Access Descriptor Based Locality Analysis for Distributed-Shared Memory Multiprocessors

A.G. Navarro
R. Asenjo
E.L. Zapata
D. Padua

September 1999
Technical Report No: UMA-DAC-99/06

Published in:

IEEE Int'l. Conf. on Parallel Processing (ICPP'99)
Aizu-Wakamatsu, Japan, pp. 86-94, September 21-24, 1999

University of Malaga
Department of Computer Architecture
C. Tecnologico • PO Box 4114 • E-29080 Malaga • Spain

Access Descriptor based Locality Analysis for Distributed-Shared Memory Multiprocessors

Angeles G. Navarro, Rafael Asenjo, Emilio L. Zapata
Dept. de Arquitectura de Computadores
Univ. de Málaga
{angeles,asenjo,ezapata}@ac.uma.es

David Padua
Dept. of Computer Science
Univ. of Illinois at Urbana-Champaign
padua@uiuc.edu

Abstract

Most of today's multiprocessors have a Distributed-Shared Memory (DSM) organization, which enables scalability while retaining the convenience of the shared-memory programming paradigm. Data locality is crucial for performance in DSM machines, due to the difference in access times between local and remote memories. In this paper, we present a compile-time representation that captures the memory locality exhibited by a program in the form of a graph known as Locality-Communication Graph (LCG). *In the LCG, each node represents a DO loop nest which can have at most one level of parallelism. Not all loops need to be represented within a node and, therefore, the LCG may contain cycles. Our representation works whether the loops represented by the nodes are perfectly nested or not, and the subscript expressions and loop limits can be affine or non-affine expressions of the loop indices. The LCG provides essential information that a parallelizing compiler can use to automatically choose a good iteration/data distribution and to schedule the communication operations required during program execution.*

1. Introduction

In this paper, we discuss a new compiler internal representation of locality in *Distributed-Shared Memory* (*DSM*) multiprocessors. We assume that the program contains a collection of DO loop nests (henceforth called *loop nests* for simplicity) where at most one level of parallelism is exploited in each nest. IF statements, WHILE loops, DO loops and other constructs control the execution of the loop nests. Because access time to remote memories is usually much higher than to local memories, data locality enhancement is clearly a key issue for performance. By locality enhancement we mean selecting an efficient data distribution in which, whenever possible, the data are placed in the local memories of the processors needing them. We reported earlier some experimental results that clearly show the significant impact of locality analysis on the efficiency of automatic parallelization [10]. Although, there have been several research projects on locality enhancement for DSMs, as discussed below, there is still room for much improvement and, for that reason, we decided to initiate this study, the first results of which are reported in this paper.

As part of this project, we have developed techniques to schedule parallel iterations and distribute the shared arrays across the local memories to minimize the parallel execution overhead resulting from load unbalance and communications. When remote accesses are unavoidable, our techniques can identify them to estimate execution costs and generate communication primitives. Our techniques are based on the *Linear Memory Access Descriptor (LMAD)* introduced in [9] which can be accurately computed in relatively complex situations, such as array reshaping resulting from subroutine calls and non-affine expressions in subindex expressions and loop bounds. One advantage of LMADs is that they can be computed inter-procedurally by applying essentially the same approach used to compute their values intra-procedurally.

Our technique uses a *Locality-Communication Graph* (*LCG*) that reflects the locality and communication patterns of a parallel program. The program is divided into *phases*, each being a DO loop nest with at most one parallel loop. These loop nests do not have to be perfectly nested. The LCG contains a directed, connected graph for each array in the program. Each node in these graphs corresponds to a phase accessing the array represented by the graph. Notice that these phases do not have to include outermost DO loops. That is, the phases could be inside one or more DO loops. The nodes are connected according to the program control flow. The connected graphs are not necessarily trees because not all loops are part of a phase. The LCG supports all the functions of our technique, including program optimization and automatic generation of communication operations.

There are a variety of approaches to automatically solve the iteration/data distribution problem [4], [5], [3], [6], [1]. However, as mentioned above, we believe that they suffer limitations that should be resolved, if possible. With these approaches, the array dimensions must be static and known at compile time, and array reshaping, therefore, is not addressed. In addition, these approaches require that the loops be perfectly nested and that the subscript expressions and loop limits be affine expressions of the loop indices. Furthermore, they do not satisfactorily solve the BLOCK-CYCLIC distribution and do not consider the *reverse distribution* case. In contrast, as mentioned above, our techniques can handle array reshaping and, as a result, can be directly applied inter-procedurally. Our techniques can handle non-affine access functions and can compute BLOCK-CYCLIC distributions, the most general type of distributions among those commonly used today. And our approach covers the reverse distribution as well.

The organization of the paper is as follows. In Section 2, we describe the main concepts in our model: the *array reference descriptor*, the *phase descriptor*, and some operations to simplify their representation. Section 3 introduces *iteration descriptors*, which describes the memory region accessed by a parallel loop iteration. In Section 4, we address the locality analysis (*intra-phase and inter-phase localities*) that generates the annotated LCG to reflect the memory locality that can be exploited in the code. In addition, we present two applications of the LCG. Due to space limitations, we have omitted formal proofs, but the theorems and statements are quite simple and we expect that the reader will have no difficulty believing their correctness.

2. Array reference descriptor and phase descriptor

To generate descriptors of memory access patterns we assume that loops have been normalized and that all arrays have been converted into one-dimensional arrays as traditionally done by conventional compilers.

