Generalized Multipartitioning of Multi-dimensional Arrays for Parallelizing Line-Sweep Computations

Alain Darte[*]
LIP, ENS-Lyon, 46, Allée d'Italie, 69007 Lyon, France.
`Alain.Darte@ens-lyon.fr`

Daniel Chavarría-Miranda Robert Fowler John Mellor-Crummey
Dept. of Computer Science MS-132, Rice University
6100 Main, Houston, TX USA
`{danich,johnmc,rjf}@cs.rice.edu`

July 15, 2002

Abstract

Multipartitioning is a strategy for decomposing multi-dimensional arrays into tiles and mapping the resulting tiles onto a collection of processors. This class of partitionings enables efficient parallelization of "line-sweep" computations that solve one-dimensional recurrences along each dimension of a multi-dimensional array. Multipartitionings yield balanced parallelism for line sweeps by assigning each processor the same number of data tiles to compute at each step of a sweep along any array dimension. Also, they induce only coarse-grain communication.

This paper considers the problem of computing generalized multipartitionings, which decompose d-dimensional arrays, $d \geq 2$, on an arbitrary number of processors. We describe an algorithm that computes an optimal multipartitioning onto all of the processors for this general case. We use a cost model to select the dimensionality of the best partitioning and the number of cuts to make along each array dimension; then, we show how to construct a mapping that assigns the resulting data tiles to each of the processors. The assignment of tiles to processors induced by this class of multipartitionings corresponds to an instance of a *latin hyper-rectangle*, a natural extension of *latin squares*, which have been widely studied in mathematics and statistics.

Finally, we describe how we extended the Rice dHPF compiler for High Performance Fortran to generate code that employs our strategy for generalized multipartitioning and show that the compiler's generated code for the NAS SP computational fluid dynamics benchmark achieves scalable high performance.

Keywords: Generalized latin squares, partitions of integers, loop parallelization, array mapping, High Performance Fortran.

AMS classification: 05A17, 05B15, 11A05, 11P81, 20N15, 68N20.

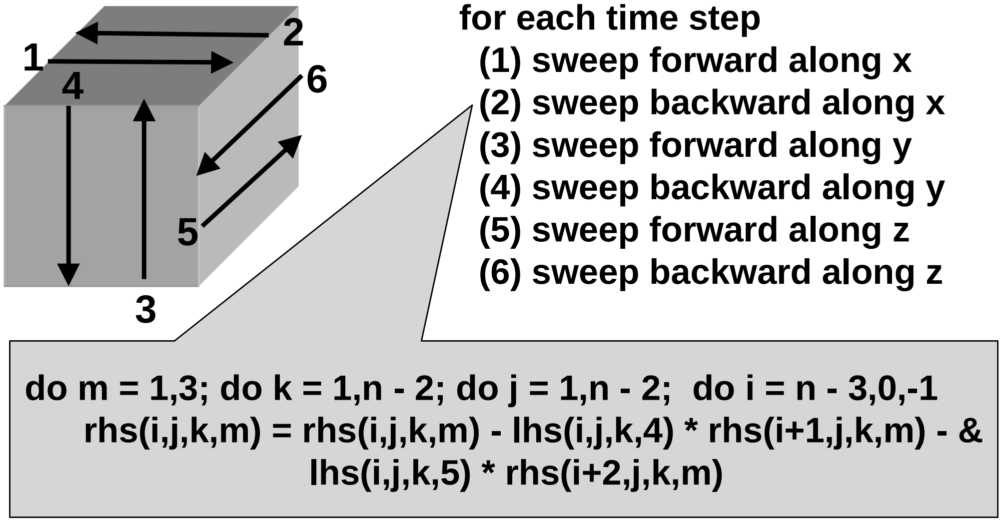
1 Introduction

Line sweeps are used to solve one-dimensional recurrences along a dimension of a multi-dimensional hyper-rectangular domain. This solution technique serves as the basis for Alternating Direction

---

[*] This work performed while a visiting scholar at Rice University.

Implicit (ADI) integration—a widely-used numerical technique for solving partial differential equations such as the Navier-Stokes equation [5, 6, 20, 22]. Figure 1 illustrates an ADI integration computation based on line sweeps over a three-dimensional data volume. In each time step, the computation alternately sweeps forward and backward over each spatial dimension of the data domain to solve a system of one-dimensional recurrences. The callout shows a code fragment that solves a recurrence using a reverse sweep over the x dimension. Besides ADI integration, line sweeps are also at the heart of a variety of other numerical methods and solution techniques [22].



```
for each time step
  (1) sweep forward along x
  (2) sweep backward along x
  (3) sweep forward along y
  (4) sweep backward along y
  (5) sweep forward along z
  (6) sweep backward along z
```

```
do m = 1,3; do k = 1,n - 2; do j = 1,n - 2;  do i = n - 3,0,-1
    rhs(i,j,k,m) = rhs(i,j,k,m) - lhs(i,j,k,4) * rhs(i+1,j,k,m) - &
                   lhs(i,j,k,5) * rhs(i+2,j,k,m)
```

Figure 1: A skeleton of three-dimensional ADI integration using line sweeps.

Parallelizing computations based on line sweeps is important because such programs address important classes of problems and they are computationally intensive. However, parallelizing multi-dimensional line-sweep computations is difficult because recurrences serialize the sweeps along each dimension. Using standard block partitionings, which assign a single hyper-rectangular volume of data to each processor, there are two reasonable parallelization strategies. A **static block unipartitioning** partitions one of the array dimensions for the entire computation. To achieve significant parallelism with this type of partitioning, one must exploit wavefront parallelism within each sweep along the partitioned dimension. In wavefront computations, there is a tension between using small messages to maximize parallelism by minimizing the length of pipeline fill and drain phases, and using larger messages to minimize communication overhead in the computation's steady state when the pipeline is full. A **dynamic block partitioning** involves partitioning some subset of the dimensions, performing line sweeps in all unpartitioned dimensions locally, and then transposing the data when necessary between sweeps so that each of the sweeps, in turn, can be performed locally. While a dynamic block partitioning achieves better efficiency during a (local) sweep over a single dimension compared to a (wavefront) sweep using a static block unipartitioning, the cost of its data transposes can be substantial.

To support better parallelization of line sweep computations over multi-dimensional arrays, a third sophisticated strategy for partitioning data and computation known as **multipartitioning** was developed [5, 6, 20, 22]. This strategy partitions arrays of $d \geq 2$ dimensions among a set of processors so that all processors are active in every step of a line-sweep computation along any array dimension, load-balance is nearly perfect and only coarse-grain communication is needed. A multipartitioning achieves this by (1) assigning each processor a balanced number of tiles in each hyper-rectangular slab defined by a pair of adjacent cuts along a partitioned data dimension and (2) ensuring that for all tiles mapped to a processor, their immediate tile neighbors in any one coordinate direction are all mapped to some other single processor. We refer to these two properties as the **balance** property, and the **neighbor** property respectively. The utility of the balance property is obvious: it enables even load balance. The neighbor property is useful because it enables a fully-vectorized, directional-shift communication to be accomplished with one message per processor. A study by van der Wijngaart [25] of strategies for hand-coded parallelizations of ADI Integration found that three-dimensional multipartitionings yield better performance than static block or dynamic block partitionings.

All of the multipartitionings described in the literature to date consider only one tile per processor per hyper-rectangular slab along a partitioned dimension. The most broadly applicable of the multipartitioning strategies in the literature is known as **diagonal multipartitioning**. In two dimensions, these partitionings can be performed on any number of processors, p; however, in three dimensions they are only useful if p is a perfect square. We consider the general problem of computing optimal multipartitionings for d-dimensional data volumes onto an arbitrary number of processors.

In the next section, we describe prior work in multipartitioning. In the subsequent sections, we present our strategy for computing generalized multipartitionings. This strategy has two principal parts: a cost-model-driven algorithm for computing the dimensionality and tile size of an optimal multipartitioning and an algorithm for computing a mapping of data tiles to processors. We present a proof and complexity analysis of the partitioning algorithm and a constructive proof for the tile to processor mapping. Finally, we describe a prototype implementation of generalized multipartitioning in the Rice dHPF compiler for High Performance Fortran. We report performance results obtained using it to parallelize a the NAS SP computational fluid dynamics benchmark.

2 Background

Johnsson *et al.* [20] describe a two-dimensional domain decomposition strategy, now known as a multipartitioning, for parallel implementation of ADI integration on a multiprocessor ring. They partition both dimensions of a two-dimensional domain to form a $p \times p$ grid of tiles using the tile-to-processor mapping $\theta(i, j) = (i - j) \bmod p$, where $0 \le i, j < p$. Using this mapping for an ADI computation, each processor exchanges data with only its two neighbors in a linear ordering of the processors, which maps nicely to a ring.

Bruno and Cappello [5, 6] devised a three-dimensional partitioning for parallelizing three-dimensional ADI integration computations on a hypercube architecture. They describe how to map a three-dimensional domain cut into $2^d \times 2^d \times 2^d$ tiles onto 2^{2d} processors using the tile-to-processor mapping $\theta(i, j, k)$ based on Gray codes. A Gray code [17] $g_s(r)$ denotes a one-to-one function defined for all integers r and s where $0 \le r < 2^s$, that has the property that $g_s(r)$ and $g_s((r+1) \bmod 2^s)$ differ in exactly one bit position. Bruno and Capello define $\theta(i, j, k) = g_d((j + k) \bmod 2^d) \cdot g_d((i + k) \bmod 2^d)$, where $0 \le i, j, k < 2^d$ and $\cdot$ denotes bitwise concatenation. This θ maps tiles adjacent along the i or j dimension to adjacent processors in the hypercube, whereas tiles adjacent along the k dimension map to processors that are exactly two hops distant. They also show that no hypercube embedding is possible in which adjacent tiles always map to adjacent processors.

Naik *et al.* [22] describe *diagonal multipartitionings* for two- and three-dimensional problems. Diagonal multipartitionings are a generalization of the Johnsson *et al.* two-dimensional partitioning strategy. This class of multipartitionings is also more broadly applicable than the Gray-code-based mapping described by Bruno and Cappello. The three-dimensional diagonal multipartitionings described by Naik *et al.* partition data into $p^{\frac{3}{2}}$ tiles, with each processor's tiles arranged along a wrapped diagonal through each of the partitioned dimensions. Figure 2 shows a three-dimensional multipartitioning of this style for 16 processors; the number in each tile indicates the processor that owns the block. In three dimensions, a diagonal multipartitioning is specified by the tile to processor mapping $\theta(i,j,k) = ((i-k) \bmod \sqrt{p})\sqrt{p} + ((j-k) \bmod \sqrt{p})$ for a domain of $\sqrt{p} \times \sqrt{p} \times \sqrt{p}$ tiles where $0 \le i,j,k < \sqrt{p}$.

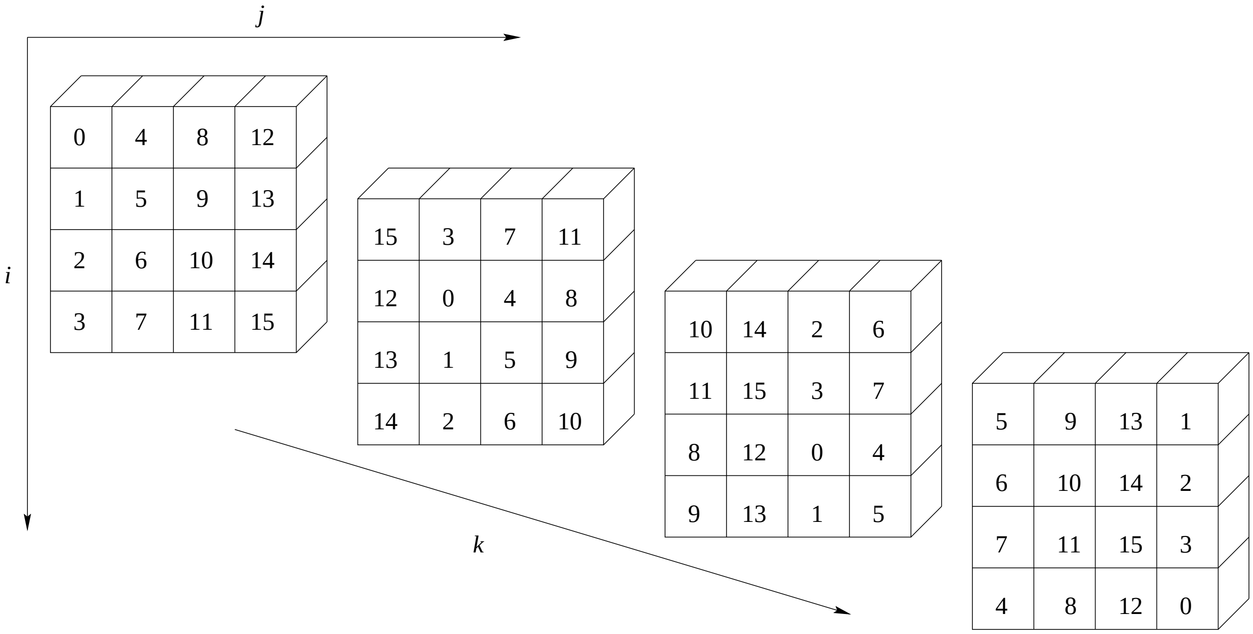


Figure 2: 3D Multipartitioning on 16 processors.

More generally, we observe that diagonal multipartitionings can be applied to partition d-dimensional data onto an arbitrary number of processors p by cutting the data into p slices in each dimension, i.e., into an array of p^d tiles. For two dimensions, this strategy yields an optimal multipartitioning (equivalent to the class of partitionings described by Johnsson *et al.* [20]). But, for $d > 2$, cutting data into so many tiles yields inefficient partitionings with excess communication, except when $p^{\frac{1}{d-1}}$ is integral.

3 Generalized Multipartitioning

Bruno and Cappello noted that multipartitionings need not be restricted to having only one tile per processor per hyper-rectangular slab of a multipartitioning [5], but they did not explore construction of partitionings of this style. We define a *generalized multipartitioning* as any partitioning of a hyper-rectangular domain in which the number of tiles in each hyper-rectangular slab defined by a pair of adjacent cuts along some partitioned dimension (hereafter, simply called a *hyper-rectangular slab*) is divisible by the number of processors and the partitioning also satisfies both the balance and neighbor constraints that we described informally in Section 1. We define generalized multipartitionings more precisely after we introduce some notation that we use throughout the rest of the paper.

- p denotes the number of processors. We write $p = \prod_{j=1}^{s} \alpha_j^{r_j}$, to represent the decomposition of p into prime factors. Each prime factor α_j of p occurs with some multiplicity r_j.

- P denotes the set of processors $\{1..p\}$.
- d is the number of dimensions of the array to be partitioned. The array is of size $n_1, \ldots, n_d$. The total number of array elements is $n = \prod_{i=1}^{d} n_i$.
- $\vec{\gamma}$ denotes an array partitioning. Each γ_i, $1 \leq i \leq d$, in $\vec{\gamma}$ represents the number of tiles into which the array is cut along its i-th dimension. We can consider a d-dimensional array as a $\gamma_1 \times \ldots \times \gamma_d$ array of tiles. In our analysis, we assume γ_i divides n_i evenly and do not consider the effect of load imbalance that arises if this assumption is not valid.

Using this notation, we define a generalized multipartitioning as an ordered pair $(\vec{\gamma}, \theta)$, where $\vec{\gamma}$ is an array partitioning (as defined above) and θ is a tile-to-processor mapping. Each tile in the array of tiles is represented by its coordinates $\vec{t}$, a d-dimensional vector with each element t_i, $1 \leq i \leq d$, such that $0 \leq t_i \leq \gamma_i - 1$. A *hyper-rectangular slab* H in dimension i is a subset of the array of tiles such that $\forall \vec{t} \in H$, $\forall \vec{u} \in H$, $t_i = u_i$.

