Using MPI @ FASRC

Plamen G Krastev, PhD
Computational Scientist and Research Consultant

February 2019
AC290

Objectives

- To inform you of available computing resources
- To provide the basic knowledge required for running your parallel MPI applications efficiently on the Odyssey cluster

Outline

- Introduction to available resources
- HPC and MPI basics
- Exercises
- Summary

Introduction to available resources

Research Computing Resources

- 78k cores HPC
- 1M+ CUDA cores
- 35PB of storage (Lustre, XFS/NFS, Isilon, gluster, Ceph, GPFS)
- 400+ virtual machines (KVM/OpenNebula)
- 2MW of research computing equipment in 3 data centers
- 24 Research Computing staff members in 5 groups
  - ARCS: Advanced Research Computing Support
  - DSRF: Data Science & Research Facilitation
  - Infrastructure as Code (OpenNebula/VMs, Containers)
  - Systems Engineering & Data Center Operations
  - Research Software Engineering
- Supporting 600 research groups and 5000+ users

Cluster Basics

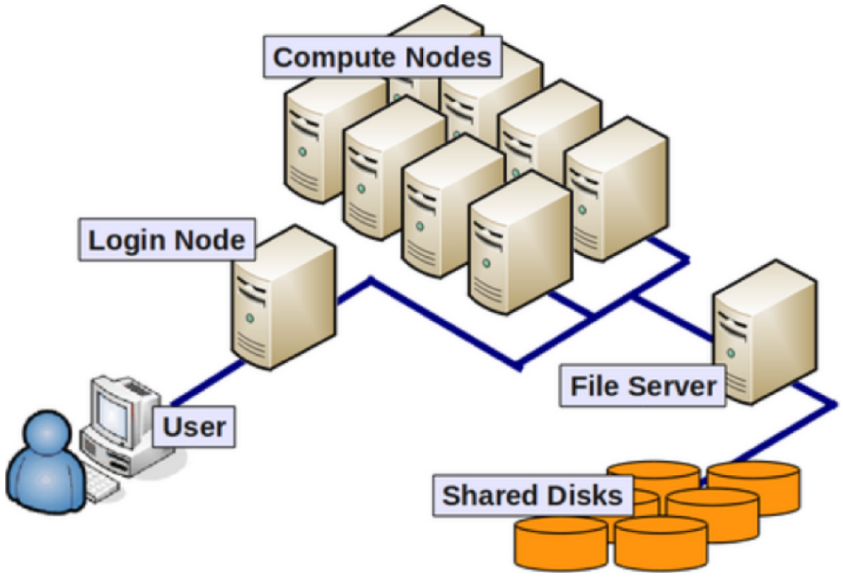
Login Nodes:

- prepare input and stage your calculations
- interact with the scheduler
- no computationally expensive processes allowed

http://rc.fas.harvard.edu/resources/access-and-login

Compute nodes:

- computational resource monitored and managed by the SLURM scheduler.
- resources are organized in partitions
- access only via scheduler
- servers are interconnected by Infiniband fabric (high throughput , low latency)



Storage:

	Home Folders	Local Scratch	Global Scratch
mount point	$HOME	/scratch	/n/regal/cs205/
size limit	100G	~ 270G/node	quota 50T/group
backup / retention	Hourly snapshots	only available during job	**90days retention** no backup
performance	Not suitable for intense I/O	Suited for small file I/O intensive jobs	Suited for large file I/O intensive jobs

Job Scheduler - SLURM

- SLURM = Simple Linux Utility for Resource Management
  User tasks (jobs) on the cluster are controlled by SLURM and isolated in cgroups so that users cannot interfere with other jobs or exceed their resource request (cores, memory, time)
- Basic SLURM commands:
  - sbatch: submit a batch job script
    ```
    > sbatch [ options for resource request ] myscript
    ```
  - srun: submit an interactive test job
    ```
    > srun --pty [ options for resource request ] /bin/bash
    ```
  - squeue: contact slurmctld for currently running jobs
    ```
    > squeue
    ```
  - sacct: contact slurmdb for accounting stats after job ends
    ```
    > sacct
    ```
  - scancel: cancel a job(s)
    ```
    > scancel some_job_ID
    ```

https://rc.fas.harvard.edu/resources/documentation/convenient-slurm-commands
https://rc.fas.harvard.edu/resources/running-jobs

SLURM Partitions

https://rc.fas.harvard.edu/resources/running-jobs/#Slurm_partitions

Partition	Nodes	Cores per Node	Core Types	Mem per Node (GB)	Time Limit	Max Jobs	Max Cores	MPI Suitable?	GPU Capable?
shared	456	32	Intel	128	7 days	none*	none*	Yes	No
general	133	64	AMD	256	7 days	none*	none*	Yes	No
serial_requeue	varies	varies	AMD/Intel	varies	7 days	none*	none*	No	Yes*
gpu_requeue	varies	varies	Intel	varies	7 days	none*	none*	No	Yes
gpu	1	24	Intel	30	none*	none*	none*	No	Yes (8 K20Xm)
test**	8	32	Intel	128	8 Hours	5	64	No	No
bigmem	6	64	AMD	512	none*	none*	none*	No	No
unrestricted	8	64	AMD	256	none*	none*	none*	Yes*	No
PI/Lab nodes	varies	varies	varies	varies	none	none	none	varies	varies
holyseasgpu	**13**	**48**	**Intel**	**96**	**none**	**none**	**none**	**yes**	**yes (2x K40m)**
seas_dgx	4	40	Intel	512	3 days*	none	none	yes	yes (8x V100)
test7	**4**	**varies**	**varies**	**varies**	**8 Hours**	**Note : Test partition with CentOS 7**			

Software on Odyssey

- CentOS 7
- Hundreds of software packages – compilers, numeric libraries, development packages, visualization tools, and much more
  - https://portal.rc.fas.harvard.edu/apps/modules
- Harvard modified environment module system LMOD
  - https://www.rc.fas.harvard.edu/resources/documentation/software-on-odyssey
  - Dynamically change user environment
  - Software is loaded incrementally

```
module load intel/17.0.4-fasrc01    # Loads compiler module
module load openmpi/3.1.1-fasrc01   # Loads OpenMPI module
module avail                        # Lists available modules
module list                         # Lists loaded modules
module purge                        # Unloads ALL modules
module-query openmpi                # Finds modules
module-query openmpi/3.1.1-fasrc01 # Gives more information
```

XSEDE

- XSEDE = Extreme Science and Engineering Development Environment
- A single virtual system that scientists can use to interactively share computing resources, data and expertise
- https://www.xsede.org

Harvard University XSEDE campus champions:

- Plamen Krastev - plamenkrastev@fas.harvard.edu
- Francesco Pontiggia - pontiggia@g.harvard.edu

HPC & MPI basics

What is High Performance Computing?

Pravetz 82 and 8M, Bulgarian Apple clones
Image credit: flickr

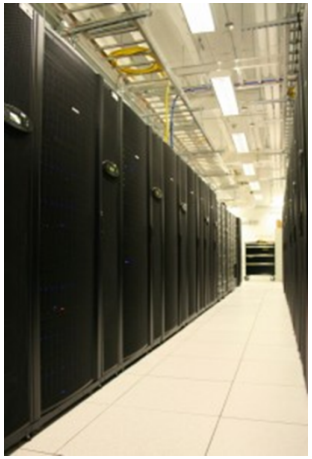
What is High Performance Computing?

