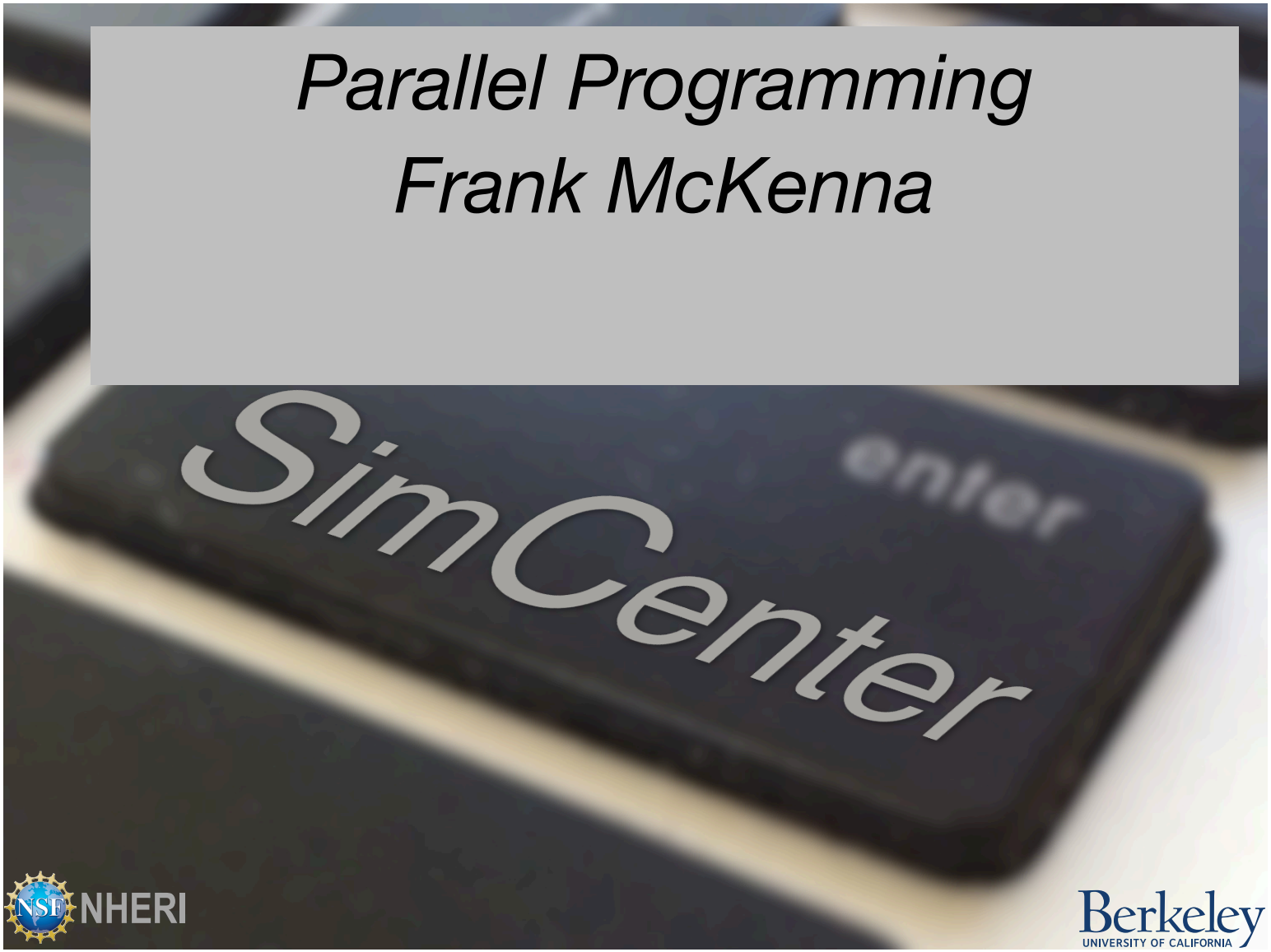
Agenda – Day 2

Time	Title	Presenter
9:00-9:10	Welcome	Frank McKenna
9:10-10:30	Debugging	Wael Elhaddad
10:30-11:00	PI / Exercises #3!	
11:00-11:15	Parallel Machines	Frank McKenna
11.15-12:00	Parallel Programming With MPI	Frank McKenna
12:00-1:00	LUNCH	
1:00-2:30	EXERCISE	YOU
2:30-3:00	Parallel Programming with OpenMP	Frank McKenna
3:00-4:30	EXERCISE	YOU
4:30-5:00	Load Balancing	Frank McKenna
Day 3	Abstraction, More C & Introducing C++	
Day 4	User Interface Design & Qt	
Day 5	SimCenter & Cloud Computing	

NSF NHERI

SimCenter NHERI
Center for Computational Modeling and Simulation


Parallel Programming
Frank McKenna
SimCenter
enter
NSF
NHERI
Berkeley
UNIVERSITY OF CALIFORNIA

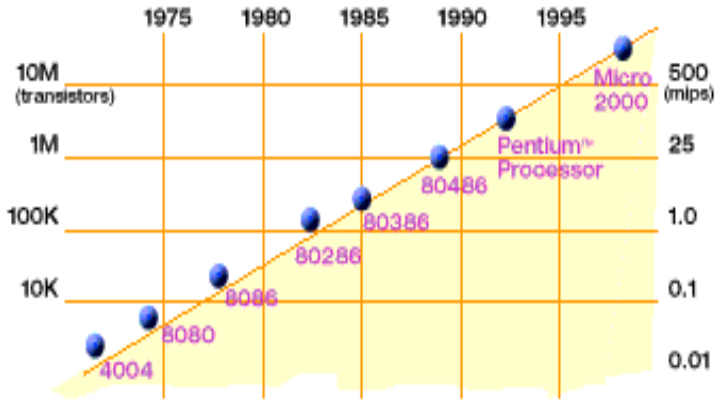
Outline

- Parallel Machines & Parallel Machine Models
- Parallel Programming
  - Message Passing With MPI
  - Shared Memory Programming with OpenMP
- Dynamic Load Balancing

Ignoring co-processors and GPUs

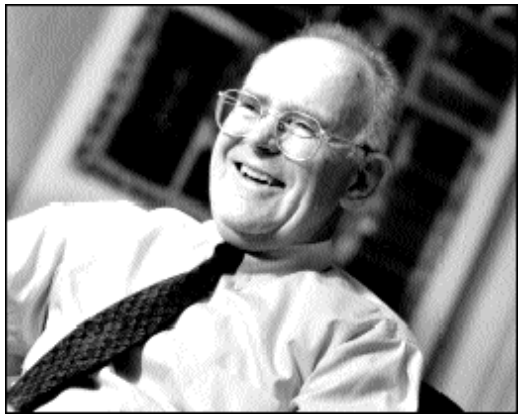
many slides source: CS267, Jim Demmell

Why is Parallel Programming Important



2X transistors/Chip Every 1.5 years
Called "Moore's Law"

Microprocessors have become smaller, denser, and more powerful.

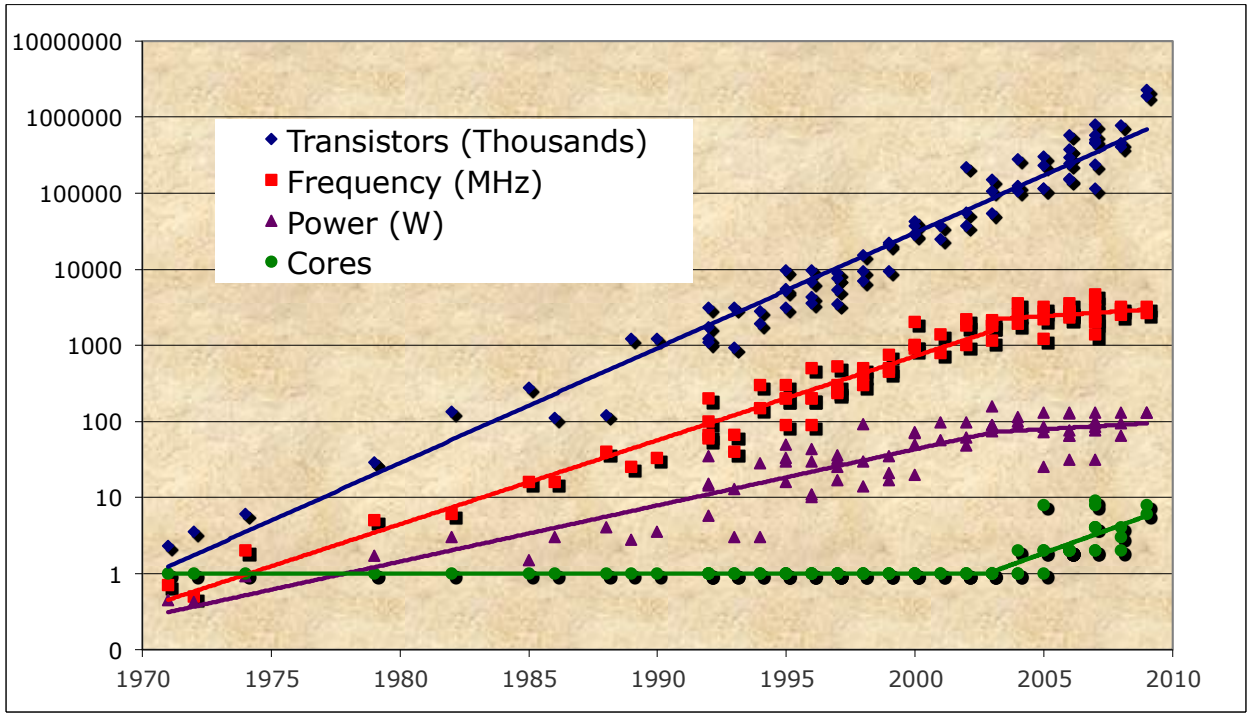


Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

Slide source: Jack Dongarra

source: CS267, Jim Demmell

Revolution in Processors



- Chip density is continuing increase ~2x every 2 years
- Clock speed is not
- Number of processor cores may double instead
- Power is under control, no longer growing

source: CS267, Jim Demmell

Multiple Cores!

What does it mean for Programmers
"The Free Lunch is Over" Herb Sutter

- Up until 2003 programmers had been relying on Hardware to make their programs go faster. No longer. They had to start programming again!
- **Performance now comes from Software**
- **To be fast and utilize the resources, Software must run in parallel, that is it must run on multiple cores at same time.**

How many cores?

1,736 Intel Xeon Skylake nodes, each with 48 cores + 192GB of RAM
4,200 Intel Knights Landing nodes, each with 68 cores + 96GB of DDR RAM

TACC DESIGNSAFE-CI

1 Intel i7 node, 6 cores + 16GB RAM

Apple A11, 6 cores + 64GB RAM

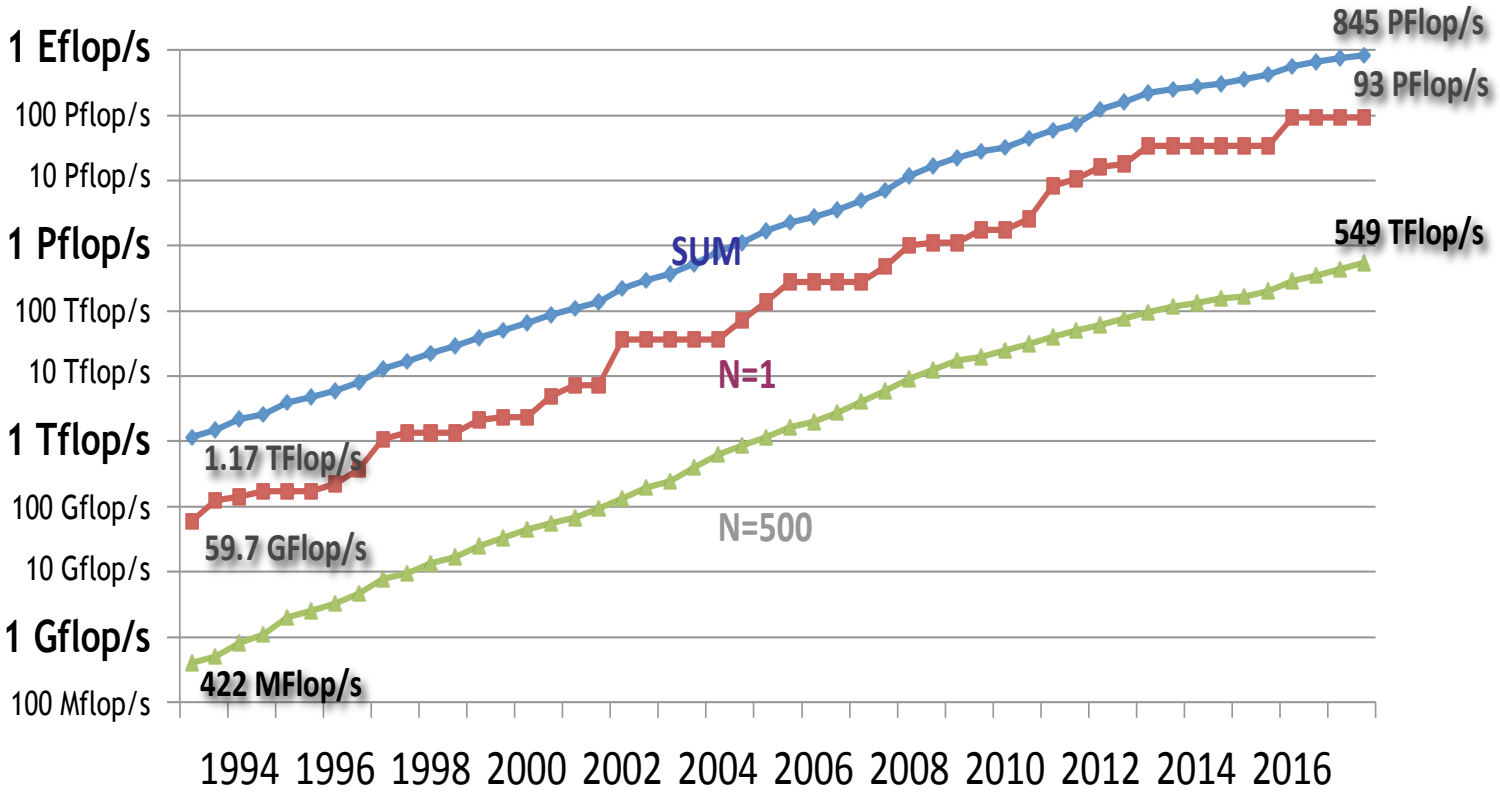
How Big & Fast!

Rmax of Linpack
Solve Ax = b

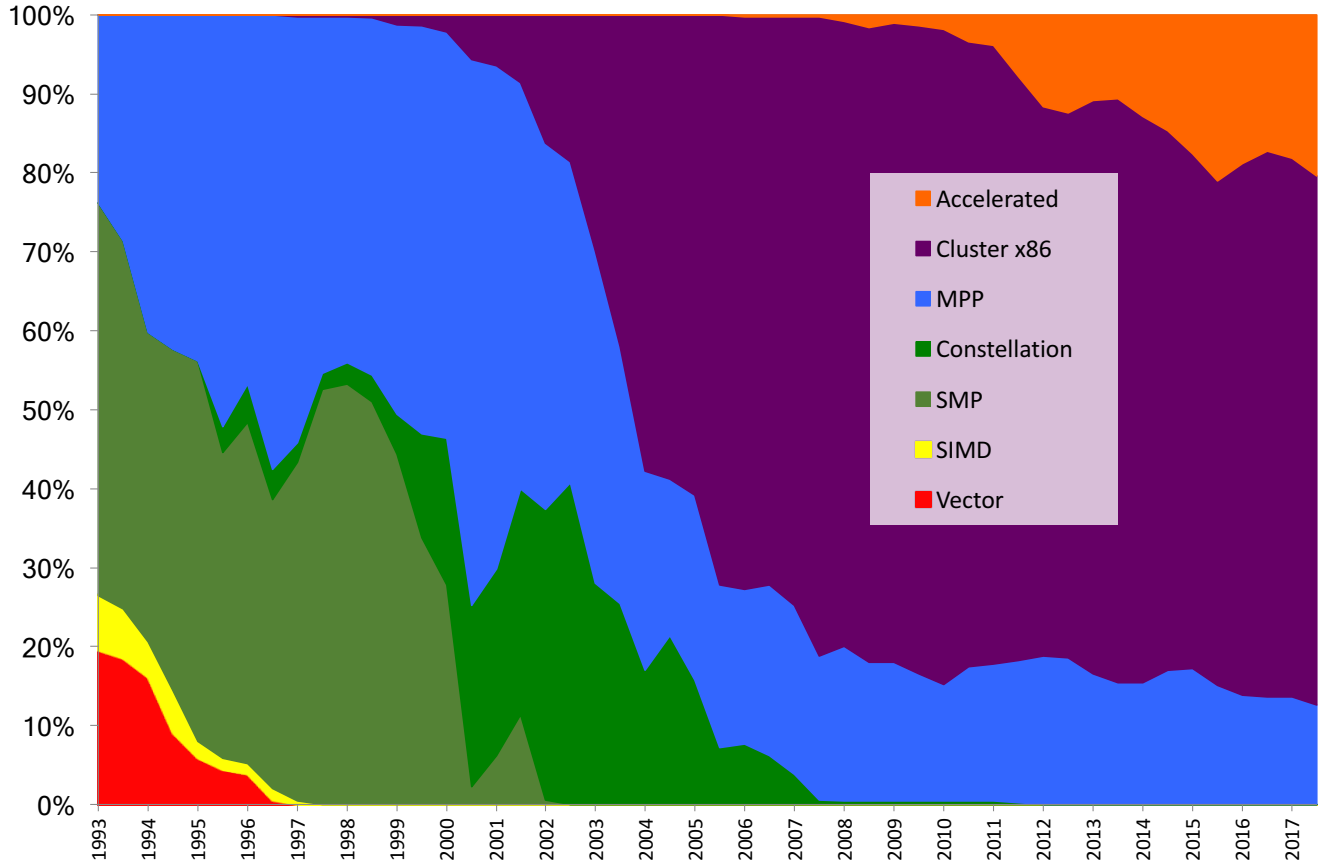
Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	2,282,544	122,300.0	187,659.3	8,806
2	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
3	DOE/NNSA/LLNL United States	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	1,572,480	71,610.0	119,193.6	
4	National Super Computer Center in Guangzhou China	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 NUDT	4,981,760	61,444.5	100,678.7	18,482
5	National Institute of Advanced Industrial Science and Technology (AIST) Japan	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2550 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR Fujitsu	391,680	19,880.0	32,576.6	1,649

15	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede2** - PowerEdge C6320P/C6420, Intel Xeon Phi 7250 68C 1.4GHz/Platinum 8160, Intel Omni-Path Dell EMC	367,024	10,680.7	18,309.2

Performance Development (2018)



From Vector Supercomputers to Massively Parallel Accelerator Systems



Moore's Law reinterpreted

- Number of cores per chip can double every two years
- Clock speed will not increase (possibly decrease)
- Need to deal with systems with millions of concurrent threads
- Need to deal with inter-chip parallelism as well as intra-chip parallelism

source: CS267, Jim Demmell

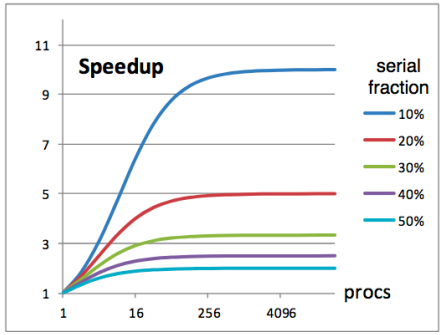
Can All Programs Be Made to Run Faster?

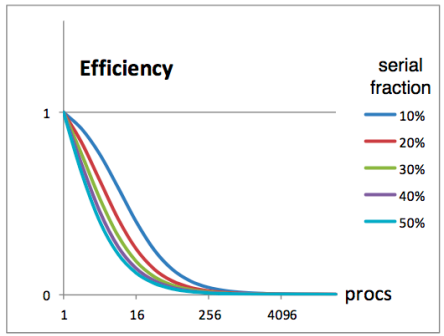
- Suppose only part of an application can run in parallel
- Amdahl's law
  - let s be the fraction of work done sequentially, so (1-s) is fraction parallelizable
  - P = number of processors

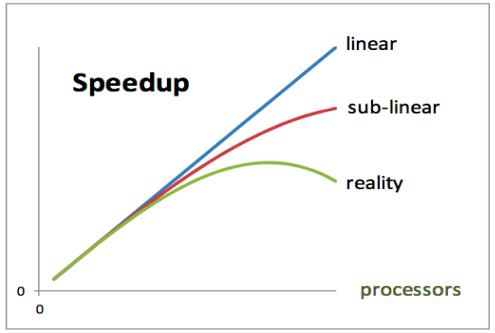
$$\text{Speedup}(P) = \text{Time}(1)/\text{Time}(P)$$
$$<= 1/(s + (1-s)/P)$$
$$<= 1/s$$

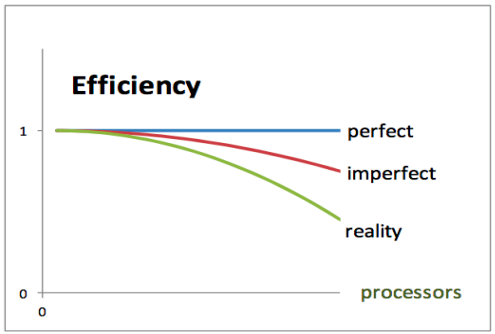
QUIZ: if 10% of program is sequential, What is the maximum speedup I Can obtain?

