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

May 1, 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.

Previously, computing a d-dimensional multipartitioning required that $p^{\frac{1}{d-1}}$ be integral, where p is the number of processors. Here, we describe an algorithm to compute a d-dimensional multipartitioning of an array of ρ dimensions for an arbitrary number of processors, for any d, $2 \leq d \leq \rho$. 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 ρ-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.

We describe an implementation of multipartitioning in the Rice dHPF compiler and performance results obtained to parallelize a line sweep computation on a range of different numbers of processors.

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

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 [4, 13, 15] — and is also at the heart of a variety of other numerical methods and solution techniques [15]. Parallelizing computations based on line sweeps is important because these computations address important classes of problems and they are computationally intensive.

The recurrences that line sweeps are used to solve serialize each computational line in a sweep along a dimension. If a dimension is partitioned, this induces a serialization between computations on different processors. Using standard block uni-partitionings, 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 and performing line sweeps in all unpartitioned data dimensions locally, then transposing the data to localize the data along the previously partitioned dimension and 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 [4, 13, 15]. 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 detail in Section 2. A study by van der Wijngaart [18] 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.

The most general class of multipartitionings described in the literature is known as *diagonal multipartitionings*. To compute a d-dimensional diagonal multipartitioning, $p^{\frac{1}{d-1}}$ must be integral, where p is the number of processors. Thus, for three dimensions, diagonal multipartitionings are restricted to cases in which the number of processors is a perfect square. In this paper, we describe an algorithm to compute a d-dimensional multipartitioning of an array of ρ dimensions for an arbitrary number of processors, for any d, $2 \leq d \leq \rho$. 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 cost. To compute the best multipartitioning of a ρ-dimensional array, we describe a cost model for selecting d, the dimensionality of the best partitioning, and the number of partitions γ_i, $1 \leq i \leq d$ along each dimension. Our new algorithm for computing multipartitionings enables them to be used for parallelizing line sweep computations effectively in a much broader set of circumstances. In practice, our algorithm will choose a three-dimensional multipartitioning for three-dimensional data, except when the number of processors p is a prime.[1]

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 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 an implementation of multipartitioning in the Rice dHPF compiler for High Performance Fortran and ongoing work to incorporate generalized multipartitionings into the dHPF compiler.

2 Background

Johnsson *et al.* [13] 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 $0 \leq i,j < p$. Using this mapping for an ADI computation requires each processor to exchange data with only its two neighbors in a linear ordering of the processors, which maps nicely to a ring.

Bruno and Cappello [4] 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 $0 \leq r < 2^s$, 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 $0 \leq i,j,k < 2^d$ 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.

[1] When computing a multipartitioning for three-dimensional data for a prime number of processors, a two-dimensional partitioning will be selected because it yields a superior computation to communication ratio.

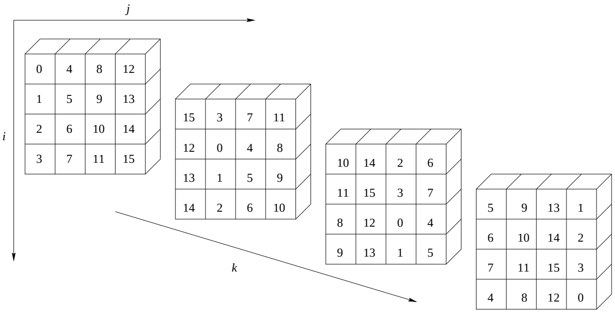

Figure 1: 3D Multipartitioning on 16 processors.

Naik *et al.* [15] describe *diagonal multipartitionings* for two and three dimensional problems, that generalizes Johnsson *et al*'s result [13]. In general, this class of multipartitionings involves partitioning data into $p^{\frac{d}{d-1}}$ tiles, where p is the number of processors and d is the number of partitioned array dimensions. Each processor handles $p^{\frac{1}{d-1}}$ 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 $0 \leq i,j,k < \sqrt{p}$. Diagonal multipartitionings in d dimensions are possible only when $p^{\frac{1}{d-1}}$ is integral; however, they are still more broadly applicable than the Gray code based mapping described by Bruno and Cappello.

3 General Multipartitioning

Bruno and Cappello noted that multipartitionings need not be restricted to having only one tile per processor per hyperplane of a multipartitioning [4]. How general can multipartitioning mappings be? A sufficient condition to support load-balanced line-sweep computation is that 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 notations 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 $\prod_{j \neq i} \gamma_j$.

