Volume Driven Data Distribution for NUMA-Machines

Felix Heine and Adrian Slowik

University of Paderborn, Germany,
felix.heine@syskoplan.de, aslowik@trace.de

Abstract. Highly scalable parallel computers, e.g. SCI-coupled workstation clusters, are NUMA architectures. Thus good *static locality* is essential for high performance and scalability of parallel programs on these machines. This paper describes novel techniques to optimize static locality at compilation time by application of *data transformations* and *data distributions*. The metric which guides the optimizations employs *Ehrhart polynomials* and allows to calculate the amount of static locality *precisely*. The effectiveness of our novel techniques has been confirmed by experiments conducted on the SCI-coupled workstation cluster of the PC^2 at the University of Paderborn.[1]

1 Introduction

Clusters of workstations promise outstanding computational power at an economically attractive price. However, good static locality is a must to utilize the aggregated power of connected nodes. To give an illuminating example, we observed the execution time of the SOR-loop to be 2.1s for one of two nodes that was assigned *all* data, while it was 28.8s for the other node with *no* local data. This huge imbalance in execution time illustrates the impact of remote memory accesses and motivates the need for data transformations and data distributions that arrange for good data locality.

1.1 Problem Formulation

We use a restricted version of the HPF block-cyclic distribution model [6] starting with a *parallel* loop nest that comprises affine loop bounds and affine index functions to multidimensional arrays. The loop nest is expected to possess exactly one parallel loop. We assume that arrays are sliced into blocks along one dimension, which are then assigned to processing nodes. In the best case, any such a block is solely accessed by the processing node that *owns* the block. Hence it is the duty of data transformations to expose a regular pattern of blocks which are accessed by unique nodes. The subsequent data distribution then has to determine an assignment of blocks to processing nodes which is consistent with this

[1] This work has been supported in part by the DFG Sonderforschungsbereich 376 "Massive Parallelität – Algorithmen, Entwurfsmethoden, Anwendungen", Paderborn

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 415–424, 2000.

pattern. We do not use the *owner computes rule*. We preserve the assignment of computations to processors that was computed in previous compiler steps.

In summary, for each array of a regular program we automatically derive a *unimodular data transformation* which *reshapes* the array, and a *block-cyclic data distribution* which *distributes* the array elements among the processing nodes. The distribution employs some cycle length and a certain offset.

1.2 Related Work

The topics of data transformation and data distribution have attracted great interest within the last decade, such that an overwhelming amount of publications has emerged in this field. But unfortunately, it is difficult to compare the efficiency of our approach to those described in the literature, because in the literature, locality improvements are measured in runtime improvements with regard to some specific target architecture. We instead provide a general technique to derive parametric formulas that map to the number of local and remote accesses, and the locality optimization we propose also takes place on this level. Thus we express the achieved improvements in terms of formulas which do not depend on details of the target architecture. Nevertheless, we also provide runtime improvements observed on a SCI-coupled workstation cluster.

Known techniques for optimizing data distribution either use integer programming [7], or heuristics based on reuse vectors [11], [1], resp. affinity graphs [2]. These techniques do not consider the geometry of the iteration space, but only inspect index functions and the nesting structure of loop nests. To the best of our knowledge none of these techniques uses Ehrhart polynomials [3] to guide the selection of data transformations and data distributions. In this sense our novel approach is unique. In its general outline to generate a set of candidates and to identify one of these candidates using complex mathematical reasoning it resembles the approach taken in [7]. However, the latter uses integer programming and also neglects the geometry of the iteration space.

2 Geometric Framework

We use a multi-grid application from the area of fluid dynamics to illustrate our approach. The computational kernel is a variant of the SOR-loop. It is a 2-dimensional loop nest with 5 references to array `U` and 1 reference to array `F`. We focus on references to array `U` throughout this text. The parallel program version shown in Fig. 1, has been produced by the automatic parallelizer of our prototype compiler. The loop nest exposes parallelism on its innermost level and now is subject to subsequent data locality optimizations:

