A Manual for the CHAOS Runtime Library *

Joel Saltz   Ravi Ponnusamy   Shamik D. Sharma   Bongki Moon
Yuan-Shin Hwang   Mustafa Uysal   Raja Das

UMIACS and Dept. of Computer Science,
University of Maryland,
College Park, MD 20742
dybbuk@cs.umd.edu

Abstract

Procedures are presented that are designed to help users efficiently program irregular problems (e.g. unstructured mesh sweeps, sparse matrix codes, adaptive mesh partial differential equations solvers) on distributed memory machines. These procedures are also designed for use in compilers for distributed memory multiprocessors. The portable CHAOS procedures are designed to support dynamic data distributions and to automatically generate send and receive messsage by capturing communications patterns at runtime.

*The CHAOS project was sponsored in part by ARPA (NAG-1-1485), NSF (ASC 9213821) and ONR (SC292-1-22913).

Fact Sheet

Principal Investigator:
Joel Saltz

The CHAOS Team:

Raja Das	Yuan-Shin Hwang
Bongki Moon	Ravi Ponnusamy
Shamik Sharma	Mustafa Uysal

Computer Science Department
University of Maryland
College Park, MD 20742

{saltz,raja,shin,bkmoon,pravi,shamik,uysal}@@cs.umd.edu

Research Collaborators:

Rice University

Reinard von Hanxeleden	Seema Hiranandani
Charles Koelbel	Ken Kennedy

Syracuse University

Alok Choudhary	Geoffrey Fox

Sanjay Ranka

To Obtain CHAOS Software:
Anonymous ftp: `hyena.cs.umd.edu`
e-mail: {bkmoon,shamik,shin,uysal}@@cs.umd.edu

Bug Reports:
e-mail: {bkmoon,shamik,shin,uysal}@@cs.umd.edu

Contents

1 Introduction

A suite of procedures has been designed and developed to efficiently solve a class of problems commonly known as irregular problems. Irregular concurrent problems are a class of irregular problems which consist of a sequence of concurrent computational phases. Patterns of data access and computational cost of each phase of these types of problems cannot be predicted until runtime. This prevents compile time optimization. In this class of problems, once runtime information is available, data access patterns are known in advance making it possible to utilize a variety of profitable pre-processing strategies. Runtime support compilation methods are being developed that are applicable to a variety of unstructured problems including explicit multigrid unstructured computational fluid dynamic solvers, molecular dynamics codes (CHARMM, AMBER, GROMOS, etc.), diagonal or polynomial preconditioned iterative linear solvers, direct simulation Monte Carlo (DSMC) codes, and particle-in-cell (PIC) codes. These problems share the characteristics of (1) arrays accessed through one or more levels of indirection, and (2) formulation of the problem as a sequence of loop nests which prove to be parallelizable.

The CHAOS procedures described in this manual can be viewed as forming a portion of a portable, compiler independent, runtime support library. The CHAOS libray has been written in C, however, interfaces have been provided to call CHAOS library routines from Fortran application programs as well. *This manual presents CHAOS procedures for Fortran application codes.*

1.1 Getting the CHAOS Library

The CHAOS procedures presented in this manual and related technical papers can be obtained from the *anonymous* ftp site `hyena.cs.umd.edu`.

`Example codes shown in this manual are distributed along with CHAOS software.`

1.2 Sneak Preview: Problems, Data Structures, and Procedures

An example of an irregular problem is presented in this section; CHAOS data structures and procedures to parallelize this example are introduced. Figure 1 illustrates a simple sequential Fortran irregular loop (loop L2) which is similar in form to loops found in unstructured computational fluid dynamics (CFD) codes and molecular dynamics codes. In Figure 1, arrays **x** and **y** are accessed by indirection arrays **edge1** and **edge2**. Note that the data access pattern associated with the inner loop, loop L2 is determined by integer arrays **edge1** and **edge2**. Because arrays **edge1** and **edge2** are not modified within loop L2, L2's data access pattern can be anticipated prior to executing it. Consequently, **edge1** and **edge2** are used to carry out preprocessing needed to minimize communication volume and startup costs. Since large data arrays are associated with a typical fluid dynamics problem, the first step in parallelizing involves partitioning data arrays x and y. The next step involves assigning equal amounts of work to processors to maintain load balance.

1.2.1 Translation Table, Dereference, and Schedule

On distributed memory machines large data arrays may not fit in a single-processor's memory hence they are divided among processors. Also computational work is divided among individual processors to achieve parallelism. Once distributed arrays have been partitioned, each

```
  C  Outer loop L1
L1  do i = 1, n_step
     ...
  C  Sweep Over Edges: Inner Loop L2
L2   do i=1, nedge
        n1 = edge1(i)
        n2 = edge2(i)
        y(n1) = y(n1) + f(x(n1), x(n2))
        y(n2) = y(n2) + g(x(n1), x(n2))
      end do
     ...
     end do
```

Figure 1: An Example code with an Irregular Loop

processor ends up with a set of globally indexed distributed array elements. Each element in a size N distributed array, A, is assigned to a particular home processor. In order for another processor to be able to access a given element, $A(i)$, of the distributed array the home processor and local address of $A(i)$ must be determined. Generally, unstructured problems solved with irregular data distributions perform more efficiently than with a regular data distribution such as BLOCK. In the case of irregular data distribution, a lookup table called *translation table* is built that for each array element, lists the home processor and the local address.

Memory considerations make it clear that it is not always feasible to place a copy of the translation table on each processor, so the translation table must be distributed between processors. This is accomplished by distributing the translation table by blocks, i.e., putting the first N/P elements on the first processor, the second N/P elements on the second processor, and so on, where P is the number of processors.

When an element $A(m)$ of distributed array A is accessed, the home processor and local offset are found in the portion of the distributed translation table stored in processor $((m-1)/N)*P+1$. A translation table lookup aimed at discovering the home processor and the offset associated with a global distributed array index is referred to as a *dereference request.*

Consider the irregular loop L2 in Figure 1 that sweeps over the edges of a mesh. In this case, distributing data arrays **x** and **y** corresponds to partitioning the mesh vertices; partitioning loop iterations corresponds to partitioning edges of the mesh. Hence, each processor gets a subset of loop iterations (edges) to work on. An edge i that has both end points ($edge1(i)$ and $edge2(i)$) inside the same partition (processor) requires no outside information. On the other hand, edges which cross partition boundaries require data from other processors. Before executing the computation for such an edge, processors must retrieve the required data from other processors.

There is typically a non-trivial communication latency, or message startup cost, on distributed memory machines. communication can be aggregated to reduce the effect of communication latency; software caching can be done to reduce communication volume. To carry out either optimization, it is helpful to have a-priori knowledge of data access patterns. In irregular problems, it is generally not possible to predict data access patterns at compile time. For example, the values of indirection arrays **edge1** and **edge2** of loop L2 in Figure 1 are known only at runtime because they depend on the input mesh. During program execution, data references of distributed arrays are pre-processed. On each processor, data needed to be exchanged are pre-computed. The results of this pre-processing is stored in a data structure called *communication schedule*. The process of analyzing the indirection arrays and generating schedules is called the *inspector* phase.

Each processor uses communication schedules to exchange required data before and after executing a loop. The same schedules can be used repeatedly, as long as the data reference patterns remain unchanged. The process of carrying communication and computation is called the *executor* phase.

1.2.2 Computing Schedules

This section presents a discussion on the process of generating and using schedules to carry out communication vectorization and software caching. Consider the example shown in Figure 1. Arrays **x**, **y**, **edge1** and **edge2** are partitioned between the processors of the distributed memory machine. The local size of data arrays **x** and **y** on each processor is **local_nnode** and indirection arrays **edge1** and **edge2** is **local_nedge**. It is assumed that arrays **x** and **y** are distributed in the same fashion and the distribution is stored in a distributed translation table. The partitioned indirection arrays **edge1** and **edge2** are called **part_edge1** and **part_edge2** respectively. To compute schedules, the local data array references (local indirection array values) are collected in an array and passed to the procedure *localize*.

In loop L2 of Figure 1, values of array **y** are updated using the values stored in array **x**. Hence, a processor may need an off-processor array element of **x** to update an element of **y**; it may update an off-processor array element of **y** also. The goal of the inspector is to pre-fetch off-processor data items before executing the loop and carry out off-processor updates after executing the loop. Hence, two sets of schedules are computed in the inspector : 1) *gather schedules* – communication schedules that can be used to fetch off-processor elements of **x**, and 2) *scatter schedules* – communication schedules that can be used to send updated off-processor elements of **y**. However, the arrays **x** and **y** are referenced in an identical fashion in each iteration of the loop L2 and also they are identically distributed, so a single schedule that represents data references of either **x** or **y** can be used for both fetching off-processor elements of **x** and sending off-processor elements of **y**.

Figure 2 contains the pre-processing code for the simple irregular loop L2 shown in Figure 1. The distribution of data arrays **x** and **y** are stored in a translation table **itable**. The globally indexed reference pattern used to access arrays **x** and **y** are collected (Loop K1) in an array **ig_ref**. The procedure *localize* dereferences the index array **ig_ref** to get the addresses and translates **ig_ref** so that valid references are generated when the loop is executed.

Off-processor elements are stored in a on-processor buffer area. The buffer area for each data array immediately follows the on-processor data for that array. For example, the buffer for data array **y** begins at **y(local_nnode+1)**. Hence, when *localize* translates **ig_ref** to **localized_edge**, the off-processor references are modified to point to buffer addresses. The procedure *localize* uses a hashtable to remove any duplicate references to off-processor elements so that only a single copy of each off-processor datum is transmitted. When the off processor data is

```
     ...
  C  Inspector Phase: build translation table, compute schedule and translate indices
 S1  itable = build_translation_table(1, local_indices, local_nnode)
 K1  do i = 1,local_nedge
       ig_ref(i) = local_edge1(i)
       ig_ref(local_nedge+i) = local_edge2(i)
     end do
 S2  call localize(itable,isched,ig_ref,localized_edge, 2*local_nedge, n_off_proc,local_nnode,1)
 K2  do i = 1, local_nedge
       local_edge1(i) = localized_edge(i)
       local_edge2(i) = localized_edge(local_nedge+i)
     end do
  C  Executor Phase: carry out communication and computation
 S4  call zero_out_buffer(y(local_nnode+1), off_proc)
 S5  call gather(x(local_nnode+1), x, isched)
 K3  do i=1, local_nedge
       n1 = local_edge1(i)
       n2 = local_edge2(i)
       y(n1) = y(n1) + f(x(n1) , x(n2))
       y(n2) = y(n2) + g(x(n1) , x(n2))
     end do
 S7  call scatter_add(y(local_nnode+1), y, isched)
     ...
```

Figure 2: Node Code for Simple Irregular Loop

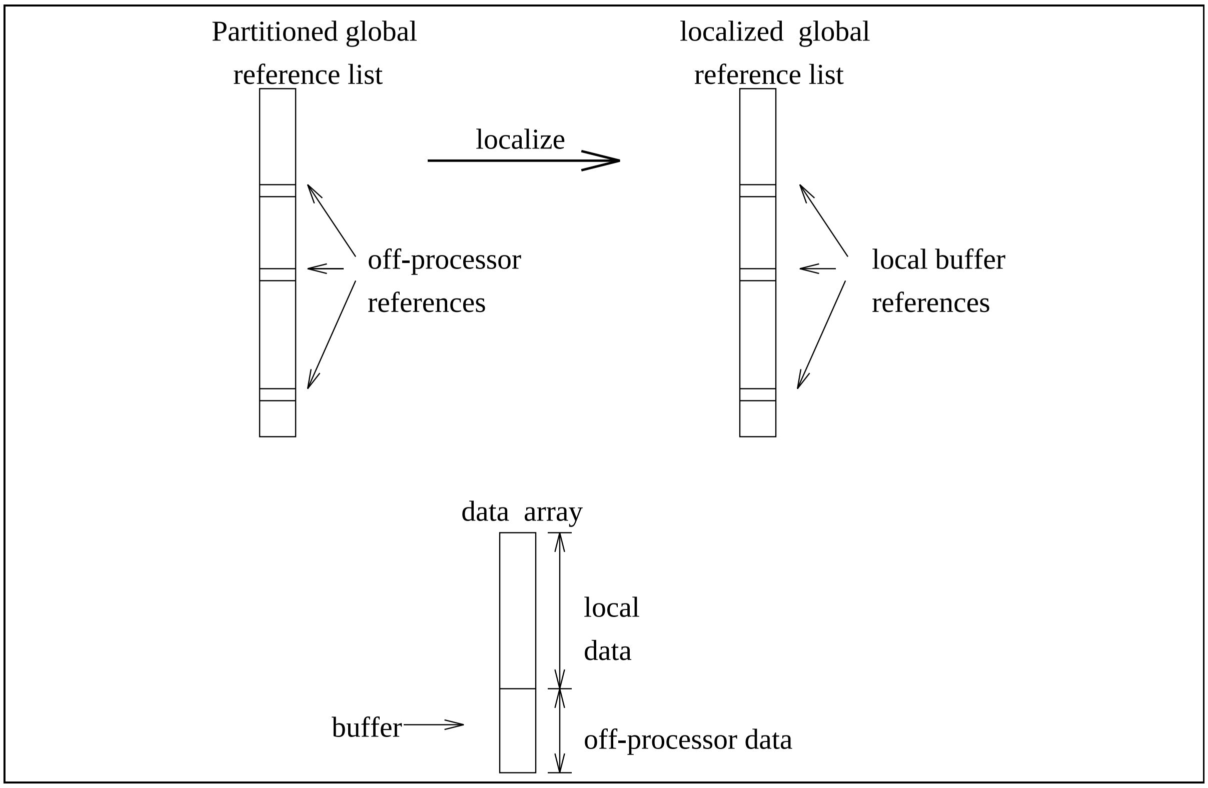

Figure 3: Index Translation by Localize Mechanism

collected into the buffer using the schedule returned by *localize*, the data is stored in a way such that execution of the loop using the **local_edge1** and **local_edge2** accesses the correct data. A sketch of how the procedure *localize* works is shown in Figure 3.

The executor code starting at S4 in Figure 2 carries out the actual loop computation. In this computation the values stored in the array **y** are updated using the values stored in **x**. During computation, accumulations to off-processor locations of array **y** are carried out in the buffer associated with array **y**. This makes it necessary to initialize the buffer corresponding to off-processor references of **y**. To perform this action the function *zero_out_buffer* is called. After the loop's computation, data in the buffer location of array **y** is communicated to the home processors of these data elements (*scatter_add*). There are two potential communication points in the executor code, i.e. the *gather* and the *scatter_add* calls. The *gather* on each processor fetches all the necessary **x** references that reside off-processor. The *scatter_add* calls accumulates the off-processor **y** values.

2 Overview of CHAOS

Solving such concurrent irregular problems on distributed memory machines using CHAOS runtime support usually involves six major phases (Figure 4). The first four phases concern mapping data and computations onto processors. The next two steps concern analyzing data access patterns in a loop and generating optimized communication calls. A brief description of these phases follows; they will be discussed in detail in later sections.

