K-State Beocat Compute Cluster

2 Head Nodes

ICR-Clymene

ICR-Helios

Slurm

Software and Hardware overview

- Rocky Linux with the Slurm batch scheduler
- Over 10,000 Intel/AMD cores on 310 compute nodes
- 53 TeraBytes of RAM memory
- Low-latency 30-100 Gbps InfiniBand/RoCE network
- 1/10/40 Gbps Ethernet to a 1 PetaByte file server
- 290 TB fast scratch space /fastscratch
- 170 32-bit NVIDIA GPUs and 4 64-bit NVIDIA P100s

310 Compute Nodes accessed through Slurm
4 Interactive nodes accessed only through OnDemand

31 Warlocks

32-128 core Epyc
128 GB - 1.5 TB
40 Gbps RoCE
40 Gbps Ethernet

16 GPUs

39 Wizards

32-64 core Skylake
96 GB - 1.5 TB
100 Gbps OmniPath
10 Gbps Ethernet

103 32-bit GPUs
4 64-bit GPUs

120 Moles

20-core Broadwell
32 GB
32 Gbps QDR
1 Gbps Ethernet

62 Dwarves

32-core Broadwell
128-512 GB
56 Gbps InfiniBand
40 Gbps Ethernet

26 GPUs

54 Heroes

24-core Broadwell
128-512 GB
40 Gbps RoCE
40 Gbps Ethernet

BeoShock Compute Cluster

2 Head Nodes

Headnode01

Headnode02

Slurm

Software and Hardware overview

- Rocky Linux with the Slurm batch scheduler
- 768 Intel cores on 21 compute nodes
- Over 7.2 TeraBytes of RAM memory
- Low-latency 10 Gbps Ethernet network
- 150 TeraByte file server
- 8 High-End NVIDIA GPUs for accelerating scientific apps

21 Compute Nodes accessed through Slurm. Also Available interactively through OnDemand graphical interface.

16 Compute nodes

36 cores
186 GB RAM
10 Gbps Ethernet

2 GPU nodes

2 NVIDIA V100s
16.4 GB GPU mem

36 cores
375 GB RAM
10 Gbps Ethernet

1 Physics node

4 NVIDIA A30 GPUs
24.6 GB GPU mem

48 cores
515 GB RAM
10 Gbps Ethernet

2 High-Memory nodes

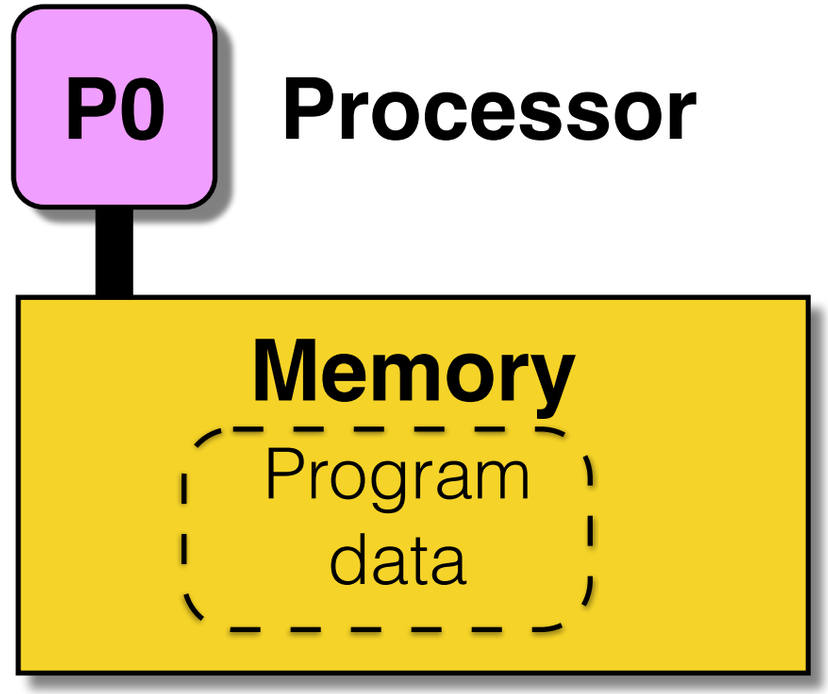
36 cores
1500 GB
10 Gbps Ethernet

Scalar Computers

One processor per computer
Operations performed one at a time
Example: Vector addition

- Load X_0
- Load Y_0
- Add X_0 + Y_0
- Store into Z_0
- Repeat for each element i

Simple and easy to program



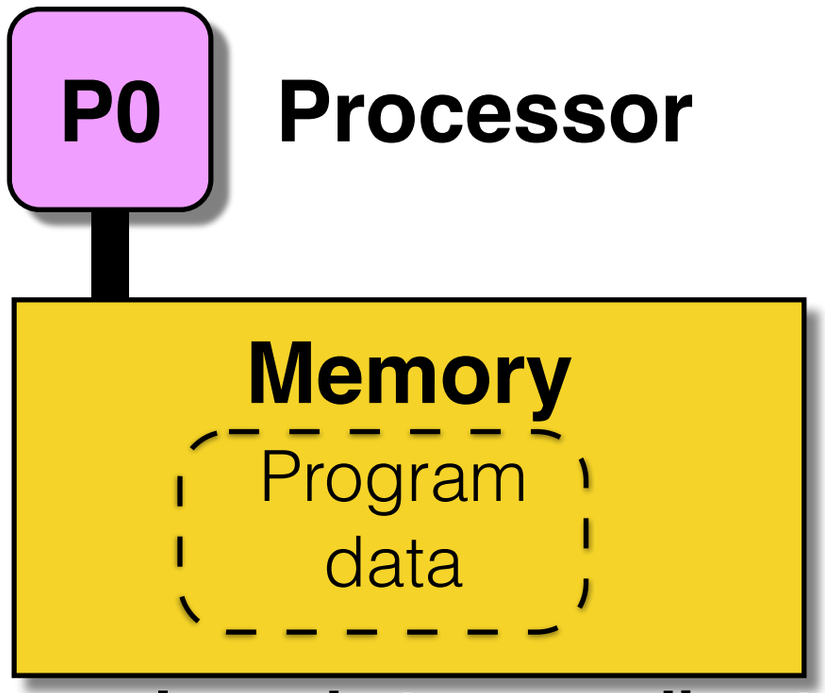
$$Z_i = X_i + Y_i$$

$$\begin{pmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \\ Z_4 \\ Z_5 \\ Z_6 \\ Z_7 \\ \dots \end{pmatrix} = \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \\ \dots \end{pmatrix} + \begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_6 \\ Y_7 \\ \dots \end{pmatrix}$$

Scalar Computers

SELENE: /homes/

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

main()
{
   int i, n = 1000000;
   double *x, *y, *z;

   printf( "Starting vector add\n");

// Allocate memory for the vectors

   x = malloc( n * sizeof(double) );
   y = malloc( n * sizeof(double) );
   z = malloc( n * sizeof(double) );

// Initialize the vectors

   for( i = 0; i < n; i++ ) {
      x[i] = (double) i;
      y[i] = (double) i * (double) i;
   }

