Learning OpenMP with the Interactive Parallelization Tool (IPT)

1 Introduction to IPT 1

2 Introduction to OpenMP 2

3 Parallelizing single for-loops 3

3.1 Parallelizing a Hello World Program 3

3.2 Parallelizing the Code for Heat Diffusion Equation 7

3.3 Parallelizing a For-Loop – Introduction to Reduction 13

3.4 Loops with dependencies 19

4 Additional Examples 20

4.1 Molecular Dynamics (MD) 20

4.2 Circuit Satisfiability 25

5 Nested For-loop Parallelization 27

6 Critical and Single Directives 30

6.1 Critical Regions 31

6.2 Atomic Directives 32

6.3 Single Directives 32

7 Practice Examples 33

7.1 Fast Fourier Transform 33

7.2 Game of Life 34

8 Loop Scheduling 35

8.1 Static 35

8.2 Dynamic 35

8.3 Guided 35

8.4 Runtime 35

9 Printing 36

10 Timing 38

1 Introduction to IPT

The **Interactive Parallelization Tool (IPT)** is a high-productivity tool for semi-automatically parallelizing existing serial programs written in C/C++. It supports parallelization with the three most popular parallel programming paradigms: OpenMP, MPI, and CUDA. The users interactively specify what to parallelize and where in their serial programs. On the basis of the specifications provided by the users, IPT can parallelize multiple types of C/C++ applications. It frees the users from the burden of learning the low-level syntax of the different parallel programming paradigms, and in manually reengineering their serial code for creating parallel versions. IPT walks the users through a series of questions related to the parallelization of the input C/C++ programs and generates parallel versions.

IPT is currently installed on the Stampede2 system at TACC. It is deployed in the cloud and can be accessed from the convenience of a web browser. The following URL has the prototype version of the IPT website: https://129.114.16.78:8443/. IPT is also available as a Docker image, and if you are interested in using this image, please send an email at rauta@tacc.utexas.edu with "IPT Docker image" in the subject line.

Before running IPT on Stampede2, users should switch to the $SCRATCH filesystem, would need to copy some code in their accounts and set up their environment. The script `sourceME.sh` that is provided in the directory at the following path on Stampede2 should be run in every new shell before starting IPT: `/scratch/01698/rauta/PEARC18_OMP`

```
$ cds; cp -r /scratch/01698/rauta/PEARC18_OMP .

$ source ./runBeforeIPT.sh
```

This script will either need to be run in each `idev` session, or it can be run once on the login node from which the `idev` command is run to start the interactive session.

IPT can then be invoked by running the following command after replacing the text "`<provide-the-path-and-name-of-the-source-code-file>`" with the actual name of the file containing code:

```
$ IPT <provide-the-path-and-name-of-the-source-code-file>
```

When using IPT, in addition to identifying the hotspots for parallelization, the users are also expected to know some concepts related to parallel programming and the programming model that they wish to select for parallelization. The key concepts that the users should know while parallelizing a code with IPT – such as, data dependency and reduction – are explained in this user-guide along with the examples.

2 Introduction to OpenMP

OpenMP (Open Multi-Processing) is an API for parallelizing C, C++, and Fortran programs for shared-memory platforms (that is, the hardware platforms consisting of multicore and manycore processors) and is supported by the commonly used C/C++/Fortran compilers. It consists of:

(1) compiler directives,
(2) library routines, and
(3) environment variables for explicitly parallelizing the regions of the programs that should be run using multiple threads.

The key steps in OpenMP parallelization include modifying the serial programs to:

(1) create group/s of threads,
(2) distribute the work to be done (computations) across the threads in the group/s (this is also known as **worksharing**), and

(3) ensure synchronization amongst threads so that correct results are produced.

Users are required to set the desired number of OpenMP threads before running the OpenMP programs. They have the option to either set the number of threads in the code (by using some predefined OpenMP functions), or at run-time by setting an environment variable.

3 Parallelizing single for-loops

A for-loop in which, the computations done in an iteration are independent of the computations done in the previous or next iterations, is a potential candidate for parallelization. In this section we will discuss the parallelization of single for-loops, that is, the for-loops that are not nested.

3.1 Parallelizing a Hello World Program

In this exercise, we will use IPT to parallelize a code named `example0.c`. This code is shown in Listing 1 and prints "Hello class!" 16 times using a for-loop.

To begin with, let us run the following command to invoke IPT and provide the path to `example0.c`:

```
$ IPT ./examples/example0.c
```

Listing 1: Code in `example0.c`

```
#include <stdio.h>
int main(){
   int i;
   for (i=0; i<16; i++){
      printf("Hello class!\n");
   }
   return 0;
}
```

Figure 1 shows the complete process of parallelizing `example0.c` (shown in Listing 1), after IPT is invoked by the aforementioned command. As and when needed, IPT requests for user-input during the parallelization process. At first, it asks the user to select a parallel programming model. IPT then analyzes the input code provided and requests further input from the user. It presents a list of non-standard library functions in the code, and prompts the user to select the function in which parallelization is required. The responses to the questions posed by IPT are shown in boldface in Figure 1.

For OpenMP programs, the users are required to select the desired type of worksharing amongst the threads, such as for-loop parallelization, or creating sections (consecutive regions) of the code that would be distributed amongst threads. The concept of worksharing is explained further in the below in this section.

After the steps shown in Figure 1, IPT generates a file with the parallel version of the source code. After parallelizing `example0.c`, IPT generates a file named `rose_example0_OpenMP.c.` The code in this file should be inspected to understand the changes made by IPT during the parallelization process. The code snippets from the serial and parallel versions of this test case are shown side-by-side in Figure 2 for comparison. IPT inserted the code for including t**he header file ("`omp.h`"). This header file is required in every OpenMP program**. IPT also inserted two compiler directives that are shown in boldface in Figure 2.

```
NOTE: We currently support only C and C++ programs.

Please select a parallel programming model from the following available options:
```

```
1. MPI
2. OpenMP
3. CUDA
2
```

OpenMP is selected

```
NOTE: As per the OpenMP standard, the parallel regions can have only one entry point,
and only one exit point.
Branching out or breaking prematurely from a parallel region is not allowed.
Please make sure that there are no return/break statements in the region selected for
parallelization. However, exit/continue statements are allowed.

A list containing the functions in the input file will be presented, and you may want
to select one function at a time to parallelize it using multi-threading.
Please choose the function that you want to parallelize from the list below
1 : main
1
```

Main function

```
Please select one of the following options (enter 1 or 2 or 3)
1. Create a parallel region (a group of threads will be created and each thread will
execute a block of code redundantly but in parallel)
2. Parallelize a for-loop (a group of threads will be created and each thread will
execute a certain number of iterations of a for-loop)
3. Create a parallel section (TBD - this mode is unavailable)
2
```

Parallelize a loop

```
Loop
Note: With your response, you will be selecting or declining
the parallelization of the outermost for-loop in the code region shown below. If
instead of the outermost for-loop, there are any inner for-loops in this code region
that you are interested in parallelizing, then, you will be able to select those at a
later stage.

  for (i = 0; i < 16; i++) {
    printf("Hello class!\n");
  }
```

IPT located the loop for parallelization

```
Is this the for-loop you are looking for?(y/n)
y

Reduction variables are the variables that should be updated by the OpenMP threads and
then accumulated according to a mathematical operation like sum, multiplication, etc.
Do you want to perform reduction on any variable ?(Y/N)
n

Do you want to use any scheduling scheme for this loop (y/n) ?
n
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)
n

Would you like to parallelize another loop?(Y/N)
n

Are you writing/printing anything from the parallelized region of the code?(Y/N)
n

Do you want to perform any timing (Y/N) ?
n
```

Figure 1. Parallelizing `example0.c` using IPT – "Hello Class" Program

Serial Version	**Parallel Version**

```
#include <stdio.h>
int main(){
   int i;
   for (i=0; i<16; i++){
      printf("Hello class!\n");
   }
   return 0;
}
```

```
#include <omp.h>
#include <stdio.h>
int main()
{
  int i;
```

Starting a team of threads

```
#pragma    omp    parallel    default(none)
private(i)
{
```

Declaring for-loop work-sharing construct

```
#pragma omp for
  for (i = 0; i < 16; i++) {
    printf("Hello class!\n");
  }
}
  return 0;
}
```

Figure 2. Comparing the serial and parallel version of `example0.c`

The first directive or pragma statement (**`#pragma omp parallel`**) inserted by IPT in the generated code is used to create a parallel region - a group of threads is formed (or forked) for executing the for-loop in parallel. This pragma statement has additional parts after the keyword **`parallel`** which are: **`default(none) private(i)`**. These are called ***clauses*** and ***are used to define the data-sharing properties of the variables***.

The **`default (none`**) clause specifies that none of the variables in the parallel region should be assigned any data-sharing attribute implicitly. The **`private(i)`** clause specifies that each thread in the group created using the **`parallel`** directive should have its own private copy of the variable **`i`** that is mentioned inside the round-brackets. ***OpenMP initializes the private variables to 0.***

Before looking into the second pragma statement or directive, let us introduce the concept of worksharing. **Worksharing** means that a task is broken into smaller units of work such that each unit of work is shared among a team of threads. In this example, the work of printing "Hello class" 16 times will be divided among the threads involved in running the program. Such a division of work amongst concurrently executing threads has the potential of reducing the overall run-time of a program. In other words, a multi-threaded program with worksharing has the potential of executing faster than running a single-threaded program.

The second directive statement in the code shown in Figure 2 - **`#pragma omp for -`** declares a **`for`** worksharing construct in the parallel region. The variables in the for-loop decorated with this second directive have the same data-sharing properties as defined in the first directive for forming the team of threads (that is, **`#pragma omp parallel`**). In addition to **`for`**, the other worksharing constructs in C/C++ are: **`sections`**, and **`single`**.

During the parallelization process of **`example0.c`** with IPT, we specified that we are not writing/printing anything from the parallelized region:

```
Are you writing/printing anything from the parallelized region of the code?(Y/N)
```

```
n
```

We lied here because we wanted that all the threads share the task of printing/writing - this is the only task that is being performed in this program. However, there can be programs in which the user may want to print the output using only one thread or control which thread executes the print statement. While parallelizing such programs, the user should respond `'y'` to the printing prompt.

