A Scheme for Automatic Data Layout in Distributed Memory Machines

Siddhartha Saha, Kamalika Chaudhuri , R Maloo , Sanjeev K Aggarwal
Department of Computer Science and Engineering
IIT Kanpur - 208016, India
Email: ska@iitk.ac.in

Abstract In this paper, we present a new scheme for automatic data distribution while compiling for distributed memory architectures. Our algorithm generates block, cyclic and block-cyclic distributions. We analyze the algorithm and show that the complexity of our algorithm is n_p^2 times the complexity of the alignment and distribution phase combined together (n_p is the number of phases). We have done simulations and the results are encouraging.

Keywords: Compiling, distributed memory.

1 Introduction

In this paper, we present a new scheme for automatic data distribution while compiling for distributed memory architectures. Improper data layout makes the target code slow. The choice for data distribution depends on the problem structure, the underlying system characteristics, number of available processors and other factors. Also different phases of a program may perform best for different data distributions. Hence, to achieve better performance the data may have to be redistributed after certain phases.

We have developed a new algorithm for data distribution and re-distribution. We discuss a cost model for programs. Our algorithm maps data arrays into distributed nodes, minimizing the cost. Our algorithm generates block, cyclic as well as block cyclic distributions.

2 Background and Related Work

A great deal of literature is available on compilers for distributed memory systems. Kremer [Kre93] and Garcia [GAL01] have used Integer Linear Programming to solve the data distribution problem. Kremer's approach creates a search space of data distribution candidates for each phase [Kre93] in the program. This approach allows data redistribution only between phases. A Phase Control Flow Graph (PCFG) is constructed, where each phase is a node and the edges represent the control flow between the phases. This information is used to perform an exhaustive analysis on all the candidate layouts in the search space. The information about the potential layouts of each phase along with the cost of data layout and the cost of re-mapping is used to build a 0-1 ILP. The solution of this ILP gives the optimal data distribution. Garcia [GAL01], however, instead of dividing the algorithm in alignment, distribution and remapping phases solves the problem globally. A single data structure, called Communication Parallelism Graph (CPG) is used, which contains all the relevant information about the data movement and the parallelism of the data under analysis. This approach uses a cost model in which cost is expressed in time units. The whole data mapping problem is modeled as a 0-1 Integer Linear Programming and is solved to obtain a data mapping which minimizes the total execution time.

3 Our Algorithm

Our algorithm works in four stages.
1. Preprocessing
2. Alignment Analysis
3. Distribution Analysis
4. Redistribution

In the preprocessing stage, two important tasks are carried out. First, the program is divided into a number of phases [Kre93]. A Phase is an outermost loop in a loop nest such that loop defines an induction variable that occurs in a subscript expression of an array reference in a loop body. Secondly, we generate *data access patterns* from a given run of the program. For example, if a program contains the statement

```
for i=1 to 100
a[i] = b[i+1];
```

then the data access patterns corresponding to these statements are (a(1),b(2)), (a(2),b(3)), ...

The second and the third stages work on the program phase by phase. For each phase, an optimal alignment of the arrays used in the phase to a global template is produced, leading to optimal data distribution. The fourth and the final stage is data redistribution; in this stage successive phases are merged based on the gain in performance. The first stage is trivial; in this paper, we focus on the steps 2, 3 and 4.

3.1 Model of a Distributed Memory Machine

A distributed memory machine can be viewed as a 4-tuple (P, R_c, W_c, D_c) where P is the number of processing nodes, and R_c, W_c and D_c are read overhead, write overhead and communication delay functions [Sar89]. It is assumed that all the processors communicate with each other through message passing and there is no shared memory.
R_c is the communication overhead function for reading data. $R_c(k, j, l, m)$ denotes the execution time incurred by processor j to receive l units of data from processor k, the communication load of the system being m at that time.
W_c is the communication overhead function for writing data. $W_c(k, j, l, m)$ denotes the execution time incurred by processor k to send l units of data to processor j, the communication load of the system being m at that time.
D_c is the communication delay overhead function. $D_c(j, k, l, m)$ denotes the delay which must elapse after processor j sends l units of data to processor k, the communications load of the system being m at that time. The delay overhead is the length of the latency period after processor j has incurred its cutout overhead W_c and before processor k has incurred its input overhead, R_c.

