TACC
TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

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

SCEC17 Workshop

December 17, 2017

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

Outline

Introduction to IPT (prototype version used)

- What is Interactive Parallelization Tool (IPT)?

Introduction to Our Approach for Teaching Parallel Programming

Parallelizing applications using IPT (hands-on session)

- Exercise-1
- Exercise-2
- Understanding performance and speed-up
- Comparing performance of the hand-written code with the generated code for exercises 1 and 2

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

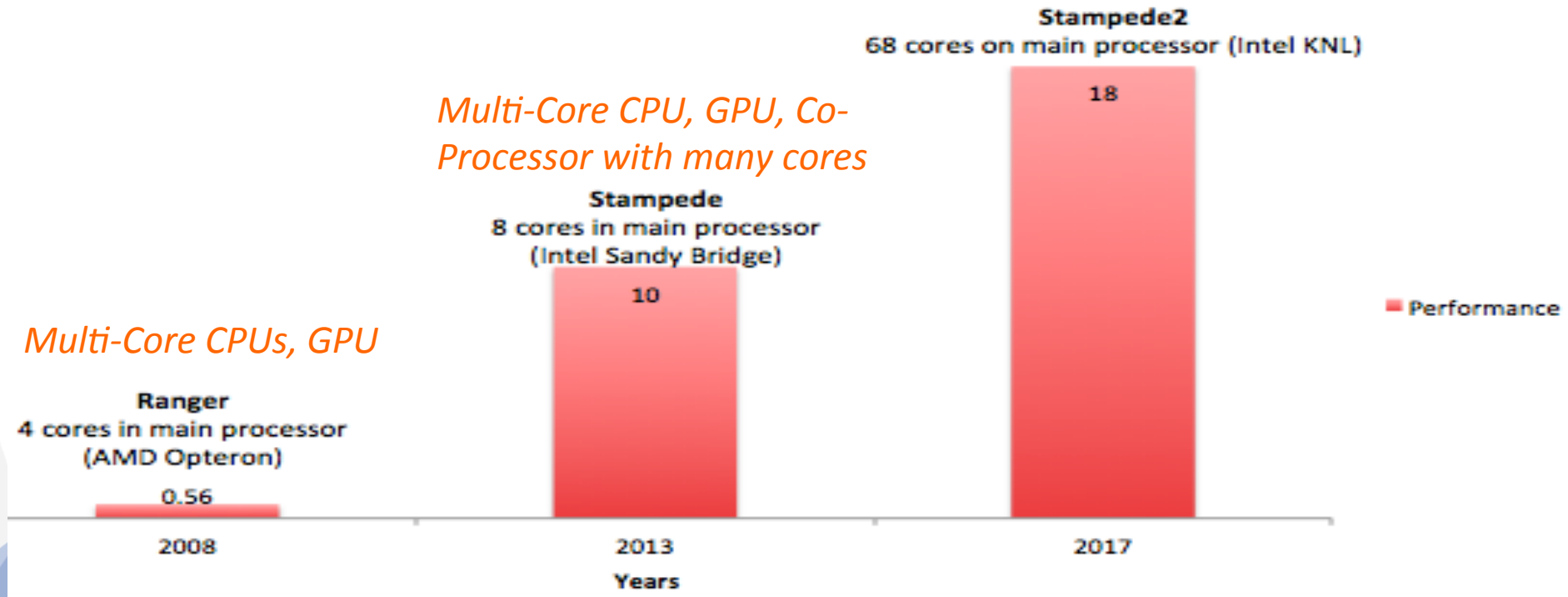
- Code modernization can be required to take advantage of the continuous advancement made in the computer architecture discipline and the programming models
  - To efficiently use many-core processing elements
  - To efficiently use multiple-levels of memory hierarchies
  - To efficiently use the shared-resources
- 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