Pravetz 82 and 8M, Bulgarian Apple clones
Image credit: flickr

What is High Performance Computing?

Summit: ORNL

Sierra: LLNL



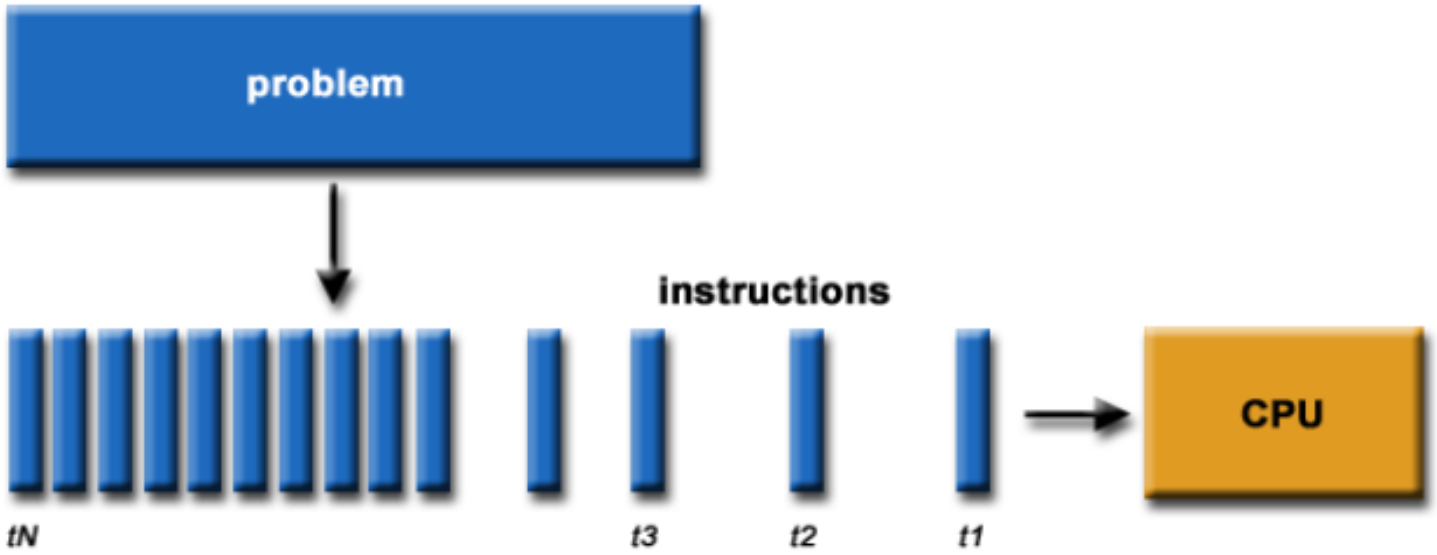
Odyssey: Harvard

Using the world's fastest and largest computers to solve large and complex problems.

Serial Computation

Traditionally software has been written for serial computations:

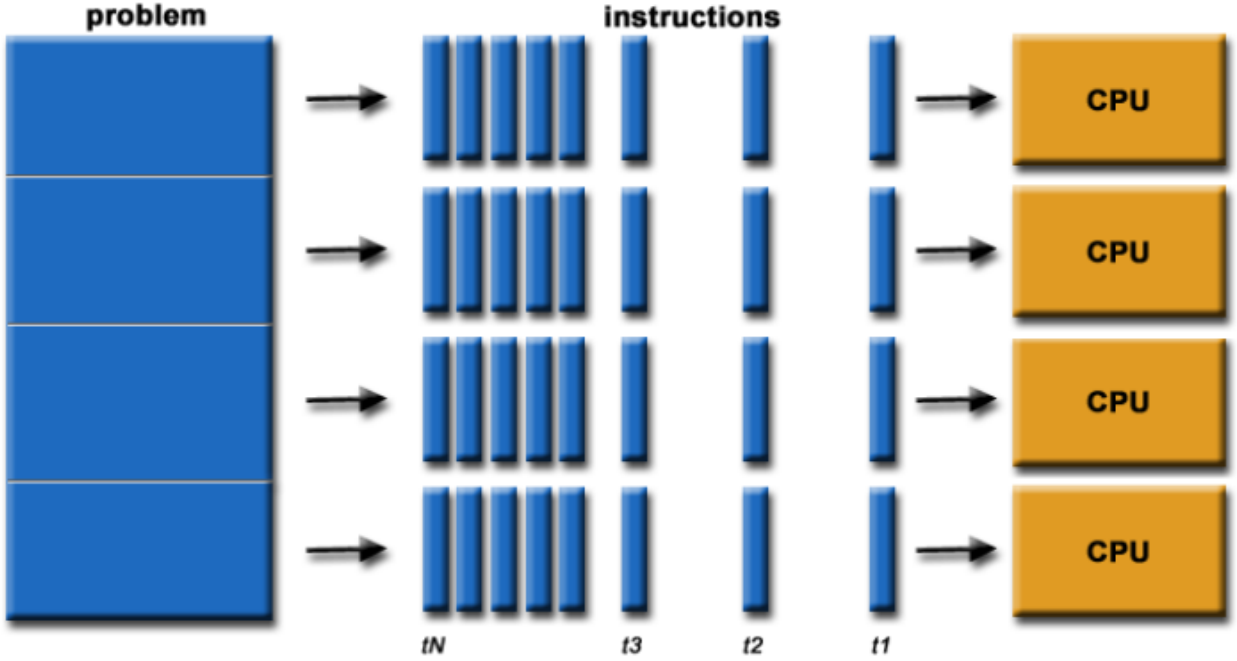
- To be run on a single computer having a single Central Processing Unit (CPU)
- A problem is broken into a discrete set of instructions
- Instructions are executed one after another
- Only one instruction can be executed at any moment in time



Parallel Computation

In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs



Why Use HPC?

Major reasons:

Save time and/or money: In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.

Solve larger / more complex problems: Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.

Provide concurrency: A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.

Use of non-local resources: Using compute resources on a wide area network, or even the Internet when local compute resources are scarce.

