CC BY

ISSN 1846-6168 (Print), ISSN 1848-5588 (Online)
https://doi.org/10.31803/tg-20241206111011

Original scientific paper
Received: 2024-12-06, Accepted: 2025-09-30

Framework for Improving Parallelization of Loops through Loop-Fission Transformation for Many-Core Systems

Akash Yadav*, Mushtaq Ahmed

Abstract: Many-core systems rely on parallel processing to achieve high-performance computing, but writing efficient parallel code remains challenging, especially for researchers with limited programming expertise. Automatic parallelization tools such as autoPar, Cetus, and Par4All help transform sequential programs into parallel ones, yet they often fail to handle certain programming constructs efficiently. This paper proposes a loop-fission transformation-based framework to improve the parallelization of dependency-free loops with control flow statements. The framework restructures loops to maximize parallel execution, overcoming limitations in existing tools. Experiments include computational statements like array assignments, addition, and multiplication to evaluate performance. Results show that the proposed approach reduces execution time by 4% to 15% on many-core systems compared to existing tools. These findings highlight the inefficiencies of current automatic parallelization techniques and demonstrate the effectiveness of the proposed method in enhancing execution performance.

Keywords: Loop Parallelization; Loop-Fission; OpenMP; Parallelizing Compilers; Superword-Level Parallelization

1 INTRODUCTION

Traditionally, CPU processing power was increased by increasing the CPU's clock speed. The CPU consumes power and generates more heat to operate at a higher clock speed. The power that can be supplied to the CPUs is limited. This limitation of power supply is known as the Power Wall problem [1]. This problem downgrades the overall system performance. However, there are more and more data-intensive and computationally intensive applications that demand high-performance processor design, e.g., big data, machine learning, and deep learning techniques and applications [2-5]. To overcome this problem, manufacturers increase the number of cores. Today, multi and many-core architectures are developing on a larger scale as parallel computing and High-Performance Computing (HPC) are now the necessary key technologies for the latest industry standards [6].

Multi-core and many-core processors are distinct types of computer processors based on the number of cores they contain. Multi-core processors feature a limited number of independent processing units within a single package, which is suitable for parallel processing in general-purpose computing tasks. On the other hand, many-core processors have a significantly larger number of cores, exceeding dozens or even hundreds, and are optimized for highly parallel workloads like scientific simulations and data analytics [7]. While multi-core processors are commonly found in desktops and laptops, many-core processors are often used in specialized applications, HPC environments, and data centers.

Both multi-core and many-core systems support parallel computing, with multi-core processors enabling low to moderate levels of parallelism for general-purpose tasks, while many-core processors excel in handling highly parallel workloads in specialized applications [8]. The many-core processors maximize parallelism and performance in specialized applications. These many-core machines can run parallel applications to give better performance. This led to the rise of parallel or concurrent programming to give optimum performance [9, 10]. Also, programmers and developers need to create applications or programs that can fully utilize the processing power of these parallel processors. However, creating a parallel program is not a simple task. It requires in-depth knowledge of parallel programming, its underlying concepts, and parallel paradigms [11]. Depending on the machine architecture, parallelism can be achieved in many ways. A parallel application can use any available method (such as message passing, shared memory, etc.) to run in parallel [12].

Most programmers find it a tedious task to write parallel codes, which leads to the requirement of parallelization tools [13, 14]. A parallelization tool can automatically convert a sequentially written program into a parallelized one. Today, many parallelization tools are available to perform automatic parallelization [15-17]. These tools insert OpenMP directives into the sequential code to achieve parallelism [18, 19]. However, this automatic conversion is still not very efficient, particularly when a program contains loops with conditional statements inside their body, such as if-else constructs [20, 21].

Many researchers and developers rely on automatic parallelization tools to simplify this process, but these tools often fail to achieve optimal performance in such cases. This limitation motivates the need for improved techniques that can effectively handle control flow within loops. This paper explores a loop-fission transformation-based approach to enhance parallelism, making automatic parallelization more effective for complex loop structures.

The rest of the paper is structured as follows. Section 2 provides information about automatic parallelization tools, loop transformation techniques, and background information on OpenMP API. The proposed framework is discussed in Section 3. Section 4 presents the experimental setup and test cases used in the experiments and results evaluation of the

proposed framework on these test cases, followed by a conclusion.

2 EXISTING TECHNIQUES FOR AUTOMATIC PARALLELIZATION

In parallel programming, many types of parallelization exist, such as Instruction-Level Parallelization (ILP), Loop-Level Parallelization (LLP), Superword-Level Parallelization (SLP) and many others [22, 23]. In SLP, similar types of independent isomorphic statements within a basic block are grouped into a single Single Instruction Multi Data (SIMD) operation. Instead of executing those isomorphic statements one at a time, this SIMD operation is executed, allowing them to run simultaneously [23]. Fig. 1 shows an example of SLP.