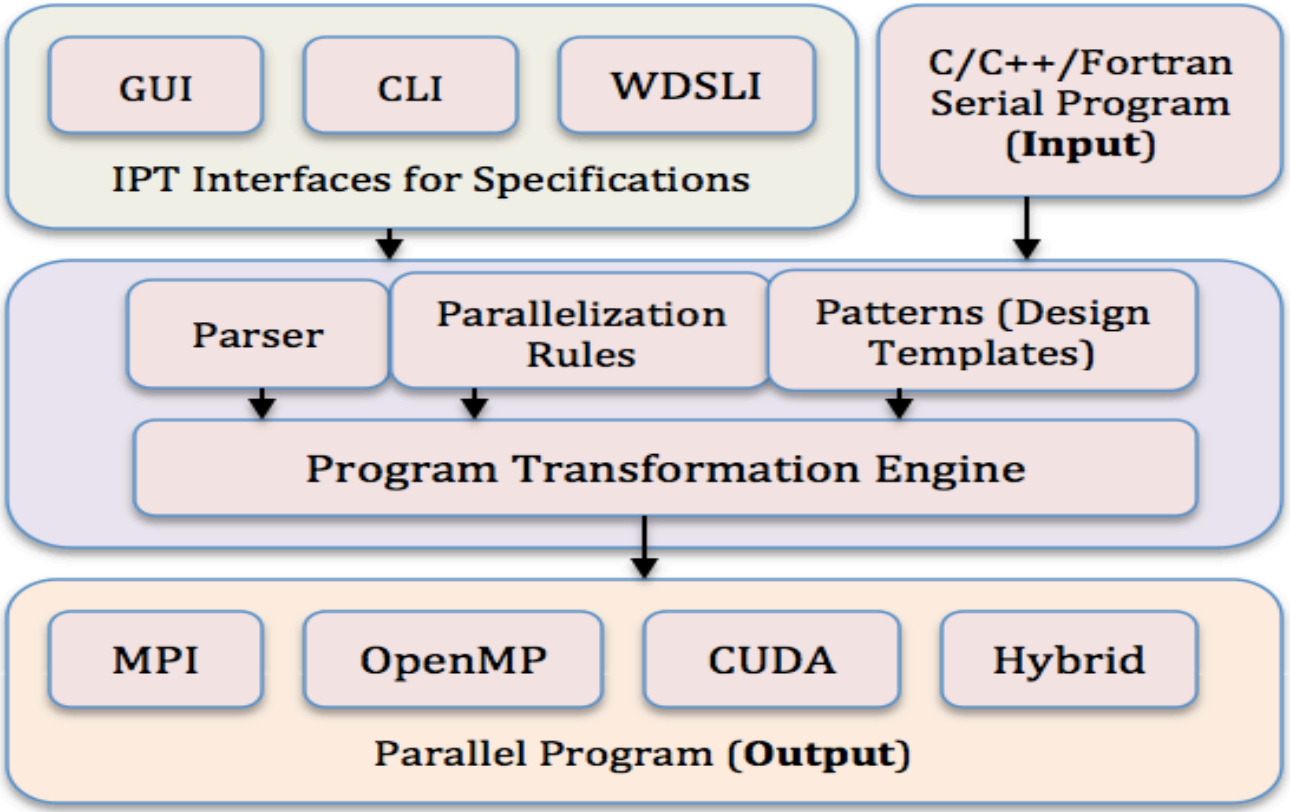


IPT – How can it help you?

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

How are we teaching Parallel Programming with IPT?

- We have classes where we introduce the concept and many details, followed by some examples

<u>The IPT training class is different</u>

- Code modification with our tool IPT
- Short introduction
- Example: serial → parallel with IPT
- Inspection of the semi-automatically parallelized code
- Learning by doing
- Focus on concepts; less important syntax taken care of by IPT
- Next example focusing on other features
- …

First: Discuss High-Level Concepts

General Concepts Related to Parallel Programming:

- Data distribution/collection/reduction
- Synchronization
- Loop dependence analysis (exercise # 2)

Must know before using IPT

Specific to OpenMP:

IPT can help with most of these

- A **structured block** having a single entry and exit point
- Threads communicate with each other by reading/writing from/to a shared memory region
- Compiler directives for creating teams of threads, sharing the work among threads, and synchronizing the threads
- Library routines for setting and getting thread attributes