Applications of HPC (not a complete list)

- Atmosphere, Earth, Environment, Space Weather
- Physics / Astrophysics – applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Sciences
- Geology, Seismology
- Mechanical and Aerospace Engineering
- Electrical Engineering, Circuit Design, Microelectronics
- Computer Science, Mathematics

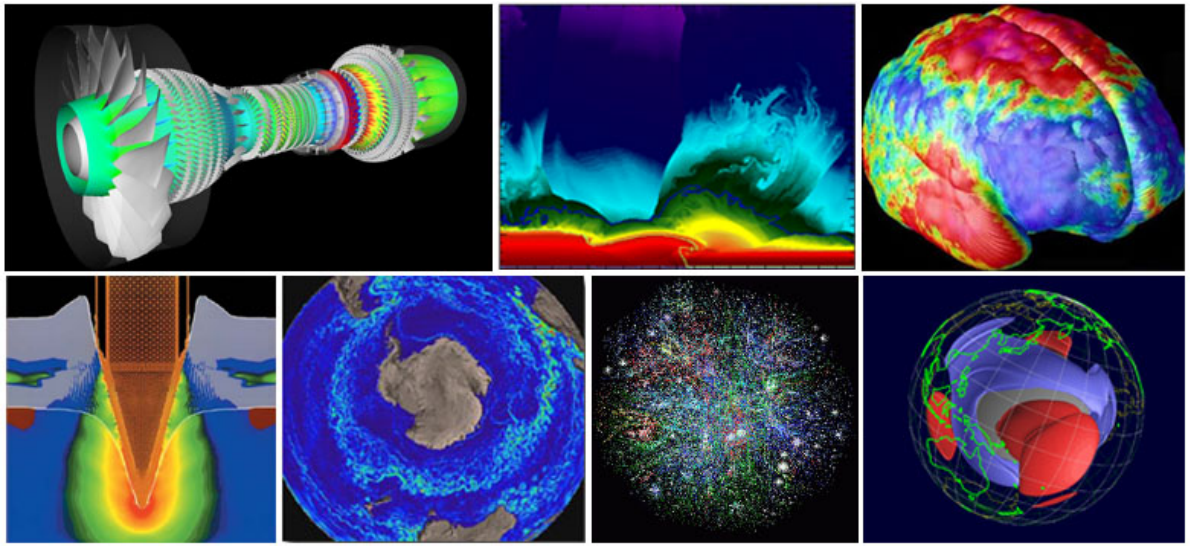
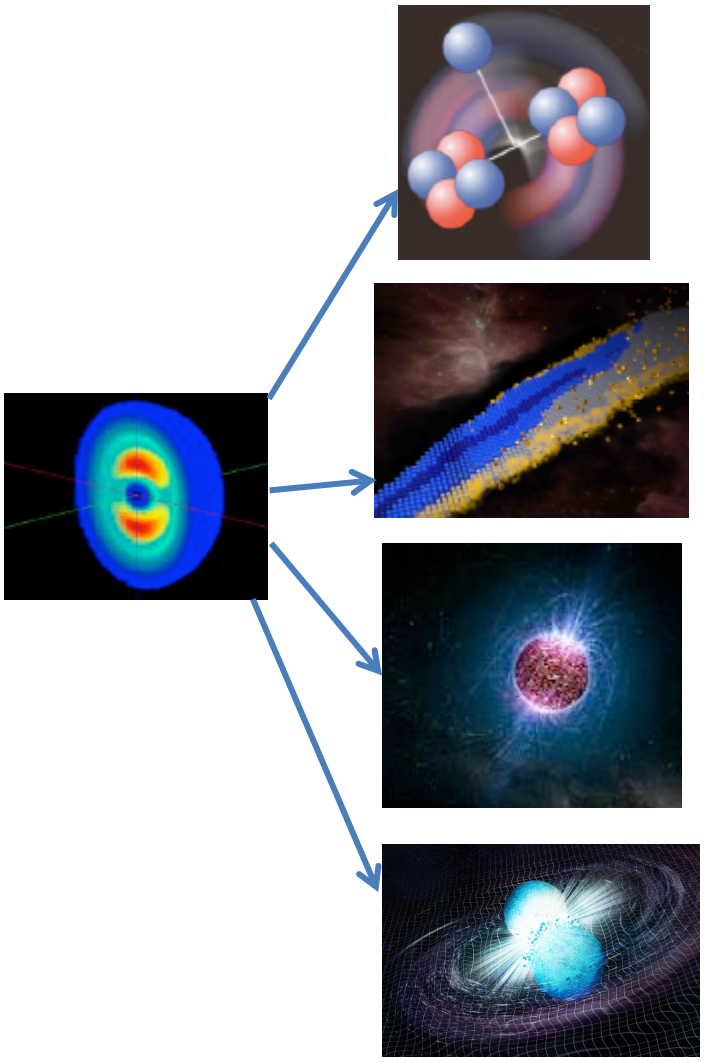


Image credit: LLNL

Computational Nuclear Physics and Astrophysics



Building complex strictures and phenomena – including nuclei, heavy-ion collisions, nucleosynthesis, neutron stars (NSs) & NSs mergers, supernova explosions, gravitational waves (GWs) and more – starting from fundamental interactions between nucleons (and quarks)

$$H = \sum_i T_i + \sum_{i<j} V_{ij} + \sum_{i<j<k} V_{ijk}$$

V_{ij} fit to many NN experimental data
V_{ijk} has O(5) parameters, typically fit to few-nucleon systems

Computational Methods:

- Quantum Monte Carlo
- Configuration Interaction (CI, shell model)
- Coupled Cluster (CC)
- Density Functional Theory (DFT)
- Greens Functions
- Relativistic Mean Field (RMF)
- Brueckner-Hartree-Fock (BHF) + relativistic extension (DBHF)
- …

Large Scale Shell-Model Calculations

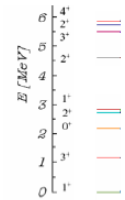
Typical eigenvalue problem $H\Psi_i = E_i\Psi_i$

Construct many-body bases states $|\phi_i\rangle$ so that $\Psi_i = \sum_n C_{in}\phi_n$

Calculate Hamiltonian matrix elements $H_{ij} = \langle\phi_j|H|\phi_i\rangle$

Diagonalize to obtain eigenvalues and eigenstates

$$\begin{pmatrix} H_{11} & H_{12} & \cdots & H_{1N} \\ \vdots & H_{22} & & \vdots \\ \vdots & & \ddots & \vdots \\ H_{N1} & & & H_{NN} \end{pmatrix} \rightarrow$$

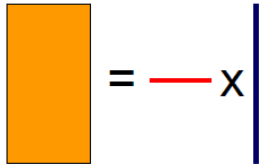


Shell-Model Code:

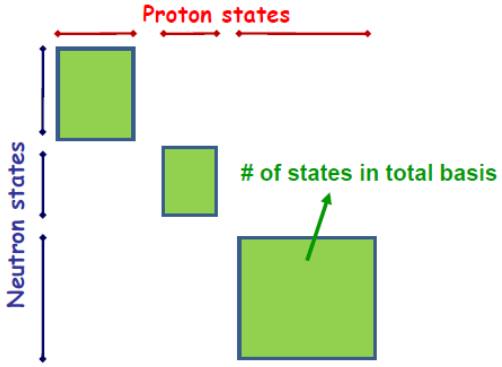
- General purpose configuration interaction (CI) code
- On-the-fly calculation of the many-body Hamiltonian
- Fortran 90, MPI+OpenMP
- 30,000+ lines of code

