Generalized Multipartitioning for Multi-dimensional Arrays*

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
C. S. Dept., MS-132, Rice University, 6100 Main St, Houston, TX USA
{danich,rjf,johnmc}@cs.rice.edu

Abstract

Multipartitioning is a strategy for parallelizing computations that require solving 1D recurrences along each dimension of a multi-dimensional array. 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 multipartitionings for d-dimensional data volumes on an arbitrary number of processors. We describe an algorithm that computes an optimal multipartitioning onto all of the processors for this general case. Finally, we describe how we extended the Rice dHPF compiler for High Performance Fortran to generate code that exploits generalized multipartitioning and show that the compiler's generated code for the NAS SP computational fluid dynamics benchmark achieves scalable high performance.

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.

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

To support better parallelization of line sweep computations, a third sophisticated strategy for partitioning data and computation known as **multipartitioning** was developed [4, 13, 15]. This strategy partitions arrays of $d \geq 2$ dimensions among a set of proces-

*This research was supported in part by the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23 as part of the prime contract (W-7405-ENG-36) between the DOE and the Regents of the University of California.

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

sors so that for a line sweep computation along any dimension of an array, all processors are active in each step of the computation, load-balance is nearly perfect, and only coarse-grain communication is needed. These properties are achieved by (1) assigning each processor a balanced number of tiles in each hyper-rectangular slab defined by a pair of adjacent cuts along a partitioned data dimension and (2) ensuring that for all tiles mapped to a processor, their immediate tile neighbors in any one coordinate direction are all mapped to some other single processor. We later refer to these two properties as the **balance** property, and the **neighbor** property respectively. A study by van der Wijngaart [18] of strategies for hand-coded parallelizations of ADI Integration found that 3D multipartitionings yield better performance than static block or dynamic block partitionings.

All of the multipartitionings described in the literature to date consider only one tile per processor per hyper-rectangular slab along a partitioned dimension. The most broadly applicable of the multipartitioning strategies in the literature is known as **diagonal multipartitioning**. In 2D, these partitionings can be performed on any number of processors, p; however, in 3D they are only useful if p is a perfect square. We consider the general problem of computing optimal multipartitionings for d-dimensional data volumes for an arbitrary number of processors.

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 an implementation of generalized multipartitioning in the Rice dHPF compiler for High Performance Fortran. We show that it yields scalable high performance when used to parallelize the NAS SP [3] computational fluid dynamics benchmark.

2. Background

Johnsson *et al.* [13] describe a 2D domain decomposition strategy, now known as a multipartitioning, for parallel implementation of ADI integration on a multiprocessor ring. They partition both dimensions of a 2D domain to form a $p \times p$ grid of tiles. They use a tile-to-processor mapping $\theta(i,j) \equiv (i-j) \bmod p$, $0 \le i,j < p$, to map from the $[i,j]$ coordinates of

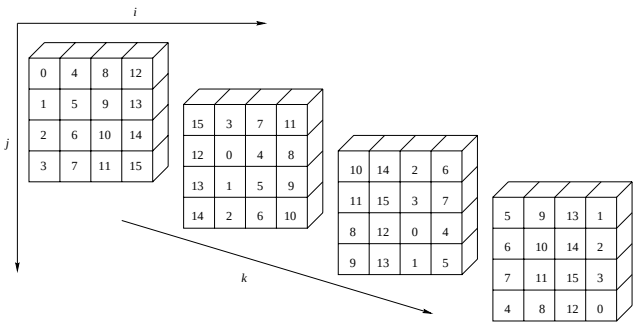

Figure 1. A 3D Multipartitioning.

each tile to its corresponding processor. This partitioning is an instance of a **latin square** [10]. Using this mapping for an ADI computation, each processor exchanges data with only its 2 neighbors in a linear ordering of the processors, which maps nicely to a ring.