Phase A :	Data Partitioning	Assign elements of data arrays to processors
Phase B :	Data Remapping	Redistribute data array elements
Phase C :	Iteration Partitioning	Allocate iterations to processors
Phase D :	Iteration Remapping	Redistribute indirection array elements
Phase E :	Inspector	Translate indices; Generate schedules
Phase F :	Executor	Use Schedules for Data Transportation; Perform computation

Figure 4: Solving Irregular Problems

1. **Data Distribution :** Phase A calculates how data arrays are to be partitioned by making use of partitioners provided by CHAOS or by the user. CHAOS supports a number of parallel partitioners that partition data arrays using heuristics based on spatial positions, computational load, connectivity, etc. The partitioners return an irregular assignment of array elements to processors, which is stored as a CHAOS construct called the *translation table*. A translation table is a globally accessible data structure which lists the home processor and offset address of each data array element. The translation table may be replicated, distributed regularly, or stored in a paged fashion, depending on storage requirements.

2. **Data Remapping :** Phase B remaps data arrays from the current distribution to the newly calculated irregular distribution. A CHAOS procedure `remap` is used to generate an optimized *communication schedule* for moving data array elements from their original distribution to the new distribution. Other CHAOS procedures, `gather`, `scatter`, and `scatter_append`, use the communication schedule to perform data movement.

3. **Loop Iteration Partitioning :** Phase C determines how loop iterations should be partitioned across processors. There are a large number of possible schemes for assigning loop iterations to processors based on optimizing load balance and communication volume. CHAOS uses the *almost-owner-computes* rule to assign loop iterations to processors. Each iteration is assigned to the processor which owns a majority of data array elements accessed in that iteration. This heuristic is biased towards reducing communication costs. CHAOS also allows the owner-computes rule.

4. **Remapping Loop Iterations :** Phase D is similar to phase B. Indirection array elements are remapped to conform with the loop iteration partitioning. For example, in Figure 1, once loop L2 is partitioned, indirection array elements **edge1(i)** and **edge2(i)** used in iteration i are moved to the processor which executes that iteration.

5. **Inspector :** Phase E carries out the preprocessing needed for communication optimizations and *index translation*.

6. **Executor :** Phase F uses information from the earlier phases to carry out the computation and communication. Communication is carried out by CHAOS data transportation primitives which use communication schedules constructed in Phase E.

In static irregular problems, Phase F is executed many times, while phases A through E are executed only once. In some adaptive problems data access patterns change periodically but reasonable load balance is maintained. In such applications, phase E must be repeated whenever the data access pattern changes. In even more highly adaptive problems, the data arrays may need to be repartitioned in order to maintain load balance. In such applications, all the phases described above are repeated.

3 Inspector

The CHAOS procedures that can be used to generate the data structures *translation tables* and *schedules* and to operate on these data structures are illustrated in this section. The functionalities of the procedures explained in this section are shown in Table 1.

3.1 Communication Schedules

To describe the inspector primitives provided by CHAOS, we shall often refer to the example code in Figure 1. In that example, one should notice that the arrays (**edge1** and **edge2**) are used to index other arrays (**x** and **y**). Since this is a indirect indexing method, we call edge1 and edge2 *indirection arrays*. As we see from the parallelized node code version (Figure 2) of the simple loop, we need to do some preprocessing before we can actually execute the loop. This preprocessing is aimed at achieving the following goals :

1. Global to Local translation of references : Determine which of the references made by an indirection array (e.g. edge1, edge2) are references to data that is now on-processor. These references are changed so that they now point to the local address. For references to data that is now off-processor, we need to assign on-processor buffer locations where these off-processor elements will be brought into. The off-processor references of the indirection array are changed so that they now point into this buffer location.

2. Communication Schedule Generation : The communication schedule specifies an optimized way to gather off-processor data, and to scatter back local copies after computation. CHAOS primitves are used to scan the data access pattern and determine which off-processor elements will be needed during computation. Duplicates among these references are removed, and the off-processor references are merged so that fewer messages will be needed for moving data. The communication schedule data structure typically contains the following :

   (a) send list — a list of local array elements that must be sent to other processors,

   (b) permutation list — an array that specifies the data placement order of incoming off-processor elements, (in a local buffer area which is designated to receive incoming data),

   (c) send size — an array that specifies the sizes of out-going messages from processor p to other processors.

Table 1: CHAOS – Inspector/Executor Procedures

Task	Functionality	Procedure
Inspector		
Schedule	compute Schedule compute Schedule compute Schedule compute Incremental Schedule hash global references create hash table deallocate hash table	localize() reglocalize() PARTI_schedule() PARTI_incremental_schedule() PARTI_hash() PARTI_creat_hash_table() PARTI_free_hash_table()
Translation Table	build translation table build translation table build paged translation table update a paged table get replication factor get table page size dereference dereference only processor dereference only offset get local indices deallocate translation table	build_translation_table() build_reg_translation_table() build_dst_translation_table() table_remap() getTableRepF() getTablePageSize() dereference() derefproc() derefoffset() getTableIndices() free_table()
Executor		
Data Exchangers	Fetch off-processor data scatter off-processor data scatter off-processor data with function fetch off-processor data scatter off-processor data scatter off-processor data scatter off-processor data gather off-processor data gather off-processor data scatter off-processor data scatter off-processor data	PREFIXgather() PPREFIXscatter() PREFIXscatter_FUNC() PREFIXmulitgather() PREFIXscatternc() PREFIXmulitscatternc() PREFIXmulitscatter_FUNCnc() PARTI_gather() PARTI_mulgather() PARTI_scatter() PARTI_mulscatter()
Miscellaneous		
	Initialize CHAOS environment Renumber data arrays	PARTI_setup() renumber()

PREFIX: data type: d for double precision, f for real, i for integer

FUNC: function : add for addition, mult for multiplication

(d) fetch size – an array that specifies the sizes of in-coming messages to processor p from other processors.

Some schedules do not need all of these entries. These variants are described below.

The following example further clarifies the process of index-translation. Let us assume that an array x, with 5 elements, x(1)...x(5), has been distributed over two processors. Similarly the indirection arrays, edge1 and edge2 are also distributed over the two processors. The local portions of each array x, edge1 and edge2, look like this :

```
     <--local area-->              <--local area-->
      --------------------        ----------------------
     | x(2) | x(5) |             | x(1) | x(3) | x(4) |
      --------------------        ----------------------
        ^       ^                   ^       ^     ^
Local:  x(1)    x(2)                x(1)    x(2)  x(3)


nedge = 5, and                   nedge = 5
edge1 = 2, 1, 4, 5 ,1            edge1 = 4, 1, 3, 4, 2
           -  -     -                                -
edge2 = 4, 1, 3, 5, 2            edge2 = 1, 3, 5, 1, 2
        -  -  -                                -     -
where the underlined references are off-processor.
```

As explained in Section 1.1.2, before we execute the loop we must bring in a copy of all off-processor references that might be made inside the loop. In this case, we have 9 off-processor references (only 5 of these are distinct). We can assign these copies memory just at the end of the local portion of the array. This new region is called the *ghost area* or the off-processor buffer.

```
 <--local area-><-ghost cells ->    <--local area-------><ghost cells>
 -------------------------------    --------------------------------
 | x(2) | x(5) | x(1) x(4) x(3)|    | x(1) | x(3) | x(4)| x(2)  x(5)|
 -------------------------------    -------------------------------
   ^       ^     ^    ^     ^          ^      ^     ^    ^     ^
 x(1)    x(2)  x(3) x(4)  x(5)       x(1)   x(2)  x(3) x(4)  x(5)

Now we must change the global references in edge1, edge2
so that they now point to the appropriate local references.
After index-translation, we have :
nedge = 5                             nedge = 5
edge1 = 1, 3, 4, 2, 3                 edge1 = 3, 1, 2, 3, 4
edge2 = 4, 3, 5, 2, 1                 edge2 = 1, 2, 5, 1, 4
```

The next task is to build a communication schedule. The fetch_size and send_size are the amounts of data that must be transferred during a "gather" operation. The send_list specifies the data that must be sent to every other processor. The permutation_list specifies where incoming data from each processor should be placed in the *ghost area*. Note that a communication schedule is dependent on the global-to-local index translation having been done, since the permutation list can only be determined after each off-processor reference has been assigned a ghost-area location during index translation. In the permutation list, the local indices are stored as offsets from the beginning of the ghostarea, instead of being stored as absolute local index values. For the example above, the communication schedule would look as follows:

```
Schedule_P0 {                      Schedule_P1 {
 fetch_size :  [ 0, 3 ]             fetch_size : [ 2, 0 ]
 send_size  :  [ 0, 2 ]             send_size  : [ 3, 0 ]
 send_list  :  p0 -> NULL           send_list  : p0 -> 1, 2, 3
               p1 -> 1, 2                        p1 -> NULL
 perm_list  :  p0 -> NULL           perm_list  : p0 -> 1, 2
               p1 -> 1, 3, 2                     p1 -> NULL
}                                  }
```

In the following sections, CHAOS primitives for performing index-translation and schedule generation have been described. Since index-translation and schedule-generation are usually performed one after the other, CHAOS primitves can combine the two-steps into a single primitive which, given an indirection array, returns a translated indirection array, as well as a communication schedule. In some cases however, it is possible to reuse the index translation process to generate different schedules. For such applications, CHAOS allows the user to use a two-step inspector process; by breaking the inspector step into index-translation and schedule-generation stages.

3.2 Single-phase Inspector

Single-step inspector primitives perform both index-translation as well as schedule-generation, with respect to a given set of indirection arrays.

3.2.1 subroutine localize()

`localize` is used to translate all the global indices in the given indirection array into local indices. It also returns the schedule corresponding to that indirection array. The schedule pointer returned by `localize` is used to gather data and store it at the end of the local array. This schedule created is such that multiple copies of the same data is not brought in during the gather phase. The elimination of duplicates is achieved by using a hash table. `localize` returns the local reference string corresponding to the global references which are passed as a parameter to it. The number of off-processor data elements are also returned by `localize` so that one can allocate enough space at the end of the local array.

Synopsis

subroutine localize(itabptr,ilsched,iglobal_refs, ilocal_refs,ndata,n_off_proc, my_size,repetition)

Parameter Declarations

integer itabptr refers to the relevant translation table pointer.

integer ilsched refers to the relevant schedule pointer (returned by `localize`).

integer iglobal_refs() the array which stores all of the global indirection array references.

integer ilocal_refs() the array which stores the local reference string corresponding to the global references (returned by `localize`).

integer ndata number of global references.

integer n_off_proc number of off-processor data (returned by `localize`).

integer my_size the size of my local data array.

integer repetition maximum number of columns or rows from which data will be gathered or scattered using the schedule returned by localize.

Return Value

None

Example

Assume that data arrays **x** and **y** are identically but irregularly distributed between processors 0 and 1. and the processors take part in a computation that involves a loop which refers to off-processor data. The indirect data array references are stored in **global_ref** and the array **my_index** has the local indices of the data arrays. The inspector and the executor code for the loop is presented here.

```
       integer i,ndata,indirection, BUFSIZE
       parameter(BUFSIZE = 4)
       integer my_index(5),global_ref(5),local_ref(5)
       double precision x(5),y(5+BUFSIZE)
       integer tabptr,schedptr, build_translation_table
c data initialization
       if(MPI_mynode() .eq. 0) then
         my_index(1) = 2
         my_index(2) = 3
         my_index(3) = 6
         my_size = 3
         global_ref(1) = 4
         global_ref(2) = 8
         global_ref(3) = 2
         ndata = 3
       else if(MPI_mynode() .eq. 1) then
         my_index(1) = 1
         my_index(2) = 4
         my_index(3) = 5
         my_index(4) = 7
```

```
         my_index(5) = 8
         my_size = 5
         global_ref(1) = 5
         global_ref(2) = 3
         global_ref(3) = 4
         global_ref(4) = 1
         global_ref(5) = 7
         ndata = 5
        else
         my_size = 0
         ndata = 0
        end if
        do 20 i=1,my_size
           x(i) = my_index(i)
           y(i) = 2*my_index(i)
20       continue
C initialize Chaos environment
        call PARTI_setup()
c the following is the inspector code
         tabptr = build_translation_table(1,my_index,my_size)
         call localize(tabptr,schedptr,global_ref,
     $        local_ref,ndata,n_off_proc,my_size,1)
c
         do 10 i=1,ndata
           global_ref(i) = local_ref(i)
10       continue
c  end of the inspector and the executor begins
         call dgather(y(my_size+1),y,schedptr)
         do 30 i=1,ndata
           indirection = global_ref(i)
           x(i) = x(i) + y(indirection)
30       continue
c end of the executor code
         if (MPI_mynode() .eq. 0) then
         WRITE(*,*) (X(I), I=1,my_size)
         WRITE(*,*) (global_ref(I), I=1,ndata)
         WRITE(*,*) (Y(global_ref(I)), I=1,ndata)
         end if
         call MPI_gsync()
         if (MPI_mynode() .eq. 1) then
         WRITE(*,*) (X(I), I=1,MY_SIZE)
         WRITE(*,*) (global_ref(I), I=1,ndata)
         WRITE(*,*) (Y(global_ref(I)), I=1,ndata)
         end if

         end
```

The distribution of data arrays are stored in a translation table *tabptr* and the communication pattern between processors are stored in a schedule *schedptr*. The procedure dgather brings in the off-processor data. After the end of the computation in processor 0 the values of x(1), x(2) and x(3) are 10.0, 19.0 and 10.0 respectively. On processor 1 the values of x(1), x(2), x(3), x(4) and x(5) are 10.0, 6.0, 8.0, 2.0 and 14.0 respectively.

3.2.2 subroutine reglocalize()

The functionality of this procedure is similar to that of the `localize` procedure, except in this case instead of passing a pointer to the translation table (for an irregular data distribution specification) we must give a regular distribution. At the present time we support BLOCK and CYCLIC distributions. Other regular distribution can be supplied by the user by writing their own regular dereference.

Synopsis

subroutine reglocalize(distribution,size,ilsched,iglobal_refs,
ilocal_refs,ndata,n_off_proc,my_size,repetition)

Parameter Declarations

integer distribution BLOCK = 1, CYCLIC = 2 or OWN = 3.

integer size The array size from which data is to be gathered or scattered.

integer ilsched refers to the relevant schedule pointer (returned by `reglocalize`).

integer iglobal_refs() the array which stores all of the global indirection array references.

integer ilocal_refs() the array which stores the local reference string corresponding to the global references (returned by `reglocalize`).

integer ndata number of global references.

integer n_off_proc number of off-processor data (returned by `reglocalize`).

integer my_size parameter will return the size of the local array.

integer repetition maximum number of columns or rows from which data will be gathered or scattered using the schedule returned by localize.

