Data Distribution Analysis and Optimization for Pointer-Based Distributed Programs

Jenq Kuen Lee Dan Ho Yue-Chee Chuang
Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan

Abstract

High Performance Fortran (HPF) provides distributed arrays to efficiently support a global name space on distributed memory architectures. The distributed data structures supported by HPF, however, are only limited to array constructs, and do not extend to pointer-based distributed structures. With the support of distributed pointers and class abstractions in parallel C++-like languages, application programmers now can build their own pointer-based aggregate objects to provide a global name space on distributed environments. However, a critical question remains if the compiler can understand the distribution pattern of pointer-based distributed objects built by application programmers, and perform optimization as effectively as the HPF compiler does with distributed arrays. In this paper, we address this challenging issue. In our work, we first present a parallel programming model which allows application programmers to build pointer-based distributed objects at application levels. Next, we propose a distribution analysis algorithm which can automatically summarize the distribution pattern of pointer-based distributed objects built by application programmers. Our work, to our best knowledge, is the first work to attempt to address this open issue. Our distribution analysis framework employs Feautrier's parametric integer programming as the basic solver, and can always obtain precise distribution information from the class of programs written in our parallel programming model with static control. Experimental results done on a 16-node IBM SP-2 machine show that the compiler with the help of distribution analysis algorithm can significantly improve the performance of pointer-based distributed programs.

Keywords: Distributed Memory Environments, SPMD Computation, Distributed Pointer Analysis, Array Data Flow Analysis, Parallel C++ Compiler, Optimizing Compiler.

1 Introduction

High Performance Fortran (HPF)[18] provides distributed arrays to support a global name space on distributed memory architectures. In HPF, programmers can specify the distribution of the data structures among processors, and the compiler can then take distribution information (how data are distributed among processors), and perform compiler optimization to utilize memory locality and reduce communication traffics[5, 11, 17, 22]. The distributed data structures supported by HPF[18], however, are only

[1]This paper is submitted to *International Conference on Parallel Processing, 1997.* The correspondence author of this work is Jenq Kuen Lee. His e-mail address is jklee@cs.nthu.edu.tw, FAX number is 886-3-5723694, phone number is 886-3-5715131 EXT. 3519, and postal address is Prof. Jenq Kuen Lee, Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan.

limited to array constructs, and do not extend to pointer-based distributed structures. With the support of distributed pointers and abstract object types in parallel C/C++ systems such as DINO[28], Split-C[4], Olden[6], and parallel C++[21, 3, 16] environments, application programmers now can build user-level pointer-based distributed structures such as distributed linked list, graph, tree, mesh, and many other application-level structures. For example, programmers or library designers can build a distributed tree with a global name space on distributed memory architectures by constructing a subtree with SPMD computation by each processor (SPMD is a parallel computation model that every processor executes the same program, but different set of data), and then link the subtrees together via distributed pointers. After the construction of the structures, the distributed tree can then be manipulated as though the shared memory and global name space are available. However, just as in the compilation process of HPF language, the compiler needs the data distribution information of the builtin distributed arrays to generate efficient communication codes in emulating the global name space on distributed memory architectures, it remains an open question how the distributed information of user-defined linked distributed structures can be passed to compilers for optimizations.

In this paper, we are interested in answering the above question from the viewpoints of fundamental compiler technology, — "Can the compiler be intelligent enough to automatically understand the distribution pattern of pointer-based distributed objects built by application programmers, and perform optimization as effectively as HPF does with distributed arrays?". In our work, we first present a parallel programming model which allows application programmers to build pointer-based distributed objects at application levels. Next, we propose a distribution analysis algorithm which can automatically summarize the distribution pattern of pointer-based distributed objects built by application programmers. Our work, to our best knowledge, is the first work to attempt to address this open issue. Previously, the research work of pointer analysis[2, 12, 13, 15, 25], have mainly focused on gathering the aliases information of pointer programs on flat shared memory parallel machines, and do not provide the distribution information needed for compiler optimization on distributed memory parallel architectures. Our distribution analysis framework employs Feautrier's parametric integer programming[23, 24] as the basic solver, and can always obtain "precise" distribution information from the class of programs written in our parallel programming model. We say the information is precise, because at any given program continuation point, we can always identify the exact memory instance which a pointer is pointing to, or the exact memory instance which a field of a memory instance is pointing to. Traditionally, in scalar data flow equations[1] and alias analysis[25], conservative estimation will return a set of memory instances which a pointer may be pointing to, while in our case, we will always return a unique one.

The summary information of the distribution information of the programs we gather from our analysis algorithm is called "raw distribution information" in this paper. The raw distribution information we gather is always precise, and can be converted into different "cooked" data distribution pattern for different compiler optimization purposes. For example, in this work, we try to infer the raw distribution patterns into a form of cooked patterns to see along a pointer direction (such as "next"), if it is distributed by "Block" (which has each processor get a consecutive block of elements), "Cyclic"(which is a round-robin rotation), "Collapse"(which mean no distribution), o r"Block-Cyclic" (which is a round-robin rotation, but each time each processor takes a small block number of elements), as simiarly in the distirbution information in HPF. In our experiments, we have successfully gathered the cooked information, "Block", "Cyclic", "Block-

Cyclic", and "Collapse" for linked list, mesh, range trees[29, 12], etc. For other optimization purpose such as affinity analysis[6], people might be interested in calculating the probability of a pointer (says "next") pointing to a remote reference.

Our distribution analysis algorithm is currently being incorporated into an experimental parallel C++ environment[19, 20, 21] at IBM SP-2 machines to experiment with unstructured and irregular scientific applications with pointer structures. The system is also part of the joint work with Power Mechanics Department, Tsing-Hua university to develop scalable methods to model unstructured gas turbine combustor[30]. Preliminary experimental results done at 16-node IBM SP-2 show that the compiler with the help of our distribution analysis algorithm can significantly improve the performance of pointer-based distributed programs.

The remainder of the paper is organized as follows. Section 2 describes our parallel programming model which allows programmers to build their own distributed data structures with SPMD computation. Section 3 presents our data distribution analysis framework which can obtain distribution patterns of the program written in our model. Next, Section 4 describes the experimental results, and Section 5 describes the related work. Finally, Section 6 concludes this paper.