Bruno and Cappello [4] devised a 3D partitioning for parallelizing 3D ADI integration computations on a hypercube architecture. They describe how to map a 3D domain cut into $2^d \times 2^d \times 2^d$ tiles on to 2^{2d} processors with a tile-to-processor mapping $\theta(i,j,k)$ based on Gray codes: 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.* [15] describe **diagonal multipartitionings** for 2D or 3D problems. Diagonal multipartitionings are a generalization of Johnsson *et al.*'s 2D partitioning strategy that are more broadly applicable than the Gray code based mapping described by Bruno and Cappello. The 3D diagonal multipartitionings described by Naik *et al.* partition the data into $p^{\frac{3}{2}}$ tiles, with each processor's tiles arranged along wrapped diagonals through the 3D volume. Figure 1 shows a 3D multipartitioning of this style for 16 processors; the number in each tile indicates the processor that owns the block. This 3D diagonal multipartitioning (there are many) is specified by the tile to processor mapping $\theta(i,j,k) \equiv ((i-k) \bmod \sqrt{p})\sqrt{p} + ((j-k) \bmod \sqrt{p})$ for a domain of $\sqrt{p} \times \sqrt{p} \times \sqrt{p}$ tiles where $0 \le i,j,k < \sqrt{p}$, where $\sqrt{p} = 4$.

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. In 2D, this yields an *optimal* multipartitioning (equivalent to those described by Johnsson *et al.*). We call a multipartitioning optimal for a particu-

lar number of processors if no other multipartitioning exists that has lower communication cost according to a cost model that considers both fixed overhead for communicating and overhead proportional to the size of the hyper-surfaces that must be communicated. For $d > 2$, diagonal multipartitionings are only optimal and efficient when $p^{\frac{1}{d-1}}$ is integral.

Bruno and Cappello noted that multipartitionings need not be restricted to having only one tile per processor per hyper-rectangular slab of a multipartitioning [4]. How general can multipartitioning mappings be? A necessary condition to support load-balanced line-sweep computation is that in any hyper-rectangular slab defined by adjacent cuts along a partitioned dimension, each processor must have the same number of tiles. We call any such slab 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 the mapping possesses the **balance** and **neighbor** properties of a multipartitioning? The answer is yes. We show that such an assignment is possible if and only if the number of tiles in each hyper-rectangular slab along any partitioned dimension is a multiple of p ("if" being the difficult part of the proof). We describe a "regular" solution (regular to be defined) that enables us to guarantee that the neighboring tiles along any one coordinate direction of all tiles mapped to a processor all belong to a single processor. This property of multipartitionings is essential for fully-vectorized, directional-shift communication to be efficient.

In Section 3.1, we define an objective function that represents the execution time of a line-sweep computation over a multipartitioned array, and in Section 3.3, we present an algorithm that computes a partitioning of a multi-dimensional array into tiles that is optimal with respect to this objective. In Section 4, 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:

- 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, α_j.
- d is the number of dimensions of the array to be partitioned. The array is of size $\eta_1, \ldots, \eta_d$. The total number of array elements $\eta = \prod_{i=1}^{d} \eta_i$.
- γ_i is the number of tiles into which the array is cut along its i-th dimension. We consider the array of elements as a $\gamma_1 \times \ldots \times \gamma_d$ array of tiles. In our analysis, we assume that γ_i divides η_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 that each slab 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 $\prod_{j \neq i} \gamma_j$. When this is true, we say that (γ_i) is a **valid partitioning**.

3. Finding the Partitioning

3.1. 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 η, the number of elements in the array. A sweep along the i-th dimension consists of a sequence of γ_i computation phases (one for each hyper-rectangular slab of tiles along dimension i), separated by $\gamma_i - 1$ communication phases. The work in each slab 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 $\prod_{j \neq i} \eta_j$ elements, i.e., $\frac{\eta}{\eta_i}$, yielding a communication volume per processor of $\frac{\eta}{p\eta_i}$. The total execution time for a sweep along dimension i can be approximated by:

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

where K_1 is a constant that depends on the sequential computation time per data element, K_2 is a constant that depends on the cost of initiating one communication phase (start-up), and $K_3(p)$ is a function of p that reflects the bandwidth-sensitive communication cost per element of hyper-surface area along a cut in dimension i.[1] Define $\lambda_i = K_2 + K_3(p)\frac{\eta}{\eta_i}$; λ_i depends on the domain size, number of processors and machine's communication parameters. The total cost, sweeping

