CCA Teams: Proposed API for Supporting Process Groups at the CCA Level

Version 0.4

Version	Date	Description
0.1	07/20/2007	Initial Version
0.2	09/17/2007	Updated version with extended description of APIs
0.3	10/05/2007	This version includes the changes proposed during the MCMD telecon on 09/28/2007 ▪ It has been decided that threads might be too complicated to deal at this level. So threads are not included ▪ emphasis on mixing multiple programming models
0.4	10/18/2007	During the September telecon and October CCA meeting presentation, it is suggested that the "global id" should be represented as an opaque object. Major changes in this version with respect to global ids and ranks.
0.5	01/17/2007	▪ based on feedback from January CCA meeting ▪ SIDl file created based on the Spec ▪ A new figure included which contains methods and classes ▪ A prototype implementation based on this spec ▪ Sample MCMD driver code

Abstract

This document proposes and describes a high level API for managing processor groups (*aka* Teams) at the CCA level. The main objective is to develop a simple, yet generic, processor group model that can map to processor groups/communicators in various parallel programming models. Our model should support basic functionalities like:

- Ability to create and destroy groups
- Assign global process id to processes
- Group translators

1. Introduction – Background and Motivation

As high-end computers move from offering thousands of processors to tens and hundreds of thousands of processors, users are increasingly challenged to find additional parallelism in their application in order to effectively utilize them. When a single parallel activity doesn't scale sufficiently, it would be desirable to be able to launch many distinct parallel tasks, each using a subset of the available processors. This type of programming goes by many names, including multi-level parallelism, multiple-program multiple-data (MPMD), multi-tasking, etc. Traditional parallel programming models provide only the most basic support for this kind of programming, if they provide any support at all. The

dynamic nature and the encapsulation provided by the component approach maps quite naturally to the concepts of MPMD programming.

The goal of the Multiple Component Multiple Data (MCMD) Working Group [1] is to define, develop, and deploy a high-level infrastructure for MCMD programming in a CCA environment. Some of the use cases that benefit from MCMD programming are:

- Hierarchical Parallelism in Computational Chemistry
- Coop Parallelism
- Ab Initio Nuclear Structure Calculations
- Coupled Climate Modeling
- Molecular Dynamics, Multiphysics Simulations

The details of these use cases are available in MCMD Working group wikipage [1]. Based on execution model (Figure 1), MCMD use cases can be broadly classified as follows:

- Single MPI job
    - Multi-level parallelism
    - Task parallelism (Molecular dynamics)
- Multiple MPI Jobs (Figure 1)
    - Coop-parallelism
    - Ab-Initio Nuclear Structure Calculations

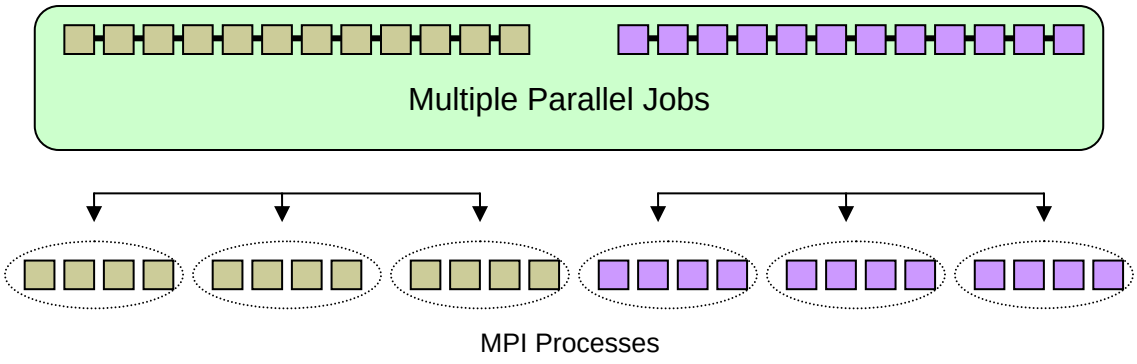

Figure 1: Generic MCMD execution model example (i.e. two jobs running on the same parallel computer). In this example, there are 2 parallel jobs (MPI/GA/PVM jobs) indicated by 2 different colors, and each box corresponds to a process (assume, MPI process). Each parallel job has multiple process groups.

