Adjusting Process Count on Demand for Petascale Global Optimization

by

Nicholas Radcliffe

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

Layne T. Watson, Chair
Adrian Sandu
Kevin Shinpaugh

Dec. 15, 2011
Blacksburg, Virginia

Key words: Message Passing Interface (MPI); Global optimization; Petascale computing; Dynamic process count

Adjusting Process Count on Demand for Petascale Global Optimization

by

Nicholas Radcliffe

(ABSTRACT)

There are many challenges that need to be met before efficient and reliable computation at the petascale is possible. Many scientific and engineering codes running at the petascale are likely to be memory intensive, which makes thrashing a serious problem for many petascale applications. One way to overcome this challenge is to use a dynamic number of processes, so that the total amount of memory available for the computation can be increased on demand. This thesis describes modifications made to the massively parallel global optimization code pVTdirect in order to allow for a dynamic number of processes. In particular, the modified version of the code monitors memory use and spawns new processes if the amount of available memory is determined to be insufficient. The primary design challenges are discussed, and performance results are presented and analyzed.

ACKNOWLEDGEMENTS

I have been very lucky to have had such a knowledgeable advisor, Dr. Layne T. Watson. His advice has been invaluable, and his prodding has helped me get to where I needed to be. I am fortunate to have taken an interesting and informative course with Dr. Adrian Sandu, and I am glad that he is on my thesis committee. I am also thankful to Dr. Kevin Shinpaugh for taking time out of his busy schedule to be on my committee. I thank Dr. Masha Sosonkina for advice and support with my research, and I thank Dr. Raphael Haftka and Dr. Michael Trosset for their advice.

I thank my lab mates for their support and kindness during my time here at Virginia Tech. I thank Dave Easterling for helping me get started when I first arrived at Virginia Tech, and I thank Shubhangi Deshpande for her help with typesetting my thesis.

I would like to thank the National Energy Research Scientific Computing Center (NERSC) for use of the Carver cluster, and Aron Ahmadia at the King Abdullah University of Science and Technology (KAUST) for use of the Shaheen and Neser clusters.

I would also like to thank my parents for making sure that I received a good education, and for their love and support over the years. Finally, I would like to thank the love of my life, Cecile Nevares, for making all the hard work worthwhile.

This work was supported in part by AFOSR Grant FA9550-09-1-0153 and AFRL Grant FA8650-09-2-3938.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1: INTRODUCTION

The ultimate goal of the work presented in this thesis is to develop a robust global optimization code that runs efficiently and effectively at the petascale. This means that the program must run efficiently, and be able to tolerate failures of any kind, on a cluster with hundreds of thousands of cores. There are a number of challenges that must be overcome before this is possible, for instance, designing the optimization code so that the speedup obtained by using multiple cores scales up to hundreds of thousands of cores. This challenge alone is enough to make the petascale daunting [1].

Beyond maintaining the efficiency of the code at the petascale, one must ensure that the code is robust and can recover from any number of failures. One possible failure results when a node in the cluster crashes. This type of failure is generally dealt with by including a checkpointing mechanism in the code. Another type of failure that can occur is insufficient main memory, which can lead to thrashing. Given the crippling effects of thrashing, a mechanism for dealing with insufficient memory would be indispensable to a large number of scientific and engineering codes that hope to run efficiently at the petascale.

The main contribution of this work is a global optimization code that is able to detect insufficient levels of available memory, and in response spawn new processes on nodes with available memory. The solution to insufficient memory presented in this thesis is specific to a particular global optimization code (`pVTdirect`), but many aspects of the design, as well as the lessons learned, can be applied to a number of parallel scientific or engineering codes, especially those that make use of the master-worker design pattern.

Chapter 2: LITERATURE REVIEW

Adaptive parallel applications are applications that can alter their process count in response to changes in available resources. Adaptive parallel applications are primarily used in grid computing due to fluctuations in available resources (a user might not want cycles being borrowed from his machine when he is using it), as well as the loose coupling of tasks. As far as the authors know, dynamically adjusting the process count of a parallel MPI application with tightly coupled processes is unique to the current work.

Tools have been developed to help users write adaptive parallel applications. In [11], a system that enables OpenMP programs to run on a network of workstations with a variable number of nodes is described. There are similar systems for grid computing, such as the system described in [12]. The adaptive parallel systems intended for grid computing are of little use for the purposes of this work, because they depend on the noninteraction of processes in the user application (i.e., the user application must be embarrassingly parallel). The communication between the masters in `pVTdirect` complicates increasing their count.

Process migration has been used to adjust the number of MPI processes running on a physical processor (although the total number of processes remains unchanged). Adaptive MPI [13] uses processor virtualization to dynamically manage resources. In particular, virtual MPI processes can be migrated from one physical processor to another, allowing applications written with Adaptive MPI to increase the process count on a particular processor (while decreasing the process count on one or more processors). Although Adaptive MPI is intended for use with applications developed in C++, it might seem as if process migration more generally could be useful for the present work. For instance, if masters lacked sufficient memory, one or more masters could be migrated to processors with more available memory. However, this is not an ideal strategy for increasing the memory available to masters in `pVTdirect` for two reasons. First, if a process is migrated to a new node, then the memory that had been used to store boxes on the previous node is lost. Second, unused processors are in general needed to increase overall available memory (see Chapter 4.2.1). Within master-worker style applications, such as `pVTdirect`, it may be wasteful to allocate one process per node, and so it is difficult to ensure enough memory without spawning an extra process on a "fresh" node.

Chapter 3: DESCRIPTION OF DIRECT

The algorithm DIRECT (DIviding RECTangles) by D. R. Jones [2] is a deterministic global optimization algorithm. DIRECT does not require the computation of the gradient of the objective function, or even objective function values (ranking information is sufficient). It performs Lipschitzian optimization, but does not require knowledge of the Lipschitz constant.

DIRECT works as follows [3]. The algorithm begins with an initial box normalized to the unit hypercube. The objective function (assumed to satisfy a Lipschitz condition) is evaluated at the center of this box. The current minimum value is initialized to this value. An evaluation counter m and an iteration counter t are initialized to $m = 1$ and $t = 0$. The following process is repeated until m or t reaches some prespecified limit (although the subroutine `pVTdirect` [3] supports several other stopping conditions).

Selection. Identify the set S of "potentially optimal" boxes. Here "potentially optimal" means that (1) for some Lipschitz constant K, the box potentially contains a point with smaller objective function value than any other box, and (2) $F(c) - K \cdot L/2 \leq f_{min} - \epsilon|f_{min}|$, where F is the objective function, c is the center point of the box, K is the same Lipschitz constant, L is the box diameter, f_{min} is the current minimum value for the objective function, and ϵ is a small, nonnegative, fixed constant.

Sampling. Select one of the potentially optimal boxes B from S. For box B, identify the set I of dimensions with maximum side length L, and let $\delta = L/3$. Sample the function at the points of the form $c \pm \delta e_i$ for each $i \in I$, where c is the center of the box and e_i is the ith standard basis vector. Update m.

Division. Divide the box containing the point c into thirds along the dimensions in I, beginning with the dimension with the least value of min $\{f(c + \delta e_i), f(c - \delta e_i)\}$, and ending with the dimension with the greatest such value. Update the minimum value.

Iteration. Remove the box B from the set of potentially optimal boxes S. If $S = \emptyset$, then increment t and go to **Selection**. Otherwise, go to **Sampling**.

The method of choosing the subbox according to both objective function value and box size gives DIRECT its local and global aspects. DIRECT performs a convex hull computation to determine potentially optimal boxes without using the Lipschitz constant directly (see Figure 1 for an illustration). From Figure 1, it is clear that if a box is on the convex hull, then the box has an objective function value that is minimal amongst all boxes of the same size (notice that the set of boxes of the same size forms a "box column", as seen in Figure 1). Since every box is ultimately examined, DIRECT will not get stuck at a local optimum, but will instead perform a global search of the feasible set. Further details can be found in [2].

VTDIRECT [4] is a Fortran 95 implementation of DIRECT that uses dynamic data structures and has options and stopping conditions not in earlier implementations of DIRECT.

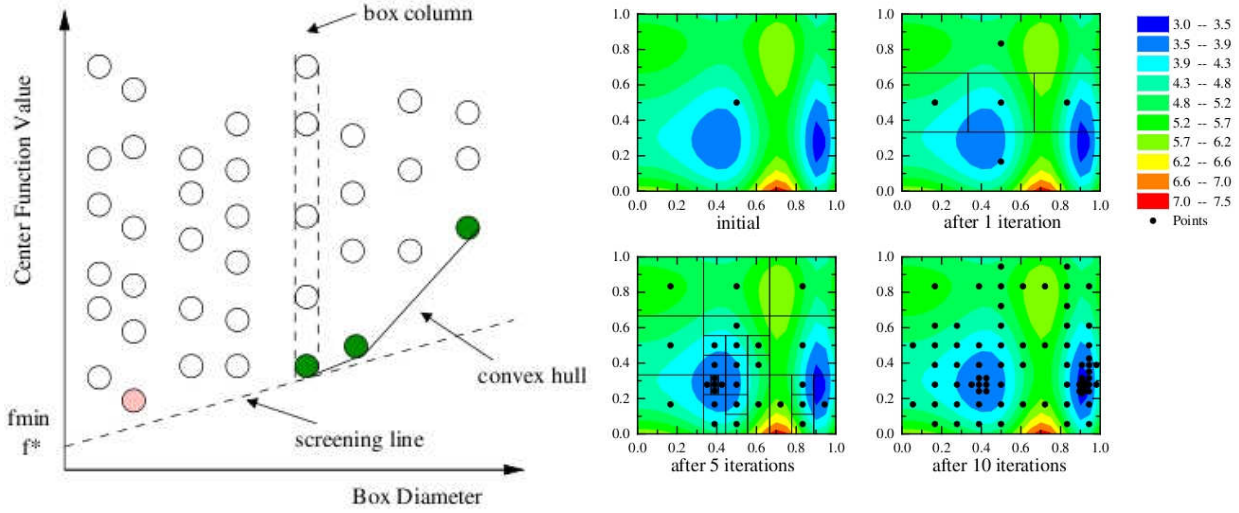

Figure 1. Illustrations of DIRECT's box columns (left), as well as VTdirect in action (right).

Based on experience from using the serial code VTDIRECT on applications such as aircraft design, cell cycle modeling, and wireless communication system design, VTDIRECT was polished and extended to include both serial and massively parallel (terascale) versions. These codes eventually became part of the ACM TOMS algorithm VTDIRECT95 [3]. In this Fortran 95 package the user callable subroutines are `VTdirect` (serial) and `pVTdirect` (parallel). `pVTdirect` is efficient at the terascale [5, 6, 7] on real applications, but likely not so at the petascale. The motivation for the present work is modifying `pVTdirect` to be efficient at the petascale, where applications in systems biology and nuclear physics await such capability.

3.1 Important Details of the Implementation

`pVTdirect`, the parallel version of `VTdirect` and the only version under consideration in this thesis, makes use of the master-worker design pattern. The masters handle the program logic, whereas the workers perform function evaluation tasks. The masters are tightly coupled, in the sense that the state of one master significantly affects the state of other masters. There is a global worker pool shared by all masters. Workers from the pool select masters to which they send requests, and masters respond to these requests by sending points at which to evaluate the objective function. Optionally, the initial box can be partitioned into *subdomains*, each with assigned masters, where masters assigned to separate subdomains operate independently. In fact, when n subdomains are used, it is almost like running n separate instances of `pVTdirect`, with the important exception that the separate subdomain optimizations share some resources, e.g., workers. Both the masters and the workers run through a main loop. Since the masters handle all the program logic,

an iteration of `pVTdirect` will be defined as an iteration of the main loop for the masters. The masters synchronize at every iteration of their main loop via `MPI_BARRIER`, whereas the workers do not synchronize at all.