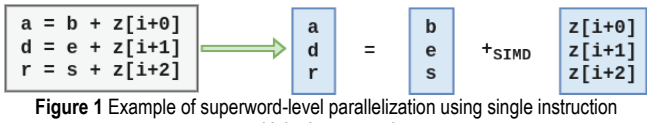


Figure 1 Example of superword-level parallelization using single instruction multiple data operation

Most of the programs spend almost 90% of their execution times in the loops [24]. Hence, LLP is a very common technique to parallelize a sequential code to speed-up the execution time [25]. Many loop transformation methods (such as loop unrolling, loop splitting, loop fission, etc.) are available that can be used to parallelize the loops. Loop unrolling generally leads to superword-level parallelism, as shown in Fig. 2.

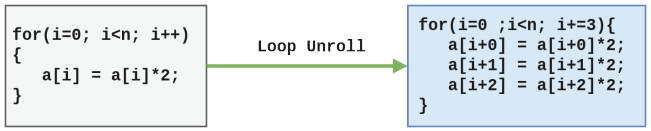


Figure 2 Example of loop unrolling transformation

SLP cannot be done when a loop contains the conditional statement in its body, as shown in Listing 1.

Listing 1 Example of a loop with conditional statements in its body

```
for(i=0;i<256;i++)
{
  if(a[i]%2==0)
  {
    a[i]=a[i]*2;
  }
}
```

In this case, loop unrolling does not create independent isomorphic statements that can be efficiently used for SIMD execution. In addition, automatic parallelization tools apply the OpenMP directive to the outer loop, which does not fully utilize parallelism. This type of loop is most common in multimedia-related areas such as image processing and video coding, such as MPEG.

2.1 Shared Memory Model and OpenMP

The shared memory model is a common programming model for parallel computing in which multiple processes or threads share access to a common memory pool, as shown in Fig. 3. This allows them to share data and communicate with each other more efficiently than other methods of interprocess communication, such as message passing.

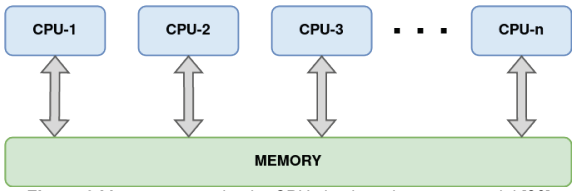


Figure 3 Memory accessing by CPUs in shared memory model [26]

One of the primary advantages of the shared memory model is its speed and efficiency. The same memory location can be accessed directly by all processes or threads, eliminating the requirement for message passing, which copies data between them [27]. This property can significantly decrease overhead and enhance performance, particularly for highly computational tasks that require a significant amount of communication between different processes. Shared memory systems can be utilised to enhance the performance of parallel computing applications, especially those involving large data sets [28, 29].

However, there are also some challenges that need to be considered when using the shared memory model. One of the most important issues is the requirement for meticulous synchronisation in order to prevent race situations and other concurrency problems. Furthermore, the approach may also be less scalable than alternative parallel programming models, such as message-passing models, for specific high-performance computing applications, especially when the number of processes or threads increases [30].

OpenMP is one of the prominent parallel programming standards for FORTRAN, C, and C++, enabling programmers to create parallel code using environment variables, runtime library functions, and compiler directives. The main objective of OpenMP is to simplify the writing of high-performance parallel programs that can utilise multiple CPU cores in a single system. OpenMP does have some restrictions, though. Its inability to enable distributed memory parallelism is one of its main drawbacks, which may limit its applicability to some complex HPC applications.

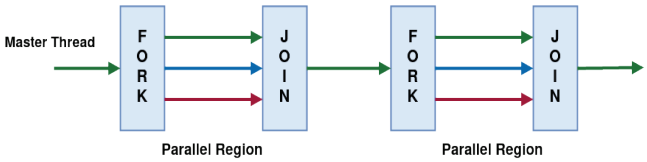


Figure 4 Parallelization by OpenMP using fork-join [31]

OpenMP uses the fork-join model to provide multi-threading-based parallelism. The fork-join model diagram is shown in Fig. 4. When the master thread encounters the parallel region, it executes the fork operation and generates

slave threads and then both master and slave threads start executing code in parallel. At the end of the parallel region, the master thread waits to complete the task assigned to the slave threads, and after completion, it executes the join operation.

Fig. 5 shows an example of parallelizing a loop using four threads. The master thread (in green colour) executes the program sequentially. When a parallel region occurs (*#pragma omp parallel for*), master threads create three slave threads and distribute the loop iterations between them (including itself) to run the loop in parallel. The loop iterations are divided equally between the master thread and slave threads. Each thread executes four iterations of loops. After completion of the task, master threads use the join operation to end the parallel execution and continue the execution of the sequential region.