2 Parallel Program Model

2.1 The Source Parallel Programming Language

In this section, we describe the programming model we provide for users to construct distributed data structures. In the model, the construction of distributed data structures is done in SPMD mode, and after the construction programmers can use it as a shared structure. Here, we can draw an analogy between our model and HPF model. In HPF, we have "extrinsic" functions. In the extrinsic mode, a program is executed in SPMD model, and the program is back to run with shared memory semantics once it gets back from extrinsic mode to the normal mode. The extrinsic mode is used to link with library routines. In our model, programmers or library designers can build a distributed tree with a global name space on distributed memory architectures by constructing a subtree with SPMD computation by each processor (SPMD is a parallel computation model that every processor executes the same program, but different set of data), and then link the subtrees together via distributed pointers. The work done above can be considered in an extrinsic mode. After the construction of the structures, the distributed tree can then be manipulated by users in the normal mode to have shared-memory semantics and global name space.

To build distributed structures in our model, programmers first need to specify the dimension of the processor array, and then three steps are used to construct the distributed data structures. The three step sequence is described below. First, programmers write SPMD programs to construct locally linked data structures. Then, a barrier synchronization is performed after the SPMD parallel step. This is shown as the step 1 in the parallel C++ code below.

```
class object {
  distributed  object *next;
  data   a;
  object();
};
```

```
object::object(){
      distributed object *q, *p, *r, *tail, *PointerArray[#P];
      /* Step  0: Specify how processor id is named */
         SetProcDim(OneDim);
      /* Step  1: SPMD Construct Locally Linked Data Structures  */
         r= NULL; tail =NULL;
         for (i=0; i<N/#P; i++)  {
            q = malloc(sizeof(object));
            if (i==0) tail = q;
            q->next =r;
            r = q;
          }
         barrier();
         /* Step 2: Broadcast */
         broadcast (r,PointerArray);
         /* Step 3: chain up */
         chain_up(tail,PointerArray[(pid+1) mod #P], ``next'');
  }
```

In the second step, a broadcast operation is performed. The broadcast operation is done as follows. Each processor running in SPMD mode provides a pointer pointing to the local linked data structure. The information is then broadcast to all the processor. Each processor then receives a vector of pointer, the value of which comes from the contribution of each processor. After the broadcast operation is done, the kth element of the gathered array coming from processor k.

In the third phase of the program, a chain_up operator is provided so that the program can chain a local pointer with the pointer broadcasted in the previous section. The pointer (such as "next" or "previous") can be specified in the chain_up operator. If two fields are specified, the first one is used as the forward pointer, and the second is used as the backward pointer. The chain_up operator in the above example, will chain PointerArray[(pid+1) mod #p] into the "next" field of pointer tail. That is

$$tail- > next = PointerArray[(pid+1)mod\#p]$$

The three steps above can be repeated arbitrary times. In addition, any one of the step can be omitted by programmers. Broadcast and chain_up are used here as communication primitives in SPMD mode while without compiler support. The model is used as a reference to demonstrate our automatic distribution analysis algorithm.

2.2 Restrictions

We restrict our parallel programming model to be with static control. The concept of the program with static control is first introduced by Feautrier[24]. In the programs, a set of unknown integer variables are allowed. The set of integer variables, which are only defined only once in the program, and whose value depends only on the outside world, are called structure parameters. The programs are with the domain of loop nests where the loops bounds, conditonal expressions in the branch, and array indices are all affine functions of the set of structure parameters.

We allow pointer expression in the program which is not seen in Feautrier[24]. However, we do not allow recursive programs[9]. The distributed data structures built by users have to be done by loops

m_i	$\in MemInst$	$= ProcSet \times LoopIndexSet \times StmtNo$
L	$\in Continuation$	$= StmtNo \times IterSet + StmtNo$
δ	$\in SummaryGraph$	$= (ProcSet \times LoopIndexSet \times StmtNo) \times FieldName \times Continuation \rightarrow MemInst$
τ	$\in MemInstForExp$	$= Expression \times IterSet \times StmtNo \rightarrow MemInst$
Q	$\in Aliases$	$= PointerVar \times IterSet \times StmtNo \rightarrow MemInst$
q	$\in PointerVar$	pointer
s_i	$\in StmtNo$	$= Z^+$
p_i	$\in ProcSet$	$= Z^+$
b, u	$\in IterSet$	$= ProcSet \times LoopIndexSet$
α, β	$\in Expression$	expression of the program
$next$	$\in FieldName$	a field name of a structure
$\#P$	$\in ProcessorNum$	$= N$
I	$\in LoopIndexSet$	$= Z^+$

Figure 1: Variable Notations

instead of recursive programs. As we are interested in demonstrating the essential idea of the existence of a sound solution to the problem, we choose a slightly restricted, but well-documented subset of the programs. We think our work provides a sound theoretic basis for the open issue –"Can the compiler be intelligent enough to automatically understand the distribution pattern of pointer-based distributed objects built by application programmers?". It can not only be used as an important basis for possible future improvements toward full solutions, but also be directly used for a specification language, where a slightly restricted program is acceptable.

3 Distribution Analysis Algorithm

3.1 Notations

In this section, we give the basic notations needed for our distribution analysis algorithm. Figure 1 gives a summary of the variables and notations used. We give the detailed descriptions for the key definitions below. First, to summarize the distribution information of distributed linked structures, we need to give a naming scheme to name the heap allocated objects (by new or malloc). Without loss of generality, throughout this paper, we will only summarize the distribution information of heap allocated objects. Our naming scheme is based on the processor number, loop index, and statement number as follows.

Definition 1 To distinguish different instances of locations created by a dynamical allocation statment in a loop, we use an absolute occurrence number. The location created by loop index $(i_1, i_2, \cdots, i_n)$, processor number, $(p_1, p_2, \cdots, p_k)$, and in the statement number $< s' >$ has the absolute occurrence number $((p_1, p_2, \cdots, p_k), (i_1, i_2, \cdots, i_n), s')$. This instance is denoted as $m_{((p_1,p_2,\cdots,p_k),(i_1,i_2,\cdots,i_n),s')}$.

Consider the following code:

Code Fragment 1

```
l= NULL;
for (i=0; i<M; i++)  {
```

```
S1:        q = malloc(sizeof(object));
S2:        q ->next = l;
S3:        l = q;
    }
L1: /* End of Loop */
```

The memory instance allocated at statement S1 is denoted as $m_{p,i,S1}$, where p is the processor number, i is the loop index, and $S1$ is the statement number. Totally, instances $m_{p,i,S1}, 0 \leq p < \#P, 0 \leq i < (M-1)$, are created in the code above, as the program is executed in SPMD mode, and each processor will execute the code above.

