Efficient Parallelization of Line-Sweep Computations

Daniel Chavarría-Miranda, Alain Darte, Robert Fowler, John Mellor-Crummey

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 8512

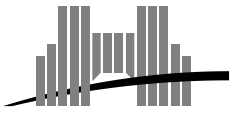
CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

On Efficient Parallelization of Line-Sweep Computations

Daniel Chavarría-Miranda,
Alain Darte, Robert Fowler,
and John Mellor-Crummey

November 2001

Research Report N° 2001-45



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

INRIA

On Efficient Parallelization of Line-Sweep Computations

Daniel Chavarría-Miranda,
Alain Darte, Robert Fowler,
and John Mellor-Crummey

November 2001

Abstract

Multipartitioning is a strategy for partitioning multi-dimensional arrays on a collection of processors. With multipartitioning, computations that require solving 1D recurrences along each dimension of a multi-dimensional array can be parallelized effectively. Previous techniques for multipartitioning yield efficient parallelizations over 3D domains only when the number of processors is a perfect square. This paper considers the general problem of computing optimal multipartitionings for d-dimensional data volumes on an arbitrary number of processors. We describe an algorithm that computes an optimal multipartitioning for this general case, which enables efficient parallelizations of line-sweep computations under arbitrary conditions. Finally, we describe a prototype implementation of generalized multipartitioning in the Rice dHPF compiler and performance results obtained when using it to parallelize a line sweep computation for different numbers of processors.

Keywords: loop parallelization, array mapping, generalized latin squares, High Performance Fortran

Résumé

Le "multi-partitionnement" est une stratégie de partitionnement multi-dimensionnel de tableaux sur un ensemble de processeurs. Grâce au multi-partitionnement, des calculs qui nécessitent la résolution de récurrences 1D le long de chaque dimension d'un tableau multi-dimensionnel peut être parallélisée efficacement. Les techniques précédentes de multi-partitionnement fournissaient des parallélisations efficaces pour des tableaux 3D mais uniquement pour un nombre de processeurs égal à un carré parfait. Dans ce rapport, nous considérons le problème général du calcul d'un multi-partitionnement optimal pour des volumes de données en dimension d pour un nombre quelconque de processeurs. We donnons un algorithme qui calcule un multi-partitionnement optimal pour ce cas général ce qui permet la parallélisation efficace des calculs par vagues 1D dans n'importe quelles conditions. Finalement, nous décrivons également un prototype d'implantation dans le compilateur dHPF de Rice et les résultats de performances obtenues lorsque le nombre de processeurs varie.

Mots-clés: parallélisation de boucles, allocation de tableaux, carrés latins généralisés, High Performance Fortran

Contents

On Efficient Parallelization of 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

11th December 2001

Abstract

Multipartitioning is a strategy for partitioning multi-dimensional arrays among a collection of processors so that line-sweep computations can be performed efficiently. The principal property of a multipartitioned array is that for a line sweep along any array dimension, all processors have the same number of tiles to compute at each step in the sweep. This property results in full, balanced parallelism. A secondary benefit of multipartitionings is that they induce only coarse-grain communication.

All of the multipartitionings described in the literature to date assign only one tile per processor per hyperplane of a multipartitioning. While this class of multipartitionings is optimal for two dimensions, in three dimensions it requires the number of processors to be a perfect square. This paper considers the general problem of computing optimal multipartitionings for multi-dimensional data volumes on an arbitrary number of processors. We describe an algorithm to compute a d-dimensional multipartitioning of a multi-dimensional array for an arbitrary number of processors. When using a multipartitioning to parallelize a line sweep computation, the best partitioning is the one that exploits all of the processors and has the smallest communication volume. To compute the best multipartitioning of a multi-dimensional array, we describe a cost model for selecting d, the dimensionality of the best partitioning, and the number of cuts along each partitioned dimension. In practice, our technique will choose a 3-dimensional multipartitioning for a 3-dimensional line-sweep computation, except when p is a prime; previously, a 3-dimensional multipartitioning could be applied only when $\sqrt{p}$ is integral.

Finally, we describe a prototype implementation of generalized multipartitioning in the Rice dHPF compiler and performance results obtained when using it to parallelize a line sweep computation for different numbers of processors.

1 Introduction

Line sweeps are used to solve one-dimensional recurrences along each dimension of a multi-dimensional discretized domain. This computational method is the basis for Alternating Direction Implicit (ADI) integration – a widely-used numerical technique for solving partial differential equations such as the Navier-Stokes equation [5, 16, 18] – and is also at the heart of a variety of other numerical methods and solution techniques [18]. Parallelizing computations based on line

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

sweeps is important because these computations address important classes of problems and they are computationally intensive.

The recurrences along a dimension that line sweeps are used to solve serialize computation of each line along that dimension. If a dimension with such recurrences is partitioned, it induces serialization between computations on different processors. Using standard block unipartitionings, in which each processor is assigned a single hyper-rectangular block of data, there are two classes of alternative partitionings. *Static block unipartitionings* involve partitioning some set of dimensions of the data domain, and assigning each processor one contiguous hyper-rectangular volume. To achieve significant parallelism for a line sweep computation with this type of partitionings requires exploiting wavefront parallelism within each sweep. 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. *Dynamic block unipartitionings* involve partitioning a single data dimension, performing line sweeps in all unpartitioned data dimensions locally, transposing the data to localize the data along the previously partitioned dimension, and then performing the remaining sweep locally. While dynamic block unipartitionings achieve better efficiency during a (local) sweep over a single dimension compared to a (wavefront) sweep using static block unipartitionings, they require transposing *all* of the data to perform a complete set of sweeps, whereas static block unipartitionings communicate only data at partition boundaries.

To support better parallelization of line sweep computations, a third sophisticated strategy for partitioning data and computation known as *multipartitioning* was developed [5, 16, 18]. Multipartitioning distributes arrays of two or more dimensions among a set of processors so that for computations performing a directional sweep along any one of the array's data dimensions, (1) all processors are active in each step of the computation, (2) load-balance is nearly perfect, and (3) only a modest amount of coarse-grain communication is needed. These properties are achieved by carefully assigning each processor a balanced number of tiles between each pair of adjacent hyperplanes that are defined by the cuts along any partitioned data dimension. We describe multipartitionings in Section 2. A study by van der Wijngaart [21] of implementation strategies for hand-coded parallelizations of ADI Integration found that 3D multipartitionings yield better performance than both static block unipartitionings and dynamic block unipartitionings.

All of the multipartitionings described in the literature to date consider only one tile per processor per hyperplane along a partitioned dimension, even the most general class known as *diagonal multipartitionings*. While diagonal multipartitionings are always possible in two dimensions, they require in 3D that the number of processors is a perfect square. We consider the general problem of computing optimal multipartitionings for d-dimensional data volumes on an arbitrary number of processors, which enables efficient parallelizations of line-sweep computations under arbitrary conditions.

In the next section, we describe prior work in multipartitioning. Then, we present our strategy for computing generalized multipartitionings. This has three parts: an objective function for computing the cost of a line sweep computation for a given multipartitioning, a cost-model-driven algorithm for computing the dimensionality and tile size of the best multipartitioning, and an algorithm for computing a mapping of tiles to processors. Finally, we describe a prototype implementation of generalized multipartitioning in the Rice dHPF compiler for High Performance Fortran. We report preliminary performance results obtained using it to parallelize a computational fluid dynamics benchmark.

2 Background

Johnsson *et al.* [16] 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. They use a 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] 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 on to 2^{2d} processors They use a tile to processor mapping $\theta(i,j,k)$ based on Gray codes. A Gray code $g_s(r)$ 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. They 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.* [18] describe *diagonal multipartitionings* for two and three dimensional problems. Diagonal multipartitionings are a generalization of Johnsson *et al*'s 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 arranged along diagonals through each of the partitioned dimensions. Figure 1 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 $0 \le i,j,k < \sqrt{p}$.