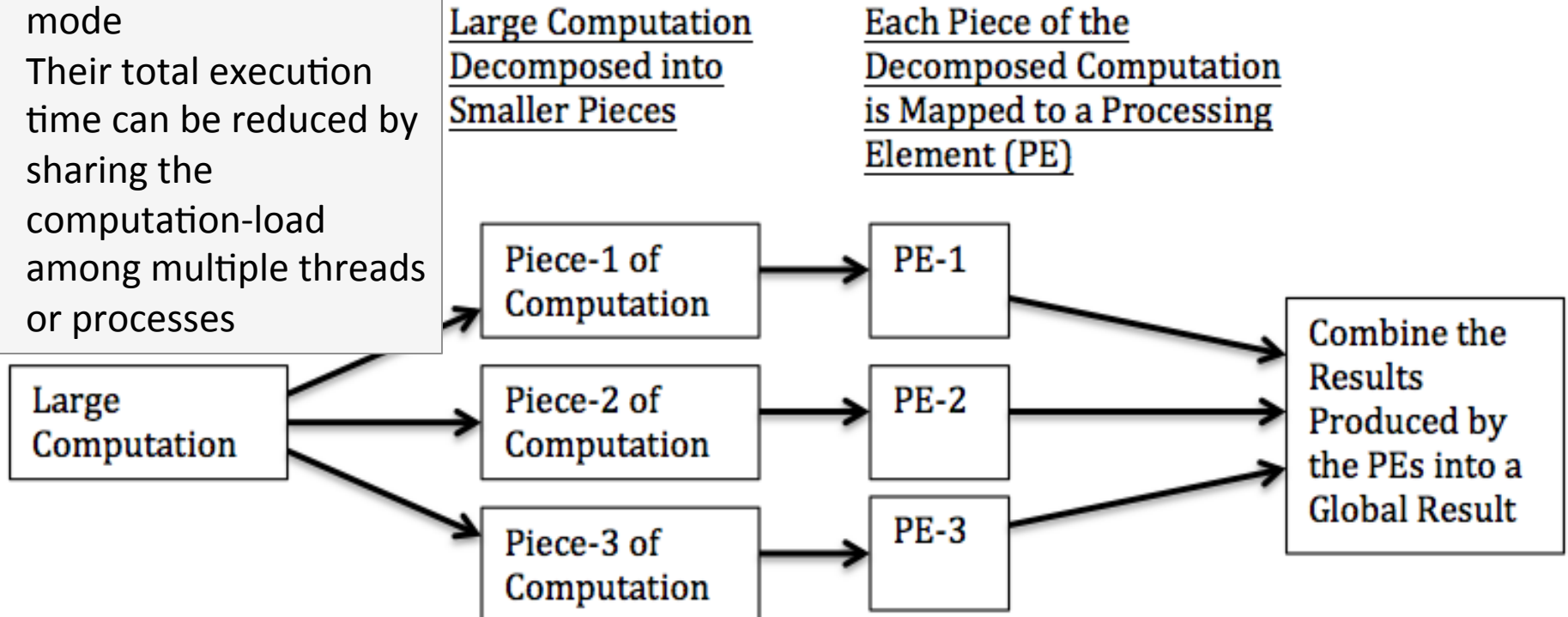
Additional Concepts Related to OpenMP:

Programmer needs to decide at run-time

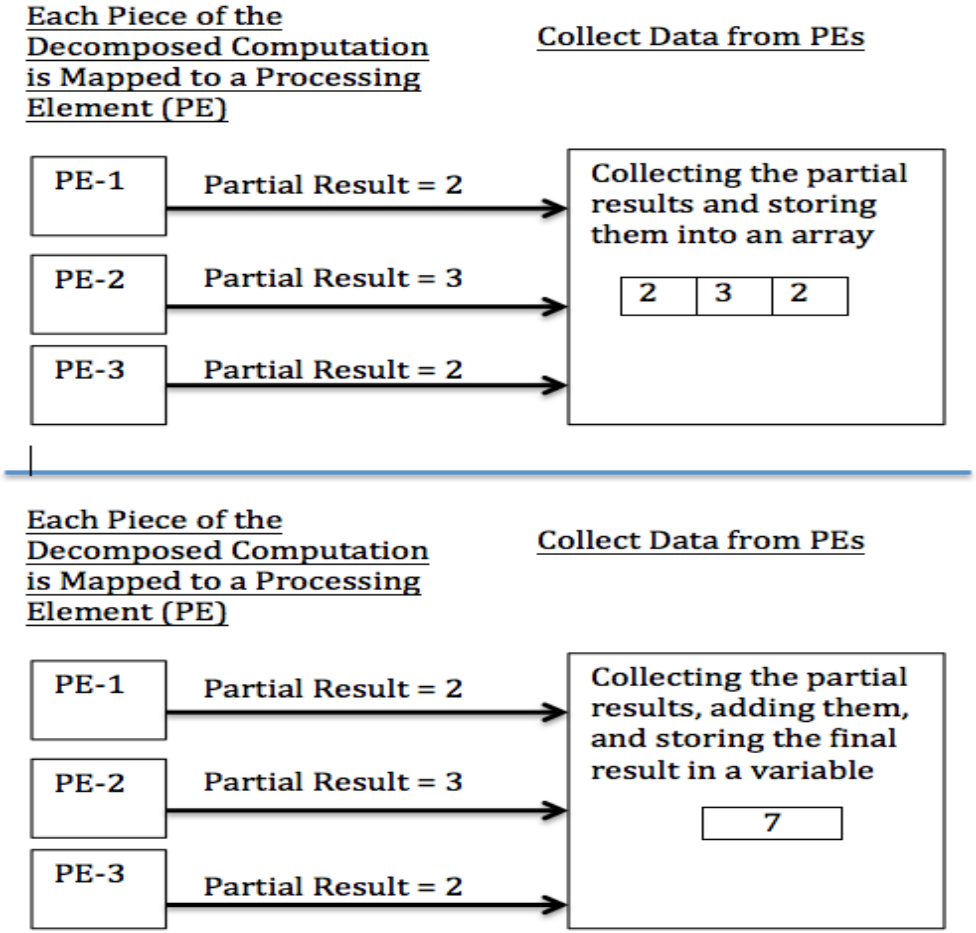
- Environment variables to control run-time behavior

