A Tool for Interactive Parallelization

Ritu Arora  
Texas Advanced Computing Center  
J.J. Pickle Research Campus  
Austin, Texas 78758  
rauta@tacc.utexas.edu

Julio Olaya  
Department of Computer Science  
University of Texas at El Paso  
El Paso, TX 79968  
jolaya@miners.utep.edu

Madhav Gupta  
Texas Advanced Computing Center  
J.J. Pickle Research Campus  
Austin, Texas 78758  
madhavg@utexas.edu

ABSTRACT

“Tell me, I’ll forget,  
Show me, I may remember,  
Involve me, I’ll understand.” - Confucius

The proverb above describes the essence of this paper and the motivation behind the development of our Interactive Parallelization Tool (IPT) that can transform serial applications into multiple parallel variants. The end-users of the IPT are required to develop an understanding about the basic concepts involved in parallel programming (viz., concurrency, data distribution and data gathering). After developing an understanding of the basic parallel programming concepts, the IPT can be used by its target audience (domain-experts and students) to semi-automatically generate parallel programs based on multiple parallel programming paradigms (MPI, OpenMP, and CUDA) and learn about these paradigms through observation and comparison. This IPT-based personalized learning approach complements the traditional methods of learning and training that usually emphasize the syntax and semantics of one or more programming standards. The main benefit of IPT is that it provides a jumpstart to the domain-experts in using modern HPC platforms for their research and development needs, and hence in lowering the adoption barriers to HPC.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming - distributed programming, parallel programming.

General Terms

Algorithms, Performance, Design

Keywords

User-guided parallelization; program transformation; hybrid programming; parallel programming; personalized learning

1. INTRODUCTION

Even though High Performance Computing (HPC) is gradually becoming indispensable for research and business growth, the programming challenges associated with the HPC application development are a key bottleneck to embracing it on a massive scale. It is often difficult for domain-experts (e.g., agronomists and data archivists) and students to develop expertise in one or more of the parallel programming paradigms and to adapt existing applications to take advantage of modern HPC platforms.

Empirical studies and reports show that (1) if given a choice, the domain-experts would prefer to focus on the scientific aspects of their research domain instead of honing their parallel programming skills; (2) the domain-experts find it difficult to adopt implicitly parallel programming languages or high-level interfaces that promise parallelization at the cost of forsaking the investment made in their existing applications; (3) investing time and effort in developing parallel applications can be risky, especially with hybrid programming techniques, as there is no guarantee that the parallel applications will perform as expected [1, 2, 3]. Such studies and reports demonstrate that there is a need to bridge the gap between the HPC experts who can develop efficient applications to utilize the HPC platforms and the domain-experts who actually have the computational problems that could use such HPC platforms. There is a need for a demonstration- or guidance-based approach that can help the domain-experts in transforming their serial applications into multiple parallel variants using explicit parallelization (hence reusing the years of investment in developing serial applications) without having to learn low-level details of parallel programming paradigms. Such an approach will help the domain-experts in developing and choosing the best performing parallel variant without being burdened with long application-development cycles and manual reengineering activity. The resulting parallel programs should be easy to read and reason about so that the domain-experts can understand the outcome of their design decisions and make changes as needed.

The reports from the HPC Initiative at the Council on Competitiveness [4] indicate a clear need for HPC workforce development. The data collected during the parallel programming contest conducted at the Texas Advanced Computing Center (TACC) [5] shows that less than 10% of the contestants, who were graduate students and postdocs from all over the nation, understood all the three popular programming paradigms – MPI [6], OpenMP [7] and CUDA [8] – and were able to develop accurate applications using them. Hence, software tools that can facilitate the teaching, learning, and the usage of parallel programming paradigms through comparison and demonstration can be of immense value towards this end. The educators can use such tools for classroom teaching to dynamically demonstrate the differences amongst the different parallel programming approaches and the impact of certain programming strategies (viz., choice of data distribution scheme, load-balancing, and synchronization) on the performance of the parallel programs.