3.2 The Trace Dependency Graph

The Trace Dependency Graph is defined as a graph $G = (V, E)$ in which each node is a trace pattern [Mal01]. A trace pattern is a data access pattern, which has been transformed to the common global template.
In the TDG, an edge between two nodes represents communication requirements. Each node is assigned a unique number in increasing order, called its ID. Suppose that two nodes Tr_i and Tr_j use the same data element t_k, with $ID(Tr_i) < ID(Tr_j)$. Then these two nodes will have an edge labelled t_k if there is no node Tr_l in between which uses t_k. Note that whether actual communication takes place through an edge of the TDG also depends on the processor assignments of the nodes. The edges through which communication actually takes place, for a given processor assignment, constitute a set called the COMM set of the TDG.

3.3 Cost Model

For building an optimal data layout for a program, we need a cost model for comparing various data layouts. We use an estimate of the parallel time taken to execute the program fragment under the given data layout as our cost model. This cost can be computed from the TDG using Algorithm 1.
Given a data distribution and a set of processor affinities, a processor assignment is computed first. This is then used to determine the estimated parallel time of execution. To estimate the cost of re-mapping two data layouts between phase boundaries, we use equation (1).

$$\begin{aligned} REMAP\_COST = max_i[\\ &\Sigma_{d\ s.t\ PA_1(d)=i\ and\ PA_2(d)\neq i} Read\_cost \\ &+\Sigma_{d\ s.t\ PA_1(d)=i\ and\ PA_2(d)\neq i} Write\_cost] \quad (1)\end{aligned}$$

where
$Read\_Cost = Read\_overhead + Delay\_in\_Read$
$Read\_overhead = R_c(i, PA_2(d), size(d), L)$
$Delay\_{}_in\_Read = D_c(i, PA_2(d), size(d), L)$
and
$Write\_cost = Write\_overhead + Delay\_in\_Write$
$Write\_overhead = W_c(PA_1(d), i, size(d), L)$
$Delay\_in\_Write = D_c(PA_1(d), i, size(d), L)$

Here $PA_1()$ and $PA_2()$ denote the processor assignment of a particular data element in the distributions 1 and 2 respectively. The communication load is estimated as

$$L = \frac{1}{K} \Sigma_{ds.tPA_1(d)\neq PA_2(d)} size(d) \quad (2)$$

Here K is a constant estimated to be the average time to perform a re-mapping.

4 Our Algorithm

The first stage involves some preprocessing on the input program. Two tasks are achieved in this stage - dividing the program into phases and generating the data access patterns by profiling information. This is a trivial task. Here we do not go into the details of this task due to lack of space.

4.1 Alignment Stage

The data alignment stage uses the Component Affinity Graph [LC91] to represent relative alignment preferences. The nodes of the graph denote components of the index domain to be aligned, and the edges denote affinity between two domain components. The edge weights denote the degree of the alignment preference. From the component affinity graph, the alignment problem reduces to the problem to partitioning the node set G into n disjoint subsets, with the restriction that no two nodes belonging to the same index domain are allowed to be in the same subset. Here n is the maximum dimensionality of any array in the program [LC91]. A heuristic algorithm [LC91] is then used to solve the problem.

