An introduction to parallelizing code with IPT and MPI

1. Introduction to IPT and MPI

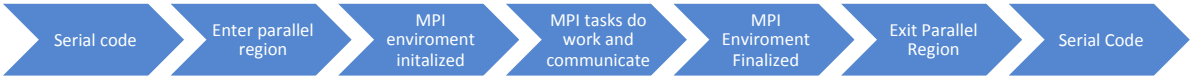
IPT stands for the interactive parallelization tool. IPT was developed as a high-level tool that allows users to parallelize programs for HCP platforms as well as provide user who are new to parallelization a way to gain familiarity with the basic concepts and syntax. After asking the user a series of questions about their program and performing some analysis, IPT inserts necessary lines of code run the program in parallel with the user's parallelization paradigm of choice.

MPI is one of the parallelization paradigms that users can choose from when using IPT. MPI stands for message passing interface and defines how message passing libraries should be written according to standards set by the MPI Forum. MPI itself is not a library, but the libraries such as MPICH and OpenMPI are implementations of MPI. Essentially, MPI defines a way information to be communicated between two systems running a program in parallel. MPI is used for programming systems with distributed memory. **Distributed memory systems** do not share address space. When running a program in parallel on distributed memory system, one process is blind to the work being performed by another process. MPI has defined functions that allow these separate processes to communicate with one another. MPI was developed before shared memory systems grew in popularity, but the MPI implementations that support MPI 2 and beyond support threading for shared memory systems and hybrid systems as well.

The purpose of this guide is to help the user learn how to use IPT as well as introduce the basics creating parallel programs with MPI. For the user to properly answer IPT's prompts they should be very familiar with code and understand how the concepts of data distribution, synchronization, and loop dependency analysis apply to the program. Examples of how to apply these concepts when using IPT will be demonstrated through out the guide.

2. Basic MPI Programs: Hello World Example

The general structure of an MPI program follows these steps: enter the parallel region, initialize MPI environment, multiple process do work in parallel region and make message passing calls between MPI processes, terminate MPI environment, exit parallel region.



Let's being paralleling a simple hello world program using MPI. The code for this example can be seen in Figure 1 below. The example prints "It is a wonderful day."

```
#include <stdio.h>

int main(){
   printf("Wonderful Day!\n");
   return 0;
}
```

Figure 1: Source code for Example0.c a hello world example

In order to parallelize this example, IPT will ask the user a series of questions to gather information about the code. The user's response to these prompts will help IPT determine how to parallelize the code. To run IPT on this example use the command:

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

Figure 2 below provides a step by step walk through of how to parallelize this example using IPT. All responses to IPT's prompts by the user have been bolded.

```
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
1

Please note that by default, the MPI Environment Initialization functions
will be set in the main function.

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

Please select a pattern from the following list that best characterizes your
parallelization needs:
(Please refer to the user-guide for the explanation on each of the patterns,
and note that not all the listed patterns may be relevant for your
application type.)
1. For-Loop Parallelization
2. Stencil
3. Pipeline
4. Data Distribution and Data Collection
5. Data Distribution
6. Data Collection
7. Point to Point
8. MPI environment initialization and finalization
9. Master-worker pattern - Data distribution - Broadcast
8

Do you want to insert the code to determine the process rank and the total
processes that are used ? (y/n)
y

Do you want to parallelize other part of the program ?(Y/N)
n

Are you printing anything?(Y/N)
n

Are you writing or reading anything from file(s) ?(Y/N)
n
Do you want to perform any timing (Y/N)?
n
```

Choosing MPI as parallelization paradigm

Creates MPI environment

Figure 2: Using IPT to parallelize a hello world program

Now let' examine the parallelized version of the code created by IPT. The output file will be formatted using the following naming convention rose_<filename>_MPI.c. For this code, the output file name is rose_exercise1_MPI.c. From this example there are some basics of MPI programs that can be explained. Notice the #include <MPI.h> has been added to include the MPI library to this program. This library as well as the functions **MPI_Init( )** and **MPI_Finalize( )** will appear in all MPI programs. All MPI programs must start and terminate with calling MPI_INIT and MPI_Finalize. These functions create and destroy the MPI environment for the program, and processes can only communicate between when these

functions have been called. MPI_Init can take either two null arguments or the two command line arguments: &argc and &argv. Remember that when running an MPI program, the code for one program is running multiple processes on separate cores which can span multiple nodes/machines. Each process is given a **rank** which acts as the process identifier. No two processes will have the same rank, and the master process is given the rank=0. Processes belong to MPI communicators. Processes inside the same **communicator** can exchange messages with one another. The default communicator constructed by MPI is **MPI_COMM_WORLD** which includes all processes in the program. As a result, these two functions seen in almost every MPI program are **MPI_Comm_size() and MPI_Comm_rank()**.

Version 1

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

int main()
{
  int rose_size;
  int rose_rank;
  MPI_Init(NULL,NULL);
  MPI_Comm_size(MPI_COMM_WORLD,&rose_size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rose_rank);
  printf("Wonderful Day!\n");
  MPI_Finalize();
  return 0;
}
```

MPI header file inserted

Initialize MPI environment

Terminate MPI environment

Figure 3: Parallelized version of Example1.c

MPI_Comm_size( ) takes the communicator as an input and returns the size of that communicator. The size of the communicator should be the number of processes requested for the job. MPI_Comm_rank returns the rank of the process. The rank acts as an id and allows the program to identify itself within the communicator. Rank can be used to not only assign work to certain processes but also plays a role in message communication by providing context for which processes is sending and receiving a message. The range of rank is between 0 and N-1 where N is the size of the communicator. Size is often used for work distribution. For example, when parallelizing a for loop, the iterations can be split on among multiple processes. The chunk size of iterations that each process receives could be determined by the dividing the number of iterations by the size of the communicator. There multiple examples of programs that do this in the later sections of the guide.

3. Running and Compiling MPI Programs

If running the program on in Stampede2 development platform, log in an create an idev session. When creating a session, request the desired number of nodes and cores desired using the -N flag for nodes and -n flag for cores. A command requesting 2 nodes and 4 cores would is constructed like this:

```
$ idev -N 2 -n 4
```

Compile the MPI program using the intel compiler mpicc. To compile the hello world program from Section 1, use the following command:

```
$ mpicc -o rose_example1_MPI rose_example1_MPI.c
```

To run the executable created after compiling using the command above use the following command. In order to run MPI program, special commands are needed to ensure the executable is run on all processed.

```
$ ibrun rose_example1
```

The ibrun command will launch the executable on each of the cores. The output of the parallelized version can be seen below in Figure 4. Since the program was run on 8 nodes the print statement was performed 8 times by 8 separate processes. Each of these processes ran a different copy of the program and allocated its own resources and memory to execute the code.