It is possible to have more than one master per subdomain. The computational work done by masters is relatively insignificant, but more than one master per subdomain may be desired—the masters store the current state of the search (in the form of box columns), and the memory available to multiple masters may be required to completely store the current state. By the nature of the computation, the memory required to store the current state of the search increases with time. This means that the current collection of masters may become unable to store the current state of the search, which may lead to thrashing. Thrashing can be avoided by increasing the number of processors in the computation, hence increasing the amount of memory available to store the current state of the search. Since the memory burden is primarily on the masters, it is necessary to spawn new masters on idle processors in order to obtain a substantial amount of extra memory. Ideally, one would like to *dynamically* increase the number of masters, rather than restarting the computation (which may last for days, or even months) with a greater number of masters. Doing this in the context of MPI and the (necessarily) distributed data structures used by `pVTdirect` is nontrivial, and constitutes the core topic of this thesis.

Chapter 4: PROBLEM DESCRIPTION

Running low on memory is a problem for masters in `pVTdirect`. `pVTdirect` was modified in order to keep track of memory use, and to spawn new masters when the amount of available memory falls below a certain threshold. Spawning new masters when memory is low, and subsequently integrating them into the running program, is a complicated and subtle task. The primary challenges are (1) determining which processes should do the spawning (this choice affects other design factors, such as how the communication scheme is handled), (2) executing the spawn when the workers behave asynchronously, (3) updating the communication scheme of the newly expanded collection of processes, and (4) integrating the spawned masters into the current job, obtaining a coherent execution unit. A modification of `pVTdirect`, called `spVTdirect`, is considered as a possible solution to the challenges described above.

The code `spVTdirect` works as follows. The number of boxes possessed by a particular master is monitored, and if the memory needed to store those boxes exceeds a user-defined threshold, a spawn request is made. When all processes have detected the spawn request, `MPI_COMM_SPAWN` is executed and new masters are spawned. All processes, both spawning and spawned, must then update their state in order to integrate the new masters into the already-running job. After the state update procedure is completed, the current iteration restarts at the top of the main loop for the masters, and the workers restart at the top of their main loop.

4.1 Choosing the Spawning Communicator

The choice of communicator used to spawn the new masters is important, because it affects how communication between workers and spawned masters will be handled. After `MPI_COMM_SPAWN` has been executed, a handle for an intercommunicator is returned. Since the local group of the intercommunicator contains the processes that performed the spawn, and the remote group contains the spawned masters, the spawned masters can only communicate directly with the processes that performed the spawn. Consequently, as far as communication is concerned, the best choice of spawning communicator is the entire collection of current processes. This choice of communicator facilitates communication between the spawned masters and all current processes.

However, using the entire collection of current processes to perform the spawn is problematic, because the spawning subroutine `MPI_COMM_SPAWN` is both blocking and collective over the set of spawning and spawned processes. If the entire collection of current processes performs the spawn, then the masters and workers must all make a collective blocking call to `MPI_COMM_SPAWN`. This is simple for the masters, which synchronize at every iteration of their main loop via `MPI_BARRIER`. However, the situation is more complicated with the workers since they operate asynchronously, in the sense that attempts to synchronize their behavior with `MPI_BARRIER` (or any collective, blocking operation) generally lead to deadlock.

4.2 Executing the Spawn

Every master monitors its own memory usage. If the amount of available memory for a master falls below a given threshold, it notifies all other processes of a need to spawn new masters (i.e., obtain more memory). After a process has received notification of a spawn request, that process first calls a spawning subroutine that executes `MPI_COMM_SPAWN`, followed by a subroutine that updates state.

Using all of the current masters, as well as the workers, to perform the spawn is a delicate procedure. Since the masters already synchronize at the top of their main loop, it would be tempting to synchronize all processes at that point, and then use a collective communication to notify all processes of a spawn request. However, all attempts to synchronize the workers with `MPI_BARRIER` have lead to deadlock. Two options for notifying all processes of a spawn request have been explored in the current work. One option is to notify all processes of a spawn request by having the requesting process write to a "spawn request" file. This can be problematic, because different processes read the spawn request file at different times, and hence one process may begin executing `MPI_COMM_SPAWN` while the others are still busy, which can lead to deadlock. A time sharing method can be used to prevent deadlock from occurring when a process performs point-to-point communications—when performing a point-to-point communication, a process goes back and forth between checking if the communication has completed, and reading the "spawn request" file to see if there is a pending request. Collective communications are only performed by the masters, and they only occur when the masters and workers are not communicating. Hence, they are never a source of deadlock during the spawn notification procedure.

The time sharing method is effective, but it is not portable, due to its reliance on a shared file system. A more portable solution is to first notify all masters of a spawn request using a type of reduction operation (technically, `MPI_ALL_REDUCE` is used), and then have the lead master notify the workers. If the `MPI_ALL_REDUCE` is executed at the top of the main loop for the masters, then the masters can prepare for a spawning event before the next iteration even begins. After the masters have been notified, the simplest solution for notifying the workers is to use code that is already in place in `pVTdirect`. In `pVTdirect`, the workers receive messages from masters at every iteration of an inner loop for the workers, and the tag associated with a message determines the response to that message. So, in `spVTdirect`, the lead master can simply send a message to each worker with a tag indicating a pending spawn request. The workers receive the messages and prepare for spawning.

On iterations without a pending spawn request (this is the vast majority of them), there is no extra overhead for the workers, and the only overhead for masters is one `MPI_ALL_REDUCE` per iteration. This overhead is minimal, and performance results have shown that the overhead has negligible impact on the runtime per iteration (see Chapter 7 for performance results).

4.2.1 Further issues with spawning

There are a few further issues related to spawning that must be considered. First, MPI does not support spawning on a cluster with a scheduler [8], as `MPI_COMM_SPAWN` requires the user to provide a list of processes in the form of a host file (the host file contains node names, like "ithaca42", not just ranks). Consequently, a Fortran 95 module designed to support spawning on scheduled clusters was developed and tested. Currently, the module (called `QSPAWN`) only provides subroutines that build a host file for spawning, but further support for spawning on scheduled clusters may be added in the future. In order to build a host file, the names of all nodes scheduled for the job must be obtained, as well as the number of cores available on each node. MPI provides support for determining node names, but not for determining the number of cores available on a node—for this, the OpenMP command `OMP_GET_MAX_THREADS` is used.

Second, the new masters should ideally be spawned on idle processors in order to obtain a substantial amount of extra memory. Where these idle processors come from is a serious concern. One possibility is to replace a worker with the spawned master. However, it is not guaranteed that a worker will be running on its own processor—it is possible for the worker to be running on a node along with other workers and a master (since masters are memory hogs, it is preferable to place them on separate nodes). So, the only solution that will work consistently is to spawn new masters on unused nodes. For clusters without a scheduler, this can be done by providing `spVTdirect` with a list of all available nodes (possibly obtained from a system administrator). For clusters with a scheduler, one solution is to use a system call to push a new job onto the scheduler's queue. Performance concerns dictate that the computation should continue, and hence state update be postponed, until after the new job is launched by the scheduler, as their may be a substantial delay before the new job is launched. After the job is launched, communication can be established between the current and newly-launched jobs, and state update can proceed as described in Chapter 4.4.

Another solution is to run multiple jobs simultaneously, allowing these jobs to share a global pool of nodes. Rather than launching each job separately, a single job with one process (but many reserved nodes) could be launched using the cluster's scheduler. The single process could then spawn all of the specified jobs using `MPI_COMM_SPAWN`, as well as maintain a list of available nodes. All involved jobs would simply take nodes off the list as they consume them, and repopulate the list with nodes as they finish with them. This idea of consolidating jobs can only work if (most of) the jobs have fluctuating resource requirements, allowing them to consume and release nodes periodically. Although the number of nodes needed by `spVTdirect` may increase with time, it never decreases. Consequently, it is not clear if this solution is feasible for `spVTdirect`.

4.3 Updating the Communication Scheme

The spawning procedure reorders some processes (the ranks of workers are translated), and add others (the spawned masters). This means that communicators must be updated in order to ensure that messages are sent to and from the correct processes. In particular, the state (ordering of processes) of the communicators after being updated must be consistent with the state of the communicators before the spawning procedure began.

The communication between the current and spawned processes depends on which subset of the current processes does the spawning. If only the set of current masters performs the spawning, then communication between the workers and the spawned masters becomes infeasible—the intercommunicator returned from the call to `MPI_COMM_SPAWN` only allows for direct communication between the spawning and spawned processes. If the workers are not involved in spawning the new masters, then communication between the workers and spawned masters must be indirect. Although indirect communication is possible (and can be coded cleanly with wrappers for the standard MPI communication subroutines), it is relatively inefficient as all communication between the workers and spawned masters must pass through one (or more) of the current masters. In particular, if there is only one master, then that master becomes a bottleneck for all communication between the workers and spawned masters. Therefore, it is preferable to have the set of all current processes execute `MPI_COMM_SPAWN`. In this case, communication between the workers and the spawned masters is direct. If every member of the current world of processes executes `MPI_COMM_SPAWN`, then the best way to update the communication scheme is as follows. The command `MPI_INTERCOMM_MERGE` is used to merge the local and remote groups of the intercommunicator returned by `MPI_COMM_SPAWN`. The processes in this merged intracommunicator must be reordered so that they are consistent with the current ordering of processes—masters have the lower ranks, starting with zero, and workers have the higher ranks, beginning with the number of masters. This allows `spVTdirect`, with an updated communication scheme, to continue to run properly.

However, the communication scheme was not originally updated as described above. This is because the authors had initially used only the set of current masters to perform the spawning. As explained above, this choice was made to simplify the spawning procedure itself, but such simplified spawning greatly complicates communication. In this case, communication is handled by using wrappers for the standard MPI communication subroutines. These wrappers are named `MC_`$\langle$ `subroutine name` $\rangle$, where `MC` stands for "many communicator". The wrapper subroutines take an array of communicators called `commArray` (rather than a single communicator) as an argument, allowing the communication scheme to adjust to increases in the number of masters. Processes are given a global rank within the collection of communicators specified by `commArray`. The global rank of a process in the ith communicator is

$$\text{rank}_{\text{global}} = N_1 + N_2 + ... + N_{i-1} + \text{rank}_{\text{local}},$$

where N_j is the size of the jth communicator for $j = 1, \ldots, i-1$, and $\text{rank}_{\text{local}}$ is the usual rank of the process within the ith communicator.

Such "many communicator" subroutines were written for both point-to-point and collective communications. For example, consider the "many communicator" subroutine for a point-to-point send communication, called `MC_SEND`. For the subroutine `MC_SEND`, the global rank of the receiving process is given as an input parameter. The global rank is used to determine the relevant communicator and the local rank of the receiving process within that communicator. If the global rank is strictly less than the size of `commArray` (1), then `commArray` (1) is the communicator and the local rank of the receiving process in `commArray` (1) is simply its global rank. If the global rank is greater than or equal to the size of `commArray` (1), then the receiving process must be an element of `commArray` (i) for some $i > 1$. In this case, the size of `commArray` (1) is subtracted from the global rank to obtain a new value, and this value is compared against the size of `commArray` (2) to determine if the receiving process is an element of this communicator. This process is repeated until the relevant communicator and local rank within that communicator are determined.

The idea behind `MC_SEND`, as well as the other "many communicator" subroutines, is to allow the communication scheme of `spVTdirect` to be updated simply and cleanly every time new masters are spawned. When new masters are spawned, only a few data structures (such as `commArray`) need to be updated. These data structures are then passed as input parameters to the "many communicator" subroutines, and the communication scheme is automatically adjusted to take into account the newly spawned masters. Stress tests were done to compare the performance of the two methods—"many communicator" and "merge"—described above. The results of these tests are presented in Chapter 5.