Return Value

None

Example

The example is similar to the one presented in the localize section except that the data arrays are regularly distributed i.e., by BLOCK.

```
      integer i,ndata,indirection, BUFSIZE, global_size, BLOCK
      integer my_index(4)
      parameter(BUFSIZE = 4)
      integer iglobal_ref(4),ilocal_ref(4)
      real buffer(3), aloc(4)
      double precision x(5),y(5+BUFSIZE)
      integer itabptr, ischedptr, build_translation_table

      BLOCK = 1
      global_size = 8
      if (MPI_numnodes() .le. global_size)  then
         my_size = global_size/MPI_numnodes()
      else
         if (MPI_mynode() .lt. global_size) then
            my_size = 1
         else
            my_size = 0
         endif
      endif
      if(MPI_mynode() .eq. 0) then
        iglobal_ref(1) = 4
        iglobal_ref(2) = 8
        iglobal_ref(3) = 2
        ndata = 3
      else if (MPI_mynode() .eq. 1) then
        iglobal_ref(1) = 5
        iglobal_ref(2) = 3
        iglobal_ref(3) = 4
        iglobal_ref(4) = 1
        ndata = 4
       else
        ndata  = 0
       end if

C initialize Chaos environment
        call PARTI_setup()
c the following is the inspector code
         call reglocalize(BLOCK,global_size,ischedptr,iglobal_ref,
     $                 ilocal_ref,ndata,n_off_proc,my_size,1)
        do 10 i=1,ndata
           iglobal_ref(i) = ilocal_ref(i)
10       continue


        do i=1, ndata
           x(i) =  MPI_mynode()*100 + i
```

```
        enddo
        do 20 i=1, my_size
           y(i) =  MPI_mynode()*100 + 2*i
20      continue

c  end of the inspector and the following is the executor code
         call dgather(y(my_size+1),y,ischedptr)

         do 30 i=1,ndata
           indirection = iglobal_ref(i)
           x(i) = x(i) +  y(indirection)
30       continue

c end of the executor code
         if (MPI_mynode() .le. 1) then
         WRITE(*,*)
     $     '[',MPI_mynode(),'] x:',(X(I), I=1,ndata)
         WRITE(*,*)
     $      '[',MPI_mynode(),'] gref:',(iglobal_ref(I), I=1,ndata)
         WRITE(*,*)
     $      '[',MPI_mynode(),'] y:',(Y(iglobal_ref(I)), I=1,ndata)
         end if
         call MPI_gsync()


C  Gather example
        do 11 i=1,my_size
            aloc(i) =  float(MPI_mynode())+0.1*i
11       continue

         call fgather(buffer,aloc,ischedptr)
 if (MPI_mynode() .le. 1) then
          write (*,*) 'Gather results'
          WRITE(*,*) MPI_mynode(), (buffer(I), I=1,n_off_proc)
         endif

         end
```

After the end of the computation in processor 0 the values of x(1), x(2), x(3), and x(4) are 9.0, 18.0, 7.0 and 4.0 respectively. On processor 1 the values of x(1), x(2), x(3), and x(4) are 15.0, 12.0, 15.0, and 10.0 respectively.

3.3 Two-phase Inspector

Instead of using `localize` or `reglocalize`, the user may also choose to perform index-translation and schedule-generation in separate steps. For this purpose, CHAOS provides the four primitives — `PARTI_create_hash_table()`, `PARTI_hash()`, `PARTI_schedule()` and `PARTI_free_hash_table()`. `PARTI_hash()` is used to enter a data access pattern (i.e an indirection array) into a hash table, where duplicates are removed and global-to-local index translation is performed. `PARTI_schedule()` is used to build a communication schedule by inspecting the entries in the hash table. `PARTI_create_hash_table` and `PARTI_free_hash_table()` are used to allocate and deallocate memory for the hash table.

3.3.1 subroutine PARTI_schedule()

Synopsis

PARTI_schedule(hash_table, combo_mask, sched, maxdim)

Parameters

integer hash_table the hash_table

integer combo_mask the combination of indirection arrays for which a schedule is needed.

integer sched the schedule that is returned.

integer maxdim maximum number of columns or rows from which data will be gathered or scattered using the schedule returned by PARTI_schedule.

Return Value

None

3.3.2 subroutine PARTI_hash()

Enters the global references into the heap, and converts them into referencres to a local array.

Synopsis

PARTI_hash(trans_table, hash_table, supermask, new_stamp, global_refs, ndata, my_size, no_off_proc, no_new_off_proc)

Parameters

integer trans_table translation table used to determine local address of any global address

integer hash_table the heap into which all the global references will be hashed. It must have been created earlier using PARTI_create_hash_table()

integer global_refs() the global references

integer ndata size of global_refs array

integer my_size the size of the local buffer - the off_processor global references will be modified so that they now point to address beginning from mysize onwards..

integer no_off_proc Conatins the number of off_processor references in global_refs (returned by `PARTI_hash`).

integer no_new_off_proc Some of the off_processor references may have already been hashed into the heap by previous indirection arrays. This parameter will have the number of new or unique off_processor references (returned by `PARTI_hash`).

integer new_stamp A heap (hash_table) can be used to hash in many indirection array - all entries hashed in by a particular indirection array have a specific id. The parameter new_stamp returns the id assigned to this particular indirection array's elements (returned by `PARTI_hash`).

integer supermask Each indirection array can specify which of the previous indirection arrays' entries it wants to reuse - this is done by passing in a parameter called the supermask. A supermask is created by converting the id's of the previous indirection arrays into a "mask" using PARTI_make_mask (id) and then using "bitwise-or" to OR these masks If a user does not care for reusing previous index analyses, a -1 should be passed.

Return value

None

3.3.3 function PARTI_create_hash_table()

Creates a heap into which each processor will enter its global references. (any data_range will do - but a good estimate improves performance)

Synopsis

PARTI_create_hash_table(data_range)

Parameters

integer data_range the maximum size of a global reference

Return Value

an integer identifying the hash table

3.3.4 subroutine PARTI_free_hash_table()

Deallocates the hash_table created with PARTI_create_hash_table()

Synopsis

PARTI_free_hash_table(hash_table)

Parameters

integer hash_table an integer identifying the hash table

Return Value

None

3.3.5 function PARTI_make_mask()

Synopsis

PARTI_make_mask(stamp)

Parameters

integer stamp

Return Value

an integer representing the bit-mask.

Converts an id to a bit-mask (see PARTI_hash)

```
      integer i,ndata,indirection, BUFSIZE, my_size
      parameter(BUFSIZE = 4)
      integer my_index(5),global_ref(5),local_ref(5)
      double precision x(5),y(5+BUFSIZE)
      integer tabptr,schedptr, build_translation_table
      integer parti_create_hash_table
      integer parti_make_mask
      integer hashptr
      integer istamp, imask, ndont_care, n_off_proc
c data initialization
      if(MPI_mynode().eq.0) then
        my_index(1) = 2
        my_index(2) = 3
```

```
         my_index(3) = 6
         my_size = 3
         global_ref(1) = 4
         global_ref(2) = 8
         global_ref(3) = 2
         global_ref(4) = 1
         ndata = 4
       else if(MPI_mynode().eq. 1) then
         my_index(1) = 1
         my_index(2) = 4
         my_index(3) = 5
         my_index(4) = 7
         my_index(5) = 8
         my_size = 5
         global_ref(1) = 5
         global_ref(2) = 3
         global_ref(3) = 4
         global_ref(4) = 1
         global_ref(5) = 7
         ndata = 5
        else
         my_size = 0
         ndata = 0
        end if
        do 20 i=1,my_size
           x(i) = my_index(i)
           y(i) = 2*my_index(i)
20       continue
C initialize CHAOS environment
        call PARTI_setup()
c the following is the inspector code
         tabptr = build_translation_table(1,my_index,my_size)
         hashptr =  PARTI_create_hash_table(8)
         call PARTI_hash(tabptr,hashptr,-1,istamp,global_ref,
     $      ndata, my_size, ndont_care, n_off_proc)
c
c The stamp returned by PARTI_hash is an integer between 1 and 32.
c PARTI_make_mask maps this integer into a  specific bit in an integer mask
c masks can then be bit-OR'ed if needed.
c
         imask =  PARTI_make_mask(istamp)
         call PARTI_schedule(hashptr, imask, schedptr, 1)
c
c  end of the inspector and the executor begins
         call dgather(y(my_size+1),y,schedptr)
         do 30 i=1,ndata
```

```
            indirection = global_ref(i)
            x(i) = x(i) +  y(indirection)
30        continue

          if (MPI_mynode().eq.0) then
          WRITE(*,*) 'Processor 0'
          WRITE(*,*) (X(I), I=1,my_size)
          end if
          call MPI_gsync()
          if (MPI_mynode().eq.1) then
          WRITE(*,*) 'Processor 1'
          WRITE(*,*) (X(I), I=1,MY_SIZE)
          end if

end
```

3.4 Inspectors for Partially Modified Data Access Patterns

The previous section described how a two-step inspector can be used instead of a single primitive such as `localize`. Both schemes provide the same functionality; indeed `localize` and `reglocalize` have been built on top of the more general primitives `PARTI_hash` and `PARTI_schedule`.

We recommend that users use the one-step inspector routines whenever possible. Two-step inspectors are usually needed when the indirection arrays are modified slightly during the course of computation. In such cases, the index analysis for the old indirection array can be reused since information is maintained in the hash-table.

To clarify how this is done, we provide the following example.

```
       integer i,ndata,indirection, BUFSIZE
       parameter(BUFSIZE = 4)
       integer my_index(5),global_ref(5),local_ref(5)
       double precision x(4),y(4+BUFSIZE)
       integer tabptr,schedptr, build_translation_table
       integer hashptr
c data initialization
       if(mynode() .eq. 0) then
         my_index(1) = 2
         my_index(2) = 3
         my_index(3) = 6
         my_size = 3
         global_ref(1) = 4
         global_ref(2) = 8
         global_ref(3) = 2
         global_ref(4) = 1
         ndata = 4
       else
```

```
         my_index(1) = 1
         my_index(2) = 4
         my_index(3) = 5
         my_index(4) = 7
         my_index(5) = 8
         my_size = 5
         global_ref(1) = 5
         global_ref(2) = 3
         global_ref(3) = 4
         global_ref(4) = 1
         global_ref(5) = 7
         ndata = 5
        end if
        do 20 i=1,my_size
           x(i) = my_index(i)
           y(i) = 2*my_index(i)
20       continue
C initialize CHAOS environment
        call PARTI_setup()
c the following is the inspector code
         tabptr = build_translation_table(1,my_index,my_size)
         call PARTI_create_hash_table(8)

       do k = 1, no_time_steps
c SSS
         if ( k .eq. 1)
           ioldmask = -1
         else
           ioldmask = PARTI_make_mask(istamp)
         endif

         call PARTI_hash(tabptr,hashptr,ioldmask,istamp,global_ref,
     $       ndata, my_size, ndont_care, n_off_proc)

         imask =  PARTI_make_mask(istamp)
         call PARTI_schedule(hashptr, imask, schedptr, 1)
c EEE

c  end of the inspector and the executor begins
         call dgather(y(my_size+1),y,schedptr)
         do 30 i=1,ndata
           indirection = global_ref(i)
           x(i) = x(i) +  y(indirection)
30       continue

c        On some weird conditions change a few entries
```

```
c        in the indirection array.

         do  i =1, ndata
           if ( i .eq. random(k) ) then
                global_ref(i) = global_ref(i) + 2
           endif
         enddo


      enddo
c end of the executor code
```

In the previous example, the indirection array changes every time step; hence the schedule must be regenerated each time. Since most of the index analysis can be resued, we call `PARTI_hash` with a pointer to the old hash table and pass the mask of the previous indirection array. Using this information, the `PARTI_hash` hashes in each entry of the new indirection array and resues the index analysis information of matching pre-existing entries in the hash-table. Index analysis is performed only for those entries that are not found in the hash table. There is a caveat to the previous example : the number of distinct stamps that `PARTI_hash` can issue is limited to 32. This implies that the previous example will work only when the number of time-steps is less than 32.

CHAOS provides two ways of working around this problem. `PARTI_clear_stamp` can clear all entries with a particular stamp. `PARTI_clear_mask` is a more general primitive with which any entry in the hash table with a particular combination of stamps can be removed. The second method is to instruct `PARTI_hash` not to issue a new stamp every iteration. For instance, in the previous example, each indirection array is used only for one schedule; indirection arrays for old time-stamps are never resued. By passing in an istamp value of -1 to `PARTI_hash` the user can direct it to reuse the last issued stamp value for all new entries.

The following code segment shows how each of this can be done. These code segments reflect the section between SSS and EEE in the previous example.

```
c  Using PARTI_clear_mask to clear entries in hash table
c SSS
         if ( k .eq. 1)
           ioldmask = -1
         else
           ioldmask = PARTI_make_mask(istamp)
         endif

         call PARTI_hash(tabptr,hashptr,ioldmask,istamp,global_ref,
     $        global_ref, ndata, mysize, ndont_care, n_off_proc)

         if ( k .gt. 1) call PARTI_clear_mask(ioldmask)

         imask =  PARTI_make_mask(istamp)
         call PARTI_schedule(hashptr, imask, schedptr, 1)
c EEE
```

```
c  Using stamp reuse to limit   stamp numbers
c SSS
        if ( k .eq. 1)
          ioldmask = -1
        else
          ioldmask = PARTI_make_mask(istamp)
        endif

        istamp = -1
        call PARTI_hash(tabptr,hashptr,ioldmask,istamp,global_ref,
    $       ndata, my_size, ndont_care, n_off_proc)

        imask =  PARTI_make_mask(istamp)
        call PARTI_schedule(hashptr, imask, schedptr, 1)
c EEE
```

3.4.1 subroutine PARTI_clear_mask()

Synopsis

PARTI_clear_mask (hash_table, mask)

Parameters

integer hash_table

integer mask

Return Value

None

Removes all entries in a hash_table which belong uniquely to this mask

3.4.2 subroutine PARTI_clear_stamp()

A wrapper around PARTI_clear_mask(). You can directly give the id of the indirection array that you want removed from the heap.

Synopsis

PARTI_clear_stamp (hash_table, stamp)

Parameters

integer hash_table

integer stamp

Return Value

None

3.5 Incremental schedules

Incremental scheduling is a method by which users can take advantage of the similarities between various data access patterns. Many of the off-processor references specified in an indirection array may be the same as the references in a previously analysed indirection array. In such cases some of the off-processor references of the second indirection array can be treated as local during schedule generation. This effectively reduces the communication volume of gathered elements.

To build incremental schedules, `PARTI_hash` must be passed masks corresponding to stamps of previous indirection arrays. This notifies the underlying layer which existing entries in the hash table can be resued. After hashing, `PARTI_incremental_schedule` is used to build the schedule.

3.5.1 subroutine PARTI_incremental_schedule()

Similar to PARTI_schedule() except that it builds a schedule only for entries in the hash_table which uniquely belong to this particular mask.

Synopsis

PARTI_incremental_schedule(hash_table, combo_mask, sched, maxdim)

Parameters

integer hash_table the hash_table

integer combo_mask the combination of indirection arrays for which a schedule is needed.

sched refers to the relevant schedule pointer (returned by `PARTI_incremental_schedule`).

maxdim maximum number of columns or rows from which data will be gathered or scattered using the returned schedule.

Return Value

None

4 Data Exchangers

The CHAOS data structure *schedule* stores the send/recieve patterns, but the CHAOS data exchangers actually move data between processors using *schedules*.