The *array reference descriptor* (*ARD*) of the s-th reference to array X in phase F_k is defined as: $\mathcal{A}_s^k(X, \vec{i}_k) = \left(\vec{\alpha}_s, \vec{\delta}_s, \vec{\lambda}_s, \tau_s\right)$. Here, $\vec{i}_k$ is the vector of indices of the loop nest F_k. There is one (and only one) element in ARD vectors $\vec{\alpha}_s$, $\vec{\delta}_s$, and $\vec{\lambda}_s$ for each loop index from the nest. Let us assume that the subscript expression of the sth reference to X is ϕ_s. Element j of $\vec{\delta}_s$ contains the absolute value of the *stride* of ϕ_s for $i_k(j)$, the jth loop index in loop nest F_k. This stride is the difference between the values ϕ_s obtained by evaluating it at two consecutive values of $i_k(j)$. The jth element of $\vec{\lambda}_s$ is 1 if the jth stride is positive and -1 otherwise. The jth element of $\vec{\alpha}_s$ is the difference between the values of ϕ_s with $i_k(j)$ evaluated at the limits of the jth loop divided by the jth element of the stride. The value of vector $\vec{\alpha}_s$ multiplied element-by-element by the stride is the *span* defined in [9].

For example, consider the two references to X shown in Figure 1 containing a loop nest of TFFT2 from the NASA benchmarks. The $P = 2^p$ and $Q = 2^q$ input parameters are constant. Initially, we do not take into account the different kinds of accesses (read/write). Therefore, for this loop nest, there are two non-affine subscript expressions: $\phi_1 = (2 \cdot P \cdot I + 2^{L-1} \cdot J + K)$ and $\phi_2 = (2 \cdot P \cdot I + 2^{L-1} \cdot J + K + P/2)$. In addition, in this example, the upper bound of J loop and the upper bound of the K loop are non-affine expressions dependent on the L index. The ARDs for the two X references are shown in Figure 2.

```
1.  F3:  doall I = 0 to Q − 1
2.           do L = 1 to p
3.             do J = 0 to P · 2^−L − 1
4.               do K = 0 to 2^L−1 − 1
5.                 ... X(2 · P · I + 2^L−1 · J + K) ...
6.                 ... X(2 · P · I + 2^L−1 · J + K + P/2) ...
7.                                  ⋮
8.               enddo
9.             enddo
10.          enddo
11.        enddo
```

Figure 1. X in phase F_3 of TFFT2

The ARD $\mathcal{A}_1^3(X, \vec{i}_3)$ from Figure 2 contains three vectors and one scalar: $\vec{\alpha}_1 = (Q, (P-2) \cdot 2^{-L} + 1, P \cdot 2^{-L}, 2^{L-1})$ containing the span divided by the stride for each loop index. The vector $\vec{\delta} = (2 \cdot P, J \cdot 2^{L-1}, 2^{L-1}, 1)$ contains the *stride* for each loop index of ϕ_1. The vector $\vec{\lambda} = (1, 1, 1, 1)$ contains the stride signs. Finally, $\tau_1 = 0$ is the *offset* of the ϕ_1 access function with respect to the basis position in the X array (which is 0).

Once we have obtained the ARDs for array X in a phase, say F_k, a *phase descriptor* (*PD*) representing all the elements of X accessed by a phase can be computed. The general form of a phase descriptor is $\mathcal{P}^k(X, i) = \bigcup_{j=1:m} \mathcal{A}_j^k(X, \vec{i}_k) = \mathcal{A}_1^k(X, \vec{i}_k) \cup \mathcal{A}_2^k(X, \vec{i}_k) \cup \ldots \mathcal{A}_m^k(X, \vec{i}_k)$. This PD has the form $\vec{\mathcal{P}}^k(X, \vec{i}_k) = \left(A, \vec{\delta}, \Lambda, \vec{\tau}\right)$ and represents $\mathtt{m} \geq 1$ of the occurrences of X within the phase. A, and Λ are matrices of dimension $\mathtt{m} \times \mathtt{n}$ which represent the spans divided by strides, and the stride signs, respectively. The vector $\vec{\delta}$ represents the vector of all possible strides for all levels of nesting and all occurrences of X. The value $\mathtt{n}$ is the dimension of vector $\vec{\delta}$. Each row of the matrices represents the $\vec{\alpha}$ and $\vec{\lambda}$ vectors, respectively, for each occurrence of X represented by $\mathcal{A}_j^k$. The vector $\vec{\tau}$ contains the offset for each occurrence. In the rest of the paper, for the sake of simplicity, we will assume that all the strides have a positive sign so that we can avoid the matrix Λ. For the code example in Figure 1, the resultant

$$\mathcal{A}_1^3(X,\vec{\imath}_3) = \left(\left(Q, \quad (P-2)\cdot 2^{-L}+1, \quad P\cdot 2^{-L}, \quad 2^{L-1}\right), \begin{pmatrix} 2\cdot P \\ J\cdot 2^{L-1} \\ 2^{L-1} \\ 1 \end{pmatrix}, (1, \quad 1, \quad 1, \quad 1), 0 \right)$$

$$\mathcal{A}_2^3(X,\vec{\imath}_3) = \left(\left(Q, \quad (P-2)\cdot 2^{-L}+1, \quad P\cdot 2^{-L}, \quad 2^{L-1}\right), \begin{pmatrix} 2\cdot P \\ J\cdot 2^{L-1} \\ 2^{L-1} \\ 1 \end{pmatrix}, (1, \quad 1, \quad 1, \quad 1), P/2 \right)$$

Figure 2. ARDs for X in phase $\mathbf{F_3}$ of TFFT2

PD for the array X is shown in Figure 3(a). In the PD for an array X, the stride vector $\vec{\delta}$ contains some strides associated with the parallel loop and some other strides associated with the sequential ones. Recall that each phase contains at most one parallel loop. For example, in Figure 3(a) the first component of stride vector $\delta_1 = 2 \cdot P$ is associated with the **doall** loop. The remaining components of the stride vector, which are associated with the sequential loops, appear in loop order: ($\delta_2 = J \cdot 2^{L-1}; \delta_3 = 2^{L-1}; \delta_4 = 1$).