Our goal of this paper is to find a distribution summary for user-defined distributed structures. The summary graph to represent the result of our summary is defined as follows.

Definition 2 A Summary Distribution Graph (SDG) is defined as

$$\delta(((p_1, p_2, \cdots, p_k), (i_1, \cdots, i_n), s), field, continuation) = \begin{cases} m_{(p_1^1, p_2^1, \cdots, p_k^1), (i_1^1, \cdots, i_n^1), s_1} & | \quad \gamma_1 \\ m_{(p_1^2, p_2^2, \cdots, p_k^2), (i_1^2, \cdots, i_n^2), s_2} & | \quad \gamma_2 \\ \cdots & \\ m_{(p_1^m, p_2^m, \cdots, p_k^m), (i_1^m, \cdots, i_n^m), s_m} & | \quad \gamma_m \end{cases}$$

where $(p_1, ..., p_k)$ is the processor number, $(i_1, ..., i_n)$ is the loop index, s is the statement number, k is the number of the dimension of the processor array, n is the depth of the nested loops, γ_i represents a set of integer intervals defined based on the constraints related to $(p_1, p_2, ..., p_k)$ and $(i_1, ..., i_n)$, and *continuation* is the program continuation defined in Figure 1.

For example, in the *Code Fragment 1*, we will have the summary graph as follows.

$$\delta((p, i, S1), next, L1) = \begin{cases} NULL & | \quad i = 0, 0 \leq p < \#P - 1 \\ m_{p,i-1,S1} & | \quad 0 < i < M, 0 \leq p < \#P - 1 \end{cases}$$

To use the notation in Definition 2, we have γ_1 as $(i = 0, 0 \leq p < \#P - 1)$ and γ_2 as $(0 < i < M, 0 \leq p < \#P - 1)$. δ states that the content of the "next" of memory instance $m_{p,i,S1}$ at a given program continuation point, $L1$, is $NULL$ if $(i = 0, 0 \leq p < \#P - 1)$, and $m_{p,i-1,S1}$ if $0 < i < M, 0 \leq p < \#P - 1$. We will show how the summary graph can be gotten in the next sub-sections.

Next, in order to calculate the summary graph, we need two auxiliary functions. The first auxiliary function, Q, is to calculate the alias information (the memory instance) for a pointer variable at a given continuation point. The function is defined as follows.

Definition 3 Given a pointer variable q, and a set program continuation point (b, s), the alias function, Q, returns the alias of q at the program continuation point (b, s). The alias function, Q is defined as

$$Q(q, b, s) = \begin{cases} m_{u_1} & | \quad \gamma_1 \\ m_{u_2} & | \quad \gamma_2 \\ \cdots & \\ m_{u_k} & | \quad \gamma_k, \end{cases}$$

where γ_i represents a set of integer intervals as similarly in Definition 2, m_{u_i} represents a memory instance, q is a pointer variable in the program, b is the iteration index, and s is the statement number.

In the definition above, b represents iteration index. As shown in Figure 1, b belongs to $IterSet$ which is equivalent to $ProcSet \times LoopIndexSet$. We again use the example in *Code Fragment 1* to show the functionality of the function, Q. Suppose we want to find out the aliases of q at statement S3 with iteration number b (let b be euqal to (p, i)).

Then we have

$$Q(q, b, S3) = Q(q, (p, i), S3) = m_{p,i,S1}, \quad 0 \leq i < M - 1, 0 \leq p < \#P.$$

Again, we will show how the alias function can be calculated in the next sub-sections. In addition, suppose we want to find out the aliases of l at statement S2 in *Code Fragment 1*. We then have

$$Q(l, b, S2) = Q(l, (p, i), S2) = \begin{cases} NULL & | \quad i = 0, 0 \leq p < \#P \\ m_{p,i-1,S1} & | \quad 0 < i < M - 1, 0 \leq p < \#P. \end{cases} \tag{1}$$

The second auxiliary function is a function, τ, to find the memory instance for a given program expression at a given program continuation point. It's defined below.

Definition 4 An auxiliary function, τ, is defined as

$$\tau(e, b, s) = \begin{cases} m_{u_1} & | \quad \gamma_1 \\ m_{u_2} & | \quad \gamma_2 \\ \cdots & \\ m_{u_k} & | \quad \gamma_k \end{cases}$$

where γ_i represents a set of integer intervals, and e is an expression in the program, b is the iteration index, m_{u_i} is a memory instance, and s is the statement number.

The auxiliary function, τ, above is similar to the function Q except for that it's more general to report the memory instance for any pointer expression (such as q->next). Finally, we need to define the notation to represent the program precedence order.

Definition 5 Lets_1, s_2 be the statement numbers, and b_1, b_2 be the iteration index. Then we define $(b_1, s_1) \prec (b_2, s_2)$ if (b_1, s_1) is executed before (b_2, s_2) in the program execution order.

3.2 Solutions of Data Distribution Analysis

In this section, we will give the equations to calculate the functions δ, Q, and τ. The result of the function, δ, directly represents the summary of the distributed information of programs. The summary information, δ, in our framework is called "raw distribution information". The raw distribution information we gather is always precise, and can be converted into a variety of "cooked" data distribution patterns, later, for different compiler optimization purposes. Our proposed framework is modelled closely based on the methodologies used in array data flow analysis[7, 23, 24, 26], which are originally used to find the data dependence among array elements. We introduce additional processor concept, pointer and aliases formula, and distribution information into the equations which greatly expand array data flow analysis framework to be able to summarize distribution pattern for pointer-based distributed structures. We give the formula below to calculate δ, Q, and τ, respectively.

Summary Graph Calculation

Suppose we have the following code fragment.

Code Fragment 2

$for\ (i{=}0;\ i < M\ ;\ i{+}{+})\ \{$

$s_1:\quad q_1->next = \alpha_1$

$\quad ...$

$s:\quad ... = malloc(sizeof(object));$

$\quad ...$

$s_2:\quad q_2->next = \alpha_2$

$\quad ...$

$s_k:\quad q_k->next = \alpha_k$

$\}$

END_of_Loop : /* End of Loop */

We want to calculate $\delta((b,s), next, END\_of\_Loop)$, as it summarizes, at the end of the loop, where the "next" field of each memory instance $m_{b,s}$ pointing. (We have b as an iteration index above.) When there is only one memory allocation statement in the program, as in the case of *Code Fragment 2*, we will abbreviate $\delta((b,s), next, END\_of\_Loop)$ as $\delta(b, next, END\_of\_Loop)$ in this paper.