To ensure that a balanced tile-to-processor mapping is possible for each hyper-rectangular slab in any dimension i, $1 \leq i \leq d$, the number of tiles in each such slab must be a multiple of p; namely, for each p should divide $\prod_{j \neq i} \gamma_j$. The tile-to-processor mapping $\theta(\vec{t})$ maps a tile's coordinates $\vec{t}$ to a processor in P. The mapping θ must satisfy two constraints for $(\vec{\gamma}, \theta)$ to be a multipartitioning. First, for any hyper-rectangular slab H in any dimension i, when θ is applied to each of the elements in the set H, it must map the coordinates of an equal number of tiles to each processor; namely, $\forall i \in P$, $\forall j \in P$, $j \neq i$, $|\theta^{-1}(i) \cap H| = |\theta^{-1}(j) \cap H|$. Second, there must exist a neighbor mapping such that for all tiles $\theta^{-1}(j)$ owned by a single processor j, $j \in P$, and any particular dimension i, $1 \leq i \leq d$, there exists a single processor that owns all tiles "right-adjacent" to any tile in $\theta^{-1}(j)$ along dimension i; similarly, there exists a single processor that owns all tiles "left-adjacent" to any tile in $\theta^{-1}(j)$ along dimension i. More formally, if $\vec{e}$ is a canonical basis vector, then we must have:

$$\theta(\vec{t}) = \theta(\vec{u}) \Rightarrow \theta(\vec{t} + \vec{e}) = \theta(\vec{u} + \vec{e})$$

when the tiles with coordinates $\vec{t}$, $\vec{u}$, $\vec{t} + \vec{e}$, and $\vec{u} + \vec{e}$ exist in the array of tiles.

In the following sections, we show how to select an optimal partitioning of a d-dimensional array into tiles and how to assign the tiles to processors so that the mapping satisfies the balance and neighbor constraints of a multipartitioning. We show that such a tile assignment is possible if and only if the number of tiles in each hyper-rectangular slab is a multiple of p ("if" being the difficult part of the proof). We describe a "regular" tile-to-processor mapping (regular to be defined) that satisfies the neighbor constraint.

In the next section, we define an objective function that represents the execution time of a parallel line-sweep computation over a multipartitioned array and present an algorithm that computes an optimal array partitioning with respect to this objective. In Section 5, we present a general theory of modular mappings for generalized multipartitioning and apply this theory to define a mapping of tiles to processors that satisfies the balance and neighbor constraints of multipartitionings.

4 Finding an Optimal Partitioning

We now develop a strategy for selecting an optimal partitioning for a multi-dimensional array onto an arbitrary number of processors. First, we define an objective function that serves as the basis for evaluating the relative cost of partitionings. Second, we cast partitioning as an optimization problem. Third, we discuss necessary properties of an optimal partitioning. Fourth, we present an algorithm for selecting an optimal partitioning by exhaustive enumeration of elementary partitionings that satisfy the criteria for optimality; we also present a proof of our algorithm's correctness. Finally, we analyze the complexity of our algorithm.

4.1 Objective Function

Consider the cost of performing a line sweep computation along each dimension of a multipartitioned array. The total computation cost is proportional to n, the number of elements in the array. A sweep along the i-th dimension consists of a sequence of γ_i computation phases, one for each hyper-rectangular slab of tiles along dimension i, separated by $\gamma_i - 1$ communication phases. Since a multipartitioning assigns a balanced amount of work to each processor in each phase of the computation, the total amount of work per processor will be roughly $\frac{1}{p}$ of the total computational work. The communication overhead, including additional computation associated with communication, is a function of the number of communication phases and the communication volume. In each communication phase, a boundary layer consisting of one or more hyperplanes of array elements is transmitted. The total communication volume in a sweep along dimension i is proportional to the number of array elements in each boundary hyperplane communicated, namely $\prod_{j=1,j\neq i}^{d} n_j = \frac{n}{n_i}$ elements. We assume here that the cost of one communication phase is an affine function of the total volume of data transmitted. The total execution time for a sweep along dimension i can be approximated by the following formula:

$$T_i(p,\gamma_i) = K_1\frac{n}{p} + (\gamma_i - 1)\left(K_2 + K_3\frac{nb_i}{n_i}\right)$$

where b_i is the number of boundary hyperplanes communicated between computations on adjacent slabs[1] and K_1, K_2, and K_3 are constants that depend, respectively, on the sequential computation time, the cost of initiating one communication phase, and the cost of transmitting one array element. If all processors can perform their communication without network contention, then the third (volume-dependent) term can be divided by p; while this would change the cost, it would not qualitatively change the way in which we use the cost function since we are reasoning for a fixed p; therefore, we don't consider this issue further. Using this model for each dimension, the cost of a complete iteration that sweeps across all of the dimensions is thus

$$T(p,\vec{\gamma}) = \sum_{i=1}^{d} T_i(p,\gamma_i) = d\left(K_1\frac{n}{p} - K_2 - K_3\sum_{i=1}^{d}\frac{nb_i}{n_i}\right) + \sum_{i=1}^{d}\gamma_i\left(K_2 + K_3\frac{nb_i}{n_i}\right)$$

Assuming that p, n, and the n_i's and b_i's are given, the only term that can be optimized is $\sum_{i=1}^{d}\gamma_i\lambda_i$, where $\lambda_i = K_2 + K_3\frac{nb_i}{n_i}$ is a constant that depends upon the domain size, the number of boundary layers communicated along that dimension, and the machine's communication parameters.

There are several cases to consider. If the number of phases (through K_2) is the critical term, the objective function can be simplified to $\sum_i \gamma_i$. If the volume of communications is the critical term, the objective function can be simplified to $\sum_i \frac{\gamma_i b_i}{n_i}$.

To illustrate the tradeoffs when partitioning data arrays with more than two dimensions, we consider the simple case of partitioning a three-dimensional data volume among 4 processors when only one boundary layer is communicated for a sweep along any dimension ($\forall_i$, $b_i = 1$). If the

---

[1] To clarify the meaning of b_i, we note that for the example shown in Figure 1, $b_1 = 2$ since the subscripts $i+1$ and $i+2$ cause the right hand side references to access two off-processor planes of data.

data domain has one dimension that is much smaller than the others, then it will be less costly to use a two-dimensional partitioning of the two larger dimensions rather than a three dimensional partitioning. For instance, if n_1 and n_2 are at least 4 times larger than n_3, then cutting each of the first two dimensions into 4 pieces ($\gamma_1 = \gamma_2 = 4$, $\gamma_3 = 1$) leads to a lower communication volume than a three-dimensional partitioning in which each dimension is cut into 2 pieces ($\gamma_1 = \gamma_2 = \gamma_3 = 2$). In this case, the cost of the extra communication phases for sweeps along the first two dimensions is offset by the absence of communication in a sweep along the unpartitioned third dimension.

4.2 The Partitioning Problem

To determine an optimal partitioning, we must minimize $\sum_i \gamma_i \lambda_i$ for general λ_i's, with the constraint that, for any fixed i, p divides $\prod_{j \neq i} \gamma_j$. From a theoretical point of view, we do not know whether this minimization problem is NP-hard, even for a fixed number of dimensions $d \geq 3$ with all λ_i equal to 1, or if a polynomial algorithm in $\log p$ exists. The only NP-hard problem we found in the literature related to product of numbers is the Subset Product Problem (SPP) [16], which is weakly NP-hard; it can be solved using dynamic programming. We suspect that this problem is strongly NP-hard. However, if p has only one prime factor, a greedy approach leads to an algorithm that is polynomial in r_1 and d, which we describe in a technical report [12].

Without an algorithm that exploits the structure of the cost function, we are left with no alternative but to search through the space of potential partitionings and select one for which the cost is minimal. If we naively consider partitioning each dimension into between 1 and p intervals (more than p partitions in a dimension would be pointless since our balance property could be satisfied with p), this gives us a search space of size p^d, which is much too large to search quickly for large p. In the next section, we examine the properties that an optimal partitioning must possess with the goal of narrowing the scope of our search.

4.3 Elementary Partitionings

We say that $\vec{\gamma}$ is a **candidate partitioning** if, for each $1 \leq i \leq d$, p divides $\prod_{j=1, j \neq i}^{d} \gamma_j$. If $\sum_i \gamma_i \lambda_i$ is minimized, we say that $\vec{\gamma}$ is an **optimal partitioning**. We show some useful properties of candidate and optimal partitionings.

Lemma 1 *Let $\vec{\gamma}$ be given. Then, $\vec{\gamma}$ is a candidate partitioning if and only if, for each factor α_j of p, appearing r_j times in the factorization of p, the total number of occurrences of α_j in all γ_i is at least $r_j + m_j$, where m_j is the maximum number of occurrences of α_j in any γ_i.*

Proof: Suppose that $\vec{\gamma}$ is a candidate partitioning. Let α_j be a factor of p appearing r_j times in the decomposition of p, let m_j be the maximum number of occurrences of α_j in any γ_i, and let k be such that α_j appears m_j times in γ_k. Since p divides $\prod_{i \neq k} \gamma_i$, α_j appears at least r_j times in this product. The total number of occurrences of α_j in all of the γ_i is thus at least $r_j + m_j$. Conversely, if this property is true for any factor α_j, then for any product of $(d-1)$ different γ_i's, the number of occurrences of α_j is at least $r_j + m_j$ minus the number of occurrences in the γ_i that is not part of the product, and thus is at least r_j. Therefore, p divides this product and $\vec{\gamma}$ is a candidate partitioning. ■

Lemma 1 enables us to view a candidate partitioning $\vec{\gamma}$ as a distribution of the factors of p into d bins, each representing a γ_i, $1 \leq i \leq d$. If a factor α_j appears r_j times in p, it should appear $(r_j + m_j)$ times in the d bins, where m_j is the maximal number of occurrences of α_j in a bin.

The following lemma shows that, for an optimal partitioning, there should be exactly $(r_j + m_j)$ occurrences for each factor α_j and that the maximum m_j should appear in at least two bins.

Lemma 2 *Let $\vec{\gamma}$ be an optimal partitioning. Then, each factor α_j of p, appearing r_j times in the decomposition of p, appears exactly $(r_j + m_j)$ times in $\vec{\gamma}$, where m_j is the maximum number of occurrences of α_j in any particular γ_i. Furthermore, the number of occurrences of α_j is m_j in at least two γ_i's.*

Proof: Let $\vec{\gamma}$ be an optimal partitioning. By Lemma 1, each factor α_j, $0 \leq j < s$, that appears r_j times in p, appears at least $(r_j + m_j)$ times in $\vec{\gamma}$. The following arguments hold independently for each factor α_j.

Suppose m_j occurrences of α_j appear in some γ_k and in no other $\gamma_i, i \neq k$. Remove one α_j from γ_k. Now, the maximum number of occurrences of α_j in any γ_i is $m_j - 1$ and we have $(r_j + m_j) - 1 = r_j + (m_j - 1)$ occurrences of α_j. By Lemma 1, we still have a candidate partitioning, and with a lower cost. This contradicts the optimality of $\vec{\gamma}$. Thus, there are at least two bins with m_j occurrences of α_j.

If c, the number of occurrences of α_j in $\vec{\gamma}$, is such that $c > r_j + m_j$, then we can remove one α_j from any nonempty bin. We now have $c - 1 \geq r_j + m_j$ occurrences of α_j and the maximum is still m_j (since at least two bins had m_j occurrences of α_j). Therefore, according to Lemma 1, we still have a candidate partitioning, and with a lower cost, again a contradiction. ■

We call the candidate partitionings that satisfy Lemma 2 **elementary partitionings**. These partitionings are the minima of the partial order defined on candidate partitionings with the relation $\vec{\gamma} \leq \vec{\gamma'}$ if γ_i divides γ'_i for all i. Since all λ_i are positive, only elementary partitionings need to be considered to find an optimal partitioning. Note, however, that there are elementary partitionings that can never be optimal. For example, in 3D, for $p = 2^2 \times 3^3 \times 5^2$, the elementary partitioning $(2^2 \times 3^2, 2^2 \times 5^2, 3^2 \times 5^2) = (36, 100, 225)$, and its permutations, can never be optimal since the partitioning $(30, 30, 30)$ is always better, whatever the positive λ_i's. However, if the largest factor were 7 instead of 5, such a partitioning will be optimal for some λ_i's, since 36 is less than $2 \times 3 \times 7 = 42$.

We present some upper and lower bounds for the maximal number of occurrences of a given factor in any bin.

Lemma 3 *In any optimal partitioning, for any factor α_j appearing r_j times in the decomposition of p, we have $\lceil \frac{r_j}{d-1} \rceil \leq m_j \leq r_j$ where m_j is the maximal number of occurrences of α_j in any bin and d is the number of bins.*

Proof: By Lemma 2, we know that the number of occurrences of α_j is exactly $r_j + m_j$, and at least two bins contain m_j elements. Thus, $r_j + m_j = 2m_j + e$, in other words $r_j = m_j + e$, where e is the total number of elements in $(d-2)$ bins, excluding two bins of maximal size m_j. Since $0 \leq e \leq (d-2)m_j$, then $m_j \leq r_j \leq (d-1)m_j$ which is equivalent to the desired inequality, since m_j is an integer. ■

4.4 Exhaustive Enumeration of Elementary Partitionings

To compute an optimal partitioning $\vec{\gamma}$, we need to consider only elementary partitionings. We compute an optimal partitioning by enumerating all elementary partitionings and selecting one with minimal cost according to our objective function. To compute all elementary partitionings, we

```
// inputs
//   r - the multiplicity of some factor of p
//   d - the number of dimensions to consider partitioning, d >= 2
// output
//   bin[1..d] - the multiplicity of instances of the factor for each dimension
// global variable
//   bin[1..d] whose initial values are ignored
void Partitions(int r, int d) {
  int m;
  for (m = (r+d-2)/(d-1); m <= r; m++) {
    P(r+m,m,2,d);
  }
}

void P(int n, int m, int c, int d) {
  int i;
  if (d==1) {
    bin[d] = n; // bin[1..d] represents an elementary partitioning of a single factor
    OutputFactorPartitioning(bin,d); // outputs contents of bin[1..d]
  }
  else {
    for (i=max(0,n-m*(d-1)); i<=min(m-1,n-c*m); i++) {
      bin[d] = i;
      P(n-i,m,c,d-1);
    }
    if (n>=m) {
      bin[d] = m;
      P(n-m,m,max(0,c-1),d-1);
    }
  }
}
```

Figure 3: Program for Generating All Possible Distributions for One Factor.

first decompose p into prime factors using a standard algorithm. Then, we essentially enumerate all elementary partitionings for each factor alone and then consider all possible combinations that arise when the independent factors are considered together. For each resulting elementary partitioning containing all of the factors, we evaluate our cost function and select a partitioning with minimal cost.

The code shown in Figure 3 enumerates all elementary partitionings for a single factor of p with multiplicity r as distributions of factor instances into d bins. This program is inspired by a program [23] for generating all partitions of a number—a partitioning problem that has been well-studied (see [3, 24]) since the mathematical work of Euler and Ramanujam. The procedure `Partitions` first selects the maximal number m of factor instances that will appear in a bin, and uses the recursive procedure `P(n,m,c,d)` to generate all distributions of n elements in d bins, from index 1 to index d, where each bin can have at most m elements and at least c bins should have m elements. The initial call is `P(r+m,m,2,d)`.

We now prove that this program enumerates the desired elementary partitionings. Let us first consider the loop in function `Partitions`. From Lemma 3, we know that the maximal number of elements in a bin is between $\lceil \frac{r}{d-1} \rceil = \frac{r+d-2}{d-1}$ and r. For each such m, there is at least one elementary partitioning with $(r+m)$ elements and two maxima equal to m. The partitioning in which the last two bins have m elements and the $(d-2)$ other bins contain a total of $(r-m)$ elements is one such partitioning. In this partitioning, a legal distribution of the $(r-m)$ elements into the leftmost $(d-2)$ bins is to let $q = \lfloor \frac{r-m}{m} \rfloor$ bins contain m elements and another one contains $(r-m-mq)$ elements. Therefore, if the function `P` is correct, the function `Partitions` is also correct.

To prove the correctness of the function `P`, we prove by induction on d (the number of bins) that there is at least one elementary partitioning if and only if $0 \leq c \leq d$ and $cm \leq n \leq dm$ and that `P` generates all of them, enumerating each one only once. These conditions are simple to understand: we need at least cm elements (so that at least c bins have m elements) and at most dm elements, otherwise at least one bin will contain more than m elements.

The terminal case is clear: if we have only one bin and n elements to distribute, the bin should contain n elements. Furthermore, if there is a partitioning, we should have $c \leq 1$ and $n = m$ if $c = 1$, i.e., $c \leq d$ and $cm \leq n \leq dm$.

The general case is trickier. We first select the number of elements i in the bin number d and recursively call `P` for the first $(d-1)$ remaining bins. If we select fewer than m elements, we will still have to select c bins with m elements for the remaining $(d-1)$ bins, with $(n-i)$ elements. Therefore, the number i that we select should be neither too small nor too large, and we should have $cm \leq n-i \leq m(d-1)$ (by induction hypothesis), i.e., $n-(d-1)m \leq i \leq n-cm$. Furthermore, i should be strictly less than m, nonnegative, and less than n. Since c is always nonnegative, the constraint $i \leq n-cm$ ensures $i \leq n$. If the parameters are correct for the bin number d, we also have $c \leq d$ and if $c = d$, then the loop has no iteration, thus for any i selected in the loop, we have $c \leq d-1$. Therefore, the recursive call `P(n-i,m,c,d-1)` has correct parameters. Finally, selecting m elements for the bin d is possible only if m is less than n, and then the remaining $(n-m)$ elements have to be distributed into $(d-1)$ bins, with a maximum of m, and at least $\max(0, c-1)$ maxima. Again, the recursive call has correct parameters, since we decreased both c and d and removed m elements.

