HeteroMPI Programmers' Reference and Installation Manual

HeteroMPI:
A Message Passing Library for Heterogeneous Networks of Computers
Version 1.3.0

Ravi Reddy, Alexey Lastovetsky

School of Computer Science and Informatics, University College Dublin, Belfield, Dublin 4, Ireland

E-mail: Manumachu.Reddy@ucd.ie, Alexey.Lastovetsky@ucd.ie

May 1, 2009

CONTENTS

1 Introduction

The tools designed for programming high-performance computations on HNOCs must provide mechanisms to automate the following tedious and error-prone tasks:

- Parameter determination characterizing the computational requirements of the parallel application and the capabilities of the machines,
- Data partitioning,
- Matching and Scheduling, and
- Task execution.

Ideally a tool must supply mechanisms to the programmer so that he or she can provide information to it that can assist in finding the most efficient implementation on HNOCs. Combining the system's detailed analysis with the programmer's high-level knowledge of the application is essential in finding more efficient mappings than either one alone is capable of achieving. The performance models used by the tools must take into account all the essential features underlying applications run on HNOCs, mainly, the speeds of the processors, the effects of paging and the speed and the bandwidth of the communication links between the processors. The model of the executing network of computers must take into consideration the essential set of machine characteristics such as computing bandwidth, communication latency, communication overhead, communication bandwidth, network contention effects and the memory hierarchy. Such a model must have enough parameters for it to be effective and accurate.

HeteroMPI is such a tool, which is an extension of MPI for programming high-performance computations on heterogeneous networks of computers. The main idea of HeteroMPI is to automate the process of selection of a group of processes, which would execute the heterogeneous algorithm faster than any other group. HeteroMPI provides features that allow the user to carefully design their parallel applications that can run efficiently on HNOCs.

The rest of the manual is organized as follows. Section 2 describes HeteroMPI. Section 3 presents the HeteroMPI API, which are extensions to MPI. Section 4 presents the library of data partitioning functions. Section 5 provides the HeteroMPI command-line user's interface. This is followed by installation instructions for HeteroMPI on UNIX platforms in section 6.

2 What is HeteroMPI

Heterogeneous MPI (**HeteroMPI**) is an extension of MPI for programming high-performance computations on heterogeneous networks of computers. It allows the application programmer to describe the performance model of the implemented algorithm in a generic form. This model allows for all the main features of the underlying parallel algorithm, which have an impact on its execution performance, such as the total number of parallel processes, the total volume of computations to be performed by each process, the total volume of data to be transferred between each pair of the processes, and how exactly the processes interact during the execution of the algorithm. Given the description of the performance model, HeteroMPI tries to create a group of processes that executes the algorithm faster than any other group of processes.

HeteroMPI provides all the features to the user to write portable and efficient parallel applications on HNOCs. These features automate all the essential steps involved in application development on HNOCs:

1). Determination of the characterization parameters relevant to the computational requirements of the applications and the machine capabilities of the heterogeneous system. The machine capabilities are determined before the application execution and supplied to the model of executing network of computers. The model of the executing network of computers is implementation-dependent. We use a static structure automatically obtained by HeteroMPI environment and saved in the form of an ASCII file. However, the parameters of the model can be updated at runtime taking into account the changing network loads.

2). Decomposition of the whole problem into a set of sub-problems that can be solved in parallel by interacting processes. This step of heterogeneous decomposition is parameterized by the number and speeds of processors and the speeds and bandwidths of the communication links between them. The Heterogeneous Data Partitioning Interface (HeteroDPI) is developed to automate this step of heterogeneous decomposition. HeteroDPI provides API that allows the application programmers to specify simple and basic partitioning criteria in the form of parameters and functions to partition the mathematical objects used in their parallel applications.

3). Selection of the optimal set of processes running on different computers of the heterogeneous network by taking into account the speeds of the processors, and the speeds and the bandwidths of the communications links between them. During the creation of this set of optimal processes, HeteroMPI runtime system solves the problem of selection of the optimal set of processes running on different computers of the heterogeneous network using an advanced mapping algorithm. The mapping algorithm is based on the performance model of the parallel algorithm in the form of the set of functions generated by the compiler from the description of the performance model, and the performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm.

4). Application program execution on the HNOCs. The command line user interface of HeteroMPI developed consists of a number of shell commands supporting the creation of a virtual parallel machine and the execution of the HeteroMPI application programs on the virtual parallel machine. The notion of virtual parallel machine enables a collection of heterogeneous computers to be used as single large parallel computer.

3 HeteroMPI's Library Interface

In this section, we describe the interfaces to the routines provided by HeteroMPI as extensions to MPI and the interfaces to the routines in the heterogeneous data partitioning interface (HPDI).

3.1 HeteroMPI runtime initialization and finalization

`HMPI_Init`

Initializes HeteroMPI runtime system

Synopsis:

```
int
HMPI_Init
(
    int* argc,
```

```
    char*** argv
)
```

Parameters:

`argc` --- Number of arguments supplied to **`main`**
`argv` --- Values of arguments supplied to **`main`**

Description: All processes must call this routine to initialize HeteroMPI runtime system. This routine must be called before any other HeteroMPI routine. It must be called at most once; subsequent calls are erroneous.

Usage:

```
int main(int argc, char** argv)
{
   int rc =  HMPI_Init(
                &argc,
                &argv
   );

   if (rc != HMPI_SUCCESS)
   {
      //Error has occurred
   }
}
```

Return values: **`HMPI_SUCCESS`** on success and an error in case of failure.

`HMPI_Finalize`

Finalizes HeteroMPI runtime system

Synopsis:

```
int
HMPI_Finalize
(
    int exitcode
)
```

Parameters:

`exitcode` --- code to be returned to the command shell

Description: This routine cleans up all HeteroMPI state. All processes must call this routine at the end of processing tasks. Once this routine is called, no HeteroMPI routine (even **`HMPI_Init`**) may be called.

Usage:

```
int main(int argc, char** argv)
{
    int rc =  HMPI_Init(
                  &argc,
                  &argv
    );

    if (rc != HMPI_SUCCESS)
    {
       //Error has occurred
    }

    rc =  HMPI_Finalize(0);

    if (rc != HMPI_SUCCESS)
    {
       //Error has occurred
    }
}
```

Return values: **HMPI_SUCCESS** on success and an error in case of failure.

3.2 HeteroMPI Group Management Functions

HMPI_Group_rank

Returns rank of the calling process

Synopsis:

```
int
HMPI_Group_rank
(
    const HMPI_Group* gid
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes

Description: This routine returns the rank of the process calling it. Only processes that are members of the group represented by the handle **gid** can call this routine.

Usage:

```
// HMPI_HOST_GROUP is a predefined group handle
// containing the host process.
HMPI_Group* gid = HMPI_HOST_GROUP;

if (HMPI_Is_member(gid))
{
   int rank =  HMPI_Group_rank(
                   gid
   );
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group represented by the handle **gid**.

HMPI_Rank

Returns rank of the process with the coordinates specified

Synopsis:

```
int
HMPI_Rank
(
    const HMPI_Group* gid,
    const int* coordinates
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes
coordinates --- coordinates representing a process in the group represented by the handle **gid**

Description: This routine returns the rank of the process in the group represented by the handle **gid** and the coordinates of the process being **coordinates**. Only processes that are members of the group represented by the handle **gid** can call this routine. (It must be noted that a topology is explicitly associated with a HeteroMPI group)

Usage:

```
// HeteroMPI target program
HMPI_Group gid;
int coordinates = 3;
```

```
if (HMPI_Is_member(&gid))
{
   int rank = HMPI_Rank(
               &gid,
               &coordinates
   );
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group represented by the handle **gid**.

HMPI_Group_coordof

Returns the coordinates of the process

Synopsis:

```
int
HMPI_Group_coordof
(
    const HMPI_Group* gid,
    int* numc,
    int** coordinates
)
```

Parameters:

gid --- Handle to the HeteroMPI group of processes. This is an input parameter.
numc --- Output parameter giving the number of coordinates representing the calling process in the group represented by the handle **gid**.
coordinates --- The values of the coordinates of the calling process in the group represented by the handle **gid**.

Description: If the process calling this routine is a member of the group given by the handle **gid**, then its coordinates are returned in **coordinates**, the initial element of which points to an integer array containing the coordinates with size **numc**. Only processes that are members of the group represented by the handle **gid** can call this routine.