To run and compile the parallelized version of the program shown in Figure 2, follow the commands in Figure 3.

Compile the code with the command below:

```
$ icc -qopenmp -o rose_example0_OpenMP rose_example0_OpenMP.c
```

Run the code with the commands shown below:

```
$ export OMP_NUM_THREADS=4
$ ./rose_example0_OpenMP
```

Figure 3. Commands to compile and run the OpenMP program

Compiling the OpenMP program: For this example, IPT generated a file with OpenMP code and named it as `rose_example0_OpenMP.c`. This file is named according to the input serial file and the programming model selected by the user. For example, if the user chooses OpenMP for parallelization, the output will be saved as `rose_<input file here>_OpenMP`. When compiling the program with the Intel compiler, the flag `-qopenmp` must be added to the compile command so that the compiler does not ignore the pragma statements added by IPT. If this flag is not used, then the compiler will ignore the pragmas, and the code will execute like a serial program. If compiling with the gcc compiler, the flag to be used is `-fopenmp` instead of `-qopenmp`.

Running the OpenMP program: Before running the program, we should set the environment variable named **`OMP_NUM_THREADS`** for controlling the number of threads using which the program should run. The export command shown in Figure 3 sets **`OMP_NUM_THREADS`** to 4 in the shell. Unless reset, all the OpenMP programs running in this shell will fork 4 threads in the parallel region. As can be seen from Figure 3, OpenMP programs are run just like serial programs.

The program output is shown in Figure 4. As expected, the program prints "`Hello class!`" sixteen times. We can verify that the output from the parallelized version is the same as the serial version.

The key benefit of parallelization is the reduction in overall run-time of the program. To measure the time that the program (or certain statements in the program) takes to run, the timing statements can be inserted in the code itself. Alternatively, the program can be run using the `time` command for measuring the overall run-time: `time ./rose_example0_OpenMP`

In Table 1, we show the time taken to run the OpenMP version of this test case with various numbers of threads. Contrary to the expectation, the overall run-time of the parallel program seems to be increasing by increasing the number of threads, and the parallel version is slower than the serial version. The program was about twice as fast when increasing the number of threads from 1 to 2. However, after 2 threads, the speed decreased instead of continuing to increase. This happens mainly because the threads are not involved in doing any significant computation. They are only writing messages to the output buffer. In contrast to the serial program, the *parallel program spent additional time in forking the group* *of threads than in doing any computation. Multiple threads compete with each other to print the message to the output buffer and slow-down each other during the process.* This is called **parallel slowdown**.

```
Hello class!
Hello class!
Hello class!
Hello class!
Hello class!
Hello class!
Hello class!
Hello class!
Hello class!
Hello class!
Hello class!
Hello class!
Hello class!
Hello class!
Hello class!
Hello class!
```

Figure 4: Output from executable `rose_example0_OpenMP`

Table 1: Timing results from running `rose_example0_OpenMP`

Threads used	Time in seconds
Serial Version (single thread)	0.003
Parallel Version with 1 thread	0.118
Parallel Version with 2 threads	0.068
Parallel Version with 4 threads	0.070
Parallel Version with 8 threads	0.077

Note that, there is only one entry and exit point to the parallel region in the OpenMP programs. Users are not allowed to prematurely branch out of the parallel region. IPT warns the user of such rules during the parallelization process. IPT also warns the user that there should be no return or break statements in the region of parallelization.

Recap

The three types of OpenMP worksharing constructs for C/C++ are:

1) **`for`**
2) **sections**
3) **single**

The two types of directives that we have covered so far are:

`1) # pragma omp parallel`
`2) # pragma omp for`

The directives can include clauses. The two types of data-sharing attribute clauses that we have covered so far are:

`1) default`
`2) private`

3.2 Parallelizing the Code for Heat Diffusion Equation

Our second example is a program that solves a 1-D time-dependent heat equation using the finite difference method. Snippets of this code, `heat_serial.c,` are shown in Figure 5. A for-loop in Figure 5 calls the function `calc_up` for `Nx - 1` times. Another loop in this code is used to set the value of $u_{x,y}$ to the value of $up_{x,y}$. We will see the process of parallelizing both these loops (shown in boldface) with IPT.

```
gettimeofday(&start, NULL);
for(t=0;t<Nt;t++) {
  for(x=1; x<Nx-1; x++){                            First loop for parallelization
      for(y=1; y<Ny-1; y++) {
        calc_up(x,y,Nx,Ny,u,up);}}

  for(x=1; x<Nx-1; x++){                            Second loop for parallelization
      for(y=1; y<Ny-1; y++){
        u[x][y] = up[x][y];}}

}
gettimeofday(&end, NULL);
```

Figure 5. Regions of code to parallelize in `heat_serial.c`

Data dependency is a situation that shows the relationship between the different variables in a code. It occurs when a program statement refers to the data of a preceding statement. **Loop-carried dependency** is a type of data dependency that occurs when the data calculated in one iteration of a loop is required in another iteration. Loop-independent dependency occurs when the data dependency occurs within the same iteration of the loop – that is, a value calculated in an iteration is used in another calculation in the same iteration. Loop-carried dependencies prevent the parallelization of a loop.

The code snippets shown in Figure 5 are good candidates for parallelization because they do not have any loop-carried dependencies - that is, the results from one iteration do not impact the results from other iterations. To parallelize this code, we can use IPT in the same way as demonstrated in section 2.1. Recall that IPT will prompt the user to select the for-loop to parallelize and will pose a few other questions. IPT automatically adds the required syntax to parallelize the code. The steps for parallelizing this example with IPT are shown in Figure 6.

```
NOTE: We currently support only C and C++ programs.

Please select a parallel programming model from the following available options:
1. MPI
2. OpenMP
3. CUDA                 OpenMP selected
2

NOTE: As per the OpenMP standard, the parallel regions can have only one entry point
and only one exit point.
Branching out or breaking prematurely from a parallel region is not allowed.
Please make sure that there are no return/break statements in the region selected
for parallelization. However, exit/continue statements are allowed.

A list containing the functions in the input file will be presented, and you may
want to select one function at a time to parallelize it using multi-threading.
Please choose the function that you want to parallelize from the list below
1 : main
1

Please select one of the following options (enter 1 or 2 or 3)
1. Create a parallel region (a group of threads will be created and each thread will
execute a block of code redundantly but in parallel)
2. Parallelize a for-loop (a group of threads will be created and each thread will
execute a certain number of iterations of a for-loop)
3. Create a parallel section (TBD - this mode is currently unavailable)
2

Loop
Note: With your response, you will be selecting or declining the parallelization of
```

Choosing a for-loop

```
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested
in parallelizing, then, you will be able to select those at a later stage.

// Boundary conditions
for (x = 0; x < Nx; x++) {
  for (y = 0; y < Ny; y++) {
    if (x == 0)
      u[x][y] = 1.0;
     else
      u[x][y] = 0.0;
  }
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.

for (y = 0; y < Ny; y++) {
  if (x == 0)
    u[x][y] = 1.0;
   else
    u[x][y] = 0.0;
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.
Note: With your response, you will be selecting or declining the parallelization of
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested
in parallelizing, then, you will be able to select those at a later stage.

////////////////////////////////////////////////////////////////////////
// Finite difference algorithm - iterate over time to reach steady state
////////////////////////////////////////////////////////////////////////
for (t = 0; t < Nt; t++) {
  for (x = 1; x < Nx - 1; x++) {
    for (y = 1; y < Ny - 1; y++) {
      calc_up(x,y,Nx,Ny,u,up);
    }
  }
  for (x = 1; x < Nx - 1; x++) {
    for (y = 1; y < Ny - 1; y++) {
      u[x][y] = up[x][y];
    }
  }
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.
Note: With your response, you will be selecting or declining the parallelization of
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested
in parallelizing, then, you will be able to select those at a later stage.

for (x = 1; x < Nx - 1; x++) {
  for (y = 1; y < Ny - 1; y++) {
    calc_up(x,y,Nx,Ny,u,up);
  }
}
Is this the for-loop you are looking for?(y/n)
```

```
y

Reduction variables are the variables that should be updated by the OpenMP threads
and then accumulated according to a mathematical operation
like sum, multiplication,etc.
Do you want to perform reduction on any variable ?(Y/N)
n
```

IPT located the loop from Figure 5

```
Do you want to use any scheduling scheme for this loop (y/n)?
n

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)
n

Would you like to parallelize another loop?(Y/N)
y
```

Parallelizing two loops

```
Please choose the function that you want to parallelize from the list below
1 : main
1

Note: With your response, you will be selecting or declining the parallelization of
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested
in parallelizing, then, you will be able to select those at a later stage.

// Boundary conditions
for (x = 0; x < Nx; x++) {
  for (y = 0; y < Ny; y++) {
    if (x == 0)
      u[x][y] = 1.0;
     else
      u[x][y] = 0.0;
  }
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.

for (y = 0; y < Ny; y++) {
  if (x == 0)
    u[x][y] = 1.0;
   else
    u[x][y] = 0.0;
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.
Note: With your response, you will be selecting or declining the parallelization of
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested
in parallelizing, then, you will be able to select those at a later stage.

////////////////////////////////////////////////////////////////////////
// Finite difference algorithm - iterate over time to reach steady state
////////////////////////////////////////////////////////////////////////
for (t = 0; t < Nt; t++) {

#pragma omp parallel default(none) shared(u,up,Nx,Ny) private(x,y)
{
```

```
#pragma omp for
  for (x = 1; x < Nx - 1; x++) {
    for (y = 1; y < Ny - 1; y++) {
      calc_up(x,y,Nx,Ny,u,up);
    }
  }
}
  for (x = 1; x < Nx - 1; x++) {
    for (y = 1; y < Ny - 1; y++) {
      u[x][y] = up[x][y];
    }
  }
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.

for (y = 1; y < Ny - 1; y++) {
  calc_up(x,y,Nx,Ny,u,up);
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.
Note: With your response, you will be selecting or declining the parallelization of
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested
in parallelizing, then, you will be able to select those at a later stage.

for (x = 1; x < Nx - 1; x++) {
  for (y = 1; y < Ny - 1; y++) {
    u[x][y] = up[x][y];
  }
}
Is this the for-loop you are looking for?(y/n)
y

Reduction variables are the variables that should be updated by the OpenMP threads
and  then  accumulated  according  to  a  mathematical  operation  like  sum,
multiplication,etc.
Do you want to perform reduction on any variable ?(Y/N)
n

Do you want to do element-wise reduction on any array?(Y/N)
n

IPT is unable to perform the dependency analysis of the array named [ u ] 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.
2

Do you want to use any scheduling scheme for this loop (y/n)?
n

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)
n

Would you like to parallelize another loop?(Y/N)
```