There have been several research projects in the past, with some dating back to the early 90s, that had the goal of making parallel programming easy for domain-experts – e.g., CAPTools [9]. Investments are being made in high-productivity programming languages, tools and paradigms, but currently, there is no single high-level, open-source, and mature domain-independent solution for parallelizing sequential applications or adapting them for multiple target platforms. There is a clear need for a tool that offers a low-risk way for scientists/engineers/students to try HPC. In order to address the aforementioned needs, we are developing an Interactive Parallelization Tool (IPT) whose core functionality is to semi-automatically parallelize existing sequential applications for multiple hardware platforms.
The IPT has a subset of functionality that is available in our framework for parallelization known as FraSPA [10, 11], and is adapted for light-weight usage through a wizard-based GUI and a command-line interface. Latest additions to FraSPA include support for parallel programming through a Graphical User Interface (GUI), and the support for generating CUDA programs. The previously available command-line interface has also been improved so that it can support the parallelization of additional classes of applications – e.g., stencil-based applications – and MPI code generation. The recent extensions made to FraSPA, along with some of its other features, have been packaged into IPT, and will be released to public under the new BSD license.

In this paper, we provide an overview of IPT, explain the usage of its GUI and command-line interface for generating parallel programs, present test cases for generating CUDA, OpenMP and MPI programs through the GUI and the command-line interface, show the benefits of the IPT in learning parallel programming through demonstration and comparison, and show that the generated code is comparable to the hand-written code.

2. OVERVIEW OF THE TOOL

The IPT uses generative programming techniques [12], design patterns [13], and end-user input to semi-automatically translate serial C/C++ applications into parallel applications for systems with both shared- and distributed-memory architectures. Support for parallelizing Fortran applications will be added in the next phase of development. The high-level overview of different components of IPT is shown in Figure 1. The knowledge of expert parallel programmers is encapsulated in the form of the rules for code generation (that are built into the parser) and the design templates. The design templates in IPT are abstractions of the structural and behavioral aspects of various parallel programming patterns (e.g., templates for handling data movement and data partitioning). Currently, IPT supports a limited set of patterns for parallelization – e.g., pipeline and stencil-based computation [13].

The end-users of IPT provide the specifications for parallelization – including what to parallelize and where – either through a command-line interface or a GUI. The end-users’ specifications are parsed, and along with the design templates, are passed to the translator program. The translator invokes the ROSE compiler [14] to perform serial to parallel transformation. The ROSE compiler, as used in this research, helps in traversing the Abstract Syntax Tree (AST) of the serial (or source) code and in modifying it to generate a parallel version (or target code). The AST modifications required for parallelizing a serial code include code deletion (viz., deleting intermediate code artifacts), code insertion (viz., inserting the #include statements), and code update (viz., changing the start and stop conditions of for-loops). The ROSE compiler supports the transformations of C, C++ and Fortran programs. The parallel programming paradigms that are currently supported by the IPT are MPI, OpenMP, CUDA and hybrid programming.

For the purpose of data-provenance, a log of the specifications for parallelization that are provided by the end-users is maintained. In future, we will extend IPT so that it can accept the edited log files as input for parallel code generation.

3. USE CASE SCENARIO

In this section, we provide a walk-through of parallelizing an existing serial program through the GUI-based and command-line interfaces of IPT. Consider a simple C program for doing matrix multiplication that needs to be parallelized using IPT. The desired output in this use case is a C + CUDA program that can run on GPUs. In Section 3.1, we explain the parallelization of this program using the GUI of IPT.

![Figure 1. Overview of the framework for generating parallel code](image)

**Figure 1.** Overview of the framework for generating parallel code

### 3.1 GUI-based Parallelization

As a first step, the user will click on the “File” menu in the GUI and select “Open Source”. This step lets the user browse the desired file and open it in the left-hand side panel of the GUI. As a second step, the user will select the desired options for the following three parameters:

- **Programming Paradigm:** CUDA or MPI or OpenMP
- **Programming Language:** C or C++ or Fortran
- **Patterns:** Outer-for-loop, nested for-loops, etc.