```
Wonderful Day!
Wonderful Day!
Wonderful Day!
Wonderful Day!
Wonderful Day!
Wonderful Day!
Wonderful Day!
Wonderful Day!
```

Figure 4: Output of Example1.c when run on 8 nodes

The parallelized version of Example1.c could have been edited to show the rank of each process. To make this change the line of code in Figure 4 has been bolded. The change in output can also be seen in Figure 4. In Example 1, the code was run on multiple processes, but no communication occurred between any of the processes. Notice that the processed do not print in order of rank. The order is random. If this program was run again, the order of the processes would likely be different. The next section will introduce various MPI communication schemes in a slightly more complicated example.

Version 2: Example 1

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

int main()
{
```

```
  int rose_size;
  int rose_rank;
  MPI_Init(NULL,NULL);
  MPI_Comm_size(MPI_COMM_WORLD,&rose_size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rose_rank);
  printf("Wonderful Day! From process %d\n", rose_rank);
  MPI_Finalize();
  return 0;
}
```

Output Version 2: Example 1

```
Wonderful Day! From process 4
Wonderful Day! From process 5
Wonderful Day! From process 6
Wonderful Day! From process 7
Wonderful Day! From process 0
Wonderful Day! From process 1
Wonderful Day! From process 2
Wonderful Day! From process 3
```

Figure 5: Code and output from Example 1 edited to show rank of each process

Often when creating MPI programs, after inserting the MPI code other changes can be made for optimizations or compiler specifications. This is because the process of parallelization is iterative. While IPT can automatically make many of the changes needed for parallelization, sometimes the programmer may want to make further changes. Figure 6 below demonstrates this design process.

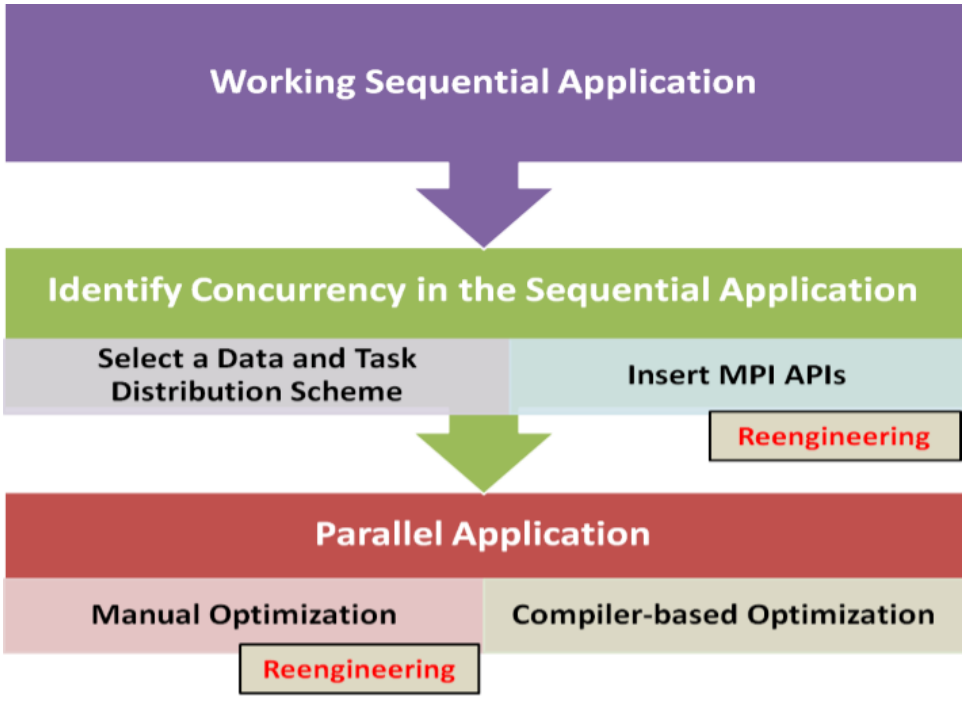


Figure 6: Graphic depicting the process of parallelization

4. Introduction to Data Distribution: Reduction Example

One of the concepts that users should understand when responding to IPT's prompts is data distribution. Data distribution occurs because multiple MPI processes have separate address space, so each process has a local value for a variable. In the case of reduction, once process needs to collect the value to calculate the true global value. **Reduction** occurs when one value is calculated by combining multiple elements using the same operation to create one result. **Reduction variables** are variables the accumulate; **reduction operations** are operations used to combine the elements. The code from Example2.c shown in Figure 5 shows a common example of reduction found in many for loops. The line where reduction occurs is shown in bold. For this example, the reduction element is sum and the reduction operation is addition.

```
#include <stdio.h>

int main(int argc,char *argv[]){
    int i, sum,upToVal;
    upToVal = 10000;
    sum = 0;

    for(i=0; i<= upToVal; i++){
         sum = sum +i;
    }
    printf("\nSum of first %d numbers is: %d\n\n",upToVal, sum);
    return 0;
}
```

Figure 7: Snippet from Exercise2.c that shows reduction

When the serial version of Exercise2.c is compiled and run, the output is **"`Sum of first 10000 numbers is: 50005000.`"** To create the parallel version, IPT will need know a little more about the program then in the when parallelizing Example1.c. In this example, parallelization hotspot is the for loop. The loop has 1000 iterations that could be split up over several MPI processes. As mentioned before, this loop contains a reduction calculation. To handle reduction, each MPI task computes a partial value of the variable sum. At the end the, the partial sums will be added together to create the true value, 50005000. One MPI task should be used to print this value. Let's parallelize this example using IPT. Figure 8 below walks the user through how to respond to IPT's prompts with reduction.

```
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
```

Using MPI

```
1

Please note that by default, the MPI Environment Initialization functions
will be set in the main function.

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

Please select a pattern from the following list that best characterizes your
parallelization needs:
(Please refer to the user-guide for the explanation on each of the patterns,
and note that not all the listed patterns may be relevant for your
application type.)
1. For-Loop Parallelization
2. Stencil
3. Pipeline
4. Data Distribution and Data Collection
5. Data Distribution
6. Data Collection
7. Point to Point
8. MPI environment initialization and finalization
9. Master-worker pattern - Data distribution - Broadcast
1

for (i = 0; i <= upToVal; i++) {
  sum = sum + i;
}
Is this the for loop you are looking for?(y/n)
y

Please specify the type of the data storage (variable, array, structure)
from which the data collection should be done at the hotspot for
parallelization:
1. Variable/s
2. Array/s
3. Structure/s (not supported
currently)
4. Both Variable/s and Array/s
5. Both Variable/s and Structure/s (not supported currently)
6. Variable/s, Array/s, Structure/s (not supported currently)
7. None of the above
1

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

Please select the reduce operation to use for variable [sum]
1.  Sum
2.  Product
3.  Min
4.  Max
5.  Logical and
6.  Bit-wise and
```