IPT has already parallelized the first loop

Second loop to be parallelized

Only one array element - u[x][y] - is updated in an iteration

```
n

Are you writing/printing anything from the parallelized region of the code?(Y/N)
n

Do you want to perform any timing (Y/N) ?
n
```

Figure 6. Parallelizing `heat_serial.c` using IPT

The snippets of the parallel version of the serial loops shown in Figure 5 are presented in Figure 7. IPT inserted the directive for a parallel region with the statement **`#pragma omp parallel.`** The directive **`#pragma omp for`** was added before the for-loop. Notice that other than the insertion of the pragmas and the statement for including the `omp.h` header file (not shown in this code snippet), there are no changes to the original code of the for-loop.

While this program was more complex than the one shown in Section 2.1 (`example0.c`), and had two for-loops that were parallelized, the parallelization process in both examples is almost the same when using IPT - the user identifies which for-loops to parallelize and responds to the high-level questions while IPT generates the parallel version.

```
For-loop 1:

#pragma omp parallel default(none) shared(u,up,Nx,Ny) private(x,y)
{
#pragma omp for
    for (x = 1; x < Nx - 1; x++) {
      for (y = 1; y < Ny - 1; y++) {
        calc_up(x,y,Nx,Ny,u,up);
      }
    }
}
```

Insertion of two pragmas - one declares a parallel region and the other for-loop worksharing

```
For-Loop 2:

#pragma omp parallel default(none) shared(u,up,Nx,Ny) private(x,y)
{
#pragma omp for
    for (x = 1; x < Nx - 1; x++) {
      for (y = 1; y < Ny - 1; y++) {
        u[x][y] = up[x][y];
      }
    }
}
```

Figure 7. Code snippet of the parallel version of heat_serial.c generated by IPT

In the output code shown in Figure 7, we notice a new clause: `shared`. When variables (or arrays) in a parallelized for-loop can be safely shared among multiple threads without the risk of their values being overwritten, then those variables (or arrays) can be specified as `shared`. The shared variables and arrays for the second parallelized loop include **`u, up, Nx,`** and **`Ny`**. Here, **`Nx,`** and **`Ny`** are the variables whose values are not updated in the for-loop. Hence, they can be safely shared amongst different threads. The elements of the array **`u`** are updated one at a time in each iteration of the second loop. That is, each thread works on updating a separate element of this array in each iteration of the loop. Hence, this array can be safely declared as shared amongst threads. The array **`up`** is not being updated in this loop – its values are only being read. Hence, it is safe to share a single copy of this array amongst all the threads.

Note that the OpenMP **`parallel`** and **`for`** directives can be combined into one directive as follows and the clauses may need to be adjusted while merging them: **`#pragma omp parallel for`**

3.3 Parallelizing a For-Loop – Introduction to Reduction

Before looking into the process of parallelizing this example with IPT, one must understand the concept of reduction in OpenMP. **Reduction** (syntax: `reduction`) is yet another data-sharing clause that is used to specify that one or more variables that are private to each thread will be subject to a `reduction` operation, such as +, -, *, max, and min, at the end of the parallel region. OpenMP supports additional reduction operations as well, such as: logical AND, logical OR, bitwise AND, bitwise OR, bitwise XOR, equivalent, not equivalent.

The variables that are specified in the reduction clause are known as reduction variables. Each thread in the team gets private copies of the reduction variables created for it. These private variables are initialized to zero. At the end of the parallel region, the reduction operations are applied to all the private copies of the reduction variables, and the results of the operations are written to the shared copies of the variables.

At a high-level it can be said that reduction is a computation that combines multiple elements (or partial results) from the threads as per the specified reduction operation to create one global result.

Note that a variable mentioned in the `reduction` clause should not be explicitly listed in the private clause (as shown below) – doing so will result in compile-time error:

```
// ERROR: private variable sum cannot be specified in a reduction clause
#pragma omp parallel default (none) private(i, sum) shared (n, b)
{
   #pragma omp for reduction(+: sum)
   for (i=0; i<n; i++){
      sum += b[i];
   }
}
```

On the contrary, the variable that is mentioned in the reduction clause should be mentioned in the `shared` clause in one particular scenario. This scenario is when a loop is decorated by the `parallel` and `for` directives on separate lines, and the reduction clause is specified with the `for` directive. In this situation, the `shared` clause of the `parallel` directive should contain the reduction variable that is specified with the `for` directive. As explained above, internally within OpenMP, the value of the reduction variable (which is implicitly private in the for-loop) is calculated by different threads and is combined as per the selected reduction operation to produce a final result. This final result is then written to the shared copy of the variable and is visible to all the threads in the group.

```
//Correct syntax
#pragma omp parallel default (none) private(i) shared (sum, n, b)
{
   #pragma omp for reduction(+: sum)
   for (i=0; i<n; i++){
      sum += b[i];
   }
}
```

When the `parallel` and `for` directives are merged into one, then the `reduction` variable should not feature in any other clause. For example, adding the `sum` variable to the `shared` clause will give an error:

```
//ERROR: In a parallel directive, the variable sum cannot be specified in
//both a shared and a reduction clause

#pragma omp parallel for shared(sum) reduction(+: sum)
```

Figure 8 shows the code of the program `example1.c`. This program has two main loop blocks, and two arrays: `x` and `y`. The first loop block initializes `x`, and the second loop section updates the value of the array `y` and variable `sum` using the temporary variable `tmp`. The second loop is computationally expensive and does not show any data dependency. Hence, this is the loop that we will parallelize. While parallelizing this loop, `sum` should be specified as a reduction variable and the reduction operation should be specified as +. This is because the value of `sum` is incremented in each iteration by adding the value of `tmp` that is calculated in that iteration (that is, `sum = sum + tmp;`). As loop iterations execute, the value of `sum` is accumulating via addition. When this loop is parallelized, to produce correct results, it is important that all the threads participating in the computation make a copy of the `sum` variable and continue updating their copy of this variable for as many iterations as they are assigned. By doing so, the threads would have calculated their local sums. Now, the local sum from all the threads should be combined to produce a global sum. The global sum is the value that the loop would produce if it were executing serially.

```
#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-loop to parallelize
      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;
}
```

The value of `sum` is incremented in each iteration

Figure 8. Snippet of example1.c

Figure 9 shows the process of parallelizing this example with IPT. A snippet from the parallel version of the code generated using IPT is shown in Figure 10.

Invoking IPT

```
$ IPT ./example1.c

NOTE: We currently support only C and C++ programs.
Please select a parallel programming model from the following available options:
1. MPI
2. OpenMP
3. CUDA
2

NOTE: As per the OpenMP standard, the parallel regions can have only one entry point
and only one exit point.
Branching out or breaking prematurely from a parallel region is not allowed.
Please make sure that there are no return/break statements in the region selected for
parallelization. However, exit/continue statements are allowed.

A list containing the functions in the input file will be presented, and you may want
to select one function at a time to parallelize it using multi-threading.
Please choose the function that you want to parallelize from the list below
1 : main
1

Please select one of the following options (enter 1 or 2 or 3)
1. Create a parallel region (a group of threads will be created and each thread will
execute a block of code redundantly but in parallel)
2. Parallelize a for-loop (a group of threads will be created and each thread will
execute a certain number of iterations of a for-loop)
3. Create a parallel section (TBD - this mode is currently unavailable)
2

Loop
Note: With your response, you will be selecting or declining the parallelization of
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested in
parallelizing, then, you will be able to select those at a later stage.

for (i = 0; i <= 30000 + 1; i++) {
  for (j = 0; j <= 30000 + 1; j++) {
    x[i][j] = ((double )((i + j) % 3)) - 0.9999;
  }
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.
```

```
for (j = 0; j <= 30000 + 1; j++) {
  x[i][j] = ((double )((i + j) % 3)) - 0.9999;
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.
Note: With your response, you will be selecting or declining the parallelization of
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested in
parallelizing, then, you will be able to select those at a later stage.

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;
  }
}
Is this the for-loop you are looking for?(y/n)
y
```

Identifying which loop to parallelize

```
Reduction variables are the variables that should be updated by the OpenMP threads and
then accumulated according to a mathematical operation like sum, multiplication,etc.
Do you want to perform reduction on any variable ?(Y/N)
y

Please select a variable to perform the reduce operation on (format 1,2,3,4 etc.).
List of possible variables are:
1. tmp type is double
2. j type is int
3. sum type is double
3
```

Explicitly identifying reduction variable

```
Please enter the type of reduction you wish for variable [sum]
1. Addition
2. Subtraction
3. Min
4. Max
5. Multiplication
1
```

Explicitly identifying reduction operation

```
Do you want to do element-wise reduction on any array?(Y/N)
n

IPT is unable to perform the dependency analysis of the array named [ y ] 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.
2
```

Only y[i][j] is updated

```
Do you want to use any scheduling scheme for this loop (y/n)?
n
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)
n

Would you like to parallelize another loop?(Y/N)
n

Are you writing/printing anything from the parallelized region of the code?(Y/N)
n
```

```
Do you want to perform any timing (Y/N) ?
n
```

Figure 9. Running IPT with exercise1.c

Two pragma statements have been inserted in the parallelized version of the code shown in Figure 10. While we have already discussed the main purpose of the `parallel` and `for` directives, we will explain the clauses that are added to these directives in this example.

