The *Double-layer Master-Slave Model* : A Hybrid Approach to Parallel Programming for Multicore Clusters

User's Manual for the HPCVL DMSM Library

Gang Liu and Hartmut L. Schmider
High Performance Computing Virtual Laboratory, Queen's University,
Kingston, Ontario, Canada K7L 3N6

Version 3.1

December 8, 2011

Abstract

This manual explains the structure and usage of a library that implements the *Double-layer Master-Slave Model* , a flexible and generally applicable model for parallel programming that can be deployed across clusters with multicore nodes. The model is based on a two-tier structure:

- The Message-Passing Interface (MPI) is used to distribute work-groups across a cluster, based on a simple Master-Slave approach.
- Locally, on each cluster node, OpenMP compiler directives are used to work through the work items in a group based on a dynamic “All Slaves” model.

Several variations of this basic structure were implemented in a library in Fortran and C. In this manual, we discuss the user interfaces for these libraries and for the necessary user-supplied functions. We also outline the capabilities of the model and its limitations.

Contents

Chapter 1

Introduction: The *Double-layer Master-Slave Model*

1.1 Parallel Programming on Multicore Clusters

Modern clusters usually consist of nodes with multiple CPU's or cores. In addition, recent core architectures support multiple threads being concurrently executed on a single core (hardware multi- or hyper-threading). The standard approach to exploit parallelism on a cluster is by use of the Message-Passing Interface MPI. This requires the communication between independent "heavy-weight" processes that are deployed across the nodes in a distributed-memory fashion. While it is possible to run multiple independent MPI processes on a single multicore cluster node, it may not be the most efficient use of resources. This is particularly true for multithreaded cores which exhibit a high degree of resource sharing between threads, and are therefore prone to run into bottlenecks, particularly connected to communication resources.

A more appropriate approach to use multicore and multi-threaded nodes is through the use of thread libraries such as *Pthread* or compiler directives such as *OpenMP*. The resulting multithreaded programs do not require communication resources and use shared memory for information exchange between "light-weight" processes instead. As a result, resource-bottlenecks are much less likely. Unfortunately, due to their dependence on the shared-memory architecture of the computer, such programs cannot be deployed across a cluster.

It therefore appears that the ideal approach for programming for a multicore, multi-threaded cluster, is a hybrid one: combining *MPI* to deploy processes across the cluster to the individual nodes, with *OpenMP* parallelization of each process "intranode" to make efficient use of the multicore and/or multi-thread structure of the hardware.

The main complicating issue with this approach is the lack of thread safety of most *MPI* libraries. This means in practice that communication resources cannot be accessed through *MPI* functions by more than one thread per process at a time, because internal communication buffers and other variables are not protected against race conditions, and are not forced to be consistent for all threads. In other words, the message envelope in MPI is not thread-specific. As a result, the onus is on the programmer to make sure that *MPI* and therefore the internode-communication subsystem are consistently only seeing only one thread per rank and that all information that is communicated is made available to all threads at a time when they need it.

The simplest approach to achieve this is to completely separate *MPI* communication from those parts of the individual processes that are executed in a multithreaded fashion. In other words, all *MPI* communication happens in the serial regions of the code, and at the time when internode-communication happens, only one thread (preferably the "master thread") is active on either end. In many cases, this approach is sufficient to obtain smoothly running and well-scaling code, and at least one of the models we discuss in this manual follows this scheme. It is arguably the safest way to write a hybrid program.

However, in some cases, there is a need to go beyond this approach. If communication is required to obtain information from another node while working through portions of the code that are *OpenMP* multi-threaded, there is no alternative but to specifically protect communication calls and to use shared data structures to pass information. This complicates issues considerably, as not only do we have to deal with race conditions on the communication resources, but also on all data structures that are involved. Most of the models discussed here are of this type. We have made an effort to keep the complexities of this approach away from the user, but an understanding of the issues is still helpfull when using our library.

1.2 Master-Slave and All-Slave Models

One of the basic principles of parallel programming is the avoidance of workload imbalances. If computing tasks are distributed among a number of independent processes, care has to be taken that the allocation of the tasks is not done too statically if there is a chance that tasks differ substantially in their execution times. If this rule is not followed, some of the processes will spend too much time waiting to obtain further work, with the corresponding "hit" on execution time and therefore scaling.