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 threads or processes



Data Distribution/Collection/Reduction



Processing Element (PE) is a thread in OpenMP

Synchronization

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

- Synchronization constructs in OpenMP:

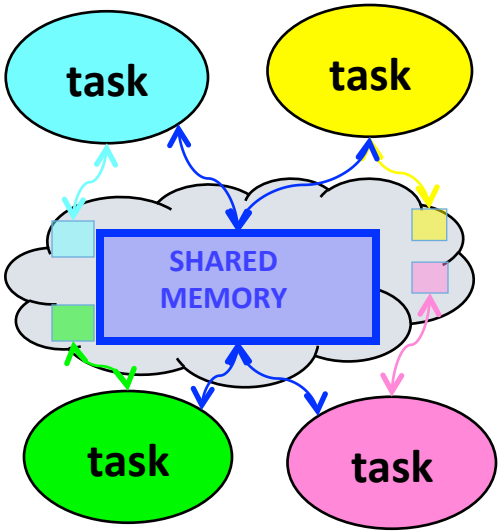
  master, single, atomic, critical, barrier, taskwait, flush, parallel {…}, ordered

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 threads or 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. One such example is example2.c

As a Second Step: Gentle Introduction to OpenMP

Shared-Data Model




Private
Memory

- Threads Execute on Cores/HW-threads
- In a parallel region, team threads are assigned (tied) to implicit tasks to do work. Think of tasks and threads as being synonymous.
- Tasks by “default” share memory declared in scope before a parallel region.
- Data: shared or private
  - Shared data: accessible by all tasks
  - Private data: only accessible by the owner task

Structured Block: Single Entry and Exit Point

OpenMP construct = Compiler Directive + Block of Code

- The block of code must must have a single entry point at the beginning, and a single exit point at the bottom, hence, it should be a structured block
  - Branching in and out of a structured block is not allowed
  - No return statements are allowed
    - exit statements are allowed though
  - Compile-time errors if the block of code is not structured

Third Step: Get Your Hands dirty with the Code but Before that, Some Heads-Up about IPT

Understanding the Questions Presented by IPT During the Parallelization Process #1

IPT analyzes the input source code, and prepares a list of the variables that are good candidates for a reduction operation at the chosen hotspot. It then prompts the user to further short-list the variables as per their needs. For example, it poses a question as follows:

```
Please select a variable to perform the reduction operation on (format 1,2,3,4 etc.).  List of
   possible variables are:

1. j type is int

2. sum type is double

2


Please enter the type of reduction you wish for variable [sum]

1. Addition

2. Subtraction

3. Min

4. Max

5. Multiplication

1
```

Understanding the Questions Presented by IPT During the Parallelization Process #2

In some cases IPT needs some information from the user while deciding whether an array should be part of the `shared` clause or `private`/`firstprivate` clause. In those cases, IPT prompts the user with a question as follows:

```
IPT is unable to perform the dependency analysis of the
  array named [ tmp ] in the region of code that you wish to
  parallelize. Please enter 1 if the entire array is being
  updated in a single iteration of the loop that you
  selected for parallelization, or, enter 2 otherwise.
```

If the user selects 1, then the array will be added to the `private`/`firstprivate` clause otherwise to the `shared` clause

Understanding the Questions Presented by IPT During the Parallelization Process #3

There may be some regions of the code that a user may want to run with one thread at a time (`critical` directive) or with only one thread in the team of threads (`single` directive). To understand such requirements of the user, IPT asks the following question:

```
Are there any lines of code that you would
  like to run either using a single thread
  at a time (hence, one thread after
  another), or using only one thread?(Y/N)
```

Errors and Bugs

While using IPT, if you see an error like the following one, then this means that you missed sourcing the file for setting the library paths:

```
c557-903$ ../../IPT matrix_mul.cc
../../IPT: error while loading shared libraries:
  librose.so.0: cannot open shared object file: No such
  file or directory
```

To fix this error:

```
c557-903$ source ../../runBeforeIPT.sh
```

Errors and Bugs # 2

The following error message indicates that the Intel compiler is not available in your user environment

```
c557-903$ icpc -qopenmp -o rose_heat_serial_OpenMP
  rose_heat_serial_OpenMP.c
-bash: icpc: command not found
```

Fix:

```
c557-903$ ml intel
```

Error and Bugs # 3

- The prototype version of IPT that is being used for today's training has only limited features
  - Code for supporting sections and parallel regions without for-loops is turned off
  - Scheduling, locks, reduction of array elements in C/C++ (supported by the OpenMP 4.5 spec) is not available in the current prototype.
  - Limited set of reduction-identifier/operation supported currently

- Using this version to parallelize *the region of code containing dynamically allocated arrays* is very likely to produce incorrect output code