// Now do the vector addition

   for( i = 0; i < n; i++ ) {
      z[i] = x[i] + y[i];
   }

// Print out a few elements of vector z

   for( i = 0; i < 100; i = i + 10) {
      printf( "z[%d] = %lf\n", i, z[i]);
   }

   exit(0);
}
```



Copy example code to your directory

beocat> cp -rp /homes/daveturner/beocat_workshop ~
beocat> cd beocat_workshop

Compiling vec_add.c

beocat> module load iomkl
beocat> icc **vec_add.c** -o **vec_add_icc**

Running vec_add.c

beocat> ./**vec_add_icc**
beocat> sbatch sb.vector

Compile with gcc

beocat> module load foss
beocat> gcc **vec_add.c** -o **vec_add_gcc**
beocat> ./**vec_add_gcc**

Vector Computers

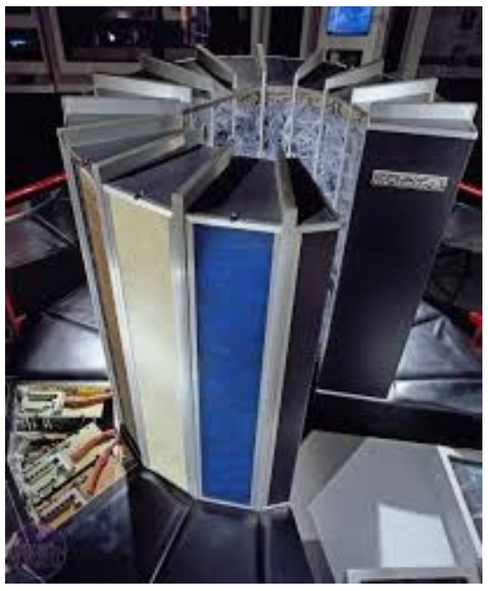
Vector computations instead of scalars

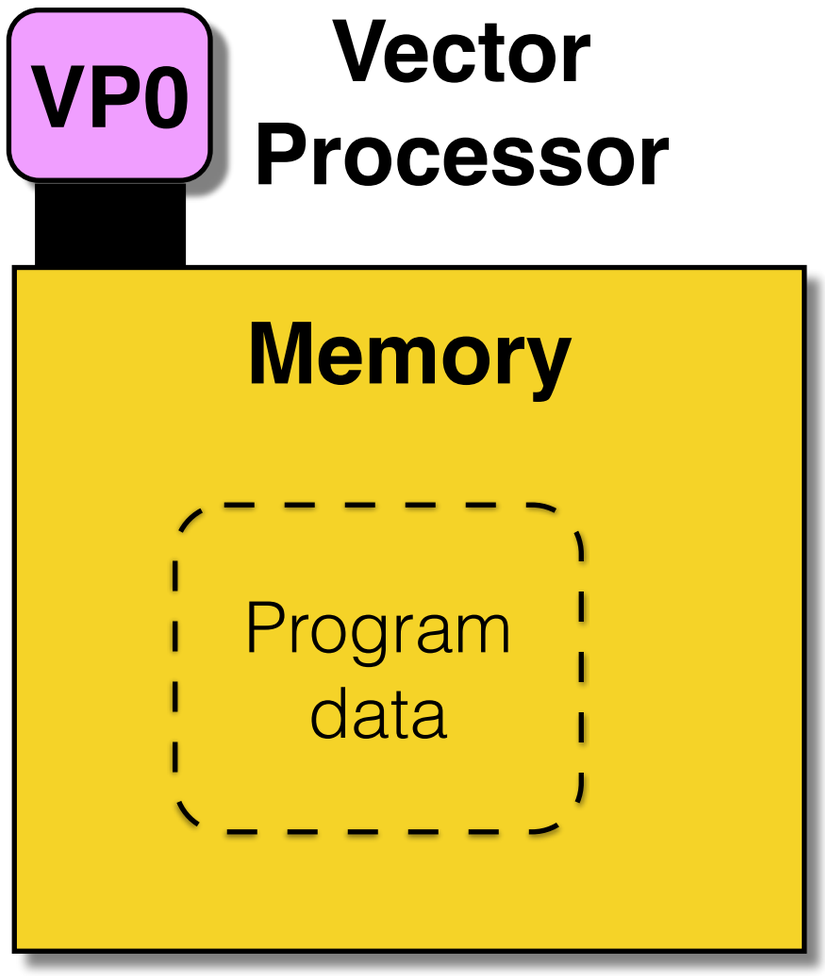
- Old Cray systems had vectors of 64 doubles
- The vector compiler provides great help

Example: Vector addition

- Load X_0 through X_{63}
- Load Y_0 through Y_{63}
- Vector unit adds each element
  X_0+Y_0, X_1+Y_1, …, $X_{63}+Y_{63}$
- Store the vector into Z_0 through Z_{63}
- Repeat for the next 64 elements
- **Potentially 64 times faster**

$$\begin{pmatrix} Z_0 \\ \vdots \\ Z_{63} \\ Z_{64} \\ \vdots \\ Z_{127} \\ \cdots \end{pmatrix} = \begin{pmatrix} X_0 \\ \vdots \\ X_{63} \\ X_{64} \\ \vdots \\ X_{127} \\ \cdots \end{pmatrix} + \begin{pmatrix} Y_0 \\ \vdots \\ Y_{63} \\ Y_{64} \\ \vdots \\ Y_{127} \\ \cdots \end{pmatrix}$$





Performance on Vector Computers

An application with 3 loops taking 30 seconds, 20 seconds, and 50 seconds

- If loops 1 and 2 are vectorized but not loop 3
  30/64 seconds + 20/64 seconds + 50 seconds => 50.78 seconds
- If all 3 loops are vectorized
  30/64 seconds + 20/64 seconds + 50/64 seconds => 1.56 seconds
- Even if all loops are vectorized you may not get 64x speedup
  since the scalar sections between loops may become significant

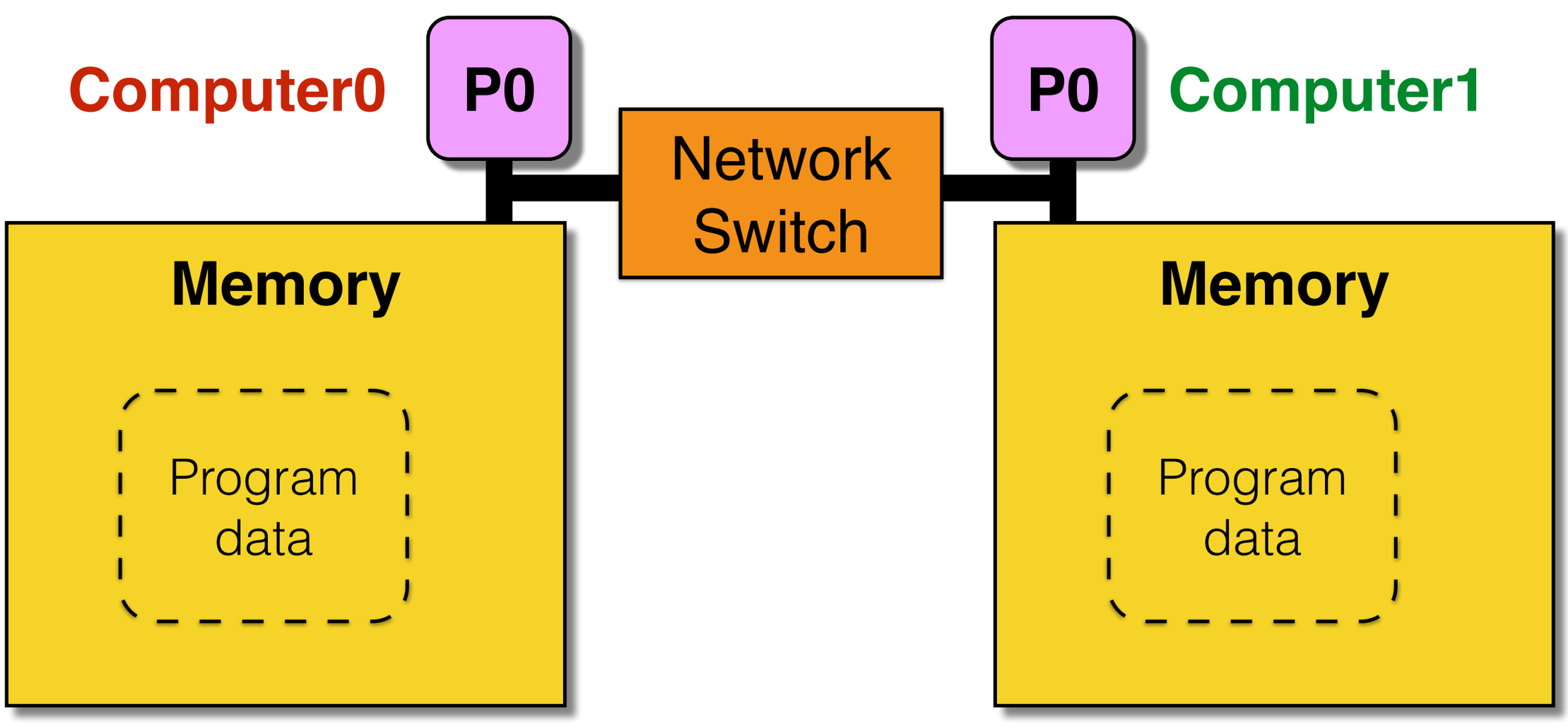
Cluster Computers

Many computers networked together

Each computer runs the same program but operates on
a different part of the data.

The programmer must distribute the data across the nodes,
divide the workload, and choreograph the communications.

Computers must exchange data to perform most calculations
32 Gbps InfiniBand or **40 Gbps** Ethernet, **1.5 *µs*** latencies

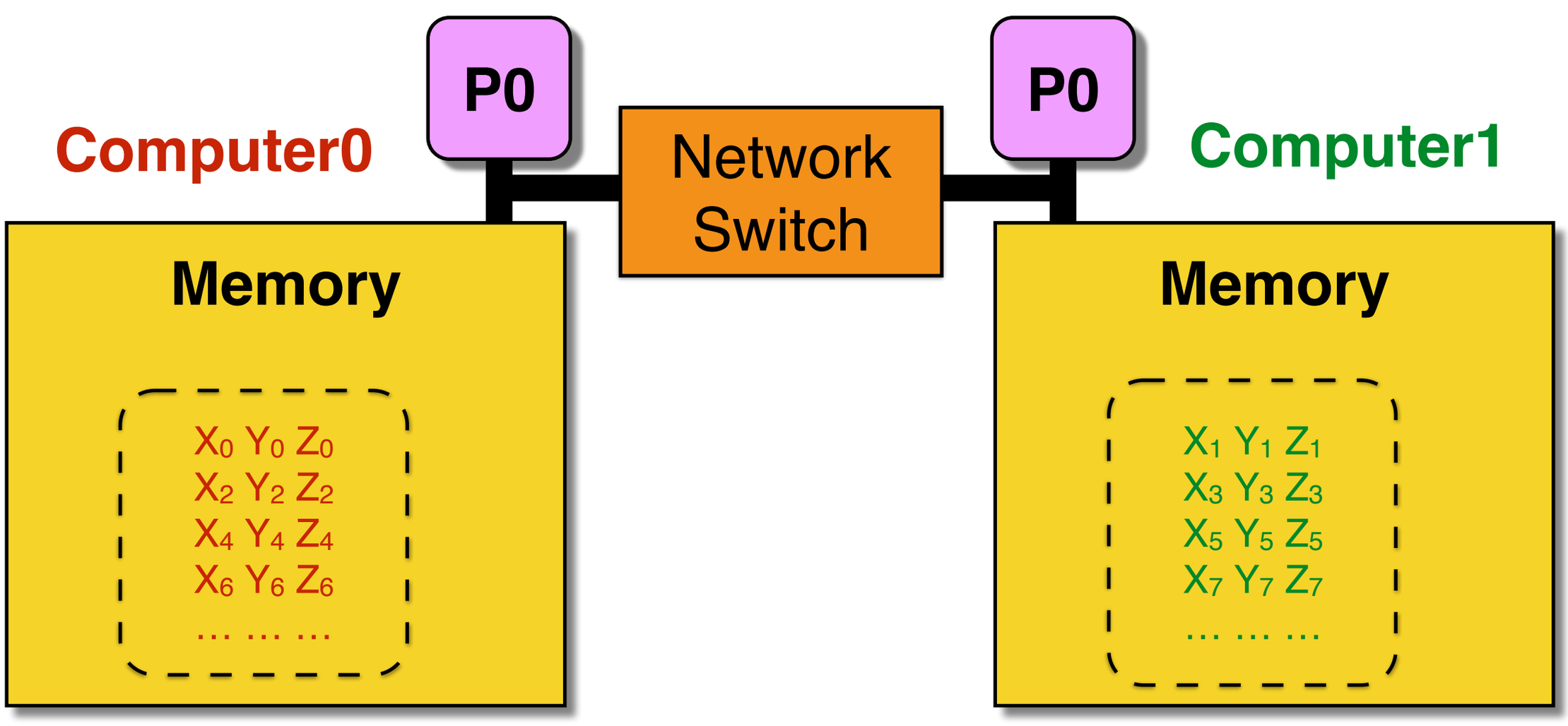


Cluster Computers

Example: Vector addition

- Computer0 loads X_0 and Y_0
- Computer0 adds $X_0 + Y_0$
- Computer0 stores into Z_0
- Computer0 repeats for even elements
- At the same time, computer1 follows the same recipe for all odd elements

