Using the Interactive Parallelization Tool to Generate Parallel Programs (OpenMP, MPI, and CUDA)

PEARC18 Tutorial

July 23, 2018

PRESENTED BY:

Ritu Arora: rauta@tacc.utexas.edu

Lars Koesterke: lars@tacc.utexas.edu

Link to the Slides and Other Material

https://tinyurl.com/y6v6ftwg

Log on to the IPT Gateway using your TACC portal id: *https://ipt.tacc.cloud*

If you run into any challenge in using the IPT Gateway, please log on to Stampede2 using your TACC portal credentials

Assumptions and Learning Objectives

- Assumptions:
  - The audience has the basic understanding of C or Fortran programming
  - The audience has access to an MPI installation
    - Your training accounts will get you access to the MPI installation on Stampede2
- Learning Objectives
  - Introduction to MPI
    - What is MPI?
    - General structure of MPI programs
    - MPI concepts – groups, communicators, process id, point-to-point communication versus collective communication
  - Basic MPI examples
  - Basic routines used for writing MPI programs
  - What is scaling and parallel efficiency?
  - More examples

Outline

- Short overview of Interactive Parallelization Tool (IPT)
- High-Level introduction to the parallel programming concepts
- Introduction to MPI
  - Concepts
  - Patterns
  - Learn to use MPI with hands-on examples

Keeping-Up with the Advancement in HPC Platforms can be an Effort-Intensive Activity

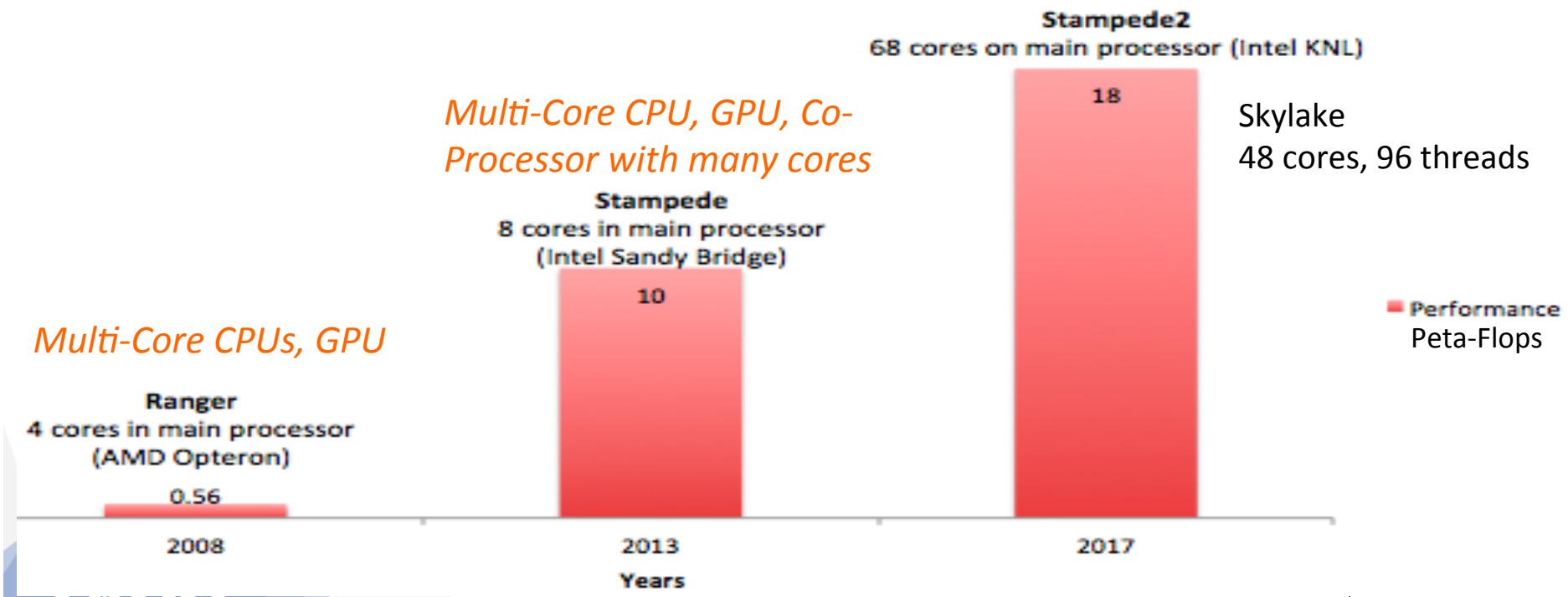
- Code modernization can be required to take advantage of the continuous advancement made in the computer architecture discipline and the programming models
  - To efficiently use many-core processing elements
  - To efficiently use multiple-levels of memory hierarchies
  - To efficiently use the shared-resources
- The manual process of code modernization can be effort-intensive and time-consuming and can involve steps such as follows:
  1. Learning about the microarchitectural features of the latest platforms
  2. Analyzing the existing code to explore the possibilities of improvement
  3. Manually reengineering the existing code to parallelize or optimize it
  4. Explore compiler-based optimizations
  5. Test, and if needed, repeat from step 3

