TOGETHER WE REACH

Introduction to Parallel Computing

Martin Čuma
Center for High Performance Computing
University of Utah
m.cuma@utah.edu

TOGETHER WE REACH

Overview

- Types of parallel computers.
- Parallel programming options.
- OpenMP, OpenACC, MPI
- Higher level languages
- Debugging, profiling and libraries
- Summary, further learning.

How to compute faster

- Faster CPU clock speed
  - – Higher voltage = more heat – not sustainable
- Work distribution
  - – Vectorization – process more than one value at a time
  - – Parallelization – spread work over multiple processing elements
  - – Specialization – application specific processors (ASIC), programmable logic (FPGA)

Computer architectures

Single processor:

- SISD – single instruction single data.

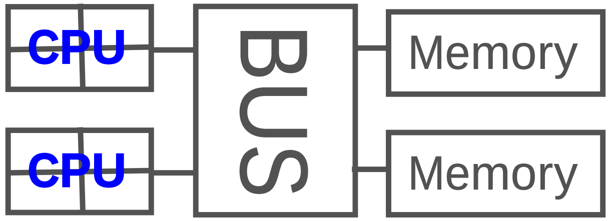
Multiple processors:

- SIMD - single instruction multiple data.
- MIMD – multiple instruction multiple data.
  - Shared Memory
  - Distributed Memory
- Current processors combine SIMD and MIMD
  - Multi-core CPUs w/ SIMD instructions (AVX, SSE)
  - GPUs with many cores and SIMT

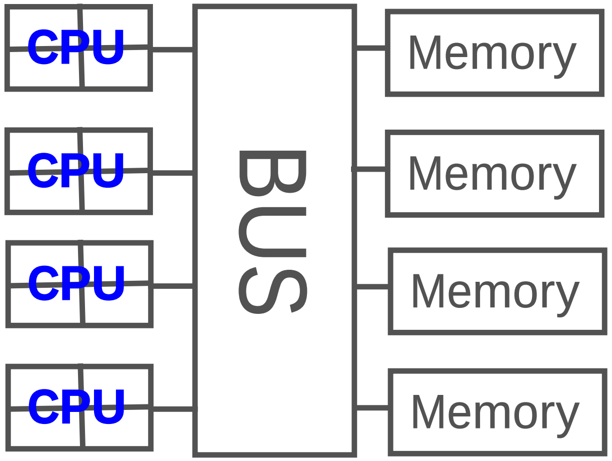
Shared memory

- All processors have access to local memory
- Simpler programming
- Concurrent memory access
- More specialized hardware
- Representatives:
  Linux clusters nodes 8-32 core GPU nodes

Dual quad-core node

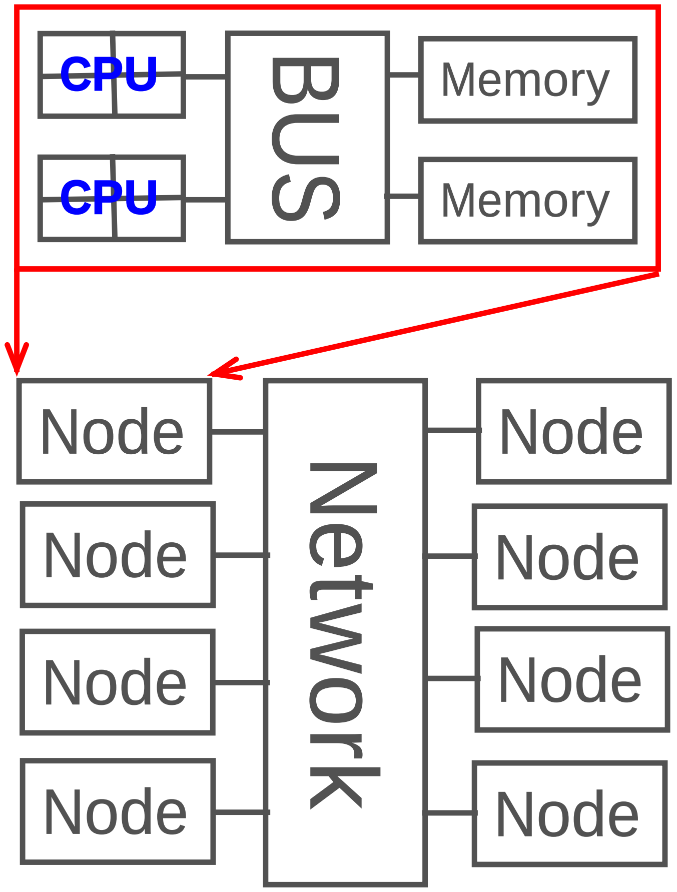


Many-CPU node (e.g. SGI)



Distributed memory

- Process has access only to its local memory
- Data between processes must be communicated
- More complex programming
- Cheap commodity hardware
- Representatives:
  Linux clusters



8 node cluster (64 cores)

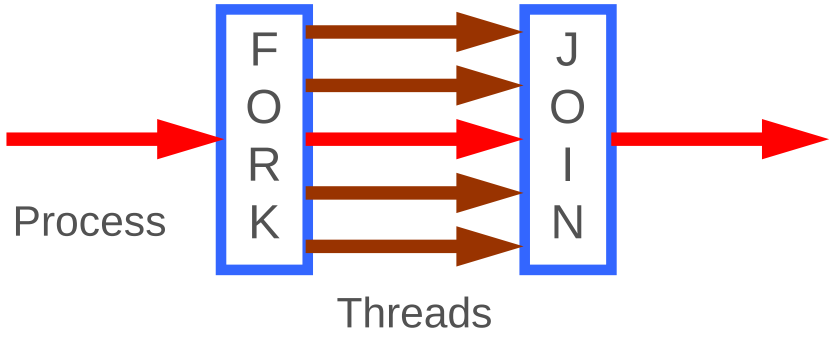
Ways of program execution

- Process (task)

  Entity that executes a program – has its own memory space, execution sequence, is independent from other processes

- Thread

  Has own execution sequence but shares memory space with the original process - a process may have many threads



Parallel programming options

Shared Memory

- Threads
  - POSIX Pthreads, OpenMP (CPU, MIC), OpenACC, CUDA (GPU)
- Processes
  - message passing, independent processes

Distributed Memory

- Independent processes
- Message passing libraries
  - General – MPI, PVM, language extensions (Co-array Fortran, UPC. …)

Higher level programming languages (Python, R, Matlab) do a combination of these approaches under the hood.