- **Even if the parallel part speeds up perfectly performance is limited by the sequential part**
- Top500 list: currently fastest machine has P~2.2M; Stampede2 has 367,000









Source: Doug James, TACC

This Does not Take into Account Overhead of Parallelism

- Parallelism overheads include:
  - cost of starting a thread or process
  - cost of communicating shared data
  - cost of synchronizing
  - extra (redundant) computation
- Each of these can be in the range of milliseconds (=millions of flops) on some systems
- Tradeoff: Algorithm needs sufficiently large units of work to run fast in parallel (i.e. large granularity), but not so large that there is not enough parallel work

source: CS267, Jim Demmell

Load Imbalance

- Load imbalance is the time that some processors in the system are idle due to
  - insufficient parallelism (during that phase)
  - unequal size tasks
- Examples of the latter
  - adapting to “interesting parts of a domain”
  - tree-structured computations
  - fundamentally unstructured problems
- Algorithm needs to balance load
  - Sometimes can determine work load, divide up evenly, before starting
    - “Static Load Balancing”
  - Sometimes work load changes dynamically, need to rebalance dynamically
    - “Dynamic Load Balancing,” eg work-stealing

Improving Real Performance

Peak Performance grows exponentially, a la Moore's Law

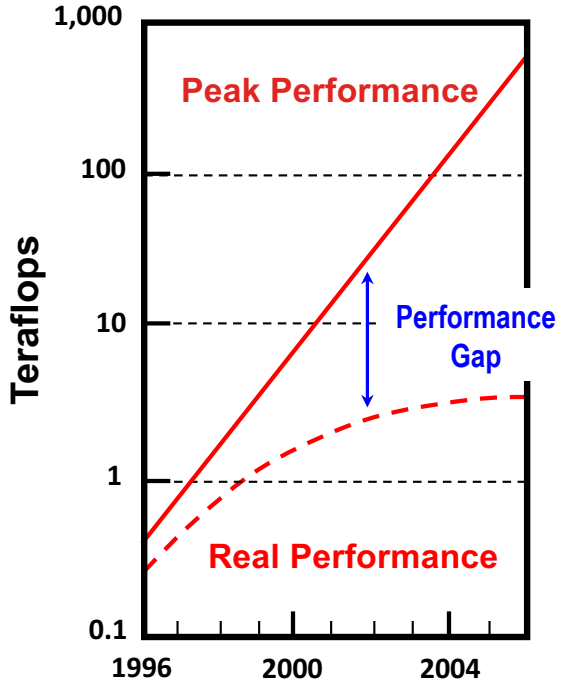
- In 1990's, peak performance increased 100x; in 2000's, it will increase 1000x

But efficiency (the performance relative to the hardware peak) has declined

- was 40-50% on the vector supercomputers of 1990s
- now as little as 5-10% on parallel supercomputers of today

Close the gap through ...

- Mathematical methods and algorithms that achieve high performance on a single processor and scale to thousands of processors
- More efficient programming models and tools for massively parallel supercomputers

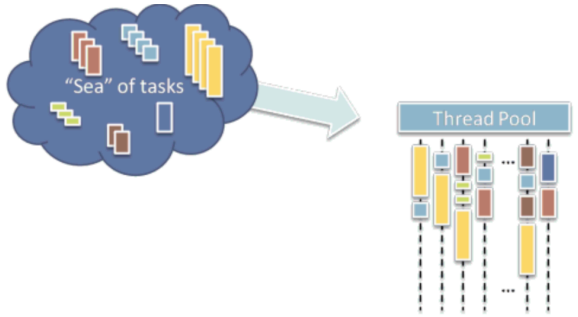


Art of Programming - II

- To take a problem, and continually break it down into a series of smaller ideally concurrent tasks until ultimately these tasks become a series of small specific individual instructions.
- Mindful of the architecture on which the program will run, identify those tasks which can be run concurrently and map those tasks onto the processing units of the target architecture.

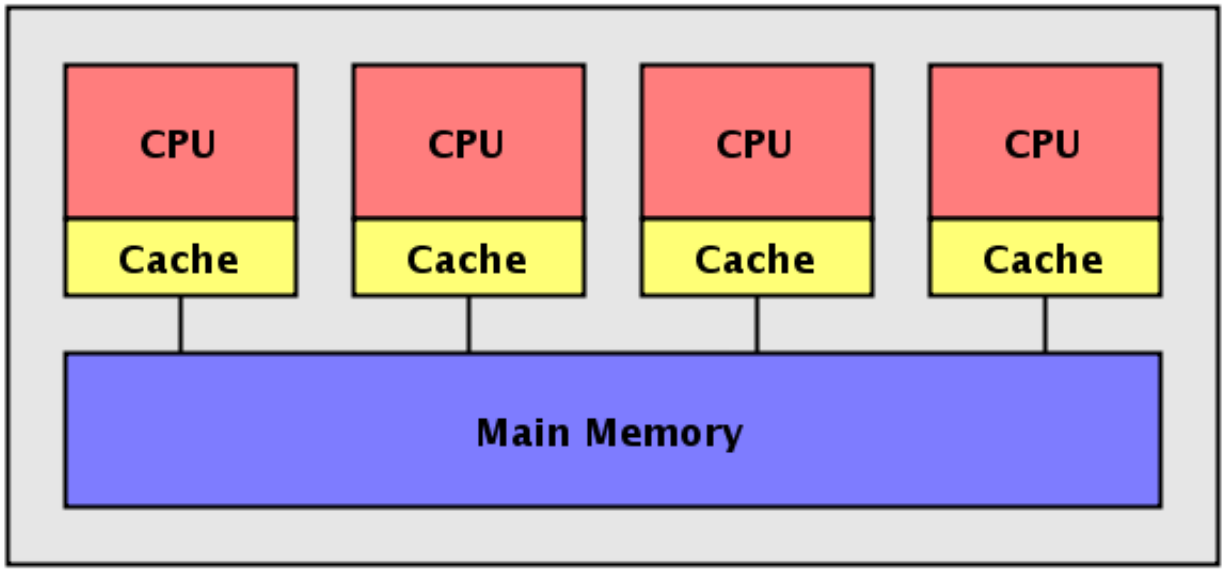
Considerations for Parallel Programming:

- Finding enough parallelism (Amdahl's Law)
- Granularity – how big should each parallel task be
- Locality – moving data costs more than arithmetic
- Load balance – don't want 1K processors to wait for one slow one
- Coordination and synchronization – sharing data safely
- Performance modeling/debugging/tuning
- Where to put the task,



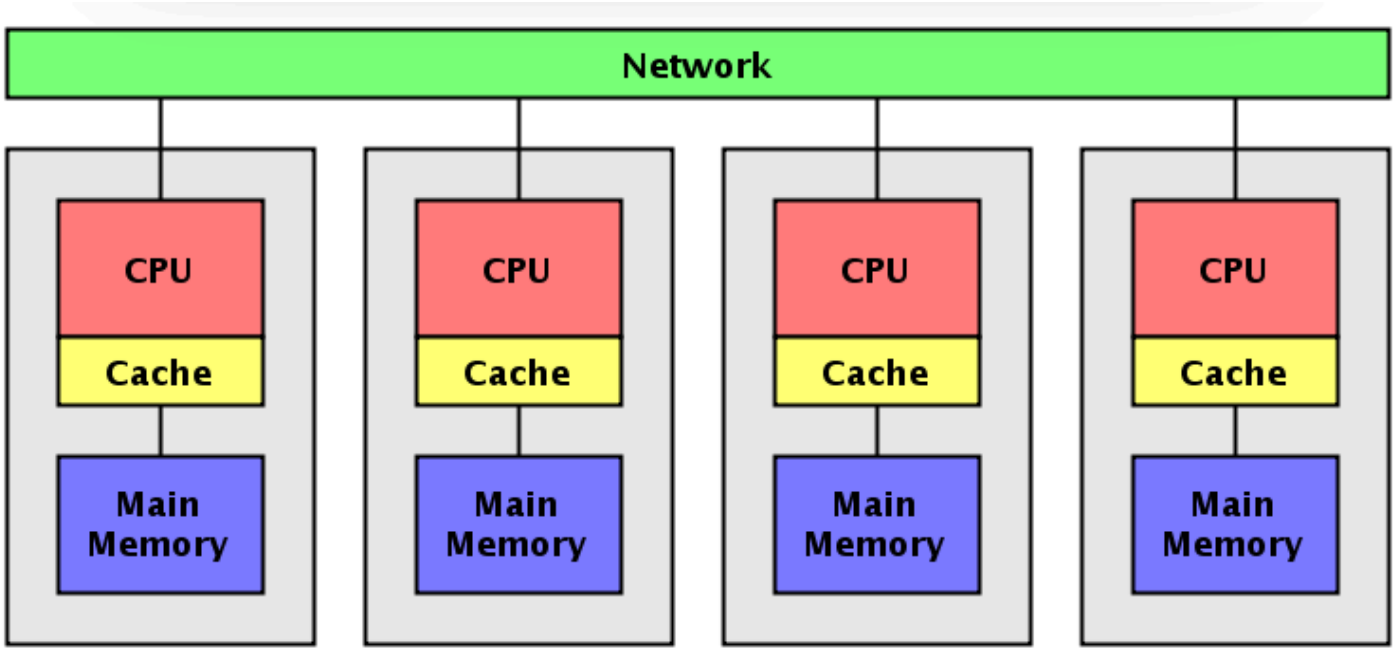
All of these things makes parallel programming even harder than sequential programming.

Simplified Parallel Machine Models



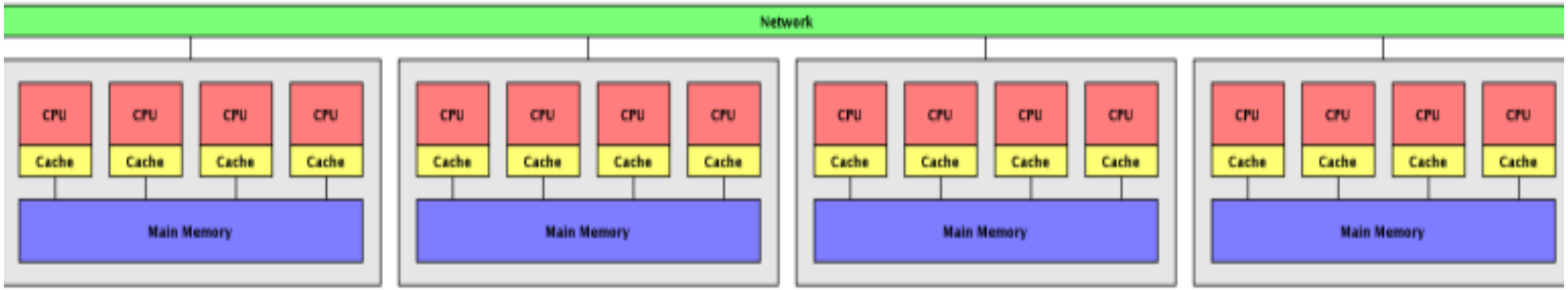
Shared Memory Model

Simplified Parallel Machine Models



Distributed Memory Model

Simplified Parallel Machine Models



Hybrid Model

Writing Programs to Run on Parallel Machines

- C Programming Libraries Exist that provides the programmer an API for writing programs that will run in parallel.
- They provide a **Programming Model** that is portable across architectures, i.e. can provide a message passing model that runs on a shared memory machine.
- We will look at 2 of these Programming Models and Libraries that support the model:
  - Message Passing Programming using MPI (message passing interface)
  - Thread Programming using OpenMP
- As will all libraries they can incur an overhead.

Outline

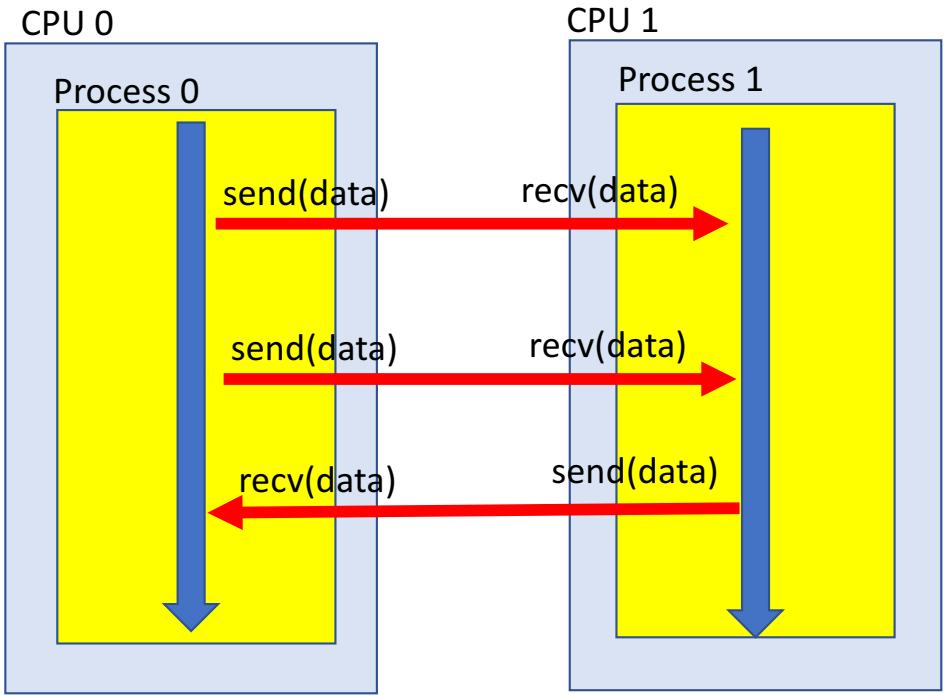
- Parallel Machines & Parallel Machine Models
- Parallel Programming
  - Message Passing With MPI
  - Shared Memory Programming with OpenMP
- Dynamic Load Balancing

Ignoring co-processors and GPUs

many slides source: CS267, Jim Demmell

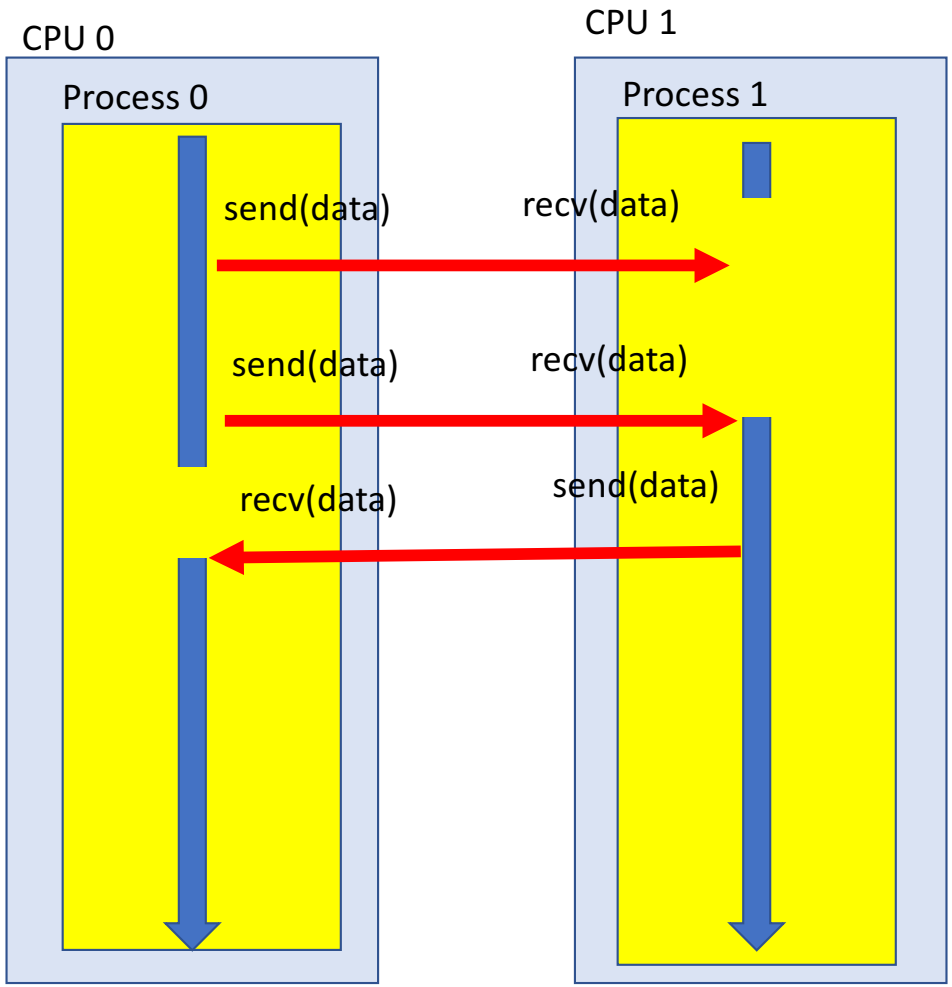
Message Passing Model

- Processes run independently in their own memory space and processes communicate with each other when data needs to be shared

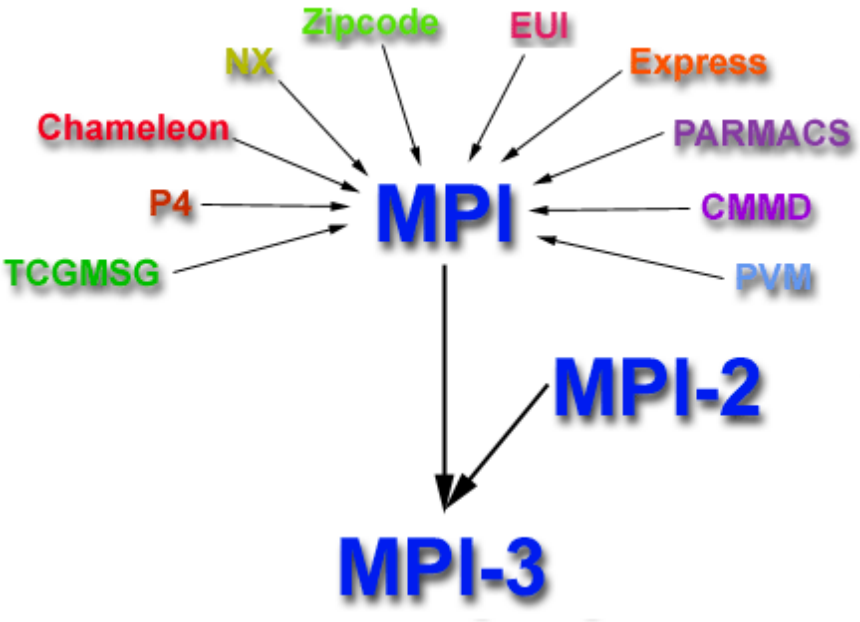


- Basically you write sequential applications with additional function calls to send and recv data.

Only get Speedup if processes can be kept busy



Programming Libraries



- Coalasced around a single standard MPI
- Allows for portable code

MPI

Provides a number of functions:

1. Enquiries
   - How many processes?
   - Which one am I?
   - Any messages Waiting?
2. Communication
   - Pair-wise point to point send and receive
   - Collective/Group: Broadcast, Scatter/Gather
   - Compute and Move: sum, product, max …
3. Synchronization
   - Barrier

REMEMBER:

What I am about to show is just C code with some functions you have not yet seen.

Hello World

Code/Parallel/mpi/hello1.c

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

int main( int argc, char **argv)
{
    int procID, numP;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numP );
    MPI_Comm_rank( MPI_COMM_WORLD, &procID );

    printf( "Hello World, I am %d of %d\n", procID, numP );
    MPI_Finalize();
    return 0;
}
```

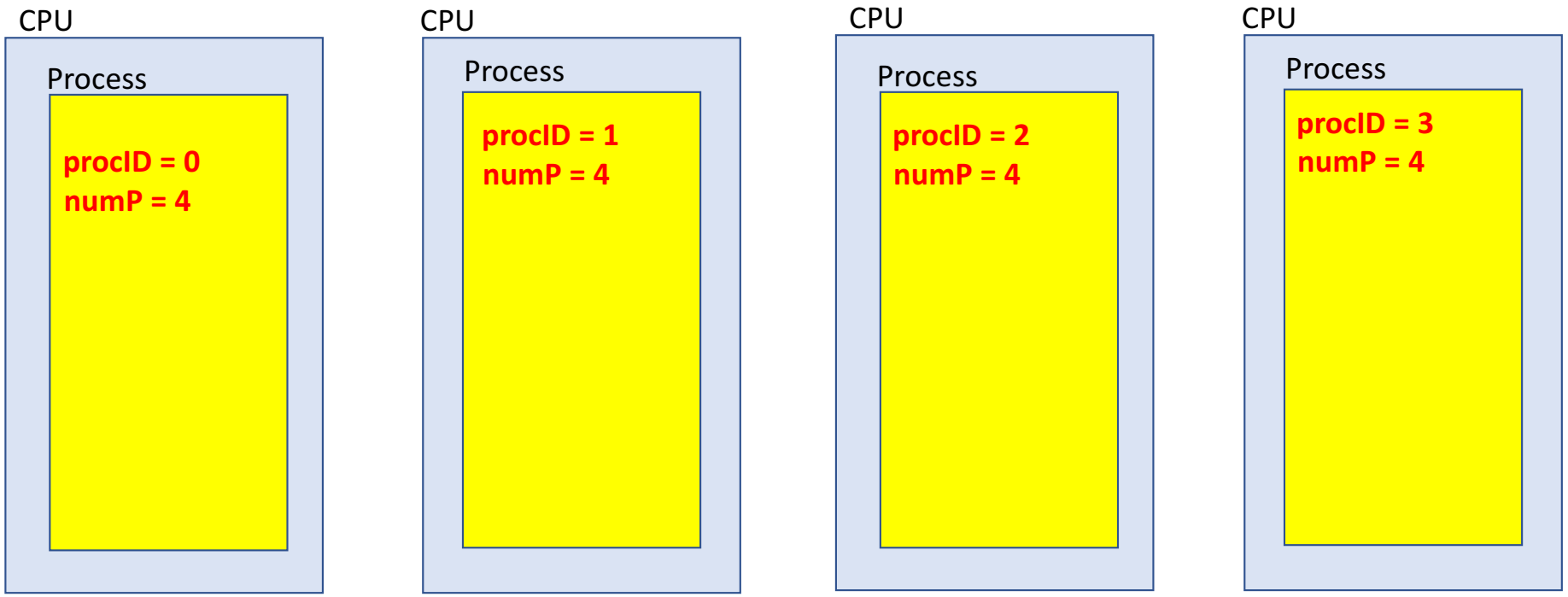
MPI functions (and MPI_COMM_WORLD) are defined in mpi.h

MPI_Init() and MPI_finalize() must be first and last functions called

MPI_COMM_WORLD is a default group containing all processes

MPI_Comm_size returns # of processes in the group

MPI_Comm_rank returns processes unique ID the group, 0 through (numP-1)


CPU
Process
procID = 0
numP = 4
CPU
Process
procID = 1
numP = 4
CPU
Process
procID = 2
numP = 4
CPU
Process
procID = 3
numP = 4

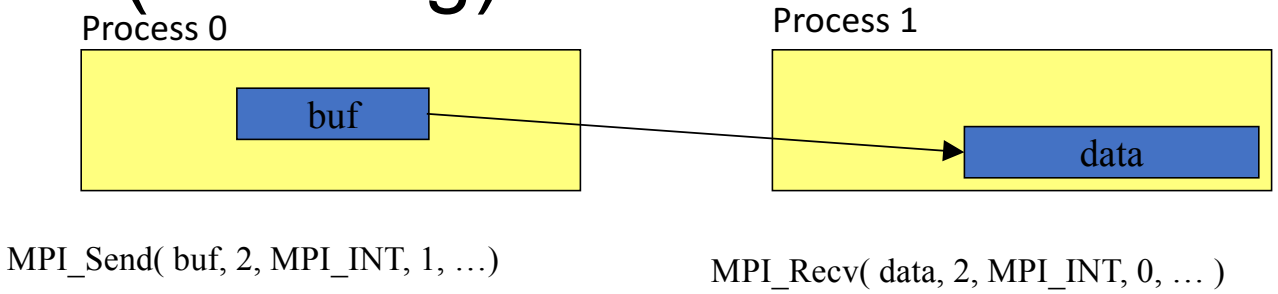
Send/Recv blocking

Code/Parallel/mpi/send1.c

