Refining High Performance FORTRAN Code from Programming Model Dependencies

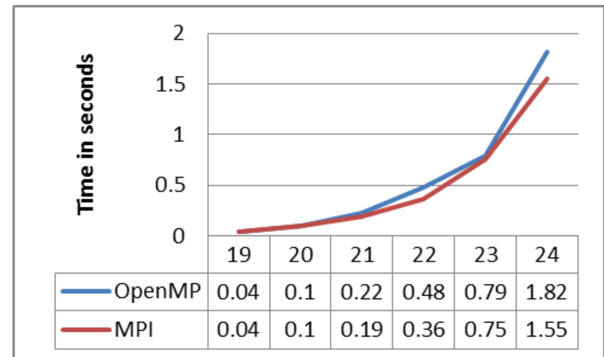
Ferosh Jacob*[⋆], Jeff Gray*, Purushotham Bangalore[†] and Marjan Mernik[‡]
*Department of Computer Science
University of Alabama
Tuscaloosa, AL 30487
Email: {fjacob, gray}@ua.edu
[†]Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35205
Email: puri@cis.uab.edu
[‡]Faculty of Electrical Engineering and Computer Science
University of Maribor
Maribor, Slovenia
Email: marjan.mernik@uni-mb.si

Abstract

For next generation applications, programmers will be required to adapt to a new style of programming to utilize the parallelism in the processors available to them. Abstractions in parallel programming languages and directives or annotations in sequential code have shown initial promise in reducing some of the burden of parallel programming. However, even with all of these advances, parallel programming still requires skill beyond that possessed by an average programmer. This is primarily due to the architecture-specific details in the domain. In this paper, we introduce a new approach to separate architecture dependencies from the program logic, enabling the programmer to execute the same computation in different platforms without actually changing the programming logic, but with capabilities to fine-tune the performance in the target platform. In this paper, we focus on refactoring existing FORTRAN code to support the latest HPC libraries.

1. Introduction

Parallel programming is essential for large simulations and cross-disciplinary scientific inquiries. Parallelism enables the decomposition of a large computational domain into numerous subdomains, and employs a large number of processors to compute the solution simultaneously in parallel on these subdomains. FORTRAN has been the preferred language among the High Performance Computing (HPC) community over the last 40 years. E. Loh argues that HPC benchmarks can be written in the "ideal HPC language, FORTRAN" in comparable lines of code with that of new programming languages [1]. As an HPC language, there exist many legacy applications written in FORTRAN.



	19	20	21	22	23	24
OpenMP	0.04	0.1	0.22	0.48	0.79	1.82
MPI	0.04	0.1	0.19	0.36	0.75	1.55

Figure 1. OpenMP/MPI programs with problem size

Using traditional approaches that do not allow separation of the computation from the underlying HPC library, a developer must maintain simultaneously the same set of simulation code for each variation of HPC APIs. As an example, the execution plot of the satisfiability problem in Figure 1 shows that even though the performance of OpenMP and MPI are comparable, for small problems the OpenMP version is faster than an MPI solution. In cases where the size of the data varies, different versions of the same program might be required if a single HPC library is used. Manually maintaining such variations induces unnecessary redundant effort that is also very prone to human errors in maintaining and updating the core algorithms. Therefore, the development of an HPC program is often limited to a specific parallel library. Otherwise, the programmer pays the price

⋆ Student

of developing and maintaining several versions of the same program.

Because of the portability issues and dearth of FORTRAN programmers, there may be some benefits in combining existing FORTRAN applications with a language-independent DSL (Domain-Specific Language). If the DSL can express the problem without any language dependency, then the computation could be executed in platforms that support parallel execution with proper configuration. The DSL should be powerful enough to express any parallel problem and configurable enough to provide optimal performance. We propose a process whereby a legacy application written in FORTRAN (using MPI, OpenMP or a combination of both) is converted to an application that is complimented by a DSL. The refactoring of the legacy application would support retargeting of the application to different HPC platforms (e.g., OpenMP, MPI, CUDA) without significant performance loss.