Represent an area by its boundary

- Factorization of problem
- Reduces dramatically memory load

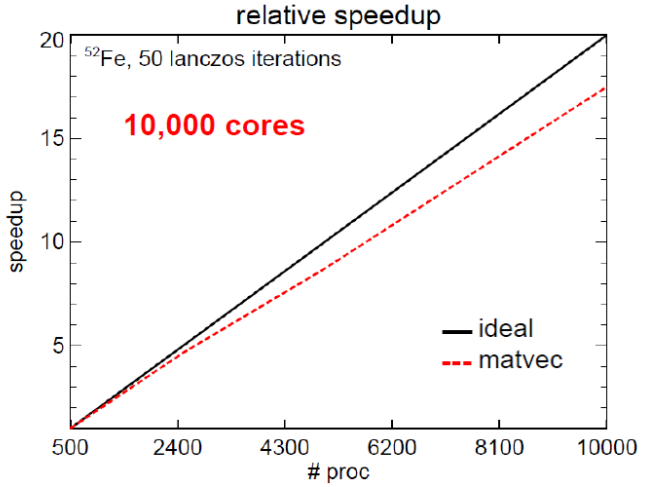


$$|\alpha\rangle = |\alpha_p\rangle \times |\alpha_n\rangle$$



Hamiltonian can be factorized similarly

(Relative) speedup of matrix-vector multiplication:



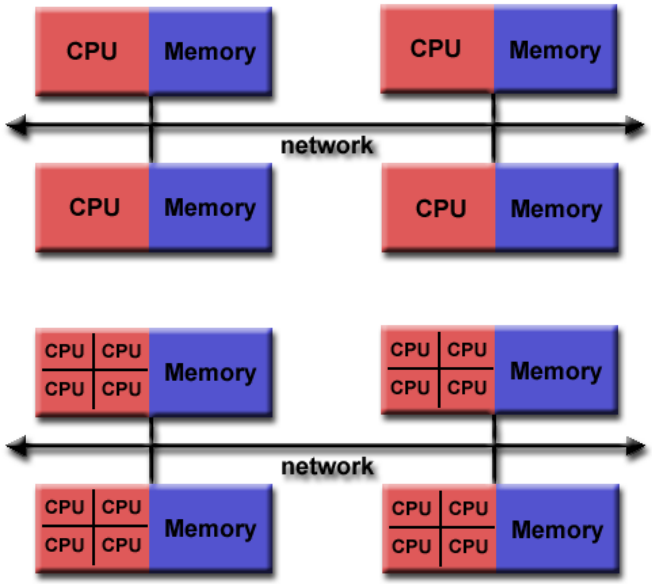
Krastev et al, Comput. Phys. Commun., 184 (12) (2013), pp. 2761-2774

What is MPI?

- M P I = Massage Passing Interface
- MPI is a specification for the developers and users of message passing libraries. By itself, it is NOT a library
- MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process
- Most recent version is MPI-3
- Actual MPI library implementations differ in which version and features of the MPI standard they support

MPI Programming Model

- Originally MPI was designed for distributed memory architectures
- As architectures evolved, MPI implementations adapted their libraries to handle shared, distributed, and hybrid architectures
- Today, MPI runs on virtually any hardware platform
  - Shared Memory
  - Distributed Memory
  - Hybrid
- Programing model remains clearly distributed memory model, regardless of the underlying physical architecture of the machine
- Explicit parallelism – programmer is responsible for correct implementation of MPI



Reasons for using MPI

- **Standardization** - MPI is the only message passing specification which can be considered a standard. It is supported on virtually all HPC platforms
- **Portability** - There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms
- **Functionality** - There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1
- **Availability** - A variety of implementations are available, both vendor and public domain

MPI Language Interfaces

- C/C++
- Fortran
- Java
- Python (pyMPI, mpi4py, pypar, MYMPI)
- R (Rmpi)
- Perl (Parallel::MPI)
- MATLAB (DCS)
- Others

Compiling MPI Programs on Odyssey

MPI Implementation	Compiler	Flag
OpenMPI MVAPICH2 Intel MPI	mpicc mpicxx mpif90	None

Intel:

```
module load  intel/17.0.4-fasrc01
module load  openmpi/3.1.1-fasrc01
mpicxx -o mpi_test.x mpi_test.cpp
```

GNU:

```
module load gcc/8.2.0-fasrc01
module load openmpi/3.1.1-fasrc01
mpicxx -o mpi_test.x mpi_test.cpp
```

https://www.rc.fas.harvard.edu/resources/documentation/software-development-on-odyssey/mpi-software-on-odyssey

Running MPI Programs on Odyssey (1)

Interactive test jobs:

(1) Start an interactive bash shell

```
> srun -p test -n 4 --pty -mem=4G -t 0-06:00 /bin/bash
```

(2) Load required modules, e.g.,

```
> module load gcc/8.2.0-fasrc01 openmpi/3.1.1-fasrc01
```

(3) Compile your code (or use a `Makefile`)

```
> mpicxx -o hello_mpi.x hello_mpi.cpp
```

(4) Run the code

```
> mpirun -np 4 ./hello_mpi.x

 Hello world from process 0 out of 4
 Hello world from process 1 out of 4
 Hello world from process 2 out of 4
 Hello world from process 3 out of 4
```

Running MPI Programs on Odyssey (2)

Batch jobs:

(1) Compile your code, e.g.,

```
> module load gcc/8.2.0-fasrc01 openmpi/3.1.1-fasrc01
> mpicxx -o mpi_hello.x mpi_hello.cpp
```

(2) Prepare a batch-job submission script

```
#!/bin/bash
#SBATCH -J mpi_job                                  # Job name
#SBATCH -o slurm.out                                # STD output
#SBATCH -e slurm.err                                # STD error
#SBATCH -p shared                                   # Queue / partition
#SBATCH -t 0-00:30                                  # Time (D-HH:MM)
#SBATCH --mem-per-cpu=4000                          # Memory per MPI task
#SBATCH -n 8                                        # Number of MPI tasks
module load gcc/8.2.0-fasrc01 openmpi/3.1.1-fasrc01 # Load required modules
srun -n $SLURM_NTASKS --mpi=pmix ./hello_mpi.x
```

(3) Submit the job to the queue

```
> sbatch mpi_test.run
```

Running MPI Programs on Odyssey (3)

- Sometimes programs can be picky about having MPI available on all the nodes it runs on so it is good to have MPI module loads in your `.bashrc` file

- Some codes are topology sensitive thus the following slurm options can be helpful
  - `--contiguous` # Contiguous set of nodes
  - `--ntasks-per-node` # Number of tasks per node
  - `--hint` # Bind tasks according to hints
  - `--distribution, -m` # Specify distribution method for tasks

- For hybrid mode jobs you would set both `-c` and `-n`

https://slurm.schedmd.com/sbatch.html
https://slurm.schedmd.com/mc_support.html
https://www.rc.fas.harvard.edu/resources/documentation/software-development-on-odyssey/hybrid-mpiopenmp-codes-on-odyssey

MPI Exercises

Exercises - Setup

- Login to the cluster

- Make a directory for this session, e.g.,

```
> mkdir ~/MPI
```

- Get a copy of the MPI examples. These are hosted at Github at https://github.com/fasrc/User_Codes/tree/master/Courses/AC290

```
> cd ~/MPI
> git clone https://github.com/fasrc/User_Codes.git
> cd User_Codes/Courses/AC290
```

- Load compiler and MPI library software modules

```
> module load gcc/8.2.0-fasrc01
> module load openmpi/3.1.1-fasrc01
```