2.1. Phase descriptor transformations

A PD can be simplified by applying transformations such as *stride coalescing* and *access descriptor union*. As mentioned above, a program is represented here by a collection of control flow graphs, one for each array in the program. The nodes of these graphs correspond to DO loop nests containing accesses to the array associated with the control flow graph. The access representation of array X can be further improved by avoiding the redundancies that may arise between PDs corresponding to different phases. In order to accomplish this task, we also need to carry out other simplification operations: *descriptors homogenization* and *offset adjustment* operations. All these operations are briefly described next. Detailed descriptions can be found in [7].

An element of the stride vector $\vec{\delta}$ may be safely deleted when it is a multiple of another and the *span* corresponding to the former is greater than the span of the latter. Our goal in the *stride coalescing* operation is to reduce the number of indices involved in the PD. Given a PD for an array X, $\mathcal{P}^k(X,\vec{\imath}_k) = \left(A, \vec{\delta}, \vec{\tau}\right)$, the *stride coalescing* operation removes *redundant stride* columns [7] from $\vec{\delta}$ and the corresponding columns from matrix A without losing the access information. The basis of the stride coalescing operation was introduced in [9].

An important case arises when the access functions are non-linear and/or the loop bounds are non-constant: the PDs are non-constant. This means that either some coefficients α_{ij} from matrix A or some δ_j from $\vec{\delta}$ are a function of the index loop variables. For example, consider the loop nest descriptor in Figure 3(a), which is expressed in terms of the loop indices L and J. The element δ_3 is a multiple of element δ_4. Therefore, we can remove δ_3 and the third column of A. However, this implies updating the fourth column of A. Figure 3(b) shows the new phase descriptor that results from the application of coalescing. In this PD, element δ_2 is again redundant with respect to δ_4. By applying the coalescing algorithm again, we get the final PD, which is shown in Figure 3(c).

In some cases, it may be possible to find for two ARDs a PD representation containing a single $\mathcal{A}_j^k$ term due to misalignments in the access functions. For example, this is the case when two ARDs describe the same access pattern but one of the regions they represent is shifted relative to the other (we say two ARDs have the same access pattern when they have a *similar* [7] size vector, $\vec{\alpha}$, and the same stride vector, $\vec{\delta}$). The ARDs of these access functions can be aggregated in such a way that they can be represented as a single row within a PD. This single row represents the union of the two ARDs. This is the goal of the access descriptor union operation. For example, applying access descriptors union to $\mathcal{A}_1^3(X,\vec{\imath}_3)$ and $\mathcal{A}_2^3(X,\vec{\imath}_3)$ ARDs from Figure 3(c) produces the PD shown in Figure 3(d).

There are several cases in which two PDs describe the same access pattern, but one of the regions is shifted with respect to the other. Since we are interested in the whole region of an array X accessed with the same access patterns, we can aggregate both PDs into one which represents the union of such regions. In order to do this, we apply a union operation similar to the one described above for ARDs. Both PDs are replaced by the result of the union operation. This is the *descriptor homogenization* operation.

The goal of the *offset adjustment* operation is to express the region accessed for each PD with respect to τ_{min}, which is the base position for array X. If the PD of F_k does not contain the minimum offset, we rewrite the smaller offset of that nest (τ_1^k) in terms of τ_{min}. For this, we define the *adjust distance* operation, R^k, as $R^k = \left\lfloor \frac{\tau_1^k - \tau_{min}}{\delta_1^k} \right\rfloor$.

3. Iteration descriptor

Consider a loop nest with a single parallel loop. To describe the elements of array X accessed by one iteration, say i, of the parallel loop in phase F_k, we use the *iteration descriptor* (*ID*) whose general form is $\mathcal{I}^k(X,i) = \bigcup_{j=1:m} \mathcal{I}_j^k(X,i) = \mathcal{I}_1^k(X,i) \cup \mathcal{I}_2^k(X,i) \cup \ldots \mathcal{I}_m^k(X,i)$,

$$\vec{\mathcal{P}}^3(X,\vec{\imath}_3) = \left\{ \begin{pmatrix} Q & (P-2)\cdot 2^{-L}+1 & P\cdot 2^{-L} & 2^{L-1} \\ Q & (P-2)\cdot 2^{-L}+1 & P\cdot 2^{-L} & 2^{L-1} \end{pmatrix}, \begin{pmatrix} 2\cdot P \\ J\cdot 2^{L-1} \\ 2^{L-1} \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ P/2 \end{pmatrix} \right\}$$

(a)

$$\vec{\mathcal{P}}^3(X,\vec{\imath}_3) = \left\{ \begin{pmatrix} Q & P/2 \\ Q & P/2 \end{pmatrix}, \begin{pmatrix} 2\cdot P \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ P/2 \end{pmatrix} \right\}$$

(c)

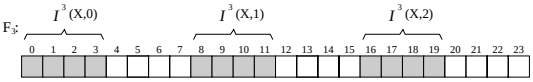
$$\vec{\mathcal{P}}^3(X,\vec{\imath}_3) = \left\{ \begin{pmatrix} Q & (P-2)\cdot 2^{-L}+1 & P/2 \\ Q & (P-2)\cdot 2^{-L}+1 & P/2 \end{pmatrix}, \begin{pmatrix} 2\cdot P \\ P/2-2^{L-1} \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ P/2 \end{pmatrix} \right\}$$

(b)

$$\vec{\mathcal{P}}^3(X,\vec{\imath}_3) = \left\{ \begin{pmatrix} Q & P \end{pmatrix}, \begin{pmatrix} 2\cdot P \\ 1 \end{pmatrix}, (0) \right\}$$

(d)

