

Mapple: A Domain-Specific Language for Mapping Distributed Heterogeneous Parallel Programs

ANJIANG WEI, Stanford University, USA
 ROHAN YADAV, Stanford University, USA
 HANG SONG, Stanford University, USA
 WONCHAN LEE, NVIDIA, USA
 KE WANG, Nanjing University, China
 ALEX AIKEN, Stanford University, USA

Optimizing parallel programs for distributed heterogeneous systems remains a complex task, often requiring significant code modifications. Task-based programming systems improve modularity by separating performance decisions from core application logic, but their mapping interfaces are often too low-level. In this work, we introduce Mapple, a high-level, declarative programming interface for mapping distributed applications. Mapple provides transformation primitives to resolve dimensionality mismatches between iteration and processor spaces, including a key primitive, *decompose*, that helps minimize communication volume. We implement Mapple on top of the Legion runtime by translating Mapple mappers into its low-level C++ interface. Across nine applications, including six matrix multiplication algorithms and three scientific computing workloads, Mapple reduces mapper code size by 14× and enables performance improvements of up to 1.34× over expert-written C++ mappers. In addition, the *decompose* primitive achieves up to 1.83× improvement over existing dimensionality-resolution heuristics. These results demonstrate that Mapple simplifies the development of high-performance mappers for distributed applications.

1 Introduction

Optimizing the performance of parallel programs on modern distributed heterogeneous systems (where nodes combine different types of processors, such as CPUs and GPUs) remains a complex and labor-intensive task despite decades of research efforts. Programmers must navigate intricate low-level programming interfaces and often make substantial code modifications to achieve high performance. As a result, even experts in high-performance computing find performance tuning a persistent challenge.

Two dominant programming paradigms exist today. The first is the Message Passing Interface (MPI)+X model [Chandra 2001; Dagum and Menon 1998; Gropp et al. 1999; Sanders and Kandrot 2010; Yang et al. 2011] where MPI is used for distributed-memory parallelism (across computers), and X is typically a shared-memory programming model used for intra-node parallelism (within one computer), such as OpenMP or CUDA. This model gives programmers fine-grained control over performance-critical aspects but significantly compromises productivity. Achieving performance gains often requires significant changes throughout the application code, making it difficult to optimize without inadvertently affecting application logic. The second paradigm is task-based programming systems [Augonnet et al. 2009; Chamberlain et al. 2007; Duran et al. 2011; Fatahalian et al. 2006; Kaiser et al. 2014; Kale and Krishnan 1993; Slaughter et al. 2015], which decouple performance decisions from the core application logic. These systems provide a dedicated interface for specifying performance-related policies, such as task and data placement across machines, enabling developers to tune program performance independently of application logic. This separation of concerns enhances modularity and makes performance tuning more systematic and less error-prone.

In task-based systems, the programming interface responsible for performance-critical decisions is known as the *mapping interface*, and the concrete implementation of these decisions is called a *mapper*. A central aspect of mapping is to partition tasks and data and assign them to processors in a distributed system. Distributed algorithms—such as those for matrix multiplication [Agarwal et al. 1995; Cannon 1969; Choi et al. 1994; Kwasniewski et al. 2019; Solomonik and Demmel 2011; Van De Geijn and Watts 1997]—rely on carefully crafted tensor partitioning and processor mappings. For instance, the Cannon’s algorithm [Cannon 1969] can experience up to a 3.5× slowdown when runtime heuristics are used instead of the intended mapping strategy.

A major limitation of existing mapping interface designs is their low-level nature and lack of abstraction, which exposes users to the complexity and intricacies of the underlying runtime system. These interfaces are often tightly coupled with the runtime, making them difficult to use, especially for application developers unfamiliar with their internals. For instance, defining a new mapper in the Chapel framework [Chamberlain et al. 2007] requires implementing 19 functions according to 33 pages of documentation [Chamberlain et al. 2011, 2010].

In this work, we introduce Mapple, a new mapping interface design. Mapple provides a high-level abstraction that hides low-level runtime details while still exposing performance-critical mapping decisions. Compared to the original interface, Mapple reduces code size by 14× without sacrificing performance (see Section 6.1).

The core challenge in any mapping interface targeting distributed applications is to determine how a multi-dimensional *iteration space*, defined by the algorithm’s logical loop structure, maps onto a multi-dimensional *processor space*, defined by the machine’s hierarchical architecture (e.g., a cluster of n nodes with m GPUs per node). This mapping process is directly responsible for the amount of data movement across processors, which is a key factor in achieving high performance on distributed systems. This challenge is compounded by the fact that these two spaces often differ in dimensionality, making the task of writing an effective mapping inherently complex.

To address the potential mismatch in dimensionality between the two spaces, Mapple introduces a set of transformation primitives that operate on the processor space. Building on a theoretical analysis, we identify a key primitive, *decompose*, which assists users in writing mappers that reduce data communication. Using this primitive, Mapple achieves up to 83% performance improvement over a commonly used heuristic for resolving dimensionality differences.

While Mapple’s transformation primitives on the processor space may appear similar to classical loop transformations, such as tiling, fusion, and fission [Chen et al. 2008, 2018a; Ragan-Kelley et al. 2013; Wolf 1992], they differ fundamentally in both purpose and scope. Traditional loop transformations aim to improve locality and parallelism within a single shared-memory system. In contrast, Mapple focuses on mapping iterations across a distributed processor space. Likewise, existing scheduling languages’ approach of separating schedules from algorithms may share some similarities to Mapple’s design [Ahrens et al. 2022; Senanayake et al. 2020; Zhang et al. 2018]. However, they do not address the problem of mapping iteration spaces onto processor spaces with potentially mismatched dimensionality, nor do they explicitly minimize data movement across heterogeneous processors.

We implement Mapple on top of the Legion [Bauer et al. 2012] parallel programming system. As a high-level interface, Mapple mappers are translated into Legion’s low-level C++ mapping interface. To evaluate Mapple, we apply it to a diverse set of nine applications, including six advanced distributed matrix multiplication algorithms and three scientific computing workloads. Mapple reduces the lines of code required to implement these mappers by 14× compared to hand-written C++ mappers, without introducing any observable performance overhead. In addition, we demonstrate that Mapple enables effective performance tuning: mappers written in Mapple outperform expert-crafted C++ mappers by up to 1.34×.

```

m = Machine(GPU)

def block2d(Tuple point, Tuple space):
    idx = point * m.size / space
    return m[*idx]

IndexTaskMap loop0 block2d

Region task_init arg0 GPU FBMEM

Layout task_finish arg1 CPU C_order

GarbageCollect systolic arg2

Backpressure systolic 1

```

(a) A Mapple mapper.

```

ShardID shard(const DomainPoint& point,
              const Domain& space,
              const size_t num_nodes) {
    auto rect2d = Rect<2>(space);
    // Implementation of external helper function is omitted
    auto blocks = this->select_num_blocks<2>(num_nodes, rect2d);
    Point<2> zeroes{0}, ones{1};
    Rect<2> blockSpace(zeroes, blocks - ones);
    auto numPoints = rect2d.hi - rect2d.lo + ones;
    Point<2> projected = point * blocks / numPoints;
    AffineLinearizedIndexSpace<2> linearizer(blockSpace);
    return linearizer.linearize(projected);
};

void map(const Task& task,
         const SliceTaskInput &input,
         SliceTaskOutput &output) {
    vector<Processor> targets = this->select_targets(task);
    auto size = targets.size();
    DomainT<2> space = input.domain;
    Point<2> n_b = this->select_num_blocks<2>(size, space);
    Point<2> zeros{0}, ones{1};
    ... // 125 lines of C++ code omitted here
    for (PointInRectIterator<2> it(blocks); it()!=NULL; it++) {
        Point<2> lo = *it; Point<2> hi = *it + ones;
        Point<2> slice_lo = n_p * lo / n_b + space.lo;
        Point<2> slice_hi = n_p * hi / n_b + space.lo - ones;
        DomainT<2, coord_t> slice_space; TaskSlice slice;
        slice.domain = {slice_lo, slice_hi};
        slice.proc = targets[index++ % targets.size()];
        output.slices.push_back(slice);
    }
}

```

(b) Simplified code excerpts from a C++ mapper.

Fig. 1. Comparison between a Mapple mapper and its partial C++ counterpart. The Mapple mapper uses a high-level, declarative design that abstracts away the complexity of low-level C++ implementations while still supporting performance optimization. The boxed `block2d` function is realized through two separate APIs in the C++ mapper, illustrating the conciseness of Mapple.

In summary, the contributions of this work are:

- (1) A high-level programming interface for mapping distributed applications to heterogeneous machines.
- (2) A set of transformation primitives for resolving dimensionality mismatches between iteration and processor spaces, including a key primitive, *decompose*, which helps minimize data communication volume.
- (3) An implementation of Mapple that reduces mapper code size by 14× compared to the low-level interface, without incurring additional overhead.
- (4) A comprehensive evaluation showing that mappers written in Mapple can outperform expert-written C++ mappers by up to 1.34×, and that the *decompose* primitive yields up to 1.83× performance improvement.

2 Design Goals and Non-Goals

Goal: A High-Level Interface Supporting Explicit Mapping and Optimization. We aim to provide a high-level programming interface that abstracts away the complexities of the underlying low-level runtime. This design allows users to focus on the logic of mapping decisions rather than the intricate details of system internals. These mapping decisions generally include:

- how to partition and distribute loops across distributed machines;
- where to place data in memory;
- how to lay out data in memory;
- whether to perform garbage collection; and
- how to make scheduling decisions.

To support this, we introduce explicit, modular abstractions and transformation primitives that make such decisions directly programmable and easy to change. For example, `block2d` specifies loop distribution, `Region` and `Layout` control data placement and layout, `GarbageCollect` handles memory management, and `Backpressure` guides scheduling. As shown in Figure 1, this approach achieves the same functionality as the C++ implementation but with significantly less code and greater clarity. We comprehensively quantify the reduction of code sizes in Section 6.1.

Moreover, our design enables systematic exploration of different mapping strategies. Programmers can easily experiment with different mappers, often outperforming hand-tuned C++ mappers. Figure 1a presents only a subset of Mapple’s features. We present the full features of Mapple in Section 7.1 and demonstrate its impact on performance tuning in Section 6.2.

Non-Goal: Portability across systems. We do not aim to support seamless portability across different systems. This non-goal reflects the inherent differences among distributed runtimes, which often prevent direct reuse of implementations. Nonetheless, we believe the ideas and abstractions introduced here can inform the design of other distributed systems.

3 The Core Challenge: Mapping Iteration Space to Processor Space

A central challenge for any mapping interface targeting distributed applications is efficiently assigning computation to hardware. In these settings, the algorithm defines a multi-dimensional *iteration space*, representing the logical structure of loops and computations. In contrast, the hardware exposes a multi-dimensional *processor space*, reflecting its hierarchical architecture—for example, a cluster of n nodes with m GPUs per node. The core problem, known as *index mapping*, is to map the iteration space onto the processor space in a way that minimizes data movement, which is critical for achieving high performance on distributed systems. Notably, the iteration and processor spaces often differ not only in size but also in dimensionality, adding further complexity to the mapping task.

