NASA Contractor Report 187442

ICASE INTERIM REPORT 13

A Manual for PARTI Runtime Primitives

Harry Berryman
Joel Saltz

NASA Contract No. NAS1-18605
September 1990

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

ICASE INTERIM REPORTS

ICASE has introduced a new report series to be called ICASE Interim Reports. The series will complement the more familiar blue ICASE reports that have been distributed for many years. The blue reports are intended as preprints of research that has been submitted for publication in either refereed journals or conference proceedings. In general, the green Interim Report will not be submitted for publication, at least not in its printed form. It will be used for research that has reached a certain level of maturity but needs additional refinement, for technical reviews or position statements, for bibliographics, and for computer software. The Interim Reports will receive the same distribution as the ICASE Reports. They will be available upon request in the future, and they may be referenced in other publications.

Robert G. Voigt
Director

A Manual for PARTI Runtime Primitives[1]

Harry Berryman and Joel Saltz

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23665
and
Computer Science Department
Yale University
New Haven, CT 06520

ABSTRACT

Primitives 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 primitives are also designed for use in compilers for distributed memory multiprocessors. Communications patterns are captured at runtime, and the appropriate send and receive messages are automatically generated.

[1] Research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665. Additional support provided by NSF grant ASC-8819374.

1 Did Somebody Say PARTI?

1.1 Overview

PARTI stands for "Parallel Automated Runtime Toolkit at ICASE." Development of PARTI has been carried out at Yale University as well as ICASE and hence has been referred to as "PARTY" in some earlier papers. The PARTI runtime primitives are designed to help users to efficiently program loops found in irregular problems (e.g. unstructured mesh sweeps, sparse matrix codes, adaptive mesh partial differential equations solvers). These primitives are also designed for use in compilers for distributed memory multiprocessors. In the context of the PARTI project, we are also developing a variety of other tools including compilers for distributed machines. These primitives are some of the basic building blocks we are using in our efforts.

The primitives in this distribution run on any of the iPSC/2 or iPSC/860 machines produced by Intel Scientific Computing. They could easily be modified to run on most distributed memory machines. This document describes the operation of the PARTI primitives and gives several examples of how to use them. The rationale of the PARTI system (the PARTI line, as it were) was presented in [2] and summarized in [4]. The mechanisms incorporated in these primitives have been outlined in [2], [5], [4]. PARTI has been used in a variety of applications, including sparse matrix linear solvers, adaptive computational fluid dynamics codes, and in a prototype compiler [4] aimed at distributed memory multiprocessors.

1.2 Primitives Available in the Release

The PARTI system is divided into several levels. Level 0 primitives allow processors to access the distributed memory of a multiprocessor with a modicum of convenience. Level 1 primitives bind mapping information to arrays. This allows the user to store and manipulate constructs that describe multiprocessor mappings of distributed multidimensional arrays. Included with this distribution are the level 0 primitives outlined next.

The level 0 *scatter* allows each processor of a distributed memory machine to move data to off-processor memory locations. The level 0 *gather* allows each processor to obtain copies of data from memory locations in other processors. Level 0 primitives are provided to support initialization and access of distributed translation tables. Such distributed tables allow a user to assign globally numbered indices to processors in an irregular pattern. By using a distributed translation table, it is possible to avoid replicating records of where distributed array elements are stored in all processors. Level 0 primitives also carry out off-processor accumulations; e.g. any processor can add to the contents of an off-processor memory location.

1.3 Primitives that exist but are not yet distributed

There are additional level 0 primitives not included with this release that support local caching of copies of off-processor data. These Level 0 primitives are presented in [3] and will be available in future PARTI releases. Level 1 primitives, also not available with this release, allow users to specify how distributed arrays are to be mapped onto sets of processors. The level 1 primitives support read, write and accumulate accesses to these mapped multidimensional arrays. The level 1 primitives also allow users to dynamically remap distributed arrays. The Level 1 primitives are described in [1]. It should be noted that use of PARTI primitives do not interfere with access to traditional message passing communications primitives. In particular, a user can call all of the iSC supplied routines when using PARTI.