Parallelizing a for loop

Data storage is occurring in variable sum

The name of the reduction variable is sum

```
7.  Logical or
8.  Bit-wise or
9.  Logical xor
10.  Bit-wise xor
11.  Max value and location
12.  Min value and location
1

Would you like to send the results after reducing the chosen variable to all
processes or to only one?(1. all 0. one).
Note: if option "0" is chosen then only one MPI process will have the
combined results. Please refer to the user-guide for further information on
this.
0

Would you like to do this MPI pattern again?(Y/N)
n

Are you printing anything?(Y/N)
y

Would you like to use
1.one process to do the printing (for printing the results to the standard
output)
2.multiple processes to do the printing
3.none of the above
1

NOTE: To make sure that one process
is printing to the data, we will need to know at which lines you are
printing data.

Is the printing 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
0 : int rose_sum0;
1 : int rose_size;
2 : int rose_rank;
3 : int i;
4 : int sum;
5 : int upToVal;
6 : MPI_Init(&argc,&argv);
7 : MPI_Comm_size(MPI_COMM_WORLD,&rose_size);
8 : MPI_Comm_rank(MPI_COMM_WORLD,&rose_rank);
9 : upToVal = 10000;
10 : sum = 0;
11 : int rose_upper_limit0;
12 : int rose_lower_limit0;
13 : int rose_range0;
14 : if (rose_rank > 0;) {
15 :    sum = 0;
```

The reduction operation is addition/summation

One process will combine the partial values of sum

Printing the final value of sum

Only want one process to print the value

```
16 : }
17 : for ( i = rose_lower_limit0;;;i <= rose_upper_limit0;;i++) {
18 :    sum = sum + i;
19 : }
20 : MPI_Reduce(&sum,&rose_sum0,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
21 : sum = rose_sum0;
22 : printf("\nSum of first %d numbers is: %d\n\n",upToVal,sum);
23 : MPI_Finalize();
24 : 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]).
22
```

Printing occurring in line 22

```
Warning: attachArbitraryText(): attaching arbitrary text to the AST as a #if
declaration: text = if(rose_rank==0) {

Warning: attachArbitraryText(): attaching arbitrary text to the AST as a #if
declaration: text =
}

Do you have any other data output calls?(Y/N)
n

Are you writing or reading anything from file(s) ?(Y/N)
n

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

Figure 8: Reduction Example - Parallelizing Example2.c with IPT

The parallelized version of this program can be seen below in Figure 9. IPT has inserted quite a few changes so let's discuss them. The changes can be summarized in the following way, MPI environment initialization, editing the for-loop bounds, communication to handle reduction, and MPI termination.

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

int main(int argc,char *argv[])

  int rose_sum0;
  int rose_size;
  int rose_rank;
  int i;
  int sum;
  int upToVal;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&rose_size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rose_rank);
  upToVal = 10000;
  sum = 0;
  int rose_upper_limit0;
  int rose_lower_limit0;
  int rose_range0;
```

MPI environment initialization

Division of work: rose_range0 is the number of iterations given to each MPI task.

```
rose_range0 = (upToVal - 0 )/ rose_size;

double rose_range_double0 = (double) (upToVal - 0 )/ rose_size;
if (rose_range_double0 <= 1 ) {
  rose_upper_limit0 = ((rose_rank+0)*1 + 0 <= upToVal ) ? (rose_rank+0)*1 +
  0 : upToVal;
  rose_lower_limit0= rose_rank*1 + 0;
}
else {
if ( rose_rank  > 0 ) {
  rose_lower_limit0 = (rose_rank-
1)*rose_range0 + rose_range0 -
  (((rose_rank-1)*rose_range0 +
rose_range0 )% 1) + 0 ;
 }
else {
rose_lower_limit0 = 0;
}
if (rose_rank < rose_size -1)  {
rose_upper_limit0 =  (rose_rank)*rose_range0 + rose_range0 -
(((rose_rank)*rose_range0 +  rose_range0) % 1) + 0 - 1;
}
else {
rose_upper_limit0 = upToVal ;
}
}
  if (rose_rank > 0) {
    sum = 0;
  }
  for (i = rose_lower_limit0; i <=
rose_upper_limit0; i++) {
    sum = sum + i;
  }

  MPI_Reduce(&sum,&rose_sum0,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
  sum = rose_sum0;

if(rose_rank==0) {
  printf("\nSum of first %d numbers is: %d\n\n",upToVal,sum);
}
  MPI_Finalize();
  return 0;
}
```

Calculating the number of iterations of the for-loop that each MPI process will compute, and finding the start and end values for the loop iteration

For loop controls modified based on calculations above. The upper and lower limits on the iterations are based on the rank of the task as well as the size of the communicator.

MPI_Reduce is a function for handling reductions. Adds up local results

Let's only the master process print. (Rank=0)

Figure 9: Parallelized Code from Example2.c a reduction example

5. MPI Communication Models

5.1 Data Movement

There are numerous modes of ways for threads to communicate in MPI. Communication may happen between just two processes, point to point communication, or between more than two processes collective communication. Collective communication is used for data movement as well as collective computation. For data movement, a value can be broadcasted, scattered, or gathered. If data is broadcast, one process will send the value to other processes. If data is scattered, values from one process will be dispersed among other processes. Gather works in opposite fashion; values from multiple processes are sent to one. In all gather, values from multiple process are gathered and then broadcast to all processes. The diagrams in Figure 10 below visually demonstrate these types of data movement.