In light of this challenge, we first survey existing approaches to index mapping. Motivated by their limitations, we introduce Mapple’s transformation primitives designed for index mapping in distributed systems. We defer the discussion of how Mapple handles dimensionality mismatches between iteration and processor spaces, using the *decompose* primitive, to Section 4.

3.1 Existing Approaches

Existing task-based systems support mapper development through one of three design styles, as illustrated in Figure 2. The *enumeration-based* interface [Adaptive MPI 2025] allows users to explicitly specify how each tile of the distributed array is mapped to a processor. While this interface is expressive, it lacks generality: enumeration-based mappers are hard-coded to a specific size of inputs or machines.

In contrast, *keyword-based* designs [Keyword Distribution 2025] provide a higher-level abstraction by letting users choose from a predefined set of standard distributions (e.g., `BlockDist` for block distribution). However, this approach sacrifices flexibility—the fixed keyword-based options cannot express non-standard mapping strategies. For example, none of the distributions used in matrix-multiplication algorithms in Section 6.1 can be represented using keyword-based interfaces.

Enumeration	Keyword	Programmatic												
<table><tr><th>Tile Index</th><th>Processor Index</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td></tr><tr><td>2</td><td>1</td></tr><tr><td>...</td><td>...</td></tr><tr><td>7</td><td>3</td></tr></table>	Tile Index	Processor Index	0	0	1	0	2	1	7	3	<p>Standard Distributions:</p> <ul style="list-style-type: none">• BlockDist• CyclicDist• BlockCyclicDist <p>Space = {1..L1, 1..L2}; Dist = new BlockDist(Space); D = Dist.createDomain(Space); var A, B: [D] int</p>	<p>3 classes to be implemented:</p> <pre>class GlobalDistribution class GlobalDomain class GlobalArray</pre> <p>19 functions to be implemented:</p> <pre>proc dsiNewRectangularDom() proc dsiAccess() proc dsiIndexToLocale() ...</pre>
Tile Index	Processor Index													
0	0													
1	0													
2	1													
...	...													
7	3													

Fig. 2. Three existing interface designs for mapping iteration space to processor space: the enumeration-based, keyword-based, and programmatic approaches.

The *programmatic* approach provides users with a low-level programming interface composed of classes and functions that must be implemented to control mapping behavior. While this design offers flexibility, it exposes significant system detail and complexity. For instance, defining a new mapping in Chapel requires implementing 19 functions based on 33 pages of documentation [Chamberlain et al. 2011, 2010], and Legion’s programmatic interface directly exposes the runtime’s internal abstractions, accompanied by 19 pages of documentation [Bauer et al. 2012]. In our experience, only users with deep expertise in the underlying runtime system are able to successfully develop mappers using such interfaces.

3.2 Motivating Examples

We present three example mappers in Mapple to illustrate how iteration spaces are mapped onto processor spaces. First, we describe a standard block distribution in Section 3.2.1 to introduce the core concepts. Next, we discuss a custom cyclic distribution in Section 3.2.2, which is not supported in keyword-based mapping interfaces. Finally, we use a mapper from Solomonik’s 2.5D matrix multiplication algorithm [Solomonik and Demmel 2011] in Section 3.2.3 to demonstrate the complexity introduced by mismatched dimensionality between the iteration and processor spaces.

3.2.1 Standard Block Distribution. Block distribution is a common default strategy widely supported by parallel programming systems. We use it here to illustrate the core concepts of index mapping in Mapple. In Mapple, users define mappers by writing a function that maps each iteration point in the iteration space to a target processor in the processor space. Figure 3 shows the function `block2D`, which implements a block distribution for an iteration space of size (6, 6) (that is, $0 \leq x < 6$ and $0 \leq y < 6$) and a processor space of size (2, 2), representing two nodes with two GPUs per node. The function assigns each iteration point to a specific processor.

The GPU processor space can be accessed using `Machine(GPU)`. In the mapper function, the iteration point `ipoint`, iteration space `ispace`, and processor space `size m.size` are all 2D integer tuples. These values are used to compute the processor indices `node_idx` and `gpu_idx`, which specify the destination processor for each iteration point. As shown in Figure 3, the shaded iteration point (2, 3) is mapped to node 0 and GPU 1. The function returns a specific processor for each iteration point, resulting in a block distribution over the processor space.

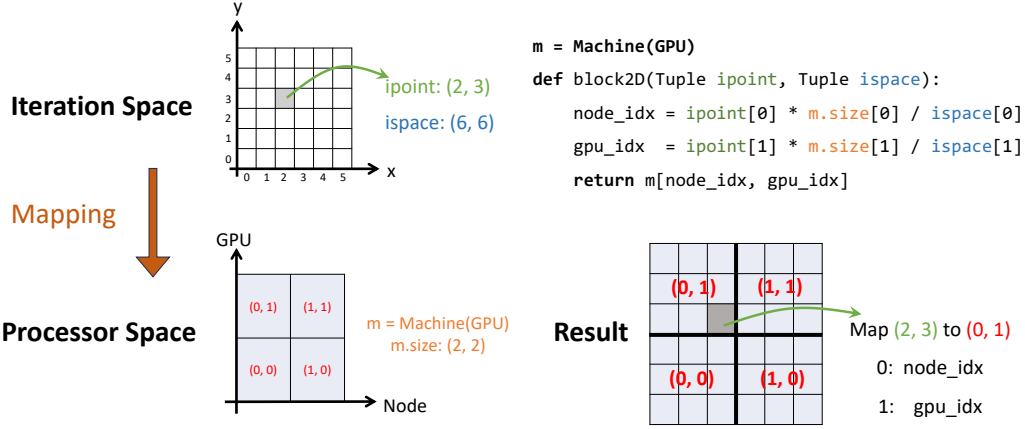


Fig. 3. Block mapping from the iteration space of (6, 6) to the processor space of (2, 2) indicating a machine with 2 nodes and 2 GPUs per node. A node index and a GPU index within the node indicate a specific GPU processor. The shaded iteration point (2, 3) is mapped to node 0 and GPU 1.

3.2.2 Custom Cyclic Distribution. We showcase a custom distribution expressible in Mapple that is not yet supported by most other parallel programming systems, which typically restrict users to a fixed set of standard, keyword-based distributions. This non-standard cyclic distribution is inspired by a simplified variant of a distributed matrix multiplication algorithm. Figure 4 illustrates the mapping function `linearCyclic`. In the resulting distribution, the subdiagonal iteration points (shaded in the iteration space) are mapped to the first processor (shaded in the processor space).

The mapper begins with the *merge* transformation primitive, which fuses dimension 0 and dimension 1 of the original 2D processor space into a single dimension. We will formally define the merge primitive in Section 3.3. After applying this transformation, the resulting processor space m becomes one-dimensional with size 4. In the mapping function, the 2D iteration point is first linearized, then assigned to processors using a round-robin distribution over the 4 processors.

3.2.3 Mapping Under Dimensionality Mismatch. We use this example to illustrate the complexity introduced by a mismatch between the dimensionality of the iteration space and the processor space, which in turn raises deeper research questions. Figure 5 shows a mapper for the Solomonik’s algorithm executed on a 2-node machine, where each node has 4 GPUs. The iteration space is three-dimensional, and the algorithm specifies a hierarchical distribution: the iteration space is first partitioned along the x-axis so that each node handles half of the x-dimension, and within each node, the 4 GPUs perform a 2D block distribution over the y-z plane. This example highlights how mismatched dimensions require careful coordination between hierarchical and multi-dimensional mapping strategies.

There is a mismatch in dimensionality between the iteration space, which is three-dimensional, and the initial processor space, which is two-dimensional. To enable the mapping required by the algorithm, we apply the *split* transformation primitive four times (highlighted in red in the code). The first two splits align the node-level processor dimension with the x-axis of the iteration space, while the last two splits align the GPU-level processor dimensions with the y and z axes. This results in a six-dimensional processor space, which we visualize as two separate 3D spaces: one for

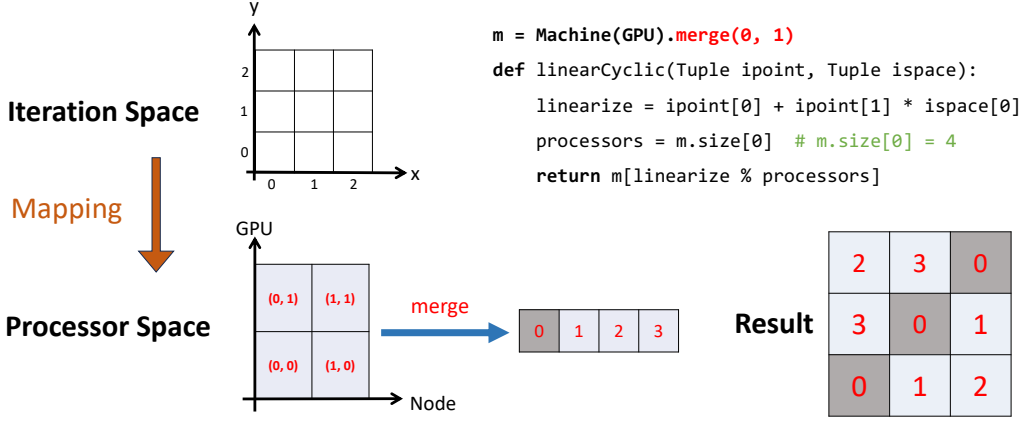


Fig. 4. A custom cyclic distribution. The merge primitive transforms the 2D processor space into a 1D space. The mapping function linearizes each 2D iteration point and applies a round-robin distribution over the resulting 1D processor space.

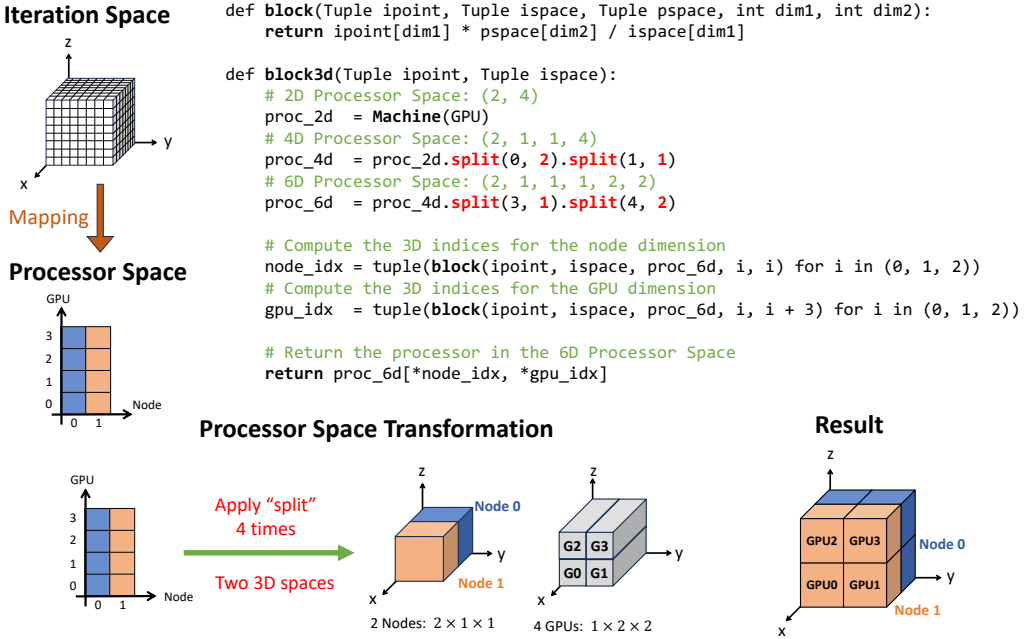


Fig. 5. A mapper illustrating the complexity caused by mismatched dimensionality between the iteration and processor spaces, as required by the Solomonik's algorithm on a 2-node machine with 4 GPUs per node. The original 2D processor space is transformed into a 6D space using the *split* primitive, visualized as two 3D spaces.