$$\begin{pmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \\ Z_4 \\ Z_5 \\ Z_6 \\ Z_7 \\ \dots \end{pmatrix} = \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \\ \dots \end{pmatrix} + \begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_6 \\ Y_7 \\ \dots \end{pmatrix}$$



Cluster Computers

Example: Matrix multiplication

Could broadcast each X and Y block to all nodes

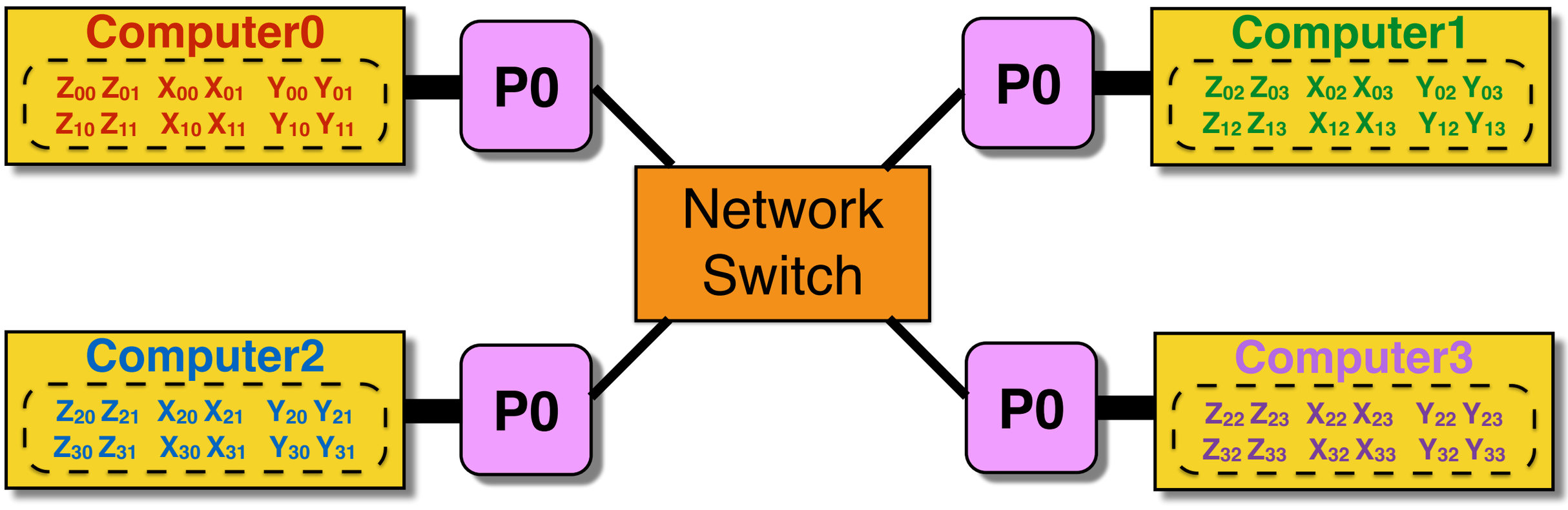
—> More memory and communication than necessary

The best algorithm is complicated but efficient

- Broadcast diagonal blocks of X down each row
- Multiply X blocks with each node's Y block
- Shift the Y block to the node above it
- Repeat with the X blocks to the right of diagonal

$$Z_{ij} = \sum_{k=0}^{3} X_{ik} * Y_{kj}$$

$$\begin{pmatrix} Z_{00} & Z_{01} & Z_{02} & Z_{03} \\ Z_{10} & Z_{11} & Z_{12} & Z_{13} \\ Z_{20} & Z_{21} & Z_{22} & Z_{23} \\ Z_{30} & Z_{31} & Z_{32} & Z_{33} \end{pmatrix} = \begin{pmatrix} X_{00} & X_{01} & X_{02} & X_{03} \\ X_{10} & X_{11} & X_{12} & X_{13} \\ X_{20} & X_{21} & X_{22} & X_{23} \\ X_{30} & X_{31} & X_{32} & X_{33} \end{pmatrix} * \begin{pmatrix} Y_{00} & Y_{01} & Y_{02} & Y_{03} \\ Y_{10} & Y_{11} & Y_{12} & Y_{13} \\ Y_{20} & Y_{21} & Y_{22} & Y_{23} \\ Y_{30} & Y_{31} & Y_{32} & Y_{33} \end{pmatrix}$$



Cluster Computers

SELENE: /homes/daveturner/biol890/compact

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

main (int argc, char **argv)
{
   int myproc, nprocs, token, tag = 0, source, destination;
   MPI_Status status;

// Get the number of processes in this run and determine which process I am

   MPI_Init (&argc, &argv);
   MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
   MPI_Comm_rank (MPI_COMM_WORLD, &myproc);

// Have each process print a Hello message to the screen

   printf ("Hello from %d of %d\n", myproc, nprocs);
   if( nprocs != 2 ) exit(-1);   // Make sure we only have 2 processes

// Now bounce a message from proc 0 to proc 1 and back

   token = 0;
   if( myproc == 0 ) {

      destination = 1;  // Sending to process 1
      MPI_Send (&token, 1, MPI_INTEGER, destination, tag, MPI_COMM_WORLD);

      source = 1;  // Receiving back from process 1
      MPI_Recv (&token, 1, MPI_INTEGER, source, tag, MPI_COMM_WORLD, &status);

   } else {   // The 2nd process

      source = 0;  // Receiving from process 0
      MPI_Recv (&token, 1, MPI_INTEGER, source, tag, MPI_COMM_WORLD, &status);

      token = token + 1;

      destination = 0;  // Sending back to process 0
      MPI_Send (&token, 1, MPI_INTEGER, destination, tag, MPI_COMM_WORLD);
   }

   printf( "Process %d has token = %d\n", myproc, token);

   MPI_Finalize( );
   exit(0);
}
```