Evolution in the HPC Landscape – HPC Systems at TACC

Multi-Core and Manycore CPUs

Multi-Core CPU, GPU, Co-Processor with many cores

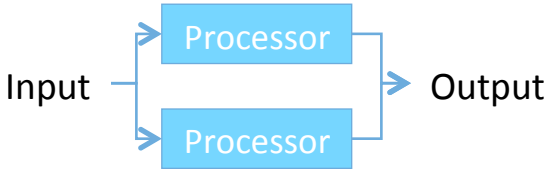
Multi-Core CPUs, GPU



Skylake
48 cores, 96 threads

Why Parallel? (1) --- Power efficiency

Power = CV^2F



Cap = 1.0c
Volts = 1.0v
Freq = 1.0f
Power = $1.0cv^2f$

Do same work with 2 processors at 0.5 freq.

Voltage ~ Frequency
2x more wires ->
~2x Capacitance

Cap = 2.2c
Volts = 0.6v
Freq = 0.5f
Power = $0.4cv^2f$

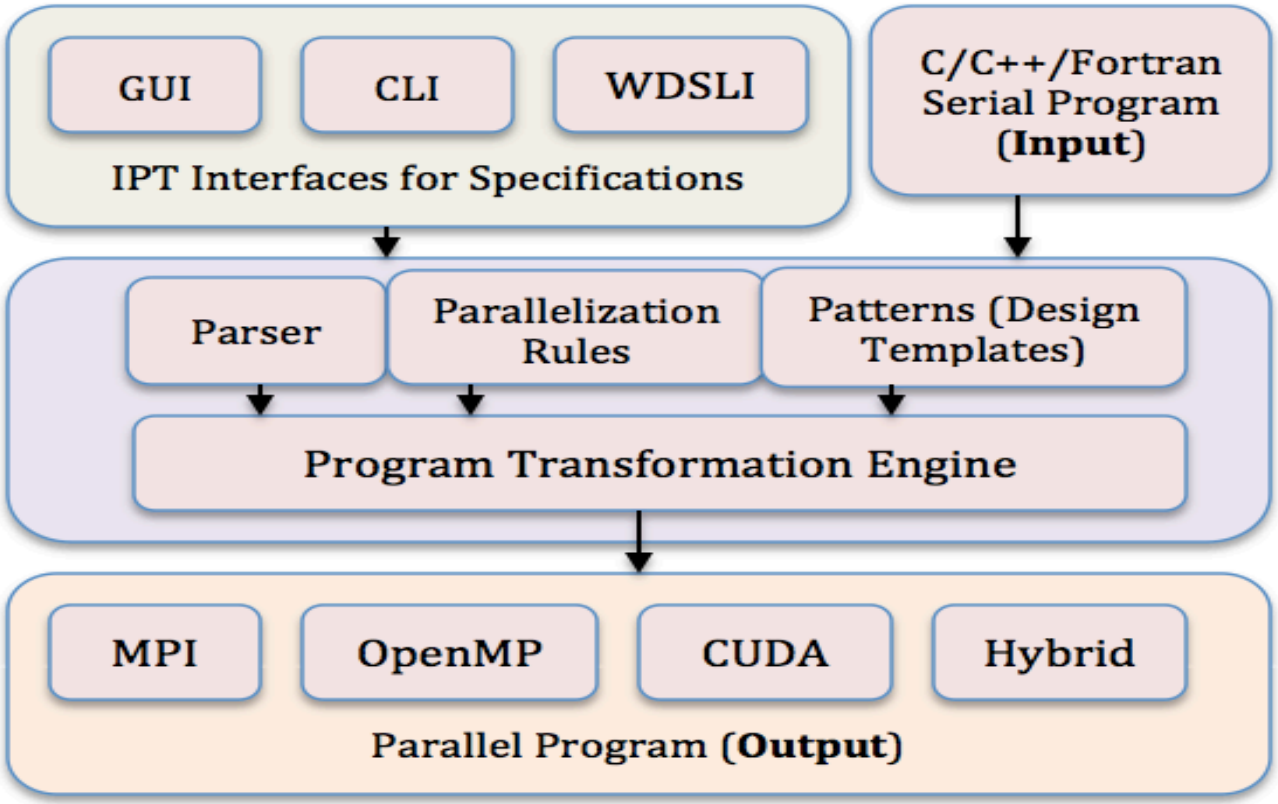
https://www.youtube.com/watch?v=cMWGeJyrc9w&index=2&list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG

IPT – How can it help you?