Transformation	Semantics
$m' = m.\text{split}(i, d)$	$m'[a_0, \dots, a_n] := m[b_0, \dots, b_{n-1}]$ $b_t = \begin{cases} a_t & t < i \\ a_i + a_{i+1} \cdot d & t = i \\ a_{t+1} & t > i \end{cases}$
$m' = m.\text{merge}(p, q)$	$m'[a_0, \dots, a_{n-2}] := m[b_0, \dots, b_{n-1}]$ $b_t = \begin{cases} a_t & t < p \text{ or } p < t < q \\ a_p \bmod m.\text{size}[p] & t = p \\ \left\lfloor \frac{a_p}{m.\text{size}[p]} \right\rfloor & t = q \\ a_{t-1} & t > q \end{cases}$
$m' = m.\text{swap}(p, q)$	$m'[a_0, \dots, a_{n-1}] := m[b_0, \dots, b_{n-1}]$ $b_t = \begin{cases} a_q & t = p \\ a_p & t = q \\ a_t & t \neq p \wedge t \neq q \end{cases}$
$m' = m.\text{slice}(i, \text{low}, \text{high})$	$m'[a_0, \dots, a_{n-1}] := m[b_0, \dots, b_{n-1}]$ $b_t = \begin{cases} a_i + \text{low} & t = i \\ a_t & t \neq i \end{cases}$
$m' = m.\text{decompose}(i, T)$	$T = (l_1, \dots, l_k)$ Formally defined in Section 4

Fig. 6. Semantics of transformation primitives expressed as mappings from the indices of the transformed processor space to the indices of the original processor space.

nodes of size (2, 1, 1) and one for GPUs of size (1, 2, 2). We formally define the *split* transformation primitive in Section 3.3.

This example also raises deeper research questions. Can we automatically determine the appropriate splitting factors? What underlying principles guide the mapping decisions in matrix multiplication algorithms? Is it possible to address dimensionality mismatches through a generalized transformation? We explore these questions in Section 4, where we introduce a new primitive called *decompose* to provide a unified solution.

3.3 Transformation Primitives

We define the semantics of each Mapple transformation primitive in Figure 6, with the exception of the *decompose* primitive, which is deferred to Section 4. Each transformation primitive is a function that takes a processor space m as input and returns a transformed processor space m' , following the mapping illustrated on the right-hand side of Figure 6. We now describe each transformation in detail.

The *split* transformation takes two arguments: a dimension index i and a splitting factor d . Given a processor space m of shape (s_0, \dots, s_{n-1}) , the operation $m' = m.\text{split}(i, d)$ produces a new processor space m' of shape $(s_0, \dots, d, s_i/d, \dots, s_{n-1})$. This transformation is invertible, allowing mappers to operate on m' while Mapple maps back to the original space m . The index relationship is:

$$m'[a_0, \dots, a_i, a_{i+1}, \dots, a_n] = m[a_0, \dots, a_i + a_{i+1} \times d, \dots, a_n].$$

Distribution	Iteration Space	Processor Space	Transformation	Mapping Function
block2D			<code>m = Machine(GPU)</code>	<pre>def block2D(Tuple ipoint, Tuple ispace): idx = ipoint * m.size / ispace return m[*idx]</pre>
block1D_x			<code>m = Machine(GPU)</code> <code>m1 = m.merge(0, 1).split(0, 1)</code>	<pre>def block1D_x(Tuple ipoint, Tuple ispace): idx = ipoint * m1.size / ispace return m1[*idx]</pre>
block1D_y			<code>m = Machine(GPU)</code> <code>m2 = m.merge(0, 1).split(0, 4)</code>	<pre>def block1D_y(Tuple ipoint, Tuple ispace): idx = ipoint * m2.size / ispace return m2[*idx]</pre>
cyclic2D			<code>m = Machine(GPU)</code>	<pre>def cyclic2D(Tuple ipoint, Tuple ispace): idx = ipoint % m.size return m[*idx]</pre>
cyclic1D_x			<code>m = Machine(GPU)</code> <code>m1 = m.merge(0, 1).split(0, 1)</code>	<pre>def cyclic1D_x(Tuple ipoint, Tuple ispace): idx = ipoint % m1.size return m1[idx]</pre>
cyclic1D_y			<code>m = Machine(GPU)</code> <code>m2 = m.merge(0, 1).split(0, 4)</code>	<pre>def cyclic1D_y(Tuple ipoint, Tuple ispace): idx = ipoint % m2.size return m2[idx]</pre>
block-cyclic			<code>m = Machine(GPU)</code>	<pre>def blockcyclic(Tuple ipoint, Tuple ispace): idx = ipoint / m.size % m.size return m[*idx]</pre>

Fig. 7. Common distributions expressed in Maple. The shaded region in the iteration space is mapped to the corresponding shaded processor. The transformation code reshapes the original (2, 2) processor space into the desired processor space, which is then used in the user-defined mapping function.

The *merge* transformation takes two dimensions p and q of the processor space and fuses them into one. Given a processor space m of shape (s_0, \dots, s_{n-1}) , the operation $m' = m.\text{merge}(p, q)$ produces a new processor space m' of shape $(s_0, \dots, s_p \cdot s_q, \dots, s_{n-1})$, where the two dimensions at positions p and q in m are combined into a single dimension at position p in m' . This transformation is invertible. The index mapping is given by:

$$m'[a_0, \dots, a_{n-2}] = m[a_0, \dots, a_p \bmod s_p, \dots, \lfloor a_p / s_p \rfloor, \dots, a_{n-2}].$$

Transformation primitives in Maple can be composed sequentially. Consider a 2D processor space m , and let $m' = m.\text{split}(0, d)$, followed by $m'' = m'. \text{merge}(0, 1)$. The resulting processor space m'' is again 2D. We now derive the index transformation from m'' to the original space m by sequentially applying the semantics of the *merge* and *split* transformations.

$$m''[a_0, a_1] = m'[a_0 \bmod d, \lfloor a_0 / d \rfloor, a_1] = m[(a_0 \bmod d) + \lfloor a_0 / d \rfloor \times d, a_1].$$

Because the expression $(a_0 \bmod d) + \lfloor a_0 / d \rfloor \times d$ simplifies to a_0 for all integers, it follows that

$$m''[a_0, a_1] = m[a_0, a_1].$$

This demonstrates that the *split* and *merge* primitives are exact inverses when applied in sequence with the same parameters.

The *swap* transformation exchanges two dimensions of the processor space. Combined with *merge*, which flattens two dimensions into one, it lets users control whether merging follows row-major or column-major order.

The *slice* transformation takes a dimension along with its lower and upper bounds, applying a constant offset to that dimension. It is useful for mapping an iteration space to a subset of the processor space, such as when assigning concurrent tasks to separate groups of processors.

Using these primitives, Mapple can transform processor spaces in flexible ways. Figure 7 illustrates how Mapple supports common distribution patterns. Suppose we want to map a 2D iteration space onto a 2-node machine with 2 GPUs per node. In the figure, the shaded region of the iteration space is mapped to the corresponding shaded processor in the processor space.

For the block2D distribution, the mapping function is equivalent to the one in Figure 3, but simpler due to tuple arithmetic support. As seen in the mapping functions for block2D, block1D_x, and block1D_y, the only difference lies in the transformed processor space. Although all use 2D processor spaces, the dimensions differ, resulting in different block distributions. The same pattern applies to cyclic2D, cyclic1D_x, and cyclic1D_y.

4 Decompose Transformation Primitive

As discussed in Section 3.2.3, mismatches between the dimensionality of the iteration and processor spaces pose significant challenges for mapping decisions. In this section, we address this issue by analyzing its underlying principles and introducing a generalized transformation primitive, *decompose*. Section 4.1 analyzes a suboptimal heuristic used by a commonly used programming system to handle dimensionality mismatches. We then formally define the *decompose* primitive in Section 4.2, grounded in a communication volume analysis. Finally, Section 4.3 describes our search-based optimization algorithm and analyzes its complexity.

4.1 Suboptimal Existing Heuristics for Resolving Dimensionality Mismatch

We analyze how existing task-based systems handle dimensionality mismatches between iteration and processor spaces. Most frameworks, to the best of our knowledge, bypass this issue by linearizing both spaces and applying a default 1D block mapping. However, they rarely document the specifics of their linearization process, making direct comparisons challenging. An exception is Chapel [Chamberlain et al. 2007], a widely used parallel programming framework, which explicitly describes its approach to dimensionality mismatch. To reveal the limitations of this strategy, we present an intuitive example showing how it can lead to suboptimal mappings.

Suppose we have 6 processors and a 2D iteration space. To match the dimensionality, we must split the 6 processors into a 2D tuple. There are four possible factorizations: (6, 1), (3, 2), (2, 3), and (1, 6). The existing algorithm used to determine the processor grid, as shown in Algorithm 1, does not consider the actual size of the iteration space.

The algorithm takes as input the number of processors to split, denoted by d , and the dimensionality of the iteration space, denoted by k . It returns an integer array of length k whose product equals d . The algorithm follows a greedy strategy to compute this array, aiming to produce factors that are as balanced as possible in magnitude. It maintains a running product for each dimension and assigns the next prime factor of d to the dimension with the smallest current product.

In the example above, the algorithm selects the grid (3, 2) by prioritizing balanced factorization, without considering the shape of the iteration space. This can lead to suboptimal mappings. We illustrate this with two iteration spaces, (12, 18) and (18, 12), corresponding to an application with spatial locality, as shown in Figure 8. Each iteration space is mapped to the processor grid (3, 2) using the block2D function, which partitions the iteration space into rectangular blocks assigned to each processor. The orange regions in the figure indicate data that must be transferred across processor boundaries due to this mapping.