Hand-on Session/Demo of IPT

Accessing Files for the Exercises

Log on to Stampede using **your_login_name**

This would be your TACC portal account user name

```
ssh <your_login_name>@stampede.tacc.utexas.edu

cds

mkdir trainingIPT

cd trainingIPT

cp -r /work/01698/rauta/trainingIPT/* .

idev
```

First Things First (1)

Logon to Stampede

How to logon to Stampede:

- You all should have a TACC portal account or the training account provided to you
  - Use this account name and password to logon
- Open a terminal on your computer (MacOS or Linux)

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

- Use Putty on a Windows laptop

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 -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 /work/01698/rauta/trainingSCEC .`
- 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 ./runBeforeIPT.sh`
- `$ module load intel`

First example (1)

Let's start with example 1

- `$ cd exercises`
- `$ cd exercise1`
- `$ ls -al`

- There is a Fortran file and a C file
- For now IPT only works with C/C++ (but you can look at the Fortran source, if that is more convenient)

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?

First example (3)

Example1

2 arrays: x and y

Loop 1 initializes x

Loop 2: Stencil update

1. $y_{i,j}$ is calculated from $x_{i,j}$
2. A temporary variable is being used
3. The sum of all elements is calculated

The latter, i.e., calculation (3) is called a reduction

```
#include <stdio.h>
#include <sys/time.h>
#define N 30000
int main(){
  int i, j;
  double x[N+2][N+2], y[N+2][N+2], sum, tmp;

  //for timing the code section
  struct timeval start,end;
  float delta;
  for(i=0; i<=N+1; i++){
    for(j=0; j<=N+1; j++){
      x[i][j] = (double) ((i+j)%3) - 0.9999;
    }
  }
  printf("\nMemory allocation done successfully\n");
  //start timer and calculation
  gettimeofday(&start, NULL);

 for(j=1; j<N+1; j++){
   for(i=1; i<N+1; i++ ){
     tmp = 0.2 * (x[i][j] + x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1]);
     y[i][j] = tmp;
     sum = sum + tmp;
   }
  }

  //stop timer and calculation
  gettimeofday(&end, NULL);
  delta = ((end.tv_sec-start.tv_sec)*1000000u + end.tv_usec-start.tv_usec)/1.e6;
  printf("\nThe total sum is: %lf\n", sum);
  //print time to completion
  printf("run time    = %fs\n", delta);
  return 0;
}
```

Loop 1

Loop 2

First example (4)

Example1

Running IPT: essentials

- Parallelize loop #2
- Instruct IPT to add a reduction
  1. Reduction variable: sum
  2. Reduction operation: add

```
#include <stdio.h>
#include <sys/time.h>
#define N 30000
int main(){
  int i, j;
  double x[N+2][N+2], y[N+2][N+2], sum, tmp;

  //for timing the code section
  struct timeval start,end;
  float delta;
  for(i=0; i<=N+1; i++){
     for(j=0; j<=N+1; j++){
        x[i][j] = (double) ((i+j)%3) - 0.9999;
     }
  }
  printf("\nMemory allocation done successfully\n");
  //start timer and calculation
  gettimeofday(&start, NULL);

 for(j=1; j<N+1; j++){
    for(i=1; i<N+1; i++ ){
      tmp = 0.2 * (x[i][j] + x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1]);
      y[i][j] = tmp;
      sum = sum + tmp;
    }
  }

  //stop timer and calculation
  gettimeofday(&end, NULL);
  delta = ((end.tv_sec-start.tv_sec)*1000000u + end.tv_usec-start.tv_usec)/1.e6;
  printf("\nThe total sum is: %lf\n", sum);
  //print time to completion
  printf("run time    = %fs\n", delta);
  return 0;
}
```

Loop 1

Loop 2

First example (5)

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

Parallel Programming MPI, OpenMP, CUDA	OpenMP (2)
Choose function	main (1)
Parallel region, loop, or section	loop (2)
Select loop	select third loop
Reduction	yes
Reduction variable	sum (3)
Reduction operation	addition (1)
Dependency analysis	select (2)
Single thread	no
Another loop	no
Printing	no

First example (6)

Example1: On screen demo

First example (7)

Example1: Parallelized code

`rose_example1_OpenMP.c`

Let's compile and run it

Compile: `icc -qopenmp rose_example1_OpenMP.c`

Select number of threads: `export OMP_NUM_THREADS=4`
Execute: `./a.out`

First example (8)

Example1: Let's check the performance

Run the code with different numbers of threads and report the timing

```
export OMP_NUM_THREADS=1;   ./a.out
export OMP_NUM_THREADS=2;   ./a.out
export OMP_NUM_THREADS=4;   ./a.out
export OMP_NUM_THREADS=8;   ./a.out
export OMP_NUM_THREADS=16;  ./a.out
export OMP_NUM_THREADS=32;  ./a.out
```

First example (8)