Note that IPT automatically determines most of the data-sharing attributes of the relevant variables or multi-variables in the code that it parallelizes. However, it **relies on the user input for adding the reduction clauses.** It presents a list of possible reduction variables, and prompts the user to select the appropriate ones and specify the type of reduction operation for the selected reduction variables.

```
#pragma omp parallel default(none) shared(sum,x,y) private(j,i,tmp)
{
                                              Reduction clause added to this pragma
#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;
    }
  }
}
```

Figure 10. Snippet of code from parallelized version of `example1.c` using OpenMP

The `pragma omp parallel` directive has three clauses. These clauses are `default(none)`, `shared(sum,x,y)`, `private(j,i,tmp)`. Let us discuss the `default` clause first. As OpenMP is a shared memory model, by default, the variables or arrays in the parallel region that have not been explicitly added to any clause will be treated as `shared`. The `default` clause in this example is applied to `none` of the variables or arrays – that is, we are telling OpenMP to not assume any variable or array to be `shared` by `default`. The OpenMP code generated using IPT always has the `default(none)`clause. IPT always explicitly defines the scope of variables in order to reduce the risk of introducing errors.

The scalar variable `tmp` and the for-loop variables, `i` and `j`, are added to the `private` clause. Each thread has its own copy of these `private` variables. The loop variables must be private because each thread running its set of loop iterations has different values of the loop-variables, and updates these values at the end of each iteration. The variable `tmp` is also private because each thread should independently update its value in each iteration. If these variables are not made `private`, the program is likely to produce unpredictable results because threads are likely to compete with each other for simultaneously updating the values of these variables at the same time. As a rule-of-thumb, all the variables that are being updated or written to in a parallel region are good candidates for getting added to the `private` clause unless, they also occur in the `reduction` clause.

Even though the arrays `y` is being updated in each iteration, it can be safely shared across the team of threads since each thread writes to a different element of this arrays, and hence, there is no risk of one thread writing over the contents of another thread. The array `x` is only being read from in the parallelized for-loop, and hence, it is safe to add it to the `shared` clause.

The **`for`** directive has the clause '**`reduction`**'. The statement **`(+ :sum)`** defines the reduction operation as addition and the reduction variable as **`sum`**. As explained above, when the `for` and `parallel` directives are on separate lines and the `reduction` clause is a part of the `for` directive, it is required to add the `reduction` variable to the `shared` clause of the `parallel` directive. Therefore, `sum` was added to the `shared` clause of the `parallel` directive. Each thread will have a `private` copy of the variable `sum`. After the loop execution, the parent thread that forked the team of threads (also known as the **master thread**) collects the `private` values of `sum` from each thread and applies the addition operation on the collected values to produce the final result (the sum total of all the private values of sum at the end of the for-loop).

Now let us compile and run the parallel version generated by IPT. The command to compile and run the parallel version is shown in Figure 11. The output of this program is also shown in Figure 11.

```
$ icc -qopenmp -o rose_example1_OpenMP rose_example1_OpenMP.c
$ export OMP_NUM_THREADS=4
$ time ./rose_example1_OpenMP
Memory allocation done successfully

The total sum is: 90000.000000
run time    = 12.841216s

real    0m22.541s
user    0m53.416s
sys     0m7.473s
```

Figure 11. Commands and output for running and compiling parallel version of `example1.c`

While the code for `example1.c` has its own timing statements, the program was run with the `time` command to print the overall time that the program took to run. The timing results from running the program with various numbers of threads is shown in Table 2.

Table 2. Timing results for `example1.c`

Number of threads	**Time (seconds)**
Serial Version	222.58
`export OMP_NUM_THREADS=2`	33.74
`export OMP_NUM_THREADS=4`	22.54
`export OMP_NUM_THREADS=8`	16.89
`export OMP_NUM_THREADS=16`	14.11

Table 2 shows the performance improvement due to parallelization. When comparing the serial run-time of the program to the run-time with 2 threads, the speed increased by over six times. Although Table 2 shows that increasing the number of threads from 2 to 16 did increase the speed, the magnitude of the increase diminished after 8 threads. Beyond 16 threads, there is no significant decrease in the overall runtime and hence, it is not efficient to run this program with more than 16 threads.

3.4 Loops with dependencies

In the previous examples, we are able to parallelize the for-loops without requiring any code modifications. However, the code of `example2.c` that is shown in Figure 12 is not amenable for parallelization due to a dependency. For updating $y_{i,j}$ in each loop-iteration shown in boldface in Figure 12, the value of $y_{i+1,j}$ is needed. Hence, the calculations in a loop iteration (`i`) are dependent on the calculations in another iteration (`i+1`). Fortunately, there is a simple solution for this example that will allow us to parallelize this code easily. If this loop is broken into two separate for-loops as shown in Figure 13, then the loops can be parallelized. After removing the dependency in the for-loop, both the sets of for-loops shown in Figure 13 can be parallelized in the same way as shown in the previous examples.

```
#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
  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];
    }
  }
  gettimeofday(&start, NULL);

  //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;
}
```

This loop has a dependency. In each iteration, an element of array `y` in updated using the value of its another element.

Figure 12. Code from `example2.c`, loop in bold-face shows data dependency

4 Additional Examples

In this section, we will look into the process of parallelizing two more applications using IPT. The source code for these applications can be found at the following path on Stampede2:

```
/scratch/01698/rauta/PEARC18_OMP/trainingIPT/exercises/exercises_3_to_7
```

4.1 Molecular Dynamics (MD)

The MD simulation application is used in multiple domains (e.g., biochemistry, biophysics, and material science) for studying the movement of atoms and molecules. Typically, the simulation is run for a fixed period of time during which the atoms and molecules are allowed to interact with each other [1]. During the interaction, the atoms and molecules exert force on each other, in a constrained or an unconstrained manner, thus giving a dynamic view of their interaction over a period of time.

```
//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]);
    }
  }

  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];
    }
  }
```

First loop computes value of `tmp[i][j]`

Second loop updates `y` and `sum` using the values of `tmp` calculated in the previous set of loops.

Figure 13. Restructuring the loops to remove dependencies

The MD simulation application that we are considering here is designed to follow the path of particles that exert force on each other and are not constrained by any walls [2]. Therefore, after colliding, the particles move past each other. This MD code uses the velocity Verlet time integration scheme and the particles in the simulation interact with a central pair potential [2].

The compute-intensive steps in this application are related to calculating the force and energies (potential energy and kinetic energy) in each time-step, as well as updating the values of the positions, velocities, and accelerations of the particles (*i.e.*, the atoms and molecules) in the simulation. The size of the system (*i.e.*, the number of particles in the simulation), the length of time between the evaluation of energies and force (*i.e.*, the size of the integration step), and the time duration for which the interaction of particles needs to be studied impact the overall runtime of the simulation. In real-world scenarios, often there are several thousand particles in an MD simulation and studying their movement using a serial implementation of the code can take a large amount of time. Therefore, parallel computing is used to reduce the overall run-time of such simulations.

Let us consider the serial version of the code for the MD simulation application. The complete code for this application is available at [2]. After profiling the code with `gprof`, we can see that the function named `compute` is the most time-consuming function and is a good candidate for parallelization. The program spends about 98.9% of its time in this function. A code snippet of the `compute` function is shown in Figure 14.

The number of iterations of the for-loop beginning at line # 1 of Figure 14 is equal to the number of particles (`np`) in the simulation. The potential energy (`pe`), force (`f`), and kinetic energy (`ke`) of each particle are calculated in each iteration, and their values across all the iterations are added together. The number of spatial dimensions for the simulation is represented by `nd` and the distance between the particles is represented by `d` in the code snippet shown in Figure 14. The values of the elements in the displacement vector `rij` are calculated for each particle in the function named `dist` in each iteration of the for-loop beginning at line # 6. The variable `PI2` shown in the code snippet in Figure 14 represents the value of the constant `pi` divided by 2, that is, (3.14/2). Except the loop-variables that are of type integer, all other values in this example are of type double.

There are no dependencies between the iterations of the for-loop at line # 1 of the code in Figure 14. Hence, this for-loop can be parallelized by distributing the computations in the loop across multiple threads or processes. Once all the threads or processes have finished their share of computations and have their local results ready, we can combine those results in a meaningful manner to obtain a global result. This global result should match the results of running the application in the serial mode as closely as possible (some rounding off errors may be permissible). Combining the locally computed values of variables while applying a mathematical operation (*e.g.,* sum or multiplication) is referred to as reduction. For combining the values of small local arrays into a large global array when the local arrays are computed using processes having separate address space - as is the case in MPI - a gather operation is used. Both reduction and gather operations are classified under the pattern named "data collection" in IPT.

In the MD example, if the for-loop at line # 1 of Figure 14 is parallelized, the values of the variables `pe` and `ke` will be computed by each thread or process participating in the computation for a certain number of iterations. However, these values should be collected together and added using `reduction` to obtain the same result as one would get without parallelization.