Our recent work has focused on a DSL named CalCon [2], which has two parts representing the core computation and the configuration concerns of a specific platform and programming model. The description of the core computation is powerful enough to express a computation in a generalized manner that is free from platform-specific issues that often tangle the code base. The configuration part of CalCon is tightly coupled to the execution environment. In addition, optimized libraries or templates for the new programming model and/or language will be required so that the new code generated delivers desired performance. With this intent, we developed Abstract APIs [3] that provide an interface for two leading GPU programming languages: CUDA and OpenCL. CUDACL [4] is a tool developed over the Abstract APIs to configure GPU parameters. By introducing CalCon, support is extended to shared memory from GPUs. CalCon has shown the benefits of separating the computation from the platform to allow retargeting and adaptation of the computation to new paradigms (e.g., from shared memory to a GPU-based solution). CalCon was targeted to the C programming language and more emphasis was given to the development phase. This paper introduces the extension of similar ideas to FORTRAN, with more emphasis on the maintenance phase of software engineering.

This paper is a first stage in refactoring the FORTRAN MPI/OpenMP code to parallel code and is structured as follows. Section 2 discusses the analysis of ten FORTRAN programs written for OpenMP in diverse domains. Section 3 explains specific details of the approach used to separate parallel concerns from the shared memory dependencies. Section 4 includes a case study to extend the approach to MPI. A list of related works is enumerated in Section 5. The paper concludes with some future works in Section 6.

Table 1. Analysis of OpenMP FORTRAN programs

No	Program Name	Total LOC	Parallel LOC	No. of blocks	R	W
1	2D Integral with Quadrature rule	601	11 (2%)	1	√	
2	Linear algebra routine	557	28 (5%)	4		√
3	Random number generator	80	9 (11%)	1		
4	Logical circuit satisfiability	157	37 (18%)	1	√	
5	Dijkstras shortest path	201	37 (18%)	1		
6	Fast Fourier Transform	278	51 (18%)	3		
7	Integral with Quadrature rule	41	8 (19%)	1	√	
8	Molecular dynamics	215	48 (22%)	4	√	√
9	Prime numbers	65	17 (26%)	1	√	
10	Steady state heat equation	98	56 (57%)	3	√	√

Table 2. Classification of OpenMP directives

Shared memory features	Parallel feature
Variable modifiers, Critical and Singular blocks, Number of threads	Parallel blocks, Reduction and Barrier blocks, Number of instances, Workshare

2. Program analysis of FORTRAN OpenMP programs

To understand the usage of OpenMP in FORTRAN programs, ten programs from different domains were selected. A Summary of the analysis is shown in Table 1. Total Lines of Code (LOC) shows the total number of lines in the program, excluding comments, print statement and blank lines. Parallel LOC shows the number of lines inside a parallel block including the OpenMP declarations. The ratio of parallel LOC to the total can be a measure to the parallel nature of the programs. The “No. of blocks” column shows the number of parallel blocks in each program. The last two columns in the table illustrate whether the parallel block has workshare and reduction operations. The workshare directive in OpenMP splits work among the available threads, and the reduction directive combines the variable values of threads to a single value when exiting the block.

The goal of the analysis was to separate parallel programming from the shared memory model dependencies. The findings after the analysis are shown in Table 2 and explanation for including a directive to one of the categories is discussed in the following subsections.

2.1. Shared Memory features

This category includes the OpenMP directives which bind the parallel programs to the shared memory architecture or

features of an OpenMP program that are not required to express the parallel nature of the problem. These directives may not be applicable if these programs are ported to a specific architecture, but are essential for performance tuning in shared memory architectures.

- **Variable modifiers:** OpenMP provides capabilities to define the scope of variables using keywords `private`, `shared`, `firstprivate`, `copyin`. These modifiers clearly have dependencies to the shared memory architecture and may be useful for a platform with different levels of shared memory (e.g., GPUs have global, constant and local shared memory).
- **Critical and Singular blocks:** If a block of code is defined as a critical block then it will be executed by only one thread at a time. In a programming model where the execution variables are not shared, the critical block is irrelevant.
- **Number of threads:** The number of threads for optimal performance is limited by either the execution environment or the context of the current problem. When executed in another environment (e.g., MPI), the important parameter will be the number of processes. Hence, in general, the number of instances may be required to express a parallel computation.