4.2 Distribution Algorithm

Our overall distribution algorithm can be given as

1. Build TDG of the program phase
   / uses the data access patterns and the optimal alignment generated by previous steps to compute the TDG*/*

Algorithm 1 : Calculate Parallel Time
Inputs: A Trace Dependence Graph (TDG) and a processor assignment$PA()$
Output: The estimated parallel time of execution of the program phase given the processor assignment
Procedure

```
1. $COMM = Calculate\_COMM(PA)$
   /*first calculates the COMM edge set, that is the edges that actually need communication given PA*/

2. $LOAD = \Sigma_{(Tr_i,Tr_j,t_k)\epsilon COMM} SIZE(t_k)/\Sigma_{n\epsilon V} TIME(n)$     /*Determines the average communication load*/

3. For every node $n\epsilon V$ do     /*for every node finds out the time of execution, including delays */
   $TOTALTIME(n) = TIME(n) + \Sigma Rc(PA(Tr_i), PA(n), SIZE(t_k), LOAD) +$
                                  $\Sigma Wc(PA(n), PA(Tr_j), SIZE(t_k), LOAD)$

4. For every node n in V do     /*finds the earliest starting time for each node */
   $EarliestStartTime(n) = max\{\{EarliestStartTime(Tr_i) + TOTALTIME(Tr_i) +$
                                  $Dc(PA(Tr_i), n, SIZE(t_k), LOAD),$
                                  such that there is an edge in $COMM$ labeled $t_k$ from $Tr_i$ to $n\}$
                              $\cup\{EarliestStartTime(m) + TOTALTIME(m)$
                                  such that $PA(m) = PA(n)$ and $ID(m) < ID(n)\}\}$

5. $PARALLELTIME = max\{EarliestStartTime(n) + TOTALTIME(n)\}$
```

```
2. Do_Initial_Scheduling()
   /* determine optimal processor assignment for each node*/

3. Create_Search_Space()
   /* build up a search space of candidate distributions */

4. Find_Optimal_Distribution()
   /* find the optimal distribution candidate */
```

4.2.1 Building the TDG

The first step uses the data access patterns generated by the preprocessing stage and the optimal alignment generated by the alignment stage to build the TDG. Each data access pattern is transformed using the optimal alignment function to produce each node of the TDG. Suppose that two nodes Tr_i and Tr_j of the TDG use the same data element t_k, with $ID(Tr_i) < ID(Tr_j)$. Then an edge labelled t_k is added between Tr_i and Tr_j if there is no node Tr_l in between which uses t_k.

4.2.2 Initial Scheduling

The procedure to do initial scheduling is described in Algorithm 2. This algorithm has similar structure to the one used by [Sar89]. However, our algorithm uses TDG in place of GR graph used by [Sar89]. The procedure first assigns each node to execute on a virtual processor. It then tries to optimize communication by merging edges in the COMM graph; finally the nodes are scheduled greedily onto the available processors.

The procedures *Merge()* is used to transfer each node assigned to one processor to another and save the old processor assignment; the procedure *Restore()* is used to restore the old processor assignment.

4.2.3 Creating the Search Space

The algorithm then builds up a search space of candidate distributions, which might yield an optimal solution. For this purpose, it looks at all trace nodes, scheduled on a particular processor. Each data element of the nodes scheduled on a given processor is examined in order of increasing index, to find the difference between the successive indexes. Based on this data, a distribution candidate of type BLOCK, CYCLIC or BLOCK_CYCLIC is proposed for the array. This is done for every dimension of the global template.
After the search space is built, the algorithm makes a second pass over the search space to modify all infeasible distribution candidates.

4.2.4 Finding the optimal distribution

The next step is to find the optimal distribution out of this search space, which is done by the procedure Find_Optimal_Distribution(). It first reinitializes the processor assignment of the trace pattern nodes, and recalculates the $COMM$ set. It then tries merging every edge in the $COMM$ set, and checks every distribution in the search space to see if they benefit by this edge merging. Along with choosing the optimal distribution, it also builds up $PA()$, the processor assignment array, by noting down the processor affinity of a particular node to a given processor under a given data distribution.

