ARACHNE: Optimizing Distributed Parallel Applications with Reduced Inter-Process Communication

YIFU HE, Shanghai Jiao Tong University, Shanghai, China
HAN ZHAO, Shanghai Jiao Tong University, Shanghai, China
WEIHAO CUI, Shanghai Jiao Tong University, Shanghai, China
SHULAI ZHANG, Shanghai Jiao Tong University, Shanghai, China
QUAN CHEN, Department of Computer Science, Shanghai Jiao Tong University, Shanghai, China
MINYI GUO, Computer Science, Shanghai Jiao Tong University, Shanghai, China

In high-performance computing (HPC), parallelization is essential for improving computational efficiency as data and computation scales exceed single-node capacity. Existing methods, such as the polyhedral model used in Pluto-Distmem, focus on loop and array optimizations within shared memory but struggle with high communication overheads and inflexibility in distributed environments. These methods often fail to effectively partition computation and manage data across nodes, leading to suboptimal performance.

This paper presents Arachne, an innovative system designed to address these shortcomings by generating distributed parallel code with minimized communication overhead. The system introduces a dynamic programming algorithm to optimally distribute computational tasks across multiple processes, ensuring minimal communication costs. It also incorporates user-friendly compiler directives, allowing programmers to influence code generation easily and accommodate a broader range of parallelization scenarios without needing in-depth knowledge of parallel architectures. Arachne significantly reduces the learning curve and need for extensive code modifications, making parallel programming more accessible and efficient. Evaluation of various HPC benchmarks demonstrates that Arachne outperforms existing methods by reducing communication overhead, lowering memory requirements, and supporting more complex parallel logic, thus enhancing the overall scalability and efficiency of HPC applications.

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**.

Additional Key Words and Phrases: High-performance computing, Parallel code generation, Compiler directives, Code optimization

1 Introduction

High-Performance Computing (HPC) increasingly relies on distributed parallelism to handle the growing scale of computation and data[23]. While shared memory models leverage single-node parallelism, large-scale applications must utilize distributed memory systems, where explicit data exchange between processes is required[34]. This shifts the burden of coordination and communication management to the programmer.

Transforming serial code into distributed parallel code involves addressing three main **challenges**: **(1)partitioning computations** for data parallelism, **(2)distributing data** among processes, and **(3)managing parallel**

Authors' Contact Information: Yifu He, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: yifu.ho@sjtu.edu.cn; Han Zhao, Shanghai Jiao Tong University, Shanghai, China; e-mail: zhao-han@cs.sjtu.edu.cn; Weihao Cui, Shanghai Jiao Tong University, Shanghai, China; e-mail: weihao@sjtu.edu.cn; Shulai Zhang, Shanghai Jiao Tong University, Shanghai, China; e-mail: zslzsl1998@sjtu.edu.cn; Quan Chen, Department of Computer Science, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: chen-quan@cs.sjtu.edu.cn; Minyi Guo, Computer Science, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: guo-my@cs.sjtu.edu.cn.

ACM 1544-3973/2025/2-ART
https://doi.org/10.1145/3716871

logic and communication. This process poses significant coding and debugging difficulties, especially for users who need to extensively understand and rewrite unfamiliar code.

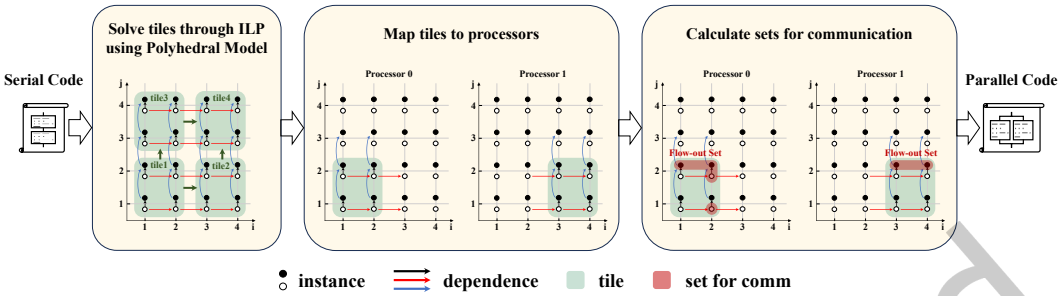


Fig. 1. **The process of PLUTO-Distmem.**

The state-of-the-art approach for parallelizing serial code in distributed environments uses the polyhedral model to identify sets of loop iterations that can be parallelized, called tiles[15]. These tiles are then mapped onto processes in a distributed environment, managing inter-tile communication to generate distributed code. The latest implementation of this approach is represented by Pluto-Distmem[14], the process of which shown in Figure 1.

However, Pluto-Distmem fails to adequately address the aforementioned challenges. This is not an isolated case. In the ADI Benchmark, while Pluto-Distmem can identify a data-parallel scheme, the tiles it generates lead to excessively high communication overhead. In a 32-process configuration, it generates **7742.25 GB** of inter-process communication, whereas the optimal distributed implementation requires only **0.25 GB**.

The root cause lies in the polyhedral model, which is designed for shared-memory environments[14], and when applied to distributed environments, it exhibits two fundamental **flaws**.

a.The model derives an affine transformation for loop dependencies, determined through an implicit loop tiling process within an integer linear programming solution. It imposes a stringent constraint: tiles must function as atomic computations, meaning all loop iteration instances within a tile must be computed before moving to the next one.

b.When multiple loops are combined into an overall linear programming problem that lacks a solution, the polyhedral model independently solves linear programming for each loop. Consequently, it fails to jointly analyze communication costs between these loops[7], resulting in significant communication overhead when switching partitioning patterns among them. While this ensures optimality for individual loops, the polyhedral model cannot find a unified solution for all loops combined.

Additionally, Pluto-Distmem struggles with memory constraints and adding parallel logic. For ADI, the original serial version requires **392MB** of memory per process, while the Pluto-generated version needs **662MB**. When distributing data due to limited memory on a single node, Pluto-Distmem is ineffective. It also cannot generate distributed reduction code for ADI, instead transferring data from all processes to a single node for serial reduction.

The **limitations** of Pluto-Distmem stem from the polyhedral model's exclusive focus on affine array accesses within nested loops. During inter-tile communication and write-back, Pluto-Distmem requires each process to write updates from other processes to the same array location, forcing every process to retain the entire array structure. Additionally, the model treats other parallelizable logic, such as reduction, as serial due to dependencies between instances.

We can find optimization opportunities for polyhedral's **flaws** and **limitations**, corresponding to the three **challenges**.

(1) For **computational partitioning**, **a.**we treat each loop iteration instance as the atomic parallel granularity, avoiding the enforced constraints of the polyhedral model and enabling finer-grained parallelism. As long as instances from the same iteration range across different loops are placed within the same partition, the communication volume between different partitions is necessary and minimal. **b.**We propose a dynamic programming[17] algorithm to minimize global communication costs across multiple loops, switching partitioning patterns only when necessary.

(2) For **data distribution**, after mapping computational partitions to different processes, each process accesses only the relevant portions of the arrays. We allocate new array spaces based on the partitions and use additional dimensions to record the indices of the array blocks handled by each process. By mapping the memory access operations on the complete arrays in the original code to the newly partitioned arrays, we achieve data distribution and reduce the memory footprint per process.

(3) For **parallel logic**, such as reduction operations, static analysis alone is insufficient without user involvement. To minimize user intervention, we devised a set of compiler directives that enable users to insert directives into the original serial code, facilitating the generation of distributed parallel code.

In this paper, we propose Arachne, a system aimed at generating distributed parallel code with minimal communication overhead. It comprises a set of user-facing compiler directives and backend components including a static dependency analyzer, computational partitioning searcher, and code generator. It enables users to generate distributed parallel code with minimal learning cost and avoid invasive modifications to the code. Users insert compiler directives at appropriate locations in the original serial code. Arachne performs comprehensive static dependency analysis on the code, extracting all loop structures and read-write information within parallel regions. The computational partitioning searcher utilizes a dynamic programming algorithm to search for parallel schemes with globally minimal communication overhead. After a series of optimizations, the code generator applies the searched computational partitioning and data distribution to the original code and adds corresponding parallel logic based on user-provided directives. Finally, it generates distributed parallel code.

Arachne effectively handles array accesses using loop indices, but its limitations arise when dealing with irregular array accesses and complex affine patterns. Static dependency analysis cannot handle runtime-dependent access patterns such as variables, pointers, or array elements as indices. Moreover, when complex affine patterns create dependencies across multiple dimensions, the searcher may fail to find a partition scheme suiting multiple loops, leading to communication costs equivalent to the all-to-all communication seen in Pluto-Distmem's outcome.

To evaluate our approach, we implemented Arachne using LLVM pass and Clang tool and used the MPI library as the communication backend. We evaluated Arachne on multiple common HPC benchmarks. Experimental results demonstrate that compared to Pluto-Distmem, Arachne achieves lower communication overhead, reduces the memory required by processes, and supports more parallel logic.

Our contributions mainly include the following:

- We identify existing issues in current distributed parallelization methods, rooted in coarse-grained parallelism and unawareness of dependencies among multiple independent loops.
- We propose a dynamic programming search algorithm across multiple loops, leading to a parallel scheme with the minimum communication overhead for the entire program.
- We design a set of compiler directives to assist users in controlling the generation of distributed parallel code more precisely.

2 Background and Motivation

Distributed memory models introduce additional complexity to user programming, necessitating explicit arrangement by the user. Transforming serial code into distributed parallel code entails addressing three crucial

challenges: (1)computational partitioning for data parallelism, (2)data distribution among processes, and (3)the communication and additional parallel logic necessitated by the above two.

2.1 Background

Current state-of-the-art research in automatic parallelization, targeting nested loops and array accesses, resides within the polyhedral model. However, the parallel objectives of the polyhedral model are primarily tailored for shared memory environments with multi-core architectures[14]. The latest advancement, Pluto-Distmem, extends the applicability of the polyhedral model to distributed memory environments. By employing the polyhedral model, Pluto-Distmem derives parallelization solutions, maps them to distributed environments, computes the data sets requiring inter-process communication, and ultimately generates end-to-end distributed code.

2.1.1 Polyhedral Represent and Dependence. In the polyhedral model, a k-layered nested loop forms a k-dimensional iteration space through its outer-to-inner loop indices. Each loop statement generates an instance during each loop iteration. The k loop indices of each instance form a vector, and the set of these vectors forms the iteration domain of the statement, appearing as a polyhedron.

Polyhedral models also represent all dependencies between instances through dependence polyhedron. If two instances have data dependencies (RAW, WAR, WAW), an arrow points from the first to the second. Polyhedral models further abstract these dependencies into directed graphs called Program Dependence Graphs (PDGs). If an $S2$ instance depends on an $S1$ instance, there exists an edge from $S1$ to $S2$ in the PDG[22].