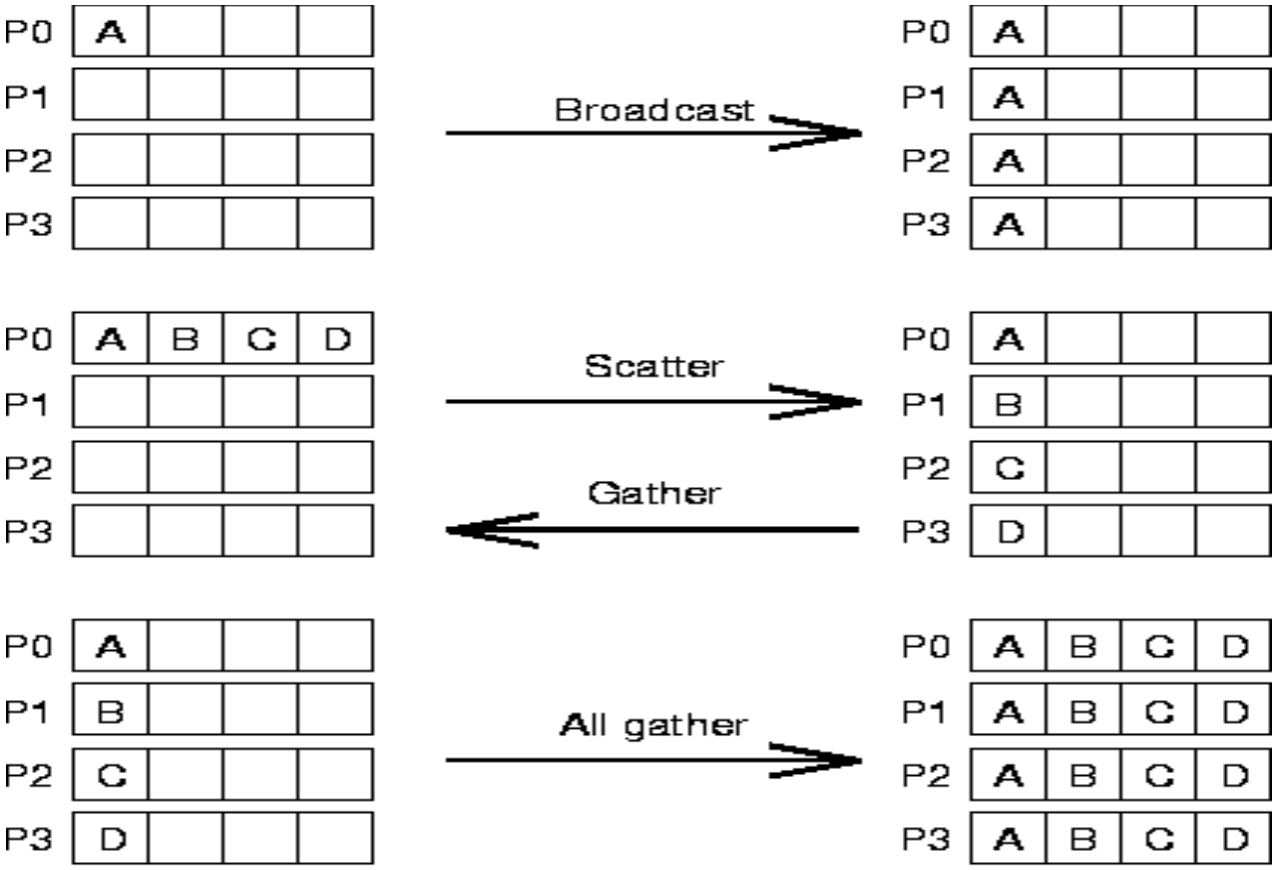


Figure 10: Types of Data Movement – Broadcast, Scatter, Gather, All Gather

For collective computations like reduction, similar types of communication take place must take place. Recall the reduction example from the Section 4. In order to find the total value of sum, the local values of sum needed to be combined by one process. The function MPI_Reduce was called to handle the computation. In order to perform the reduction, the MPI processes needed to communicate their local values of sum to the master process. This communication would occur in a gather like fashion where the

master process would receive all the local values. Then the master process would sum the local values to create the true value of sum. A diagram to demonstrate this communication can be seen in Figure 11.

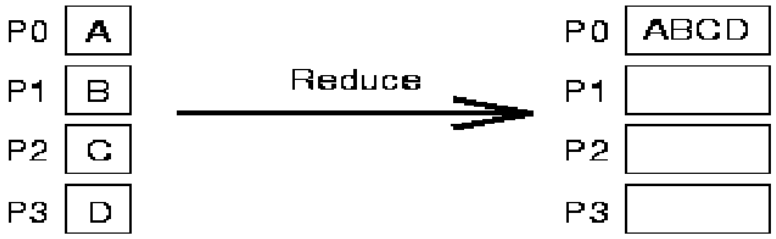


Figure 11: Diagram showing how process communication during the collective computation

6. Additional Examples

6.1 Searching Example

This example demonstrates how parallelization can be used to significantly speed up for-loops used while searching for a value. The main program calls the search function with parameters A, B, and C. The function searches for a value j in the range from A to B where f(j)=C. The function **`search`** has a for loop which is a hotspot for parallelization. The serial code for the function search can be seen in Figure 12 below.

```
int search ( int a, int b, int c ){
  int fi;
  int i;
  int j;
  j = -1;
  for ( i = a; i <= b; i++ ){
    fi = f ( i );
    if ( fi == c ){
      j = i;
      break;
    }
  }
  return j;
}
```

Search returns the value j from the range of [a,b] where f(j)=c. If there is no value in the range, the return -1. This for loop is a parallelization hotspot.

Figure 12: Snippet that shows code for function search in search.c

The work of this loop can be distributed over multiple MPI tasks because no data dependencies between iterations. Each iteration is independent from the other iterations in the loop. This means that no matter what order the iterations were run in, their output would be the same. Figure 13 below serves as a guide to answering IPT's prompts in this example.

```
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
1

Please note that by default, the MPI Environment Initialization functions
will be set in the main function.

Please choose the function that you want to parallelize from the list below
1 : main
2 : search
3 : f
4 : timestamp
5 : cpu_time
2

Please select a pattern from the following list that best characterizes your
parallelization needs:
(Please refer to the user-guide for the explanation on each of the patterns,
and note that not all the listed patterns may be relevant for your
application type.)
1. For-Loop Parallelization
2. Stencil
3. Pipeline
4. Data Distribution and Data Collection
5. Data Distribution
6. Data Collection
7. Point to Point
8. Master-worker pattern- Data distribution - Broadcast
1

for (i = a; i <= b; i++) {
  fi = f(i);
  if (fi == c) {
    j = i;
    break;
  }
}
Is this the for loop you are looking for?(y/n)
y

Please specify the type of the data storage (variable, array, structure)
from which the data collection should be done at the hotspot for
parallelization:
1. Variable/s
2. Array/s
3. Structure/s (not supported currently)
4. Both Variable/s and Array/s
5. Both Variable/s and Structure/s (not supported currently)
6. Variable/s, Array/s, Structure/s (not supported currently)
7. None of the above
7

Would you like to do this MPI pattern again?(Y/N)
n
```

No data vollection across MPI processes occurs in loop.

```
Are you printing anything?(Y/N)
n

Are you writing or reading anything from file(s) ?(Y/N)
n
Do you want to perform any timing (Y/N)?
n
```

Figure 13: IPT prompts for parallelizing search.c

The insertions from IPT in the function search can be seen in Figure 14 below. One should expect to see the three main changes to the code by IPT: MPI initialization, distribution of loop iterations, and MPI finalization. Like in almost all MPI programs there are calls to MPI_Comm_size( ) and MPI_Comm_rank( ). While in the function search there no calls of the functions MPI_Init( )and MPI_Finalize( ), these functions are called in the main program before and after respectively before the function search( ) is called.

```
int search(int a,int b,int c)
{
  int rose_size;
  int rose_rank;
  int fi;
  int i;
  int j;
  j = - 1;
  MPI_Comm_size(MPI_COMM_WORLD,&rose_size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rose_rank);
  int rose_upper_limit0;
  int rose_lower_limit0;
  int rose_range0;
rose_range0 = (b - i )/ rose_size;
double rose_range_double0 = (double) (b - i )/ rose_size;
if (rose_range_double0 <= 1 ) {
rose_upper_limit0 = ((rose_rank+0)*1 + i <= b ) ? (rose_rank+0)*1 + i : b;
rose_lower_limit0= rose_rank*1 + i;
}
else {
if ( rose_rank  > 0 ) {
rose_lower_limit0 = (rose_rank-1)*rose_range0 + rose_range0 - (((rose_rank-
1)*rose_range0 + rose_range0 )% 1) + i ;
 }
else {
rose_lower_limit0 = i;
}
if (rose_rank < rose_size -1)  {
rose_upper_limit0 =
(rose_rank)*rose_range0 + rose_range0 -
(((rose_rank)*rose_range0 +  rose_range0)
% 1) + i - 1;
}
else {
rose_upper_limit0 = b ;
```

