Logistics

• Any issues with Pthreads or MPI on
  – Osage/Redwing?
  – G4?
  – IBM SP @ NERSC???

• Pthread homework (#4) due today
  – Any Questions????????

• START THE MPI HOMEWORK NOW!
  – The due date will NOT change
    ▪ 4/22/2005
  – Using the queues takes time!
  – If you wait until the last minute that is YOUR fault.
Program Skeleton for GA/MPI

/* initialize MPI space */
if (MPI_SUCCESS != (MPI_Init(&argc, &argv)))
    (void)fprintf(stderr,"MPI_Init failed\n");

/*---------------- Initialize GA and MA----------------*/
GA_Initialize( );
stack = 100000; /* stack size per node*/;
heap = 100000; /* heap size per node*/;
if (!(MA_init(MT_F_DBL,stack,heap)))
    GA_Error("MA_init failed",(stack+heap));

... ... do_something_useful( );

GA_Terminate( );
MPI_Finalize( );
A Code initialization snippet [2]

GA_sync();
X_ROWS = argvalrow;
X_COLS = argvalcol;
LEN_Y  = X_COLS;
LEN_Z  = X_ROWS;
/*--- Start set timers-------------*/
Wtimeall = -Wall_Time();  Ctimeall = -CPU_Time();
/*-- Create GA entities -------------------*/
dims[0] = X_ROWS; dims[1] = X_COLS;
g_X    = NGA_Create(MT_F_DBL, 2, dims, "X matrix", NULL);
g_Y    = NGA_Create(MT_F_DBL, 1, &LEN_Y, "Y vector", NULL);
g_Zcomp = NGA_Create(MT_F_DBL, 1, &LEN_Z, "Z computed vector", NULL);
g_Zanal = GA_Duplicate(g_Zcomp,"Z analytic vector");
A Code initialization snippet [3]

/*------- zero memory allocated -----------------*/
GA_Zero(g_X);                      GA_Zero(g_Y);
GA_Zero(g_Zcomp); GA_Zero(g_Zanal);
if (me == 0) {
    (void)printf(" Matrix/Vector Sizes
    X(%7d,%7d)*Y(%7d) = Z(%7d)\n",
                              X_ROWS,X_COLS,LEN_Y,LEN_Z);
}
(void)printf(" Matrix/Vector Distributions\n"));
GA_Print_distribution(g_X);
GA_Print_distribution(g_Y);
GA_Print_distribution(g_Zcomp);
GA_Print_distribution(g_Zanal);
void genX(int g_buffer, int rowdim_X, int coldim_X) {
    a = A_VAL;
    b = B_VAL;
    c = C_VAL;
    dim_of_GA = GA_Ndim(g_buffer);
    me=GA_Nodeid();
    if (dim_of_GA != 2)
        GA_Error("genX: expected 2 dimensional GA got:",dim_of_GA);
    NGA_Inquire(g_buffer, &type, &ndim, dims_GA);
    NGA_Distribution(g_buffer,me,dimlo,dimhi);
GenX the GA way [2]

\[ lds = \text{dimhi}[1]-\text{dimlo}[1]+1; \]
\[ \text{patchsize} = (\text{dimhi}[0]-\text{dimlo}[0]+1) * (lds); \]
\[ \text{buffer\_local} = (\text{double} \,*\, \text{mymalloc}(\ldots," \ldots")); \]
\[ \text{count} = 0; \]
\[ \text{for} \ (i = \text{dimlo}[0]; i \leq \text{dimhi}[0]; i++) \ { \]
\[ \quad \text{for} \ (j = \text{dimlo}[1]; j \leq \text{dimhi}[1]; j++) \ { \]
\[ \quad \quad \text{buffer\_local}[\text{count}] = \]
\[ \quad \quad \quad (\text{double})(i*\text{dims\_GA}[1] + j + 1); \]
\[ \quad \quad \text{count}++; \]
\[ \quad } \]
GenX the GA way [3]

NGA_Put(g_buffer,dimlo,dimhi,buffer_local,&lds);
printf("matrix printed in C order\n");
GA_Print(g_buffer);