Compiling token_pass.c

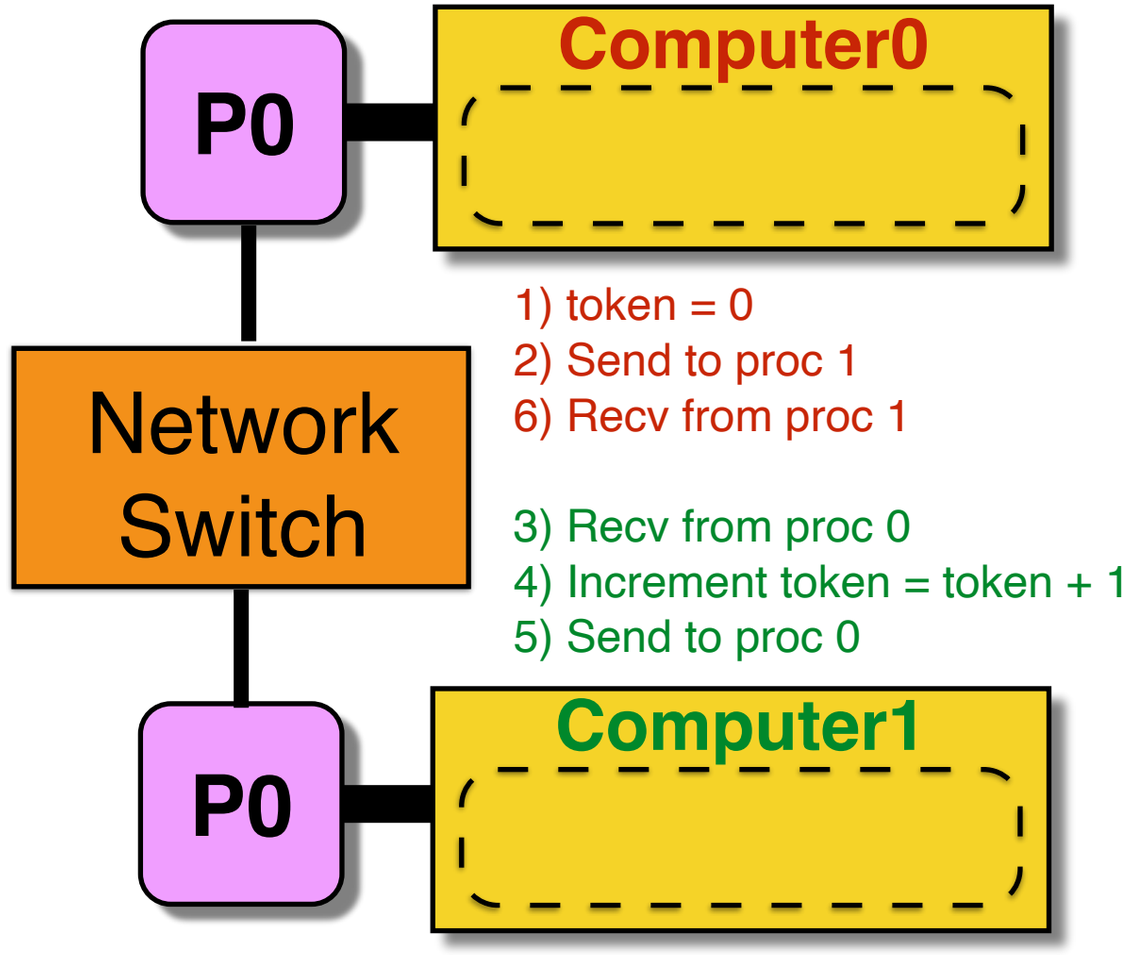
beocat> module load iomkl
beocat> mpicc **token_pass.c** -o **token_pass**