Algorithm 2: Do Initial Scheduling
Inputs: Trace Dependency Graph (TDG) and Number of Processors (P)
Output: A processor assignment PA() to each node in the TDG, which optimizes inter-processor communication
Procedure

```
1. For each node n in V { $PA(n) = ID(n)$}        /*Initialize, by setting each node to execute on a virtual processor*/

2. $PAR\_TIME = Determine\_Time(G, PA)$

3. $COMM = Create\_COMM(G)$
   /*Try merging each communication edge efficiently, so that parallel execution time is decreased*/

4. For each edge ($Tr_i$, $Tr_j$, $t_k$) in $COMM$ do        /*Now schedule the graph nodes on P processors*/
        $Merge(G, PA(Tr_i), PA(Tr_j))$; $NEW\_TIME = Determine\_Time(G, PA)$;
        If $(NEW\_TIME < PAR\_TIME)$
            then $PAR\_TIME = NEW\_TIME$
            else $Restore(G)$

5. For every node n in V do
        $PAR\_TIME = \infty$
        For proc = 1 to P do Merge(G, proc, PA(n))
        $NEW\_TIME = Determine\_time(G, PA)$
        If $(NEW\_TIME < PAR\_TIME)$ { $P_{min} = proc$; $PAR\_TIME = NEW\_TIME$}
        $Restore(G)$; $Merge(G, Pmin, PA(n))$
```

The algorithm makes use of the function $Accommodate()$. This function takes as input a candidate data distribution for the program and a data element, and produces as output the processor number where this data element will reside, given the data distribution. The algorithm also uses a set LDB. This is a set of three-tuples $(d, n, proc)$, where d denotes a candidate distribution, n denotes a trace pattern node and $proc$ denotes a processor number. LDB records the affinity of a trace pattern node for a given processor if the data is distributed in the form of the candidate distribution. For recording the affinity of a trace pattern node to a given processor, we assign it to the processor where its Left Hand Side (LHS) or in other words, the value it is computing, resides.

4.3 Data Redistribution

Our alignment and distribution algorithms operate on one phase at a time. The redistribution algorithm considers all the phases in the program and tries to merge phases based on performance. The redistribution algorithm works on a data structure called the Phase Affinity Tree [Mal01]. Initially the tree consists of a dummy node as root and all the phases as children of the root. As phases are merged, they are replaced by a sub-tree containing a dummy node as root and the phases as its children.

The redistribution algorithm proceeds in stages. In the first stage, it considers merging adjacent phases. In the second stage, it considers merging three adjacent phases and so on. Phases are merged only if merging gives a benefit; the *Merge - Benefit* for two phases ϕ_1 and ϕ_2 is defined as

$$\begin{aligned} Merge\_Benefit(\phi_1, \phi_2) = PAR\_TIME(\phi_1 \cup \phi_2) \\ -PAR\_TIME(\phi_1) - PAR\_TIME(\phi_2) \\ -REMAP\_COST(\phi_1, \phi_2) \end{aligned} \quad (3)$$

Here $PAR\_TIME(\phi)$ denotes the estimated parallel time of execution of phase ϕ assuming an optimal alignment and distribution for ϕ, and $REMAP\_COST(\phi_1, \phi_2)$ is the cost of re-mapping between the optimal data distributions for phases ϕ_1 and ϕ_2.

4.4 Example

Consider the following program fragment

```
for i= 1 to 15 step 2
a[i] = a[i+1] + a[i+8] + a[i+9]
```

We consider applying our algorithm on this program fragment for a machine with $P = 4$ and $R_c(j, k, s, L) = W_c(j, k, s, L) = D_c(j, k, s, L) = s, \forall j, k, L$. Since there is only one array involved, the alignment stage is trivial. The TDG out-putted by the distribution phase of the program is shown in Figure (1). As suggested by the figure, the best distribution candidate for this example will be a Block Cyclic distribution with a block size of 2, distributed across the four processors. After initial scheduling, nodes 1 and 5 of the graph will be scheduled on Processor 1, 2 and 6 on Processor 2 and so on. The AAE values for the processors are given below:

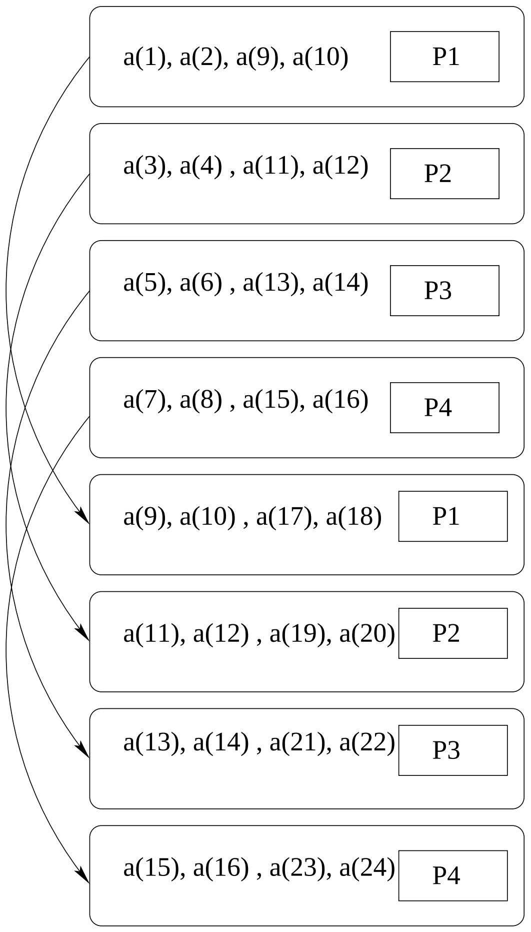
a(1), a(2), a(9), a(10)
P1
a(3), a(4) , a(11), a(12)
P2
a(5), a(6) , a(13), a(14)
P3
a(7), a(8) , a(15), a(16)
P4
a(9), a(10) , a(17), a(18)
P1
a(11), a(12) , a(19), a(20)
P2
a(13), a(14) , a(21), a(22)
P3
a(15), a(16) , a(23), a(24)
P4

Figure 1. Trace Dependency Graph along with Processor Assignment for example Program

AAE(1) =	1,2,9,10,17,18
AAE(2) =	3,4,11,12,19,20
AAE(3) =	5,6,13,14,21,22
AAE(4) =	7,8,15,16,23,24

After the creation of search space, the search space will consist of the elements $(Blocksize = 2, Procs = 4, BLOCK\_CYCLIC), (Blocksize = 1, Procs = 4, CYCLIC), (Blocksize = 6, Procs = 4, BLOCK)$. Out of these candidate distributions, our algorithm will choose the optimal distribution to be $(Blocksize = 2, Procs = 4, BLOCK\_CYCLIC)$.

4.5 Analysis

The preprocessing stage involves a single pass through the program to divide the program into phases, and a single pass through a given run of the program to generate the data access patterns. Let there be n_a arrays in the program, with the maximum dimensionality of all arrays being M. Then the CAG will consist of at most $n_a * M$ nodes. Generating such a graph takes a single pass through the program. Finding a maximum weighted bipartite matching on a graph containing M nodes will take $O(M^3 \log M)$ time, and there will be n_a such passes [LC91]. This makes the total complexity of the alignment stage to be $O(n_a * M^3 \log M)$.
To analyze the complexity of the distribution phase, let n be the number of nodes in the TDG, m be the number of edges in TDG, and P be the number of processors. Also let d be the total number of data elements accessed by the program. Then building the TDG takes $O(n^2 d)$ time in the worst case.
Once the TDG is built calculating the initial schedule takes $O((n * P + m) * (m + n^2 d))$ time, and creating the search space takes $O(S + P * M * (n \log n + D))$ time. Here S denotes the total number of BLOCK and CYCLIC(1) distributions. It has been shown [Kre93] that the total number of possible BLOCK and CYCLIC(1) distributions for p^k processors, where p is a prime number, and a M dimensional program template is $\Sigma_{i=1}^{M} 2^i * \binom{M}{i} * \binom{k-1}{i-1}$. Finally we take $O(m * (S + d * P) * (m + n^2 d))$ time to find the optimal distribution from the search space. This makes the complexity of the distribution stage as $O(P * M * (n \log n + D) + m * (s + d * P) * *(m + n^2 * d))$
The re-mapping stage calls the alignment and distribution stages as a subroutine in a pass. Suppose there are n_p phases in the program; then there will be at most n_p^2 such calls. This makes the total complexity of the program to be n_p^2 times the complexity of the alignment and distribution phase combined together.