4.1 subroutine PREFIXgather()

PREFIX can be d (double precision), i (integer) , f (real) or c (character). The PREFIXgather procedure uses a schedule and copies of data values obtained from other processors are placed in memory pointed to by $buffer$. Also passed to PREFIXgather is a pointer to the location from which data is to be fetched *on the calling processor*. This pointer is designated here as $aloc$.

Synopsis

PREFIXgather(buffer,aloc,schedinfo)

Parameter Declarations

TYPE buffer() pointer to buffer for copies of gathered data values

TYPE aloc() location from which data is to be fetched from calling processor

integer schedinfo refers to the relevant schedule

Return Value

None

Assume that a schedule has already been obtained and that real numbers are gathered, i.e., fgather is used. On each processor, aloc points to the arrays from which values are to be obtained. Buffer points to the location into which will be placed copies of data values obtained from other processors.

```
  real buffer(3), aloc(4)
  integer ischedptr

     do 10 i=1,4
       aloc(i) = float(mynode()) + 0.1*i
10   continue

     call fgather(buffer,aloc,ischedptr)
     WRITE(*,*) (buffer(i), i=1,n_off_proc)
```

On processor 0, buffer(1) is 1.4 and on processor 1, buffer(1), buffer(2) and buffer(3) are now equal to 0.3, 0.4 and 0.1. The size of off-processor elements n_off_proc is returned by the procedure **reglocalize**.

4.2 subroutine PREFIXscatter()

PREFIX can be d (double precision), i (integer) , f (real) or c (character). PREFIXscatter uses a schedule produced by a call to another subroutine which creates a schedule (eg: reglocalize). Copies of data values to be scattered to other processors are placed in memory pointed to by buffer. Also passed to PREFIXscatter is a pointer to the location to which copies of data are to be written *on the calling processor*. This pointer is designated here as aloc.

Synopsis

PREFIXscatter(buffer,aloc,schedinfo)

Parameter Declarations

TYPE buffer() points to data values to be scattered from a given processor

TYPE aloc() points to first memory location on calling processor for scattered data to be placed

integer schedinfo refers to the relevant schedule.

Return Value

None

Example

We assume that a schedule has already been obtained by calling a subroutine such as reglocalize. Our example will assume that we wish to scatter real precision numbers, i.e., that we will be calling fscatter. On each processor, aloc points to the arrays to which values are to be scattered. "buffer" points to the location from which data will be obtained to scatter.

```
  real buffer(3), aloc(4)
  integer ischedptr
     do 11 i=1,4
       aloc(i) = 10.0
11   continue
   if(mynode().eq.0) then
     buffer(1) = 555.55
   endif

   if(mynode().eq.1) then
     buffer(1) = 666.66
     buffer(2) = 777.77
     buffer(3) = 888.88
   endif

   call fscatter(buffer,aloc,ischedptr)
```

On processor 0, the first four elements of aloc are 888.88, 10.0, 666.66, and 777.77. On processor 1, the first four elements of aloc are 10.0, 10.0, 10.0 and 555.55.

4.3 subroutine PREFIXscatter_FUNC()

PREFIX can be d (double precision), i (integer) , f (real) or c (character). FUNC can be add, sub or mult . PREFIXscatter stores data values to specified locations. PREFIXscatter_FUNC allows one processor to specify computations that are to be performed on the contents of given memory location of another processor. The procedure is in other respects analogous to PREFIXscatter.

Synopsis

PREFIXscatter_FUNC(buffer,aloc,ischedinfo)

Parameter Declarations

TYPE buffer() points to data values that will form operands for the specified type of remote operation.

TYPE aloc() points to first memory location on calling processor to be used as targets of remote operations.

integer schedinfo refers to the relevant schedule.

Return Value

None

Example

We assume that a schedule has already been obtained by calling a subroutine such as reglocalize. Our example will assume that we wish to scatter and add real numbers, i.e. that we will be calling fscatter_add. On each processor, aloc points to the arrays to which values are to be scattered and added. 'buffer' points to the location from which values will be obtained to scatter and add.

```
  real buffer(3), aloc(4)
  integer ischedptr
     do 13 i=1,4
       aloc(i) = 10.0
13   continue
   if(mynode().eq.0) then
     buffer(1) = 555.55
```

```
endif

if(mynode().eq.1) then
  buffer(1) = 666.66
  buffer(2) = 777.77
  buffer(3) = 888.88
endif

call fscatter_add(buffer,aloc,ischedptr)
```

On processor 0, the first four elements of aloc are 898.88, 10.0, 676.66 and 787.77. On processor 1, the first three elements of aloc are 10.00, 10.00, 10.00 and 565.55.

4.4 subroutine PREFIXmultigather()

This primitive is an extension of the regular gather primitive, such that it allows one to specify multiple schedules to be used to gather data from the target array into the buffer. It also allows multi-column or multi-row gathers if the columns are distributed in the same way. The different schedules to be used are passed in an array data structure to the function, and the number of schedules need to be specified as a parameter to the function.

Synopsis

PREFIXmultigather(buffer,aloc,n_scheds, scheds,base_shift,dilation,repetition)

Parameter Declarations

TYPE buffer() buffer for copies of gathered data values

TYPE aloc() location from which data is to be fetched from calling processor

integer n_scheds number of schedules passed

integer scheds() array of pointers where each pointer points to a schedule

integer base_shift size of the distributed dimension.

integer dilation the factor by which the distance between two data accesses on any dimension changes when that dimension is transposed.

integer repetition maximum number of columns or rows from which data will be accessed.

Return Value

None

4.5 subroutine PREFIXmultiscatter()

This primitive is an extension of the regular scatter primitive, but it allows one to specify multiple schedules to be used to scatter data from the target array into the buffer. It also allows multi-column or multi-row scatters if the columns are distributed in the same way. The different schedules to be used are passed in an array data structure to the function, and the number of schedules need to be specified as a parameter to the function.

Synopsis

PREFIXmultiscatter(buffer,aloc,n_scheds,
scheds,base_shift,dilation,repetition)

Parameter Declarations

TYPE buffer() buffer for copies of scattered data values

TYPE aloc() location from which data is to be scattered from calling processor

integer n_scheds number of schedules passed

integer scheds() array of pointers where each pointer points to a schedule.

integer base_shift size of the distributed dimension.

integer dilation the factor by which the distance between two data accesses on any dimension changes when that dimension is transposed.

integer repetition maximum number of columns or rows to which data will be scattered.

Return Value

None

Example

Data can be scattered to multi-columns or rows using more than one schedules.

4.6 subroutine PREFIXmultiscatter_FUNC()

FUNC can be add, sub or mult . This primitive is an extension of the regular scatter_FUNC primitive, but it allows one to specify a number of schedules to be used to perform computations on the contents of a given memory location of another processor. It also allows multi-column or multi-row operations if the columns or rows are distributed in the same way. The different schedules to be used are passed in an array data structure to the function, and the number of schedules need to be specified as a parameter to the function.

Synopsis

PREFIXmultiscatter_FUNC(buffer,aloc,n_scheds,
scheds,base_shift,dilation,repetition)

Parameter Declarations

TYPE buffer() points to data values that will form operands for the specified type of remote operation.

TYPE aloc() points to first memory location on calling processor to be used as targets of remote operations.

integer n_scheds number of schedules passed

integer scheds() array of pointers where each pointer points to a schedule.

integer base_shift size of the distributed dimension.

integer dilation the factor by which the distance between two data accesses on any dimension changes when that dimension is transposed.

integer repetition maximum number of columns or rows from which data will be accessed.

Return Value

None

Example

Simple arithmetic operations can be performed on the memory location of another processor similar to the way a gather is performed.

4.7 subroutine PREFIXscatternc()

This works just like a normal scatter function except it does an on-processor gather before it does the scatter. The on-processor gather it does is done according to pattern stored in the schedule.

Synopsis

PREFIXscatternc(buffer,aloc,schedinfo)

Parameter Declarations

TYPE buffer() array from which data is to be scattered.

TYPE aloc() array to which data is to be scattered. processor

integer schedinfo refers to the relevant schedule pointer

Return Value

None

Example

This works similar to the scatter_addnc function.

4.8 subroutine PREFIXmultiscatternc()

This primitive is an extension of the regular multiscatter primitive, such that an on processor gather is performed before the scatter occurs. The on-processor gather is done according to a pattern stored in the schedule.

Synopsis

PREFIXmultiscatternc(buffer,aloc,n_scheds,
scheds,base_shift,dilation,repetition)

Parameter Declarations

TYPE buffer() buffer for copies of scattered data values

TYPE aloc() location from which data is to be scattered from calling processor

integer n_scheds number of schedules passed

integer scheds() array of pointers where each pointer points to a schedule.

integer base_shift size of the distributed dimension.

integer dilation the factor by which the distance between two data accesses on any dimension changes when that dimension is transposed.

integer repetition maximum number of columns or rows to which data will be scattered.

Return Value

None

Example

Data can be scattered to multi-columns or rows using more than one schedules.

4.9 subroutine PREFIXscatter_FUNCnc()

This works just like a normal scatter_FUNC function except it does a on-processor gather before it does the scatter_FUNC. The on-processor gather it does is done according to pattern stored in the schedule.

Synopsis

void PREFIXscatter_FUNCnc(buffer,aloc,schedinfo)

Parameter Declarations

TYPE buffer() array from which data is to be scattered.

TYPE aloc() array to which data is to be scattered. processor

integer schedinfo refers to the relevant schedule pointer

Return Value

None

4.10 subroutine PREFIXmultiscatter_FUNCnc()

FUNC can be add, sub or mult . This primitive is an extension of the regular multiscatter_FUNC primitive, such that it allows the user to perform an on-processor gather before the scatter_FUNC operation.

Synopsis

subroutine PREFIXmultiscatter_FUNCnc(buffer,aloc,n_scheds, scheds,base_shift,dilation,repetition)

Parameter Declarations

TYPE buffer() points to data values that will form operands for the specified type of remote operation.

TYPE aloc() points to first memory location on calling processor to be used as targets of remote operations.

integer n_scheds number of schedules passed

integer scheds() array of pointers where each pointer points to a schedule.

integer base_shift size of the distributed dimension.

integer dilation the factor by which the distance between two data access on any dimension changes when that dimension is transposed.

integer repetition maximum number of columns or rows from which data will be accessed.

Return Value

None

Example

Simple arithmetic operations can be performed on the memory location of another processor similar to the way a gather is performed.

4.11 Data Exchangers For General Data Structures

4.11.1 subroutine PARTI_gather()

Synopsis

subroutine PARTI_gather(sched, data, target, size)

Parameter Declarations

integer sched refers to the relevant schedule.

¡any type¿ data() points to data values that will form operands for the specified type of remote operation.

¡any type¿ target() points to first memory location on calling processor to be used as targets of remote operations.

integer size size of the data structure.

Return Value

None

Similar to TYPEgather() except that this subroutine can be used to gather any type of data structure. The size of the data structure must be provided.

4.11.2 subroutine PARTI_scatter()

Synopsis

subroutine PARTI_scatter(sched, data, target, size, func)

Parameter Declarations

integer sched refers to the relevant schedule.

data() points to data values that will form operands for the specified type of remote operation.

target() points to first memory location on calling processor to be used as targets of remote operations.

integer size size of the data structure.

func function to be used.

Return Value

None

Similar to scatter_FUNC subroutine, but the function is provided by the user. The parameter 'func' specifies the operation for scatter routine. For standard functions the following constants can be used:

```
NULL       : no operation (scatter function)
PARTI_add  : integer addition
PARTI_sub  : integer substraction
PARTI_mul  : integer multiplication
PARTI_cadd : character addition
PARTI_csub : character substraction
PARTI_cmul : character multiplication
PARTI_fadd : float addition
PARTI_fsub : float substraction
PARTI_fmul : float multiplication
PARTI_dadd : double addition
PARTI_dsub : double substraction
PARTI_dmul : double multiplication
```

4.11.3 subroutine PARTI_mulgather()

Synopsis

PARTI_mulgather(sched, size, narray, data, target [, data, target, ...])

Parameter Declarations

integer sched refers to the relevant schedule.

integer size size of the data structure.

integer narray number of arrays.

data() points to data values that will form operands for the specified type of remote operation.

target() points to first memory location on calling processor to be used as targets of remote operations.

Return Value

None

The gather routine for multiple arrays.

4.11.4 subroutine PARTI_mulscatter()

Synopsis

PARTI_mulscatter(sched, func, size, narray, data, target [, data, target, ...])

Table 2: CHAOS Procedures for Adaptive Problems

Task	Functionality	Procedure
Inspector		
Light-Weight Schedule	compute schedule	schedule_proc()
Executor		
Data	exchange data	PREFIXscatter_append()
	restore data	PREFIXscatter_back()
Transportation	exchange data	PREFIXmultiscatter_append()
	restore data	PREFIXmultiscatter_back()
	exchange data	PREFIXmultiarr_scatter_append()
	restore data	PREFIXmultiarr_scatter_back()

Parameter Declarations

integer sched refers to the relevant schedule.

func function to be used.

integer size size of the data structure.

integer narray number of arrays.

data() points to data values that will form operands for the specified type of remote operation.

target() points to first memory location on calling processor to be used as targets of remote operations.

Return Value

None

The scatter routine for multiple arrays.

5 CHAOS Procedures for Adaptive Problems

5.1 Introduction

In a class of highly adaptive problems, patterns of data access vary frequently. As a result, pre-processing, or *inspector* must be carried out whenever change in data access pattern occurs. The implication is that the processing cost of *inspector* can hardly be amortized because the communication schedule produced by the *inspector* procedures for one time step may not be reused for the consequent time steps. Inspector procedures for application codes in which partial change in data access patterns occur, are discussed in Section 3.4.

A set of primitives for highly adaptive problems have been developed. These procedures are particularly suitable for applications where order of data storage and computation is not strictly maintained. The procedure developed for adaptive problems compute *light-weight schedules* at very low cost. These procedures compute space requirements for data migration on each processor and determine how collective communications can be performed. Data transportation routines developed for this routines uses these schedules and perform irregular communications efficiently.

5.2 function schedule_proc()

This function returns a communication schedule as **PARTI_schedule** does, but the schedule generated by schedule_proc function does not have information on addresses in destination processors for off-processor data items.

Synopsis

integer schedule_proc(proc, ndata, new_ndata, maxdim)

Parameter declarations

integer proc() a list of destination processor indices

integer ndata the number of local array elements

integer new_ndata number of new local array elements

integer maxdim maximum number of columns or rows from which the distributed arrays will be exchanged

Return value

an integer value representing a light-weight communication schedule

5.3 subsection PREFIXscatter_append()

PREFIX can be **d** (double precision), **i** (integer), **f** (float), **c** (character). The PREFIXscatter_append procedure uses a schedule produced by a call to **schedule_proc()**. The ownership of distributed array *data* is exchanged among participating processors and a copy of new local portion of the distributed array is placed in memory pointed to by *target*.