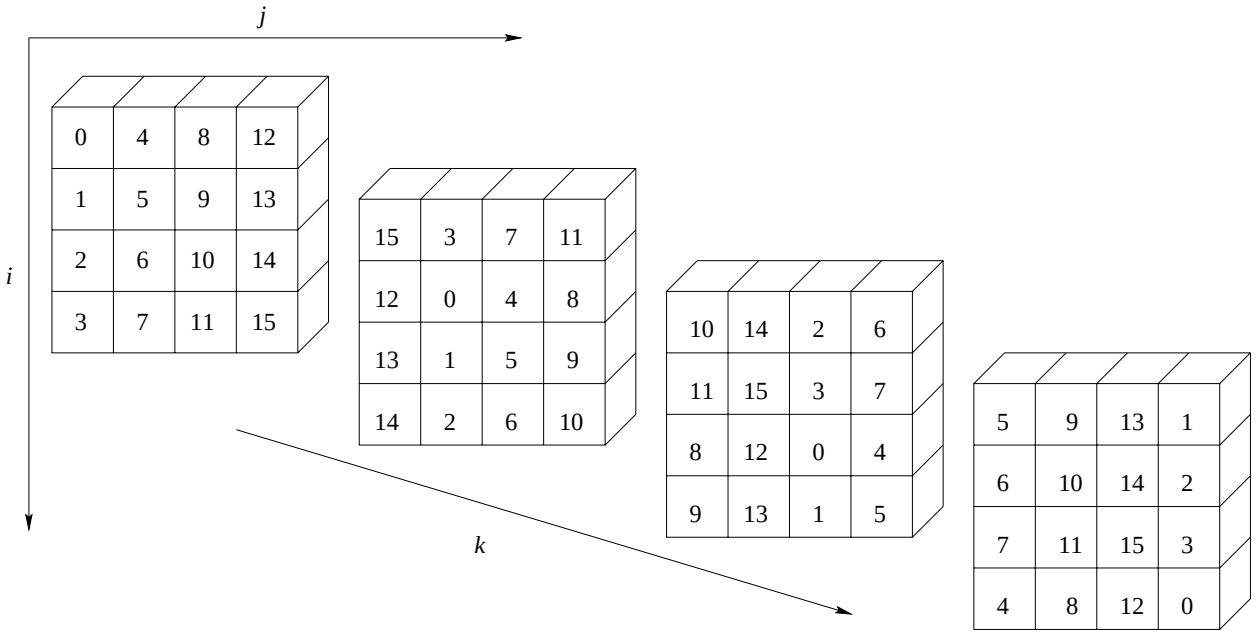


Figure 1: 3D Multipartitioning on 16 processors.

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 yields a unique optimal multipartitioning (equivalent to the class of partitionings described by Johnsson *et al.* [16]). 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 hyperplane of a multipartitioning. [5] How general can multipartitioning mappings be? Obviously, to support load-balanced line-sweep computation, in any hyperplane of the partitioning, each processor must have the same number of tiles. We call any hyperplane in which each processor has the same number of tiles *balanced*. This raises the question: can we find a way to partition a d-dimensional array into tiles and assign the tiles to processors so that each hyperplane is balanced? The answer is yes. However, such an assignment is possible if and only if the number of tiles in each hyperplane along any dimension is a multiple of p. We describe a "regular" solution (regular to be defined) to this general problem that enables us to guarantee that the neighboring tiles of a processor's tiles along a direction of a data dimension all belong to a single processor — an important property for efficient computation on a multipartitioned distribution.

In Section 4, we define an objective function that represents the execution time of a line-sweep computation over a multipartitioned array. In Section 5, we present an algorithm that computes a partitioning of a multidimensional array into tiles that is optimal with respect to this objective. In Section 6, we develop a general theory of modular mappings for multipartitioning. We apply this theory to define a mapping of tiles to processors so that each line sweep is perfectly balanced over the processors.

We use the following notation in the subsequent sections:

- 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.
- 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 $n = \prod_{i=1}^{d} n_i$.
- γ_i, for $1 \leq i \leq d$, is the number of tiles into which the array is cut along its i-th dimension. We consider the 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 alignment or boundary problems that must be handled when applying our mappings in practice if this assumption is not valid.

To ensure each hyperplane is balanced, the number of tiles it contains must be a multiple of p; namely, for each $1 \leq i \leq d$, p should divide $\prod_{\substack{j=1 \\ j \neq i}}^{d} \gamma_j$.

4 Objective Function

We consider the cost of performing a line sweep computation along each dimension of a multipartitioned array. The total computation cost is proportional to the number of elements in the array,

n. A sweep along the i-th dimension consists of a sequence of γ_i computation phases (one for each hyperplane of tiles along dimension i), separated by $\gamma_i - 1$ communication phases. The work in each hyperplane is perfectly balanced, with each processor performing the computation for its own tiles. The total computational work for each processor is roughly $\frac{1}{p}$ of the total work in the sequential computation. The communication overhead is a function of the number of communication phases and the communication volume. Between two computation phases, a hyperplane of array elements is transmitted – the boundary layer for all tiles computed in first phase. The total communication volume for a phase communicated along dimension i is $\prod_{\substack{j=1 \\ j \neq i}}^{d} n_j$ elements, i.e., $\frac{n}{n_i}$. Therefore, the total execution time for a sweep along dimension i can be approximated by the following formula:

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

where K_1 is a constant that depends on the sequential computation time, K_2 is a constant that depends on the cost of initiating one communication phase (start-up), and K_3 is a constant that depends of the cost of transmitting one array element. The total cost of the algorithm, sweeping in all dimensions, is thus

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

Remark: we assume here that the cost of one communication phase is actually an affine function of the volume of transmitted data. If all communications are in fact performed with perfect parallelism, with no overhead, then the third term is divided by p. However, this will not change the technique we describe later.

Assuming that p, n, and the n_i's are given, the only term that can be optimized is $\sum_{i=1}^{d} \gamma_i \lambda_i$ where each $\lambda_i = K_2 + K_3 \frac{n}{n_i}$ is a constant that depends upon the domain size and the machine's communication parameters (and possibly the number of processors with the remark above).

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 $\sum_i \frac{\gamma_i}{n_i}$, which means it is preferable to partition dimensions that are larger into relatively more pieces. For example, in 3D, even for a square number of processors (e.g., $p = 4$), if the data domain has one very small dimension, then it is preferable to use a 2D partitioning with the two larger ones rather than a 3D partitioning. Indeed, 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 smaller volume of communication than a "classical" 3D partitioning in which each dimension is cut into 2 pieces ($\gamma_1 = \gamma_2 = \gamma_3 = 2$). The extra communication while sweeping along the first two dimensions is offset by the absence of communication in the local sweep along the last dimension.

5 Finding the Partitioning

In this section, we address the problem of minimizing $\sum_i \gamma_i \lambda_i$ for general λ_i's, with the constraint that, for any fixed i, p divides the product of the γ_j's excluding γ_i. We give a practical algorithm, based on an exhaustive search, exponential in s and the r_i's (see the decomposition of p into prime factors), but whose complexity in p grows slowly. Furthermore, in practice, the algorithm is much

faster than the exponential worse-case may suggest, both because the average complexity is much lower and because practical p's are not very large.

From a theoretical point of view, we do not know whether this minimization problem is NP-complete, even for a fixed dimension $d \geq 3$, even if all λ_i are equal to 1, or if an algorithm polynomial in $\log p$ or even in $\log s$ and the $\log r_i$'s exists. The only NP-complete problem we found in the literature related to product of numbers is the Subset Product Problem (SPP) [13], which is weakly NP-complete (it can be solved thanks to dynamic programming). We suspect that our problem is strongly NP-complete, even if the input is s and the r_i's, instead of p. However, if p has only one prime factor, we point out that a greedy approach leads to a polynomial (i.e., polynomial in $\log r$) algorithm (see Section 5.1).

We say that $(\gamma_i)_{1 \leq i \leq d}$ – or (γ_i) for short [1] – is a **valid solution** if, for each $1 \leq i \leq d$, p divides $\prod_{\substack{j=1 \\ j \neq i}}^{d} \gamma_j$. Furthermore, if $\sum_i \gamma_i \lambda_i$ is minimized, we say that (γ_i) is an **optimal solution**. We start with some basic properties of valid and optimal solutions.