4.4 Integration of the Spawned Masters

The current job has n_m masters and n_w workers, whereas the spawned job has n_m masters and zero workers (notice that the spawned job is not intended to run on its own). These two jobs need to be integrated into a coherent execution unit with $2n_m$ masters and n_w workers. The integration of the two jobs is complicated by the inconsistencies in state between the current and spawned masters—at the point of spawning, the current masters have generally run through quite a few iterations of the main loop, whereas the spawned masters are just beginning. Dealing with the difference in state between the current and spawned masters is tricky, and it is all too easy for subtle problems to arise when attempting to integrate the spawned masters into the already-running job.

After the intercommunicator has been merged, the difference in state between the current and spawned masters must be dealt with in order to successfully integrate the spawned masters into the current job. One solution is to transfer state from one of the current masters to the spawned masters. Although this solution may seem obvious, the real

challenge is in the details of making this seemingly simple solution work. For a trivial example of the challenges involved, consider the following facts about `pVTdirect`. In `pVTdirect`, the lead master in a subdomain (i.e., the master with rank zero) behaves differently from the other masters in the same subdomain. Since, in general, the lead master is the only master guaranteed to exist, it makes sense to transfer the state of the lead master to the spawned masters. However, if done naively, this would mean that all spawned masters would think they were lead masters, which is obviously problematic. This problem is trivial—it is only meant to illustrate the sort of problems that can arise when the states of all processes are not properly updated after spawning new masters.

A conservative approach is taken to updating state after a spawning event. In general, an aspect of the state of a process is reset when it is not clear how to properly maintain and/or augment that aspect. This means that some information is lost; however, the lost information has no effect on the mathematical correctness of the algorithm. Succinctly, `VTdirect`, `pVTdirect`, and `spVTdirect` all produce exactly the same set of boxes and function values. Note that some aspects of state, such as the iteration counter, *must* be fabricated for the spawned masters (since the iteration number can be used as a stopping condition).

Since state is reset when it is not clear how to update and/or augment it, it is beneficial for the states of data structures to not be persistent across iterations (note that it is sufficient for the state of a data structure to be determined by a simple formula, i.e., the ith element of an array is the rank of the ith master). If the state of a data structure is not persistent (or if it can be determined by a simple formula), then its state is trivial to update after a spawning event. Hence, the less state that is persistent, the easier it is to update the state of a process after a spawning event. For example, consider the array `lcConvex`, which contains the convex hull box counters for every master in a subdomain. Since convex hull boxes are reassigned to different masters at every iteration, the values of `lcConvex` are recomputed at every iteration, and hence it is safe to reset `lcConvex` to an arbitrary state after a spawning event.

Now consider the following example of state that is persistent across iterations. When a worker chooses a master in order to make a request, the worker chooses from the set of busy masters, which requires the 2-dimensional arrays `masterID` and `masterStat`. The array `masterID` holds the ranks of every master in every subdomain, and the array `masterStat` holds the status ('busy' or 'idle') of every master in every subdomain. Updating the contents of `masterID` after a spawning event is trivial (its contents are determined by a simple formula), but it is not obvious how to update the contents of `masterStat` while maintaining the statuses of the spawning masters. Hence, a conservative strategy that simply (re)initializes the contents of `masterStat` is used.

4.4.1 Derived data types

The data structures with derived types used for storing box-related information do not have to be updated for any process. The main data structure holding box information is `mHead` (technically, `mHead` is just the initial link in a larger data structure). `mHead` is a fairly complex data structure, basically a dynamic list of matrices, where columns in the matrices represent box columns. If a particular box column grows beyond the height of the matrix, the column can be extended (multiple times, if necessary) using fixed-length arrays. The boxes stored by a particular process are unique to that process, and so the data structure pointed to by `mHead` need not be transferred to spawned masters—the spawned masters initialize `mHead` and fill it in with box information from scratch. This means that there is a substantial imbalance in the number of boxes stored by spawning and spawned masters. A dynamic load balancing mechanism is used to balance the number of boxes stored by each master (Chapter 5).

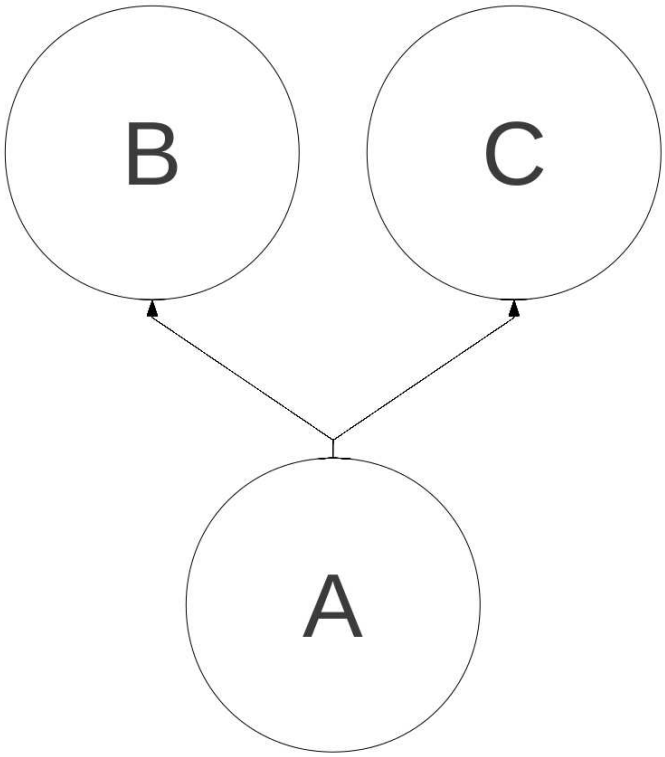

Figure 2. Illustration of the "many communicator" method.

Chapter 5: Comparison of the "Many Communicator" and "Merge" Methods

Stress tests were performed to compare the two methods—the "merge" method and the "many communicator" method—for updating the communication scheme after a spawning event. As described above, the "merge" method requires that the spawning process be collective over all processes. In this case, the intercommunicator returned from `MPI_COMM_SPAWN` is simply merged into an intracommunicator. The "many communicator" method was designed so that the spawning process need only be collective over the current set of masters. This method uses an array of communicators to implement a communication scheme that can adjust to increases in the number of masters.

In Figure 2, an initial communicator consisting of all current masters (A) spawns two child communicators (B and C), both containing new masters. In this situation, the "many communicator" method would be used to effectively merge A, B, and C into a single communicator. In Figure 3 (left), communicator A contains all current processes,

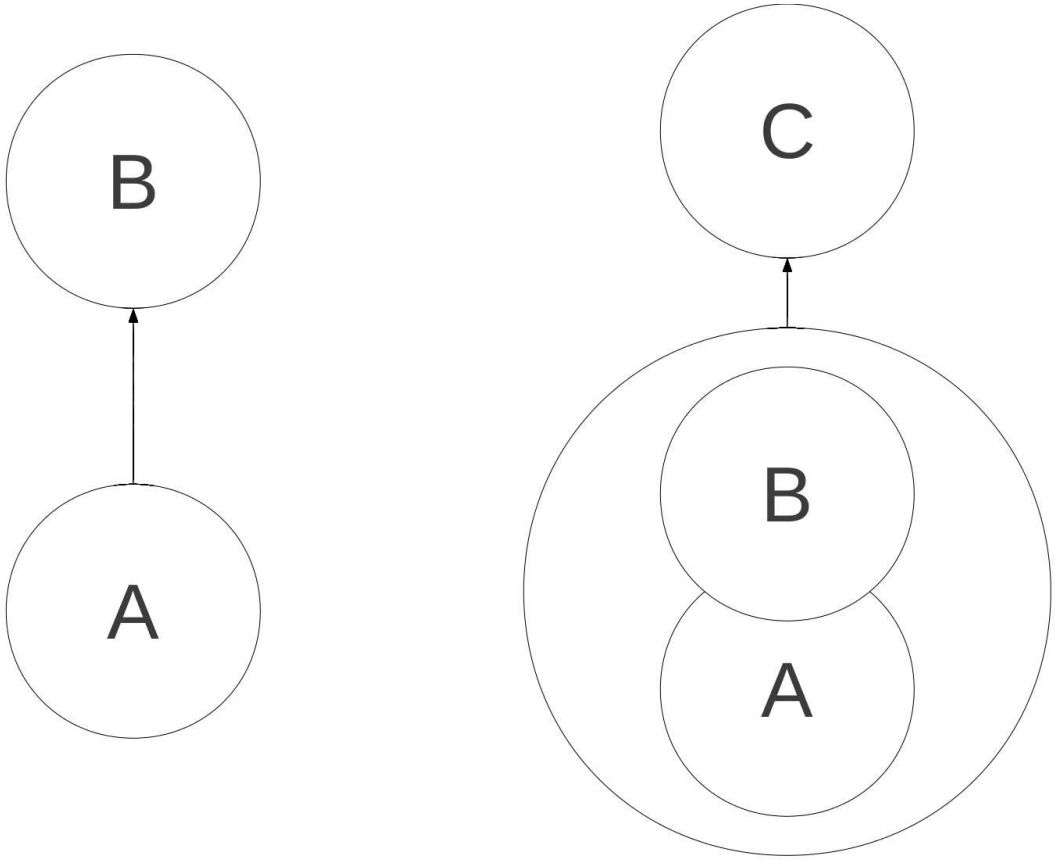

Figure 3. Illustration of the "merge" method.

and it spawns new masters (B). The illustration on the right in Figure 3 shows the two communicators (A and B) merged into an unnamed communicator, which then spawns C.

5.1 Experimental Setup

For the "many communicator" method, four sets of tests were performed—broadcast and gather tests, as well as two types of point-to-point tests. In one type of point-to-point test, a wild card value (`MPI_ANY_SOURCE`) was used as the source argument for the receiving process; in the other type of point-to-point test, a specific value was used as the source argument. All four sets of tests ran through 20 iterations, increasing the number of integers communicated at each iteration. At each iteration, two extraneous sends and receives were needed to handle the timing. This did not substantially affect the accuracy of the timing measurements because the communications being timed sent much more data than the communications needed for taking measurements. For the send and receive tests, the source was chosen at random from one of two communicators, and the destination was chosen at random from the other communicator. For the broadcast and gather tests, the root process was chosen at random. Further, the tests for each message length were performed 1000

times, and the mean communication time was determined. To do this, a separate program was used to execute the timing program 1000 times, and to obtain data such as means and standard deviations, for each message length.

5.1.1 Broadcast and gather

Two communicators were used in all the tests, each containing 32 processes. One was a parent communicator that spawned the other. The tests consisted of 20 iterations of communication, where the message length increased with each iteration. For broadcast, $500 \cdot k$ integers were sent to every process at iteration k, making for a total of $32,000 \cdot k$ integers broadcast at each iteration. For gather, $500 \cdot k$ integers were gathered from each process, making for a total of $32,000 \cdot k$ integers gathered at each iteration. Although $32,000 \cdot k$ integers are communicated during each subroutine call, the message length at iteration k will be considered $500 \cdot k$ (integers per process, with 64 processes involved in each communication). The root was chosen as a random element of the parent communicator.

5.1.2 Send and receive

Two communicators—a parent and a child communicator—were used in all the tests, and each communicator contained four processes. The tests consisted of 20 iterations of communication, where the message length was 5,000,000 times the iteration number. The source was selected as a random number between zero and seven, and the destination was chosen as follows. First, a random number, h, between zero and three was generated. If the value of the source was between zero and three, then the destination was $h + 4$; otherwise, the value of the destination was simply h. Notice that this guaranteed that the source and destination were in distinct communicators.