In the next two sections, we evaluate the complexity of our algorithm. The principal challenge is to determine the bounds on the number of possible elementary partitionings. Our algorithm for exhaustively enumerating the elementary partitionings of a single factor α_j is exponential in r_j, the multiplicity factor. Using this building block to compute elementary partitionings of all factors yields an algorithm that is also exponential in s, the number of unique prime factors of p, but whose complexity in p grows slowly. In practice, the algorithm is much faster than the exponential worst case may suggest, both because the average complexity is much lower and because p is not huge in practice.

4.4.1 Complexity Issues: First Hints

Here we consider some simple upper bounds on the number of elementary partitionings $S(p)$ for an integer p. We are not interested in an exact formula, but in the order of magnitude. In cases of practical interest, we expect that the number of bins d will be small, i.e., 3, 4, or 5, but p may be large, up to several thousand nodes.

Let us first try a rough estimation. For each factor α_j of p, we need to put $(r_j + m_j)$ instances of α_j into d bins; by Lemma 3, m_j is at most r_j. This implies that we need to distribute at most $2r_j$ instances of α_j into the bins. Each factor instance can be put in d different bins; thus, there are at most d^{2r_j} possibilities for each factor α_j, and $d^{(2\sum_j r_j)}$ for all factors. This is at most $d^{2\log_2 p}$ in the worst case, when p is a power of 2, which is equal to $p^{2\log_2 d}$.

We can compute a tighter bound if we consider the structure of the distribution of the factors dictated by Lemma 2. For each each factor α_j, we must select two bins that will contain m_j elements, for which we have $\frac{d(d-1)}{2}$ choices. The other $(r_j - m_j)$ elements must be distributed among the remaining bins, which can be done in at most $(d-2)^{r_j - m_j}$ ways. When $d = 3$, m_j must satisfy $\lceil \frac{r_j}{2} \rceil \leq m_j \leq r_j$; thus, the number of different multiplicities of α_j that we must consider distributing is thus $3(\lfloor \frac{r_j}{2} \rfloor + 1) \leq 3r_j$; therefore, the total number of possible partitionings when considering the combination of all possible distributions of each of the factors must be less than $3^s \prod_{i=1}^{s} r_j$. We come back to such an expression later when discussing our final upper bound. When $d > 3$, the number of possibilities for each factor α_j is less than:

$$\begin{aligned}\sum_{m=\lceil \frac{r_j}{d-1} \rceil}^{r_j} \frac{d(d-1)}{2}(d-2)^{r_j-m_j} &\leq \frac{d(d-1)}{2}\sum_{m_j=1}^{r_j}(d-2)^{r_j-m_j} = \frac{d(d-1)}{2}\sum_{i=0}^{r_j-1}(d-2)^i \\ &\leq \frac{d(d-1)}{2}\frac{(d-2)^{r_j}-1}{d-3} \leq \frac{d(d-1)(d-2)^{r_j}}{2(d-3)}\end{aligned}$$

When p has only a single factor, such as 2, the number of partitionings can be of order $(d-2)^{\log_2 p}$, i.e., $p^{\log_2(d-2)}$ which is roughly the square root of what we had previously. This is better, but still a polynomial in p. We would like a tighter bound.

In the previous approach, we evaluated the number of possible positions for each element to be put in the d bins. A dual view is to count, for each bin, how many factor instances it can contain. This view leads to a sharper upper bound when d is small and r_j can be large. We can select for each bin a number between 0 and r_j, thus $(r_j + 1)$ possibilities. The total number of partitionings is less than $(r_j + 1)^d$ for each factor, thus less than $(\log_2 p + 1)^d$ if p is a power of 2, and $S(p) \leq (\prod_j (r_j + 1))^d$ in the general case. When r_j is larger than $d \geq 3$, $(r_j + 1)^d$ is less than $(d + 1)^{r_j}$, which is roughly our previous upper bound. With this scheme, we are more likely to obtain a better evaluation of the total number of partitionings.

In Section 4.4.2 we show that $S(p) = O\left(\left(\frac{d(d-1)}{2}\right)^{\frac{(1+o(1))\log p}{\log\log p}}\right)$, which is a better upper bound when d small and p can be large, and that this bound is tight in order of magnitude. It is no surprise that this expression is similar to the number of dividers of an integer (see [19]), since $\prod_j (r_j + 1)$ is indeed the number of dividers of p.

4.4.2 Complexity for Computing an Optimal Partitioning

Here we consider the cost of each of the steps for computing an optimal partitioning. Decomposing p into prime factors using a standard algorithm has a complexity of $O(\sqrt{p})$. We use the `Partitions` function to generate, for each factor, all elementary partitionings: those that satisfy Lemma 2. We combine them while keeping track of the best overall partitioning. For evaluating each partitioning, we need to build the corresponding γ_i's and add them. Each γ_i is at most p and is obtained by at most $\sum_j r_j \leq \log_2 p$ multiplications of numbers less than p. Therefore, building each γ_i costs at most $(\log_2 p)^3$. The overall complexity, excluding the cost of the decomposition of p into prime factors, is thus the product of $(\log_2 p)^3$ and the complexity of the function `Partitions`, which corresponds to the number of partitionings generated by the algorithm since each partitioning is generated only once and the amount of work performed by the function is linearly proportional to the number of partitionings it enumerates. Therefore, it remains to evaluate the number of partitionings $S(p)$ generated by the function `Partitions`, i.e., the number of elementary partitionings.

Consider first the case of a number p, product of unique prime factors, in particular the product of the first s prime numbers: $p = \prod_{i=1}^{s} \pi_i$ where π_i is the i-th prime number. For each factor, there are $\frac{d(d-1)}{2}$ possible distributions, in which two bins each have one element; therefore, the total number of partitionings is $\left(\frac{d(d-1)}{2}\right)^s$. Now, the i-th prime number is equivalent to $i \log i$ (see for example [19]). Therefore, when p grows, we have

$$\log p = \sum_{i=1}^{s} \log \pi_i \sim \sum_{i=1}^{s} \log(i \log i) \sim \sum_{i=1}^{s} \log i \sim \int_1^s \log x \, \mathrm{d}x \sim s \log s$$

since divergent series with nonnegative equivalent terms are equivalent. Therefore, $\log p \sim s \log s$ and $\frac{\log p}{\log \log p} \sim s$. The total number of partitionings for p is thus $\left(\frac{d(d-1}{2}\right)^{\frac{\log p}{\log \log p}(1+o(1))}$. We will prove later that this situation (p is the product of unique prime factors) is actually representative of the worst case, in order of magnitude.

Before that, we can give several simpler upper bounds, exploiting well-known results concerning $d(p)$ the number of dividers of p (see again the book of Hardy and Wright [19, Chap. XVIII]). The number of dividers of $p = \prod_{i=1}^{s} \alpha_i^{r_i}$ is $d(p) = \prod_{i=1}^{s}(r_i + 1)$. For all $\epsilon > 0$, for p sufficiently large, $d(p) \leq 2^{(1+\epsilon) \log p / \log \log p}$. Furthermore, the average order of $d(p)$, which is $(\sum_{i=1}^{p} d(i))/p$, is equivalent to $\log p$, while the average order of $d(p)^d$ is of order $(\log p)^{2^d - 1}$. As seen before, the number of elementary partitionings $S(p)$ satisfies $S(p) \leq d(p)^d$. We can deduce immediately the following upper bounds:

- For all $\epsilon > 0$, for p large enough, $S(p) \leq 2^{d(1+\epsilon) \log p / \log \log p}$. Thus $\log S(p) = O(\log p / \log \log p)$.
- Combined with the previous lower bound, when p is the product of first primes, $\log(d(d-1)/2) \leq \limsup \{\log S(p) \log \log p / \log p\} \leq d \log 2$.
- The average order of $S(p)$ is less than $(\log p)^{2^d - 1}$.

In other words, the worst-case complexity of the exhaustive search is not polynomial in $\log p$, but is smaller than any function p^r for any real number $r > 0$. Furthermore, on average, the behavior of the search is that of a polynomial (in $\log p$) algorithm. We did not try to get an equivalent of the average order of $S(p)$, i.e., an exact order of magnitude. However, it remains to compute the exact order of magnitude of the worst case, i.e., to show that $\limsup \{\log S(p) \log \log p / \log p\} = \log(d(d-1)/2)$. Actually, we only need to find an upper bound better than the one obtained using $d(p)$.

Theorem 1 *For $\epsilon > 0$, when d is fixed and large p,* $\log S(p) \leq (1+\epsilon) \log(d(d-1)/2) \log p / \log \log p$.

Proof: Let $C(r)$ be the number of partitionings generated by a factor appearing r times in the decomposition of p. The total number of partitionings $S(p)$ is $\prod_{i=1}^{s} C(r_i)$. To find an upper bound for $S(p)$, we use a similar technique as in [19, page 261-262], which consists of finding an upper bound for:

$$\frac{S(p)}{p^\delta} = \prod_{i=1}^{s} \left(\frac{C(r_i)}{\alpha_i^{\delta r_i}} \right)$$

for any particular $\delta > 0$, and δ will then be chosen as a function of p.

We first find an upper bound individually for each $C(r)/\alpha^{\delta r}$. When it is important to be accurate for a small r, we will use upper bounds in d^r (this also makes the computation simpler), and we will use upper bounds in r^d when r can be large (remember that $r^d \leq d^r$ when $r \geq d \geq 3$). First note that, roughly, $C(r)$ is less than $(r+1)^d$. Furthermore, since $1/e \leq \log 2$ and $x \leq e^{x/e}$, we have $x \leq e^{x \log 2} = 2^x$. Thus, for any $\delta > 0$:

$$C(r)/\alpha^{r\delta} \leq (r+1)^d/2^{r\delta} = ((r+1)/2^{r\delta/d})^d \leq (1 + r/2^{r\delta/d})^d \leq (1 + d/\delta)^d \tag{1}$$

This is true whatever r and α.

When α is large, we can bound the number of partitionings in a more accurate way. When $r = 1$, the number of generated partitionings is $d(d-1)/2$. When $r \geq 2$ and $d \geq 4$, we saw (Section 4.4.1) that $C(r) \leq \frac{d(d-1)(d-2)^r}{2(d-3)}$, and for $d = 3$, the number of partitionings is less than $rd(d-1)/2 = 3r$. When $d \geq 4$ and $r \geq 2$, it is easy to see that the function $\log(\frac{d(d-1)(d-2)^r}{2(d-3)})/r$ is maximal for $r = 2$. Furthermore, for $r = 2$, we have:

$$\frac{d(d-1)(d-2)^2}{2(d-3)} \leq \left(\frac{d(d-1)}{2}\right)^2 \Leftrightarrow \frac{(d-2)^2}{d-3} \leq \frac{d(d-1)}{2} \Leftrightarrow 2(d-2)^2 \leq d(d-1)(d-3)$$

But when $d \geq 5$, $2 \leq d-3$ and $(d-2)^2 \leq d(d-1)$ and when $d = 4$, $2(d-2)^2 = 8$ and $d(d-1)(d-3) = 12$. Thus, for all $r \geq 2$, we have $\log(C(r))/r \leq \log(d(d-1)/2) = \log C(1)$. And

$$\alpha \geq C(1)^{1/\delta} \Rightarrow \alpha^{r\delta} \geq C(1)^r \Rightarrow \frac{C(r)}{\alpha^{r\delta}} \leq \frac{C(r)}{C(1)^r} \leq 1 \tag{2}$$

We use Inequality (1) when $\alpha \leq C(1)^{1/\delta}$ and Inequality (2) otherwise. Since there are at most $C(1)^{1/\delta}$ different α's in the first case, we get $\frac{S(p)}{p^\delta} \leq (1 + d/\delta)^{d\,C(1)^{1/\delta}}$, thus:

$$\log S(p) \leq d\,C(1)^{1/\delta} \log(1 + d/\delta) + \delta \log p \leq C(1)^{1/\delta} d^2/\delta + \delta \log p$$

With $\delta = (1 + \epsilon/2) \log C(1) / \log\log p$, we get:

$$\log S(p) \leq \frac{d^2}{(1+\epsilon/2)\log C(1)} (\log p)^{1/(1+\epsilon/2)} \log\log p + (1+\epsilon/2) \log C(1) \log p / \log\log p$$

For large p, the first term is $o(\log p / \log\log p)$, therefore for p large enough, we get:

$$\log S(p) \leq (1+\epsilon) \log C(1) \log p / \log\log p$$

which is the desired inequality. ■

5 Finding a Tile-to-processor Mapping

In Section 4, we described how, given a number of processors and the dimensions of an array, we compute an array partitioning that minimizes an objective function [2]. A partitioning specifies an array (of tiles) whose size is $\vec{\gamma}$ for which the objective is minimized. So far, we have assumed that a *multipartitioning tile-to-processor assignment* can be computed. Such an assignment maps tiles to processors in a way that satisfies the key multipartitioning properties; namely, each processor will have the same number of tiles assigned to it in each slab of the array, and a multipartitioning

---

[2] We use an objective function that measures the $(d-1)$ dimensional 'surface' area of the cuts—an approximation of communication cost for a line sweep computation using the partitioning.

"neighbor mapping" function exists for each processor. This assumption has not yet been proven valid. An assignment with the first property is a **F-hyper-rectangle**, a generalization of the notion of **latin square** that is described in the second reference book on latin squares by Dénes and Keedwell [13, page 392]. Despite this reference, we have not found any paper that gives a method for constructing such an assignment, or even an existence proof, for our general case. Furthermore, even if such a proof exists, which we are not aware of, the constructive proof we give below is of interest to us because:

- it has the neighbor property,
- the tile-to-processor mapping is given by a simple formula, and conversely, for each processor, the list of tiles assigned to it can be easily formulated, which is very desirable for code generation,
- it gives a new insight to the properties of "modular" mappings (defined below).

By our partitioning algorithm, $\vec{\gamma}$ is an elementary partitioning. All elementary partitionings possess the property that for each $1 \leq i \leq d$, p divides $\prod_{j \neq i} \gamma_j$. This property is obviously a necessary condition. We prove in this section that this is also a sufficient condition. For any candidate partitioning $\vec{\gamma}$, optimal or not (with or without the additional property of Lemma 2), we can compute a valid multipartitioning tile-to-processor assignment for the tiles defined by the partitioning. To accomplish this, we use *modular mappings*, which we define in the following section (Section 5.1). In Section 5.2, we review some results on the validity of modular mappings that were shown previously by Darte, Dion, and Robert [11], along with some additional properties that are more important to our problem at hand. Finally, in Section 5.3, we give a constructive proof of the existence of a multipartitioning tile-to-processor assignment using modular mappings. The solution we build is one particular assignment. It is not unique and experiments might show that other assignments may yield faster execution times because of a difference in communication costs associated with their neighbor mappings. However, our current objective function (Section 4.1) does not account for network topology when constructing tile-to-processor mappings, so all valid mappings are considered equally good.

5.1 Modular Mappings

Consider the assignment in Figure 2. We have to find a formula that describes it. Let us try an assignment in the form of a linear [3] mapping modulo p (the number of processor), $ax+by+cz \bmod p$ (a, b, and c can be chosen between 0 and $p-1$, i.e., modulo p), and processors can be arranged differently, as long as the load-balancing property is satisfied.

To simplify the discussion, let us first consider a "smaller" example, with $p = 4$ and $\gamma_1 = \gamma_2 = \gamma_3 = 2$. Consider the first horizontal slab (for $z = 0$): the four numbers, 0, 1, 2, and 3 should appear exactly once, and the 0 is in position $(0,0)$. First, a and b are nonzero, otherwise 0 appears twice, either in the first row, or in the first column. Furthermore, if 1 appears in position $(0,1)$, then 3 cannot appear in position $(1,0)$, otherwise $1+3 = 0 \bmod 4$ appears in position $(1,1)$, which again is not acceptable. Therefore, either $a = 1$ and $b = 2$ (and the symmetric case), or $a = 2$ and $b = 3$ (and the symmetric case). The possible values for c have to be found. Obviously c is not 0 otherwise 0 appears twice in the slab $y = 0$ for example. Consider the first case $a = 1$ and $b = 2$. If $c = 1$, then 1 appears twice in the slab $y = 0$ (for $(1,0,0)$ and for $(0,0,1)$). If $c = 2$, then 2 appears

---

[3] Adding a constant, i.e., considering an affine mapping is not more powerful. Numbers are just all shifted by the same amount.

twice in the slab $x = 0$ (for $(0, 1, 0)$ and for $(0, 0, 1)$. And if $c = 3$, then 0 appears twice in the slab $y = 0$ (for $(0, 0, 0)$ and for $(1, 0, 1)$). Now, in the second case, if $a = 2$ and $b = 3$, the possible values for c have to be determined. A similar study shows that $c = 1$ is not possible (conflict for the slab $x = 0$), $c = 2$ is not possible (conflict for the slab $y = 0$), and $c = 3$ is not possible (conflict for the slab $x = 0$).