For the (12, 18) iteration space, the total inter-processor communication volume is 96 elements, while for the (18, 12) iteration space it is 84 elements. This difference highlights the sensitivity of communication cost to how the iteration space is aligned with the processor grid. If the algorithm had taken the shape of the iteration space into account and chosen the processor grid (2, 3) for the

Algorithm 1: Greedy Heuristic for Processor Grid Selection (Suboptimal)

Input:

d # Number of processors

k # Dimensionality of the iteration space

```

1 Function Greedy( $d, k$ ):
2   primes  $\leftarrow$  PrimeFactorization( $d$ )           # Sorted list of prime factors,  $d = p_1 \leq \dots \leq p_n$ 
3   factors  $\leftarrow$  new int[ $k$ ]
4   for  $i \in \{0, \dots, k-1\}$  do
5     factors[ $i$ ]  $\leftarrow$  1
6   for  $p \in$  primes do
7      $j \leftarrow$  ArgMin(factors)                   # Index of smallest current product
8     factors[ $j$ ]  $\leftarrow$  factors[ $j$ ]  $\times$   $p$ 
9   Sort(factors, descending=True)                 # Sort for consistent ordering
10  return factors

```

Application

```

for i in range(L1):
  for j in range(L2):
    B[i][j] = 0.2 * (A[i][j]
                     + A[i][j-1]
                     + A[i][j+1]
                     + A[i-1][j]
                     + A[i+1][j])

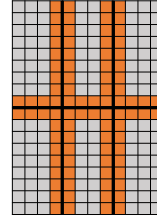
```


Heuristic Mapper

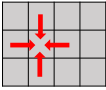
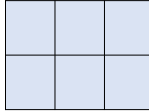
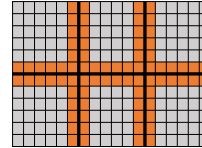
```

m = Machine(GPU).merge(0, 1)
m = m.split(0, 3)
def block2D(Tuple ipoint, Tuple ispace):
  idx = ipoint * m.size / ispace
  return m[*idx]

```

Iteration Space: (12, 18)

Inter-Processor
Communication
= Count()
= 96

Spatial Locality**Processor Space: (3, 2)****Iteration Space: (18, 12)**


Inter-Processor
Communication
= Count()
= 84

Fig. 8. For applications with spatial locality, the mapper computed by Algorithm 1 selects a fixed processor grid of (3, 2) for 2D block mapping. Data elements that must be transferred across processor boundaries are shown in orange. The (12, 18) iteration space incurs higher inter-processor communication volume than the (18, 12) iteration space, indicating that the chosen mapping is suboptimal for the former. A lower communication volume for the (12, 18) space could be achieved by using a (2, 3) processor grid instead.

(12, 18) space, the communication volume could have matched the more efficient 84-element case, avoiding unnecessary communication overhead.

4.2 Formal Definition of Decompose and Intuitive Solution

We formally define the optimization problem that the decompose primitive aims to solve, based on an analysis of inter-processor communication volume. We also present the underlying intuition to clarify how the solution addresses this problem.

Suppose we apply `decompose` to a processor space M of size (d_0, \dots, d_{n-1}) , written as $M' = M.\text{decompose}(i, (l_1, \dots, l_k))$. The `decompose` primitive splits the i -th dimension d_i into k natural numbers d_{i_1}, \dots, d_{i_k} , producing a new processor space M' of size $(d_0, \dots, d_{i_1}, \dots, d_{i_k}, \dots, d_{n-1})$. The transformation preserves the total size of the processor space, so it must satisfy $\prod_{m=1}^k d_{i_m} = d_i$.

Conceptually, `decompose` is a shorthand for applying a sequence of `split` transformations. Specifically, applying $m_k = m_1.\text{decompose}(i, T)$ is equivalent to performing $m_{n+1} = m_n.\text{split}(i + n - 1, d_{i_n})$ for $1 \leq n < k$.

There can be multiple ways to factor d_i into k natural numbers d_{i_1}, \dots, d_{i_k} . The `decompose` primitive selects a factorization by solving the following optimization problem:

$$\begin{aligned} & \text{minimize} && \sum_{m=1}^k \frac{d_{i_m}}{l_m} \\ & \text{subject to} && \prod_{m=1}^k d_{i_m} = d_i, \quad d_{i_m} \in \mathbb{N} \quad \forall m \end{aligned}$$

An equivalent formulation introduces the *workload vector* $w_m = \frac{l_m}{d_{i_m}}$ for $1 \leq m \leq k$, where w_m represents the amount of iteration space assigned to each processor in the m -th dimension. Rewriting the problem in terms of the workload vector gives:

$$\begin{aligned} & \text{minimize} && \sum_{m=1}^k \frac{1}{w_m} \\ & \text{subject to} && \prod_{m=1}^k w_m = \frac{\prod_{m=1}^k l_m}{d_i}, \quad \frac{l_m}{w_m} \in \mathbb{N} \quad \forall m \end{aligned}$$

We now illustrate the optimization problem using the two examples shown in Figure 9. The left-hand side depicts a 2D case with $k = 2$ and processor count $d_i = 4$. To compute the communication volume, we count the number of elements transferred across processor boundaries. This volume equals $2 \cdot (w_1 + w_2) \cdot d_i - 2 \cdot (l_1 + l_2)$, where the first term is the total perimeter of the d_i blocks of size (w_1, w_2) , and the second term is the perimeter of the entire iteration space of size (l_1, l_2) .

Given fixed input dimensions l_1 and l_2 and processor count d_i , the second term is constant, and $w_1 \cdot w_2 = \frac{l_1 \cdot l_2}{d_i}$ is also fixed. Therefore, minimizing the communication volume reduces to minimizing $w_1 + w_2$, which is equivalent to minimizing $\frac{1}{w_1} + \frac{1}{w_2}$. This matches the objective of our optimization formulation in the 2D case.

On the right-hand side, the figure shows a 3D example with $k = 3$ and $d_i = 16$. Here, the communication volume corresponds to the total inter-block surface area. The surface area S of the d_i cuboids of size (w_1, w_2, w_3) is given by:

$$2S = 2(w_1 w_2 + w_1 w_3 + w_2 w_3) \cdot d_i - 2(l_1 l_2 + l_1 l_3 + l_2 l_3).$$

Given fixed input dimensions l_1, l_2, l_3 and processor count d_i , the term $w_1 w_2 w_3 = \frac{l_1 l_2 l_3}{d_i}$ is constant. Thus, minimizing S is equivalent to minimizing $w_1 w_2 + w_1 w_3 + w_2 w_3$. Under the fixed-product constraint, this can be transformed into minimizing $\frac{1}{w_1} + \frac{1}{w_2} + \frac{1}{w_3}$, again aligning with the objective of our optimization formulation in the 3D case.

This analysis generalizes to any k -dimensional case. Let S denote the total inter-surface area in the k -dimensional setting. Then,

$$2S = SA(w_1, w_2, \dots, w_k) \cdot d_i - SA(l_1, l_2, \dots, l_k),$$

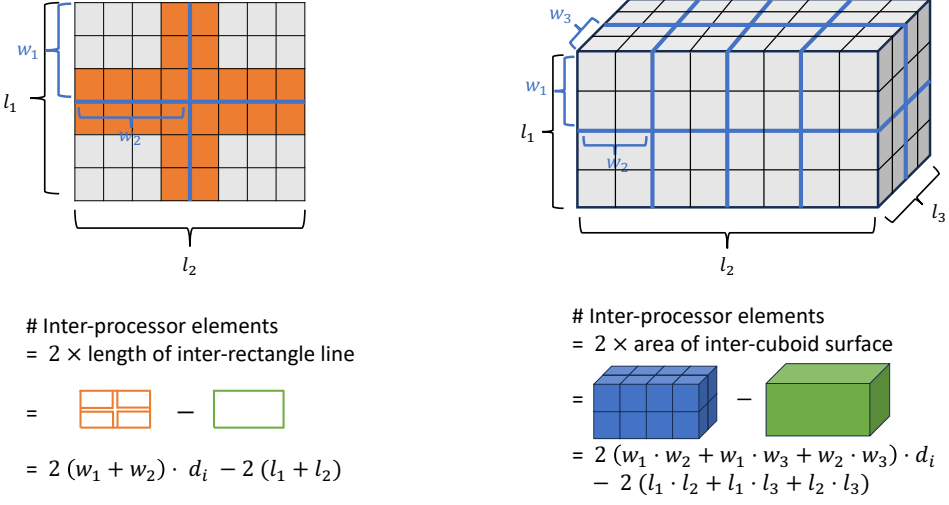


Fig. 9. We show two examples of how to compute the communication volume (namely, the number of inter-processor elements). In the 2D example, we compute the length of the inter-rectangle line by summing up the perimeter of all 4 rectangles of (w_1, w_2) and subtracting the perimeter of the (l_1, l_2) rectangle. In the 3D example, we compute the area of the inter-cuboid surface by summing up the surface area of all 16 cuboids of (w_1, w_2, w_3) and subtracting the surface area of the (l_1, l_2, l_3) cuboid.

where $SA(x_1, x_2, \dots, x_k)$ denotes the surface area of a k -dimensional hyperrectangle, defined as

$$SA(x_1, x_2, \dots, x_k) = 2 \cdot \left(\prod_{m=1}^k x_m \right) \cdot \left(\sum_{m=1}^k \frac{1}{x_m} \right).$$

Given that $\prod_{m=1}^k w_m = \frac{\prod_{m=1}^k l_m}{d_i}$ is constant for fixed input size and processor count, minimizing S reduces to minimizing $\sum_{m=1}^k \frac{1}{w_m}$ —which matches the objective of the optimization problem introduced earlier.

The solution to the optimization problem is not immediately obvious. In the following, we present an intuitive justification for the solution, which is based on the classic inequality stated below.

THEOREM. *Given a list of n positive numbers a_1, a_2, \dots, a_n , the following inequality holds:*

$$\frac{1}{n} \sum_{m=1}^n a_m \geq \left(\prod_{m=1}^n a_m \right)^{1/n},$$

with equality if and only if $a_1 = a_2 = \dots = a_n$.

This is the well-known arithmetic-geometric mean (AM-GM) inequality. It states that the arithmetic mean of a set of positive numbers is always greater than or equal to their geometric mean, with equality only when all values are equal.

Note that in the optimization problem, the term $\frac{\prod_{m=1}^k l_m}{d_i}$ is a constant determined by the input size and the number of processors. We can now apply the AM-GM inequality to the objective:

$\frac{1}{k} \sum_{m=1}^k \frac{1}{w_m} \geq \left(\prod_{m=1}^k \frac{1}{w_m} \right)^{1/k}$. Using the constraint $\prod_{m=1}^k w_m = \frac{\prod_{m=1}^k l_m}{d_i}$, we obtain:

$$\sum_{m=1}^k \frac{1}{w_m} \geq k \cdot \left(\frac{d_i}{\prod_{m=1}^k l_m} \right)^{1/k}.$$

Equality is achieved when all terms $\frac{1}{w_m}$ are equal, which implies $w_1 = w_2 = \dots = w_k$. In practice, however, exact equality may not be attainable due to the integrality constraint that each $\frac{l_m}{w_m}$ must be a natural number.

In the 2D iteration space example shown in Figure 8, the mapping for the (18, 12) space is optimal because the workload vector is $(w_1, w_2) = \left(\frac{l_1}{d_i}, \frac{l_2}{d_i} \right) = (6, 6)$, where the two components are equal. Similarly, the 3D example in Figure 9 uses an iteration space of (4, 8, 4) mapped onto 16 processors. The solution shown in the figure yields a workload vector $(w_1, w_2, w_3) = (2, 2, 2)$, which satisfies the sufficient condition for attaining the minimum communication volume.