[1] On a parallel machine in which the network bandwidth available is directly proportional to the number of processors, $K_3(p)$ would be proportional to $\frac{1}{p}$, whereas on a bus-based system for which available bandwidth is fixed, $K_3(p)$ would be a constant.

in all dimensions, is thus

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

Assuming that p, η, and the η_i's are given, the first term is a constant, and what we want to minimize is the second term $\sum_{i=1}^{d}\gamma_i\lambda_i$.

Remark: 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 $\sum_i \frac{\gamma_i}{\eta_i}$, 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 a short extent in one dimension, it is preferable to use a 2D partitioning of the other 2 dimensions rather than a 3D partitioning. Indeed, if η_1 and η_2 are at least 4 times larger than η_3, then cutting each of the first 2 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 ($\forall i$, $\gamma_i = 2$). The extra communication while sweeping along the first 2 dimensions is offset by the absence of communication in the local sweep along the last one.

We now address the problem of minimizing $\sum_i \gamma_i\lambda_i$ with the constraint that, for any fixed i, p divides the product of the γ_j's, $j \neq i$. We give a practical algorithm, based on an (optimized) exhaustive search, exponential in s (the number of distinct 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 $\forall i$, $\lambda_i = 1$, or if there is an algorithm polynomial in $\log p$ or even in the s values $\log r_i$. If p has only one prime factor, a greedy approach leads to a polynomial (polynomial in $\log p$) algorithm (see [8]). However, we do not know if an extension of this greedy approach can lead to a polynomial algorithm for an optimal partitioning in the general case.

3.2. Elementary Partitionings

If (γ_i) is a valid partitioning such that $\sum_i \gamma_i\lambda_i$ is minimized, we say that (γ_i) is an **optimal partitioning**. Using the fact that for each $1 \leq i \leq d$, p divides $\prod_{j\neq i}\gamma_j$ and that the objective function increases when the γ_i increase (the λ_i are positive), we can show the following result. (The proof is not difficult, we omit it due to space constraints.)

Lemma 1 *Let (γ_i) be an optimal partitioning. Then, each factor α_j of p, appearing r_j times in the decomposition of p, appears exactly $(r_j + m_j)$ times in (γ_i), where m_j is the maximum number of occurrences of α_j in any γ_i. Furthermore, the number of occurrences of α_j is m_j in at least two γ_i's.*

We can thus restrict to **elementary partitionings**, those that satisfy the conditions of Lemma 1. We can interpret (and manipulate) an elementary partitioning as a distribution of the factors of p into d bins, satisfying a particular constraint on the number of occurrences. Elementary partitionings are those which are not a "multiple" of another possible size; in other words, these are the sizes for which a multipartitioning exists that cannot be obtained by composing it (by paving) from multiple instances of a smaller multipartitioning. For example, in 3D, with 8 processors, only the partitionings $4\times4\times2$, $8\times8\times1$, and their permutations are elementary. With $p = 5\times3\times2$, only the partitionings $10\times15\times6$, $15\times30\times2$, $10\times30\times3$, $5\times30\times6$, $30\times30\times1$ (and permutations) are elementary.

3.3. Exhaustive Enumeration

We now give an algorithm that finds an optimal partitioning by generating all possible elementary partitionings (γ_i), which satisfy the necessary optimality conditions given by Lemma 1, and determining which one yields the lowest cost partitioning. We also evaluate how many candidate partitions there are to give the complexity of our algorithm. For the complexity, we are not interested in the exact number of elementary partitionings, 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 shown in Figure 2 generates, in linear time, all possible distributions of r_j instances of a factor α_j of p into d bins that satisfy the $(r_j + m_j)$ optimality condition of Lemma 1. This program 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 multiplicity m of the factor under consideration that may appear in any bin, and uses the recursive procedure `P(n,m,c,t,d)` to generate all distributions of n elements in $(d - t + 1)$ bins (from index t to index d), where each bin can have at most m instances of the factor and at least c bins must have m instances of the factor. 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 2. Program for generating all possible distributions for one factor.