Lemma 1 *Let (γ_i) be given. Then, (γ_i) is a valid solution if and only if, for each factor α of p, appearing r_α times in the decomposition of p, the total number of occurrences of α in all γ_i is at least $r_\alpha + m_\alpha$, where m_α is the maximum number of occurrences of α in any γ_i.*

Proof: Suppose that (γ_i) is a valid solution. Let α be a factor of p appearing r_α times in the decomposition of p, let m_α be the maximum number of occurrences of α in any γ_i, and let i_0 be such that α appears m_α times in γ_{i_0}. Since p divides the product of all γ_i excluding γ_{i_0}, α appears at least r_α times in this product. The total number of occurrences of α in all of the γ_i is thus at least $r_\alpha + m_\alpha$. Conversely, if this property is true for any factor α, then for any product of $(d-1)$ different γ_i's, the number of occurrences of α is at least $r_\alpha + m_\alpha$ minus the number of occurrences in the γ_i that is not part of the product, and thus is at least r_α. Therefore, p divides this product and (γ_i) is a valid solution. ■

Thanks to Lemma 1, we can interpret (and manipulate) a valid solution (γ_i) as a distribution of the factors of p into d bins. If a factor α appears r_α times in p, it should appear $(r_\alpha + m_\alpha)$ times in the d bins, where m_α is the maximal number of occurrences of α in a bin. As far as the minimization of $\sum_i \lambda_i \gamma_i$ is concerned, no other prime number should appear in the γ_i without increasing the objective function. The following lemma shows that, for an optimal solution, there should be exactly $(r_\alpha + m_\alpha)$ occurrences for each factor α and that the maximum m_α should appear in at least two bins.

Lemma 2 *Let (γ_i) be an optimal solution. Then, each factor α of p, appearing r_α times in the decomposition of p, appears exactly $(r_\alpha + m_\alpha)$ times in (γ_i), where m_α is the maximum number of occurrences of α in any particular γ_i. Furthermore, the number of occurrences of α is m_α in at least two γ_i's.*

Proof: Let (γ_i) be an optimal solution. By Lemma 1, each factor α, $0 \leq j < s$, that appears r_α times in p, appears at least $(r_\alpha + m_\alpha)$ times in (γ_i). The following arguments hold independently for each factor α.

[1] In other words, (γ_i) denotes the set of all particular γ_i's as opposed to a particular γ_i

Suppose m_α occurrences of α appear in some γ_{i_0} and no other γ_i. Remove one α from γ_{i_0}. Now, the maximum number of occurrences of α in any γ_i is $m_\alpha - 1$ and we have $(r_\alpha + m_\alpha) - 1 = r_\alpha + (m_\alpha - 1)$ occurrences of α. By Lemma 1, we still have a valid solution, and with a smaller cost. This contradicts the optimality of (γ_i). Thus, there are at least two bins with m_α occurrences of α.

If c, the number of occurrences of α in (γ_i), is such that $c > r_\alpha + m_\alpha$, then we can remove one α from any nonempty bin. We now have $c - 1 \geq r_\alpha + m_\alpha$ occurrences of α and the maximum is still m_α (since at least two bins had m_α occurrences of α). Therefore, according to Lemma 1, we still have a valid solution, and with smaller cost, again a contradiction. ■

We call **locally optimal solutions** the valid solutions that satisfy Lemma 2. These solutions are the minima of the partial order defined on valid solutions with the relation $S = (\gamma_i) \leq S' = (\gamma'_i)$ if γ_i divides γ'_i for all i. Since all λ_i are positive, only locally optimal solutions need to be considered to find an optimal solution. Note however that there are locally optimal solutions that can never be optimal. For example, in 3D, for $p = 2^2 \times 3^3 \times 5^2$, the locally optimal solution $(2^2 \times 3^2, 2^2 \times 5^2, 3^2 \times 5^2) = (36, 100, 225)$ (and its permutations) can never be optimal since the solution $(30, 30, 30)$ is always better, whatever the positive λ_i's. However, with a 7 instead of a 5, these solutions can now be optimal for some particular λ_i's since 36 is less than $2 \times 3 \times 7 = 42$.

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

Lemma 3 *In any optimal solution, for any factor α appearing r_α times in the decomposition of p, we have $\lceil \frac{r_\alpha}{d-1} \rceil \leq m_\alpha \leq r_\alpha$ where m_α is the maximal number of occurrences of α in any bin and d is the number of bins.*

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

5.1 A Greedy Algorithm for Partitioning when p is a Power of a Prime

Here we describe a greedy algorithm for computing an optimal partitioning in the simplest case, namely, when p is a power of a prime, $p = \alpha^r$. The algorithm has to apportion factors of α among d bins (one for each array dimension) in a way that satisfies Lemma 2 and minimizes the total cost. Satisfying Lemma 2 requires that the total number of occurrences of α in all of the bins must be $r + m$, where m is the maximum in any one bin, and at least two bins contain m instances of α. The cost of the i-th bin in the solution is $\alpha^{\beta_i}\lambda_i$, where β_i is the number of instances of α in the bin. This yields a total cost of $\sum_{i=1}^{d} \alpha^{\beta_i}\lambda_i$. A greedy algorithm is shown in Figure 2. We show that this algorithm computes the optimal partitioning when $p = \alpha^r$ and that the complexity of this algorithm is $O((d + \log p)\log d)$.

Theorem 1 *The greedy algorithm shown in Figure 2 is optimal if p is a power of a prime.*

Proof: We prove the result by induction on r. Let us first consider the case $r = 1$. We need to put $(r + m)$ copies of α in the bins and $m \leq r$, according to Lemma 3. Therefore, any optimal solution has $(d-2)$ empty bins and 2 bins with one element, and the greedy algorithm computes an optimal solution in one iteration of the while loop by applying case 2.

inputs:
d: the number of dimension, $d \geq 2$; let D represent $\{i \mid 1 \leq i \leq d\}$.
r: $p = \alpha^r$ and α is prime.
λ_i, $i \in D$: the cost coefficient for each dimension.
outputs:
β_i, $i \in D$: the power of α in the bin for each dimension.

Algorithm GreedyPartitioning
let t = m = 0; // *t is the total number of α placed, m is the maximum number of α in any bin*
for all $i \in D$, **let** $\beta_i = 0$ and $C(i) = \lambda_i$; // *$C(i)$ is the current cost of dimension i*
let $C(0) = +\infty$; // *add a dummy dimension to simplify notations*
while (t < r + m)
let $\mathcal{B}_m = \{i \mid i \in D \wedge \beta_i = m\}$; **let** $s = \{0\} \cup (D \setminus \mathcal{B}_m)$; // *add $\{0\}$ to s to avoid an empty set*
let $j \in s$ be the dimension with smallest cost $C(j)$;
let $k, l \in \mathcal{B}_m$ be two dimensions with smallest costs $C(k)$ and $C(l)$;
if $C(j) < C(k) + C(l)$ **then** // *add one α to bin j, increase t* **(case 1)**
β_j += 1; t += 1; $C(j) = \alpha C(j)$;
else // *add one α to bins k and l, and increase both m and t as appropriate* **(case 2)**
β_k += 1; $C(k) = \alpha C(k)$; β_l += 1; $C(l) = \alpha C(l)$; m += 1; t += 2;
endif
endwhile

Figure 2: Greedy Algorithm for Partitioning r Instances of α among d Dimensions.

We now prove the induction step. Assume that the greedy algorithm gives an optimal solution for α^r. Let us show that it also gives an optimal solution for α^{r+1}. Denote by S_r the solution obtained by the greedy algorithm for r and let $\mathcal{B}_m$ be the set of bins with maximal size for S_r. To build the greedy solution for $r+1$, we evaluate the cost of two alternatives: S_1 and S_2. S_1 extends S_r by adding a single element into a bin b_j not in $\mathcal{B}_m$ with smallest cost (case 1). S_2 extends S_r by adding two elements into two bins b_k and b_l, both in $\mathcal{B}_m$, with smallest costs (case 2). The greedy algorithm selects the lowest cost solution of S_1 and S_2 as S_{r+1}. Before showing that either S_1 or S_2 is an optimal solution, let us make two simple remarks:

- If we remove one element in any bin of a valid solution for α^{r+1}, we obtain a valid solution for α^r. Indeed, because of Lemma 1, if the total number of elements is at least $r+m$, then after removing one element, the total number of elements is at least $r+m-1$, which is at least $(r-1)+m'$ where $m' \leq m$ is the new maximum.