The options that are in bold in the list given above are the ones that are applicable to the use case presented in this section. As a third step, the user will highlight the for-loop to parallelize. As a fourth step, the user will click on “Select” from the “Selection” menu to register their selection of the for-loop. The steps three and four are shown through the screen-shot in Figure 2.

As a fifth step, the user is required to select the device parameters (viz. input and output variables) from the list presented by IPT. As the sixth step, the user will click the “Run” button to generate the target (or parallel) program. As a seventh step, the user can review the generated program and click on “Save” button in the right-hand side panel of the GUI. This will cause the file with the generated program to be saved in the same directory as the serial code. The file is named as per the default file-naming convention of IPT. For the CUDA programs that are generated and saved through IPT, the file-name is made up of a prefix which is the name of the serial code (the source program), a suffix “.par” and extension “.cu”: <source-program>_<version>. The user has the option to edit both the source and target programs through the GUI if they wish to do so. A snippet of the generated code is shown in Figure 3 and a screen-shot associated with the sixth and seventh steps is shown in Figure 4.
Highlight the for-loop to parallelize and register this selection.

1. void kernel01(float *a)
2. __global__ void kernel(int * a, ...)
3. {
4.   int j, k, sum, i;
5.   i = blockIdx.x * blockDim.x + threadIdx.x;
6.   j = blockIdx.y * blockDim.y + threadIdx.y;
7.   if(i < row){
8.     if(j < col){
9.       for (k = 0; k < row; k++) {
10.          indexA = ((i * row) + k);
11.          indexB = ((k * col) + j);
12.          sum += (a[indexA] * b[indexB]);
13.        }
14.   }
15.   }
16.   indexC = ((i * col) + j);
17.   c[indexC] = sum;
18.   sum = 0;
19. }

13. //other code
14. /***** Starting Parallelization *****/
15. //declare device variables
16. float elapsedTime; cudaEvent_t start, stop;
17. cudaEventCreate(&start);
18. cudaEventCreate(&stop);
19. int * device_a; int * device_b; int * device_c;
20. //Allocate memory space in the GPU
21. cudaMalloc((void **) &device_a, sizeof(a));
22. cudaMalloc((void **) &device_b, sizeof(b));
23. cudaMalloc((void **) &device_c, sizeof(c));
24. //Copy from host to device
25. cudaMemcpy(device_a, a, ...);
26. cudaMemcpy(device_b, b, ...);
27. //launch kernel function
28. dim3 numThreads(32, 32);
29. dim3 blocks((row+31)/32, (col+31)/32);
30. cudaEventRecord(start, 0);
31. kernel01<<<blocks, numThreads>>>(device_a, ...);
32. cudaEventRecord(stop, 0);
33. cudaEventSynchronize(stop);
34. cudaEventElapsedTime(&elapsedTime, start, stop);
35. printf("the elapsed time is %f\n", elapsedTime);
36. //copy back from device to host
37. cudaMemcpy(c, device_c, sizeof(c), cudaMemcpyDeviceToHost);
38. cudaFree(device_a);
39. cudaFree(device_b);
40. cudaFree(device_c);
41. //other code
42. /***** Ending Parallelization *****/
43. //other code

Figure 2. Highlight the for-loop to parallelize and register this selection.

Figure 3. Snippet of generated CUDA code

The generated program contains all the necessary statements for the kernel code, the code to initialize the accelerator, management of data or program transfers between the host and accelerator, and initiation of accelerator shutdown. The comments are inserted by IPT 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. For example, the number of GPU threads is 32 by default but can be changed by the end-user to any desired value. In the next version of IPT, we will prompt the user for different numbers of threads, block size etc. and will incorporate optimization options like the usage of asynchronous calls and double buffering. The process of generating parallel code with OpenMP directives and with both OpenMP + Offload directives is very similar to the use case presented in this section. However, the user needs to provide additional information as prompted by IPT.
3.2 Command-Line Option for Parallelization

In order to generate a CUDA version of a given C/C++ program through IPT, the end-user can either provide the specifications for parallelization interactively, or through an input text file. Let us assume that the end-user wants to provide the specifications through an input file. As shown in Figure 5, the end-user fills in the desired specifications for parallelization in a given template – they specify the programming model, programming language, the parallel programming pattern, and the details about the parallel operation desired on their preferred region of code. After writing the input text file, the user needs to invoke the translator program by providing the name of the input source file as an argument. The internal mechanism for generating parallel code via GUI or command-line interface is the same. Hence, the generated code via the two methods is also the same. The process of using the command-line interface interactively to provide the specifications for parallelization is shown in Section 5.