The small example we just studied (for an array of size $2 \times 2 \times 2$ and four processors) is an example for which the class of one-dimensional modular mappings (a linear map modulo the number of processors) is not large enough to contain a valid solution. We will therefore study a larger class of mappings, the class of **multi-dimensional modular mappings** $M_{\vec{m}} \; : \; \mathbb{Z}^d \longrightarrow \mathbb{Z}^{d'}$ defined by $M_{\vec{m}}(\vec{i}) = (M\vec{i}) \bmod \vec{m}$ (the modulo is component-wise) where $\vec{i}$ is the vector of coordinates in the array of tiles (an integral d-dimensional vector), M is an integral $d' \times d$ matrix, and $\vec{m}$ is an integral positive vector of dimension d'. Each tile is assigned to a "processor number" in the form of a vector. The product of the components of $\vec{m}$ is equal to the number of processors. It then remains to define a one-to-one mapping from the hyper-rectangle $\{\vec{j} \in \mathbb{Z}^{d'} \mid \vec{0} \leq \vec{j} < \vec{m}\}$ (inequalities component-wise) onto the processor numbers. This can be done by viewing the processors as a virtual grid of dimension d' of size $\vec{m}$. The mapping $M_{\vec{m}}$ is then an assignment of each tile (described by its coordinates in the d-dimensional array of tiles) to a processor (described by its coordinates in the d'-dimensional virtual grid).

Note: in Section 5.3, we will consider only the case $d' = d - 1$. Modular mappings into a space of dimension lower than $(d - 1)$ are captured by this representation when some components of $\vec{m}$ are equal to 1. Modular mappings into a space of larger dimension could also be studied in a similar way, but they are not needed for our purpose.

Consider again Figure 2. There are 16 processors that can be represented as a 2-dimensional grid of size 4×4. For example, the processor number $7 = 4 + 3$ can be represented as the vector $(3, 1)$. In general, they can be expressed as (r, q) where r and q are the rest and the quotient, respectively, of the Euclidian division by 16. The assignment in the figure corresponds to the modular mapping $(i - k \bmod 4, j - k \bmod 4)$.

The following definitions summarize the notions of modular mappings that satisfy the load-balancing property.

Definition 1 (Modular mapping)

A modular mapping $M_{\vec{m}} \; : \; \mathbb{Z}^d \longrightarrow \mathbb{Z}^{d'}$ *is defined as* $M_{\vec{m}}(\vec{i}) = (M\vec{i}) \bmod \vec{m}$ *where* M *is an integral* $d' \times d$ *matrix and* $\vec{m}$ *is an integral positive vector of dimension* d'.

Definition 2 (Rectangular index set)

Given a positive integral vector $\vec{b}$, *the rectangular index set defined by* $\vec{b}$ *is the set* $\mathcal{I}_{\vec{b}} = \{\vec{i} \in \mathbb{Z}^n \mid 0 \leq \vec{i} < \vec{b}\}$ *(component-wise) where* n *is the dimension of* $\vec{b}$.

Definition 3 (Slab)

Given a rectangular index set $\mathcal{I}_{\vec{b}}$, *a slab* $\mathcal{I}_{\vec{b}}(i, k_i)$ *of* $\mathcal{I}_{\vec{b}}$ *is defined as the set of all elements of* $\mathcal{I}$ *whose* i*-th component is equal to* k_i *(an integer between* 0 *and* $b_i - 1$*).*

Definition 4 (One-to-one modular mapping)

Given a hyper-rectangle (or any more general set) $\mathcal{I}_{\vec{b}}$, *a modular mapping* $M_{\vec{m}}$ *is a one-to-one mapping from* $\mathcal{I}_{\vec{b}}$ *onto* $\mathcal{I}_{\vec{m}}$ *if and only if for each* $\vec{j} \in \mathcal{I}_{\vec{m}}$ *there is one and only one* $\vec{i} \in \mathcal{I}_{\vec{b}}$ *such that* $M_{\vec{m}}(\vec{i}) = \vec{j}$.

Definition 5 (Equally-many-to-one modular mapping) *Given a hyper-rectangle (or any more general set)* $\mathcal{I}_{\vec{b}}$*, a modular mapping* $M_{\vec{m}}$ *is a equally-many-to-one modular mapping from* $\mathcal{I}_{\vec{b}}$ *onto* $\mathcal{I}_{\vec{m}}$ *if and only if the number of* $\vec{i} \in \mathcal{I}_{\vec{b}}$ *such that* $M_{\vec{m}}(\vec{i}) = \vec{j}$ *does not depend on* $\vec{j}$*.*

Definition 6 (Load-balancing property)

Given a rectangular index set $\mathcal{I}_{\vec{b}}$*, a modular mapping* $M_{\vec{m}}$ *has the load-balancing property for* $\mathcal{I}_{\vec{b}}$ *if and only if for any slab* $\mathcal{I}_{\vec{b}}(i, k_i)$*, the restriction of* $M_{\vec{m}}$ *to* $\mathcal{I}_{\vec{b}}(i, k_i)$ *is a equally-many-to-one mapping onto* $\mathcal{I}_{\vec{m}}$*.*

Note that the images of the slab $\mathcal{I}_{\vec{b}}(i, k_i)$ are obtained from the images of the slab $\mathcal{I}_{\vec{b}}(i, 0)$ by adding (modulo $\vec{m}$) k_i times the i-th column of M. Therefore, all values in $I_{\vec{m}}$ are images of the same number of elements in the slab $\mathcal{I}_{\vec{b}}(i, k_i)$ if and only if the same is true for the slab $\mathcal{I}_{\vec{b}}(i, 0)$. In other words, for modular mappings (as opposed to arbitrary mappings), the load-balancing property can be checked only for the slabs that contain 0 (the slabs $\mathcal{I}_{\vec{b}}(i, 0)$). Furthermore, if $\vec{b}[i]$ denotes the vector obtained from $\vec{b}$ by removing the i-th component and $M[i]$ denotes the matrix obtained from M by removing the i-th column, then the images of $\mathcal{I}_{\vec{b}}(i, 0)$ under $M_{\vec{m}}$ are the images of $\mathcal{I}_{\vec{b}[i]}$ under the modular mapping $M[i]_{\vec{m}}$. We therefore have the following property.

Lemma 4 *Given a hyper-rectangle* $\mathcal{I}_{\vec{b}}$*, a modular mapping* $M_{\vec{m}}$ *has the load-balancing property for* $\mathcal{I}_{\vec{b}}$ *if and only if each mapping* $M[i]_{\vec{m}}$ *is a equally-many-to-one modular mapping from* $\mathcal{I}_{\vec{b}[i]}$ *to* $\mathcal{I}_{\vec{m}}$*.*

We now address the following issues. First, we need to check that a given modular mapping has the load-balancing property. This is the topic of Section 5.2. Second, given an array of tiles $\mathcal{I}_{\vec{b}}$ defined by a candidate partitioning $\vec{b}$, we need to find a modular mapping that has the load-balancing property. This is the topic of Section 5.3. We will have to define an adequate matrix M and also to choose an adequate shape $\vec{m}$ for the virtual grid of processors.

5.2 Validity of Modular Mappings

Let us first recall some of the theory of one-to-one modular mappings developed by Lee and Fortes [21] and by Darte, Dion, and Robert [11]. We will then extend some of these results to equally-many-to-one modular mappings. We will make use of Hermite and Smith forms. We recall here the definitions for square matrices.

Definition 7 (Smith normal form)

Given a square matrix A *of with integral components, there exist integral matrices* Q_1*,* Q_2*, and* S *such that:*

- Q_1 *and* Q_2 *are unimodular (i.e., with determinant* 1 *or* -1*),*
- S *is nonnegative,* $S = diag(s_1, \ldots, s_r, 0, \ldots, 0)$ *where* r *is the rank of* A*, and* s_i *divides* s_{i+1} *for* $1 \leq i < r$*,*
- $A = Q_1 S Q_2$*.*

Definition 8 (Left Hermite normal form)

Given a square matrix A *with integral components, there exist integral matrices* Q *and* H *such that:*

- Q *is unimodular (i.e., with determinant* 1 *or* -1*),*

- *H is lower triangular and nonnegative,*
- *each nondiagonal element is smaller than the diagonal element of the same row,*
- *$A = HQ$.*

We can also apply the Hermite decomposition by first permuting the rows of A, i.e., considering the rows of A in a different order when "triangularizing" the matrix A into H. We will consider these $d!$ Hermite forms in the rest of our study (where d is the size of A). We will also use the following notations: $a\mathbb{Z}$ denotes the set of integers that are multiple of a, and $\vec{m}\mathbb{Z}$ denotes the product $m_1\mathbb{Z} \times \ldots \times m_{d'}\mathbb{Z}$ for a vector $\vec{m}$ of size d'. A modular mapping $M_{\vec{m}}$ is a linear mapping from the group $\mathbb{Z}^d$ into the quotient group $\mathbb{Z}^{d'}/\vec{m}\mathbb{Z}$.

5.2.1 The Lattice $G = \mathbf{Ker}(M_{\vec{m}})$

We denote by G the set $\mathrm{Ker}(M_{\vec{m}}) = \{\vec{i} \mid M\vec{i} = \vec{0} \bmod \vec{m}\}$, i.e., the set of pre-images of the zero of $\mathbb{Z}^{d'}/\vec{m}\mathbb{Z}$. The set G is essential for our study.

Lemma 5 *G is a subgroup of $\mathbb{Z}^d$. G is also a sub-lattice of $\mathbb{Z}^d$, whose determinant divides the cardinality of $\mathbb{Z}^{d'}/\vec{m}\mathbb{Z}$, i.e., $\prod_{i=1}^{d'} m_i$.*

Proof: G is the pre-image of the subgroup of $\mathbb{Z}^{d'}/\vec{m}\mathbb{Z}$ whose single element is the zero of $\mathbb{Z}^{d'}/\vec{m}\mathbb{Z}$. Therefore G is a subgroup of $\mathbb{Z}^d$. Now, consider the equivalence relation $\vec{i} \equiv \vec{i'} \Leftrightarrow \vec{i} - \vec{i'} \in G$. Let π be the canonical surjective map from $\mathbb{Z}^d$ onto the quotient group $\mathbb{Z}^d/G$ such that $\pi(\vec{i})$ is the class of $\vec{i}$ with respect to the equivalence relation $\equiv$. There is a unique map $\overline{M}_{\vec{m}}$ from $\mathbb{Z}^d/G$ into $\mathbb{Z}^{d'}/\vec{m}\mathbb{Z}$ such that $\overline{M}_{\vec{m}} \circ \pi = M_{\vec{m}}$, and $\overline{M}_{\vec{m}}$ is one-to-one (this is the classical factorization of a linear map between groups). The image of $\mathbb{Z}^d/G$ under $\overline{M}_{\vec{m}}$ is a subgroup of $\mathbb{Z}^{d'}/\vec{m}\mathbb{Z}$, thus its cardinality divides the cardinality of $\mathbb{Z}^{d'}/\vec{m}\mathbb{Z}$ (Lagrange's theorem). Since $\overline{M}_{\vec{m}}$ is one-to-one, this holds also for $\mathbb{Z}^d/G$. Finally, since $\mathbb{Z}^d$ is a lattice and G is a subgroup of $\mathbb{Z}^d$, G is also a lattice, and its determinant is the cardinality of $\mathbb{Z}^d/G$. ■

The map $\overline{M}_{\vec{m}}$ is a one-to-one linear map from the equivalence classes of $\equiv$ onto the image of $\mathbb{Z}^d$ under $M_{\vec{m}}$. Now, given a set $S \in Z^d$ of representatives of the equivalence relation $\equiv$ (i.e., a set for which the restriction of the surjective map π is one-to-one), the restriction of $M_{\vec{m}}$ to S is also a one-to-one mapping, but we lose the linear properties of $M_{\vec{m}}$ or $\overline{M}_{\vec{m}}$, since S has no particular algebraic structure. This will typically be our case when considering the rectangular index set $\mathcal{I}_{\vec{b}}$. We have the following result.

Lemma 6 *If a basis of G is given by a triangular (up to a permutation of the rows) matrix H, then the rectangular index set $\mathcal{I}_{\vec{b}}$ where $\vec{b}$ is the vector of diagonal components of H is a set of representatives of the equivalence relation $\equiv$.*

Proof: First, if $\vec{b}$ is defined as the diagonal components of H, then $\mathcal{I}_{\vec{b}}$ and $\mathbb{Z}^d/G$ have the same number of elements. It remains to check that all elements of $\mathcal{I}_{\vec{b}}$ have different images under π, i.e., belong to different equivalence classes. Let $\vec{i}$ and $\vec{i'}$ in $\mathcal{I}_{\vec{b}}$ such that $\vec{i} \equiv \vec{i'}$, i.e., $\vec{i} - \vec{i'} \in G$. Let $\vec{i} - \vec{i'} = \vec{j}$. By definition, $|\vec{j}| < \vec{b}$ (component-wise), and $\vec{j} = H\vec{k}$ for some integral vector $\vec{k}$. Such a triangular system has only one integral solution, $\vec{j} = \vec{0}$. ■

The previous lemma provides the means of computing a rectangular index for which the map $M_{\vec{m}}$ is one-to-one. Now, we need to compute a basis for the lattice G. This can be done as follows.

Consider $\vec{i} \in G$: $M\vec{i} = \vec{0} \bmod \vec{m} \Leftrightarrow \exists \vec{k} \in \mathbb{Z}^{d'}$ such that $M\vec{i} = \theta_{\vec{m}}\vec{k}$ where $\theta_{\vec{m}}$ is the diagonal matrix diag($\vec{m}$). Let $\overline{\theta}_{\vec{m}}$ be the comatrix of $\theta_{\vec{m}}$, i.e., the matrix such that $\overline{\theta}_{\vec{m}}\theta_{\vec{m}} = \det(\theta_{\vec{m}})I_{\vec{d'}}$. We get:

$$M\vec{i} = \theta_{\vec{m}}\vec{k} \Leftrightarrow (\overline{\theta}_{\vec{m}}M)\vec{i} = \det(\theta_{\vec{m}})\vec{k} \Leftrightarrow Q_1SQ_2\vec{i} = \det(\theta_{\vec{m}})\vec{k} \Leftrightarrow SQ_2\vec{i} = \det(\theta_{\vec{m}})Q_1^{-1}\vec{k}$$

where Q_1SQ_2 is the Smith form of $\theta_{\vec{m}}M$. Now, there exists $\vec{k} \in \mathbb{Z}^{d'}$ such that $M\vec{i} = \theta_{\vec{m}}\vec{k}$ if and only if there exists $\vec{k'} \in \mathbb{Z}^{d'}$ such that $S\vec{j} = \det(\theta_{\vec{m}})\vec{k'}$ and $\vec{i} = Q_2^{-1}\vec{j}$. It remains to solve the diagonal system $S\vec{j} = \det(\theta_{\vec{m}})\vec{k'}$. With $p = \det(\theta_{\vec{m}}) = \prod_{i=1}^{d'} m_i$, there exists an integer k'_i such that $s_i j_i = pk'_i$ if and only if p divides $s_i j_i$ iff $\frac{p}{\gcd(s_i,p)}$ divides $\frac{s_i}{\gcd(s_i,p)}j_i$ iff j_i is a multiple of $\frac{p}{\gcd(s_i,p)}$ (this is true even if $s_i = 0$ with the convention that $\gcd(0,p) = p$). Thus, with $S' = \text{diag}(\frac{p}{\gcd(s_1,p)}, \ldots, \frac{p}{\gcd(s_d,p)})$, we just proved that $\vec{i} \in G$ if and only if there exists $\vec{k''} \in \mathbb{Z}^d$ such that $\vec{i} = Q_2^{-1}S'\vec{k''}$. In other words, $Q_2^{-1}S'$ is a basis for G.

