Course 5: Message Passing

Kristaps Dzonsons

12 December, 2011

Course site: *http://kristaps.bsd.lv/minicourse_12_2011*

Environment

In prior lectures, we focussed on the environment of high-performance computing:

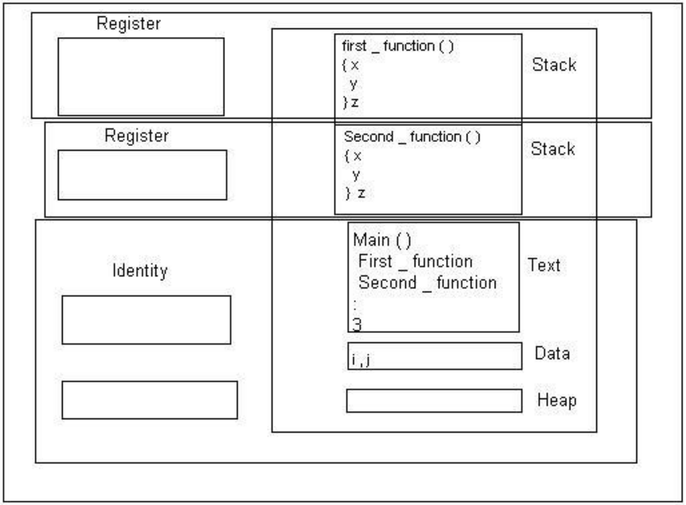
UNIX	an operating system popular in non-desktop environments
C	a low-level, minimal programming language
development	pthreads, fork/join, vector-parallel compiler extensions
hardware	multi-core, multi-processor, NUMA

In this lecture, we consider clustering and message passing.

Review: Threading

Threading is orchestrated by a kernel and compiler. It is an implementation of “shared memory”, where executables have the illusion of operating over a single, contiguous block of memory.

Review: Threading

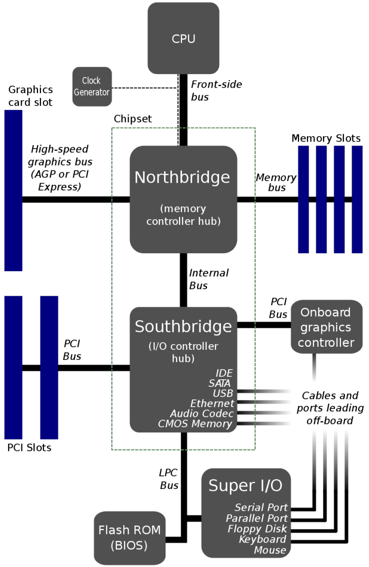


Review: Threading

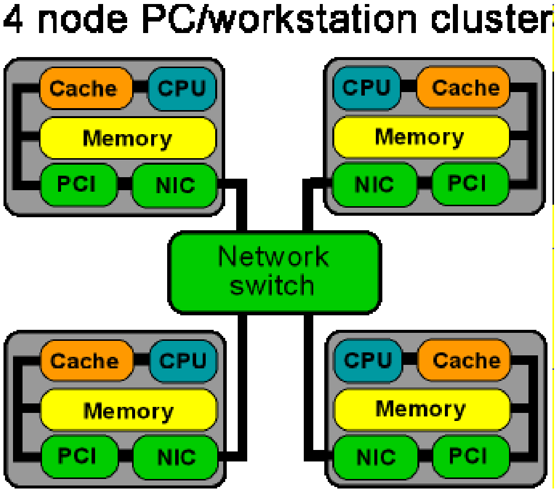
This extends a programmer's natural expectations of process memory to multiple processing units.

What are the problems with this model of computation? Consider: hardware (multiple non-contiguous memory images) and algorithm complexity (locality-aware synchronisation).

Introduction: Single Machine



Introduction: Connected Machines



Introduction: Connected Machines

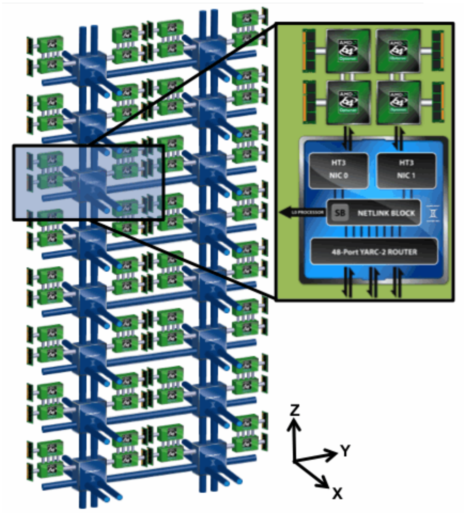


Image courtesy of Cray, Inc.

Introduction: Cluster

A cluster is a disjoint set of memory images. In practise: connected computers.