5 Conclusion

In conclusion, we have developed a new heuristic algorithm to perform automatic data layout while compiling for distributed memory machines. Our scheme works in four stages - preprocessing, alignment,data distribution and redistribution. Some initial simulations of our scheme have been done in [Mal01]. The results of these simulations were encouraging. However, more extensive simulations of the algorithm need to be carried out.

References

[GAL01] J. Garcia, E. Ayguade, and J. Labarta. A framework for integrating data alignment, distribution and redistribution in distributed memory multiprocessors. *IEEE Trans. on Parand Dist Systems*, 12(4), 2001.

[Kre93] U Kremer. *Automatic Data Layout for distributed memory machines*. PhD thesis, Rice University, 1993.

[LC91] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13, 1991.

[Mal01] R Maloo. *A Framework for Automatic Data Layout for Distributed Memory Machines*. MTech thesis, IIT Kanpur, 2001.

[Sar89] V. Sarkar. *Partitioning and scheduling parallel programs for multiprocessors*. MIT Press, Cambridge, Massachussets, 1989.

Algorithm 3: Create Search Space
Input: The Trace Dependency Graph $G = (V, E)$ and A processor assignment to each node of the TDG, PA()
Output: $SEARCH\_SPACE$: the search space of all candidate data distributions
Procedure

```
1. $SSPACE = \phi$

2. For $proc$ = 1 To $P$ Do          /*Compute the set of all array elements assigned to this processor*/
       $AAEproc =\{ n|n$ is an array element accessed by a node scheduled on proc$\}$
                                      /* Generate candidate distribution for each dimension */

3. For $dim$ = 1 to $MAXDIMS$ do      /*Now try all combination of candidate distribution for each dimension */
       $bsize = 1$
       For Every Data element $da_i$ in $AAEproc$ do
           $DIFF$ = difference in index of dimension $dim$ between $da_i$ and $da_{i-1}$
           If $DIFF = 1$ then $bsize = bsize + 1$
           else
               $procs = (DIFF + bsize - 1)/bsize$
               $SSPACE = SSPACE \cup \{procs, dim, (bsize, procs, block\_cyclic)\}$
               $bsize = 1$

4. $SEARCH\_SPACE$ = $\{((b_1,p_1,t_1), \ldots,(b_{maxdims}, p_{maxdims}, t_{maxdims}))|$
                         $(pr_i,i,(b_i,p_i,t_i))\ \epsilon SSPACE \cap pr_1 = pr_2 = \ldots = pr_{maxdims}\}$
                         /*Insert all possible BLOCK and CYCLIC(1) candidates */

5. $SEARCH\_SPACE$ = $SEARCH\_SPACE \cup$
                    $\{((1, p_1, CYCLIC)\ , \ldots,\ (1,p_{maxdims},CYCLIC))|$
                     $p_1.p_2..p_{maxdims} <= P$ and for some $i$, $p_i > 1\ \}$

6. $SEARCH\_SPACE$ = $SEARCH\_SPACE \cup$
                    $\{((BLOCK, p_1, t_1)\ , \ldots,\ (BLOCK, p_{maxdims}, t_{maxdims}))|$
                     $b_i = Number\_elements(i)/p_i$ and $p_1.p_2..p_{maxdims} <= P$ and $p_i > 1$ for some $i\}$
                      /*Now we modify infeasible distribution candidates in the search space */

7. For each distribution candidate $((b_1, p_1, t_1)\ , \ldots,\ (b_{maxdims}, p_{maxdims}, t_{maxdims}))$ in $SEARCH\_SPACE$ Do
       $ModCount = 1$
       While $(p_1.p_2 \ldots p_{maxdims} > P)$ Do
           $Dim2mod = (ModCount\%MAXDIMS) + 1$
           $K = Min\_Prime\_Divisor(pDim2mod)$
           $bDim2mod = bDim2mod * K$
           $pDim2mod = pDim2mod/K$
           $ModCount = ModCount + 1$
```

