

# Refining High Performance FORTRAN Code from Programming Model Dependencies

Ferosh Jacob, Department of Computer Science, University of Alabama

Jeff Gray, Department of Computer Science, University of Alabama

Purushotham Bangalore, Department of Computer and Information Sciences, University of Alabama at Birmingham

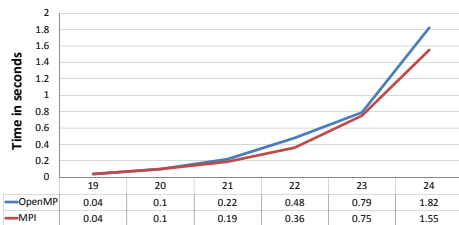
Marjan Mernik, Faculty of Electrical Engineering and Computer Science, University of Maribor

Contact: fjacob@crimson.ua.edu

## Introduction

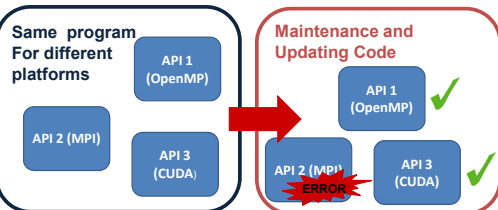
For next generation applications, programmers will be required to adapt to a new style of programming to utilize the parallelism in the processors available to them. Even with new advances, parallel programming still requires skill beyond that possessed by an average programmer. This is primarily due to the architecture-specific details in the domain. In this poster, 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.

## Traditional Approaches to Parallel Programming

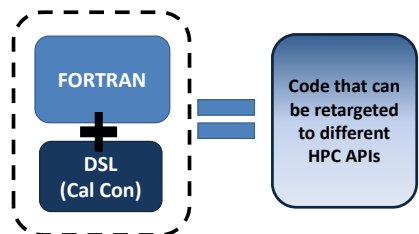


Using traditional approaches, a developer must maintain simultaneously the same set of simulation code for each variation of HPC APIs. For example, the figure above shows that even though the performance of OpenMP and MPI are comparable, for small problems OpenMP is faster than an MPI solution. In cases where the size of data varies, different versions of the same program might be required if a single HPC library is used.

Manually maintaining the variations needed causes unnecessary redundant effort and is also prone to human errors in maintaining and updating the core algorithms.



## Separating Architecture Dependencies from Program Logic



By combining existing FORTRAN applications with a language-independent DSL (Domain Specific Language), then the computation could be executed in platforms that support parallel execution with the proper configuration. This poster introduces a DSL named CalCon.

## Analysis of FORTRAN OpenMP Programs

To understand the usage of OpenMP in FORTRAN, ten programs from different domains were selected. The table summarizes the analysis: No. of block refers to number of parallel blocks, R to reduction operations, and W to workshare operations.

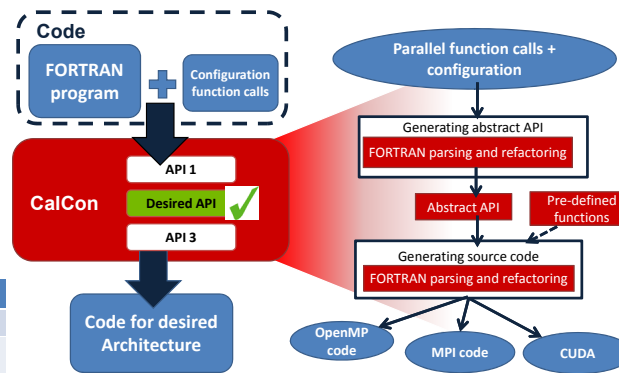
No	Program Name	Total LOC	Parallel LOC	No. of blocks	R	W
1	2D Integral with Quadrature rule	601	11 (2%)	1	✓	
2	Linear algebra routine	557	28 (5%)	4		✓
3	Random number generator	80	9 (11%)	1		
4	Logical circuit satisfiability	157	37 (18%)	1	✓	
5	Dijkstra's shortest path	201	37 (18%)	1		
6	Fast Fourier 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	✓✓	

## Classifications of OpenMP Directives

Shared Memory Features	Parallel Features
• Variable modifiers	• Parallel blocks
• Critical blocks	• Reduction blocks
• Singular blocks	• Barrier blocks
• Number of threads	• Number of instances
	• Workshare

## Internal details of the proposed approach

CalCon 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 architecture details) specified in the DSL.



```

Refined FORTRAN program
call parallel(instance_num,'satisfiability')
i1o2 = ( ( instance_num - id ) * i1o &
+ ( ( instance_num - id ) * i1i ) * i1i ) &
/ ( instance_num - id ) * i1i &
i1i2 = ( ( instance_num - id - 1 ) * i1o &
+ ( ( instance_num - id + 1 ) * i1i ) &
/ ( instance_num - id ) * i1i &
solution_num_local = 0
do i = i1o2, i1i2 - 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 ( i1i, i1o, thread_num ) &
!$omp private ( bvec, i, id, i1o2, i1i2,
, solution_num_local, value ) &
!$omp reduction ( + : solution_num ).
final:
    
```

## Related Work

Sequential code to Parallel	Related Work
CUDA	• PFC • PAT • hiCUDA • CUDA-lite • PGI
OpenCL	• CUDACL

## Conclusion and Future Work

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 poster 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 future work.

## References

- J. Jacob, D. Whittaker, S. Thapaliya, P. Bangalore, M. Mernik, and J. Gray. "CUDACL: A tool for CUDA and OpenCL programmers." In Proceedings of 17th International Conference on High Performance Computing, Goa, India, December 2010, 11 pages., in press.
- J. 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, p. 339-35.
- [E. Loh, "The ideal HPC programming language." Communications of the ACM, vol. 53, no. 7, pp. 42-47, 2010.
- [A. Basumalik, and R. Eigenmann, "Towards automatic translation of OpenMP to MPI." In Proceedings of the 16th Annual International Conference on Supercomputing, Cambridge, MA, June 2005, p.189-198.
- R. Allen and K. Kennedy. "PFC: A program to convert FORTRAN to parallel form." Rice University, Houston, TX, Technical Report MASC-TR82-6, March, 1982.

## MPI Case Study: Integral Estimation Using the Quadrature Rule

### MPI Program for Integral Estimation

```

!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)
    
```

### Refined Parallel Program for Integral Estimation

```

!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 ).
    
```