These examples suggest that the optimal solution is achieved when the iteration space is divided among the processor dimensions in a balanced manner. However, in practice, this ideal may not be attainable due to the integrality constraint that each $\frac{l_m}{w_m}$ must be a natural number. To address this, we introduce a search-based algorithm for solving the optimization problem, which we describe in detail in Section 4.3.

4.3 Algorithm Implementation and Complexity Analysis

At a high level, the integrality constraint requires us to enumerate all possible ways of factoring d_i into k positive integers d_{i_1}, \dots, d_{i_k} such that $\prod_{m=1}^k d_{i_m} = d_i$, and then select the factorization that minimizes the objective function $\sum_{m=1}^k \frac{d_{i_m}}{l_m}$. The main challenge lies in efficiently and exhaustively enumerating all valid factorizations to ensure optimality.

We begin with a simple case: suppose $k = 3$ and $d_i = 16 = 2^4$. To enumerate all possible factorizations, we must determine all ways to distribute the four factors of 2 across the three dimensions. This reduces to finding all non-negative integer solutions to the equation $x_1 + x_2 + x_3 = 4$, which can be solved using recursion or backtracking.

A more complex example is when $k = 3$ and $d_i = 48 = 2^4 \cdot 3^1$. In this case, we must find all non-negative integer solutions to both $x_1 + x_2 + x_3 = 4$ and $y_1 + y_2 + y_3 = 1$, corresponding to the distribution of the prime factors 2 and 3, respectively. Each valid factorization corresponds to a tuple of the form $(2^{x_1} \cdot 3^{y_1}, 2^{x_2} \cdot 3^{y_2}, 2^{x_3} \cdot 3^{y_3})$ for some choice of (x_n, y_n) .

In the general case where $d_i = p_1^{a_1} \cdot p_2^{a_2} \cdot \dots \cdot p_t^{a_t}$, the prime factorization naturally decomposes the enumeration task. We can independently enumerate the placement strategies for each prime factor by solving t separate integer partition problems of the form $z_1 + \dots + z_k = a_j$ for each p_j , and then compute the Cartesian product of their solutions to generate all valid factorizations.

We now analyze the complexity of the search-based algorithm. Since the algorithm exhaustively enumerates all valid factorization strategies, its complexity is determined by the size of the search space. As shown in the earlier example, enumerating non-negative integer solutions to the equation $x_1 + x_2 + x_3 = 4$ is equivalent to counting integer partitions with repetition, which can be mapped to the problem of selecting two dividers among six gaps. This corresponds to the number of combinations $\binom{6}{2} = 15$.

In general, for a processor count with prime factorization $d_i = p_1^{a_1} \cdot p_2^{a_2} \cdot \dots \cdot p_t^{a_t}$, the total number of factorization strategies is:

$$\prod_{j=1}^t \binom{a_j + k - 1}{k - 1}$$

Each term in the product counts the number of ways to distribute the a_j copies of prime p_j across k dimensions. In practice, the exponents a_j are usually small (typically less than 10), and the number of dimensions k is often no greater than 3. As a result, the search space remains small, and the algorithm is computationally efficient in real-world scenarios.

We believe that enumeration is necessary to guarantee an optimal solution. Consider a greedy algorithm that, at each step, assigns prime factors of $d_i = 72$ (i.e., 2, 2, 2, 3, 3) to dimensions in a way that minimizes the maximum difference between elements of the workload vector. Given an iteration space of $(l_1, l_2) = (8, 9)$, this greedy strategy produces a suboptimal workload vector of $(\frac{4}{3}, \frac{3}{4})$, which results in imbalanced partitioning. In contrast, our search-based algorithm identifies the optimal factorization that yields a perfectly balanced workload vector of (1, 1).

5 Implementation

We describe the challenges and implementation of translating Mapple into task-based runtime systems. These systems employ a highly asynchronous and pipelined execution model, where tasks progress through multiple stages to maximize throughput. Mapping decisions—determining where tasks and data are placed—must be made at various points along this pipeline and are exposed via many distinct callback functions, each tied to a specific stage in a task’s lifecycle. This results in a fragmented and low-level interface that mirrors the internal structure of the runtime rather than providing an intuitive programming model for developers.

The core challenge is to design a high-level, unified mapping abstraction that can be faithfully translated into this low-level execution model. To illustrate this semantic gap, we first describe the runtime’s execution semantics in Section 5.1, then present our translation strategy in Section 5.2.

5.1 Execution Semantics of the Runtime System

Task-based runtime systems are designed to maximize throughput through an asynchronous, pipelined execution model, where tasks advance through several distinct stages during their lifetime. To illustrate the low-level nature of the mapping interface, we present the execution semantics of such systems. This exposes a key source of complexity: the interface is tightly coupled to internal execution stages and operates at a low level of abstraction. Consequently, a central challenge is to design a high-level, centralized mapping DSL that abstracts away low-level details while still capturing critical mapping decisions.

Before presenting the semantics, we first describe the key relationships between tasks that govern how and when they execute. Consider the following simple program:

```
task f(a, b) {
  g(a); // writes a
  h(b); // writes b
  k(a, b); // reads a, b and writes b
}
```

The task $f(a, b)$ is the *parent* of child tasks $g(a)$, $h(b)$, and $k(a, b)$. Each child begins only after the parent starts, and the parent completes only after all its children finish. Since tasks execute sequentially, parallelism arises from asynchronous launches. The order in which the parent invokes its children induces a program order \leq . In this example, $g(a) \leq h(b) \leq k(a, b)$, meaning $k(a, b)$ has *sibling predecessors* $g(a)$ and $h(b)$.

Task dependencies are represented by the relation \leq , indicating that one task may read or write a value produced by another. In this example, $g(a) \leq k(a, b)$ and $h(b) \leq k(a, b)$ capture that $k(a, b)$ may read values written by $g(a)$ and $h(b)$, and therefore cannot begin execution until both complete. However, this does not prevent all progress on $k(a, b)$ —in particular, its *mapping* (i.e., assignment

<i>State</i>	
Execution State S	$::= N[n] * N[n]$
Execution Log L	$::= e \text{ set}$
Execution Log Entry e	$::= \text{enqueued}(t) \mid \text{mapped}(t, p) \mid \text{launched}(t, p) \mid \text{executed}(t, p)$
Node Queues N	$::= (t \text{ queue})$
Processor p	$::= id$
<i>Tasks</i>	
Point P	$::= (i_1, \dots, i_n)$
Task t	$::= \text{Index}(id, P \text{ set})$
<i>Relations</i>	
Task Tree Relation r	$::= \text{parent}(t_p, t_c) \mid \text{task}(t)$
Task Tree T	$::= r \text{ set}$
Dependence Relation d	$::= t \leq t'$
Dependencies D	$::= d \text{ set}$
Sibling Relation w	$::= t \leq t'$
Siblings W	$::= w \text{ set}$

Fig. 10. Abstract Syntax for Execution Model

to a processor) can proceed before its dependencies finish. Since there is no dependency between $g(a)$ and $h(b)$, they may execute in parallel.

Figure 10 defines the structures used in the execution semantics. The execution state consists of two vectors of n queues—one for tasks *enqueued* on each node, and one for tasks that have been *mapped*. These represent two of the four stages in a task’s lifetime; the others are when a task is *launched* and when it is *executed*. A task progresses through these stages as follows:

- **Enqueued:** A task is enqueued once its parent and all sibling predecessors have been enqueued, preserving program order and control dependencies.
- **Mapped:** A task is mapped after all sibling predecessors it depends on have been mapped, ensuring their processor locations are known for scheduling data movement.
- **Launched:** A task can launch after all dependence predecessors have executed, ensuring their effects are complete and visible.
- **Executed:** A task is considered executed once it finishes running and all of its children have also completed, since their effects contribute to the parent’s semantics.

Figure 11 formalizes the task execution rules described above using an operational semantics. Each judgment has the form

$$L; (E, M) \xrightarrow{T; D; W} L'; (E', M')$$

which should be read as: given a task tree T , a dependence relation D , and a sibling predecessor relation W , the current execution log L and execution state (E, M) , representing the sets of enqueued and mapped tasks on each node, transition to a new execution log L' and new state (E', M') . The rules rely on two user-supplied mapping functions, SHARD and MAP, with the following signatures:

$$\begin{aligned} \text{SHARD: } & t \rightarrow \text{distribute}((t * p) * (t * p)) + \text{local}(t) \\ \text{MAP: } & t \rightarrow p \end{aligned}$$

$$\begin{array}{c}
task(t) \in T \\
t = \text{Index}(id, P) \\
\text{enqueued}(t) \notin L \\
\text{parent}(t_p, t) \in T \\
\text{launched}(t_p, p) \in L \\
\forall t_s : \text{parent}(t_p, t_s) \in T \wedge t_s \leq t, \text{enqueued}(t_s) \in L \\
\hline
L; (E, M) \xrightarrow{T;D;W} L \cup \{\text{enqueued}(t)\}; (E +_p t, M) \quad [\text{ENQUEUE}]
\end{array}$$

$$\begin{array}{c}
t = \text{Index}(id, P) \\
\text{SHARD}(t) = \text{distribute}((t_1, p_1), (t_2, p_2)) \\
t = (id, P) \\
t_1 = (id_1, P_1) \\
t_2 = (id_2, P_2) \\
id_1, id_2 \text{ fresh (not in } T, E, M, L) \\
P = P_1 \cup P_2 \\
E' = E -_i t \\
\hline
L; (E, M) \xrightarrow{T;D;W} L; ((E' +_{p_1} t_1) +_{p_2} t_2, M) \quad [\text{DISTRIBUTE}]
\end{array}$$

$$\begin{array}{c}
\text{SHARD}(t) = \text{local}(t) \\
E' = E -_p t \\
\hline
L; (E, M) \xrightarrow{T;D;W} L; (E', M +_p t) \quad [\text{LOCAL}]
\end{array}$$

$$\begin{array}{c}
\forall t' : t' \leq t, \text{mapped}(t', _) \in L \\
\text{MAP}(t) = p \\
M' = M -_i t \\
\hline
L; (E, M) \xrightarrow{T;D;W} L \cup \{\text{mapped}(t, p)\}; (E, M') \quad [\text{MAP}]
\end{array}$$

$$\begin{array}{c}
\forall t' : t' \leq t, \text{executed}(t', _) \in L \\
\text{mapped}(t, p) \in L \\
\hline
L; (E, M) \xrightarrow{T;D;W} L \cup \{\text{launched}(t, p)\}; (E, M) \quad [\text{LAUNCH}]
\end{array}$$

$$\begin{array}{c}
\forall t' : t' \leq t, \text{executed}(t', _) \in L \\
\forall c : \text{parent}(t, c), \text{executed}(c, _) \in L \\
\text{mapped}(t, p) \in L \\
\hline
L; (E, M) \xrightarrow{T;D;W} L \cup \{\text{executed}(t, p)\}; (E, M) \quad [\text{EXECUTE}]
\end{array}$$

Fig. 11. Transition relation judgments.