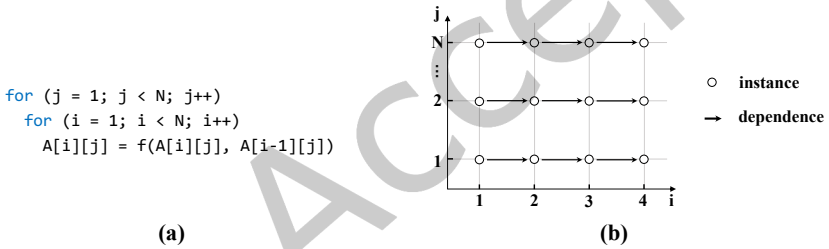


Fig. 2. **Code with sync-free parallelism and its dependence.**

2.1.2 Method of Polyhedral Model. The polyhedral model finds sync-free parallelism and pipeline parallelism in loops by solving integer linear programming (ILP)[30]. Pipeline parallelism is the core means used by the polyhedral model to achieve parallelization.

For nested loops where there is no dependence in one dimension, the polyhedral model can solve an ILP to obtain an affine transformation that maps the iteration domain to the processor domain, generating a new outer loop indexed by processors, which can run in parallel without any synchronization overhead[18]. For the code in Figure 2(a), the dependence graph in Figure 2(b) reveals that all computations have no data dependence along the j direction, while each instance depends on the previous instance along the i direction. The polyhedral model maps the j-dimensional computations to the processors.

Most programs lack sync-free parallelism and require synchronization to expose parallelism. The polyhedral model can achieve parallelism through pipelining for certain loops: loops with dependencies in each dimension, whose iterations can execute with a fixed delay relative to the iterations they depend on[15]. This is visualized by finding a hyperplane where all dependence directions in the iteration space have positive components relative to the hyperplane's normal vector. This implies a pipeline direction, enabling concurrent execution of instances on the same hyperplane.

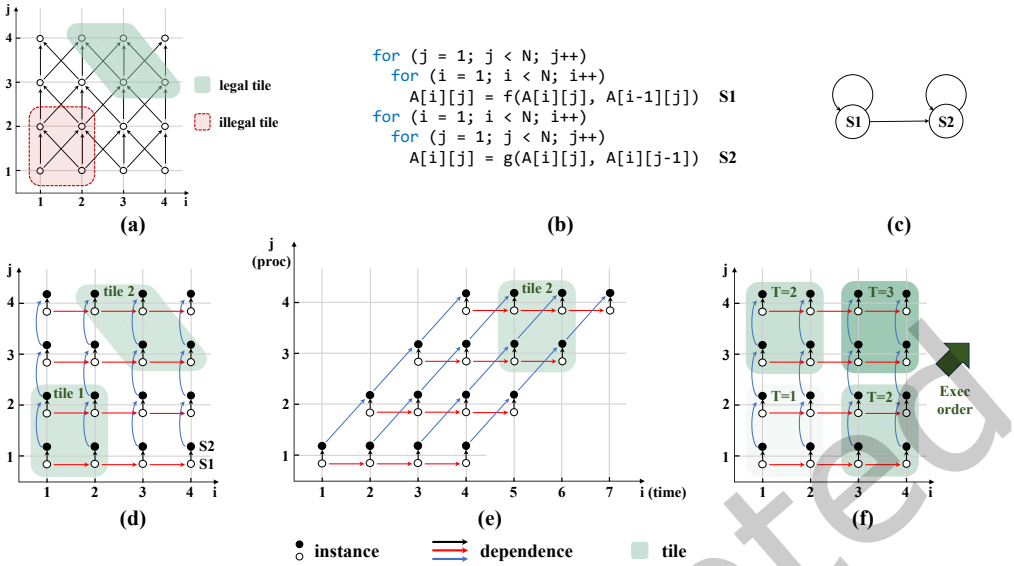


Fig. 3. **Code with pipeline parallelism and tile selection.**

In these cases, the polyhedral model solves an ILP to obtain another affine transformation. The solution suggests a loop tiling scheme, determining the tile's shape, and the affine transformation maps the tile edges perpendicular to the axes. The polyhedral model then determines the parallel computation order based on data dependencies between tiles. In linear programming representing pipelined parallelism, there is an additional constraint: each tile is **atomic**, meaning all instances in a tile are computed before moving to the next tile. If tiles cannot be executed as atoms, a mutual dependency arises, violating pipelined parallelism[30]. Figure 3(a) shows the shapes of a legal tile and an illegal tile in such dependencies.

For example, in Figure 3, (d) demonstrates two tiling methods for the code in (b), corresponding to two ILP solutions. Tile 1's edges are perpendicular to the axes, and the dependence graph after the affine transformation is shown in (f), while the tiling method of Tile 2 corresponds to the affine transformation shown in (e). All instances within a tile can execute without additional dependencies. (f) shows the computation order corresponding to Tile 1's tiling method, where the diagonal represents the pipeline direction, and all tiles on the same plane execute concurrently in each time step.

If loops have more complex dependencies, making it impossible to solve an ILP that satisfies these parallelism constraints for the entire program, the polyhedral model attempts to fission the loops and continues to seek parallelism opportunities in each part. In the PDG, a unidirectional arrow between nodes indicates that all instances of one statement can execute entirely before those of another, allowing such loops to be split into two independent sub-loops. The polyhedral model finds each strongly connected component in the PDG and seeks the two aforementioned parallelism opportunities in each independent component. Strongly connected components have only unidirectional arrows between them, requiring synchronization to ensure correct parallel execution.

The PDG of the code shown in Figure 3(b) corresponds to the dependence graph in (c). $S1$ and $S2$ instances depend on their previous instances along j and i directions, respectively, resulting in loops pointing to themselves. Furthermore, each $S2$ instance depends on an $S1$ instance, creating a unidirectional arrow from $S1$ to $S2$. Thus, parallelism can be sought independently for $S1$ and $S2$, with synchronization added between them to ensure correctness.

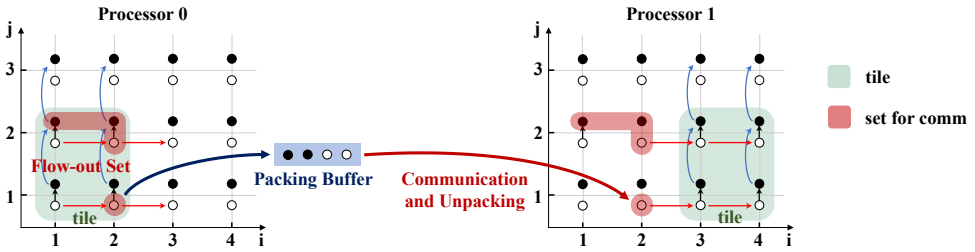


Fig. 4. **Inter-process communication in Pluto-Distmem.**

2.1.3 Polyhedral for Distributed Memory. Pluto-Distmem explores the application of the polyhedral model to distributed memory systems, achieving an automated end-to-end system capable of transforming serial code into distributed parallel MPI code. Building on the ILP within the polyhedral model, Pluto-Distmem simultaneously determines tiling dimensions, computation order, loop structure, and hierarchy.

Within the parallel dimensions derived from the polyhedral model, Pluto-Distmem assigns tiles to different processes and computes the elements required for communication between each tile. As in Figure 4, it employs Flow-out Sets to represent data needed by other processes after being written, and Write-out Sets to signify data aggregation to a single node upon completing the parallel computation region. After executing a tile's instances, Pluto-Distmem packages the necessary data sets into buffer and sends them via MPI to the respective processes. Upon reception, each process unpacks the data and writes it back to the original data location[16, 37].

2.2 Problems

However, Pluto-Distmem fails to effectively address the three challenges, due to its deficiencies in distributed environments and its limited scope of application.

2.2.1 Deficiencies in Distributed Environments. The polyhedral model's parallelization objectives are designed for shared-memory systems, and applying it to distributed environments reveals two fundamental flaws: (1)the constraint for atomic tiles and (2)the inability to jointly analyze multiple independently processed loops.

Pipeline parallelism is the most prevalent application scenario for the polyhedral model. As discussed in Section 2.1.2, when seeking such parallelism, there exists a constraint that tiles must be atomic. When considering multiple loops with complex dependencies, the constraint of atomic tiles restricts the polyhedral model to achieving only coarse-grained parallelism at the tile level, resulting in a limited solution space. Since it is unable to find a unified tile partition for multiple loops, the polyhedral model performs loop fission and independently solves the ILP for each loop. This approach prevents the polyhedral model from jointly analyzing the dependencies and communication between these independent loops.

In shared-memory environments, these two flaws typically do not affect parallelization, as there is no data inconsistency among threads. However, in distributed environments, maintaining data consistency requires inter-process communication between loops, resulting in substantial communication overhead in the parallel schemes derived by the polyhedral model.

Taking the ADI code in Figure 5(a) as an example, this represents a common computational pattern in HPC applications. Due to reverse dependencies, the dependency chains between iterations form a circular pattern (as bold arrows in Figure 5(b)) in the iteration space. A dependency chain implies that all instances must be executed in the order defined by the original code, and the circular pattern indicates that tiling along this dimension is illegal for the polyhedral model.

The polyhedral model fails to identify a pipeline direction for all loops in Figure 5(a), and instead resorts to loop fission to explore independent parallelization. $S1$ and $S2$ exhibit sync-free parallelism along the j dimension,

while $S3$ and $S4$ do so along the i dimension, leading the polyhedral model to add a synchronization point between them for independent parallel execution, as in Figure 5(c). In shared memory, synchronization requires a barrier for all threads, while in distributed memory, it involves all-to-all communication between processes, causing significantly higher overhead than on-demand communication.

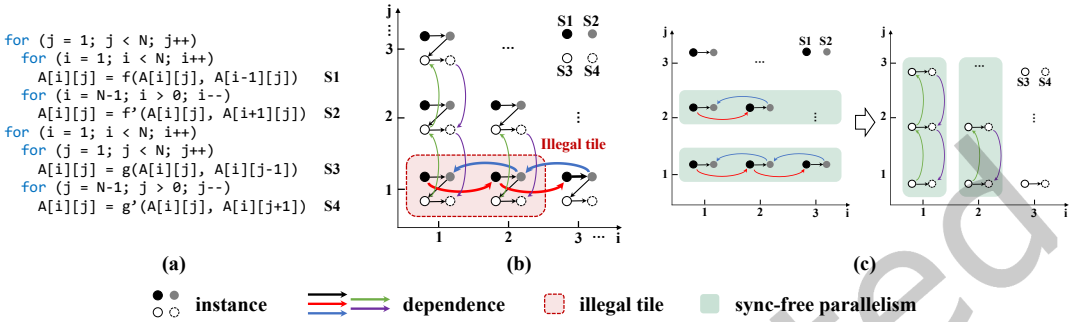


Fig. 5. **Polyhedral model's handling of codes with complex dependencies.**

2.2.2 Limitations of Use Cases. The polyhedral model's limitations in managing data distribution and process execution make it unsuitable for large-scale distributed computations requiring complex parallel patterns[7].

The polyhedral model focuses on affine accesses to arrays within nested loops, optimizing loop parallelism and data locality. It requires each process to hold a complete definition of all array structures to write data updates from other processes at the corresponding positions, even if each process only accesses specific portions of the array. This becomes impractical for large-scale data partitioning across multiple processes to accommodate memory limitations.

Furthermore, the polyhedral model is limited in supporting other parallel patterns in distributed environments, such as array reductions. It always synchronizes by collecting data from all processes to a single node, where the reduction is performed serially. Additionally, the polyhedral model lacks effective strategies for handling local arrays defined within loops in distributed scenarios, such as promoting dimensions to facilitate inter-process communication.