2 Installation

The PARTI primitives come in a single tar file. To install, change to the directory where you wish to put the PARTI subdirectory and type:

```
tar xof parti.tar
```

This should create the following directory structure:

parti/examples/unst sweep over unstructured mesh, described in section 4.

parti/examples/free a conjugate gradient linear equation solver cg.c and cg_host.c not discussed in this documentation. (Free prize included in every copy of PARTI!). Also included is simple.c, a simple example involving several of the primitives.

parti/src source for the PARTI primitives

parti/tests test programs to verify correct installation

A makefile should be present in the PARTI directory. At the beginning of this makefile are several macros to be modified by the user.

NFLAG This macro is passed to the C compiler and linker when compiling and/or linking node programs. It should have one of the following values:

-node -sx for iPSC/2 machines with weitek floating point accelerators

-node -i860 for iPSC/860 machines

-node for vanilla iPSC/2 machines

NARC This macro indicates the archive to be used in creating the PARTI library. It should be set to one of the following:

ar for any iPSC/2

ar860 for an iPSC/860

LIB This macro should be set to the directory where the party library will be installed. It is prudent to use the full path name here. This directory must exist before the system is installed.

INCL This macro should be set to the directory where the PARTI include files will reside. It is prudent to use the full path name here. This directory must exist before the system is installed.

NPROCS This indicates the largest number of processors that the tests should be run on. Eight and sixteen are good values.

Make sure that the directories pointed to by `LIB` and `INCL` exist. If they do not, any attempt to install the party system there will fail. There are several objects to make. Typing the following make commands in the listed order should be sufficient to install and check the PARTI system on your computer.

make will compile the PARTI library but not install it in the designated directories.

make install will install the PARTI system in the designated directories.

make clean will remove object and executable file from various subdirectories.

make test will run several tests to see if everything has been compiled correctly.

3 Function Descriptions

3.1 Header Files

There are two header files which go with the PARTI library. The first is `parti.h`. This file contains the definitions of all structures, macro definition and function definitions needed to run the PARTI primitives. *It must be included in all programs that use the PARTI system.* The second include file, `parti_more.h`, is used only when the system is compiled. It defines such things as message types, and static buffer lengths. It should not be necessary to include this file in applications which use PARTI.

Two of the primitives `schedule` and `build_translation_table` are functions that carry out preprocessing. `schedule` and `build_translation_table` allocate elements of structures `schedule_struct` and `trans_table` and then return pointers to structures. The above structures are defined in parti.h; macro definitions define `struct schedule_struct` as SCHED and define `struct trans_table` as TTABLE. parti.h also defines macros STRIPED and BLOCKED used in the procedure `build_translation_table`.

3.2 Level 0 primitives

Level 0 primitives consist of routines to *gather* and/or *scatter* (read and write) values to elements of one dimensional arrays $\texttt{aloc}^j$ defined on each processor j. Each $\texttt{aloc}^j$ is local to processor j; it is not viewed as a distributed array by the Level 0 Primitives.

Level 0 gathers and scatters are accomplished by using three routines: `schedule`, `gather` , and `scatter` . `Gather` corresponds to the "gather exchanger" in [1], similarly `scatter` corresponds to "scatter exchanger."

`Schedule` on processor P^i is passed a list of indices K^j into each $aloc^j$ from which data is to be fetched and produces a schedule `S` that can be used by either `gather` or `scatter`.

On processor P^i, gather (or `PREFIXgather` in section 3.4) inputs

1. a buffer into which the fetched elements are to be placed
2. the location of array $aloc^i$
3. the schedule `S` produced by `schedule`

`gather` executes sends and receives that fetch from each processor P^j the appropriate elements from the array $aloc^j$. Then it places these elements into the user-supplied buffer. `Scatter` (or `PREFIXscatter`) is passed

1. a buffer from which each scattered datum is to be obtained
2. the location of array $aloc^i$
3. the schedule `S` produced by Schedule

`Scatter` executes sends and receives that put on each processor P^j the appropriate elements from the buffer. Then `scatter` places these elements into the appropriate elements of array $aloc^j$.

In addition to the Level 0 exchanger, we have developed versions of gathers and scatters that perform remote operations on distributed array data. For example, the PREFIXscatter_add adds data elements $D_1, ..., D_{n_j}$ to elements $aloc^j(k_1)$,.., $aloc^j(k_{n_j})$. Similar exchanges perform distributed subtractions and multiplications.

Level 0 primitives have also been developed to support the declaration and use of distributed translation tables. These distributed translation tables can be used to describe distributed data array mappings (see discussion on indirect distributions, section 3.7).