Run on 2 processes on the head node

beocat> mpirun -np 2 ./**token_pass**

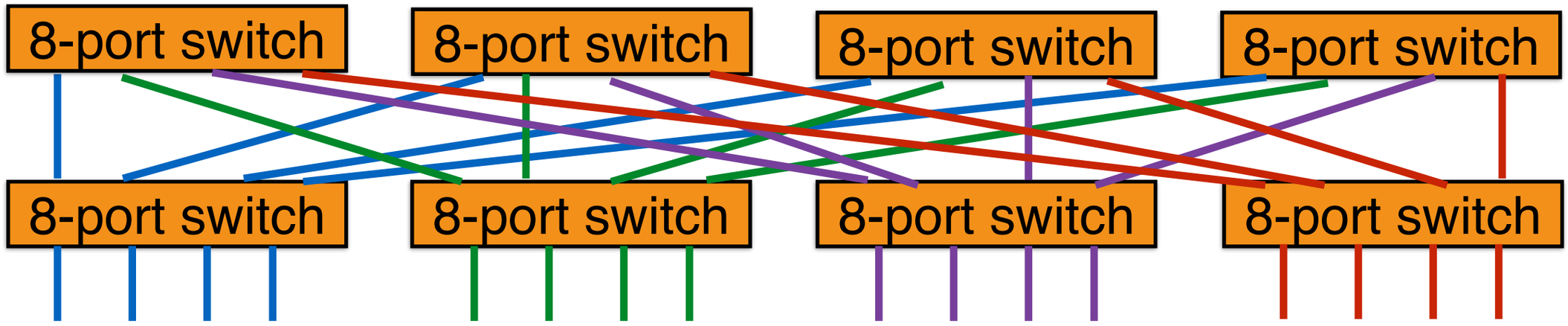
Communicate between 2 nodes

beocat> sbatch sb.token

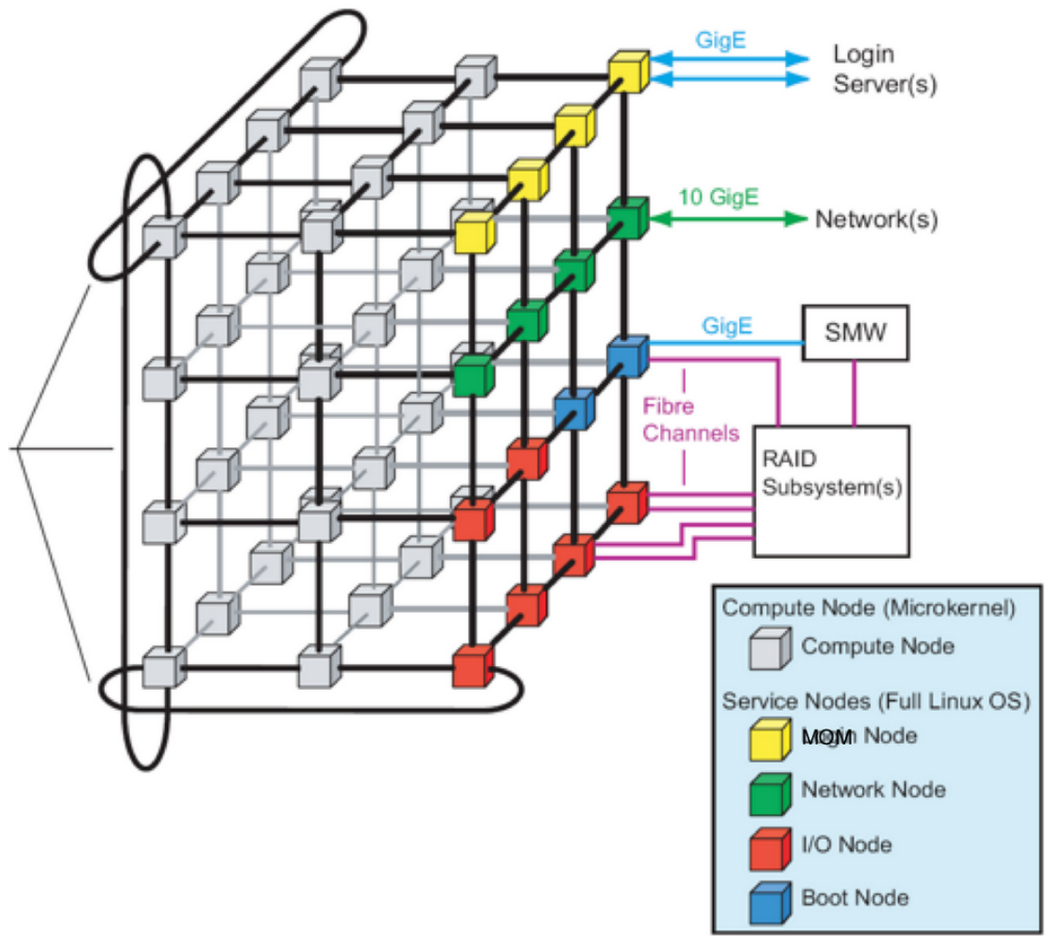


Network Topology

Layered switches for clusters - same bandwidth up as out



Distributed-memory supercomputers - Blue Waters 3D Torus



Multicore Computers

Many processing cores per computer

One program using many cores.

Each process usually does the same task
on different data.

Can be programmed in MPI where each process
runs its own program and has its own memory.

★Many programs use MPI for historical reasons

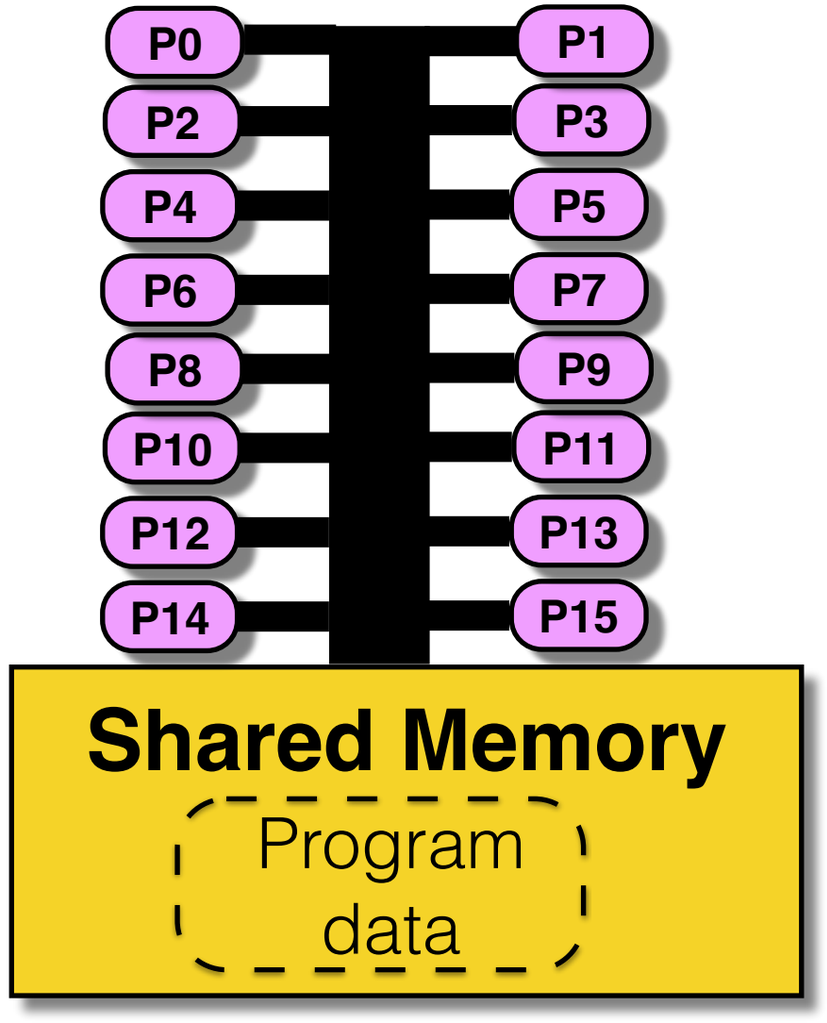
★MPI copies data between process's memory space

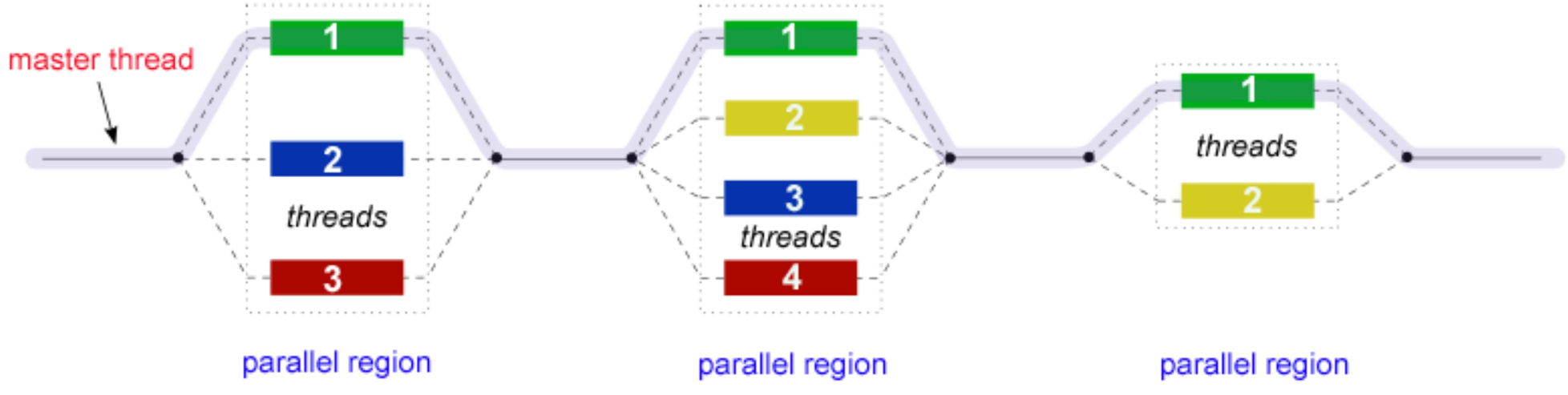
Can be programmed in OpenMP where there
is one program that sometimes uses all processes
with all sharing the same memory.

★Easier to program since all data is shared

★Efficient since data is not moved around

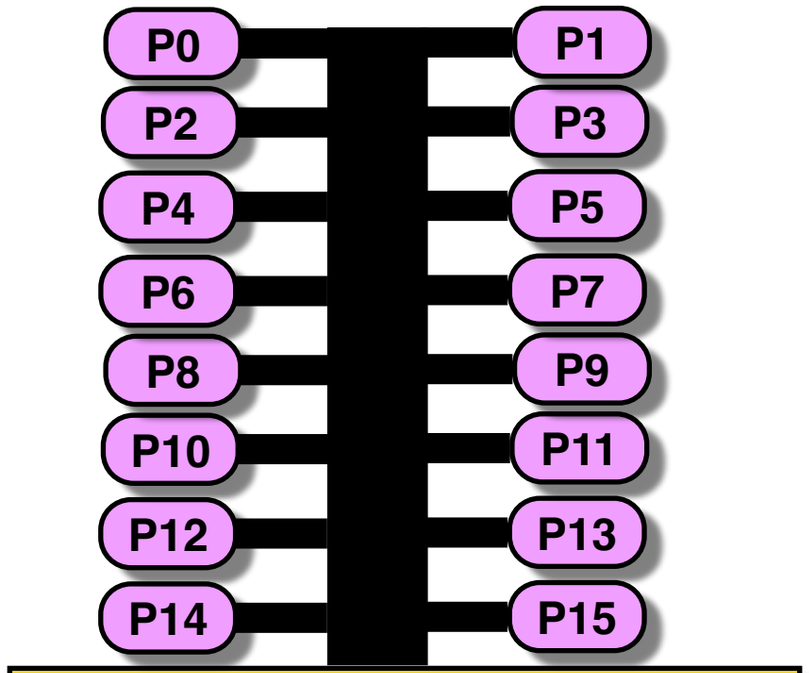
★Can only be run on one compute node





Multicore Computers

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

main()
{
   int i, n = 1000000;
   double *x, *y, *z;
   int nthreads;