5.1.3 The "merge" method

For the "merge" method, one simply needs to use `MPI_INTERCOMM_MERGE` to obtain a merged intracommunicator, and then this communicator can be used with existing MPI communication subroutines. Thus, ignoring spawning concerns, the "merge" method is quite simple to implement. Whenever possible, the experimental setup for the "merge" method was the same as the setup for the "many communicator" method. In particular, the number of integers sent per communication at each iteration, the number of timing tests performed at each iteration, and the selection of the source, destination, and root were the same as described above for the "many communicator" method.

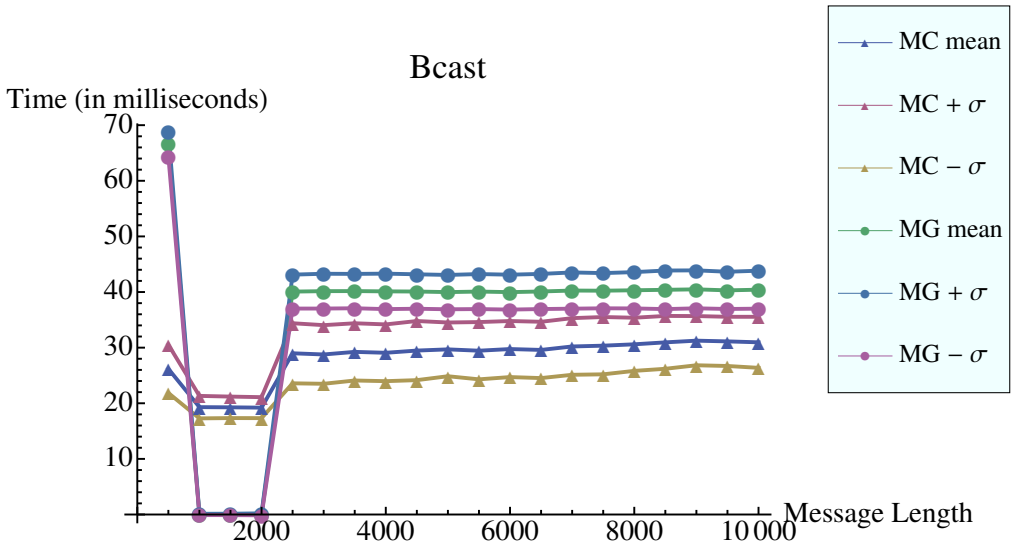

Figure 4. "Many communicator" (triangles) and "merge" method (circles) broadcasts on System G.

5.1.4 Hardware/software

The experiments were conducted on System G, which is the world's largest power-aware compute research cluster. System G has working power-aware features, power and thermal sensors on-board and accessible via software, and high performance processors and interconnects. The cluster consists of 325 Apple MacPro (dual processor quad core Xeon 2.8 GHz) systems with 8GB memory per node and a Mellanox QDR Infiniband interconnect. Users have access to the 30+ thermal sensors and 30+ power sensors in each MacPro. The version of MPI used for the tests was Open MPI 1.4.1.

5.2 Results

In general, the results for the "many communicator" method are illustrated in plots using triangles, and circles are used for the "merge" method. The middle curve for each method represents the mean, and the curves above and below the middle curve show one standard deviation above ($+\sigma$ in the plot legend) and below ($-\sigma$) the mean, respectively. The three curves for each method form a band that likely contains the "true" runtime curve for the method.

5.2.1 Broadcast and gather

For the broadcast and gather tests (Figures 4 and 5), with a few exceptions, the "many communicator" method generally performed better. For message lengths of 1000 to 2000

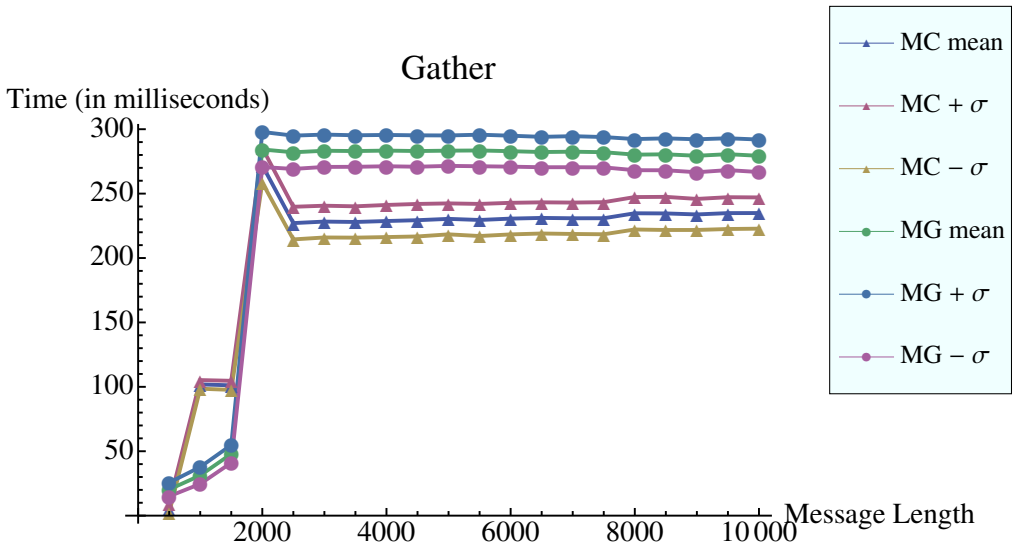

Figure 5. "Many communicator" (triangles) and "merge" method (circles) gathers on System G.

integers per process, the "merge" method outperformed the "many communicator" method for the broadcast tests. The runtimes for both methods were relatively low for these message lengths, producing a "dip" in both bands. For message lengths of 2500 to 10,000, the "many communicator" method performed better than the "merge" method. Notice that there was no overlap in bands whenever the "many communicator" method outperformed the "merge" method.

5.2.2 Send and receive

For the send/receive tests using `MPI_ANY_SOURCE` as the source argument, the "merge" method consistently outperformed the "many communicator" method (see Figure 6). The performance of the "merge" method was also more consistent—the standard deviation for the "merge" method was generally less than 0.5, whereas the standard deviation for the "many communicator" method was between about two and five. It is not clear why the "merge" method performed better than the "many communicator" method. It is possible that the relatively poor performance of the "many communicator" method is due to using `MPI_IPROBE` to determine the relevant communicator. A more efficient method for determining the communicator could yield better runtimes for the "many communicator" method.

The performance of both methods was nearly identical for the send/receive tests using a specific value for the source argument (plot not shown). Since only two communicators were used in the tests, it was quite simple to determine the communicator that contained the

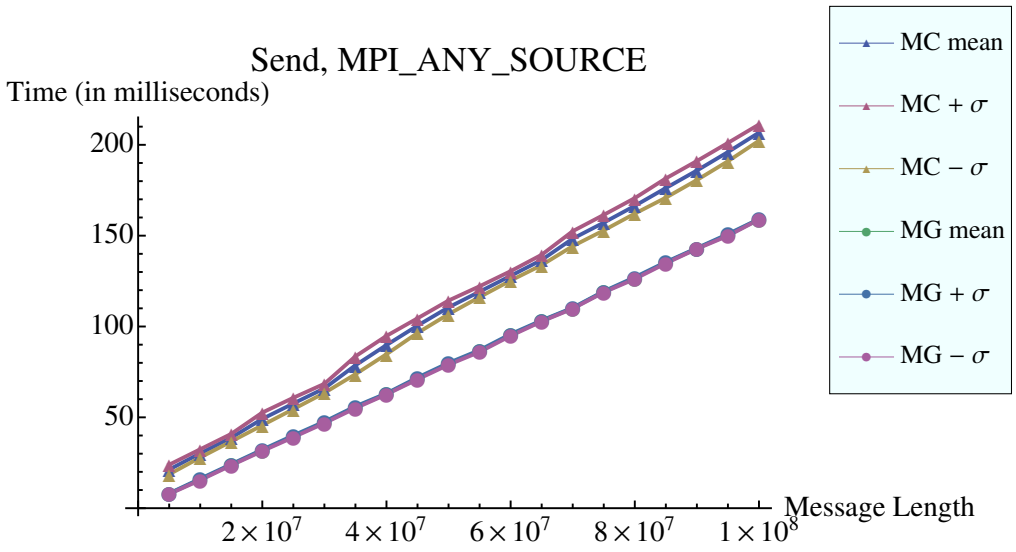

Figure 6. "Many communicator" (triangles) and "merge" method (circles) point-to-point communications with wild card source on System G.

destination, as well as the local rank of the destination process within that communicator. Consequently, most of the runtime was taken up by the transmission time associated with a standard `MPI_SEND`. Presumably, this is why the runtimes for the two methods were so similar. This situation is not unrealistic, as the number of spawning events should generally be small.

5.2.3 Buggy timing code

The stress tests discussed above were not the original tests performed. The initial stress tests produced unreliable results due to a bug in the timing function, which produced a large amount of artificial noise in the runtime data. The timing code allowed certain processes to initiate a send operation long before the receiving process was ready. Since the bug was due to processes "jumping ahead" to later iterations, running each iteration separately by hand produced accurate timing results. This "fix" for the bug was quickly determined, although the source of the bug was not determined until much later.

The tests were performed on a variety of clusters, using multiple compilers and implementations of MPI in order to determine if the observed noise was due to a hardware/software/compiler problem. Of course, the noise was reproduced on all clusters, using any compiler or implementation of MPI. Figures 7–10 show some of the results using various clusters, etc.

Clusters at the National Energy Research Scientific Computing Center (Carver) and the King Abdullah University of Science and Technology (Shaheen and Neser) were used

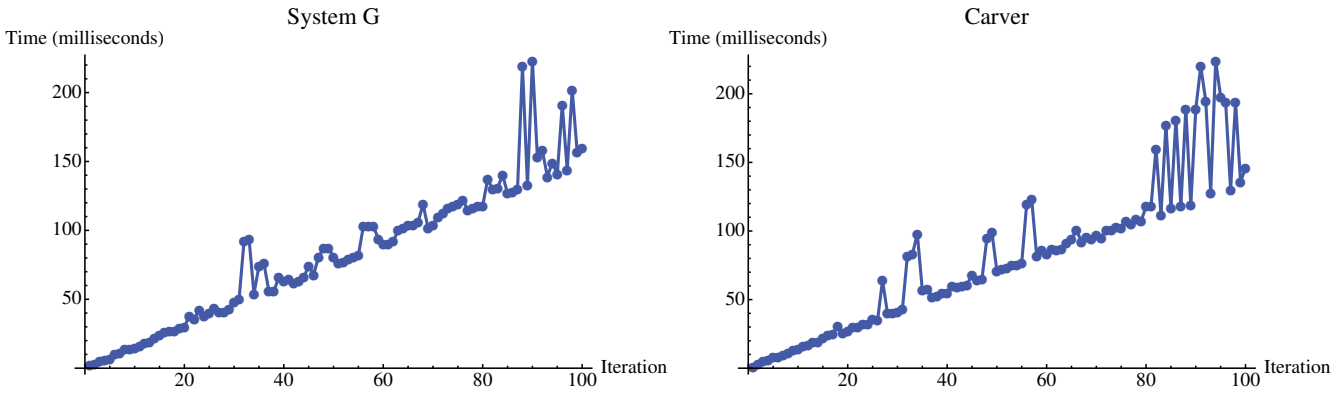

Figure 7. Point-to-point communication times for MPI_SEND on System G (left) and Carver (right).