3.3 schedule()

This procedure carries out the preprocessing needed for carrying out optimized gather exchanger and scatter exchanger routines. Every processor must participate in this procedure call. On each processor, a schedule is passed a list of processors and local indices from which a gather procedure on that processor can later obtain data (or to which a scatter procedure on that processor can later write data). schedule returns a pointer to a structure of type SCHED, this pointer is used in gather, scatter and scatter_FUNC operations (Sections 3.4, 3.5, 3.6.

Synopsis

SCHED *schedule(local,proc,ndata)

Parameter declarations

int *local local index to be gathered from or scattered to

int *proc processors to be gathered from or scattered to

int ndata number of data involved in gather or scatter

Return value

Returns pointer to structure of type SCHED which can be used in PREFIXgather, PREFIXscatter, PREFIXscatter_add, PREFIXscatter_sub, PREFIXscatter_mult.

Example

Node 0 schedules a fetch of elements 1 and 2 from a (so far unspecified) array on node 1; node 1 schedules a fetch of element 1 from an array on node 0 and 0 from an array on node 1.

```
int local[2], proc[2], ndata;
SCHED *schedinfo;



if(mynode()==0){
  proc[0] = 1;
  local[0] = 1;
  proc[1] = 1;
  local[1] = 2;
  ndata = 2;
}

if(mynode()==1){
  proc[0] = 0;
  local[0] = 1;
  proc[1] = 1;
  local[1] = 0;
  ndata = 2;
}


schedinfo = schedule(local,proc,ndata);
```

3.4 PREFIXgather()

PREFIX can be d (double precision), i (integer) , f (floating point) or c (character) This procedure is the gather exchanger procedure described above and in [1]. PREFIXgather uses a schedule produced by a call to schedule, the schedule is passed to PREFIXgather in structure SCHED schedinfo. Copies of data values obtained from other processors are placed in memory pointed to by buffer. Also passed to PREFIX gather is a pointer to the location from which data is to be fetched *on the calling processor.* This pointer is designated here as aloc, aloc corresponds to $aloc^i$ above and in [1].

Synopsis

void PREFIXgather(schedinfo,buffer,aloc)

Parameter Declarations

SCHED *schedinfo information obtained from schedule's preprocessing of reference pattern

TYPE *buffer pointer to buffer for copies of gathered data values

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

Return Value

None

Example

We assume that schedule has already been called with the parameters presented in Section 3.3. Our example will assume that we wish to gather double precision numbers, i.e. that we will be calling dgather. 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.

```
double buffer[2], aloc[3];
SCHED *schedinfo;


 for(i=0;i<3;i++){
   aloc[i] = mynode() + 0.1*i;
 }

 dgather(schedinfo,buffer,aloc);
```

On processor 0, buffer[0] and buffer[1] are now equal to 1.1 and 1.2. On processor 1, buffer[0] and buffer[1] are now equal to 0.1 and 1.0.

3.5 PREFIXscatter()

PREFIX can be d (double precision), i (integer) , f (floating point) or c (character). This procedure is the scatter exchanger procedure described above and in [1]. PREFIXscatter uses a schedule produced by a call to schedule, the schedule is passed to PREFIXscatter in structure SCHED schedinfo. Copies of data values to be scattered to other processors are placed in memory pointed to by buffer. Also passed to PREFIX scatter 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, aloc corresponds to $aloc^i$ above and in [1].

Synopsis

void PREFIXscatter(schedinfo,buffer,aloc)

Parameter Declarations

SCHED schedinfo information obtained from schedule's preprocessing of reference pattern

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

Return Value

None

Example

We assume that schedule has already been called with the parameters presented in Section 3.3. Our example will assume that we wish to scatter double precision numbers, i.e. that we will be calling dscatter. On each processor, *aloc points to the arrays to which

values are to scattered. *buffer points to the location from which will be obtained data that will be scattered The processor and local_array index to which the values are to be scattered was designated during an earlier call to schedule.

```
double buffer[2], aloc[3];
SCHED *schedinfo;


 for(i=0;i<3;i++){
   aloc[i] = 10.0;
 }

 if(mynode()==0){
   buffer[0] = 444.44;
   buffer[1] = 555.55;
 }

 if(mynode()==1){
   buffer[0] = 666.66;
   buffer[1] = 777.77;
 }

 dscatter(schedinfo,buffer,aloc);
```

On processor 0, the first three elements of aloc are 10.0, 666.66 and 10.0. On processor 1, the first three elements of aloc are 777.77, 444.44 and 555.55.