```
#include <mpi.h>
#include <stdio.h>
int main( int argc, char **argv) {
  int procID;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank( MPI_COMM_WORLD, &procID );

  if (procID == 0) { // process 0 sends
    int buf[2] = {12345, 67890};
    MPI_Send( &buf, 2, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (procID == 1) { // process 1 receives

    int data[2];
    MPI_Recv( &data, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    printf( "Received %d %d\n", data[0], data[1] );
  }

  MPI_Finalize();
  return 0;
}
```

NOTE the PAIR of Send/Recv

MPI Basic (Blocking) Send

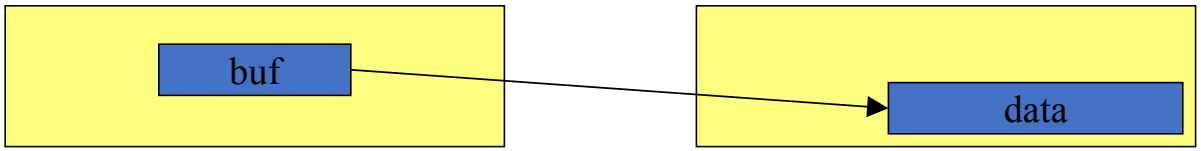


`MPI_SEND(start, count, datatype, dest, tag, comm)`

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**. The message has an identifier **tag.**
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI_CHAR	char
MPI_WCHAR	wchar_t - wide character
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT **MPI_LONG_LONG**	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_C_COMPLEX **MPI_C_FLOAT_COMPLEX**	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_C_BOOL	_Bool
MPI_INT8_T **MPI_INT16_T** **MPI_INT32_T** **MPI_INT64_T**	int8_t int16_t int32_t int64_t
MPI_UINT8_T **MPI_UINT16_T** **MPI_UINT32_T** **MPI_UINT64_T**	uint8_t uint16_t uint32_t uint64_t
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

MPI Basic (Blocking) Receive



MPI_Send(buf, 2, MPI_INT, 1, …)

MPI_Recv(data, 2, MPI_INT, 0, …)

`MPI_RECV(start, count, datatype, source, tag, comm, status)`

- Waits until a matching (both **`source`** and **`tag`**) message is received from the system, and the buffer can be used
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**
- **`tag`** is a tag to be matched or **`MPI_ANY_TAG`**
- receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error
- **`status`** contains further information (e.g. size of message)

Retrieving Further Information

- **`Status`** is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

Not Quite So Simple as ensuring PAIRS of send/recv

Code/Parallel/mpi/send2.c

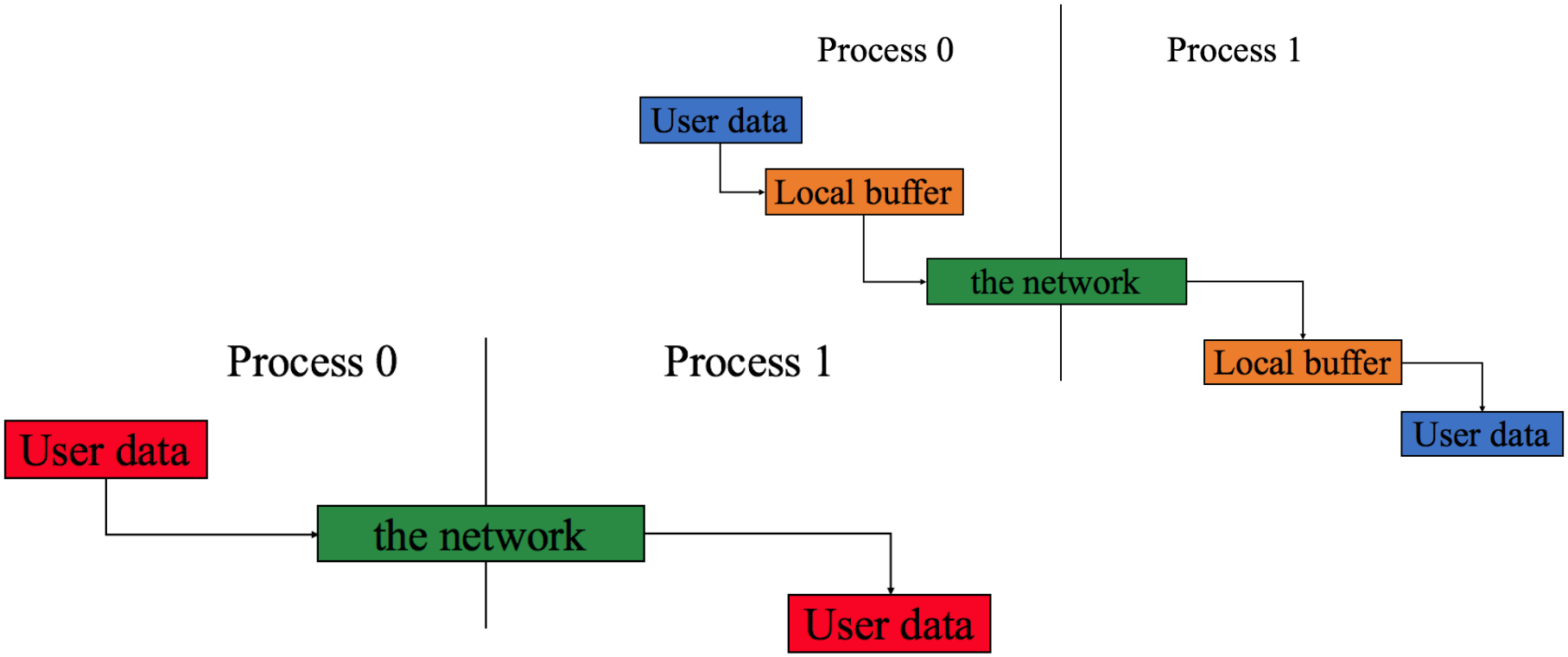
```
#include <mpi.h>
#include <stdio.h>
#define DATA_SIZE 1000
int main(int argc, char **argv) {
   int procID, numP;
   MPI_Status status;
   int buf[DATA_SIZE];

   MPI_Init( &argc, &argv );
   MPI_Comm_size( MPI_COMM_WORLD, &numP );
   MPI_Comm_rank( MPI_COMM_WORLD, &procID );

   If (procID == 0) {
      for (int i=0; i<DATA_SIZE; i++) buf[i]=1+i;
      MPI_Send(&buf, DATA_SIZE, MPI_INT, 1, 0, MPI_COMM_WORLD);
      MPI_Recv(&buf, DATA_SIZE, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
      printf("%d Received %d %d\n", procID, buf[0], buf[DATA_SIZE-1]);
   } else if (procID == 1) {
      for (int i=0; i<DATA_SIZE; i++) buf[i]=DATA_SIZE-i;
      MPI_Send(&buf, DATA_SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD);
      MPI_Recv(&buf, DATA_SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
     printf("%d Received %d %d\n", procID, buf[0], buf[DATA_SIZE-1]);
   }
   MPI_Finalize();
   return 0;
}
```

```
mpi >mpicc send2.c; mpirun -n 2 ./a.out
Buffer Size: 1000
0 Received 1000 1
1 Received 1 1000
```

```
mpi >mpicc send2.c; mpirun -n 2 ./a.out
Buffer Size: 10000
^Cmpi >
mpi >
```

DEADLOCK .. PROGRAM HANGS .. WHY?

Why Deadlock? .. Where Does the Data Go



If large message & insufficient data, the send() must wait for buffer to clear through a recv()

source: CS267, Jim Demmell

Current Problem:

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

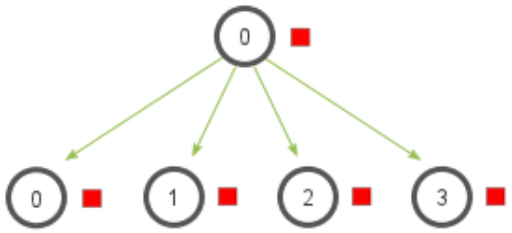
Could revise order
this requires some code rewrite:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

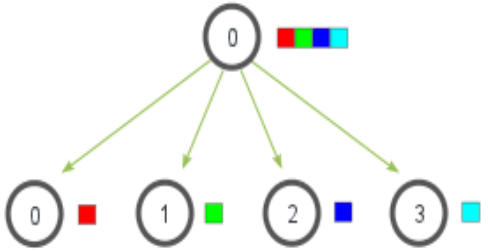
Alternatives use non-blocking sends.

Some Collective Functions

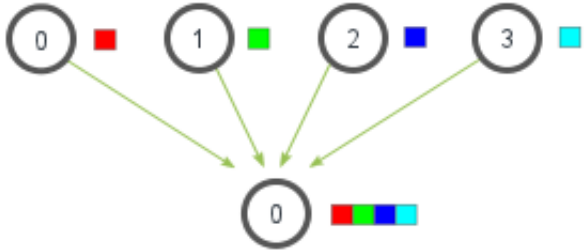
MPI_Bcast



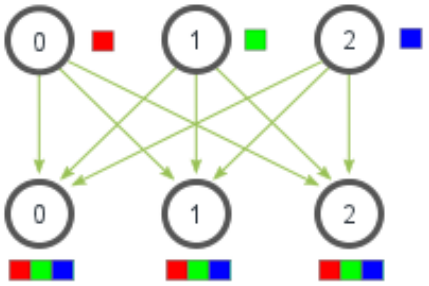
MPI_Scatter



MPI_Gather



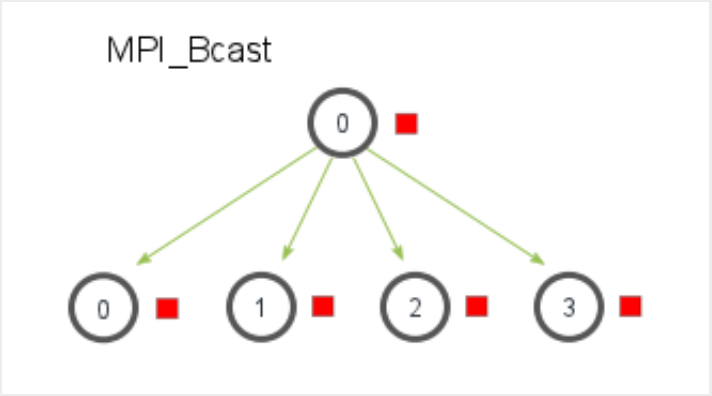
MPI_Allgather



Broadcast

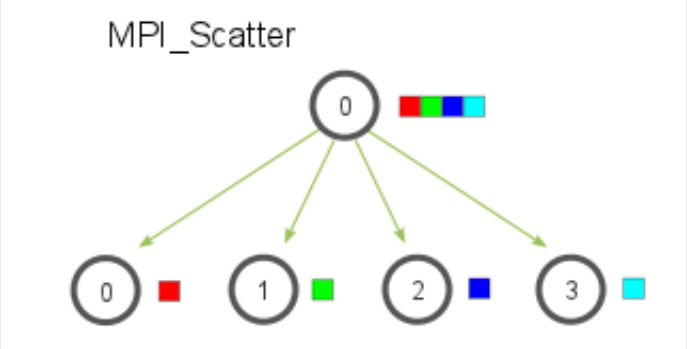
Code/Parallel/mpi/bcast1.c

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

int main( int argc, char **argv) {
   int procID; buf[2];

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &procID );

    if (procID == 0) {
       buf[0] = 12345;
       buf[1] = 67890;
    }

    MPI_Bcast(&buf, 2, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Process %d data %d %d\n", procID, buf[0], buf[1]);

    MPI_Finalize();
    return 0;
}
```



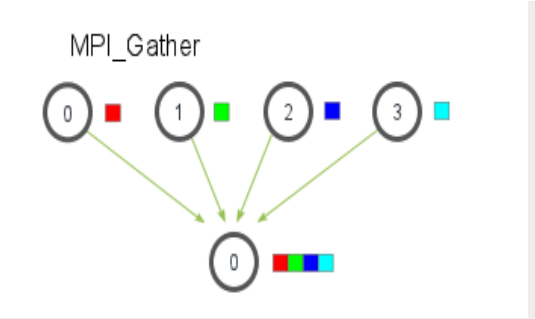
Scatter

Code/Parallel/mpi/scatter1.c




```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define LUMP 5
int main(int argc, char **argv) {
   int numP, procID;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numP);
   MPI_Comm_rank(MPI_COMM_WORLD, &procID);

  int *globalData=NULL;
  int localData[LUMP];

  if (procID == 0) { // malloc and fill in with data
    globalData = malloc(LUMP * numP * sizeof(int) );
    for (int i=0; i<LUMP*numP; i++)
      globalData[i] = i;
  }
  MPI_Scatter(globalData, LUMP, MPI_INT, &localData, LUMP, MPI_INT, 0, MPI_COMM_WORLD);
  printf("Processor %d has first: %d last %d\n", procID, localData[0], localData[LUMP-1]);

  if (procID == 0)  free(globalData);

  MPI_Finalize();
}
```

```
#include "mpi.h"
#include <stdio.h>
#define LUMP 5
int main(int argc, const charr **argv) {
     int procID, numP, ierr;
     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &numP);
     MPI_Comm_rank(MPI_COMM_WORLD, &procID);

      int *globalData=NULL;
      int localData[LUMP];
      if (procID == 0) { // malloc global data array only on P0
         globalData = malloc(LUMP * numP * sizeof(int) );
      }
      for (int i=0; i<LUMP; i++)
           localData[i] = procID*10+i;

     MPI_Gather(localData, LUMP, MPI_INT, globalData, LUMP, MPI_INT, 0, MPI_COMM_WORLD);

     if (procID == 0) {
          for (int i=0; i<numP*LUMP; i++)
                printf("%d ", globalData[i]);
          printf("\n");
     }
     if (procID == 0) free(globalData);
     MPI_Finalize();
```

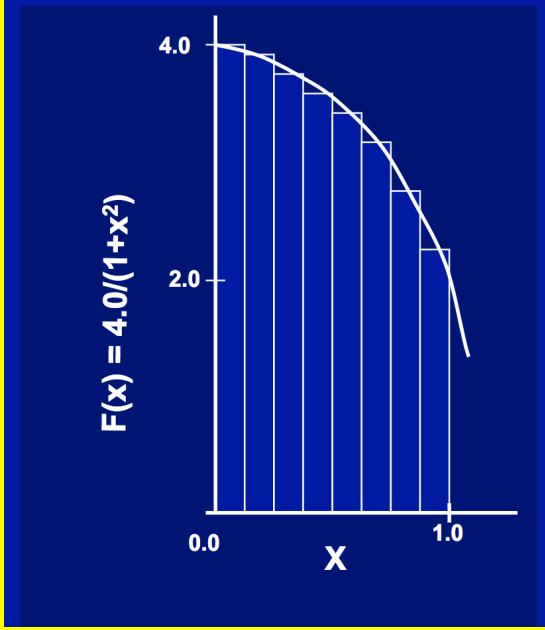


MPI can be simple

- Claim: most MPI applications can be written with only 6 functions (although which 6 may differ)
- Using point-to-point:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_SEND`
  - `MPI_RECEIVE`
- Using collectives:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_BCAST/MPI_SCATTER`
  - `MPI_GATHER/MPI_ALLGATHER`
- You may use more for convenience or performance

Exercise: Parallelize Compute PI using MPI

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

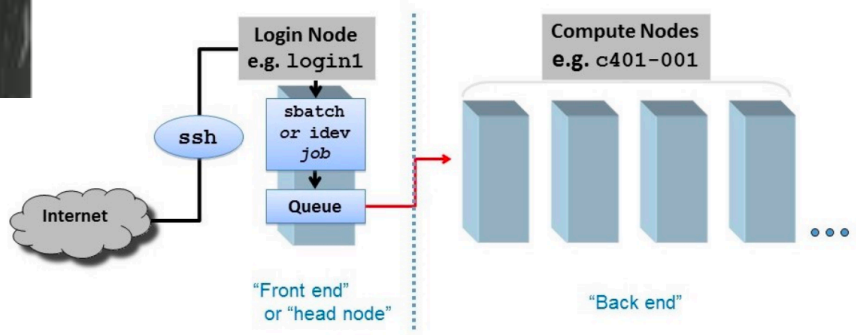
Source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere

```
#include <stdio>
static int long numSteps = 100000;
int main() {
  double pi = 0; double time=0;
  // your code
  for (int i=0; i<numSteps; i++) {
    // your code
  }
   // your code
  printf("PI = %f, duration: %f ms\n",pi, time);
  return 0;
}
```

Stampede2

TACC

DESIGNSAFE-CI



sbatch script

```
#!/bin/bash
#----------------------------------------------------------------
# Generic SLURM script – MPI Hello World
#
# This script requests 1 node and 8 cores/node (out of total 64 avail)
# for a total of 1*8 = 8 MPI tasks.
#----------------------------------------------------------------
#SBATCH -J myjob          # Job name
#SBATCH -o myjob.%j.out   # stdout; %j expands to jobid
#SBATCH -e myjob.%j.err   # stderr; skip to combine stdout and stderr
#SBATCH -p development    # queue
#SBATCH -N 1              # Number of nodes, not cores (64 cores/node)
#SBATCH -n 8              # Total number of MPI tasks (if omitted, n=N)
#SBATCH -t 00:00:10       # max time

module petsc    # load any needed modules, these just examples
module load list

ibrun ./a.out
```

1. Put your pi.c into your github repository and push it to your fork

```
cp pi.c ~/SimCenterBootcamp/Code/Parallel/mpi/pi.c
cd  ~/SimCenterBootcamp/Code/Parallel/mpi
git add pi.c
git commit –m “pi.c initial import”
git push
```

2. Login to stampede2

```
ssh yourlogin@stampede2.tacc.utexas.edu
```

3. Now on stampede2 login node, clone the repository on stampede2

```
git clone http://???????
```

4. cd to the ~/SimCenterBootCamp/Code/Parallel/mpi directory

```
cd  ~/SimCenterBootcamp/Code/Parallel/mpi
```

5. Compile hello1.c or whatever you want

```
mpicc hello1.c
```

6. Submit the job to SLURM queue

```
sbatch submit.sh
```

7. Look at the output

```
cat myjob.*.out
```

8. Make changes to your code, compile & run!

Your on your own

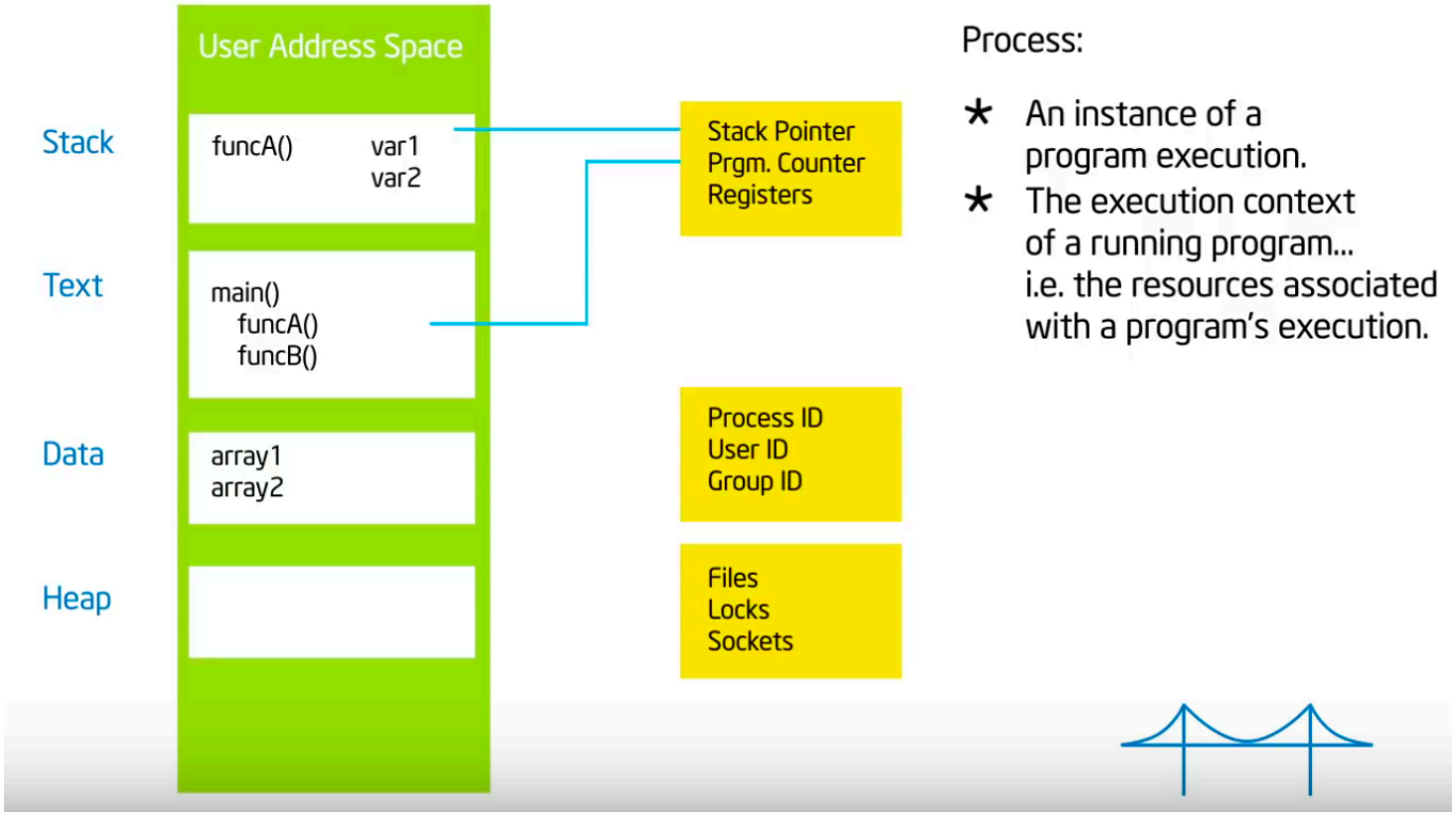
Outline

- Parallel Machines & Parallel Machine Models
- Parallel Programming
  - Message Passing With MPI
  - Shared Memory Programming with OpenMP
- Dynamic Load Balancing