If you know **what** to parallelize and **where**, IPT can help you with the syntax (of MPI/OpenMP/CUDA) and typical code reengineering for parallelization

- Main purpose of IPT: *a tool to aid in learning parallel programming*
- Helps in learning parallel programming concepts without feeling burdened with the information on the syntax of MPI/OpenMP/CUDA
- C and C++ languages supported as of now, Fortran will be supported in future

IPT: High-Level Overview



Before Using IPT

- It is important to know the logic of your serial application before you start using IPT
  - IPT is not a 100% automatic tool for parallelization
- Understand the high-level concepts related to parallelization
  - Data distribution/collection
    - For example: reduction
  - Synchronization
  - Loop/Data dependency
- Familiarize yourself with the user-guide

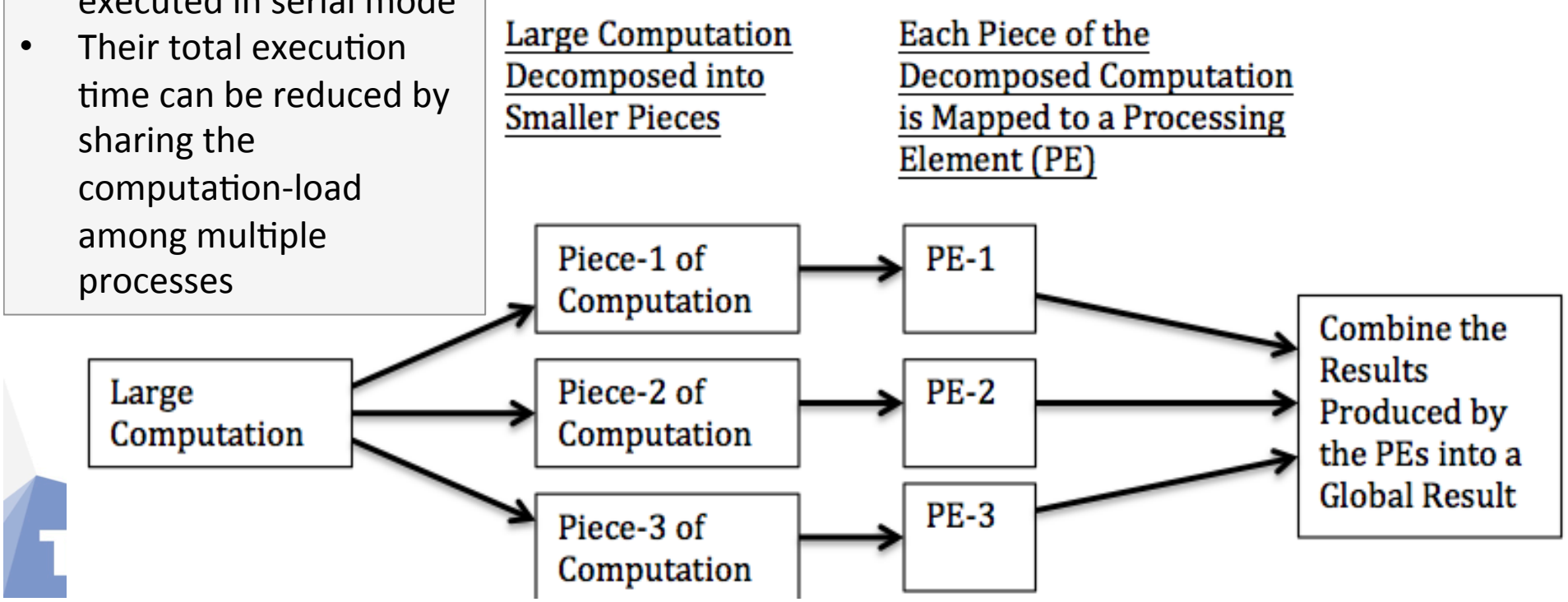
Concepts Related to Parallel Programming

- General
  - Loop dependence analysis and parallelization
  - Data distribution
    - Split/block big arrays into smaller ones
    - Scatter parts of arrays to multiple processors
    - Broadcast arrays or variables
  - Data collection
    - Gather
    - Reduction
  - Synchronization

- Specific to MPI
  - Group, Rank, Size,
  - Communicator
  - Types of communication
  - Environment variables that can impact the run-time behavior of the program (not covered in today's session)