MPI Exercises - Overview

1. MPI Hello World program
2. Parallel FOR loops in MPI – dot product
3. Scaling – speedup and efficiency
4. Parallel Matrix-Matrix multiplication
5. Parallel Lanczos algorithm

https://github.com/fasrc/User_Codes/tree/master/Courses/AC290

Exercise 1: MPI Hello World

```
#include <iostream>
#include <mpi.h>
using namespace std;
// Main program.....................................
int main(int argc, char** argv){
  int i;
  int iproc;
  int nproc;
// Initialize MPI..................................
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&iproc);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);

  for ( i = 0; i < nproc; i++ ){
    MPI_Barrier(MPI_COMM_WORLD);
    if ( i == iproc ){
      cout << "Hello world from process " << iproc
           << " out of " << nproc << endl;
    }
  }
// Shut down MPI...................................
  MPI_Finalize();
  return 0;
}
```

Exercise 1: MPI Hello World

(1) Description – a simple parallel "Hello World" program printing out the number of MPI parallel processes and process IDs

(2) Compile the program

```
> cd ~/MPI/User_Codes/Courses/AC290/Example1
> make
```

(3) Run the program (the default is setup to 4 MPI tasks)

```
> sbatch sbatch.run
```

(4) Explore the output (the "omp_hello.dat" file), e.g.,

```
> cat mpi_hello.dat
Hello world from process 0 out of 4
Hello world from process 1 out of 4
Hello world from process 2 out of 4
Hello world from process 3 out of 4
```

(5) Run the program with a different MPI process number – e.g., 2, 4, 8

Parallelizing DO / FOR Loops

In almost all scientific and technical applications, the hot spots are likely to be found in DO / FOR loops.

Thus parallelizing DO / FOR loops is one of the most important tasks when you parallelize your program.

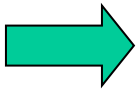
The basic technique of parallelizing DO / FOR loops is to distribute iterations among MPI processes and to let each process do its portion in parallel.

Usually, the computations within a DO / FOR loop involve arrays whose indices are associated with the loop variable. Therefore distributing iterations can often be regarded as dividing arrays and assigning chunks (and computations associated with them) to MPI processes.

Block Distribution

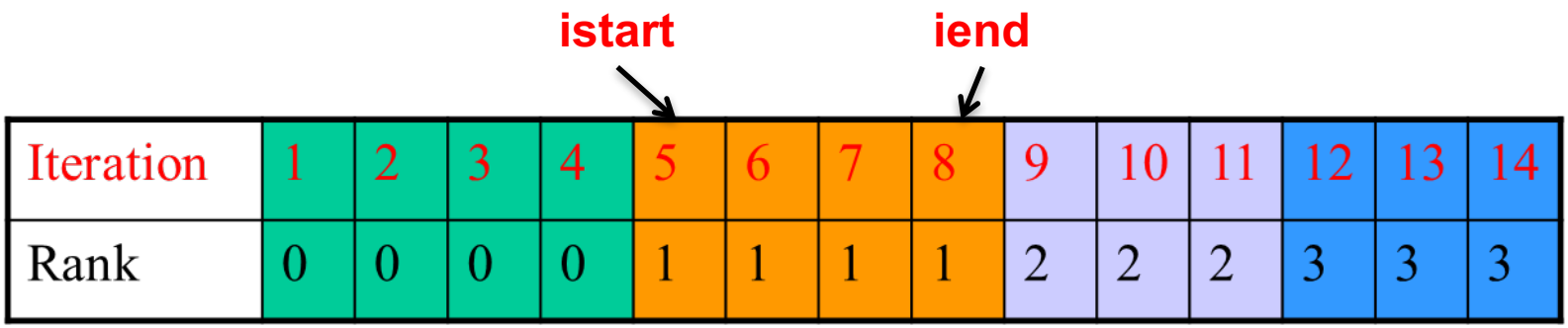
In block distribution, iterations are divided into **p** parts, where **p** is the number of MPI processes to be executed in parallel.

```
DO i = n1, n2
   computation
ENDDO
```



```
DO i = istart, iend
   computation
ENDDO
```

Example: Distributing 14 iterations over 4 MPI tasks

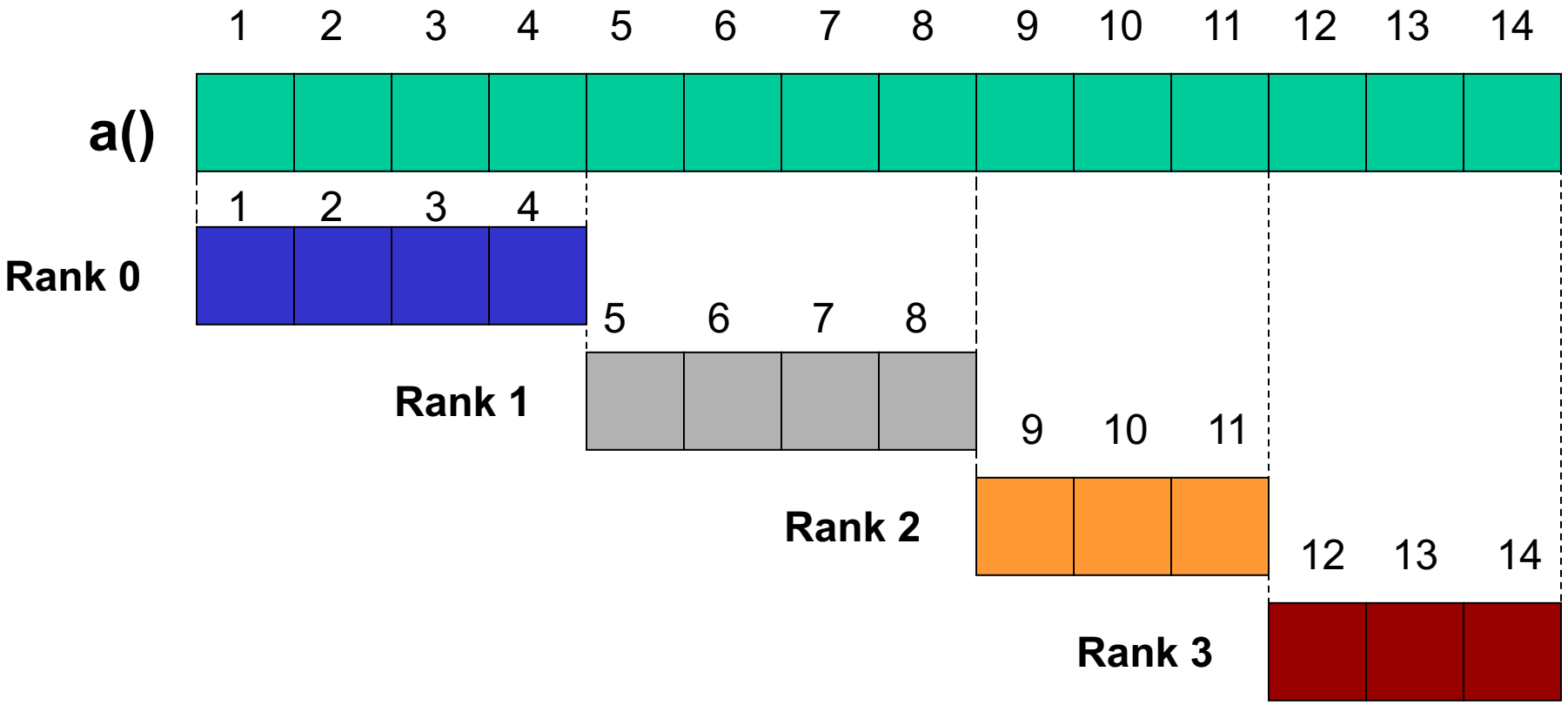


Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	0	0	0	1	1	1	1	2	2	2	3	3	3

Shrinking Arrays

Extremely important for efficient memory management!