2.3 Opportunities

In response to the issues found within the polyhedral model, we can identify opportunities for resolution.

2.3.1 Fine-Grained Parallelism with Overall Communication. The two core flaws of the polyhedral model can be addressed by adopting finer-grained partitioning and optimizing communication strategies.

Observing the tile partition graph within the polyhedral model, we note that on determined dimensions, any tile partition containing the same number of instances generates equivalent communication requirements between tiles. As long as multiple loop instances with the same iteration range can be in the same calculation partition, the communication volume between this calculation partition and other partitions is necessary and minimal. In Figure 6, the partitions of each stage only require communication to satisfy dependency arrows with adjacent partitions.

By considering each instance as the atomic parallel granularity instead of tiles, we avoid the enforced constraints of the polyhedral model and jointly analyze communication costs across multiple loops. As depicted by the dependency arrows that span different statements in Figure 6, the partitioning corresponding to the four statements belongs to the same process, inter-process communication via all-to-all is not required. This approach addresses the second flaw of the polyhedral model.

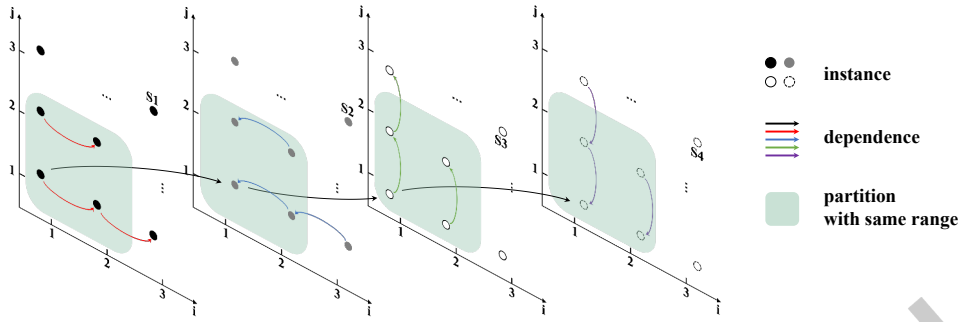


Fig. 6. **Partition with instance granularity in multiple loops.**

2.3.2 User-Assisted Code Generation. The limitations of Pluto-Distmem in fully automating distributed parallel code generation, particularly for data distribution and complex parallel patterns, necessitate user intervention to achieve optimal results.

Data partitioning can be automatically handled through static analysis while considering fine-grained parallelism. For local arrays, dimension promotion can eliminate dependencies between instances, achieving parallelism and meeting inter-process communication requirements. Moreover, by introducing additional dimensions to record partition indices, memory accesses to the complete array in the original code can be mapped to different blocks of the partitioned array, facilitating data distribution and reducing the memory footprint of each process.

We aim to facilitate the generation of parallel logic with few user interventions. To minimize user learning costs and mitigate invasive code modifications, we propose a set of compiler directives designed for insertion at appropriate locations within the original serial code to assist code generation.

3 Overview

To address the aforementioned challenges, we propose Arachne, which combines user-facing compiler directives with backend components. By adding a small number of auxiliary directives to serial code, Arachne enables the end-to-end generation of distributed parallel MPI code, simplifying the programming process. The backend consists of a static dependency analyzer, a computational partitioning searcher, and a code generator.

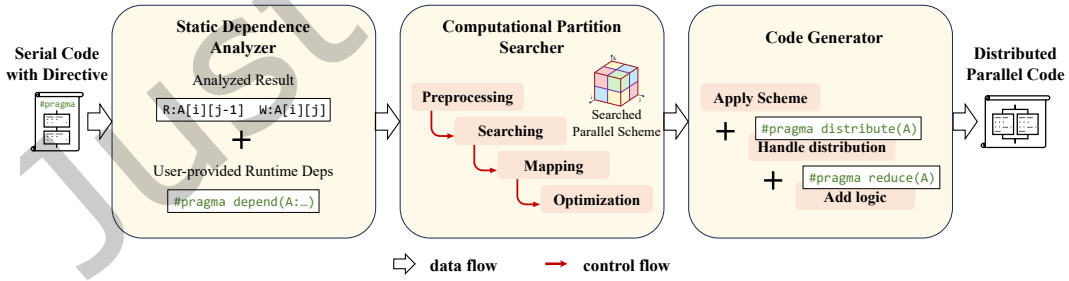


Fig. 7. **Overview of Arachne.**

As shown in Figure 7, the static dependency **analyzer** examines the original serial code, extracting loop structures and array read/write access information, and provides this data, along with user-specified runtime dependencies, to the searcher. The computational partitioning **searcher**, the core of Arachne, operates in four steps: preprocessing, searching, mapping, and optimization. It preprocesses the code using static analysis results to extract parallelizable regions, promote local arrays, and fission loops for finer-grained parallelism. Next, it

searches for a parallel scheme that minimizes communication overhead, potentially selecting from multiple partitioning patterns. The mapping step assigns each computational partition to different processes. Finally, the optimization step employs loop expansion to minimize communication where possible. The code **generator** applies the parallel scheme to the code, manages data distribution and index remapping for user-specified arrays, and integrates parallel logic based on user directives. The final output is distributed parallel MPI code.

4 Distributed Memory Code Generation

This section describes the backend design and workflow of Arachne, considering instance granularity and overall communication cost.

4.1 Dependence and Parallelization

We use a portion of ADI code augmented with initialization logic as an example to illustrate the different parallelization opportunities arising from two types of data dependencies. Unlike Pluto-Distmem's coarse-grained analysis, which targets tiles, our analysis is performed at a fine-grained instance level. Figure 8(a) depicts the dependency graph and the PDG.

Since computational partitioning essentially involves dividing the loop range, a line perpendicular to the i-dimension can be drawn on the dependency graph to represent partitioning along the i-dimension. All dependency arrows intersecting this line correspond to data that requires synchronization between partitions through communication, which can be categorized into the following two types: **(1)**dependencies between different instances of the same statement, and **(2)**dependencies between two different statements. The first type appears in the PDG as self-loops (as red arrows), while the second type is represented as unidirectional arrows between two statements (as blue arrows).

The first type establishes a clear execution order among instances of a statement, where each subsequent instance depends on the preceding one, thereby creating a dependency chain. Consequently, one partition must wait for the completion of the previous one, which can lead to process idle time if these partitions are assigned to different processes. Properly allocating computational partitions across other dimensions can maximize process utilization.

For the second type, as long as there is a unidirectional dependency between two statements, either completing all instances within a single partition first (corresponding to loop fusion, i.e., using tiles as the minimal computational unit) or executing instances of one statement across multiple partitions first (corresponding to loop fission) can ensure correctness. This flexibility allows using instances as computational partition units to expose greater parallelism.

4.2 Optimal Partition Searching

4.2.1 Preprocessing. The dependency analysis is not a novel contribution of this work, but it is essential for extracting loop structures and data read-write states. Based on these results, we perform preprocessing to expose additional parallelism by extracting potential parallel regions, handling local arrays within loops, and applying loop fission.

To avoid the influence of non-accessing loop indices on access analysis, we represent the loop hierarchy using a tree structure called a loop tree. Each node in the loop tree encompasses all access information within that loop level and its nested sub-loops. We identify potential parallel regions by conducting a breadth-first traversal of the entire loop tree. This process filters out nodes where the loop index does not affect array accesses, thereby simplifying the loop tree to focus solely on the components relevant to parallelization.

For the nested loops represented by the simplified loop tree, we search for a reference array, each dimension of which is accessed by loop variables at each loop level. Based on the access patterns to this array, we standardize

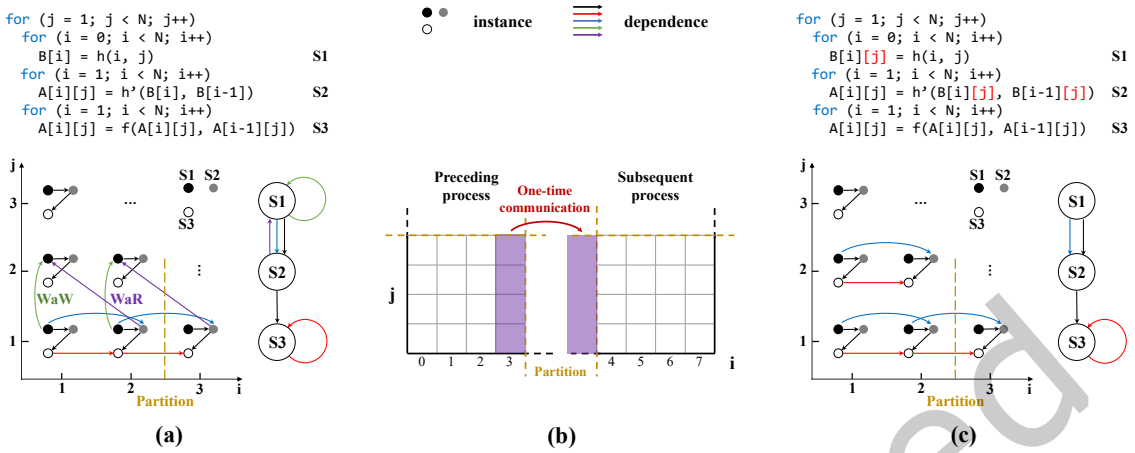


Fig. 8. **Handling of local arrays.**

loop variables that exhibit identical memory access behaviors across different loops, ensuring consistent treatment of distinct variables, such as i and k accessing the first dimension of A in $S5$ and $S6$ from Figure 9(a).

When reference array is present, local arrays with fewer dimensions than the reference array may impact the code parallelizability. As discussed in Section 2.3, the optimal parallelization of the complete ADI code involves partitioning both the j and i dimensions. For demonstration purposes, we continue to use the portion of ADI code in Figure 8(a). The local array B defined within the i-dimensional loop (whether inside or outside the loop body) introduces multiple dependencies on the same position within B across different j iterations, including WAR and WAW dependencies. These dependencies form a loop from $S1$ to itself and bidirectional arrows between $S1$ and $S2$, indicating complete non-parallelizability along the j dimension.

Privatizing array B for each iteration j can eliminate dependencies described above, thereby enabling parallelization. However, this approach introduces additional inter-process communication overhead in distributed environments. As shown in Figure 8(b), the preceding process must send all $B[3]$ from different j instance to the subsequent process, as the latter requires $B[3]$ to compute its first $i = 4$ iteration. Although it is possible to perform one communication for each j instance, frequent communication introduces more overhead from starting each. Additionally, because array B is privatized, the two processes must synchronize their computations for the same j instance, leading to additional waiting overhead.

By performing Array Promotion as depicted in Figure 8(c), we move the definition of local arrays out of the loop body and promote their dimensions to accommodate the loop levels. This operation eliminates data dependencies along the j dimension on the dependency graph, allowing processes to compute all instances of $S1$ before $S2$ without waiting. Simultaneously, it provides the necessary array space to store the process data that requires communication. The final one-time communication avoids the overhead associated with frequent communication initiations. The additional memory footprint introduced by Array Promotion will be further discussed and addressed in Section 4.3.