Process of Parallelizing a Large Number of Computations in a Loop

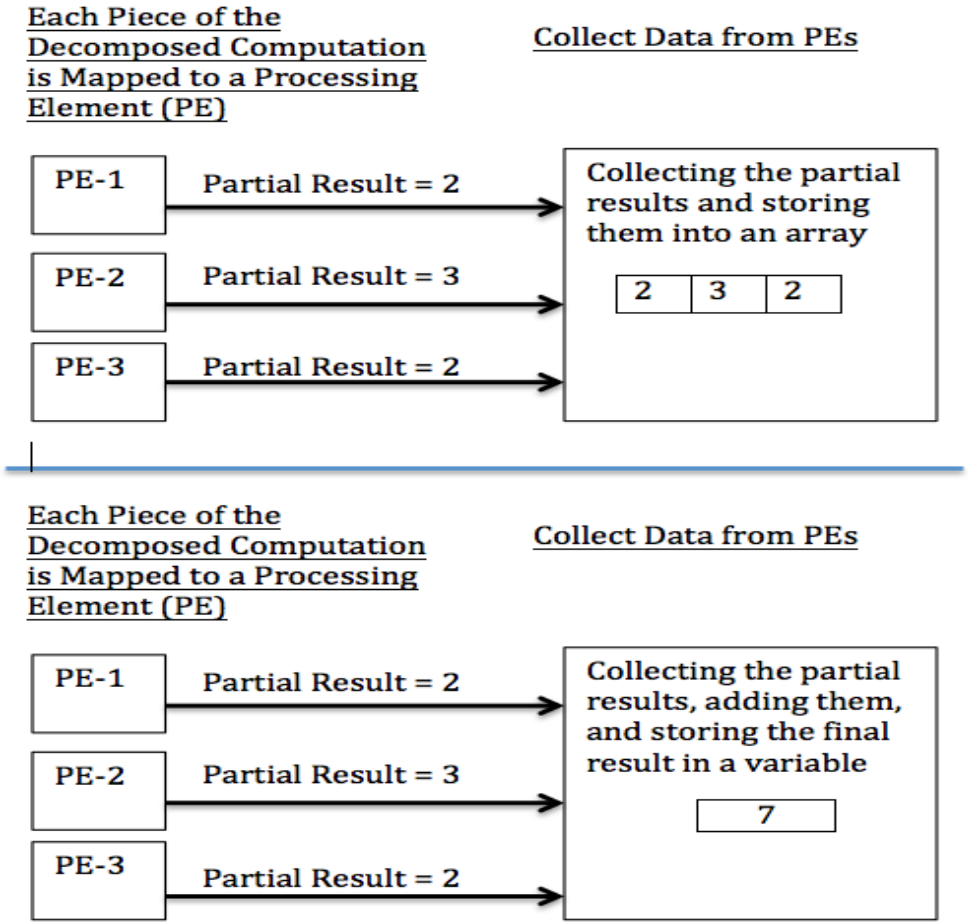
- Loops can consume a lot of processing time when executed in serial mode
- Their total execution time can be reduced by sharing the computation-load among multiple processes



Loop/Data Dependency

- Loop dependence implies that there are dependencies between the iterations of a loop that prevent its parallel processing
  - Analyze the code in the loop to determine the relationships between statements
- Analyze the order in which different statements access memory locations (data dependency)
- On the basis of the analysis, it may be possible to restructure the loop to allow multiple processes to work on different portions of the loop in parallel
- For applications that have hotspots containing ante-dependency between the statements in a loop (leading to incorrect results upon parallelization), code refactoring should be done to remove the ante-dependency prior to parallelization.

Data Distribution/Collection/Reduction



Processing Element (PE) is a process in MPI

Synchronization

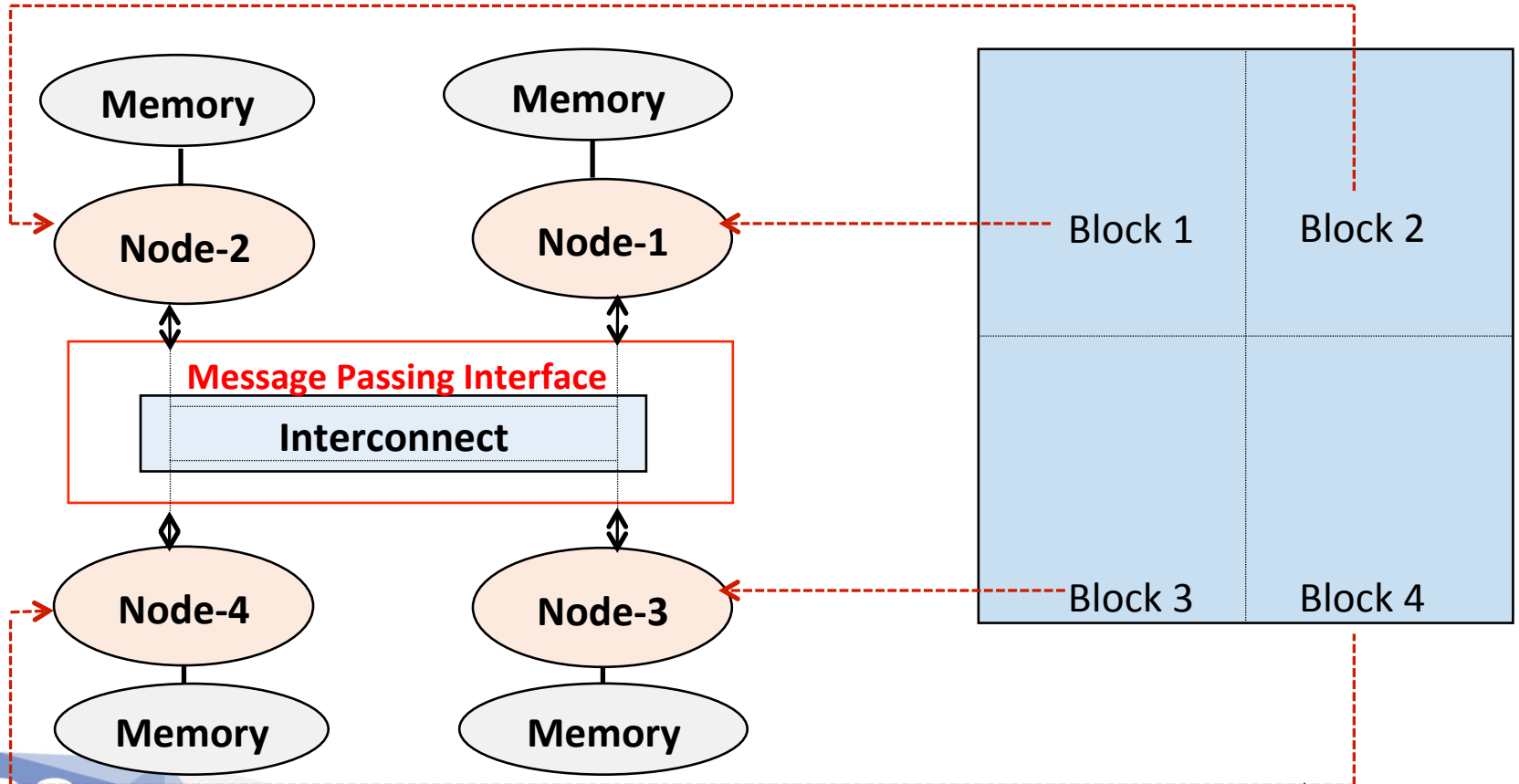
- Synchronization helps in controlling the execution of processes relative to other processes in a team