Block distribution of 14 iterations over 4 cores. Each MPI process needs only part of the array a()



The **para_range** subroutine

Computes the iteration range for each MPI process

FORTRAN implementation

```
subroutine para_range(n1, n2, nprocs, irank, istart, iend)
  integer(4) :: n1          !Lower limit of iteration variable
  integer(4) :: n2          !Upper limit of iteration variable
  integer(4) :: nprocs      !Number of MPI ranks
  integer(4) :: irank       !MPI rank ID
  integer(4) :: istart      !Start of iterations for rank iproc
  integer(4) :: iend        !End of iterations for rank iproc
  iwork1 = ( n2 - n1 + 1 )  /  nprocs
  iwork2 = MOD(n2 - n1 + 1, nprocs)
  istart = irank * iwork1 + n1 + MIN(irank, iwork2)
  iend = istart + iwork1 - 1
  if ( iwork2 > irank ) iend = iend + 1
  return
end subroutine para_range
```

The **para_range** subroutine, cont'd

Computes the iteration range for each MPI process

C / C++ implementation

```
void para_range(int n1, int n2, int &nprocs, int &irank, int &istart, int &iend){
  int iwork1;
  int iwork2;
  iwork1 = ( n2 - n1 + 1 )  /  nprocs;
  iwork2 = ( ( n2 - n1 + 1 ) %  nprocs );
  istart = irank * iwork1 + n1 + min(irank, iwork2);
  iend = istart + iwork1 - 1;
  if ( iwork2 > irank ) iend = iend + 1;
}
```

Exercise 2: Parallel MPI for / do loops

(1) Description – Program performs a dot product of 2 vectors in parallel

(2) Review the source code and compile the program

```
> cd ~/MPI/User_Codes/Courses/AC290/Example2
> make
```

(3) Run the program (the default is setup to 4 MPI tasks)

```
> sbatch sbatch.run
```

(4) Explore the output (the "mpi_dot.dat" file), e.g.,

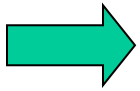
```
> cat mpi_dot.dat
Global dot product: 676700
Local dot product for MPI process 0: 11050
Local dot product for MPI process 1: 74800
Local dot product for MPI process 2: 201050
Local dot product for MPI process 3: 389800
```

(5) Run the program with a different MPI process number – e.g., 2, 4, 8

Cyclic Distribution

In cyclic distribution, the iterations are assigned to processes in a round-robin fashion.

```
DO i = n1, n2
   computation
ENDDO
```



```
DO i = n1+iproc, n2, nproc
   computation
ENDDO
```

Example: Distributing 14 iterations over 4 cores in round-robin fashion

Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	1	2	3	0	1	2	3	0	1	2	3	0	1

Scaling and Efficiency

How much faster will the program run?

Speedup:

$$S(n) = \frac{T(1)}{T(n)}$$

Time to complete the computation on **one** MPI process

Time to complete the computation on **n** MPI processes

Efficiency:

$$E(n) = \frac{S(n)}{n}$$

Tells you how efficiently you parallelized your code

Example 3: Scaling – Compute PI in Parallel

Monte-Carlo Approximation of PI:

1. Inscribe a circle in a square

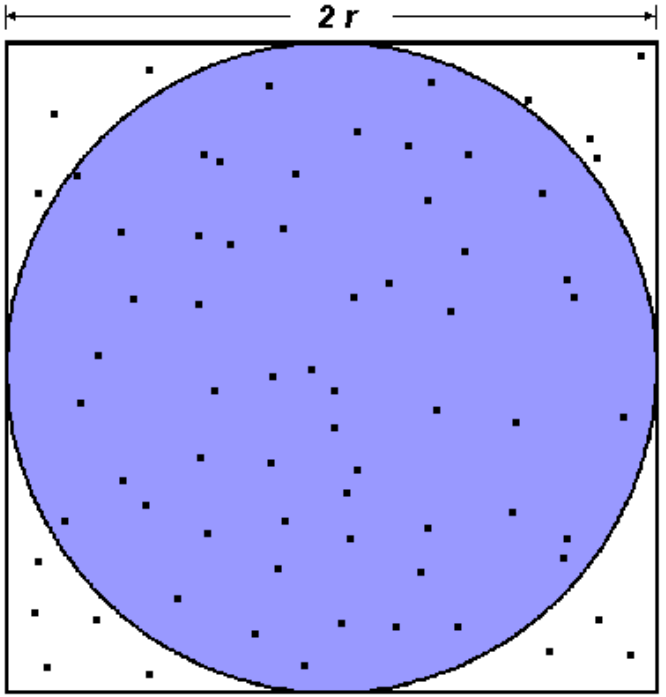
2. Randomly generate points in the square

3. Determine the number of points in the square that are also in the circle

4. Let r be the number of points in the circle divided by the number of points in the square

5. PI ~ 4 r

6. Note that the more points generated, the better the approximation



$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

Example 3: Scaling – Compute PI in Parallel

(1) Description – Program performs parallel Monte-Carlo approximation of PI

(2) Review the source code and compile the program

```
> cd ~/MPI/User_Codes/Courses/AC290/Example3
> make
```

(3) Run the program

```
> sbatch sbatch.run
```

(4) Explore the output (the "mpi_dot.dat" file), e.g.,

```
> cat mpi_pi.dat
…
Elapsed time = 3.429223 seconds
Pi is approximately 3.1416008000000000, Error is 0.0000081464102069
Exact value of PI is 3.1415926535897931
…
```