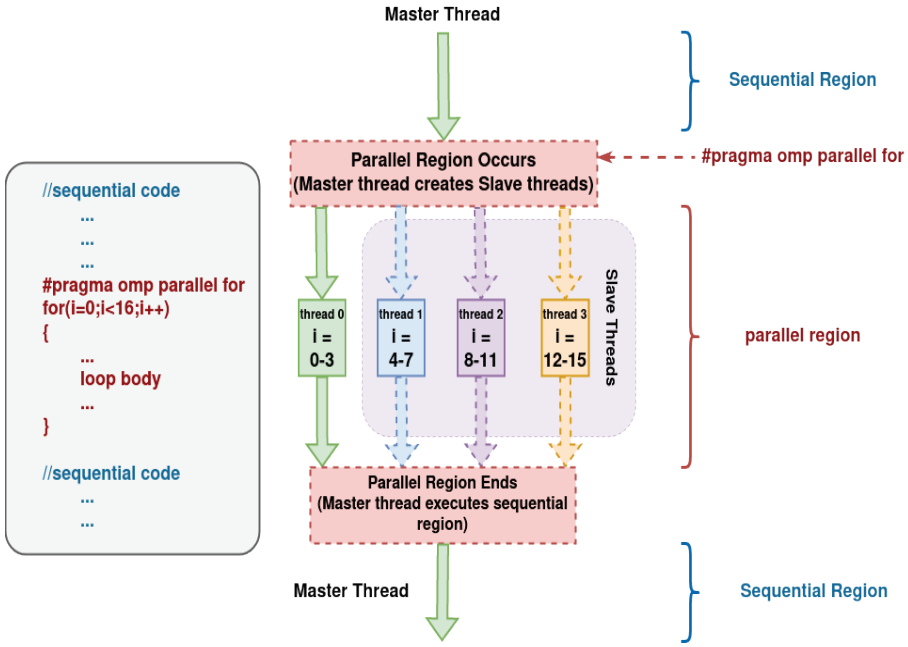


Figure 5 Parallelizing a for loop with 16 iterations using OpenMP with four threads

2.2 Automatic Parallelization Tools

Several automatic parallelization compilers can convert sequential code into its parallelized equivalent by inserting OpenMP directives [32]. In our experiments, we used AutoPar and Cetus, as they are free, up-to-date, and well-suited for parallelizing scientific codes. Both are source-to-source compilers that utilize the OpenMP API for parallelization.

```
for (i = 0; i < n ; i ++)
{
    a[i] = a[i] + b[i];
}
```

(a) Simple loop

```
for (i = 0; i < n ; i ++)
{
    if (a[i] != 0)
    {
        a[i] = a[i] + b[i];
    }
}
```

(b) Loop with control flow statement

Figure 6 Example of two different loops

The following subsections provide brief background information about autoPar and Cetus. These subsections also contain examples of how autoPar and Cetus parallelize the two loops with different body statements, as shown in Fig. 6. Listings 2–5 are code snippets that highlight the OpenMP directives inserted by autoPar and Cetus. The full programs are not shown here. In the complete codes, loop variables such as int i are declared. The tools also generate the long type casting and fixed values like 100000. These are shown exactly as produced.

2.2.1 autoPar

The autoPar is an auto parallelization tool in ROSE. ROSE is a compiler-based infrastructure for building program transformation and analysis tools [33]. It is developed at Lawrence Livermore National Laboratory. It is an open-source source-to-source compiler. ROSE tools support a wide range of programming languages, including C/C++, Python, PHP, and FORTRAN. ROSE includes an open-source automatic parallelization compiler called autoPar, which can currently parallelize C and C++ programmes by inserting OpenMP pragma in the sequential code. Listings 2 and 3 show how autoPar parallelizes the loops from Fig. 6 with an iteration count of 100000. Instead of the full code, the listings highlight only the changes made by the automatic parallelization tool, specifically the addition of OpenMP directives to parallelize the loops. All variables used in the loops are already declared in the original code, and the tool itself applies the typecasting of constants, such as long.

Listing 2 Parallelization of the simple loop by autoPar

```
#pragma omp parallel for private(i)
for (i=0; i<=((long )100000)-1;i+=1){
  a[i] = a[i] + b[i];
}
```

Listing 3 Parallelization of the loop with conditional statements in its body by autoPar

```
#pragma omp parallel for private(i)
for (i=0; i<=((long )100000)-1;i+=1){
  if (a[i] != 0) {
    a[i] = a[i] + b[i];
  }
}
```

2.2.2 Cetus

Cetus is a source-to-source C compiler written in Java and was developed and maintained by the ParaMount research group at Purdue University [34]. It is considered the most powerful and easy-to-use tool. Currently, it parallelizes only C programs.

Listing 4 Parallelization of the simple loop by Cetus

```
#pragma cetus private(i)
#pragma loop name main#0
#pragma cetus parallel
#pragma omp parallel for if((10000<(1L+3L*n)))) private(i)
for(i = 0; i < n; i++)
{
a[i] = a[i] + b[i];
}
```

Cetus's analysis passes can perform two types of data dependency analysis: Banerjee-Wolfe Test and Range Test. Its analysis pass also includes array privatization, symbolic expression manipulators, reduction recognition, etc. Transformation passes of Cetus include loop parallelizer, induction variable substitution pass, and loop dependent analysis. Automatic parallelization capability is a built-in capability of the Cetus compiler.