3.6 PREFIXscatter_FUNC()

PREFIX can be d (double precision), i (integer) , f (floating point) 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

void PREFIXscatter_FUNC(schedinfo,buffer,aloc)

Parameter Declarations

SCHED *schedinfo information obtained from schedule's preprocessing of reference pattern.

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.

Return Value

None

Example

We assume that schedule has already been called with the parameters presented in Section 3.3. Our example will assume that we wish to scatter and add double precision numbers, i.e. that we will be calling dscatter_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 will be obtained the values to be scattered and added. The processor and local_array index to which the values are to be scattered and added was designated during an earlier call to schedule.

```
double buffer[2], aloc[3];
SCHED *schedinfo;


 for(i=0;i<3;i++){
   aloc[i] = 10.0;
 }

 if(mynode()==0){
   buffer[0] = 444.44;
   buffer[1] = 555.55;
}

if(mynode()==1){
  buffer[0] = 666.66;
  buffer[1] = 777.77;
}

dscatter_add(schedinfo,buffer,aloc);
```

On processor 0, the first three elements of aloc are 10.0, 676.66 and 10.0. On processor 1, the first three elements of aloc are 787.77, 454.44 and 565.55.

3.7 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_ta 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 = $\lceil M/NP \rceil$.

build_translation_table returns a pointer to a structure of type TTABLE; this pointer is used in dereference, defined in section 3.8.

Synopsis

TTABLE *build_translation_table(part,indexarray,ndata)

Parameter Declarations

int part how translation table will be mapped - may be BLOCKED or STRIPED

int *indexarray each processor P specifies list of globally numbered indices for which P will be responsible

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

Return Value

structure of type TTABLE; this structure contains a given processor's portion of the distributed translation table

Example

An example to demonstrate the use of both build_translation_table and dereference can be found in Section 3.8.

3.8 dereference()

dereference accesses distributed translation table constructed in build_translation_table.

dereference is passed a pointer to a structure of type TTABLE; this structure defines the irregularly distributed mapping and was created in procedure build_translation_table. 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

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

Parameter declarations

int *global list of global indices we wish to locate in distributed memory

int *local local indices obtained from the distributed translation table that correspond to the global indices passed to dereference

Table 1: Values obtained by dereference

Processor	proc[0]	local[0]	proc[1]	local[1]
0	0	0	1	0
1	1	1	0	1

int *proc array of distributed translation table processor assignments for each global index passed to dereference

int ndata number of elements to be dereferenced

TTABLE *index_table distributed translation table datastructure created in build_translation_ta

Return value

None

Example

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 initialize a translation table. Processor 0 calls build_translation_table and assigns indices 0 and 3 to processor 0, processor 1 calls build_translation_table and assigns indices 1 and 2 to processor 1. The translation table is partitioned between processors in blocks.

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

```
#include <stdio.h>
#include "parti.h"

main()
{
  int size, i, *index_array;
  int *deref_array;
  int *local, *proc;
  TTABLE *table;


  size = 2;
  index_array = (int *) malloc(sizeof(int)*size);
  deref_array = (int *) malloc(sizeof(int)*size);
  local = (int *) malloc(sizeof(int)*size);
  proc = (int *) malloc(sizeof(int)*size);

/*Assign indices 0 and 3 to processor 0 */
if(mynode()==0)
    {
    index_array[0] = 0;
    index_array[1] = 3;
    }
/*Assign indices 1 and 2 to processor 1 */
if(mynode()==1)
    {
    index_array[0] = 1;
    index_array[1] = 2;
    }

  /* set up a translation table */

  table = build_translation_table(BLOCKED,index_array,size);


/* Processor 0 seeks processor and local indices
for global array indices 0 and 1 */
if(mynode()==0)
    {
    deref_array[0] = 0;
    deref_array[1] = 1;
    }
```

```
/* Processor 1 seeks processor and local indices
for global array indices 2 and 3 */
if(mynode()==1)
    {
    deref_array[0] = 2;
    deref_array[1] = 3;
    }

 /* Dereference a set of global variables */

  dereference(table,deref_array,local,proc,size);

/* local and proc return the processors and local indices where
global array indices are stored.
In processor 0, proc[0] = 0, proc[1] = 1, local[0] = 0 , local[1] = 0;
In processor 1, proc[0] = 1, proc[1] = 0, local[0] = 1 , local[1] = 1;
*/
}
```

Now assume that processor 0 needs to know to values of distributed array elements 0,1, and 3 while processor 1 needs to know the value of element 2. 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.