Figure 3. (a) PD of X in $\mathbf{F_3}$; (b) after removing δ_3; (c) after removing δ_2; (d) after the access descriptor union



$$\mathcal{I}^3(X,i) = \left\{ (4), (1), (8\cdot i) \right\}$$

Figure 4. IDs for X associated with parallel iterations i=0, i=1 and i=2 of $\mathbf{F_3}$

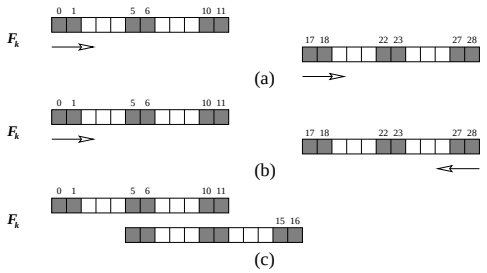
where $\mathcal{I}^k(X,i) = \left\{ B, \vec{\delta}_B, \vec{\tau}_B(i), \vec{\Delta} \right\}$. The ID represents a super-set of all elements of array X accessed in such a parallel iteration. A simple way to compute an ID is to manipulate the corresponding PD as follows: B is computed from A by removing the sizes associated with the parallel loop; $\vec{\delta}_B$ represents the stride associated with the sequential loops, and $\vec{\tau}_B(i)$ is the extended offset vector. $\vec{\tau}_B(i)$ contains, for $j = 1 : m$, $\tau_B(j,i) = \tau_j + i \cdot \delta_P(j)$, where $\delta_P(j)$ is the stride associated with the parallel loop of the jth occurrence. $\tau_B(j,i)$ points to the first memory position of the sub-region accessed by the $\phi_j(\vec{\imath}_k)$ access function in the i-th iteration of the parallel loop. $\vec{\Delta}$ is originally a null vector that will be computed next. Figure 4 (with $\vec{\Delta}$ omitted) shows a graphic representation of the IDs associated with each iteration of the parallel loop of F_3 for the array X of the TFFT2 example of Figure 2 when $Q = 3$ and $P = 4$. The shaded memory positions of X represent the data sub-regions described by each $\mathcal{I}^3(X,i)$. The $\vec{\Delta}$ component of each ID term can be used to take advantage of *storage symmetry*. It is used to represent as a single term several $\mathcal{I}^k_j(X,i)$ terms from the original form of the ID. For each type of storage symmetry we define a *distance*: a) *Shifted storage*: With a shifted storage distance, two array sub-regions with the same access pattern but shifted can be represented by a single ID term. In this situation, we define the *shifted storage distance*, Δ_d; b) *Reverse storage*: this represents two array sub-regions that are accessed with a reverse access pattern (this means that one access function is increasing and the other is decreasing with respect to the parallel loop index). In this case, we define the *reverse storage distance* Δ_r; c) *Overlapping storage*: this represents two array sub-regions which are partially overlapped. In this case, we calculate the *overlapping distance*, Δ_s. In Figure 5, we show some examples of storage symmetry and their corresponding IDs and distances. The three kinds of storage symmetry we have just defined are not exclusive to each other, and they can appear at the same phase descriptor.

4. Memory access locality analysis

In developing our analysis and transformation algorithms, we have assumed: i) The total number of processors, H, to be involved in the execution of the program is known at compile time; and ii) The iterations of each parallel loops are statically distributed between the H processors involved in the execution of the code following a BLOCK-CYCLIC pattern.

The first part of the algorithm is to identify when it is possible to distribute iterations and data in such a way that all accesses to an array X are to local memory. We call this part of the algorithm *memory access locality analysis*. Obviously, is not always possible to find a static iteration/data distribution such that all accesses required by a given processor are local. In such cases, a remote access (communication) to the memory of the processor owning the required data is necessary. Actually, the communication operation is implemented in our model via a *put* [2] operation. This operation is an example of what is known as single-sided communication primitives. In our approach, the data distribution may change dynamically from phase to phase. In fact, with our locality analysis framework, it is possible to identify sets of consecutive phases that cover the same data sub-region of an array X for a number of parallel iterations scheduled on each phase. For this set of phases, we can select a static data distribution for X such that all accesses to this array are going to be local.

In this Section, we analyze the conditions that must hold to ensure the memory access locality for each phase (*intra-phase locality*) and between phases (*inter-phase locality*). The goal of this analysis is to compute the LCG. We assign an attribute to each node of the LCG identifying the type



$$\mathcal{I}^k(X,i) = \Big((2), (1), (5\cdot i), (\Delta_d = 17) \Big)$$
$$\mathcal{I}^k(X,i) = \Big((2), (1), (5\cdot i), (\Delta_r = 27) \Big)$$
$$\mathcal{I}^k(X,i) = \Big((2), (1), (5\cdot i), (\Delta_s = 5) \Big)$$

Figure 5. (a) Shifted st. $\Delta_{\mathbf{d}} = 17$; (b) Reverse st. $\Delta_{\mathbf{r}} = 27$; (c) Overlapping st. $\Delta_{\mathbf{s}} = 5$

of memory access for that array in the corresponding phase. When the memory access for an array X in a phase is write only, the associated node in the X graph is marked with the attribute `W`; when the memory access is read only, the attribute is `R`; and, finally, for read and write accesses, the attribute is `R/W`. A special case arises when array X is privatizable in a phase (we restrict the definition of privatizable array given in [10] because we consider the value of X is not lived after the execution of F_k): the corresponding node is marked with attribute `P`. Figure 6 shows the LCG for a fragment of our motivating TFFT2 example. This LCF comprises two graphs: one for array X and one for array Y. Each graph contains 8 nodes (phases). As noted earlier, the nodes are labeled with an access attribute: `W`, `R`, `P`, and `R/W`. On each graph, the edges connecting nodes are also annotated with additional labels: `L`, which means that is possible to exploit memory access locality between the connected nodes, and `C`, which means that it is not possible to assure memory access locality. This `C` label stands for "communication", because the lack of memory access locality implies non-local accesses or, in other words, the necessity of communication between processors. In these cases, the communication operations will be placed just after the execution of the source connected phase and before the execution of the drain connected phase. The locality labels, `L` and `C`, will be determined at the end of the locality analysis.