If the computational burden associated with each of the tasks is heterogeneous and unpredictable, a dynamic schedule of the tasks to the processes needs to be adopted. For distributed-memory systems, this often takes the form of a "Master-Slave Model". In this model, one process - the Master - is dedicated to work distribution, while the others - the Slaves - work on individual tasks. Once a Slave has completed a task, it sends a signal to the Master and obtains a new one. If the communication times for signalling the Master and supplying Slaves with new tasks are small compared with task execution, this model stands to avoid workload imbalances and scale well in the limit of a large number of tasks. When used with a large number of processes, the dedication of one process to "management" is deemed an acceptable price for the resulting scalability. On a distributed-memory cluster, the Master process usually is necessary to insure that no tasks are performed more than once or skipped, as well as to supply the nodes with the necessary information to perform the tasks. Of course, the independence of the tasks is a precondition for this model to work.

On a shared-memory system such as a multicore node or an SMP server, it is usually not necessary to dedicate a specific processor for the Master tasks. This is because the shared-memory structure of the machine makes all necessary information for task execution readily available, and the role of the Master can be played by a simple protected counter variable. Each process that has run out of work accesses this variable and supplies itself with the next task. We call this an "All-Slaves Model". This model is mostly used on shared-memory systems, where it combines the advantages of the Master-Slave model with the additional bonus of the participation of all available processes in the workload execution.

In this manual, we are discussing a combination of these two approaches in an *MPI/OpenMP* hybrid framework. This is called the *Double-layer Master-Slave Model* .

1.3 The *Double-layer Master-Slave Model* (DMSM)

It is easy to see that an *MPI* implementation of the "Master-Slave Model" can readily be expanded to a hybrid model if each of the tasks is further divided into sub-tasks which are then executed in an *OpenMP* parallel fashion. In practice, it is easier to combine multiple tasks into "workload packages" and distribute these packages across nodes using an *MPI* master-slave approach. Each of the tasks in the package is then internally distributed on each node via an "All-Slaves Model".

1. On a cluster, one node (usually the one running the process with rank 0) is chosen as the master node where thread 0 is dedicated to be the Master. All other nodes are used as Slaves. Communication between them is done via *MPI* calls.

2. In the beginning, each Slave signals to the Master that it is idle, and the Master sends out instructions and data to all Slaves establishing their workload. This workload consists of many smaller tasks that are handled internally by the Slave nodes.

3. Any Slave node works through such a workload package without any further communication to the Master node. Once the workload is executed, a signal is sent to the Master requesting another package. This is executed in the same manner until a stop signal is received, at which point execution ends.

4. The Master continues to serve any requests for new packages until all the available work is distributed. Then a stop signal is passed to the Slaves. During this process, the Master insures that no package is distributed more than once or skipped.

Step 3. in the above Master-Slave scheme - the workload execution on the nodes - is in itself done through OpenMP:

1. Once a new workload package is received on a Slave node, a task counter is reset, and multiple *OpenMP* threads begin to work on the package.

2. Each process (thread) assigns itself to a given task in the package and moves the task counter forward to avoid double execution or skipping.
3. The task counter has to be protected by placing updates into a critical region or by use of locks.
4. Once all tasks in a package are executed (or at least assigned), a single thread communicates with the Master for a new package.

The last point on this level establishes a distinction between different variations of the model. A more robust implementation waits until all tasks in a package have been performed completely before communicating with the Master for a new package outside of the *OpenMP* parallel region. A more flexible version of the code obtains a new package from the Master as soon as the first thread in the *OpenMP* set runs out of tasks, even if others in the set are still working on tasks from the old package. This has to be done from inside the *OpenMP* parallel region and therefore needs careful protection.

1.4 The Structure of this Manual

This manual is meant as a User's Guide for the DMSM library.