2.2. Parallel features

This category includes the parallel features that are required to express the parallel computation. An assumption is that the programmer is using the Single Program Multiple Data (SPMD) programming style. The features identified from the parallel programs analyzed are explained as follows:

- **Parallel blocks:** In the Single Program Multiple Data (SPMD) style, there should be a way to distingush the code to be executed by a single thread and the code to be executed in parallel. Parallel blocks can define the code that must be executed in parallel.
- **Reduction and Workshare blocks:** Reduction and Workshare are included as parallel features not in the same meaning as they are used in OpenMP. Workshare refers to setting up the data for executing and reduction refers to distributing the data after execution.
- **Number of instances:** This refers to the number of threads in shared memory or processes in MPI. In a massively parallel environment like a GPU, the number of threads is usually limited by the context of the problem. As such, the number of threads is a problem detail and not an implementation detail.
- **Barrier:** Barrier is a key concept in parallel programming that provides synchronization among the instances.

```
! Refined FORTRAN program
call parallel(instance_num, satisfiability )
  ilo2 = ( ( instance_num  - id      ) * ilo   &
         + (               id      ) * ihi ) &
         / ( instance_num           )
  ihi2 = ( ( instance_num  - id - 1 ) * ilo  &
         + (               id + 1 ) * ihi ) &
         / ( instance_num           )
  solution_num_local = 0

  do i = ilo2, ihi2 - 1
    call i4_to_bvec ( i, n, bvec )
    value = circuit_value ( n, bvec )
    if ( value == 1 ) then
      solution_num_local = solution_num_local + 1
    end if
  end do
  solution_num = solution_num + solution_num_local
call parallelend( satisfiability )

! Configuration file for FORTRAN program above
block  satisfiability
init:
!$omp parallel &
!$omp shared ( ihi, ilo, thread_num ) &
!$omp private ( bvec, i, id, ilo2, ihi2,
                j, solution_num_local, value ) &
!$omp reduction ( + : solution_num ).
final:.
```

Figure 2. Refined program for the satisfiability problem

2.3. Analysis Conclusion

A DSL that uses only the parallel features can express parallel problems in a platform-independent manner. Most of the programs involve an initialization segment that initializes the execution of the parallel part, and a code segment that is used to collect data from the parallel instances.

3. Proposed approach to express parallel programs in FORTRAN

To address the learning curve challenges of adopting a new parallel API, the CalCon DSL uses FORTRAN function calls to specify the parallel features in a program. The program looks like a typical FORTRAN program with some additional function calls that are not defined inside the program. These function calls are later parsed by the CalCon compiler, which replaces the function calls with generated FORTRAN code based on the configuration (machine or architecture details) specified in the DSL.

In a shared memory programming model, the programs are written such that communication is explicit and synchronization is implicit. This is in contrast to most distributed programming models, where synchronization is explicit and communication is explicit. To an approach that can serve as a substitute for both of these programming models, both communication and synchronization has to be explicit. Similar to OpenMP, if not specified as a parallel block, code will be executed only by a single/main thread.

Example source code and configuration is shown in Figure 2 for the circuit satisfiability problem. From our analysis, a

parallel program consists of blocks of code that have to be executed in parallel. Before and after each parallel block there can be a code segment that is machine or architecture dependent. As shown in Figure 2, the machine dependent blocks are separated in a configuration file.

3.1. Implementation details of the approach

If the targeted platform is shared memory, we start with a template of an OpenMP FORTRAN program with the required definitions and libraries. The program file is read by a FORTRAN parser and written into the template line by line; any occurrences of pre-defined functions (`parallel`, `endparallel`) in the program are replaced by the corresponding blocks defined in the configuration file. The template has pre-defined variables like `instance_num` (total number of instances), and `id` (identifier for the current instance). Users can define their own functions as `distribute`, which is explained in the case study.

4. An MPI case study: Integral estimation using the quadrature rule

In this section, we explain our approach applied to the implementation of an MPI program to estimate an integral shown in the equation $\frac{50}{\pi}\int_0^{10}\frac{dx}{2500x^2+1}$.