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 the loop selects a different number of elements for the current 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 at most 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 one valid partitioning, that all possible numbers of elements are selected and only those.

Let us first consider the loop in function `Partitions`. Thanks to Lemma 1, it is easy to see 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; for instance, the $r-m$ elements could be distributed so that $q = \lfloor \frac{r-m}{m} \rfloor$ bins contain m elements and one contains $(r-m-mq)$ elements. Thus, if the function `P` is correct, `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 $cm \leq n-i \leq (d-t)m$, i.e., $n-(d-t)m \leq i \leq n-cm$. Furthermore, i should be strictly less than m, nonnegative, and at most 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 at most 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.

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 elementary partitionings, which satisfy Lemma 1 for each factor, with the function `Partitions` and we combine them while keeping track of the best overall solution. The overall complexity (excluding the cost of the decomposition of p into prime factors) is the product of the complexity of the function `Partitions` (which is the number of solutions generated by the algorithm) times $(\log_2 p)^3$ (to built the γ_i's and evaluate them). We proved that the total number of generated solutions (i.e., the number of elementary partitionings) is $O\left(\left(\frac{d(d-1)}{2}\right)^{\frac{(1+o(1))\log p}{\log\log p}}\right)$

and that this bound is tight. (The proof is too long to be provided here but is available in the extended version of this paper [8].)

4. Finding the Mapping

In Section 3, 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. Up to this point, we have assumed that we will be able to assign tiles to processors so that the assignment posesses the *balance* and *neighbor* properties of a multipartitioning. This has not yet been shown, and we need to prove it. We point out that an assignment with the *balance* property is a generalization of the notion of **latin square** that is known as as an **F-hyper-rectangle** [10, page 392]. However, despite this reference, we have not found any paper that gives a construction for such an assignment, or even an existence proof, for our general case. Furthermore, even if such a proof exists, which we are not aware of, our constructive proof is of interest because:

- its tile-to-processor mappings have the neighbor property,
- its tile-to-processor mappings are given by a simple formula, and conversely, for each processor, the list of tiles assigned to it can be easily formulated, which is handy for use in a run-time library,
- it gives a new insight to the properties of "modular" mappings (defined below).

Therefore, we make no further reference to latin squares and F-hyper-rectangles and proceed with a presentation of our proof.

The only property we know so far is that the (γ_i) is a valid partitioning, namely, for each i, p divides $\prod_{j\neq i}\gamma_j$. Our main result is that this condition is sufficient to guarantee a mapping of processors to tiles that possesses both the balance and neighbor properties. Our proof is constructive. For any valid partitioning (γ_i), optimal or not, with or without the additional property of Lemma 1, we give an automatic way to assign a processor number to each tile so that the properties are 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 [8] 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 3.1), the network topology is not taken into account yet and all valid mappings are considered equally good.