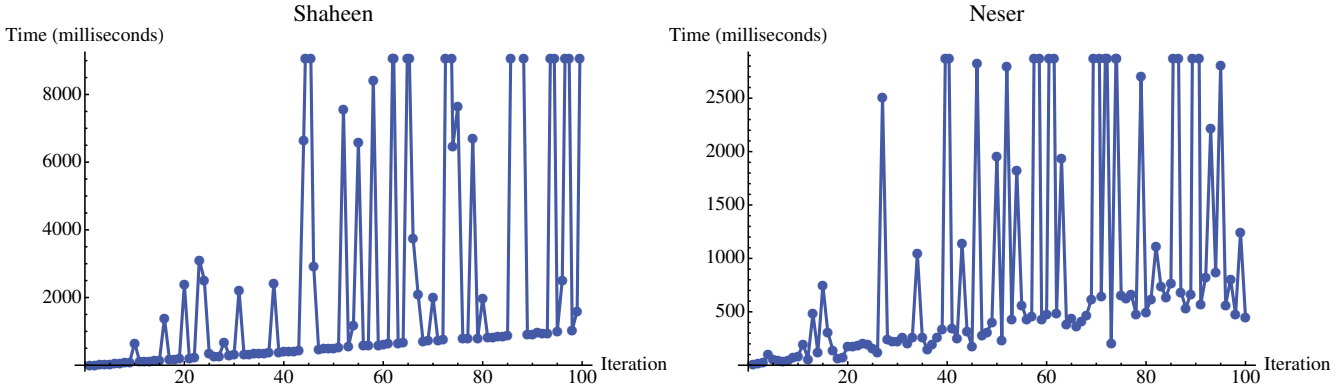

Figure 8. Point-to-point communication times for MPI_SEND on Shaheen (left) and Neser (right).

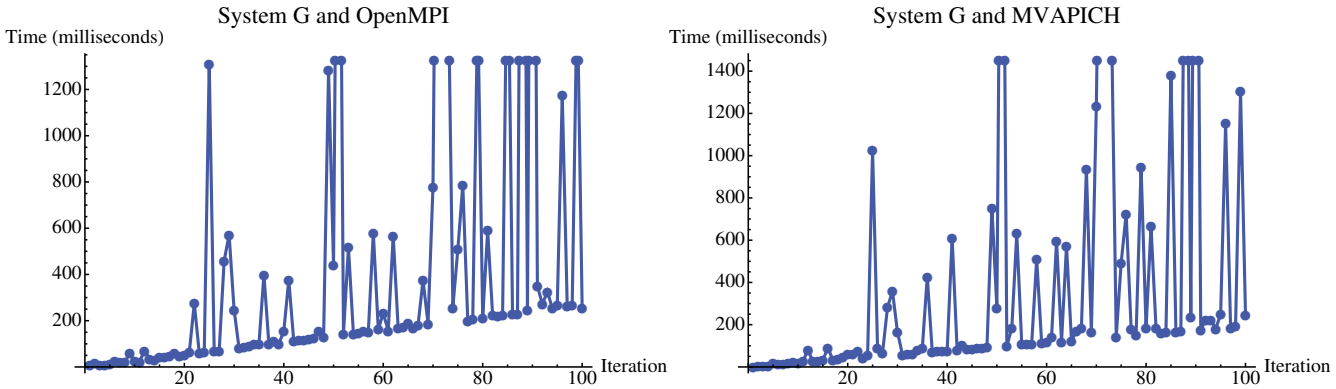

Figure 9. Point-to-point communication times for MPI_SEND on System G using Open MPI (left) and MVAPICH (right).

to confirm that the observed noise was not unique to the cluster used for the initial tests (System G). Carver is a liquid cooled IBM iDataPlex system with 3,200 processor cores

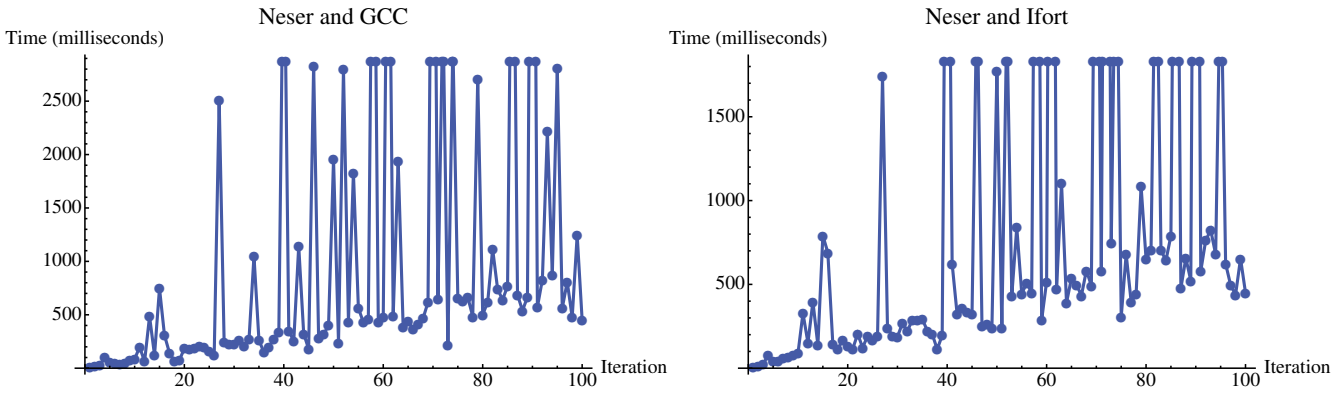

Figure 10. Point-to-point communication times for MPI_SEND on Neser using GCC (left) and IFORT (right).

and 400 compute nodes. The nodes are interconnected by 4X QDR InfiniBand technology, so that 32 Gb/s of point-to-point bandwidth is available for high-performance message passing and I/O. Shaheen is an IBM Blue Gene/P. Each of 16 racks contains 1024 quad-core PowerPC 450 compute nodes running at 850MHz. I/O on the Blue Gene/P is provided via quad-core PowerPC 450 I/O nodes (850MHz, 4GB RAM). Neser is an x86_64 compute cluster consisting primarily of 128 IBM System $\times$ 3550 compute nodes. Each compute node is equipped with 2 quad-core Intel "Harpertown" E5420 CPUs.

Chapter 6: DYNAMIC LOAD BALANCING

The main source of memory use for masters is storing boxes, so "memory load" and "box load" are essentially interchangeable for masters. The box load on masters is monitored to determine when new masters must be spawned. If we define *spawn count* to be the number of spawning events that have occurred at a certain point of the execution of the program, then the memory threshold for masters is roughly $1-1/2^{1+s}$, where s is the current number of spawning events, i.e., new masters are spawned when one half of the currently available memory is used. The threshold for sufficient memory is intentionally low because boxes are not transferred from the current to the spawned masters. Rather, the rate at which boxes are accumulated decreases (resp. increases) temporarily for the current (resp. spawned) masters. Notice that each spawning event doubles the number of masters (and because of this exponential increase in masters, the number of spawning events is limited by a user-defined parameter).

As mentioned above, the box load is strongly imbalanced immediately after new masters are spawned, as the spawned masters have not had time to accumulate boxes. Since spawning new masters creates an imbalance in box load, a dynamic load balancing mechanism is required. To this end, *load deviation* for the ith master is defined as

$$dev_i = \frac{bc_i - bc_a}{bc_t},$$

where bc_t is the total box count across all masters, bc_a is the average box count, and bc_i is the box count for the ith master. Load deviation measures the extent (either positive or negative) to which a master's box count deviates from the average box count. Notice that the sum of load deviations for all masters is zero, and that $|dev_i| \leq 1$.

In order to dynamically balance the box load for all masters, the number of boxes received by a master after the convex hull computation is initially $\lfloor (1-dev_i)T/N \rfloor$, where T is the total number of new boxes obtained from the convex hull computation, and N is the number of masters. After the initial distribution of boxes, remaining boxes are distributed pseudorandomly amongst the masters. This essentially scales a master's share of new boxes by $1 - dev_i$, so that masters with below average box loads will receive more boxes than those with above average loads. In Chapter 7.3, it is shown that this method is effective in dynamically balancing box load after new masters are spawned.

Chapter 7: PERFORMANCE RESULTS

A variety of optimization problems were used to test the modifications made to `pVTdirect`. Several toy problems, as well as a realistic nuclear physics problem (MFDn), were used. Since both `pVTdirect` and `spVTdirect` consider the objective function to be a black box, the runtime per iteration for both `pVTdirect` and `spVTdirect` should only depend on the runtime of the objective function. Consequently, it is useful to test the performance of `spVTdirect` using objective functions with a variety of runtimes and runtime patterns (i.e., the runtime per iteration might increase with iterations, or stay roughly constant). Four of the toy problems have negligible runtimes (on the order of 10^{-4} seconds), one toy problem (GRSleep) has runtimes around one second, and the real-world physics problem has (parallel) runtimes ranging from about eight to fifteen seconds.

A single spawning event was artificially made to occur at the seventh iteration of `spVTdirect`. For the toy problems, `pVTdirect` was run with either nine or twelve processes, and `spVTdirect` was run with eight or ten processes (chosen to meet the restraints set by `pVTdirect` on the ratio of masters to workers). `spVTdirect` spawned either one or two new masters, so that the numbers of processes used by `pVTdirect` and `spVTdirect` were identical after the spawning event. For the MFDn objective function, `pVTdirect` was run with six processes, and `spVTdirect` was run with five processes. `spVTdirect` spawned one new master, so that `pVTdirect` and `spVTdirect` were both running with six processes after the spawning event. Every instance of MFDn was run with six processes.

7.1 Toy Problems

The four toy problems with negligible runtimes were the Griewank function (GR), the Quartic function (QU), Schwefel's function (SC), and Michalewicz's function (MI), all taken from [3]. Figures 11 and 12 show runtimes per iteration for `pVTdirect` and `spVTdirect` with GR, QU, SC, and MI as objective functions. The runtimes for `spVTdirect` are shown with triangles, and those for `pVTdirect` are shown with circles. Notice that the runtimes per iteration for `spVTdirect` are quite similar to the runtimes per iteration for `pVTdirect`. Predictably, the runtime for `spVTdirect` is much larger for the seventh iteration (when the spawning event occurred). The runtime per iteration for `spVTdirect` generally stabilizes to values that consistently hover slightly above the values for `pVTdirect`. See the plot of runtimes per iteration for MI (Figure 12, right) for a nice illustration of this effect.

The toy problems with negligible runtimes were convenient for comparing the *total* number of iterations and function evaluations for `spVTdirect` and `pVTdirect`, as well as comparing other global properties. The total number of iterations and objective function evaluations, the minimum box diameter, and the stopping rule used to end the search were always identical for `spVTdirect` and `pVTdirect`. Although there are very minor differences

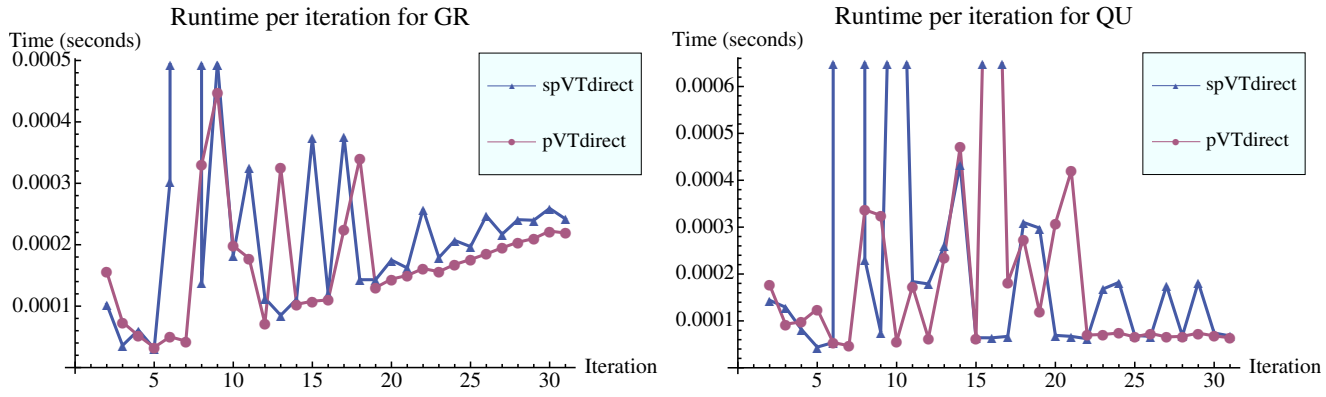

Figure 11. Comparison of runtimes per iteration for the GR (left) and QU (right) objective functions.