- The present Chapter 1 serves as a general introduction and explains the working of the Double-layer Master Slave model.
- The next Chapter 2 outlines the different variations of the model as implemented in the DMSM library.
- Chapter 3 focusses on the user interfaces necessary to run the *Double-layer Master-Slave Model* in the simple case where work-related data are available on the nodes, and results do not have to be collected from the nodes by the library.
- Chapter 4 deals with cases where work-related data are supplied to the nodes dynamically by the library.
- Chapter 5 discusses cases where computed results are collected dynamically from the nodes by the library.
- Chapter 6 deals with extended usage of the *Double-layer Master-Slave Model* library where individual jobs are executed in *parallel.* item Chapter 7 addresses technical issues, such as installation, compilation, and availability of the library.

Each of the chapters is divided into sections.

Chapter 2

The DMSM Library

The library documented in this manual was developed by Gang Liu at *HPCVL* in 2011 to enable the deployment of a large number of independent computing tasks with unpredictable execution times across a cluster with multithreaded multicore nodes. At present, the library is implemented in Fortran 90 and C.
The current version of the library is 3.1.

2.1 Overview

- Nine variations of the *Double-layer Master-Slave Model* are implemented. See below (Section 2.3) for details.
- In the simplest case, where data required for all tasks already reside on the slave nodes, and results are either directly written by the Slaves or sent to the Master after completion of all tasks, the user only supplies a simple job routine (`DO_THE_JOB`) (Chapter 3).
- In a more complex situation where task-related data are sent along to the Slaves whenever a new task package is requested, the user needs to supply an additional job group preparation routine (`JOB_GROUP_PREPARATION`) (Chapter 4).
- In the most general case that involves the collection of results from a previous package whenever a new one is issued, the user supplis a data collection routine as well (`RESULT_COLLECTION`) (Chapter 5).
- Each individual job may still be run in parallel using either *OpenMP* or *MPI* (Chapter 6).

2.2 Naming Convention

All library routines, internal variables and constants, modules used by the library, and files generated are named with a prefix `DMSM_`. It is best not to use this prefix for user routines etc. to avoid conflicts and overlaps.

2.3 Nine Variations

Depending on the complexity and flexibility of the basic algorithm, the *Double-layer Master-Slave Model* can be used in nine variations. These are labeled by an integer variable that serves as an argument in the standard function call to the library. Whenever applicable, the main thread on the Master node always serves as the Master to distribute job packages.

Table 2.1: Variations of the *Double-layer Master-Slave Model*

Arg	**Other threads on Master**	**Slave Communication**
11	none	serial
12	pre-allocated	serial
13	dynamic	serial
21	none	parallel 0
22	pre-allocated	parallel 0
23	dynamic	parallel 0
31	none	parallel any
32	pre-allocated	parallel any
33	dynamic	parallel any

The first column of the above table indicates the integer label that must be passed to the library to select the corresponding variation of the model. This label is computed according to $l = 10m + s$ where m describes the behaviour of the threads on the "master node", and s indicates nature of the communication with threads on the slave nodes.

The second column "**Other threads on Master**" indicates what task are performed by the non-master threads - if any - on the Master node. To make good use of the available resources, they can either work through packages according to a *pre-allocated* static schedule, or *dynamically* obtain job packages from the Master and work through them using an All-Slaves scheme.

The last column "**Slave Communication**" explains how *MPI* communication takes place on the side of the Slave nodes. "serial" indicates that work packages are obtained before a parallel thread-set is invoked, i.e. from the *serial* region of the code. "parallel 0" means that a specific thread (number *0*) obtains new packages from inside the *OpenMP parallel* region when needed. "parallel any" means that *any* of the slave threads can communicate with the Master from inside the *parallel* region when a new work package is needed.

Clearly, Variation 11 is the most static but also the most "safe" since no communication from inside *OpenMP* parallel regions takes place at all, and therefore *OpenMP* and *MPI* parallelism are completely separated. Variation 33 is the most complex and flexible, and therefore is most suitable to avoid load imbalances. Anytime a slave thread detects that no further work items are left unallocated, it obtains another work package from the Master. In addition, threads on the Master node work on their own work package to make optimal use of the computing resources.

2.4 Three Special Cases

Specific values for the number of *MPI* processes and the number of threads per process in the *Double-layer Master-Slave Model* lead to simpler models.