Each thread when working in parallel updates certain number of elements of the array `f`. If the threads are used for parallel computation, this array can be in a shared memory region. Each thread can work on updating the values of certain number of elements in the same array. Hence, no extra step to combine the updated elements of the array is needed.

```
1.  for ( k = 0; k < np; k++ ){
2.      //Compute the potential energy (pe) and forces (f).
3.      for ( i = 0; i < nd; i++ ){
4.        f[i+k*nd] = 0.0;
5.      }
6.      for ( j = 0; j < np; j++ ){
7.        if ( k != j ){
8.           d = dist ( nd, pos+k*nd, pos+j*nd, rij );
9.           if ( d < PI2 ) {
10.            d2 = d;
11.          } else{
12.            d2 = PI2;}
13.          pe = pe + 0.5 * pow ( sin ( d2 ), 2 );
14.          for ( i = 0; i < nd; i++ ){
15.            f[i+k*nd] = f[i+k*nd] - rij[i] * sin ( 2.0 * d2 ) / d; }
16.       }
17.      }
18.      //Compute the kinetic energy (ke).
19.      for ( i = 0; i < nd; i++ ) {
20.          ke = ke + vel[i+k*nd] * vel[i+k*nd]; }
21.  }
```

Figure 14. Snippet of the `compute` function – serial version of the MD code

The steps for parallelizing the MD code with IPT are shown in Figure 15. After the user follows the steps shown in Figure 15, IPT generates an output file named `rose_md_OpenMP.c`. The generated code is shown in Figure 16. The code inserted by IPT is highlighted in boldface. As can be noticed from the OpenMP version of the code, IPT inserted the directives for starting a team of parallel threads for executing the for-loop and performing the reduction operation.

```
NOTE: We currently support only C and C++ programs.

Please select a parallel programming model from the following available options:
1. MPI
2. OpenMP
3. CUDA
2

NOTE: As per the OpenMP standard, the parallel regions can have only one entry point
and only one exit point.
Branching out or breaking prematurely from a parallel region is not allowed.
Please make sure that there are no return/break statements in the region selected
for parallelization. However, exit/continue statements are allowed.

A list containing the functions in the input file will be presented, and you may
want to select one function at a time to parallelize it using multi-threading.
Please choose the function that you want to parallelize from the list below
1 : main
2 : compute
3 : cpu_time
4 : dist
5 : initialize
6 : r8mat_uniform_ab
7 : timestamp
8 : update
2

Please select one of the following options (enter 1 or 2 or 3)
1. Create a parallel region (a group of threads will be created and each thread will
execute a block of code redundantly but in parallel)
2. Parallelize a for-loop (a group of threads will be created and each thread will
execute a certain number of iterations of a for-loop)
3. Create a parallel section (TBD – this mode is currently unavailable)
2

Loop
Note: With your response, you will be selecting or declining the parallelization of
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested
in parallelizing, then, you will be able to select those at a later stage.

for (k = 0; k < np; k++) {
/*
  Compute the potential energy and forces.
*/
  for (i = 0; i < nd; i++) {
    f[i + k * nd] = 0.0;
  }
  for (j = 0; j < np; j++) {
    if (k != j) {
      d = dist(nd,pos + k * nd,pos + j * nd,rij);
/*
  Attribute half of the potential energy to particle J.
*/
      if (d < PI2) {
        d2 = d;
      }
       else {
        d2 = PI2;
      }
      pe = pe + 0.5 * pow((sin(d2)),2);
      for (i = 0; i < nd; i++) {
        f[i + k * nd] = f[i + k * nd] - rij[i] * sin(2.0 * d2) / d;
      }
    }
  }
/*
  Compute the kinetic energy.
*/
  for (i = 0; i < nd; i++) {
    ke = ke + vel[i + k * nd] * vel[i + k * nd];
  }
}
Is this the for-loop you are looking for?(y/n)
y
```

Selected loop from Figure 11

```
Reduction variables are the variables that should be updated by the OpenMP threads
and then accumulated according to a mathematical operation like sum,
multiplication,etc.
Do you want to perform reduction on any variable ?(Y/N)
y

Please select a variable to perform the reduce operation on (format 1,2,3,4
etc.).  List of possible variables are:
1. nd type is int
2. k type is int
3. np type is int
4. d type is double
5. PI2 type is double
6. d2 type is double
7. pe type is double
8. ke type is double
7,8
```

Identifying pe and ke as reduction variables

```
Please enter the type of reduction you wish for variable [pe]
1. Addition
2. Subtraction
3. Min
4. Max
5. Multiplication
1

Please enter the type of reduction you wish for variable [ke]
1. Addition
2. Subtraction
3. Min
4. Max
5. Multiplication
1
Do you want to do element-wise reduction on any array?(Y/N)
n

IPT is unable to perform the dependency analysis of the array named [ f ] 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.
2
```

Single value updated per iteration

```
IPT is unable to perform the dependency analysis of the array named [ rij ] 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.
2

Do you want to use any scheduling scheme for this loop
(y/n) ?
n

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)
n

Would you like to parallelize another loop?(Y/N)
n

Are you writing/printing anything from the parallelized region of the code?(Y/N)
n

Do you want to perform any timing (Y/N) ?
n
```

Entire array not updated

Figure 15. Running IPT for parallelizing MD code

```
#pragma omp parallel default(none) shared(pe,ke,np,f,pos,vel,nd,PI2)
private(k,i,j,d,d2) firstprivate(rij)
{

#pragma omp for reduction ( + :pe,ke)
  for (k = 0; k < np; k++) {
  //Compute the potential energy and forces.
    for (i = 0; i < nd; i++) {
      f[i + (k * nd)] = 0.0;
    }
    for (j = 0; j < np; j++) {
      if (k != j) {
        d = dist(nd,(pos + (k * nd)),(pos + (j * nd)),rij);
        //Attribute half of the potential energy to particle J.
        if (d < PI2) {
          d2 = d;
        }
        else {
          d2 = PI2;
        }
        pe = (pe + (0.5 * pow(sin(d2),2)));
        for (i = 0; i < nd; i++) {
          f[i + (k * nd)] = (f[i + (k * nd)] - ((rij[i] * sin((2.0 * d2))) /
d));
        }
      }
    }
    //Compute the kinetic energy.
    for (i = 0; i < nd; i++) {
      ke = (ke + (vel[i + (k * nd)] * vel[i + (k * nd)]));
    }
  }
}
```

OpenMP reduction clause with sum operation

Figure 16, Snippet of the parallel version of MD code generated by IPT

4.2 Circuit Satisfiability

Circuit Satisfiability is a problem that determines whether a boolean circuit produces an output of 1 with a set of input values. The complete code for this application is available at [3, 4]. A snippet of this code is shown in Figure 17. The for-loop shown in this code snippet is a good candidate for parallelization as there are no data dependencies in this loop. Recall that in order to run IPT we need to know which loop we want to parallelize as well as identify any reduction variables. In this example, `solution_num` should be reduced to obtain the correct results. Since we are adding the values of the individual processes into `solution_num`, our operation will be sum.

```
for (i = 0; i < ihi; i++) {
  i4_to_bvec(i,n,bvec);
  value = circuit_value(n,bvec);
  if (value == 1) {
    solution_num = solution_num + 1;
    printf("  %2d  %10d:  ",solution_num,i);
    for (j = 0; j < n; j++) {
      printf(" %d",bvec[j]);
    }
    printf("\n");
  }
}
```

Reduction of `solution_num`

Figure 17. Snippet of the main function in `circuit.c`

```
NOTE: We currently support only C and C++ programs.

Please select a parallel programming model from the following available options:
1. MPI
2. OpenMP
3. CUDA
2

NOTE: As per the OpenMP standard, the parallel regions can have only one entry point
and only one exit point.
Branching out or breaking prematurely from a parallel region is not allowed.
Please make sure that there are no return/break statements in the region selected
for parallelization. However, exit/continue statements are allowed.

A list containing the functions in the input file will be presented, and you may
want to select one function at a time to parallelize it using multi-threading.
Please choose the function that you want to parallelize from the list below
1 : main
2 : circuit_value
3 : i4_to_bvec
4 : timestamp
1

Please select one of the following options (enter 1 or 2 or 3)
1. Create a parallel region (a group of threads will be created and each thread will
execute a block of code redundantly but in parallel)
2. Parallelize a for-loop (a group of threads will be created and each thread will
execute a certain number of iterations of a for-loop)
3. Create a parallel section (TBD – this mode is currently unavailable)
2

Loop

for (i = 1; i <= n; i++) {
  ihi = ihi * 2;
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.
Note: With your response, you will be selecting or declining the parallelization of
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested
in parallelizing, then, you will be able to select those at a later stage.

for (i = 0; i < ihi; i++) {
  i4_to_bvec(i,n,bvec);
  value = circuit_value(n,bvec);
  if (value == 1) {
    solution_num = solution_num + 1;
    printf("  %2d  %10d:  ",solution_num,i);
    for (j = 0; j < n; j++) {
      printf(" %d",bvec[j]);
    }
    printf("\n");
  }
}
```

Loop to parallelize from Figure 14

```
Is this the for-loop you are looking for?(y/n)
y

Reduction variables are the variables that should be updated by the OpenMP threads
and then accumulated according to a mathematical operation like sum,
multiplication,etc.
Do you want to perform reduction on any variable ?(Y/N)
y

Please select a variable to perform the reduce operation on (format 1,2,3,4
etc.).  List of possible variables are:
1. i type is int
2. n type is int
3. value type is int
4. solution_num type is int
4
```

Selecting `solution_num` as a reduction variable

```
Please enter the type of reduction you wish for variable [solution_num]
1. Addition
2. Subtraction
3. Min
4. Max
5. Multiplication
1

Do you want to do element-wise reduction on any array?(Y/N)
n

IPT is unable to perform the dependency analysis of the array named [ bvec ] 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.
1

Do you want to use any scheduling scheme for this loop (y/n) ?
n

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)
n
```

```
Would you like to parallelize another loop?(Y/N)
n

Are you writing/printing anything from the parallelized region of the code?(Y/N)
n

Do you want to perform any timing (Y/N) ?
n
```

Figure 18, Parallelizing `circuit.c` with IPT

A snippet of the code generated using IPT is shown in Figure 19. From this snippet, we can see that IPT inserted two directives. The first is **`#pragma omp parallel`** which creates a team of threads to run in parallel. The second is **`#pragma omp for`** which tells the compiler that the iterations of the for-loop decorated with this directive will be divided amongst the threads running in parallel. The **`reduction`** clause indicates that `solution_num` is a reduction variable and the reduction operation is sum (+).

```
#pragma omp parallel default(none) shared(solution_num,ihi,n)
private(i,value,j) firstprivate(bvec)
{

#pragma omp for reduction ( + :solution_num)
  for (i = 0; i < ihi; i++) {
    i4_to_bvec(i,n,bvec);
    value = circuit_value(n,bvec);
    if (value == 1) {
      solution_num = solution_num + 1;
      printf("  %2d  %10d:  ",solution_num,i);
      for (j = 0; j < n; j++) {
        printf(" %d",bvec[j]);
      }
      printf("\n");
    }
  }
}
```