Assume $s_1, s_2, ..., s_k$ are the only statements which have the writes to the "next" field of objects in the program. As we want to summarize the memory instances with the "next" field, we are only interested in the write to the "next" field. Therefore, we only need to consider statements $s_1, s_2, ..., s_k$. To calculate δ, without loss of generality, we can first try to find out the set of iterations and instances at statment s_i writing to the "next" field of memory instance $m_{b,s}$. Let M_{s_i} be the set. Then we have

$$M_{s_i}(b, next) = \{(u, \tau(\alpha_i, u, s_i)) | Q(q_i, u, s_i) = m_{b,s}, u \in IterSet_{s_i}\}.$$

In the formula above, we have $Q(q_i, u, s_i)$ to find out the alias of q_i at iteration u, and to see if the alias happens to be $m_{b,s}$, where $u \in IterSet_{s_i}$, and $IterSet_{s_i}$ represents all the legal iteration indexes for statement s_i. If the alias is $m_{b,s}$, the content of "next" of $m_{b,s}$ will be $\tau(\alpha_i, u, s_i)$. As defined in Definition 4, given an expression α_i, τ will return the memory instance which α_i is representing at the program continuation point (u, s_i). We will show later in this section how to calculate the auxiliary function τ.

At statement s_i, there could be several writes to the memory instance $m_{b,s}$, and we are only interested in the one with the lexicographic maximum (the last write). Let K_{s_i} be the set of the last write. Then we have

$$K_{s_i}(b, next) = Max_{\prec} M_{s_i}(b, next),$$

where $Max_{\prec} M_{s_i}(b, next)$ is defined as follows.

$$\begin{aligned} Max_{\prec} M_{s_i}(b, next) \equiv \{(u, \tau(\alpha_i, u, s_i)) | (u, \tau(\alpha_i, u, s_i)) \in M_{s_i}(b, next)\ and \\ \forall (u', \tau(\alpha_i, u', s_i)) \in M_{s_i}(b, next) \Rightarrow (s_i, u') \prec (s_i, u)\} \end{aligned}$$

But since we have statements, $s_1, ..., s_k$, we need to summarize all the statements to find the last write to the next field of the memory instance $m_{b,s}$. Therefore, we have

$$\delta'(b, next, END - of - Loop) = Max_{\prec}\{K_{s_i}(b, next), i = 1, ..., k\}$$

Finally, the summary graph only needs the second tuple of the above information, so we have

$$\delta(b, next, END-of-Loop) = \{j|(u,j) \in \delta'(b, next, END-Of-Loop)\}.$$

The Calculation of Aliases Identity

Next, we show how to calculate the alias function, Q. Suppose we have the code fragment below.

Code Fragment 3

```
for (i=0; i <M ; i++)
{
s1 :     q = β1
         . . .
s3 :     q = β2
         . . .
sa :     = q . . . /* Use of q */
         . . .
sk :     q = βk
}
```

Suppose q is a pointer variable, and $s_1, s_2, ..., s_k$ are the only statements with writes to the variable q. Then, the aliases of q at statement s_a with iteration number b can be calculated from the write sequence of s_1, s_2,..., s_k.

Let $R_{s_i,s_a}(q,b)$ represents the set of writes produced at statement s_i to potentially become the alias of q at statement s_a with iteration b. Then we have

$$R_{s_i,s_a}(q,b) = \{(u, \tau(\beta_i, u, s_i))|(s_i, u) \prec (s_a, b), u \in IterSet_{s_i}\}.$$

The formula above lists all the legal iteration indexes, u ($u \in Iterset_{s_i}$) such that (s_i, u) precedes (s_a, b), while the alias is $\tau(b_i, u, s_i)$. Each element in $R_{s_i,s_a}(q,b)$ is a potential alias to q at iteration b, but only the one with the lexicographic maximum (the last write) of the set is the real alias of q at iteration b. Let T_{s_i,s_a} represent the set with the last write. Then we have

$$T_{s_i,s_a}(q,b) = Max_{\prec} R_{s_i,s_a}(q,b).$$

$Max_{\prec} R_{s_i,s_a}(q,b)$ is defined as follows.

$$Max_{\prec} R_{s_i,s_a}(q,b) \equiv \{(u, \tau(\beta_i, u, s_i))|(u, \tau(\beta_i, u, s_i)) \in R_{s_i,s_a}(q,b) \; and$$
$$\forall (u', \tau(\beta_i, s_i, u')) \in R_{s_i,s_a}(q,b) \Rightarrow (s_i, u') \prec (s_i, u)\}$$

Note that if the set is empty, that means q is $\perp$ or defined at a previous continuation point. Here, since we have statements, $s_1, ..., s_k$, we need to summarize all the statements. Therefore, we have

$$Q'_{s_a}(q,b) = Max_{\prec}\{R_{s_i,s_a}(q,b), i = 1, ..., k\}$$

Finally, we only need to pick up the second field of each element of Q' to form the set of aliases.

$$Q(q, b, s_a) = \{j|(u,j) \in Q'_{s_a}(q,b)\}$$

Auxiliary Function: τ

Next, we give the equations for the auxiliary function, τ, which finds the memory instance for a given program expression, α, at statement s_i with iteration index b. We have

$$\tau(\alpha, b, s_i) = \begin{cases} m_{b,s_i} & if \quad \alpha\ is\ malloc \\ Q(q, b, s_i) & if \quad \alpha\ is\ q \\ NULL & if \quad \alpha\ is\ NULL \\ \delta(Q(q,b,s_i), next, (s_i, b)) & if \quad \alpha\ is\ q \rightarrow next. \end{cases}$$

The algorithm we gave earlier in this section for the calculation of the summary graph only calculated the summary information at the end of a given for-loop. The summary graph information at the end of the allocation subroutine (such as constructor) is the final result we need for our optimization. However, as we can see in the calculation of the auxiliary function, τ, we might need the summary graph information at arbitrary continuation point of a given program. Therefore, we re-write our formula below to be used at an arbitrary continuation point of a given program.

Summarized Graph Calculation at Arbitrary Continuation Point

Suppose we have the following code fragment.