Note that if M is unimodular, the above construction can be simplified. Indeed, in this case, $M\vec{i} = \theta_{\vec{m}}\vec{k} \Leftrightarrow \vec{i} = M^{-1}\theta_{\vec{m}}\vec{k}$, thus $M^{-1}\theta_{\vec{m}}$ is a basis for G. We will use this later, focusing only on unimodular matrices (even triangular). Note also that given a lattice described by a nonsingular square matrix B, we can always build back a matrix M and a vector $\vec{m}$ such that the corresponding lattice G for $M_{\vec{m}}$ has basis B. This can be done as follows. Write the Smith normal form of B, $B = Q_1SQ_2$. Let $M = Q_1^{-1}$ and let $\vec{m}$ be the vector whose components are the diagonal components of S. As seen before, a basis for G is then Q_1S, which is also a basis of the lattice described by B since B and Q_1S differ by multiplication on the right by a unimodular matrix.

5.2.2 One-To-One Modular Mappings

We are now ready to discuss necessary and sufficient conditions for a modular mapping $M_{\vec{m}}$ to be one-to-one from a rectangular index set $\mathcal{I}_{\vec{b}}$ onto the rectangular set $\mathcal{I}_{\vec{m}}$ (here we identify $Z^{d'}/\vec{m}\mathbb{Z}$ with the set $\mathcal{I}_{\vec{m}}$ forgetting about the group structure).

First, of course, the cardinality of $\mathcal{I}_{\vec{b}}$ and $\mathcal{I}_{\vec{m}}$ have to be the same, i.e., $\prod_{i=1}^{d} b_i = \prod_{i=1}^{d'} m_i$. Also, the determinant of the lattice G has to be equal to the cardinality of $\mathcal{I}_{\vec{m}}$. Otherwise, according to Lemma 5, the cardinality of the range of $M_{\vec{m}}$ divides (strictly) the cardinality of $\mathcal{I}_{\vec{m}}$, therefore some elements of $\mathcal{I}_{\vec{m}}$ have no pre-image, and the mapping cannot be one-to-one onto $\mathcal{I}_{\vec{m}}$.

Lemma 6 gives a sufficient condition for a mapping $M_{\vec{m}}$ to be one-to-one from a rectangular index set $\mathcal{I}_{\vec{b}}$ onto the rectangular index set $\mathcal{I}_{\vec{m}}$. If the lattice G has a basis expressed by a triangular (up to a permutation of the rows) matrix whose diagonal components are the components of $\vec{b}$, and if the determinant of G is equal to the cardinality of $\mathcal{I}_{\vec{m}}$ (equivalently if $\mathcal{I}_{\vec{b}}$ and $\mathcal{I}_{\vec{m}}$ have the same cardinality), then the mapping is a one-to-one mapping from $\mathcal{I}_{\vec{b}}$ onto $\mathcal{I}_{\vec{m}}$. It turns out that the converse is also true (see details in [11]) thanks to a famous (in covering/packing theory) theorem due to Hajós [18].

Theorem 2 *A modular mapping $M_{\vec{m}}$ is one-to-one from $\mathcal{I}_{\vec{b}}$ onto $\mathcal{I}_{\vec{m}}$ if and only if the lattice G has a basis given by a triangular (up to a permutation of the rows) matrix whose diagonal components are the components of $\mathcal{I}_{\vec{b}}$, and $\mathcal{I}_{\vec{b}}$ and $\mathcal{I}_{\vec{m}}$ have the same cardinality.*

To find the diagonal components of this triangular basis, we just have to compute the $d!$ left Hermite forms of a given basis. For example, if the basis of G is given by the matrix $B = \begin{pmatrix} 2 & 4 \\ 3 & 12 \end{pmatrix}$, then:

$$\begin{pmatrix} 2 & 4 \\ 3 & 12 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 3 & 6 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 2 & 4 \\ 3 & 12 \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 3 & 0 \end{pmatrix}\begin{pmatrix} 1 & 4 \\ 0 & -1 \end{pmatrix}$$

Therefore, among the 6 possibilities – $\vec{b} \in \{(1,12),(12,1),(2,6),(6,2),(3,4),(4,3)\}$ – the mapping is one-to-one from $\mathcal{I}_{\vec{b}}$ onto $\mathcal{I}_{\vec{m}}$ only for $\vec{b} = (2,6)$ and $\vec{b} = (4,3)$. We need to build a mapping with such property. To begin, we compute the Smith normal form of B:

$$\begin{pmatrix} 2 & 4 \\ 3 & 12 \end{pmatrix} = \begin{pmatrix} 2 & -1 \\ 3 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 12 \end{pmatrix} \begin{pmatrix} 1 & 8 \\ 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 2 & -1 \\ 3 & -1 \end{pmatrix}^{-1} = \begin{pmatrix} -1 & 1 \\ -3 & 2 \end{pmatrix}$$

Therefore the mapping $(i,j) \longrightarrow (-i+j \bmod 1, -3i+2j \bmod 12)$ is such a mapping. We can even remove the first component of the mapping (since $x \bmod 1$ is always 0), and simply consider the mapping $(i,j) \longrightarrow -3i+2j \bmod 12$. The table below depicts the values $-3i+2j \bmod 12$ in the plane (i,j) (i increases from left to right, j from bottom to top). The reader can check that only the "boxes" $(2,6)$ and $(4,3)$ are suitable.

...	...	...	...	...	...	...	...	...
...	0	9	6	3	0	9	6	...
...	10	7	4	1	10	7	4	...
...	8	5	2	11	8	5	2	...
...	6	3	0	9	6	3	0	...
...	4	1	10	7	4	1	10	...
...	2	11	8	5	2	11	8	...
...	0	9	6	3	0	9	6	...
...	...	...	...	...	...	...	...	...

We point out that the techniques developed here (with the help of Hajós' theorem) are also useful in systolic array-like design (see [10] and [9]) for generating "juggling schedules" corresponding to "locally sequential, globally parallel" partitionings.

5.2.3 Equally-Many-To-One Modular Mappings

We now generalize the previous results (when possible) to the case of equally-many-to-one modular mappings. The situation turns out to be much more difficult. We will say that a rectangular index set $\mathcal{I}_{\vec{b}}$ is a multiple of a rectangular index set $\mathcal{I}_{\vec{b}'}$ if $\vec{b}$ and $\vec{b}'$ have same size and each component of $\vec{b}$ is a multiple of the corresponding component of $\vec{b}'$. In other words, $\mathcal{I}_{\vec{b}}$ can be paved (partitioned) by disjoint copies of $\mathcal{I}_{\vec{b}'}$.

The results of the previous section provide immediately a necessary and sufficient condition for a modular mapping $M_{\vec{m}}$ to be equally-many-to-one from $\mathcal{I}_{\vec{b}}$ onto $\mathcal{I}_{\vec{m}}$.

Lemma 7 *If $M_{\vec{m}}$ is a equally-many-to-one modular mapping from $\mathcal{I}_{\vec{b}}$ onto $\mathcal{I}_{\vec{m}}$, then the cardinality of $\mathcal{I}_{\vec{b}}$ is a multiple of the cardinality of $\mathcal{I}_{\vec{m}}$, and the determinant of the lattice G is equal to the cardinality of $\mathcal{I}_{\vec{m}}$.*

Proof: The first part is obvious. If $M_{\vec{m}}$ is equally-many-to-one from $\mathcal{I}_{\vec{b}}$ onto $\mathcal{I}_{\vec{m}}$, by Definition 5, there exists an integer k such that each element of $\mathcal{I}_{\vec{m}}$ has exactly k pre-images. Therefore, the cardinality of $\mathcal{I}_{\vec{b}}$ is exactly k times the cardinality of $\mathcal{I}_{\vec{m}}$.

The second part comes from Lemma 5: the image of $\mathbb{Z}^d$ is a set whose cardinality is the cardinality of the determinant of G. If the determinant of G is not equal to the cardinality of $\mathcal{I}_{\vec{m}}$, then some element in $\mathcal{I}_{\vec{m}}$ has no pre-image at all. Since $\vec{0}$ has at least one pre-image ($\vec{0}$ always belongs to a rectangular index set since we assumed $\vec{b}$ to be positive), not all elements of $\mathcal{I}_{\vec{m}}$ have the same number of pre-images. ■

Lemma 8 *If $M_{\vec{m}}$ is a one-to-one modular mapping from $\mathcal{I}_{\vec{b}'}$ onto $\mathcal{I}_{\vec{m}}$, then $M_{\vec{m}}$ is a equally-many-to-one modular mapping from any multiple $\mathcal{I}_{\vec{b}}$ of $\mathcal{I}_{\vec{b}'}$ onto $\mathcal{I}_{\vec{m}}$.*

Proof: This is clear. Each element in $\mathcal{I}_{\vec{b}'}$ gives rise to a different value in $\mathcal{I}_{\vec{m}}$. Furthermore, this property is true for any translated copy of $\mathcal{I}_{\vec{b}'}$ since a translation of the index set $\mathcal{I}_{\vec{b}'}$ corresponds to a translation (modulo $\vec{m}$) of their images. We already made this remark in Lemma 4. Therefore, if $M_{\vec{m}}$ is one-to-one from $\mathcal{I}_{\vec{b}'}$ onto $\mathcal{I}_{\vec{m}}$, $M_{\vec{m}}$ is also one-to-one from any translated copy of $\mathcal{I}_{\vec{b}'}$ onto $\mathcal{I}_{\vec{m}}$. Since $\mathcal{I}_{\vec{b}}$ is the union of disjoint copies of $\mathcal{I}_{\vec{b}}$, $M_{\vec{m}}$ is a equally-many-to-one mapping from $\mathcal{I}_{\vec{b}}$ onto $M_{\vec{m}}$, since each element of $M_{\vec{m}}$ has as many pre-images as there are copies of $\mathcal{I}_{\vec{b}'}$ in the partitioning of $\mathcal{I}_{\vec{b}}$. ■

We need to generalize Theorem 2 to the case of equally-many-to-one modular mappings. The theorem of Hajós used to prove Theorem 2 is a theorem of group theory, for a group that is the direct sum of particular subsets ("beginning" of cyclic groups). The direct sum (that Hajós denoted by $\oplus$) means that each element of the group can be decomposed, in a unique way, as the sum of elements, each one in a different subset. In our case, the number of subsets is the dimension of the index set we work with (d). It turns out that Hajós proved an extension of his result to more general "direct sums" (denoted by $\overset{k}{\oplus}$), where each element can be decomposed in exactly k ways into the sum of elements, each one in a different subset. He proved a less-known result of the same kind up to dimension 3, but gives a counter-example for 4 such subsets. For more details, see the original paper by Hajós (in German) [18] or Fuchs' book [15].

We can easily use Hajós' extended result to adapt the proof of Theorem 2, up to dimension 3: a modular mapping is equally-many-to-one from $\mathcal{I}_{\vec{b}}$ onto $\mathcal{I}_{\vec{m}}$ if and only if the lattice G has a determinant equal to the cardinality of $\mathcal{I}_{\vec{m}}$, and one of the $d!$ left Hermite forms of a basis of G is such that each diagonal component divides the corresponding component of $\vec{b}$. Thanks to Lemma 4, this will give a necessary and sufficient condition for mappings with the load-balancing property up to dimension 4 (one more than for equally-many-to-one mappings). However, because of Hajós' counter-example with 4 subsets, these results are more likely to be wrong for higher dimensions, even if we did not try to build a counter-example neither for equally-many-to-one mappings, nor for mappings with the load-balancing property. It is possible that the very constrained conditions needed for the load-balancing property (the mapping has to be equally-many-to-one for each "face" of the index set) forces us to look for a counter-example in even higher dimensions, but nevertheless, we think that such a counter-example is more likely to exist.

The extension of Theorem 2 to equally-many-to-one modular mappings has a very interesting consequence. When this extension is true (with the previous discussion, at least up to dimension 3), it implies that each index set $\mathcal{I}_{\vec{b}}$ for which the equally-many-to-one property is true is a multiple of a smaller index set for which the one-to-one property holds. But in higher dimensions, it is very possible that $\mathcal{I}_{\vec{b}}$ can be partitioned using several smaller rectangular index sets, for each of which the one-to-one property is true, but that $\mathcal{I}_{\vec{b}}$ cannot be partitioned using only one of them. This seems to be related to results (much simpler compared to Hajós' theorem) due to N. G. de Bruijn who characterized the rectangular index sets, partitioned into given smaller rectangular index sets, that can be partitioned using only one of them. In higher dimensions, this situation is more likely to happen. Therefore, in higher dimensions, we do not know if a simple necessary and sufficient condition for a modular mapping to be equally-many-to-one can be given. Nevertheless, for our practical concern, we are mainly interested in building **one** such mapping, and for this, using the sufficient conditions from the if-part of Theorem 2 and from Lemma 8 will be enough.

5.2.4 Modular Mappings With the Load-Balancing Property

Lemma 4 makes the link between modular mappings with the load-balancing property and equally-many-to-one modular mappings. Using the various necessary conditions and sufficient conditions of the two previous sections, for one-to-one mappings and equally-many-to-one mappings respectively, we can now give conditions for the load-balancing property.

Theorem 3 *If $M_{\vec{m}}$ is a modular mapping with the load-balancing property for $\mathcal{I}_{\vec{b}}$, then the following holds:*

- *for each dimension j, $\prod_{i \neq j} b_i$ is a multiple of the cardinality of $\mathcal{I}_{\vec{m}}$,*
- *the determinant of the lattice G is equal to the cardinality of $\mathcal{I}_{\vec{m}}$,*
- *for any basis B of G, each row of B has coprime components.*

Proof: The first property is clear. This is the same as our validity condition for $\vec{\gamma}$ in Section 4.3 and a consequence of the first condition of Lemma 7.

The second property is a consequence of Lemma 5 again. The determinant of the lattice G for $M_{\vec{m}}$ divides the cardinality of $\mathcal{I}_{\vec{m}}$. If the determinant is not equal to $\mathcal{I}_{\vec{m}}$, it is strictly smaller, and some values in $\mathcal{I}_{\vec{m}}$ have no pre-image at all in the whole set $\mathbb{Z}^n$, therefore no pre-image in $\mathcal{I}_{\vec{m}}$. In this case, the mapping cannot have the load-balancing property (in terms of Section 4.3, some processors have no tiles at all to compute).

The third property comes from Lemma 4. The link between the lattice G for $M_{\vec{m}}$ and the lattice $G[i]$ for $M[i]_{\vec{m}}$ ($M[i]$ is the matrix obtained by removing the i-th column of M, $M[i]_{\vec{m}}$ is the corresponding mapping modulo $\vec{m}$). The mapping $M_{\vec{m}}$ has the load-balancing property if and only if, for each dimension i, $M[i]_{\vec{m}}$ is equally-many-to-one when restricted to the i-th "face" of $\mathcal{I}_{\vec{b}}$, i.e., $\mathcal{I}_{\vec{b}}(i, 0)$. Following Lemma 7, the determinant of the lattice $G[i]$ has to be equal to the cardinality of $\mathcal{I}_{\vec{m}}$. The lattice $G[i]$ is the set of vectors in $\mathbb{Z}^{d-1}$ whose image under $M[i]_{\vec{m}}$ is $\vec{0}$. Embedding the set $\mathbb{Z}^{d-1}$ into $\mathbb{Z}^d$ (according to i), $\mathbb{Z}^{d-1}$ is the set of vectors in $\mathbb{Z}^d$ such that the i-th component is 0, and $G[i]$ can be viewed as the projection of the intersection of G with the set $\{\vec{j} \in \mathbb{Z}^d \mid j_i = 0\}$. This is the reverse operation we did to obtain Lemma 4. To say it differently, with $\vec{j}[i]$ the vector obtained from $\vec{j}$ by removing the i-th component, we have:

$$\begin{aligned}(\vec{j} \in G \text{ and } j_i = 0) \quad &\Leftrightarrow \quad (M\vec{j} = \vec{0} \bmod \vec{m} \text{ and } j_i = 0) \Leftrightarrow (M[i]\vec{j}[i] = \vec{0} \bmod \vec{m} \text{ and } j_i = 0) \\ &\Leftrightarrow \quad (\vec{j}[i] \in G[i] \text{ and } j_i = 0)\end{aligned}$$

In other words, instead of computing $G[i]$ following the steps given in Section 5.2.1 starting from the matrix $M[i]$, we can build $G[i]$ directly from G (which is computed starting from the matrix M). Starting from a basis B of G, we can multiply B (on the right) by any unimodular matrix Q, and BQ is still a basis of G. Furthermore, as we do for computing Hermite normal forms, we can choose Q such that the i-th row of B has only one nonzero component (which is the gcd – greater common divisor – of the elements of this row). We then remove the i-th row of B and the column that contains the nonzero component of the i-th row, and we obtain a basis $B[i]$ for $G[i]$. By construction, the determinant of B is equal (up to sign) to the determinant of $B[i]$ times the nonzero element of the i-th row. Therefore, since the determinant of G divides the cardinality of $\mathcal{I}_{\vec{m}}$, this is true also for the determinant of $B[i]$, and there is equality if and only if the nonzero element was equal to 1 (or −1), i.e., if and only if the elements of the i-th row are coprime. ■

Theorem 3 will give us some hints on how to build a desired mapping with the load-balancing property. Let us give a sufficient condition now, which is a direct consequence of the if-part of Theorem 2 and of Lemma 8, and of the link between $G[i]$ and G revealed in the previous lemma.