IPT generates pragmas for parallelization

Figure 19. Parallelized snippet from `circuit.c`

5 Nested For-loop Parallelization

Figure 20 shows an example of nested for-loops that perform matrix multiplication - the matrices `a` and b, are multiplied, and their product is saved in matrix `mult.` The process for nested for-loop parallelization with IPT is similar to that of a single for-loop parallelization. To parallelize this code, choose the outer for-loop shown in Figure 20. The variable dependency analysis will also be the same as with a single for-loop. The steps for parallelization with IPT are shown in Figure 21. Running IPT with nested for-loops is similar to all of our previous examples with for-loops. IPT will prompt the user by asking which for-loop to parallelize and gives the option to choose the inner or outer loops

```
for (int i =0; i<M; ++i){
   for(int j=0; j<N; ++j){
      for(int k=0; k<K; ++k){
           mult[i][j] += a[i][k]*b[k][j];
      }
```

```
    }
}
```

Figure 20: Code snippet of serial matrix multiplication

```
NOTE: We currently support only C and C++ programs.

Please select a parallel programming model from the following available options:
1. MPI
2. OpenMP
3. CUDA
2

NOTE: As per the OpenMP standard, the parallel regions can have only one entry point
and only one exit point.
Branching out or breaking prematurely from a parallel region is not allowed.
Please make sure that there are no return/break statements in the region selected
for parallelization. However, exit/continue statements are allowed.

A list containing the functions in the input file will be presented, and you may
want to select one function at a time to parallelize it using multi-threading.
Please choose the function that you want to parallelize from the list below
1 : main
1

Please select one of the following options (enter 1 or 2 or 3)
1. Create a parallel region (a group of threads will be created and each thread will
execute a block of code redundantly but in parallel)
2. Parallelize a for-loop (a group of threads will be created and each thread will
execute a certain number of iterations of a for-loop)
3. Create a parallel section (TBD - this mode is currently unavailable)
2

Loop
Note: With your response, you will be selecting or declining the parallelization of
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested
in parallelizing, then, you will be able to select those at a later stage.

for (int i = 0; i < M; ++i) {
  for (int k = 0; k < K; k++) {
    a[i][k] = i;
  }
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.

for (int k = 0; k < K; k++) {
  a[i][k] = i;
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.
Note: With your response, you will be selecting or declining the parallelization of
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested
in parallelizing, then, you will be able to select those at a later stage.

for (int k = 0; k < K; ++k) {
  for (int j = 0; j < N; j++) {
    b[k][j] = j;
  }
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.

for (int j = 0; j < N; j++) {
  b[k][j] = j;
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.
Note: With your response, you will be selecting or declining the parallelization of
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested
in parallelizing, then, you will be able to select those at a later stage.

for (int i = 0; i < M; ++i) {
  for (int j = 0; j < N; j++) {
    mult[i][j] = 0;
  }
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.

for (int j = 0; j < N; j++) {
  mult[i][j] = 0;
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.
Note: With your response, you will be selecting or declining the parallelization of
the outermost for-loop in the code region shown below. If instead of the outermost
for-loop, there are any inner for-loops in this code region that you are interested
in parallelizing, then, you will be able to select those at a later stage.

for (int i = 0; i < M; ++i) {
  for (int j = 0; j < N; ++j) {
    for (int k = 0; k < K; ++k) {
      mult[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

Target for parallelization, nested for-loops

```
Is this the for-loop you are looking for?(y/n)
y

Reduction variables are the variables that should be updated by the OpenMP threads
and then accumulated according to a mathematical operation like sum,
multiplication,etc.
Do you want to perform reduction on any variable ?(Y/N)
n

Do you want to do element-wise reduction on any array?(Y/N)
n

IPT is unable to perform the dependency analysis of the array named [ mult ] 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.
2
                                                     Single element updated
Do you want to use any scheduling scheme for this loop
(y/n) ?
n

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)
n

Would you like to parallelize another loop?(Y/N)
n

Are you writing/printing anything from the parallelized region of the code?(Y/N)
n

Do you want to perform any timing (Y/N) ?
n
```

Figure 21. Parallelizing matrix multiplication with IPT

The output of the nested for-loop parallelization can be seen in Figure 22. Two pragma statements have been added, just like in single for-loop parallelization. The first pragma, **`#pragma omp parallel`** creates the parallel region and performs variable analysis. The second **`#pragma omp for`**, specifies that the for-loop worksharing construct will be used.

```
#pragma omp parallel default(none) shared(mult,a,b)
{

#pragma omp for
  for (int i = 0; i < M; ++i) {
    for (int j = 0; j < N; ++j) {
      for (int k = 0; k < K; ++k) {
        mult[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

Figure 22. Snippet of the parallelized matrix multiplication code

6 Critical and Single Directives

In all of the previous examples, each thread ran the same code. Therefore, when IPT prompted the following, "`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)`" the response has been "`n`". However, in some cases it might be required to have the threads perform an operation one at a time or ensure that only one thread performs an operation. To handle these situations the directives `critical`, `atomic`, and `single` can be used. The following sections will explain these directives and how to apply them when using IPT.

6.1 Critical Regions

When a critical region is created, only one thread at a time can execute the code in that region. This region acts as a lock, meaning that other threads cannot enter the region until the thread currently in the region finishes. Critical directives can be used to help prevent race-conditions by ensuring that an action is serially performed by all the threads. Let us consider a simple example in Figure 23.

Serial Version

```
#include <stdio.h>
int main(){
    int x=0;
    x=x+1;
    return 0;
}
```

Parallel Version

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

int main(){
int x=0;
#pragma omp parallel{
    #pragma omp critical
     x=x+1;

}
}
```

Figure 23. Comparing the serial version with the parallel version having a critical statement

In the serial version, the statement `x = x+1` will be executed once and the final value of `x` will be 1. If the same code is run in parallel, it would be expected that the value of `x = N` where `N = #threads`. This implementation would be analogous to a for-loop that runs for `N` number of times. To ensure the final value of `x=N`, each thread should the statement `x=x+1` one at a time. As an example, if this code is run using three threads, the final result should be `x=3`.

The values of `x` should be updated as follows: thread A**:** `x=1`, thread B: `x=2`, and thread C: `x=3`. In order to achieve this result, one must ensure the value of `x` is updated by the threads one at a time. Otherwise, `x` could be read as `x=0` by threads A and B which both update `x=1`. Thread C could be reading the value of `x` as 1 and updating `x=2`. The statement **`#pragma omp critical`** creates a critical region so that if one thread is working in the critical region, no other threads can enter until the working thread finishes. By adding in this critical statement, the final result is `x=3`. This result could have also been achieved by using a `reduction` clause. In order to use the `critical` directive, the user should respond to the prompts as shown in Figure 24.

```
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)
Y

Please select from one of the following options:
1. Execute the entire code region using one thread
2. Execute the code region with multiple threads, but let only one thread
run at a time
2
```

Figure 24. Prompts and responses to use critical directive

IPT should then generate a file named `numberedCode.c`. If the user elects to show this file, it will be printed on the screen with a number added to each line for identification. The user can then select the lines in the code that should be added to the `critical` region. The file `numberedCode.c` can also be opened in a new shell window to look-up the line-numbers in case the user does not want to display its content in the shell in which IPT is running.

6.2 Atomic Directives

The `atomic` directive is similar to the `critical` directive but can be applied in fewer situations. This directive is used to restrict the access to a single memory location and protect data values. Additionally, the `critical` directive works by using locks – this is computationally intensive. However, the `atomic` directive provides some performance advantages over `critical` when used to protect statements that update variables. Consider the serial version shown in Figure 23. The code can also be parallelized using the `atomic` directive. The statement **`#pragma omp atomic update`** declares that `x=x+1` will be performed atomically. This means that no other threads can view can read from `x` until the action is complete. IPT automatically selects between `atomic` and `critical` once the user specifies to "**`Execute the code region with multiple threads, but let only one thread run at a time`**".

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

int main(){
int x=0;
#pragma omp parallel default(none), shared(x), private(none){

    #pragma omp atomic update
       x=x+1;

    }
}
```

Figure 25. Example of parallelized code using `#pragma omp atomic`

6.3 Single Directives

The `single` directive is used to serialize a section of code in the parallel region such that only one thread from the team will execute the code in this region. All other threads will skip this region. By default, there is an implied barrier at the end of the `single` region to prevent the thread that is executing in this region from getting behind the other threads. As a result, the other threads will execute independently until the end of the `single` region is reached. At this point these threads will wait for the thread executing in the `single` region to complete, and then all threads will proceed. If this wait is not desired, then the `no wait` clause can be added to the `single` directive statement. A parallelized version of the serial code shown in Figure 23 can also be created using the `single` directive, and is shown in Figure 28.

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

int main(){
int x=0;
#pragma omp parallel default(none), shared(x){

    #pragma omp single
     x=x+1;
    }
}
```

Figure 26. Example using `single` directive

The result from the code above would be "`x=1;`". This is because only one thread would update the value of `x`, even if the code was run by 4 threads. IPT will insert a `single` directive instead of `critical` if the user chooses option #1, **"`Execute an entire region of code using one thread`".** This is the only difference between creating and `single` and `critical` region when using IPT.

7 Practice Examples

This section allows the users to practice running IPT on their own. In some examples the users will be given the region of the program to parallelize and in others, the users will be asked to identify this region on their own. In all cases, the user will not be told how to respond to IPT when prompted. Each exercise has a key for the correct responses to the IPT prompts as well as a copy of the parallelized output.

7.1 Fast Fourier Transform

Fast Fourier is an algorithm developed to efficiently compute the discrete Fourier transform of a function. For this example, try parallelizing the hotspot shown in Figure 27. It is a good candidate for parallelization because this loop has no data dependencies so the iterations of the loop can be easily run in parallel. Complete code of the serial version is available at [5].