4. EVALUATION METRICS

All the development and testing related to IPT is being done on the Stampede supercomputer at TACC. Using various test cases, IPT was evaluated as per the following criteria:

- The number of LoC in IPT that are reused for generating applications across different domains.

The command-line interface and GUI-based interface are being evaluated for usability on the basis of the end-user feedback.

5. TEST CASES AND RESULTS

Three test-cases are discussed in this section – Circuit Satisfiability problem [15], Seismic Tomography code [16, 17], and Poisson Solver [18, 19] – to demonstrate the versatility of IPT. The Circuit Satisfiability problem is used to demonstrate the generation of OpenMP + Offload code for the coprocessors, the Seismic Tomography code is used to demonstrate generation of CUDA code for GPUs, and the Poisson Solver is used to show the generation of MPI code.
Circuit Satisfiability is an embarrassingly parallel application that simulates the actual circuit and determines whether a combination of inputs to the circuit of logical gates produces an output of 1. The application involves an exhaustive search of all the possible combinations of the specified number of bits in the input. A snippet of the serial code for this application is shown in Figure 6. The for-loop at line #2 of the code snippet in Figure 6 needs to be parallelized with a reduce operation using OpenMP + Offload directives so that is can run on coprocessors. The GUI interface was used for parallelizing this test case and the complete process is shown in the demo at [20]. Besides selecting the for-loop to parallelize, the end-user selects the variable to reduce from the list of variables provided by IPT and also selects the variables that need to be copied to the coprocessor and back to the host.

```c
1. wtime1 = gettime();
2. for (i = 0; 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 %10d: ",solution_num,i);
8.   for (j = 0; j < n; j++) {
9.     printf(" %d",bvec[j]);
10.   } printf("\n");
11.   solution_num = (solution_num + 1);
12. }

Figure 6. Snippet of Circuit Satisfiability Code - Serial
```

Seismic travel-time tomography is a technique that is used by geoscientists to create models for determining the velocity structure of the crust of the Earth. In order to resolve the earth’s structure with accuracy, this technique is extended to determine the uncertainty in the velocity (caused due to measurement errors in seismic travel-times) by using a tomographic algorithm designed and implemented by J. E. Vidale [16] and J. A. Hole [17]. The original version of the serial implementation is a set of programs written in C and Fortran. The Fortran programs were translated to C some time ago for educational purposes. Later, the section of the code that was the execution-bottleneck was rewritten in CUDA to run on GPUs. The serial implementation of this code (written in C) was also parallelized for GPUs using the GUI-based interface of IPT. The process of transforming this application using the GUI and the generated code are shown in the video demo at [21]. A code region with multiple nested for-loops was selected for parallelization in this test case. The IPT generated CUDA kernel which is 58 lines of code and added/modified additional 35 lines of code for declaring device variables, allocating memory space on GPU, copying from host to device, launching the CUDA kernel, and for copying back the data from device to host.

```c
wtime1 = gettime();
/***** Starting Parallelization *****/
#pragma offload target (mic) in (n, bvec) out(solution_num)
#pragma omp parallel default(none) shared(ihi, n,solution_num) private(i,value,j) firstprivate(bvec)
{
    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]);
            }
        }
    }
/***** Ending Parallelization *****/
wtime2 = gettime();
```