Synopsis

subroutine PREFIXscatter_append(sched, data, target)

Parameter declarations

integer sched a light-weight schedule data exchange

TYPE data() a distributed array pointer

TYPE target() a pointer to buffer where the new copy of local portion of the array will be placed

Return value

None

5.4 subsection PREFIXscatter_back()

PREFIX can be **d** (double precision), **i** (integer), **f** (float), **c** (character). The PREFIXscatter_back procedure uses the same schedule which was previously used by PREFIXscatter_append() function to reverse the effects of data exchange done by PREFIXscatter_append(). Each element of the distributed array will be restored into its original position within the processor which owned it before data exchange.

Synopsis

subroutine PREFIXscatter_back(sched, data, target)

Parameter declarations

integer sched a light-weight schedule for data exchange

TYPE data() a distributed array pointer

TYPE target() a pointer to buffer from which the copy of local portion of the array is to be restored

Return value

None

5.5 subsection PREFIXmultiscatter_append()

This subroutine is an extension to the **PREFIXscatter_append()** so that it allows one to specify a number of schedules to be used to exchange distributed arrays. It also allows data exchange of multidimensional arrays. The different schedules to be used are passed in an array of schedules to **PREFIXmultiscatter_append()**, and the number of schedules needs to be specified as a parameter to this subroutine.

Synopsis

subroutine PREFIXmultiscatter_append(nsched,scheds,data,target, base_shift,dilation,repetition)

Parameter declarations

integer nsched the number of schedules passed

integer scheds an array of schedules

TYPE data() a distributed array

TYPE target() a pointer to buffer where the new copy of local portion of the array will be placed

integer base_shift size of the distributed dimension

integer dilation *Not used*

integer repetition *Not used*

Return value

None

5.6 subroutine PREFIXmultiscatter_back()

The functionality of this subroutine is the same to that of **PREFIXscatter_back()** except that it allows one to use a number of schedules .

Synopsis

subroutine PREFIXmultiscatter_back(nsched,scheds,data,target, base_shift,dilation,repetition)

Parameter declarations

integer nsched the number of schedules passed

integer scheds an array of schedules

TYPE data() a distributed array whose ownership will be restored

TYPE target() a pointer to buffer from which the copy of the array is to be restored

integer base_shift size of the distributed dimension

integer dilation *Not used*

integer repetition *Not used*

Return value

None

5.7 subroutine PREFIXmultiarr_scatter_append()

This subroutine is another extension to the **PREFIXscatter_append()**, which can be used to exchange arrays which are distributed in a similar manner.

Synopsis

subroutine PREFIXmultiarr_scatter_append(sched,narrays, $data_1, target_1, \ldots, data_n, target_n$)

Parameter declarations

integer sched a schedule for

integer narrays the number of pairs of $data_i$ and $target_i$

TYPE $data_i$**()** array to be distributed

TYPE $target_i$**()** a pointer to buffer where the new copy of local portion of the array will be placed

Return value

None

5.8 subroutine PREFIXmultiarr_scatter_back()

This subroutine is another extension to the **PREFIXscatter_append()**, which can be used to exchange data of multiple arrays which are distributed indentically.

Synopsis

subroutine PREFIXmultiarr_scatter_back(sched,narrays, $data_1, target_1, \ldots, data_n, target_n$)

Parameter declarations

integer sched a schedule for

integer narrays the number of pairs of $data_i$ and $target_i$

TYPE $data_i$**()** a distributed array whose data will be restored

TYPE $target_i$**()** a pointer to buffer from which the copy of the array is to be restored

Return value

None

5.9 Example

```
      integer ndata
      double precision x(ndata), y(ndata), load(i)

      do k = 1, timestep            ! Loop L1
c computation
         do i = 1, ndata            ! Loop L2
            do j = 1, load(i)
```

```
        x(i) = y(i) + ...
      enddo
      load(i) = x(i) ...
   enddo

end do
```

Consider the above sequential code L2. Loop L2 is executed many times inside the loop L1. Computation load for iteration i depends on the value of **load(i)** and it is updated in every iteration k of the loop L1. Note that there is no communication involved. To efficiently run this code on a parallel machine, load in processors must be balanced for every iteration of loop L1.

Let us see how to parallelize this code. Assume that data arrays **data**, **x** and **y** are identically (and maybe irregularly) distributed among several processors, and they need to be remapped to balance the workload. The data array **workload** stores the load information per each data point, and user specified load balancing primitive *load_balancer()* generates a list of destination processor indices using the load information passed. A light-weight schedule **sched** is computed using the procedure *schedule_proc*. Sice arrays **x**, **y** and **load** are distributed identically, the same schedule can be used to tranport all the arrays to new locations.

```
        integer sched, ndata, new_nd, BUFSIZE
        parameter (BUFSIZE = 1000)
        integer proc(BUFSIZE)
        real workload(BUFSIZE), data(BUFSIZE)
        double precision x(BUFSIZE), y(BUFSIZE), load(BUFSIZE)

     do k = 1, timestep                 ! Loop L1

c compute the workload of each data point
        do i = 1, ndata
            workload(i) = ...
        enddo

c initialize CHAOS environment
        call PARTI_setup()

c produce a list of destination processor indices
        call load_balancer(proc,workload,ndata)

c build up a schedule for data exchange
        sched = schedule_proc(proc,ndata,new_nd,1)

c check the buffer size for the new local portion of arrays
        if (new_nd .gt. BUFSIZE) then
            write(*,*) 'Exceed the local buffer size.'
            call exit
```

```
        endif

c exchange distributed data arrays among participating processors
        call fscatter_append(sched,data,data)
        call dmultiarr_scatter_append(sched,3,x,x,y,y,load,load )

c local computation
        do i = 1, new_nd                 ! Loop L2
           ...
        enddo

c restore data distributed arrays
        call fscatter_back(sched,data,data)
        call dmultiarr_scatter_back(sched,3,x,x,y,y,load,load)

    end do
```

6 Translation Table

6.1 function build_translation_table()

In order to allow a user to assign globally numbered indices to processors in an irregular pattern, it is useful to be able to define and access a distributed translation table. By using a distributed translation table, it is possible to avoid replicating records of where distributed array elements are stored in all processors. The distributed table is itself partitioned in a very regular manner. A processor that seeks to access an element I of a irregularly distributed data array is able to compute a simple function that designates a location in the distributed table; the location of the actual array element sought is obtained from the distributed table.

The procedure build_translation_table constructs a distributed translation table. It assumes that distributed array elements are globally numbered. Each processor passes build_translation_table a set of indices for which it will be responsible. The distributed translation table may be striped or blocked across the processors. With a striped translation table, the translation table entry for global index I is stored in processor (I modulo number_of_processors); the local index of the translation table is (I/ number_of_processors). In a blocked translation table, translation table entries are partitioned into a number of equal sized ranges of contiguous integers, these ranges are placed in consecutively numbered processors. With blocked partitioning, the block corresponding to index I is (I/B) and the local index is (I modulo B), where B is the size of the block. Let M be the maximum global index passed to build_translation_table by any processor and NP represent the number of processors; B = $\lceil M/NP \rceil$.

Synopsis

function build_translation_table(part,indexarray,ndata)

Parameter Declarations

integer part how translation table will be mapped - may be BLOCKED(=1) or STRIPED (=2)

integer indexarray() each processor P specifies list of globally numbered indices for which P will be responsible

integer ndata number of indices for which processor P will be responsible

Return Value

integer which refers to the translation table corresponding to the input data.

Example

For a detailed example refer to Section 6.2.

6.2 subroutine dereference()

The subroutine dereference accesses a translation table and determines owner processors and local addresses on the owner processors for a list of global numbered array elements. The subroutine dereference is passed a pointer to a translation table; this structure defines the irregularly distributed mapping created. dereference is passed an array with global indices that need to be located in distributed memory; dereference returns arrays local and proc that contain the processors and local indices corresponding to the global indices.

Synopsis

subroutine dereference(global,local,proc,ndata,index_table)

Parameter declarations

integer global() list of global indices we wish to locate in distributed memory

integer local() local indices obtained from the distributed translation table that correspond to the global indices passed to dereference

integer proc() array of distributed translation table processor assignments for each global index passed to dereference

integer ndata number of elements to be dereferenced

integer index_table refers to the relevant translation table

Return value

None

Example

Table 3: Values obtained by dereference

Processor	proc(1)	local(1)	proc(2)	local(2)
0	0	1	1	1
1	1	2	0	2

A one dimensional distributed array is partitioned in some irregular manner so we need a distributed translation table to keep track of where one can find the value of a given element of the distributed array.

In the example below, we show how a translation table is initialized. Processor 0 calls build_translation_table and assigns indices 1 and 4 to processor 0, processor 1 calls build_translation_table and assigns indices 2 and 3 to processor 1. The translation table is partitioned between processors in blocks.

Processor 0 then uses the translation table to dereference global variables 1 and 2, processor 1 uses the translation table to dereference global variables 3 and 4. On each processor, dereference carries out a translation table lookup. The values of proc and local are returned by dereference are shown in Table 3). The user gets to specify the processor to which each global index is assigned, note however that build_translation_table assigns local indices.

```
       program dref
c
       integer size, i, index_array(2)
       integer deref_array(2)
       integer local(2), proc(2)
       logical build_translation_table

c initialize CHAOS environment
        call PARTI_setup()

c   Assign indices 1 and 4 to processor 0
       my_size = 2
       if(mynode().eq.0) then
         index_array(1) = 1
         index_array(2) = 4
       endif
c   Assign indices 2 and 3 to processor 1
       if(mynode().eq.1) then
         index_array(1) = 2
         index_array(2) = 3
       endif
c   set up a translation table
       itable = build_translation_table(1,index_array,my_size)
c    Processor 0 seeks processor and local indices
c     for global array indices 0 and 1 */
       if(mynode().eq.0) then
```

```
          deref_array(1) = 1
          deref_array(2) = 2
        endif
c    Processor 1 seeks processor and local indices
c     for global array indices 2 and 3 */
        if(mynode().eq.1) then
          deref_array(1) = 3
          deref_array(2) = 4
        endif

 c   Dereference a set of global variables
        call dereference(table,deref_array,local,proc,size)

c local and proc return the processors and local indices where
c global array indices are stored.
c In processor 0, proc(0) = 0, proc(1) = 1, local(0) = 1 , local(1) = 1
c In processor 1, proc(0) = 1, proc(1) = 0, local(0) = 2 , local(1) = 2

        stop
        end
```

Now assume that processor 0 needs to know to values of distributed array elements 1,2, and 4 while processor 1 needs to know the value of element 3. We call dereference to find the processors and the local indices that correspond to each global index. At this point schedule can be called and gathers and scatters carried out.

6.3 function build_reg_translation_table()

Builds a translation table for a regular distribution. It returns an id for the translation table built. The distribution could be blocked or cyclic for the current implementation. No other regular distribution is supported as before, however provisions for an analytic user function and parameterized general block distributions have been made. The translation tables returned by this function DO NOT explicitly list all global indices.

Synopsis

build_reg_translation_table (dist_type, data_size)

Parameter declarations

integer dist_type distribution type BLOCK = 1 and CYCLIC = 2

integer data_size global data array size

Return value

an integer representing a translation table

6.4 function build_dst_translation_table()

Builds a translation table for a degenerate irregular distribution. 'index' is the list of global indices for which the calling processor is responsible for. 'nindex' denotes the size of the 'index' array. 'repf' is a floating-point number between 0 and 1 (inclusive), denoting the replication factor for the translation table storage (more on this later). 'psize' is the page size for the page decomposition of the translation table contents. The function returns an ID for the table generated to be used by dereference functions.

Synopsis

build_dst_translation_table (index, nindex, repf, psize)

Parameter declarations

integer index() index list to be dereferenced

integer nindex size of the index_array

real repf replication factor

integer psize page size

Return value

an integer reprezenting a translation table

This type of translation tables list all the global indices and their location assignment in the distributed memory. The global list of indices are decomposed into pages of size 'psize' and the storage is managed in page level (rather than individual index level). The pages are distributed between the processors using block distribution. Furthermore, each processor replicates (repf*(N-N/P)) pages in its local memory, where N denotes the total number of pages and P is the number of processors in the system.

6.5 function init_ttable_with_proc()

A partial translation table that stores only processor information for a distribution can be built using the function init_ttable_with_proc().

Synopsis

function init_ttable_with_proc(part,proc,ndata)

Parameter Declarations

integer part how translation table will be mapped - may be BLOCKED(=1))

integer proc() list of owner processor numbers for a global data array

integer ndata size of of proc array

Return Value

integer which refers to the translation table corresponding to input data.

Example

6.6 subroutine free_table()

It deallocates storage space associated with a translation table.

Synopsis

free_table(trans_table)

Parameter declarations

integer trans_table translation table id

Return value

None

6.7 subroutine derefproc()

Same as dereference, but the function does not return offset assignment of global indices. Translation table argument is generic, both irregular and regular distributions uses the same function.

Synopsis

derefproc(trans_table,index_array,proc,ndata)

Parameter declarations

integer trans_table translation table table id

integer index_array() index list to be dereferenced

integer ndata size of the index_array

integer proc() list of processor numbers

Return value

None

6.8 subroutine derefoffset()

Same as dereference, but the function does not return processor assignment of global indices. Translation table argument is generic, both irregular and regular distributions uses the same function.

Synopsis

derefoffset(trans_table,index_array,local,ndata)

Parameter declarations

integer trans_table translation table table id

integer index_array() index list to be dereferenced

integer ndata size of the index_array

integer local() list of local offset

None

Return value

None

6.9 subroutine remap_table()

Updates the translation table entries to reflect the changes in the distribution of data. This function does not perform any action if the old distribution was regular. *oldt* is the translation table containing the current distribution of data. *newIndex* is the new set of global indices for which the calling processor is responsible for. *nindex* is the size of *newIndex*. The size of the global data space should be same for the new distribution and current distribution. The effect of this function is that, it updates translation table, and the result is translation table contains entries for the new data distribution. (Note: Space is re-used, user should not free the old translation table space).

Synopsis

remap_table(oldt, newIndex, nindex)

Parameter declarations

integer oldt translation table table id

integer newIndex() new index set

integer nindex size of the new index list

Return value

None

6.10 function tableGetReplication()

Returns the replication factor for a distributed translation table. For regular distribution it returns a symbolic TT_ERROR.

Synopsis

real tableGetReplication(table)

Parameter declarations

integer table translation table table id

Return value

a real number which gives the replication factor.

6.11 function tableGetPageSize()

Returns the page size for a distributed translation table. For regular distribution it returns a symbolic TT_ERROR.

Synopsis

integer tableGetPageSize(table)

Parameter declarations

integer table translation table table id

Return value

an integer number which gives the page size of the table

6.12 subroutine tableGetIndices()

Returns the list of indices owned by processors in global address.

Synopsis

tableGetIndices(table, indices, nindex)

Parameter declarations

integer table translation table table id

integer indices() local set of indices

integer nindex size of the local indices

Return value

None

7 Miscellaneous Procedures

7.1 subroutine PARTI_setup

Synopsis

PARTI_setup()

Parameter declarations

None

Return value

None

The procedure PARTI_setup is called once at the begining of the application program. This procedure sets up buffer space and environment variables.