```
for (i = 0; i < (2 * n); i = (i + 2)) {
  z0 = ggl(&seed);
  z1 = ggl(&seed);
  x[i] = z0;
  z[i] = z0;
  x[i + 1] = z1;
  z[i + 1] = z1;
 }
}
```

Figure 27. Code snippet Fast Fourier Transform

After attempting to run IPT, compare the output to that in Figure 28. To see the full response to IPT prompts check key in the Appendix.

```
#pragma omp parallel default(none) shared(x,z,seed,n) private(i,z0,z1)
{

#pragma omp for
 for (i = 0; i < (2 * n); i = (i + 2)) {
       z0 = ggl(&seed);
       z1 = ggl(&seed);
       x[i] = z0;
       z[i] = z0;
       x[i + 1] = z1;
       z[i + 1] = z1;
   }
 }
```

Figure 28. Snippet of the Parallelized Code – FFT

Table 3: Summarized Key for FFT Example

Prompts	**Response**
Model, Function, Option	2 (OpenMP), 1(main), 2(for-loop)
Which for-loop?	Third loop
Reduction variables and arrays	No
Variable analysis for [x]	2 (otherwise)
All other prompts	No

7.2 Game of Life

Now let us parallelize the game of life program. The hotspot for parallelization is the for-loop in which the `compute` function is called. While the compute function itself does all the calculations, this for-loop runs for `NTIMES` number of iterations (as shown in Figure 29). As such, each thread will perform a certain number of calculations based on the user input and the number of threads available.

```
t1=gettime();
 for (k = 0; k < NTIMES; k++) {
 // compute new matrix
   temp = compute(life, temp, M, N);

 // swap old and new matrices
   ptr = temp;
   temp = life;
   life = ptr;
}
t2= gettime();
```

Figure 29. Snippet code calling `compute` function - Game of Life (serial code)

The results from parallelizing this example are shown in Figure 30. IPT added **`#pragma omp parallel`** to declare the parallel region. IPT also added **`#pragma omp for`** to indicate which type of worksharing construct will be used in this parallel region. See the appendix for the full walk through with IPT.

```
T1 = gettime();

#pragma    omp    parallel    default(none)    shared(NTIMES,M,N)    private(k)
firstprivate(temp,life,ptr)
{

#pragma omp for
 for (k = 0; k < NTIMES; k++) {
  // compute new matrix
  temp = compute(life,temp,M,N);
  // swap old and new matrices
  ptr = temp;
  temp = life;
  life = ptr;
 }
}
t2 = gettime();
```

Figure 30. Snippet of the Parallelized Code – Game of Life

8 Loop Scheduling

In the previous sections of the guide, it did not matter which thread ran which iteration of the parallelized loops. If the user wishes to provide more control over how loop iterations are assigned to threads, scheduling directives can be added. IPT allows the user to choose from the following scheduling options: `affinity`, `dynamic`, `guided`, `static`, and `runtime`. Each option will be explained in greater detail below. For all the options except for runtime, IPT will prompt the user for the chunk size. The syntax for each option is shown in Table 4 below.

8.1 Static

Static scheduling distributes `x` iterations to each loop where `x` is the chunk size specified by the user. Let us assume, a loop runs for 16 iterations, with 4 threads, and a chunk size of 4. With `static` scheduling, thread # 1 will be assigned iterations 1-4, thread # 2 will be assigned iterations 5-8, thread # 3 will be assigned iterations 9-12, thread # 4 will be assigned iterations 13-16. When thread # 1 finishes running iteration # 1, it will run the next one that it was assigned which in this case is iteration 2. If the thread number and chunk size stay constant but instead there were a total of 32 iterations, the assignment for iterations 1-16 would stay the same but thread # 1 would also be assigned iterations 17-20, and thread # 2 will get 21-24, thread # 3 will get 25-28, thread # 4 will get 29-32. This type of scheduling is most effective when the cost of each iteration is computationally similar.

8.2 Dynamic

Dynamic scheduling groups `x` iterations into a chunk where `x` is the chunk size specified by user. These chunks are then distributed among the various threads. This distribution is `dynamic`, because it occurs at run time. Threads that finish their chunks first will receive new chunks first. The default chunk size for this option is one. The chunks may be assigned to be distributed differently among the threads each time the program is run.

8.3 Guided

Guided distributes threads in chunks at run-time like dynamic, but the size of the chunk is variable. For guided, the parameter `chunk_size` acts as a guide for the minimum size chunk that a thread can receive. The exception for this is the last chunk – this chuck, in most cases will be smaller than `chunk_size`.

8.4 Runtime

This option allows the user to define the scheduling at `runtime` setting the environment variable `OMP_SCHEDULE` or using the function `omp_set_schedule`.

Table 4: Syntax For-loop Scheduling

Scheduling Option	Syntax (chunk size=4)
`Static`	`#pragma omp for schedule(static,4)`
`Dynamic`	`#pragma omp for schedule(dynamic,4)`
`Guided`	`#pragma omp for schedule(guided,4)`

Runtime	#pragma omp for schedule(runtime)

9 Printing

When there are print statements in a `parallel` region, either all the threads can execute these statements, or we can specify that only one thread should execute them. When the users need to control the printing of output, they should respond "**`y`**" when prompted with "**`Are you writing/printing anything from the parallelized region of the code?(Y/N)`**". IPT will then ask if only one thread is printing or if every thread will. If the user selects to have all threads print, IPT will not make any insertions, however if the user wishes for only one thread to print the output then IPT will create a `single` region for the print statement so that only one thread executes. IPT will create a temporary file that displays the code with numbers that correspond to each line. The user can view this temporary file and select the line that is printing output. Figure 31 shows an example of printing with one thread in a parallel region and with all threads in the team. The keys for IPT options required for generating the code in Figure 31 is shown in Tables 5 and 6.

Single thread printing

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

int main()
{
  int i;
  int x = 0;

#pragma  omp  parallel  default(none)
shared(x) private(i)
{

#pragma omp for reduction ( + :x)
  for (i = 0; i < 10; i++) {
    x = x + 1;
  }
}

#pragma omp single
  printf("I can print!\n");
}
```

All threads printing

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

int main()
{
  int i;
  int x = 0;

#pragma  omp  parallel  default(none)
shared(x) private(i)
{

#pragma omp for reduction ( + :x)
  for (i = 0; i < 10; i++) {
    x = x + 1;
  }
}
  printf("I can print!\n");
}
```

Figure 31. Compares single thread printing to all threads printing

Table 5: Key for running IPT with single thread printing

Prompts	***Responses***
Model, Function, Option	**2 (OpenMP), 1 (main), 2 (for-loop)**
Which for-loop?	**First (only loop in this program)**
Reduction variables and arrays	**Yes, int x, Addition**
Variable analysis for [x]	**2 (otherwise)**
Single thread and another loop	**No**

Are you writing/printing anything from the parallelized region of the code?(Y/N)	**Y**
Would you like to use 1.one thread to do the writing (for printing the results to the standard output or writing to a file) 2.multiple threads to do the writing (to a file) 3.none of the above	**1**
Is the writing happening from the same function that is being parallelized? (y/n)	**y**
For your convenience we have generated a file called numberedCode.C with line numbers corresponding to each line of your code. You can chose to show this file here or open this file in a different terminal. Do you want to show the file ? (Y/N)	**y**
Select the line numbers from the NumberCode.txt file in which you want to perform the operation, and enter the line numbers here (format:[a],[a-b],[a,b]).	**7 : printf("I can print!\n");**
Do you have any other data output calls?(Y/N)	**N**
Timing	**N**

Table 6: Key for running IPT with all threads printing

Prompts	***Responses***
Model, Function, Option	**2 (OpenMP), 1 (main), 2 (for-loop)**
Which for-loop?	**First (only loop in this program)**
Reduction variables and arrays	**Yes, int x, Addition**
Variable analysis for [x]	**2 (otherwise)**
Single thread and another loop	**No**
Are you writing/printing anything from the parallelized region of the code?(Y/N)	**Y**
Would you like to use 1.one thread to do the writing (for printing the results to the standard output or writing to a file)	**2**

`2.multiple threads to do the writing (to a file)`
 	
`3.none of the above`	

`Timing`	**`No`**

10 Timing

IPT can insert the required code for measuring the execution time for a statement or a region of code. IPT prompts the users asking if they want to time certain regions of the code, and if they responds positively, IPT prompts them to specify the line numbers of the code to time. Figure 32 shows the IPT prompts for timing.

```
Do you want to perform any timing (Y/N) ?
Y

Please choose the section of code that you want to time.
For your convenience we have generated a file called numberedCode.C with
line numbers corresponding to each line of your code. You can chose to show
this file here or open this file in a different terminal.
Do you want to show the file ? (Y/N)
Y

0 : int i;
1 : #pragma omp parallel default(none) private(i)
2 : #pragma omp for
3 : for ( i = 0;i < 4;i++) {
4 :     printf("Hello world!\n");
5 : }
6 : return 0;
Select the line numbers from the NumberCode.txt file in which you want to
perform the operation, and enter the line numbers here (format:[a],[a-
b],[a,b]).
4

Do you want to insert code to output the timing (Y/N)?
Y
```

Figure 32. IPT timing prompts for timing.c

The code generated by IPT is shown in Figure 33 and the output is shown in Figure 34. Notice IPT has inserted lines to call various timing functions from the OpenMP library. While in this example we choose to have the timing for a line printed to the command-line, these statements could have been suppressed. Please note that printing the timing for lines of code called with very high frequency may cause runtime issues and effect performance.

Program code
`#include <omp.h>`
`#include <stdio.h>`
`int main()`

```
{
  int i;

#pragma omp parallel default(none) private(i)
{

#pragma omp for
  for (i = 0; i < 16; i++) {
    double Openmp_time_start, Openmp_time_end, Openmp_delta_time;
    Openmp_time_start = omp_get_wtime( );
    printf("Hello world!\n");
    Openmp_time_end = omp_get_wtime( );
    Openmp_delta_time = Openmp_time_end - Openmp_time_start;
    printf("Time taken for the process is %lf\n",Openmp_delta_time);
  }
}
  return 0;
}
```

Figure 33. Parallelized version of a hello world program

```
Program output – code was run using 4 threads
Hello world!
Time taken for the process is 0.000138
Hello world!
Time taken for the process is 0.000007
Hello world!
Time taken for the process is 0.000006
Hello world!
Time taken for the process is 0.000007
```

Figure 34. Output from parallelized version of a hello world program

7. References

[1] Molecular Dynamics (MD) Simulation Application. Website accessed on August 10, 2017.
https://en.wikipedia.org/wiki/Molecular_dynamics

[2] Molecular Dynamics (MD) code. Website accessed on August 10, 2017.
https://people.sc.fsu.edu/~jburkardt/c_src/md/md.html

[3] Circuit Stability code. Web accessed September 20, 2019
https://people.sc.fsu.edu/~jburkardt/c_src/satisfy/satisfy.c

[4] Circuit Satisfiability code on GitHub:

https://github.com/ritua2/IPT/blob/master/PEARC18_OMP/exercises/exercises_3_to_7/circuit.c

[5] Fast Fourier Transform code. Web accessed September 20, 2019
https://people.sc.fsu.edu/~jburkardt/c_src/fft_serial/fft_serial.html

8. Appendix

Full Key for Fast Fourier Transform:

```
NOTE: We currently support only C and C++ programs.