Determining size of communicator and rank of MPI process

Calculating the number of iterations that each MPI task will receive which is depended upon [a,b] and size of communicator. The loop upper and lower bounds of the loop vary depending on the process rank and number of iterations give to each process.

```
}
}
  for (i = rose_lower_limit0; i <= rose_upper_limit0; i++) {
    fi = f(i);
    if (fi == c) {
      j = i;
      break;
    }
  }
  return j;
}
```

Figure 14: Snippet from search function that shows IPT insertions from parallelized version of search.c

The table below shows the results from both the serial and parallelized version. The parallel version was run on 4 knights landing nodes on TACC's Stampede2 supercomputer. The speed was almost about 25 times as fast. It is important to note that not all programs run time will increase by the same degrees as this example. There are even cases when parallelization will be slower than the serial version. This is why it is important for the programmer to understand what areas are hotspots for parallelization and try to optimize these regions to make them more efficient.

Table 1: Testing serial and parallel version program outputs

Program	search.c	rose_search_MPI.c
Command to compile (intel compilers)	icc search.c	mpicc rose_search_MPI.c
Command to run	./a.out	ibrun ./a.out
Output	SEARCH_SERIAL:C version Search the integers from A to B for a value J such that F(J) = C. A = 1 B = 2147483647 C = 45 Found J = 1674924981 Verify F(J) = 45 **Elapsed CPU time is 100.01** SEARCH_SERIAL: Normal end of execution.	SEARCH_SERIAL:C version Search the integers from A to B for a value J such that F(J) = C. A = 1 B = 2147483647 C = 45 Found J = 1674924981 Verify F(J) = 45 **Elapsed CPU time is 4.03** SEARCH_SERIAL: Normal end of execution.

6.4 Prime Numbers Example

This program finds the number of prime numbers between 0 and N where N= 10, 100, 1000, 10000. The function prime( ) is called each time to determine the number of primes between 0 and N, and the results are printed to the screen. The function prime( ) determines the if a number is prime by brute force, checking if a number can be evenly divided. This program takes about N^2 to complete.

```
int prime_number(int n {
  int i;
  int j;
  int prime;
  int total;

  total = 0;

  for ( i = 2; i <= n; i++ ){
    prime = 1;
    for ( j = 2; j < i; j++ ){
      if ( ( i % j ) == 0 ){
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
  return total;
}
```

Figure 15: Snippet that shows hot spot for parallelization in compute function from prime.c

When calling IPT with this example, the things to keep in mind are the target region, for loop parallelization, reduction of the variable total, and printing. The hotspot for parallelization in this example is the function prime_numbers( ). The pattern for parallelization is the for loop scheme since each loop iteration can be run independently of one another. The variable total reduced so it is important to call out this data storage to IPT. Lastly in order to print only the final results and avoid printing meaningless lines, IPT should be told to print in only one process. In order to achieve the results, follow the IPT prompts shown below in Figure 16.

```
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
1

Please note that by default, the MPI Environment Initialization functions
will be set in the main function.

Please choose the function that you want to parallelize from the list below
1 : cpu_time
2 : prime_number
```

```
3 : timestamp
4 : main
2

Please select a pattern from the following list that best characterizes your
parallelization needs:
(Please refer to the user-guide for the explanation on each of the patterns,
and note that not all the listed patterns may be relevant for your
application type.)
1. For-Loop Parallelization
2. Stencil
3. Pipeline
4. Data Distribution and Data Collection
5. Data Distribution
6. Data Collection
7. Point to Point
8. Master-worker pattern- Data distribution - Broadcast
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.

for (i = 2; i <= n; i++) {
  prime = 1;
  for (j = 2; j < i; j++) {
    if (i % j == 0) {
      prime = 0;
      break;
    }
  }
  total = total + prime;
}
Is this the for loop you are looking for?(y/n)
y

Please specify the type of the data storage (variable, array, structure)
from which the data collection should be done at the hotspot for
parallelization:
1. Variable/s
2. Array/s
3. Structure/s (not supported currently)
4. Both Variable/s and Array/s
5. Both Variable/s and Structure/s (not supported currently)
6. Variable/s, Array/s, Structure/s (not supported currently)
7. None of the above
1

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

```
Please select the reduce operation to use for variable [total]
1.  Sum
2.  Product
3.  Min
4.  Max
5.  Logical and
6.  Bit-wise and
7.  Logical or
8.  Bit-wise or
9.  Logical xor
10.  Bit-wise xor
11.  Max value and location
12.  Min value and location
1

Would you like to send the results after reducing the chosen variable to all
processes or to only one?(1. all 0. one).
Note: if option "0" is chosen then only one MPI process will have the
combined results. Please refer to the user-guide for further information on
this.
0

Would you like to do this MPI pattern again?(Y/N)
n

Are you printing anything?(Y/N)
y

Would you like to use
1.one process to do the printing (for printing the results to the standard
output)
2.multiple processes to do the printing
3.none of the above
1

NOTE: To make sure that one process is printing to the data, we will need to
know at which lines you are printing data.

Is the printing happening from the same function that is being parallelized?
(y/n)
n

Please choose the function that you want to parallelize from the list below
1 : cpu_time
2 : prime_number
3 : timestamp
4 : main
4

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 rose_size;
1 : int rose_rank;
2 : int n_hi;
```

```
3 : int n_low;
4 : int p;
5 : int primes;
6 : int primes_part;
7 : int n;
8 : int n_factor;
9 : double wtime;
10 : MPI_Init(NULL,NULL);
11 : MPI_Comm_size(MPI_COMM_WORLD,&rose_size);
12 : MPI_Comm_rank(MPI_COMM_WORLD,&rose_rank);
13 : n_low = 1;
14 : n_hi = 1000000;
15 : n_factor = 10;
16 : printf(" N # Primes \n");
17 : while ( n_low <= n_hi  {
18 :    primes = prime_number(n_low);
19 :    printf(" %8d %8d\n",n_low,primes);
20 :    n_low = n_low * n_factor;
21 : }
22 : MPI_Finalize();
23 : 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]).
16,19