/* get a patch back a X[0:1,0:2] or 2x3 patch*/
dimlo[0]=0;dimhi[0]=1; dimlo[1]=0;dimhi[1]=2;
lds=3;
set_it(buffer_local,patchsize,(double)-1.0);
NGA_Get(g_buffer,dimlo,dimhi,buffer_local,&lds);
printf("matrix patch printed in C order\n");
(void)free(buffer_local);
Output on 1 process

Total number of processes 1; full rank of matrix X(5,4)
Matrix/Vector Sizes X( 5, 4)*Y( 4) = Z( 5)
Matrix/Vector Distributions
Array Handle=-1000 Name:'X matrix' Data Type:double
Array Dimensions:5x4
Process=0 owns array section: [0:4,0:3]
Array Handle=-999 Name:'Y vector' Data Type:double
Array Dimensions:4
Process=0 owns array section: [0:3]
Array Handle=-998 Name:'Z computed vector' Data Type:double
Array Dimensions:5
Process=0 owns array section: [0:4]
Array Handle=-997 Name:'Z analytic vector' Data Type:double
Array Dimensions:5
Output on 1 process [2]

Process=0 owns array section: [0:4]
g_buffer=-1000,dim_of_GA=2,me=0
g_buffer=-1000,dim_of_GA=2,me=0
<0> dimlo=0 dimhi=4
<1> dimlo=0 dimhi=3
matrix printed in C order

1.00000  2.00000  3.00000  4.00000
5.00000  6.00000  7.00000  8.00000
9.00000 10.00000 11.00000 12.00000
13.00000 14.00000 15.00000 16.00000
17.00000 18.00000 19.00000 20.00000
### Output on 1 process [3]

**global array:** \( \text{X matrix}[1:4,1:5] \), handle: -1000

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00000</td>
<td>5.00000</td>
<td>9.00000</td>
<td>13.00000</td>
<td>17.00000</td>
</tr>
<tr>
<td>2</td>
<td>2.00000</td>
<td>6.00000</td>
<td>10.00000</td>
<td>14.00000</td>
<td>18.00000</td>
</tr>
<tr>
<td>3</td>
<td>3.00000</td>
<td>7.00000</td>
<td>11.00000</td>
<td>15.00000</td>
<td>19.00000</td>
</tr>
</tbody>
</table>

\[
\text{buff}[0][0]<0> = 1.000000e+00 \\
\text{buff}[0][1]<1> = 2.000000e+00 \\
\text{buff}[0][2]<2> = 3.000000e+00 \\
\text{buff}[1][0]<3> = 5.000000e+00 \\
\text{buff}[1][1]<4> = 6.000000e+00 \\
\text{buff}[1][2]<5> = 7.000000e+00 \\
\]

---
Output on 1 process [4]

matrix patch printed in C order
1.00000   2.00000   3.00000
5.00000   6.00000   7.00000

/* get a patch back */
dimlo[0]=0;dimhi[0]=1;
dimlo[1]=0;dimhi[1]=2;
lds=3;
Output on 2 procs

ARMCI configured for 2 cluster nodes
Total number of processes 2; full rank of matrix $X(5,4)$
Matrix/Vector Sizes $X(5,4) \times Y(4) = Z(5)$
Matrix/Vector Distributions
Array Handle=-1000 Name:'X matrix' Data Type:double
Array Dimensions:5x4
Process=0 owns array section: [0:4,0:1]
Process=1 owns array section: [0:4,2:3]
Array Handle=-999 Name:'Y vector' Data Type:double
Array Dimensions:4
Process=0 owns array section: [0:1]
Process=1 owns array section: [2:3]
Array Handle=-998 Name:'Z computed vector' Data Type:double
Array Dimensions:5
Process=0 owns array section: [0:2]
Process=1 owns array section: [3:4]
Output on 2 procs [2]

Array Handle=-997 Name: 'Z analytic vector' Data Type: double
Array Dimensions: 5
Process=0 owns array section: [0:2]
Process=1 owns array section: [3:4]