In order to implement the high-level infrastructure for MCMD programming, we need to have an abstract interface for process group management at the CCA level. In order to express and manage hierarchical parallelism though the use of processor groups, it is

essential to support processor groups at the component level. This promotes parallelism at the component level, parallelism within the component, and parallelism within a subroutine. [2] explains library level support for hierarchical process group management. CCA process group management should have the following capabilities:

- Support various execution models
    - E.g. coop parallelism vs. single mpirun
- Support different programming Models (Table1)
    - MPI, Global Address Space (GAS) models including Global Arrays (GA), CAF, etc.
- Global process and group ids
- Should be MPI-Friendly (minimum requirement)
- ID translators
    - E.g. to facilitate the translation of CCA Group to MPI group

Table 1 presents an example of different approaches (i.e. MPI and GA processor group management).

MPI	GA
▪ Groups and Communicators ▪ WORLD Group is always by default ▪ MPI_COMM_WORLD ▪ Parent level synchronization for creating groups	▪ Groups only ▪ Default group can be changed ▪ As per design, parent level sync is not required

Table 1: Differences between MPI and GA processor groups

2. MCMD Programming Model

We need a simple, yet generic, processor group model that can map to processor groups in various parallel programming models. This model should support the following environments:

- Single executable parallel job
- Multi partition application (see Figure 1. e.g. multiple parallel jobs, cooperative parallelism)
    - This multipartition application can be
        - homogenous (e.g. all parallel jobs are MPI based) OR

        - heterogeneous i.e. jobs may contain a combination of multiple programming models like MPI, GA, PVM, etc.
- Distributed application
    - Multiple parallel jobs on different machines

3. Supporting Process Groups at the CCA Level

3.1 CCA Team

A process "group" or "team" is referred to an ordered collection of processes (similar to groups in MPI). From here onwards, we will use the terminology "team" instead of "group" because the concept of CCA Team is more general than an MPI group. The CCA team can contain a list of processes from multiple parallel runs (or distributed processes across various machines). For example, let us say we have 2 parallel MPI jobs as follows. A CCA team may contain 64 processes, which includes processes from each of these jobs below.

- ***Job #1:*** mpirun –np 32 ocean.x (running ocean model on 32 processes)
- ***Job #2:*** mpirun –np 32 land.x (running land model on 32 processes)

As per CCA specification, CCA does not specify how these processes communicate with each other (within a job or across jobs). It is entirely up to the user to define a communication model for processes communicating within an job (e.g. using MPI,GA, PVM, etc) or across jobs (e.g. sockets, Babel RMI).

3.2 CCA Representation for group id, membership

It is desirable for CCA teams and component management capabilities to be independent of:

- Parallel programming models
    - MPI, PVM, CAF, GA support idea of processor groups/teams. MPI model must be supported
- Languages (e.g. PGAS Languages)

If the components developed from different parallel models (e.g. GA in NWChem and MPI in ScaLAPACK, PETSc) interact, then there should be a universal way of representing team ids, team membership, etc.

3.3 Global Id Specification

In MPI, all processes within a single mpirun are given a unique MPI process rank starting from 0 to N-1 (where N is number of MPI processes). Since we consider a generic execution model as in Figure 1, there should be a unique way of specifying a "global id" for each unit of execution (processes across multiple mpiruns). This global id should be a combination of: <*global id*> = <*job id*> + <*process rank*>.

Each process in the MCMD environment is associated with a global id object. This object contains more information about the process like global rank (similar to MPI rank), job id, etc.

3.4 MCMD Team Management Initialization

There is a need for a MCMD team management initialization, where all processes from all jobs participate in initializing a parallel MCMD execution environment. For example, in ccaffeine [3] CCA framework, "ccafe-batch" is used as an executable for SCMD (single component multiple data) jobs, which initializes the parallel environment (e.g. calls MPI_Init). Likewise, there is a need for an executable for MCMD application, which initializes the MCMD environment. Let us call this executable as "ccafe-mcmd".

3.4.1 MCMD Directory Service

The MCMD generic execution model (Figure 1) contains multiple MPI jobs. A directory service registers the number of MPI jobs and the size of each job (i.e. number of processes) in a MCMD execution environment. This is achieved by the participation of all root processes (for example, process 0) from MPI jobs. The root processes interact with each other (using files, sockets, or Babel RMI) and collects the information about every MPI job. This approach is scalable as only one process from each job is participating, rather than all the processes in the MCMD execution environment.