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 Euclidean division by 4. The assignment in the figure corresponds to $(i-k \bmod 4, j-k \bmod 4)$, which is what we call a **multi-dimensional modular mapping**, i.e., a mapping $M_{\vec{m}}$ from $\mathbb{Z}^d$ to $\mathbb{Z}^{d'}$ defined by an integral $d\times d'$ matrix M and an integral positive vector $\vec{m}$ of dimension d' with $M_{\vec{m}}(\vec{i}) = (M\vec{i}) \bmod \vec{m}$. With such a 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 $\{\vec{j} \in \mathbb{Z}^{d'} \mid \vec{0} \leq \vec{j} < \vec{m}\}$ 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). (Actually, we 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. Given $\vec{b} \in \mathbb{N}^n$, the **hyper-rectangle** defined by $\vec{b}$ is the set $\mathcal{I}_{\vec{b}} = \{\vec{i} \in \mathbb{Z}^n \mid \vec{0} \leq \vec{i} < \vec{b}\}$ (component-wise). A **slice** $\mathcal{I}_{\vec{b}}(i, k_i)$ of $\mathcal{I}_{\vec{b}}$ is defined as the set of all elements of $\mathcal{I}$ whose i-th component is equal to k_i (an integer between 0 and $b_i - 1$). Given a hyper-rectangle $\mathcal{I}_{\vec{b}}$ (or any more general set), a modular mapping $M_{\vec{m}}$ is **one-to-one from $\mathcal{I}_{\vec{b}}$ onto $\mathcal{I}_{\vec{m}}$** if and only if for each $\vec{j} \in \mathcal{I}_{\vec{m}}$ there is one and only one $\vec{i} \in \mathcal{I}_{\vec{b}}$ such that $M_{\vec{m}}(\vec{i}) = \vec{j}$. $M_{\vec{m}}$ is **equally-many-to-one from $\mathcal{I}_{\vec{b}}$ onto $\mathcal{I}_{\vec{m}}$** if and only if the number of $\vec{i} \in \mathcal{I}_{\vec{b}}$ such that $M_{\vec{m}}(\vec{i}) = \vec{j}$ does not depend on $\vec{j}$. Finally, $M_{\vec{m}}$ has the **load-balancing property** for $\mathcal{I}_{\vec{b}}$ if and only if for any slice $\mathcal{I}_{\vec{b}}(i, k_i)$, the restriction of $M_{\vec{m}}$ to $\mathcal{I}_{\vec{b}}(i, k_i)$ is equally-many-to-one onto $\mathcal{I}_{\vec{m}}$.

Because a modular mapping is linear, it is easy to see that the load-balancing property needs to be checked only for the slices that contain $\vec{0}$ (the slices $\mathcal{I}_{\vec{b}}(i, 0)$). Furthermore, if $\vec{b}[i]$ denotes the vector obtained from $\vec{b}$ by removing the i-th component and $M[i]$ denotes the

matrix obtained from M by removing the i-th column, then the images of $\mathcal{I}_{\vec{b}}(i,0)$ under $M_{\vec{m}}$ are the images of $\mathcal{I}_{\vec{b}[i]}$ under the modular mapping $M[i]_{\vec{m}}$. We therefore have the following properties.

Lemma 2 *Given an hyper-rectangle $\mathcal{I}_{\vec{b}}$, a modular mapping $M_{\vec{m}}$ has the load-balancing property for $\mathcal{I}_{\vec{b}}$ if and only if each mapping $M[i]_{\vec{m}}$ is equally-many-to-one from $\mathcal{I}_{\vec{b}[i]}$ to $\mathcal{I}_{\vec{m}}$.*

Lemma 3 *If $M_{\vec{m}}$ is a one-to-one modular mapping from $\mathcal{I}_{\vec{b}'}$ onto $\mathcal{I}_{\vec{m}}$, then $M_{\vec{m}}$ is an equally-many-to-one modular mapping from any multiple $\mathcal{I}_{\vec{b}}$ of $\mathcal{I}_{\vec{b}'}$ onto $\mathcal{I}_{\vec{m}}$.*

Lemmas 2 and 3 explain why we focus on one-to-one modular mappings first, then on equally-many-to-one modular mappings, and finally on modular mappings with the load-balancing property. In the extended version of this paper [8], 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 on one-to-one modular mappings by Lee and Fortes [14] and Darte, Dion, and Robert [7]. As in [7], the theory we developed is linked to a famous (in covering/packing theory) theorem due to Hajós [12], which has previously been used to generate "juggling schedules" for systolic-like array designs (see [9]). These earlier papers all consider "one-to-one"-like problems; however, many questions remain open in the equally-many-to-one case because the extension of Hajós' theorem to a similar "equally-many-to-one" case is true only up through 3 dimensions. 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 more constrained matrix such that any submatrix obtained by removing one column is equally-many-to-one for the corresponding $\vec{b}$ and $\vec{m}$. In other words, to use the terminology in [9], we need to juggle simultaneously in all dimensions.