Code Fragment 4

```
      for (i=0; i <M ; i++)
      {
s :       ... = malloc(...)
          . . .
s1 :      q1-> next = α1
          . . .
sa :      ... = ... /*To calculate the summary graph at this point */
s2 :      q2-> next = α2
          . . .
sk :      qk-> next = αk
      }
```

Assume $s_1, s_2, ..., s_k$ are the only statements which have the writes to the "next" field of objects in the program. As we want to summarize the memory instances with the "next" field, we are only interested in the write to the "next" field. Therefore, we only need to consider statements $s_1, s_2, ..., s_k$. To calculate δ, without loss of generality, we can first try to find out the set of iterations and instances at statment s_i writing to the "next" field of memory instance $m_{b,s}$ preceding the the continuation point (t, s_a), where t is the iteration index. Let M_{s_i,s_a} be the set. Then we have

$$M_{s_i,s_a}(b, next, t) = \{(u, \tau(\alpha_i, u, s_i)) | (s_i, u) \prec (s_a, t), Q(q_i, u, s_i) = m_{b,s}, u \in IterSet_{s_i}\}.$$

At statement s_i, there could be several writes to the memory instance $m_{b,s}$, and we are only interested in the one with the lexicographic maximum (the last write). Let K_{s_i,s_a} be the set of the last write. Then we have

$$K_{s_i,s_a}(b, next, t) = Max_{\prec} M_{s_i,s_a}(b, next, t).$$

where $Max_{\prec} M_{s_i,s_a}(b, next, t)$ is defined as follows.

$$Max_{\prec}M_{s_i,s_a}(b,next,t) \equiv \{(u,\tau(\alpha_i,u,s_i))|(u,\tau(\alpha_i,u,s_i)) \in M_{s_i,s_a}(b,next,t)\ and$$
$$\forall(u',\tau(\alpha_i,u',s_i)) \in M_{s_i,s_a}(b,next,t) \Rightarrow (s_i,u') \prec (s_i,u)\}$$

But since we have statements, $s_1, ..., s_k$, we need to summarize all the statements. Therefore, we have

$$\delta'(b,next,(t,s_a)) = Max_{\prec}\{K_{s_i,s_a}(b,next,t), i = 1,...,k\}.$$

The summary graph only needs the second tuple of the above information, so we have

$$\delta(b,next,(t,s_a)) = \{j|(u,j) \in \delta'(b,next,(t,s_a))\}.$$

3.3 Examples

In the following, we give two examples to demonstrate how our framework works. The systematic solver to solve the problem will be given in the next sub-section by using parametric integer programming.

Our first example is a program to build a one dimensional linked list as follows.

Code Fragment 5

```
      UserObject::UserObject(){
         distributed UserObject *q, *r,*tail, *PointerArray[#P];
  L0:    r=NULL, tail =NULL;
  L1:    for (i=0; i < (N/#P)-1; i++) {
  S1:       q = malloc(sizeof(UserObject));
  S2:       if (i==0) tail = q;
  S3:       q->next = r;
  S4:       r = q;
         }
  L2:    /* Barrier */
  L3:    broadcast (r, PointerArray);
  L4:    chain_up(tail, PointerArray[(pid+1) mod #P],''next'');
  L5:  }
```

We need to first calculate the summary graph at the program point "L2". In addition, the only write to the "next" field of the objects is in statement S3, so we have

$$\delta(b,next,L2) = Max_{\prec}M_{S_3}(b,next,L2). \tag{2}$$

We then calculate M_{S3}.

$$M_{S3}(b,next,L2) = \{(u,\tau(r,u,S3))|Q(q,u,S3) = m_{b,s_1}, u \in IterSet_{S3}\} \tag{3}$$

Next, we calculate $Q(q,u,S3)$. After a series of calculations according to the formula in Section 3.2, we have

$$Q(q,u,S3) = m_{u,S1}. \tag{4}$$

We then calculate $\tau(r,u,S3)$ as follows.

$$\tau(r,u,S3) = Q(r,u,S3)$$

Next, we have

$$Q(r,(pid,i),S3) = \begin{cases} NULL & | \quad i = 0 \\ Q(q,(pid,i-1),S4) & | \quad 0 < i < (N/\#P)\&\&(0 \leq pid \leq (\#P-1)) \end{cases} \tag{5}$$

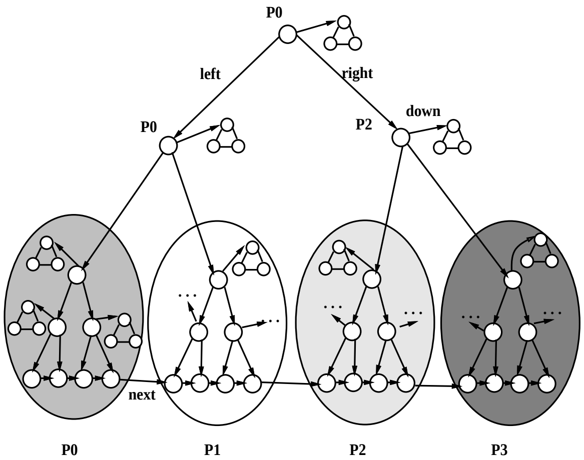

Figure 2: Two Dimensional Range Tree

Combining equations 2, 3, 4, and 5, we get

$$\delta((pid, i, S1), next, L3) = \begin{cases} m_{pid,i-1,S1} & | \quad 0 < i < (N/\#P) \&\& (0 \leq pid \leq (\#P-1)) \\ NULL & | \quad i = 0 \end{cases} \tag{6}$$

Next, we need to do the broadcast and chain-up operations, and we need to know where r and $tail$ are pointing to.

$$Q(r, pid, L2) = Q(q, pid, L2) = m_{pid,N/\#P-1}$$

$$Q(tail, pid, L2) = m_{pid,0}$$

The chain-up operation basically does the following

$$M_{(pid,0,S1)} - > next = m_{(pid+1)mod\#P,N\#P-1,S1} \tag{7}$$

Combining results at (6) and (7), we have the summary information as follows.

$$\delta((pid, i, S1), ''next'', L5) = \begin{cases} m_{pid,i-1,S1} & | \quad 0 < i < N/\#P-1, 0 \leq pid \leq (\#P-1) \\ m_{(pid+1)mod\#P,N/\#P-1,S1} & | \quad i = 0, 0 \leq pid \leq (\#P-1) \end{cases} \tag{8}$$

The distribution information we get from (8) is a precise information, and is a "raw" information. The raw distribution information can then be converted into different "cooked" data distribution patterns depending on the purpose of compiler optimizations. In this example, we can further infer that the distribution along the "next" direction is by "Block" distribution.