The SHARD function takes an index task and either partitions it into smaller tasks for distribution across nodes or assigns it to the local node. MAP then selects a specific processor within the target

node for execution. Although these are two *separate* callback functions in the low-level interface, Mapple *unifies* them by interpreting both as part of a single index transformation from iteration space to processor space.

Figure 11 formalizes the rules described above as execution judgments. The rules use a new operator $+_i$, which adds a task to the i th queue of a vector of queues:

$$(Q_1, \dots, Q_i, \dots, Q_n) +_i t = (Q_1, \dots, t : Q_i, \dots, Q_n)$$

We also need the dual dequeuing operation:

$$(Q_1, \dots, Q_i : t, \dots, Q_n) -_i t = (Q_1, \dots, Q_i, \dots, Q_n)$$

We briefly summarize each rule. [ENQUEUE] places a task on the same node as its parent once the parent has launched and all sibling predecessors are enqueued. [DISTRIBUTE] partitions an index task into independent sub-tasks for mapping. [LOCAL] selects the target node based on the user-defined SHARD function. [MAP] assigns the task to a specific processor within that node using the MAP function. [LAUNCH] and [EXECUTE] mark the start and completion of execution and involve no further mapping decisions.

The semantics of task execution underscore the low-level nature of mapping in task-based systems. To achieve high performance, tasks progress through multiple pipelined stages, each requiring user-defined mapping decisions. For brevity, we focus on SHARD and MAP, though in practice, 19 distinct callback APIs are invoked across a task’s lifetime. This design fragments mapping logic across numerous low-level callbacks, making the APIs complex and opaque to most application developers.

5.2 Translation

The translation poses two key challenges. First, as discussed in Section 5.1, mapping logic is spread across 19 callback functions triggered at different stages of a task’s lifetime. While this mirrors the runtime’s internal pipeline, it forces developers to work with a scattered and unintuitive interface (Figure 1). Second, the low-level API lacks abstraction over the machine architecture—there is no logical view of the hardware, nor support for aligning mapping decisions with loop structures or restructuring index spaces to facilitate coordination.

We implement Mapple by targeting Legion’s programmatic mapping interface, comprising roughly 7,000 lines of C++ code—much of it dedicated to handling low-level runtime details. Mapple overrides a set of callback functions invoked at different stages of the task execution pipeline to decide task placement. The two core overrides, corresponding to the user-defined SHARD and MAP functions, are shown in Figure 1b and explained in Section 5.1.

In Mapple, the machine’s logical view is defined with $\mathfrak{m} = \text{Machine}(\text{GPU})$, and users apply index transformations using provided primitives. The runtime invokes the SHARD function to assign a task to a node, followed by MAP to select a processor on that node. Both are implemented by interpreting user-defined mapping functions written in Mapple, evaluated per iteration point. The core translation computes the node (for SHARD) and processor (for MAP) identifiers via a sequence of invertible transformations over the processor space. As defined in Figure 6, these transformations yield a pair of unsigned integers representing coordinates in the original 2D processor space specified by Machine. These coordinates are then returned as the node and processor identifiers.

6 Results

Experiments were run on a cluster where each node has 40 IBM Power9 CPU cores and 4 NVIDIA V100 GPUs connected with NVLink 2.0 and an Infiniband EDR interconnect. To account for performance variability, each measurement was repeated five times, and the average is reported.

Benchmark. Our approach is evaluated using a diverse set of nine benchmarks selected to demonstrate the generality and effectiveness of Mapple across representative distributed workloads. Matrix multiplication is one of the most fundamental and heavily studied problems in parallel computing, serving as a cornerstone for many scientific applications. It is also a classic stress test for distributed systems, due to the need to balance computation and communication across complex processor hierarchies. The first six benchmarks therefore, focus on advanced parallel matrix multiplication algorithms: Cannon’s [Cannon 1969], SUMMA [Van De Geijn and Watts 1997], PUMMA [Choi et al. 1994], Johnson’s [Agarwal et al. 1995], Solomonik’s [Solomonik and Demmel 2011], and COSMA [Kwasniewski et al. 2019]. They are divided into 2D and non-2D categories. The 2D algorithms, such as Cannon’s, PUMMA, and SUMMA, enhance communication efficiency by partitioning matrices into 2D tiles and mapping them onto processor space, utilizing techniques like pipelining. Conversely, non-2D algorithms like Johnson’s, Solomonik’s, and COSMA, employ 3D partitioning or optimize processor grids to minimize communication overhead and boost performance.

The remaining three benchmarks are drawn from scientific simulation workloads, which represent a different class of distributed applications with distinct computation models and data access patterns. Circuit [Bauer et al. 2012] simulates electrical circuit behavior by modeling currents and voltages across interconnected nodes and wires. Stencil [Van der Wijngaart and Mattson 2014] operates on a 2D grid, updating each point’s value based on a stencil pattern derived from its neighbors. Pennant [Ferenbaugh 2015] models unstructured mesh Lagrangian staggered-grid hydrodynamics, essential for simulating compressible flow.

Together, these benchmarks span a wide spectrum of distributed computation patterns. Demonstrating strong results on these diverse workloads provides convincing evidence of Mapple’s ability to express complex mapping strategies and deliver high performance across application domains.

6.1 Lines of Code Reduction

We show the lines of code comparison between Mapple and C++ mappers in Table 1. For each mapper, we count the number of non-blank, non-comment lines of code written for the Mapple mapper and the C++ mapper. Using Mapple to implement mappers leads to a reduction in lines of code from 406 to 29 on average, i.e., a 14× reduction across all benchmarks. In Figure 12, we present example functions utilized by the mappers for distributed matrix-multiplication algorithms. The DSL effectively captures all the mapping decisions made by the C++ mappers in a concise manner.

We manually verify that both approaches make *identical* mapping decisions and achieve *matching performance* (i.e., identical throughput), indicating that any overhead introduced by Mapple is negligible. This confirms the fidelity of our Mapple mappers when compared against prior low-level C++ implementations [Bauer et al. 2012; Yadav et al. 2022a].

Table 1. Lines of Code (LoC) comparison between DSL and C++ mappers. The DSL achieves a 14× average reduction in LoC while maintaining the same performance as the C++ mappers.

Application	1	2	3	4	5	6	7	8	9	Avg.
LoC in C++	347	306	379	447	437	430	428	433	448	406
LoC in Mapple	16	14	16	38	38	38	33	38	32	29
LoC Reduction	22×	22×	24×	12×	12×	11×	13×	11×	14×	14×

Helper functions, Global variable	<pre> def block_primitive(Tuple ipoint, Tuple ispace, Tuple pspace, int dim1, int dim2): return ipoint[dim1] * pspace[dim2] / ispace[dim1] def cyclic_primitive(Tuple ipoint, Tuple ispace, Tuple pspace, int dim1, int dim2): return ipoint[dim1] % pspace[dim2] m_2d = Machine(GPU) </pre>
Solomonik's (function 1)	<pre> def hierarchical_block3D(Tuple ipoint, Tuple ispace): # split the 0th dimension into 3 dimensions m_4d = m_2d.decompose(0, ispace); # split the GPU dimension into 3 dimensions # sub iteration space for each node: ispace / m_4d[:-1] m_6d = m_4d.decompose(3, ispace / m_4d[:-1]) upper = tuple(block_primitive(ipoint, ispace, m_6d, i, i) for i in (0,1,2)) lower = tuple(cyclic_primitive(ipoint, ispace, m_6d, i, i + 3) for i in (0,1,2)) return m_6d[*upper, *lower] </pre>
Cannon's PUMMA SUMMA	<pre> def hierarchical_block2D(Tuple ipoint, Tuple ispace): # Similar to hierarchical_block3D except for the dimension of iteration space m_3d = m_2d.decompose(0, ispace) m_4d = m_3d.decompose(2, ispace / m_3d[:-1]) upper = tuple(block_primitive(ipoint, ispace, m_4d, i, i) for i in (0, 1)) lower = tuple(cyclic_primitive(ipoint, ispace, m_4d, i, i + 2) for i in (0, 1)) return m_4d[*upper, *lower] </pre>
Solomonik's (function 2)	<pre> def linearize_cyclic(Tuple ipoint, Tuple ispace): linearized = ipoint[0] + ispace[0] * ipoint[1] + ispace[0] * ispace[1] * ipoint[2] # cyclic over node dimension and GPU dimension node_idx = linearized % m_2d.size[0] gpu_idx = (linearized / m_2d.size[0]) % m_2d.size[1] return m_2d[node_idx, gpu_idx] </pre>
COSMA	<pre> def special_linearize3D(Tuple ipoint, Tuple ispace): # split the node dimension as equal as possible m_5d = m_2d.decompose(0, (1, 1, 1)) gx = m_5d.size[2] gy = m_5d.size[1] linearized = ipoint[0] + ipoint[1] * gx + ipoint[2] * gx * gy return m_2d[linearized % m_2d.size[0], 0] </pre>
Johnson's	<pre> def conditional_linearize3D(Tuple ipoint, Tuple ispace): grid_size = ispace[0] > ispace[2] ? ispace[0] : ispace[2] linearized = ipoint[0] + ipoint[1] * grid_size + ipoint[2] * grid_size * grid_size return m_2d[linearized % m_2d.size[0], 0] </pre>

Fig. 12. Example functions used by the mappers for the distributed matrix-multiplication algorithms.

Table 2. Performance improvements of Mapple mappers over expert-designed C++ mappers.

Application	1	2	3	4	5	6	7	8	9
Mapple Tuned Speedup	1.34×	1.02×	1.04×	1.09×	1.09×	1.09×	1.09×	1.07×	1.31×

6.2 Performance Tuning

We demonstrate that Mapple facilitates effective performance tuning by enabling the creation of mappers that outperform expert-designed C++ mappers. As shown in Table 2, Mapple achieves up to 1.34× speedup over the baseline C++ implementations. Even with a simple search-based algorithm, Mapple uncovers substantial optimization opportunities, helping users discover mappers that achieve higher throughput.

We analyze the sources of the observed performance improvements. For the six matrix multiplication applications (indexed 4 to 9), the speedup is entirely due to tuning the index mappings, although other features of Mapple contribute to code size reduction. In contrast, for the three scientific applications, the performance gains stem from assigning memories differently than the expert-written mappers—a capability enabled by Mapple’s full feature set, as discussed in Section 7.1.

We also present a case study highlighting the critical role of precise control over mapping strategies. Our investigation demonstrates significant performance variations resulting from minor alterations in the `hierarchical_block2D` function for the Cannon’s, PUMMA, and SUMMA algorithms. Specifically, we contrast the mapping function specified by the algorithm with an alternative approach integrating runtime heuristics: the runtime system dynamically assigns the iteration point to one of the four GPUs on the node, selecting the GPU with the least workload at runtime, rather than adhering to a predetermined distribution.