- If both the number of *MPI* processes and the number of threads per process is 1, the workload is executed in serial. It is best to choose the number of jobs per group to be 1 as well in this case.
- Otherwise, if only the number of *MPI* processes is 1, the library performs a single-layer “all slaves” model that is implemented entirely with OpenMP.
- If only the number of threads per *MPI* process is chosen to be 1, then no OpenMP parallelism is used and the system defaults to a “master-slaves” model. Jobs are distributed in job groups and the processed serially. If a standard “master-slaves” model is desired, the number of jobs per group needs to be set to 1. Note that it is also possible to incorporate a pre-existing *parallel* program (using either *OpenMP* or *MPI*) into the job execution routine to perform each individual job. In this case, many such parallel jobs can be executed simultaneously. This ”extended usage” of the model will be discussed further in Chapter 6.

Chapter 3

Library Interfaces

The central part of the DMSM library consists of 4 basic interfaces that must be called in a fixed sequence. This may be done by the user directly, or through an additional "wrapper" interface that calls the basic four in the correct order. In the following, we are going to discuss the usage of these interfaces with a focus on the **Fortran version**. An additional section (3.7) is devoted to the differences between Fortran and C interfaces. The routines are as follows:

- `DMSM_INITIALIZE` initializes the DMSM library and requires the specification of the basic framework for the *Double-layer Master-Slave Model* . Its usage is explained in section 3.1.
- `DMSM_WORKING` is the core routine of the DMSM library. It requires the specification of the job routines and is explained in section 3.2.
- `DMSM_JOB_DISTRIBUTION_CHECKUP` checks the proper execution of the work distribution and does not require any arguments. It is explained in section 3.3.
- `DMSM_FINALIZE` finalizes the usage of the DMSM library. No arguments are required. It is explained in section 3.4
- `DMSM_ALL` is a wrapper interface for the above routines and requires the same arguments as `DMSM_INITIALIZE` and `DMSM_WORKING`. It is explained in section 3.5.

In addition to these interfaces, the user has to supply at least one subroutine that encapsulates the execution of a single work task. The interface of that routine is discussed in section 3.6.

3.1 DMSM_INITIALIZE

This subroutine is used to initialize the DMSM library and requires the specification of the basic framework for the *Double-layer Master-Slave Model* . It must be called first before any other DMSM interfaces are used. *MPI* must be initialized first through a call to `MPI_INIT` before calling this routine. The routine has the following interface:

```
SUBROUTINE DMSM_INITIALIZE( &
   THREADS_PER_PROCESS, &
   JOB_DISTRIBUTION_PLAN, &
   TOTAL_JOBS, &
   NUM_OF_JOBS_PER_GROUP)
```

The 4 arguments of the subroutine are used to define basic parameters of the *Double-layer Master-Slave Model* . They are listed in Table 3.1.

Table 3.1: Arguments of `DMSM_INITIALIZE`

	Name	**Type**	**Function/Purpose**
1	`THREADS_PER_PROCESS`	integer	Requested number of threads per process
2	`JOB_DISTRIBUTION_PLAN`	integer	Specification of the variation of the *Double-layer Master-Slave Model* , see Table 2.1
3	`TOTAL_JOBS`	integer	Total number of jobs to be performed
4	`NUM_OF_JOBS_PER_GROUP`	integer	Number of jobs contained in each job group

3.2 DMSM_WORKING

This subroutine causes a full run of the *Double-layer Master-Slave Model* to be executed. The basic parameters of the run are expected to have been previously specified by a call to `DMSM_INITIALIZE`. The interface requires up to three user-supplied subroutines to be specified as arguments. These subroutines need be declared `EXTERNAL` by the user, or – preferably – an `INTERFACE` declaration has to be provided. All their arguments are of type integer. The interface of `DMSM_WORKING` is:

```
SUBROUTINE DMSM_WORKING( &
   DO_THE_JOB, &
   JOB_GROUP_PREPARATION, &
   RESULT_COLLECTION, &
   COLLECTION_ENABLED)
```

The arguments of this routine are listed in Table 3.2.