Example1: Let's check the performance

Run the code with different numbers of threads and report the timing

```
export OMP_NUM_THREADS=1;   ./a.out        38.3
export OMP_NUM_THREADS=2;   ./a.out        19.5
export OMP_NUM_THREADS=4;   ./a.out        16.9
export OMP_NUM_THREADS=8;   ./a.out        16.4
export OMP_NUM_THREADS=16;  ./a.out        11.0
export OMP_NUM_THREADS=32;  ./a.out         9.3
```

First example (9)

Example1: Let's inspect the parallel version of the code

1. Header file in all routines with OpenMP content
2. Parallel region
   1. Threads are spawned
   2. All code within the curly brackets **{}** is executed by all threads
3. Worksharing for following loop (j loop)
   1. Worksharing = every thread is executing a different chunk of the loop

```
#include <omp.h>
#include <stdio.h>
#include <sys/time.h>
#define N 30000

int main()
{
  int i;
  int j;
  double x[30002UL][30002UL];
  double y[30002UL][30002UL];
  double sum;
  double tmp;
//for timing the code section
  struct timeval start;
  struct timeval end;
  float delta;
  for (i = 0; i <= 30000 + 1; i++) {
    for (j = 0; j <= 30000 + 1; j++) {
      x[i][j] = (((double )((i + j) % 3)) - 0.9999);
    }
  }
  printf("\nMemory allocation done successfully\n");
//start timer and calculation
  gettimeofday(&start,0);

#pragma omp parallel default(none) shared(sum,x,y) private(j,i,tmp)
{

#pragma omp for reduction ( + :sum)
  for (j = 1; j < 30000 + 1; j++) {
    for (i = 1; i < 30000 + 1; i++) {
      tmp = (0.2 * ((((x[i][j] + x[i - 1][j]) + x[i + 1][j]) + x[i][j - 1]) + x[i][j + 1]));
      y[i][j] = tmp;
      sum = (sum + tmp);
    }
  }
}
…
```

1: OpenMP header file

This code is now OpenMP parallel

2: Parallel region

3: Worksharing

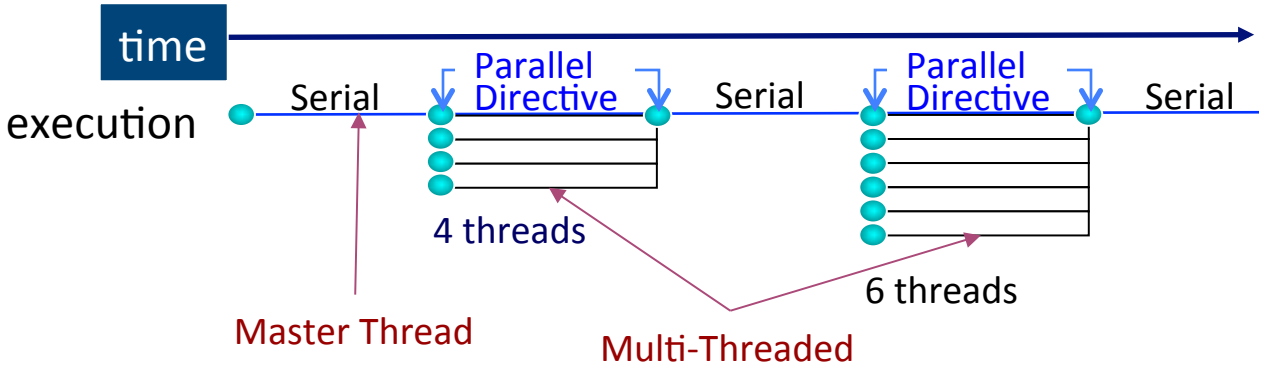
First example (10)

Example1: Look at the 2 OpenMP statements in the parallel code

```
#pragma omp parallel default(none) shared(sum,x,y) private(j,i,tmp)
{
#pragma omp for reduction ( + :sum)
for … {
…
}
```

Execution Model

- Programs begin as a single process: master thread
- Master thread executes in serial mode until the parallel construct is encountered
- After executing the statements in the parallel region, team threads synchronize and terminate (join) but master continues



OpenMP Syntax

Compiler directive syntax:

```
#pragma omp construct [clause [[,]clause]…]        C/C++
!$omp construct [clause [[,]clause]…]              F90
```

Example

Fortran

```
print*,"serial"

!$omp  parallel num_threads(4)
     …
!$omp end parallel

print*,"serial"
```

C/C++

```
printf("serial\n");

#pragma omp parallel num_threads(4)
{
      …
}

printf("serial\n");
```

Parallel Region & Worksharing

Use OpenMP directives to specify Parallel Region & Worksharing constructs

```
parallel

end parallel
```

Code block	Each Thread Executes
do / for	Worksharing
sections	Worksharing
single	Worksharing (one thread)

Sentinels "!$omp" and "#pragma omp") not shown here

Work-sharing Directives assign threads to units of work.
There is an implied barrier at the end of a worksharing construct!