Listing 5 Parallelization of the loop with the conditional statement in its body by Cetus

```
#pragma cetus private(i)
#pragma loop name main#0
#pragma cetus parallel
#pragma omp parallel for if((10000<(1L+(3L*n)))) private(i)
for(i=0;i<n;i++){
  if(a[i]!=0){
    a[i]=a[i]+b[i];
  }
}
```

The listings illustrate how Cetus applies OpenMP to the loops in Fig. 6 for parallelization. Listings 4 and 5 show that Cetus parallelizes both loops in the same manner.

3 PROPOSED FRAMEWORK

As discussed in Section 2.2, parallelization tools apply the same parallelization technique to transform the loops with or without the control flow statements. Most of the time, compilers apply if-conversion to optimize the loop, but it is only applicable when the condition and data associated with it remain the same in every iteration of the loops.

These types of loops can be easily parallelized. But when there is an array-based if condition where data associated with the condition changes in every iteration (e.g., if($a[i]$=0)), it becomes complex to optimally parallelize them. Fig. 7 shows the basic working of automatic parallelization compilers to parallelize loops.

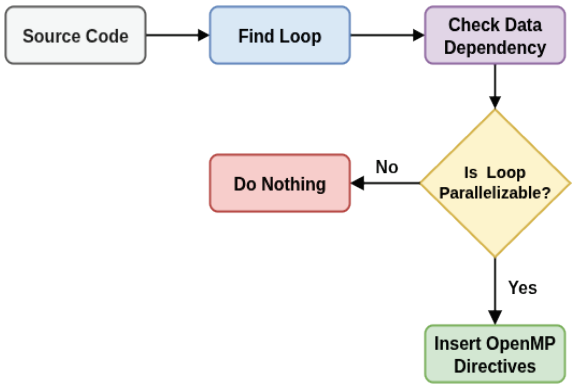


Figure 7 Basic approach of automatic parallelization compilers for parallelizing loops

The proposed framework (named as loopFiss) uses loop-fission transformation to parallelize data-dependency-free loops, even with control statements, for both multi-core and many-core systems. However, it is particularly suited for many-core systems, as HPC workloads typically run on such architectures. In loop-fission, a loop is split into separate loops, as shown in Fig. 8. Independent computations are parallelized, while statements with dependencies, such as data dependencies or control flow constraints, remain sequential, ensuring efficient execution and correctness.

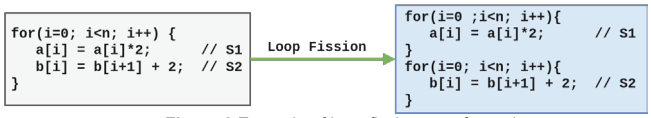


Figure 8 Example of loop-fission transformation

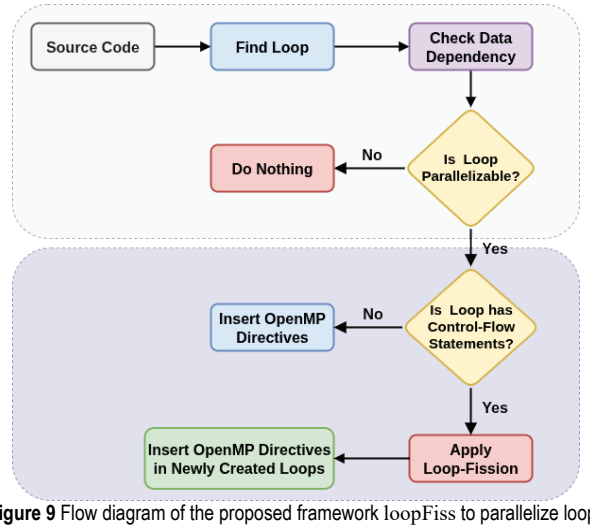


Figure 9 Flow diagram of the proposed framework loopFiss to parallelize loops

The proposed approach checks for dependency-free loops that can be parallelized. Fig. 9 shows the flow diagram of the proposed approach. If the loop has no control flow statements, then OpenMP pragma directives are inserted to parallelize the loop. If a control flow statement is present

inside the loop body, then loop-fission transformation is applied.

For example, to parallelize the loop given in Listing 6, first, it searches for the dependency-free loop with control flow statements.

Listing 6 Syntax of parallelizable loop

```
for(i=0;for_cond;i++){
  if(if_cond){
    body statements containing arrays;
  }
}
```

Then, the loop-fission is applied to the loop to break it into two loops. The first loop stores every array index for which condition is true in a temporary array *trueArr, and* then the second loop uses the index store in *trueArr* to run the statements in parallel. With loop-unrolling transformation, the second loop can use superword-level parallelism to speed-up the processing time.

Algorithm 1 Steps of Proposed Framework loopFiss

```
1:  Create an array trueArr of size N;
2:  Initialize m←0;
3:  while for_cond is true:
4:      if if_cond is true:
5:          trueArr[m]←i;
6:          m←m+1;
7:      end if
8:  end while
9:  Set k ← 0:
10: Apply OpenMP-API to parallelize the loop in Step 11
11: while k< m :
12:     i ← trueArr[k];
13:     Use the value of i as an index for the arrays in body statements;
14: end while
```

Algorithm 1 shows the overall steps of parallelizing the loop in the proposed framework. Listing 7 shows an example of a parallelized loop using this framework. The proposed loop-fission method divides the original loop into two loops. The first loop finds the elements that satisfy the condition. The second loop performs the computations on these elements in parallel. This separation makes it possible to parallelize loops that contain conditional statements.

