Interactive Parallelization Tool – Version 1.0

Please Note: This excerpt of the IPT documentation is meant for giving a quick tool demo

1. Overview of using the IPT – Command Line Interface

This section contains the relevant information for guiding the end-user through the process of parallelization using IPT. The main topics that are covered include user supplied input, MPI, OpenMP, and CUDA parallelization.

1.1. User Supplied Input

The end-users interactively supply the input to IPT through the keyboard. Wherever IPT needs specific line-numbers, the end-user is prompted to select the numbers from the file called `numberedCode.C`. This file is generated by IPT during the parallelization process and helps in getting the hooks for parallelization (information on where to parallelize) from the end-users.

Please note that before starting the parallelization process, you should be able to compile your serial program using the GCC compiler. If the serial program is not free from compile-time or link-time errors while using the GCC compiler, IPT will not be able to parallelize it correctly. Also, if you are parallelizing functions whose definitions are in the header files or other included files/subclasses, please copy them into the main file that is specified as input to IPT – this step is required for only the current version of IPT. Though IPT automatically imports the related (or included) files to create an AST, it cannot edit those header files or included files and save the changes.

1.2. Dependency Analysis

Once the serial program is provided as input to IPT and a parallelization pattern is selected, IPT will do a variable dependency analysis. This will depend upon the type of parallel programming paradigm selected by the end-user, but in essence, IPT automatically determines, the relevant variables or multi-variables that feature in the section of code to be parallelized and prompts the end-user for some clarifications. Please note that for the OpenMP + Offload and CUDA modes of parallelization, IPT will prompt the end-users to specify whether each variable is input, or output or both or neither.

1.3. Analysis – Warning messages

In certain scenarios, IPT will display warning messages for the end-users. Messages about the scopes not being the same can be displayed at any stage during the parallelization process. Such warnings can be ignored as they are inherently handled by IPT. Also after the end-user has finished supplying input, and

the parallelization process is complete, a list of errors will appear on the screen when IPT runs the consistency tests. These errors can be ignored and indicate that no such methods as those inserted in the program can be found. This happens primarily because the IPT uses the GCC compiler for compiling the generated parallel program. Once the parallelization process is finished, the end-user should compile with the supported compiler for the paradigm.

```
Warning: SageBuilder::buildNondefiningFunctionDeclaration_T(): scope parameter m
ay not be the same as the topScopeStack() (e.g. for member functions)
```

Figure 1: Snippet of Warning Messages

<OTHER SECTIONS TO BE COPIED FROM THE MAIN DOCUMENT AND INSERTED HERE>

2. Running the IPT

To run IPT, copy the serial file in the same directory as IPT. Make sure the serial program has no errors. Then type the following command into the console:

```
./IPT <input filename here>
```

3. Test Case 1: Circuit Satisfiability

Circuit Satisfiability is a problem that determines whether a given Boolean circuit has an assignment of its inputs that makes the output true. In the code snippet shown in Figure 41, we can see a for-loop at line # 2 that is an ideal candidate for parallelization. In the following subsections we will be parallelizing this for-loop at line # 2 using MPI, OpenMP, and CUDA.

```
1.        t1 = gettime();
2.        for ( i = ilo; i < ihi; i++ ){
3.          i4_to_bvec ( i, n, bvec );
4.          value = circuit_value ( n, bvec );
5.          if ( value == 1 ) {
6.            solution_num = solution_num + 1;
7.            printf ( " %2d %1ld: ", solution_num, i );
8.            for ( j = 0; j < n; j++ ){
9.              printf ( " %1ld", bvec[j] );
10.           }
11.           printf ( "\n" );
12.         }
13.       }
14.       t2 = gettime();
```