Parallel programming options hierarchy

- Instruction level (ILP)
  - Instruction pipelining, speculative execution, branch prediction, …
- Vector (SIMD)
- Multi-core/Multi-socket SMP
- Accelerators (GPU, MIC)
- FPGA, ASIC
- Distributed clusters

Compiler (not your problem)

OpenMP

OpenACC

Very specialized

MPI

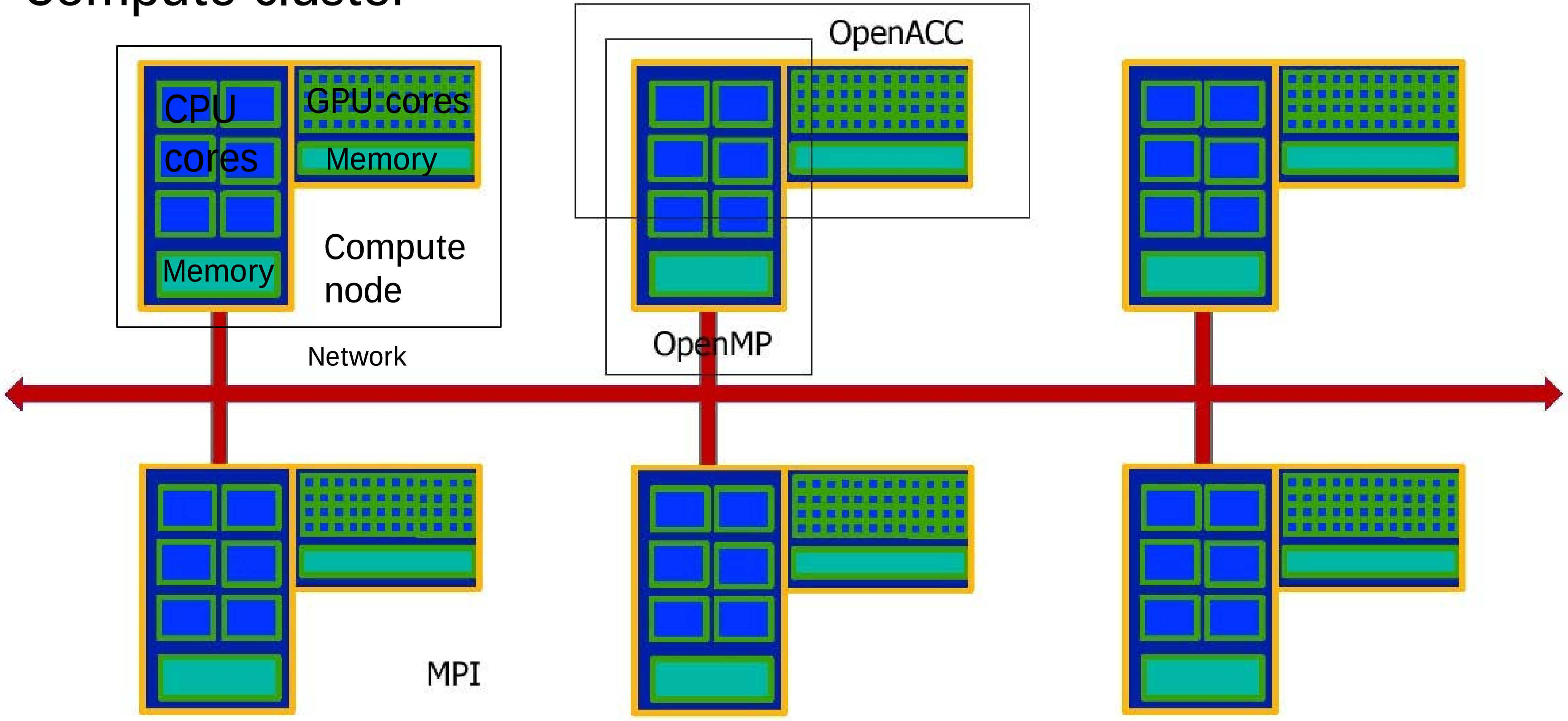
THE UNIVERSITY OF UTAH™

Mapping programming options to the hardware

TOGETHER WE REACH

Center for High-Performance Computing

Compute cluster



OpenMP basics

- Compiler directives to parallelize (CPU or GPU)
  - Fortran – source code comments

    `!$omp parallel/!$omp end parallel`
  - C/C++ - #pragmas

    `#pragma omp parallel`
- Small set of subroutines
- Degree of parallelism specification
  - `OMP_NUM_THREADS` or `omp_set_num_threads(INTEGER n)`

OpenACC Basics

- Compiler directives to offload to GPU
  - Fortran – source code comments

    `!$acc kernels/!$acc end kernels`
  - C/C++ - #pragmas

    `#pragma acc kernels`
- Small set of subroutines
- Degree of parallelism specification
  - `OMP_NUM_THREADS` or `omp_set_num_threads(INTEGER n)`