Do you have any other data output calls?(Y/N)
n

Are you writing or reading anything from file(s) ?(Y/N)
n

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

Figure 16: IPT prompts for parallelizing prime.c

The Figure 17 below shows only the changes made in the function prime_number( ). IPT also would insert calls of MPI_Init( ) and MPI_Finialize( ) in the main function of the program. Statements that allow only the rank=0 process to print were also inserted so that output would be easy to understand. For printing results, it will often be desired to have only one process print. Notice that the controls for the for loop have been changed based on the communicator size and the process rank. When the parallel version of this code was run on 4 process, the code finished in about 4 minutes. When the serial version of this program was run, the code took over 17 minutes. The time to run the serial version of this program is exponentially related to the size of n, so when n=1,000,000 the execution time is very long.

```
int prime_number(int n){
  int rose_total0;
  int rose_size;
  int rose_rank;
  int i;
  int j;
```

```
  int prime;
  int total;
  total = 0;
  MPI_Comm_size(MPI_COMM_WORLD,&rose_size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rose_rank);
  int rose_upper_limit0;
  int rose_lower_limit0;
  int rose_range0;
rose_range0 = (n - 2 )/ rose_size;
double rose_range_double0 = (double) (n - 2
)/ rose_size;
if (rose_range_double0 <= 1 ) {
  rose_upper_limit0 = ((rose_rank+0)*1 + 2 <= n ) ? (rose_rank+0)*1 + 2 : n;
  rose_lower_limit0= rose_rank*1 + 2;
}
else {
  if ( rose_rank  > 0 ) {
    rose_lower_limit0 = (rose_rank-1)*rose_range0 + rose_range0 -
    (((rose_rank-1)*rose_range0 + rose_range0 )% 1) + 2 ;
   }
  else {
    rose_lower_limit0 = 2;
  }
  if (rose_rank < rose_size -1)  {
    rose_upper_limit0 =  (rose_rank)*rose_range0 + rose_range0 -
    (((rose_rank)*rose_range0 +  rose_range0) % 1) + 2 - 1;
  }
  else {
    rose_upper_limit0 = n ;
  }
}
  if (rose_rank > 0) {
    total = 0;
  }
  for (i = rose_lower_limit0; i <= rose_upper_limit0; i++) {
    prime = 1;
    for (j = 2; j < i; j++) {
      if (i % j == 0) {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
  MPI_Reduce(&total,&rose_total0,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
  total = rose_total0;
  return total;
}
```

Distribution of iterations determined by number of processes. Loop controls for each process determined by process rank and chunk size of iterations.

MPI_Reduce( ) called to aggerate local copies of the value total to global value.

Figure 17: Snippet from parallelized version of function compute from prime.c

6.3 Molecular Dynamics

In this example the program, md.c is a program that runs a molecular dynamics simulation. In the simulation, particle motion is based on principled of traditional physical mechanics such as force and

energy. The particles in the simulation are not constrained by any barriers and if they collide, the particles pass through one another. In order to run the simulation, the force and energy of each particle must be calculated. These values are based on the particles' positions and velocities. While there are differential equations the model particle position and velocity over time, this program uses constant incremental discrete time steps to simplify the calculation. In the snippet from Figure 18 below shows the hotspot for parallelization in the compute function.

```
void compute ( int np, int nd, double pos[], double vel[], double mass,
  double f[], double *pot, double *kin ){
  double d;
  double d2;
  int i;
  int j;
  int k;
  double ke;
  double pe;
  double PI2 = 3.141592653589793 / 2.0;
  double rij[3];

  pe = 0.0;
  ke = 0.0;

  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++ ){
```

Start of for loop which computes pe and ke for each particle.

Variable pe is reduced

Variable ke is reduced

```
      ke = ke + vel[i+k*nd] * vel[i+k*nd];
    }
  }

  ke = ke * 0.5 * mass;

  *pot = pe;
  *kin = ke;
  return;
}
```

Figure 18: Snippet that shows hot spot for parallelization in compute function from md.c

The for loop in the compute function has no data dependency because the calculations for one particle independent of from other particles. One should notice the when computing the values of variables, reduction of f, pe, and ke by addition is occurring but otherwise, parallelization is relatively like previous examples. This data storage will need to be called out to IPT. To parallelize this example follow the prompts in Figure 19 below.

```
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
1

Please note that by default, the MPI Environment Initialization functions
will be set in the main function.

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 a pattern from the following list that best characterizes your
parallelization needs:
(Please refer to the user-guide for the explanation on each of the patterns,
and note that not all the listed patterns may be relevant for your
application type.)
1. For-Loop Parallelization
2. Stencil
3. Pipeline
4. Data Distribution and Data Collection
5. Data Distribution
6. Data Collection
7. Point to Point
8. Master-worker pattern- Data distribution - Broadcast
```

```
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.

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

Please specify the type of the data storage (variable, array, structure)
from which the data collection should be done at the hotspot for
parallelization:
1. Variable/s
2. Array/s
3. Structure/s (not supported currently)
4. Both Variable/s and Array/s
5. Both Variable/s and Structure/s (not supported currently)
6. Variable/s, Array/s, Structure/s (not supported currently)
7. None of the above
4

Please select the arrays from which you want to collect data (format 1,2,3,4
etc. with 1 is the first array in the list, 2 is the second array in the
```

```
list etc.)
Possible arrays are:
1. f  type is double []
2. pos  type is double []
3. rij  type is double [3]
4. vel  type is double []
5. pot  type is double *
6. kin  type is double *
1
For array #0
Would you like to collect data from:
1. Distributed stencil array
2. Partially calculated value of an array in a for loop
2
Is this a
1. 1-D array
2. 2-D array
1
Please enter the size of the array: nd*np

Please select the variables 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

Please select the reduce operation to use for variable [pe]
1.  Sum
2.  Product
3.  Min
4.  Max
5.  Logical and
6.  Bit-wise and
7.  Logical or
8.  Bit-wise or
9.  Logical xor
10.  Bit-wise xor
11.  Max value and location
12.  Min value and location
1

Would you like to send the results after reducing the chosen variable to all
processes or to only one?(1. all 0. one).
Note: if option "0" is chosen then only one MPI process will have the
combined results. Please refer to the user-guide for further information on
this.
 1

Please select the reduce operation to use for variable [ke]
1.  Sum
2.  Product
```

Collecting data from an array will result in gather operations

ke and pe are selected for reduction

Reduction with sum operation

```
3.  Min
4.  Max
5.  Logical and
6.  Bit-wise and
7.  Logical or
8.  Bit-wise or
9.  Logical xor
10.  Bit-wise xor
11.  Max value and location
12.  Min value and location
1

Would you like to send the results after reducing the chosen variable to all
processes or to only one?(1. all 0. one).
Note: if option "0" is chosen then only one MPI process will have the
combined results. Please refer to the user-guide for further information on
this.
1
Would you like to do this MPI pattern again?(Y/N)
n

Are you printing anything?(Y/N)
n

Are you writing or reading anything from file(s) ?(Y/N)
n
Do you want to perform any timing (Y/N)?
n
```