Gentle Introduction to MPI

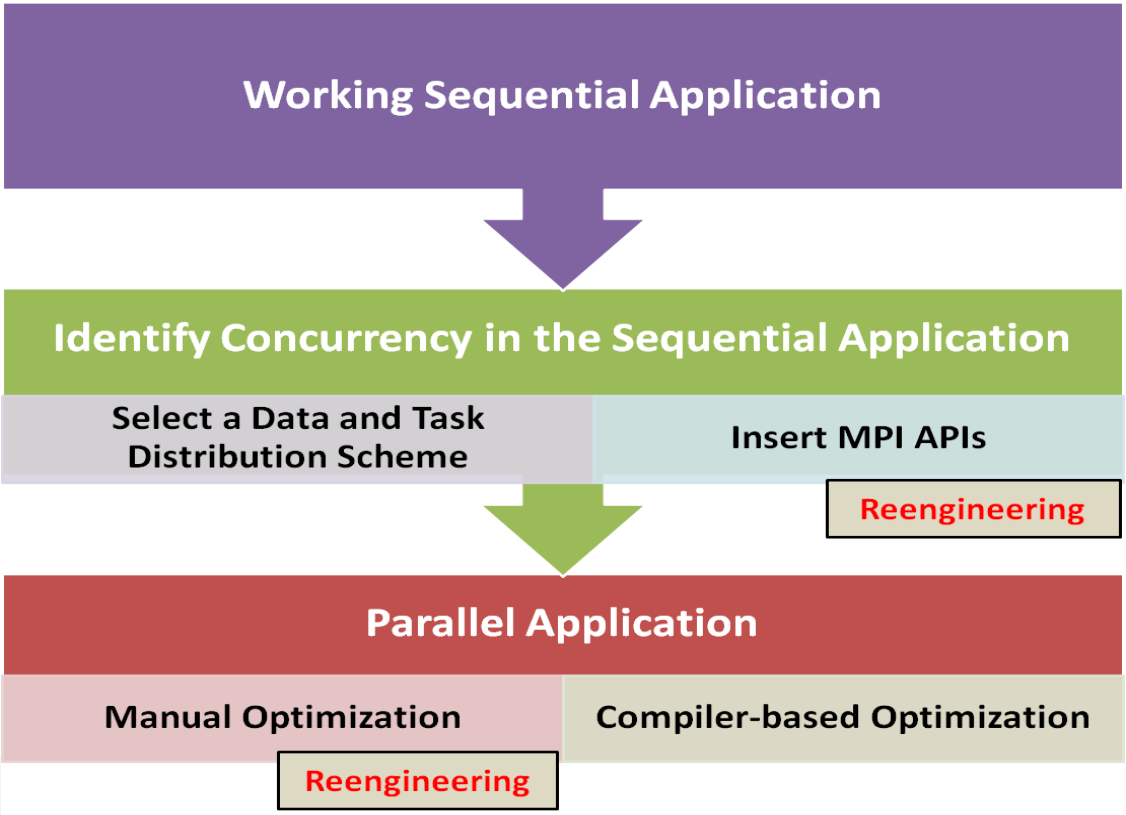
Message Passing Interface (MPI)

- MPI is a specification for message passing library that is standardized by MPI Forum
- Multiple vendor-specific implementations: MPICH, OpenMPI
- MPI implementations are used for programming systems with distributed memory
  - Each process has a different address space
  - Processes need to communicate with each other
    - Synchronization
    - Data Exchange
  - Can also be used for shared memory and hybrid architectures
- MPI specifications have been defined for C and Fortran languages, are credited for portability and performance, recent version is MPI-3