Figure 41: Snippet of the Serial Code – Circuit Satisfiability Problem

3.1. MPI Mode

For the circuit satisfiability example, we will parallelize the for-loop in which we call the compute function. After the computations are finished, we will need to add the results from the individual processes into the variable `solution_num`. To parallelize with the IPT, we must know the location of the for-loop, the variable to be reduced, the type of reduction and the preferred reduction operation. Since we are adding the values of the individual processes into `solution_num`, our operation will be sum. The complete parallelization process is depicted below in Figure 42. At the end of the parallelization process, a file named "rose_circuit_serial_OpenMP.cpp" will be generated by the tool that contains the MPI version of the test case.

```
Which of the following patterns would you like.

   1. Single For-Loop Parallelization
   2. Nested For-Loop Parallelization
   3. Stencil
   4. Pipeline
   5. Data Distribution and Data Collection
   6. Data Distribution
   7. Data Collection.

1

Please enter the name of the function that you wish to parallelize.

main

for(i = 1;i <= n;i++) {ihi =(ihi * 2);}

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

n

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

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

y


Would you like to do the data collection of a

   1. Variable
   2. Array.
```

```
1



Please enter the number of variables that you would like to perform reduction
operations on. If there are no variables to reduce please enter 0.

1

Please select a variable to perform the reduce operation on. List of possible
variables are:

n type is int

bvec type is long [30UL]

solution_num type is int

i type is long

solution_num



Please select the reduce operation to use for variable

      1. Sum
      2. Product
      3. Min
      4. Max

1

Would you like to send the results after reducing the chosen variable to all
processes or to only one?(1. all 0. one) 1

Warning:      SageBuilder::buildNondefiningFunctionDeclaration_T():      scope
parameter may not be the same as the topScopeStack() (e.g. for member
functions)

Variable Declarations complete.

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

n

Are you writing anything?(Y/N)

n

Running Consistency Tests
```

Figure 42: Snippet of the Parallelization Process (MPI) – Circuit Satisfiability

As shown in the code snippet in Figure 43, we have parallelized the desired for-loop. Note in lines 2-6, the values for the upper and lower limit of the for-loop are automatically calculated and inserted into the serial program. Also, notice lines 19 and 20 which contain the reduce operation. For this operation, the IPT creates a temporary variable which it uses to store results in the individual calculations. These results are accumulated in line 19 and copied over to the actual variable in line 20. Additional lines of code that are standard to developing an MPI program are automatically added to the generated code that can be compiled using the available MPI compiler.

```
1.  t1 = gettime();
2.  rose_lower_limit0 = rose_rank * ((ihi - ilo) / rose_size) + ilo;
3.  if (rose_rank == rose_size - 1)
4.   rose_upper_limit0 = rose_lower_limit0 + …;
5.  else
6.   rose_upper_limit0 = rose_lower_limit0 + (ihi - ilo) / rose_size;
7.  for (i = rose_lower_limit0; i < rose_upper_limit0; i++) {
8.       i4_to_bvec(i,n,bvec);
9.       value = circuit_value(n,bvec);
10.      if (value == 1) {
11.        solution_num = (solution_num + 1);
12.        printf(" %2d %1ld: ",solution_num,i);
13.        for (j = 0; j < n; j++) {
14.           printf(" %1ld",bvec[j]);
15.        }
16.       printf("\n");
17.     }
18.    }
19.    MPI_Allreduce(&solution_num,&rose_solution_num0,1,…);
20.    solution_num = rose_solution_num0;
21.    t2 = gettime();
```

Figure 43: Snippet of the Parallelized Code (MPI) – Circuit Satisfiability

Notice that the variables rose_lower_limit and rose_upper_limit were created to calculate the bounds of the for-loop. The IPT has also created the MPI variables and inserted function calls as specified by the end-user. Keep in mind that the placement of these functions was based on the end-user input. As such, it is imperative that the end-user has a basic understanding of their own code and the basic concepts involved in parallelization.

3.2. OpenMP Mode

For parallelizing the same loop in OpenMP without offload mode, we will perform the same operations as we did for MPI. As seen from the parallelization process in Figure 44, we need to only specify the location of the for-loop and the variable to reduce. Note the IPT will give the user feedback by printing the results of the variable dependency analysis. The complete process is depicted below.

```
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(i = 1;i <= n;i++) {ihi =(ihi * 2);}

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

n

for(i     =     ilo;i     <     ihi;i++)     {i4_to_bvec(i,n,bvec);value     =
circuit_value(n,bvec);if(value    ==    1)    {solution_num    =(solution_num    +
1);printf("  %2d  %11d:  ",solution_num,i);for(j  =  0;j  <  n;j++)  {printf("
%lld",bvec[j]);}printf("\n");}}

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

y

Adding i to private variables.

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

1
```

```
Please select a variable to perform the reduce operation on. List of possible
variables are:

ihi type is long
n type is int
solution_num type is int
t1 type is double
ilo type is long
bvec type is long [30UL]
value type is int
t2 type is double

solution_num

…<some messages displayed by IPT>…

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

Running Consistency Tests
```