Results communicated with all MPI processes

Figure 19: IPT prompts for parallelizing md.c

In Figure 20 below the changes to the function compute are shown. As noted by IPT during parallelization, the calls to MPI_Init( ) and MPI_Finialize( ) are made in the main program. Since different processes split the iterations of outer for loop, the loop control variables have changed. These are calculated in the beginning of the function after the innovation of MPI_Comm_rank( ) and MPI_Comm_size( ).

```
void compute(int np,int nd,double pos[],double vel[],double mass,double
f[],double *pot,double *kin){
  double rose_+ke0;
  double rose_+pe0;
  int rose_size;
  int rose_rank;
  double d;
  double d2;
  int i;
  int j;
  int k;
  double ke;
  double pe;
  double PI2 = 3.141592653589793 / 2.0;
  double rij[3];
  pe = 0.0;
  MPI_Comm_size(MPI_COMM_WORLD,&rose_size);
```

```
  MPI_Comm_rank(MPI_COMM_WORLD,&rose_rank);
  ke = 0.0;
  int rose_upper_limit0;
  int rose_lower_limit0;
  int rose_range0;
rose_range0 = (np - 0 )/ rose_size;
double rose_range_double0 = (double) (np - 0 )/ rose_size;
if (rose_range_double0 <= 1 ) {
rose_upper_limit0 = ((rose_rank+1)*1 + 0 <= np ) ? (rose_rank+1)*1 + 0 : np;
rose_lower_limit0= rose_rank*1 + 0;
}
else {
if ( rose_rank  > 0 ) {
rose_lower_limit0 = (rose_rank-1)*rose_range0 + rose_range0 - (((rose_rank-
1)*rose_range0 + rose_range0 )% 1) + 0 ;
 }
else {
rose_lower_limit0 = 0;
}
if (rose_rank < rose_size -1)  {
rose_upper_limit0 =  (rose_rank)*rose_range0 + rose_range0 -
(((rose_rank)*rose_range0 +  rose_range0) % 1) + 0 - 0;
}
else {
rose_upper_limit0 = np ;
}
}
double * rose_temp_f;
int f_displacement [rose_size];
int f_recvcounts [rose_size];
int f_array_indexes [nd*np];
int f_array_length = 0;
  if (rose_rank > 0) {
    pe = 0;
  }
  if (rose_rank > 0) {
    ke = 0;
  }
  for (k = rose_lower_limit0; k < rose_upper_limit0; k++) {
/*
  Compute the potential energy and forces.
*/
for (i = 0; i < nd; i++) {
  f[i + k * nd] = 0.0;
  int f_index_appendable = 1;
 for (int rose_f_index = 0;rose_f_index <f_array_length;rose_f_index++ ) {
   if (f_array_indexes[rose_f_index] == (0 + i + k * nd*1)){
   f_index_appendable = 0;
   }
 }
 if (f_index_appendable == 1) {
   f_array_indexes[f_array_length] = 0 + i
+ k * nd*1;
   f_array_length++;
   }
 }
    for (j = 0; j < np; j++) {
```

Iteration distribution and calculating for loop boundaries

Code for calculating the number of elements to be gathered from each MPI process and the displacement of the elements in the global/result array.

```
      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;
          int f_index_appendable = 1;
for (int rose_f_index = 0;rose_f_index <f_array_length;rose_f_index++ ) {
  if (f_array_indexes[rose_f_index] == (0 + i + k * nd*1)){
    f_index_appendable = 0;
  }
}
if (f_index_appendable == 1) {
  f_array_indexes[f_array_length] = 0 + i + k * nd*1;
  f_array_length++;
}
        }
      }
    }
/*
  Compute the kinetic energy.
*/
    for (i = 0; i < nd; i++) {
      ke = ke + vel[i + k * nd] * vel[i + k * nd];
    }
  }
  MPI_Allreduce(&ke,&rose_+ke0,1,MPI_
DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  ke = rose_+ke0;
  MPI_Allreduce(&pe,&rose_+pe0,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  pe = rose_+pe0;
  rose_temp_f = (double *) malloc (sizeof(double)*f_array_length);
for (int rose_index =0; rose_index < (f_array_length ); rose_index++) {
  rose_temp_f[rose_index] = f[f_array_indexes [rose_index]];
}
MPI_Allgather(&f_array_length,1,MPI_INT,f_recvcounts,1,MPI_INT,MPI_COMM_WORD
  );
int current_f_dis = 0;
for (int rose_dis_index =0; rose_dis_index < rose_size; rose_dis_index++) {
  f_displacement[rose_dis_index] = current_f_dis;
  current_f_dis += f_recvcounts[rose_dis_index];
}
MPI_Allgatherv(rose_temp_f,f_recvcounts[rose_rank],MPI_DOUBLE,f,f_recvcount,
  f_displacement,MPI_DOUBLE,MPI_COMM_WORLD);
  ke = ke * 0.5 * mass;
   *pot = pe;
   *kin = ke;
  return ;
}
```

Calls to MPI_Allgatherv and MPI_Allreduce

Figure 20: Snippet from parallelized version of function compute from md.c

6.4 Circuits Satiability

The program circuits.c is an a determines the number of solutions to the circuit satisfy problem. To find the solutions and exhaustive search is used. There is no known polynomial-time algorithm to solve the general case because the solution is nondeterministic polynomial (NP). While using an exhaustive search can be time consuming to run in series, this issue can be solved by parallelizing the program. There are no data dependencies between loop iterations because each circuit evaluation is independent. The hot spot for parallelization is in the main function of the program in shown in Figure 21 below

```
int main ( int argc, char *argv[] ) {
# define N 23

  int bvec[N];
  int i;
  int ihi;
  int j;
  int n = N;
  int solution_num;
  int value;

  printf ( "\n" );
  timestamp ( );
  printf ( "\n" );
  printf ( "SATISFY\n" );
  printf ( "  C version\n" );
  printf ( "  We have a logical function of N logical arguments.\n" );
  printf ( "  We do an exhaustive search of all 2^N possibilities,\n" );
  printf ( "  seeking those inputs that make the function TRUE.\n" );
/*
  Compute the number of binary vectors to check.
*/
  ihi = 1;
  for ( i = 1; i <= n; i++ ){
    ihi = ihi * 2;
  }
  for ( i = 1; i <= n/2; i++ ){
    printf("\ntest\n");
  }
  printf ( "\n" );
  printf ( "  The number of logical variables is N = %d\n", n );
  printf ( "  The number of input vectors to check is %d\n", ihi );
  printf ( "\n" );
  printf ( "   #       Index    ---------Input Values----------------------
-\n" );
  printf ( "\n" );
/*
  Check every possible input vector.
*/
  solution_num = 0;
  for ( i = 0; i < ihi; i++ ){
```

This loop is the hot spot for parallelization. The variable solution_num is being reduced.

```
    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" );
    }
  }
  printf ( "\n" );
  printf ( "  Number of solutions found was %d\n", solution_num );
/*
  Shut down.
*/
  printf ( "\n" );
  printf ( "SATISFY\n" );
  printf ( "  Normal end of execution.\n" );
  printf ( "\n" );
  timestamp ( );
  return 0;
# undef N
}
```