MPI Basics

- Communication library
- Language bindings:
  - C/C++ - `int MPI_Init(int argv, char* argc[])`
  - Fortran - `MPI_Init(INTEGER ierr)`
- Quite complex (100+ subroutines)

  but only small number used frequently
- User defined parallel distribution

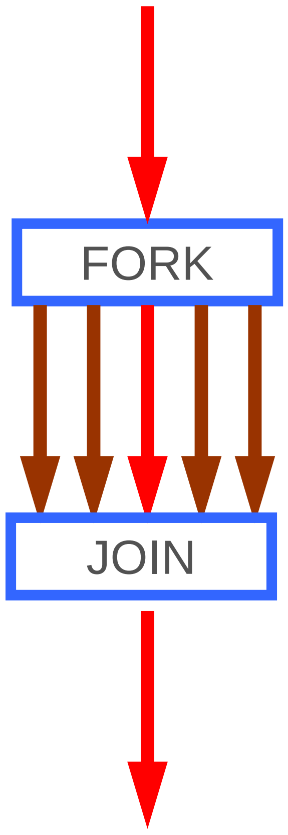
Program example

- saxpy – vector addition: $\bar{z} = a\bar{x} + \bar{y}$
- simple loop, no cross-dependence, easy to parallelize

```
subroutine saxpy_serial(z, a, x, y, n)
integer i, n
real z(n), a, x(n), y(n)

do i=1, n
  z(i) = a*x(i) + y(i)
enddo
return
```

OpenMP program example

```
subroutine saxpy_parallel_omp(z, a, x, y, n)
integer i, n
real z(n), a, x(n), y(n)

!$omp parallel do
do i=1, n
   z(i) = a*x(i) + y(i)
enddo
return

$ gcc –fopenmp saxpy.f
$ export OMP_NUM_THREADS=16
$ ./a.out
```



OpenMP caveats

- Data dependencies
  - Private (thread-local) variables

```
x = a(i)
b(i) = c + x
```

  - Flow dependence – rearrangement

```
a(i) = a(i+1) + x
```

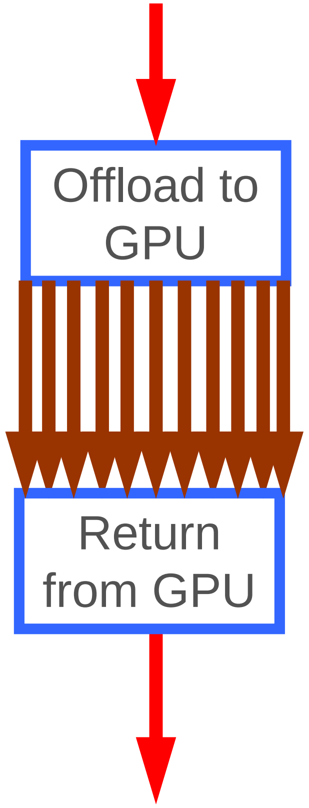
  - Reduction (sum over threads)

```
x += a(i)
```

- Scheduling
  - What runs on what thread – schedule, task,…
- Advanced features
  - Thread affinity (to CPU core)
  - Vectorization
  - Accelerator offload

OpenACC program example

```
subroutine saxpy_parallel_oacc(z, a, x, y, n)
integer i, n
real z(n), a, x(n), y(n)

!$acc kernels datain(x,y) dataout(z)
do i=1, n
  z(i) = a*x(i) + y(i)
enddo
return

$ pgcc -acc -Minfo=accel saxpy.f
$ pgaccelinfo To verify that GPU is available
$ ./a.out
```



OpenACC caveats

- Data dependencies (Like in OpenMP)
- Data locality
  - Transfers from host to GPU and back take time – need to minimize them

```
#pragma acc data [copyin, copyout, create,...]
```

- Parallel regions
  - More explicit execution control (warps, threads)

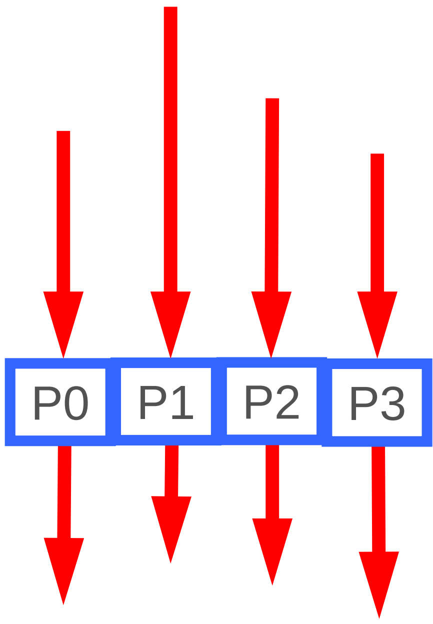
```
#pragma acc parallel
```

- Procedure calls
  - If procedure is executed on the GPU

```
#pragma acc routine
```

MPI program example

```
subroutine saxpy_parallel_mpi(z, a, x, y, n)
integer i, n, ierr, my_rank, nodes, i_st, i_end
real z(n), a, x(n), y(n)

call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD,my_rank,ierr)
call MPI_Comm_size(MPI_COMM_WORLD,nodes,ierr)
i_st = n/nodes*my_rank+1
i_end = n/nodes*(my_rank+1)

do i=i_st, i_end
   z(i) = a*x(i) + y(i)
enddo
call MPI_Finalize(ierr)
return
```

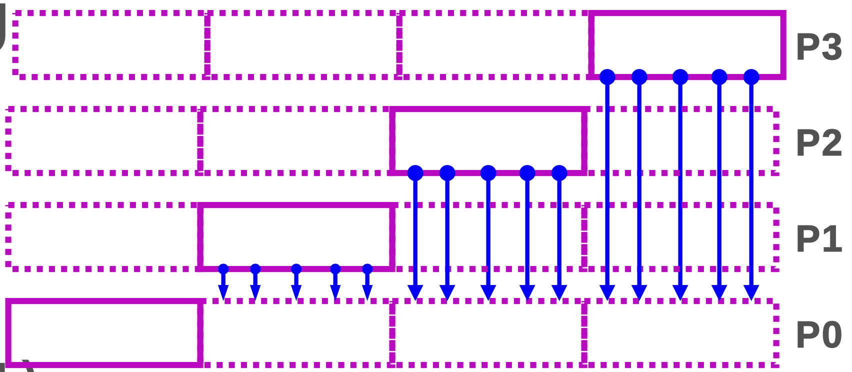


z(i) operation on 4 processes (tasks)

z(1 … n/4)	z(n/4+1 … 2*n/4)	z(2*n/4+1 … 3*n/4)	z(3*n/4+1 … n)