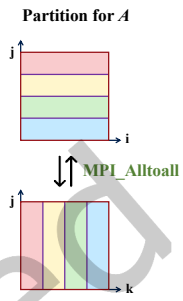
After exposing more of the code's parallelism, we greedily apply loop fission to the output loop tree generated by the previous steps, decomposing nested loops into independent loops as much as possible. The objective of loop fission is to independently analyze the parallelism within each loop, search for computational partitions, and insert communication primitives with finer granularity.

Figure 10(a) illustrates the preprocessing applied to the complete ADI code. It extracts potential parallel regions excluding the T-loop, standardizes loop variables based on the reference array A, promotes the dimensions of the local array B, and performs loop fission wherever possible, as each statement can be fissioned after B's promotion.

```
for (t = 0; t < T; t++)
  for (j = 1; j < N; j++)
    for (i = 0; i < N; i++)
      B[i] = h(i, j)                    S1
    for (i = 1; i < N; i++)
      A[i][j] = h'(B[i], B[i-1])        S2
    for (i = 2; i < N; i++)
      A[i][j] = f(A[i][j], A[i-1][j])   S3
    for (i = N-1; i > 0; i--)
      A[i][j] = f'(A[i][j], A[i+1][j])  S4
  for (k = 1; k < N; i++)
    for (j = 2; j < N; j++)
      A[k][j] = g(A[k][j], A[k][j-1])   S5
    for (j = N-1; j > 0; j--)
      A[k][j] = g'(A[k][j], A[k][j+1])  S6
```

(a) Original Code of complete ADI

```
for (t = 0; t < T; t++)
  for (j = j_start; j < j_end; j++)
    for (i = 0; i < N; i++)
      B[i] = h(i, j)                    S1
    for (i = 1; i < N; i++)
      A[i][j] = h'(B[i], B[i-1])        S2
    for (i = 2; i < N; i++)
      A[i][j] = f(A[i][j], A[i-1][j])   S3
    for (i = N-1; i > 0; i--)
      A[i][j] = f'(A[i][j], A[i+1][j])  S4
  MPI_Alltoall(A)
  for (k = k_start; k < k_end; k++)
    for (j = 2; j < N; j++)
      A[k][j] = g(A[k][j], A[k][j-1])   S5
    for (j = N-1; j > 0; j--)
      A[k][j] = g'(A[k][j], A[k][j+1])  S6
```



(b) PLUTO-Distmem Method

Fig. 9. **Complete ADI code and Pluto-Distmem Method.**

4.2.2 Searching. From the previous analysis, we draw two conclusions. **(1)** The communication cost induced by any computational partition, regardless of the partition shape, remains constant along specific dimensions. As illustrated in Figure 3(d), two different partition shapes drawn on the two-dimensional plane result in the same arrows, which represent the communication requirements. The critical factor determining the communication is whether each dimension has been partitioned. **(2)** If a dimension is partitioned in a subsequent loop but was not partitioned in a previous loop, or vice versa, all processes must synchronize the updated data from the previous loop through all-to-all communication to maintain consistency. In Figure 5(c) and Figure 9(b), Pluto-Distmem's parallelization of different dimensions across the two loops in ADI results in an all-to-all communication of A.

To search a parallel scheme that minimizes the overall communication cost across the entire code, we derive a subtree for each array from the loop tree obtained in Section 4.2.1. Each node within these subtrees corresponds to a read or write operation on the respective array. We further abstract the analysis by assigning a flag to each loop index within the subtree, representing the data dependency between the array dimension and the loop dimension: **0** indicates direct data access in this dimension, demonstrating synchronization-free parallelism; **1** represents cross-iteration access that can satisfy dependency requirements through inter-process communication; **2** denotes serial code or complete access to the array along this dimension, preventing parallelism; **-1** signifies no data access for the array along this dimension. From each array x, we abstract a $M \times N$ flag matrix $\mathbf{F}_x$ from its subtree, where M is the number of fissioned nested loops, and N is the maximum depth of the loop tree, corresponding to each loop level.

The searching process for a partitioning scheme can be abstracted as a dynamic programming problem, aiming to find a path through the two-dimensional flag matrix, as detailed in Algorithm 1. Specifically, if there exists a dimension without cross-iteration dependencies in all flag matrices, partitioning the computation along this dimension will incur no communication cost (lines 4-7).

In the general case, we traverse the matrix to find the overall optimal scheme. We define a two-dimensional matrix $\mathbf{opt}_x$ of the same size as F_x, where $opt(i, j)$ represents the optimal solution with its communication cost from the matrix's starting point $(0, 0)$ to the current point (i, j). The first row and first column of opt serve as boundary conditions, with their optimal values initialized accordingly (line 8). For each point (i, j) in the matrix beyond these boundaries, the optimal solution is derived from the optimal solutions of the point above $(i - 1, j)$

Algorithm 1 Search for Partition Scheme with Minimum Communication

1: **Input:** An M*N flag matrix F generated from the subtree for each array
2: **Output:** A specific partitioning scheme
3: **for** each dimension N **do**
4: **for** each matrix F **do**
5: **if** all flags in a dimension are 0 **then**
6: Partition in this dimension without communication
7: Exit algorithm
8: **for** each matrix F **do** Initialize first row and first column
9: **for** each matrix F **do**
10: **for** each row **do**
11: **for** each point (i, j) in the row **do**
12: **if** $opt(i-1, j)$ includes dimension j **and** $F(i, j)$ fits $opt(i-1, j)$ **then**
13: $opt(i, j) = opt(i-1, j)$
14: **else**
15: **if** $opt(i, j-1)$ can be extended by adding partitioning in dimension j **then**
16: $opt(i, j) = opt(i, j-1) - merge\_cost(i, j)$
17: **else**
18: Choose lowest cost between left $opt(i, j-1)$ and upper $opt(i-1, j)$ connected with all-to-all communication
19: Aggregate partitioning patterns for each $opt(i, N-1) = \bigcap opt_x(i, N-1)$

and the point to the left $(i, j-1)$. The state transition equation, which represents the communication cost, is defined as follows (lines 9-18).

$$\text{opt}(i,j) = \begin{cases} \text{opt}(i-1,j), & \text{if } (i,j) \text{ fits upper;} \\ \text{opt}(i,j-1) - \text{merge\_cost}(i,j), & \text{if } (i,j) \text{ fits left;} \\ \min\{\text{opt}(i-1,j) + \text{alltoall\_cost}(i-1,i), \text{opt}(i,j-1)\}, & \text{else.} \end{cases}$$

We define **merge_cost(i,j)** as the difference in communication cost when adding the j-dimension of point (i, j) to the partitioning scheme recorded in $opt(i, j-1)$. As further explained in Section 4.2.3 with Figure 11(a) and (b), partitioning along three dimensions reduces the communication cost compared to partitioning along two dimensions. As stated in Conclusion (1), only the choice of dimensions affects the cost. **alltoall_cost(i-1,i)** is defined as the communication cost incurred when synchronizing data between processes via all-to-all communication between the (*i-1*)-th and *i*-th loops. This operation implies that the two loops adopt different partitioning patterns, as outlined in Conclusion (2), where consecutive rows using the same partitioning pattern exhibit locally optimal communication cost.

After completing the traversal of each row, we take the intersection of the optimal solutions $opt(i, N-1)$ for each matrix, aggregating the partitioning patterns for the loop. Ultimately, $opt(M-1, N-1)$ records the partitioning scheme and the corresponding communication cost for the entire program (line 19).

Figure 10(b) demonstrates the searching process for complete ADI. In the F matrix for A and the promoted B, 1 indicates cross-loop access within that loop, and the corresponding opt matrix records the results of the search. Since neither the i nor the j dimension consists entirely of 0, parallelization cannot be achieved by partitioning along just one of them (line 4-7). From $(S2, j)$ to $(S4, j)$, both opt_B and opt_A record partitioning along the (j) dimension, as their corresponding values in the F matrix are all 0 (line 12-13).

In opt_A, $(S5, j)$ cannot follow the same pattern as $(S4, j)$ because F_A matrix records 1 in the corresponding position. However, since both the dimensions i and j have only one element marked as 1, the rest being 0, it is feasible to partition simultaneously between these two dimensions. Merging the j dimension into $(S5, i)$ results in lower communication cost compared to performing two separate all-to-all communications, one between $S2$ and $S3$ and the other between $S4$ and $S5$, as two-dimensional partitioning does not require all-to-all communication (line 15-16). Finally, opt_A records $(i + j)$ at both $(S5, j)$ and $(S6, j)$, and its intersection with opt_B's record of (j) at $(S6, j)$ results in the final partitioning scheme being $(i + j)$ (line 19).

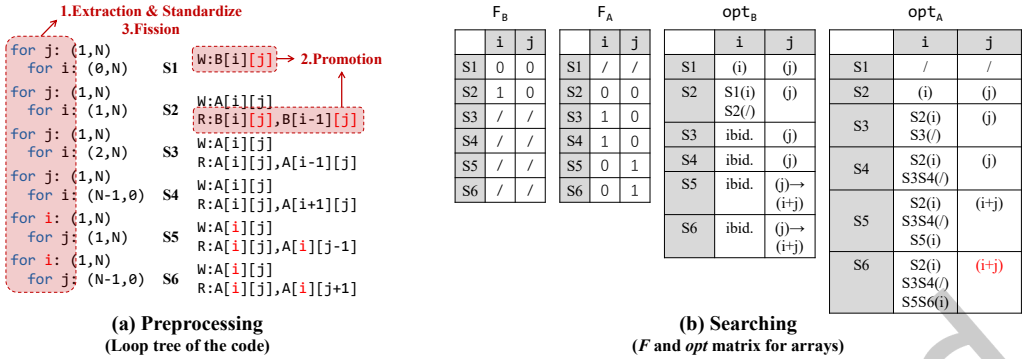


Fig. 10. **Example of preprocessing and searching.**

4.2.3 Mapping. The result of searching typically involve partitioning instances and arrays across multiple dimensions. In this section, we explain how to map these multi-dimensional partitions to different processes to prevent load imbalance and process idleness, thereby ensuring overall execution time. Note that this work does not address the topology relationship between processes and processors, such as placing certain processes on the same processor to further reduce inter-process communication time. This aspect is orthogonal to our work.

Consider the ADI code within a three-dimensional context in Figure 11, where the array A is three-dimensional and each nested loop has dependency chains across different dimensions. Our objective is to seek a parallel scheme for partitions across multiple dimensions to avoid the synchronization costs associated with all-to-all communication, which is the approach taken by Pluto-Distmem.

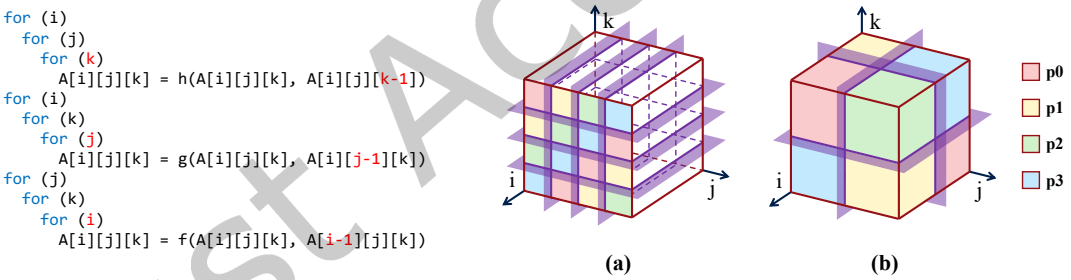


Fig. 11. **Different ways of mapping partition to processes.**

Figure 11(a) and (b) illustrate partitioning schemes in two and three dimensions, respectively. Even in the presence of dependency chains—necessitating sequential computation from 0 to N in each dimension—all processes are computing at any moment within each loop, with no idle time. Both partitioning schemes exhibit a communication complexity of $O(N^2)$, significantly outperforming the $O(N^3)$ complexity required for all-to-all communication.