Ignoring co-processors and GPUs

many slides source: CS267, Jim Demmell

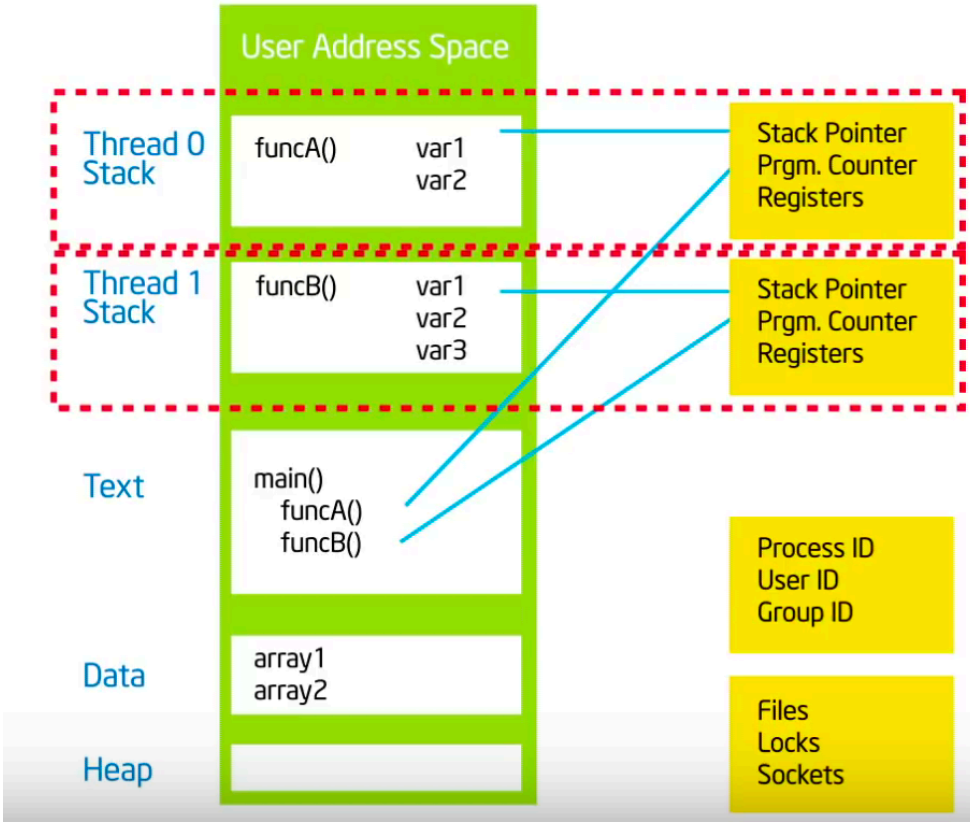
Process



Process:

* An instance of a program execution.
* The execution context of a running program... i.e. the resources associated with a program's execution.

Threads

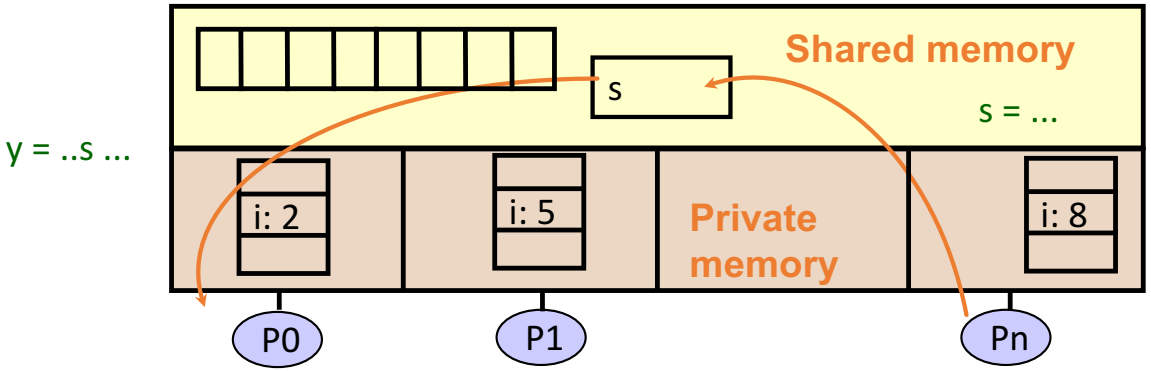


Threads:

* Threads are "light weight processes"
* Threads share Process state among multiple threads. This greatly reduces the cost of switching context.

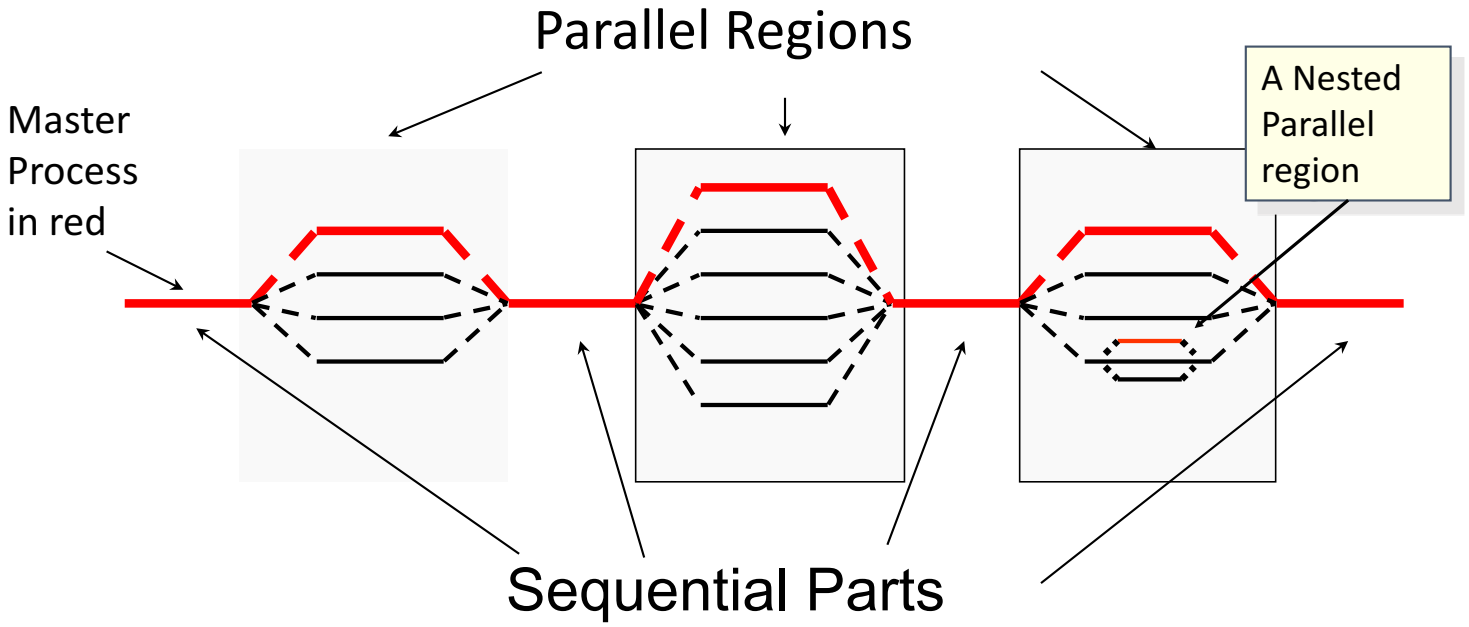
Threads

- Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables



Programming Model for Threads

- Master Process spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



Runtime Library Options for Shared Memory

POSIX Threads (pthreads)
OpenMP

- OpenMP provides multi-threaded capabilities to C, C++ and Fortran Programs
- In a threaded environment, threads communicate by sharing data
- Unintended sharing of data causes **race conditions**
- **Race Condition**: program output is different every time you run the program, a consequence of the threads being scheduled differently
- OpenMP provides constructs to control what blocks of code are run in parallel and also constructs for providing access to shared data using synchronization
- Synchronization has overhead consequences, you have to minimize them to get good speedup.

- Mostly Set of Compiler directives (#pragma) applying to structured block

```
#pragma omp parallel
```

- Some runtime library calls

```
omp_num_threads(4);
```

- Being compiler directives, they are built into most compilers .
- Just have to activate it when compiling

```
gcc hello.c -fopenmp
```

```
icc hello.c /Qopenmp
```

Hello World

Code/Parallel/openmp/hello1.c

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

int main( int argc, char *argv[] )
{
  #pragma omp parallel
  {
     int id = omp_get_thread_num();
     int numP = omp_get_num_threads();
     printf("Hello World, I am  %d of %d\n",id,numP);
  }
  return 0;
}
```

```
openmp >gcc-7.2 hello1.c -fopenmp; ./a.out
Hello World, I am  0 of 4
Hello World, I am  3 of 4
Hello World, I am  1 of 4
Hello World, I am  2 of 4
openmp >
```

```
openmp >export env OMP_NUM_THREADS=2
openmp >./a.out
Hello World, I am  0 of 2
Hello World, I am  1 of 2
openmp >
```

Hello World – changing num threads

Code/Parallel/openmp/hello2.c

```
#include <openmp.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
  omp_set_num_threads(2);
  #pragma omp parallel
   {
        int id = omp_get_thread_num();
        int numP = omp_get_num_threads();
        printf("Hello World, I am  %d of
   }
  return 0;
}
```

Runtime function to request a certain number of threads

Runtime function to return actual number of threads in the team

Code/Parallel/openmp/hello3.c

```
#include <openmp.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
#pragma omp parallel num_threads(2)
   {
        int id = omp_get_thread_num();
        int numP = omp_get_num_threads();
        printf("Hello World, I am  %d of %d\n",id,numP);
   }
  return 0;
}
```

Different # threads in different blocks

Code/Parallel/openmp/hello4.c

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

int main(int argc, const char **argv) {

#pragma omp parallel num_threads(2)
  {
    int id = omp_get_thread_num();
    int numP = omp_get_num_threads();
    printf("Hello World, I am  %d of %d\n",id,numP);
  }
#pragma omp parallel num_threads(4)
  {
    int id = omp_get_thread_num();
    int numP = omp_get_num_threads();
    printf("Hello World Again, I am  %d of %d\n",id,numP);
  }
  return(0);
}
```

```
openmp >gcc-7.2 hello4.c -fopenmp; ./a.out
Hello World, I am  0 of 2
Hello World, I am  1 of 2
Hello World Again, I am  1 of 4
Hello World Again, I am  2 of 4
Hello World Again, I am  3 of 4
Hello World Again, I am  0 of 4
openmp >
```

Hello World – shared variables & RACE CONDITIONS

Code/Parallel/openmp/hello5.c

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

int main(int argc, const char **argv) {

 int id, numP;

#pragma omp parallel num_threads(4)
  {
    id = omp_get_thread_num();
    numP = omp_get_num_threads();
    printf("Hello World from %d of %d th
  }

  return(0);
}
```

```
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 1 of 4 threads
Hello World from 2 of 4 threads
Hello World from 3 of 4 threads
Hello World from 0 of 4 threads
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 0 of 4 threads
Hello World from 3 of 4 threads
Hello World from 2 of 4 threads
Hello World from 1 of 4 threads
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 0 of 4 threads
Hello World from 0 of 4 threads
Hello World from 2 of 4 threads
Hello World from 3 of 4 threads
```

What We Want Threads To DO

- Work Independently With Controlled Access at times to Shared Data
  - Parallel Tasks Constructs
    - omp parallel
    - Opr for
  - Shared Data

Simple Vector Sum

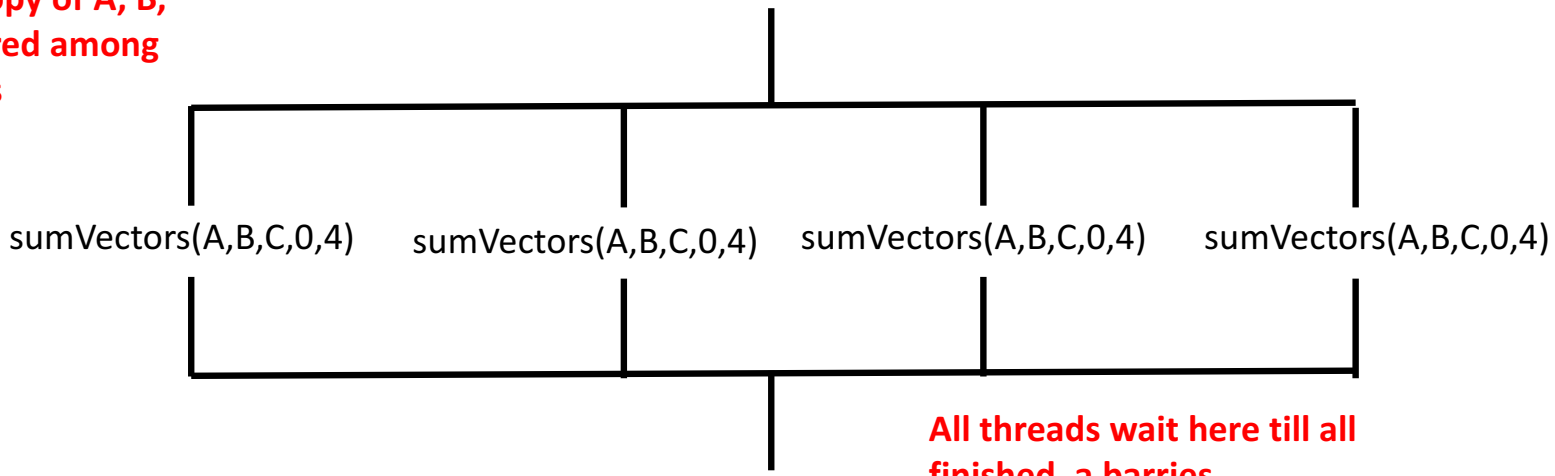
Code/Parallel/openmp/sum1.c

```
#include <omp.h>
#include <stdio.h>
#define DATA_SIZE 10000

void sumVectors(int N, double *A, double *B, double *C, int tid, int numT);
I
nt main(int argc, const char **argv) {
  double a[DATA_SIZE], b[DATA_SIZE], c[DATA_SIZE];
  int num;
  for (int i=0; i<DATA_SIZE; i++) { a[i] = i+1; b[i] = i+1; }
  double tdata = omp_get_wtime();
#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    num = numT;
    sumVectors(DATA_SIZE, a, b, c, tid, numT);
  }
  tdata = omp_get_wtime() - tdata;
  printf("first %f last %f in time %f using %d threads\n
  return 0;
}
```

```
void sumVectors(int N, double *A, double *B, double *C, int tid, int numT)
  // determine start & end for each thread
  int start =   tid * N / numT;
  int end = (tid+1) * N / numT;
  if (tid == numT-1)
    end = N;


  // do the vector sum for threads bounds
  for(int i=start; i<end; i++) {
    C[i] = A[i]+B[i];
  }
}
```

Implicit Barrier in Code

A single copy of A, B, and C shared among all threads




```
openmp >export env OMP_NUM_THREADS=1; ./a.out
first 2.000000 last 200000.000000 in time 0.000902 using 1 threads
openmp >export env OMP_NUM_THREADS=2; ./a.out
first 2.000000 last 200000.000000 in time 0.000678 using 2 threads
openmp >export env OMP_NUM_THREADS=4; ./a.out
first 2.000000 last 200000.000000 in time 0.000652 using 4 threads
openmp >export env OMP_NUM_THREADS=8; ./a.out
first 2.000000 last 200000.000000 in time 0.000693 using 8 threads
```

The for is such an obvious candidate for threads:

Code/Parallel/openmp/sum2.c

```
#include <omp.h>
#include <stdio.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  double a[DATA_SIZE], b[DATA_SIZE], c[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++) { a[i] = i+1; b[i] = i+1; }
  double tdata = omp_get_wtime();
#pragma omp parallel
  {
   #pragma omp for
   for (int i=0; i<DATA_SIZE; i++)
      c[i] = a[i]+b[i];
  }
  tdata = omp_get_wtime() - tdata;
  printf("first %f last %f in time %f \n",c[0], c[DATA_SIZE-1], tdata);
  return 0;
}
```

Code/Parallel/openmp/sum3.c

```
#pragma omp parallel for
    for (int i=0; i<DATA_SIZE; i++)
     c[i] = a[i]+b[i];
```

How About Dot Product?

Code/Parallel/openmp/dot1.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  int nThreads = 0;
  double dot = 0, a[DATA_SIZE], sum[64];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;
  for (int i=0; i<64; i++) sum[i] = 0;

#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    if (tid == 0)  nThreads = numT;
    for (int i=tid; i<DATA_SIZE; i+= numT)
      sum[tid] += a[i]*a[i];
  }
  for (int i=0; i<nThreads; i++)
      dot += sum[i];
  dot = sqrt(dot);
  printf("dot %f\n", dot);
  return 0;
}
```

Create a shared array to store data

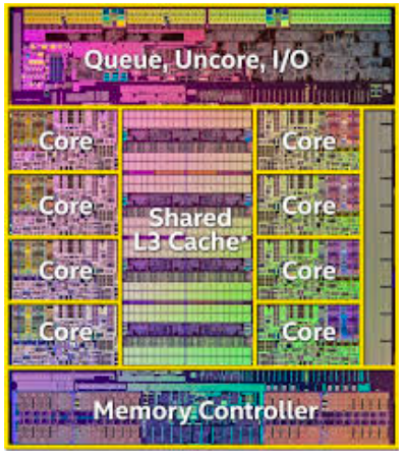
Iterate over big array using thread id and number of threads

Combine sequentially

Poor Performance?

- Want high performance for shared memory: Use Caches!
  - Each processor has its own cache (or multiple caches)
  - Place data from memory into cache
  - Writeback cache: don't send all writes over bus to memory

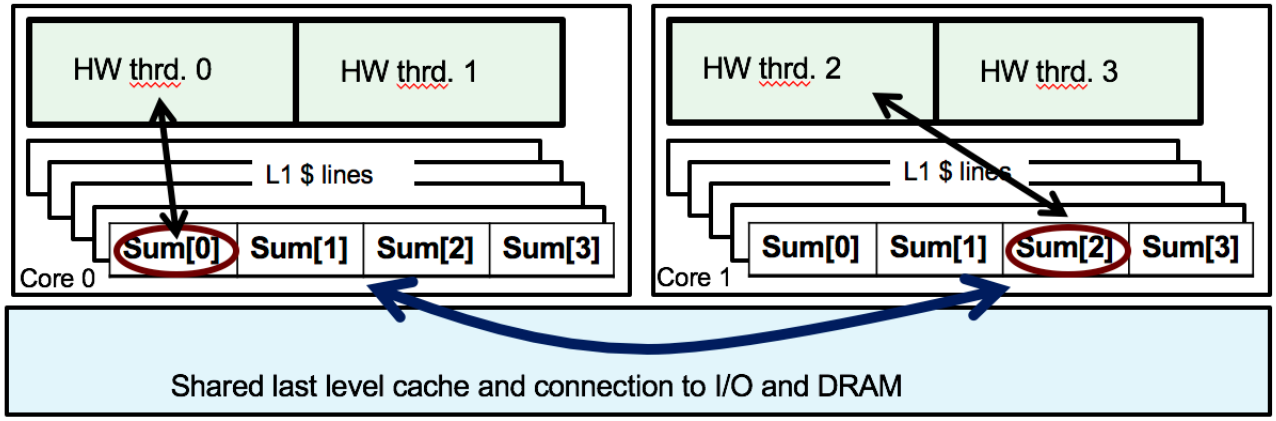
L3 Cache Actually Shared



- Problem is in multi-threaded model with all threads wanting to WRITE same spatially temporal data we have contention at the cache line in the L3 cache

False Sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to "slosh back and forth" between threads … This is called **false sharing** or sequential **consistency**.