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 $\prod_{j \neq i} n_j$ 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. Define $\lambda_i = K_2 + K_3 \frac{n}{n_i}$, λ_i depends on the domain size, number of processors and machine's communication parameters. The total cost of the algorithm, sweeping in all dimensions, is thus

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

Remark: if all communications are performed with perfect parallelism, with no overhead, then the term with K_3 is actually divided by p. We assume here that, in general, the cost of one communication phase is an affine function of the volume of transmitted data.

Assuming that p, n, and the n_i's are given, what we can try to minimize is $\sum_{i=1}^{d} \gamma_i \lambda_i$.

There are several cases to consider. If the number of phases 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 (the number of factors) and the r_i's (see the decomposition of p into prime factors), but whose complexity in p grows slowly.

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 there is an algorithm polynomial in $\log p$ or even in $\log s$ and the $\log r_i$'s. We suspect that our problem is strongly NP-complete, even if the input is s and the r_i's, instead of p. 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 [10]). However, 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.

5.1 Basic Properties of Potentially Optimal Partitionings

We say that $(\gamma_i)_{1 \leq i \leq d}$ – or (γ_i) for short – is a **valid solution** if, for each $1 \leq i \leq d$, p divides $\prod_{j \neq i} \gamma_j$. 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 must be 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 must 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 can appear in the γ_i without increasing the objective function. The following lemma refines the result of Lemma 1 for a potentially optimal solution.

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 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, $0 \le j < s$, 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.

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, containing fewer than m_α occurrences. We now have $c-1 \ge r_\alpha + m_\alpha$ 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 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 $\lceil \frac{r_\alpha}{d-1} \rceil \le m_\alpha \le r_\alpha \le (d-1)m_\alpha$ 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, $r_\alpha + m_\alpha = 2 * m_\alpha + e$ where e is the total number of elements in $(d-2)$ bins, excluding two bins of maximal size m_α. Since $0 \le e \le (d-2)m_\alpha$, then $m_\alpha \le r_\alpha \le (d-1)m_\alpha$. Finally, any valid solution requires that p divides the product of all of the factor instances in each group of $d-1$ bins. Thus, there must be r_α instances of α in $d-1$ bins, and thus $m_\alpha \ge \lceil \frac{r_\alpha}{d-1} \rceil$. ■

5.2 Exhaustive Enumeration of Potentially Optimal Partitionings

We now give an algorithm that finds an optimal solution by generating all possible partitionings (γ_i) that satisfy the necessary optimality conditions given by Lemma 2, and determining which one yields the lowest cost partitioning. We also evaluate how many candidate partitions there are and present the complexity of our algorithm. For the complexity, we are not interested in the exact number of solutions that respect the conditions of Lemma 2, 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.

The C program of Figure 2 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 [16] for generating all partitions of a number, which is a well-studied problem (see [17]) 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 exactly c bins should have m elements. Therefore the initial call is `P(r+m,m,2,1,d)`.