Theorem 4 *A modular mapping $M_{\vec{m}}$ has the load-balancing property for $\mathcal{I}_{\vec{b}}$ if the following conditions are satisfied:*

- *the determinant of the lattice G for $M_{\vec{m}}$ is equal to the cardinality of $\mathcal{I}_{\vec{m}}$,*
- *if B is a basis of the lattice G, for each dimension i, there is permutation of the rows of B such that the i-th row of B is now the first and such that the (unique for this permutation) left Hermite form of B is HQ where the first diagonal component of B is 1 and each diagonal component of H divides the corresponding (taking the permutation into account) component of $\vec{b}$.*

Proof: Consider a dimension i. To make the explanations simpler, let us get rid of the permutation mentioned in the conditions of the lemma by permuting the axes accordingly. Now, the matrix H is still a basis for G since this is B multiplied on the right by a unimodular matrix. H' the lower-right submatrix of H, obtained by removing the first row and first column, is a basis of $G[i]$, as we showed in the proof of Theorem 3. This matrix H' satisfies the condition of Theorem 2: its determinant is equal to the determinant of B (since the first component of H is 1), which is equal to the cardinality of $\mathcal{I}_{\vec{m}}$, therefore $M[i]_{\vec{m}}$ is a one-to-one mapping from $\mathcal{I}_{\vec{h}}$ onto $\mathcal{I}_{\vec{m}}$, where $\vec{h}$ is the diagonal of H'. Now, thanks to Lemma 8, we know that $M[i]_{\vec{m}}$ is a equally-many-to-one modular mapping from the slab $\mathcal{I}_{\vec{b}}(i,0)$ onto $\mathcal{I}_{\vec{m}}$ since $\mathcal{I}_{\vec{b}[i]}$ is a multiple of $\mathcal{I}_{\vec{h}}$. Finally, since this is true for any dimension i, Lemma 4 shows that $M_{\vec{m}}$ is a modular mapping with the load-balancing property for the index set $\mathcal{I}_{\vec{b}}$. ■

Note that in many practical cases, the previous conditions are also necessary. For example, when for a dimension i, the cardinality of $\mathcal{I}_{\vec{b}}(i,0)$ is equal to $\mathcal{I}_{\vec{m}}$ (and not simply a multiple), the conditions are necessary (Theorem 2) since equally-many-to-one is in this case one-to-one. Also, when $d \leq 4$, then each modular mapping $M[i]_{\vec{m}}$ is a mapping in dimension less or equal to 3 and the conditions of Lemma 8 are also necessary. Furthermore, as pointed out previously, the fact that the extension of Hajós' theorem does not hold for a dimension d does not mean that the conditions of Theorem 4 are not necessary for dimension $(d+1)$. The load-balancing property makes the problem much more constrained. We would need a counter-example to give a complete answer to such a question.

5.3 Generating A Valid Modular Mapping

We are now ready, thanks to the previous observations, to build a modular mapping $M_{\vec{m}}$ with the load-balancing property for an index set $\mathcal{I}_{\vec{b}}$ (which is given, $\vec{b}$ is the vector whose components are the γ_i's of Section 4.3). We can choose the matrix M and the modulo vector $\vec{m}$, but with the constraint that the cardinality of $\mathcal{I}_{\vec{m}}$ (the product of the components of $\vec{m}$) is given, equal to the number of processors p. The only property of $\vec{b}$ we exploit is that $\vec{b}$ is a candidate partitioning (with the meaning of Section 4.3), which means that the product of any $(d-1)$ components of $\vec{b}$ is a multiple of p. We will choose the matrix M to be unimodular so that the lattice G is easy to compute (as we noticed in Section 5.2.1). A basis of G is simply given by $M^{-1}\theta_{\vec{m}}$, where $\theta_{\vec{m}} = \operatorname{diag}(m_1, \ldots, m_d)$. Note also that, when M is unimodular, the first condition of Theorem 4 is automatically fulfilled since the determinant of G is the determinant of $M^{-1}\theta_{\vec{m}}$, thus the determinant of $\theta_{\vec{m}}$, i.e., the cardinality of $\mathcal{I}_{\vec{m}}$.

We will choose the matrix M with the following form:

$$M = \begin{pmatrix} N & 0 \\ * & 1 \end{pmatrix}$$

where N will be computed by induction on the dimension. Therefore, finally, M will be even triangular, with 1's on the diagonal. We have the following preliminary result.

Lemma 9 *Suppose that m_d divides b_d, and that the modular mapping $N_{\vec{m}'}$ – in dimension $(d-1)$ – defined by N and $\vec{m}'$ has the load-balancing property for $I_{\vec{b}'}$, where $\vec{b}'$ and $\vec{m}'$ are the vectors defined by the $(d-1)$ first components of $\vec{b}$ and $\vec{m}$. Then, the modular mapping $M_{\vec{m}}$ defined by M and $\vec{m}$ has the load-balancing property for $\mathcal{I}_{\vec{b}}$ if it is equally-many-to-one from the last slab $\mathcal{I}_{\vec{b}}(d,0)$ onto $\mathcal{I}_{\vec{m}}$.*

Proof: In order to check that the mapping defined by M and $\vec{m}$ has the load-balancing property for the rectangular index set $\mathcal{I}_{\vec{b}}$, we have to make sure that it is equally-many-to-one for all slabs $\mathcal{I}_{\vec{b}}(i,0)$, $1 \le i \le d$ (Lemma 4). To prove this lemma, we only have to prove that this is true for the slabs $\mathcal{I}_{\vec{b}}(i,0)$, $i < d$, if N has the properties stated, since this is supposed to be true for $\mathcal{I}_{\vec{b}}(d,0)$.

Without loss of generality, let us consider the first dimension, i.e., the first slab $\mathcal{I}_{\vec{b}}(1,0)$. Given $\vec{j} \in \mathbb{Z}^d/\vec{m}\mathbb{Z}$, let us count the number of vectors $\vec{i} \in \mathcal{I}_{\vec{b}}$, such that $M\vec{i} = \vec{j} \bmod \vec{m}$ and $i_1 = 0$.

$$M\vec{i} = \vec{j} \bmod \vec{m} \Leftrightarrow N\vec{i}' = \vec{j}' \bmod \vec{m}' \text{ and } \vec{\lambda}.\vec{i}' + i_d = j_d \bmod m_d$$

where $\vec{i}'$ and $\vec{j}'$ are defined the same way as $\vec{b}'$ and $\vec{m}'$, and $\vec{\lambda}$ is the row vector formed by the first $(d-1)$ component of the last row of M. Now, because of the load-balancing property of $N_{\vec{m}'}$, there are exactly n vectors $\vec{i}' \in \mathcal{I}_{\vec{b}'}$ such that $i_1 = 0$ and $N\vec{i}' = \vec{j}' \bmod \vec{m}'$, where n is a positive integer that does not depend on $\vec{j}'$. It remains to count the number of values i_d, between 0 and $b_d - 1$, such that $i_d = j_d - \vec{\lambda}.\vec{i}' \bmod m_d$. Since m_d divides b_d, there are exactly b_d/m_d such values, whatever the value $x = (j_d - \vec{\lambda}.\vec{i}' \bmod m_d)$. These are the values $x + km_d$, with $0 \le k < b_d/m_d$. Therefore, $\vec{j}$ has nb_d/m_d pre-images in $\mathcal{I}_{\vec{b}}$ and this number does not depend on $\vec{j}$. ■

We define the vector $\vec{m}$ according to the following formula:

$$\forall i,\ 1 \le i \le d,\ m_i = \frac{\gcd\left(p, \prod_{j=i}^{d} b_j\right)}{\gcd\left(p, \prod_{j=i+1}^{d} b_j\right)} \tag{3}$$

(By convention, a product of no terms is equal to 1). The vector $\vec{m}$ defined this way has several properties that will make a recursive construction of M possible. In the rest of the proofs, we will make an extensive use of the relation $\gcd(a,bc) = \gcd(a,b)\gcd\left(\frac{a}{\gcd(a,b)}, c\right)$.

Lemma 10 *The vector $\vec{m}$ defined by Equation 3 has the following properties: (1) the components of $\vec{m}$ are positive integers, (2) $m_1 = 1$, (3) the product of the components of $\vec{m}$ is equal to p, (4) each component m_i divides b_i, (5) the definition of the m_i's for $i < d$ is the same if we only consider the $(d-1)$ first components of $\vec{b}$ and the processor number $\frac{p}{\gcd(p,b_d)}$. Furthermore, if $\vec{b}$ is a candidate partitioning for p, then $(b_1, \dots, b_{d-1})$ is a candidate partitioning for $\frac{p}{\gcd(p,b_d)}$, (6) If $\vec{b}$ is a candidate partitioning for p, then for all k, $2 \le k \le d$, $\prod_{i=1}^{k-1} b_i$ is a multiple of $\prod_{i=2}^{k} m_i$.*

Proof: The first property is obvious. The second property ($m_1 = 1$) comes from the fact that $\prod_{i=2}^{d} b_i$ is a multiple of p, therefore $m_1 = p/p = 1$. The third property is also elementary. When computing the product of the m_i, the denominator of m_i cancels with the numerator of m_{i+1}, only the first numerator remains, which is p.

Let us consider the fourth property. By definition we have:

$$m_i \gcd\left(p, \prod_{j=i+1}^{d} b_j\right) = \gcd\left(p, \prod_{j=i}^{d} b_j\right) = \gcd(p, b_i) \gcd\left(\frac{p}{\gcd(p, b_i)}, \prod_{j=i+1}^{d} b_j\right)$$

Since $\gcd\left(p, \prod_{j=i+1}^{d} b_j\right)$ is a multiple of $\gcd\left(\frac{p}{\gcd(p,b_i)}, \prod_{j=i+1}^{d} b_j\right)$, $\gcd(p, b_i)$ is a multiple of m_i. Therefore, m_i divides both p and b_i.

For the fifth property, we have:

$$m_i = \frac{\gcd\left(p, \prod_{j=i}^{d} b_j\right)}{\gcd\left(p, \prod_{j=i+1}^{d} b_j\right)} = \frac{\gcd(p, b_d) \gcd\left(\frac{p}{\gcd(p,b_d)}, \prod_{j=i}^{d-1} b_j\right)}{\gcd(p, b_d) \gcd\left(\frac{p}{\gcd(p,b_d)}, \prod_{j=i+1}^{d-1} b_j\right)} = \frac{\gcd\left(\frac{p}{\gcd(p,b_d)}, \prod_{j=i}^{d-1} b_j\right)}{\gcd\left(\frac{p}{\gcd(p,b_d)}, \prod_{j=i+1}^{d-1} b_j\right)}$$

Furthermore, if $\vec{b}$ is a candidate partitioning for p, then for all $i < d$, p divides $\prod_{j=1, j \neq i}^{d} b_j$, therefore $\frac{p}{\gcd(p,b_d)}$ divides $\prod_{j=1, j \neq i}^{d-1} b_j$. Thus, the first $(d-1)$ components of $\vec{b}$ form a candidate partitioning for $\frac{p}{\gcd(p,b_d)}$.

For the last property, we have:

$$\prod_{i=2}^{k} m_i = \prod_{i=2}^{k} \frac{\gcd\left(p, \prod_{j=i}^{d} b_j\right)}{\gcd\left(p, \prod_{j=i+1}^{d} b_j\right)} = \frac{\gcd(p, \prod_{j=2}^{d} b_j)}{\gcd(p, \prod_{j=k+1}^{d} b_j)} = \gcd\left(\frac{p}{\gcd(p, \prod_{j=k+1}^{d} b_j)}, \prod_{j=2}^{k} b_j\right)$$

Thus, $\prod_{i=2}^{k} m_i$ divides $\frac{p}{\gcd(p, \prod_{j=k+1}^{d} b_j)}$. But, if $\vec{b}$ is a candidate partitioning for p, p divides $\prod_{i \neq k} b_i$. Thus, $\frac{p}{\gcd(p, \prod_{j=k+1}^{d} b_j)}$ divides $\prod_{i=1}^{k-1} b_i$ and this is *a fortiori* true for $\prod_{i=2}^{k} m_i$. ■

Because $m_1 = 1$, we will be able to drop, at the end of the construction, the first component of the mapping, and we will end up with a mapping from $\mathbb{Z}^d$ into a subgroup of $\mathbb{Z}^{d-1}$ (or of smaller dimension if some other components of m are equal to 1). Also, the fact that $m_1 = 1$ will make our life easier when computing a basis of a lattice $G[i]$ (i.e., the pre-images of zero under the restricted mapping $M[i]_{\vec{m}}$). Indeed, the two following constructions are possible:

- Compute a basis B for G with $B = M^{-1}\theta_{\vec{m}}$, where $\theta_{\vec{m}} = \text{diag}(\vec{m})$. Manipulate the columns of B so that the i-th row has only one nonzero component. The square submatrix defined by removing the i-th row and the column that contains the nonzero component of the i-th row is a basis for $G[i]$ (see Section 5.2.1).

- Consider $M[i]_{\vec{m}}$ directly, i.e., remove the i-th column of M. $M[i]_{\vec{m}}$ is not square, but we may also remove the first row since $m_1 = 1$. Now, we get a square submatrix. If this submatrix is unimodular, then we can easily compute the corresponding $G[i]$.

We now assume that N is triangular with 1's on the diagonal. It remains to specify how we choose the last row of M so that, given N, the modular mapping $M_{\vec{m}}$ is equally-many-to-one from the slab $\mathcal{I}_{\vec{b}}(d,0)$ onto $\mathcal{I}_{\vec{m}}$. For that, since $m_1 = 1$ and the previous remark, we only need to consider the matrix $\tilde{M}$ obtained from M by removing the last column and first row.

$$\tilde{M} = \begin{pmatrix} \vec{u} & T \\ \rho & \vec{z} \end{pmatrix}$$

where $\vec{u}$ is a column vector and $\vec{z}$ is a row vector, both with $(d-1)$ components, and ρ is an integer. The matrix $(\vec{u}, T)$ is the matrix obtained by removing the first row of N, thus T is a triangular matrix with 1's on the diagonal. With I the identity matrix of size $(d-2)$, we write:

$$\begin{pmatrix} I & 0 \\ \vec{t} & 1 \end{pmatrix} \tilde{M} = \begin{pmatrix} I & 0 \\ \vec{t} & 1 \end{pmatrix} \begin{pmatrix} \vec{u} & T \\ \rho & \vec{z} \end{pmatrix} = \begin{pmatrix} \vec{u} & T \\ \vec{t}.\vec{u} + \rho & \vec{t}T + \vec{z} \end{pmatrix}$$

and we define ρ and $\vec{z}$ such that, for the last row of the previous matrix product, only the first component is nonzero, equal to 1. In other words, $\vec{z} = -\vec{t}T$ and $\rho = 1 - \vec{t}.\vec{u}$. The row vector $\vec{t}$ has $(d-2)$ components and is to be defined. We can now compute the inverse of $\tilde{M}$ starting from the previous equality (and we first multiply both sides of the equality by the adequate matrix so that the last row of matrices is now the first):

$$\tilde{M}^{-1} = \begin{pmatrix} 1 & 0 \\ \vec{u} & T \end{pmatrix}^{-1} \begin{pmatrix} \vec{t} & 1 \\ I & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ * & T^{-1} \end{pmatrix} \begin{pmatrix} \vec{t} & 1 \\ I & 0 \end{pmatrix}$$

We denote by F_t the matrix $\begin{pmatrix} \vec{t} & 1 \\ I & 0 \end{pmatrix}$ and by $\tilde{\theta}_{\vec{m}}$ the matrix $\text{diag}(m_2, \ldots, m_d)$. The matrix $\tilde{M}^{-1}\tilde{\theta}_{\vec{m}}$ is a basis of the lattice $G[d]$. Furthermore, to find the diagonal components of its left Hermite form, it is sufficient to compute the left Hermite form of $F_t\tilde{\theta}_{\vec{m}}$, since the left Hermite form of $\tilde{M}^{-1}\tilde{\theta}_{\vec{m}}$ is obtained by multiplying the left by a triangular matrix with 1's on the diagonal (T^{-1} is triangular with diagonal components equal to 1). This will not change the diagonal components of a decomposition HQ. It turns out that the left Hermite form of F_t (and similarly of F_t multiplied by a diagonal matrix) is easy to compute in a symbolic way.