- Sequential Consistency problem is pervasive and performance critical in shared memory

Solution?

source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere

AVOID FALSE SHARING

Code/Parallel/openmp/dot2.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000
#define PAD 64

int main(int argc, const char **argv) {
  int nThreads = 0;
  double dot = 0, a[DATA_SIZE], sum[64][PAD];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;
  for (int i=0; i<64; i++) sum[i][0]= 0;

#pragma omp parallel
  {
     int tid = omp_get_thread_num();
     int numT = omp_get_num_threads();
     if (tid == 0)  nThreads = numT;
     for (int i=tid; i<DATA_SIZE; i+= numT)
       sum[tid][0]+= a[i]*a[i];
  }
  for (int i=0; i<nThreads; i++)
      dot += sum[i][0];
  dot = sqrt(dot);
  printf("dot %f \n", dot);
  return 0;
```

Pad the shared array to store data to avoid false sharing

SYNCHRONIZATION

Code/Parallel/openmp/dot3.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  double dot = 0;
  double a[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;

#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    double sum = 0.;
    for (int i=tid; i<DATA_SIZE; i+= numT)
      sum += a[i]*a[i];
#pragma omp_critical
    dot += sum;
  }
  dot = sqrt(dot);
    printf("dot %f  \n", dot);
  return 0;
}
```

REDUCTION

Code/Parallel/openmp/dot4.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  double dot = 0;
  double a[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;

#pragma omp parallel reduction(+:dot)
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    double sum = 0.;
#pragma omp for
    for (int i=tid; i<DATA_SIZE; i+= numT)
      sum += a[i]*a[i];

    dot += sum;
  }
  dot = sqrt(dot);
   printf("dot %f  \n", dot);
  return 0;
}
```

Additional Reduction Operators

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

Are Pitfalls to Parallel Loops might only occur for experienced programmer!

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent ... So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

Note: loop index "i" is private by default

```
X = 0.5*dx;
for (int i=0; i<numSteps; i++) {
    pi += 4./(1.+x*x);
    x += dx;
 }
```

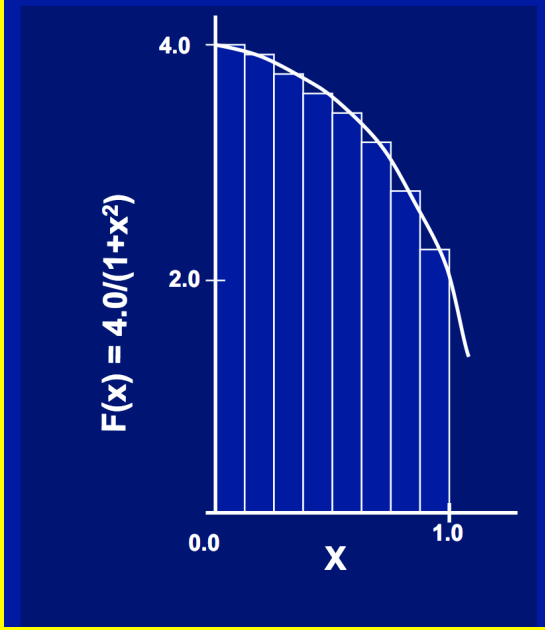
```
#pragma omp parallel for reduction(+:pi)
  for (int i=0; i<numSteps; i++) {
      x = (i+0.5)*dx;
      pi += 4./(1.+x*x);
  }
```

Remove loop carried dependence

- OpenMP provides multi-threaded capabilities to C, C++ and Fortran Programs
- In a threaded environment, threads communicate by sharing data
- Unintended sharing of data causes **race conditions**
- **Race Condition**: program output is different every time you run the program, a consequence of the threads being scheduled differently
- OpenMP provides constructs to control what blocks of code are run in parallel and also constructs for providing access to shared data using synchronization
- Synchronization has overhead consequences, you have to minimize them to get good speedup.

Exercise: Parallelize Compute PI using OpenMP

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

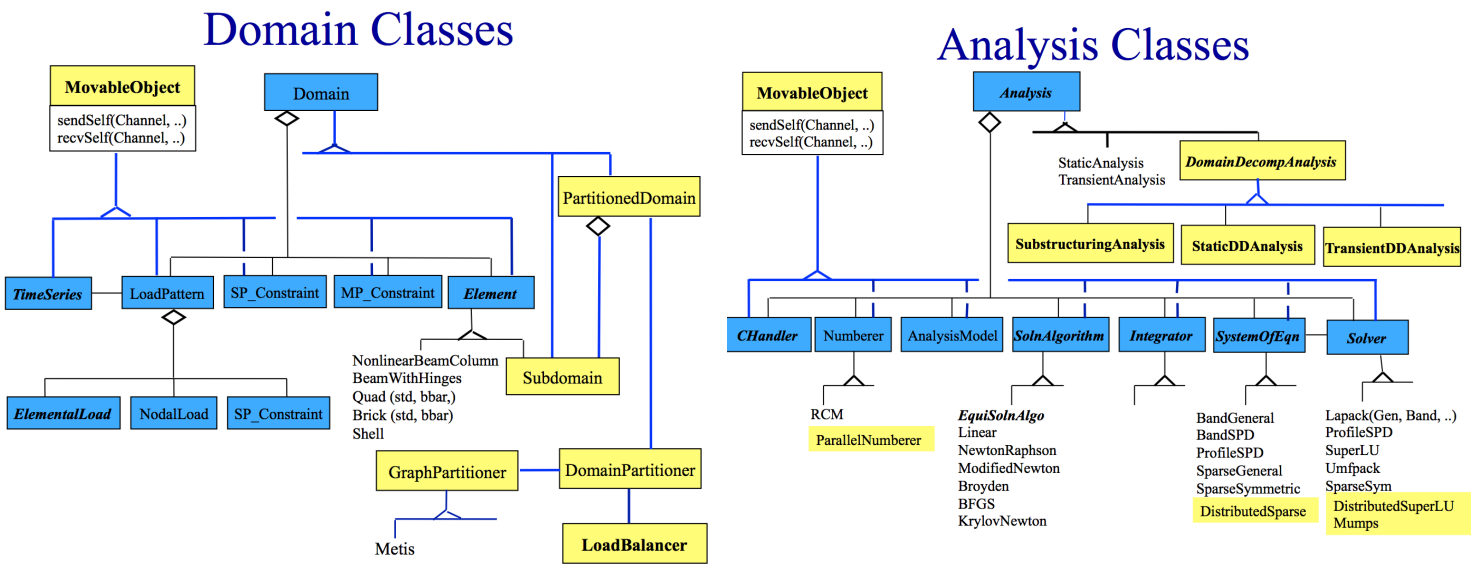
Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere

```
#include <stdio>
static int long numSteps = 100000;
int main() {
  double pi = 0; double time=0;
  // your code
  for (int i=0; i<numSteps; i++) {
    // your code
  }
   // your code
  printf("PI = %f, duration: %f ms\n",pi, time);
  return 0;
}
```

What is OpenSees?

- OpenSees is an Open-Source Software Framework written in C++ for developing nonlinear Finite Element Applications for both sequential and **PARALLEL** environments.



The OpenSees Interpreters

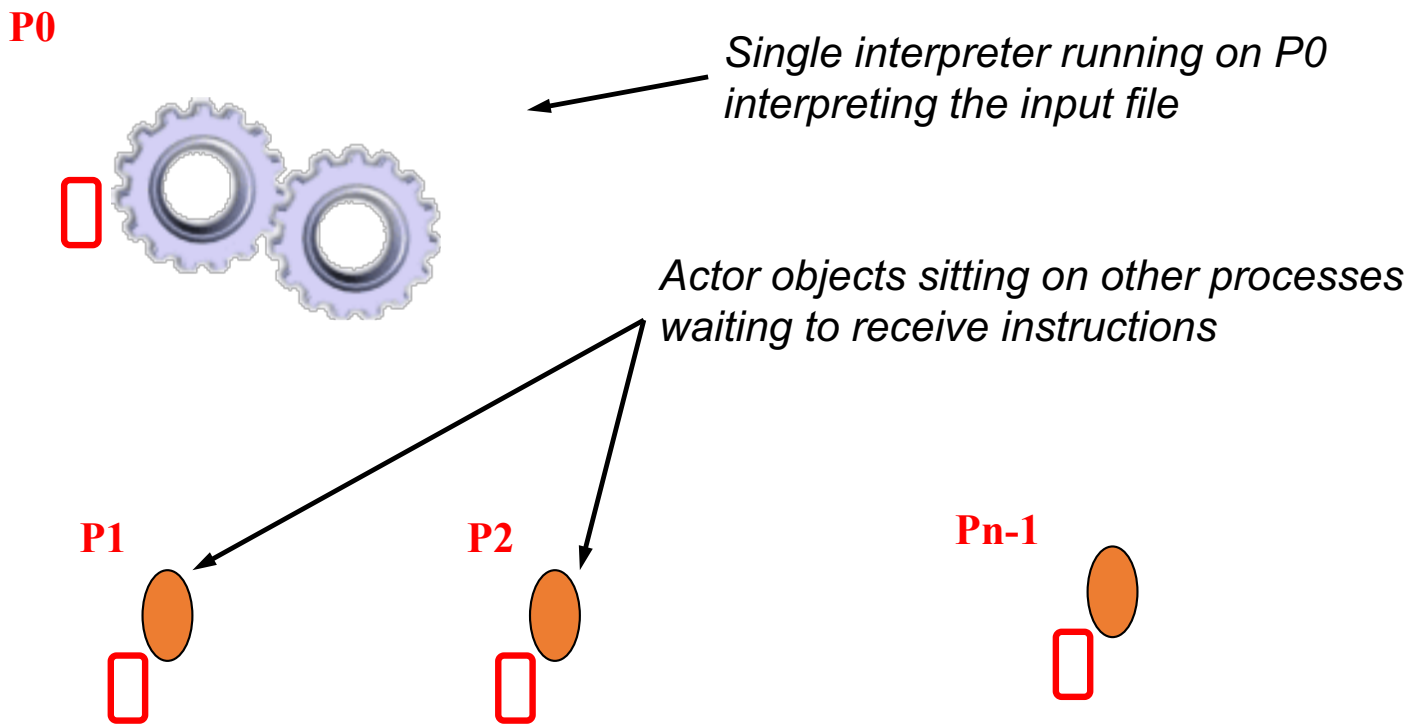
- There are 4 released interpreters:
  1. OpenSees.exe,
  2. OpenSeesTk.exe
  3. OpenSeesSP.exe
  4. OpenSeesMP.exe

So What are OpenSeesSP.exe and OpenSeesMP.exe ?

Parallel OpenSees Interpreters

- OpenSeesSP: An application for large models which will parse and execute the exact same script as the sequential application. The difference being the element state determination and equation solving are done in parallel.
- OpenSeesMP: An application for **BOTH** large models and parameter studies.

OpenSeesSP:
An application for Large Models



Modified Commands

- System command is modified to accept new parallel equation solvers

  system Mumps

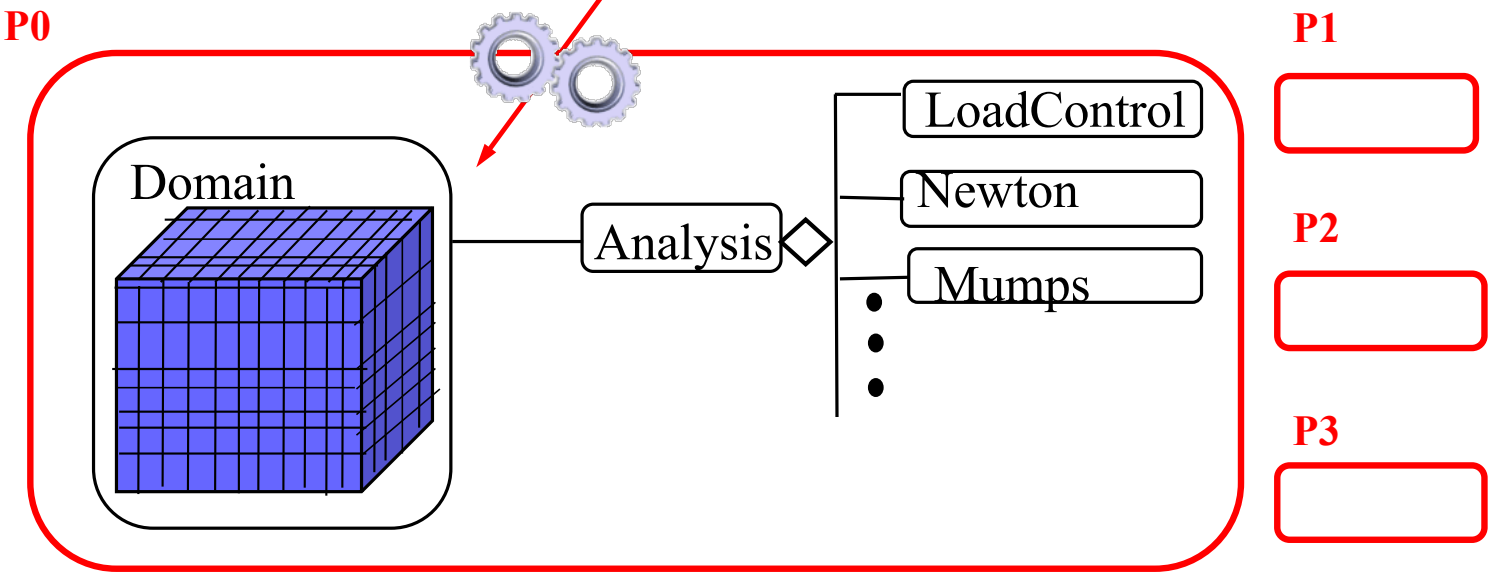
  system Diagonal

Model Built and Analysis Constructed in P0

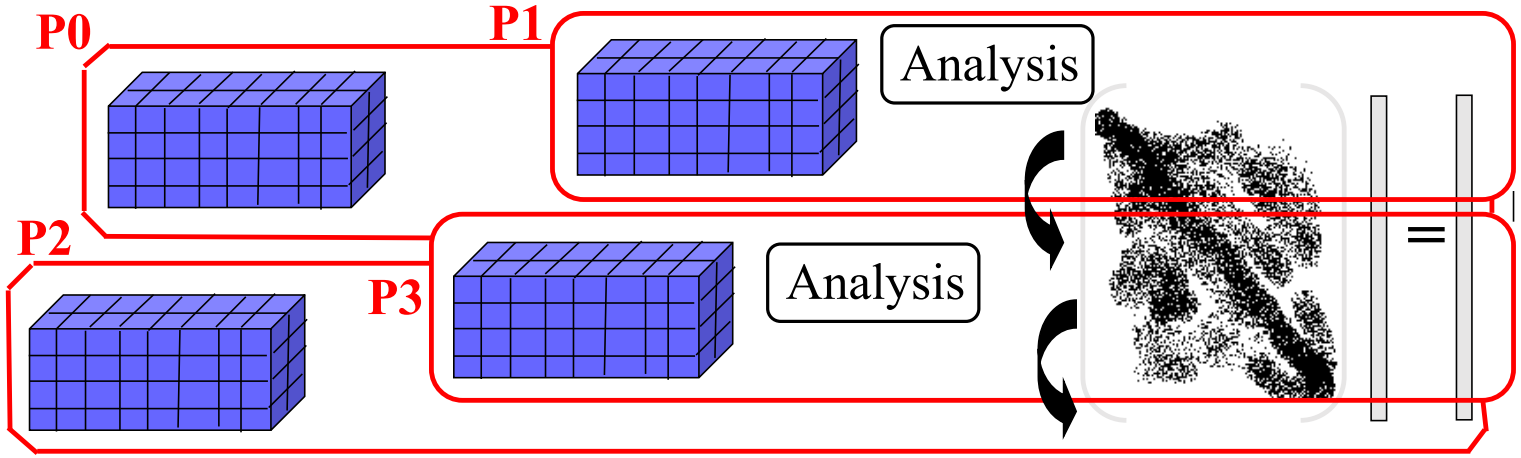
```
#build the model
source model.tcl
#build the analysis
system Mumps
constraints Transformation
numberer Plain
test NormDispIncr 1.0e-12 10 3
algorithm Newton
integrator LoadControl
analysis Static
```

Single interpreter running on P0

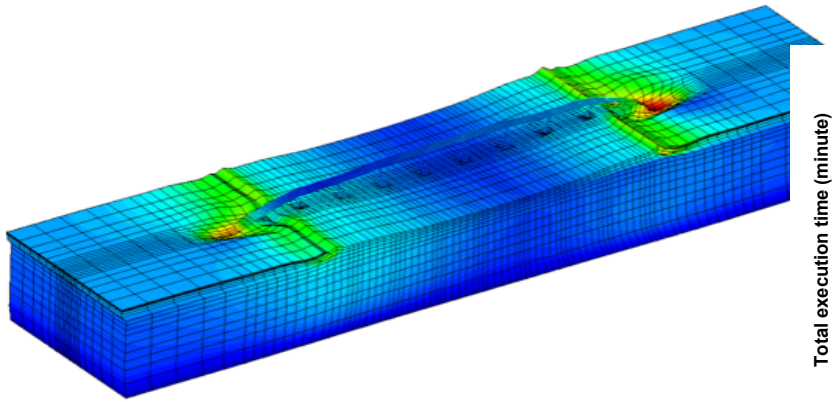
Interpreting the input file



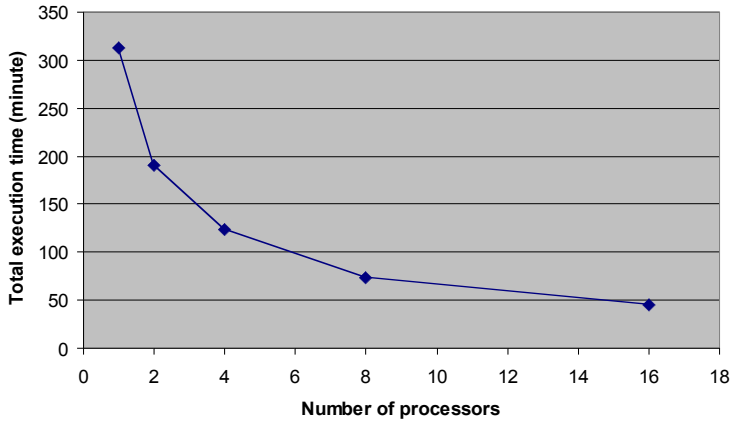
```
#build the model
source modelP.tcl
#build the analysis
system Mumps
constraints Transformation
numberer Plain
test NormDispIncr 1.0e-12 10 3
algorithm Newton
integrator LoadControl
analysis Static
analyze 10
```

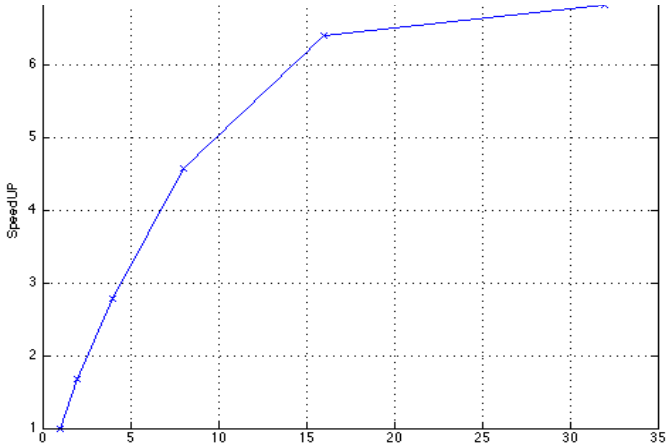


Example Results:
Humboldt Bay Bridge Model



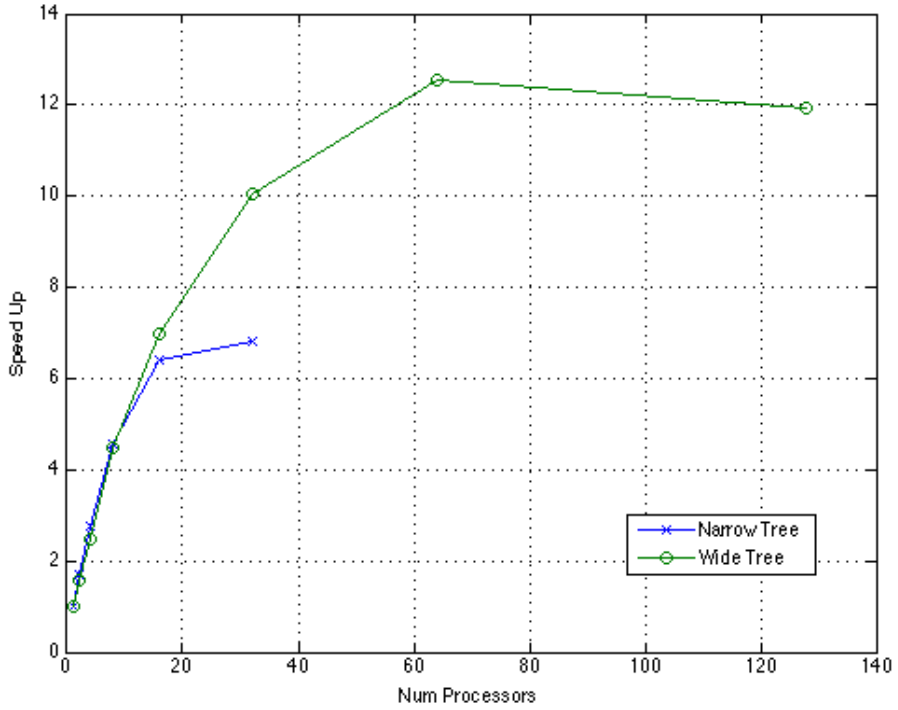
100,000+ DOF Model
Implicit Integration
Mumps Direct Solver





Effect of Shape of Elimination Tree on Performance:

50,000 DOF+ Models
Implicit Integration
Direct Solver



* Note the fatter the elimination tree, the better the parallel performance of the direct solver – NOT GREAT SCALABILITY

Explicit Central Difference (Diagonal Solver)

Run	el. size (m)	Elements	Nodes	DOFs
A	20	54,026	59,032	156,768
B	10	404,751	424,512	1,193,283
C	5	3,130,301	3,208,822	9,307,563
D	2.5	24,615,801	24,928,842	73,515,123

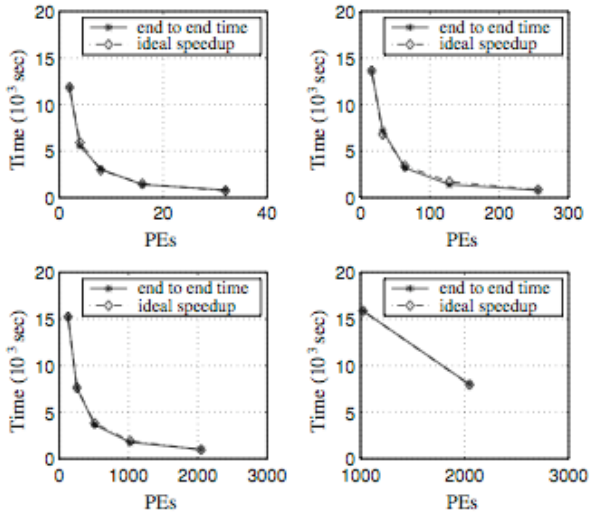
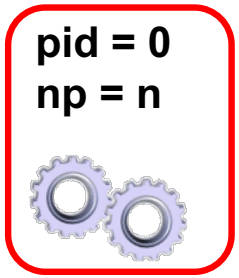
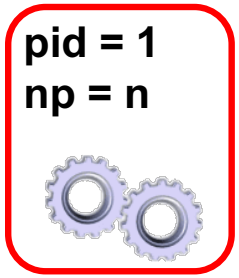


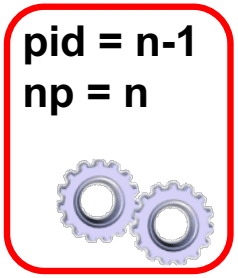
Fig. 18 Fixed-size, scalability plot at SDSC's DataStar. Upper row is runs A (*left*) and B (*right*), lower row is runs C (*left*) and D (*right*) (Table 3)

- Scales well with problem size

OpenSeesMP: An application for Large Models and Parameter Studies







Each process is running an interpreter and can determine it's unique process number and the total number of processes in computation

Based on this script can do different things