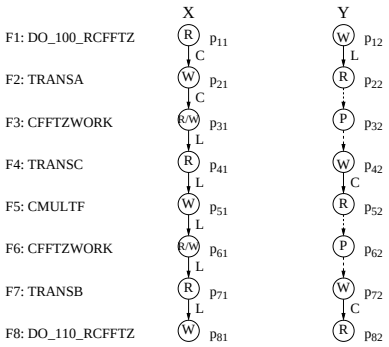


Figure 6. LCG for a TFFT2 code section

4.1. Intra-phase locality

Let $\mathcal{I}^k(X,i)$ be the ID for array X in the i-th parallel iteration of phase F_k. Let us assume that this i-th iteration is scheduled in the processor PE. We say that all *accesses to X are local* to processor PE if the region described by $\mathcal{I}^k(X,i)$ belongs to processor PE. From here, we propose an intuitive idea.

Theorem 1. Let us suppose that the i-th parallel iteration of phase F_k is scheduled in processor PE. The sufficient condition to ensure that all accesses to array X in phase F_k are local to processor PE (**intra-phase locality**) is that this processor local memory holds $\mathcal{I}^k(X,i)$ and a) X is privatizable, or b) X is non-privatizable and there is not overlapping storage for array X in phase F_k, or c) X is non-privatizable, there is overlapping storage for X in F_k and accesses to array X in that phase are only reads. □

Theorem 1 sets three different situations:

a) Array X is privatizable. If the local memory of each processor contains a copy of the region of the privatizable array X accessed in the corresponding parallel iteration (which is described in $\mathcal{I}^k(X,i)$), we can guarantee that all accesses to X in the phase F_k are local, as we show in Figure 7(a) for array Y. *Array replication* is a particular case of this situation.

b) Array X is non-privatizable and there is not overlapping storage for array X in phase F_k ($\nexists\ \Delta_s$). In this case, when the local memory of processor PE (which executes the i-th parallel iteration of phase F_k) contains the region described in $\mathcal{I}^k(X,i)$, all the access functions in this phase only generate local accesses to the memory of processor PE, as we see in Figure 7(b) for array Y.

c) Array X is non-privatizable and there is overlapping storage for X in F_k ($\exists\ \Delta_s$), then $\mathcal{I}^k(X,i)$ contains overlap sub-regions that are replicated in other processor memories. This is shown in Figure 7(c). In this case, when accesses are reads only, it is not necessary to update the replicated sub-regions (overlap sub-regions). Therefore, no communications will be necessary, and all accesses will be local to the memory of processor PE.

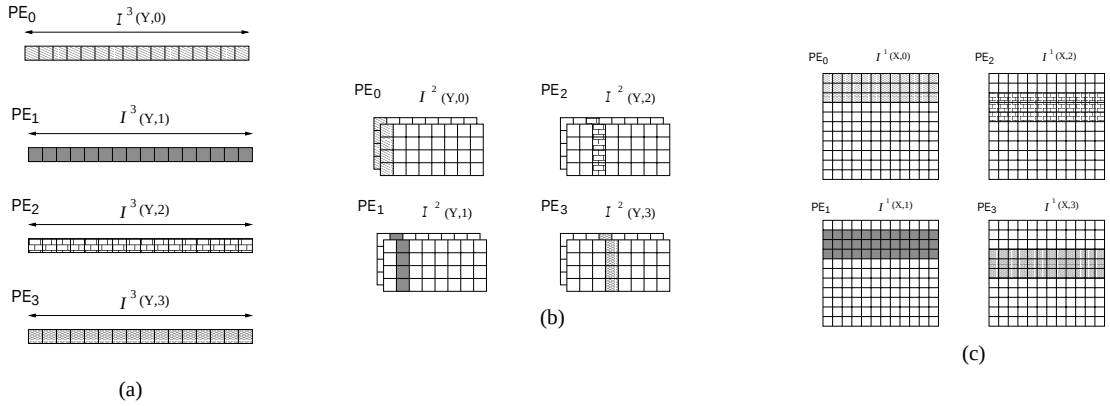


Figure 7. (a) Y is priv.; (b) Y is non-priv. and non-overlap.; (c) X is non-priv., overlap. and is only-read

Theorem 1 can be extended for a collection of parallel iterations. Then, for each parallel iteration scheduled in a processor the corresponding ID must be allocated in the processor local memory.

4.2. Inter-phase locality

Once we have established the intra-phase locality condition in a phase (which implies there will be no communication operations in the execution of that phase), the next step is to extend the locality analysis and to determine when two phases F_k and F_g ($k < g$) access the same local region of X, so as to avoid communication operations between the execution of these phases (*inter-phase locality*).

There are two concepts that help us to relate the sub-region accessed by a number of parallel iterations to those parallel iterations: the *upper limit* and the *memory gap* [7]. The upper limit of array X for a parallel iteration i, $\mathcal{UL}(\mathcal{I}^k(X,i))$, represents the farthest memory position of the sub-region described by the ID of X in the i-th iteration of the parallel loop of phase F_k. In general, the upper limit of array X for a chunk of p parallel iterations, starting in the iteration i, $\mathcal{UL}(\mathcal{I}^k(X,i),p)$, represents the farthest memory position of all the sub-regions described by the IDs of X from the ith to the $(i+p-1)$th iterations of the parallel loop. There may be phases in the program in which the sequential loops do not access all the memory positions between two consecutive parallel iterations. In other words, there could be memory gaps between the farthest memory position of the sub-region associated with the ID of the i-th parallel iteration and the lower memory position of the next sub-region (associated with the ID of the (i+1)-th parallel iteration). To cover these cases, we define the memory gap, h^k, of array X in phase F_k. Figure 8 shows the upper limit of each ID for the parallel iterations $i = 0$, $i = 1$ and $i = 2$ of F_3 and the memory gap for the array X in the TFFT2 example of Figure 4.