We now prove the correctness of the program. The procedure `P` selects a number of elements for the bin number t and makes a recursive call with parameter $t+1$ for the selection in the next bin. It is thus clear that all generated solutions are different since each iteration of a loop selects a different number of elements for each bin. It remains to prove that all solutions generated by `P` are valid (the total number of elements should be $r+m$, each bin should have less than m elements, and there should be at least c bins with m elements), and that all solutions are generated. For that we prove that `P(n,m,c,t,d)` is always called with parameters for which there exists at least a valid solution, that all possible numbers of elements are selected and only those.

```
// 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 2: 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 $\lceil\frac{r}{d-1}\rceil$ 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 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 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 a 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 call `P` for the remaining bins. If we select strictly less than m elements (this selection is 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 $cm \leq n-i \leq m(d-t)$, 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 positive, 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 an 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 (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.

5.3 Complexity of the Exhaustive Enumeration

For generating all optimal solutions to our minimization problem, we first decompose p into prime factors (complexity $O(\sqrt{p})$ by a standard algorithm, but could be less), we then generate all potentially optimal solutions 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) times $(\log_2 p)^3$. Therefore, it remains to evaluate the number of solutions generated by the function `Partitions`.

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 copy of each element), so the total number of solutions is $\left(\frac{d(d-1)}{2}\right)^s$. Now, the i-th prime number is approximated by $i \log i$ (see for example the Prime Pages [5]). Therefore, when p grows, we have

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

since divergent series with equivalent nonnegative 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 $\left(\frac{d(d-1)}{2}\right)^{\frac{\log p}{\log \log p}(1+o(1))}$, thus at least of order $p^{\frac{f(d)(1+o(1))}{\log \log p}}$, for a small function $f(d)$ of d. We can prove that this situation (when p is the product of single prime factors) is actually representative of the worse case (in order of magnitude). The proof is too long to be provided here but is available in the extended version of this paper [10].

Theorem 1 *When p grows, the total number of generated solutions is less than $p^{\frac{f(d)(1+o(1))}{\log \log p}}$ where $f(d)$ is a small function of d.*

6 Finding the Mapping

In Section 5, we determined a particular way of cutting the array so as to optimize communications: after partitioning, we get an array (of tiles) whose size is (γ_i) for which the objective is minimized. But until now, we made the assumption that we will be indeed able to assign tiles to processors so that each slice of the array contains exactly the same number of tiles per processor (load-balancing property). This is not sure yet.

The only property we have until now is that the (γ_i) form is a **valid solution**: for each $1 \leq i \leq d$, p divides $\prod_{j \neq i} \gamma_j$, the defining property of a completely balanced multipartitioning. Our main result is that this condition is sufficient to guarantee a mapping of processors to tiles. Our proof is constructive. For any valid solution (γ_i), optimal or not, with or without the additional property of Lemma 2, we give an automatic way to assign a processor number to each tile so that the load-balancing property is satisfied. This assignment is done through the use of modular mappings, defined below. The proof of our construction is much too long to be given here. We refer the reader to the extended version of this paper [10] for details of the proof and interesting properties of modular mappings.

The solution we build is one particular assignment, out of a set of legal mappings. It is not unique, and more experiments might show that they are not all equivalent in terms of execution time, for example because of communication patterns. But, currently, with our objective function (Section 4), the network topology is not taken into account yet and 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? 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 remainder and the quotient of the Euclidian division by 16. The assignment in the figure corresponds to the assignment $(i - k \bmod 4, j - k \bmod 4)$, which is what we call a **multi-dimensional modular mapping**.

Definition 1 *A mapping $M_m : \mathbb{Z}^d \longrightarrow \mathbb{Z}^{d'}$ defined by $M_m(\vec{i}) = (M\vec{i}) \bmod \vec{m}$ where M is an integral $d \times d'$ matrix and $\vec{m}$ is an integral positive vector of dimension d' is a* **modular mapping**.

With a multi-dimensional mapping, each tile is assigned to a "processor number" in the form of a vector. The product of the components of $\vec{m}$ is equal to the number of processors. It then remains to define a one-to-one mapping from the hyper-rectangle $\{\vec{j} \in \mathbb{Z}^{d'} \mid \vec{0} \leq \vec{j} < \vec{m}\}$ (inequalities component-wise) onto the processor numbers. This can be done by viewing the processors as a virtual grid of dimension d' of size $\vec{m}$. The mapping $M_{\vec{m}}$ is then an assignment of each tile (described by its coordinates in the d-dimensional array of tiles) to a processor (described by its coordinates in the d'-dimensional virtual grid). (Note: in our construction, we will need only the case $d' = d - 1$.)

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

Definition 2 *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 *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 *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 *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 *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 $\mathcal{I}_m$.*

Because a modular mapping is linear, it is easy to see that 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 also have the following straightforward result.

Lemma 5 *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 $\mathcal{I}_m$.*

Lemmas 4 and 5 explain why we focus on one-to-one modular mappings first, then on many-to-one modular mappings, and finally on modular mappings with the load-balancing property. In the extended version of this paper [10], we explore the properties of such modular mappings, in order to define a provably adequate matrix M and shape $\vec{m}$ for the virtual grid of processors. Our results are linked to previous works by Lee and Fortes [14] and Darte, Dion, and Robert [9] to the case of one-to-one modular mappings. As in [9], the theory we developed is linked to a famous (in covering/packing theory) theorem due to Hajos [12]. Our results are also connected (through the use of Hajos' theorem) to scheduling techniques used in systolic-like array design (see [8] and [11]) for generating "juggling schedules". However, unlike these two works, which are "one-to-one"-like problems, many questions remain open in the many-to-one case because the extension of Hajos' theorem to a similar "many-to-one" case is true only up to dimension 3 included. Also, while it is easy to build a one-to-one mapping (just take $\vec{m} = \vec{b}$ and the identity matrix!), here we need a much more constrained matrix, such that any submatrix obtained by removing one column is many-to-one for the corresponding $\vec{b}$ and $\vec{m}$. In other words, to use the terminology [11], we need to juggle simultaneously in all dimensions!

We just give here the steps of our construction. We 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 also 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 choose the matrix M with the following form:

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

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

Lemma 6 *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 $\mathcal{I}_b(0, d)$ 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 $\mathcal{I}_b(0, i)$, $1 \le i \le d$ (Lemma 4). To prove this lemma, we only have to prove that this is true for the slices $\mathcal{I}_b(0, i)$, $i < d$ if N has the properties stated.

Without loss of generality, let us consider the first dimension, i.e., the first slice $\mathcal{I}_b(0, 1)$. 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 $i_1 = 0$. Now $(M\vec{i} = \vec{j} \bmod \vec{m}) \Leftrightarrow (N\vec{i}' = \vec{j}' \bmod \vec{m}'$ 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:

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

(By convention, an "empty" product is equal to 1). The vector $\vec{m}$ defined this way has several properties that will make a recursive construction of M possible (see [10] again).

Because $m_1 = 1$, we will be able to drop, at the end of the construction, the first component of the mapping, and 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). Once N is built, we write:

$$M = \begin{pmatrix} N & 0 \\ \vec{\lambda} & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ \vec{u} & T & 0 \\ \rho & \vec{z} & 1 \end{pmatrix}$$

and we define ρ and $\vec{z}$ such that $\vec{z} = -\vec{t}T$ and $\rho = 1 - \vec{t}.\vec{u}$, where the row vector $\vec{t}$, with $(d-2)$ components, is defined by the following (decreasing) recurrence:

- $r_{d-1} = m_d$,
- for $1 \leq i \leq d-2$, $t_i = \frac{r_{i+1}}{\gcd(b_{i+1}, r_{i+1})}$ and $r_i = \gcd(t_i m_{i+1}, r_{i+1})$.

This schema corresponds to the C program of Figure 3 (where the matrix M has rows and columns from 1 to d as in the presentation of this paper). In our current implementation, we of course take the final matrix modulo the corresponding values of $\vec{m}$. 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 [9] (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 in Figure 3.

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

  for (i=1; i<=d; i++)
    for (j=1; j<=d; j++)
      if ((i==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);
    }
  }
}
```