```
# source in the model and analysis procedures
set pid [getPID]
set np [getNP]

# build model based on np and pid
source modelP.tcl
doModel {$pid $np}

# perform gravity analysis
system Mumps
constraints Transformation
numberer Parallel
test NormDispIncr 1.0e-12 10 3
algorithm Newton
integrator LoadControl 0.1

analysis Static

set ok [analyze 10]
return $ok
```

New Commands added to OpenSeesMP:

- A Number of new commands have been added:

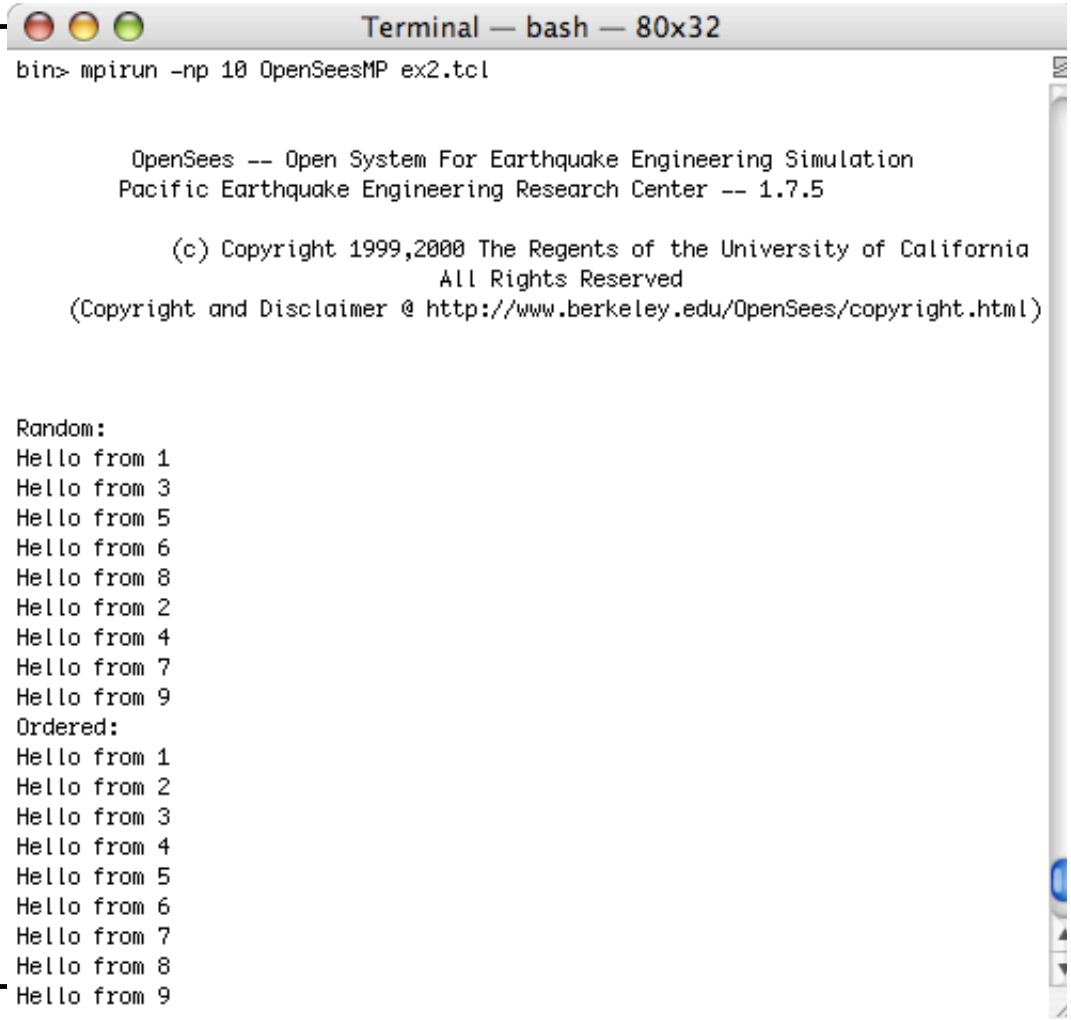
1. getNP returns number of processes in computation.
2. getPID returns unique pocess id {0,1, .. NP-1}
3. send -pid pid? data pid = { 0, 1, .., NP-1}
4. recv -pid pid? variableName pid = {0,1 .., NP-1, ANY}
5. barrier
6. domainChange

- These commands have been added to ALL interpreters (OpenSees, OpenSeesSP, and OpenSeesMP)

Example

ex2.tcl

```
set pid [getPID]
set np [getNP]
if {$pid == 0 } {
   puts "Random:"
   for {set i 1 } {$i < $np} {incr i 1} {
      recv -pid ANY msg
      puts "$msg"
   }
} else {
   send -pid 0 "Hello from $pid"
}
barrier
if {$pid == 0 } {
   puts "\nOrdered:"
   for {set i 1 } {$i < $np} {incr i 1} {
      recv -pid $i msg
      puts "$msg"
   }
} else {
   send -pid 0 "Hello from $pid"
}
```



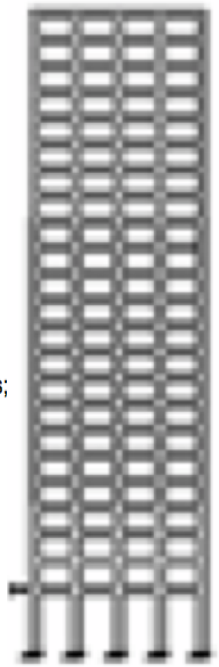
Steel Building Study

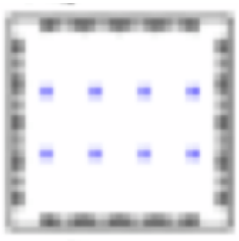
```
set pid [getPID]
set np [getNP]
set recordsFileID [open "peerRecords.txt" r]
set count 0;

foreach gMotion [split [read $recordsFileID] ¥n] {
  if {[expr $count % $np] == $pid} {

  source model.tcl
  source analysis.tcl

  set ok [doGravity]

  loadConst -time 0.0

  set gMotionList [split $gMotion "/"]
  set gMotionDir [lindex $gMotionList end-1]
  set gMotionNameInclAT2 [lindex $gMotionList end]
  set gMotionName [string range $gMotionNameInclAT2 0 end-4 ]

  set Gaccel "PeerDatabase $gMotionDir $gMotionName -accel 384.4 -dT dT -nPts nPts"
  pattern UniformExcitation 2 1 -accel $Gaccel

  recorder EnvelopeNode -file $gMotionDir$gMotionName.out -node 3 4 -dof 1 2 3 disp

  doDynamic [expr $dT*$nPts] $dT

   wipe
  }

  incr count 1;
}
```

7200 records
2 min a record
240 hours or 10 days
Ran on 2000 processors
on teragrid in less than 15 min.

Concrete Building Study

```
set pid [getPID]
set np [getNP]
set count 0;
source parameters.tcl
source ReadSMDFileNewFormat.tcl;
foreach GMfile $iGMFile {
   foreach Factor1248 $iFactor1248 {

      if {[expr $count % $np] == $pid} {

         set inFile $GMdir/$GMfile.AT2
         set outFile $GMdir/$GMfile.g3;
         ReadSMDFileNewFormat $inFile $outFile dt npts;

         wipe
         source GravityAnalysisScript.tcl

         loadConst -time 0.0;
         wipeAnalysis

         source  EQ_Recorder.tcl
         source EQAnalysisScript.tcl

        if {$ok == 0} {
           puts "Process $pid $GMfile x $Factor1248 FINISHED OK modelTime [getTime]]"
        } else {
           puts "Process $pid $GMfile x $Factor1248 FINISHED FAIL modeTime [getTime] desiredTime $TmaxAnalysis
        }
        incr count 1
      }
   }
}
```





113 records, 4 intensities
3 hour a record, 1356 hours or 56.5 days.

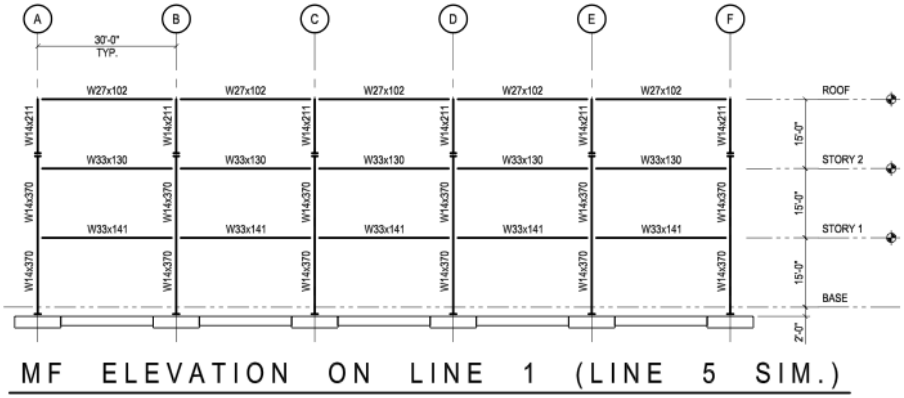
Ran on 452 processors
On XSEDE in less than 5 hours.

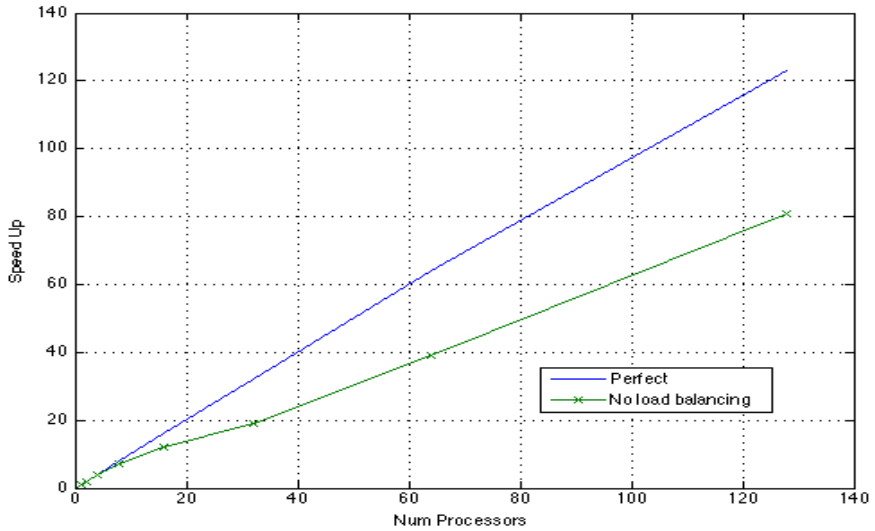
ATC-63/2 Project

3 buildings, 2 config a building, 44 records, 12 intensities, 5 hour a record, would have, taken 15840 hours or 660 days or 1.8 years. Ran on 44 processors of a XSEDE Ranger in less than 7 days*.*
(ATC-63/2 project using NEEShub)

Example Results Parameter Study: Moment Frame Reliability Analysis

100+ DOF Model
Explicit Integration,
Mumps Direct Solver
8000 Earthquake Simulations





* If more simulations than processors SpeedUp With "**Load Balancing**" will be "**Linear**"

Modified Commands

- Some existing commands have been modified to allow analysis of large models in parallel:
  1. numberer

     numberer ParallelPlain

     numberer ParallelRCM
  2. system

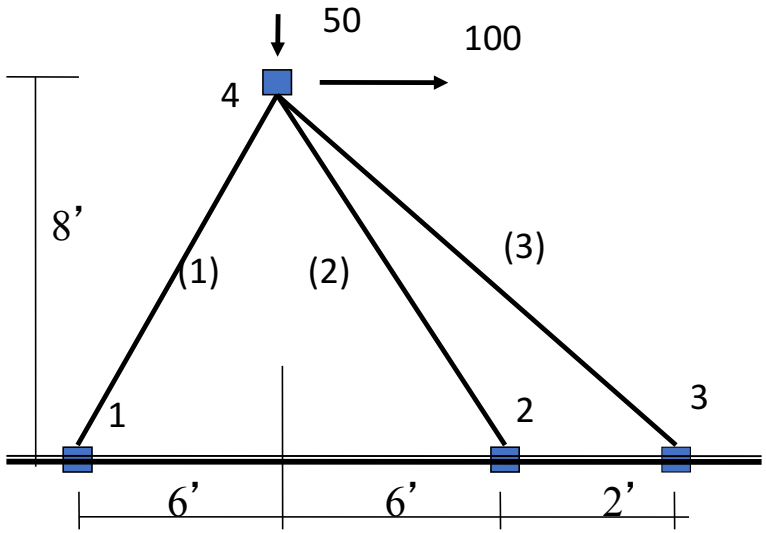
     system Mumps <-ICNTL14 %?>
  3. integrator

     integrator ParallelDisplacementControl node? Dof? dU?
- Use these ONLY IF PARALLEL MODEL

Example Parallel Model:

ex4.tcl

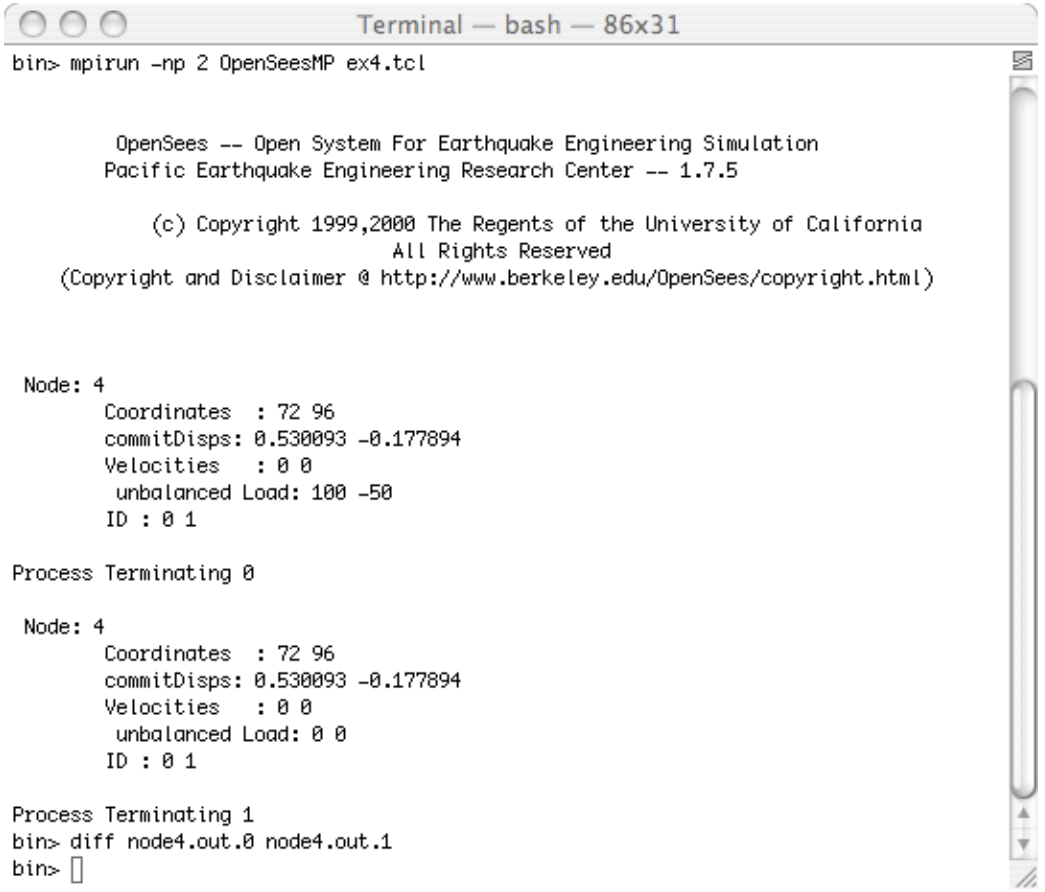
```
set pid [getPID]
set np [getNP]
if {$np != 2}  exit

model BasicBuilder -ndm 2 -ndf 2
uniaxialMaterial Elastic 1 3000
if {$pid == 0} {
    node 1   0.0  0.0
    node 4  72.0 96.0
    fix 1 1 1
    element truss 1 1 4 10.0 1
    pattern Plain 1 "Linear" {
        load 4 100 -50
    }
} else {
    node 2 144.0  0.0
    node 3 168.0  0.0
    node 4  72.0 96.0
    fix 2 1 1
    fix 3 1 1
    element truss 2 2 4 5.0 1
    element truss 3 3 4 5.0 1
}
```



	E	A
1	3000	10
2	3000	5
3	3000	5

Example Parallel Analysis:

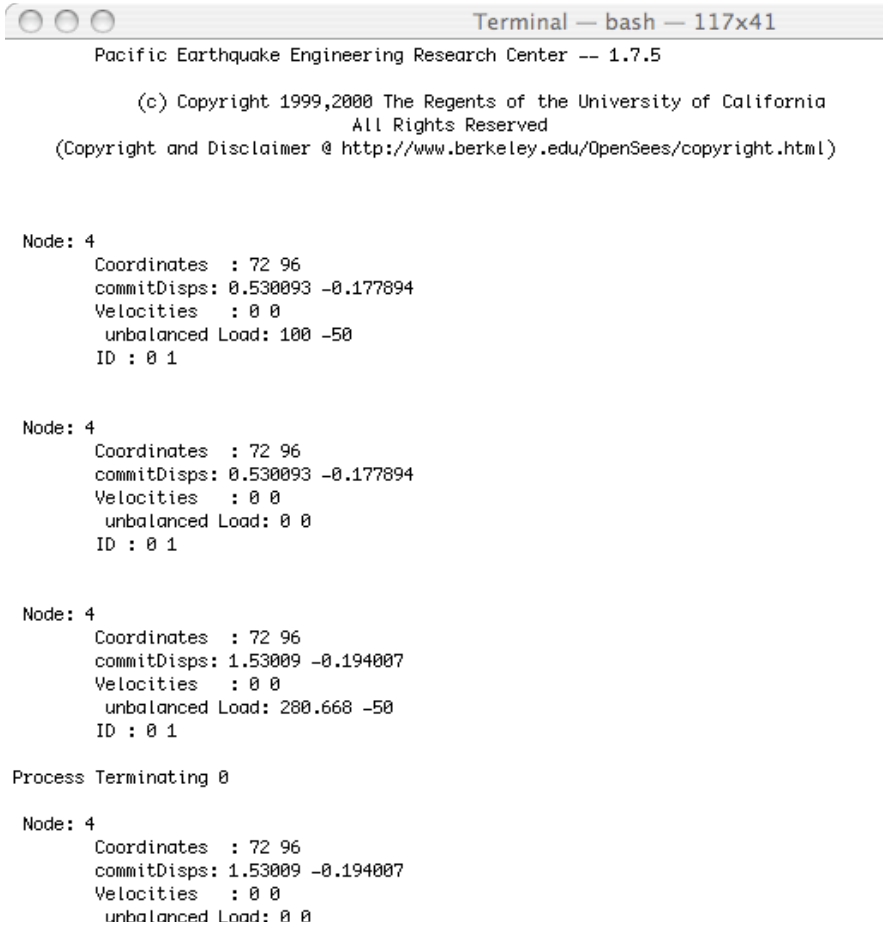
```
#create the recorder
recorder Node -file node4.out.$pid -node 4 -dof 1 2 disp

#create the analysis
constraints Transformation
numberer ParallelPlain
system Mumps
test NormDispIncr 1.0e-6 6 2
algorithm Newton
integrator LoadControl 0.1
analysis Static