- Let T and U be two valid solutions for α^{r+1} such that the cost of T is less than the cost of U. Assume that there is a bin b_i such that b_i has β_T elements in T and β_U elements in U with $\beta_T \geq \beta_U \geq 1$. Then, if we remove one element in this bin both in T and U, we obtain two valid solutions T' and U' for α^r (according to the previous remark) such that the cost of T' is less than the cost of U'. Indeed, the cost of the bin b_i goes down from $\lambda_i \alpha^{\beta_T}$ to $\lambda_i \alpha^{\beta_T - 1}$ for T, and is thus reduced by $\lambda_i \alpha^{\beta_T}(1 - 1/\alpha)$. The reduction for U can be computed analogously. Therefore, since $\beta_T \geq \beta_U$, the cost decrease is higher for T than for U, and since the cost of T was less than the cost of U, this remains true for T' and U'.

We now go back to the greedy algorithm. Suppose there exists a solution S' that is strictly

better than S_{r+1}. We show that this is not possible. We first show that no bin in S' can have strictly more elements than in S_r. We keep the notations of Figure 2 for the indices j, k, and l: S_1 is obtained from S_r by adding one element in bin b_j, S_2 by adding one element both in bins b_k and b_l. We distinguish bins in $\mathcal{B}_m$ (those with maximal size in S_r) from the other ones.

- If S' has a bin b_i not in $\mathcal{B}_m$ with strictly more elements than the corresponding bin in S_r, we move in S_1 one element from the bin b_j to the bin b_i, and we get a new valid solution S'_1 (with at most as many elements in b_i than S'), not better than S_1 (otherwise i would have been selected instead of j in the greedy algorithm) thus strictly worse than S'. Then, we remove one element from b_i, both in S' and S'_1. According to the second remark above, we get two valid solutions S'' and S''_1 for α^r such that the cost of S'' is strictly less than the cost of S''_1. But S''_1 is exactly S_r. This contradicts the fact that S_r is optimal by induction hypothesis.

- Now, suppose that S' has a bin b_i in $\mathcal{B}_m$ with strictly more elements than the corresponding bin in S_r, thus a bin with strictly more elements than any bin in S_r (by definition of $\mathcal{B}_m$). Consider b_t and b_s two bins with maximal size in S' (there are at least two such bins according to Lemma 2). In S', they have more elements than b_i, thus strictly more elements than any bin in S_r, in particular the corresponding bins in S_r. As we did for S_1, we move in S_2 one element from b_k to b_t, and one element from b_l to b_s, and we get a new valid solution S'_2 (with at most as many elements in b_s and b_r than S'), not better than S_2 (otherwise s and t would have been selected instead of k and l in the greedy algorithm), thus strictly worse than S'. Then, we remove one element each from b_t and b_s, both in S' and in S'_2, and we get two valid solutions S'' and S''_2 for $p = \alpha^r$ such that the cost of S'' is strictly less than the cost of S''_2. But S''_2 is exactly S_r, again a contradiction.

In other words, no bin in S' can have strictly more elements than the corresponding bin in S_r, thus the total cost of S' is at most the total cost of S_r. If we remove any element from S', we get a valid solution for α^r (according to the first remark above) with a cost strictly smaller than the cost of S', thus better than S_r, which again is not possible. Therefore, the best solution for α^{r+1} can indeed be built from an optimal solution for α^r by selecting the best solution obtained by adding one element in a bin which is not of maximal size, or two elements in two bins of maximal size. ■

Here we consider the complexity of GreedyPartitioning. Since $m \leq r$ by Lemma 3, the while loop executes at most $2r$ trips. With λ equal to the maximal λ_i, the cost for each bin is always less than $\lambda\alpha^r = \lambda p$. The dimensions with the smallest costs can be maintained in two sets: one for bins of maximum size, and one for other bins. Selecting the bins with minimal cost in each set can be done in $O(d \log(\lambda p))$, while updating the costs is done by a multiplication by α, thus in $O(\log \alpha \log(\lambda p))$. This yields an overall complexity of the greedy algorithm of $O(\log(\lambda p)(rd + r \log \alpha)) = O((d+1) \log(\lambda p) \log p)$ since $p = r \log \alpha$, thus $O(d(\log p)^2)$ if λ is not considered.

We do not know if an extension of this greedy approach can lead to a polynomial algorithm for an optimal solution in the general case where $p \neq \alpha^r$.

5.2 Exhaustive Enumeration of Locally Optimal Partitionings

In this section, we present an algorithm that finds an optimal solution by generating all possible partitionings (γ_i) that satisfy the necessary conditions given by Lemma 2 (the locally optimal solutions), and determining which one yields the lowest cost. We also evaluate how many candidate partitions there are and present the complexity of our algorithm.

Note that instead of generating all possible distributions in bins given by Lemma 2, we could try to explore all possible values of (γ_i) directly and to check whether they are valid (in the sense

that the product of any $(d-1)$ values is a multiple of p). However, this would require generating all possible values from 1 to p for each bin and would consider roughly p^d possible solutions (without counting the cost of checking). We want to do better.

5.2.1 Complexity Issues: First Hints

Before describing our exhaustive enumeration algorithm, we first give some upper bounds on the number of solutions $S(p)$ that respect the conditions of Lemma 2, for an integer p. We are not interested in an exact formula, but in the order of magnitude, especially when the number of bins d is fixed (and small, equal to 3, 4, or 5), but when p can be large (up to 1000 for example), since this is the situation we expect to encounter in practice when computing multipartitionings.

Let us first try a rough estimation. For each factor, we need to put $(r+m)$ factor instances in the bins, where m is at most r, thus at most $2r$ instances. Each factor instance can be put in d different bins, thus there are at most d^{2r} possibilities for each factor, and $d^{(2\sum r_i)}$ for all factors, and $d^{2\log_2 p}$ in the worse case (when p is a power of 2), which is equivalent to $p^{2\log_2 d}$.

We can be a little bit more accurate: we can try to give an upper bound for a given m between the bounds given by Lemma 3. For each m, we select two bins – $\frac{d(d-1)}{2}$ choices – in which we put m elements each. It remains to distribute $(r-m)$ elements, which can be done in at most $(d-2)^{r-m}$ ways. When $d=3$, since m goes from $\lceil \frac{r}{2} \rceil$ to r, the number of possibilities for each factor is thus $3(\lfloor \frac{r}{2} \rfloor + 1) \leq 3r$, and the total number of solutions is less than $3^s \prod_{i=1}^{s} r_i$ when combining all factors. We will come back to such an expression later when discussing our final upper bound. When $d > 3$, the number of possibilities for each factor is less than:

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

When there is a single factor (such as 2) the number of solutions 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 to do even better.

In the previous approach, we evaluated the number of possible positions for each element to be put in the d bins. Another (dual) view is to count, for each bin, how many elements it can contain. This view leads to a sharper upper bound when d is small and r can be large. We can select for each bin a number between 0 and r, thus $(r+1)$ possibilities. The total number of solutions is thus less than $(r+1)^d$ for each factor, thus less than $(\log_2 p + 1)^d$ if p is a power of 2, and $S(p) \leq (\prod_i (r_i+1))^d$ in the general case. When r is larger than $d \geq 3$, $(r+1)^d$ is less than $(d+1)^r$, which is roughly our previous upper bound. Thus, with this scheme, we are more likely to obtain a better evaluation of the total number of solutions. We will indeed be able to show (Section 5.2.3) 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 is fixed and p can be large, and that this bound is tight in order of magnitude. It is no surprise that we get a worse-case with a similar expression as the number of dividers of an integer (see [15]) since $\prod_i (r_i+1)$ is indeed the number of dividers of p.

We now first detail an algorithm that generates exactly all locally optimal solutions.

