500.00Mhz  Mem: 1024.0G  128*128 max streams
AVG Util:    1.5% Traps:      0.1 Strms:     17.3
Free Mem: 832.3G MemR/s: 474.7M Flop/s:    1.8M Strms [res: 2,061, act: 523]

   PID USER       S   PRI NICE   P     SIZE      TIME   UTIL SysUTL TASK
     0 root       Ru    0    0 128    16.9G 5d+17:49   0.9%   0.9% mtk
 32798 root       Ru  100    0 128     6.0G 2d+17:32   0.5%   0.5% clockd.v1
 32855 root       Ru  100    0   1   146.8M 01:37:57   1.6%   0.0% dashd
 32839 root       Ru  100    0   1   148.6M    31:25   0.5%   0.0% mtarund
 32824 root       Ru  100    0   1   146.7M    27:38   0.5%   0.0% syslogd
     1 root       Sl  100    0   1   272.1M    00:00   0.0%   0.0% init
 32789 root       Sl  100    0   1   147.6M    00:00   0.0%   0.0% bash
 32790 root       Sl  100    0   1   147.6M    00:00   0.0%   0.0% bash
 32795 root       Sl  100    0   1   146.2M    00:05   0.0%   0.0% rememd
 32827 root       Sl  100    0   1   146.7M    04:03   0.0%   0.0% prngd
 32828 root       Sl  100    0   1   148.1M    00:00   0.0%   0.0% bash
 32829 root       Sl  100    0   1   148.1M    00:00   0.0%   0.0% bash
 32842 root       Sl  100    0   1   146.6M    00:00   0.0%   0.0% portmap
```

Cursor is here, see next slide for options

`128 * 128 max streams`
is 128 streams on each of 128 processors

`mtatop` Command Options

- **While `mtatop` is running "interactively," some useful commands are:**
  - **`c:` displays CPU usage**
  - **`p:` enables you to view process specific info (see the next slide)**
  - **`u:` enables you to filter by user name**
  - **`t:` returns you to the default display**
- **Batch (`mtatop` is not interactive) options that you can add to the `mtatop` command:**
  - **`-b:` a *batch mode* snapshot of the system; provides the typical `mtatop` output and CPU usage**
    - **Appending `-pid` *`process_ID`* to the *`-b`* option provides additional information about the process**

mtatop - Process Information

```
XMT: nid00033 UP: 4d+22:37:47                                       [10:43:41]
Total: 128 cpus  @500.00Mhz  Mem: 1024.0G  128*128 max streams
AVG Util:   19.3% Traps:     0.5 Strms:    57.8
Free Mem: 583.8G MemR/s:   8.1G Flop/s:    1.4M Strms [res: 7,570, act: 6,029]

-----------------------------------------------------
Util:     21.1% Traps:   195.6 Strms:   60.5
MemR/s:   67.6M Flop/s:   1.0K Strms [res: 12997 act 12994]
-----------------------------------------------------
Process Name:           futurestress
Process ID:             25544
User Name:              someuser
Parent Process ID:      32839
Process Group ID:       25544
UID:                    95762
Cpu:                    11.3%
Processors:             128
User Time:              4min 22sec
State:                  Running
Nice value:             0
Priority:               100
Resident Size:          266,708,443,136
Program Text Size:      2,588,672
Program Data Size:      266,707,910,656
Shared Size:            532,480
System Calls:           1
Blocked System Calls:   0
FS Bytes:               14,272
Networking Bytes:       3,748
```

At the cursor, type ***p***
You are then prompted for the process ID

Exercise 1

Assignment:

Write a loop that initializes each element of an integer array to its index value squared, followed by a loop that sums the elements of the array.