Figure 7. Snippet of Circuit Satisfiability Code - Parallel

Poisson Solver is a representative application from the class of stencil-based computations that is used in the Computational Fluid Dynamics (CFD) and in the earth sciences domains. In earth sciences, Poisson solver is used for modeling and investigating many aspects of the mechanical behavior of faults and fractures in the earth’s brittle crust [19]. In this case-study, a solution to a 2D Poisson problem with a five-point stencil is considered. The solution involves iterative computation of values at each point in the computational domain using the neighboring cells from the previous iteration till the convergence criterion is satisfied. A snippet of the serial code of Poisson Solver is shown in Figure 8.

```c
1. //other code
2. NTIMES = atoi(argv[3]);
3. a = allocMatrix<double>(a, M, N);
4. b = allocMatrix<double>(b, M, N);
5. f = allocMatrix<double>(f, M, N);
6. start = 0;
7. //other code
8. printMatrix<double>(a, M, N);
9. t1 = gettime();
10. for (k = 0; k<NTIMES && norm>=tolerance; k++){
11.   b = compute(a, f, b, M, N);
12.   ptr = a;
13.   a = b;
14.   b = ptr;
15.   norm = normdiff(b, a, M, N);
16. }
17. t2 = gettime();
```

Figure 8. Snippet of the Poisson Solver Code - Serial
To parallelize the Poisson Solver application, the matrices \( a \) and \( b \) (see Figure 8) should be blocked and the cells at the border of the blocks should exchange values with their neighbors after initialization. The neighboring blocks should exchange the value of the border-cells of matrix \( b \) in every iteration of the for-loop starting at line # 10 of Figure 8. Apart from exchanging boundary values, the value of \( \text{norm} \) computed in every iteration of the for-loop is also reduced. The IPT does static-analysis of the code and prompts the end-user for getting precise specifications for parallelization. A screenshot of some of the steps in the parallelization process is shown in Figure 9. A snippet from the generated code is shown in Figure 10. The code inserted by IPT is in bold-face. The statement for including the template for exchanging the boundary values of the matrices is at lines # 20, 21 and 24 of Figure 10. The exchange template is written using C++ and MPI and is approximately 128 lines. The code for setting the MPI Cartesian topology and data partitioning is part of the other templates and functions that are inserted by IPT. In the generated code, the user gets block or linear mapping of data. They can select cyclic or block-cyclic schemes too.

```
for (k = 0; k < NTIMES && (norm >= tolerance); k++) {
    if (this is the for loop you are looking for?)
        \( a = \text{compute}(a, f, b, M, \text{P}) \);
    ptr = a; \( a = b; b = \text{ptr} \); \( \text{norm} = \text{normdiff}(b, a, M, N) \); } 

//other code
2. double rose_norm;
3. MPI_Comm comm2d, rowcomm, colcomm;
4. LinearMapping<int> rowmap, colmap;
5. MPI_Init(&argc, &argv);
6. MPI_Comm_size(MPI_COMM_WORLD, &rose_size);
7. MPI_Comm_rank(MPI_COMM_WORLD, &rose_rank);
8. MPI_Comm(comm2d, rowcomm, colcomm);
9. LinearMapping<int> rowmap, colmap;
10. MPI_Init(&argc, &argv);
11. MPI_Comm_size(MPI_COMM_WORLD, &rose_size);
12. MPI_Comm_rank(MPI_COMM_WORLD, &rose_rank);
13. MPI_Comm(comm2d, rowcomm, colcomm);
14. myrows = rowmap.getMyCount();
15. mycols = colmap.getMyCount();
16. tempM_fraspa = M;
17. tempN_fraspa = N;
18. M = myrows;
19. //other code
20. b = exchange<double>(b, M+2, N+2, P, Q, p, q, comm2d, ...);
21. a = exchange<double>(a, M+2, N+2, P, Q, p, q, comm2d, ...);
22. for (k = 0; k < NTIMES && norm >= tolerance; k++) {
      b = compute(a, f, b, M, N);
      b = exchange<double>(b, M+2, N+2, P, Q, p, q, comm2d, ...);
      ptr = a;
      a = b;
      b = ptr;
      \( \text{norm} = \text{normdiff}(b, a, M, N) \);
  }
23. MPI_Allreduce(&norm, &rose_norm, 1, MPI_DOUBLE, ...);
24. \( \text{t2} = \text{gettime()} \);
```