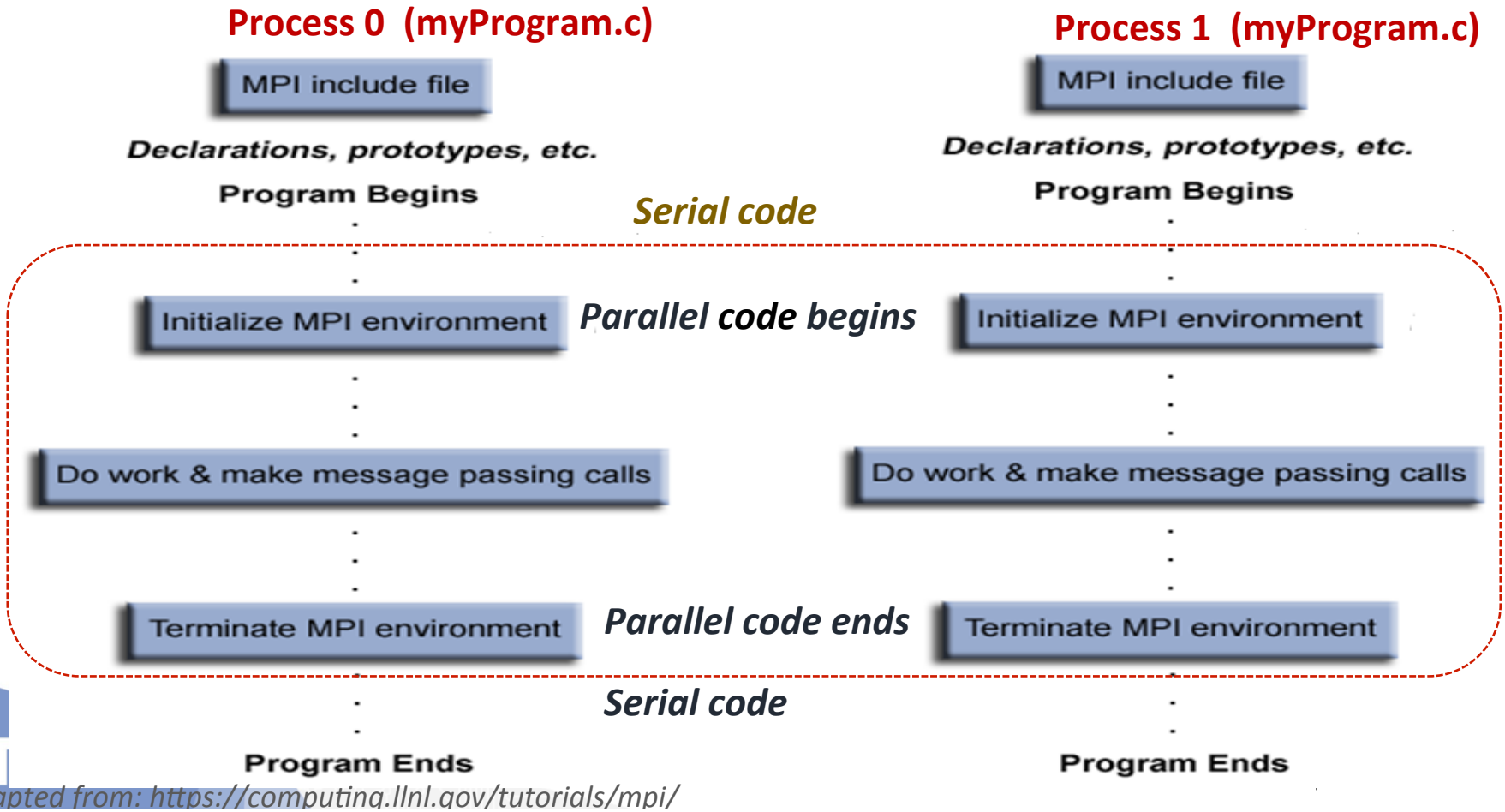
Divide & then Conquer with MPI



Explicit Parallelization with MPI (traditional way)