Extending our awareness of memory to a cluster, we must consider:

interconnect	the hardware device connecting computers
topology	the way in which computers are connected

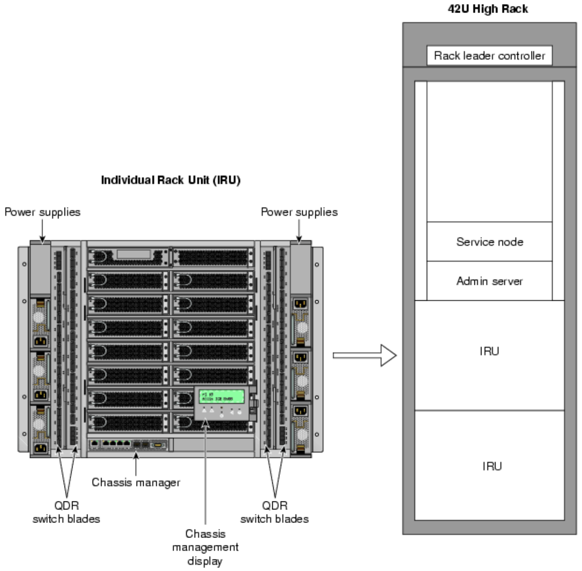
Introduction: Case Study

Consider the following hardware found on KTH's Lindgren supercomputer (Cray XE6).

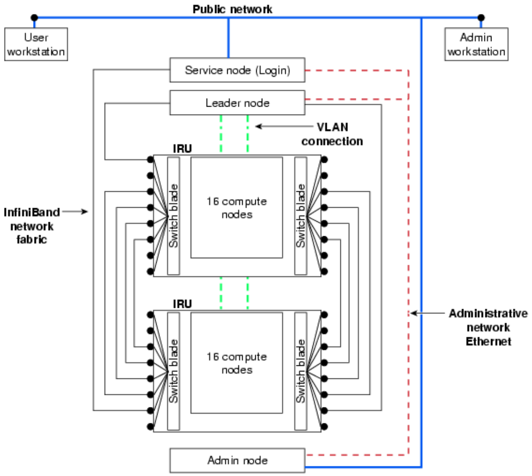
- 1516 compute nodes (processors)
- 36348 compute cores (24-core processors)
- 64 KB L1 cache (per-core)
- 512 KB L2 cache (per-core)
- 24 MB L3 cache (per-processor)
- 32 GB RAM (per-processor)

Introduction: Case Study

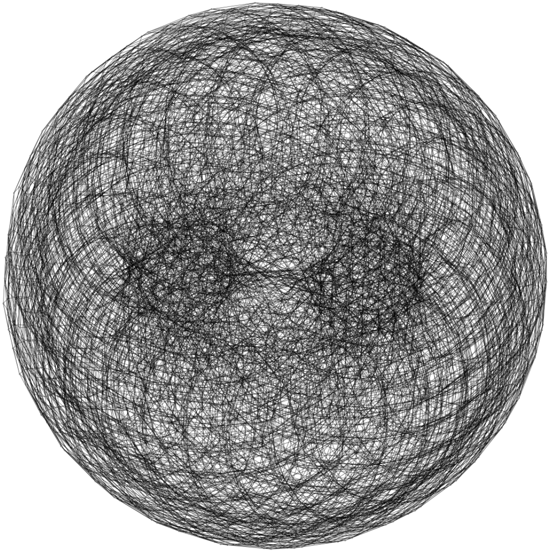
Introduction: Case Study



Introduction: Case Study



Introduction: Case Study

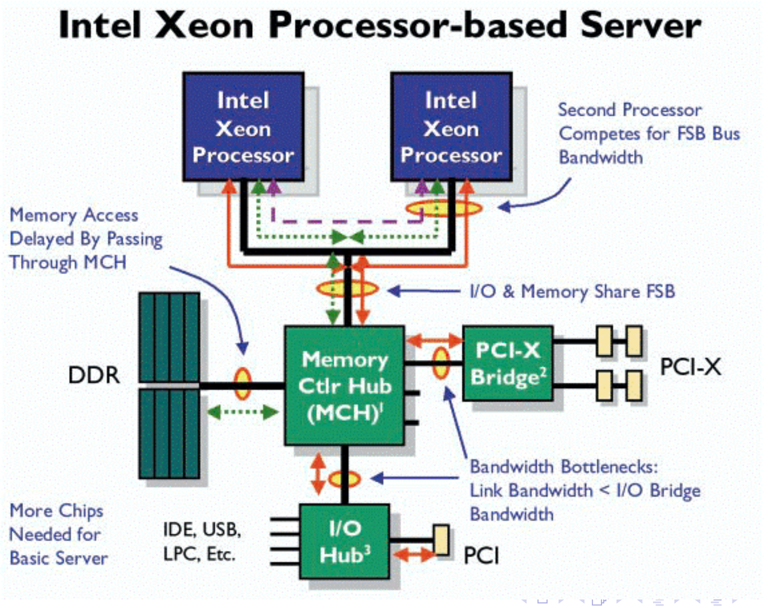


Problem: Scalability

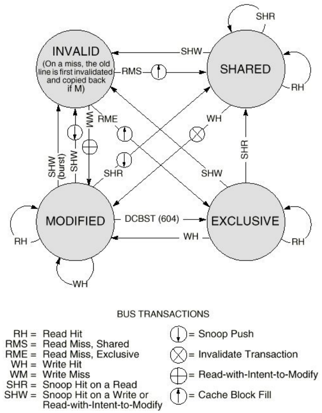
Our simple, single-machine shared memory system had a simple method of cache coherence: keeping cache memory and “backing store memory” in sync.