Algorithm 4 : Find_Optimal_Distribution

Inputs: SEARCH_SPACE - the distribution candidate space built up by Create_Search_Space()
The Trace Dependency Graph G = (V,E)
Outputs: CURRENT_SSN : the desired distribution candidate
PA() : a processor assignment array for the desired distribution candidate

Procedure

```
1. For each Trace Pattern $Tr_i$ in V do          /*Reinitialize the processor assignment array */
        $PA(Tr_i) = ID(Tr_i)$

2. $PAR\_TIME = Determine\_Time(G, PA)$

3. $COMM = Create\_COMM(G)$
   /*Now try merging each communication edge and try all distributions for the merger */

4. For each edge ($Tr_i$,$Tr_j$,$t_k$) in $COMM$ do
        $Merge(G, PA(Tr_i), PA(Tr_j))$

5. $NEW\_PAR\_TIME$ = $Determine\_time(G, PA)$
   /*if merging this edge gives a benefit, check it for all distributions else restore the graph and proceed */

6. If $NEW\_PAR\_TIME < PAR\_TIME$
        For each candidate distribution $d$ in $SEARCH\_SPACE$ Do
        /*if $Tr_i$ and $Tr_j$ are executed at the same processor */
            If ($Accommodate(d, LHS(Tr_i)) = Accommodate(d, LHS(Tr_j))$)
                LDB = LDB $\cup(d, Tr_i, Accommodate(d, LHS(Tr_i)) \cup (d, Tr_j, Accommodate(d, LHS(Tr_j))$
                $NEW\_PAR\_TIME = Determinetime\_withdistr(G, d)$
                If $NEW\_PAR\_TIME < PAR\_TIME$
                    $PAR\_TIME = NEW\_PAR\_TIME$
                    $CURRENT\_SSN = d$
                    Initialize $PA(n) = ID(n), \forall n$
                    For all $(d, n, proc)$ in LDB, set $PA(n) = proc$
   Else
        $Restore\_Graph()$

7. Exit with $CURRENT\_SSN$ and $PA()$
```

Algorithm 5: Main Algorithm
Inputs: Division of the program into phases and Data access patterns for the program
Output: An optimal data layout scheme for the overall program
Procedure

```
1. $G = Phase\_Affinity\_Tree$

2. $distance = 1$

3. Generate Alignment and Data Distribution for every phase of the program,
   using the Alignment Analyzer and Distribution Analyzer routines

4. if $(distance \geq n_{phases})$ then exit with the current data layout

5. for each phase $i \epsilon G$      if $(i + distance > n_{phases})$ break
       Compute optimal Alignment and Data Distribution for the combined phases $i$,$i+1$,...,$i + distance$ taken together
       merge_benefit = $\Sigma_{j=i}^{i+distance} COST(j, align_j, distr_j) +$
                       $\Sigma_{j=i}^{i+distance-1} REMAP\_COST(align_j, distr_j, align_{j+1}, distr_{j+1}) -$
                       $COST(merged - phase(i, \cdots, i + distance),$
                             $combined - align_{i,i+distance},$
                             $combined - distr_{i,i+distance})$
       If merge_benefit $> 0$ put an edge $ed_i$ between phases $i$ and $i + distance$ with weight merge_benefit

6. Choose the edge $ed_j$ with the highest weight

7. while $(ed_j \neq NULL)$
          G = $merge(j, \cdots, j + distance)$
          For every phase $k$ such that $j - distance \leq k \leq j$
              If ($k + distance \leq n_{phases}$) Perform step 5 for phase k
          Choose the edge $ed_j$ with the highest weight

8. $distance = distance + 1$

9. Goto step 4
```