MPI program example

- Result on the first CPU



```
include "mpif.h"
integer status(MPI_STATUS_SIZE)
if (my_rank .eq. 0 ) then
   do j = 1, nodes-1
      do i= n/nodes*j+1, n/nodes*(j+1)
         call MPI_Recv(z(i),1,MPI_REAL,j,0,MPI_COMM_WORLD,
&        status,ierr)
      enddo
   enddo
else
   do i=i_st, i_end
      call MPI_Send(z(i),1,MPI_REAL,0,0,MPI_COMM_WORLD,ierr)
   enddo
endif
```

Data | Count

Sender

Recipient

MPI program example

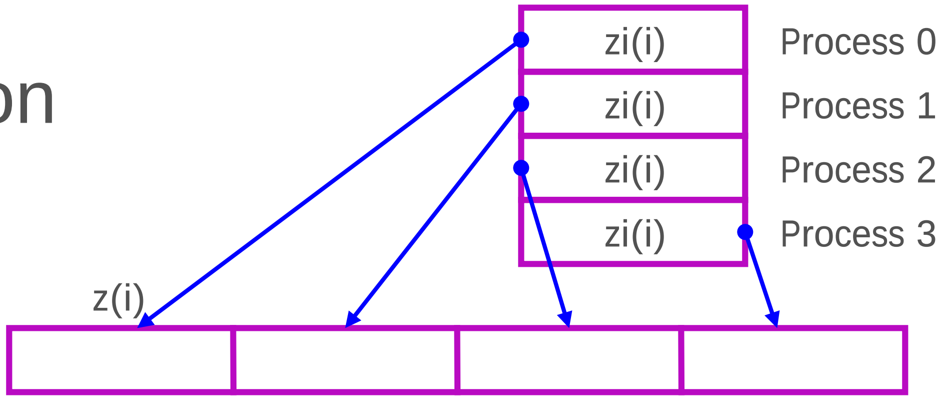
- Collective communication

```
real zi(n)
j = 1
do i=i_st, i_end
   zi(j) = a*x(i) + y(i)
   j = j +1
enddo
call MPI_Gather(zi,n/nodes,MPI_REAL,z,n/nodes,MPI_REAL,
&                0,MPI_COMM_WORLD,ierr)
```

Send data

Receive data

Root process



- Result on all nodes

```
call MPI_AllGather(zi,n/nodes,MPI_REAL,z,n/nodes,
&                MPI_REAL,MPI_COMM_WORLD,ierr)
```

No root process

MPI caveats

- Explicit task based parallelism
  - manual work distribution
  - task communication and synchronization
- Communication patterns
  - due to different data distribution
- Many advanced features
  - blocking vs. non-blocking communication
  - derived data types
  - topologies
  - …

broadcast
reduction
gather/scatter
...

MPI distributions

- Different networks
  - Ethernet
  - InfiniBand
  - Intel OmniPath
  - most MPI distributions now come with multiple networks support
- Several distributions follow the MPI standard
  - MPICH, MVAPICH2
  - Intel MPI, Cray MPI,…
  - OpenMPI
  - Ensure that build and run is done with the same distribution (ABI compatibility)

But wait, my program is not in C or Fortran

Interpreted languages are popular

- Matlab, Python, R

Each has some sort of parallel support, but most likely it will not perform as well as using OpenMP or MPI with C/Fortran.

Try to parallelize (and optimize) your Matlab/Python/R code and if it’s still not enough consider rewriting in C++ or Fortran.

Cluster running options for Matlab, Python, R

- Using parallelization in the program run through interactive or batch job
  - multi-threading and/or multi-processing packages (parfor, mpi4py, R parallel, Rmpi, …)
- Using built in job submission
  - Matlab Parallel Server, rslurm, python Dask, snakemake
- Independent calculations in parallel
  - launching concurrent calculations in a job

Matlab

Threads

- Built in Matlab functions. Vector/matrix operations threaded (and vectorized) through Intel MKL library, many other functions also threaded

Tasks (processes)

- *Parallel Computing Toolbox* allows for task based parallelism
- *Parallel Server* can distribute tasks to multiple nodes
- Great for independent calculations, when communication is needed uses MPI under the hood

https://www.chpc.utah.edu/documentation/software/matlab.php

Matlab tasks

- Parallel program

```
function t = parallel_example
parfor idx = 1:16
  A(idx) = idx;
end
```

Will launch loop iterations on multiple workers

- Parallel worker pool on a single machine

```
poolobj=parpool('local',8);
parallel_example;
delete(poolobj);
```

Starts multiple workers pool

- Parallel pool on a cluster

```
c = parcluster;
c.AdditionalProperties.QueueName = 'kingspeak';
...
j = c.batch(@parallel_example, 1, {}, 'Pool', 4);
j.State
j.fetchOutputs{:}
```

Submits cluster job

Matlab examples

- Parallel worker pool on a single node
  - best run from a SLURM job loop_parallel_onenode.m, run_matlab_onenode.m, run_matlab_onenode.slr
  - https://git.io/CHPC-Intro-to-Parallel-Computing-Matlab
  - `sbatch run_matlab_onenode.slr`
- Parallel worker pool on a multiple nodes
  - must run from inside of Matlab
  - start Matlab on interactive node inside of a FastX session

```
ml matlab
matlab &
```