Here we present our construction of a modular mapping $M_{\vec{m}}$ with the load-balancing property for an index set $\mathcal{I}_{\vec{b}}$ (which is given, $\vec{b}$ is the vector whose components are the γ_i's found in Section 3.3). 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 $\mathcal{I}_{\vec{m}}$ (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 partitioning: 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 4 *Suppose that m_d divides b_d and that the modular mapping $N_{\vec{m}[d]}$ – in dimension $(d-1)$ – has the load-balancing property for $I_{\vec{b}[d]}$. Then, the modular mapping $M_{\vec{m}}$ – in dimension d – has the load-balancing property for $\mathcal{I}_{\vec{b}}$ if it is equally-many-to-one from the last slice $\mathcal{I}_{\vec{b}}(d,0)$ onto $\mathcal{I}_{\vec{m}}$.*

Proof: In order to check that the mapping defined by M and $\vec{m}$ has the load-balancing property for the rectangular index set $\mathcal{I}_{\vec{b}}$, we have to make sure that it is equally-many-to-one for all slices $\mathcal{I}_{\vec{b}}(i,0)$, $1 \leq i \leq d$ (Lemma 2). Since we assume that this is true for $i = d$, we only have to prove it for the slices $\mathcal{I}_{\vec{b}}(i,0)$ with $i < d$.

Without loss of generality, let us consider the first dimension, i.e., the first slice $\mathcal{I}_{\vec{b}}(1,0)$. Given $\vec{j} \in \mathcal{I}_{\vec{m}}$, let us count the number of vectors $\vec{i} \in \mathcal{I}_{\vec{b}}$ such that $M\vec{i} = \vec{j} \bmod \vec{m}$ and $i_1 = 0$. By definition of M and N, $(M\vec{i} = \vec{j} \bmod \vec{m}) \Leftrightarrow (N\vec{i}[d] = \vec{j}[d] \bmod \vec{m}[d]$ and $\vec{\lambda}.\vec{i}[d] + i_d = j_d \bmod m_d)$ where $\vec{\lambda}$ is the row vector formed by the first $(d-1)$ component of the last row of M. Because $N_{\vec{m}[d]}$ has the load-balancing property for $\mathcal{I}_{\vec{b}[d]}$, there are exactly n vectors $\vec{i}' \in \mathcal{I}_{\vec{b}[d]}$ such that $i'_1 = 0$ and $N\vec{i}' = \vec{j}[d] \bmod \vec{m}[d]$, where n is a positive integer that does not depend on $\vec{j}[d]$. 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 exactly $(nb_d)/m_d$ pre-images in $\mathcal{I}_{\vec{b}}(1,0)$ and this number does not depend on $\vec{j}$. ■

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

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

(By convention, an "empty" product is equal to 1.) Thanks to the previous lemma and the properties of the vector $\vec{m}$ defined this way, we will be able to build M in a recursive manner (see [8]). Because $m_1 = 1$, we will be able to drop, at the end, the first component of the mapping and get a mapping from $\mathbb{Z}^d$ into a subgroup of $\mathbb{Z}^{d-1}$ (or of smaller dimension if some other components of $\vec{m}$ are equal to 1). Once N is built, we write:

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

```
// Precondition: d >= 2
void ModularMapping(int d) {
  int i,j,r,t;
  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);
    }
  }
}
```

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

and we define ρ and $\vec{z}$ (a row vector) 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 (with the help of an intermediate vector $\vec{r}$):

- $r_{d-1} = m_d$,
- 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 recurrence is linked to the *symbolic* computation of some **Hermite form** that we use to be able to apply Lemma 4 and prove the validity of the recursive construction. See details in [8].

This schema is implemented by the C program shown in Figure 3 (rows and columns are from 1 to d). In our actual implementation of this algorithm, we augment the basic kernel presented to compute the final matrix modulo the corresponding values of $\vec{m}$ as well as apply some strategies (*e.g.,* alternating signs of $\vec{t}$, or pre-permuting the components of $\vec{b}$) to make coefficients smaller.

5. Experiments

We extended the Rice dHPF compiler for High Performance Fortran to generate code based on generalized multipartitionings.