Specifically, the partitioning scheme in (a) requires inter-process communication only in the first two loops because each process holds complete data along the i-dimension, eliminating the need for communication in the last loop. In contrast, partitioning scheme in (b) requires communication in all three loops. The difference between the two schemes lies in the amount of communication data required for the same number of processes (e.g., four in Figure 11). Scheme (a) requires communicating the data contained within six planes between all processes, whereas scheme (b) only requires that within three planes, as indicated by the purple planes in

Figure 11. Algorithm 1 derives the partitioning scheme shown in (b). By integrating the third dimension into the scheme of the first two dimensions, the algorithm identifies a partitioning scheme with reduced communication costs (line 15).

The mappings in Figure 11(a) and (b) ensure that each process performs $1/P$ of the total computation, guaranteeing load balancing and eliminating idle waiting. The partition mapping is computed as follows. For a D-dimensional array being partitioned, all partition blocks in $D-1$ dimensions are mapped to different processes, and rotational shifting distribution is performed along the remaining dimension. Assuming the number of processes is $P = x^{D-1}$, where x is a positive integer, for a process p with a computational partition in $D-1$ dimensions, if the partition block in dimension d is at coordinate i_d, then the coordinate of process p's next partition block in dimension d is $(i_d + 1) \bmod P$ or $(i_d - 1) \bmod P$.

Figure 12(a) and (b) show the final results of the complete ADI code using ARACHNE method, along with the corresponding partitioning and mapping of A. The number of stages equals the number of processes (four in this case). Each process calculates the start and end index of the computation partition for each stage based on its MPI rank. For statements with dependencies along the i dimension, $S1$ to $S3$ are computed sequentially from $i = 0$ to $i = N-1$, corresponding to stage 0 to 4, while $S4$ is computed in the reverse order. For $S5$ and $S6$, which have dependencies along the j dimension, stages are partitioned with reference to the j dimension.

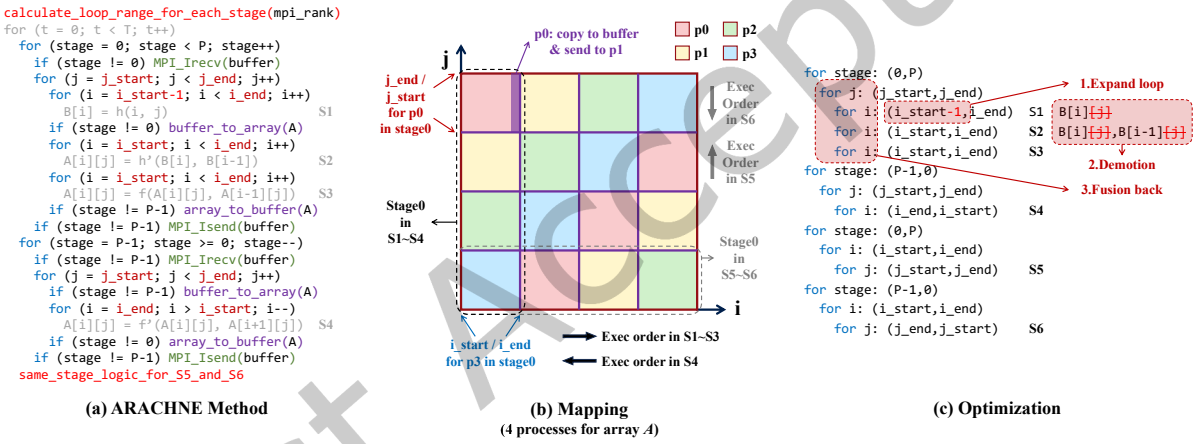


Fig. 12. **Example of mapping and optimization.**

4.3 Communication Optimization

Section 4.1 introduces two types of dependencies. One type leads to dependency chains, which can be parallelized through the multi-dimensional mapping discussed in Section 4.2.3. The other type allows avoiding communication by overlapping instances from other computational partitions. In a distributed environment, the computational overhead of accessing data is significantly lower than the communication cost.

Taking the example in Figure 13(a), where yellow lines represent the original computational partitions identified by the algorithm. We observe that for dependencies between $S1$ and $S2$ (indicated by blue lines), $S1$ does not have any additional dependency requirements in the preceding computational partition. Represented by green lines, a new computational partition is created by utilizing overlapping instance (as shown in Figure 13(b)). In this new partition, an additional instance of $S1$ is computed by the subsequent process to avoid communication induced by the second type of dependency.

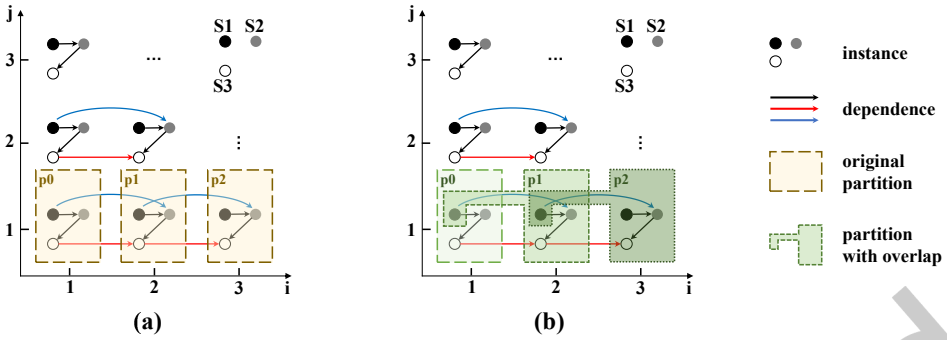


Fig. 13. **Comparison of Original partition and partition with overlap.**

The communication avoidance algorithm, as illustrated in Algorithm 2, achieves communication avoidance by expanding the loop ranges for each process. This expansion allows the computation to overlap with instances from the previous partition. Here, R_x and W_x represent the cross-iteration read and write requirements of an array x within a loop, respectively, while T_x denotes the updated requirements after expansion. The set Req is the collection of all the latest T_x during the processing, and E represents the final expansion computed for the current loop.

Algorithm 2 Loop Expansion for Communication Avoidance

1: **Input:** Loops with original computational partitions
2: **Output:** Expanded loops applied partitions with overlapping
3: **for** each fissioned loop, traversed backward **do**
4: **for** each variable x occurring in the loop **do**
5: **if** x is write-only **and** $T_x \in Req$ **then**
6: Expand this loop as $E = \texttt{argmax}_{t \in Req}|t|$
7: **for** each variable x occurring in the loop **do**
8: **if** x is read-write **and** exist inter-iteration access for W_x **then**
9: $T_x = W_x$
10: **if** x is read-only **and** exist inter-iteration access for R_x **then**
11: **if** $T_x \in Req$ **then**
12: $T_x = \texttt{argmax}_{t \in \{T_x, R_x\}}|t| + E$
13: **else**
14: Calculate expand requirement for v as $T_x = R_x + E$
15: Add T_x into Req
16: **if** Still have T_x in Req are unsatisfied **then**
17: Satisfy all remaining T_x via communication

Take the code shown in Figure 14 as an example, corresponding to the code partitioned along the j and i dimensions in Figure 8(c), where the start and end value of j and i differ for each process. The specific logic for calculating these range is omitted. $S3$ satisfies the first type of dependency, where its cross-iteration read $(i-1)$ is necessarily fulfilled through communication. Since $S3$ does not have a cross-iteration write ($W_A = 0$), there are no expansion requirements ($T_A = 0$) for this step. If a cross-iteration write existed, the expansion requirements would need to be recorded, as each array element involves the computation of multiple instances. This would require expanding within the corresponding process to ensure that the writes to the communication buffer are correct (lines 8-9). As $S3$ has no expansion requirements, $S2$ does not need to be expanded. Additionally, the array B in $S2$ does not belong to the dependency chain, so R_B are recorded (lines 13-15), and loop expansion is completed in $S1$ (lines 5-6).

Figure 12(c) shows the optimizations achieved after searching. Using Algorithm 2, loop expansion in Figure 14 is implemented to satisfy communication avoidance, after which B is demoted. Subsequently, the original loops are re-fused according to different stages to enhance data locality.

```
for (t = 0; t < T; t++)
  for (stage = 0; stage < P; stage++)
    for (j = j_x; j < j_y; j++)
      for (i = i_x-1; i < i_y; i++)
        B[i][j] = h(i, j)                    S1   E=argmax_{t∈Req}|t|=T_B=-1
      for (i = i_x; i < i_y; i++)
        A[i][j] = h'(B[i][j], B[i-1][j])     S2   Req{T_A=0}, R_B=-1 ➔ E=argmax_{t∈Req}|t|=0 ➔ T_B=R_B+E=-1, Req{T_A=0, T_B=-1}
      for (i = i_x; i < i_y; i++)
        A[i][j] = f(A[i][j], A[i-1][j])      S3   R_A=-1, W_A=0 ➔ T_A=W_A=0, Req{T_A=0}
```

Fig. 14. **Example of loop expansion.**

This method is particularly effective for local arrays, which are typically defined within a loop and first written at the level of the loop to which they belong. This means their instances have no preceding dependencies, creating a condition that allows for overlapping computations to avoid communication.

In Section 4.2.1, we promote the dimensions of local arrays to accommodate the needs for inter-process communication, which results in additional memory footprint. Once communication for a local array is completely eliminated through overlapping computation, we demote it back to its original definition to reduce runtime memory usage and leverage data locality. Additionally, if there is no need for communication between fissioned loops, we re-fuse them back into their original nested loop structure.

4.4 Data Distribution

The objective of data distribution is to reduce the memory footprint of each process. After computational partitioning, a process only needs to access a portion of the arrays, creating opportunities for array partitioning. There are two common computational modes in HPC applications: one involves solving a global numerical solution across the entire computational space, and the other simulates the micro-behavior at each grid point. Both modes are directly amenable to data distribution and do not require the addition of any extra computational logic or data structures to adapt to distributed parallelization.

Two key factors must be considered: (1) ensuring that the size of data definitions for each array within each process meets the needs of the entire program under various partitioning patterns; and (2) defining additional data space to accommodate data dependencies in computations. The limitation for applying this method of data distribution is that the code's memory access patterns only require data from neighboring computational partitions.

We utilize continuous memory spaces and array reshaping to replace the original array definitions, remapping memory accesses from the original complete arrays to the new reshaped arrays. Specifically, additional dimensions are used to index the stages of the computation. If all arrays involved in nested loops can be partitioned, the entire loop index and range are remapped. However, if only some arrays within a loop can be partitioned and other data need to be maintained across all processes, the loop remains unchanged, and only the memory accesses for the partitioned arrays are remapped. The corresponding code modifications for these two cases are shown in Figure 15 and Figure 16, respectively.