  - loop_parallel.m, parallel_multinode.m

```
parallel_multinode
```

R

Threads

- Under the hood threading with specially built (or Microsoft) R for vector/matrix operations using MKL
- *parallel* R library

Tasks (processes)

- *parallel* R library (uses *multicore* for shared and *snow* for distributed parallelism)
- Parallelized **apply* functions, e.g. *mclapply*
- *Rmpi* library provides MPI like functionality
- Many people run multiple independent R instances in parallel

https://www.chpc.utah.edu/documentation/software/r-language.php

Parallel R on a cluster

- Load libraries

```
library(parallel)
library(foreach)
library(doParallel)
```

`hostlist.txt` comes from a job script

```
srun -n $SLURM_NTASKS > hostlist.txt
```

- Start R cluster

```
hostlist <- paste(unlist(read.delim(file="hostlist.txt",
header=F, sep =" ")))
cl <- makeCluster(hostlist)
registerDoParallel(cl)
clusterEvalQ(cl,.libPaths("/uufs/chpc.utah.edu/sys/installdir/
RLibs/3.5.2i"))
```

this is only needed if running on multiple nodes

- Run parallel loop

```
r <- foreach(icount(trials), .combine=rbind) %dopar% {}
```

- Stop R cluster

```
stopCluster(cl)
```

R examples

- Parallel R on one node
  - best run from a SLURM job
    parallel-onenode-iris.R, R-parallel-onenode-iris.slr
  - https://git.io/CHPC-Intro-to-Parallel-Computing-R
  - `sbatch R-parallel-onenode-iris.slr`
- Parallel R multiple nodes
  - must specify list of nodes where R workers run
    parallel-multinode-iris.R, R-parallel-multinode-iris.slr-
  - `sbatch R-parallel-onenode-iris.slr`
- Submit SLURM job directly from R - rslurm
  - SLURM-aware apply function, some issues with results collection
  - rslurm-example.R

Python

Threads

- No threads in Python code because of GIL (Global Intepreter Lock)
- C/Fortran functions can be threaded (e.g. *NumPy* - Anaconda)

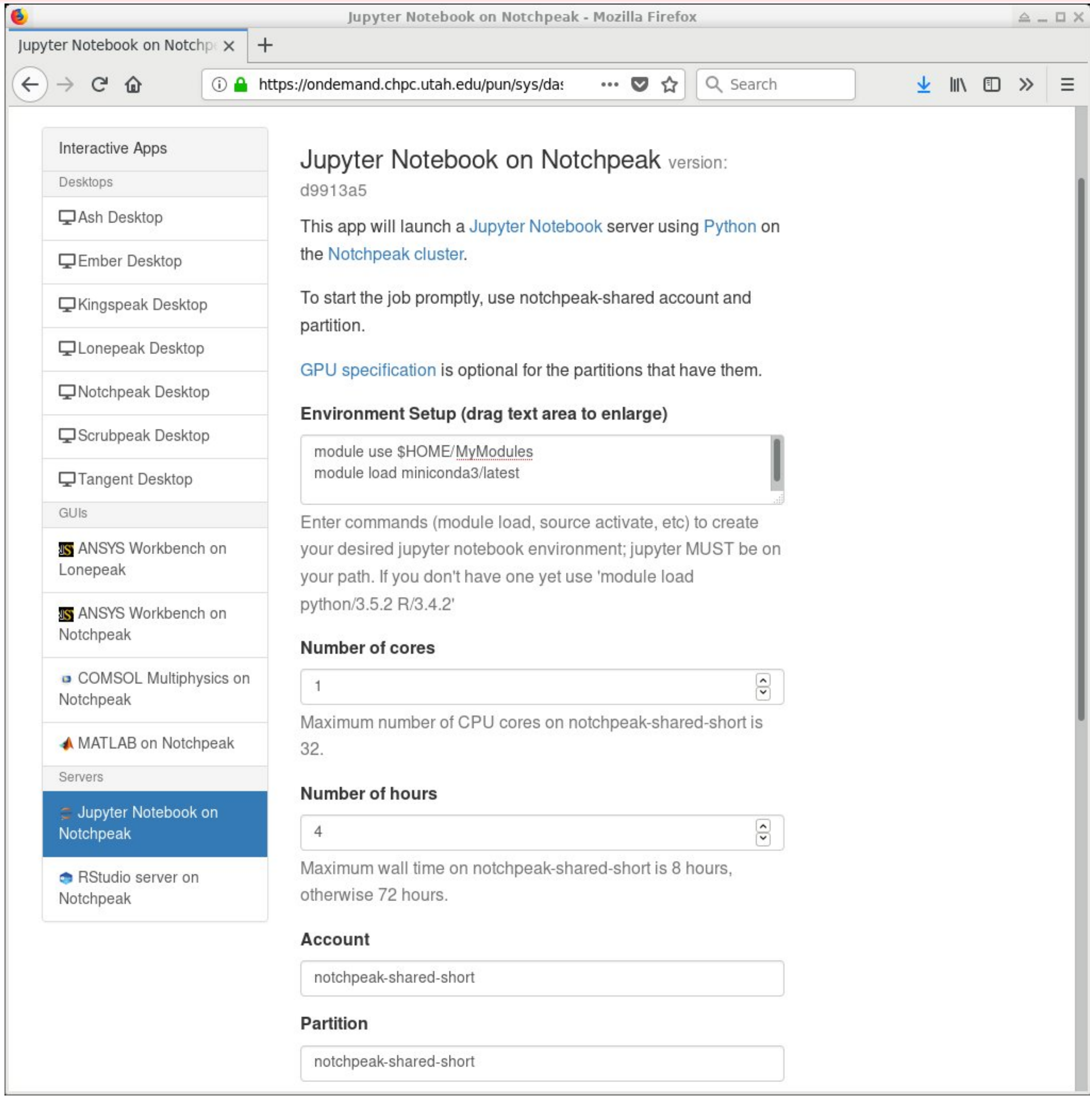
Tasks (processes)