The throughput result depicted in Figure 13 underscores the pronounced performance discrepancies between the two approaches. The x-axis indicates different machine sizes, and the y-axis indicates the throughput per node. The “Algorithm Specification” line shows the throughput achieved by correctly implementing the corresponding algorithms, which match the performance results reported in prior work. The “Runtime Heuristics” line shows the throughput achieved by the alternative approach. In the 1-node scenario, the slowdown can reach up to 3.5×. This disparity stems from the additional data movement induced by divergent mapping decisions. Furthermore, the runtime heuristics-based mapping leads to out-of-memory (OOM) errors on 32-GPU runs for the PUMMA and SUMMA algorithms, because mapping decisions influence where the data is physically materialized in memory. This example shows that it is important to precisely control the mapping to minimize data movement and optimize memory resource utilization.

6.3 Performance Study of the Decompose Primitive

To investigate the impact of the decompose primitive, we compare it against the default heuristics for reshaping machine spaces from Algorithm 1. We evaluate the end-to-end performance of both mappers on stencil-pattern applications, which are a dominant communication pattern in image processing and scientific computing. Each point in the iteration space updates its value based on its nearest neighbors, making these applications highly sensitive to how computations are partitioned and mapped. As a result, demonstrating effectiveness on stencil applications provides strong evidence of the practical benefits of decompose for reducing communication in widely used computational workloads.

To ensure a fair comparison, we systematically vary the 2D stencil parameters as shown in Table 3. We test six aspect ratios from 1 : 1 to 1 : 32, reflecting shapes commonly seen in scientific

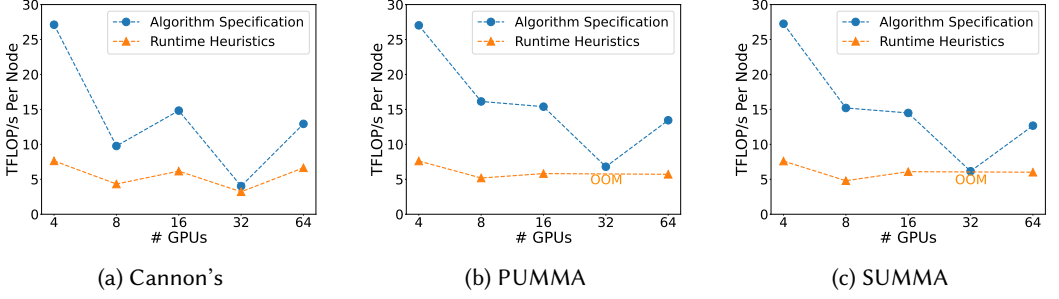


Fig. 13. Throughput comparison between the original hierarchical_block2D mapping function (specified by the Cannon's, PUMMA, and SUMMA algorithm) and a runtime heuristics-based mapper. The runtime heuristics-based mapping can lead to out-of-memory error (denoted by "OOM").

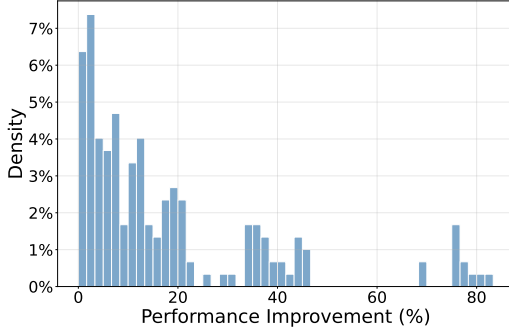


Fig. 14. The distribution of improvement percentage over different configurations.

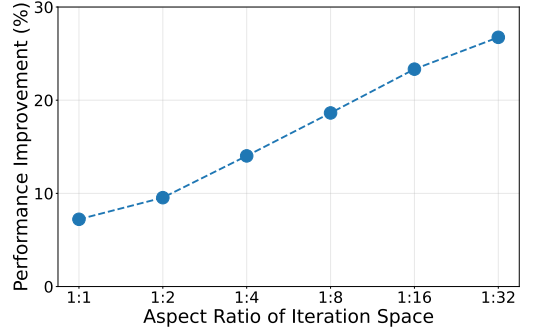


Fig. 15. Geometric mean of improvement percentage w.r.t. aspect ratios of iteration space.

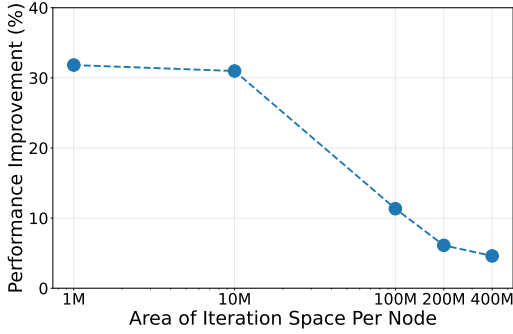


Fig. 16. Geometric mean of improvement percentage w.r.t. area of iteration space per node.

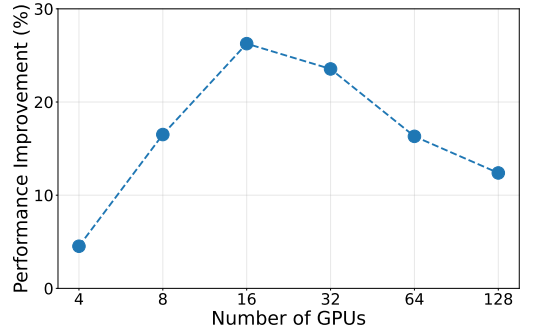


Fig. 17. Geometric mean of improvement percentage w.r.t. machine sizes.

simulations (e.g., liquid films [Nave et al. 2010], shock tubes [Wong et al. 2019]). As discussed in Section 4.1, Algorithm 1 can lead to suboptimal results. We also vary the iteration space per node across five sizes (10^6 to 4×10^8) to capture different communication-to-computation ratios, and scale the number of GPUs from 4 to 128 in powers of two. In total, we evaluate 180 configurations.

Table 3. Parameter space in performance comparison

Parameter	Values
Aspect ratio of total iteration space ($x : y$)	1 : 1, 1 : 2, 1 : 4, 1 : 8, 1 : 16, 1 : 32
Area of iteration space per node ($x * y / \#nodes$)	10^6 , 10^7 , 10^8 , 2×10^8 , 4×10^8
Number of GPUs	4, 8, 16, 32, 64, 128

Terminals: TaskName, RegionName, var, int

```

Program ::= Statement+
Statement ::= TaskMap | DataMap | DataLayout | FuncDef |
            IndexTaskMap TaskName var | ...
TaskMap ::= Task TaskName Proc+
DataMap ::= Region TaskName RegionName Proc Memory+
Proc ::= CPU | GPU | OMP
Memory ::= SYSMEM | FBMEM | ZCMEM
DataLayout ::= Layout TaskName RegionName Proc Constraint+
Constraint ::= SOA | AOS | C_order | F_order | Align == int
FuncDef ::= def var(var+): FuncStmnt+
FuncStmnt ::= var = Expr | return Expr
Expr ::= var | var(Expr+) | Machine(Proc) | Expr.Expr | Expr Op Expr | (Expr) |
        Expr[Expr] | *Expr | Expr ? Expr : Expr | Primitive
Primitive ::= split | merge | reorder | slice | decompose

```

Fig. 18. Mapple’s Grammar

We report the percentage improvement, with the distribution shown in Figure 14, ranging from 0% to 83%. The geometric mean improvement is 16%. We also plot the geometric mean improvement across varying values of each parameter. In Figure 15, the y-axis shows the geometric mean improvement percentage, and the x-axis shows the iteration space aspect ratio. As the ratio increases from 1 : 1 to 1 : 32, the improvement rises from 7% to 27%, highlighting the growing advantage of mappers using the decompose primitive. This trend is expected, as Algorithm 1 evenly splits processors—a strategy that performs well when the iteration space is nearly square.

Figure 16 shows how the improvement percentage varies with the iteration space size per node. As the area increases from 10^6 to 4×10^8 , the improvement drops from 32% to 5%. Larger iteration spaces reduce the communication-to-computation ratio, diminishing the relative benefit of minimizing communication. Figure 17 plots improvement against the number of nodes and GPUs. The improvement peaks at 26% on 4 nodes (16 GPUs). From 1 to 4 nodes, growing inter-node communication enhances the benefit of reducing bandwidth-bound traffic. Beyond 4 nodes, inter-rack latency becomes the bottleneck, diminishing the impact of improved mappings.

7 Discussion

7.1 Additional Performance Optimization Features in Mapple

While Section 3 and Section 4 focus on index mapping, the full optimization space includes several additional dimensions that contribute to the performance gains observed in the three scientific applications evaluated in Section 6.2. Figure 18 presents a simplified grammar capturing the core

constructs of Mapple. A primary axis of control is *processor selection*, expressed via the TaskMap directive. This determines whether a given task is assigned to GPUs, CPUs, or the OpenMP runtime. The decision is made per task and depends on factors such as task granularity, GPU memory capacity, and kernel launch overhead. For example, small tasks may favor CPUs to avoid the cost of GPU launches, and large-memory tasks may also favor CPUs if they exceed GPU memory limits.

Another critical dimension is *memory placement*, specified through the DataMap construct. This governs where task arguments are stored: in GPU FrameBuffer memory for high-speed access, in ZeroCopy regions to enable CPU-GPU sharing, or in host memory when capacity is a concern. Each location involves trade-offs between access latency, available memory, and transfer costs. This mapping is defined per task and per argument.

The optimization space also includes *memory layout*, configured with the DataLayout statement. This involves selecting between layouts such as Struct of Arrays (SoA) versus Array of Structures (AoS), specifying memory ordering (e.g., Fortran vs. C order), and applying alignment constraints (e.g., 128-byte alignment). These layout choices directly impact cache behavior and performance. This decision is made per task, per data region, and per target processor. Additional features, not shown in the figure, include scheduling policies (e.g., task prioritization), garbage collection strategies, and load-balancing directives.

7.2 Generalization of the Decompose Primitive

While Section 4 addresses isotropic communication across iteration space dimensions, the decompose primitive can be extended to anisotropic patterns, such as uneven halo widths and dimension-specific all-to-all exchanges (e.g., for data transposes). This generalization supports a much wider range of computations on block-structured grids. In both cases, only the objective function changes; the same minimization procedure from Section 4.3 still applies.

7.2.1 Anisotropic halo communication. To generalize, we extend the communication area from Section 4.2 to a volume that captures the total data exchanged between distributed machines. In isotropic cases, where halo width is uniform, volume reduces to area. For anisotropic patterns, varying halo width across dimensions must be considered. Letting h_n denote the width in dimension n , the total communication volume V in a k -dimensional space is $V = \sum_{n=1}^k d_{i_n} h_n \prod_{m \neq n} l_m$. For $l_n = w_n d_{i_n}$, the volume, V , can also be expressed as $V = \left(\sum_{n=1}^k \frac{h_n}{w_n} \right) \left(\prod_{m=1}^k l_m \right)$, where $\prod_{m=1}^k l_m$ is constant for a certain iteration space.