4.5 Limitations

The novelty of this work lies in the Searching and the Optimization process following the acquisition of dependency information, without introducing new methods for static dependency analysis. Therefore, existing static dependency analysis techniques affect Arachne's applicability. Arachne is not suitable for user codes with irregular accesses to the array, such as those using variables, pointers, or array elements as indices. These runtime-dependent access patterns are beyond the capabilities of static analysis.

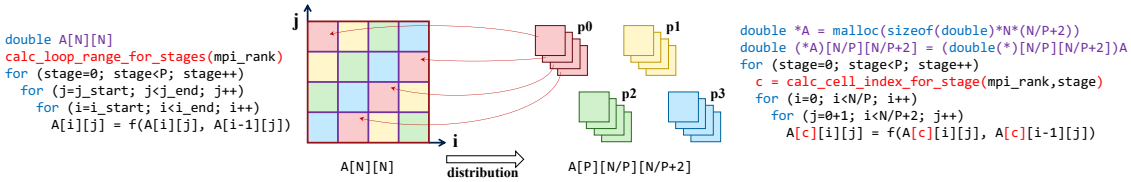


Fig. 15. **Distribution of all arrays.**

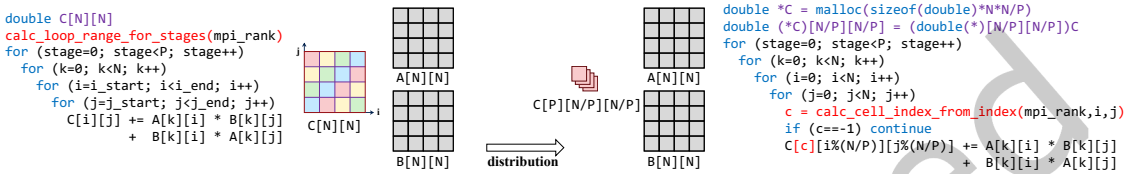


Fig. 16. **Distribution of part of arrays.**

Although the proposed method works well for array accesses using loop indices, complex affine access patterns that create dependencies across all dimensions will prevent the Partition Searching process from finding a partition scheme suiting multiple loops, leading to non-reduced communication costs equivalent to the all-to-all communication seen in Pluto-Distmem's outcome.

5 Directive Design

We developed a set of compiler directives as the front-end for Arachne to give users better control over the data distribution and parallel logic, extending beyond the automatic searching driven by static analysis in Arachne's back-end.

5.1 Design Principles and Advantages

Our directive design in Figure 17 is based on the widely-used and user-friendly OpenMP syntax. One advantage of adopting OpenMP is its ease of use, enabling users to specify complex parallel logic with minimal learning and fewer intrusive code modifications.

Another benefit is the ability to combine OpenMP directives to achieve multi-level parallelism: between nodes, through automatically generated MPI code for process-level parallelism, and within nodes, using OpenMP for thread-level parallelism. This approach improves program scalability across nodes while maximizing resource utilization and computational efficiency on clusters by leveraging both distributed and shared memory systems. Thus, our work is orthogonal to shared memory optimization techniques, such as the polyhedral model.

5.2 Partition of Computation

Computational partitioning is crucial for parallelization. For users unfamiliar with distributed parallelism or the parallelizable parts of their code, no extra information is required—simply use **`start`** and **`end`** directives to mark parallelizable regions. Data dependency analyzer can automatically manage inter-array dependencies for most access patterns.

For users who understand their code's dependencies and parallel potential, with the **`partition`** directive, the **`dimension`** clause allows specifying one or more dimensions for computational partitioning. We assume users have deeper knowledge of their code than any static analysis, enabling them to uncover parallel opportunities beyond what dependency analysis can detect. Additionally, users can provide implicit information that static

Partition of Computation

```
#pragma arach {start | end}

#pragma arach partition [clause[,clause]…]
    for-loops
clause:
    dimension(list)
    depend(var-dims[;var-dims]…)
    exclude(list)
    reduction(operator:list)
var-dims:
    var,index:dim[[,dim]…]->dim[[,dim]…]
```

Data-related Options

```
#pragma arach distribute data(list) [clause]
clause:
    dimension(list)

#pragma arach broadcast data(list)

#pragma arach {scatter | gather | allgather}
data(list) [clause]
clause:
    dimension(list)
```

Parallel Logic

```
#pragma arach reduce(operator:list)
                              [distribute] [all]
    structured-block
operator:
    +, *, &, |, ^, &&, ||, max, min

#pragma arach single {root | map}
    structured-block

#pragma arach barrier
```

Fig. 17. **Directives design corresponding to three categories.**

analysis cannot access, expanding the potential for parallelization. For example, the **depend** clause allows users to explicitly specify data dependencies that static analysis might miss, such as WAW dependencies from indirect array accesses. In HPC applications, using arrays as indices for sequential or blocked computations is common. Applying the depend clause to the data array avoids the limitations of static analysis when accessing data through an index array. The **exclude** clause lists variables that should not be partitioned, typically local loop variables or those unrelated to loop dimensions. Variables not excluded will be partitioned based on the search results. The **reduction** clause, which serves a similar purpose, will be explained later and can be used with or independently of computational partitioning.

5.3 Data-Related Operations

In migrating to distributed environments, users may handle arrays in different ways. For smaller data sets, frequent read-only access, or cases where all data must be initialized on a single node before distribution, users might prefer each process to maintain a complete data definition. This minimizes inter-process communication and memory remapping overhead. For larger arrays that need to fit within node memory limits, data distribution is preferred. Thus, in distributed programs, the data view can be a mix of complete and distributed arrays.

We provide the **distribute** directive for users to declare array distribution at the data definition stage, with arrays defaulting to a complete definition if this directive is not used. Distribution follows the computational partition. To facilitate data movement across nodes during partitioning, we offer the following directives: (1)One-to-many: **broadcast** copies data from the root to all processes, while **scatter** distributes data from the root to all processes; (2)Many-to-one: **gather** consolidates data from all processes to the root; (3)Many-to-many: **allgather** collects data from all processes and distributes it to all, equivalent to `gather` followed by `broadcast`. Users can also specify dimensions for `scatter/gather/allgather` using an optional `dimension clause`, corresponding to the computational partition dimensions. If not specified, the system automatically determines the target processes based on the partitioning.

5.4 Parallel Logic

We provide a variety of directives to support complex parallel logic in distributed environments. For global numerical problems, the program reduces data from all processes to derive a global result. Without data distribution, users can use directives to gather data to the root for final computation. With data distribution, each process reduces its portion locally, and the results are aggregated to produce the global outcome. The **reduce** directive supports this reduction pattern across nodes, offering binary operations and `max/min` options. The **all** clause ensures the results are updated across all processes.

For process execution, some code logic should be executed by only one process in parallel regions, such as special data handling by the root or processing edges of the computational domain. The **single** directive specifies

that only one process executes the code block, with the **`root`** or **`map`** clause designating the specific process. The **`barrier`** directive establishes a synchronization point, ensuring all processes reach the barrier before proceeding to maintain data consistency.

6 Evaluation

6.1 Setup

Our experiments were conducted on a distributed cluster comprising 64 nodes, interconnected by Mellanox MT2892 Family ConnectX-6 Dx with a network speed of up to 100Gbps. Each node within the cluster is equipped with dual Intel Xeon Gold 6248 CPUs, each offering 20 cores running at 2.50GHz, and 192GB of memory. The nodes are operated under Linux kernel version 4.18.0 and utilize GCC version 9.3.0 for compilation, coupled with OpenMPI 4.1.2 for managing distributed processes.

Arachne integrates a modular static dependence analyzer, which can be decoupled and replaced with other prevalent static dependency analysis tools, such as the Integer Set Library (ISL)[3]. We implemented a simple static dependence analyzer targeting loop and array accesses and the partition searcher as LLVM passes[6]. Additionally, the code generator was developed using Clang LibTooling[5]. The complete system encompasses over 12,000 lines of C++ code. We compare Arachne with Pluto-Distmem, which generates distributed code using default parameters (‘–isldep –lastwriter –tile –parallel –distmem –commopt‘).

6.2 Benchmarks

Our performance evaluation focuses on widely used benchmarks and applications within the HPC domain, specifically in areas such as linear algebra, physical simulations, and stencil computations. We selected eight small-kernel applications from PolyBench[1] and five real applications from the NAS Parallel Benchmarks (NPB)[2, 4]. This selection is three-fold.

(1)Representation of Scientific Computing Patterns: Both NPB and PolyBench cover a wide range of computational patterns common in HPC, reflecting typical scenarios. For instance, the alternating direction implicit (ADI) method and successive over-relaxation (SOR) method are included in NPB's SP and BT benchmark, while the FT benchmark utilizes data across entire dimensions, demonstrating comprehensive computational diversity.

(2)Capability to Search Parallel Schemes from the Static Analysis: Our evaluation assesses the effectiveness of deriving parallel schemes solely from static analysis, without adding any directives to PolyBench, FT, SP and BT. For the remaining unselected NPB benchmarks, IS and CG involve random memory access, MG uses the same pointer to reference different arrays, and LU is parallelizable due to its mathematical properties, though dependency analysis suggests otherwise. These cases exceed the capabilities of static analysis, as discussed in Section 4.5, and therefore are not included in the evaluation.

PolyBench's simple computational and data access patterns allow Pluto-Distmem to generate adequate parallel code. However, for NPB, which features more complex constructs such as local arrays and result reduction, Pluto-Distmem fails to transform. Thus, we compare Arachne against C++ versions of NPB-MPI, which are based on the official Fortran versions[4] and achieve near-optimal performance.

(3)Different Types of Communication: The selected benchmarks also vary in their communication requirements. ADI and SP involve neighbor communications, Floyd relies on broadcasting, and FT requires all-to-all between different partitioning patterns. Polybench benchmarks not unselected, which exhibit sync-free parallelism, making them less suitable for evaluation. The problem sizes for all benchmarks are listed in Table 1.

Besides Pluto-Distmem, no other comprehensive system was found that can generate distributed parallel code from serial implementations. We evaluated scalability from 1 to 64 nodes. To focus on communication

Benchmark	Problem size	Description
adi	T = 500, N = 4096	Alternating Direction Implicit
floyd	N = 6144	Floyd-Warshall Algorithm
fdtd	T = 1200, N = 8192	Finite-Difference Time-Domain
jacobi	T = 1200, N = 8192	Jacobi Method
covcol	N = 8192	Covariance Matrix Computation
dsyr2k	N = 8192	Double Symmetric Rank-2k Update
strmm	N = 8192	Triangular Matrix-Matrix Multiplication
trmm	N = 8192	Triangular Matrix Multiply
EP	M = 32	Embarrassingly Parallel
DC	10000000	Data Cube
FT	1024 * 1024 * 512	Fourier Transform
SP	160 * 160 * 160	Scalar Pentadiagonal
BT	160 * 160 * 160	Block Tri-diagonal

Table 1. **Problem size of benchmarks.**

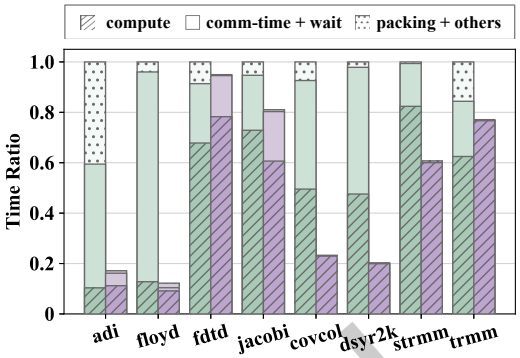


Fig. 18. **Normalized time ratio.**

performance and avoid interference from intra-node process interactions, we ran a single process per node and limited each process to a single OpenMP thread.