#perform the analysis
analyze 10

# print to screen node 4
print node 4
```



Parallel Displacement Control and domainChange!

ex5.tcl

```
source ex4.tcl

loadConst - time 0.0

if {$pid == 0} {
    pattern Plain 2 "Linear" {
        load 4 1 0
    }
}

domainChange

integrator ParallelDisplacementControl 4 1 0.1
analyze 10
```



Things to Watch For

1. Deadlock (program hangs)
   - send/recv messages
   - Opening files for writing & not closing them
2. Race Conditions (different results every time run problem)
   - parallel file system.
3. Load Imbalance
   - poor initial task assignment.

Outline

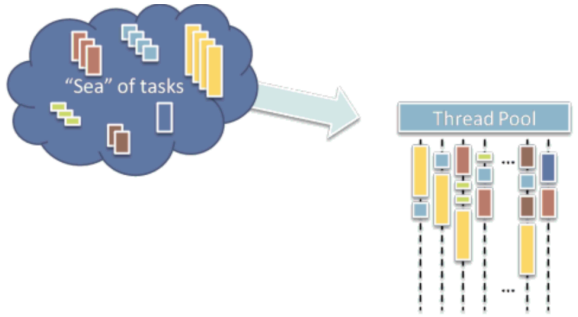
- Parallel Machines & Parallel Machine Models
- Parallel Programming
  - Message Passing With MPI
  - Shared Memory Programming with OpenMP
- Dynamic Load Balancing

Ignoring co-processors and GPUs

many slides source: CS267, Jim Demmell

Considerations for Parallel Programming:

- Finding enough parallelism (Amdahl's Law)
- Granularity – how big should each parallel task be
- Locality – moving data costs more than arithmetic
- Load balance – don't want 1K processors to wait for one slow one
- Coordination and synchronization – sharing data safely
- Performance modeling/debugging/tuning
- Where to put the task,



All of these things makes parallel programming even harder than sequential programming.

Load Balancing Solutions:

The key question: When is information (task costs, task dependencies, and task locality) about the load balancing problem known. Leads to a spectrum of solutions:

- Static scheduling. All information is available to scheduling algorithm, which runs before any real computation starts.
  - Off-line algorithms, eg graph partitioning, DAG scheduling (Pegasus)
- Semi-static scheduling. Information may be known at program startup, or the beginning of each time step, or at other well-defined points. Offline algorithms may be used even though the problem is dynamic.
- Dynamic scheduling. Information is not known until mid-execution.

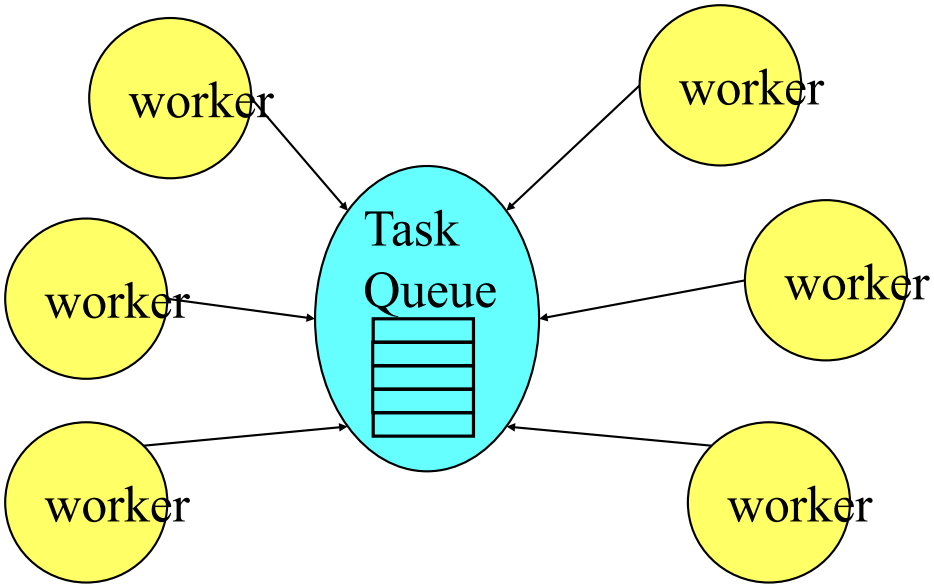
Dynamic Scheduling (Dynamic Load Balancing)

In dynamic load balancing we Adjust Task Assignment **as Processing Takes Place**. 2 Main Approaches:

- **Work Sharing**: processors balance the workload using a globally shared work queue(s) that contain tasks to be computed. If a task generates additional tasks, these can be pushed back to task queue to share.
- **Work Stealing**: processors keep own queues, idle processors search among the other processors in the computation in order to find surplus work.

Work Sharing
(Centralized Scheduling /Self Scheduling)

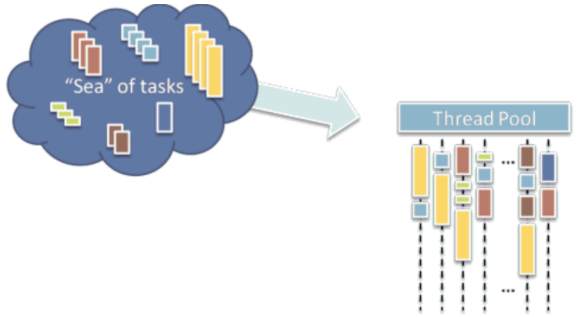
- Keep a queue of tasks waiting to be done
  - May be done by manager process
  - Or a shared data structure protected by locks
- When free, Worker process requests some tasks!



Source: Prof. Demmel, CS267, UC Berkeley

Considerations for Parallel Programming:

- Finding enough parallelism (Amdahl's Law)
- Granularity – how big should each parallel task be
- Locality – moving data costs more than arithmetic
- Load balance – don't want 1K processors to wait for one slow one
- Coordination and synchronization – sharing data safely
- Performance modeling/debugging/tuning
- Where to put the task,



All of these things makes parallel programming even harder than sequential programming.

Variations on Work Sharing

- Typically, don't want to grab smallest unit of parallel work, e.g., a single iteration
  - Too much contention at shared queue
- Instead, choose a chunk of tasks of size K.
  - If K is large, access overhead for task queue is small
  - If K is small, we are likely to have even finish times (load balance)
- A number of Variations:
  - Guided self-scheduling
  - Tapering

Source: Prof. Demmel, CS267, UC Berkeley

Guided Self-Scheduling

- Idea: use larger chunks at the beginning to avoid excessive overhead and smaller chunks near the end to even out the finish times.
  - The chunk size K_i at the i^{th} access to the task pool is given by

$$K_i = \text{ceiling}(R_i/p)$$

  - where R_i is the total number of tasks remaining and
  - p is the number of processors
- See Polychronopoulos & Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," IEEE Transactions on Computers, Dec. 1987.

Source: Prof. Demmel, CS267, UC Berkeley

Tapering

- Idea: the chunk size, K_i is a function of not only the remaining work, but also the task cost variance
  - variance is estimated using history information
  - high variance => small chunk size should be used
  - low variance => larger chunks OK

- See S. Lucco, "Adaptive Parallel Programs," PhD Thesis, UCB, CSD-95-864, 1994.
  - Gives analysis (based on workload distribution)
  - Also gives experimental results -- tapering always works at least as well as GSS, although difference is often small

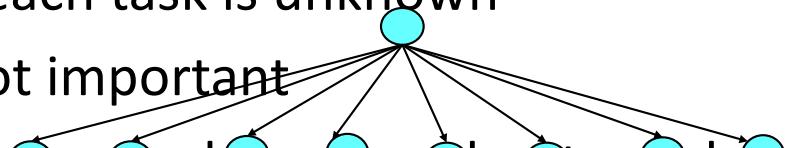
Source: Prof. Demmel, CS267, UC Berkeley

When is Work Sharing a Good Idea?

Useful when:

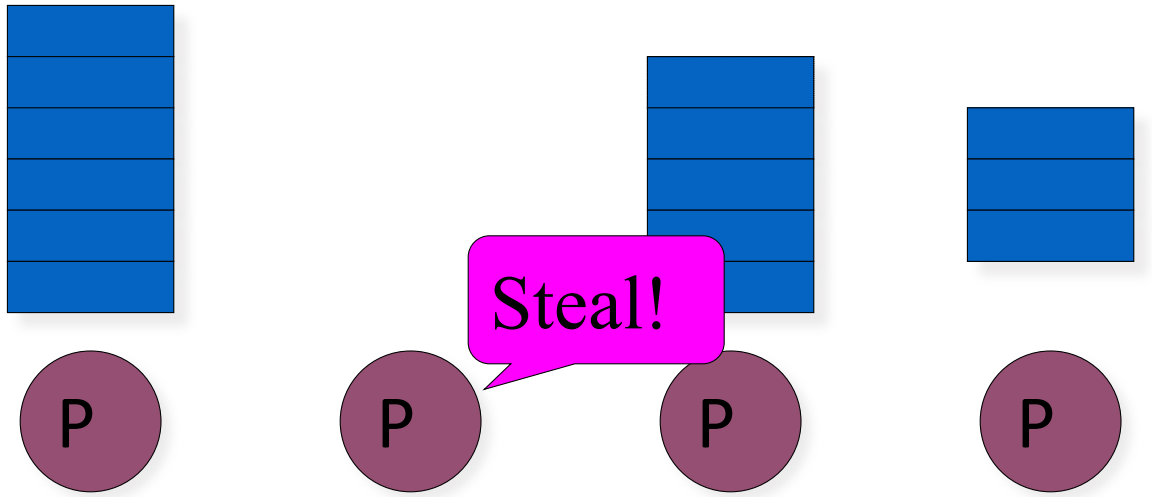
- A batch (or set) of tasks without dependencies
  - can also be used with dependencies, but most analysis has only been done for task sets without dependencies

- The cost of each task is unknown
- Locality is not important
- Shared memory machine, or at least number of processors is small – centralization is OK



Work Stealing
(Work Crews/Distributed Load Balancing)

- Each processor processes tasks in its queue
- When finished, steals work from a processor that is busy
- Requires asynchronous communication



Load Balancing Solutions:

The key question: When is information (task costs, task dependencies, and task locality) about the load balancing problem known. Leads to a spectrum of solutions:

- Static scheduling. All information is available to scheduling algorithm, which runs before any real computation starts.
  - Off-line algorithms, eg graph partitioning, DAG scheduling (Pegasus)
- Semi-static scheduling. Information may be known at program startup, or the beginning of each timestep, or at other well-defined points. Offline algorithms may be used even though the problem is dynamic.
- Dynamic scheduling. Information is not known until mid-execution.

Source: Prof. Demmel, CS267, UC Berkeley

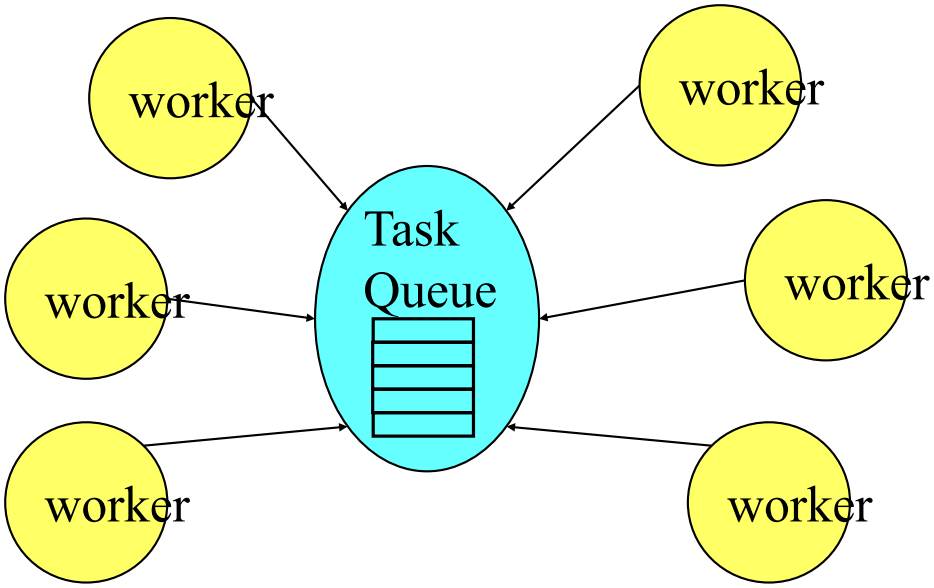
Dynamic Scheduling (Dynamic Load Balancing)

In dynamic load balancing we Adjust Task Assignment **as Processing Takes Place**. 2 Main Approaches:

- **Work Sharing**: processors balance the workload using a globally shared work queue(s) that contain tasks to be computed. If a task generates additional tasks, these can be pushed back to task queue to share.
- **Work Stealing**: processors keep own queues, idle processors search among the other processors in the computation in order to find surplus work.

Work Sharing
(Centralized Scheduling /Self Scheduling)

- Keep a queue of tasks waiting to be done
  - May be done by manager process
  - Or a shared data structure protected by locks
- When free, Worker process requests some tasks!



Source: Prof. Demmel, CS267, UC Berkeley

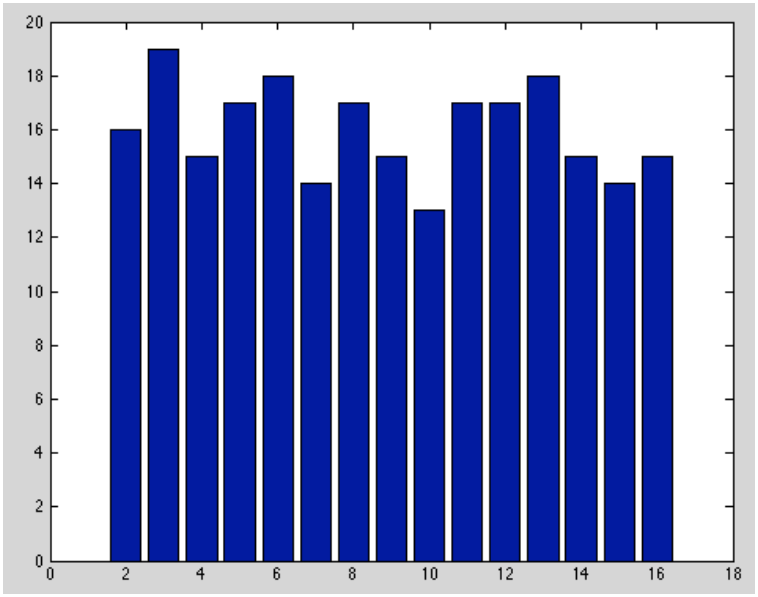
Work Sharing

MRF_MP2.tcl

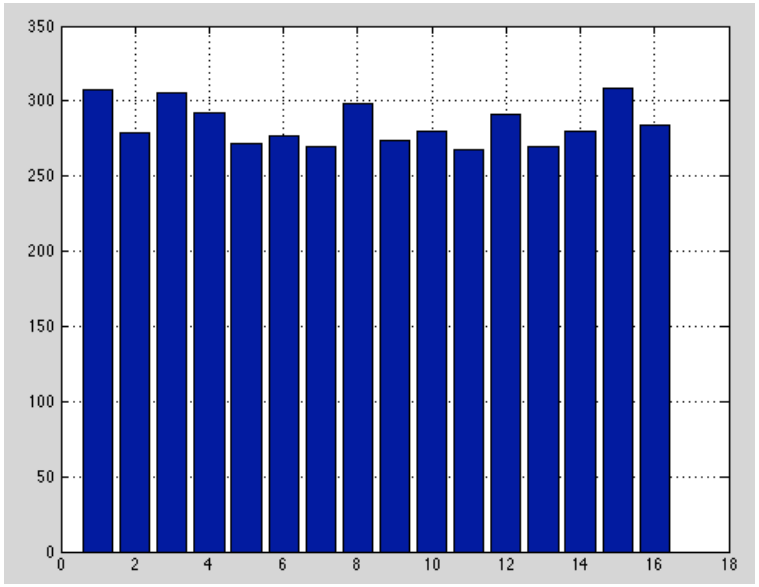
```
set np [getNP]
set pid [getPID]
# set some parameters & open some local files
….
if {$pid == 0} {
   foreach Hazard $Hazards {

      set dataDir [format "Oak-%i-50-Output" $Hazard] ;
      file mkdir $dataDir/DriftAcceleration;
      file mkdir $dataDir/ForceDeformation

      foreach eqDir $eqDirections {
         for {set i 1 } { $i <= $eqNum } { incr i} {
            recv -pid ANY pidWorker
            send -pid $pidWorker "$Hazard $eqDir $i"
         }
      }
   }
   # tell all processors we are done
   for {set i 1} {$i < $np} {incr i 1} {
      recv -pid ANY pidWorker
      send -pid $pidWorker "DONE"
   }
} else {
```

```
} else {
   set done NOT_DONE;
   while {$done != "DONE"} {
      send -pid 0 $pid
      recv -pid 0 task

      set Hazard [lindex $task 0]

      if {$Hazard == "DONE"} {
         set done "DONE"
         break;
      }
      set eqDir [lindex $task 1]
      set i    [lindex $task 2]

      set tStart [clock clicks -milliseconds]
      set GMdir [format "./GMfiles/Oak-%i-50/" $Hazard];
      source buildModelWithGravity.tcl
      source addRecorders.tcl
      source Dynamic.EQ.tcl
      source processResults.tcl

      set tEnd [clock clicks -milliseconds]
      puts $timeOut "DURATION: $i $eqDir $Hazard [expr ($tEnd-$tStart)/1000.]"
   }
}
 set tEnd [clock clicks -milliseconds]
puts $timeOut "TOTAL DURATION: [expr ($tEnd-$tStartAll)/1000.]"
puts  "PID: $pid DURATION: [expr ($tEnd-$tStartAll)/1000.]"
# close local files
….
```

Results using 16 Processors



 # Tasks Completed *(pid vs # tasks)*



Duration *(pid v time)*

Variations on Work Sharing

- Typically, don't want to grab smallest unit of parallel work, e.g., a single iteration
  - Too much contention at shared queue
- Instead, choose a chunk of tasks of size K.
  - If K is large, access overhead for task queue is small
  - If K is small, we are likely to have even finish times (load balance)
- A number of Variations:
  - Guided self-scheduling
  - Tapering

Source: Prof. Demmel, CS267, UC Berkeley

Guided Self-Scheduling

- Idea: use larger chunks at the beginning to avoid excessive overhead and smaller chunks near the end to even out the finish times.
  - The chunk size K_i at the i^{th} access to the task pool is given by

$$K_i = \text{ceiling}(R_i/p)$$

  - where R_i is the total number of tasks remaining and
  - p is the number of processors
- See Polychronopoulos & Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," IEEE Transactions on Computers, Dec. 1987.

Source: Prof. Demmel, CS267, UC Berkeley

Tapering

- Idea: the chunk size, K_i is a function of not only the remaining work, but also the task cost variance
  - variance is estimated using history information
  - high variance => small chunk size should be used
  - low variance => larger chunks OK

- See S. Lucco, "Adaptive Parallel Programs," PhD Thesis, UCB, CSD-95-864, 1994.
  - Gives analysis (based on workload distribution)
  - Also gives experimental results -- tapering always works at least as well as GSS, although difference is often small

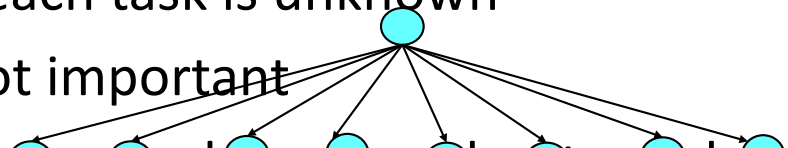
Source: Prof. Demmel, CS267, UC Berkeley

When is Work Sharing a Good Idea?

Useful when:

- A batch (or set) of tasks without dependencies
  - can also be used with dependencies, but most analysis has only been done for task sets without dependencies

- The cost of each task is unknown
- Locality is not important
- Shared memory machine, or at least number of processors is small – centralization is OK

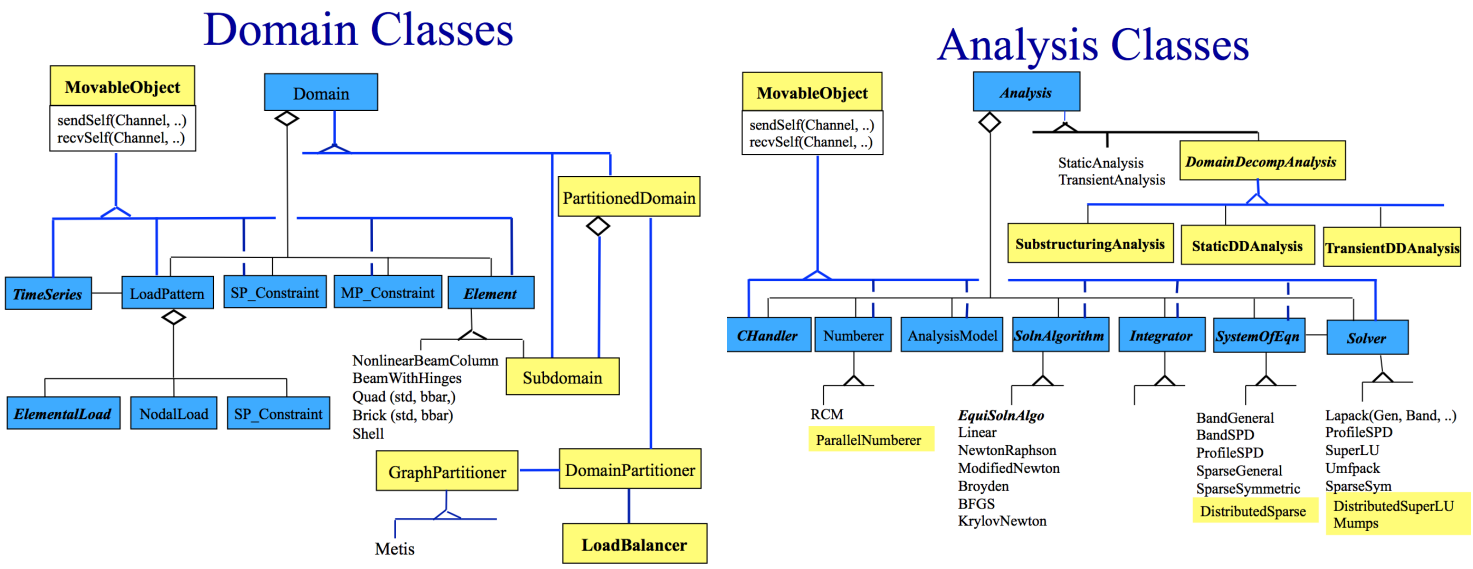


Extra Slides

many slides source: CS267, Jim Demmell

What is OpenSees?

- OpenSees is an Open-Source Software Framework written in C++ for developing nonlinear Finite Element Applications for both sequential and **PARALLEL** environments.



Parallel OpenSees Interpreters

- OpenSeesSP: An application for large models which will parse and execute the exact same script as the sequential application. The difference being the element state determination and equation solving are done in parallel.
- OpenSeesMP: An application for **BOTH** large models and parameter studies.

New Commands added to OpenSeesMP:

- A Number of new commands have been added:

1. getNP returns number of processes in computation.
2. getPID returns unique pocess id {0,1, .. NP-1}
3. send -pid pid? data pid = { 0, 1, .., NP-1}
4. recv -pid pid? variableName pid = {0,1 .., NP-1, ANY}
5. barrier
6. domainChange

- These commands have been added to ALL interpreters (OpenSees, OpenSeesSP, and OpenSeesMP)

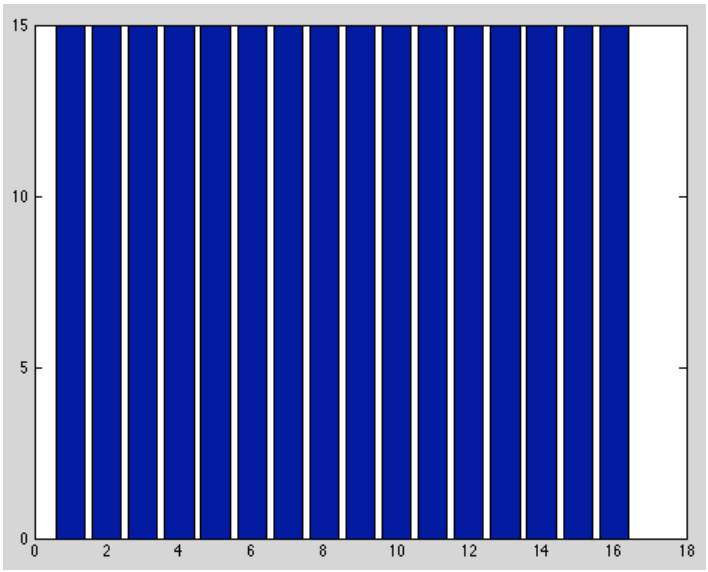
Parameter Study

MRF_MP1.tcl

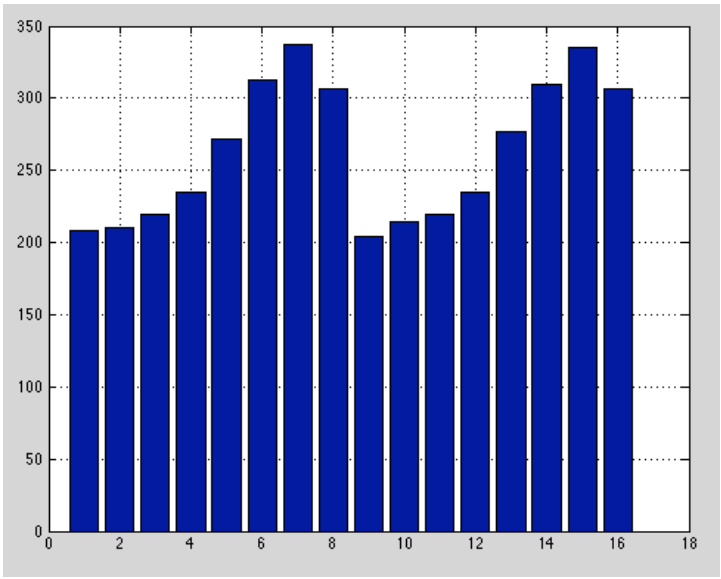
```
set np [getNP]
set pid [getPID]
set count 0
# set some parameters & open some local files
….
if {$pid == 0} {
   foreach Hazard $Hazards {
      set dataDir [format "Oak-%i-50-Output" $Hazard]
      file mkdir $dataDir/DriftAcceleration;
      file mkdir $dataDir/DriftDeformation;
}
barrier
foreach Hazard $Hazards {
   foreach eqDir $eqDirections {
      for { set i 1 } { $i <= $eqNum } { incr i} {
         if {[expr $count % $np] == $pid} {
            set tStart [clock clicks –milliseconds]
            set GMdir [format " /GMfiles/Oak-%i-50/" $Hazard]
            source buildModelWithGravity.tcl
            source addRecorders.tcl
            source Dynamic.EQ.tcl
            source processResults.tcl
            set tEnd [clock clicks -milliseconds]
            puts $timeOut "DURATION: $i $eqDir $Hazard [expr ($tEnd-$tStart\)/1000.]"
         }
         incr count 1
      }
   }
}
set tEnd [clock clicks -milliseconds]
puts $timeOut "TOTAL DURATION: [expr ($tEnd-$tStartAll)/1000.]"
puts  "PID: $pid DURATION: [expr ($tEnd-$tStartAll)/1000.]"
# close local files
….
```

To ensure only 1 processor does stuff to file system

Results using 16 Processors



 # Tasks Completed *(pid vs # tasks)*



Duration *(pid v time)*

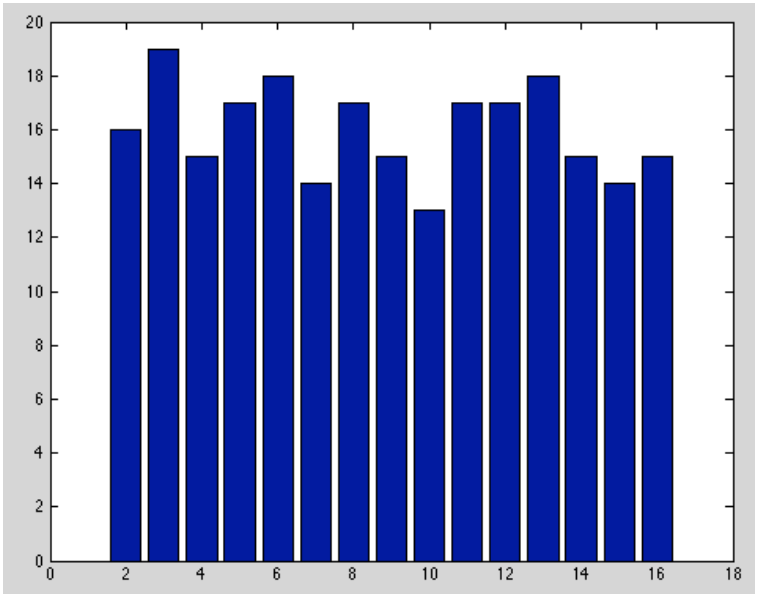
Work Sharing

MRF_MP2.tcl

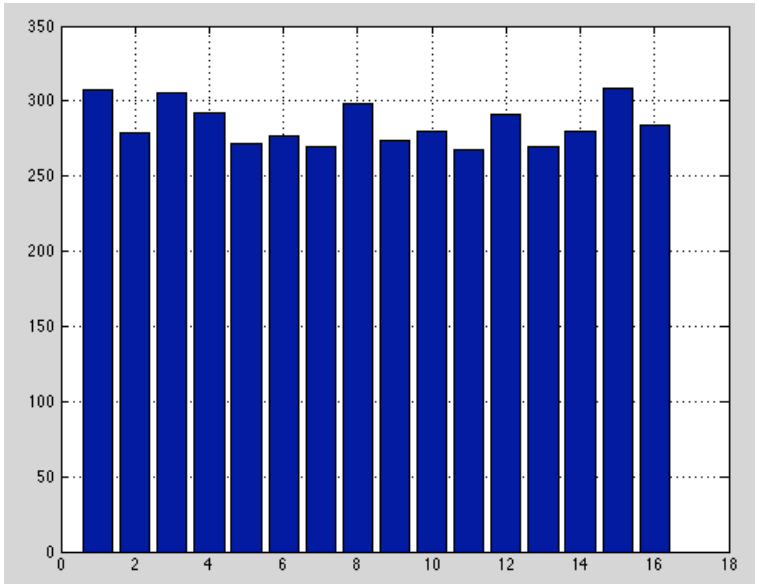
```
set np [getNP]
set pid [getPID]
# set some parameters & open some local files
….
if {$pid == 0} {
   foreach Hazard $Hazards {