- Several libraries that use MPI under the hood, most popular is *mpi4py*
- More-less MPI function compatibility, but slower communication because of the extra overhead
- Also many other data-parallel libraries, e.g. *Dask*

https://www.chpc.utah.edu/documentation/software/python.php

Python - Jupyter

- Several options listed at https://www.chpc.utah.edu/documentation/software/jupyterhub.php
- The easiest is to use Open OnDemand


Jupyter Notebook on Notchpeak - Mozilla Firefox
https://ondemand.chpc.utah.edu/pun/sys/da:
Interactive Apps
Desktops
Ash Desktop
Ember Desktop
Kingspeak Desktop
Lonepeak Desktop
Notchpeak Desktop
Scrubpeak Desktop
Tangent Desktop
GUIs
ANSYS Workbench on Lonepeak
ANSYS Workbench on Notchpeak
COMSOL Multiphysics on Notchpeak
MATLAB on Notchpeak
Servers
Jupyter Notebook on Notchpeak
RStudio server on Notchpeak

Jupyter Notebook on Notchpeak version: d9913a5

This app will launch a Jupyter Notebook server using Python on the Notchpeak cluster.

To start the job promptly, use notchpeak-shared account and partition.

GPU specification is optional for the partitions that have them.

Environment Setup (drag text area to enlarge)

```
module use $HOME/MyModules
module load miniconda3/latest
```

Enter commands (module load, source activate, etc) to create your desired jupyter notebook environment; jupyter MUST be on your path. If you don't have one yet use 'module load python/3.5.2 R/3.4.2'

Number of cores: 1

Maximum number of CPU cores on notchpeak-shared-short is 32.

Number of hours: 4

Maximum wall time on notchpeak-shared-short is 8 hours, otherwise 72 hours.

Account: notchpeak-shared-short

Partition: notchpeak-shared-short

Python tasks

- Our personal favorite is to ignore all the Python parallel efforts, divide the data into independent parts and run multiple Python processes on parts of the data concurrently
- Only works if data can be split
- Use various approaches for independent parallel calculations listed at https://www.chpc.utah.edu/documentation/software/serial-jobs.php
- More on this later

Python- Dask

- With relatively small effort one can use Dask
- Install Miniconda

```
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash ./Miniconda3-latest-Linux-x86_64.sh -b -p $HOME/software/pkg/miniconda3
mkdir -p $HOME/MyModules/miniconda3
cp /uufs/chpc.utah.edu/sys/installdir/python/modules/miniconda3/latest.lua $HOME/MyModules/miniconda3
```

- Use own miniconda and install Jupyter and Dask

```
module use $HOME/MyModules
module load miniconda3/latest
conda install jupyter dask
```

- Start Open OnDemand Jupyter notebook
  - log into ondemand.chpc.utah.edu with CHPC credentials

Python- Dask

- Go to Interactive Apps - Jupyter Notebook on notchpeak
- In the Environment Setup text box, put:

```
module use $HOME/MyModules
module load miniconda3/latest
```

- Use notchpeak-shared-short for account and partition, and select your choice of CPU cores and walltime hours (within the listed limits). Then hit Launch to submit the job.
- Once the job starts, hit the blue Connect to Jupyter button
- Open one of the following notebooks: dask_embarrass.ipynb, dask_slurmcluster.ipynb, dask_slurm_xarray.ipynb
- DASK also allows to submit jobs to SLURM (last 2 examples)

Independent calculations

- Different approaches based on the nature of the calculations
  - Runtime length, variability, number of calculations
- Similar runtime, small calculation count
  - Shell script in a SLURM job

```
#!/bin/bash
for (( i=0; i < $SLURM_NTASKS ; i++ )); do
  /path_to/myprogram $i &
done
wait
```

  - srun –multi-prog

```
srun --multi-prog my.conf
cat my.conf
0-11 ./example.sh %t
```

https://www.chpc.utah.edu/documentation/software/serial-jobs.php

Variable runtime

- Mini-scheduler inside of a job
  - to launch calculations till all are done
  - GNU Parallel - https://www.gnu.org/software/parallel/
  - TACC Launcher - https://www.tacc.utexas.edu/research-development/tacc-software/the-launcher
  - CHPC Submit - https://www.chpc.utah.edu/documentation/software/serial-jobs.php#submit
- Workflow managers
  - More on this later
  - Makeflow, Swift, Snakemake, Pegasus
- Distributed computing resources
  - Open Science Grid - https://opensciencegrid.org/

Debuggers

- Useful for finding bugs in programs
- Several free
  - `gdb` – GNU, text based, limited parallel
  - `ddd` – graphical frontend for gdb
- Commercial that come with compilers
  - `pgdbg` – PGI, graphical, parallel but not intuitive
  - `pathdb, idb` – Pathscale, Intel, text based
- Specialized commercial
  - `totalview` – graphical, parallel, CHPC has a license
  - **`ddt`** - Distributed Debugging Tool
  - **Intel Inspector XE** – memory and threading error checker
- How to use:
- `http://www.chpc.utah.edu/docs/manuals/software/par_devel.html`