Please select a parallel programming model from the following available
options:
1. MPI
2. OpenMP
3. CUDA
2

NOTE: As per the OpenMP standard, the parallel regions can have only one
entry point and only one exit point.
Branching out or breaking prematurely from a parallel region is not allowed.
Please make sure that there are no return/break statements in the region
selected for parallelization. However, exit/continue statements are allowed.

A list containing the functions in the input file will be presented, and you
may want to select one function at a time to parallelize it using multi-
threading.
Please choose the function that you want to parallelize from the list below
1 : main
2 : ccopy
3 : cfft2
4 : cffti
5 : cpu_time
6 : ggl
7 : step
8 : timestamp
1

Please select one of the following options (enter 1 or 2 or 3)
1. Create a parallel region (a group of threads will be created and each
thread will execute a block of code redundantly but in parallel)
2. Parallelize a for-loop (a group of threads will be created and each
thread will execute a certain number of iterations of a for-loop)
3. Create a parallel section (TBD - this mode is currently unavailable)
2

Loop
Note: With your response, you will be selecting or declining the
parallelization of the outermost for-loop in the code region shown below. If
instead of the outermost for-loop, there are any inner for-loops in this
code region that you are interested in parallelizing, then, you will be able
```

```
to select those at a later stage.

/*
  LN2 is the log base 2 of N.  Each increase of LN2 doubles N.
*/
for (ln2 = 1; ln2 <= 20; ln2++) {
  n = 2 * n;
/*
  Allocate storage for the complex arrays W, X, Y, Z.
  We handle the complex arithmetic,
  and store a complex number as a pair of doubles, a complex vector as a
doubly
  dimensioned array whose second dimension is 2.
*/
  w = ((double *)(malloc(n * sizeof(double ))));
  x = ((double *)(malloc((2 * n) * sizeof(double ))));
  y = ((double *)(malloc((2 * n) * sizeof(double ))));
  z = ((double *)(malloc((2 * n) * sizeof(double ))));
  first = 1;
  for (icase = 0; icase < 2; icase++) {
    if (first) {
      for (i = 0; i < 2 * n; i = i + 2) {
        z0 = ggl(&seed);
        z1 = ggl(&seed);
        x[i] = z0;
        z[i] = z0;
        x[i + 1] = z1;
        z[i + 1] = z1;
      }
    }
     else {
      for (i = 0; i < 2 * n; i = i + 2) {
/* real part of array */
        z0 = 0.0;
/* imaginary part of array */
        z1 = 0.0;
        x[i] = z0;
/* copy of initial real data */
        z[i] = z0;
        x[i + 1] = z1;
/* copy of initial imag. data */
        z[i + 1] = z1;
      }
    }
/*
  Initialize the sine and cosine tables.
*/
    cffti(n,w);
/*
  Transform forward, back
*/
    if (first) {
      sgn = +1.0;
      cfft2(n,x,y,w,sgn);
      sgn = - 1.0;
      cfft2(n,y,x,w,sgn);
/*
```

```
  Results should be same as the initial data multiplied by N.
*/
      fnm1 = 1.0 / ((double )n);
      error = 0.0;
      for (i = 0; i < 2 * n; i = i + 2) {
        error = error + pow(z[i] - fnm1 * x[i],2) + pow(z[i + 1] - fnm1 *
x[i + 1],2);
      }
      error = sqrt(fnm1 * error);
      printf("  %12d  %8d  %12e",n,nits,error);
      first = 0;
    }
     else {
      ctime1 = cpu_time();
      for (it = 0; it < nits; it++) {
        sgn = +1.0;
        cfft2(n,x,y,w,sgn);
        sgn = - 1.0;
        cfft2(n,y,x,w,sgn);
      }
      ctime2 = cpu_time();
      ctime = ctime2 - ctime1;
      flops = 2.0 * ((double )nits) * (5.0 * ((double )n) * ((double )ln2));
      mflops = flops / 1.0E+06 / ctime;
      printf("  %12e  %12e  %12f\n",ctime,ctime / ((double )(2 *
nits)),mflops);
    }
  }
  if (ln2 % 4 == 0) {
    nits = nits / 10;
  }
  if (nits < 1) {
    nits = 1;
  }
  free(w);
  free(x);
  free(y);
  free(z);
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.
Note: With your response, you will be selecting or declining the
parallelization of the outermost for-loop in the code region shown below. If
instead of the outermost for-loop, there are any inner for-loops in this
code region that you are interested in parallelizing, then, you will be able
to select those at a later stage.

for (icase = 0; icase < 2; icase++) {
  if (first) {
    for (i = 0; i < 2 * n; i = i + 2) {
      z0 = ggl(&seed);
      z1 = ggl(&seed);
      x[i] = z0;
      z[i] = z0;
      x[i + 1] = z1;
      z[i + 1] = z1;
    }
  }
   else {
    for (i = 0; i < 2 * n; i = i + 2) {
/* real part of array */
      z0 = 0.0;
/* imaginary part of array */
      z1 = 0.0;
      x[i] = z0;
/* copy of initial real data */
      z[i] = z0;
      x[i + 1] = z1;
/* copy of initial imag. data */
      z[i + 1] = z1;
    }
  }
/*
  Initialize the sine and cosine tables.
*/
  cffti(n,w);
/*
  Transform forward, back
*/
  if (first) {
    sgn = +1.0;
    cfft2(n,x,y,w,sgn);
    sgn = - 1.0;
    cfft2(n,y,x,w,sgn);
/*
  Results should be same as the initial data multiplied by N.
*/
    fnm1 = 1.0 / ((double )n);
    error = 0.0;
    for (i = 0; i < 2 * n; i = i + 2) {
      error = error + pow(z[i] - fnm1 * x[i],2) + pow(z[i + 1] - fnm1 * x[i
+ 1],2);
    }
    error = sqrt(fnm1 * error);
    printf("  %12d  %8d  %12e",n,nits,error);
    first = 0;
  }
   else {
    ctime1 = cpu_time();
    for (it = 0; it < nits; it++) {
      sgn = +1.0;
      cfft2(n,x,y,w,sgn);
      sgn = - 1.0;
      cfft2(n,y,x,w,sgn);
    }
    ctime2 = cpu_time();
    ctime = ctime2 - ctime1;
    flops = 2.0 * ((double )nits) * (5.0 * ((double )n) * ((double )ln2));
    mflops = flops / 1.0E+06 / ctime;
    printf("  %12e  %12e  %12f\n",ctime,ctime / ((double )(2 *
nits)),mflops);
  }
```

```
}
Is this the for-loop you are looking for?(y/n)
n

OK - will find the next loop if available.

for (i = 0; i < 2 * n; i = i + 2) {
  z0 = ggl(&seed);
  z1 = ggl(&seed);
  x[i] = z0;
  z[i] = z0;
  x[i + 1] = z1;
  z[i + 1] = z1;
}
Is this the for-loop you are looking for?(y/n)
y

Reduction variables are the variables that should be updated by the OpenMP
threads and then accumulated according to a mathematical operation like sum,
multiplication,etc.
Do you want to perform reduction on any variable ?(Y/N)
n

Do you want to do element-wise reduction on any array?(Y/N)
n

IPT is unable to perform the dependency analysis of the array named [ x ] 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.
2

IPT is unable to perform the dependency analysis of the array named [ z ] 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.
2

Do you want to use any scheduling scheme for this loop (y/n) ?
n

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)
n

Would you like to parallelize another loop?(Y/N)
n

Are you writing/printing anything from the parallelized region of the
code?(Y/N)
n

Do you want to perform any timing (Y/N) ?
n
```

Full Key For Game of Life Example:

```
Please enter which type of parallel program you want to create. Type 1 for
MPI, 2 for OpenMP or 3 for Cuda.
2

Please note that the default setting for the initialization function is set
to main.
Would you like this program to be in offload mode?(Y/N)
n

Would you like to parallelize a for-loop?(Y/N)
y

Please enter the function in which you wish to insert a pragma(function to
parallelize).
main

for(j = 0;j <(N + 2);j++) temp[0][j] =(temp[M + 1][j] = 0);
Is this the for-loop you are looking for?(y/n)
n

for(i = 1;i <(M + 1);i++) temp[i][0] =(temp[i][N + 1] = 0);
Is this the for-loop you are looking for?(y/n)
n

for(k = 0;k < NTIMES;k++) {temp = compute(life,temp,M,N);ptr = temp;temp =
life;life = ptr;}
Is this the for-loop you are looking for?(y/n)
y

Adding k to private variables.

Please enter the number of variables to reduce. If there are no variables to
reduce please enter 0.
0

Adding NTIMES to shared variables.

firstprivate variables added temp with type int **

firstprivate variables added life with type int **

firstprivate variables added ptr with type int **

Adding M to shared var

Adding N to shared var
Are there any lines of code that you would like to run by a single thread at
a time?(Y/N)n

Would you like to parallelize another loop?(Y/N)
N
```