```
#define M 1000000
int array[M];
int i;
for( i = 0; i<M; i++)
//???
int sum=0;
for(i=0; i<M; i++)
//???
printf("%d\n", sum);
```

Lunch

Programming for Performance – Part 1

Loops

The compiler can automatically parallelize 3 kinds of loops:

- **Loops without loop-carried dependences,**
- **First-order linear recurrences, and**
- **Reductions.**

This is our basic palette and we strive to express all our programs in these forms.

Inductive Loops

Before the compiler will consider parallelizing a loop, the loop must be *inductive*.

- **Single entrance and single exit,**
- **Controlled by a linear induction variable (incremented by an invariant amount each iteration), and**
- **Exit is controlled by comparing the induction variable against an invariant.**

The key here is that the compiled code must be able to determine, a priori, how many iterations will be executed.

Example of Parallelizing Your Code

This loop parallelizes:

```
void foo() {
      int i;
      int my_array[10000];
      for (i = 0; i < 10000; i++) {
            my_array[i] = i;
      }
      return;
}
```

Example 2 of Parallelizing Your Code

This loop does not parallelize:

```
void foo(int *a, int *b) {
      int i;
      for (i = 0; i < 10000; i++) {
            a[i] = b[i];
      }
}
```

- a and b may point to overlapping memory

```
foo(x+5000, x);
```

Example 3 of Parallelizing Your Code

If you know that `a` and `b` will not point to the same memory, you can use a `pragma` to instruct the compiler it is safe

- This code will now parallelize

```
void foo(int *a, int *b) {
      int i;
  #pragma mta noalias *a
      for (i = 0; i < 10000; i++) {
            a[i] = b[i];
      }
}
```

Compiler Transformations for Parallelism

The compiler attempts to restructure code to find or enhance parallelism:

- Scalar expansion
- Making associative operations atomic
- Reductions
- Recurrences

You can view the ways the compiler restructured your code in Canal (text-based) or in the canal tab of Apprentice2 (GUI-based).

Scalar Expansion

This loop cannot be parallelized because of the interaction of the reads from `t` and the writes to `t` in subsequent iterations (an anti-dependence):

```
for (i = 0; i < n; ++i) {
  t = sqrt(a[i + 1]);
  a[i] = t + 5;
}
```

Scalar Expansion

- **The compiler resolves this conflict by converting the scalar integer `t` into an array of integers**
- **The compiler then splits the original single loop into two loops to preclude the conflict between reading `a[i+1]` and writing `a[i]` on the next iteration.**

```
for (i = 0; i < n; ++i) {
   t[i] = sqrt(a[i + 1]);
   a[i] = t[i] + 5;
}
```

Scalar Expansion

- Viewing this loop in the canal tab of Apprentice2, we see:

```
       | for (i = 0; i < n; ++i) {
5 P:e  |   t = sqrt(a[i + 1]);
++ function sqrt inlined
5 P    |   a[i] = t + 5;
       | }
```

Performed scalar expansion

Reductions

- The compiler attempts to recognize loops that calculate sums, products, minimums, and maximums over an array. For example:

```
int min = MAX_VAL;
for (i = 0; i < n; i++) {
  if (x[i] < min)
    min = x[i];
}
```

- The compiler converts these to reductions
  - Each thread computes the min/max/sum/product over a subsection of the array.
  - Threads then combine results to determine the final value.

Reductions

- **Viewing this in the canal tab of Apprentice2, we see:**

```
      | for (i = 0; i < n; i++) {
3 P:$ |   if (x[i] < min)
** reduction moved out of 1 loop
      |     min = x[i];
      | }
```

Converted to reduction

Recurrences

Some loops use values computed by early iterations in later iterations. These recurrences usually prevent parallelization.

The compiler recognizes first-order linear recurrences and rewrites them so they can be solved in parallel. For example:

```
for( i = 2; i < n; i++ )
     X(i) = X(i - 1) + Y(i)
```

Generally, the compiler can handle recurrences of the form

```
X(i) = X(i - k)*F(i) + G(i)
```

Where `k` is a small constant.

Recurrences

- Viewing a parallelized recurrence in the canal tab of Apprentice2, we see:

Computed as a linear recurrence

```
2    | for (i = 5; i < n; i++) {
3 L  |    x[i] = x[i - 3]*f[i] + g[i];
     | }
```

- In the loop annotations below, we also see:

```
Loop    2 in main at line 6 in region 1
        Stage 1 of recurrence
        ...
Loop    3 in main at line 7 in region 1
        Stage 2 of recurrence
        ...
```

Canal

Canal generates analysis of the parallelization and optimizations made to the application by the compiler

- Can use a fat object (`.o`) file
- Can use a program library (`.pl`) file (compiled with a skinny `.o`)

`-pl` *`pl_file`*	Specify the program library file (program compilation in *skinny* mode)
`-o` *`o_file`*	Specify the object file (program compiled in *fat* mode)
`-P profile`	If the application was compiled with the profile option and a profile file was created, include it in the Canal report
`-e`	Executable, required if the `-P` option is included
`source_file`	The source file

Additional Canal Options

`-a`	A s part of the optimization process, annotate generated loops. (Normally, Canal annotates only your source code.)
`-all_inlined`	Annotate inlined routines from the standard libraries.
`-f`	Ignore source file time-stamp.
`-nl`	Print line numbers with the source code.
`-pb`	Insert page breaks between source files.
`-s`	Print a description of Canal annotations.
`-help`	Print a usage message.
`-v`	Print the version number and exit.

Compiler Annotations

- **Optimizations performed by the compiler appear on the left side of the Canal output**

P	The compiler parallelized the loop automatically
p	An assertion in the source code caused the loop to parallelize
D	An assertion caused the code to parallelize, but would not have without the assertion
L	A parallelized linear recurrence or reduction
-	The compile could not parallelize the loop
S	The marked statement prevented the loop from parallelizing
s	Did not parallelize because the loop was too small
X	Did not parallelized because the loop was not inductive
U	The compiler completely unrolled the loop
e	The compile expanded the scalar variable to a vector and distributed it to all the loops
$	The statement is atomic (uses the full/empty bit)

Canal

Annotated source code listing

Program library file

Source file

```
%> canal -pl matreduce2.pl mat_reduce2.c

…
          |     for (i=0; i<MAXARRAY; i++) {
          |        for (j=0; j<MAXARRAY; j++) {
          |
     4 PP |           x[i][j] = i+j+2;
     4 PP |           y[i][j] = i+j+2;
     4 PP |           dist[i][j] = 0;
          |
          |        }
          |     }
```

Two nested loops, both parallelized

Loop number

Canal

Annotated source code listing

Loop number

```
Loop    4 in report_example at line 4 in loop 3

        Parallel section of loop from level 2

        Loop summary: 1 memory operations, 0 floating point
           operations

         1 instructions, needs 50 streams for full utilization
           pipelined
```

User Runtime

User runtime can be used to control how an application runs

- **The runtime controls:**
  - Trap handling
  - Asynchronous operating system calls
  - Debugging support
  - Event logging
  - Resource acquisition and management
  - Work scheduling and management
- **The MTA_PARAMS environment variable can be used to control the user runtime**
  - Many options are also controlled with the `mtarun` commend

```
$ export  MTA_PARAMS="param1 n param2 n"
```

MTA_PARAMS

`num_procs n`	Sets the maximum number of processors to use
`stream_linit n`	Sets the maximum number of streams to use per processor
`num_readypools n`	Sets the maximum number of ready pools for the entire task
`max_readypool_retries n`	Sets the maximum number of retries that an idle thread can make to try to find new work
`ft_trapsoptions`	Enables various floating-point trap options
`no_preserve`	Prevents the runtime from reserving 3 streams for attaching the debugger
`pc_hash n, m, l`	Sets hash size, age threshold, or dump threshold for an event
`echo`	Prints a list of parameters to the screen
`debug_data_prot`	Waits for the debugger to attach instead of exiting when a data protection or poison error occurs
`do_backtrace`	Dumps the registers for all streams

Compiler Directives and Assertions

When the compiler cannot determine safe automatic parallelization options, you, as the human behind the code, can help

- **By adding directives or assertions to the source code, you can help the compile parallelize your code**
  - **For a complete list of compile directives and assertions see the "*Compile Directives and Assertions*" chapter in the *Cray XMT Programming Environment Users Guide,* S-2479**

Common Directives and Assertions

A short list of common directives and assertions follows:

- These are typically written:

```
#pragma mta assert parallel
```

- But can also be written:

```
_Pragma ("mta assert parallel")
```

Parallelization Pragmas

Help the compiler find parallelism

- `mta assert noalias *var1, *var2`
- `mta assert no dependence *var1, *var2` (or `nodep`)
- `mta assert parallel` (use as last resort)
- `mta assert par_newdelete`

Control parallelism and scheduling

- `mta parallel [on|off|default|singleprocessor |multiprocessor|future]`
- `mta recurrence [on|off|default]`
- `mta restructure [on|off|default]`
- `mta [block|dynamic|interleave] schedule`
- `mta use` *n* `streams`

The `noalias` Pragma

```
void foo(int *x, int*y, int*z) {
  #pragma mta noalias *x, *y
  for (int i = 0; i < N; i++) {
    z[i] = x[i] + y[i];
  }
}
```

- Must appear within the scope and after the declarations of the listed variables
- Asserts that the listed variables are not used as aliases for any other variables
- Can also use `restrict` pointers to get the same affect
  - `void foo(int * restrict x, int *y, int *z) {`

The `no dependence` Pragma

```
#pragma mta assert noalias *IA
#pragma mta assert no dependence *IA
for (int i = 0; i < N; i++) {
  IA[i][1] = IA[i][INDEX[i]];
}
```

- Appears immediately before a loop
- Asserts that any memory location accessed in the loop through any variable on the no dependence list is accessed by *exactly* one iteration of the loop
- Variables on the list must be `noalias` or *restrict* pointers

The `assert parallel` Pragma

```
#pragma mta assert parallel
for (int i = 0; i < N; i++) {
  printf("May appear out of order %d\n", i);
}
```

- Asserts that the iterations of the loop can be executed concurrently without any synchronization
- Does not force the compiler to parallelize the loop, but it is a *very* strong suggestion
- Should be used only when other techniques to parallelize your loop fail
  - It limits the types of optimizations and transformations the compiler can perform on the loop
  - You are asserting that the loop is parallel *as written*
    - In the end, compiler semantics inhibit loop transformations that could produce invalid results

The `par_newdelete` pragma

```
#pragma mta par_newdelete
aclass an_array[1000];
#pragma mta par_newdelete
aclass *another_array = new double[2000];
#pragma mta par_newdelete
delete [] another_array;
```

- **May appear before an array declaration, `new`, or `delete`**
- **Before an array declaration or `new`, indicates that the constructors for the array's elements should be fired in parallel**
- **Before an array delete, indicates that the destructors for the array's elements should be fired in parallel**

Parallelization and Restructuring pragmas

`#pragma mta parallel` *`mode`*

- Allows you to change the parallelization mode within a source module
- Affects all routines that appear after the pragma
- *mode* can be on, off, default, single processor, multiprocessor, or future
  - Ignored with `-nopar`

`#pragma mta recurrence [on|off|default]`

- Enables or disables parallelization of reductions and recurrences
  - Ignored with `-nopar`

`#pragma mta restructure [on|off|default]`

- Enables or disables loop restructuring
  - Ignored with `-nopar`

Scheduling pragmas

`#pragma mta block schedule`

- Iterations are broken into <number of threads> equal size chunks and each thread is assigned a chunk
  - Can allow reuse of register data because adjacent iterations are executed by the same stream
  - Best when each iteration does about the same amount of work; otherwise, may lead to load imbalances.

`#pragma mta dynamic schedule`

- Threads are assigned one iteration at a time and receive a new iteration when they complete their current one
  - Avoids load balancing problems (threads that receive cheap iterations will do more iterations).
  - Adds overhead due to keeping track of which iterations have been assigned (a global, atomic counter).

Scheduling pragmas

`#pragma mta block dynamic schedule`

- A mixture of block and dynamic
- Each thread receives a block of iterations, then receives a new block after it completes the previous one
  - Better load balancing than block, but worse than dynamic
  - Fewer updates of the global atomic counter, thus lower overhead than dynamic

`#pragma mta interleave schedule`

- Each thread receives evenly spaced iterations
  - For example, thread A receives 1,21,41,… while thread B receives 2,22,42,…
  - Works well for triangular loop nests