Table 3.2: Arguments of DMSM_WORKING

	Name	Type	Args	Function/Purpose
1	DO_THE_JOB	subroutine	1	Executes a single job
2	JOB_GROUP_PREPARATION	subroutine	4	Communicates initial data for a job group from the master to a slave node.
3	RESULT_COLLECTION	subroutine	2	Communicates available results from a slave node to the master.
4	COLLECTION_ENABLED	logical	n/a	If true, results are collected dynamically throughout, otherwise only once at the end.

Note that while the first of these routines (`DO_THE_JOB`) **must** be supplied by the user, the other two can be "skipped" if the nodes need not be supplied with supporting data before the processing of a job package, and if no result collection inside the library is required, respectively. The interface of `DMSM_WORKING` is overloaded, so that the first argument is always meant to be `DO_THE_JOB`. If the routine is called with two arguments, the second one is supposed to be `JOB_GROUP_PREPARATION`. If with three, the second is `RESULT_COLLECTION`, and the third `COLLECTION_ENABLED`. If all four are specified they have to be in the above order.

Section 3.6 discusses the function of `DO_THE_JOB`, while the other two will be explained in later chapters (4 and 5).

3.3 DMSM_JOB_DISTRIBUTION_CHECKUP

The purpose of this routine is to check if all jobs of a run of `DMSM_WORKING` have been properly executed. It produces a list of all jobs together with the *MPI* processes and threads that executed them and writes this list and other detailed information, such as timings, about job execution into a file `DMSM_journal.txt` that is written to the working directory on the master node. It requires no arguments.

3.4 DMSM_FINALIZE

This routine finalizes the use of the *Double-layer Master-Slave Model* library by deallocating arrays and releasing resources. It should be called after all jobs have finished and no further use of `DMSM` routines is made. It requires no arguments.

3.5 DMSM_ALL

The routine DMSM_ALL is a wrapper that makes consecutive calls to all 4 previously discussed routines: DMSM_INITIALIZE, DMSM_WORKING, DMSM_CHECKING, and DMSM_FINALIZE. The arguments of DMSM_ALL are the combination of the ones of DMSM_INITIALIZE and DMSM_WORKING, i.e. this routine has the following interface:

```
SUBROUTINE DMSM_ALL( &
   THREADS_PER_PROCESS, &
   JOB_DISTRIBUTION_PLAN, &
   TOTAL_JOBS, &
   NUM_OF_JOBS_PER_GROUP, &
   DO_THE_JOB, &
   JOB_GROUP_PREPARATION, &
   RESULT_COLLECTION, &
   COLLECTION_ENABLED)
```

The meaning of the arguments are shown in tables 3.1 and 3.2. The overloading structure for the last four arguments in DMSM_ALL is the same as in DMSM_WORKING.

3.6 DO_THE_JOB

The most important user-supplied routine for the *Double-layer Master-Slave Model* is obviously the one that executes a specific single job. This routine must be supplied by the user. It only requires a single integer argument, namely the number of the job to be done. In Fortran, this lies between 1 and the maximum number TOTAL_JOBS which is the third argument in the call to subroutine DMSM_INITIALIZE (see Table 3.1).

It is recommended to make an INTERFACE declaration of the routine to make sure that its name is properly passed to the DMSM library. The actual code for the routine can then be done in a separate file. The following code snippet declares a user defined function inside an INTERFACE and uses it in a single-argument call to DMSM_WORKING.

```
program main
...
interface
     subroutine single_job(m)
       implicit  none
       integer :: m
     end subroutine single_job
end interface
...
call DMSM_WORKING(single_job)
...
end program main
```

This routine is repeatedly called by all work threads in the framework of the *Double-layer Master-Slave Model* from inside a parallel region if applicable. It should therefore be programmed in a *threadsafe* manner, meaning that different jobs should not write data into the same shared location. Specifically, the use of external shared data structures such as common blocks, modules, or global variables, should be done with great caution, as should output into shared files.