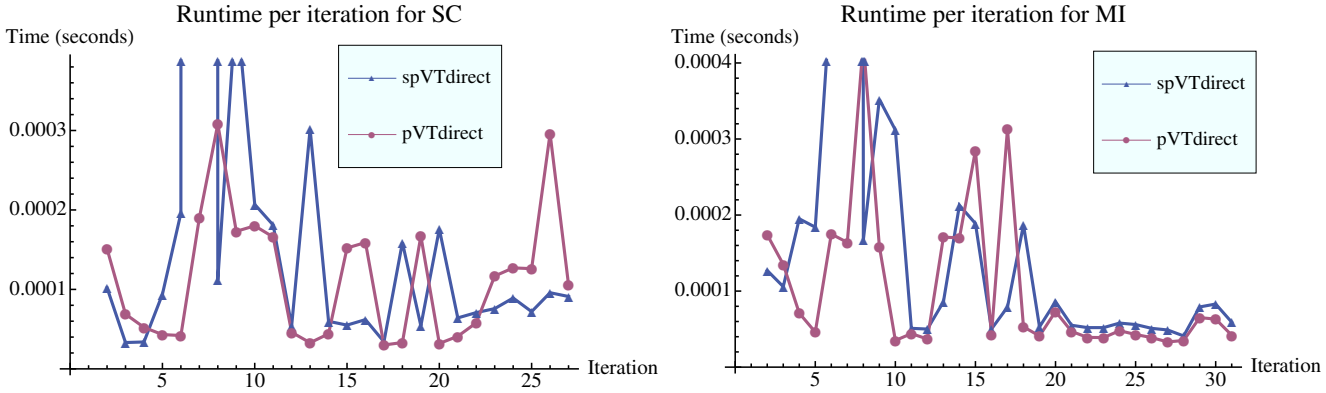

Figure 12. Comparison of runtimes per iteration for the SC (left) and MI (right) objective functions.

in performance, the boxes examined at every iteration are identical for `pVTdirect` and `spVTdirect`.

In Figure 13, the runtimes per iteration are plotted for `pVTdirect` and `spVTdirect` with the objective function MISleep—a variant of the MI toy problem discussed above—that is designed to run for about one second. The runtimes for `spVTdirect` and `pVTdirect` are shown with triangles and circles, respectively. From the plot in Figure 13, it is clear that the runtimes per iteration for `pVTdirect` and `spVTdirect` with the MISleep objective function are nearly identical, with the exception of the seventh iteration for `spVTdirect` (where the spawning event occurs). The same basic pattern is seen in other similar variants of the toy problems (GRSleep, etc.). One can conclude that the overhead for `spVTdirect` has negligible impact on the runtime per iteration when the objective function has a sufficiently long runtime (according to the tests done for this work, a runtime of at least one second is sufficiently long).

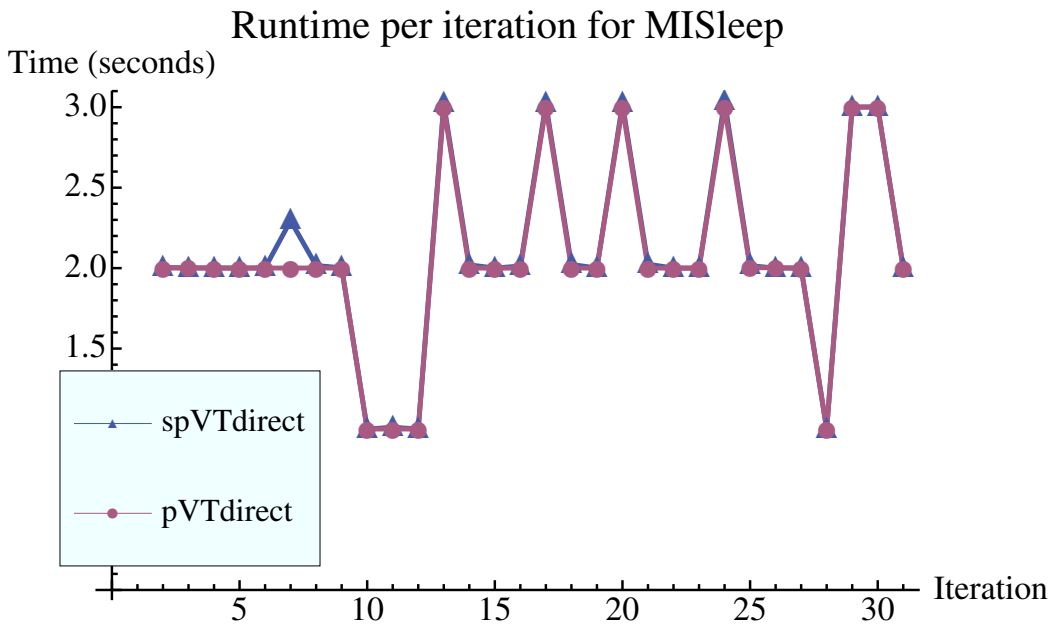

Figure 13. Comparison of runtimes per iteration for the MISleep objective function.

7.2 MFDn

MFDn, which stands for "many fermion dynamics nuclear", is a nuclear physics code [9] developed at Iowa State University that computes theoretical values for certain observables relevant to the spectra of atomic nuclei. The computed values for these observables can be compared to empirical values using a χ^2 function, and a value is obtained representing the goodness of fit. Since MFDn has an input file containing several input parameters, one can vary these parameters, and observe the goodness of fit obtained by each setting of input parameters. This suggests the use of an optimization algorithm in order to find an optimal set of parameters, where an "optimal" set of parameters means a set of parameters that yields a minimal χ^2 value. For the problem considered here, there are only three input parameters that vary. Also, the output of the objective function is not simply the χ^2 value for the (sequences) of computed and empirical values. Instead, MFDn is run twice with two separate input files. The input files are identical with the exception of a single parameter, which is not amongst the three that are varied. The output of the objective function is the sum of the two χ^2 values for the two runs of MFDn.

The computation of the MFDn objective function is very complex and involves finding a solution to the Schrödinger equation with a large, sparse, and irregularly structured Hamiltonian matrix [10]. One reason for this complexity is that MFDn is itself a parallel computation, and so it must be spawned using `MPI_COMM_SPAWN`. Another reason is that two instances of MFDn need to be run in order to determine the output of the objective function (each instance of MFDn uses a different input data set). A third reason is that

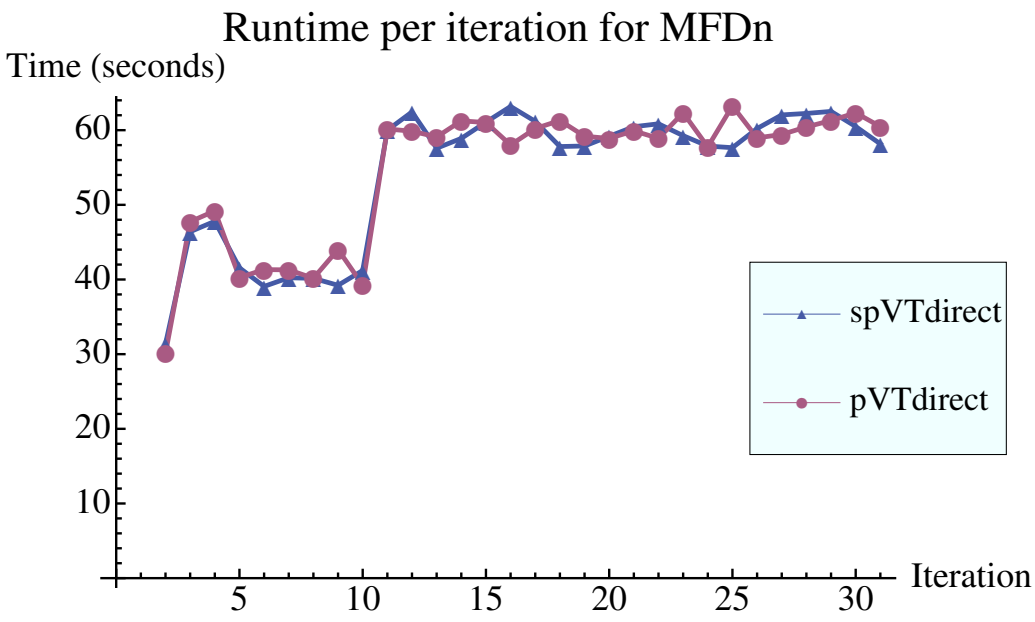

Figure 14. Comparison of runtimes per iteration for the MFDn objective function.

multiple processes may execute `MPI_COMM_SPAWN` simultaneously, and hence (on shared file systems) multiple processes may attempt to access the executable simultaneously, causing the program to crash. Notice that even though MFDn runs on separate processes from the worker that spawned it, the worker's call to the objective function does not complete until both instances of MFDn complete, because the objective function waits for a completion message from MFDn. This means that the computation of the MFDn objective function takes on the order of eight to fifteen seconds to complete. The runtime of this objective function is not entirely consistent because the computation is done in parallel.

MPI does not provide any means for locking executables to prevent race conditions when calling `MPI_COMM_SPAWN`, so it is up to the user to prevent such conditions. Fortunately, MPI *does* provide support for locking files when reading or writing to them. So, one way to prevent race conditions when using `MPI_COMM_SPAWN` is to have each process read a value from a file, where the value indicates which process currently "owns" the executable. A process continually reads from the file until it reads "f" (for "free"), in which case the process writes its own rank to the file, executes `MPI_COMM_SPAWN`, and then writes "f" to the file once both instances of MFDn have been spawned.

Figure 14 shows runtimes per iteration for `pVTdirect` and `spVTdirect`, both with MFDn as objective function. The triangles and circles show the runtimes per iteration for `spVTdirect` and `pVTdirect`, respectively. The runtimes per iteration for `spVTdirect` with objective function MFDn are roughly the same as runtimes per iteration for `pVTdirect`, ignoring the variations in runtime for both `pVTdirect` and `spVTdirect`. The overhead in `spVTdirect` does not have a significant impact on its performance.