For generation time, simple benchmarks like those in PolyBench typically complete end-to-end in under 1 second, making a detailed analysis unnecessary. However, for more complex benchmarks like SP, which contains over two thousand lines of code and deeply nested loops, our static analyzer requires around 180 seconds to analyze dependencies. Despite this, the actual search and generation time remains under 1 second.

6.3 Time Breakdown

In this section, we analyzed the time breakdown for PolyBench on a 32-node setup to compare the parallel schemes generated by Arachne and Pluto-Distmem, as shown in Figure 18. This node count highlights the differences in computation scale and communication volume between the two schemes. By examining the time spent in different phases, we can better understand the performance differences observed in our scalability experiments.

In ADI and Floyd, Arachne showed significant performance improvements over Pluto-Distmem. As discussed earlier, Pluto-Distmem requires frequent all-to-all communication, with each process sending its computed segment to all others during each iteration, resulting in communication volumes of $array\_size \times (mpi\_size - 1)$. This leads to substantial overhead for packing and unpacking data. In contrast, Arachne only communicates array edges per iteration, reducing communication volume by several orders of magnitude. For FDTD and Jacobi, the parallel schemes identified by Arachne and Pluto-Distmem were similar, resulting in only slight performance differences due to runtime variations. Consequently, performance gains were less pronounced.

The latter four PolyBench applications exhibit sync-free parallelism, where each process performs computations independently without requiring data from other processes. In Covcol and Dsyr2k, a significant amount of waiting time was observed. Pluto-Distmem did not account for the number of instances when mapping dimensions to processes, leading to load imbalances and causing earlier finishing processes to wait at synchronization points. In contrast, Arachne ensures an even partition of workloads, eliminating synchronization waiting times. For STRMM and STMM, Arachne showed modest improvements over Pluto-Distmem, primarily due to reduced waiting times and lower runtime overhead from Pluto-Distmem.

6.4 Scalability

This section presents the scalability of parallel code generated by Arachne and Pluto-Distmem on a constant data size across different node configurations, as shown in Figure 19. For simpler benchmarks like PolyBench,

Pluto-Distmem can generate one parallel scheme using the polyhedral model, but this often results in significant inter-process communication and uneven load distribution.

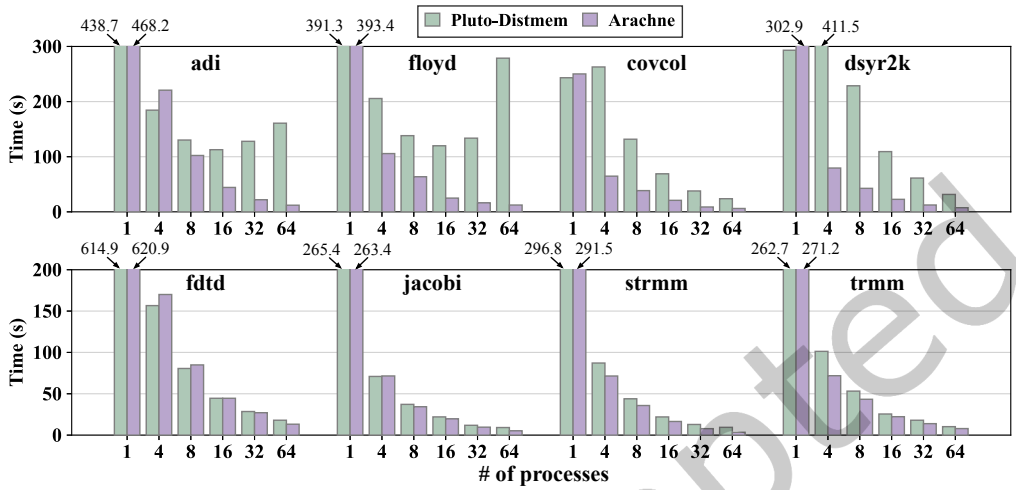


Fig. 19. **Result of scalability experiment on PolyBench.**

In the case of ADI, Arachne underperforms compared to Pluto-Distmem at lower parallelism (4 nodes), as the polyhedral model's loop tiling better exploits cache locality. For both ADI and Floyd, as the number of processes increases, the performance of Pluto-Distmem decreases due to the linear scaling of communication costs with the number of processes, overshadowing the benefits of computational parallelism. In contrast, Arachne's optimal computational partition continues to improve performance as parallelism increases.

For Covcol and Dsyr2k, Arachne consistently outperforms Pluto-Distmem across all parallelism levels, as Pluto-Distmem's poor load balancing leads to process waiting times. Arachne's even partitioning ensures no waiting, yielding better performance. In FDTD, Jacobi, STRMM, and TRMM, the improvements with Arachne over Pluto-Distmem are modest, with the performance enhancement ratio increasing with the number of processes.

The results in Table 2 show the performance of NPB benchmarks, where Arachne's transformed versions are compared to C++ versions of official NPB-MPI. Arachne consistently achieves more than 95% of the official version's performance. For FT, SP and BT, which involve nested loop array accesses, no manual directives were used. In SP and BT, optimal computational partitioning requires mapping two dimensions across all processes, meaning the number of processes per dimension must be equal—i.e., the total number of processes should be a square number.

The performance loss in FT, SP and BT stems from Arachne overestimating the required communication volume compared to the optimal implementation. To avoid infinite loop expansion in nested calls, Arachne uses the computational domain (i.e. a function) as the basic unit, which leads to some redundant communication before each computational domain. For EP and DC, which exhibit strong scalability, the parallel performance scales linearly with the number of processes. Unlike traditional loop-indexed array access patterns, their parallelism requires user-specified computational partitions and result reduction directives. Arachne's code generation based on `reduction` directives closely matches the performance of official MPI versions.

	EP		DC		FT		SP		BT	
Procs	Official	Ours	Official	Ours	Official	Ours	Official	Ours	Official	Ours
1	345.56s	346.74s	103.83s	104.36s	1029.48s	1037.19s	406.33s	410.21s	818.72s	826.50s
4	86.70s	87.15s	26.95s	27.75s	321.25s	323.14s	213.84s	217.49s	417.25s	423.41s
8	43.53s	43.60s	13.47s	13.92s	188.78s	191.22s	/	/	/	/
16	21.73s	21.96s	6.73s	7.11s	88.29s	89.68s	60.76s	62.95s	124.14s	127.52s
32	10.87s	11.23s	3.25s	3.51s	48.66s	49.03s	/	/	/	/
64	5.67s	5.79s	1.79s	1.85s	24.85s	26.33s	22.61s	23.68s	45.23s	47.11s
Average Efficiency	98.60%		95.58%		98.04%		96.77%		97.30%	

Table 2. **Result of scalability experiment on NPB.**

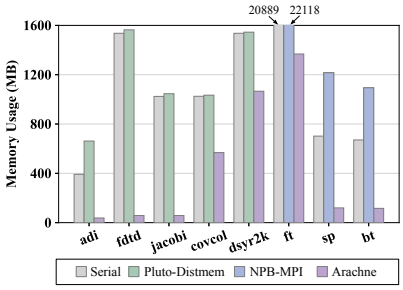


Fig. 20. **Result of memory usage.**

6.5 Load Imbalance

In this section, we address another key factor affecting performance in parallel computing: load balancing. In the generated code, due to the presence of barriers, the end-time discrepancy across processes is less than 1 second, meaning that waiting for one process delays all others. Figure 21(a) shows the load imbalance across 32 processes. By using dimensional partitioning and shifted mapping, Arachne significantly reduces load imbalance compared to Pluto-Distmem.

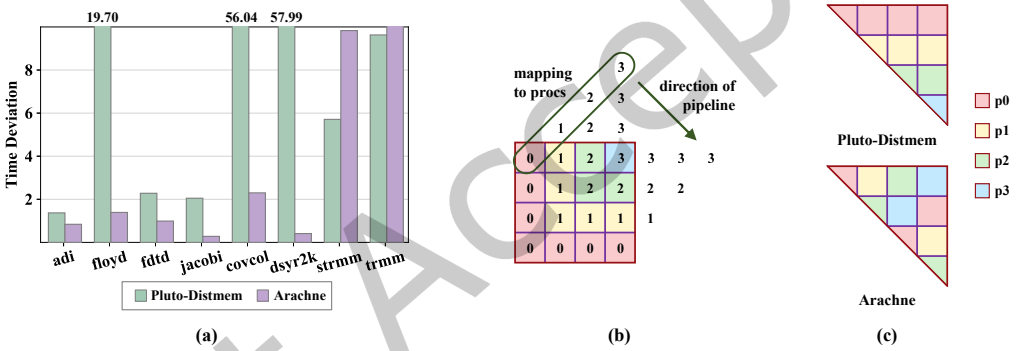


Fig. 21. **Deviation of computational time on PolyBench.**

Pluto-Distmem faces two main issues that contribute to its load imbalance: **(1)**Pluto-Distmem uses the polyhedral model to determine directions for pipeline parallelism and maps concurrently executable tiles to processes at each time step. This mapping creates notable load discrepancies; for example, as shown in Figure 21(b), Process 0 computes six more tiles than Process 3, as seen in the Floyd benchmark. **(2)**As discussed in Section 6.3, Pluto-Distmem does not account for the number of instances per dimension when mapping dimensions to processes, further contributing to load imbalance in cases like Covcol and Dsyr2k, as shown in Figure 21(c).

6.6 Memory Usage with Data Distribution

In this section, we discuss another critical objective: reducing memory consumption to accommodate the memory constraints of individual nodes. As noted in Section 4.4, if a computational partition only requires data from adjacent partitions, memory usage can be minimized through index remapping. We tested memory usage on five PolyBench applications across 32 nodes and three NPB applications on 16 nodes, comparing memory consumption against their original serial versions and parallelized versions generated by Pluto-Distmem or official MPI versions without data distribution, results shown in Figure 20.

Pluto-Distmem does not modify array definitions and uses its own runtime, while declaring additional buffers for data transfer. Our tests confirmed that in all five Pluto-Distmem versions of PolyBench, each process required more memory than in the original serial versions, with ADI consuming significantly more memory due to the large volume of data transmitted by each process. When all arrays are distributed, each process in a 32-node configuration requires only 3.78% of the original data volume, even accounting for communication buffers. Even if only some data is distributed, memory usage decreases with more processes, achieving over a 30% reduction in applications like Covcol and Dsyr2k. For FT, SP and BT running on 16 nodes, memory usage per process was reduced to just 6.55% and 17.09% of the serial versions, respectively.

7 Related Work

In the realm of distributed memory parallelization[20, 21, 32, 36], several attempts[8–10, 24, 42, 43] have faced significant limitations. Most efforts[20, 21, 32], including those leveraging perfectly nested loops with uniform dependences, only address specific aspects of parallelization and code generation, leaving comprehensive solutions out of reach. Recent work[38] highlights challenges in distributed memory code generation for mixed regular/irregular computations, while other methods[12, 35] optimize spatial data distribution in affine programs, improving locality in distributed systems. The bipartite graph model[12] focuses on reducing data transfers to minimize time, while a multi-objective optimization method[35] leverages affine transformations for improved performance in distributed systems.