      set dataDir [format "Oak-%i-50-Output" $Hazard] ;
      file mkdir $dataDir/DriftAcceleration;
      file mkdir $dataDir/ForceDeformation

      foreach eqDir $eqDirections {
         for {set i 1 } { $i <= $eqNum } { incr i} {
            recv -pid ANY pidWorker
            send -pid $pidWorker "$Hazard $eqDir $i"
         }
      }
   }
   # tell all processors we are done
   for {set i 1} {$i < $np} {incr i 1} {
      recv -pid ANY pidWorker
      send -pid $pidWorker "DONE"
   }
} else {
```

```
} else {
   set done NOT_DONE;
   while {$done != "DONE"} {
      send -pid 0 $pid
      recv -pid 0 task

      set Hazard [lindex $task 0]

      if {$Hazard == "DONE"} {
         set done "DONE"
         break;
      }
      set eqDir [lindex $task 1]
      set i    [lindex $task 2]

      set tStart [clock clicks -milliseconds]
      set GMdir [format "./GMfiles/Oak-%i-50/" $Hazard];
      source buildModelWithGravity.tcl
      source addRecorders.tcl
      source Dynamic.EQ.tcl
      source processResults.tcl

      set tEnd [clock clicks -milliseconds]
      puts $timeOut "DURATION: $i $eqDir $Hazard [expr ($tEnd-$tStart)/1000.]"
   }
}
 set tEnd [clock clicks -milliseconds]
puts $timeOut "TOTAL DURATION: [expr ($tEnd-$tStartAll)/1000.]"
puts  "PID: $pid DURATION: [expr ($tEnd-$tStartAll)/1000.]"
# close local files
….
```

Results using 16 Processors



 # Tasks Completed *(pid vs # tasks)*



Duration *(pid v time)*

Work Stealing

MRF_MP3.tc

```
set np [getNP]
set pid [getPID]
# set some parameters & open some local files
# define some procedures for work stealing
….
if {$pid == 0} {

   # create task queues
   createTaskQueues $np

   # add tasks to the different queus
   set count 0
   foreach Hazard $Hazards {
      set dataDir [format "Oak-%i-50-Output" $Hazard] ;
      file mkdir $dataDir/DriftAcceleration;
      file mkdir $dataDir/ForceDeformation;

      foreach eqDir $eqDirections {
         for {set i 1 } { $i <= $eqNum } { incr i} {
            set thePID [expr $count % $np]
            addTask $thePID $Hazard $eqDir $i
            incr count 1
         }
      }
   }
}

barrier
```

Work Stealing

MRF_MP3.tcl

```
#
# process my own tasks
#

set done NOT_DONE;
while {$done != "DONE"} {
   set task [getTask $pid]

   if {[llength $task] == 0} {
      set done "DONE"
      break;
   }

   set Hazard [lindex $task 0]
   set eqDir  [lindex $task 1]
   set i      [lindex $task 2]

   set tStart [clock clicks -milliseconds]

   set GMdir [format "./GMfiles/Oak-%i-50/" $Hazard];
   source buildModelWithGravity.tcl
   source addRecorders.tcl
   source Dynamic.EQ.tcl
   source processResults.tcl

   set tEnd [clock clicks -milliseconds]
   puts $timeOut "DURATION: $i $eqDir $Hazard [expr ($tEnd-$tStart)/1000.]"
}
```

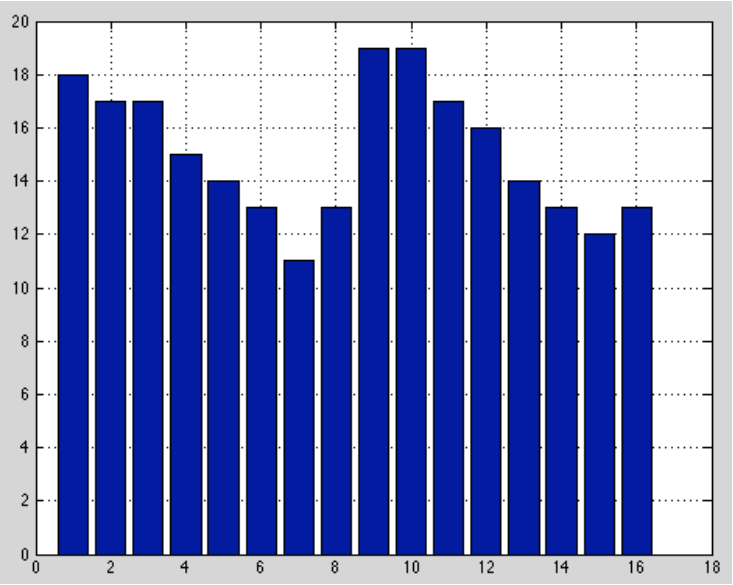
```
#
# done with my own queue, start stealing from others
#

set numDone 1;
set done "NOTDONE"
while {$done == "NOTDONE"} {

   set task [stealTask]

   if {$task == ""} {
      set done "DONE"
      break
   }

   set Hazard [lindex $task 0]
   set eqDir  [lindex $task 1]
   set i      [lindex $task 2]

   set tStart [clock clicks -milliseconds]

   set GMdir [format "./GMfiles/Oak-%i-50/" $Hazard];
   source buildModelWithGravity.tcl
   source addRecorders.tcl
   source Dynamic.EQ.tcl
   source processResults.tcl

   set tEnd [clock clicks –milliseconds]
   puts $timeOut "DURATION: $i $eqDir $Hazard [expr ($tEnd-$tStart)/1000.]
}
```

Work Stealing

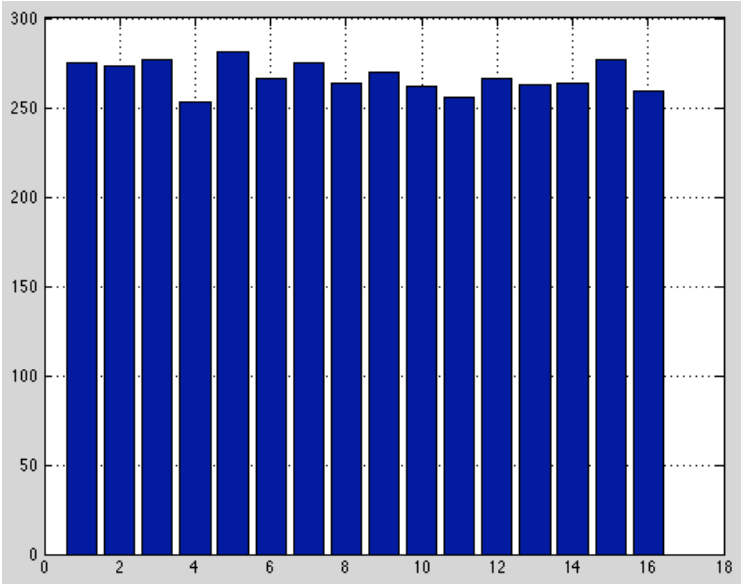
```
proc createTaskQueues {np} {
   # open some files to store tasks
   for {set i 0} {$i < $np} {incr i 1} {
                set taskFile [open tasks.$i w]
   }
   close $taskFile
}

proc addTask {thePID Hazard eqDir i} {
   set taskFile [open tasks.$thePID a]
   puts $taskFile "$Hazard $eqDir $i"
   close $taskFile
}

proc getTask {thePID} {
   if {[file size tasks.$thePID] == 0} {
     return ""
   }
   set taskFile [open tasks.$thePID r+]
   seek $taskFile 0
   gets $taskFile task
   set restData [read $taskFile]
   chan truncate $taskFile 0
   seek $taskFile 0
   set data [split $restData "\n"]
   foreach line $data {
      if {[llength $line] != 0} {
         puts $taskFile $line
      }
   }
   close $taskFile
   return $task
```

```
proc getRandomPID {np} {
   if {$np == 0} {
     return 0;
   }
   set maxFactor [expr $np + 1]
   set value [expr int([expr rand() * 100])]
   set value [expr int([expr $value % $maxFactor])]
   return [expr $value % $np]
}
proc stealTask {} {
   global pid
   global np
   global numDone
   global processesDone
   set foundOne 0
   set task "";
   while {$foundOne == 0 && $numDone < $np} {
     set otherProcess [getRandomPID [expr $np-$numDone-1]]
     set task [getTask [lindex $processesDone $otherProcess]]
     if {[llength $task] == 0} {
       set newProcessesDone {}
       for {set i 0} {$i < $np-$numDone} {incr i 1} {
         if {$i != $otherProcess} {
           lappend newProcessesDone [lindex$processesDone $i]
         }
       }
       set numDone [expr $numDone+1]
       set processesDone $newProcessesDone
     } else {
       set foundOne 1
     }
   }
   return $task
}
```

Results using 16 Processors



 # Tasks Completed *(pid vs # tasks)*



Duration *(pid v time)*

Work Stealing

MRF_MP3.tc

```
set np [getNP]
set pid [getPID]
# set some parameters & open some local files
# define some procedures for work stealing
….
if {$pid == 0} {

   # create task queues
   createTaskQueues $np

   # add tasks to the different queus
   set count 0
   foreach Hazard $Hazards {
      set dataDir [format "Oak-%i-50-Output" $Hazard] ;
      file mkdir $dataDir/DriftAcceleration;
      file mkdir $dataDir/ForceDeformation;

      foreach eqDir $eqDirections {
         for {set i 1 } { $i <= $eqNum } { incr i} {
            set thePID [expr $count % $np]
            addTask $thePID $Hazard $eqDir $i
            incr count 1
         }
      }
   }
}

barrier
```

Work Stealing

MRF_MP3.tcl

```
#
# process my own tasks
#

set done NOT_DONE;
while {$done != "DONE"} {
   set task [getTask $pid]

   if {[llength $task] == 0} {
      set done "DONE"
      break;
   }

   set Hazard [lindex $task 0]
   set eqDir  [lindex $task 1]
   set i      [lindex $task 2]

   set tStart [clock clicks -milliseconds]

   set GMdir [format "./GMfiles/Oak-%i-50/" $Hazard];
   source buildModelWithGravity.tcl
   source addRecorders.tcl
   source Dynamic.EQ.tcl
   source processResults.tcl

   set tEnd [clock clicks -milliseconds]
   puts $timeOut "DURATION: $i $eqDir $Hazard [expr ($tEnd-$tStart)/1000.]"
}
```

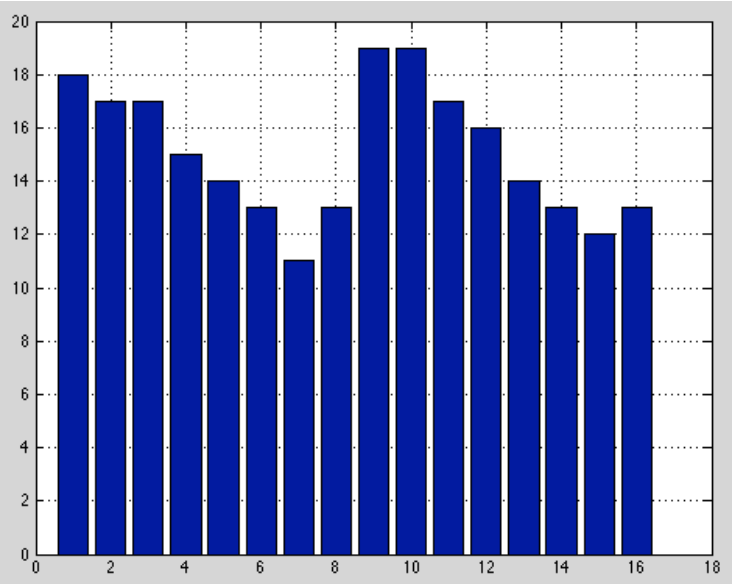
```
#
# done with my own queue, start stealing from others
#

set numDone 1;
set done "NOTDONE"
while {$done == "NOTDONE"} {

   set task [stealTask]

   if {$task == ""} {
      set done "DONE"
      break
   }

   set Hazard [lindex $task 0]
   set eqDir  [lindex $task 1]
   set i      [lindex $task 2]

   set tStart [clock clicks -milliseconds]

   set GMdir [format "./GMfiles/Oak-%i-50/" $Hazard];
   source buildModelWithGravity.tcl
   source addRecorders.tcl
   source Dynamic.EQ.tcl
   source processResults.tcl

   set tEnd [clock clicks –milliseconds]
   puts $timeOut "DURATION: $i $eqDir $Hazard [expr ($tEnd-$tStart)/1000.]
}
```

Work Stealing

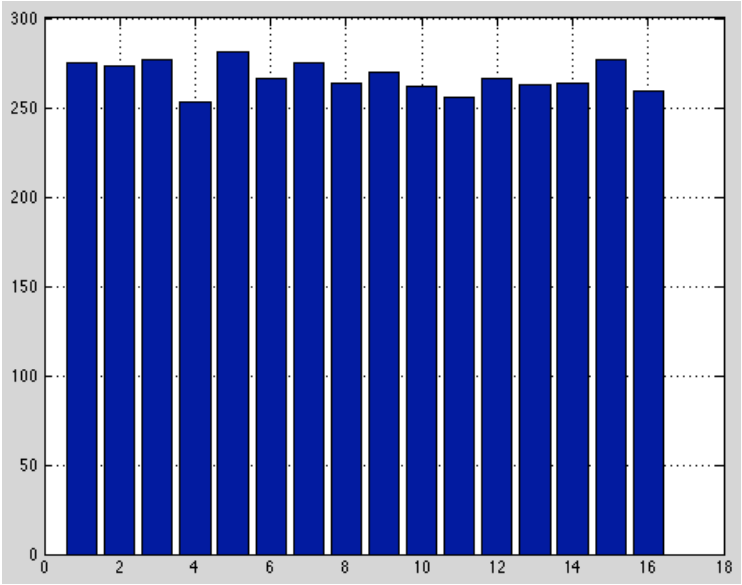
```
proc createTaskQueues {np} {
   # open some files to store tasks
   for {set i 0} {$i < $np} {incr i 1} {
                set taskFile [open tasks.$i w]
   }
   close $taskFile
}

proc addTask {thePID Hazard eqDir i} {
   set taskFile [open tasks.$thePID a]
   puts $taskFile "$Hazard $eqDir $i"
   close $taskFile
}

proc getTask {thePID} {
   if {[file size tasks.$thePID] == 0} {
     return ""
   }
   set taskFile [open tasks.$thePID r+]
   seek $taskFile 0
   gets $taskFile task
   set restData [read $taskFile]
   chan truncate $taskFile 0
   seek $taskFile 0
   set data [split $restData "\n"]
   foreach line $data {
      if {[llength $line] != 0} {
         puts $taskFile $line
      }
   }
   close $taskFile
   return $task
```

```
proc getRandomPID {np} {
   if {$np == 0} {
     return 0;
   }
   set maxFactor [expr $np + 1]
   set value [expr int([expr rand() * 100])]
   set value [expr int([expr $value % $maxFactor])]
   return [expr $value % $np]
}
proc stealTask {} {
   global pid
   global np
   global numDone
   global processesDone
   set foundOne 0
   set task "";
   while {$foundOne == 0 && $numDone < $np} {
     set otherProcess [getRandomPID [expr $np-$numDone-1]]
     set task [getTask [lindex $processesDone $otherProcess]]
     if {[llength $task] == 0} {
       set newProcessesDone {}
       for {set i 0} {$i < $np-$numDone} {incr i 1} {
         if {$i != $otherProcess} {
           lappend newProcessesDone [lindex$processesDone $i]
         }
       }
       set numDone [expr $numDone+1]
       set processesDone $newProcessesDone
     } else {
       set foundOne 1
     }
   }
   return $task
}
```

Results using 16 Processors



 # Tasks Completed *(pid vs # tasks)*



Duration *(pid v time)*