Figure 9. Parallelizing through command-line interface

Figure 10. Snippet of the Poisson Solver Code – Parallel
The design templates in IPT are reusable across a diverse range of applications—e.g., the templates for inter-process communication and data-distribution for stencil-based computations can be used for parallelizing Poisson’s Solver as well as a trivial Game of Life problem [10]. The run-time comparison of the manually written code and the generated code for the test cases presented in this section is shown in Figures 11, 12, and 13. The generated and manual versions were run with the same problem sizes and input conditions. As can be noticed from the results in these figures, there is no significant difference in the performance of the generated code and the manually written code.

In this research we are quantifying end-user productivity in terms of the reduction in (1) the number of lines of code written for parallelizing existing code for multiple target platforms, and (2) the time-to-parallel-implementation, and hence the production of scientific results. The time-to-implementation includes the learning-curve associated with the different approaches to parallelization. As can be observed through the test cases presented in this section, the end-user productivity is significantly enhanced in terms of the effort involved in parallelization and hence, in getting started with using HPC platforms.

We are studying the impact that IPT has on enhancing the learning experience of its end-users. The end-users who participated in the preliminary study were computational scientists from diverse backgrounds: quantum chemistry, nuclear physics, and geosciences. These end-users had prior programming experience and were familiar with at least one parallel programming paradigm. The preliminary study has led to the following observations: (1) IPT reduces the effort involved in navigating to the hot-spots for parallelization, (2) IPT reduces the chances of errors during parallelization and hence reduces the effort involved in troubleshooting the errors, and (3) by following the comments inserted by IPT, the end-users can understand the changes made to their existing code for parallelization purposes. Further studies are being done for usability analysis of IPT.

6. RELATED WORK

In the pursuit of raising the level of abstraction of parallel programming, several research efforts have resulted in libraries, toolkits, languages and language extensions [22-25]. However, due to space constraints, only three related efforts are presented in this section and are compared to IPT.

Computer Aided Parallelization Tools (CAPTools) [9] is a software tool from 1990’s for interactively transforming serial Fortran applications into parallel applications with MPI or OpenMP directives. Literature review shows that CAPTools does not support the parallelization of C/C++ applications. It does not have support for CUDA. The task of data partitioning for MPI seems to be more involved in CAPTools than in IPT. Also, in contrast to IPT, CAPTools does not seem to let the user make any direct selection for parallelization or to apply an action directly on the selected region of code.

Intel’s Concurrent Collections (CnC) [26] provides a parallel programming model through which the end-user can graphically express parallelism in their applications. The end-users are required to identify data-dependencies and control-dependencies in their applications. The end-users of CnC for C++ provide the workflow using the graphical or textual symbols. The CnC translator then generates a header file and a code-template. The end-user edits the code-template file to provide the computation kernel. The end-user then writes the C++ code for the application and includes the generated header file. Finally, the object code is linked to Intel’s Threaded Building Block. In comparison to CnC, the end-user effort in using IPT is low in terms of the amount of manual editing and restructuring of the code/compute kernels.

Par4all is a compiler for automatically transforming C and Fortran programs into OpenMP, CUDA, and OpenCL code [27]. It is also based on a source-to-source compilation tool. In contrast to Par4all, IPT provides support for MPI and hybrid programming. From literature review it is not clear as to how Par4all supports
the various parallel programming patterns like stencil-based computations or irregular meshes.

7. CONCLUSION
In this paper, we reported recent progress on our tool for interactive parallelization— the support for generating CUDA code, support for a GUI interface, and improvement to the previous command-line interface to generate MPI code. Through the test cases presented in the paper, we have shown the usage and benefits of our tool. The magnitude of the impact of our tool in research and education is a direct function of (1) the importance of the HPC in science and engineering, and (2) the challenges that the domain-experts and students face in climbing the learning curve for parallel programming.

8. REFERENCES
[27] Par4all: http://www.par4all.org/documentation.html#users-guide