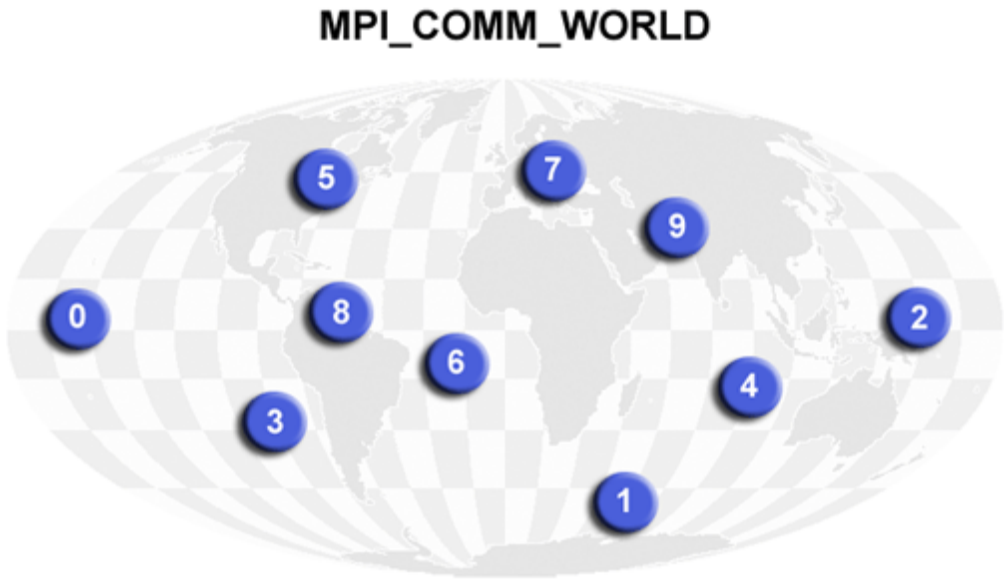


General Structure of MPI Programs



Adapted from: https://computing.llnl.gov/tutorials/mpi/

Concept of Communicators and Groups



Source: https://computing.llnl.gov/tutorials/mpi/

- Communicators and groups are objects that are used to define which collection of processes may communicate with each other
- Most MPI routines require a communicator as an argument
- **`MPI_COMM_WORLD`** is the predefined communicator that includes all MPI processes
- Multiple communicators and groups can be defined

First Things First (1)

Logon to Stampede

How to logon to Stampede:

- Please fill out the sign-up sheet.
  - Use this account name and password to logon
- Open a terminal on your computer (MacOS or Linux)
- Use Putty on a Windows laptop

```
$ ssh <username>@stampede.tacc.utexas.edu
```

First Things First (2)

Start an interactive session on Stampede with `idev`

How to launch an **`idev`** job on Stampede:

- Idev: Interactive development
- When asked, accept to use the reservation
- Once the job is running and the prompt returns: check hostname

- `$ idev -N 1 -n 4 -m 120`
- `$ hostname`

- Now all of you have a single node where you can edit the code, run IPT, and run the serial and parallel code

Retrieve the Example Files

Copy the files from Ritu's account

- `$ cp -pr /scratch/01698/rauta/PEARC18_MPI .`

Please ignore the 'Permission denied'

- Now all of you have a single node where you can edit the code, run IPT, and run the serial and parallel code

Get IPT ready for use by executing a shell script

- `$ source ./sourceME.sh`
- `$ module load intel`

First example (1)

Let's start with example 1

- `$ cd examples`
- `$ cd example1`
- `$ ls -al`

- There is one C file
- For now IPT only works with C/C++

First example (2)

Let's start with example 1

- Before using IPT inspect the code (we will do this together here)
- Use one of these: 'more', 'vi', or 'emacs'

```
$ more example1.c
$ vi example1.c
$ emacs example1.c
```

IPT will ask you a lot of questions
These are the important ones (for now)

- Which loop should IPT parallelize?
- Is there a reduction?

Serial Program: `example1.c`

```
#include <stdio.h>

int main(){

  printf("Wonderful Class!\n");

  return(0);

}
```

Compiling:

```
icc -o example1 example1.c
```

Running:

```
./example1
Wonderful Class!
```

Steps for Using IPT to Parallelize example1.c

Either login to the IPT Gateway or Stampede2

If on the IPT Gateway:

If on Stampede2:

Steps for Using IPT to Parallelize example1.c

Online demo

Explaining the Steps for Parallelization : `example1.c` to `mpiExample1.c`

```
#include <stdio.h>
#include "mpi.h"
int main(){
  printf("Wonderful Class!\n");
  return(0);
}
```

Include the header file "mpi.h"

First example

Example1: All steps (I will demo this example in a minute)

Parallel Programming MPI, OpenMP, CUDA	MPI (1)
Choose function	main (1)
Choose pattern	MPI environment (8)
Insert ‘rank’ and ‘size’	y
More?	no
Printing	n
Reading	n
Timing	no (but try it later)

Explaining the Steps for Parallelization: example1.c to rose_example1_MPI.c

```
#include <stdio.h>
#include "mpi.h"          <-- Include the header file "mpi.h"
int main(){
  MPI_Init(NULL, NULL);   <-- Start up MPI
  MPI_Comm_size(MPI_COMM_WORLD, &rose_size)
  MPI_Comm_rank(MPI_COMM_WORLD, &rose_rank)
  printf("Wonderful Class!\n");
  MPI_Finalize();         <-- Shut down MPI
  return(0);
}
```

Compiling and Running the Program

Compile with

```
mpicc rose_example1_MPI.c
```

Run with

```
ibrun ./a.out
```

Compiling `mpiExample1.c` on Stampede2

Only MPI-2 is available (MVAPICH2 and Intel MPI)

Intel compiler available as a default option

The GCC compiler module is also available but Intel is recommended

Compiling `mpiExample1.c` on Stampede2

Compiling the example program

```
login1$ mpicc -o mpiExample1 mpiExample1.c
```

Compiler	Program	File Extension
`mpicc`	C	.c
`mpicxx`	C++	Intel: .C/c/cc/cpp/cxx/c++ PGI: .C/c/cc/cpp/cxx/
`mpif90`	F77/F90	.f, .for, .ftn, .f90, .f95, .fpp

Running mpiExample1.c

- To run your application on TACC resources
  - Please consult the user-guide and write a job script (**`myJob.sh`**)
    https://portal.tacc.utexas.edu/user-guides/stampede2
  - Submit the job to the SLURM queue