```
for(i = 0; i < n; i++)
  for(j = 0; j < i; j++)
    a[i][j] = …;
```

The `use` *n* `streams` pragma

- For each parallel region, the compiler attempts to determine the number of streams per processor, which are necessary to saturate the processors
- Occasionally, you may find that raising this number improves performance

```
#pragma mta use n streams
```

  - Request at least *n* streams per processor
  - Request is passed to the runtime, which may or may not grant the requested number depending on available resources
  - If multiple loops with "use n streams" pragmas are combined into a single parallel region, the compiler uses the largest requested number

The `for all streams` Pragma

```
#pragma mta for all streams
 {
     …/* execute this set of statements
once for every stream allocated to this
parallel region. */
     …
 }
```

- **can be used in conjunction with other pragmas:**

```
#pragma mta use 100 streams //NOTE: no guarantee
#pragma mta for all streams
{ … }
```

The `for all streams i of n` Pragma

```
int thrID, numThreads;
…
#pragma mta for all streams thrID of numThreads
{
      …
      int myQuota = arraySize / numThreads;
      int myStart = myQuota*thrID;
      for(int j = myStart; j < myStart + myQuota; j++)
      {
            array[j] = …
      }
```

Other pragmas

- `mta inline`
- `mta no inline`
- `mta single round required`
- `mta loop future` (Covered later)

The `future` construct

On the MTA, `future` has two meanings:

- as a type qualifier (*a la* `sync`)

```
future int x$;
```

  loads and stores wait for and leave `x$` full

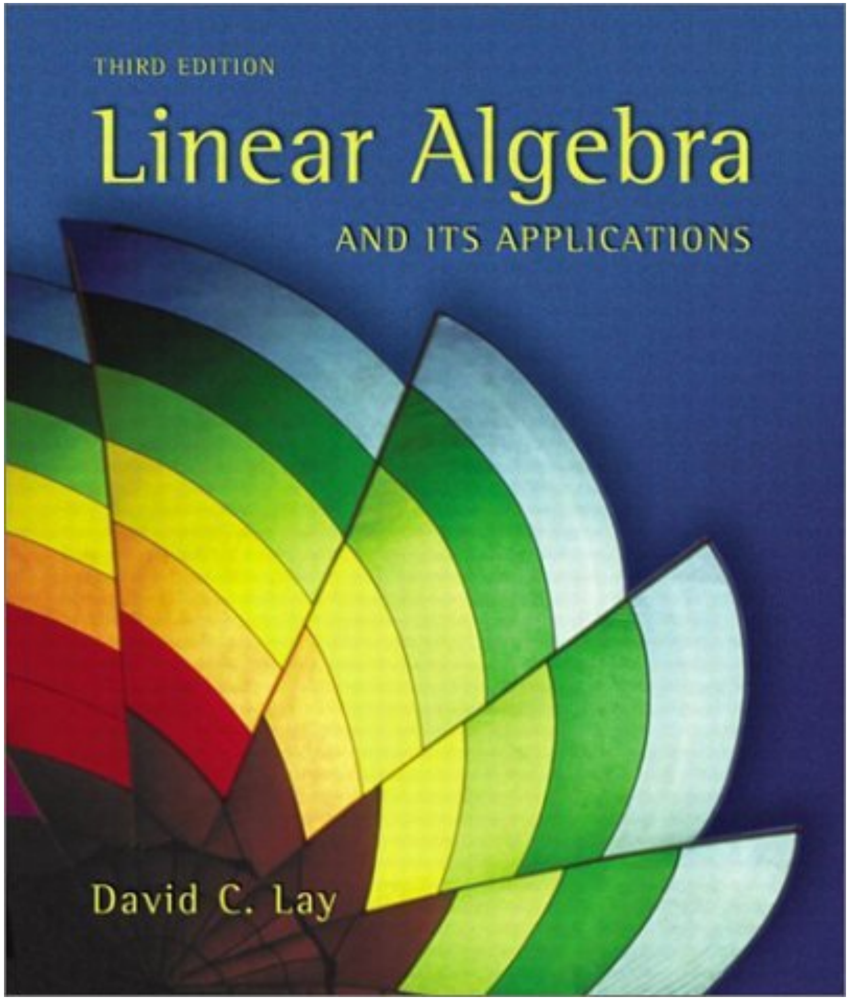
- as a statement