// Set the number of threads to use

   nthreads = 4;
   omp_set_num_threads( nthreads );

   printf(" nthreads at the start is %d of %d\n",
            omp_get_num_threads(), omp_get_max_threads( ));

// Allocate memory for the vectors

   x = malloc( n * sizeof(double) );
   y = malloc( n * sizeof(double) );
   z = malloc( n * sizeof(double) );

// Initialize the vectors

   for( i = 0; i < n; i++ ) {
      x[i] = (double) i;
      y[i] = (double) i * (double) i;
   }

// Now do the vector addition using multiple threads

#pragma omp parallel for
   for( i = 0; i < n; i++ ) {
      if( i == 0 ) printf(" nthreads in the for loop is %d of %d\n",
                          omp_get_num_threads(), omp_get_max_threads( ));
      z[i] = x[i] + y[i];
   }

// Print out a few elements of vector z

   for( i = 0; i < 100; i = i + 10) {
      printf( "z[%d] = %lf\n", i, z[i]);
   }

   exit(0);
}
```



Compiling vec_add_omp.c

beocat> module load iomkl
beocat> icc **vec_add_omp.c** -o **vec_add_omp** **-qopenmp**

beocat> module load foss
beocat> gcc **vec_add_omp.c** -o **vec_add_omp** **-fopenmp**

Run on 4 cores in 1 compute node

beocat> ./**vec_add_omp**

//omp_set_num_threads() line and recompile

beocat> ./**vec_add_omp**
beocat> export OMP_NUM_THREADS=2
beocat> ./**vec_add_omp**

Multicore Clusters

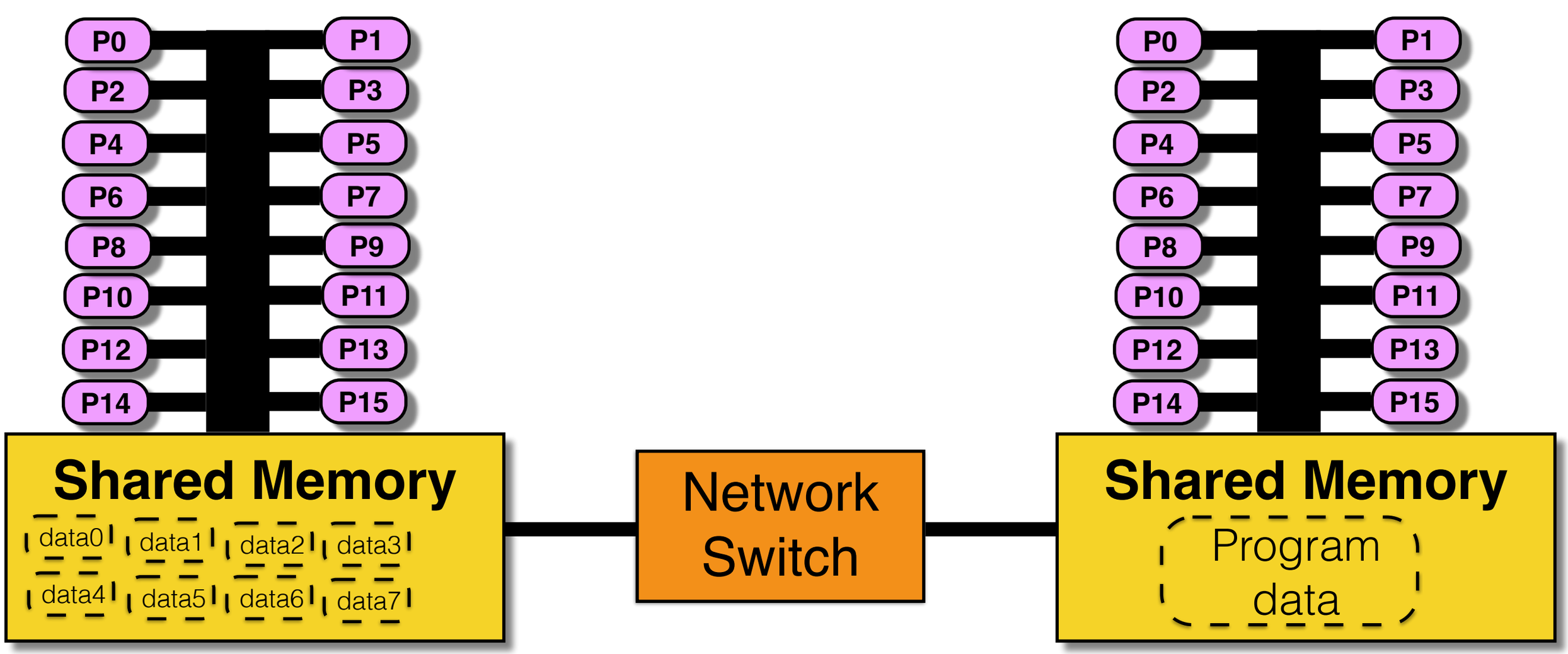
Many multicore computers networked together

Can run MPI between nodes and MPI between cores

- ★ Easier since hybrid programming is not needed
- ★ Can be less efficient

Can run MPI between compute nodes and OpenMP within a compute node

- ★ Most efficient but requires 2 levels of parallel programming



Vector Multicore Computers

Vector computations on each core

- Broadwell processors have 256-bit units
  vectors of 4 doubles or 8 floats
- Intel Phi processors have 512-bit units
  vectors of 8 doubles or 16 floats
- The vector compiler provides help
- **Potential 4x, 8x, or 16x speedup!!!**

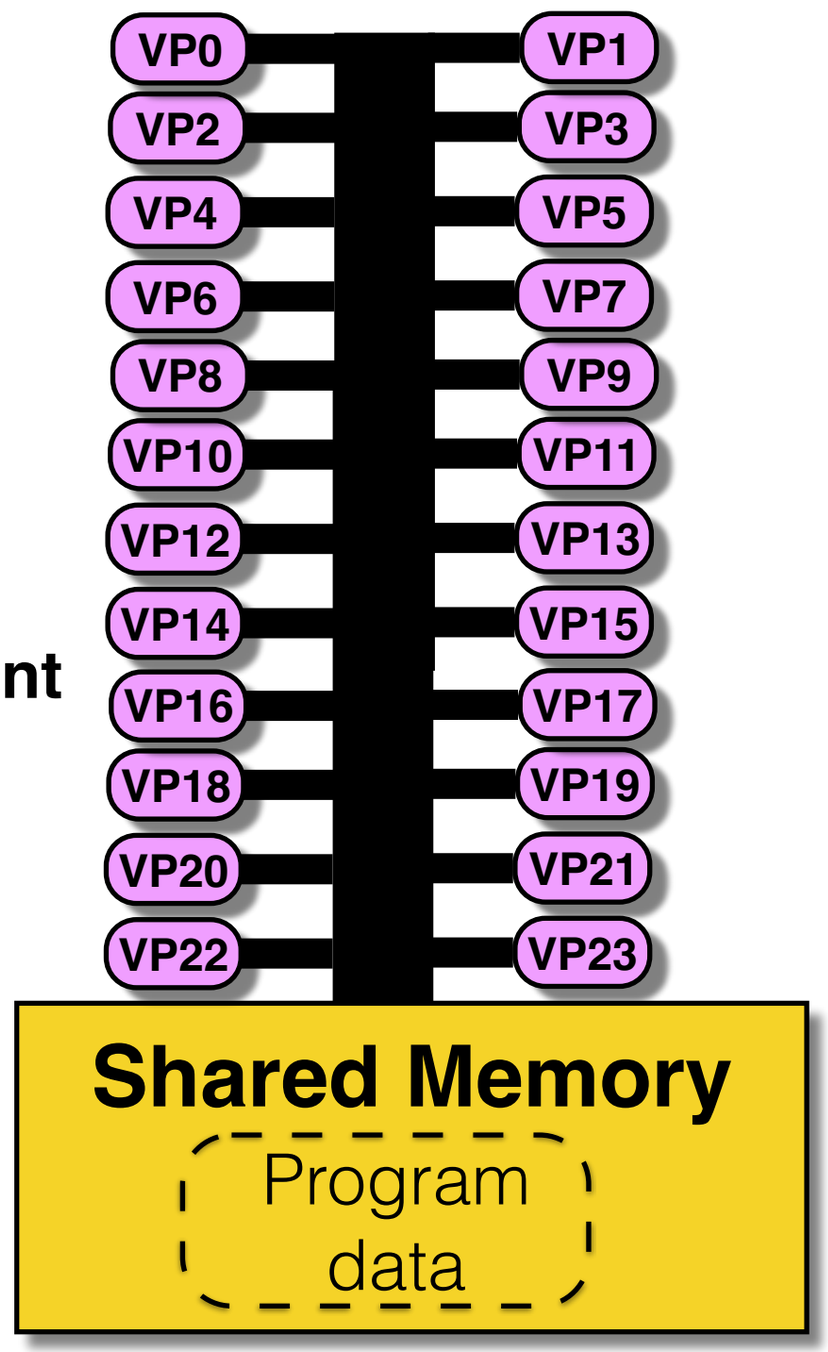
Intel vector compiler is buggy and hard to use

icc -O3 -mavx -qopt-report=5 **code.c** -o **code.x**

Requires compiler directives and memory alignment