Listing 7 Code snippet of the output program after applying the loopFiss approach

```
int trueArr[n];
int i,m=0;
#pragma omp parallel for simd private(i)
for(i=0;i<n;i++){
  if(a[i]!=0){
    trueArr[m++]=i;
  }
}

#pragma omp parallel for simd private(z)
for(int z=0;z<m;z++){
  i=trueArr[z];
  a[i]=a[i]+b[i];
}
```

4 EXPERIMENTAL EVALUATION

In this section, the experimental setup and test cases used in the evaluation are described, followed by a discussion of the experimental results. The experiment is conducted on a 72-core Intel® Xeon® processor with 64GB of RAM. For the C-compiler, GCC 9.4.0 is used to compile all test cases as it supports OpenMP 4.5 specifications. The detailed specifications of the hardware and software used for the experiments are shown in Tab. 1.

Table 1 Details of hardware/software specifications used for experiments

Tools	Specification
Ubuntu OS	Ubuntu 20.04.2 (64-bit)
Processor	Intel® Xeon® CPU E5-2699 v3
Clock speed	2.30GHz
Total Cores	72
RAM	64 GB
Cetus	Version 1.4.4
autoPar	ROSE-0.11.46.0
Compiler	GCC 9.4.0

A Python-based generator program is used to generate test cases that contain array-based statements in a loop body. The code snippet shown in Listing 6 is used as the template for test cases in the experiments.

Table 2 Test cases with their description and example of statements used as the loop body

Test Case	Example	Description
AASSIG	$a[i] = a[i] + int_value$	Assign a value to array elements
AADD	$c[i] = a[i] + b[i]$	Addition of two arrays
AMUL	$c[i] = a[i] * b[i]$	Vector Multiplication of two arrays
AOPs:	$b[i] = a[i] + int_value$ $c[i] = a[i] * b[i]$	Multiple Computation with different arrays

Different statements such as array value assignment, array addition, array multiplication and their combinations are used as the body of the control statement (if-statement). Tab. 2 shows the different test cases with descriptions.

The experiments are performed to compare the execution time efficiency of the proposed approach with existing approaches used by autoPar and Cetus using different sizes of arrays for every test case. The arrays of varying sizes from 10K to 200k are used in experiments, and the number of loop iterations in each test case is equal to the size of the array. The test cases are run ten times, and their average value is used as the result value to minimize the error. In addition, during the execution of the test case, no other tasks are assigned to the system to keep the system resources free.

4.1 Result Analysis

Tab. 3 shows the execution times of experiments for different array sizes for test cases *AASSIG* and *AADD*. In *AASSIG*, the assignment operation ($a[i] = a[i] + int_value$) involves arithmetic along with array access. This makes it more computationally intensive than a simple array assignment. In *AADD*, the operation represents element-wise addition of two arrays.

Table 3 Execution time comparison between Cetus, autoPar and loopFiss approach for test cases *AASSIGN* and *AADD*

No. of Iterations	Execution Time (in ms.)					
	AASIGN			AADD		
	Cetus	autoPar	loopFiss	Cetus	autoPar	loopFiss
10000	1.195	1.186	1.201	1.341	1.331	1.346
30000	1.389	1.383	1.394	1.571	1.563	1.571
50000	1.487	1.484	1.468	1.677	1.671	1.650
80000	1.697	1.692	1.663	1.938	1.934	1.892
100000	1.814	1.810	1.769	2.281	2.274	2.214
130000	2.090	2.087	2.032	2.655	2.650	2.565
150000	2.396	2.392	2.321	3.103	3.090	2.982
180000	2.782	2.775	2.679	3.481	3.473	3.331
200000	3.023	3.014	2.901	3.830	3.818	3.639

Results show that for the small number of inputs, the proposed approach takes more time compared to the existing methods. However, as the input size increases, it performs better. From the results, it can be observed that the operation type also affects the execution time. Compared to simple operations like array assignment and addition, a complex operation like multiplication or multiple operations in a single statement take more time.

The proposed approach shows significant improvement in execution time for complex operations, as shown in Fig. 10 and Fig. 11, which show the execution time for test cases *AMUL* and *AOPs*, respectively.