Usage:

```
HMPI_Group gid;

if (HMPI_Is_member(&gid))
{
   int numc;
```

```
    int* coordinates;

    int rc =  HMPI_Group_coordof(
                  &gid,
                  &numc,
                  &coordinates
    );

    if (rc != HMPI_SUCCESS)
    {
       //Failure
    }

    free(coordinates);
 }
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group represented by the handle **gid**. **HMPI_SUCCESS** is returned on success.

HMPI_Coordof

Returns the coordinates of the process with a specified rank.

Synopsis:

```
int
HMPI_Coordof
(
    const HMPI_Group* gid,
    int rank,
    int* numc,
    int** coordinates
)
```

Parameters:

gid --- Handle to the HeteroMPI group of processes. This is an input parameter.
rank --- The rank of the process whose coordinates are returned.
This is an input parameter.
numc --- Output parameter giving the number of coordinates of the process whose rank is **rank** in the group represented by the handle **gid**.
coordinates --- The values of the coordinates of the process whose rank is **rank** in the group represented by the handle **gid**.

Description: The coordinates of the process whose rank is **rank** in the group represented by the handle **gid** are returned in **coordinates**, the initial element of which points to an integer

array containing the coordinates with size **numc**. Only processes that are members of the group represented by the handle **gid** can call this routine.

Usage:

```
HMPI_Group gid;

if (HMPI_Is_member(gid))
{
   int rank = 0;
   int numc;
   int* coordinates;

   int rc = HMPI_Coordof(
               &gid,
               rank,
               &numc,
               &coordinates
   );

   if (rc != HMPI_SUCCESS)
   {
      //Failure
   }

   free(coordinates);
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group represented by the handle **gid**. **HMPI_SUCCESS** is returned on success.

HMPI_Group_topo_size

Returns the number of coordinates that can specify a process in a group

Synopsis:

```
int
HMPI_Group_topo_size
(
    const HMPI_Group* gid
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes

Description: This routine returns the number of coordinates used to specify a process, which is a member of the group represented by the handle **gid**. Only processes that are members of the group represented by the handle **gid** can call this routine.

Usage:

```
HMPI_Group gid;

if (HMPI_Is_member(&gid))
{
   int numc =  HMPI_Group_topo_size(
                   &gid
   );
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group represented by the handle **gid**.

HMPI_Group_topology

Returns the number of processes in the group in each dimension of the topology of the group.

Synopsis:

```
int
HMPI_Group_topology
(
    const HMPI_Group* gid,
    int* numc,
    int** coordinates
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes.
numc --- Output parameter giving the number of dimensions of the topology specifying the arrangement of the processes, which are members of the group represented by the handle **gid**.
coordinates --- Output parameter giving the number of processes in each dimension of the topology specifying the arrangement of the processes, which are members of the group represented by the handle **gid**.

Description: This routine returns the number of dimensions of the topology and the number of processes in each dimension of the topology representing the arrangement of the processes,

which are members of the group represented by the handle **gid**. The number of processes in each dimension are returned in **coordinates**, the initial element of which points to an integer array with number **numc** of elements containing the number of dimensions. Only processes that are members of the group represented by the handle **gid** can call this routine.

Usage:

```
HMPI_Group gid;

if (HMPI_Is_member(&gid))
{
   int numc;
   int* coordinates;
   int rc =  HMPI_Group_topology(
                &gid,
                &numc,
                &coordinates
   );

   if (rc != HMPI_SUCCESS)
   {
      //Failure
   }

   free(coordinates);
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group represented by the handle **gid**. **HMPI_SUCCESS** is returned on success.

HMPI_Group_parent

Returns the rank of the parent of a group

Synopsis:

```
int
HMPI_Group_parent
(
    const HMPI_Group* gid
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes.

Description: This routine returns the rank of the parent of the group represented by the handle **gid**. Only processes that are members of the group represented by the handle **gid** can call this routine.

Usage:

```
HMPI_Group* gid;
if (HMPI_Is_member(gid))
{
   int prank =  HMPI_Group_parent(gid);
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group represented by the handle **gid**.

HMPI_Group_size

Returns the number of processes in the group

Synopsis:

```
int
HMPI_Group_size
(
    const HMPI_Group* gid
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes

Description: This routine returns the number of processes in the group represented by the handle **gid**. Only processes that are members of the group represented by the handle **gid** can call this routine.

Usage:

```
HMPI_Group* gid;

if (HMPI_Is_member(gid))
{
   int size = HMPI_Group_size(gid);
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group represented by the handle **gid**.

HMPI_Is_host

Is the calling process the host?

Synopsis:

```
unsigned char
HMPI_Is_host()
```

Description: This routine returns **true** if the process calling this function is the host process otherwise **false**. Any process can call this function.

Usage:

```
if (HMPI_Is_host())
{
   printf("I'm the host\n");
}
else
{
   printf("I'm not the host\n");
}
```

Return values: Value of 1 is returned if the process is the member of the group. 0 otherwise.

HMPI_Is_parent

Is the calling process the parent process of the group?

Synopsis:

```
unsigned char
HMPI_Is_parent
(
    const HMPI_Group* gid
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes.

Description: This routine returns **true** if the process calling this routine is the parent of the group represented by the handle **gid** otherwise **false**. Only processes that are members of the group represented by the handle **gid** can call this routine.

Usage:

```
HMPI_Group* gid;

if (HMPI_Is_parent(gid))
{
   printf("I'm the parent of the group gid\n");
}
else
{
   printf("I'm not the parent of the group gid\n");
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group represented by the handle **gid**.

HMPI_Is_member

Am I a member of the group?

Synopsis:

```
unsigned char
HMPI_Is_member
(
    const HMPI_Group* gid
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes.

Description: This function returns **true** if the process calling this routine is the member of the group represented by the handle **gid** otherwise **false**. Only processes that are members of the group represented by the handle **gid** can call this routine.

Usage:

```
HMPI_Group* gid;

if (HMPI_Is_member(gid))
{
   printf("I'm a member of the group gid\n");
}
else
```

```
{
   printf("I'm not a member of the group gid\n");
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group represented by the handle **gid**.

HMPI_Is_free

Am I a member of the predefined group **HMPI_FREE_GROUP**?

Synopsis:

```
unsigned char
HMPI_Is_free()
```

Description: This routine returns **true** if the process is free and is member of the predefined group **HMPI_FREE_GROUP** and **false** otherwise. Any process can call this function.

Usage:

```
if (HMPI_Is_free())
{
   printf("I'm a free process and member of"
          " HMPI_FREE_GROUP \n");
}
else
{
   printf("I'm not a free process and not a member of"
          " HMPI_FREE_GROUP \n");
}
```

Return values: Value of 1 is returned if the process is not the member of any other group other than **HMPI_FREE_GROUP**. 0 otherwise.

HMPI_Get_comm

Returns an MPI communicator with communication group of MPI processes

Synopsis:

```
const MPI_Comm*
HMPI_Get_comm
(
    const HMPI_Group* gid
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes.

Description: This routine returns an MPI communicator with communication group of MPI processes defined by **gid**. This is a local operation not requiring inter-process communication. Application programmers can use this communicator to call the standard MPI communication routines during the execution of the parallel algorithm. This communicator can safely be used in other MPI routines.

Usage:

```
HMPI_Group* gid;
MPI_Comm* comm;

if (HMPI_Is_member(gid))
{
   comm = HMPI_Get_comm(gid);
   if (comm == NULL)
   {
      //error
   }
}
```

Return values: This call returns **NULL** if the process is not a member of the group represented by the handle **gid**.

HMPI_Group_create

Create a HeteroMPI group of processes

Synopsis:

```
int
HMPI_Group_create
(
    HMPI_Group* gid,
    const HMPI_Model* model,
    const int* model_parameters,
    int paramc
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes. This is an output parameter.
model --- handle that encapsulates all the features of the performance model in the

form of a set of functions generated by the compiler from the description of the performance model (input parameter)

`model_parameters` --- parameters of the performance model (input parameter)

`paramc` --- number of parameters of the performance model (input parameter)

Description: This routine tries to create a group that would execute the heterogeneous algorithm faster than any other group of processes. In HeteroMPI, groups are not absolutely independent on each other. Every newly created group has exactly one process shared with already existing groups. That process is called a *parent* of this newly created group, and is the connecting link, through which results of computations are passed if the group ceases to exist. **`HMPI_Group_create`** is a collective operation and must be called by the parent and all the processes, which are not members of any HeteroMPI group.