```
x = (double *) _mm_malloc( n * sizeof(double), 64);
__assume_aligned( x, 64);
__assume_aligned( y, 64);
#pragma ivdep
for( j = 0;  j < n;  j++ ) {
   x[j] += y[j] * y[j];
}
```

24-core Broadwell processor



Memory Hierarchy

Memory closer to the processor is faster
but more expensive, so smaller

Two Haswell E5-2680 processor chips

—> 2 x 12-core Xeon chips ==> 24 cores

—> 168 registers per core

—> **128 KB L1 cache at 700 GB/sec**

—> **1 MB L2 cache at 228 GB/sec**

—> **12 MB L3 cache at 112 GB/sec**

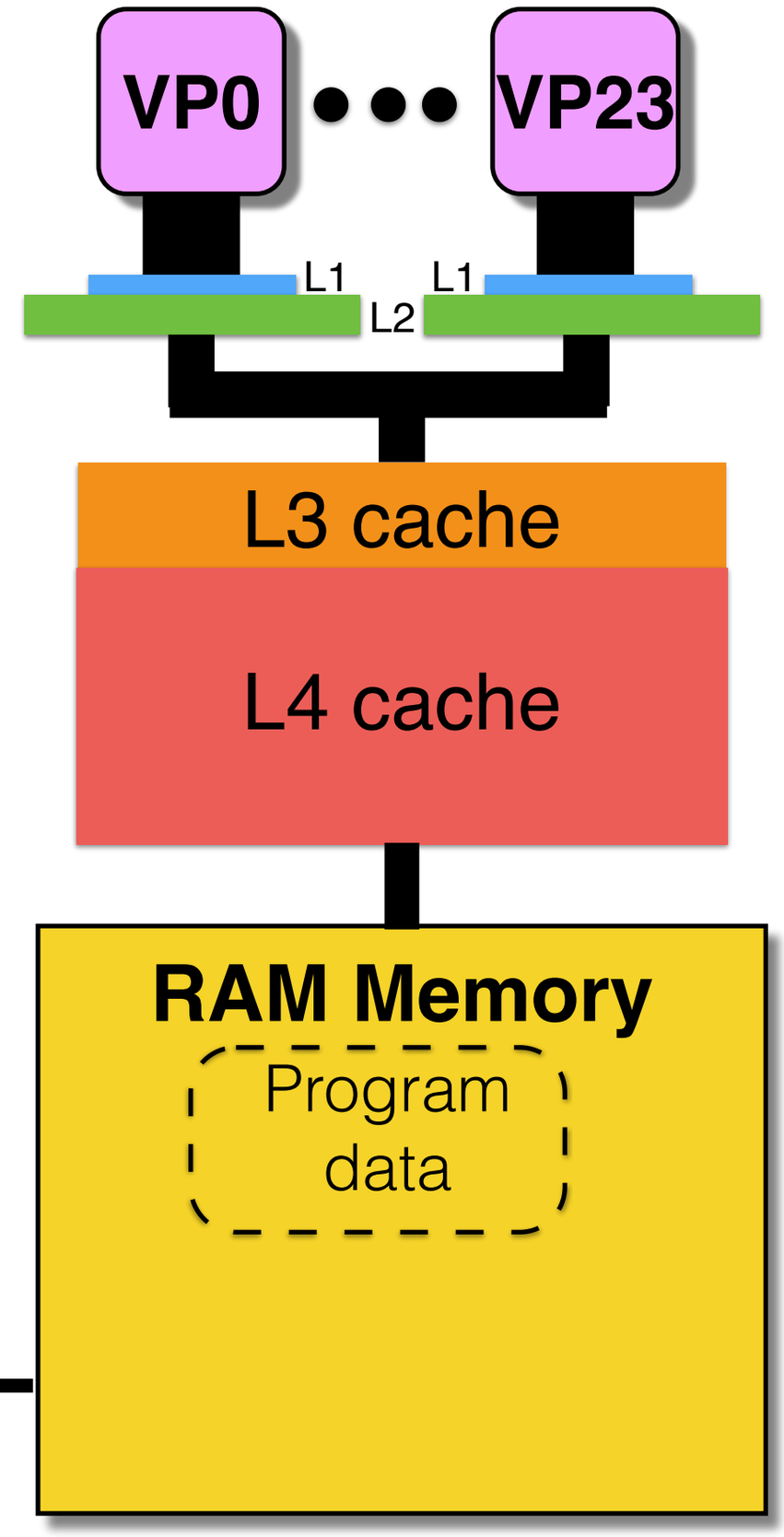
—> **256 MB L4 cache at 42 GB/sec**

—> **up to 768 GB RAM at 17 GB/sec**

—> **40 Gbps network or 5 GB/sec**

$Z_i = X_i + Y_i$ leaves data in the caches

$W_i = X_i * Y_i$ would start with the data in cache



Parallel Computing

The goal of parallel computing is to get an **N times speedup** on **N processes**.
Each compute node runs an identical program.
Each node has a process number which is used to divide the work up.
Nodes must exchange data across the network to perform most calculations.

An MPI communication package is used to do communications in a portable manner.
Overview of message passing with MPI: **https://computing.llnl.gov/tutorials/mpi/**

Programming multicore systems: MPI or OpenMP
Overview of OpenMP: **https://computing.llnl.gov/tutorials/openMP/**

Communications Between Processes

Communication between compute nodes

- ★ **30 Gbps** InfiniBand with a **1.5 *μs*** latency
- ★ **40 Gbps** Ethernet (RoCE) with a **1.5 *μs*** latency

Communication within a node

- ★ MPI: **60 Gbps** with a **0.4 *μs*** latency
- ★ OpenMP: Data has shared access
- ★ MPI-3 also can do shared memory programming

Network topology

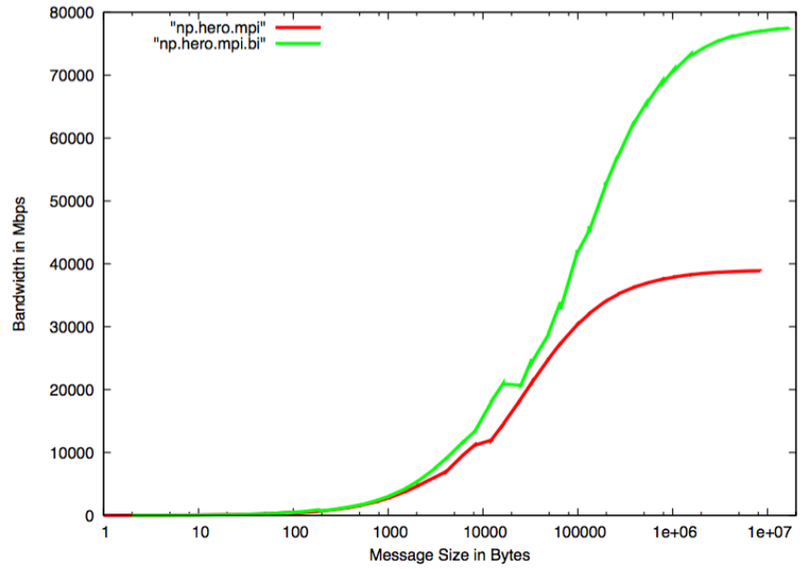
- ★ Clusters have switch hierarchies
- ★ Cray systems can have 3-dimensional grids
- ★ Mapping the algorithm to the network topology is critical for large supercomputers to keep data communications local. This minimizes the contention for communication paths.

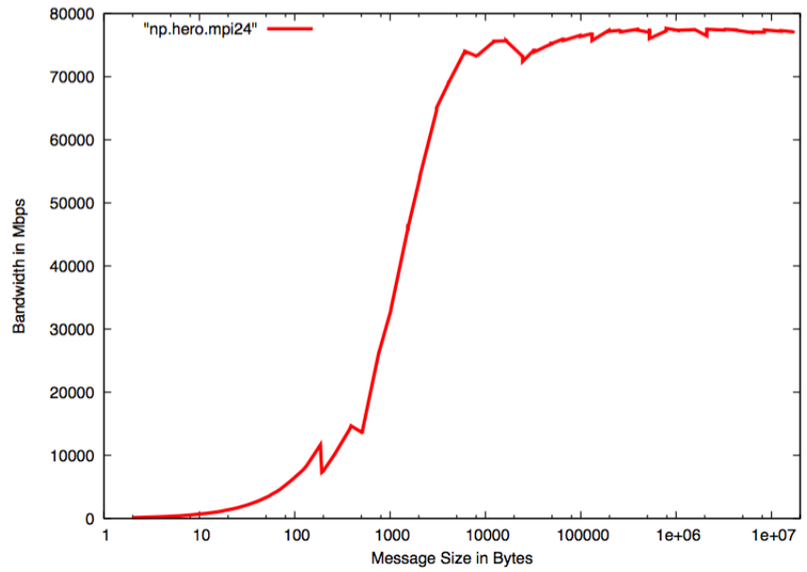
Accessing the file server

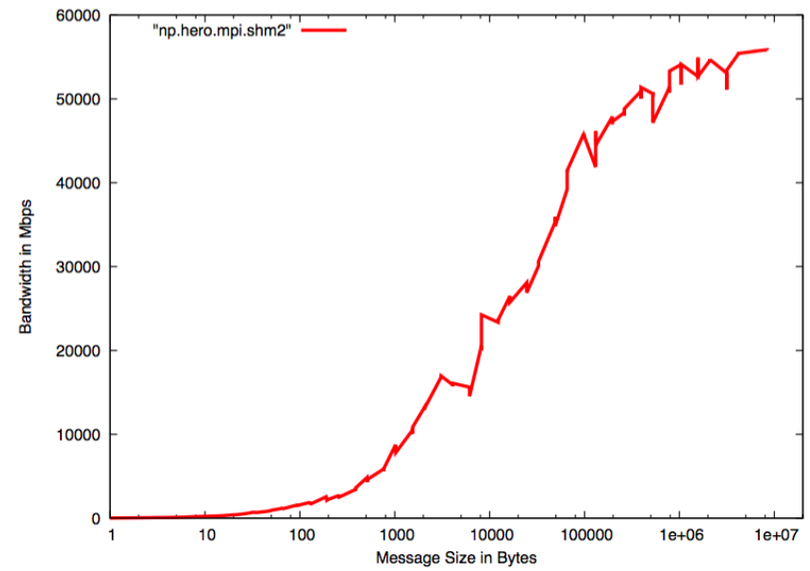
- ★ **10 Gbps** Ethernet for the Elf and Mage nodes
- ★ **40 Gbps** Ethernet for the Hero nodes

Accessing Beocat from the outside world

- ★ **10 Gbps** Ethernet soon to be **100 Gbps** Ethernet







Application Scaling

or How many processes should I run on???

Measure the run time of an application on 1, 2, 4, 8, and 16 cores.

★ Time your run using the 'real' part of the *time* function

★ **time** ./**your_executable**