Figure 21: Snippet from circuits.c the shows region that to parallelize

Notice variable solution_num is being reduced, this will be important when answering IPT's prompts. While this example is very similar to previous ones, there are two regions for parallelization. Follow IPT's prompts shown in Figure 22 below for guidance.

```
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
1

Please note that by default, the MPI Environment Initialization functions
will be set in the main function.

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 a pattern from the following list that best characterizes your
parallelization needs:
(Please refer to the user-guide for the explanation on each of the patterns,
and note that not all the listed patterns may be relevant for your
application type.)
1. For-Loop Parallelization
```

```
2. Stencil
3. Pipeline
4. Data Distribution and Data Collection
5. Data Distribution
6. Data Collection
7. Point to Point
8. MPI environment initialization and finalization
9. Master-worker pattern - Data distribution - Broadcast
1

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.

for (i = 1; i <= n / 2; i++) {
  printf("\ntest\n");
}
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");
  }
}
Is this the for loop you are looking for?(y/n)
y

Please specify the type of the data storage (variable, array, structure)
from which the data collection should be done at the hotspot for
parallelization:
1. Variable/s
2. Array/s
3. Structure/s (not supported currently)
4. Both Variable/s and Array/s
5. Both Variable/s and Structure/s (not supported currently)
6. Variable/s, Array/s, Structure/s (not supported currently)
7. None of the above
1
```

Parallelize this loop, which has reduction

Data storage happening in variable solution_num

```
Please select the variables 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

Please select the reduce operation to use for variable [solution_num]
1.  Sum
2.  Product
3.  Min
4.  Max
5.  Logical and
6.  Bit-wise and
7.  Logical or
8.  Bit-wise or
9.  Logical xor
10.  Bit-wise xor
11.  Max value and location
12.  Min value and location
1

Would you like to send the results after reducing the chosen variable to all
processes or to only one?(1. all 0. one).
Note: if option "0" is chosen then only one MPI process will have the
combined results. Please refer to the user-guide for further information on
this.
1

Would you like to do this MPI pattern again?(Y/N)
n

Are you printing anything?(Y/N)
n

Are you writing or reading anything from file(s) ?(Y/N)
n

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

Printing truncated from this example

Figure 21: Reponses to IPT prompts for circuits.c

The parallelized version of main by IPT is show below in Figure 22. One should expect to see a few key changes. First there should be an call of MPI_Init( ) as well as MPI_Comm_size and MPI_Comm_rank. These three functions appear at the beginning of almost all MPI programs, and MPI_Init must be called in order to initialize the MPI environment. Since the pattern of for-loop parallelization was chosen, there should be calculations to distribute the loop iterations among the MPI processes. There are changes to the upper and lower bounds on the for loop since each process will be distributed a chunk of iterations to run. Also, since there was reduction in the loop, IPT should have added in a call to MPI_Allreduce( ) to

calculate the global value of solution_num from the each processes' local value. Lastly, there is the invocation of MPI_Finialize( ) terminal the session and end communication between the processes.

```
int main(int argc,char *argv[]){
  int rose_solution_num0;
  int rose_size;
  int rose_rank;
# define N 23
  int bvec[23];
  int i;
  int ihi;
  int j;
  int n = 23;
  int solution_num;
  int value;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&rose_size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rose_rank);
  printf("\n");
  timestamp();
  printf("\n");
  printf("SATISFY\n");
  printf("  C version\n");
  printf("  We have a logical function of N logical arguments.\n");
  printf("  We do an exhaustive search of all 2^N possibilities,\n");
  printf("  seeking those inputs that make the function TRUE.\n");
/*
  Compute the number of binary vectors to check.
*/
  ihi = 1;
  for (i = 1; i <= n; i++) {
    ihi = ihi * 2;
  }
  for (i = 1; i <= n / 2; i++) {
    printf("\ntest\n");
  }
  printf("\n");
  printf("  The number of logical variables is N = %d\n",n);
  printf("  The number of input vectors to check is %d\n",ihi);
  printf("\n");
  printf("   #       Index    ---------Input Values-----------------------
\n");
  printf("\n");
/*
  Check every possible input vector.
*/
  solution_num = 0;
  int rose_upper_limit0;
  int rose_lower_limit0;
  int rose_range0;
rose_range0 = (ihi - 0 )/ rose_size;
double rose_range_double0 = (double) (ihi                                  - 0
)/ rose_size;
if (rose_range_double0 <= 1 ) {
rose_upper_limit0 = ((rose_rank+1)*1 + 0                                   <=
ihi ) ? (rose_rank+1)*1 + 0 : ihi;
```

IPT added variable rose_solution_num to store the local value of solution_num for each MPI process.

Initializing MPI environment. These functions are in almost all MPI programs.

Distributing iterations of for loop to MPI processes and calculating new loop controls

```
rose_lower_limit0= rose_rank*1 + 0;
}
else {
if ( rose_rank  > 0 ) {
rose_lower_limit0 = (rose_rank-1)*rose_range0 + rose_range0 - (((rose_rank-
1)*rose_range0 + rose_range0 )% 1) + 0 ;
 }
else {
rose_lower_limit0 = 0;
}
if (rose_rank < rose_size -1)  {
rose_upper_limit0 =  (rose_rank)*rose_range0 + rose_range0 -
(((rose_rank)*rose_range0 +  rose_range0) % 1) + 0 - 0;
}
else {
rose_upper_limit0 = ihi ;
}
}
  if (rose_rank > 0) {
    solution_num = 0;
  }
  for (i = rose_lower_limit0; i < rose_upper_limit0; 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");
    }
  }
  MPI_All_reduce(&solution_num,&rose_solution_num0,1,MPI_INT,MPI_SUM,MPI_COMM
_WORLD);
  solution_num = rose_solution_num0;
//added this for testing. If the already parallelized loop is not the last
one, then there is no error, else segementation fault
  printf("\n");
  printf("  Number of solutions found was %d\n",solution_num);
/*
  Shut down.
*/
  printf("\n");
  printf("SATISFY\n");
  printf("  Normal end of execution.\n");
  printf("\n");
  timestamp();
  MPI_Finalize();
  return 0;
# undef N
}
```

MPI_reduce was called. The variable solution_num is the send_data parameter. The variable rose_solution_num0 is the receive data variable. Global value of solution_num. temporary stored in

Terminate MPI environment

Figure 22: Parallelized version of the function main from circuits.c

7. References

Search Example code. Web accessed Oct 1, 2019.
https://people.sc.fsu.edu/~jburkardt/c_src/search/search.c

Prime Number Example code. Web accessed Oct 1, 2019.
https://people.sc.fsu.edu/~jburkardt/c_src/prime_mpi/prime_mpi.html

Molecular Dynamics (MD) code. Web accessed May 10, 2017.
https://people.sc.fsu.edu/~jburkardt/c_src/md/md.html

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