Usage:

```
HMPI_Group gid1, gid2, gid3;

int modelp[1] = {5};
unsigned char is_parent_of_nid2 = 0;
unsigned char is_parent_of_nid3 = 0;

// The parent used in the creation of abstract network
// gid1 is the host
if (HMPI_Is_member(HMPI_HOST_GROUP))
{
   HMPI_Group_create(
       &gid1,
       &HMPI_Model_simple,
       modelp,
       1
   );
}

if (HMPI_Is_free())
{
   HMPI_Group_create(
       &gid1,
       &HMPI_Model_simple,
       NULL,
       0
   );
}

// The parent used in the creation of group gid2 is the
// member of group gid1 whose coordinates are given
// {2}
if (HMPI_Is_member(&gid1))
```

```
{
   int numc;
   int** coordinates = (int**)malloc(
                               sizeof(int*)
   );
   int rc = HMPI_Group_coordof(
                &gid1,
                &numc,
                coordinates,
   );
   if ((*coordinates)[0] == 2)
   {
      is_parent_of_nid2 = 1;
   }

   free(coordinates[0]);
   free(coordinates);
}

if (is_parent_of_nid2
    || HMPI_Is_free()
)
{
   HMPI_Group_create(
       &nid2,
       &HMPI_Model_simple,
       modelp,
       1
   );
}

// The parent used in the creation of the group gid3 is
// the member of abstract network nid2 whose
// coordinates are given by {3}
if (HMPI_Is_member(&nid2))
{
   int numc;
   int** coordinates = (int**)malloc(
                               sizeof(int*)
   );

   int result =  HMPI_Group_coordof(
                     &gid2,
                     &numc,
                     coordinates,
   );
```

```
    if ((*coordinates)[0] == 3)
    {
       is_parent_of_gid3 = 1;
    }

    free(coordinates[0]);
    free(coordinates);
 }

 if (is_parent_of_nid3
     || HMPI_Is_free()
 )
 {
    HMPI_Group_create(
        &gid3,
        &HMPI_Model_simple,
        modelp,
        1
    );
 }
```

Return values: **HMPI_SUCCESS** on success and an error in case of failure.

HMPI_Group_auto_create

Create a HeteroMPI group of processes with optimal number of processes

Synopsis:

```
int
HMPI_Group_auto_create
(
    HMPI_Group* gid,
    const HMPI_Model* model,
    const int* model_parameters,
    int paramc
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes. This is an output parameter.
model --- handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model (input parameter)
model_parameters --- parameters of the performance model (input parameter)
paramc --- number of parameters of the performance model (input parameter)

Description: This routine allows application programmers not to bother about finding the optimal number of processes that can execute the parallel application. They can specify only the rest of the parameters thus leaving the detection of the optimal number of processes to the HeteroMPI runtime system. **`HMPI_Group_auto_create`** is a collective operation and must be called by the parent and all the processes, which are not members of any HeteroMPI group.

The parameters **`model_parameters`** and **`param_count`** are input parameters. User fills only the input-specific part of the parameter **`model_parameters`** and ignores the return parameters specifying the number of processes to be involved in executing the algorithm and their performances.

During the execution of this function, process arrangements are evaluated serially, that is, one after the other. The parent of the group evaluates the process arrangements.

Return values: **`HMPI_SUCCESS`** on success and an error in case of failure.

`HMPI_Group_pauto_create`

Create a HeteroMPI group of processes with optimal number of processes

Synopsis:

```
int
HMPI_Group_pauto_create
(
    HMPI_Group* gid,
    const HMPI_Model* model,
    const int* model_parameters,
    int paramc
)
```

Parameters:

`gid` --- handle to the HeteroMPI group of processes. This is an output parameter.
`model` --- handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model (input parameter)
`model_parameters` --- parameters of the performance model (input parameter)
`paramc` --- number of parameters of the performance model (input parameter)

Description: This routine allows application programmers not to bother about finding the optimal number of processes that can execute the parallel application. They can specify only the rest of the parameters thus leaving the detection of the optimal number of processes to the HeteroMPI runtime system. **`HMPI_Group_pauto_create`** is a collective operation and must be called by the parent and all the processes, which are not members of any HeteroMPI group.

The parameters **model_parameters** and **param_count** are input parameters. User fills only the input-specific part of the parameter **model_parameters** and ignores the return parameters specifying the number of processes to be involved in executing the algorithm and their performances.

During the execution of this function, process arrangements are evaluated parallely. The process arrangements are evaluated by all the processes available for computation. The process arrangements are divided amongst the processes based on the speeds of the processors. This function call may be more efficient than the function call, ***HMPI_Group_auto_create****.*

Return values: **HMPI_SUCCESS** on success and an error in case of failure.

HMPI_Group_heuristic_auto_create

Uses user-supplied heuristics to create a HeteroMPI group of processes with optimal number of processes

Synopsis:

```
typedef int (*HMPI_Heuristic_function)(
        int np, int *dp, const int* modelp, int paramc);
int
HMPI_Group_heuristic_auto_create
(
    HMPI_Group* gid,
    const HMPI_Model* model,
    HMPI_Heuristic_function hfunc,
    const int* model_parameters,
    int paramc
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes. This is an output parameter.
model --- handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model (input parameter)
hfunc --- User-supplied heuristic function (input parameter)
model_parameters --- parameters of the performance model (input parameter)
paramc --- number of parameters of the performance model (input parameter)

Description: This routine has the same functionality as **HMPI_Group_auto_create** except that it allows application programmers to supply heuristics that minimize the number of process arrangements evaluated.

Application programmers provide the heuristic function **hfunc**. The input parameter **np** is the number of dimensions in the process arrangement. The input parameter **dp** is an integer array of

size **np** containing the number of processes in each dimension of the process arrangement. The input parameters **modelp** and **paramc** are the parameters supplied to the performance model. The function **HMPI_Group_heuristic_auto_create** evaluates a process arrangement only if the heuristic function **hfunc** returns **true**.

During the execution of this function, process arrangements are evaluated serially, that is, one after the other. The parent of the group evaluates the process arrangements.

A simple heuristic function is shown below, which returns a value **true** only if the process arrangement is a square grid.

```
int Square_grid_only(
    int np, int *dp, void *modelp, int paramc){
    if ((np == 2) && (dp[0] == dp[1]))
       return true;
    return false;
}
```

The function evaluates process arrangements that are square grids only if this heuristic function is provided as an input.

Return values: **HMPI_SUCCESS** on success and an error in case of failure.

HMPI_Group_heuristic_pauto_create

Uses user-supplied heuristics to create a HeteroMPI group of processes with optimal number of processes

Synopsis:

```
typedef int (*HMPI_Heuristic_function)(
        int np, int *dp, const int* modelp, int paramc);
int
HMPI_Group_heuristic_pauto_create
(
    HMPI_Group* gid,
    const HMPI_Model* model,
    HMPI_Heuristic_function hfunc,
    const int* model_parameters,
    int paramc
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes. This is an output parameter.
model --- handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description

of the performance model (input parameter)
hfunc --- User-supplied heuristic function (input parameter)
model_parameters --- parameters of the performance model (input parameter)
paramc --- number of parameters of the performance model (input parameter)

Description: This routine has the same functionality as **HMPI_Group_pauto_create** except that it allows application programmers to supply heuristics that minimize the number of process arrangements evaluated.

Application programmers provide the heuristic function **hfunc**. The input parameter **np** is the number of dimensions in the process arrangement. The input parameter **dp** is an integer array of size **np** containing the number of processes in each dimension of the process arrangement. The input parameters **modelp** and **paramc** are the parameters supplied to the performance model. The function **HMPI_Group_heuristic_pauto_create** evaluates a process arrangement only if the heuristic function **hfunc** returns **true**.

During the execution of this function, process arrangements are evaluated parallely. The process arrangements are evaluated by all the processes available for computation. The process arrangements are divided amongst the processes based on the speeds of the processors. This function call may be more efficient than the function call, ***HMPI_Group_heuristic_auto_create****.*

A simple heuristic function is shown below, which returns a value **true** only if the process arrangement is a square grid.

```
int Square_grid_only(
    int np, int *dp, void *modelp, int paramc){
    if ((np == 2) && (dp[0] == dp[1]))
       return true;
    return false;
}
```

The function evaluates process arrangements that are square grids only if this heuristic function is provided as an input.

Return values: **HMPI_SUCCESS** on success and an error in case of failure.

HMPI_Group_pauto_d_create

Create a HeteroMPI group of processes with optimal number of processes

Synopsis:

```
int
HMPI_Group_pauto_d_create
(
    HMPI_Group* gid,
```

```
    const HMPI_Model* model,
    const int* model_parameters,
    int paramc
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes. This is an output parameter.
model --- handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model (input parameter)
model_parameters --- parameters of the performance model (input parameter)
paramc --- number of parameters of the performance model (input parameter)

Description: This routine allows application programmers not to bother about finding the optimal number of processes that can execute the parallel application. They can specify only the rest of the parameters thus leaving the detection of the optimal number of processes to the HeteroMPI runtime system. **HMPI_Group_pauto_d_create** is a collective operation and must be called by the parent and all the processes, which are not members of any HeteroMPI group.