```
future x$(i) {
  return printf("i is %d\n", i);
}
```

  – the statement purges `x$` (sets empty)
  – arguments are passed by value
  – enqueued and executed asynchronously in ~FIFO order
  – the return value is stored to `x$`

Exercise 2

`/mnt/lustre/Workshop/Exercise2`



End of Day 1

Shared Memory Programming Considerations

Writing Programs Correctly

- Simple: Shared variables should be declared sync or future (or should be accessed only via the generics) or be protected by a lock
  - Data race-free
- Caveats
  - Deadlock/Livelock
  - Performance bottlenecks
    - Large critical sections
    - Hot spots

Intentionally *Race-y* Code

- **One way to improve efficiency of shared memory programs is to avoid synchronization if possible – or make sure it's as fine-grain as possible**

Example: Initialize Graph Search

Thread 1

```
…
if (!n.initialized) {
  n.initialized = true
  n.id = _id
  n.visited = false
  …
}
…
```

Thread 2

```
…
if (!n.initialized) {
  n.initialized = true
  n.id = _id
  n.visited = false
  …
}
…
```

This race is *benign*: the order of operations does not affect the final result.

Example: Graph Search

Thread 1

```
…
if (!n.visited) {
  n.visited = true
  if (n.id==target) {
    total++
  }
}
…
```

Thread 2

```
…
if (!n.visited) {
  n.visited = true
  if (n.id==target) {
    total++
  }
}
…
```

This race is *real*: the order of operations affects the final result.

Example: Search with Critical Section

```
readfe(&global_lock)
  if (!n.visited) {


    n.visited = true
  if (n.id == target) {
          total++
        }

      }
writeef(&global_lock)
      ...
```

Creating a critical section around node access eliminates the race, but creates a potential performance bottleneck by serializing access to the node.

Hotspot potential

Example: Search with a Shorter Critical Section

```
…
if (!n.visited) {
  readfe(&n.lock)
  if (!n.visited) {
    n.visited = true
    if (n.id == target) {
      total++
    }
  }
  writeef(&n.lock)
}
…
```

The critical section is "less critical" because only threads that arrive at approximately the same time will need to execute the critical section.

Synchronized operations ensure that the compiler will not remove the second conditional.

Example: Search Without Locks

```
                    …
if (int_fetch_add(&n.visited,1)
==0) {

   // only first one gets here
   if (n.id == target) {
      total++
   }

}
…
```

Debugging with MDB

Debugging an Application: `mdb`

- `mdb` is the XMT text-based debugger
  - Derived from gdb version 3.3.5 (very old)
    - No C++ conveniences
    - No thread support
  - On the Cray XMT support was added for:
    - MTA extensions to C/C++
    - Threads (`info threads`, `thread`, `bt` options, etc.)
    - Watchpoints and breakpoints in a multithreaded environment
  - "Remote" debugging model
    - `mdb` runs on the service node and communicates with the application via IPC (sockets)

Debugging: `mdb`

Start programs with mdb

- `% mdb a.out`

Attach to already running programs

- `% mdb a.out 11111`
- OR
- `% mdb a.out`