3.4.2 What does ccafe-mcmd provide?

- Directory service for each process to register
- Interoperate with the underlying CCA MCMD layer to create the following
  - Global CCA Team
    - contains all processes
    - In case of single mpirun, this maps to MPI_COMM_WORLD
    - In case of multiple mpiruns, this global team includes all processes from multiple mpiruns
  - Initialize data structures
    - Assign global id for each process
    - Job information (each mpirun is one separate job)
      - Number of jobs (i.e. mpiruns)
      - Size of each job (i.e. number of processes in each mpirun)
      - Assign a unique job id for every job (say, range from 0…N-1)

3.5 Local or Collective Call

All CCA calls (e.g. port creation, framework services, etc) are local to the calling process. Similarly CCA Team calls are designed to be local. They cannot be collective call for the following reason(s):

- According to CCA specification, CCA does not enforce or assume what programming model the user will be using. User has the option to select any programming model.
    - Since the programming model is not visible to CCA layer, CCA cannot use a collective call.

Limitations:

- As of now there is no support for
    - distributed systems environment (No use cases)
    - threads
    - dynamic team management (e.g. MPI dynamic processes)
        - However there are plans to expand the MCMD spec to incorporate this feature

Open Issue(s):

- Should any of the CCA Team calls be collective?

4. API Specification

As mentioned in previous sections, our APIs (Figure 2) should support basic functionalities like:

- Ability to create and destroy groups
- Global process/group ids
- Group translators

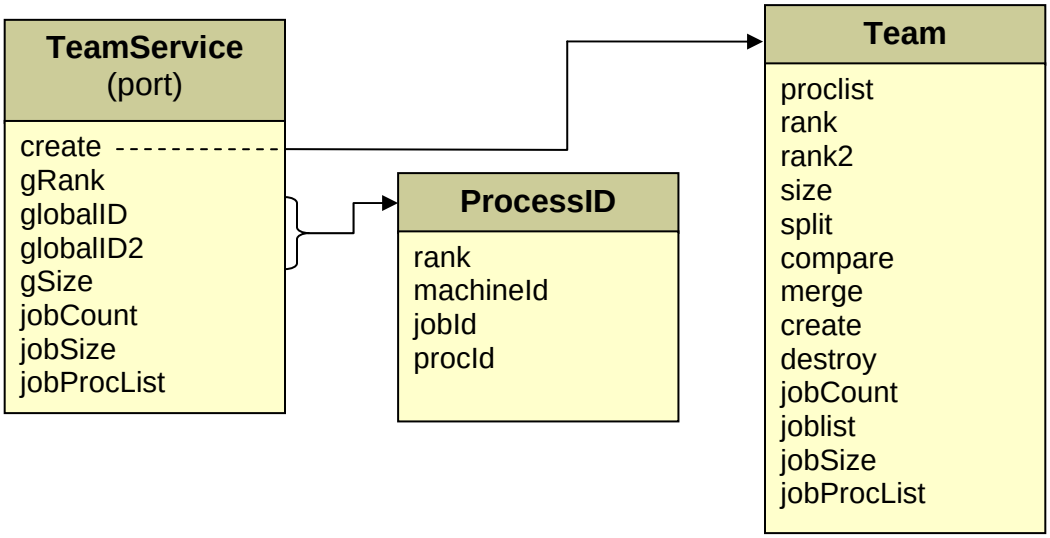

Figure 2: MCMD team service: classes and methods.

4.1 Team Creation

Listed below is the API to create a CCA team

4.1.1. create() - Create a new CCA Team

Format

```
Team create(in array<int> plist, in int size)
```

- (input) plist - list of processes (specified by global ranks) to appear in the new team to be created. This list should be in sorted order
- (input) size - number of elements in array plist (i.e. the size of the new team)
- (output) new team's object/handle is returned, which is derived from the above parameters in the order defined by plist.

Purpose

To create a new CCA team, i.e. a collection of processes

Issues/Notes

There is no concept of "parent" groups in the CCA team layer. All teams are created based on the global ranks. This contradicts the MPI model, where the list of process global ranks is not always based on MPI_COMM_WORLD (world group).

4.2 Parallel MCMD Environment

The flow of the MCMD program depends on the rank of a process and finding out about the rest of the world. In case of multiple mpiruns, it also depends on the job information (like job id, size of each job, etc). Below are some APIs that can be used to know about your parallel MCMD environment.

4.2.1 Global ID

a. globalID() – global Id object of the calling process in the global team

Format

```
ProcessID globalID(in int jobid, in int pid)
```

- globalID - returns the global id object of the any process given the jobid and the local rank (within the job)

Purpose

To determine the global ID of a process

4.2.2 Global ID by Rank

a. globalIDbyRank() – global Id object of the calling process in the global team

Format

```
ProcessID globalID(in int rank)
```

- globalID - returns the global id object of the any process given its global rank

Purpose

To determine the global ID of the a process

4.2.3. Rank and Size

a. gRank() – determines global rank of the calling process in the global team

Format

```
int gRank()
```

- gRank() - returns the global rank of the calling process

Purpose

To determine the global rank of the calling process

b. gSize() – determines the size of the global team

Format

```
int gSize()
```

- (output) Returns the global size of the global team

Purpose

To find out how many processes are involved in the execution of this MCMD program.