7.2.2 Transpose via all-to-all communication along certain dimensions. For transpose operations via all-to-all communication, the data volume per partition along the n -th dimension is given by $v_n = \frac{d_{i_n}-1}{d_{i_n}} \prod_{m=1}^k w_m$. This reflects that each partition splits its data evenly into d_{i_n} groups, one of which remains local while the others are sent to distinct peers along the n -th dimension. Thus, the total communication volume within a pencil of d_{i_n} aligned processors is $d_{i_n} v_n$. To optimize the mapping, we minimize the total communication volume, $(V + \sum_{n \in \mathbb{T}} V_n^*)$, where \mathbb{T} denotes the set of dimensions requiring transposes. The volume along dimension n , V_n^* , is defined as:

$$V_n^* = v_n \prod_{m=1}^k d_{i_m} = \left(1 - \frac{1}{d_{i_n}} \right) \left(\prod_{m=1}^k w_m \right) d_i.$$

8 Related Work

8.1 Task-based Programming Systems

Table 4 summarizes mapping features in HPC systems, all of which are supported by Mapple. Systems like Legion [Bauer et al. 2012] and StarPU [Augonnet et al. 2009] offer low-level C/C++

Table 4. Mapping features exposed to application control by different systems and covered by Mapple.

Systems	Task Placement	Data Placement	Data Layout	Scheduling	Load Balancing
Legion [Bauer et al. 2012]	✓	✓	✓	✓	✓
StarPU [Augonnet et al. 2010]	✓	✓	✓	✓	✓
Chapel [Chamberlain et al. 2007]	✓		✓		✓
Charm++ [Kale and Krishnan 1993]	✓	✓		✓	✓
PaRSEC [Danalis et al. 2015]	✓		✓	✓	✓
X10 [Charles et al. 2005]	✓	✓		✓	✓
HPX [Heller et al. 2017]	✓	✓		✓	✓
OpenMP [Chandra et al. 2001]	✓				✓
OmpSs [Duran et al. 2011]	✓			✓	✓
Mapple	✓	✓	✓	✓	✓

mapping interfaces via asynchronous callbacks, requiring deep familiarity with internal runtime abstractions. Other HPC systems offer less direct mapping control. ZPL [Deitz et al. 2004], Chapel [Chamberlain et al. 2007], and X10 [Charles et al. 2005] expose predefined data distributions (e.g., blocked, cyclic). HPX [Heller et al. 2017] and PaRSEC [Danalis et al. 2015] rely on schedulers with tunable policies for heterogeneous execution. OpenMP [Chandra et al. 2001], OmpSs [Duran et al. 2011], Charm++ [Kale and Krishnan 1993], and Chapel’s low-level APIs allow programmers to specify task placement directly in the source code, ranging from processor annotations (OpenMP, OmpSs) to custom C++ mapping classes (Charm++). Task-based systems in data analytics and AI such as MapReduce [Dean and Ghemawat 2008], Spark [Zaharia et al. 2010], Dask [Rocklin 2015], Ray [Moritz et al. 2018], and Pathways [Barham et al. 2022] rely on runtime mapping heuristics. This eases development but limits performance tuning when heuristics fall short. Mapple offers similar ease of use while enabling fine-grained control.

8.2 Loop Transformations

A large body of work on compiler algorithms uses loop transformations to maximize parallelism and minimize communication, including unimodular transformations [Banerjee 2007; Banerjee et al. 1990; Wolf 1992; Wolf and Lam 1991b; Wolfe 1982] (i.e., combinations of loop interchange, reversal, and skewing), loop block/tiling, fusion, fission, reindexing, and scaling [Lim et al. 1999, 2001; Wolf and Lam 1991a; Xue 2000], and affine transformations used by the polyhedral model [Acharya et al. 2018; Bondhugula 2013; Bondhugula et al. 2008]. These works share a similar motivation to Mapple, though the goals and specific transformations are different as explained in Section 1.

Another line of work uses scheduling languages [Baghdadi et al. 2019; Chen et al. 2008, 2018a; Kjolstad et al. 2017; Ragan-Kelley et al. 2012; Senanayake et al. 2020; Yadav et al. 2022b; Zhang et al. 2018] to separate the algorithm from the schedule, based on traditional loop optimizations such as split, collapse, and loop reordering. This separation enables programmers to just change the schedule when moving to different hardware. A growing body of research focuses on auto-scheduling [Adams et al. 2019; Chen et al. 2018b; Mullapudi et al. 2016; Tollenaere et al. 2023; Zheng et al. 2020, 2022] to reduce the manual effort required in writing schedules. Task-based systems share a similar approach in separating the algorithm from its schedule or mapping. However, prior work on scheduling languages does not address the challenges of mapping iteration spaces onto processor spaces of differing dimensionality in distributed settings.

9 Conclusion

We present Mapple, a high-level interface that simplifies the development of mappers for task-based parallel systems. Mapple overcomes key limitations of existing mapping interfaces by abstracting low-level runtime details while exposing transformation primitives to resolve dimensionality

mismatches between iteration and processor spaces. At the core of Mapple is the *decompose* primitive, which minimizes communication volume based on theoretical analysis. Across nine distributed applications, Mapple reduces mapper code size by 14× and achieves up to 1.34× speedup over expert-written C++ mappers. Moreover, the *decompose* primitive outperforms existing heuristics by up to 1.83×. These results demonstrate the practical effectiveness of Mapple in enabling the development of concise, high-performance mappers for distributed applications.

References

- Aravind Acharya, Uday Bondhugula, and Albert Cohen. 2018. Polyhedral auto-transformation with no integer linear programming. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 529–542.
- Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.
- Adaptive MPI 2025. Adaptive MPI. <https://charm.readthedocs.io/en/latest/ampi/04-extensions.html#user-defined-initial-mapping>.
- Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, Mahesh Joshi, and Prasad Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development* 39, 5 (1995), 575–582.
- Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 269–285.
- Cédric Augonnet, Samuel Thibault, and Raymond Namyst. 2010. *StarPU: a runtime system for scheduling tasks over accelerator-based multicore machines*. Ph.D. Dissertation. INRIA.
- Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In *European Conference on Parallel Processing*. Springer, 863–874.
- Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.
- Utpal Banerjee. 2007. *Loop transformations for restructuring compilers: the foundations*. Springer Science & Business Media.
- Utpal Banerjee et al. 1990. *Unimodular transformations of double loops*. University of Illinois at Urbana-Champaign, Center for Supercomputing
- Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. 2022. Pathways: Asynchronous distributed dataflow for ML. *Proceedings of Machine Learning and Systems* 4 (2022), 430–449.
- Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.
- Uday Bondhugula. 2013. Compiling affine loop nests for distributed-memory parallel architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12.
- Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 101–113.
- Lynn Elliot Cannon. 1969. *A cellular computer to implement the Kalman filter algorithm*. Montana State University.
- Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.
- Bradford L Chamberlain, Sung-Eun Choi, Steven J Deitz, David Iten, and Vassily Litvinov. 2011. Authoring user-defined domain maps in Chapel. In *Cray Users Group Conference (CUG)*.
- Bradford L Chamberlain, Steven J Deitz, David Iten, and Sung-Eun Choi. 2010. User-defined distributions and layouts in Chapel: Philosophy and framework. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*. 12–12.
- Rohit Chandra. 2001. *Parallel programming in OpenMP*. Morgan kaufmann.
- Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP*. Morgan kaufmann.
- Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices* 40, 10 (2005), 519–538.

- Chun Chen, Jacqueline Chame, and Mary Hall. 2008. A framework for composing high-level loop transformations. *Technical Report 08-897, USC Computer Science Technical Report* (2008).
- Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018a. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.
- Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018b. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems* 31 (2018).
- Jaeyoung Choi, David W Walker, and Jack J Dongarra. 1994. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience* 6, 7 (1994), 543–570.
- Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.
- Anthony Danalis, Heike Jagode, George Bosilca, and Jack Dongarra. 2015. Parsec in practice: Optimizing a legacy chemistry application through distributed task-based execution. In *2015 IEEE International Conference on Cluster Computing*. IEEE, 304–313.
- Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
- Steven J Deitz, Bradford L Chamberlain, and Lawrence Snyder. 2004. Abstractions for dynamic data distribution. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings*. IEEE, 42–51.
- Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters* 21, 02 (2011), 173–193.
- Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, et al. 2006. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 83–es.
- Charles R Ferenbaugh. 2015. PENNANT: an unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 4555–4572.
- William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.
- Thomas Heller, Patrick Diehl, Zachary Byerly, John Biddiscombe, and Hartmut Kaiser. 2017. Hpx—an open source c++ standard library for parallelism and concurrency. *Proceedings of OpenSuCo 5* (2017).
- Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. 1–11.
- Laxmikant V Kale and Sanjeev Krishnan. 1993. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. 91–108.
- Keyword Distribution 2025. Keyword Distribution. <https://chapel-lang.org/docs/1.28/modules/dists/BlockDist.html>.
- Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.
- Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefer. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–22.
- Amy W Lim, Gerald I Cheong, and Monica S Lam. 1999. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th international conference on Supercomputing*. 228–237.
- Amy W Lim, Shih-Wei Liao, and Monica S Lam. 2001. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*. 103–112.
- Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 561–577.
- Ravi Teja Mullaipudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–11.
- J-C Nave, XD Liu, and Sanjoy Banerjee. 2010. Direct numerical simulation of liquid films with large interfacial deformation. *Studies in Applied Mathematics* 125, 2 (2010), 153–177.
- Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédéric Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics (TOG)* 31, 4 (2012), 1–12.

- Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.
- Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, Vol. 130. Citeseer, 136.
- Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.
- Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A sparse iteration space transformation framework for sparse tensor algebra. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.
- Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: A high-productivity programming language for HPC with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.
- Edgar Solomonik and James Demmel. 2011. Communication-optimal parallel 2.5 D matrix multiplication and LU factorization algorithms. In *Euro-Par 2011 Parallel Processing: 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29-September 2, 2011, Proceedings, Part II* 17. Springer, 90–109.
- Nicolas Tollenare, Guillaume Iooss, Stéphane Pouget, Hugo Brunie, Christophe Guillon, Albert Cohen, P Sadayappan, and Fabrice Rastello. 2023. Autotuning convolutions is easier than you think. *ACM Transactions on Architecture and Code Optimization* 20, 2 (2023), 1–24.
- Robert A Van De Geijn and Jerrell Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274.
- Rob F Van der Wijngaart and Timothy G Mattson. 2014. The parallel research kernels. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.
- Michael Edward Wolf. 1992. *Improving locality and parallelism in nested loops*. stanford university.
- Michael E Wolf and Monica S Lam. 1991a. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 30–44.
- Michael E Wolf and Monica S Lam. 1991b. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel & Distributed Systems* 2, 04 (1991), 452–471.
- Michael Joseph Wolfe. 1982. *Optimizing supercompilers for supercomputers*. University of Illinois at Urbana-Champaign.
- Man Long Wong, Daniel Livescu, and Sanjiva K Lele. 2019. High-resolution Navier-Stokes simulations of Richtmyer-Meshkov instability with reshock. *Physical Review Fluids* 4, 10 (2019), 104609.
- Jingling Xue. 2000. *Loop tiling for parallelism*. Vol. 575. Springer Science & Business Media.
- Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022a. DISTAL: the distributed tensor algebra compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 286–300.
- Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022b. SpDISTAL: Compiling distributed sparse tensor computations. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.
- Chao-Tung Yang, Chih-Lin Huang, and Cheng-Fang Lin. 2011. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications* 182, 1 (2011), 266–269.
- Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.
- Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.
- Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.
- Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 874–887.