Let $\mathcal{I}^k(X,i)$ and $\mathcal{I}^g(X,i')$ be the IDs of X for i and i' parallel iterations in phases F_k and F_g ($g > k$). Let h^k and h^g be the memory gap of X in each of these phases. We define the **balanced locality condition** for array X as:

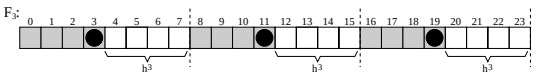
$$\mathcal{UL}(\mathcal{I}^k(X,i),p_k) + h^k = \mathcal{UL}(\mathcal{I}^g(X,i'),p_g) + h^g \quad (1)$$

$$1 \le p_k \le \left\lceil \frac{u_{k1}+1}{H} \right\rceil \quad (2)$$

$$1 \le p_g \le \left\lceil \frac{u_{g1}+1}{H} \right\rceil \quad (3)$$

where u_{k1} and u_{g1} represent the upper bounds of the parallel loops in phases F_k and F_g. $0 \le i + p_k - 1 \le u_{k1}$ and $0 \le i' + p_g - 1 \le u_{g1}$. Equation 1 determines the number of consecutive parallel iterations (p_k, p_g) that needs to be scheduled in each phase in order to ensure that the sub-region they describe is the same. Therefore, if these iterations are scheduled in processor PE, the accesses to this region are local when the region is stored in the PE local memory. That is, in processor $PE = 0$ will be scheduled $i = 0 : p_k - 1$ of F_k and $i' = 0 : p_g - 1$ of F_g; In $PE = 1$ will be scheduled $i = p_k : 2 \cdot p_k - 1$ of F_k and $i' = p_g : 2 \cdot p_g - 1$ of F_g, and so on following a CYCLIC(p_k) and CYCLIC(p_g) (BLOCK-CYCLIC) scheduling of the parallel iterations. Equations 2 and 3 limit the maximum number of parallel iterations that can be scheduled in a processor for phases F_k and F_g, thereby taking care of the load balance. By solving the system of Equations 1-3, we get the unknowns p_k and p_g, which give us the size of the chunk in the BLOCK-CYCLIC distributions as we describe in [7].

Theorem 2. To ensure that all accesses to array X in phases F_k and F_g are carried out without communication operations between the execution of these phases (**inter-phase locality**) one of these conditions suffices: 1) X is non-privatizable in phases F_k and F_g, and the intra-phase locality condition is fulfilled in phase F_k, and the balanced locality condition holds; 2) X is privatizable in phase F_k (F_g), and the intra-phase locality condition is fulfilled in phase F_g (F_k); 3) X is privatizable in phases F_k and F_g. □



$$\mathcal{UL}(\mathcal{I}^3(X,0)) = 3$$
$$\mathcal{UL}(\mathcal{I}^3(X,1)) = 11$$
$$\mathcal{UL}(\mathcal{I}^3(X,2)) = 19$$

Figure 8. Symbol ● represents the upper limit of each $\mathcal{I}^3(\mathbf{X},\mathbf{i})$; $\mathbf{h^1} = 4$ is the memory gap

Theorem 2 is based on Theorem 1 which establishes the conditions to avoid communications within a phase. Three situations may arise when we analyze the transition between the intra-phase locality condition in phase F_k and the intra-phase locality condition in phase F_g:

1. Array X is non-privatizable in phases F_k and F_g, and the intra-phase locality condition is fulfilled in phase F_k. There will be communication operations for array X between the execution of these phases if the balanced locality condition does not hold. For example, the balanced locality condition for array X in phases F_2 and F_3 of TFFT2 is expressed as:

$$p_2 + 2 \cdot Q \cdot P - P = 2 \cdot P \cdot p_3 \quad (4)$$

$$1 \leq p_2 \leq \left\lceil \frac{P}{H} \right\rceil \quad (5)$$

$$1 \leq p_3 \leq \left\lceil \frac{Q}{H} \right\rceil \quad (6)$$

The integer solution that verifies Equation 4 is $p_2 = P$, $p_3 = Q$. However, this solution does not verify Equations 5 and 6. This fact points out that all accesses to X in phases F_2 and F_3 are local if the processor PE executes all iterations of phases F_2 and F_3. That is, F_2 and F_3 would be executed sequentially, and the local memory of PE should store the whole array to avoid communication operations. Actually, this means that there will be communication operations if we do not take into account the sequential execution possibility.

On the contrary, let us analyze the balanced locality condition for non-privatizable array X in phases F_3 and F_4 of TFFT2, which is in Figure 9(c). In this case, we get $\left\lceil \frac{Q}{H} \right\rceil$ integer solutions, which verifies the balanced locality condition. Suppose that we select as a solution for the balanced locality condition $p_3 = p_4 = 1$. Figure 9(a)(b) shows the IDs associated with $p_3 = p_4 = 1$ for processor $PE = 0$. We can see that the covered data region is the same in the two phases and that the intra-phase locality condition is fulfilled in phase F_g for $p_g = 1$. As a result, all accesses to X in F_3 and F_4 are local.