. . . or was it so simple?

Problem: Scalability


Intel Xeon Processor-based Server
Intel Xeon Processor
Intel Xeon Processor
Second Processor Competes for FSB Bus Bandwidth
Memory Access Delayed By Passing Through MCH
I/O & Memory Share FSB
DDR
Memory Ctlr Hub (MCH)¹
PCI-X Bridge²
PCI-X
Bandwidth Bottlenecks: Link Bandwidth < I/O Bridge Bandwidth
More Chips Needed for Basic Server
IDE, USB, LPC, Etc.
I/O Hub³
PCI

Problem: Scalability

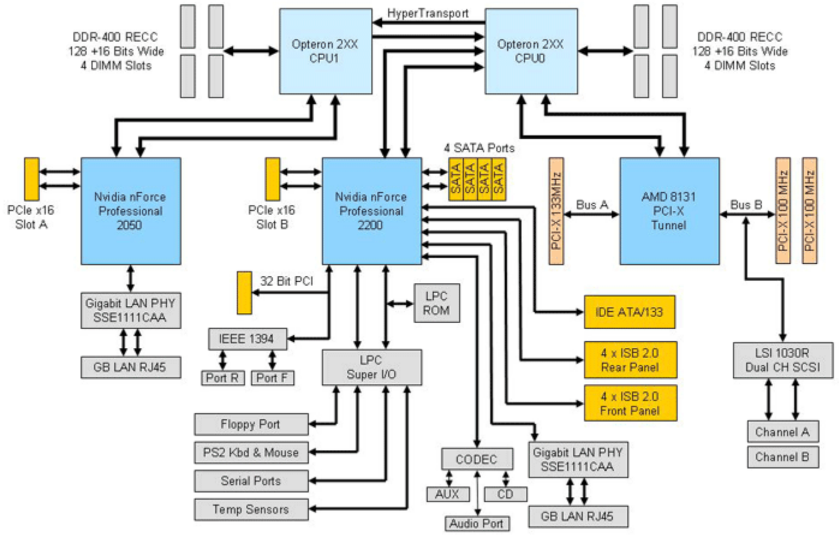


Problem: Scalability

Even in the simple case, we had to rely on a “total view” of cache transfers to maintain coherence.

How do we pull that off on a cluster, or even NUMA? Do we want to at all?

Problem: Scalability



Solution: Message Passing

One solution is to bypass the mess and stop pretending that virtual memory encompasses the entire machine. Instead, pass memory messages between cooperating machines.

How do we do this so that it scales upward (multi-machine, multi-cluster, . . .) and downward (multi-core) without being incredibly complex?

Solution: MPI

The Message Passing Interface (MPI) is a standardised message-passing protocol for high-performance computing.

gamelab uses Open MPI (it doesn't matter: utilities and libraries are standardised). Open MPI also used on RoadRunner (1/500, 2009), K-Computer (1/500, 2011).

Case Study: Source

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

static float
f(int start, int end, const float *v1, const float *v2)
{
        int             i;
        float           res;

        for (res = 0.0, i = start; i < end; i++)
                res += v1[i] * v2[i];

        return(res);
}
```

Case Study: Source

```c
int
main(int argc, char *argv[])
{
        float *v1, *v2, rres, res;
        int i, sz = 50000000, rank, size, start, end;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        v1 = calloc(sz, sizeof(float));
        v2 = calloc(sz, sizeof(float));

        for (i = 0; i < sz; i++)
                v1[i] = v2[i] = (float)i;

        start = (rank * sz) / size;
        end = (rank == size - 1) ? sz : start + (sz / size);
        res = f(start, end, v1, v2);

        MPI_Reduce(&res, &rres, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

        if (0 == rank)
                printf("%g\n", rres);

        MPI_Finalize();
        return(EXIT_SUCCESS);
}
```

Case Study: Running

```
% ftp http://kristaps.bsd.lv/minicourse_12_2011/example8.c
% cc -I/usr/local/include -L/usr/local/lib \
> -g -W -Wall -o example8 example8.c -lmpi -lpthread
% mpirun example8
4.88498e+22
% mpirun -np 2 example8
4.12441e+22
```