Debuggers - parallel

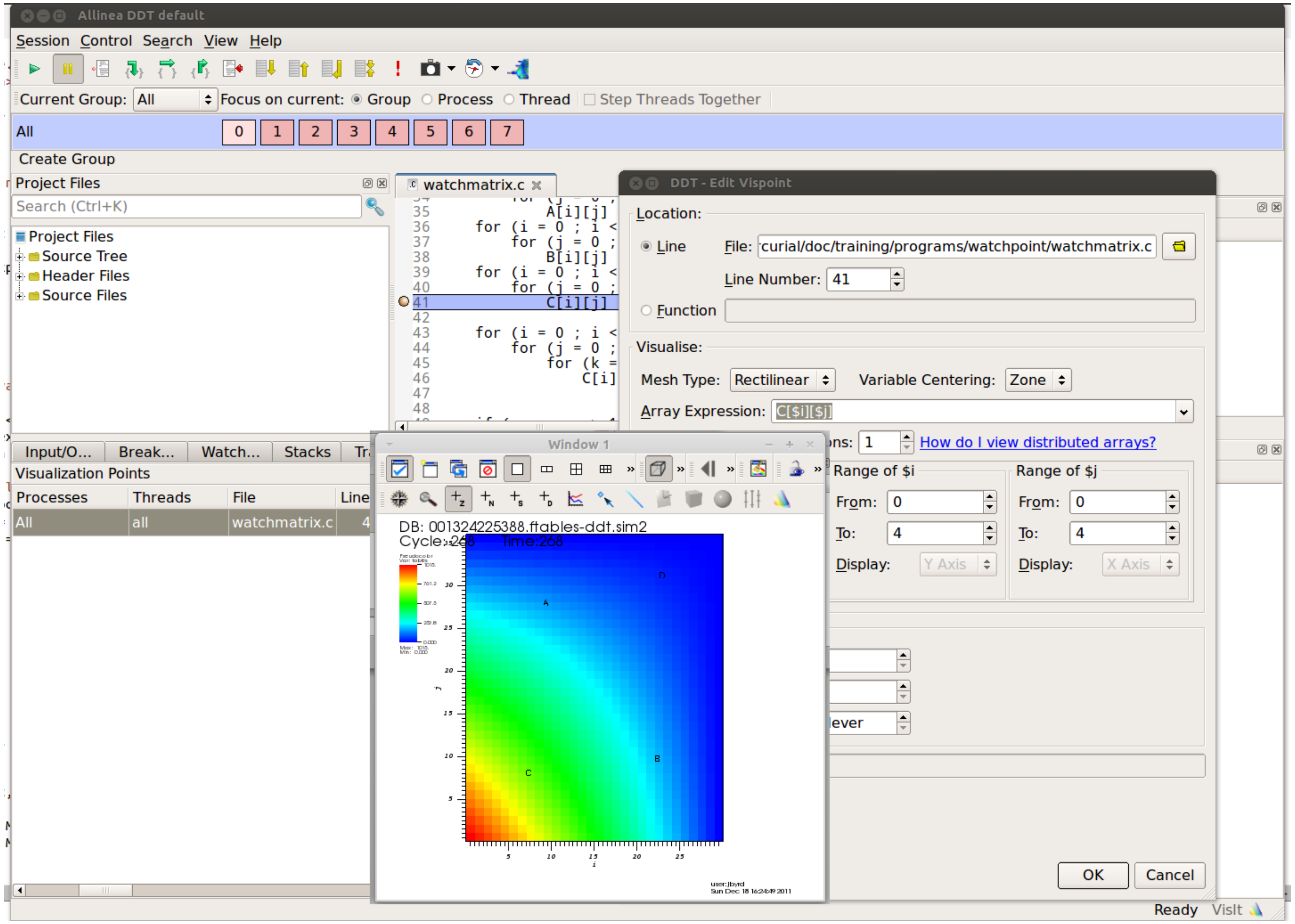
- Parallel debugging more complex due to interaction between processes
- DDT is the debugger of choice at CHPC
- Expensive but academia get discount
- How to run it:
  - compile with –g flag
  - run `ddt` command
  - fill in information about executable, parallelism, …
- Details:

`https://www.chpc.utah.edu/documentation/software/debugging.php`

- Further information

`https://www.allinea.com/products/ddt`

Debuggers – parallel



Allinea DDT default

Session Control Search View Help

Current Group: All | Focus on current: ● Group ○ Process ○ Thread ☐ Step Threads Together

All | 0 1 2 3 4 5 6 7

Create Group

Project Files
- Source Tree
- Header Files
- Source Files

watchmatrix.c

```
35            A[i][j]
36      for (i = 0 ; i <
37          for (j = 0 ;
38              B[i][j]
39      for (i = 0 ; i <
40          for (j = 0 ;
41              C[i][j]
42
43      for (i = 0 ; i <
44          for (j = 0 ;
45              for (k =
46                  C[i]
47
48
```

Input/O... | Break... | Watch... | Stacks

Visualization Points

Processes	Threads	File	Line
All	all	watchmatrix.c	4

DDT - Edit Vispoint

Location:

● Line File: curial/doc/training/programs/watchpoint/watchmatrix.c

Line Number: 41

○ Function

Visualise:

Mesh Type: Rectilinear Variable Centering: Zone

Array Expression: C[$i][$j]

ns: 1 How do I view distributed arrays?

Range of $i		Range of $j	
From:	0	From:	0
To:	4	To:	4
Display:	Y Axis	Display:	X Axis