Next, we will look at our second example to build a two dimensional range tree[29][12]. Figure 2 illustrates an example of the construction of a distributed range tree. An interesting multi-dimensional tree structure which is a binary tree of binary tree, and the leave node is again linked together. The source program of the construction of the distributed range tree in SPMD mode is listed in Appendix. We first calculate the summary information at program point L1. It summarizes the lower part of the subtree at each processor.

$$\delta((p, i, j, S1), left, L1) = \begin{cases} m_{p,i+1,2*j,S1} & | \quad 0 \leq i < (N-1) \&\& 0 \leq j < 2^i \\ NULL & | \quad i = N-1, 0 \leq j < 2^i \end{cases} \tag{9}$$

$$\delta((p, i, j, S1), right, L1) = \begin{cases} m_{p,i+1,2*j+1,S1} & | \quad 0 \leq i < (N-1) \&\& 0 \leq j < 2^i \\ NULL & | \quad i = N-1, 0 \leq j < 2^i \end{cases} \tag{10}$$

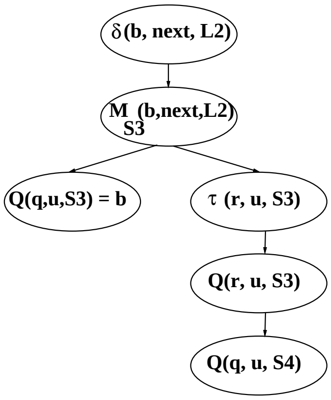

Figure 3: The Unfolded Tree of the Call-Graph

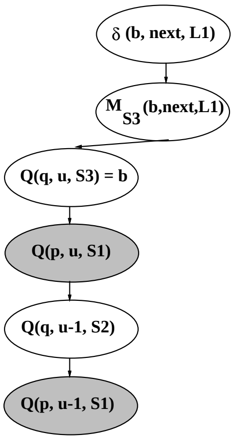

Figure 4: The Unfolded Tree with a Cycle

Next, we will try to summarize the information at program point L2. It summarize the "next" information in the leave nodes. We have

$$\delta((p, N-1, j, S1), next, L2) = \begin{cases} m_{p,N-1,j+1,S1} & | \quad 0 \le j < (2^{N-1}) - 1 \\ m_{p+1,N-1,0,S1} & | \quad 0 \le p < \#P-1, j = 2^{N-1} \\ NULL & | \quad p = \#P-1, j = 2^{N-1}. \end{cases} \tag{11}$$

Next, we summarize the information at program point L3. It constructs the upper tree of the distributed tree. We have

$$\delta((p, stride, S2), left, L3) = \begin{cases} m_{p,0,0,S1} & | \quad stride = 1, (p\ mod(stride * 2)) = 0 \\ m_{p,stride/2,S2}, & | \quad 1 < stride < \#P, (p\ mod\ (stride * 2)) = 0 \end{cases} \tag{12}$$

$$\delta((p, stride, S2), right, L3) = \begin{cases} m_{p+stride,0,0,S1} & | \quad stride = 1, (p\ mod\ (stride * 2)) = 0 \\ m_{p+stride,stride/2,S2} & | \quad 1 < stride < \#P, (p\ mod\ (stride * 2)) = 0. \end{cases} \tag{13}$$

Similarly, we can summarize the subtree in the "down" direction. In addition, because the summary information in (9), (10), (11), (12), and (13) do not over-write each other, so the final summary information is the add up of those information. If they over-write each other, we need to again calculate the set of the last write elements. The distribution information we get above is a precise, but a "raw" information. We can further conclude that along the "next"direction, the distribution is by block. In addition, if we will follow the "left" pointer, it will never cross processor boundaries to reach remote references, and similarly, it will not cross the processor boundary in the "down"direction.

3.4 Analysis of the Formula

Our equations developed in Section 3.2 can be solved through a sequence of applications of parametric integer programming. The parametric integer programming[23] is originally developed for array data flow analysis. It's basically a form of integer programming, but allowing unknow variables in the equation so that you will get a parametric solution for the unknown variables in your program. Figure 3 shows the unfolded call-graph in calculating $\delta(b, next, L2)$ in the first example of our previous sub-section. In the process, δ, Q, and τ, might call each other. The calling sequences in Figure 3 results in no cycles as eventually every part of the call-graph develops into a leaf node. In this case, we can apply parametric integer programming to the leaf nodes first, and then apply them into the upper level of the tree one by one. Theorem 1 below states the result.

Theorem 1 *If the unfolded call-graph in calculating summary distribution graph, δ, is cycle-free, the function, δ, can be calculated via a sequence of applications of parametric integer programming (PIP).*

Proof: The formal proof will be given in the final version of the paper.

Next, let's consider the code fragment below.

Code Fragment 6

```
Sa : p = malloc(...);
for (i = 0; i <M ; i++)
{
S1 :      q = p
          . . .
S2 :      p = q
          . . .
S3 :      q-> next = ...
}
L1 : /* To calculate the Summary Information here */
```

Figure 4 shows part of the unfolded call-graph in calculating $\delta(b, next, L1)$ for the code above. We have a cycle in calculating the alias of p that it again depends on the alias of p. We call such a graph having a cycle. Fortunately, the cycle case is a trivial case of our problem. For example, in this case, we have

$$Q(p, u, S1) = \begin{cases} m_{S_a} & | \quad u = 0 \\ Q(p, u-1, S1) & | \quad u > 0 \end{cases}$$

And we can can reduce the equations above into the equation below to avoid the cycle problem.

$$Q(p, u, S1) = m_{S_a}, u \geq 0$$

Theorem 2 *If the unfolded call-graph of the summary distribution graph, δ, has a cycle of the calling sequence of the alias identity function, Q, a cycle-reduction technique can be used to break the cycle.*

Proof: The formal proof will be given in the final version of the paper.

Combining the Theorem 1 and Theorem 2, we have the results that our function, δ, can always be calculated by applying a sequence of PIP (parametric integer programming) solvers.

4 Experiments

Our distribution analysis algorithm is currently being incorporated into an experimental parallel C++ environment[19, 20, 21] at IBM SP-2 machines to experiment with unstructured and irregular scientific applications with pointer structures. Preliminary experimental results done at 16-node IBM SP-2 (code translation partly by hand codes) show that the compiler with the help of our distribution analysis algorithm can significantly improve the performance of pointer-based distributed programs.