Figure 3 shows important parts of a conventional MPI program to estimate the integral using the quadrature rule. The program shows the general structure of an MPI program and is divided into three sections: 1) Setting up the data, 2) Parallel Execution, and 3) Collecting results. The following discussion identifies these three sections in the MPI program and explains how our approach is applied to the problem.

4.1. MPI program for integral estimation

An MPI program has some initialization code that specifies the number of processes, the id of each process (`MPI_Comm_size, MPI_Comm_rank`). These lines appear at the beginning of the program. The three sections of integral estimation program are explained as follows:

1) **Setting up the data:** In this step, the master process divides the data so that it can be made accessible to other processes. The master process broadcasts or sends the variables to another process, in this case after calculating the corresponding values of `my_a` and `my_b` to each process. The calculation and sending of `my_b` is not shown in the code of Figure 3 to improve clarity. Usually, this code is executed by the master thread (`my_id==0`); hence, the code is inserted inside the conditional shown in Figure 3.
2) **Parallel execution:** This section includes the code to be executed in parallel. Depending on whether the master thread is participating in the execution, the code is inserted inside the `else` condition. Before execution, each process receives the data sent or broadcasted by the master thread.
3) **Collecting results:** After finishing the execution, every process has a total (`my_total`) and the final total (`total`) that is the sum of individual process totals. `MPI_Reduce` achieves this in the program.

```
!Part 1: Master process setting up the data
if ( my_id == 0 ) then    do p = 1, p_num - 1
      my_a = ( real ( p_num - p,      kind = 8 ) * a   &
             + real (          p - 1, kind = 8 ) * b ) &
             / real ( p_num     - 1, kind = 8 )
      target = p
      tag = 1
      call MPI_Send ( my_a, 1, MPI_DOUBLE_PRECISION, &
                      target, tag, &MPI_COMM_WORLD, &
                      error_flag )
    end do

!Part 2: Parallel execution
  else
    source = master
    tag = 1
    call MPI_Recv ( my_a, 1, MPI_DOUBLE_PRECISION, source,
        ↪  tag, &
      MPI_COMM_WORLD, status, error_flag )
    my_total = 0.0D+00
    do i = 1, my_n
      x = ( real ( my_n - i,      kind = 8 ) * my_a   &
          + real (        i - 1, kind = 8 ) * my_b ) &
          / real ( my_n     - 1, kind = 8 )
      my_total = my_total + f ( x )
    end do
    my_total = ( my_b - my_a ) * my_total / real
                                        ( my_n, kind = 8 )
  end if

!Part 3: Results from different processes are collected to
! calculate the final result
  call MPI_Reduce ( my_total, total, 1,
                    MPI_DOUBLE_PRECISION, & MPI_SUM,
                    master, MPI_COMM_WORLD, error_flag)
```

Figure 3. MPI program for integral estimation

4.2. Refined parallel program for integral estimation

The refined parallel program using the new approach is shown in Figure 4. The refined parallel program that uses our approach is independent of any architecture or language. The template used has definitions and libraries required for MPI execution, as well as pre-defined variables for receiving the process identifier (`id`) and total number of processes (`instance_num`). The new code after the refined program will be inserted inside the `if (id=0)` condition and the parallel block code would be inserted into the `else` condition.

5. Related works

There has been much effort in converting sequential code to parallel code even before the introduction of the GPU

```
!Work share part
    do p = 1, instance_num - 1
      my_a = ( real ( instance_num - p,      kind = 8 ) * a
           ↪    &
             + real (          p - 1, kind = 8 ) * b ) &
             / real ( instance_num     - 1, kind = 8 )
      call distribute (my_a)
    end do

!Declaring parallel block
 call parallel(num, quadrature )
    my_total = 0.0D+00
    do i = 1, my_n
      x = ( real ( my_n - i,      kind = 8 ) * my_a   &
          + real (        i - 1, kind = 8 ) * my_b ) &
          / real ( my_n     - 1, kind = 8 )
      my_total = my_total + f ( x )
    end do
    my_total = ( my_b - my_a ) * my_total / real
                                         ( my_n, kind = 8 )
 call endparallel( quadrature );

! Configuration file for FORTRAN program above
block  quadrature
init:
    source = master
    tag = 1
    call MPI_Recv ( my_a, 1, MPI_DOUBLE_PRECISION, source,
         ↪  tag, &
      MPI_COMM_WORLD, status , error_flag ).
final:
call MPI_Reduce ( my_total, total, 1,
                   MPI_DOUBLE_PRECISION, & MPI_SUM,
                   master, MPI_COMM_WORLD, error_flag).
distribute param:
      call MPI_Send ( param, 1, MPI_DOUBLE_PRECISION, &
                    target , tag, &MPI_COMM_WORLD, &
                    error_flag ).
```