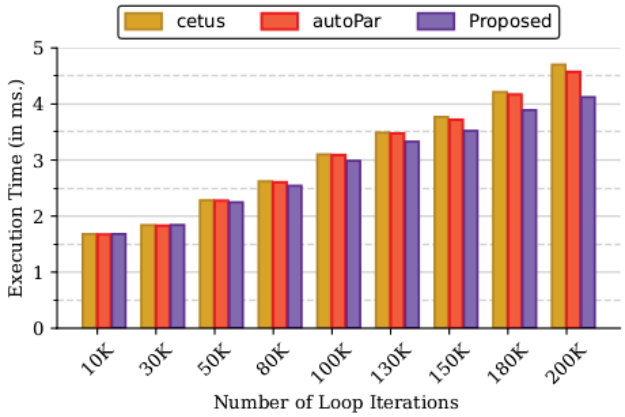


Figure 10 Execution time comparison between Cetus, autoPar and loopFiss for test cases AMUL (1K = 1000)

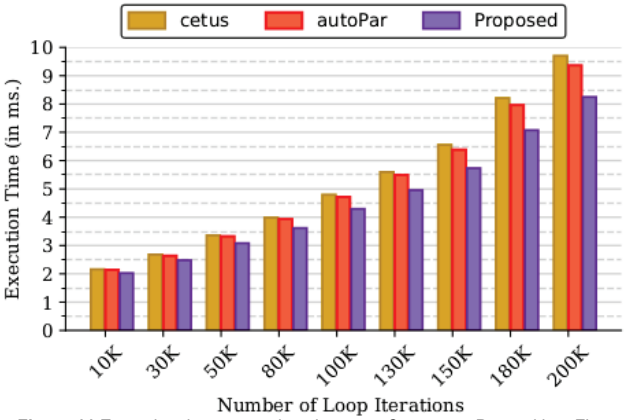


Figure 11 Execution time comparison between Cetus, autoPar and loopFiss Approach for test cases AOPs (1K = 1000)

The speed-up metric is used to determine the performance of methods. The speed-up improvement percentage between the proposed approach and the existing ones is calculated using Eq. (1).

$$Speed\ Up\ (\%) = \left(1 - \frac{ET_{\text{proposed}}}{ET_{\text{existing}}}\right) \cdot 100 \tag{1}$$

Where: ET_{proposed} is the execution time of the proposed approach, and ET_{existing} is the execution time of the existing approach.

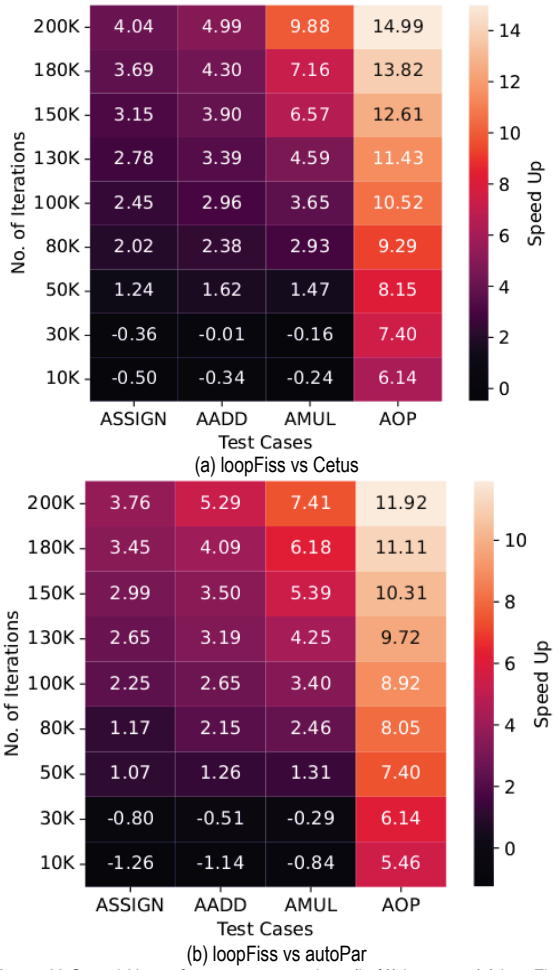


Figure 12 Speed-Up performance comparison (in %) between (a) loopFiss approach vs Cetus, and (b) loopFiss approach vs autoPar

Fig. 12 shows the speed-up differences between the proposed approach and existing approaches for all the test cases. From the results, it can be observed that the proposed approach shows low performance (−ve values) when the input size is small compared to large input sizes, but for large-size inputs, it outperforms the existing methods and shows a significant improvement in execution time.

When the size of the array is small, the proposed approach takes more time compared to existing methods because it breaks the loops and then parallelizes them by creating threads, whereas the existing methods parallelize

only one loop. When performing parallelization, the overhead associated with dividing the work among threads and synchronizing their operations has a significant impact on the overall execution time. For small input sizes, this overhead may outweigh the benefits of parallel execution, resulting in longer execution times when parallelizing two loops compared to parallelization of a single loop.

Fig. 12(a) shows that compared to Cetus, the proposed approach gives a speed-up of 4% to 15% for simple to complex operations. Similarly, compared to autoPar, it shows a speed-up improvement of approximately 3.7% to 12% for the same, as shown in Fig. 12(b). The results analysis indicate that in performing parallelization using OpenMP, simple operations such as array addition exhibit shorter execution times compared to multiplication or more complex computations. This outcome can be attributed to the inherent computational complexity associated with the operation. Additionally, the overhead of parallelization, encompassing thread creation, synchronization, and data communication, has a greater impact on complex operations compared to the simple ones.