4 Example: A Sweep over an Unstructured Mesh

The following example can be found in the distribution, in file unst.c in the examples directory. This unstructured mesh sweep program inputs mapping information from a host using procedure get_unst_mesh(). build_translation_table and dereference are then employed to allow the user to partition indices between processors in an irregular fashion. gen_fetch_list produces a list of off-processor array elements that need to be fetched, these lists are then passed to schedule. schedule calculates the information needed to carry out the mesh sweep, sweep.

There is also a host program, unst_host.c. The host program is not described here.

```
/**************************************************************/
/* PARTI program to sweep over an arbitrary unstructured mesh */
/*                                                            */
/* This program reads in an unstructured mesh structure,      */
/* and carries out a sweep over the unstructured mesh.        */
/* This is the node program. The host program (unst_host.c)   */
/* is required to run this, as is a data file in a format     */
/* described in the comments of the host program. In this     */
/* program, the unstructured  mesh is stored in a global data */
/* structure. This program:                                   */
/*                                                            */
/*     1) gets unstructured mesh (w/ help from unst_host.c)   */
/*     2) does lots of memory and address stuff on it         */
/*     3) generates a vector x                                */
/*     4) multiplies x by the matrix, getting y               */
/*                                                            */
/* by Scott Berryman, ICASE/NASA Langley Research Center      */
/* 30 Aug 1990                                                */
/**************************************************************/

#include <cube.h>
#include <stdio.h>
#include <math.h>
#include "parti.h"

/* define maximum size of sparse matrix */

#define MAX_NONZEROS 163840
#define MAX_ROWS     32768

/* sparse matrix data structure in traditional CSR format */

  int Size, Myrows, Nrows, Mynonzeros;
  int Cols[MAX_NONZEROS], Ncols[MAX_ROWS];
  float Vals[MAX_NONZEROS];
  /* Extra data structures needed for parallel version: */
  /*                                                    */
  /* Row[] contains a list of matrix rows for which     */
  /* a given processor is to be responsible.            */
  /*                                                    */
  /* Local[j],Proc[j] represent the proc/offset pair    */
  /*        for column j.                               */
  /* Fetch_p[i],Fetch_l[i] represent the proc/offset    */
  /*        of the ith off-processor column.            */

  int Row[MAX_ROWS], Local[MAX_NONZEROS], Proc[MAX_NONZEROS];
  int *Fetch_p, *Fetch_l, Nfetch;


main()
{
  int i, j;
  TTABLE *table;
  SCHED *sr;
  float *x, *y;

  /* Get unstructured mesh from unst_host.c. The source for this procedure
  is in the distribution but is not described here.           */

  get_unst_mesh();

  /* Build the translation table.                             */
  /* IN: Row[i]                 OUT: table                    */

  table = build_translation_table(BLOCKED,Row,Myrows);

  /* Look up address of Cols and put them in Local and Proc.  */
  /* This step identifies what local and off-processor array  */
  /* locations will be involved in the mesh sweep.            */
  /* IN: Cols[i],table          OUT: Local[i],Proc[i]         */
  dereference(table,Cols,Local,Proc,Mynonzeros);

  /* Loop through all proc/offset pairs and decide which        */
  /* must be fetched from other processors.                     */
  /* IN: Local[i],Proc[i]      OUT: Fetch_l[i],Fetch_p[i]       */

  gen_fetch_list();

  /* Allocate memory for vectors.  Set x[i] = i for local i.    */

  x = (float *) malloc(sizeof(float)*Myrows);
  y = (float *) malloc(sizeof(float)*Myrows);
  for (i=0; i<Myrows; i++) x[i] = i;

  /* Build the communications schedule.                         */
  /* IN: Fetch_l[i],Fetch_p[i]   OUT: sr                        */

  sr = schedule(Fetch_l,Fetch_p,Nfetch);

  /* Do a sweep over the unstructured mesh.                     */

  sweep(sr,x,y);

}



/* Unstructured mesh sweep
   (requires the schedule be built and passed in).              */

sweep(sr,x,y)
SCHED *sr;     /* <--- communication schedule */
float *x, *y;  /* <--- input and result vectors */
{
  int myproc, bcount, count, i, j;
  float *buffer;

  /* allocate local buffer to gather data into */

  buffer = (float *) malloc(sizeof(float)*Nfetch);

  /* gather data using previously computed communications schedule */

  fgather(sr,buffer,x);


  myproc = mynode();
  bcount = 0;
  count = 0;
  for(i=0;i<Myrows;i++){
    y[i] = -1.0*x[i]/(float)Ncols[i];
    for(j=0;j<Ncols[i];j++){  /* for each nonzero link .... */
      if(Proc[count]==myproc){  /* if col[count] is local */
        y[i] += x[Local[count]];
      }else{                    /* otherwise look in buffer */
        y[i] += buffer[bcount++];
      }
      count++;
    }
  }
  free(buffer);
}


/* This function takes the Local[i],Proc[i] */
/* address for each nonzero col in the matrix*/
/* and puts nonlocal ones into Fetch_l[i],Fetch_p[i] */
gen_fetch_list()
{
  int count,i,myproc;
  myproc = mynode();
  /* count offnode refs */
  Nfetch = 0;
  for(i=0;i<Mynonzeros;i++){
   Nfetch += (Proc[i]!=myproc);
  }

   /* for each ref */
  Fetch_p = (int *) my_malloc(sizeof(int)*Nfetch*2);
  Fetch_l = &Fetch_p[Nfetch];
  count = 0;
  for(i=0;i<Mynonzeros;i++){
    if(Proc[i]!=myproc){    /* if Col[i] refers to an off-proc location.. */
      Fetch_p[count] = Proc[i]; /* add it to the fetch list */
      Fetch_l[count] = Local[i];
      count++;
    }
  }
}

/* local definition of malloc to catch running out of memory */
long my_malloc(n)
long n;
{
  long tmp;

  tmp = malloc(n);
  if(((char *) tmp) == NULL){
    printf("Out of memory on node %d.\n",mynode());
    exit();
  }
  return(tmp);
}
```