5.2.2 Generating All Distributions in Bins

The C program of Figure 3 generates in linear time all possible distributions into d bins satisfying the $(r+m)$ optimality condition of Lemma 2 of a given factor appearing r times in the decomposition of p. It is inspired by a program [19] for generating all partitions of a number, which is a well-studied problem (see [3, 20]) since the mathematical work of Euler and Ramanujam. The procedure `Partitions` first selects the maximal number m in a bin, and uses the recursive procedure `P(n,m,c,t,d)` that generates all distributions of n elements in $(d-t+1)$ bins (from index t to index d), where each bin can have at most m elements and at least c bins should have m elements. Therefore the initial call is `P(r+m,m,2,1,d)`.

```
// Precondition: d >= 2
void Partitions(int r, int d) {
  int m;
  for (m = (r+d-2)/(d-1); m <= r; m++) {
    P(r+m,m,2,1,d);
  }
}

void P(int n, int m, int c, int t, int d) {
  int i;
  if (t==d) {
    bin[t] = n;
  } else {
    for (i=max(0,n-(d-t)*m); i<=min(m-1,n-c*m); i++) {
      bin[t] = i;
      P(n-i,m,c,t+1,d);
    }
    if (n>=m) {
      bin[t] = m;
      P(n-m,m,max(0,c-1),t+1,d);
    }
  }
}
```

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

Let us first consider the loop in function `Partitions`. Thanks to 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. Furthermore, for each such m, there is indeed at least one valid solution with $(r+m)$ elements and two maxima equal to m (if $d \geq 2$), for example the solution where the first two bins have m elements and the $(d-2)$ other bins contain a total of $(r-m)$ elements, one possibility being with the $(r-m)$ elements distributed so that $q = \lfloor \frac{r-m}{m} \rfloor$ 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 $(d-t+1)$ (the number of bins) that there is at least one valid solution if and only if $c \leq d-t+1$ and $cm \leq n \leq (d-t+1)m$ and that `P` generates all of them (and only once) if these conditions are satisfied. These conditions are simple to understand: we need at least cm elements (so that at least c bins have m elements)

and at most $(d-t+1)m$ 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 solution, we should have $c \leq 1$ and $n = m$ if $c = 1$, i.e., $c \leq d-t+1$ and $cm \leq n \leq (d-t+1)m$.

The general case is more tricky. We first select the number of elements i in the bin number t and recursively calls `P` for the remaining bins. If we select strictly less than m elements (this selection is thus in the loop), we will still have to select c bins with m elements for the remaining $(d-t)$ bins, with $(n-i)$ elements. Therefore, the number i that we select should not be too small, nor too large, and we should have $cm \leq n-i \leq m(d-t)$ (by induction hypothesis), i.e., $n-(d-t)m \leq i \leq n-cm$. 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 t, we also have $c \leq d-t+1$ and if $c = d-t+1$, then the loop has no iteration, thus for any i selected in the loop, we have $c \leq d-t$. Therefore the recursive call `P(n-i,m,c,t+1,d)` has correct parameters. Finally, if we select m elements for the bin t (this selection is thus after the loop), this is possible only if m is less than n of course, and then it remains to put $(n-m)$ elements into $(d-t)$ 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 $(d-t)$ and removed m elements.

5.2.3 Complexity of the Exhaustive Enumeration

For generating an optimal solution to our minimization problem, we first decompose p into prime factors (complexity $O(\sqrt{p})$ by a standard algorithm, this will be the dominant complexity), we then generate all locally optimal solutions (those that satisfy Lemma 2) for each factor (with the function `Partitions`), and we combine them while keeping track of the best overall solution. For evaluating each solution, 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 $\sum_i r_i \leq \log_2 p$ 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 the complexity of the function `Partitions` (which is the number of solutions generated by the algorithm since each solution is generated only once) times $(\log_2 p)^3$. Therefore, it remains to evaluate the number of solutions $S(p)$ generated by the function `Partitions`, i.e., the number of locally optimal solutions.

Consider first the case of a number p, product of simple 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 (picking two bins where to put one element), so the total number of solutions is $\left(\frac{d(d-1)}{2}\right)^s$. Now, the i-th prime number is equivalent to $i \log i$ (see for example [15]). 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 solutions 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 (when p is the product of single prime factors) is actually representative of the worse case (in order of magnitude), as claimed in Section 5.2.1.

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 [15, 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 locally optimal solutions $S(p)$ satisfies $S(p) \leq d(p)^d$. We can therefore 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), we get that $\log(d(d-1)/2) \leq \limsup\{\log S(p)\log\log p/\log p\} \leq d\log 2$.
- The average order of $S(p)$ is less than $(\log p)^{2^d-1}$.

In other words, the worse-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, in average, the behaviour 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, to be more complete in this research report, it remains to compute, as promised, the exact order of magnitude of the worse-case, i.e., to show that $\limsup\{\log S(p)\log\log p/\log p\} = \log(d(d-1)/2)$. Actually, it only remains to find an upper bound better than the one obtained thanks to $d(p)$.

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

Proof: Let $C(r)$ be the number of solutions generated by a factor appearing r times in the decomposition of p. The total number of solutions $S(p)$ is $\prod_{i=1}^{s} C(r_i)$. To find an upper bound for $S(p)$, we use a similar proof technique as in [15, page 261-262], with consists in 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, with a rough approximation, $C(r)$ 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 $\delta > 0$:

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

This is true whatever r and α.

When α is large, we can bound the number of solutions in a more accurate way. Remember that when $r = 1$, the number of generated solutions is $d(d-1)/2$. When $r \geq 2$ and $d \geq 4$, we saw (Section 5.2.1) that $C(r) \leq \frac{d(d-1)(d-2)^r}{2(d-3)}$, and for $d = 3$, the number of solutions 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:

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

When p grows, 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. ■

6 Finding the Mapping

In Section 5, 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 $(\gamma_i)_{1\leq i\leq d}$ 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 slice of the array and a multipartitioning "neighbor mapping" function exists for each processor. This assumption has not yet been proven valid. We point out that an assignment with the first property is a generalization of the notion of **latin square**, mentioned in the second reference book on latin squares by Dénes and Keedwell [11, page 392] as an **F-hyper-rectangle**. However, despite this reference, we did not find so far any paper that gives a construction mechanism for such an assignment, or even an existence proof, in our general case. Furthermore, even if such a proof exists, which we would not be aware of, the constructive proof we give below is of interest for us because:

- it has the neighbor property mentioned below,
- 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).

[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.

Therefore, we will give here the complete proof, with no further reference to latin squares and F-hyper-rectangles.

The only property guaranteed so far is that a (γ_i) partitioning selected is a **valid solution** for the partitioning problem; namely, 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 valid solution (γ_i), 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 6.1). In Section 6.2, we review some results on the validity of modular mappings that were shown previously by Darte, Dion, and Robert [10], along with some additional properties that are more important to our problem at hand. Finally, in Section 6.3, we give a constructive proof for 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) does not account for network topology when constructing tile-to-processor mappings, so all valid mappings are considered equally good.

6.1 Modular Mappings

Consider the assignment in Figure 1. Can we give 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 slice (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). What are the possible values for c? Obviously c is not 0 otherwise 0 appears twice in the slice $y = 0$ for example. Consider the first case $a = 1$ and $b = 2$. If $c = 1$, then 1 appears twice in the slice $y = 0$ (for $(1, 0, 0)$ and for $(0, 0, 1)$). If $c = 2$, then 2 appears twice in the slice $x = 0$ (for $(0, 1, 0)$ and for $(0, 0, 1)$. And if $c = 3$, then 0 appears twice in the slice $y = 0$ (for $(0, 0, 0)$ and for $(1, 0, 1)$). Now, in the second case, if $a = 2$ and $b = 3$, what are the possible values for c? A similar study shows that $c = 1$ is not possible (conflict for the slice $x = 0$), $c = 2$ is not possible (conflict for the slice $y = 0$), and $c = 3$ is not possible (conflict for the slice $x = 0$).

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** $M_m : \mathbb{Z}^d \longrightarrow \mathbb{Z}^{d'}$ defined by $M_m(\vec{i}) = (M\vec{i}) \bmod \vec{m}$ (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

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

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 $\vec{m}$. (Don't take this representation as a physical view of the processors. It is just a way of numbering processors.) 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 the next section, 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 study.

Consider again Figure 1. 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 (r, q) where r and q are the rest and the quotient 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 and of modular mappings that satisfy the load-balancing property.