    ```
    login1$ sbatch myJob.sh
    ```
  - Remember that Stampede2 has 48 cores per Skylake node and 68 cores on the KNL nodes
    - Also note that the words "**tasks**" and "**MPI processes**" mean the same thing in this lecture

Job Script for Stampede: `myJob.sh`

```
#!/bin/bash
#SBATCH -J myMPI                    # Job Name
#SBATCH -o myMPI.o%j                # Name of the output file
#SBATCH -n 8                        # Requests 8 tasks/node
#SBATCH -N 1                        # Requests 1 node
#SBATCH -p skx-normal               # Queue name skx-normal
#SBATCH -t 01:30:00                 # Run time (hh:mm:ss) - 1.5 hours
#SBATCH -A xxxxx                    # Mention your account name (xxxxx)
set -x                              # Echo commands
ibrun ./mpiExample1                 # Run the MPI executable
```

Note : `ibrun` is a wrapper for `mpirun/mpiexec` that is exclusive to TACC resources

36

Output from `rose_example1_MPI.c`

```
login3$ cat myMPI.o2339942
…
TACC: Starting up job 2339942
TACC: Setting up parallel environment for MVAPICH ssh-based mpirun.
TACC: Setup complete. Running job script.
TACC: starting parallel tasks...
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
…
TACC: Shutting down parallel environment.
TACC: Cleaning up after job: 2339942
TACC: Done.
```

Manually print ‘size’ and ‘rank’

```
#include <stdio.h>
#include "mpi.h"
int main(){
  MPI_Init(NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &rose_size)
  MPI_Comm_rank(MPI_COMM_WORLD, &rose_rank)
  printf("Wonderful Class! ??? \n");
  MPI_Finalize();
  return(0);
}
```

Include the header file "mpi.h"

Start up MPI

Shut down MPI

Output from `mpiExample1.c`

```
login3$ cat myMPI.o2339942
…
TACC: Starting up job 2339942
TACC: Setting up parallel environment for MVAPICH ssh-based mpirun.
TACC: Setup complete. Running job script.
TACC: starting parallel tasks...
Wonderful Class! Rank 3  Size 4
Wonderful Class! Rank 2  Size 4
Wonderful Class! Rank 0  Size 4
Wonderful Class! Rank 1  Size 4
…
TACC: Shutting down parallel environment.
TACC: Cleaning up after job: 2339942
TACC: Done.
```

Lessons to Learn: Every MPI Program…

Includes the MPI header file (`mpi.h`)

Has a routine to initialize the MPI environment (`MPI_Init`)

Has a routine to terminate the MPI environment (`MPI_Finalize`)

C	Fortran
`#include "mpi.h"`	`include 'mpif.h'`

`MPI_Xxx(. . .);`	`CALL MPI_XXX(. . ., ierr)`
	
	`Call mpi_xxx(. . ., ierr)`

`MPI_Init(NULL, NULL)`
 	
`MPI_Init(&argc, &argv)`
	`MPI_INIT(ierr)`

`MPI_Finalize()`	`MPI_FINALIZE(ierr)`

2nd Example: Reduction

```
cd ../example2
```

```
#include <stdio.h>
int main(int argc,char *argv[]){
    int i, sum,upToVal;
    upToVal = 10000;
    sum = 0;
    for(i=0; i<= upToVal; i++){
        sum = sum +i;  }
    printf("\nSum of first %d numbers is: %d\n\n",upToVal, sum);
    return 0;
}
```

2nd Example: Compile and execute

```
icc example2.c
./a.out
```

Steps for Using IPT to Parallelize example2.c

Online demo

Example2: All steps (I will demo this example in a minute)

Parallel Programming MPI, OpenMP, CUDA	MPI (1)
Choose function	main (1)
Parallel region, loop, or section	loop (1)
Select loop	select first loop
Type of storage	variable (1)
Reduction variable	sum (1)
Reduction operation	addition (1)
All processes	yes (1)
Dependency analysis	select (2)
Same pattern again	no
Printing	n
Writing or reading	n
Timing	no (but try it later)

Environment Management Routines (1)

`MPI_Init` initializes the MPI execution environment, must be called before any other MPI routine is called, and is invoked only once in an MPI program

`MPI_Finalize` terminates the MPI execution environment and must be called in the last

`MPI_Comm_size` determines the number of processes (**`size`**) in a communicator (**`comm`**)

```
C: MPI_Comm_size (comm,&size)
Fortran: MPI_COMM_SIZE (comm,size,ierr)
```

Note: If `comm` is `MPI_COMM_WORLD`, then `size` is total number of processes in the program.

Environment Management Routines (2)

`MPI_Comm_rank` determines the number of processes within a communicator, ranges from 0 to N-1

```
C: MPI_Comm_rank (comm,&rank)
Fortran: MPI_COMM_RANK (comm,rank,ierr)
```

`MPI_Wtime` is a timer routine that returns elapsed wall clock time in seconds

```
C: MPI_Wtime()
Fortran: MPI_WTIME()
```