C/C++

Worksharing: Loop

```
1  #pragma omp parallel
2  {
3    #pragma omp for
4    for (i=0; i<N; i++)
5    {
6      a[i] = b[i] + c[i];
7    }
8  }
```

Or

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

Line 1 Team of threads formed (parallel region).

Line 3-7 Loop iterations are split among threads.
implied barrier at }

Each loop iteration must be independent of other iterations.

First example (11)

Example1: How about the clauses in the 'omp parallel' statement?

```
#pragma omp parallel default(none) shared(sum,x,y) private(j,i,tmp)
{
…
}
```

Private variables are used to avoid race conditions

Every thread needs a private copy of
- loop indices i and j
- Scalar variable tmp

The input and output arrays are shared
- arrays: x and y

Every thread is writing to a different
- array element

```
#pragma omp parallel default(none) shared(x,y) private(j,i,tmp)
{

#pragma omp for reduction ( + :sum)
  for (j = 1; j < 30000 + 1; j++) {
    for (i = 1; i < 30000 + 1; i++) {
      tmp = (0.2 * ((((x[i][j] + x[i - 1][j]) + x[i + 1][j]) + x[i][j - 1]) + x[i][j + 1]));
      y[i][j] = tmp;
      sum = (sum + tmp);
    }
  }
}
…
```

Private Data Example

- In the following loop, each thread needs its own private copy of temp
- If temp were shared, the result would be unpredictable since each thread would be writing/reading to/from the same memory location

```
#pragma omp parallel for shared(a,b,c,n) private(temp,i)
for (i=0; i<n; i++){
    temp = a[i] / b[i];
    c[i] = temp + cos(temp);
}
```

- A **lastprivate(temp)** clause will copy the last loop(stack) value of temp to the (global) temp storage when the parallel for is complete.
- A **firstprivate(temp)** would copy the global temp value to each stack's temp.

First example (12)

Example1: How about the clauses in the 'omp for' statement?

```
#pragma omp for reduction (+:sum)
for …
{
…
}
```

Reduction variables store the local results of each thread. The local results are combined with a reduction operation to produce a global result.

A reduction is performed in the loop

Variable sum is updated by all threads

```
#pragma omp parallel default(none) shared(x,y) private(j,i,tmp)
{

#pragma omp for reduction ( + :sum)
  for (j = 1; j < 30000 + 1; j++) {
    for (i = 1; i < 30000 + 1; i++) {
      tmp = (0.2 * ((((x[i][j] + x[i - 1][j]) + x[i + 1][j]) + x[i][j - 1]) + x[i][j + 1]));
      y[i][j] = tmp;
      sum = (sum + tmp);
    }
  }
}
…
```

C/C++ Reduction

- Operation that combines multiple elements to form a single result
- A variable that accumulates the result is called a reduction variable
- In parallel loops reduction operators and variables must be declared

```
float asum=0.0, aprod=1.0;

#pragma omp parallel for reduction(+:asum) reduction(*:aprod)
  for (i=0; i<n; i++){
    asum  = asum  + a[i];
    aprod = aprod * a[i];
  }
```

Each thread has a private **asum** and **aprod**, initialized to the operator's identity

- **After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction**

First example (13)

Example1: Environment variable OMP_NUM_THREADS

```
export OMP_NUM_THREADS=4; ./a.out
```

OMP_NUM_THREADS sets the default number of threads

There are other ways to change the number of threads

Function call within code: `omp_set_num_threads(8)`

Clause at the omp parallel statement

```
#pragma omp parallel numthreads(8)
```

OpenMP Environment Variables

variable	description
OMP_NUM_THREADS=*integer*	Set to default no. of threads to use
OMP_SCHEDULE="*schedule-type[, chunk_size]*"	Sets "runtime" in loop schedule clause: "…omp for/do schedule(runtime)"
OMP_DISPLAY_ENV=*anyvalue*	Prints runtime environment at beginning of code execution.

[…] = optional

Second example (1)

Let's start with example 2

- `$ cd ..`
- `$ cd exercise2`
- `$ ls -al`

- There is a Fortran file and a C file
- For now IPT only works with C/C++ (but you can look at the Fortran source, if that is more convenient)

If your idev session has expired:
Look at previous slides and start a new session
Do not forget to source the runBeforeIPT.sh script, etc.