Definition 1 (Modular mapping)
A modular mapping $M_m : \mathbb{Z}^d \longrightarrow \mathbb{Z}^{d'}$ *is defined as* $M_m(\vec{i}) = (M\vec{i}) \bmod \vec{m}$ *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* $\mathcal{I}_b = \{\vec{i} \in \mathbb{Z}^n \mid 0 \leq \vec{i} < \vec{b}\}$ *(component-wise) where* n *is the dimension of* $\vec{b}$.

Definition 3 (Slice)
Given a rectangular index set $\mathcal{I}_b$, *a slice* $\mathcal{I}_b(i, k_i)$ *of* $\mathcal{I}_b$ *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 an hyper-rectangle (or any more general set) $\mathcal{I}_b$, *a modular mapping* M_m *is a one-to-one mapping from* $\mathcal{I}_b$ *onto* $\mathcal{I}_m$ *if and only if for each* $\vec{j} \in \mathcal{I}_m$ *there is one and only one* $\vec{i} \in \mathcal{I}_b$ *such that* $M_m(\vec{i}) = \vec{j}$.

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

Definition 6 (Load-balancing property)
Given a rectangular index set $\mathcal{I}_b$, *a modular mapping* M_m *has the load-balancing property for* $\mathcal{I}_b$ *if and only if for any slice* $\mathcal{I}_b(i, k_i)$, *the restriction of* M_m *to* $\mathcal{I}_b(i, k_i)$ *is a many-to-one mapping onto* $\mathcal{I}_m$.

Note that the images of the slice $\mathcal{I}_b(i, k_i)$ are obtained from the images of the slice $\mathcal{I}_b(i, 0)$ by adding (modulo $\vec{m}$) k_i times the i-th column of M. Therefore, all values in I_m are images of the same number of elements in the slice $\mathcal{I}_b(i, k_i)$ if and only if the same is true for the slice $\mathcal{I}_b(i, 0)$. In other words, for modular mappings (as opposed to arbitrary mappings), the load-balancing

property can be checked only for the slices that contain 0 (the slices $\mathcal{I}_b(i,0)$). 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 $\mathcal{I}_b(i,0)$ under M_m are the images of $\mathcal{I}_{b[i]}$ under the modular mapping $M[i]_m$. We therefore have the following property.

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

We now address the following questions. How can we check that a given modular mapping has the load-balancing property? This is the topic of Section 6.2. Given an array of tiles $\mathcal{I}_b$ defined by a valid solution (b_i), how can we find a modular mapping that has the load-balancing property? This is the topic of Section 6.3. We will have to define an adequate matrix M and also to choose an adequate shape $\vec{m}$ for the virtual grid of processors.

6.2 Validity of Modular Mappings

Let us first recall some of the theory of one-to-one modular mappings developed by Lee and Fortes [17] and Darte, Dion, and Robert [10]. We will then extend some of these results to many-to-one modular mappings. We will make use of Hermite and Smith forms. We only 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 lower 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_m is a linear mapping from the group $\mathbb{Z}^d$ into the quotient group $\mathbb{Z}^{d'}/\vec{m}\mathbb{Z}$.

6.2.1 The Lattice $G = \mathbf{Ker}(M_m)$

We denote by G the set $\mathrm{Ker}(M_m) = \{\vec{i} \mid M\vec{i} = \vec{0} \bmod \vec{m}\}$, 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 $\overline{M}_m$ from $\mathbb{Z}^d/G$ into $\mathbb{Z}^{d'}/\vec{m}\mathbb{Z}$ such that $\overline{M}_m \circ \pi = M_m$, and $\overline{M}_m$ is one-to-one (this is the classical factorization of a linear map between groups). The image of $\mathbb{Z}^d/G$ under $\overline{M}_m$ 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 $\overline{M}_m$ 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 $\overline{M}_m$ is a one-to-one linear map from the equivalence classes of $\equiv$ onto the image of $\mathbb{Z}^d$ under M_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_m to S is also a one-to-one mapping, but we loose the linear properties of M_m or $\overline{M}_m$ since S has no particular algebraic structure. This will be typically our case when considering the rectangular index set $\mathcal{I}_b$. 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 $\mathcal{I}_b$ 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 $\mathcal{I}_b$ and $\mathbb{Z}^d/G$ have the same number of elements. It remains to check that all elements of $\mathcal{I}_b$ have different images under π, i.e., belong to different equivalence classes. Let $\vec{i}$ and $\vec{i}'$ in $\mathcal{I}_b$ 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 $\vec{k}$. It is easy to check that such a triangular system has only one integral solution, $\vec{j} = \vec{0}$. ■

The previous lemma gives an automatic way to compute a rectangular index for which the map M_m is one-to-one. But how can we compute a basis for the lattice G? This can be easily done as follows (as described in [10]). Consider $\vec{i} \in G$: $M\vec{i} = \vec{0} \bmod \vec{m} \Leftrightarrow \exists \vec{k} \in \mathbb{Z}^{d'}$ such that $M\vec{i} = \theta_m\vec{k}$ where θ_m is the diagonal matrix $\mathrm{diag}(\vec{m})$. Let $\overline{\theta}_m$ be the comatrix of θ_m, i.e., the matrix such that $\overline{\theta}_m\theta_m = \det(\theta_m)I_{d'}$. We get:

$$M\vec{i} = \theta_m\vec{k} \Leftrightarrow (\overline{\theta}_m M)\vec{i} = \det(\theta_m)\vec{k} \Leftrightarrow Q_1 S Q_2 \vec{i} = \det(\theta_m)\vec{k} \Leftrightarrow S Q_2 \vec{i} = \det(\theta_m) Q_1^{-1}\vec{k}$$

where Q_1SQ_2 is the Smith form of $\theta_m M$. Now, there exists $\vec{k} \in \mathbb{Z}^{d'}$ such that $M\vec{i} = \theta_m\vec{k}$ if and only if there exists $\vec{k}' \in \mathbb{Z}^{d'}$ such that $S\vec{j} = \det(\theta_m)\vec{k}'$ and $\vec{i} = Q_2^{-1}\vec{j}$. It remains to solve the diagonal system $S\vec{j} = \det(\theta_m)\vec{k}'$. With $p = \det(\theta_m) = \prod_{i=1}^{d'} m_i$, there exists an integer k_i' such that $s_i j_i = p k_i'$ 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 $\gcd(0,p) = p$). In other words, 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, $M\vec{i} = \theta_m \vec{k} \Leftrightarrow \vec{i} = M^{-1}\theta_m \vec{k}$, thus $M^{-1}\theta_m$ 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_m has basis B. This can be done as follows. Write the Smith normal form of B, $B = Q_1 S Q_2$. 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 $Q_1 S$, which is also a basis of the lattice described by B since B and $Q_1 S$ differ by multiplication on the right by a unimodular matrix.

6.2.2 One-To-One Modular Mappings

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

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

Lemma 6 gives a sufficient condition for a mapping M_m to be one-to-one from a rectangular index set $\mathcal{I}_b$ onto the rectangular index set $\mathcal{I}_m$. 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 $\mathcal{I}_m$ (equivalently if $\mathcal{I}_b$ and $\mathcal{I}_m$ have the same cardinality), then the mapping is a one-to-one mapping from $\mathcal{I}_b$ onto $\mathcal{I}_m$. It turns out that the converse is also true (see details in [10]) thanks to a famous (in covering/packing theory) theorem due to Hajós [14].

Theorem 3 *A modular mapping M_m is one-to-one from $\mathcal{I}_b$ onto $\mathcal{I}_m$ 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 $\mathcal{I}_b$, and $\mathcal{I}_b$ and $\mathcal{I}_m$ have same cardinality.*

To find the diagonal components of these 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 $\mathcal{I}_b$ onto $\mathcal{I}_m$ only for $\vec{b} = (2,6)$ and $\vec{b} = (4,3)$. Can we build a mapping with such a property? Yes. 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-like array design (see [9] and [8]) for generating "juggling schedules" corresponding to "locally sequential globally parallel" partitionings.

6.2.3 Many-To-One Modular Mappings

We now generalize the previous results (when possible) to the case of many-to-one modular mappings. The situation turns out to be much more difficult. We will say that a rectangular index set $\mathcal{I}_b$ is a multiple of a rectangular index set $\mathcal{I}_{b'}$ 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, $\mathcal{I}_b$ can be paved (partitioned) by disjoint copies of $\mathcal{I}_{b'}$.