In the case of 2 parallel processors and a cyclic distribution of the columns of array `U`, we observe the access pattern shown in Fig. 2. It illustrates a *default-distribution* which results in 50% remote accesses, provided iteration point (i, j) is executed by processor $P_{j \bmod 2}$. This situation can not be remedied by a data distribution, because most array elements are accessed by both processors. Since

```
DO I = 2, M+N-2
  FORALL J = MAX(1, I-M+1), MIN(I-1, N-1)
    U(S, I-J, J-1) = (F(S, I-J, J) + U(S, I-J, J-1) + U(S, I-J-1, J)
                                   + U(S, I-J, J+1) + U(S, I-J+1, J)/4.0
  ENDFORALL
END DO
```

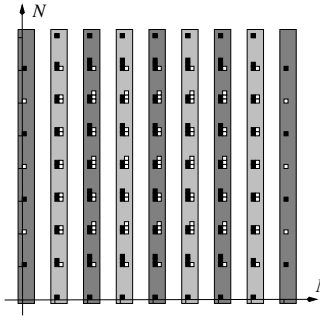
Fig. 1. SOR loop nest from multi-grid

we do not consider replication, some accesses are forced to be remote, no matter what the distribution parameters are. Nevertheless, the situation can be improved significantly by a preceding data transformation.

Now we introduce some convenient abbreviations and define fundamental geometric abstractions suited to rank transformations and distributions.

A loop nest N defines the *iteration space* $\mathcal{I}_N$. An array `X` that is accessed by a reference R_l, $l \geq 1$, defines the *index space* $\mathcal{D}_{\mathtt{X}}$. By f_l we refer to the index function of reference R_l. Furthermore, P denotes the number of processors, d the distribution dimension, B the block size, and j the parallel dimension. Henceforth, we omit subscripts if there is no risk of confusion.

It is well known that an iteration space $\mathcal{I}$ and an index space $\mathcal{D}$ both define convex polytopes [4], [8]. We represent a convex polytope $\mathcal{P}$ as usual by a set of inequations, i.e. $\mathcal{P} = \{\boldsymbol{x} \in \mathbb{Z}^k \mid \boldsymbol{A} \cdot \boldsymbol{x} + \boldsymbol{C} \cdot \boldsymbol{n} + \boldsymbol{b} \geq 0\}$. The Ehrhart polynomial E of a parameterized convex polytope $\mathcal{P}$ is a function in parameters of the polytope and maps to the number of integral points contained within $\mathcal{P}$ [3]. The fundamental idea of our approach is to encode iteration points that cause local accesses by convex polytopes. Then the Ehrhart polynomials provide the means to judge the quality of a data transformation and data distribution. We employ the usual condensed notation of Ehrhart polynomials, two examples are shown in Fig. 2. The notation $E(M,N) = [10,5]_N \cdot M$ abbreviates the distinction of the two cases $E(M,N) = 10 \cdot M$ if $N \bmod 2 = 0$, and $E(M,N) = 5 \cdot M$ if $N \bmod 2 = 1$, respectively. This evaluation scheme extends to higher dimensional *cyclic-coefficients*. Moreover, a polytope may have a set of Ehrhart polynomials.



access, local memory of processor P_0

access, local memory of processor P_1

P_i	number of local accesses G_i
P_0	$\frac{1}{4}(5 \cdot M \cdot N - [10,5]_N \cdot M - [6,5]_M \cdot N) + \begin{bmatrix} 12 & 10 \\ 6 & 5 \end{bmatrix}_{N,M}$)
P_1	$\frac{1}{4}(5 \cdot M \cdot N - [0,5]_N \cdot M - [4,5]_M \cdot N + \begin{bmatrix} 0 & 0 \\ 4 & 5 \end{bmatrix}_{N,M}$)

Fig. 2. Accesses to array `U` and default-distribution of array `U`

In this case its polynomials are defined for convex subsets of the *parameter space*, called the *validity domains*.

Now our two primitives for data distribution, *block aggregation* and *convolution of blocks*, have to be translated into terms of convex polytopes. We begin with the primitive that addresses block aggregation. It captures the effect of data distribution with equally sized *blocks*:

Let $\boldsymbol{x}' = f_l(\boldsymbol{x}) \in \mathcal{D}$ denote the index point accessed by reference R_l at iteration point $\boldsymbol{x} \in \mathcal{I}$. Because data distribution applies to dimension d the block identified by $x'_d = \lfloor x_d/B \rfloor$ is accessed at $\boldsymbol{x}$. The non-linear expression $\lfloor x_d/B \rfloor$ can be transformed into a linear expression at the expense of an additional unknown b and a constraint C_d. If we encounter an equation containing $\lfloor x_d/B \rfloor$, we replace it by a new free variable b and additionally constrain the admissible range of x_d to satisfy C_d: $B \cdot b \leq x_d < B \cdot (b+1)$.