Second example (2)

Continue with example 2

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

```
$ more example2.c
$ vi example2.c
$ emacs example2.c
```

IPT will ask you all the usual questions

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

Second example (3)

Can you guess why this loop cannot be so easily parallelized?

```
for(j=1; j<N+1; j++){
   for(i=1; i<N+1; i++ ){
      tmp[i][j] = 0.167 * (x[i][j] + x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1] + y[i+1][j]);
      y[i][j] = tmp [i][j];
      sum = sum + tmp[i][j];
   }
 }
```

Second example (3)

But example 2 requires some code modifications

- Inside the loop, $y_{i,j}$ is updated (same as in example 1)
- However, the right-hand side refers also to $y_{i+1,j}$
- Loops can be parallelized easily when loop iterations are independent
  - e.g., when you can execute the loop iterations in any order

```
for(j=1; j<N+1; j++){
   for(i=1; i<N+1; i++ ){
      tmp[i][j] = 0.167 * (x[i][j] + x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1] + y[i+1][j]);
      y[i][j] = tmp [i][j];
      sum = sum + tmp[i][j];
   }
 }
```

Second example (4)

Code modifications

- In this loop the loop iterations are not independent
- Solution: Create 2 loop nests
  - The first one to calculate the temporary array (tmp)
  - The second one to copy tmp into y and to calculate the sum
- Then parallelize both loops separately

```
 for(j=1; j<N+1; j++){
    for(i=1; i<N+1; i++ ){
       tmp[i][j] = 0.167 * (x[i][j] + x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1] + y[i+1][j]);
}
}
 for(j=1; j<N+1; j++){
    for(i=1; i<N+1; i++ ){
       y[i][j] = tmp [i][j];
       sum = sum + tmp[i][j];
}
}
```

Second example (5)

Example 2: Serial source code

Loop 2: Stencil update

1. Split the loop into 2
   - This is done by you, not by IPT
2. Run the code through IPT
3. Parallelize both loops

```
#include <stdio.h>
#include <sys/time.h>


#define N 30000

int main(){
  int i, j;
  double x[N+2][N+2], y[N+2][N+2], tmp[N+2][N+2];
  double sum=0;

  //for timing the code section
  struct timeval start,end;
  float delta;

  for(i=0; i <= N+1; i++){
     for(j=0; j <= N+1; j++){
         x[i][j] = (double) ((i+j)%3) - 0.9999;
         y[i][j]= x[i][j] + 0.0001;
     }
  }

  //start timer and calculation
  gettimeofday(&start, NULL);

  for(j=1; j<N+1; j++){
    for(i=1; i<N+1; i++ ){
       tmp[i][j] = 0.167 * (x[i][j] + x[i-1][j] + x[i+1][j] +
                            x[i][j-1] + x[i][j+1] + y[i+1][j]);
       y[i][j] = tmp [i][j];
       sum = sum + tmp[i][j];
    }
  }
  //stop timer and calculation
  gettimeofday(&end, NULL);
  delta = ((end.tv_sec-start.tv_sec)*1000000u +
                                    end.tv_usec-start.tv_usec)/1.e6;
  printf("\nThe total sum is: %lf\n", sum);
  //print time to completion
  printf("run time    = %fs\n", delta);
  return 0;
}
```

Loop 2

Second example (8)

Example 2: Let's check the performance

Run the code with different numbers of threads and report the timing

```
export OMP_NUM_THREADS=1;   ./a.out
export OMP_NUM_THREADS=2;   ./a.out
export OMP_NUM_THREADS=4;   ./a.out
export OMP_NUM_THREADS=8;   ./a.out
export OMP_NUM_THREADS=16;  ./a.out
export OMP_NUM_THREADS=32;  ./a.out
```

How about using the information learnt thus far to generate MPI or CUDA code?

Ritu will present a demo.

Using IPT from a Web Browser

- IPT will be made available through a science gateway so that you can use it to generate parallel programs through a web browser – NSF funded Agave project and the Science Gateway Community Institute (SGCI) project are providing the plumbing and the resources

- Generated parallel programs could be compiled and run on XSEDE resources or could be downloaded to run on local systems

- URL: https://ipt.tacc.cloud

Early User Group

- If you wish to be part of our early user group for IPT, we would love to connect with you.

- Please feel free to email us for more information on the public release of IPT(rauta@tacc.utexas.edu, lars@tacc.utexas.edu)

References

1. OpenMP API specification for parallel programming: http://www.openmp.org/

2. Ritu Arora, Julio Olaya, and Madhav Gupta. 2014. A Tool for Interactive Parallelization. In Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment (XSEDE '14). ACM, New York, NY, USA, , Article 51 , 8 pages. DOI: http://dx.doi.org/10.1145/2616498.2616558

3. Video-demo of parallelizing a Molecular Dynamics Code using IPT: https://www.youtube.com/watch?v=JH7o_k9Bxd0

4. Stampede User-Guide: https://portal.tacc.utexas.edu/user-guides/stampede#cluster-modes

Thank You!

We are grateful for the support received through:

- NSF Grant # 1642396
- NSF Grant # 1359304
- TACC STAR Scholars program
- Extreme Science and Engineering Discovery Environment (XSEDE) - NSF grant # ACI-105357
- NSF Science Gateway Community Institute