The parameters **model_parameters** and **param_count** are input parameters. User fills only the input-specific part of the parameter **model_parameters** and ignores the return parameters specifying the number of processes to be involved in executing the algorithm and their performances.

This function has additional constraints:
1). It is valid only for 1D and 2D process arrangements
2). It starts with the total number of processes in the communication universe. It obtains all the possible two dimensional process arrangements for this total number of processes. It evaluates these process arrangements parallely. The process arrangements are evaluated by all the processes available for computation. The process arrangements are divided amongst the processes based on the speeds of the processors. It determines the best process arrangement using the estimated execution times.
Now it decrements the total number of processes by one. It obtains again all the possible two dimensional process arrangements. If the estimated execution times are more than the optimal estimated execution time calculated before, the algorithm stops.

Return values: **HMPI_SUCCESS** on success and an error in case of failure.

HMPI_Group_heuristic_pauto_d_create

Uses user-supplied heuristics to create a HeteroMPI group of processes with optimal number of processes

Synopsis:

```
typedef int (*HMPI_Heuristic_function)(
        int np, int *dp, const int* modelp, int paramc);
int
HMPI_Group_heuristic_pauto_d_create
(
    HMPI_Group* gid,
    const HMPI_Model* model,
    HMPI_Heuristic_function hfunc,
    const int* model_parameters,
    int paramc
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes. This is an output parameter.
model --- handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model (input parameter)
hfunc --- User-supplied heuristic function (input parameter)
model_parameters --- parameters of the performance model (input parameter)
paramc --- number of parameters of the performance model (input parameter)

Description: This routine has the same functionality as **HMPI_Group_pauto_d_create** except that it allows application programmers to supply heuristics that minimize the number of process arrangements evaluated.

Application programmers provide the heuristic function **hfunc**. The input parameter **np** is the number of dimensions in the process arrangement. The input parameter **dp** is an integer array of size **np** containing the number of processes in each dimension of the process arrangement. The input parameters **modelp** and **paramc** are the parameters supplied to the performance model. The function **HMPI_Group_heuristic_pauto_d_create** evaluates a process arrangement only if the heuristic function **hfunc** returns **true**.

This function has additional constraints:
1). It is valid only for 1D and 2D process arrangements
2). It starts with the total number of processes in the communication universe. It obtains all the possible two dimensional process arrangements for this total number of processes. It evaluates these process arrangements parallely. The process arrangements are evaluated by all the processes available for computation. The process arrangements are divided amongst the processes based on the speeds of the processors. It determines the best process arrangement using the estimated execution times.
Now it decrements the total number of processes by one. It obtains again all the possible two dimensional process arrangements. If the estimated execution times are more than the optimal estimated execution time calculated before, the algorithm stops.

A simple heuristic function is shown below, which returns a value **true** only if the process arrangement is a square grid.

```
int Square_grid_only(
    int np, int *dp, void *modelp, int paramc){
    if ((np == 2) && (dp[0] == dp[1]))
       return true;
    return false;
}
```

The function evaluates process arrangements that are square grids only if this heuristic function is provided as an input.

Return values: **HMPI_SUCCESS** on success and an error in case of failure.

HMPI_Group_free

Free a HeteroMPI group of processes

Synopsis:

```
int
HMPI_Group_free
(
    const HMPI_Group* gid
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes

Description: This routine deallocates the resources associated with a group object **gid**. **HMPI_Group_free** is a collective operation and must be called by all the processes, which are members of the communication universe HMPI_COMM_WORLD.

Usage:

```
HMPI_Group gid;

// Create a HeteroMPI group of processes

if (HMPI_Is_member(HMPI_COMM_WORLD_GROUP))
{
   HMPI_Group_free(&gid);
}
```

Return values: **HMPI_SUCCESS** on success and an error in case of failure.

3.3 HeteroMPI Runtime updation Functions

HMPI_Recon

Updates the estimation of processor performances dynamically

Synopsis:

```
typedef void (*HMPI_Benchmark_function)(
        const void*, int, void*);

int
HMPI_Recon
(
    HMPI_Benchmark_function func,
    const void* input_p,
    int num_of_parameters,
    void* output_p
)
```

Parameters:

func --- Benchmark user function executed by all the physical processors.
input_p --- Input parameters to the user function.
num_of_parameters --- Number of input parameters to the user function.
output_p --- Return parameter on the execution of the user function.

Description: All the processors execute the benchmark function **func** in parallel, and the time elapsed by each of the processors to execute the code is used to refresh the estimation of its speed. This is a collective operation and must be called by all the processes available for computation.

This routine allows updating the estimation of processor performances dynamically, at runtime, just before using the estimation by the programming system. It is especially important if computers, executing the HeteroMPI program, are used for other computations as well. In that case, the real performance of processors can dynamically change dependent on the external computations. The use of this routine allows writing parallel programs sensitive to such dynamic variation of the workload of the underlying computer system.

Usage:

```
double Perf_func (
        double l, double w, double h, double delta)
```

```
{
   double m,x,y,z;
   for (m = 0.0, x = 0.0; x < l; x += delta)
       for (y = 0.; y < w; y += delta)
           for (z = 0.; z < h; z += delta)
               m += XYZ_func(x,y,z);
   return m * delta * delta * delta;
}

void Benchmark_function
(
   const void* input_p,
   int num_of_p,
   void* output_p
)
{
   double* params = (double*)input_p;
   double result =   Perf_func(
                         params[0],
                         params[1],
                         params[2],
                         params[3]
   );
   *(double*)(output_p) = result;
   return;
}

// All processes available for computation must call
// this function
{
   double output_p;
   int rc = HMPI_Recon(
                Benchmark_function,
                input_p,
                4,
                &output_p
   );

   if (rc != HMPI_SUCCESS)
   {
      //An error has occurred
   }
}
```

Return values: **HMPI_SUCCESS** on success and an error in case of failure.

HMPI_Recon_t

Updates the estimation of processor performances dynamically

Synopsis:

```
typedef double (*HMPI_Benchmark_function_t)(
        const void*, int, void*);

int
HMPI_Recon_t
(
    HMPI_Benchmark_function func,
    const void* input_p,
    int num_of_parameters,
    void* output_p
)
```

Parameters:

func --- Benchmark user function executed by all the physical processors.
input_p --- Input parameters to the user function.
num_of_parameters --- Number of input parameters to the user function.
output_p --- Return parameter on the execution of the user function.

Description: All the processors execute the benchmark function **func** in parallel, and the time elapsed by each of the processors to execute the code is used to refresh the estimation of its speed. This is a collective operation and must be called by all the processes available for computation.

This routine allows updating the estimation of processor performances dynamically, at runtime, just before using the estimation by the programming system. It is especially important if computers, executing the HeteroMPI program, are used for other computations as well. In that case, the real performance of processors can dynamically change dependent on the external computations. The use of this routine allows writing parallel programs sensitive to such dynamic variation of the workload of the underlying computer system.

Usage:

```
double Perf_func (
        double l, double w, double h, double delta)
{
   double m,x,y,z;
   for (m = 0.0, x = 0.0; x < l; x += delta)
       for (y = 0.; y < w; y += delta)
           for (z = 0.; z < h; z += delta)
```

```
                m += XYZ_func(x,y,z);
    return m * delta * delta * delta;
}

double Benchmark_function
(
    const void* input_p,
    int num_of_p,
    void* output_p
)
{
    double* params = (double*)input_p;
    // Time the benchmark function
    double start = MPI_Wtime();
    double result =   Perf_func(
                          params[0],
                          params[1],
                          params[2],
                          params[3]
    );
    double end = MPI_Wtime();
    *(double*)(output_p) = result;
    return (end-start);
}

// All processes available for computation must call
// this function
{
    double output_p;
    int rc = HMPI_Recon_t(
                Benchmark_function,
                input_p,
                4,
                &output_p
    );

    if (rc != HMPI_SUCCESS)
    {
       //An error has occurred
    }
}
```

Return values: **HMPI_SUCCESS** on success and an error in case of failure.