Lemma 11 *Given a matrix* $A = \begin{pmatrix} & \vec{v} & \\ S & & 0 \end{pmatrix}$ *of size* n*, where* $\vec{v}$ *is a row vector of size* n *and* S *is a diagonal matrix* $diag(s)$ *of size* $(n-1)$*, the diagonal* $\vec{h}$ *of* H*, where* $A = HQ$ *is the left Hermite form of* A *is defined by the following formulas:*

- $r_n = v_n$ *and* $r_i = \gcd(v_i, r_{i+1})$ *for* $1 \leq i < n$*,*
- $h_{i+1} = \frac{r_{i+1}s_i}{r_i}$ *for* $1 \leq i < n$ *and* $h_1 = r_1$*.*

Proof: Let us compute the left Hermite form of A, first manipulating (i.e., multiplying by a unimodular matrix on the right of A) the last two columns of A so that the last component of the first row is 0. Then, we calculate the two columns $(n-2)$ and $(n-1)$ so that the $(n-1)$-th component of the first row is 0, etc. until we reach the first two columns. We let $r_n = v_n$ and we let r_i be the component that appears on the first row of A and i-th column just after we zeroed the $(i+1)$-th component of this row. Just before this change, the $(i+1)$-th component of the first row was equal to r_{i+1} and the i-th component was v_i. Therefore, by an adequate unimodular transformation (changing only the columns i and $(i+1)$), when the $(i+1)$-th component becomes zero, the gcd of v_i and r_{i+1} appears on the i-th column, thus $r_i = \gcd(v_i, r_{i+1})$. The following matrices illustrate this mechanism:

$$\begin{pmatrix} v_1 & \dots & v_i & r_{i+1} & 0 & \dots & 0 \\ s_1 & 0 & \dots & \dots & \dots & \dots & 0 \\ 0 & \ddots & & & & & \vdots \\ \vdots & & s_i & 0 & \dots & \dots & 0 \\ 0 & \dots & 0 & * & h_{i+2} & \dots & 0 \\ \vdots & & \vdots & * & * & \ddots & \vdots \\ 0 & \dots & 0 & * & * & \dots & h_n \end{pmatrix} \Longrightarrow \begin{pmatrix} v_1 & \dots & r_i & 0 & 0 & \dots & 0 \\ s_1 & 0 & \dots & \dots & \dots & \dots & 0 \\ 0 & \ddots & & & & & \vdots \\ \vdots & & * & h_{i+1} & \dots & \dots & 0 \\ 0 & \dots & 0 & * & h_{i+2} & \dots & 0 \\ \vdots & & \vdots & * & * & \ddots & \vdots \\ 0 & \dots & 0 & * & * & \dots & h_n \end{pmatrix}$$

Furthermore, the determinant of the 2×2 submatrix, as defined by the columns i and $(i+1)$ and the rows 1 and i (a triangular matrix), does not change. It was equal to $s_i r_{i+1}$, and it is now equal to $r_i h_{i+1}$. Therefore, $h_{i+1} = \frac{s_i r_{i+1}}{r_i}$. After these transformations, the matrix is triangular and h_1, the first component of the diagonal, is equal to the remaining nonzero element r_1. ■

Applying Lemma 11 to the matrix $F_t \tilde{\theta}_{\vec{m}}$ of size $(d-1)$, we get:

- $r_{d-1} = m_d$ and $r_i = \gcd(t_i m_{i+1}, r_{i+1})$ for $1 \leq i < d-1$,
- $h_{i+1} = \frac{r_{i+1} m_{i+1}}{r_i}$ for $1 \leq i < d-1$ and $h_1 = r_1$.

To prove that $M_{\vec{m}}$ is equally-many-to-one from the slab $\mathcal{I}_{\vec{b}}(d, 0)$ onto $\mathcal{I}_{\vec{m}}$, it is now sufficient to define $\vec{t}$ so that h_i divides b_i for all $i < d$ (Lemma 8 and Theorem 2) since the fact that $\mathcal{I}_{\vec{h}}$ and $\mathcal{I}_{\vec{m}}$ have same cardinality was already guaranteed because $\tilde{M}$ is unimodular. We define the vector $\vec{t}$ by the following formula:

$$t_i = \frac{r_{i+1}}{\gcd(b_{i+1}, r_{i+1})} \text{ for } 1 \leq i \leq d-2 \tag{4}$$

We then have the following relation:

$$\begin{aligned} r_i &= \gcd(t_i m_{i+1}, r_{i+1}) = \gcd\left(\frac{r_{i+1} m_{i+1}}{\gcd(b_{i+1}, r_{i+1})}, r_{i+1}\right) \\ &= \gcd(m_{i+1}, r_{i+1}) \gcd\left(\frac{r_{i+1}}{\gcd(b_{i+1}, r_{i+1})}, \frac{r_{i+1}}{\gcd(m_{i+1}, r_{i+1})}\right) \\ &= \frac{r_{i+1} \gcd(m_{i+1}, r_{i+1})}{\gcd(b_{i+1}, r_{i+1})} \text{ since } b_{i+1} \text{ is a multiple of } m_{i+1} \text{ (Lemma 10, fourth property)} \end{aligned}$$

We therefore have the following relation for the components of $\vec{h}$:

$$h_{i+1} = \frac{r_{i+1} m_{i+1}}{r_i} = \frac{m_{i+1} \gcd(b_{i+1}, r_{i+1})}{\gcd(m_{i+1}, r_{i+1})}$$

We can then check that h_{i+1} divides b_{i+1}. Indeed:

$$\begin{aligned} h_{i+1} \text{ divides } b_{i+1} &\Leftrightarrow m_{i+1} \gcd(b_{i+1}, r_{i+1}) \text{ divides } b_{i+1} \gcd(m_{i+1}, r_{i+1}) \\ &\Leftrightarrow \gcd(m_{i+1} b_{i+1}, m_{i+1} r_{i+1}) \text{ divides } \gcd(m_{i+1} b_{i+1}, b_{i+1} r_{i+1}) \end{aligned}$$

But m_{i+1} divides b_{i+1}, thus $m_{i+1} r_{i+1}$ divides $b_{i+1} r_{i+1}$ and, finally, $\gcd(m_{i+1} b_{i+1}, m_{i+1} r_{i+1})$ divides $\gcd(m_{i+1} b_{i+1}, b_{i+1} r_{i+1})$.

Now, we need to prove the tricky case of $h_1 = r_1$. We prove the following result by (decreasing) induction on k, from $k = d-1$ to $k = 1$ (the case we are interested in):

Lemma 12 *If $\vec{b}$ is a candidate partitioning for p then, for $1 \leq k \leq d-1$, $\left(r_k \prod_{i=2}^{k} m_i\right)$ divides $\left(\prod_{i=1}^{k} b_i\right)$.*

Proof: Since $r_{d-1} = m_d$, the case $k = d-1$ comes from the second and third properties of Lemma 10: $r_{d-1} \prod_{i=2}^{d-1} m_i = p$ which divides $\prod_{i=1}^{d-1} b_i$ since $\vec{b}$ is a candidate partitioning for p.

Now, assume that the result is true for k: $r_k \prod_{i=2}^{k} m_i$ divides $\prod_{i=1}^{k} b_i = b_k \prod_{i=1}^{k-1} b_i$. We also know that $\prod_{i=2}^{k} m_i$ divides $\prod_{i=1}^{k-1} b_i$ (sixth property of Lemma 10), i.e., $\prod_{i=1}^{k-1} b_i = \lambda \prod_{i=2}^{k} m_i$ for some integer λ. Thus, r_k divides λb_k, and thus $\frac{r_k}{\gcd(b_k, r_k)}$ divides λ. Multiplying by $\prod_{i=2}^{k} m_i$, we obtain that $\frac{r_k m_k}{\gcd(b_k, r_k)} \prod_{i=2}^{k-1} m_i$ divides $\prod_{i=1}^{k-1} b_i$, and *a fortiori* $\frac{r_k \gcd(m_k, r_k)}{\gcd(b_k, r_k)} \prod_{i=2}^{k-1} m_i$ divides $\prod_{i=1}^{k-1} b_i$. Going back to the link between r_{k-1} and r_k, this proves that $r_{k-1} \prod_{i=2}^{k-1} m_i$ divides $\prod_{i=1}^{k-1} b_i$, which is the property for $(k-1)$. ■

We have just proved the following result.

Lemma 13 *With the vector $\vec{t}$ given by Equation 4, the mapping $M_{\vec{m}}$ is equally-many-to-one from the slab $\mathcal{I}_{\vec{b}}(d, 0)$ onto $\mathcal{I}_{\vec{m}}$, when $\vec{b}$ is a candidate partitioning for p.*

We are now ready to put all pieces together, using an induction argument.

Lemma 14 *Let $\vec{b}$ be a candidate partitioning for p. If N is built recursively in dimension $(d-1)$, just as M is built in dimension d, but for a number of processors equal to $\frac{p}{\gcd(p, b_d)}$, and if the vector $\vec{b}'$ defined by the first $(d-1)$ components of $\vec{b}$, then $M_{\vec{m}}$ is a modular mapping with the load-balancing property for $\mathcal{I}_{\vec{b}}$.*

Proof: We induce on the dimension, assuming that the construction is correct for the dimension $(d-1)$ (and this is obviously true for $d = 1$).

First, thanks to the fifth property of Lemma 10, $\vec{b}'$ is indeed a candidate partitioning for $p' = \frac{p}{\gcd(p, b_d)}$, and the vector $\vec{n}$ defined from $\vec{b}'$ and p' by Equation 3 is equal to $\vec{m}'$. Therefore, by induction hypothesis, the modular mapping defined by N and $\vec{m}'$ has the load-balancing for $\mathcal{I}_{\vec{b}'}$. Now, thanks to Lemma 9, and the fact that m_d divides b_d, we just have to show that $M_{\vec{m}}$ is equally-many-to-one from the slab $\mathcal{I}_{\vec{b}}(d, 0)$ onto $\mathcal{I}_{\vec{m}}$. This is the result of Lemma 13. ■

Figure 4 presents an implementation of the schema just presented (where the matrix M has rows and columns from 1 to d as in the presentation of this paper).

In our current implementation, we of course take the final matrix modulo the corresponding values of $\vec{m}$ (or at least in absolute value less than the corresponding value of $\vec{m}$) and, as mentioned before, we drop the first row of M and component of $\vec{m}$ (which is equal to 1). We also play some tricks, variants of the previous program (alternating signs of t for example, or pre-permuting the components of $\vec{b}$) to make coefficients smaller. We also use Theorem 3 in [11] (injectivity of $M_{\lambda\vec{m}}$ for $\mathcal{I}_{\lambda\vec{b}}$) to reduce the components of M, dividing the components of $\vec{b}$ by their gcd. But the basic kernel is the one presented above. For example, for $p = 30 = 2 \times 3 \times 5$ and the candidate partitioning $\vec{b} = (10, 15, 6)$ in dimension 3, the basic kernel leads to $\vec{m} = (1, 5, 6)$ and the mapping $(i, j, k) \to (i + j \bmod 5, k - i - 2j \bmod 6)$, which corresponds for example to the "linearization" $(i, j, k) \to 6(i + j \bmod 5) + (k - i - 2j) \bmod 6$. The appendix contains 6 tables that give the corresponding values for (i, j), when k goes from 0 to 5.

```
// Precondition: d >= 2
// input:
//   m - an integral positive modulo vector of length d
//   b - an integral positive d-vector: the number of partitionings in each dimension
// output:
//   M - an integral d x d matrix
void ModularMapping(int d, int m[d], int b[d], int M[d][d]) {
  int i, j, k;
  for (i=1; i<=d; i++) {
    for (j=1; j<=d; j++) {
      if ((j==1) || (i==j)) M[i][j] = 1; else M[i][j] = 0;
    }
  }
  for (i=2; i<=d; i++) {
    r = m[i];
    for (j=i-1; j>=2; j--) {
      t = r/gcd(r, b[j]);
      for (k=1; k<=i-1; k++) {
        M[i][k] = M[i][k] - t*M[j][k];
      }
      r = gcd(t*m[j],r);
    }
  }
}
```

Figure 4: Code for computing a modular mapping matrix M.

6 Experiments

We have incorporated support for *generalized* multipartitionings in the Rice dHPF compiler for High Performance Fortran.

Multipartitioning within the dHPF compiler is implemented as a generalization of `BLOCK`-style HPF partitionings [7, 8]. The dHPF compiler analyzes communication and reduces loop bounds as if a multipartitioned template were a standard `BLOCK` partitioned template mapped onto an array of processors of symbolic extent. The main difference comes in the interpretation that the compiler gives to the `PROCESSORS` directive. When using multipartitioning, a template cannot specify the number of processors on a per dimension basis because each hyperplane defined by a partitioning along a multipartitioned template dimension is distributed among all processors. A multipartitioned template is therefore partitioned into tiles according to the rank and extent of the virtual processor array. These tiles are then assigned to the processors as described in previous sections.

There are several important issues for generating correct and efficient code for multipartitioned distributions. First, the order in which a processor's tiles are enumerated has to satisfy any loop-carried dependences present in the original loop from which the multipartitioned loop has been generated. Second, communication that has been fully vectorized out of a loop nest should not be performed on a tile-by-tile basis; instead it should be performed for all of a processor's tiles at once. Communication aggregation is more tricky than for diagonal multipartitionings since generalized multipartitionings have multiple tiles per hyper-rectangular slab, but it is possible because generalized multipartitionings also possess the *neighbor* property described earlier in Section 1. Third, communication caused by loop-carried dependences should not be performed on a tile-by-tile basis either. Instead, communication should be vectorized for all tiles within a hyper-rectangular slab along the partitioned dimension.

By using a multipartitioned data distribution in conjunction with sophisticated data-parallel compiler optimizations, we are closing the performance gap between compiler-generated and hand-coded implementations of line-sweep computations. Earlier results and details about dHPF's compilation techniques can be found elsewhere [8, 7, 1, 2]. Here we present the result of applying generalized multipartitioning in a compiler-based parallelization of the NAS SP application benchmark [4, 8], a computational fluid dynamics code on the "class B" problem size of 102^3.

The most important analysis and code generation techniques used to obtain high-performance multipartitioned applications by the dHPF compiler are:

- partial replication of computation to reduce communication frequency and volume,
- communication vectorization,
- aggressive communication placement, and
- intra-variable and inter-variable communication aggregation.

In addition, we use an extended on-home directive (inspired by the HPF/JA `EXT_HOME` directive[14]) to partially replicate computation into a processor's shadow regions, and the HPF/JA `LOCAL` directive to eliminate unnecessary communication for values that were previously explicitly computed in a processor's shadow region.

We performed these experiments on a SGI Origin 2000 with 128 250MHz R10000 CPUs. Each CPU has 32KB of L1 instruction cache, 32KB of L1 data cache and an unified, two-way set associative L2 cache of 4MB.

Table 1 compares the performance of a hand-coded MPI version of the SP benchmark developed at NASA Ames Research Center with an MPI version generated by the dHPF compiler.[4] The hand-coded version uses 3D diagonal multipartitioning and, thus, can only be run on a perfect square number of processors. The dHPF-generated code MPI uses generalized multipartitioning which enables the code to be run on arbitrary numbers of processors. As Table 1 shows, the performance of the dHPF-generated code is quite close to (and sometimes exceeds) the performance of the hand-coded MPI for numbers of processors that are perfect squares. When the number of processors is a perfect square, the generalized multipartitionings used by the dHPF-generated code are exactly diagonal multipartitionings. These measurements show that our implementation of generalized multipartitionings is efficient in the case of diagonal multipartitionings, in which each processor has one tile per hyperplane of the partitioning. Both the hand-coded and dHPF-generated versions of SP deliver roughly linear speedup on numbers of processors that are perfect squares.

In the measurements taken of the dHPF-generated code for numbers of processors that are not perfect squares, we see that generalized multipartitionings deliver near linear speedup in these cases as well. The cases we have measured exploiting generalized multipartitioning are ones in which the factors of the number of processors are small primes. Performance would be lower for numbers of processors that are prime or have large prime factors because computation would be divided into a large number of phases and communication volume grows in proportion to the number of phases. Currently, the code generated by dHPF cannot exploit generalized multipartitionings when the

---

[4] All speedups presented are relative to the original sequential version of the code.