Matrix/Vector Distributions
<0> dimlo=0 dimhi=4
<1> dimlo=2 dimhi=3

matrix printed in C order
3.00000  4.00000
7.00000  8.00000
11.00000 12.00000
15.00000 16.00000
19.00000 20.00000
Output on 2 procs [3]

buff[0][0]<0> = 1.000000e+00
buff[0][1]<1> = 2.000000e+00
buff[0][2]<2> = 3.000000e+00
buff[1][0]<3> = 5.000000e+00
buff[1][1]<4> = 6.000000e+00
buff[1][2]<5> = 7.000000e+00
matrix patch printed in C order
  1.00000  2.00000  3.00000
  5.00000  6.00000  7.00000
<0> dimlo=0 dimhi=4
<1> dimlo=0 dimhi=1
matrix printed in C order
  1.00000  2.00000
  5.00000  6.00000
  9.00000 10.00000
13.00000 14.00000
17.00000 18.00000
Output on 2 procs [4]

global array: X matrix[1:4,1:5], handle: -1000

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00000</td>
<td>5.00000</td>
<td>9.00000</td>
<td>13.00000</td>
<td>17.00000</td>
</tr>
<tr>
<td>2</td>
<td>2.00000</td>
<td>6.00000</td>
<td>10.00000</td>
<td>14.00000</td>
<td>18.00000</td>
</tr>
<tr>
<td>3</td>
<td>3.00000</td>
<td>7.00000</td>
<td>11.00000</td>
<td>15.00000</td>
<td>19.00000</td>
</tr>
</tbody>
</table>

matrix patch printed in C order

1.00000  2.00000  3.00000
5.00000  6.00000  7.00000
GA MPL requirements

• What does GA need from the message passing library (MPL) it uses.
  ➢ initialization and termination of processes in an SPMD program,
  ➢ synchronization,
  ➢ functions that returns the number of processes and calling process id,
  ➢ broadcast,
  ➢ reduction operation for integer and double datatypes, and
  ➢ a function to abort the running parallel job in case of an error.
Other GA requirements

- The MPL must be
  - Initialized before the GA library initialization.
  - Terminated after the GA library is terminated.
- GA provides two functions
  - ga_nnodes returns the number of processes
  - ga_nodeid the calling process id (MPI-like rank)
  - Starting with release 3.0, these functions return the same values as their message-passing counterparts.
- Although MPL offer their own barrier (global synchronization) function,
  - this operation does not wait for completion of the outstanding GA communication operations.
  - The ga_sync( ) operation should be used for synchronization
    - It has the desired effect of waiting for all the outstanding GA operations to complete.
    - Essentially a GA aware wrapper around the MPL “barrier”
There are two ways to deal with data

• For routines that use the GA toolkit in a distributed fashion there are two ways to deal with the data

  ➢ One way is to use the actual distribution (where possible) that GA has assigned.
    ▪ Often requires too much memory
  ➢ Block the routine based on an arbitrary designation of work load.
    ▪ Balance computation and communication.
    ▪ Control over memory utilization.
Function genX full GA blocks

me=GA_Nodeid();
NGA_Inquire(g_buffer, &type,&dim_of_GA,dims_GA);
NGA_Distribution(g_buffer,me,dimlo,dimhi);
lds = dimhi[1]-dimlo[1]+1;
patchsize = (dimhi[0]-dimlo[0]+1)*(lds);
buffer_local = (double *)mymalloc(patchsize," local buffer in
   genX",MT_F_DBL,&handle_loc_X);
count = 0;
for (i=dimlo[0];i<=dimhi[0];i++) {
   for (j=dimlo[1];j<=dimhi[1];j++) {
      buffer_local[count] =
         sqrt((a*(double)i+b*(double)j+c));
      count++;   }  }
NGA_Put(g_buffer,dimlo,dimhi,buffer_local,&lds);
(void)myfree(buffer_local);
Function genX with blocking