Figure 3: Program for generating a mapping with the load-balancing property.

7 Multipartitionings in dHPF

We have implemented support for Naik-style diagonal multipartitionings in our dHPF compiler. We are currently in the process of implementing support for *general* multipartitionings.

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 diagonal multipartitionings the number of processors cannot be specified on a per dimension basis. 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 section 2). Figure 1 illustrates a 3D diagonal multipartitioning on 16 processors. In general, each processor handles a total of $p^{\frac{1}{d-1}}$ tiles where p is the number of processors and d is the number of multipartitioned dimensions of the array. However, diagonal multipartitionings are restricted to having exactly **one** tile per hyperplane.

There are several important issues for correctly generating efficient code for diagonal multipartitioned distributions:

- **Tile Iteration Order**: 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.

- **Inter-loop nest Communication Aggregation**: 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. This is possible because multipartitioning guarantees that the neighboring tiles for a particular processor will be the same for all of its owned tiles.

In the case of generalized multipartitionings, we might have distributions in which we have more than one tile per processor on a single hyperplane. In order to generate high-performing code, we have to address these challenges:

- **Extended Tile Iteration Order**: For a single hyperplane, a processor may need to enumerate several tiles. The enumeration order does not have any bearing on correctness because dependences are being carried across hyperplanes instead of within a single hyperplane.

- **Intra-loop nest Communication Aggregation**: Communication cause by a loop-carried dependence may require several of a processor's tiles on a single hyperplane to send or receive data. We desire that this communication event should be executed as a single unit, instead of once per tile. This is possible because generalized multipartitionings provide the same neighborhood guarantee as simpler, diagonal multipartitionings.

8 Experiments

Our implementation of multipartitioning in dHPF currently supports only Naik-style diagonal multipartitionings. By using a multipartitioned data distribution in conjunction with several other important compilation techniques we have been able to obtain near hand-coded performance on the NAS benchmarks SP and BT [3, 7]. These results and details on the compilation techniques have been described in [7, 6, 1, 2].