The results of the previous section give immediately a necessary condition and a sufficient condition for a modular mapping M_m to be many-to-one from $\mathcal{I}_b$ onto $\mathcal{I}_m$.

Lemma 7 *If M_m is a many-to-one modular mapping from $\mathcal{I}_b$ onto $\mathcal{I}_m$, then the cardinality of $\mathcal{I}_b$ is a multiple of the cardinality of $\mathcal{I}_m$, and the determinant of the lattice G is equal to the cardinality of $\mathcal{I}_m$.*

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

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 $\mathcal{I}_m$, then some element in $\mathcal{I}_m$ 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 $\mathcal{I}_m$ have the same number of pre-images. ■

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

Proof: This is clear. Each element in $\mathcal{I}_{b'}$ gives rise to a different value in $\mathcal{I}_m$. Furthermore, this property is true for any translated copy of $\mathcal{I}_{b'}$ since a translation of the index set $\mathcal{I}_{b'}$ corresponds to a translation (modulo $\vec{m}$) of their images. We already made this remark in Lemma 4. Therefore, if

M_m is one-to-one from $\mathcal{I}_{b'}$ onto $\mathcal{I}_m$, M_m is also one-to-one from any translated copy of $\mathcal{I}_{b'}$ onto $\mathcal{I}_m$. Since $\mathcal{I}_b$ is the union of disjoint copies of $\mathcal{I}_b$, M_m is a many-to-one mapping from $\mathcal{I}_b$ onto M_m, since each element of M_m has as many pre-images as there are copies of $\mathcal{I}_{b'}$ in the partitioning of $\mathcal{I}_b$. ■

Can we generalize Theorem 3 to the case of many-to-one modular mappings? The theorem of Hajós used to prove Theorem 3 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 (here 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 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) [14] or the book of Fuchs [12].

We can easily use Hajós' extended result to adapt the proof of Theorem 3, up to dimension 3: a modular mapping is many-to-one from $\mathcal{I}_b$ onto $\mathcal{I}_m$ if and only if the lattice G has a determinant equal to the cardinality of $\mathcal{I}_m$, 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 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 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 many-to-one for each "face" of the index set) makes that we need 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 3 to many-to-one modular mappings has also 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 $\mathcal{I}_b$ for which the 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 $\mathcal{I}_b$ can be partitioned using several smaller rectangular index sets, for each of which the one-to-one property is true, but that $\mathcal{I}_b$ 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 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 3 and from Lemma 8 will be sufficient.

6.2.4 Modular Mappings With the Load-Balancing Property

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

Theorem 4 *If M_m is a modular mapping with the load-balancing property for $\mathcal{I}_b$, then the following holds:*

- *for each dimension j, $\prod_{i \neq j} b_i$ is a multiple of the cardinality of $\mathcal{I}_m$,*
- *the determinant of the lattice G is equal to the cardinality of $\mathcal{I}_m$,*
- *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 (γ_i) in Section 5 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_m divides the cardinality of $\mathcal{I}_m$. If the determinant is not equal to $\mathcal{I}_m$, it is strictly smaller, and some values in $\mathcal{I}_m$ have no pre-image at all in the whole set $\mathbb{Z}^n$, therefore no pre-image in $\mathcal{I}_m$. In this case, the mapping cannot have the load-balancing property (in terms of Section 5, some processors have no tiles at all to compute).

The third property comes from Lemma 4 and the link between the lattice G for M_m and the lattice $G[i]$ for $M[i]_m$ ($M[i]$ is the matrix obtained by removing the i-th column of M, $M[i]_m$ is the corresponding mapping modulo $\vec{m}$). The mapping M_m has the load-balancing property if and only if, for each dimension i, $M[i]_m$ is many-to-one when restricted to the i-th "face" of $\mathcal{I}_b$, i.e., $\mathcal{I}_b(i,0)$. Following Lemma 7, the determinant of the lattice $G[i]$ has to be equal to the cardinality of $\mathcal{I}_m$. The lattice $G[i]$ is the set of vectors in $\mathbb{Z}^{d-1}$ whose image under $M[i]_m$ 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:

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

In other words, instead of computing $G[i]$ following the steps given in Section 6.2.1 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 $\mathcal{I}_m$, 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 4 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 3 and of Lemma 8, and of the link between $G[i]$ and G revealed in the previous lemma.

Theorem 5 *A modular mapping M_m has the load-balancing property for $\mathcal{I}_b$ if the following conditions are satisfied:*

- *the determinant of the lattice G for M_m is equal to the cardinality of $\mathcal{I}_m$,*

- *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. Furthermore, 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 4. This matrix H' satisfies the condition of Theorem 3: its determinant is equal to the determinant of B (since the first component of H is 1), which is equal to the cardinality of $\mathcal{I}_m$, therefore $M[i]_m$ is a one-to-one mapping from $\mathcal{I}_h$ onto $\mathcal{I}_m$, where $\vec{h}$ is the diagonal of H'. Now, thanks to Lemma 8, we know that $M[i]_m$ is a many-to-one modular mapping from the slice $\mathcal{I}_b(i,0)$ onto $\mathcal{I}_m$ since $\mathcal{I}_{b[i]}$ is a multiple of $\mathcal{I}_h$. Finally, since this is true for any dimension i, Lemma 4 shows that M_m is a modular mapping with the load-balancing property for the index set $\mathcal{I}_b$. ■

Note that in many practical cases, the previous conditions are also necessary. For example, when for a dimension i, the cardinality of $\mathcal{I}_b(i,0)$ is equal to $\mathcal{I}_m$ (and not simply a multiple), the conditions are necessary (Theorem 3) since many-to-one is in this case one-to-one. Also, when $d \leq 4$, then each modular mapping $M[i]_m$ 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 5 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.

6.3 Generating A Valid Modular Mapping

We are now ready, thanks to the previous observations, to build a modular mapping M_m with the load-balancing property for an index set $\mathcal{I}_b$ (which is given, $\vec{b}$ is the vector whose components are the γ_i's of Section 5). The freedom we have is that we can choose the matrix M and the modulo vector $\vec{m}$, but with the constraint that the cardinality of $\mathcal{I}_m$ (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 valid solution (with the meaning of Section 5), 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 6.2.1). A basis of G is simply given by $M^{-1}\theta_m$, where $\theta_m = \mathrm{diag}(m_1, \ldots, m_d)$. Note also that, when M is unimodular, the first condition of Theorem 5 is automatically fulfilled since the determinant of G is the determinant of $M^{-1}\theta_m$, thus the determinant of θ_m, i.e., the cardinality of $\mathcal{I}_m$.

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 $N_{m'}$ – in dimension $(d-1)$ – defined by N and $\vec{m}'$ has the load-balancing property for $I_{b'}$, 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_m defined by M and $\vec{m}$ has the load-balancing property for $\mathcal{I}_b$ if it is many-to-one from the last slice $\mathcal{I}_b(d,0)$ onto $\mathcal{I}_m$.*

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

Without loss of generality, let us consider the first dimension, i.e., the first slice $\mathcal{I}_b(1,0)$. Given $\vec{j} \in \mathbb{Z}^d/\vec{m}\mathbb{Z}$, let us count the number of vectors $\vec{i} \in \mathcal{I}_b$, 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 $N_{m'}$, there are exactly n vectors $\vec{i}' \in \mathcal{I}_{b'}$ 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 $0 \leq k < b_d/m_d$. Therefore, $\vec{j}$ has nb_d/m_d pre-images in $\mathcal{I}_b$ and this number does not depend on $\vec{j}$. ■

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

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

(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 valid solution for p, then $(b_1, \ldots, b_{d-1})$ is a valid solution for $\frac{p}{\gcd(p,b_d)}$, (6) If $\vec{b}$ is a valid solution for p, then for all k, $2 \leq k \leq d$, $\prod_{i=1}^{k-1} b_i$ 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 cancel 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 valid solution 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 valid solution 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 valid solution 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 $M[i]_m$). Indeed, the two following constructions are possible:

- Compute a basis B for G with $B = M^{-1}\theta_m$, where $\theta_m = \mathrm{diag}(\vec{m})$. 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 6.2.1).