2. Array X is privatizable in phase F_k (F_g). By definition the value of X in F_g (F_k) does not depend on the value of X in phase F_k (F_g). In these cases, the intra-phase locality condition (Theorem 1) holds in the phase where array X is privatizable. For the other phase, if the intra-phase locality condition is fulfilled, we are assured that, in F_k and F_g, all accesses to X are local and that they do not generate any communication operations. Here, we say that phases F_k and F_g are *un-coupled*.

3. Array X is privatizable in phases F_k and F_g. Again, these two phases are un-coupled.

Table 1. Classification of labels for edges in a LCG

$F_k - F_g$	Overl. ($\exists\ \Delta_s$)		Non overl. ($\nexists\ \Delta_s$)	
	Bal. loc.	Non-bal. loc.	Bal. loc.	Non-bal. loc.
$R - R$	L	C	L	C
$R - W$	L	C	L	C
$R - R/W$	L	C	L	C
$R - P$	D	D	D	D
$W - R$	C	C	L	C
$W - W$	C	C	L	C
$W - R/W$	C	C	L	C
$W - P$	C	C	D	D
$R/W - R$	L	C	L	C
$R/W - W$	L	C	L	C
$R/W - R/W$	L	C	L	C
$R/W - P$	D	D	D	D
$P - W$	D	D	D	D
$P - R/W$	D	D	D	D
$P - P$	D	D	D	D

Table 1 summarizes all possible cases in a LCG when Theorem 2 is applied. In the first column of the table, we show all the possible attribute combinations between the two nodes for which we are analyzing the inter-phase locality. Bear in mind that by using memory access locality analysis we aim to find out how to label the edges in the LCG. We have divided the table columns into two major parts to show whether there is parallel iteration overlapping in phase F_k ($\exists\ \Delta_s$). Each major column, is sub-divided into two additional ones, to show whether there is verification of the balanced locality condition in phases F_k and F_g. By applying Theorem 2, the edges will be labeled with C if there are communication operations between the execution of the two connected phases, or with L if it is possible to avoid communications by exploiting locality (i.e., Theorem 2 holds). In Table 1, for the labels set to L we assume that the intra-phase locality condition of phase F_k holds. We use the label D to annotate those edges that bind two un-coupled phases. For example, to build the LCG of TFFT2, shown in Figure 6, we just needed to apply the 4-th and 5-th columns of Table 1 because there is not overlapping storage in any phase (i.e., $\nexists\ \Delta_s$). For array Y, phases (F_2, F_3), (F_3, F_4) and (F_5, F_6), (F_6, F_7), are un-coupled; thus, the corresponding edges (the dashed edges in Figure 6) are first

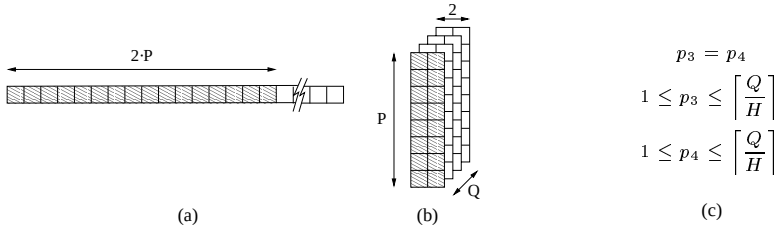


Figure 9. (a) $\mathcal{I}^3(\mathbf{X}, \mathbf{0})$ is shaded; (b) $\mathcal{I}^4(\mathbf{X}, \mathbf{0})$ is shaded; (c) the balanced locality condition

marked with D, and later removed.

4.3. Locality analysis applications

Once the LCG has been built, we can use it for at least two applications: (a) implementing an *integer programming model* to find out the iteration/data distribution that minimizes the parallel execution overhead; and (b) generating the communication routines between the phases that have been marked with label C.

a) The set of nodes of a LCG that are connected consecutively with L edges covers a common data sub-region of array X; we call this a *chain* of nodes. There can be more than one chain for an array, and each column of the LCG has at least one chain. In the LCG, the chains of array X are separated by C edges. The nodes of the chains define the constraints of our integer programming model. Those nodes $(k-g)$ of the LCG connected with C represent a redistribution (*Global communications*) or an updating of overlapped subregions (*Frontier communications*), which are the communication patterns that our approach can handle. These nodes take part in the definition of the objective function (where $C^{kg}(X, p_k)$ is the communications cost function) because communications are one of the overhead sources. The other overhead source is the accumulation of the parallel load unbalance assigned to processors in each phase (where $D^k(X, p_k)$ is the load unbalance cost function). Due to space limitations, we leave the full description in [8], where we present and validate by measurements the cost functions. Table 2 illustrates all the constraints for the LCG of the TFFT2 code example (Figure 6). The objective function that models the parallel overhead is shown in Equation 7, where X_j represents one of the two arrays of the program.

$$o.f. = Min \left\{ \sum_{j=1:2} \sum_{k=1:8} D^k(X_j, p_k) + C^{kg}(X_j, p_k) \right\} \quad (7)$$

The solution to this problem allows us to find p_k. As a reminder, this is the size of the chunk of parallel iterations scheduled in each node (phase), following a CYCLIC(p_k) scheme. Once the iteration distribution of a node has been selected, we can compute the data distribution for each array of that node. To do this, the data distribution function must hold the intra-phase locality requirements imposed by the IDs associated with the chunks of parallel iterations scheduled in each processor. However, it is possible to find a global data distribution for all the nodes of the chain such that all the accesses to X in any of the connected nodes of the chain are in the local memory of the processors. The reason is that all nodes belonging to the same chain cover the same data region of array X (inter-phase locality). Thus, the data allocation procedure of array X only takes place before the first node of the chain. Therefore, for each array X in the LCG, after the execution of the last phase of a chain, and before the execution of the first phase of the next chain, a data allocation procedure (redistribution) must re-allocate the data of X. We describe this procedure in [8].

b) In [8] we also outline, how to automatically generate the communication routines for array X when there is a C edge connecting two nodes in a LCG. In the communications routines generation we take into account two communications patterns: the Frontier Communications and the Global Communications. In addition, message aggregation is performed in our approach.