3.7 The C interfaces

The interfaces of the `C` version of the `DMSM` routines discussed above are almost identical to the ones in Fortran. The one for the initialization routine is:

```
void DMSM_Initialize( \
   int threads_per_process, \
   int job_distribution_plan, \
   int total_jobs, \
   int num_of_jobs_per_group)
```

The meaning of the arguments is the same as in Fortran. The function is of type `void`, i.e. it does not return a value. For the "working routine" the interface is:

```
void DMSM_Working( \
   void (*do_the_job)(int), \
   void (*job_group_preparation)(int,int,int,int), \
   void (*result_collection)(int,int), \
   int collection_enabled)
```

Since `C` does not support overloading of functions, this interface requires four arguments even if job preparation and/or result collection is not required. In those cases, the corresponding arguments may be replaced by `NULL`. The argument `collection_enabled` is of type `int`. If it is `1` results are collected dynamically throughout, if it is `0` they are collected only once at the end. If `result_collection` is specified as `NULL`, i.e. not used `collection_enabled` is ignored. All specified functions are of type `void`, i.e. no return value. They need to be prototyped, for instance:

```
void  single_job (int);
...
main (){
...
  DMSM_Working(single_job,NULL,NULL,0);
...
}
```

for the simple case where only a job execution function `single_job` is defined and used as an arument in `DMSM_Working`. As the only argument, `single_job` receives a job number ranging from 0 to `total_jobs-1`.

Chapter 4

Job Group Preparation (JOB_GROUP_PREPARATION)

In many cases all necessary data to do the jobs in the *Double-layer Master-Slave Model* can be distributed among the nodes before the execution of the model starts. In this case the communication of the initial data for a job group or package is not necessary and the JOB_GROUP_PREPARATION argument of DMSM_WORKING can be skipped in Fortran or NULL in C. In some cases it is necessary to supply the job data dynamically at the time when a new job package is issued by the master. In this case the user can supply such a routine with 4 integer input arguments:

```
interface
   subroutine my_job_preparation(first_job,last_job,call_rank,dest_rank)
    integer :: first_job,last_job,call_rank,dest_rank
   end my_job_preparation
end interface
```

The first two arguments first_job and last_job define the first and last job number in the group of jobs that is being prepared. The third argument is the rank of the calling *MPI* process, which can be 0 (if the routine is called by the Master) or a positive integer (if it is called by a slave node). The last argument is the rank number of the slave node that is being supplied with job group data.

The routine is called by the DMSM library when a new job group is assigned to an *MPI* process. The details of the implementation of this routine are left to the user. Usually, the routine will contain at least one set of point-to-point *MPI* communication calls, i.e. an MPI_SEND call for the case when it is called by the Master, and an MPI_RECV call for when it is called by a slave node. It is recommended to use blocking *MPI* calls for this to avoid possible race conditions. In cases where job group data need to be initialized inside the Master node, the communication of the data can be replaced by a simple copy from one data structure to another.

The interface for the job group preparation function in `C` is virtually identical to the one in Fortran:

```
void my_job_preparation(\
   int first_job,int last_job,int call_rank,int dest_rank)
```

In many cases, a job preparation routine communicates job related data from the Master into a temporary data structure on the slave. This information is over-written every time a new job group is requested. This implies a race condition in those cases where the new job data are requested by a single thread inside the OMP parallel region of a slave process. This is because new data are requested whenever the communicating thread runs out of work, at a time when other threads might still be doing work that requires job related data from the previous job group. To avoid this race condition, the DMSM library provides a pair of lock routines that protect access to job related data until they are not needed anymore. The first of these routines is:

```
DMSM_WAIT_FOR_INITIAL_LOCKS
void DMSM_Wait_For_Initial_Locks()
```

in Fortran and `C`, respectively. **This routine should be called inside the job group preparation routine before updating job related data for a slave process**. This will cause the job preparation routine to halt until it is safe to update these data. The signal that this is the case comes from call to the second of the lock routines:

```
DMSM_UNSET_AN_INITIAL_LOCK
void DMSM_Unset_An_Initial_Lock()
```

This routine should be called in the job routine after the point where the specific job related data have been copied to local data structures and can therefore be safely overwritten. When all threads on an *MPI* process that are still doing work have called this routine, the thread that is waiting after a call to `DMSM_WAIT_FOR_INITIAL_LOCKS` can proceed to update data for a new job group. All other work (lock declaration, initialization, setting, and deleting) is done by the library.