```
real   0m7.261s
user   0m0.000s
sys    0m0.010s
```

Calculate the speedup compared to the single core run.

	Run Time	Speed up	Memory
1 core	10 hours		10 GB
2 cores	5 hours	2.0x	12 GB
4 cores	2.8 hours	3.6x	14 GB
8 cores	1.5 hours	6.7x	14 GB
16 cores	1.1 hours	9.1x	14 GB

Not much speedup when you double the cores from 8 to 16.
The scaling will change with the size of system you are using.
Also check the memory utilization.

Application Scaling

or How many compute nodes should I run on???

Measure the run time of an application on 1, 2, and 4 compute nodes.
Calculate the speedup compared to the single compute node run.

Scaling to multiple nodes will be difficult on Beocat

- ★ Lots of powerful cores on each node can overwhelm the network
- ★ Getting a multi-node job through the queue can be slow

Spreading the memory across compute nodes may be necessary.

Job Interactions

Jobs can interact with other jobs running on the same compute node.

★You own the processor cores you are allocated, so other jobs can't use them.

- If another job exceeds its memory allocation it can affect yours.
- All jobs on a compute node share the memory bandwidth.
- All jobs on a compute node share the network connection.

—> Multiple jobs doing heavy IO to the file server could slow each other down.

You can check to see if your jobs interact with themselves.

★Run an 8 core job alone on one compute node (request all cores but run on 8)

★Run two 8-core jobs on the same machine at the same time

You must submit both jobs to a given node that you know is open.

★Does it take longer to run two 8-core jobs than one 8-core job???

Application Profiling

You can also add profiling to a code to measure the performance of each function. An application programmer would do this at the start of optimizing an application.

```
#include “mytimer_c.h”

Timer_Start( “section_name” );
    section to be timed
Timer_Stop( );
   ...
Timer_Report( “time.out” );
```

```
Selene m time.abyss
    time in seconds     +children       calls    name
    409.606 ( 19.9%)     416.676    362833951  NSC handle
    391.084 ( 19.0%)     783.976            1  cntlCoverage
    314.702 ( 15.3%)    2043.067            1  runControl
    221.231 ( 10.8%)     221.231     84895836  rcvBufMsg
    210.667 ( 10.2%)     403.334            1  GenerateAdjacency
    195.801 (  9.5%)     195.801    826749932  checkMessage
    111.441 (  5.4%)     945.150    741854096  NSC pumpNetwork
     60.177 (  2.9%)     594.387            2  NetworkAssembly
     47.114 (  2.3%)      58.690    298011483  procBranchAssembly
     15.529 (  0.8%)      69.064           15  removeMarked
     12.860 (  0.6%)      25.644      3786906  assembleContig
     12.291 (  0.6%)      12.291            2  mergeFAfiles
      8.769 (  0.4%)       9.088     31339019  erode
      7.579 (  0.4%)       7.579            1  NetworkPopBubbles
      6.263 (  0.3%)      11.416           15  NetworkTrim
      5.653 (  0.3%)      34.667            1  NetworkDiscoverBubbl
      5.215 (  0.3%)       5.215      2290507  markNeighbors
      4.667 (  0.2%)     267.039            3  LoadSequences
      4.275 (  0.2%)       9.507            1  splitAmbiguous
      4.220 (  0.2%)      23.373            2  markAmbiguous
      3.513 (  0.2%)      30.684            2  Erode
      1.886 (  0.1%)    2057.281            1  Abyss Total
      1.022 (  0.0%)       3.970            2  NSC cntlErode
      0.946 (  0.0%)      75.370           15  NSC rmMarked
      0.291 (  0.0%)       0.434            2  completeOperation
      0.215 (  0.0%)       0.215       525381  NSC branchTrim
      0.144 (  0.0%)      87.814           15  NSC ctrlTrimRound
      0.080 (  0.0%)       0.080            1  openBubbleFile
      0.036 (  0.0%)       0.036            1  SeqColHash
      0.001 (  0.0%)       0.001          943  NSC parseControlMsg
      0.000 (  0.0%)      87.815            2  NSC ctrlTrim
      0.000 (  0.0%)       0.000            4  cleanup

Total time is 2057.280645 secs    (0 hours  34 mins  17.281 secs) on hero08
Timer overhead ~ 0.100000 usecs  no cutoff for now
```

Additional Information

Oklahoma University - Supercomputing in Plain English
http://www.oscer.ou.edu/education.php

message passing with **MPI**: **https://computing.llnl.gov/tutorials/mpi/**

Overview of **OpenMP**: **https://computing.llnl.gov/tutorials/openMP/**