Figure 4. Refined program for integral estimation

[5]. Automatic parallelization of sequential code may not be the perfect solution for some cases, as revealed by our analysis. Parallel FORTRAN Converter (PFC) [6], [7] is a program to convert sequential programs written in FORTRAN to FORTRAN 8x, which is a version of FORTRAN compatible with vector computers. This program replaces loops with array operations wherever possible by studying the data dependency in the program. PAT [8] is another tool supporting interactive conversion of sequential code to parallel code in FORTRAN. Another research direction is converting code from one model into another [9], [10]. These efforts are not targeted for FORTRAN. Furthermore, they miss the opportunity to identify the parallel features from a program. Hence, when porting to a new architecture, the program has to be rewritten.

6. Conclusion

For the evolution of high performance FORTRAN code, it is necessary to separate the code of the core computation from the machine or architecture dependencies that may come from usage of a specific API. We analyzed ten FORTRAN programs from diverse domains to understand the usage of OpenMP in scientific code. The analysis revealed that programs often share a common structure such that platform and machine details could be specified in a different file. A case study is included to show that the approach can be extended to other architectures.

Future work includes refactoring the legacy code to the approach specified in this paper with minimum input from the user. Another direction will be focused on executing the parallel programs to a GPU. Conducting a user study to explore the advantages and disadvantages from a human factors perspective is another direction of work.

References

[1] F. Jacob, D. Whittaker, S. Thapaliya, P. Bangalore, M. Mernik, and J. Gray, "CUDACL: A tool for CUDA and OpenCL programmers," in *Proceedings of the 17th International Conference on High Performance Computing*, Goa, India, in press.

[2] F. Jacob, R. Arora, P. Bangalore, M. Mernik, and J. Gray, "Raising the level of abstraction of GPU-programming," in *Proceedings of the 16th International Conference on Parallel and Distributed Processing*, Las Vegas, NV, July 2010, pp. 339–345.

[3] F. Jacob, "Extending abstract GPU APIs to shared memory," in *Proceedings of the ACM International Conference companion on Object Oriented Programming Systems Languages and Applications companion*, Reno, NV, October 2010, pp. 217–218.

[4] E. Loh, "The ideal HPC programming language," *Communications of the ACM*, vol. 8, no. 6, pp. 42–47, 2010.

[5] R. Allen and K. Kennedy, "Automatic loop interchange," *SIGPLAN Notes*, vol. 39, no. 4, pp. 75–90, 2004.

[6] A. Basumallik and R. Eigenmann, "Towards automatic translation of OpenMP to MPI," in *Proceedings of the 19th International Conference on Supercomputing*, Cambridge, MA, June 2005, pp. 189–198.

[7] R. Allen and K. Kennedy, "PFC: A program to convert FORTRAN to parallel form," in *Technical Report MASC-TR82-6*, Rice University, Houston, TX, March 1982.

[8] B. Appelbe, K. Smith, and C. McDowell, "Start/Pat: A parallel-programming toolkit," *IEEE Software*, vol. 6, no. 4, pp. 29–38, 1989.

[9] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," in *Proceedings of the 14th symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, February 2009, pp. 101–110.

[10] Y.-S. Kee, J.-S. Kim, and S. Ha, "ParADE: An OpenMP programming environment for SMP cluster systems," in *Proceedings of the 17th International Conference on Supercomputing*, Phoenix, AZ, November 2003, pp. 12–15.