```
…
(mdb) attach 11111
```

PID

Debugging: `mdb`

- `info threads`
  - Print information about a running program's threads
  - Qualifiers can specify other information
    - `/a`: include runtime threads (negative ids)
    - `/n`: total number only
    - `/v`: verbose mode
    - `/b`: at breakpoint
- `info pc`
  - Print possible source lines from which the PC is derived

Debugging: mdb

- `info registers`
  - Print out integer registers and their contents
- `info opa`
  - Print out instructions that may have been responsible for data traps

Debugging Without Core Files

`MTA_PARAMS: do_backtrace`

- Dump registers of the stream that hit the fatal error
- If the runtime is corrupted, sometimes the register dump fails

`MTA_PARAMS: debug_data_prot`

- Rather than die on data prot (SEGV), wait for the debugger to attach
- It is unwise to have it on all the time because the program will wait, consuming resources, until the debugger attaches

Debugging without core files

`SIGQUIT`

- **If a program appears to be stuck, you can send it the SIGQUIT signal (`mtarun kill -QUIT` *`pid`*) which will print the register state of all threads in the program (does not terminate the program)**
- **Can result in a really large and unmanageable amount of information**

Debugging: Attaching to a "Bad" Program

Attach to a program that has data prot or is ignoring domain signal:

```
% mdb -rm -socket 172.30.79.242:1234 a.out
  …
(mdb) run
```

- Hit `Ctrl-C` to stop the program
- Look for threads in the 'bad' state or threads that are running in the `__data_prot_handler`

Common User Problems

- Users write buggy software. The following problems require close examination:
  - Deadlock/Livelock: no threads making progress
  - Improper synchronization
    - Incorrect initialization (e.g., `purge()` or `writexf()` of `sync`/`future` variable)
    - Off-by-one errors surrounding synchronization
  - Application appears to be making progress, but very slowly
    - Appears mostly on large systems
    - Other symptoms include `mtatop` exiting and “processor not checking in” message on console
  - Hot spotting
    - Use performance tools to isolate.

Less Common User Problems

- User programs may accidentally modify runtime data structures
  - This is a bug in the user program, but manifests itself as a runtime problem
  - These problems are notoriously difficult to find

Less Common User Problems

- **Data prot due to forwarding to a bad location**
  - Address format is typically `0xbad`)
    - Use `mdb info opa`
  - Check for invalid pointer values
    - Use before initializing
    - Incorrect pointer arithmetic
    - Bad type casting
  - Assertions by the runtime
    - Check for similar

Software

- Compiler performs all the standard scalar optimizations (copy propagation, common sub-expression elimination, loop-invariant code motion, etc.)
- When debugging an application
  - For –g1, stores are not moved beyond *sequence points*
  - For –g2, loads and stores are not moved beyond *sequence points*

Compiler Flags for Debugging

- `-g`: Can view variable values inside mdb
  - Internally: compiler won't move stores

```
y = 0;
....
x = 4;
y = 5;
....
z = 6;
```

In mdb, should see y = 0

In mdb, should see y = 5

- `-g2`: Can view and set variables in mdb
  - Internally: won't move loads or stores

```
a = 1;
....
x = y;
....
b = 1;
```

If we set y here in mdb, x should see new y.

If we set y here in mdb, x should see old y.

Software

- By definition, variables that are declared *volatile* are not considered for any optimization
- No reordering (*memory fence*) around loads/stores of sync and future variables
- No reordering around the following MTA generics:
  - `readfe/writeef` (sync load/store)
  - `readff/writeff` (future load/store)
  - `touch`
  - `int_fetch_add` of sync/future

Software

- Reordering of generics that are indifferent to full/empty state is allowed:
  - `readxx/writexf` (normal load/store)
  - `purge`
  - `int_fetch_add` of a normal variable
- Reordering can be enabled via a compiler directive

```
#pragma mta may reorder
```

Example: Enabling Reordering

```
sync int A$[10000];

…

#pragma mta may reorder A
for (i=0; i<10000; i++) {
  A$[i] = i;
}
```

Tells compiler that accesses to A$ may be reordered, enabling parallelization

Loop is not parallelized because accesses to A $ may not be reordered