Figure 44: Snippet of the Parallelization Process (OpenMP) – Circuit Satisfiability

From the parallelized code snippet in Figure 45, we can see there are some lines of code inserted by IPT. The first is the `pragma omp parallel` in line 1 which contains our variable analysis and tells the compiler to run the following code in parallel. The second in the `pragma omp for reduction`. This tells the compiler that there is a for-loop being run in parallel. The reduction of the variable also appears in the same pragma. Notice however that the variable being reduced is a temp variable. IPT creates the temp variable and uses it to store values of the calculations. This is then copied back by the assignment statement in line 20.

```
1. #pragma omp parallel default(none)
   shared(solution_num,temp_solution_num,ihi,ilo,n,bvec)
   private(i,value,j)
2. {
3. #pragma omp for reduction ( +: temp_solution_num )
4.   for (i = ilo; i < ihi; i++) {
5.       {
6.        i4_to_bvec(i,n,bvec);
7.        value = circuit_value(n,bvec);
```

```
8.          if (value == 1) {
9.             solution_num = (solution_num + 1);
10.            printf(" %2d %11d: ",solution_num,i);
11.            for (j = 0; j < n; j++) {
12.              printf(" %11d",bvec[j]);
13.            }
14.            printf("\n");
15.          }
16.       }
17.       temp_solution_num = solution_num;
18.       }
19.       }
20.       solution_num = temp_solution_num;
```

Figure 45: Snippet of the Parallelized Code (OpenMP) – Circuit Satisfiability

3.3. CUDA Mode

For parallelizing the loop at line # 2 of Figure 41 with CUDA, the process of providing specifications is shown in Figure 46. As can be seen from the parallelization process in Figure 46, we need to only specify the location of the for-loop and the variable to reduce. Note the IPT will give the user feedback by printing the results of the variable dependency analysis. The complete process is depicted below.

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

 3

Please enter the function in which you wish to insert the kernel call (or
parallelize the for-loop).

main

Would you like

1. Single for-loop parallelization involving arrays
2. Nested for-loop Parallelization involving arrays
3. Single for-loop involving a variable and reduction operation
4. Nested for-loop involving variables and multiple reduction operations
3

<…other prompts>
```

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

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

y

Please enter the name of the variable to reduce.

solution_num

Please enter the number of iterations of the for-loop.

ihi

Running Consistency Tests
```

Figure 46: Snippet of the parallelization process (CUDA) – Circuit Satisfiability

A snippet of the generated CUDA code is shown in Figure 47. IPT inserts code for declaring device variables, allocating memory space on GPU, copying from host to device, launching the CUDA kernel, copying back the data from device to host, inserting device function calls and adding code for the device functions. Comments are inserted at appropriate places so that the user can understand the purpose of the code inserted or changed during the parallelization process. There are some default values in the generated program, *e.g.*, the number of GPU threads and blocks to use for the computation. These can be changed by the end-user to any desired value.

```
wtime1 = gettime();

//Starting Parallelization

//Inserting code for memory allocation, grid-size & block-size

host_solution_num= (int*)malloc((1)*(ihi)*sizeof(int));

cudaMalloc((void **) &device_solution_num,( 1)*(ihi)*sizeof(int));

dim3 dimGrid(ihi/1024,1);

dim3 dimBlock(1024,1);

//Inserting Kernel Call
```

```
kernel0<<<dimGrid,dimBlock>>>(device_solution_num,ihi,n,value,j,1,ihi);

/*
  for (col = 0; col < ihi; i++) {
    i4_to_bvec(::i,n,bvec);
    value = circuit_value(n,bvec);
    if (value == 1) {
      solution_num[i] = (solution_num[i] + 1);
      printf ( "  %2d  %10d:  ", solution_num, i );
      for (j = 0; j < n; j++) {
        printf ( " %d", bvec[j] );
      }
      printf ( "\n" );
    }
  }
*/
//Copying from Device to Host

cudaMemcpy(host_solution_num,device_solution_num,(ihi)*sizeof(int),
cudaMemcpyDeviceToHost);

//Code for reduction operation

 for(long row = 0; row < 1; ++row){for(long col = 0; col < ihi; ++col)    {
total_solution_num+= host_solution_num[row*ihi+ col];    }  }

solution_num= total_solution_num;

//  Ending Parallelization
```

Figure 47: Snippet of the generated parallel code (CUDA) – Circuit Satisfiability