block_size_i = 500;  block_size_j = 500;
NGA_Inquire(g_buffer, &type, &dim_of_GA, dims_GA);
NGA_Distribution(g_buffer, me, dimlo, dimhi);
patchsize = block_size_i*block_size_j;
buffer_local = (double *)mymalloc(patchsize," local buffer in genX",MT_F_DBL,&handle_loc_patch_X);
for (ilo=dimlo[0];ilo<=dimhi[0];ilo += block_size_i) {
    ihi = MMIN(dimhi[0],ilo + block_size_i-1);
    for (jlo=dimlo[1];jlo<=dimhi[1];jlo += block_size_j) {
        jhi = MMIN(dimhi[1],jlo + block_size_j-1);
        count = 0;
        for (i=ilo;i<=ihi;i++) {
            for (j=jlo;j<=jhi;j++) {
                buffer_local[count] = sqrt((a*(double)i+b*(double)j+c));
                count++;
            }
        }
        putlo[0]=ilo;putlo[1]=jlo;puthi[0]=ihi;puthi[1]=jhi;
        lds = jhi - jlo + 1;
        NGA_Put(g_buffer,putlo,puthi,buffer_local,&lds);}}
(void)myfreebuffer_local);
GA_Create

• If array can’t be created return a failure status.
• Otherwise an integer handle is returned.
  ➢ This handle represents a global array object in all operations involving that array.
  ➢ This is the only piece of information the programmer needs to store for that array.
    § All the properties of the object (data type, distribution data, name, number of dimensions and values for each dimension) can be obtained from the library based on the handle at any time
    § It is not necessary to keep track of this information explicitly in the application code.
      ♦ You need only keep track of the handles for each object.

• Note that regardless of the distribution type at most one block can be owned on a processor.
Destroying Arrays

- **Global arrays can be destroyed by calling the function**
  - C: `void GA_Destroy(int g_a)`
  - Fortran: `subroutine ga_destroy(g_a)`

- **They take as an argument a handle representing a valid global array.**
  - It is a fatal error to call `ga_destroy` with a handle pointing to an invalid array.

- **All active global arrays are destroyed implicitly when the user calls `ga_terminate`.**
  - Do not rely on this 😊
GA one-sided operations

• GA provides one-sided, noncollective communication operations that access data in global arrays without cooperation with the process or processes that hold the referenced data.
  - These processes do not know what data items in their own memory are being accessed or updated by remote processes.
  - The calling process does not even have to know process ids and location in memory where the referenced data resides.

• The one-sided operations that GA provides fall into three categories:
  - Remote blockwise write/read
    - ga_put, ga_get
  - Remote atomic update
    - ga_acc, ga_read_inc, ga_scatter_acc
  - Remote elementwise write/read
    - ga_scatter, ga_gather
Put and Get Operations

• Put and get are two powerful operations for inter-process communication, performing remote write and read.
  ➢ The data is simply accessed as if it were in shared memory, via the indices of the global object.

• Put copies data from the local array to the global array section, which is
  ➢ void NGA_Put(int g_a, int lo[], int hi[], void *buf, int ld[])

• All the arguments are provided in one call:
  ➢ lo and hi specify where the data should go in the global array; ld specifies the stride information of the local array buf.
  ➢ It is only required to present the n-dimensional view of the local memory buffer, that by itself might be one-dimensional.
Put and Get Operations [2]

• The operation is transparent to the user, which means the user doesn't have to worry about where the region defined by lo and hi is located.
  - It can be in the memory of one or many remote processes,
  - owned by the local process, or
  - even mixed (part of it belongs to remote processes and part of it belongs to a local process).