lever

OK Cancel

Ready VisIt

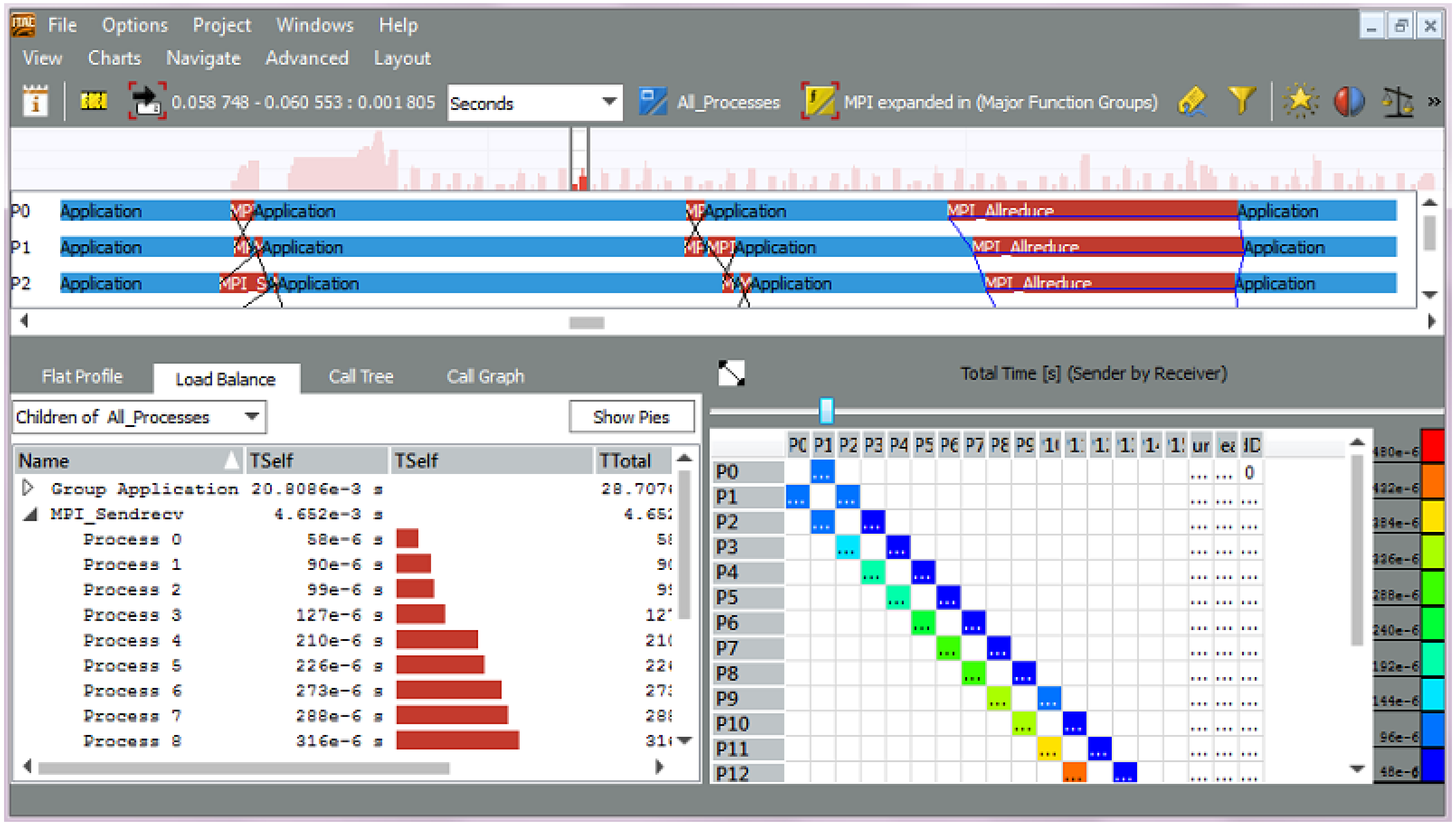
Window 1

DB: 001324225388.ftables-ddt.sim2
Cycle: 268 Time:268

Profilers

- Measure performance of the code
- Serial profiling
  - discover inefficient programming
  - computer architecture slowdowns
  - compiler optimizations evaluation
  - gprof, pgprof, pathopt2, Intel tools
- Parallel profiling
  - target is inefficient communication
  - **Intel Trace Collector and Analyzer, AdvisorXE, VTune**

Profilers - parallel



Name	TSelf	TSelf	TTotal
▷ Group Application	20.8086e-3 s		28.707
◢ MPI_Sendrecv	4.652e-3 s		4.65
Process 0	58e-6 s	🟥	5
Process 1	90e-6 s	🟥	9
Process 2	99e-6 s	🟥	9
Process 3	127e-6 s	🟥	12
Process 4	210e-6 s	🟥	21
Process 5	226e-6 s	🟥	22
Process 6	273e-6 s	🟥	27
Process 7	288e-6 s	🟥	28
Process 8	316e-6 s	🟥	31

Libraries

- Use libraries for common operationsSerial
  - BLAS, LAPACK – linear algebra routines
  - MKL, ACML – hardware vendor libraries
- Parallel
  - ScaLAPACK, PETSc, FFTW
  - MKL – dense and sparse matrices
- Design a new code around existing library
  - PETSc, Trilinos,…

Single executable across desktops and clusters

- MPICH, MVAPICH2 and Intel MPI are cross-compatible using the same ABI
  - Can e.g. compile with MPICH on a desktop, and then run on the cluster using MVAPICH2 and InfiniBand
- Intel and PGI compilers allow to build "unified binary" with optimizations for different CPU platforms
  - But in reality it only works well under Intel compilers
- On a desktop

```
module load intel mpich
mpicc –axCORE-AVX512,CORE-AVX2,AVX program.c –o program.exe
mpirun –np 4 ./program.exe
```

- On a cluster

```
srun –N 2 –n 24 ...
module load intel mvapich2
mpirun –np $SLURM_NTASKS ./program.exe
```

- `https://www.chpc.utah.edu/documentation/software/single-executable.php`

Summary

- Shared vs. Distributed memory parallelism
- OpenMP, OpenACC and MPI for low level parallelism
- Different approaches for higher level languages
- Many ways to run independent calculations in parallel
- There are tools for debugging, profiling

To learn more

- CHPC lectures
  - https://www.chpc.utah.edu/presentations/index.php
- XSEDE HPC Summer Boot Camp
  - OpenMP, OpenACC, MPI
  - https://www.youtube.com/XSEDETraining
- Petascale Computing Institute
  - Wide range of parallel programming topics
  - videos at https://bluewaters.ncsa.illinois.edu/bw-petascale-computing-2019/agenda
- XSEDE online training
  - https://www.xsede.org/web/xup/online-training