We proceed with the primitive for the convolution of blocks. It captures the effect of a *cyclic* data distribution. Let therefore $b = f_l(\boldsymbol{x})$ denote the expression that evaluates to the block accessed by reference R_l at iteration point $\boldsymbol{x} \in \mathcal{I}_N$. Then this block b will be assigned to processor $b \bmod P$. The expression $b \bmod P$ must be transformed into a linear expression to fit into our linear framework. We replace it by $(b - P \cdot z)$, where z is a new free variable, and additionally constrain the expression b to satisfy C_b: $P \cdot z \leq b < P \cdot (z+1)$.

Thus we can use the primitives above to describe sets of iteration points without leaving the domain of convex polytopes.

3 Data Transformation

Our method to select data transformations and data distributions can be subdivided into two phases: The first phase computes a set of *optimal* transformations and distributions for each reference *separately*. This approach is guaranteed to succeed in the case of injective index functions [5], and optimality coincides with the absence of remote accesses. The second phase ranks these candidate transformations; it compares their associated Ehrhart polynomials considering *all* references in concert and selects the *best* transformation among all candidate transformations.

Fig. 3 shows the three main steps in the generation of a candidate transformation. During step one basis vectors are selected which span subspaces of the iteration space such that these subspaces are accessed by exactly one processor (a). Then these vectors are mapped to the index space, where they span subspaces accessed by at most one processor (b). Within the next step, an unimodular transformation is determined which makes these subspaces orthogonal to one of the axes (c) [5]. Then the resulting array is sliced into blocks along the selected axis. Each of these blocks is either unused or it is used by exactly one processor, which leads to a certain *utilization pattern* of memory blocks.

Finally, an offset is determined to shift the pattern such that it matches the data distribution. The result is a transformation which turns *all* accesses performed by *one* reference into local accesses.

3.1 Ranking References

We first show how to rank a data transformation with respect to a single reference R. We start with the convex polytope of the iteration space $\mathcal{I}$ and decompose it into subspaces $\mathcal{I}_p$, such that subspace $\mathcal{I}_p$ is executed by processor P_p. Thus

$$\mathcal{I} = \{\boldsymbol{x} \in \mathbb{Z}^k \mid \boldsymbol{A} \cdot \boldsymbol{x} + \boldsymbol{C} \cdot \boldsymbol{n} + \boldsymbol{b} \geq 0\}$$

for appropriate matrices $\boldsymbol{A}, \boldsymbol{C}$, and a vector $\boldsymbol{b}$. The Ehrhart polynomial I of $\mathcal{I}$ maps to the number of iterations to be executed by all parallel processors. In case of the multi-grid-example shown in Fig. 1 we obtain the Ehrhart polynomial

$$I(M, N) = M \cdot N - M - N + 1$$

Because we assume a cyclic mapping of iteration points in the parallel dimension j onto a total of P parallel processors, it follows that

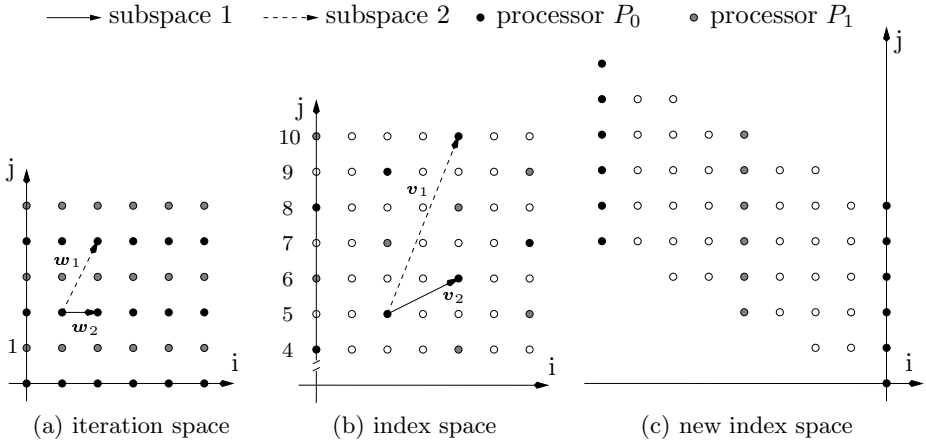
$$\mathcal{I}_p \;=\; \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{I} \;\wedge\; x_j \bmod P = p\}$$