5 CONCLUSION AND FUTURE WORK

This work is motivated by the fact that writing a parallel program is a difficult task, specifically for researchers who don't have enough programming knowledge. Therefore, automatic parallelization tools are developed to convert sequential programs into parallel versions to perform computations faster. However, there are still many programming constructs that cannot be efficiently parallelized by these tools. This paper introduced a framework to effectively parallelize data-dependency-free loops with control flow statements using the loop-fission method.

In experiments, various statements like array assignment, addition, multiplication, etc., are used as the body of control flow statements to compare the approaches using execution time as the performance metric. Experimental results show that our approach improves the execution time by 4% to 15% on many-core systems compared to existing tools. These improvements are modest, reflecting the specific test cases and array sizes used in our experiments. To better assess performance over a wider range, future studies will consider larger array sizes that grow more rapidly. Additionally, future work will focus on modifying the approach so that it can be applied to loops containing more than one conditional statement, thereby extending its applicability to more complex scenarios.

6 REFERENCES

[1] Bhaskar, A. V. (2022). A new method of power analysis of network-on-chip using analytical modelling. In *IEEE Seventh International Conference on Parallel, Distributed and Grid Computing (PDGC2022)*, 222–227. https://doi.org/10.1109/PDGC56933.2022.10053136

[2] Lai, Z., Li, S., Tang, X., Ge, K., Liu, W., Duan, Y., Qiao, L., & Li, D. (2023). Merak: An efficient distributed DNN training framework with automated 3D parallelism for giant foundation models. *IEEE Transactions on Parallel and Distributed Systems, 34*(5), 1466–1478. https://doi.org/10.1109/TPDS.2023.3247001

[3] Dozier, J. (2022). Revisiting topographic horizons in the era of big data and parallel computing. *IEEE Geoscience and Remote Sensing Letters, 19*, 1–5. https://doi.org/10.1109/LGRS.2021.3125278

[4] Mohseni, M., Othman, B. A., Raturi, P., Mishra, A. B., Priya, S., & Saravanan, V. (2022). The role of parallel computing towards implementation of enhanced and effective industrial internet of things (IoT) through MANOVA approach. In *IEEE 2nd International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE2022)*, 160–164. https://doi.org/10.1109/ICACITE53722.2022.9823698

[5] Tripathi, A. K., Sharma, K., Bala, M., Kumar, A., Menon, V. G., & Bashir, A. K. (2021). A parallel military-dog-based algorithm for clustering big data in cognitive industrial internet of things. *IEEE Transactions on Industrial Informatics, 17*(3), 2134–2142. https://doi.org/10.1109/TII.2020.2995680

[6] Hoe, M., & Dargham, J. (2020). High performance computing (HPC) applications in industry 4.0 (I4.0) for the betterment of humanity. In *IEEE 8th R10 Humanitarian Technology Conference (R10-HTC2020)*, 1–6. https://doi.org/10.1109/R10-HTC49770.2020.9356990

[7] Groen, M., & Gramoli, V. (2016). Multicore vs manycore: The energy cost of concurrency. In *Euro-Par 2016: Parallel Processing* (Dutot, P.-F., & Trystram, D., Eds.), 545–557. Springer International Publishing. https://doi.org/10.1007/978-3-319-43659-3_40

[8] Geer, D. (2005). Chip makers turn to multicore processors. *Computer, 38*(5), 11–13. https://doi.org/10.1109/MC.2005.160

[9] Rogowski, M., Aseeri, S., Keyes, D., & Dalcin, L. (2023). mpi4py.futures: MPI-based asynchronous task execution for Python. *IEEE Transactions on Parallel and Distributed Systems, 34*(2), 611–622. https://doi.org/10.1109/TPDS.2022.3225481

[10] Wang, X., Yang, R., & Liu, X. (2022). Research on general system of remote sensing satellite data preprocessing based on MPI + CUDA. In *IEEE 3rd International Conference on Geology, Mapping and Remote Sensing (ICGMRS2022)*, 607–612. https://doi.org/10.1109/ICGMRS55602.2022.9849313

[11] Watkinson, N., Shivam, A., Nicolau, A., & Veidenbaum, A. (2019). Teaching parallel computing and dependence analysis with Python. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW2019)*, 320–325. https://doi.org/10.1109/IPDPSW.2019.00061

[12] Mitrovski, J., Djinevski, L., Gusev, M., & Arsenovski, S. (2021). TPC-H benchmark Q3, Q6 and Q12 sequential, OpenMP parallel, and CUDA parallel implementation. In *IEEE 44th International Convention on Information, Communication and Electronic Technology (MIPRO2021)*, 938–943. https://doi.org/10.23919/MIPRO52101.2021.9597197

[13] Arabnejad, H., Bispo, J., Barbosa, J. G., & Cardoso, J. M. P. (2018). An OpenMP-based parallelization compiler for C applications. In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/Sustain Com)*, 915–923. https://doi.org/10.1109/BDCloud.2018.00135

[14] Sah, S., & Vaidya, V. G. (2012). A review of parallelization tools and introduction to EasyPar. *International Journal of Computer Applications, 56*(12). https://doi.org/10.5120/8944-3108

[15] Palkowski, M., & Bielecki, W. (2017). TRACO: Source-to-Source Parallelizing Compiler. *Computing and Informatics, 35*(6), 1277–1306.