Chapter 5

Result Collection (RESULT_COLLECTION)

Sometimes, the results of jobs that have been executed by the slave nodes need to be communicated back dynamically to the master node. This should be done "in bulk" after some jobs have finished to avoid interference between multiple active threads on multiple slave nodes. In such a case, the user can supply a routine that collects all available results and sends them to the Master. This routine needs to have two integer arguments:

```
interface
   subroutine my_result_collection(call_rank,source_rank)
    integer :: call_rank,source_rank
   end my_result_collection
end interface
```

As with the job group preparation, the interface for the data collection function in `C` is virtually identical to the one in Fortran:

```
void my_result_collection(\
   int call_rank,int source_rank)
```

The first argument is the rank of the calling *MPI* process, which can be 0 (if the routine is called by the Master) or a positive integer (if it is called by a slave node). The second argument is the rank number of the slave node that is returning result data. If the value of `collection_enabled` (the fourth argument of a `DMSM_WORKING` call) is equal to `1` the result collection routine is called whenever a new job group is assigned, if it is equal to `0` it is called only once when all jobs are done. The details of the implementation of this routine are left to the user.

In many cases, a result collection routine communicates result data from a temporary data structure on the slave process to the master. If both `call_rank` and `source_rank` are 0 (i.e. both are on the Master), results from a temporary data structure should be copied to the final result structure. Otherwise, the slave sends information about the amount of result data to the master, followed by the data themselves. A specific signal (such as a zero) is suggested to indicate “currently no results”. Once the results are communicated, the temporary structure is usually “emptied” to avoid the same results being sent repeatedly. Inside the library, the result collection routine is called only from a critical region to protect the final result structure on the Master. If this data structure is updated *only* by this routine the structure is safe.

If the temporary data structure for results is shared, one thread may send it to the Master and update it, while another has new results and writes them into the structure. To avoid this race condition, the DMSM library provides another pair of lock routines that protect the temporary job result data while they are being communicated and reset. These routines are:

```
CALL DMSM_SET_NODE_RESULT_LOCK
void DMSM_Set_Node_Result_Lock()
```

and

```
CALL DMSM_UNSET_NODE_RESULT_LOCK
void DMSM_Unset_Node_Result_Lock();
```

in Fortran and `C`, respectively. **The first routine should be called inside the job result collection routine before communicating and resetting the temporary data structure that contains results, as well as in the job routine before updating that structure. The second routine should be called after communication/reset or after an update, respectively.**

Chapter 6

Extended Usage

6.1 Individual Jobs Parallelized Using *OpenMP*

It is possible to set the number of threads per *MPI* process in `DMSM_INITIALIZE` to 1 and reduce the model to a simple *MPI* Master-Slave model, while still use multiple OpenMP threads for each job to be executed in parallel. The first argument in `DMSM_INITIALIZE` only sets the number of OpenMP threads that are used by the DMSM library. As a result, many OpenMP parallel jobs may be executed simultaneously on multiple slave nodes. However, no computations can be performed on the Master node in this model. Note that the number of threads on the Slave nodes has to be set outside the DMSM library in the user code.

6.2 Individual Jobs Parallelized Using *MPI*

This library, reduced to the pure MPI Master-Slave Model, can be used to distribute independent *MPI* program runs to the nodes of a cluster. This implies that *MPI* is used on two different levels: one (the "distribution" level) to communicate program runs and the corresponding data to the nodes, and another (the "job" level) to execute individual jobs on the nodes. The latter are usually considerably more communication intensive than the former, and therefore the shared-memory structure of a single node is desirable because of fast SHM-layer communication.

To implement such a model, it is necessary to generate two different *MPI* communicators. One - we call it the "distribution communicator" - contains only one process per node which we call the "local master". This communicator is used to distribute jobs and related data from a head node to all job nodes using the library with `THREADS_PER_PROCESS` set to 1. The other (called the "job communicator") contains all processes that execute a specific job internally.