Multipartitioning within the dHPF compiler is implemented as a generalization of `BLOCK`-style HPF partitionings [5, 6]. The dHPF compiler analyzes communication and reduces loop bounds as if a multipartitioned template is a standard `BLOCK` partitioned template mapped onto an array of processors of symbolic extent. The main difference comes in the interpretation that the compiler gives to the `PROCESSORS` directive. When using multipartitioning, the number of processors cannot be specified on a per dimension basis for dimensions of the template because each hyperplane defined by a partitioning along a multipartitioned template dimension is distributed among all processors. A multipartitioned template is partitioned into tiles according to the rank and extent of the virtual processor array. These tiles are then assigned in a skewed-cyclic fashion to the processors as described in 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. Second, communication that has been fully vectorized out of a loop nest should not be performed on a tile-by-tile basis; instead it should be performed for all of a processor's tiles at once. Communication aggregation is more tricky than for diagonal multipartitionings since generalized multipartitionings have multiple tiles per hyperrectangular slab, but it is possible because generalized multipartitionings also possess the *neighbor* property described earlier in Section 1. Third, communication caused by loop-carried dependences should not be performed on a tile-by-tile basis either. Instead, communication should be vectorized for all tiles within a hyperrectangular slab along the partitioned dimension.

By using a multipartitioned data distribution in conjunction with sophisticated data-parallel compiler optimizations, we are closing the performance gap between compiler-generated and hand-coded implementations of line-sweep computations. Earlier results and details about dHPF's compilation techniques can be found elsewhere [6, 5, 1, 2]. Here we present results from applying generalized multipartitioning in the context of a compiler-based parallelization of the NAS SP computational fluid dynamics application benchmark [3, 6] for the "class B" problem size of 102^3.

The most important analysis and code generation techniques used to obtain high-performance multipartitioned applications by the dHPF compiler are: partial replication of computation to reduce communication frequency and volume, communication vectorization, aggressive communication placement, and communication aggregation to reduce the number of mes-

# CPUs	hand-coded	dHPF	% diff.
1	0.95	0.91	3.84
2		1.43	
4	2.96	2.93	1.00
6		5.06	
8		7.57	
9	7.95	8.04	-1.14
12		11.80	
16	16.64	16.25	2.34
18		18.54	
20		19.03	
24		22.25	
25	27.44	24.32	11.38
32		32.22	
36	38.46	38.83	-0.97
45		39.78	
49	48.37	51.49	-6.46
50		47.35	
64	76.74	59.84	22.02
72		66.96	
81	81.40	70.63	13.23

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

sages. In addition, we use an extended on-home directive (inspired by the HPF/JA `EXT_HOME` directive[11]) to partially replicate computation into a processor's shadow regions, and the HPF/JA `LOCAL` directive to eliminate unnecessary communication for values that were previously explicitly computed in a processor's shadow region.

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

Table 1 compares the performance of a hand-coded MPI version of the SP benchmark developed at NASA Ames Research Center with an MPI version generated by the dHPF compiler.[2] The hand-coded version uses 3D diagonal multipartitioning and thus can only be run on a perfect square number of processors. The dHPF-generated code MPI uses generalized multipartitioning which enables the code to be run on arbitrary numbers of processors. As Table 1 shows, the performance of the dHPF-generated code is quite close to (and sometimes exceeds) the performance of the hand-coded MPI for numbers of processors that are perfect squares. When the number of processors is a perfect square, the generalized multipartitionings used by the dHPF-generated code are exactly diagonal multipartitionings. These measurements show that our implementation of generalized multipartitionings is efficient in the case of diagonal multipartitionings, in which each processor has one tile per hyperplane of the partitioning. Both the hand-coded and dHPF-generated versions of SP deliver roughly linear speedup on numbers of processors that are perfect squares.