Figure 5 gives an illustration of our experiment with a program performing a smoothing algorithm on a 2-d linked mesh. It's used to demonstrate performance benefits for the compiler to have the distribution

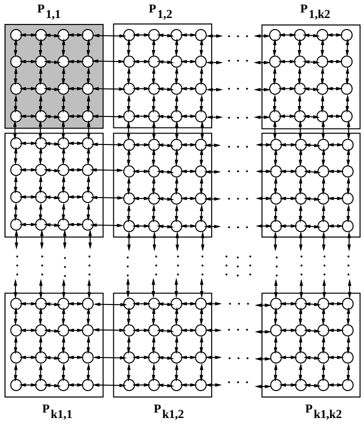

Figure 5: Two Dimensional Linked Mesh Structures

Local Block Size	512x512	256x256	128x128
A	218.7	66.3	25.4
B	70.29	19.78	6.67
C	55.76	12.73	3.38
A/B Improvement	3.11	3.35	3.80
B/C Improvement	1.26	1.55	1.97

Table 1: The execution time (second) of smoothing algorithm on a 16-node SP-2 machine.

information obtained from our algorithm. Suppose that programmers decide to build a linked mesh distributed among processors. Figure 5 gives an illustration of such a structure. After the linked structure created, programmers will write the smoothing algorithms on the global name space as though there are shared memories available.

Table 1 shows the preliminary performance results we get from a smoothing algorithm by running 200 iterations. Experimental results are done on a 16-node IBM SP-2 machine. The IBM SP-2 machine is a workstation clustered connected via fast switches. There are three experimental data. First, in the version A, the distribution analysis algorithm is not incorporated into the compiler. The program does not have the knowledge that where the boundary is. Therefore, the program has to check each pointer reference to see if it's a local reference or a remote reference. A message passing scheme with the efforts of two round trips is used to request a remote reference for a local processor. This results in a very slow program, as there is little support from the compiler to emulate the global name space on distributed memory architectures.

In the version B, the distribution analysis algorithm is incorporated into the compiler. The compiler recognizes the distribution of the programs, and partitions the smoothing algorithm into two loops, one is with boundary areas, and the other is completely with local computation. In such cases, the program does not need to check each pointer reference to be local or remote. It significantly out-performs the version without distribution analysis algorithms. Finally, in the version C, the distribution analysis algorithm is also incorporated into the compiler. The compiler further utilizes the distribution analysis to group messages in blocks. It further improves the program performance.

5 Related Work

Previously, the research work of pointer analysis[25, 12, 13, 2, 8] have mainly focused on gathering the aliases information of pointer programs on flat shared memory parallel machines, and do not provide the distribution information needed for compiler optimization on distributed memory parallel architectures. The work done in [9, 14] can infer the shape of the data structures to be a tree, linked list, or a graph, but still do not provide the processor concept and distribution information for distributed memory architectures. In the treatment of distributed pointer problems, *Olden* project[6] addressed the importance of gathering the information of affinity analysis (the probability of a pointer pointing to a remote reference) in their optimizations with distributed pointer-based programs. Their system allows programmers to provide a quantified hint to the compiler, while our distribution analysis algorithm makes the compiler understand the information automatically. In addition, our distribution information can also be used for *Olden's* compiler optimization. Rinard and Diniz presented a commutativity analysis work[27] which can also optimized programs by analyzing the commutativity properties of distributed programs.

Our framework in finding the distribution pattern for programs is modelled based on the methodologies used in array data flow analysis[7, 23, 24, 26], which are originally used to find the data dependence among array elements. We introduce additional processor concept, pointer and aliases formula, and distribution information into the equations which greatly expand array data flow analysis framework to be able to summarize distribution pattern for pointer-based distributed structures. The array data flow analysis framework was known to be able to find precise array data flow information in the domain of loop nests where the loops bounds, conditional expressions in the branch, and array indices are affine functions of loop indices. Therefore, our analysis algorithm is able to always find precise data distribution information. The "precise" information means that at any given program continuation point, we can identify the exact memory instance which a pointer is pointing to, or exact memory instance which a field of a memory instance is pointing to. Traditionally, in scalar data flow equation and alias analysis[25], conservative estimation will return a set of memory instances which a pointer might be pointing to, while in our case, we will always return a unique one.

6 Conclusion

In this paper, we present a parallel programming model which allows application programmers to build pointer-based distributed objects at application levels. We then propose a distribution analysis algorithm which can automatically summarize the distribution pattern of pointer-based distributed objects built by application programmers. Our work, to our best knowledge, is the first work to attempt to address this open issue. Our distribution analysis framework employs Feautrier's parametric integer programming as the basic solver, and can always obtain precise distribution information from the class of programs written in our parallel programming model. Experiments results done at 16-node IBM SP-2 show that the compiler with the help of distribution analysis algorithm can significantly improve the performance of pointer-based distributed programs.

References

[1] A. Aho, R. Sethi, J. Ullman. *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1985.

[2] Francois Bodin. *Data Structure Analysis in C Programs*, Technical Reports, IRISA, 1991.

[3] Bodin, F., Beckman, P., Gannon, D., Narayana, S., and Yang, S. Distributed pC++: Basic Ideas for an Object Parallel Language, *Scientific Programming*, 2(3), Fall 1993.

[4] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lummetta, T. von Eicken, and K. Yelick. *Parallel Programming in Split-C*, In Proceedings of Supercomputing '93, pp. 262-273, 1993.

[5] Bozkus, Z., Choudhary, A., Fox, G., Haupt, T., Ranka, S., and Wu, M. Y. Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers, *J. Parallel Distrib. Comput.*, 21, 1 (April 1994), 15-26.

[6] M. C. Carlisle, A. Rogers. *Software Caching and Computation Migration in Olden*, Proceedings of ACM SIGPLAN Conference on Principles and Practice of Parallel Programming, pp. 29-39, July 1995.

[7] J. F. Collard, P. Feautrier. *Fuzzy Array DataFlow Analysis*, Proceedings of ACM SIGPLAN Conference on Principles and Practice of Parallel Programming, pp. 92-101, July 1995.

[8] Vincent A. Guarna, Jr. *A Technique For Analyzing Pointer and Structure References in Parallel Restructuring Compilers*, International Conference on Parallel Processing, pp. 212-220, 1988.

[9] William Harrison. *The Interprocedural Analysis and Automatic Parallelization of Scheme Programs*, Lisp and Symbolic Computation, 2:3/4, pp. 179-396, 1989.