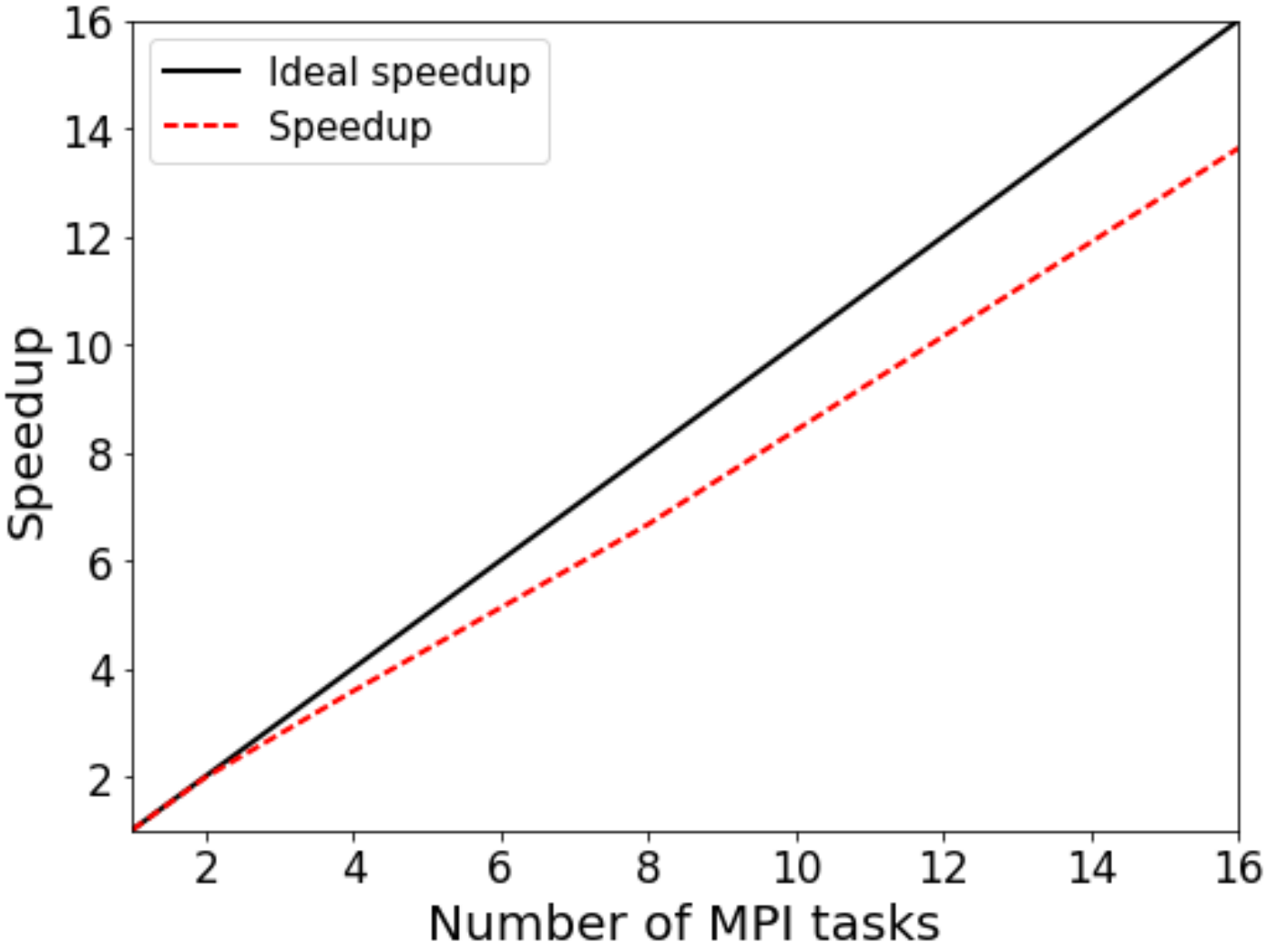
(5) Run the program with a different number of MPI processes – 1, 2, 4, 8 – and record the run times for each case. This will be needed to compute the speedup and efficiency (the default is set to run on 1, 2, 4, 8, and 16 MPI tasks)

Example 3: Scaling – Compute PI in Parallel

You may use the `speedup.py` Python code to calculate the speedup and efficiency. It generates the below table plus a speedup figure

# MPI tasks	Walltime	Speedup	Efficiency (%)
1	12.27	1.00	100.00
2	6.19	1.98	99.11
4	3.43	3.58	89.43
8	1.84	6.67	83.36
16	0.90	13.63	85.21

Example 3: Scaling – Compute PI in Parallel



Exercise 4: Matrix Multiplication

- A standard problem in computing is matrix multiplication:

  C = A x B

- In this example we take a naïve approach to parallelizing matrix multiplication

- The matrix A is divided up based on its rows by the number of ranks. Each sub array is sent to its relevant ranks by using the `MPI_Scatter` command

- Matrix B is simply sent to all processors using the `MPI_Bcast` command.

- Each rank computes its subset of C based on the part of A it received

- The full solution for C is brought together using `MPI_Gather`

Exercise 4: Matrix Multiplication

(1) Description – A simple algorithm for matrix multiplication demonstrating the use of `MPI_Bcast`, `MPI_Scatter`, and `MPI_Gather`

(2) Compile the program

```
> cd ~/MPI/User_Codes/Courses/AC290/Example4
> make
```

(3) Run the program (the default is setup to 4 ranks)

```
> sbatch sbatch.run
```

(4) Explore the output (the "`mmult.out`" file).

(5) Run the program with different MPI process number – e.g., 1, 2, 4, 8. See how the run time varies depending on number of ranks (**HINT**: use `sacct`, `time`, or `MPI_Wtime` to get duration). Try varying the size of the matrix allowed to see how long it takes and then do scaling tests to see how well this code is parallelized.

Exercise 5: Parallel Lanczos diagonalization

(1) Description – Program performs parallel Lanczos diagonalization of a random symmetric matrix of dimension 100 x 100

(2) Review the source code and compile the program

```
> cd ~/MPI/User_Codes/Courses/AC290/Example5
> make
```

(3) Run the program

```
> sbatch sbatch.run
```

(4) Explore the output (the "planczos.dat" file), e.g.,

```
> cat planczos.out
…
 iteration:           50
           1   50.010946087355691        50.010946087355670
           2   5.1987505309251540        5.1987505309260547
           3   5.1856783411618199        5.1856783411660166
           4   5.0014143249256930        5.0014143250821714
           5   4.8032829796748748        4.8032831650372225
 Lanczos iterations finished...
```

(5) Run the program with a different number of MPI processes – 1, 2, 4, 8.

Summary and hints for efficient parallelization

- Is it even worth parallelizing my code?
  - Does your code take an intractably long amount of time to complete?
  - Do you run a single large model or do statistics on multiple small runs?
  - Would the amount of time it take to parallelize your code be worth the gain in speed?
- Parallelizing established code vs. starting from scratch
  - Established code: Maybe easier / faster to parallelize, but my not give good performance or scaling
  - Start from scratch: Takes longer, but will give better performance, accuracy, and gives the opportunity to turn a “black box” into a code you understand

Summary and hints for efficient parallelization

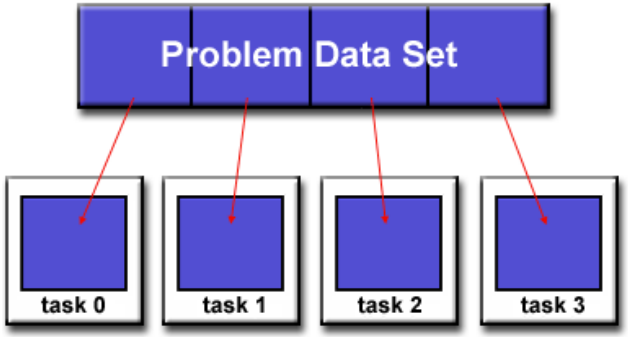
- Increase the fraction of your program that can be parallelized. Identify the most time consuming parts of your program and parallelize them. This could require modifying your intrinsic algorithm and code's organization
- Balance parallel workload
- Minimize time spent in communication
- Use simple arrays instead of user defined derived types
- Partition data. Distribute arrays and matrices – allocate specific memory for each MPI process
- For I/O intensive applications implement parallel I/O in conjunction with a high-performance parallel filesystem, e.g., Lustre