The polyhedral model[11, 26, 27, 31, 45, 46], designed for shared-memory, uses mathematical methods to expose parallelism by transforming loop structures and optimizing locality. Recent adaptations of this model for distributed memory[14, 44] demonstrate its potential in handling both task and data distribution across nodes, but they often require significant manual intervention. Our approach is orthogonal to this model, applying polyhedral techniques at the process level for thread parallelism and local data optimization while our system manages task and data distribution. DISTAL[44], a tensor algebra compiler, abstracts data and computation distribution for heterogeneous distributed systems but requires users to rewrite algorithms in a specialized language, posing accessibility challenges.

Griebl's work[25] provides insights into scheduling and allocation for distributed architectures but lacks in generating efficient communication code, a challenge PLUTO-Distmem[14] addresses by mapping polyhedral outputs onto distributed systems. However, its high communication costs and limited adoption hinder scalability, which our work addresses using PLUTO-Distmem as a baseline.

R. Dathathri[19] focuses on data movement in heterogeneous architectures but does not address broader parallel code generation. Recent efforts[13, 13, 28, 29, 33, 40, 41] aim to bridge the gap between thread-based and distributed-memory paradigms. MPI-RICAL[40] assists domain decomposition in MPI using Transformer models, focusing on suggesting MPI calls but not automating parallelization. Basumallik's approach[13] simplifies data synchronization across nodes but introduces communication overhead due to its oversimplified strategy. Saà-Garriga[39] maps the OpenMP shared memory model to MPI, enhancing multithreading but struggling with complex computational patterns.

8 Conclusion

In this paper, we presented ARACHNE, a system designed to generate distributed parallel code with minimal communication overhead. ARACHNE addresses critical challenges in computational partitioning, data distribution, and communication handling. Through dynamic programming and user-friendly compiler directives, ARACHNE enables efficient parallelization and reduces memory usage. Experimental results demonstrate that ARACHNE outperforms PLUTO-Distmem, achieving lower communication overhead and supporting more parallel logic.

Acknowledgments

This work is partially sponsored by the National Key Research and Development Program of China (2023YFB3001504), the National Natural Science Foundation of China (62302302, 62232011). Quan Chen is the corresponding author.

References

[1] 2012. PolyBench. https://web.cs.ucla.edu/~pouchet/software/polybench/.
[2] 2022. NPB-CPP. https://github.com/GMAP/NPB-CPP.
[3] 2023. Integer Set Library. https://libisl.sourceforge.io/.
[4] 2023. NAS Parallel Benchmark. https://www.nas.nasa.gov/software/npb.html.
[5] 2024. Clang LibTooling. https://clang.llvm.org/docs/LibTooling.html.
[6] 2024. LLVM Pass. https://llvm.org/docs/WritingAnLLVMPass.html.
[7] Randy Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann.
[8] Saman P. Amarasinghe and Monica S. Lam. 1993. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 126–138. doi:10.1145/155090.155102
[9] Jennifer-Ann M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. 1995. Data and Computation Transformations for Multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Santa Barbara, California, USA, July 19-21, 1995*, Jeanne Ferrante, David A. Padua, and Richard L. Wexelblat (Eds.). ACM, 166–178. doi:10.1145/209936.209954
[10] Jennifer-Ann M. Anderson and Monica S. Lam. 1993. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 112–125. doi:10.1145/155090.155101
[11] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley (Eds.). IEEE, 193–205. doi:10.1109/CGO.2019.8661197
[12] A.P. Bagliy, N.M. Krivosheev, and B.Ya Steinberg. 2023. Automatic mapping of sequential programs to parallel computers with distributed memory. *Procedia Computer Science* 229 (2023), 236–244. doi:10.1016/j.procs.2023.12.025 12th International Young Scientists Conference in Computational Science, YSC2023.
[13] Ayon Basumallik and Rudolf Eigenmann. 2005. Towards automatic translation of OpenMP to MPI. In *Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, Cambridge, Massachusetts, USA, June 20-22, 2005*, Arvind and Larry Rudolph (Eds.). ACM, 189–198. doi:10.1145/1088149.1088174
[14] Uday Bondhugula. 2013. Compiling affine loop nests for distributed-memory parallel architectures. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, William Gropp and Satoshi Matsuoka (Eds.). ACM, 33:1–33:12. doi:10.1145/2503210.2503289
[15] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 101–113. doi:10.1145/1375581.1375595
[16] Uday Bondhugula, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2007. Automatic mapping of nested loops to FPGAs. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 101–111.
[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press. http://mitpress.mit.edu/books/introduction-algorithms
[18] Alain Darte, Yves Robert, and Frédéric Vivien. 2000. *Scheduling and automatic parallelization.* Birkhäuser.
[19] Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. 2013. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, Christian Fensch, Michael F. P. O'Boyle, André Seznec, and François Bodin (Eds.). IEEE Computer Society, 375–386. doi:10.1109/PACT.2013.6618833
[20] Damien de Montis, Jean-Baptiste Besnard, and Christophe Alias. 2021. *A Polyhedral Approach for Auto-Parallelization using a Distributed Virtual Machine.* Ph. D. Dissertation. INRIA, LIP-ENS Lyon; Paratools.
[21] Rui Ding, Rongcai Zhao, and Liguo Fu. 2013. Code Generation for Accurate Array Redistribution on Automatic Distributed-Memory Parallelization. In *14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2013, Honolulu, Hawaii, USA, 1-3 July, 2013*. IEEE Computer Society, 267–274. doi:10.1109/SNPD.2013.38
[22] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *Int. J. Parallel Program.* 20, 1 (1991), 23–53. doi:10.1007/BF01407931
[23] Ian T. Foster. 1995. *Designing and building parallel programs - concepts and tools for parallel software engineering.* Addison-Wesley.

[24] Georgios I. Goumas, Nikolaos Drosinos, Maria Athanasaki, and Nectarios Koziris. 2006. Message-passing code generation for non-rectangular tiling transformations. *Parallel Comput.* 32, 10 (2006), 711–732. doi:10.1016/J.PARCO.2006.07.003

[25] Martin Griebl et al. 2004. *Automatic parallelization of loop programs for distributed memory architectures.* Univ. Passau Passau, Germany.

[26] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 12 (jul 2015), 50 pages. doi:10.1145/2743016

[27] Jing Guo, Robert Bernecky, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2014. Polyhedral methods for improving parallel update-in-place. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques, Sanjay Rajopadhye and Sven Verdoolaege (Eds.). Vienna, Austria.*

[28] Khaled Hamidouche, Joel Falcou, and Daniel Etiemble. 2011. A framework for an automatic hybrid MPI+OpenMP code generation. In *Proceedings of the 19th High Performance Computing Symposia* (Boston, Massachusetts) *(HPC '11)*. Society for Computer Simulation International, San Diego, CA, USA, 48–55.

[29] Roy Harel, Iftach Mosseri, Hanoch Levin, et al. 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 (2020), 1–31. doi:10.1007/s10766-019-00640-3

[30] François Irigoin and Rémi Triolet. 1988. Supernode Partitioning. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and Peter Mager (Eds.). ACM Press, 319–329. doi:10.1145/73560.73588

[31] Caigui Jiang, Chengcheng Tang, Amir Vaxman, Peter Wonka, and Helmut Pottmann. 2015. Polyhedral patterns. 34, 6, Article 172 (nov 2015), 12 pages. doi:10.1145/2816795.2818077

[32] Martin Kong, Raneem Abu Yosef, Atanas Rountev, and P. Sadayappan. 2023. Automatic Generation of Distributed-Memory Mappings for Tensor Computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*, Dorian Arnold, Rosa M. Badia, and Kathryn M. Mohror (Eds.). ACM, 64:1–64:13. doi:10.1145/3581784.3607096

[33] Abdellah Kouadri-Mostefaoui, Daniel Millot, Christian Parrot, and Frédérique Silber-Chaussumier. 2009. Prototyping the automatic generation of MPI code from OpenMP programs in GCC. In *GROW 2009: 1st International Workshop on GCC Research Opportunities.* 1–11.

[34] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. 1994. *Introduction to Parallel Computing.* Benjamin/Cummings.

[35] Artem S. Lebedev and Shamil G. Magomedov. 2021. Automatic Parallelization of Affine Programs for Distributed Memory Systems. In *Futuristic Trends in Network and Communication Technologies*, Pradeep Kumar Singh, Gennady Veselov, Anton Pljonkin, Yugal Kumar, Marcin Paprzycki, and Yuri Zachinyaev (Eds.). Springer Singapore, Singapore, 91–101.

[36] Johannes Pahlke and Ivo F. Sbalzarini. 2024. Proven Distributed Memory Parallelization of Particle Methods. *CoRR* abs/2401.02180 (2024). doi:10.48550/ARXIV.2401.02180 arXiv:2401.02180

[37] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2010. Combined iterative and model-driven optimization in an automatic parallelization framework. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 1–11.

[38] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. Distributed memory code generation for mixed Irregular/Regular computations *(PPoPP 2015)*. Association for Computing Machinery, New York, NY, USA, 65–75. doi:10.1145/2688500.2688515

[39] Albert Saà-Garriga, David Castells-Rufas, and Jordi Carrabina. 2015. OMP2MPI: Automatic MPI code generation from OpenMP programs. *CoRR* abs/1502.02921 (2015). arXiv:1502.02921 http://arxiv.org/abs/1502.02921

[40] Nadav Schneider, Tal Kadosh, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023. MPI-RICAL: Data-Driven MPI Distributed Parallelism Assistance with Transformers. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (Denver, CO, USA) *(SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 2–10. doi:10.1145/3624062.3624063

[41] Jannek Squar, Tim Jammer, Michael Blesel, Michael Kuhn, and Thomas Ludwig. 2020. Compiler Assisted Source Transformation of OpenMP Kernels. In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*. 44–51. doi:10.1109/ISPDC51135.2020.00016

[42] Peiyi Tang and John N. Zigman. 1994. Reducing data communication overhead for DOACROSS loop nests. In *Proceedings of the 8th international conference on Supercomputing, ICS 1994, Manchester, UK, July 11-15, 1994*, John R. Gurd and William Jalby (Eds.). ACM, 44–53. doi:10.1145/181181.181261

[43] Jingling Xue. 2000. *Loop Tiling for Parallelism.* The Kluwer International Series in Engineering and Computer Science, Vol. 575. Kluwer. http://www.springer.com/computer/communication+networks/book/978-0-7923-7933-1

[44] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: the distributed tensor algebra compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 286–300. doi:10.1145/3519939.3523437

[45] Jie Zhao and Peng Di. 2020. Optimizing the Memory Hierarchy by Compositing Automatic Transformations on Computations and Data. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 427–441. doi:10.1109/MICRO50266.2020.00044

[46] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. 2013. Improving high level synthesis optimization opportunity through polyhedral transformations *(FPGA '13)*. Association for Computing Machinery, New York, NY, USA, 9–18. doi:10.1145/2435264.2435271

Received 31 May 2024; revised 23 October 2024; accepted 6 January 2025