Thus every set $\mathcal{I}_p$ also is a convex polytope, and the Ehrhart polynomial I_p of $\mathcal{I}_p$ exists. For example

$$I_p(M, N) = \frac{1}{2}(M \cdot N - M - \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}_{p,M} \cdot N + \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}_{p,M})$$

for our running example. To compute the index point within the transformed index space, we have to apply the transformation $\boldsymbol{x} \mapsto \boldsymbol{T} \cdot \boldsymbol{x} + \boldsymbol{T}_n \cdot \boldsymbol{n} + \boldsymbol{t}$ to the index point $f(\boldsymbol{x})$. Then we can investigate the block-cyclic distribution of the array in order to detect whether the array element $t(f(\boldsymbol{x})) = \boldsymbol{x}''$ that is accessed at iteration point $\boldsymbol{x}$ is a local array element of processor P_p. The according constraint C_p thus reads

$$C_p : \; B \cdot p \leq \pi_d(\boldsymbol{x}'') - (B \cdot P) \cdot z_2 < B \cdot (p + 1)$$



(a) iteration space (b) index space (c) new index space

Fig. 3. Steps in the generation of a candidate transformation

Note that $\pi_d(\boldsymbol{x})$ denotes the projection onto component x_d. Thus the set of iteration points $\mathcal{L}_p$ that spawn accesses to local array elements is equal to

$$\mathcal{L}_p = \{\boldsymbol{x} \in \mathcal{I}_p \mid C_p(\boldsymbol{x}) = true\}$$

We observe that the set $\mathcal{L}_p$ also is a convex polytope. The set $\mathcal{R}_p$ that spawns remote accesses is equal to the difference $\mathcal{R}_p := \mathcal{I}_p - \mathcal{L}_p$, which is not convex in the general case. Nevertheless, the polynomial of $\mathcal{R}_p$ exists and maps to the number of remote accesses. For reference R_1 = `U(I-J, J-1)` of our running example and $B = 1$, $t = id$, we obtain:

$$L_0(M,N) = \frac{1}{4}(M \cdot N - [2,1]_N \cdot M - [2,1]_M \cdot N + \begin{bmatrix} 4 & 2 \\ 2 & 1 \end{bmatrix}_{N,M})$$

Note that L_0 is a specialization of $L(M,N,p)$ with $p = 0$. Hence we can compute the polynomial $|L_0(M,N) - L_1(M,N)|$, which denotes the *imbalance* of remote memory accesses for the processors involved. In Sect. 6 we will provide further comments on the effect of such an imbalance.

3.2 Ranking Transformations

At this point we conclude that the construction of $\mathcal{L}_p$ as shown above allows to determine the local-remote access ratio of any reference R_l. We start with the iteration space as a parametric polytope, introduce a new parameter p to select iteration points executed by processor P_p and further restrict this polytope to contain only those points with local accesses. If we omit the parameter p that represents a processor P_p, we yield the desired polytope $\mathcal{L}_l$.

The volume of the polytope $\mathcal{L}_l$ is represented by an Ehrhart polynomial L_l, which serves as a metric to rank a transformation with respect to reference R_l. Thus the sum $L = \sum_l L_l$ along all references, complemented by the polynomial G that represents the total number of memory accesses, reflect the local-remote access ratio of an entire loop nest N and provides the desired metric to rank the combination of a linear data transformation and a block-cyclic data distribution.

We do not consider the case of multiple validity domains for the polytopes $\mathcal{L}_l$. In this case, one would need more information regarding the parameters of the program in order to choose the right validity domain.