The most important analysis and code generation techniques used to obtain high-performing multipartitioned applications are:

- Non-owner computes computation partitionings
- Communication vectorization
- Aggressive communication placement
- Intra-variable and inter-variable communication aggregation
- Prefetchable dynamic array references

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.

Tables 1 and 2 show the speedups for both the dHPF-generated and hand-coded versions of the NAS SP benchmark, for classes 'A' and 'B' respectively. Each table presents the speedups for the hand-coded

# CPUs	hand-coded	dHPF	% diff.
1	1.01	0.96	5.50
4	4.21	3.21	23.72
9	11.60	7.51	35.30
16	16.21	14.13	12.85
25	21.00	20.13	4.15
36	30.69	26.92	12.26

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

# CPUs	hand-coded	SP dHPF	% diff.
1	0.80	0.78	2.67
4	2.86	2.52	12.13
9	7.74	6.17	20.26
16	13.01	11.29	13.22
25	22.15	17.11	22.75
36	36.52	25.69	29.65

Table 2: Comparison of hand-coded and dHPF speedups for NAS SP (class B).

# CPUs	hand-coded	dHPF	% diff.
1	1.06	1.09	-2.64
4	3.28	3.34	-1.77
9	7.73	7.26	6.14
16	14.21	13.49	5.10
25	21.08	20.66	1.98
36	29.78	28.77	3.40

Table 3: Comparison of hand-coded and dHPF speedups for NAS BT (class A).

# CPUs	hand-coded	dHPF	% diff.
1	0.98	0.92	5.85
4	3.37	2.91	13.48
9	4.91	5.63	-14.70
16	12.30	12.83	-4.34
25	19.09	19.91	-4.30
36	30.95	28.80	6.93

Table 4: Comparison of hand-coded and dHPF speedups for NAS BT (class B).

version, the dHPF version and the differences between them. Speedups are relative to the respective sequential version of NAS SP. The average difference is 16%.

Tables 3 and 4 show the speedups for both the dHPF-generated and hand-coded versions of the NAS BT benchmark, for classes 'A' and 'B' respectively. Speedups are relative to the respective sequential version of NAS BT. The average difference is 1.5%.

9 Conclusions

We described an algorithm for generating multipartitioned data distributions that is applicable for all numbers of processors. For arrays of two or more dimensions, our algorithm will compute generalized multipartitionings. These partitionings minimize the cost with respect to an objective function based on a simple communication model. This objective function minimizes the cost of communication in line sweep computations.

We have generalized the concept of multipartitionings, to support fully parallel execution for line-sweep computations using an arbitrary number of processors. Previous work on multipartitionings required that for an d-dimensional multipartitioning the $(d-1)^{th}$ root of the number of processors had to be an integer. Our extensions allow for d-dimensional multipartitionings on any number of processors.

Using a simplified execution cost model, we developed a fast algorithm to select an optimal data partitioning (lowest communication cost, full processor utilization). The data array is multipartitioned in such a way that the number of tiles in each slice is a multiple of the number of processors.

We have shown that, having a partition 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 also give a constructive method for building this mapping using new techniques based on modular mappings. These techniques assign the optimal tiles obtained by the partitioning algorithm, to the physical processors that are going to compute on them.

We have also started the implementation of these algorithms in the dHPF compiler, which already supported diagonal multipartitionings.

References

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

[2] V. Adve and J. 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] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[4] 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, Jan. 1988.

[5] C. Caldwell. The prime pages. `http://www.utm.edu/research/primes`, 2001.

[6] D. Chavarría-Miranda and J. 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] D. Chavarría-Miranda, J. Mellor-Crummey, and T. Sarang. Data-parallel compiler support for multipartitioning. Technical Report CS-TR01-374, Dept. of Computer Science, Rice University, Mar. 2001.

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

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

[10] A. Darte, J. Mellor-Crummey, R. Fowler, and D. Chavarría. On efficient parallelization of line-sweep computations. Technical Report CS-TR01-377, Dept. of Computer Science, Rice University, Apr. 2001.

[11] 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.

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

[13] S. L. Johnsson, Y. Saad, and M. H. Schultz. Alternating direction methods on multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 8(5):686–700, 1987.

[14] H. J. Lee and J. A. Fortes. On the injectivity of modular mappings. In P. Cappello, R. M. Owens, J. Earl E. Swartzlander, and B. W. Wah, editors, *Application Specific Array Processors*, pages 237–247, San Francisco, California, Aug. 1994. IEEE Computer Society Press.

[15] N. 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.

[16] J. 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.

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

[18] 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.