- Consider $M[i]_m$ directly, i.e., remove the i-th column of M. $M[i]_m$ 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_m is many-to-one from the slice $\mathcal{I}_b(d, 0)$ onto $\mathcal{I}_m$. 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 $\tilde{\theta}_m$ the matrix $\mathrm{diag}(m_2, \ldots, m_d)$. The matrix $\tilde{M}^{-1}\tilde{\theta}_m$ 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 $F_t\tilde{\theta}_m$ since the left Hermite form of $\tilde{M}^{-1}\tilde{\theta}_m$ is obtained by multiplying on 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 $\mathrm{diag}(s)$ 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* $1 \le i < n$,
- $h_{i+1} = \frac{r_{i+1}s_i}{r_i}$ *for* $1 \le i < n$ *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 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 illustrates this mechanism:

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

Furthermore, the determinant of the 2×2 submatrix defined by the columns i and $(i+1)$ and the rows 1 and i (a triangular matrix) does not change. It was equal to $s_i r_{i+1}$, 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 $F_t\tilde{\theta}_m$ 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_m is many-to-one from the slice $\mathcal{I}_b(d,0)$ onto $\mathcal{I}_m$, it is now sufficient to define $\vec{t}$ so that h_i divides b_i for all $i < d$ (Lemma 8 and Theorem 3) since the fact that $\mathcal{I}_h$ and $\mathcal{I}_m$ 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 $\gcd(m_{i+1} b_{i+1}, b_{i+1} r_{i+1})$.

It remains 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 valid solution 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 valid solution 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 just proved the following result.

Lemma 13 *With the vector $\vec{t}$ given by Equation 4, the mapping M_m is many-to-one from the slice $\mathcal{I}_b(d,0)$ onto $\mathcal{I}_m$, when $\vec{b}$ is a valid solution for p.*

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

Lemma 14 *Let $\vec{b}$ be a valid solution for p. If N is built recursively in dimension $(d-1)$ the same way M is built in dimension d, but for a number of processors equal to $\frac{p}{\gcd(p,b_d)}$ and the vector $\vec{b}'$ defined by the first $(d-1)$ components of $\vec{b}$, then M_m is a modular mapping with the load-balancing property for $\mathcal{I}_b$.*

Proof: We make an induction 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 valid solution 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 $\mathcal{I}_{b'}$. Now, thanks to Lemma 9, and the fact that m_d divides b_d, we just have to show that M_m is many-to-one from the slice $\mathcal{I}_b(d,0)$ onto $\mathcal{I}_m$. This is the result of Lemma 13. ■

The schema we just presented corresponds to the following program (where the matrix M has rows and columns from 1 to d as in the presentation of this paper):

```
// Precondition: d >= 2
void ModularMapping(int d) {

  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] -= t*M[j][k];
      }
      r = gcd(t*m[j],r);
    }
  }
}
```

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 $\vec{m}$). 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 [10] (injectivity of $M_{\lambda m}$ for $\mathcal{I}_{\lambda b}$) 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 valid solution $\vec{b} = (10, 15, 6)$ in dimension 3, the basic kernel leads to $\vec{m} = (1, 5, 6)$ and the mapping $(i, j, k) \rightarrow (i + j \bmod 5, k - i - 2j \bmod 6)$, which corresponds for example to the "linearization"

$(i, j, k) \to 6(i + j \bmod 5) + (k - i - 2j) \bmod 6$. The following 6 tables give the corresponding values for (i, j), when k goes from 0 to 5.

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

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

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

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

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

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

The brave reader can check that this is indeed a correct 3D multipartitioning.

7 Experiments

We have implemented preliminary 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 partitioning [6, 7]. The partitioned dimensions of the template are distributed onto a virtual array of processors that has the correct size for the rank of the multipartitioning. Internally, the compiler analyzes communication and loop bounds reduction as if the multipartitioned template was a standard `BLOCK` partitioned template onto a larger array of processors. The main difference comes in the interpretation that the compiler gives to the `PROCESSORS` directive. For a `BLOCK` partitioned template, the number of processors onto which each dimension is partitioned determines the data sizes of the tiles. The number of processors may be different for each dimension (i.e. `processors p(2, 3); distribute t(block, block) onto p`). In the case of multipartitionings, the number of processors cannot be specified on a per dimension basis since each multi-partitioned dimension is completely distributed among all processors. The tiles are partitioned according to the rank of the multipartitioning and then assigned in a skewed-cyclic fashion to the processors as presented in the previous sections.

There are several important issues for correctly generating 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. If the tiles are not enumerated in the order indicated by the loop-carried dependences, then it is possible to execute the loop correctly, but in a serialized manner induced by data exchange-related synchronization. Second, communication, which has effectively been vectorized out of a loop nest, should not be performed on a tile-by-tile basis, but instead should be executed once for all of a processor's tiles. Communication aggregation is thus more tricky but is possible because generalized multipartitioning provide the same neighborhood guarantee as simpler, diagonal multipartitionings: the neighboring tiles for a particular processor will be the same for all of its owned tiles.

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 [7, 6, 1, 2]. Here we present some preliminary results applying generalized multipartitioning in a compiler-based parallelization of the NAS SP application benchmark [4, 7], a computational fluid dynamics code.

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.

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 shows the speedups obtained for both the dHPF-generated and hand-coded versions of the NAS SP benchmark using the class 'B' problem size (102^3). The hand-coded version implements three-dimensional diagonal multipartitionings, thus its results are only available for numbers of processors which are perfect squares. The compiler-generated version uses generalized multipartitioning to execute on other numbers of processors. The table presents the speedups for the hand-coded version (where available), the dHPF version and the differences between them. All

# CPUs	hand-coded	dHPF	% diff.
1	0.80	0.87	-8.30
2		1.30	
4	2.86	2.60	10.16
6		4.14	
8		6.35	
9	7.74	6.98	10.84
12		9.72	
16	13.00	13.97	-6.87
18		15.84	
20		16.44	
25	22.15	21.32	3.87
32		27.84	
36	36.51	32.38	12.79
49	51.78	41.32	25.32
50		38.88	
64	74.95	51.43	13.44

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

speedups presented are relative to the sequential version of NAS SP. Overall, the performance of the compiler-generated code is similar to that of the hand-coded versions with the exception of the gap between the versions for a 49 processor execution, which is wider for reasons that are currently unknown.

The performance differences observed between the hand-coded and compiler-generated versions are due in large part to a difference how off-processor values are stored and accessed in the two versions. In the dHPF-generated code, each data tile is extended with overlap areas (ghost regions around the tile's boundary) into which off-processor data is unpacked. Overlap areas enable a loop operating on the tile to reference all data uniformly without having to distinguish between local and off-processor data. The hand-coded version uses a clever buffering scheme in which iterations of a loop that need off-processor data are peeled off the main body of the loop. Then, in the peeled loop references to off-processor data read their values directly out of a message buffer without having to unpack it. In the dHPF-generated code, the use of extra data space for overlap areas degrades data cache efficiency, which appears to account for most of the observed performance differences.

One other factor that effects the execution efficiency of the dHPF-generated code 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 fails to effectively exploit reuse of data tiles across multiple loop nests. Currently, 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 could 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 this would improve cache utilization by facilitating reuse of tile data among multiple loop nests.

8 Conclusions

The paper describes an algorithm for computing multipartitioned data distributions. These distributions are important because they support fully parallel execution of line-sweep computations. For arrays of 2 or more dimensions, our algorithm will compute an optimal multipartitioning that minimizes cost according to an objective function that measures communication in line sweep computations. Previously, optimal multipartitionings could be computed for d dimensional data only when $p^{\frac{1}{d-1}}$ is integral.

We have shown that, having a partitioning in which the number of tiles in each slice is a multiple of the number of processors — an obvious necessary condition — is also a sufficient condition for a balanced mapping of tiles to processors. We give a constructive method for building the mapping (which assigns the tiles to the physical processors that should compute upon them) using new techniques based on modular mappings.

We have constructed a prototype code generator that exploits generalized multipartitionings in the Rice dHPF compiler; however, these partitionings could be exploited by hand-coded implementations as well. Preliminary performance results for generalized multipartitioning code generated by dHPF show encouraging scalability for small numbers of processors.

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] 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.

[7] 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.

[8] 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.

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

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

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

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

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.