3.3 Final Selection

To finally select a transformation that performs well for the entire loop nest, we *symbolically compare* the Ehrhart polynomials of different transformations and keep the best among all candidate transformations. Given a finite set of transformations, which will be constructed in Sect. 4, we compare the polynomials L of these transformations. Without further knowledge on program parameters, we first simplify periodic coefficients, i.e., we replace them by their arithmetic average, we unify program parameters, and then we compare the coefficients

$$E_M(M,N,p) = \frac{1}{4} \cdot (5 \cdot M \cdot N - \left[\begin{smallmatrix} 10 & 5 \\ 0 & 5 \end{smallmatrix}\right]_{p,N} \cdot M - \left[\begin{smallmatrix} 6 & 5 \\ 4 & 5 \end{smallmatrix}\right]_{p,M} \cdot N + \left[\left[\begin{smallmatrix} 12 & 10 \\ 6 & 5 \end{smallmatrix}\right]_{N,M}, \left[\begin{smallmatrix} 0 & 0 \\ 4 & 5 \end{smallmatrix}\right]_{N,M} \right]_p)$$

$$E_N(M,N,p) = \frac{1}{4} \cdot (6 \cdot M \cdot N - \left[\begin{smallmatrix} 12 & 6 \\ 0 & 6 \end{smallmatrix}\right]_{p,N} \cdot M - 6 \cdot N + \left[\begin{smallmatrix} 12 & 6 \\ 0 & 6 \end{smallmatrix}\right]_{p,N})$$

Fig. 4. Ehrhart polynomials associated with default-distributions

of the polynomials in descending order of their degree. Fig. 4 shows Ehrhart polynomials E_M and E_N, of our running example for 2 parallel processors. The polynomial E_M represents a default-distribution along the M-axis, whereas E_N represents a distribution along the N-axis. Both polynomials map to the number of local accesses, which implies that the data distribution along the N-axis is superior to that along the M-axis, because $\frac{6}{4} \cdot M \cdot N > \frac{5}{4} \cdot M \cdot N$ for significant problem parameters M, N. Moreover, these terms do not depend on the processor coordinate p. The distribution of array `U` along the M-axis (E_M) is illustrated in Fig. 2.

4 Enumerating Transformations

Although the result of the preceding subsection allows to rank a data transformation or a given program formulation and therefore provides a precious result by itself, we are interested in enumerating *candidate transformations* that provide locality in order to pick the best one by means of metric L.

We first search for $n-1$ vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{n-1}$ within the n-dimensional iteration space $\mathcal{I}$ that span disjoint subspaces of dimension $n-1$. If $\mathcal{I}_{\boldsymbol{o}}$ is such a subspace identified by some origin $\boldsymbol{o}$, i.e., $\mathcal{I}_{\boldsymbol{o}} = \{\boldsymbol{x} \in \mathcal{I} \mid \boldsymbol{x} = \boldsymbol{o} + \sum_{i=1}^{n-1} k_i \cdot \boldsymbol{w}_i, k_i \in \mathbb{Q}\}$, the following implication should hold:

$$\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{I}_{\boldsymbol{o}} \;\Rightarrow\; x_p \bmod P = x'_p \bmod P$$

Thus we intend to assign a subspace $\mathcal{I}_{\boldsymbol{o}}$ to a unique processor. In terms of generating vectors $\boldsymbol{w}_i$ we require for arbitrary iteration points $\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{I}$ that

$$\boldsymbol{x}' = \boldsymbol{x} + \sum_{i=1}^{n-1} (k_i \cdot \boldsymbol{w}_i) \;\Rightarrow\; x_p \bmod P = x'_p \bmod P \tag{1}$$

Theorem 4.1 gives a sufficient condition that allows for the selection of $\boldsymbol{w}_i$. Note that below p denotes the parallel dimension of the loop nest:

Theorem 4.1 Let $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{n-1}$ denote linearly independent generating vectors from $\mathbb{Z}^n$ such that $\boldsymbol{w}_i = (w_{1,i}, \ldots, w_{n,i})^t$. Then these vectors $\boldsymbol{w}_i$ satisfy constraint (1) above, if:

i) $\forall i:\ \gcd(w_{1,i}, \ldots, w_{n,i}) = 1$ and
ii) $\forall j:$ there exists at most one i: $w_{j,i} \neq 0$ and
iii) $\forall i:\ w_{p,i} \bmod P = 0$

■

The following implication applies to the index space:

Theorem 4.2 Let $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{n-1}$ denote linearly independent vectors from $\mathbb{Z}^n$ satisfying constraint (1). Let $f(\boldsymbol{x}) = \boldsymbol{F} \cdot \boldsymbol{x} + \boldsymbol{F}_n \cdot \boldsymbol{n} + \boldsymbol{f}$ denote an index function having a square and invertible access matrix $\boldsymbol{F}$. Let further $\boldsymbol{v}_i = \boldsymbol{F} \cdot \boldsymbol{w}_i$ denote images of vectors $\boldsymbol{w}_i$ under the linear part of the index function f. Then:

$$f(\boldsymbol{x}') = f(\boldsymbol{x}) + \sum_{i=1}^{n-1} k_i \cdot \boldsymbol{v}_i \ \Rightarrow\ x_p \bmod P = x'_p \bmod P$$

■

Thus vectors $\boldsymbol{v}_i$ mentioned in Theorem 4.2 span subspaces of the index space which are accessed by at most one processor. The formulation of Theorem 4.2 implies that only those index points are involved which have a counterpart in the iteration space. Fig. 3 (a) illustrates generating vectors $\boldsymbol{w}_i$, whereas Fig. 3 (b) illustrates vectors $\boldsymbol{v}_i$ of both theorems above.

It remains to compute the transformation $\boldsymbol{T}$. Let $\boldsymbol{v}'_i = \boldsymbol{T} \cdot \boldsymbol{v}_i$ denote the image of $\boldsymbol{v}_i$ under transformation $\boldsymbol{T}$. If there exists an index j such that all vectors $\boldsymbol{v}'_i$ have a 0-entry in their jth component, then it is efficient to distribute along this dimension. We compute such a transformation $\boldsymbol{T}$ by application of Gaussian elimination combining vectors $\boldsymbol{v}_i$ into a matrix of n rows and $n-1$ columns. Its rank is $n-1$, because vectors $\boldsymbol{v}_i$ are linearly independent. Now we eliminate the entries of the last row by Gaussian elimination and place that row on a desired level. The elimination algorithm emits the transformation $\boldsymbol{T}$.

5 Data Distribution

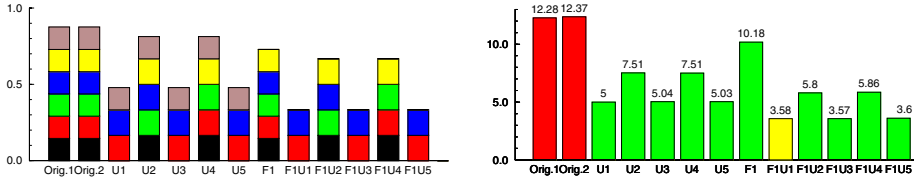
So far, we have computed the transformation matrix. It remains to determine the distribution parameters. This is done in two steps. First we determine the resulting utilization pattern, then an offset for the transformation function is computed

5.1 The Utilization Pattern

First, we introduce an important prerequisite, the notion of an array slice:

Definition 5.1 A *slice* of an array `X` with respect to a dimension d is a subspace of the index space $\mathcal{D}_{\mathtt{X}}$ which results from the evaluation of a fixed coordinate in dimension d. We say that a processor *owns* a slice $\mathcal{S}_k$, if it accesses elements within the slice but no other processor does access its elements. ■

The utilization pattern consists of slices that are owned by a specific processor and of unused slices. Using '*' to denote unused slices, we can describe the



(a) predicted performance (b) observed performance

Fig. 5. Estimated and observed performance of several multi-grid versions

pattern for the candidate transformation in Fig.3 (c) as '0,*,*,*,1,*,*,*'. We have a slice owned by processor 0, followed by three unused slices, followed by a slice owned by processor 1, etc. This pattern repeats cyclically. The blocks with unused slices always have the same size [5], in this case three.

Therefore, a simple iterative algorithm can be used to compute the pattern. Three cases must be distinguished: In the first case, the pattern fits immediately to a cyclic distribution, like the pattern '0, *, 1, *, 2, *' fits in the case of 3 processors. In the second case, a reversal transformation must be applied to the array to make the pattern fit to a distribution, which for example is true for the pattern '2, *, 1, *, 0, *'. In the third case, the pattern cannot be mapped to the distribution. Hence these transformations are removed from the set of valid candidates.