4.2.4 Global process ID APIs

Each process in the MCMD environment is represented with a unique global rank (integer). However, a global rank does not give enough information about the MCMD environment of a process. This issue is addressed by associating each process with a global id object (of type "ProcessID"), which is created and initialized by default through ccafe-mcmd. This section provides some global id specific APIs (or methods) that can be accessed using this object/handle to know more about the calling process. . For example, in case of C++, the functions will be invoked as follows:

```
ProcessID gid = globalIdByRank(2); // global id obj of process
                                   // with global rank as 2
jobid  = gid.jobId();
pid    = gid.procId();
```

ID specific APIs are as follows.

Format

```
int jobId()
int machineId()
int procId()
int rank()
```

- jobId() - returns the job id of any process with the given global id (i.e. *gid*). Job ids range from 0 to n-1, where "n" is the total number of jobs/mpiruns returned by jobCount (For example, in case of single mpirun execution model, this is always 1. In case of multiple mpiruns this "n" corresponds to the number if mpiruns).
- machineId() – returns the machine id of where the corresponding process resides
- procId() - Given the global id, this function returns the process id of that corresponding process within the job. Process ids range from 0 to j-1, where "j" is the job size (see jobSize()).
- rank() - returns the global rank of the process

Purpose

Determine the process specific information from the global ID object/handle.

Issues/Notes

The pids (returned by procId() are unique only within the job. Across multiple jobs this pids will be replicated. i.e. the first process in all the jobs will have the process id of 0 (i.e. similar to MPI rank). In the MCMD environment, the unique process ids are specified by global ranks (i.e. *gRank*)

4.2.4 Job Information

Format

```
int jobCount()
int jobSize(in int jobid)
int jobProcList(in int jobid, in int size,
                        in array<int>ranklist)
```

- jobCount() - total number of MPI jobs/mpiruns being run in the current MCMD environment
- jobSize() – returns the number of processes belong to this job id
- jobProcList() – returns the list of processes (i.e. global rank list) that belong to the given job id
    - input parameters – jobid, size of the job
    - output – ranklist (i.e. ordered list of processs in this jobid)

Purpose

To determine the job information of any processes/jobs in the current MCMD environment

4.2.5 Global ID Translators

As explained earlier globalID and globalIdbyRank can be used as ID translators as it determines the global id object of any process.

4.3 Team APIs

When a team is created using the create() call explained in earlier Section 4.1.1, it returns an object or handle of type “Team”. This section provides some Team specific APIs (or methods) that can be accessed using this object/handle to know more about the given team. Team specific APIs are as follows:

All the below functions/APIs will be invoked through the team object or handle. For example, in case of C++, the functions will be invoked as follows:

```
Team oceanTeam = create (...)
myTeamRank = oceanTeam.rank();
myTeamSize = oceanTeam.size();
```

4.3.1 Team rank

Format

```
int rank()
int rank2(in int gid)
```

- rank - returns the local rank of the calling process within this team

- rank2 - returns the local rank of the any process (given its global rank) within this team.
    - Returns -1 if not part of the team.

Purpose

To determine the local rank (or pid) of the calling process within this team. The rank ranges from 0 to n-1, where n is the size of the team

4.3.2 Team size

Format

```
int size()
```

- Returns the size of this team

Purpose

To determine the local rank (or pid) of the calling process within this team. The rank ranges from 0 to n-1, where n is the size of the team

4.3.3 Process list

Format

```
void procList(in array<int> plist, in int size)
```

- Returns the sorted list of processes (i.e. global ranks) that belong to this team
    - output – plist (i.e. ordered list of process in this team)

Purpose

To determine the global ranks of all processes in this team

4.3.4 Split

Format

```
Team split(in int n)
```

- Splits the given team into “n” teams. It returns the new team object/handle created, in which the calling process belong.

Purpose

This is an abstract function used to split the given team into multiple teams.

4.3.5 Compare

Format

```
int compare(in Team t1)
```

- Compares the current team object with the given team t1. If the context of two teams are same, then it returns 0, else 1.