3.4 HeteroMPI Estimation Functions

HMPI_Timeof

Predict the total time of execution of the algorithm on the underlying hardware without its real execution

Synopsis:

```
double
HMPI_Timeof
(
    const HMPI_Model* model,
    const int* model_parameters,
    int paramc
)
```

Parameters:

model--- handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model (input parameter)
model_parameters --- parameters of the performance model (input parameter)
paramc --- number of parameters of the performance model (input parameter)

Description: This routine allows application programmers to predict the total time of execution of the algorithm on the underlying hardware without its real execution. This function allows the application programmers to write such a parallel application that can follow different parallel algorithms to solve the same problem, making choice at runtime depending on the particular executing network and its actual performance. This is a local operation that can be called by any process, which is a member of the group associated with the predefined communication universe **HMPI_COMM_WORLD** of HeteroMPI.

HMPI_Timeof can thus be used to estimate the execution time on HNOCs for each possible set of model parameters **model_parameters**. Application programmers can use this function creatively to design best possible heuristics for the set of parameters. Depending on the time estimated for each set, the optimal values of the parameters are determined. These values are then passed to the performance model during the actual creation of the group of processes using the function **HMPI_Group_create**.

Usage:

```
algorithm bcast(int p, int n, int ITER, int rooot) {
   coord I=p;
   node {
      I>=0: bench*1;
   };
   link {
```

```
      I>=0&&I!=rooot: length*(n*n*ITER*sizeof(double))
                      [rooot]->[I];
   };
   parent[0];
   scheme {
      int i, k;
      for (k = 0; k < ITER; k++)
          for (i = 0; i < p; i++)
             if (i != rooot)
                (100/ITER)%%[rooot]->[i];
   };
};

int main() {
    int p;
    HMPI_Group gid;
    …
    p = HMPI_Group_size(HMPI_COMM_WORLD_GROUP);
    if (HMPI_Is_host()) {
       int param_count = 4;
       int model_params[4] = {
           p,
           N,
           ITER,
           root
       };
       double time;
       time = HMPI_Timeof(
                  &HMPI_Model_bcast,
                  &model_params,
                  param_count
       );
       time = (double)time/(double)ITER;
       printf("Number of bytes broadcast = %d,
               time=%0.9f\n", N*N*8, time);
    }
}
```

Return values: -1.0 in case of failure.

HMPI_Group_timeof

Returns the predicted total time of execution of the algorithm on the underlying hardware

Synopsis:

```
double
HMPI_Group_timeof
```

```
(
    const HMPI_Group* gid
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes

Description: This routine returns the predicted total time of execution of the algorithm on the underlying hardware. This function is a collective operation that must be called by all the members of the group.

Return values: -1.0 in case of failure.

3.5 HeteroMPI Processor Information Functions

HMPI_Get_number_of_processors

Returns the number of physical processors of the underlying distributed memory machine

Synopsis:

```
int
HMPI_Get_number_of_processors()
```

Description: This routine returns the number of physical processors of the underlying distributed memory machine. This is a collective operation and must be called by all the processes in the group associated with the predefined communication universe **HMPI_COMM_WORLD** of HeteroMPI.

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group **HMPI_COMM_WORLD_GROUP**. **HMPI_SUCCESS** is returned on success.

HMPI_Get_processors_info

Returns the relative performances of the physical processors of the underlying distributed memory machine

Synopsis:

```
int
HMPI_Get_processors_info
(
    double* relative_performances
)
```

Parameters:

Relative_performances --- Output parameter containing the relative

performances of the physical processors of the underlying distributed memory machine

Description: This routine returns the relative performances of the physical processors of the underlying distributed memory machine. This is a collective operation and must be called by all the processes in the group associated with the predefined communication universe **HMPI_COMM_WORLD** of HeteroMPI.

Usage:

```
int p = HMPI_Get_number_of_processors();
double speeds = (double*)malloc(
                          sizeof(double)
                          *
                          p
);

int rc = HMPI_Get_processors_info(
             speeds
);

if (rc != HMPI_SUCCESS)
{
   //An error has occurred
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group **HMPI_COMM_WORLD_GROUP**. **HMPI_SUCCESS** is returned on success.

HMPI_Get_processes_info

Returns the relative performances of the processes running on the physical processors of the underlying distributed memory machine

Synopsis:

```
int
HMPI_Get_processes_info
(
    double* relative_performances
)
```

Parameters:

Relative_performances --- Output parameter containing the relative performances of the processes running on the physical processors of the underlying distributed

memory machine

Description: This routine returns the relative performances of the processes running on the physical processors of the underlying distributed memory machine. This is a collective operation and must be called by all the processes in the group associated with the predefined communication universe **HMPI_COMM_WORLD** of HeteroMPI.

Usage:

```
int p = HMPI_Group_size(HMPI_COMM_WORLD_GROUP);
double speeds = (double*)malloc(
                          sizeof(double)
                          *
                          p
);

int rc = HMPI_Get_processes_info(
              speeds
);

if (rc != HMPI_SUCCESS)
{
   //An error has occurred
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group **HMPI_COMM_WORLD_GROUP**. **HMPI_SUCCESS** is returned on success.

HMPI_Group_performances

Returns the relative performances of the processes in a group

Synopsis:

```
int
HMPI_Group_performances
(
    const HMPI_Group* gid,
    double* relative_performances
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes
Relative_performances --- Output parameter containing the relative performances of the processes in the group represented by the handle **gid**

Description: This routine returns the relative performances of the processes in the group represented by the handle **gid**. This is a collective operation and must be called by all the processes in the group given by the handle **gid**.

Usage:

```
HMPI_Group gid;

if (HMPI_Is_member(gid))
{
   int p = HMPI_Group_size(&gid);
   double speeds = (double*)malloc(
                          sizeof(double)
                          *
                          p
   );

   int rc = HMPI_Group_performances(
                gid,
                speeds
   );

   if (rc != HMPI_SUCCESS)
   {
      //An error has occurred
   }
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group given by the handle **gid**. **HMPI_SUCCESS** is returned on success.

3.6 HeteroMPI Synchronization Functions

HMPI_Barrier

Barrier for the members of the group

Synopsis:

```
int HMPI_Barrier
(
    const HMPI_Group* gid
)
```

Parameters:

gid --- handle to the HeteroMPI group of processes

Description: Has same functionality as **MPI_Barrier**. This is a collective operation and must be called by all the processes in the group given by the handle **gid**.

Usage:

```
HMPI_Group gid;

if (HMPI_Is_member(&gid))
{
   HMPI_Barrier(&gid);
}
```

Return values: **HMPI_SUCCESS** on success and an error in case of failure.

3.7 HeteroMPI Debugging and Version Functions

HMPI_Printf

Print formatted strings to the host processor.

Synopsis:

```
int HMPI_Printf
(
    const char* format,
    ...
)
```

Parameters:

format --- Format string in printf-fashion.

Description: Prints formatted strings to standard output on the virtual host processor from any virtual processor of the computing space. Any process can call this function.

Usage:

```
HMPI_Group gid;

if (HMPI_Is_member(&gid))
{
   HMPI_Printf(
      "Hello, My node rank is %d, My Globalrank “
      “is %d\n ",
      HMPI_Group_rank(&nid),
```

```
        HMPI_Group_rank(HMPI_COMM_WORLD_GROUP)
    );
}
```

Return values: **HMPI_SUCCESS** on success and an error in case of failure.

HMPI_Strerror

Return a string associated with error code.

Synopsis:

```
int
HMPI_Strerror
(
    int errnum,
    char* message
)
```

Parameters:

errnum --- Error code from any HeteroMPI routine call.
message --- Output parameter. Error message associated with the error code. The message must represent storage that is at least **HMPI_MAX_ERROR_STRING** characters long.

Description: An error message string corresponding to the error number **errnum** is returned in **message**. Any process can call this function.

Usage:

```
char message[HMPI_MAX_ERROR_STRING];

int rc = HMPI_Init(
            &argc,
            &argv
);

if (rc != HMPI_SUCCESS)
{
   HMPI_Strerror(
       rc,
       message
   );

   HMPI_Printf(
    "Error during HETEROMPI initialization. Reason is %s\n",
```

```
        message
      );
}
```

Return values: **`HMPI_SUCCESS`** on success and error on failure.

`HMPI_Debug`

Turn the diagnostics on/off.

Synopsis:

```
int
HMPI_Debug
(
    int yesno
)
```

Parameters:

`yesno` --- yes (1) or no (0)

Description: Produces detailed diagnostics. Any process can call this function. This is the only function apart from **`HMPI_Get_version`** that can be called before **`HMPI_Init`** or after **`HMPI_Finalize`**.

`HMPI_Get_version`

Returns the version of the HeteroMPI API in the format x.y

Synopsis:

```
int
HMPI_Get_version
(
    int *version,
    int *sub_version
)
```

Parameters:

`version` --- Major version
`sub_version` --- Minor version

Description: Returns the version of HeteroMPI. Any process can call this function. This is one of the few functions that can be called before **`HMPI_Init`** or after **`HMPI_Finalize`**.

Usage:

```
int version, sub_version;
HMPI_Get_version(&version, &sub_version);
```

4 Heterogeneous Data Partitioning Interface (HeteroDPI)

The core of scientific, engineering or business applications is the processing of some mathematical objects that are used in modeling corresponding real-life problems. In particular, partitioning of such mathematical objects is a core of any data parallel algorithm. Our analysis of various scientific, engineering and business domains resulted in the following short list of mathematical objects commonly used in parallel and distributed algorithms: **sets** (ordered and non-ordered), **dense matrices** (and multidimensional arrangements), **graphs**, and **trees**.

Based on this classification, we suggest an API for partitioning mathematical objects commonly used in scientific and engineering domains for solving problems on networks of heterogeneous computers. These interfaces allow the application programmers to specify simple and basic partitioning criteria in the form of parameters and functions to partition their mathematical objects. These partitioning interfaces are designed to be used along with various programming tools for parallel and distributed computing on heterogeneous networks.

4.1 Sets

`Partition_unordered_set`

Partition a non-ordered set

Synopsis:

```
typedef double (*User_defined_metric)(
      int p, const double *speeds, const int *actual,
      const int *ideal)

int Partition_unordered_set (
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int n,
    const int *w, int type_of_metric,
    User_defined_metric umf, double *metric, int *np)
```

Description: This routine partitions a set into **`p`** disjoint partitions.

Return values: **`0`** on success and **`-1`** in case of failure.

`Partition_ordered_set`

Partition a well-ordered set

Synopsis:

```
int Partition_ordered_set (
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int n,
    const int *w, int processor_reordering,
    int type_of_metric, User_defined_metric umf,
    double *metric, int *np)
```

Description: This routine partitions a well-ordered set into **p** disjoint contiguous partitions.

Parameters:

Parameter **p** is the number of partitions of the set. Parameters **speeds** and **psizes** specify speeds of processors for **pn** different problem sizes. These parameters are 1D arrays of size **p×pn** logically representing 2D arrays of shape **[p][pn]**. The speed of the **i**-th processor for **j**-th problem size is given by the **[i][j]**-th element of **speeds** with the problem size itself given by the **[i][j]**-th element of **psizes**. Parameter **mlimits** gives the maximum number of elements that each processor can hold.

Parameter **n** is the number of elements in the set, and parameter **w** is the weights of its elements.

Parameter **type_of_metric** specifies which metric should be used to determine the quality of the partitioning. If **type_of_metric** is **USER_SPECIFIED**, then the user provides a metric function **umf**, which is used to calculate the quality of the partitioning. If **type_of_metric** is **SYSTEM_DEFINED**, the system-defined metric is used.

The output parameter **metric** gives the quality of the partitioning, which is the deviation of the partitioning achieved from the ideal partitioning satisfying the partitioning criteria. If the output parameter **metric** is set to **NULL**, then the calculation of metric is ignored.

If **w** is not **NULL** and the set is well ordered, then the user needs to specify if the implementations of this operation may reorder the processors before partitioning (Boolean parameter **processor_reordering** is used to do it). One typical reordering is to order the processors in the decreasing order of their speeds.

Return values: **0** on success and **-1** in case of failure.

Get_set_processor

For an ordered set, returns the processor owning the set element at index **i**

Synopsis:

```
int Get_set_processor (
    int i, int n, int p, int processor_reordering,
    const int *np)
```

Return values: **-1** in case of failure.

Get_my_partition

For a set, returns the number of elements allocated to processor **i**

Synopsis:

```
int Get_my_partition (
    int i, int p, const double *speeds, int n)
```

Return values: **-1** in case of failure.

4.2 Dense Matrices

Partition_matrix_2d

Partition a matrix amongst processors arranged in a 2D grid

Synopsis:

```
int Partition_matrix_2d (
   int p, int q,
   int pn, const double *speeds, const int *psizes,
   const int *mlimits, int m, int n,
   int type_of_distribution, int *w, int *h, int *trow,
   int *tcol, int *ci, int *cj )
```

Parameters:

The parameter **p** is the number of processors along the row of the processor grid. The parameter **q** is the number of processors along the column of the processor grid.

Parameters **speeds** and **psizes** specify speeds of processors for **pn** different problem sizes. These parameters are 1D arrays of size **p×q×pn** logically representing arrays of shape **[p][q][pn]**. The speed of the **(i, j)**-th processor for **k**-th problem size is given by the **[i][j][k]**-th element of **speeds** with the problem size itself given by the **[i][j][k]**-th element of **psizes**. Parameter **mlimits** gives the maximum number of elements that each processor can hold.

The parameters **m** and **n** are the sizes of the generalized block along the row and the column.

The input parameter **type_of_distribution** specifies if the distribution is **CARTESIAN, ROW-BASED,** and **COLUMN-BASED**.

Output parameter **w** gives the widths of the rectangles of the generalized block assigned to different processors. This parameter is an array of size **p**×**q**.

Output parameter **h** gives the heights of rectangles of the generalized block assigned to different processors. This parameter is an array of size **p**×**q**×**p**×**q** logically representing array of shape **[p][q][p][q]**.

Output parameter **trow** gives the top leftmost point of the rectangles of the generalized block assigned to different processors from the first row of the generalized block. This parameter is an array of size **p**×**q**.

Output parameter **tcol** gives the top leftmost point of the rectangles of the generalized block assigned to different processors from the first column of the generalized block. This parameter is an array of size **p**×**q**.

Output parameters **ci**, and **cj** are each an array of size **m**×**n**. The coordinates of the processor in its processor grid to which the matrix element at row **i** and column **j** of the generalized block is assigned is given by **ci[i×n+j]**, and **cj[i×n+j]** respectively. If the application programmer sets these parameters to **NULL**, then these parameters are ignored.

Description: This routine partitions a matrix into **p** disjoint partitions amongst processors arranged in a 2D grid.

Return values: **0** on success and **-1** in case of failure.

Partition_matrix_1d_dp

Partition a matrix amongst processors arranged in a linear array

Synopsis:

```
int Partition_matrix_1d_dp(
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int m, int n,
    Get_lower_bound lb, DP_function dpf,
    int type_of_distribution,
    int *w, int *h, int *trow, int *tcol, int *c)
```

Parameters:

The parameter **p** is the number of number of disjoint rectangles the matrix is partitioned into. Parameters **speeds** and **psizes** specify speeds of processors for **pn** different problem sizes. These parameters are 1D arrays of size **p×pn** logically representing 2D arrays of shape **[p][pn]**. The speed of the **i**-th processor for **j**-th problem size is given by the **[i][j]**-th element of **speeds** with the problem size itself given by the **[i][j]**-th element of **psizes**. Parameter **mlimits** gives the maximum number of elements that each processor can hold.

The parameters **m** and **n** are the sizes of the generalized block along the row and the column.

The input parameter **type_of_distribution** specifies if the distribution is **ROW-BASED** or **COLUMN-BASED**.

Output parameter **w** gives the widths of the rectangles of the generalized block assigned to different processors. This parameter is an array of size **p**. Output parameter **h** gives the heights of rectangles of the generalized block assigned to different processors. This parameter is an array of size **p**×**p**. Output parameter **trow** gives the top leftmost point of the rectangles of the generalized block assigned to different processors from the first row of the generalized block. This parameter is an array of size **p**. Output parameter **tcol** gives the top leftmost point of the rectangles of the generalized block assigned to different processors from the first column of the generalized block. This parameter is an array of size **p**.

Output parameter **c** is an array of size **m**×**n**. The coordinates of the processor in its processor array to which the matrix element at row **i** and column **j** of the generalized block is assigned is given by **c[i×n+j]**. If the user sets these parameters to **NULL**, then these parameters are ignored.

Description: This routine partitions a matrix into **p** disjoint partitions amongst processors arranged in a linear array.

Return values: **0** on success and **-1** in case of failure.

Partition_matrix_1d_iterative

Partition a matrix amongst processors arranged in a linear array

Synopsis:

```
int Partition_matrix_1d_iterative(
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int m, int n,
    Get_lower_bound lb, Iterative_function cf,
    int *w, int *h, int *trow, int *tcol, int *c)
```

Parameters:

Application programmers provide a cost function **cf** that tests the optimality of a partition from a finite set of partitions. The initial partition in this finite set of partitions is obtained using a problem-specific strategy. The cost function **cf** is called iteratively for each of the partitions in the subset of partitions. The return value of this function gives an optimality value. At each step of the iteration, the optimality value is compared to the lower bound of the optimal solution to the optimization problem. Application programmers specify a function **lb**, which is used to calculate the lower bound of their optimization problem. The iteration stops when the function returns an optimality value less than or equal to the lower bound or a negative return value indicating that the partitioning cannot be improved and that the current partition is optimal.

Description: Partitions a matrix into **p** disjoint partitions amongst processors arranged in a linear array.

Return values: **0** on success and **-1** in case of failure.

Partition_matrix_1d_refining

Partition a matrix amongst processors arranged in a linear array

Synopsis:

```
int Partition_matrix_1d_refining(
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int m, int n,
    Get_lower_bound lb, Refining_function cf,
    int *w, int *h, int *trow, int *tcol, int *c)
```

Parameters:

Application programmers provide a refinement function **rf** that refines an old partition giving a new better partition. A negative return value of this function suggests that the old partition cannot be refined further. This function is iteratively called. The partition for the first call of this refining function is obtained using a problem-specific strategy. Application programmers specify a function **lb**, which is used to calculate the lower bound of their optimization problem. The iteration stops when the refinement function **rf** returns an optimality value less than or equal to the lower bound indicating that the current partition is optimal.

Description: Partitions a matrix into **p** disjoint partitions amongst processors arranged in a linear array.

Return values: **0** on success and **-1** in case of failure.

Get_matrix_processor

Returns the coordinates (**i**,**j**) of the processor owning the matrix element at row **r** and column **c**

Synopsis:

```
typedef struct {int i; int j;} Processor;
int Get_matrix_processor(
    int r, int c, int p, int q, int *w, int *h, int *trow,
    int *tcol, int type_of_distribution, Processor *root)
```

Return values: **0** on success and **-1** in case of failure.

Get_my_width

Returns the width of the rectangle owned by the processor with coordinates (**i**,**j**)

Synopsis:

```
int Get_my_width(
    int i, int j, int p, int q, const double *speeds,
    int type_of_distribution, int m, int n)
```

Description: Currently only applicable to two-dimensional processor arrangements.

Return values: **-1** in case of failure.

Get_my_height

Returns the height of the rectangle owned by the processor with coordinates (**i**,**j**)

Synopsis:

```
int Get_my_height(
    int i, int j, int p, int q, const double *speeds,
    int type_of_distribution, int m, int n)
```

Description: Currently only applicable to two-dimensional processor arrangements.

Return values: **-1** in case of failure.

Get_diagonal

Obtain the number of elements owned by the processor with coordinates (**i**,**j**) on the diagonal of the matrix

Synopsis:

```
int Get_diagonal(
    int i, int j, int p, int q, int *w, int *h, int *trow,
    int *tcol)
```

Description: Currently only applicable to dense square matrices and two-dimensional processor arrangements.

Return values: **-1** in case of failure.

Get_my_elements

Obtain the number of elements owned by the processor with coordinates (**i**,**j**) in the upper or lower half of the matrix including the diagonal elements

Synopsis:

```
int Get_my_elements(
```

```
    int n, int g, int i, int j, int p, int q, int *w, int *h,
    int *trow, int *tcol, int type_of_distribution,
    char upper_or_lower)
```

Description: Currently only applicable to dense square matrices and two-dimensional processor arrangements.

Return values: **-1** in case of failure.

Get_my_kk_elements

Obtain the number of elements owned by the processor with coordinates (**i**,**j**) in the upper or lower half of the matrix starting from (**k**,**k**) including the diagonal elements

Synopsis:

```
int Get_my_kk_elements(
    int n, int g, int k, int i, int j, int p, int q, int *w,
    int *h, int *trow, int *tcol, int type_of_distribution,
    char upper_or_lower)
```

Description: Currently only applicable to dense square matrices and two-dimensional processor arrangements.

Return values: **-1** in case of failure.

4.3 Graphs

Partition_graph

Partition a graph

Synopsis:

```
int Partition_graph (
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int n, int m,
    const int *vwgt, const int *xadj,
    const int *adjacency, const int *adjwgt,
    int nopts, const int *options, int *vp, int *edgecut)
```

Parameters:

Parameter **p** is the number of partitions of the graph. Parameters **speeds** and **psizes** specify speeds of processors for **pn** different problem sizes. These parameters are 1D arrays of size **p×pn** logically representing 2D arrays of shape **[p][pn]**. The speed of the **i**-th processor for **j**-th problem size is given by the **[i][j]**-th element of **speeds** with the problem size itself

given by the **[i][j]**-th element of **psizes**. Parameter **mlimits** gives the maximum number of elements that each processor can hold.

The parameters **n** and **m** are the number of vertices and edges in the graph. The parameters **vwgt** and **adjwgt** are the weights of vertices and edges of the graph. In the case in which the graph is unweighted (i.e., all vertices and/or edges have the same weight), then either or both of the arrays **vwgt** and **adjwgt** can be set to **NULL**. The parameters **vwgt** is of size **n**. The parameter **adjwgt** is of size **2m** because every edge is listed twice (i.e., as (*v*, *u*) and (*u*, *v*)).

The parameters **xadj** and **adjacency** specify the adjacency structure of the graph represented by the compressed storage format (CSR). The adjacency structure of the graph is stored as follows. The adjacency list of vertex **i** is stored in **adjacency** starting at index **xadj[i]** and ending at but not including **xadj[i+1]**. The adjacency lists for each vertex are stored consecutively in the array **adjacency**.

The parameter **options** is an array of size **nopts** containing the options for the various phases of the partitioning algorithms employed in partitioning the graph. These options allow integration of third party implementations, which provide their own partitioning schemes.

The parameter **vp** is an array of size **n** containing the partitions to which the vertices are assigned. Specifically, **vp[i]** contains the partition number in which vertex **i** belongs to. The parameter **edgecut** contains the number of edges that are cut by the partitioning.

Description: This routine partitions a graph into **p** disjoint partitions.

Return values: **0** on success and **-1** in case of failure.

Partition_bipartite_graph

Partition a bipartite graph

Synopsis:

```
int Partition_bipartite_graph (
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits,
    int n, int m, const int *vtype, const int *vwgt,
    const int *xadj, const int *adjacency,
    const int *adjwgt, int type_of_partitioning,
    int nopts, const int *options, int *vp, int *edgecut)
```

Parameters:

The meaning of the parameters **p**, **pn**, **speeds**, **psizes**, **mlimits**, **n**, **m**, **vwgt**, **adjwgt**, **xadj**, **adjacency** is identical to meaning of the corresponding parameters of **Partition_graph**.

The parameter **vtype** specifies the type of vertex. The only values allowed are 0 and 1 representing the two disjoint subsets the bipartite graph is composed of.

The parameter **type_of_partitioning** specifies whether the partitioning of subsets is done separately or not. It can take only one of the values **PARTITION_SUBSET** and **PARTITION_OTHER**.

The parameter **options** is an array of size **nopts** containing the options for the various phases of the partitioning algorithms employed in partitioning the graph. These options allow integration of third party implementations, which provide their own partitioning schemes.

The parameter **vp** is an array of size of size **n** containing the partitions to which the vertices are assigned. Specifically, **vp[i]** contains the partition number in which vertex **i** belongs to. The parameter **edgecut** contains the number of edges that are cut by the partitioning.

Description: This routine partitions a bipartite graph into **p** disjoint partitions.

Return values: **0** on success and **-1** in case of failure.

Partition_hypergraph

Partition a hypergraph

Synopsis:

```
int Partition_hypergraph (
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits,
    int nv, int nedges, const int *vwgt, const int *hptr,
    const int *hind, const int *hwgt, int *vp,
    int nopts, const int *options, int *edgecut)
```

Parameters:

The meaning of the parameters **p**, **pn**, **speeds**, **psizes**, and **mlimits** is identical to meaning of the corresponding parameters of **Partition_graph**.

The parameters `nv` and `nedges` are the number of vertices and number of hyperedges in the hypergraph.

The parameters `vwgt` is an array of size `nv` that stores the weights of the vertices and `hwgt` is an array of size `nedges` that stores the weights of hyperedges of the graph. If the vertices in the hypergraph are unweighted, then `vwgt` can be `NULL`. If the hyperedges in the hypergraph are unweighted, then `hwgt` can be `NULL`.

The parameter **hptr** is an array of size **nedges**+1 and is an index into **hind** that stores the actual hyperedges. Each hyperedge stores the sequence of the vertices that it spans, in consecutive locations in **hind**. Specifically, **i**-th hyperedge is stored starting at location **hind[hptr[i]]** up to but not including **hind[hptr[i+1]]**.

The parameter **options** is an array of size **nopts** containing the options for the various phases of the partitioning algorithms employed in partitioning the graph. These options allow integration of third party implementations, which provide their own partitioning schemes.

The parameter **vp** is an array of size of size **n** containing the partitions to which the vertices are assigned. Specifically, **vp[i]** contains the partition number in which vertex **i** belongs to. The parameter **edgecut** contains the number of hyperedges that are cut by the partitioning.

Description: This routine partitions a hypergraph into **p** disjoint partitions.

Return values: **0** on success and **-1** in case of failure.

4.4 Trees

Partition_tree

Partition a tree

Synopsis:

```
int Partition_tree (
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits,
    int n, int nedges, const int *nwgt, const int *xadj,
    const int *adjacency, const int *adjwgt,
    int *vp, int *edgecut)
```

Parameters:

The meaning of the parameters **p**, **pn**, **speeds**, **psizes**, and **mlimits** is identical to meaning of the corresponding parameters of **Partition_graph**.

The parameters **n** and **nedges** are the number of vertices and edges in the tree. The parameters **nwgt** is an array of size **n** that stores the weights of the vertices and **adjwgt** is an array of size **nedges** that stores the weights of edges of the tree. If the vertices in the tree are unweighted, then **nwgt** can be **NULL**. If the edges in the tree are unweighted, then **adjwgt** can be **NULL**.

The parameters **xadj** and **adjacency** specify the adjacency structure of the tree.

```
nettype grid(int p, int q) {
   coord I=p, J=q;
};
```

Figure 1: Specification of a simple performance model in the HeteroMPI's performance definition language. The performance model definition is in the file "`grid.mpc`".

The parameter **`vp`** is an array of size of size **`n`** containing the partitions to which the vertices are assigned. Specifically, **`vp[i]`** contains the partition number in which node **`i`** belongs to. The parameter **`edgecut`** contains the number of edges that are cut by the partitioning.

Description: This routine partitions a tree into **`p`** disjoint subtrees.

Return values: **`0`** on success and **`-1`** in case of failure.

5 HeteroMPI Program Execution

5.1 HeteroMPI Environment

Currently, the HeteroMPI programming environment includes a *compiler* and a *library*.

The compiler compiles the description of this performance model to generate a set of functions. The functions make up an algorithm-specific part of the HeteroMPI runtime system.

The library consists of extensions to MPI and Heterogeneous Data Partitioning Interface (HeteroDPI).

5.2 Building HeteroMPI Application

A sample performance model and the HeteroMPI application using the performance model are shown in Figures 1 and 2:

Outlined below are steps to build and run a HeteroMPI application.

1). Compile the performance model file. The generated file is "**`grid.c`**".

```
shell$ mpcc grid.mpc
```

2). Create the executable.

```
shell$ mpicc –I<HMPI Installation directory>/include –o Test_group_create Test_group_create.c -L<HMPI Installation directory>/lib –lhmpi –lmpc –lm
```

```
#include <math.h>
#include <stdio.h>
#include <sys/time.h>
#include <hmpi.h>
#include "grid.c"

int main() {
    int param_count, model_params[2];
    struct timeval start, end;
    gettimeofday(&start, NULL);

    HMPI_Group gid;
    HMPI_Init(&argc, &argv);
    if (HMPI_Is_host()) {
       int gsize, p, q;
       param_count = 2;
       gsize = HMPI_Group_size(HMPI_COMM_WORLD_GROUP);
       p = q = sqrt(gsize);
       if ((p == 0) && (q == 0))
          p = q = 1;
       model_params[0] = p;
       model_params[1] = q;

       printf("Total number of processes available for computation
              is %d\n", gsize);
       printf("Creating a grid (%d, %d) of processes\n", p, q);
    }
    if (HMPI_Is_host())
       HMPI_Group_create (&gid, &MPC_NetType_grid,
                          model_params, param_count)
    else
       HMPI_Group_create (&gid, &MPC_NetType_grid,
                          NULL, 0)
    // Distribute computations using the optimal speeds of processes
    if (HMPI_Is_member(&gid)){
      // computations and communications are performed here
    }
    HMPI_Group_free(&gid);
    gettimeofday(&end, NULL);
    if (HMPI_Is_host()) {
       double tstart = start.tv_sec + (start.tv_usec/pow(10, 6));
       double tend = end.tv_sec + (end.tv_usec/pow(10, 6));
       printf("Time taken for group creation(sec)=%f\n",
              tend-tstart);
    }
    HMPI_Finalize(0);
}
```

Figure 2: A sample HeteroMPI program. The HeteroMPI program is written in the file "`Test_group_create.c`".

More advanced examples can be studied from the "*tests*" directory.

5.3 Running HeteroMPI Application

Run the target program using *mpirun*.

```
<output…>
Total number of processes available for computation is 9
Creating a grid (3, 3) of processes
Time taken for group creation(sec)=0.262353
```

During the HeteroMPI runtime initialization, the structure of the network (performances of the processors, values of the parameters of the communication model) is obtained. This information is used by the runtime mapping algorithms. This initialization process takes time. To avoid this penalty, there are two options that can be provided to the HeteroMPI program, which allow initialization to take place from a file.

To write the structure/state of the executing network to a file, use the option "*-wtopo*". For example, consider the execution of the program *Test_group_create* using OpenMPI on a network consisting of three hosts {hcl01, hcl02, hcl03}. The structure of the network is saved/written to a file "topostruct.txt".

```
shell$ mpirun –np 1 --host hcl01 Test_group_create –wtopo
topostruct.txt : –np 1 --host hcl02 Test_group_create : –np 1 --
host hcl03 Test_group_create
```

To read the structure/state of the executing network from a file, use the option "*-rtopo*". For example, consider the execution of the program *Test_group_create* using OpenMPI on three hosts {hcl01, hcl02, hcl03}. The structure of the network is read from the file "topostruct.txt".

```
shell$ mpirun –np 1 --host hcl01 Test_group_create –rtopo
topostruct.txt : –np 1 --host hcl02 Test_group_create : –np 1 --
host hcl03 Test_group_create
```

The topology files can be created once and used for multiple runs.

6 HeteroMPI Installation Guide for UNIX

This section provides information for programmers and/or system administrators who want to install HeteroMPI for UNIX.

6.1 System Requirements

The following table describes system requirements for HeteroMPI for UNIX.

Component	Requirement
Operating System	Tested on Linux, Solaris, FreeBSD
C compiler	Any ANSI C compiler

MPI	LAM MPI MPICH MPI 1.0.0 or higher with chp4 device OpenMPI

6.2 Contents of HeteroMPI distribution

HeteroMPI for Unix distribution contains the following:

Directory	Contents
README, AUTHORS	Copyright information, Contact information
INSTALL	Installation instructions
configure…	Configure scripts for installation
docs	HeteroMPI manual for programmers
man	Manual pages for HeteroMPI API
src	Source code for HeteroMPI
include	Header files
tests	Tests for testing HeteroMPI library
mpC	Source of mpC

6.3 Installation

You should have MPI installed on your system. Please make sure that **`mpicc`** and **`mpirun`** tools are in your **`PATH`** environment variable.

Unpack the HeteroMPI distribution, which comes as a tar in the form heterompi-x.y.z.tar.gz. To uncompress the file tree use:

`shell$ tar -zxvf` heterompi-x.y.z.tar.gz

where x.y.z stands for the installed version of the HeteroMPI library (say 1.2.1, 2.0.0, or 3.1.1).

The directory 'heterompi-x.y.z' will be created; execute

```
shell$ cd heterompi-x.y.z
shell$ mkdir build
shell$ ../configure –prefix=<HMPI Installation directory>
shell$ make all install
```

HeteroMPI installation directorycontains the following:

Directory	Contents
bin	Binaries *mpcc*
include	Header files
share	Manual pages for HeteroMPI API
lib	Libraries *libhmpi.a*, *libmpc.a*

After you have successfully installed HeteroMPI, use the tests in the directory "*tests*" to test the installation.