7.2 subroutine renumber

Synopsis

renumber(table, index, array, size, rarray)

Parameter declarations

Table 4: Choas runtime data mapping procedures

Task	Function	Primitive
Data Partitioning	initializing hash table	init_rdg_hash_table()
	generating local graph	eliminate_dup_edges()
	generating distributed graph	generate_rdg()
Loop iteration Partitioning	generating loop iteration graph	dref_rig()
	partitioning loop iteration graph	iteration_partitioner()
Remap	generating schedule to remap array indices and other aligned data arrays	remap()

integer table translation table id

integer index() local index set in global address

integer array() array to be renumbered

integer size size of array index list

integer rarray() array with renumbered values

Return value

None

8 Runtime Data Distribution

In scalable multiprocessor systems, high performance demands that computational load be balanced evenly among processors and that inter-processor communication be minimized. Over the past few years a lot of study has been carried out in the area of mapping irregular problems onto distributed memory multicomputers. As a result of this, several general heuristics have been proposed for efficient data mapping. Currently these partitioners must be coupled to user programs manually. A standard interface can, however, be used to link these partitioners with programs at runtime. We use a distributed data structure to represent array access patterns that arise in particular loops of a program. This data structure is passed to the graph partitioner. The partitioner returns a data distribution. In this section, we present the CHAOS procedure that can be used to generate the distributed data structure, link the partitioners, and support data redistribution.

8.1 Runtime Data Graph

To implement the data partitioning, we generate a distributed data structure called the *Runtime Data Graph* or **RDG**. The runtime graph is generated from the distributed array access patterns in the user selected loops. Here it is assumed that all distributed arrays considered for RDG generation are to be partitioned in the same way and also that they are of the same size. Node

```
                 Selected Loop
      do i=1,5
S1      y(IA(i)) = x(IB(i))
      end do
                       Input
      IA = {1, 3, 4, 2, 5}
      IB = {4, 2, 1, 5, 5}
              Runtime Data Graph
      1 --> 4
      2 --> 3, 5
      3 --> 2
      4 --> 1
      5 --> 2
Runtime Data Graph in Compressed Row Format
      adjacency_list_array = {4, 3, 5, 2, 1, 2}
      adjacency_list_pointr = {1, 2, 4, 5, 6, 7}
```

Figure 5: An example of runtime data graph generation

i of an **RDG** represents element i of all distributed arrays if there is more than one distributed array used for graph generation.

An **RDG** is constructed by executing a modified version of the loop which forms a list of edges instead of performing numerical calculations. The graph partitioners that we consider divide the graph into equal subgraphs with as few edges as possible between them. The intent is that two nodes in RDG will be linked if one is used to compute the other and they are to be allocated to the same processor; There is an edge between nodes i and j in the RDG if, on some iteration of the loop, an assignment statement writes element i of an array (i.e. $x(i)$ appears on the left-hand side) and references element j (i.e. $y(j)$ appears on the right-hand side), or vice-versa.

Figure 5 shows a simple example of sequential generation of a runtime graph. The statement S1 in the figure has indirections, IA and IB, both on the left and the right hand sides. In this example, the arrays x and y, of the same size 5, are considered for graph generation. Each vertex in the graph represents an array element. Hence, the runtime graph will have 5 vertices. The RDG is constructed by adding an *undirected* edge between the node pairs representing the left hand side (IA(i)) and the right hand side (IB(i)) array indices, for each loop iteration i. For instance, during the first loop iteration an edge between vertex 1 and 4 and also an edge between 4 and 1 are added to the graph. The run time graph is stored in a format closely related to compressed sparse row format.

Figure 6 shows the parallel RDG generation steps. Initially, data arrays and loop iterations are divided among processors in uniform blocks. Each processor generates a local RDG using the array access patterns that occur in local loop iterations. For clarity, the local RDG is shown as an adjacency matrix in the figure. The local graph is then merged to form a distributed graph. While merging, if we view the local graph as an adjacency matrix stored in compressed row format, then processor P_0 collects all entries of the first N/P rows in the matrix from all other processors, where N is the number of nodes (array size) and P is the number of processors. Processor P_1 collects the next N/P rows of the matrix and so on. Processors remove duplicate

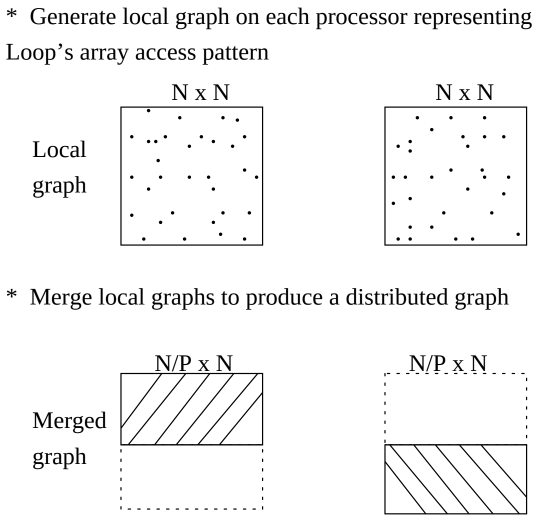

Figure 6: Parallel generation of runtime data graph

```
  do i = 1, nedges
     n1 = nde(i,1)
     n2 = nde(i,2)
     n3 = nde(i,3)
S1   y(n1) = y(n1) + x(n1) + x(n2) + x(n3)
S2   y(n2) = y(n2) - x(n1) + x(n2)
  enddo
```

Figure 7: An example sequential loop

entries when they collect adjacency list entries.

8.2 Data Mapping Procedures

In this section, we present the CHAOS procedures that can be used for data mapping on MIMD machines. The kernel shown in Fig. 7 is an example of the kind of loop that commonly occurs in a variety of sparse or unstructured code. We use this kernel as a running example to illustrate the procedures. The CHAOS procedures carry out data mapping and loop iteration partitioning in *parallel.* To perform operations in parallel, the loop iterations of the selected loops and the arrays to be partitioned are distributed among processors in uniform blocks. Figure 8 shows the modified version of the sequential loop and the initial data and loop distribution. The array *local_ind_list* in the figure has a list of local array descriptors assigned to each processor and *n_local* is the size of *local_ind_list*. The array *local_iter_list* has a list of local iteration numbers. Figure 9 shows parallel pre-processing code, for the example code in Figure 7, with data mapping procedures incorporated.

```
  do i = 1, nlocal_edges
     n1 = local_nde(i,1)
     n2 = local_nde(i,2)
     n3 = local_nde(i,3)
S1   y(n1) = y(n1) + x(n1) + x(n2) + x(n3)
S2   y(n2) = y(n2) - x(n1) + x(n2)
  enddo
```

Assuming the size of arrays x and y to be 6, value of nedges to be 10 and nde = {(1,2,3), (1,3,4), (4,2,3),(2,4,1), (2,3,6),(6,4,2), (2,4,3),(3,4,5),(3,5,4), (6,5,3)} the initial data and loop iteration distributions for 2 processors are:

Processor 0	Processor 1
local_iter_list = {1,2,3,4,5}	local_iter_list = {6,7,8,9,10}
nlocal_edges = 5	nlocal_edges = 5
local_ind_list = {1,2,3}	local_ind_list = {4,5,6}
n_local = 3	n_local = 3
local_nde = {(1,2,3), (1,3,4), (4,2,3), (2,4,1), (2,3,6)}	local_nde = {(6,4,2), (2,4,3), (3,4,5), (3,5,4), (6,5,3)}

Figure 8: Modified example code for parallel pre-processing

8.2.1 subroutine eliminate_dup_edges()

The procedure **eliminate_dup_edges** generates a local runtime graph on each processor. This procedure is called *once for each statement* in the loops that accesses the distributed arrays on both the left hand side and the right hand side of the statement. For all these statement, the left hand side (*lhs_ind*) array indices and the corresponding list of right hand side(*rhs_ind*) array indices, for each local iteration, are generated separately. For example, in Figure 9 statements S1 and S2 access the distributed arrays both on left and right hand sides. The statement S1 has 2 distinct array indices on the right hand side for each left hand side index, whereas statement S2 has only one. The left hand side and right hand side index pairs are generated separately for each statement and the procedure is called twice.

The procedure **eliminate_dup_edges** inputs the lists *lhs_ind* and *rhs_ind*, the size of *lhs_ind* (n_count) and the number of unique *rhs_ind* indices (*n_dep*) for each *lhs_ind*. This procedure generates the local graph by adding an *undirected* edge between the left hand side and right hand index lists and stores it in a hash table. An undirected edge between nodes i and j is formed by adding j to the adjacency node list of node i and vice versa.

The procedure eliminates any duplicate edges and also self edges as there is no potential for communication. The current version of the procedure does not distinguish the edges connecting the same pairs of nodes but arise due to different pairs of arrays. The future version of this procedure will take this case into account and it will produce graphs with weighted edges.

Synopsis

```
  btable = build_translation_table(type, local_ind_list, n_local)
C-- Initialize hash table to store runtime data graph
  n_dep1    = 2
  n_dep2    = 1
  hashindex = init_rdg_hash_table(2*(n_dep1+n_dep2)*nlocal_edges)
  jcount    = 1
  icount    = 0
  kcount    = 1
  lcount    = 1
  do i = 1, nlocal_edges
     n1 = local_nde(i,1)
     n2 = local_nde(i,2)
     n3 = local_nde(i,3)
C S1   y(n1) = y(n1) + x(n1) + x(n2) + x(n3)
     lhs_ind1(icount+1)  = n1
     rhs_ind1(jcount)    = n2
     jcount              = jcount + 1
     rhs_ind1(jcount)    = n3
     jcount              = jcount + 1
C S2   y(n2) = y(n2) - x(n1) + x(n2)
     lhs_ind2(icount+1)  = n2
     rhs_ind2(lcount)    = n1
     lcount              = lcount + 1
     icount              = icount + 1
  enddo
C--Generate local run time graph
   call eliminate_dup_edges(hashindex, lhs_ind1, rhs_ind1, n_dep1, icount)
   call eliminate_dup_edges(hashindex, lhs_ind2, rhs_ind2, n_dep2, icount)
C--Generate distributed run time graph
   call generate_rdg(hashindex, local_ind_list,
 &                   n_local, csr_ptr, csr_cols, ncols)
C-- call parallel graph partitioner
   call parallel_rsb(local_ind_list, n_local, csr_ptr, csr_cols, ntable)
C-- Remap array indices
   call remap(ntable, local_ind_list, sched, new_ind_list, new_size)
   call dgather(x, x, sched)
   call dgather(y, y, sched)
```

Figure 9: Parallel preprocessing code for data mapping

eliminate_dup_edges(hashindex, lhs_ind, rhs_ind, n_count, n_dep)

Parameter declarations

integer hashindex an integer identifying the hash table that stores RDG

integer lhs_ind() list of left hand side array indices of all local iterations

integer rhs_ind() a list of right hand side unique indices of each local iterations

integer n_count number of left hand side indices

integer n_dep number of unique right hand indices for each left hand side index in the considered statement

Return value

None

Example

Assume that in the example kernel shown in Figure 7, two processors are employed, and arrays x and y are to be mapped in a conforming manner and are of the same size (6).

The RDG is generated based on the access pattern of the arrays x and y in the modified loop shown Figure 9. The statements S1 and S2 in the kernel are not executed while generating the graph. Initially, the arrays x and y and the loop iterations are distributed in uniform blocks as shown in Figure 8. The modified loop is executed in parallel to form a local list of array access patterns on each processor as shown in Figure 8. The lists lhs_ind1 and rhs_ind1 in Figure 9 have the left hand side and right hand side array access patterns in statement S1 and lhs_ind2 and rhs_ind2 have the patterns in S2. All processors then call the procedure **eliminate_dup_edges** with the list of left hand side and right hand side array indices. The RDG generated by the procedure is shown in Figure 10. The RDG is stored in the hash table which is identified by an integer (*hashindex*). The RDG obtained using S1 does not get altered by the statement S2 as there are no new pair of array indices.

8.2.2 subroutine generate_rdg()

Once array access patterns in a loop have been recorded in a hash table by the procedure **eliminate_dup_edges**, the procedure **generate_rdg** can be called on each processor to flatten its local adjacency list in the hash table into an adjacency list data structure (closely related to Compressed Sparse Row (CSR) format). This procedure performs a global scatter operation (resolving collisions by appending lists) and then combines these local lists into a complete graph, also represented in CSR format. This data structure is distributed so that each processor stores the adjacency lists for a subset of the array elements.

Synopsis

generate_rdg(hashindex, local_ind_list, n_local, csr_ptr, csr_col, ncols)

Parameter declarations

Processor 0	Processor 1
lhs_ind1 = {1,1,4,2,2}	lhs_ind1 = {6,2,3,3,6}
rhs_ind1 = {2,3,3,4,2,3,4,1,3,6}	rhs_ind1 = {4,2,4,3,4,5,5,4,5,3}
lhs_ind2 = {1,1,4,2,2}	lhs_ind2 = {6,2,3,3,6}
rhs_ind2 = {2,3,2,4,3}	rhs_ind2 = {4,4,4,5,5}

after *eliminate_dup_edges* call for statement S1 - RDG in hash table

RDG = {(1,2), (2,1),(1,3),(3,1), (1,4),(4,1),(2,4),(4,2),(4,3),(3,4), (2,3), (3,2),(2,6),(6,2) }	RDG = {(6,4),(4,6),(6,2),(2,6), (2,4),(4,2),(2,3),(3,2),(3,4),(4,3), (3,5),(5,3),(6,5),(5,6),(6,3),(3,6) }

after *eliminate_dup_edges* call for statement S2 - RDG in hash table

RDG = {(1,2), (2,1),(1,3),(3,1) (1,4),(4,1),(2,4),(4,2),(4,3),(3,4), (2,3), (3,2),(2,6),(6,2) }	RDG = {(6,4),(4,6),(6,2),(2,6), (2,4),(4,2),(2,3),(3,2),(3,4),(4,3), (3,5),(5,3),(6,5),(5,6),(6,3),(3,6) }

Figure 10: Example output - eliminate_dup_edges()

integer hashindex an integer identifying the hash table which stores RDG

integer local_ind_list() list of array descriptors

integer n_local size of *local_ind_list*

integer csr_ptr() list of csr format pointers pointing into csr_col

integer csr_col() adjacency list for indices occur in local iterations

integer ncols size of csr_cols returned by generate_rdg

Return value

None

Example:

Processor 0	Processor 1
after flattening	
csr_col = {2,3,4,1,4,3,6,1,4,2,1,2,3,2}	csr_col = {6,4,3,2,4,5,6,6,2,3,6,3,4,2,5,3}
csr_ptr = {1,4,8,11,14,14,15}	csr_ptr = {1,1,4,8,11,13,17}
after merging	
csr_col = {2,3,4,1,3,4,6,1,2,4,5,6}	csr_col = {1,2,3,6,3,6,2,3,4,5}
csr_ptr = {1,4,8,13}	csr_ptr = {1,5,7,11}

Figure 11: Example output - generate_rdg()

The flattening process groups edges for each array element together and then stores in a list (csr_col). A list of pointers (csr_ptr) identifies the beginning of the edge list for each array element in csr_col. Since the flattening process uses only the local hash table, there is no communication between processors. In the merging process, processor P0 collects the adjacency list for arrays indices {1,2,3} and P1 for array indices {4,5,6} as shown in Figure 11.