Some experiments were conducted in [10] and [7] to probe the effectiveness of the iteration/data distributions and communication generation obtained with our approach for a set of six real codes. The parallelization procedure is as follows. First, the Polaris parallelizer marked the parallel loops of each code. Then, the LCG of each code was built, and the integer programming problem of each code was derived. Here we used GAMS to obtain the iteration/data distributions of each phase; the solutions of the integer programming problems were obtained in few seconds in a R1000. Finally, those distributions were hand-coded for each program, including the communications generation when necessary. These parallel codes were executed in a Cray T3D. We achieved parallel efficiencies of over 70% in the Cray for 64 processors.

5. Conclusions

We have shown a compile-time representation that handles array reshaping, non-affine access functions and non-perfectly nested loops, that is suitable for general inter-

Table 2. Constraints for TFFT2 code

	X	Y
Locality const.	$p_{31} = p_{41}$ $P \cdot p_{41} = Q \cdot p_{51} \quad p_{51} = p_{61}$ $p_{61} = p_{71} \quad 2 \cdot Q \cdot p_{71} = p_{81}$	$p_{12} = Q \cdot p_{22}$ $P \cdot p_{32} = Q \cdot p_{52}$ $2 \cdot Q \cdot p_{62} = p_{82}$
Load bal. const.	$1 \leq p_{11}, p_{81} \leq \left\lceil \frac{P \cdot Q}{H} \right\rceil \quad 1 \leq p_{31}, p_{41} \leq \left\lceil \frac{Q}{H} \right\rceil$ $1 \leq p_{21}, p_{51}, p_{61}, p_{71} \leq \left\lceil \frac{P}{H} \right\rceil$	$1 \leq p_{12}, p_{82} \leq \left\lceil \frac{P \cdot Q}{H} \right\rceil \quad 1 \leq p_{32}, p_{42} \leq \left\lceil \frac{Q}{H} \right\rceil$ $1 \leq p_{22}, p_{52}, p_{62}, p_{72} \leq \left\lceil \frac{P}{H} \right\rceil$
Storage const.	$p_{81} \cdot H \leq \Delta_d^{81} = P \cdot Q$ $p_{81} \cdot H \leq \frac{\Delta_r^{81}(1)}{2} = \frac{P \cdot Q}{2}$ $p_{81} \cdot H \leq \frac{\Delta_r^{81}(2)}{2} = \frac{2 \cdot P \cdot Q}{2}$	$p_{12} \cdot H \leq \Delta_d^{12} = P \cdot Q$ $Q \cdot p_{22} \cdot H \leq \Delta_d^{22} = P \cdot Q$ $p_{82} \cdot H \leq \Delta_d^{82} = P \cdot Q$ $p_{82} \cdot H \leq \frac{\Delta_r^{82}(1)}{2} = \frac{P \cdot Q}{2}$ $p_{82} \cdot H \leq \frac{\Delta_r^{82}(2)}{2} = \frac{2 \cdot P \cdot Q}{2}$
Affinity const.	$p_{11} = p_{12} \quad p_{21} = p_{22} \quad p_{31} = p_{32} \quad p_{41} = p_{42}$ $p_{51} = p_{52} \quad p_{61} = p_{62} \quad p_{71} = p_{72} \quad p_{81} = p_{82}$	

procedural analysis. Using this representation, we build the LCG graph to capture the memory locality exhibited by a program (intra-phase locality and inter-phase locality). The LCG graph summarizes the phases of the program for which all accesses can be local or communication operations are required. This graph is a powerful tool that can be easily handled by the compiler in later optimization stages: iteration partition, data distribution, array replication, and communication scheduling. The framework comprising these last steps is presented in [7], where an integer programming problem results in the iterations/data distributions (including array replication) that minimize the number of remote accesses while keeping a good load balance for all the parallel loops in the code. Experimental results support the effectiveness of the data distributions and the control of communication operations derived with our approach for the set of tested programs.

References

[1] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, Alburquerque, New Mexico, June 1993.

[2] R. Barriuso and A. Knies. SHMEM user's guide for Fortran. Cray Research Inc., August 1994. Revision 2.2.

[3] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali, and Stodghill. Solving alignment using elementary linear algebra. In K. P. et al., editor, *Proceedings of LCPC'94*, number 892 in LNCS. Springer Verlag, Ithaca, N.Y., August 1994.

[4] S. Chatterjee, J. Gilbert, and R. Schreiber. The alignment-distribution graph. In U. Banerjee, editor, *Proceedings of LCPC'93*, number 768 in LNCS. Springer Verlag, Portland, OR., August 1993.

[5] C. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. *J. Parallel and Distributed Computing*, 19:90–102, 1993.

[6] K. Kennedy and U. Kremer. Automatic data layout for high performance Fortran. In *Supercomputing'95*, San Diego, CA, December 1995.

[7] A. G. Navarro and E. Zapata. An automatic iteration/data distribution method based on access descriptors for DSM multiprocessors. Technical report, Department of Computer Architecture, University of Málaga, 1999.

[8] A. G. Navarro and E. L. Zapata. An automatic iteration/data distribution method based on access descriptors for DSMM. In *Twelfth International Workshop on Languages and Compilers for Parallel Computing (LCPC'99)*, The University of California, San Diego, La Jolla, CA USA, August 4–6 1999.

[9] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of array access patterns for compiler optimizations. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1994.

[10] Y. Paek, A. Navarro, E. Zapata, and D. Padua. Parallelization of Benchmarks for scalable shared memory machines. In *IEEE Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 401–408, Paris, France, October 12–18 1998.