[16] Dever, M. (2015). *AutoPar: Automating the parallelization of functional programs* (PhD thesis). Dublin City University.

[17] Dave, C., Bae, H., Min, S.-J., Lee, S., Eigenmann, R., & Midkiff, S. (2009). Cetus: A source-to-source compiler infrastructure for multicores. *Computer, 42*(12), 36–42. https://doi.org/10.1109/MC.2009.385

[18] Arabnejad, H., Bispo, J. A., Cardoso, J. A. M. P., & Barbosa, J. G. (2020). Source-to-source compilation targeting OpenMP-based automatic parallelization of C applications. *Journal of Supercomputing, 76*, 6753–6785. https://doi.org/10.1007/s11227-019-03109-9

[19] Dagum, L., & Menon, R. (1998). OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering, 5*(1), 46–55. https://doi.org/10.1109/99.660313

[20] Prema, S., Jehadeesan, R., & Panigrahi, B. (2017). Identifying pitfalls in automatic parallelization of NAS parallel benchmarks. In *IEEE National Conference on Parallel Computing Technologies (PARCOMPTECH2017)*, 1–6. https://doi.org/10.1109/PARCOMPTECH.2017.8068329

[21] Liang, X., Humos, A. A., & Pei, T. (2017). Vectorization and parallelization of loops in C/C++ code. In *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)* (pp. 203–206). The Steering Committee of the World Congress in Computer Science.

[22] Singh, D., & Garg, R. (2023). Performance analysis of a multicore approach proposed for efficient community detection and recommendation system. *IETE Technical Review*, 0(0), 1–15.

[23] Zhou, H., & Xue, J. (2016). A compiler approach for exploiting partial SIMD parallelism. *ACM Transactions on Architecture and Code Optimization (TACO), 13*(1), 1–26. https://doi.org/10.1145/2886101

[24] Lee, J. et al. (2014). Improving performance of loops on DIAM-based VLIW architectures. SIGPLAN Not. 49, 5 (May 2014), 135–144. https://doi.org/10.1145/2666357.2597825

[25] Devan, P. S., & Kamat, R. (2014). A Review-LOOP Dependence Analysis for Parallelizing Compiler. *International Journal of Computer Science and Information Technologies, 5*(3), 4038–4046.

[26] Barney, B. (n.d.). LLNL HPC Tutorials: OpenMP Programming Model. Retrieved from https://hpc-tutorials.llnl.gov/openmp/programming_model/

[27] Naik, D., Ramesh, D., Gandomi, A. H., & Gorojanam, N. B. (2022). Parallel and distributed paradigms for community detection in social networks: A methodological review. *Expert Systems with Applications, 187*, 115956. https://doi.org/10.1016/j.eswa.2021.115956

[28] Rauber, T., & Rünger, G. (2023). Parallel Programming Models (pp. 109–168). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-031-28924-8

[29] Eichstädt, J., Vymazal, M., Moxey, D., & Peiró, J. (2020). A comparison of the shared-memory parallel programming models OpenMP, OpenACC, and Kokkos in the context of implicit solvers for high-order FEM. *Computer Physics Communications, 255*, 107245. https://doi.org/10.1016/j.cpc.2020.107245

[30] He, G., Vialle, S., & Baboulin, M. (2022). Parallel and accurate K-means algorithm on CPU-GPU architectures for spectral clustering. *Concurrency and Computation: Practice and Experience, 34*(14), e6621. https://doi.org/10.1002/cpe.6621

[31] Liu, X., Lv, H., & Liu, S. (2021). Design and implementation of multi-core parallel compiler based on OpenMP. *Journal of Physics: Conference Series, 1802*, 032137. https://doi.org/10.1088/1742-6596/1802/3/032137

[32] Harel, R., Mosseri, I., Levin, H., Alon, L.-o., Rusanovsky, M., & Oren, G. (2020). Source-to-source parallelization compilers for scientific shared-memory multi-core and accelerated multiprocessing: Analysis, pitfalls, enhancement, and potential. *International Journal of Parallel Programming, 48*(1), 1–31. https://doi.org/10.1007/s10766-019-00640-3

[33] Quinlan, D. (2000). Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters, 10*(02n03), 215–226. https://doi.org/10.1142/S0129626400000214

[34] Lee, S.-I., Johnson, T. A., & Eigenmann, R. (2003). Cetus: An extensible compiler infrastructure for source-to-source transformation. In *International Workshop on Languages and Compilers for Parallel Computing* (pp. 539–553). Springer. https://doi.org/10.1007/978-3-540-24644-2_35

Authors' contacts:

Akash Yadav, Dr.
(Corresponding author)
GLA University,
Mathura, UP- 281406, India
Malaviya National Institue of Technology,
Jaipur, Rajasthan-302017, India
akash.yadav@gla.ac.in, 2019rcp9150@mnit.ac.in

Mushtaq Ahmed, Dr.
Malaviya National Institue of Technology,
Jaipur, Rajasthan-302017, India
mahmed.cse@mnit.ac.in