8.2.3 function init_rdg_hash_table()

The hash table used in procedures **eliminate_dup_edges** and **generate_rdg** can be initialized by using the procedure **init_rdg_hash_table**. This procedure is called with the initial number of expected entries in the hash table, for memory allocation. Extra memory space is automatically allocated when the hash table entries overflow the initial allocation.

Synopsis

function init_rdg_hash_table(size)

Parameter declarations

integer size number of expected entries in the hash table

Return value

An integer identifying the hash table

8.3 Loop Iteration Partitioning Procedures

Upon identifying the new array distribution, loop iteration partitioning procedures are called to distribute the loop iterations. These procedures, by distributing the loop iterations, balance computation among processors and reduce off-processor memory accesses. Figure 12 shows pre-processing code for mapping loop iterations of the kernel shown in Figure 7. In the following section the loop iteration partitioning procedures **dref_rig** and **iteration_partitioner** are discussed.

8.3.1 subroutine dref_rig()

To partition loop iterations, we use the *Runtime Iteration Graph*(**RIG**) which lists, for each loop iteration, the distinct distributed array elements referenced. For example, in Figure 12, each loop iteration accesses three different array indices $(n1, n2, n3)$ of the distributed arrays. In this case, the **RIG** has a list of n1, n2, and n3 for all local iterations. Using the **RIG**, procedure **dref_rig** generates a *Runtime Iteration Processor Assignment graph* (**RIPA**) that has a list of processors that own array elements in the **RIG**. The future version of this procedure will support a **RIG** with a weight associated with each entry in the graph.

Synopsis

dref_rig(ttable, rig, niter, n_ref, ripa)

```
C-- Create translation table with the current loop iteration list
  ttable = build_translation_table(1, local_iter_list, nlocal_edges)
  icount = 1
  n_ref  = 3
  do i = 1, nlocal_edges
     n1 = local_nde(i,1)
     n2 = local_nde(i,2)
     n3 = local_nde(i,3)
C S1   y(n1) = y(n1) + x(n1) + x(n2) + x(n3)
C S2   y(n2) = y(n2) - x(n1) + x(n2)
     rig(icount)   = n1
     rig(icount+1) = n2
     rig(icount+2) = n3
     icount        = icount + 3
  enddo
C-- Generate runtime iteration graph
  call dref_rig(ntable, rig, nlocal_edges, n_ref, ripa)
C-- Partition runtime iteration graph
  call iteration_partitioner(ripa,  nlocal_edges, n_ref, ltable)
C-- Remap loop iterations
  call remap(ltable, local_iter_list, sched, new_iter_list, new_loop_size)
  call igather(local_nde(1,1),  local_nde(1,1), sched)
  call igather(local_nde(1,2),  local_nde(1,2), sched)
  call igather(local_nde(1,3),  local_nde(1,3), sched)
```

Figure 12: Example code with loop iteration partitioning procedures

Parameter declarations

- **integer ttable** an integer identifying the distribution translation table which describes the distribution arrays returned by the partitioner
- **integer rig()** list of distinct indices accessed for each local iteration
- **integer niter** number of local iterations
- **integer n_ref** number of unique indices accessed per iteration
- **integer ripa()** list of processors that own each entry in rig

Return value

None

8.3.2 subroutine iteration_partitioner()

The current version of mapper procedure **iteration_partitioner** inputs the **RIPA** and assigns iterations to processors by assigning each iteration to the processor that owns the most data.

Processor 0	Processor 1
new_ind_list = 2,4,5	new_ind_list = 1,3,6
n_ref = 3	n_ref = 3
rig = {1,2,3,1,3,4,4,4,2,3,2,4,1,2,3,6}	rig = {6,4,2,2,4,3,3,4,5,3,5,4,6,5,3}
after dref_rig	
ripa = {1,0,1,1,1,0,0,0,1,0,0,1,0,1,1}	ripa = {1,0,0,0,0,1,1,0,0,1,0,0,1,0,1}
after iteration_partitioner and *remap* followed by *gather*	
new_iter_list = {3,4,6,7,8,9}	new_iter_list = {1,2,5,10}

Figure 13: Example output - dref_ref() and iteration_partitioner()

The new loop iteration distribution is described by a translation table.

Synopsis

iteration_partitioner(ripa, niter, n_ref, itable)

Parameter declarations

integer ripa() list of processors that own each entry in rig

integer niter number of local iterations

integer n_ref number of unique indices accessed per iteration

integer itable an integer identifying distribution translation table

Return value

None

Example

Assuming array elements {2,4,5} are distributed to P0 and array elements {1,3,6} are distributed to P1 in Figure 12 and loop iterations are partitioned initially as shown in Figure 8, the Figure 13 shows the new loop distribution.

The procedure **dref_rig** returns a list of processor numbers(ripa) which own indices referred in each local iteration. This list is obtained by dereferencing the translation_table (*ttable*). The *ttable* describes the new distribution of arrays. Note that the loop iteration partitioning can only be done after the arrays are remapped (Section 8.4) based on the new distribution returned by the partitioner. The procedure **iteration_partitioner** assigns a loop iteration to a processor which owns the maximum number of indices accessed in that iteration. Ties are broken arbitrarily. The procedure returns the loop iteration mapping for its initial list of iterations (*local_iter_list*) in the form of a translation table(*ltable*). Each processor then calls the procedure **remap** to obtain a schedule (*sched*). This schedule can be used to get the new iteration list (*new_iter_list*) using CHAOS data exchager (*gather*).

8.4 Data Remapping

Once the new array distribution has been identified the array index list must be remapped based on the new distribution. The data arrays associated with the distributed array must also be remapped. The procedure **remap** inputs the translation table (*newtable*) describing the new array mapping and a list of initial array indices (*local_ind_list*). It returns a schedule (*sched*) and a new index list(*new_ind_list*). A schedule stores send/receive patterns and this can be used to move data among processors using the CHAOS data exchangers described in Section 4. The returned schedule can be used to remap the data arrays associated with the distributed array descriptors.

8.4.1 subroutine remap()

Synopsis

remap(newtable, local_ind_list, sched, new_ind_list, nind)

Parameter declarations

integer newtable translation table index describing new mapping

integer local_ind_list() list of initial local indices (in global numbering)

integer sched schedule index returned

integer new_ind_list() list of new local indices (in global number ing)

integer nind number of new local indices

Return value

None

Example

In Figure 9, initially arrays x and y and any data arrays associated with them are distributed in a conforming manner. The procedure **parallel_rsb**, on each processor, returns a new distribution for the initial local array elements of x and y. This distribution is returned in the form of a translation table (*newtable*). The procedure **remap** returns a new list of indices (*new_ind_list*) for arrays x and y based on the new partition for each processor. It also returns a schedule (*sched*) which is used to remap the data arrays. These associated arrays are actually transported using the CHAOS data exchange procedures(**gather**).

9 Parallel Partitioners

In this section we present parallel partitioners that are distributed along with the CHAOS runtime library. The first two partitioners, namely recursive coordinate bisection and inertial bisection partitioners, use spatial information. The third parallel partitioner, recursive spectral bisection, uses graph connectivity information.

Table 5: Parallel Partitioners

Partitioner	Input	Procedure
Recursive coordinate bisection	geometry	CoorBisecMap()
	geometry	CoorBisec()
	geometry and weight	CoorWeighBisecMap()
	geometry and weight	CoorWeighBisec()
Recursive coordinate bisection	geometry	InerBisecMap()
	geometry	InerBisec()
	geometry and weight	InerWeighBisecMap()
	geometry and weight	InerWeighBisec()
Spectral bisection	connectivity	parallel_rsb()

9.1 Geometry Based Partitioners

There are two types of geometry partitioners: ones which use computational load information and ones which don't. There are implemented in two fashions: coloring and remapping. In the 'coloring' implementation, only processor assignment of each element are returned, whereas in the 'remapping' implementation, arrays used to specify the geometry information are automatically redistributed and the size of data arrays and indices of new local elements are returned. Therefore, there are four possible versions of partitioners as shown in Table 5.

9.1.1 Recursive Bisection (Coloring)

The procedure PREFIXBisecMap returns a processor assignment list *maparray*. PREFIX can be 'Coor' for recursive coordinate partitioner and 'Iner' for Inertial bisection partitioner.

PREFIXBisecMap (maparray, ndata, ndim, x, [, arg...])

Parameters

integer maparray() result of partitioning

integer ndata number of elements

integer ndim number of dimensions

double precision x() {, arg...} coordinate arrays

9.1.2 Weighted Recursive Bisection (Coloring)

Here the partitioners use an additional information to partition data. Computational load at each point is also considered for partitioning. The load is specified by an array *load*.

PREFIXWeighBisecMap (maparray, load, ndata, ndim, x [, arg...])

Parameters

integer maparray() result of partitioning

integer load() loads of elements

integer ndata number of elements

integer ndim number of dimensions

double precision x() {, arg...} ; coordinate arrays

9.1.3 Recursive Bisection with Remapping

PREFIXBisec (remaplevel, myindex, ndata, ndim, x [, arg...])

Parameters

integer remaplevel remap arrays every remaplevel levels

integer myindex() indexes of local elements

integer ndata() number of elements

integer ndim number of dimensions

double precision x() {, arg...} ; coordinate arrays

The arrays used to specify the geometry information are automatically remapped to the new distribution. The parameter remaplevel is used to specify how often that coordinate arrays should be remaped, i.e. the coordinate arrays will be remapped and moved every remaplevel levels.

The parameter myindex is used to keep track of how elements are remapped and moved. Users input the indexes of current local elements. The partitioner returns indexes of new local elements after remapping.

Users place the current number of local elements in 'ndata'. Partitioners returns the number of new local elements in 'ndata'.

9.1.4 Weighted Recursive Bisection with Remapping

PREFIXWeighBisec (remaplevel, myindex, load, ndata, ndim, x [, arg...])

Parameters

integer remaplevel remap arrays every remaplevel levels

integer myindex() indexes of local elements

integer load() loads of elements

integer ndata number of elements

integer ndim number of dimensions

double precision x() {, arg...} coordinate arrays

9.2 Recursive Spectral Partitioner

parallel_rsb (myindex, n_local, csr_ptr, csr_col, ntable)

Parameters

integer myindex() indexes of local elements

integer n_local number of local elements

integer csr_ptr() list of csr format pointers pointing into csr_col

integer csr_col() adjacency list for all local indices

integer ntable distribution returned in the form a translation table

The csr format representation of RDG (see Section 8) is passed to the procedure **parallel_rsb**. The parallel partitioner returns a pointer to a translation table (*ntable*) which describes the new array distribution. The parallel version is based on the sequential single level spectral partitioner provided by Horst Simon. For efficiency reasons, user might want to use the multilevel version of the partitioner.

References

[1] R. Das, Y.-S. Hwang, M. Uysal, J. Saltz, and A. Sussman. Applying the CHAOS/PARTI library to irregular problems in computational chemistry and computational aerodynamics. In *Proceedings of the Scalable Parallel Libraries Conference, Mississippi State University, Starkville, MS*, pages 45–56. IEEE Computer Society Press, October 1993.

[2] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect access to distributed arrays. Technical Report CS-TR-3076 and UMIACS-TR-93-42, University of Maryland, Department of Computer Science and UMIACS, May 1993. Appears in LCPC '93.

[3] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. Technical Report CS-TR-3163 and UMIACS-TR-93-109, University of Maryland, Department of Computer Science and UMIACS, October 1993. Submitted to Journal of Parallel and Distributed Computing.

[4] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[5] Seema Hiranandani, Joel Saltz, Piyush Mehrotra, and Harry Berryman. Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, 12(4):415–422, August 1991.

[6] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 140–152, July 1988.

[7] R. Ponnusamy, R. Das, J. Saltz, J. Saltz, and D. Mavriplis. The dybbuk runtime system. In *IEEE COMPCON*, San Francisco, February 1993.

[8] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Proceedings Supercomputing '93*, pages 361–370. IEEE Computer Society Press, November 1993. Also available as University of Maryland Technical Report CS-TR-3055 and UMIACS-TR-93-32.

[9] Ravi Ponnusamy, Joel Saltz, Alok Choudhary, Yuan-Shin Hwang, and Geoffrey Fox. Runtime support and compilation methods for user-specified data distributions. Technical Report CS-TR-3194 and UMIACS-TR-93-135, University of Maryland, Department of Computer Science and UMIACS, November 1993. Submitted to IEEE Transactions on Parallel and Distributed Systems.

[10] J. Saltz, H. Berryman, and J. Wu. Runtime compilation for multiprocessors. *Concurrency: Practice and Experience*, 3(6):573–592, 1991.

[11] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.

[12] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[13] Shamik D. Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel Saltz. Run-time and compile-time support for adaptive irregular problems. Submitted to Supercomputing 1994, April 1994.

[14] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, and K. Crowley. PARTI primitives for unstructured and block structured problems. *Computing Systems in Engineering*, 3(1-4):73–86, 1992. Papers presented at the Symposium on High-Performance Computing for

[15] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 2, pages 26–30, 1991.