The above communicators may be implemented by the user, for instance through calls to `MPI_COMM_SPLIT`. However, we provide a facilitating routine that will generate these communicators. The interface for this routine is:

```
DMSM_GEN_COMM_MPI_ALL( &
   PROCS_PER_JOB, &
   ALL_COMM, ALL_RANK, ALL_SIZE, &
   JOB_COMM, JOB_RANK, JOB_SIZE, &
   DIST_COMM, DIST_RANK, DIST_SIZE )
```

The first two arguments are *input* arguments, all others are *output*. The first argument specifies how many *MPI* processes will be used per job execution. For instance, if we want to use the Master-Slave model on 10 nodes with 8 cores each for job execution for a total of 80 processes, this number would be 8. The second argument is the name of the original communicator that includes *all* processes. This will usually be `MPI_COMM_WORLD`. Table 6.1 shows all arguments.

Table 6.1: Arguments of `DMSM_GEN_COMM_ALL`

	Name	**Type**	**In/Out**	**Explanation**
1	`PROCS_PER_JOB`	Integer	In	*MPI* processes per job execution
2	`ALL_COMM`	Communicator	In	Global communicator
3	`ALL_RANK`	Integer	Out	Rank of calling process in global communicator
4	`ALL_SIZE`	Integer	Out	Size of global communicator
5	`JOB_COMM`	Communicator	Out	Job (local) communicator
6	`JOB_RANK`	Integer	Out	Rank of calling process in job communicator
7	`JOB_SIZE`	Integer	Out	Size of job communicator
8	`DIST_COMM`	Communicator	Out	Distribution communicator
9	`DIST_RANK`	Integer	Out	Rank of calling process in distribution communicator
10	`DIST_SIZE`	Integer	Out	Size of distribution communicator

Arguments labeled as type "Communicator" are `INTEGER` in Fortran and `MPI_Comm` in `C`. Note that each node has their own `JOB_COMM` of the same name and each process belongs to one of the `DIST_COMM`s. No ambiguity occurs as the job communicators are mutually exclusive. For completeness, here is the `C` interface for this routine:

```
void DMSM_Gen_Comm_MPI_All( \
   int Procs_Per_Job,\
   MPI_Comm All_Comm, int *All_Rank, int *All\_Size,\
   MPI_Comm *Job_Comm, int *Job_Rank, int *Job_Size,\
   MPI_Comm *Dist_Comm, int *Dist_Rank, int *Dist_Processes);
```

To run such an *MPI* based two-layered model with the library, a call to `DMSM_GEN_COMM_MPI_ALL` should be followed by a call to another, simplified wrapper routine:

```
DMSM_MPI_ALL( &
   TOTAL_JOBS, &
   JOBS_PER_GROUP, &
   DO_THE_JOB, &
   JOB_GROUP_PREPARATION, &
   RESULT_COLLECTION, &
   COLLECTION_ENABLED)
```

were the types and meaning of the arguments are as in Tables 3.1 and 3.2. The interface is similar to the one of `DMSM_ALL`, but the first two arguments are not reuired as they pertain to the OpenMP all-slaves part of the model which is not used here. Again for completeness, the `C` interface is:

```
void DMSM_MPI_All( \
   int Total_Num_Of_Jobs,\
   int Num_Of_Jobs_Per_Group,\
   void (*Do_The_Job) (int),\
   void (*Job_Group_Preparation) (int,int,int,int),\
   void (*Result_Collection) (int,int),\
   int Collection_Enabled);
```

Chapter 7

Technical issues

7.1 Hardware and Software Requirements

The following components are required to use the library successfully:

- Operating system: Preferably Unix based (Solaris, Linux, AIX, Irix, etc.)
- Standard tools: *MPI* library.
- Fortran 90 and C compilers, preferably native to the operating system. However, the cross-platform gnu compilers should work as well.

7.2 Library Components & Installation

The DMSM library is distributed in the form of source code. This archive can easily be extracted via the *tar* command. All files are expanded into a single directory to avoid complicated installation procedures.

For the Fortran version of the library, there is only one component that includes all subroutines, functions, and modules. This is called **dmsm.f90**. For the C version, additional header files are required. The C components are called **dmsm.c** and **dmsm.h**.