Purpose

Compares two teams

4.3.6 Merge

Format

```
Team merge(in Team t1)
```

- Merges the current team object with the given team t1, and returns the newly created Team object/handle

Purpose

Merges two teams

4.3.7 Create - Create a new sub-team within this team

Format

```
Team create(in array<int> plist, in int size)
```

- (input) plist - list of processes (specified by global ranks) to appear in the new team to be created. This list should be in sorted order
- (input) size - number of elements in array plist (i.e. the size of the new team)
- (output) new team's object/handle is returned, which is derived from the above parameters in the order defined by plist.

Purpose

To create a new CCA sub-team within the current team, i.e. a collection of processes

Issues/Notes

This model supports the MPI model given how sub-teams are created. For example: Here the current team can be considered as the parent team.

4.3.8 Destroy

Format

```
void destroy()
```

- Destroys the current team

Purpose

To destroy a previously created CCA team

4.4 Team ID Translators

CCA Team Id translators are part of the CCA Team APIs. As mentioned earlier, all the below functions/APIs will be invoked through the team object or handle. Team ID translators facilitate in translating the global ids in CCA team to respective job and process ids, which can then be used to create a process group in any programming model (say, MPI groups/communicators).

4.4.1 Team Job Information

Format

```
int jobCount()
int jobList(in array<int> jlist, in int njobs)
int jobSize(in int jobid)
int jobProcList(in int jobid, in int size, in array<int>ranklist)
```

- jobCount() - There are many processes in a team. These processes can come from one or many MPI jobs. jobCount() can be used to determine the number of MPI jobs involved in this team.
- jobList() – To determine the list of job ids part of this team (jlist). The number of jobs (njobs) is determined using jobCount().
- jobSize() – To determine the number of processes part of this team which also belongs to the given job id. Returns the number of processes belong to this job id and also part of this team
- jobProcList() – returns the list of processes (i.e. global rank list) that belong to the given job id and also part of this team.
    - input parameters – jobid, size of the job
    - output – ranklist (i.e. ordered list of processs in this jobid)

Purpose

To determine the job specifc information (global id) of any processes/jobs with in the current team. This facilitates id translation with in a CCA team.

5. MCMD Example

Here is a very simple MCMD program in execution. This is just a pseudo code for illustration purposes.

Note: MCMD::Init initialized the MCMD parallel environment, where all processes from all jobs must participate (collective across all parallel jobs)

```
#define OCEAN 0
#define LAND  1
```

```
MCMD::TeamService ts = <get mcmd TeamService port>;
ts.init()// initializes the MCMD parallel environment

/* get my global rank and ID*/
int my_rank   = ts.gRank();
ProcessID gid = ts.globalIDbyRank(my_rank);

/* get job info */
int my_jobId   = gid.jobId();
int njobs      = ts.jobCount();
int my_jobsize = ts.jobSize(my_jobid);

if(njobs !=2 ) Abort();

/* Create teams where each team consists of procs from same job */
if(my_jobid == OCEAN)
{
   int *gRanklist   = new int[my_jobsize];
   int *mpiRanklist = new int[my_jobsize];

   ts.jobProcList(my_jobid, my_jobsize, gRanklist);
   Team ocean_team = ts.create(gRanklist, my_jobsize);

   /* my local team rank */
   int team_rank = ocean_team.rank();
    . . .
   . . .
}
else if(my_jobid == LAND)
{
   int *gRanklist  = new int[my_jobsize];
   int *gaRanklist = new int[my_jobsize];

   ts.jobProcList(my_jobid, my_jobsize, gRanklist);
   Team land_team = ts.create(gRanklist, my_jobsize);

    . . .
   . . .

}
```

5.1 Sample MCMD Driver code (c++)

```
int32_t
mcmd::Driver_impl::go_impl () {
  // DO-NOT-DELETE splicer.begin(mcmd.Driver.go)
    gov::cca::Port mcmdport = svc.getPort("mcmdport");
    if(mcmdport._not_nil())
    {
       mcmd::TeamService ts = ::babel_cast< mcmd::TeamService >(mcmdport);

       // initialize XM's mcmd service – implementation specific for now
       xm::TeamService ts_xm = ::babel_cast< xm::TeamService > (ts);
       ts_xm.init();

       int32_t jobcnt  = ts.jobCount();
       int32_t grank  = ts.gRank();
       int32_t gsize   = ts.gSize();
       …….
       mcmd::Team t1 = ts.create(ranks, teamsize);
       …..
             printf("%d: A new team is created. size=%d, rank=%d\n",
                      grank, t1.size(), t1.rank());
       … …
     }
}
```