A Example Program 1 : NEIGHBORCOUNT

A.1 Sequential Version of Program

```
PROGRAM NeighborCount
CC    The purpose of this program is to take a graph and determine each node's
CC in/out degree.  The graph is represented by arrays X, Y, edge1 and edge2
CC where arrays X and Y are the vertices' placement on a grid, and edge1 and
CC edge2 represent the end points of each edge.

parameter (idim=43)
parameter (NE=125)

INTEGER edge1(NE), edge2(NE)
INTEGER IO_Degree(idim), X(idim), Y(idim)


C Initialize edges arrays
edge1(1) = INT(rand()*idim) + 1
edge2(1) = INT(rand()*idim) + 1
do 200 i = 2, NE
  edge1(i)=INT(rand()*idim) + 1
  edge2(i)=MOD(INT(rand()*idim-1)+edge1(i),idim)+1
 200 continue

C Initialize (X,Y) coord arrays (not used in program
X(1) = INT(rand()*1000) + 1
Y(1) = INT(rand()*1000) + 1
do 300 i = 1, idim
  X(i) = INT(rand()*1000) + 1
  Y(i) = INT(rand()*1000) + 1
 300 continue

do 400 i = 1, idim
  IO_Degree(i) = 0
 400 continue


C Calculate neighbors
do 100 i = 1, NE
IO_Degree(edge1(i)) = IO_Degree(edge1(i))+1
IO_Degree(edge2(i)) = IO_Degree(edge2(i))+1
 100 continue


CC    The output is in the form of :
CC
```

```
CC  Node #   IO_Degree

do 110 i = 1, idim
  print *,i,IO_Degree(i)
  tot = tot + IO_Degree(i)
 110  continue

print *,'Total = ',tot


end
```

A.2 Parallel Version of Program (1) : Regular (block) Distribution

```
PROGRAM NeighborCount
CC    The purpose of this program is to take a graph and determine each node's
CC in/out degree.  The graph is represented by arrays X, Y, edge1 and edge2
CC where arrays X and Y are the vertices' placement on a grid, and edge1 and
CC edge2 represent the end points of each edge.

CC    In this version of the program, the X, Y, edge1, edge2 and IO_Degree
CC arrays are initialized assuming a block distribution.

#include "mpi.fort.h"

integer idim, ne, buff
C idim is the number of vertices in the graph
parameter (idim=43)
C NE is the number of edges in the graph
parameter (NE=125)
C buff is the amouunt of buffer space being allocated for
C off proceessor data storage
parameter (buff=NE)

INTEGER build_reg_translation_table

INTEGER rand

C edge1(i) and edge2(i) hold the endpoints for edge #i
INTEGER edge1(NE),edge2(NE)

C newedge1 and newedge2 hold the adjusted values of the endpoints
C  (adjusted based on which vertices "my" processor has and
C  where it has them)
INTEGER newedge1(NE + buff),newedge2(NE + buff)

C edge1 and edge2 are combined into tempedgearr so tempedgearr can be
C passed to XXXXXXXXXXXX and the adjusted values are returned
C in newtempedgearr
INTEGER tempedgearr(NE + buff), newtempedgearr(NE + buff)

INTEGER IO_Degree(idim + buff)
C The X and Y arrays represent the x,y coordinates of the vertices
INTEGER X(idim), Y(idim)

INTEGER sch, tt, count, tot
INTEGER BLOCKarr1(idim)
```

```
        INTEGER size(512), loops(512)


call PARTI_setup()
procs=MPI_numnodes()

C Each processor must know how many everyone else has to create BLOCKarrs
        do 50 i = 1,procs
                size(i) = idim/procs
                if (i-1 .lt. (idim - size(i)*procs)) size(i)=size(i)+1
 50     continue
        mysize=size(MPI_mynode()+1)


        do 60 i = 1,procs
                loops(i)=NE/procs
                if (i-1 .lt. (NE - loops(i)*procs)) loops(i)=loops(i)+1
 60     continue
        myloops=loops(MPI_mynode()+1)


C Initialize edges arrays
do 100 i = 1, myloops
  edge1(i)=MOD(INT(rand()),idim) + 1
  edge2(i)=MOD(MOD(INT(rand()),idim-1)+edge1(i),idim)+1
 100 continue


C Initialize (X,Y) coord arrays (not used in this program)
do 200 i = 1, idim
  X(i) = MOD(INT(rand()),1000) + 1
  Y(i) = MOD(INT(rand()),1000) + 1
 200 continue

C  Initialize IO_Degree Array
do 250 i = 1, myloops
  IO_Degree(i) = 0
 250 continue


C Initialize BLOCK distribution arrays to represent initial
C   distribution of the data and indirection arrays
        offset=0
        do 301 i = 1, MPI_mynode()
          offset=offset + size(i)
```

```
 301    continue
        do 302 i = 1, mysize
          BLOCKarr1(i) = offset + i
 302    continue


C Partition Data
tt = build_reg_translation_table(1,idim)



C Inspector
CC    The values in the edge end point indirection arrays are adjusted using
CC localize to refer to local indices for on-processor data segments and to
CC buffer space for off-processor data segments.
do 600 i = 1, myloops
  tempedgearr(i) = edge1(i)
     tempedgearr(i+myloops) = edge2(i)
 600 continue

call localize(tt, sch, tempedgearr, newtempedgearr, myloops*2,
     $ noff, mysize, 1)

do 650 i = 1, myloops
  edge1(i) = newtempedgearr(i)
     edge2(i) = newtempedgearr(i+myloops)
 650 continue


C Executor
C Calculate neighbors
do 700 i = 1, myloops
  IO_Degree(edge1(i)) = IO_Degree(edge1(i))+1
  IO_Degree(edge2(i)) = IO_Degree(edge2(i))+1
 700 continue


CC    After the local portion of the indirection arrays have been polled, each
CC processor sends any information which it has stored in its buffer areas to
CC the processor which owns the associated node.
call iscatter_add(IO_Degree(mysize+1),IO_Degree,sch)


CC    The output is in the form of :
CC
CC  Home Processor #   Node #   IO_Degree
```

```
tot = 0
do 800 i = 1, mysize
print *,MPI_mynode(),BLOCKarr1(i),IO_Degree(i)
tot=tot+IO_Degree(i)
 800  continue

CC   The following information is printed out for debugging purposes to
CC make sure that the correct number of edge end points were counted
print *,MPI_mynode(),'‘s TOTAL =',tot

call MPI_gisum(tot,1,ibuf)
print *,'Total Total = ',tot

print *,'Total should be ',2*NE

end
```

A.3 Parallel Version of Program (2) : Irregular Distribution

```
PROGRAM NeighborCount
CC    The purpose of this program is to take a graph and determine each node's
CC in/out degree.  The graph is represented by arrays X, Y, edge1 and edge2
CC where arrays X and Y are the vertices' placement on a grid, and edge1 and
CC edge2 represent the end points of each edge.

CC    In this version of the program, the X, Y, edge1, edge2 and IO_Degree
CC arrays are initialized assuming a block distribution.

#include "mpi.fort.h"

parameter (idim=43)
parameter (NE=125)
parameter (buff=NE)

INTEGER build_translation_table

INTEGER edge1(NE),edge2(NE)
INTEGER newedge1(NE + buff),newedge2(NE + buff)
INTEGER arr(NE + buff), new2edge(NE + buff)
INTEGER IO_Degree(idim + buff),IO_Degree2(idim + buff)
DOUBLE PRECISION  X(idim), Y(idim)

INTEGER sch1, sch2, sch3, tt1, tt2, count, tot
INTEGER BLOCKarr1(idim)
INTEGER BLOCKarr2(NE)
INTEGER maparr(idim), newdistarr1(idim + buff)
INTEGER newdistarr2(2 * NE + buff)
INTEGER rig(NE * 2), ripa(NE * 2)

INTEGER size(512), loops(512)

call PARTI_setup()
procs=MPI_numnodes()

C Each processor must know how many everyone else has to create BLOCKarrs
do 50 i = 1,procs
size(i) = idim/procs
if (i-1 .lt. (idim - size(i)*procs)) size(i)=size(i)+1
 50 continue
mysize=size(MPI_mynode()+1)


do 60 i = 1,procs
```

```
loops(i)=NE/procs
if (i-1 .lt. (NE - loops(i)*procs)) loops(i)=loops(i)+1
 60  continue
myloops=loops(MPI_mynode()+1)

C Initialize edges arrays
edge1(1) = INT(rand()*idim) + 1
edge2(1) = INT(rand()*idim) + 1
do 100 i = 2, myloops
  edge1(i)=INT(rand()*idim) + 1
  edge2(i)=MOD(INT(rand()*idim-1)
     $ +edge1(i),idim)+1
 100 continue

C Initialize (X,Y) coord arrays
X(1) = INT(rand()*1000) + 1
Y(1) = INT(rand()*1000) + 1
do 200 i = 1, idim
  X(i) = INT(rand()*1000) + 1
  Y(i) = INT(rand()*1000) + 1
 200 continue

C  Initialize IO_Degree Array
do 250 i = 1, myloops
  IO_Degree(i) = 0
 250 continue

C Initialize BLOCK distribution arrays to represent initial
C   distribution of the data and indirection arrays
offset=0
do 301 i = 1, MPI_mynode()
  offset=offset + size(i)
 301  continue
do 302 i = 1, mysize
  BLOCKarr1(i) = offset + i
 302 continue

offset=0
do 401 i = 1, MPI_mynode()
  offset=offset + loops(i)
 401 continue
do 402 i = 1, myloops
  BLOCKarr2(i) = offset + i
 402 continue

C Partition Data
```

```
CC    The X and Y coordinates are used to create a partitioning scheme for the
CC IO_Degree data arrays.  The data arrays are then redistributed from block to
CC the new distribution.
call PARTI_setup()
call CoorBisecMap(maparr, mysize, 2, X, Y)
tt1 = init_ttable_with_proc(1, maparr, mysize)

C ReMap data from old distribution to new distribution
call remap(tt1, BLOCKarr1, sch1, newdistarr1, newnumlocalind)
    tt1 = build_translation_table(1, newdistarr1, newnumlocalind)
call igather(IO_Degree2, IO_Degree, sch1)
mysize=newnumlocalind

C Partition Loops
CC    Then the loops are partitioned using the new distribution information in
CC order to try to maximize on-processor work.  The indirection arrays are then
CC gathered so that each processor has the portions of the indirection arrays
CC which correspond to the loop iterations they own.
c   collect all local ind array values for IO_Degree into rig
count = 1
do 500 i = 1, myloops
  rig(count) = edge1(i)
  rig(count+1) = edge2(i)
  count = count + 2
 500 continue

call dref_rig(tt1, rig, myloops, 2, ripa)
call iteration_partitioner(ripa, myloops, 2, tt2)
call remap(tt2, BLOCKarr2, sch2, newdistarr2, newmyloops)

call igather(newedge1, edge1, sch2)
call igather(newedge2, edge2, sch2)



C Inspector
CC    The values in the edge end point indirection arrays are adjusted using
CC localize to refer to local indices for on-processor data segments and to
CC buffer space for off-processor data segments.
do 600 i = 1, newmyloops
  arr(i) = newedge1(i)
     arr(i+newmyloops) = newedge2(i)
 600 continue
call localize(tt1, sch3, arr, new2edge, newmyloops*2,
     $ noff, mysize, 1)
```

```
C Executor
C Calculate neighbors
do 700 i = 1, newmyloops
IO_Degree2(new2edge(i)) = IO_Degree2(new2edge(i))+1
IO_Degree2(new2edge(i+newmyloops))=IO_Degree2(new2edge(i+newmyloops))+1
 700 continue


CC    After the local portion of the indirection arrays have been polled, each
CC processor sends any information which it has stored in its buffer areas to
CC the processor which owns the associated node.
call iscatter_add(IO_Degree2(mysize+1),IO_Degree2,sch3)


CC    The output is in the form of :
CC
CC  Home Processor #   Node #   IO_Degree
tot = 0
do 800 i = 1, mysize
print *,MPI_mynode(),newdistarr1(i),IO_Degree2(i)
tot=tot+IO_Degree2(i)
 800  continue

CC   The following information is printed out for debugging purposes to
CC make sure that the correct number of edge end points were counted
print *,MPI_mynode(),'`s TOTAL =',tot

call MPI_gisum(tot,1,ibuf)
print *,'Total Total = ',tot

print *,'Total should be ',2*NE

end
```

B Example Program 2 : Simplified ISING

B.1 Sequential Version of Program

```
program ising_simulation
parameter (ixdim=64)
parameter (iydim=64)
parameter (frac=20)
parameter (particles=16*ixdim*iydim)
parameter (timeunits=10)

C  This program will simulate a grid of cells and the energy particles
C within those cells.  The grid will be initialized in a uniformly
C distributed pattern.  The simulation will allow a particle to
C move one cell left, right up or down (or stay in place) per time
C unit.  For this program it is assumed there is a one-way permeable
C wall through which particles can leave, but not enter the simulation.

integer x(particles), y(particles)
integer count
INTEGER rand

C Initialize Grid (assuming perfect divisibility)
        do 100 i = 1, particles
          x(i) = MOD(rand(),ixdim) + 1
          y(i) = MOD(rand(),iydim) + 1
 100    continue

C Do "timeunits" particle movement cycles
        do 200 k = 1, timeunits
        do 300 i = 1, particles
          if ( MOD(rand(),100) .lt. frac ) then
                ival = MOD(rand(),4)
                if (ival.eq.0) then
                        y(i)=y(i)-1
                else
                if (ival.eq.1) then
                        y(i)=y(i)+1
                else
                if (ival.eq.2) then
                        x(i)=x(i)-1
                else
                if (ival.eq.3) x(i)=x(i)+1
                endif
                endif
                endif
          endif
```

```
          if (x(i).lt.1) x(i)=ixdim
          if (x(i).gt.ixdim) x(i)=1
          if (y(i).lt.1) y(i)=iydim
          if (y(i).gt.iydim) y(i)=1

 300 continue
 200 continue

C Calculate number of particles left in grid (can do something else instead)

print *,'Particles in grid : ', particles

end
```

B.2 Parallel Version of Program

```
program ising_simulation
#include "mpi.fort.h"

parameter (ixdim=64)
parameter (iydim=64)
parameter (frac=20)
parameter (particles=16*ixdim*iydim)
parameter (timeunits=100)

C  This program will simulate a grid of cells and the energy particles
C within those cells.  The grid will be initialized in a uniformly
C distributed pattern.  The simulation will allow a particle to
C move one cell left, right up or down (or stay in place) per time
C unit.  For this program it is assumed that when a particle leaves
C  the top, it goes to the bottom, and the same for left and right
C and vice-versa.

integer x(particles), y(particles)
integer sched, schedule_proc
integer dest_proc(particles)
integer count,part
INTEGER rand

call PARTI_setup()
procs=MPI_numnodes()
menode=MPI_mynode()
part=INT(ixdim/procs)
localloops=particles/procs

C Random number "hack" so that all processors don't re-use same random
C    numbers.
do 50 i = 1, menode * localloops
  l=rand()
 50 continue

C Initialize Grid (assuming perfect divisibility)
C  NOTE : The x-coordinates are appropriate for the given processor
do 100 i = 1, localloops
  x(i) = MOD(rand(),part) + part*menode + 1
  y(i) = MOD(rand(),iydim) + 1
 100 continue

C Do "timeunits" particle movement cycles
do 200 k = 1, timeunits
```

```
do 300 i = 1, localloops
  if (iprocmap(x(i),ixdim).ne.menode) print *,'Wrong Processor'
  if ( MOD(rand(),100) .lt. frac ) then
ival = MOD(rand(),4)
if (ival.eq.0) then
y(i)=y(i)-1
else
if (ival.eq.1) then
y(i)=y(i)+1
else
if (ival.eq.2) then
x(i)=x(i)-1
else
if (ival.eq.3) x(i)=x(i)+1
endif
endif
endif
  endif
  if (x(i).lt.1) x(i)=ixdim
  if (x(i).gt.ixdim) x(i)=1
  if (y(i).lt.1) y(i)=iydim
  if (y(i).gt.iydim) y(i)=1
C Create destination processor array listing which processor each particle
C    should be on for the next time iteration
  dest_proc(i) = iprocmap(x(i),ixdim)
 300 continue
C Creating schedule for moving particles between machines
sched = schedule_proc(dest_proc,localloops,newlocalloops,1)
localloops=newlocalloops
C Moving particles' x and y coordinates to processor which "owns" the
C     cell in which that particle resides
call iscatter_append(x,x,sched)
call iscatter_append(y,y,sched)
C Freeing memory used by schedule since this schedule is never useful again
call free_sched(sched)
 200 continue

C Check number of particles left in grid (can do something else instead)
print *,'Particles on machine ',menode,': ',localloops

call MPI_gisum(localloops,1,itemp)
if (menode.eq.0) print *,'Total = ',localloops

end
```

```
function iprocmap(xcoord,xdim)
integer xcoord,xdim

  iprocmap = INT((xcoord-1)/(INT(xdim/MPI_numnodes())))

end
```