5.2 The Offset

Up to now, we just know the portion $\boldsymbol{T}$ of the transformation $t(\boldsymbol{x}) = \boldsymbol{T} \cdot \boldsymbol{x} + \boldsymbol{T}_n \cdot \boldsymbol{n} + \boldsymbol{t}$. The offset $\boldsymbol{T}_n \cdot \boldsymbol{n} + \boldsymbol{t}$ should be chosen such that every processor accesses those slices that it owns itself. This property is satisfied for the index function without offset. We have to determine the offset of transformation t such that it compensates the offset of f.

In the context of our simple processor-mapping the iteration point $\mathbf{0}$ is executed by processor P_0. Moreover, slice $\mathcal{S}_0$ is always owned by processor P_0. Thus it is sufficient to choose the offset such that iteration point $\mathbf{0}$ causes an access to slice S_0. Starting with $t(f(\mathbf{0})) = \mathbf{0}$ we yield $\boldsymbol{T}_n = -\boldsymbol{T} \cdot \boldsymbol{F}_n \ \wedge \ \boldsymbol{t} = -\boldsymbol{T} \cdot \boldsymbol{f}$.

6 Results and Conclusion

We have applied our techniques to a multi-grid application and investigated their impact on its execution time. Fig. 5 (a) shows ratios of remote accesses to be executed, whereas Fig. 5 (b) shows the execution time in seconds we observed on a SCI-cluster of 8 nodes and a matrix size of 512×512.

The bars from left to right represent two default versions, of which the first had all data on one node, while the second one employed a default distribution provided by the shared memory interface, 5 versions that have been optimized

for one of the 5 references to array `U`, one that has been optimized for array `F` and combinations optimized for `U` *and* `F`.

Stacked bars in sub-figure (a) are subdivided to indicate contributions of single references. Absent hatch patterns indicate that the according references do not contribute remote accesses. The real execution time observed on the SCI-cluster shown in sub-figure (b) is seen to conform strikingly well to that estimated by inspection of the remote-to-local ratio. The small deviation is due to imbalances in remote references across processors. These cause some processors to stall at synchronization barriers [5]. Compared to the worst default-parallel version, which takes 12.37s to complete, the selected version of the multi-grid application needs only 3.58s. It has been optimized for arrays `U` and `F` (bar F1U1) in concert. This gives an improvement of approx. 3.5.

From this example we conclude that our novel techniques significantly boost the performance of regular programs on NUMA-architectures. They are suited to improve data distributions of explicitly parallel programs and to guide data distribution optimizations of parallelizing compilers. Future work will address a broader set of application programs and more workstations.

Acknowledgments: We are grateful to Philippe Clauss who provided the initial implementation of Ehrhart polynomials.

References

[1] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *PPOPP 95, Santa Clara, CA USA*, pages 166–178, June 1995.

[2] E. Ayguade, J. Garcia, M. Girones, and J. Labarta. Detecting and using affinity in an automatic data distribution tool. *Lecture Notes in Computer Science*, 892:61–75, 1995.

[3] P. Clauss. Counting Solutions to Linear and Nonlinear Constraints through Ehrhart Polynomials. In *ACM Int. Conf. on Supercomputing*. ACM, May 1996.

[4] P. Feautrier. Compiling for massively parallel architectures: a perspective. *Microprogramming and Microprocessors*, 41:425–439, 1995.

[5] F. Heine. Optimierung der Datenverteilung für SCI-gekoppelte Workstation-Cluster. Master's thesis, Universität-GH Paderborn, May 1999.

[6] C. H. Koelbel. *The High Performance Fortran handbook*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, Jan. 1994.

[7] U. Kremer. *Automatic Data Layout for Distributed Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, Oct. 1995.

[8] C. Lengauer. Loop parallelization in the polytope model. Technical report, Universität Passau, Fakultät für Mathematik und Informatik, 1993.

[9] A. Slowik. *Volume Driven Selection of Loop and Data Transformations for Cache-Coherent Parallel Processors*. PhD thesis, Universität-GH Paderborn, 1999. To appear (submitted).

[10] D. K. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, Intitut de Recherche en Informatique et Systèmes Aléatoires, Dec. 1993.

[11] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada*, pages 30–44, June 1991.