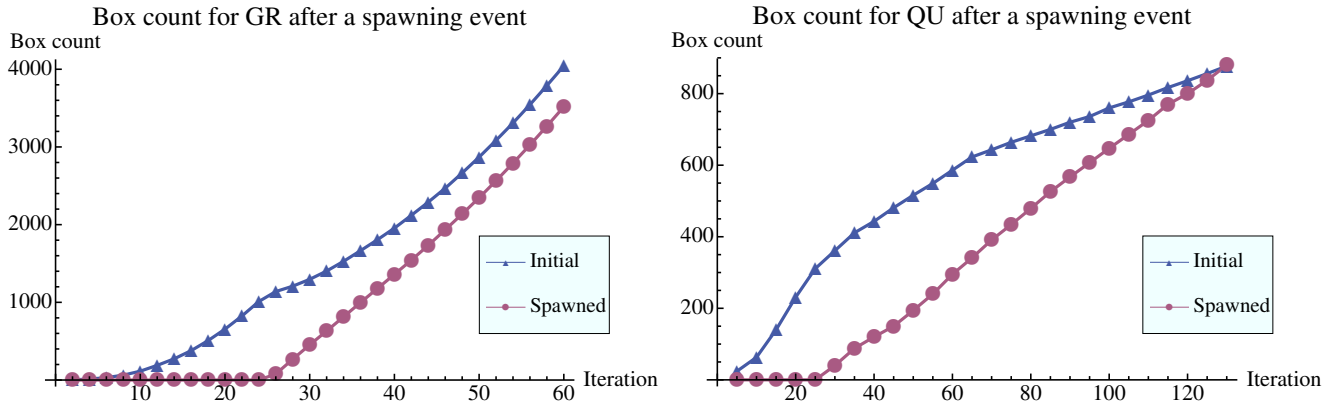

Figure 15. Box count versus iteration for GR (left) and QU (right).

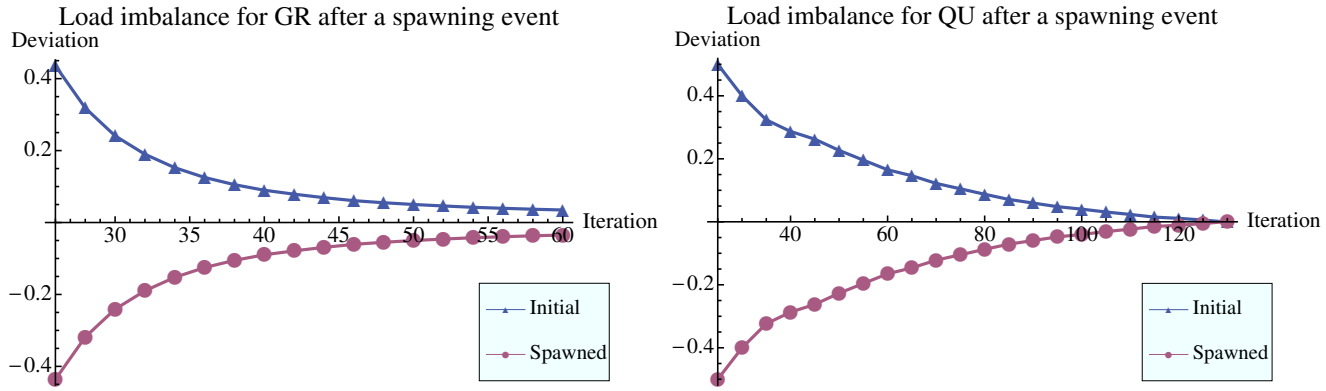

Figure 16. Load deviation versus iteration for GR (left) and QU (right).

7.3 Performance of the Dynamic Load Balancing Mechanism

In order to test the dynamic load balancing mechanism, `spVTdirect` was run with one master using GR and QU as objective functions. The amount of memory available on a node is read from an input file. The value for available memory was set artificially low so that a spawning event would occur at iteration 25, increasing the number of masters to two. Box count and load deviation were monitored before and after the spawning event. Before the spawning event, the spawned master did not exist, and hence its box count is assumed to be zero. After the spawning event, the box count for the spawned master increased until it was almost identical to the box count for the current master (Figure 15).

The tests for load deviation began at iteration 25, when the new master was spawned. Load deviation was initially 0.5 for the spawning master, and -0.5 for the spawned master for both objective functions. For both GR and QU, the load deviation moved toward zero for both masters; however, load deviation approached zero more quickly (and smoothly) for GR (Figure 16). This was due to the fact that the number of new boxes added per iteration was

greater for GR than for QU, and that (for technical reasons) the lead master must receive at least one box per iteration. In general, at most three new boxes were added per iteration for QU, and since the lead master received at least one box, the spawned master could receive at most two boxes, regardless of the load deviation. This observation inspired a possible modification to the load balancing mechanism that increases the number of boxes added per iteration, thereby balancing the box load in fewer iterations. An input parameter for `pVTdirect` and `spVTdirect`, called the "aggressive" switch, specifies that all boxes on the convex hull should be selected, not just those meeting the minimum improvement condition. For 20 iterations after a spawning event, the aggressive switch is turned on. This number of iterations was selected based on the observation that 20 iterations was generally sufficient to reduce load deviation to about 0.1. When the temporary aggressive switch is used (Figure 17), it takes far fewer iterations for load deviation to reach 0.1 (about 7 iterations, versus 50 iterations when the "aggressive" switch is not used at all). A serious problem with turning the "aggressive" switch on temporarily is that it can destroy determinism—if `spVTdirect` is run on two machines, one with more memory than the other, then `spVTdirect` may spawn new masters on one machine and not the other. This can lead to different minimum values being found at the end of the two runs on the different machines. In order to avoid this loss of determinism, the "aggressive" switch is not modified after spawning.

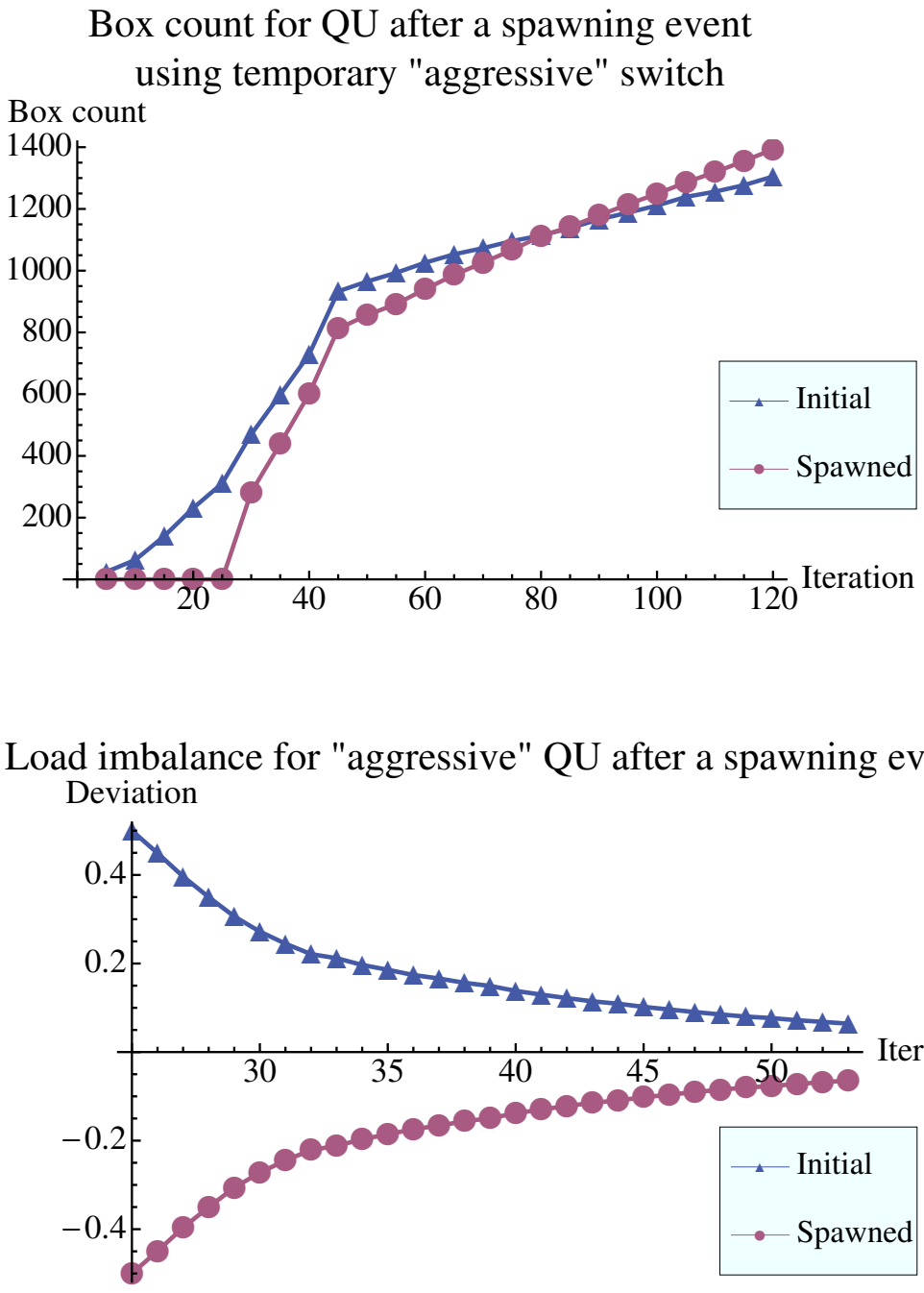

Figure 17. Box count (top) and deviation (bottom) versus iteration for QU using temporary "aggressive" switch.

Chapter 8: CONCLUSIONS

This work shows that it is possible to dynamically adjust the number of masters in the global optimization code `pVTdirect`, and hence prevent (or at least substantially delay) thrashing when the amount of available memory becomes insufficient. Performance results show that the extra communication overhead in `spVTdirect` has a negligible impact on the performance of the application.

There were a number of lessons learned during the course of this work that should be useful to anyone designing a master-worker style parallel code with the capability to adjust process count on demand. Updating state after new processes are spawned can be quite subtle if some of the processes are tightly coupled and/or there are persistent aspects of program state, i.e., aspects of state that are persistent across iterations of a main loop. One way to deal with the problem of updating/fabricating state is to design the code so that processes are only loosely coupled, i.e., the state of a given process has minimal effect on the state of any other process. If the processes *must* be tightly coupled, then a reasonable design choice is to prevent state from being persistent across iterations. If processes are unaware of any state from the previous iteration, then integrating spawned processes into the computation should be simple. Another useful design choice is to regularly synchronize all masters. This should simplify the task of notifying all processes of a spawn request, assuming all processes are involved in the spawning procedure. Of course, all processes need not be involved, but the present work has shown that this can simplify communication between the current and spawned processes. Synchronizing the workers may also be beneficial, at the risk of creating an unreasonable amount of idle time for the workers.

One final point is that the number of workers could also be dynamically adjusted on demand. This would require only minor modifications to `spVTdirect`—the spawn notification method used for masters could be used for spawning new workers, and the state update would be simpler than for spawning masters. Adjusting the number of workers on demand would be useful in many situations. For instance, the user could supply parameters specifying that some minimal amount of progress has to be made by the search in a fixed amount of time. If sufficient progress is not made, then more workers could be spawned on demand to perform more function evaluations, and hopefully speed up the progress of the search.

References

[1] Balaji, P.; Buntinas, D.; Goodell, D.; Gropp, W.; Kumar, S.; Lusk, E.; Thakur, R.; Larsson, J. (2009), "MPI on a million processors," in *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag 20–30, Berlin, Heidelberg*.

[2] Jones, D. R.; Perttunen, C.D.; Stuckman, B. E. (1993), "Lipschitzian optimization without the Lipschitz constant," *J. Optim. Theory Appl.*, **79**, 1, 157–181.

[3] He, J.; Watson, L. T.; Sosonkina, M. (2009), "Algorithm 897: VTDIRECT95: serial and parallel codes for the global optimization algorithm DIRECT," *ACM Transactions on Mathematical Software*, **26**, 3, Article 17, 24 pp..

[4] He, J.; Watson, L. T.; Ramakrishnan, N.; Shaffer, C. A.; Verstak, A.; Jiang, J.; Bae, K.; Tranter, W. H. (2002), "Dynamic data structures for a direct search algorithm," *Comput. Optim. Appl.*, **23**, 1, 5–25.