5 Acknowledgements

We would like to thank Seema Hiranandani, Jeff Scroggs and Janet Wu for their help in debugging the primitives presented here. We also thank Janet Wu for her formulation of the build_translation_table primitive. We would like to thank Adam Rifkin for his careful proofing of this manual. Finally, we would like to thank Bob Voigt and Martin Schultz for their support during this project's long (and continuing) incubation period. It takes time to put together a good PARTI!

References

[1] H. Berryman, J. Saltz, and J. Scroggs, *Execution time support for adaptive scientific algorithms on distributed memory machines*ICASE Report 90-41, May 1990.

[2] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and K. Crowley, *Principles of runtime support for parallel processors*, in Proceedings of the 1988 ACM International Conference on Supercomputing , St. Malo France, July 1988, pp. 140–152.

[3] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman, *A scheme for supporting automatic data migration on multicomputers*, in Proceedings of the Fifth Distributed Memory Computing Conference, Charleston S.C., 1990.

[4] J. Saltz, H. Berryman, and J. Wu, *Runtime compilation for multiprocessors*, ICASE Report 90-59, 1990.

[5] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, *Run-time scheduling and execution of loops on message passing machines*, Journal of Parallel and Distributed Computing, 8 (1990), pp. 303–312.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA CR-187442 ICASE Interim Report 13	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle A MANUAL FOR PARTI RUNTIME PRIMITIVES		5. Report Date September 1990
		6. Performing Organization Code
7. Author(s) Harry Berryman Joel Saltz		8. Performing Organization Report No. Interim Report No. 13
		10. Work Unit No. 505-90-21-01
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		11. Contract or Grant No. NAS1-18605
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Interim Report

16. Abstract

Primitives 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 primitives are also designed for use in compilers for distributed memory multiprocessors. Communications patterns are captured at runtime, and the appropriate send and receive messages are automatically generated.

17. Key Words (Suggested by Author(s)) distributed memory, sparse matrix, unstructured mesh, tools, primitives, compiler	18. Distribution Statement 59 - Mathematical and Computer Sciences (General) 61 - Computer Programming and Software Unclassified - Unlimited

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 26	22. Price A03

NASA-Langley, 1990