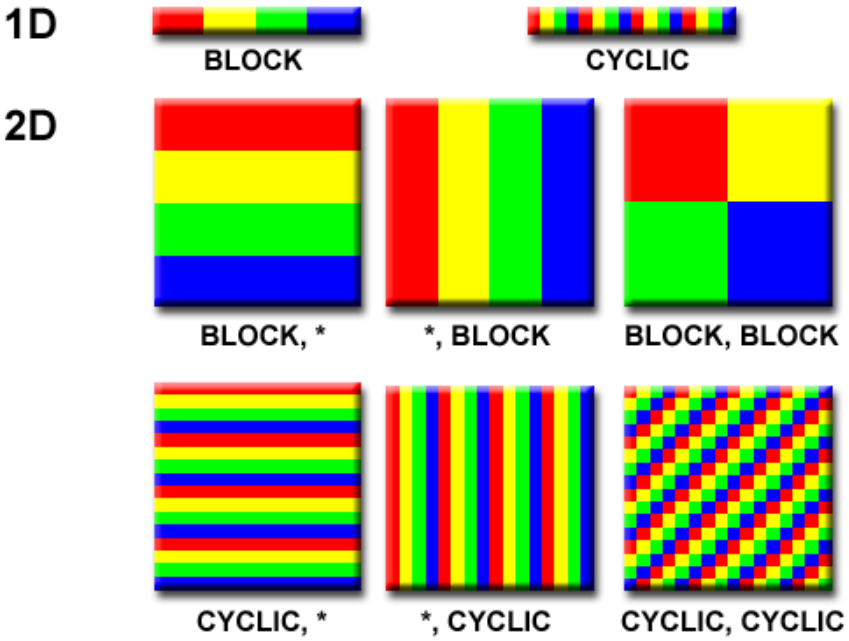
Extra Slides

Designing parallel programs – partitioning

One of the first steps in designing a parallel program is to break the problem into discrete “chunks” that can be distributed to multiple parallel tasks.

Domain Decomposition:
Data associate with a problem is partitioned – each parallel task works on a portion of the data



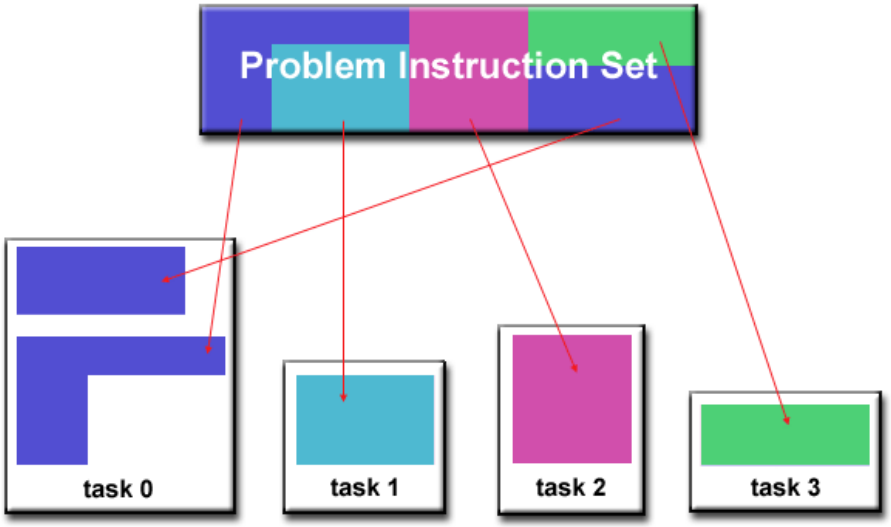


There are different ways to partition the data

Designing parallel programs – partitioning

One of the first steps in designing a parallel program is to break the problem into discrete “chinks” that can be distributed to multiple parallel tasks.

Functional Decomposition:
Problem is decomposed according to the work that must be done. Each parallel task performs a fraction of the total computation.



Designing parallel programs – communication

Most parallel applications require tasks to share data with each other.

Cost of communication: Computational resources are used to package and transmit data. Requires frequently synchronization – some tasks will wait instead of doing work. Could saturate network bandwidth.

Latency vs. Bandwidth: Latency is the time it takes to send a minimal message between two tasks. Bandwidth is the amount of data that can be communicated per unit of time. Sending many small messages can cause latency to dominate communication overhead.

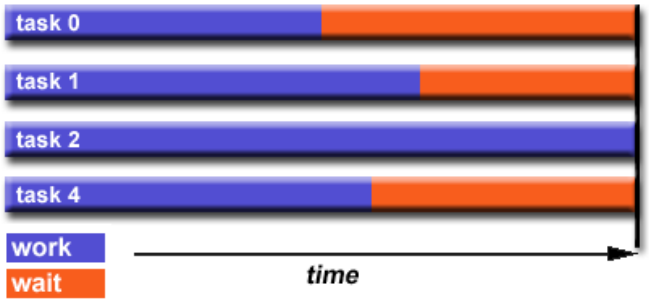
Synchronous vs. Asynchronous communication: Synchronous communication is referred to as blocking communication – other work stops until the communication is completed. Asynchronous communication is referred to as non-blocking since other work can be done while communication is taking place.

Scope of communication: Point-to-point communication – data transmission between tasks. Collective communication – involves all tasks (in a communication group)

This is only partial list of things to consider!

Designing parallel programs – loadbalancing

Load balancing is the practice of distributing approximately equal amount of work so that all tasks are kept busy all the time.



How to Achieve Load Balance?

Equally partition the work given to each task: For array/matrix operations equally distribute the data set among parallel tasks. For loop iterations where the work done for each iteration is equal, evenly distribute iterations among tasks.

Use dynamic work assignment: Certain class problems result in load imbalance even if data is distributed evenly among tasks (sparse matrices, adaptive grid methods, many body simulations, etc.). Use scheduler – task pool approach. As each task finishes, it queues to get a new piece of work. Modify your algorithm to handle imbalances dynamically.

Designing parallel programs – I/O

The Bad News:

- I/O operations are inhibitors of parallelism
- I/O operations are orders of magnitude slower than memory operations
- Parallel file systems may be immature or not available on all systems
- I/O that must be conducted over network can cause severe bottlenecks

The Good News:

- Parallel file systems are available (e.g., Lustre)
- MPI parallel I/O interface has been available since 1996 as a part of MPI-2

I/O Tips:

- Reduce overall I/O as much as possible
- If you have access to parallel file system, use it
- Writing large chunks of data rather than small ones is significantly more efficient
- Fewer, larger files perform much better than many small files
- Have a subset of parallel tasks to perform the I/O instead of using all tasks, or
- Confine I/O to a single tasks and then broadcast (gather) data to (from) other tasks

Big Thanks to ^^^

Plamen Krastev – Computational Scientist and Research Consultant

February 2019
AC290