In the measurements taken of the dHPF-generated code for numbers of processors that are not perfect squares, we see that generalized multipartitionings deliver near linear speedup in these cases as well. The cases we have measured exploiting generalized multipartitioning are ones in which the factors of the number of processors are small primes. Performance would be less for numbers of processors that are prime or have large prime factors because computation would be divided into a large number of phases and communication volume grows in proportion to the number of phases. Currently, the code generated by dHPF cannot exploit generalized multipartitionings when the block size on any processor falls below the shift width associated with communication operations, which happens when a dimension is partitioned many times (as occurs with large primes and prime factors). This limitation prevents experiments with generalized multipartitionings using the 102^3 problem size of the SP benchmark on numbers of processors that are large primes or have large prime factors.[3]

Overall, these preliminary experiments show that generalized multipartitionings are of practical as well as theoretical interest and can be used to efficiently parallelize applications using multipartitioning in a wider range of cases.

6. Conclusions

This paper describes an algorithm for computing an optimal multipartitioning of d-dimensional arrays, $d > 2$, onto an arbitrary number of processors, p. Our algorithm minimizes cost according to an objective function that measures communication in line sweep computations. Previously, optimal multipartitionings could be computed only when $p^{\frac{1}{d-1}}$ is integral. We show that a partitioning in which the number of tiles in each hyperrectangular slab is a multiple of the num-

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

[3] To be perfectly clear, this limitation applies only to code generated by the dHPF compiler; the *technique* of generalized multipartitioning itself is completely general.

ber of processors — an obvious necessary condition — is also a sufficient condition for a multipartitioned mapping of tiles to processors. We present a constructive method for building the mapping of tiles to processors using new techniques based on modular mappings and demonstrate experimentally that code using generalized multipartitionings is both scalable and efficient.

Currently, when we multipartition a d-dimensional array onto p processors, we force *all* processors to participate in the computation; however, this may lead to suboptimal performance. If the partitioning is not **compact**, *i.e.*, the number of tiles per processor is large relative to a diagonal multipartitioning (more precisely, when $\prod_{i=1}^{d} \gamma_i$ is large compared to $p^{\frac{d}{d-1}}$), and the cost of communicating at tile boundaries is not small compared to the cost of the computation on tile data (the relative cost of communication to computation is proportional to the surface to volume ratio in the partitioning: $\sum_{i=1,d} \frac{\gamma_i}{\eta_i}$), it will be faster to drop back to a nearby lower number of processors for which a compact partitioning exists. For example, table 1 shows that for the 102^3 problem size, a $5 \times 10 \times 10$ decomposition on 50 processors is slower than a $7 \times 7 \times 7$ decomposition on 49 processors for NAS SP. Given a cost function (see Section 3.1) that models the cost of computation as well as communication, our algorithm could be used to search for the most efficient partitioning, which will occur on some number of processors between $\lfloor p^{\frac{1}{d-1}} \rfloor^{d-1}$ (for which a diagonal multipartitioning is possible) and p as long as the communication term is not dominant.

Acknowledgments

The authors wish to gratefully acknowledge the anonymous reviewers for their thoughtful comments which helped us improve the presentation of this paper.

References

[1] V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi. High Performance Fortran compilation techniques for parallelizing scientific codes. In *SC'98: 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 *SIGPLAN'98 Conference on Programming Language Design and Implementation*, Montreal, Canada, Jun. 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 *3rd Conference on Hypercube Concurrent Computers and Applications*, pages 1073–1087, Pasadena, CA, Jan. 1988.

[5] D. Chavarría-Miranda and J. Mellor-Crummey. Towards compiler support for scalable parallelism. In *5th Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, LNCS 1915, pages 272–284, Rochester, NY, May 2000. Springer-Verlag.

[6] D. Chavarría-Miranda, J. Mellor-Crummey, and T. Sarang. Data-parallel compiler support for multipartitioning. In *European Conference on Parallel Computing (Euro-Par)*, Manchester, United Kingdom, Aug. 2001.

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

[8] A. Darte, J. Mellor-Crummey, R. Fowler, and D. Chavarría. On efficient parallelization of line-sweep computations. Research Report RR2001-45, LIP, ENS-Lyon, France, 2001.

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

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

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

[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 *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, `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 *Supercomputing 1993*, pages 102–111. IEEE Computer Society Press, 1993.