• Get is the reverse operation of put. It copies data from a global array section to the local array. It is
  - void NGA_Get(int g_a, int lo[], int hi[], void *buf, int ld[])

• Similar to put,
  - lo and hi specify where the data should come from in the global array, and ld specifies the stride information of the local array buf.
  - The local array must have the stride properly represented

Accumulate

- Accumulate performs atomic remote update to a patch in the GA.
- Since the operations are atomic, the same portion of a global array can be referenced by these operations issued by multiple processes and the GA will assure the correct and consistent result of the updates.
- Accumulate combines the data from the local array with data in the global array section, which is
  - void NGA_Acc(int g_a, int lo[], int hi[], void *buf, int ld[], void *alpha)
- The function performs
  - global array section (lo[], hi[]) += alpha * buf
Read and Increment

- **Read_inc remotely updates a particular element in the global array, which is**
  - `long NGA_Read_inc(int g_a, int subscript[], long inc)`
- **This function applies to integer arrays only.**
- **It atomically reads and increments an element in an integer array.**
- **It performs**
  - `a(subscripts) += inc`
  - `and returns the original value (before the update) of a(subscript).`
Scatter

- Scatter transfers a specified set of elements to a GA.
  - one-sided operation
- Scatter puts array elements into a global array, which is:
  - `void NGA_Scatter(int g_a, void *v, int *subsarray[], int n)`
- It performs (in C notation)
- `for(k=0; k<= n; k++) {
  a[subsArray[k][0]][subsArray[k][1]][subsArray[k][2]]... = v[k];
}`
Scatter Example

• **Scatter the 5 elements into a 10x10 global array**

  - **Element 1**  \( v[0] = 5 \)
    - \( \text{subsArray}[0][0] = 2 \)
    - \( \text{subsArray}[0][1] = 3 \)
  - **Element 2**  \( v[1] = 3 \)
    - \( \text{subsArray}[1][0] = 3 \)
    - \( \text{subsArray}[1][1] = 4 \)
  - **Element 3**  \( v[2] = 8 \)
    - \( \text{subsArray}[2][0] = 8 \)
    - \( \text{subsArray}[2][1] = 5 \)
  - **Element 4**  \( v[3] = 7 \)
    - \( \text{subsArray}[3][0] = 3 \)
    - \( \text{subsArray}[3][1] = 7 \)
  - **Element 5**  \( v[4] = 2 \)
    - \( \text{subsArray}[4][0] = 6 \)
    - \( \text{subsArray}[4][1] = 3 \)
Scatter example result.
Gather

• Gather is the reverse operation of scatter.
  - One-sided operation
• It gets the array elements from a global array into a local array.
  - void NGA_Gather(int g_a, void *v, int *subsarray[], int n)
• It performs (in C notation)
• for(k=0; k<= n; k++){
  - v[k] =
    a[subsArray[k][0]][subsArray[k][1]][subsArray[k][2]]...;
• }

Periodic One-Sided Operations

• Periodic interfaces to the one-sided operations support some computational fluid dynamics problems on multidimensional grids.
  ➢ They provide an index translation layer that allows the use of put, get, and accumulate operations possibly extending beyond the boundaries of a global array.
  ➢ The references that are outside of the boundaries are wrapped to the appropriate values of the global array.
  ➢ If you have an application that requires their use make sure you read the user manual carefully.
    ▪ We don’t have such an application in CS/CE 425.
Inter-process Synchronization

- **GA provides three types of synchronization calls to support different synchronization styles.**
  - **Lock with mutex:**
    - is useful for a shared memory model.
    - One can lock a mutex, to exclusively access a critical section.
  - **Fence:**
    - Guarantees that the GA operations issued from the calling process are complete.
    - The fence operation is local.
  - **Sync: (a barrier)**
    - It synchronizes processes and ensures that all Global Array operations completed.
    - Sync operation is collective.
Mutex/Lock example

• Use one mutex and the lock mechanism to enter the critical section.
  
  status = GA_Create_mutexes(1);
  if(!status) {
      GA_Error("ga_create_mutexes failed",0);
  }
  
  GA_Lock(0);
  ... do something in the critical section
  NGA_Put(g_a, ...);

  ...
  
  GA_Unlock(0);
  if(!GA_Destroy_mutexes())
      GA_Error("mutex not destroyed",0);
Fence Operations

• Fence blocks the calling process until all the data transfers corresponding to the GA operations initiated by the calling process complete.

• The typical scenario used is:
  1. Initialize the fence
  2. Global array operations
  3. Fence

• This would guarantee the operations between step 1 and 3 are complete.