[10] Gwan-Hwan Hwang, Jenq Kuen Lee, and Dz-Ching Ju. An Array Operation Synthesis Scheme to Optimize Fortran 90 Programs, Proceedings of ACM SIGPLAN Conference on Principles and Practice of Parallel Programming, pp. 112-122, July 1995.

[11] Gwan-Hwan Hwang, Jenq Kuen Lee, and Dz-Ching Ju. Array Operation Synthesis to Optimize HPF, Proceedings of International Conference on Parallel Processing, Aug. 1996.

[12] L. Hendren, J. Hummel, A. Nicolau. *Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs*, Proceedings of ACM SIGPLAN PLDI, 1992.

[13] J. Hummel, L. Hendren, A. Nicolau. *A General Data Dependence Test for Dynamic, Pointer-Based Data Structures*, Proceedings of ACM SIGPLAN PLDI, 1994.

[14] R. Ghiya and L. Hendren. *Is it a Tree, a DAG or Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C*, Proceedings of ACM POPL, 1996.

[15] Dennis Gannon, Vincent Guarna, and Jenq Kuen Lee, *Static Analysis and Runtime Support for Parallel Execution of C*, in Languages and Compilers for Parallel Computing, David Gelernter, Alexandru Niclau, and David Padau, editors, The MIT Press, pp. 254-274, 1990.

[16] P. Karamcheti, Andrew Chien. *Concert – Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware*, Supercomputing '93, 1993.

[17] Koelbel, C. Compile-Time Generation of Regular Communications Patterns, *Proc. Supercomputing '91*, Albuquerque, Nov. 18-22, 1991, pp. 101-110.

[18] Koelbel, C., Loveman, D., Schreiber R., Steele, G., and Zosel, M. *The High Performance Fortran Handbook*, MIT-press, Cambridge, 1994.

[19] Jenq Kuen Lee, In-Kuen Tsaur, San-Yih Hwang. *Parallel Array Object I/O Support on Distributed Environments*, to appear in Journal of Parallel and Distributed Computing, 1997.

[20] Lee, J. K., and Ho, D. Integrating Lattice Parallelism into a Collection-Oriented Language, *CD-ROM Proc. HPC Asia '95*, Taipei, Taiwan, Sep. 1995.

[21] Jenq Kuen Lee and D. Gannon. *Object-Oriented Parallel Programming: Experiments and Results*, *Proceedings of Supercomputing '91*, New Mexico, November, 1991.

[22] J. Li and M. Chen. *The data alignment phase in compiling programs for distributed-memory machines*, Journal of parallel and Distributed Computing, Vol. 13, pp. 213-221, Oct. 1991.

[23] Paul Feautrier. *Parametric Integer Programming*, RAIRO Recherche Operationnelle, 22:243-268, Sep. 1988.

[24] Paul Feautrier. *Dataflow Analysis of Array and Scalar References*, International Journal of Parallel Programming, 20(1):23-52, Feb. 1991.

[25] W. Landi, B. Ryder, S. Zhang. *Interprocedural Modification Side Effect Analysis with Pointer Aliasing*, Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 56-68, 1993.

[26] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. *Array Data-FLow Analysis and Its Use in Array Privatization*, Proceedings of Principles of Programming Language, 1993.

[27] M. Rinard, Pedro Diniz. *Commutativity Analysis: A New Analysis Framework for Parallelizing Compilers*, Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 54-66, 1996.

[28] M. Rosing, . Schnabel, R. Weaver. *Scientific Programming Languages for Distributed Memory Multiprocessors: Paradigms and Research Issues*, Languages, Compilers, and Runtime Environments for Distributed Memory Machines, editors, Joel Saltz, P. Mehrotra, North Holland, 1992.

[29] Hanan Samet. *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1990.

[30] Shieh, Y. L., Lee, J. K., Tsai, J. H., and Lin, C. A. Computations of Three-Dimensionally Evolving Jets with Vortex Methods on Parallel Machines with Distributed Memory, In Acer, A. et al. (Eds.). *Parallel Computational Fluid Dynamics: New Algorithms and Applications*, Elsevier Science, 1995, pp. 119-126.

Appendix

An Example with a Multi-Dimensional Range Tree.

```
   /*  Build the local subtree */
   for (i=0; i<N; i= 2*i)
     {
       /* To build one level subtree */
       for (j=0; j <= 2**i; j++)
         {
S1:         next_nodes[j]= new node;
            if (i!= (N-1) next_nodes[j]->down =Alloc_2ndDimTree();
            if ((i==0)&&(j==0)) root = next_nodes[0];
            else {
              if (j % 2) cur_nodes[j/2]->left=next_nodes[j];
              else       cur_nodes[j/2]->right=next_nodes[j];
            }
         }
        /*  House-keeping before going into next level */
       for (k=1; k<2*i; k++)
         {
            cur_nodes[k]= next_nodes[k];
         }
       }
L1:   /* Chain-Up the next pointer in the leave nodes */
      for (i=1; i< 2^N-1;i++)
        cur_nodes[i]->next=cur_nodes[i+1];
      broadcast(ProcArray, cur_nodes[0]);
      chain_up(cur_nodes[2^n],ProcArray[pid+1],''next'');
```

```
L2:   /* Link up the local subtree between each processor */
       for (stride=1; stride<=#P, stride= stride*2)
         {
            broadcast(ProcArray,root);
            if ((pid % stride)==0)   {
S2:              root1= new node;
                 root1->down =Alloc_2ndDimTree();
                 root1->left =root;
                 chain_up(root1,ProcArray[pid+stride], ``right'');
                 root=root1;
             }
          }
L3:   /* END of Program */
      }

Alloc_2ndDimTree() {
   node *root,rrot1, cur_nodes[MAXSIZE],next_nodes[MAXSIZE];
   for (i=0; i<M; i= 2*i)
     {
       /* To build one level subtree */
       for (j=0; j <= 2**i; j++)
         {
S3:         next_nodes[j]= new node;
            if ((i==0)&&(j==0)) root = next_nodes[0];
            else {
              if (j % 2) cur_nodes[j/2]->left=next_nodes[j];
              else       cur_nodes[j/2]->right=next_nodes[j];
            }
         }
        /*  House-keeping before going into next level */
       for (k=1; k<2*i; k++)
         {
            cur_nodes[k]= next_nodes[k];
         }
       }
      /* Chain-Up the next pointer in the leave nodes */
      for (i=1; i< 2**M-1;i++)
        cur_nodes[i]->next=cur_nodes[i+1];
      return(root);
}
```