# CPUs	hand-coded	dHPF	% diff.
1	1.01	0.93	7.88
2		1.47	
4	2.95	3.02	-2.52
6		5.33	
8		7.79	
9	7.98	7.91	0.87
12		12.54	
16	12.04	16.80	-39.56
18		19.49	
20		19.49	
24		22.57	
25	25.97	26.78	-3.13
32		33.25	
36	38.84	40.96	-5.45
45		41.87	
49	42.06	54.25	-29.00
50		49.23	
64	66.34	67.36	-1.54
72		70.83	
81	88.32	73.50	16.78

Table 1: Comparison of hand-coded and dHPF speedups for NAS SP (class B).

block size on any processor falls below the shift width associated with communication operations, which happens when a dimension is partitioned many times (as occurs with large primes and prime factors). This limitation prevents experiments with generalized multipartitionings using the 102^3 problem size of the SP benchmark on numbers of processors that are large primes or have large prime factors.[5]

Currently, one limitation on the efficiency of the dHPF-generated code for generalized multipartitionings when the number of tiles per hyperplane of a multipartitioning is greater than one (e.g., when the number of processors in a 3D partitioning is not a perfect square) is that the dHPF-generated code does not yet exploit reuse of data tiles across multiple loop nests. For a sequence of loop nests, dHPF-generated code executes one loop nest for each of the data tiles in a hyperplane of the data and then advances to the next loop nest. For a sequence of loop nests with compatible tile enumeration order, the tile enumeration loops should be fused so that all of the compatible loop nests in the sequence are performed on one tile before advancing to the next tile. When data tiles are small enough to fit into one or more caches, this strategy would improve cache utilization by facilitating reuse of tile data among multiple loop nests.

Overall, these preliminary experiments show that generalized multipartitionings are of practical as well as theoretical interest and can be used to parallelize applications efficiently using multipartitioning on a wider class of numbers of processors than possible using only diagonal multipartitionings.

---

[5] This limitation applies only to code generated by the dHPF compiler; the *technique* of generalized multipartitioning itself is completely general.

7 Conclusions

This paper describes an algorithm for computing an optimal multipartitioning of d-dimensional arrays, $d > 2$, onto an arbitrary number of processors, p. Our algorithm minimizes cost according to an objective function that measures communication in line sweep computations. Previously, optimal multipartitionings could be computed only when $p^{\frac{1}{d-1}}$ is integral. We show that a partitioning in which the number of tiles in each hyper-rectangular slab is a multiple of the number of processors — an obvious necessary condition — is also a sufficient condition for a multipartitioned mapping of tiles to processors. We present a constructive method for building the mapping of tiles to processors using new techniques based on modular mappings and demonstrate experimentally that code using generalized multipartitionings is both scalable and efficient.

Currently, when we multipartition a d-dimensional array onto p processors, we force *all* processors to participate in the computation; however, this may lead to suboptimal performance. If the partitioning is not *compact*, *i.e.*, the number of tiles per processor is large relative to a diagonal multipartitioning (more precisely, when $\prod_{i=1}^{d} \gamma_i$ is large compared to $p^{\frac{d}{d-1}}$), and the cost of communicating at tile boundaries is not small compared to the cost of the computation on tile data (the relative cost of communication to computation is proportional to the surface to volume ratio in the partitioning: $\sum_{i=1,d} \frac{\gamma_i}{\eta_i}$), it will be faster to drop back to a nearby lower number of processors for which a compact partitioning exists. For example, table 1 shows that for the 102^3 problem size, a $5 \times 10 \times 10$ decomposition on 50 processors is slower than a $7 \times 7 \times 7$ decomposition on 49 processors for NAS SP. Given a cost function (see Section 4.1) that models the cost of computation as well as communication, our algorithm could be used to search for the most efficient partitioning, which will occur on some number of processors between $\lfloor p^{\frac{1}{d-1}} \rfloor^{d-1}$ (for which a diagonal multipartitioning is possible) and p as long as the communication term is not dominant.

References

[1] Vikram Adve, Guohua Jin, John Mellor-Crummey, and Qing Yi. High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes. In *Proceedings of SC98: High Performance Computing and Networking*, Orlando, FL, Nov 1998.

[2] Vikram Adve and John Mellor-Crummey. Using Integer Sets for Data-Parallel Program Analysis and Optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[3] George E. Andrews. *Theory of Partitions*, volume 2 of *Encyclopedia of Mathematics and its applications*. Addison-Wesley, 1976.

[4] D. Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[5] J. Bruno and P. Cappello. Implementing the beam and warming method on the hypercube. In *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 1073–1087, Pasadena, CA, January 1988.

[6] John Bruno and Peter Cappello. Implementing the 3D alternating direction method on the hypercube. *Journal of Parallel and Distributed Computing*, 23(3):411–417, December 1994.

[7] Daniel Chavarría-Miranda and John Mellor-Crummey. Towards compiler support for scalable parallelism. In *Proceedings of the Fifth Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, Lecture Notes in Computer Science 1915, pages 272–284, Rochester, NY, May 2000. Springer-Verlag.

[8] Daniel Chavarría-Miranda, John Mellor-Crummey, and Trushar Sarang. Data-parallel compiler support for multipartitioning. In *European Conference on Parallel Computing (Euro-Par)*, Manchester, United Kingdom, August 2001.

[9] A. Darte, R. Schreiber, B. R. Rau, and F. Vivien. A constructive solution to the juggling problem in systolic array synthesis. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'00)*, pages 815–821, Cancun, Mexico, May 2000.

[10] Alain Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *INTEGRATION, The VLSI Journal*, pages 293–304, 1991.

[11] Alain Darte, Michèle Dion, and Yves Robert. A characterization of one-to-one modular mappings. *Parallel Processing Letters*, 5(1):145–157, 1996.

[12] Alain Darte, John Mellor-Crummey, Robert Fowler, and Daniel Chavarría. On efficient parallelization of line-sweep computations. Technical Report CS-TR01-377, Dept. of Computer Science, Rice University, April 2001.

[13] J. Dénes and A. D. Keedwell. *Latin Squares: New Developments in the Theory and Applications.* North Holland, 1991.

[14] Japanese Association for High Performance Fortran. HPF/JA language specification (version 1.0). Available at URL `http://www.tokyo.rist.or.jp/jahpf/spec/index-e.html`, January 1999.

[15] Lazslo Fuchs. *Abelian Groups.* Pergamon Press, Los Angeles, CA, 1960.

[16] M. Garey and D. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., New York, NY, 1979.

[17] Frank Gray. Pulse code communication. United States Patent Number 2,632,058. March 17, 1953. Available online at `http://patft.uspto.gov/netahtml/srchnum.htm`.

[18] G. Hajós. Über einfache und mehrfache Bedeckung des n-dimensionalen Raumes mit einen Würfelgitter. *Math. Zschrift*, 47:427–467, 1942.

[19] G. H. Hardy and E. M. Wright. *Introduction to the Theory of Numbers.* Oxford University Press, 1979.

[20] S. Lennart Johnsson, Youcef Saad, and Martin H. Schultz. Alternating direction methods on multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 8(5):686–700, 1987.

[21] Hyuk J. Lee and José A.B. Fortes. On the injectivity of modular mappings. In Peter Cappello, Robert M. Owens, Jr Earl E. Swartzlander, and Benjamin W. Wah, editors, *Application Specific Array Processors*, pages 237–247, San Francisco, California, August 1994. IEEE Computer Society Press.

[22] N.H. Naik, V. Naik, and M. Nicoules. Parallelization of a class of implicit finite-difference schemes in computational fluid dynamics. *International Journal of High Speed Computing*, 5(1):1–50, 1993.

[23] Joe Sawada. C program for computing all numerical partitions of n whose largest part is k. Information on Numerical Partitions, Combinatorial Object Server, University of Victoria, `http://www.theory.csc.uvic.ca/~cos/inf/nump/NumPartition.html`, 1997.

[24] N. J. A. Sloane. The on-line encyclopedia of integer sequences. `http://www.research.att.com/~njas/sequences`, 2001.

[25] R. F. Van der Wijngaart. Efficient implementation of a 3-dimensional ADI method on the iPSC/860. In *Proceedings of Supercomputing 1993*, pages 102–111. IEEE Computer Society Press, 1993.

A A Generalized Multipartitioning Example

Here we show an example generalized multipartitioning for 30 processors ($p = 2 \times 3 \times 5$). Assuming that we are partitioning a cube (namely, $\lambda_1 = \lambda_2 = \lambda_3$), our partitioning algorithm selects a three-dimensional elementary partitioning $\vec{\gamma} = (10, 15, 6)$. The basic kernel leads to $\vec{m} = (1, 5, 6)$ and the modular mapping $(i, j, k) \rightarrow (i + j \bmod 5, k - i - 2j \bmod 6)$. Linearization of this mapping gives the tile-to-processor mapping $\theta(i, j, k) \rightarrow 6(i + j \bmod 5) + (k - i - 2j) \bmod 6$. The following 6 tables show the corresponding values for each (i, j) plane, for $0 \leq k \leq 5$.

The brave reader can check that this is indeed a correct three-dimensional multipartitioning that obeys the balance and neighbor properties.

0	11	16	21	26	1	6	17	22	27
10	15	20	25	0	11	16	21	26	1
14	19	24	5	10	15	20	25	0	11
18	29	4	9	14	19	24	5	10	15
28	3	8	13	18	29	4	9	14	19
2	7	12	23	28	3	8	13	18	29
6	17	22	27	2	7	12	23	28	3
16	21	26	1	6	17	22	27	2	7
20	25	0	11	16	21	26	1	6	17
24	5	10	15	20	25	0	11	16	21
4	9	14	19	24	5	10	15	20	25
8	13	18	29	4	9	14	19	24	5
12	23	28	3	8	13	18	29	4	9
22	27	2	7	12	23	28	3	8	13
26	1	6	17	22	27	2	7	12	23

$k = 0$

1	6	17	22	27	2	7	12	23	28
11	16	21	26	1	6	17	22	27	2
15	20	25	0	11	16	21	26	1	6
19	24	5	10	15	20	25	0	11	16
29	4	9	14	19	24	5	10	15	20
3	8	13	18	29	4	9	14	19	24
7	12	23	28	3	8	13	18	29	4
17	22	27	2	7	12	23	28	3	8
21	26	1	6	17	22	27	2	7	12
25	0	11	16	21	26	1	6	17	22
5	10	15	20	25	0	11	16	21	26
9	14	19	24	5	10	15	20	25	0
13	18	29	4	9	14	19	24	5	10
23	28	3	8	13	18	29	4	9	14
27	2	7	12	23	28	3	8	13	18

$k = 1$

$$\begin{vmatrix}
2 & 7 & 12 & 23 & 28 & 3 & 8 & 13 & 18 & 29 \\
6 & 17 & 22 & 27 & 2 & 7 & 12 & 23 & 28 & 3 \\
16 & 21 & 26 & 1 & 6 & 17 & 22 & 27 & 2 & 7 \\
20 & 25 & 0 & 11 & 16 & 21 & 26 & 1 & 6 & 17 \\
24 & 5 & 10 & 15 & 20 & 25 & 0 & 11 & 16 & 21 \\
4 & 9 & 14 & 19 & 24 & 5 & 10 & 15 & 20 & 25 \\
8 & 13 & 18 & 29 & 4 & 9 & 14 & 19 & 24 & 5 \\
12 & 23 & 28 & 3 & 8 & 13 & 18 & 29 & 4 & 9 \\
22 & 27 & 2 & 7 & 12 & 23 & 28 & 3 & 8 & 13 \\
26 & 1 & 6 & 17 & 22 & 27 & 2 & 7 & 12 & 23 \\
0 & 11 & 16 & 21 & 26 & 1 & 6 & 17 & 22 & 27 \\
10 & 15 & 20 & 25 & 0 & 11 & 16 & 21 & 26 & 1 \\
14 & 19 & 24 & 5 & 10 & 15 & 20 & 25 & 0 & 11 \\
18 & 29 & 4 & 9 & 14 & 19 & 24 & 5 & 10 & 15 \\
28 & 3 & 8 & 13 & 18 & 29 & 4 & 9 & 14 & 19
\end{vmatrix}$$

$k = 2$

$$\begin{vmatrix}
3 & 8 & 13 & 18 & 29 & 4 & 9 & 14 & 19 & 24 \\
7 & 12 & 23 & 28 & 3 & 8 & 13 & 18 & 29 & 4 \\
17 & 22 & 27 & 2 & 7 & 12 & 23 & 28 & 3 & 8 \\
21 & 26 & 1 & 6 & 17 & 22 & 27 & 2 & 7 & 12 \\
25 & 0 & 11 & 16 & 21 & 26 & 1 & 6 & 17 & 22 \\
5 & 10 & 15 & 20 & 25 & 0 & 11 & 16 & 21 & 26 \\
9 & 14 & 19 & 24 & 5 & 10 & 15 & 20 & 25 & 0 \\
13 & 18 & 29 & 4 & 9 & 14 & 19 & 24 & 5 & 10 \\
23 & 28 & 3 & 8 & 13 & 18 & 29 & 4 & 9 & 14 \\
27 & 2 & 7 & 12 & 23 & 28 & 3 & 8 & 13 & 18 \\
1 & 6 & 17 & 22 & 27 & 2 & 7 & 12 & 23 & 28 \\
11 & 16 & 21 & 26 & 1 & 6 & 17 & 22 & 27 & 2 \\
15 & 20 & 25 & 0 & 11 & 16 & 21 & 26 & 1 & 6 \\
19 & 24 & 5 & 10 & 15 & 20 & 25 & 0 & 11 & 16 \\
29 & 4 & 9 & 14 & 19 & 24 & 5 & 10 & 15 & 20
\end{vmatrix}$$

$k = 3$

$$\begin{vmatrix}
4 & 9 & 14 & 19 & 24 & 5 & 10 & 15 & 20 & 25 \\
8 & 13 & 18 & 29 & 4 & 9 & 14 & 19 & 24 & 5 \\
12 & 23 & 28 & 3 & 8 & 13 & 18 & 29 & 4 & 9 \\
22 & 27 & 2 & 7 & 12 & 23 & 28 & 3 & 8 & 13 \\
26 & 1 & 6 & 17 & 22 & 27 & 2 & 7 & 12 & 23 \\
0 & 11 & 16 & 21 & 26 & 1 & 6 & 17 & 22 & 27 \\
10 & 15 & 20 & 25 & 0 & 11 & 16 & 21 & 26 & 1 \\
14 & 19 & 24 & 5 & 10 & 15 & 20 & 25 & 0 & 11 \\
18 & 29 & 4 & 9 & 14 & 19 & 24 & 5 & 10 & 15 \\
28 & 3 & 8 & 13 & 18 & 29 & 4 & 9 & 14 & 19 \\
2 & 7 & 12 & 23 & 28 & 3 & 8 & 13 & 18 & 29 \\
6 & 17 & 22 & 27 & 2 & 7 & 12 & 23 & 28 & 3 \\
16 & 21 & 26 & 1 & 6 & 17 & 22 & 27 & 2 & 7 \\
20 & 25 & 0 & 11 & 16 & 21 & 26 & 1 & 6 & 17 \\
24 & 5 & 10 & 15 & 20 & 25 & 0 & 11 & 16 & 21
\end{vmatrix}$$

$k = 4$

$$\begin{vmatrix}
5 & 10 & 15 & 20 & 25 & 0 & 11 & 16 & 21 & 26 \\
9 & 14 & 19 & 24 & 5 & 10 & 15 & 20 & 25 & 0 \\
13 & 18 & 29 & 4 & 9 & 14 & 19 & 24 & 5 & 10 \\
23 & 28 & 3 & 8 & 13 & 18 & 29 & 4 & 9 & 14 \\
27 & 2 & 7 & 12 & 23 & 28 & 3 & 8 & 13 & 18 \\
1 & 6 & 17 & 22 & 27 & 2 & 7 & 12 & 23 & 28 \\
11 & 16 & 21 & 26 & 1 & 6 & 17 & 22 & 27 & 2 \\
15 & 20 & 25 & 0 & 11 & 16 & 21 & 26 & 1 & 6 \\
19 & 24 & 5 & 10 & 15 & 20 & 25 & 0 & 11 & 16 \\
29 & 4 & 9 & 14 & 19 & 24 & 5 & 10 & 15 & 20 \\
3 & 8 & 13 & 18 & 29 & 4 & 9 & 14 & 19 & 24 \\
7 & 12 & 23 & 28 & 3 & 8 & 13 & 18 & 29 & 4 \\
17 & 22 & 27 & 2 & 7 & 12 & 23 & 28 & 3 & 8 \\
21 & 26 & 1 & 6 & 17 & 22 & 27 & 2 & 7 & 12 \\
25 & 0 & 11 & 16 & 21 & 26 & 1 & 6 & 17 & 22
\end{vmatrix}$$

$k = 5$