[5] He, J; Verstak, A.; Watson, L. T.; Sosonkina, M. (2009), "Performance modeling and analysis of a massively parallel DIRECT - part 1," *Int. J. High Perform. Comput. Appl.*, **23**, 1, 14–28.

[6] He, J.; Verstak, A.; Sosonkina, M.; Watson, L. T. (2009), "Performance modeling and analysis of a massively parallel DIRECT - part 2," *Int. J. High Perform. Comput. Appl.*, **23**, 1, 29–41.

[7] Panning, T. D.; Watson, L. T.; Allen, N. A.; Chen, K. C.; Shaffer, C. A.; Tyson, J. J. (2008), "Deterministic parallel global parameter estimation for a model of the budding yeast cell cycle," *J. of Global Optimization*, **40**, 4, 719-738.

[8] Squyres, J., "The spawn of MPI", cw.squyres.com Feb. 2005, ClusterWorld magazine, Nov. 2011 <http://cw.squyres.com/columns/2005-02-CW-MPI-Mechanic.pdf>.

[9] Vary, J. P., "The Many-Fermion-Dynamics Shell-Model code," Iowa State University, 1994, Unpublished.

[10] Sternberg, P.; Ng, E. G.; Yang, C.; Maris, P.; Vary, J. P. ; Sosonkina, M.; Le, H. V. (2008), "Accelerating configuration interaction calculations for nuclear structure," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08), IEEE Press Article 15 (12 pages), Piscataway, NJ, USA*.

[11] Scherer, A.; Lu, H; Gross, T.; Zwaenepoel, W. (1999), "Transparent adaptive parallelism on NOWs using OpenMP," in *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '99), ACM 96–106, New York, NY*.

[12] Godard, E.; Setia, S.; White, E. L. (2000), "DyRecT: software support for adaptive parallelism on NOWs," in *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing (IPDPS '00), Springer-Verlag 1168–1175, London, UK*.

[13] Huang, C; Lawlor, O.; Kalé, L. V. (2003), "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958 306–322, College Station, Texas*.

[14] Snir, M.; Otto, S.; Huss-Lederman, S.; Walker, D.; Dongarra, J. (2000), *MPI—the Complete Reference, Vol. 1: the MPI Core, 2nd ed*, MIT Press: Cambridge, MA.

[15] Gropp, W.; Huss-Lederman, S.; Lumsdaine, A.; Lusk, E.; Nitzberg, B.; Saphir, W.; Snir, M. (2000), *MPI—the Complete Reference, Vol. 2: the MPI Extension, 2nd ed*, MIT Press: Cambridge, MA.

APPENDIX A: LIST OF FIGURES

APPENDIX B: spVTdirect.f 5

```
! This file (spVTdirect.f95) contains the module spVTdirect_MOD that
! defines data types, subroutines, and functions used in the parallel
! VTDIRECT implementation.
!
MODULE spVTdirect_MOD
USE VTDIRECT_GLOBAL  ! Module (shared_modules.f95) for data types,
  ! parameters, global variables, interfaces, and internal subroutines.
USE VTDIRECT_COMMSUB ! Module (shared_modules.f95) for subroutines
  ! used by VTdirect and spVTdirect in common.
USE VTDIRECT_CHKPT   ! Module (shared_modules.f95) for data types,
  ! subroutines, functions, global variables used by checkpointing.
USE OMP_LIB         ! OpenMP library.
USE MPI            ! MPI library functions.
!
!OptResults: Contains the search results for a subdomain.
!  fmin     - Minimum function value.
!  x        - The point coordinates associated with 'fmin'.
!  max_iter  - Number of iterations.
!  max_evl   - Number of evaluations.
!  min_dia   - Minimum diameter.
!  status    - Search return status.
!
TYPE OptResults
  REAL(KIND = R8) :: fmin
  REAL(KIND = R8), DIMENSION(:), POINTER :: x
  INTEGER :: max_iter
  INTEGER :: max_evl
  REAL(KIND = R8) :: min_dia
  INTEGER :: status
END TYPE OptResults

! MPI message types.

! A nonblocking function evaluation request.
INTEGER, PARAMETER :: NONBLOCK_REQ = 1
! A blocking function evaluation request.
INTEGER, PARAMETER :: BLOCK_REQ = 2
! Point(s) responding to a nonblocking request.
INTEGER, PARAMETER :: POINT_NBREQ = 3
! Point(s) responding to a blocking request.
INTEGER, PARAMETER :: POINT_BREQ = 4
! No more point for evaluation.
INTEGER, PARAMETER :: NO_POINT = 5
! Returned point function value(s).
INTEGER, PARAMETER :: FUNCVAL = 6
! All search work is done.
INTEGER, PARAMETER :: MYALL_DONE = 7
! Updating the global counter for the blocked workers.
INTEGER, PARAMETER :: COUNT_DONE = 8
! Search results.
INTEGER, PARAMETER :: RESULT_DATA = 9
! Converting a master to a worker.
INTEGER, PARAMETER :: BE_WORKER = 11
! Termination.
INTEGER, PARAMETER :: TERMINATE = 13
! Updating intermediate search results.
INTEGER, PARAMETER :: UPDATES = 14
! Transferring state to spawned masters.
INTEGER, PARAMETER :: STATE_TRANSFER_TAG = 15
! Creating intracommunicator from parent and child intercommunicators.
INTEGER, PARAMETER :: INTERCOMM_CREATE_TAG = 16
! Request to spawn more masters.
INTEGER, PARAMETER :: SPAWN_REQ = 17
! Handle for the spawning host file.
INTEGER, PARAMETER :: QSPAWN_UNIT = 18
! MPI error.
INTEGER, PARAMETER :: DMPI_ERROR = 40

CONTAINS

SUBROUTINE spVTdirect_finalize()
! MPI finalization.
!
! On input: None.
!
! On output: None.
!
! Local variable.
```

```
INTEGER :: ierr

CALL MPI_FINALIZE(ierr)
RETURN
END SUBROUTINE spVTdirect_finalize

SUBROUTINE spVTdirect_init(iflag)
! MPI initialization.
!
! On input: None.
!
! On output:
! iflag - Initialization status.
!
INTEGER, INTENT(OUT) :: iflag

iflag = 0
CALL MPI_INIT(iflag)
IF (iflag /= MPI_SUCCESS) THEN
  iflag = DMPI_ERROR
END IF
RETURN
END SUBROUTINE spVTdirect_init

SUBROUTINE spVTdirect(N, L, U, OBJ_FUNC, X, FMIN, PROCID, STATUS, &
                 SWITCH, MAX_ITER, MAX_EVL, MIN_DIA, OBJ_CONV, EPS, &
                 MIN_SEP, W, BOX_SET, NUM_BOX, N_SUB, N_MASTER, &
                 BINSIZE, RESTART, TOTAL_MEM)
IMPLICIT NONE
! This is a parallel implementation of the DIRECT global
! unconstrained optimization algorithm described in:
!
!   D.R. Jones, C.D. Perttunen, and B.E. Stuckman, Lipschitzian
!   optimization without the Lipschitz constant, Journal of Optimization
!   Theory and Application, Vol. 79, No. 1, 1993, pp. 157-181.
!
! The algorithm to minimize f(x) inside the box L <= x <= U is as follows:
!
!   1. Normalize the search space to be the unit hypercube. Let c_1 be
!      the center point of this hypercube and evaluate f(c_1).
!   2. Identify the set S of potentially optimal rectangles.
!   3. For all rectangles j in S:
!      3a. Identify the set I of dimensions with the maximum side length.
!          Let delta equal one-third of this maximum side length.
!      3b. Sample the function at the points c +- delta*e_i for all i
!          in I, where c is the center of the rectangle and e_i is the ith
!          unit vector.
!      3c. Divide the rectangle containing c into thirds along the
!          dimensions in I, starting with the dimension with the lowest
!          value of f(c +- delta*e_i) and continuing to the dimension
!          with the highest f(c +- delta*e_i).
!   4. Repeat 2.-3. until stopping criterion is met.
!
! On input:
!
! N is the dimension of L, U, and X.
!
! L(1:N) is a real array giving lower bounds on X.
!
! U(1:N) is a real array giving upper bounds on X.
!
! OBJ_FUNC is the name of the real function procedure defining the
!   objective function f(x) to be minimized. OBJ_FUNC(C,IFLAG) returns
!   the value f(C) with IFLAG=0, or IFLAG/=0 if f(C) is not defined.
!   OBJ_FUNC is precisely defined in the INTERFACE block below.
!
! Optional arguments:
!
! SWITCH =
!   1   select potentially optimal boxes on the convex hull of the
!       (box diameter, function value) points (default).
!   0   select as potentially optimal the box with the smallest function
!       value for each diameter that is above the roundoff level.
!       This is an aggressive selection procedure that generates many more
!       boxes to subdivide.
!
! MAX_ITER is the maximum number of iterations (repetitions of Steps 2-3)
!   allowed; defines stopping rule 1. If MAX_ITER is present but <= 0
!   on input, there is no iteration limit and the number of iterations
!   executed is returned in MAX_ITER.
!
! MAX_EVL is the maximum number of function evaluations allowed; defines
!   stopping rule 2. If MAX_EVL is present but <= 0 on input, there is no
!   limit on the number of function evaluations, which is returned in
!   MAX_EVL.
!
```

```
! MIN_DIA is the minimum box diameter allowed; defines stopping rule 3.
!    If MIN_DIA is present but <= 0 on input, a minimum diameter below
!    the roundoff level is not permitted, and the box diameter of the
!    box containing the smallest function value FMIN is returned in
!    MIN_DIA.
!
! OBJ_CONV is the smallest acceptable relative improvement in the minimum
!    objective function value 'FMIN' between iterations; defines
!    stopping rule 4. OBJ_CONV must be positive and greater than the round
!    off level.  If absent, it is taken as zero.
!
! EPS is the tolerance defining the minimum acceptable potential
!    improvement in a potentially optimal box.  Larger EPS values
!    eliminate more boxes from consideration as potentially optimal,
!    and bias the search toward exploration.  EPS must be positive and
!    greater than the roundoff level.  If absent, it is taken as
!    zero.  EPS > 0 is incompatible with SWITCH = 0.
!
! MIN_SEP is the specified minimal (weighted) distance between the
!    center points of the boxes returned in the optional array BOX_SET.
!    If absent or invalid, MIN_SEP is taken as 1/2 the (weighted) diameter
!    of the box [L, U].
!
! W(1:N) is a positive real array.  It is used in computing the distance
!    between two points X and Y as SQRT(SUM( (X-Y)*W*(X-Y) )) or scaling the
!    dimensions W*(U-L) for domain decomposition.  If absent, W is taken as
!    all ones.
!
! BOX_SET is an empty array (TYPE HyperBox) allocated to hold the desired
!    number of boxes.
!
! N_SUB is the specified number of subdomains that run the DIRECT search
!    in parallel. If absent or invalid (out of the range [1,32]), it is
!    taken as 1 (default).
!
! N_MASTER is the specified number of masters per subdomain. If absent or
!    invalid, it is taken as 1 (default).
!
! BINSIZE is the number of function evaluations per task to be sent at one
!    time. If absent, it is taken as 1 (default).
!
! RESTART
!    0, checkpointing is off (default).
!    1, function evaluations are logged to a file 'pvtdirchkpt.dat*' tagged
!       with the subdomain ID and the master ID.
!    2, the program restarts from checkpoint files on all masters and new
!       function evaluation logs will be appended to the end of the files.
!       When 'N_MASTER' has changed for the recovery run, at every
!       iteration, each master reads all the checkpoint logs for that
!       iteration from all checkpoint files. A new set of checkpoint files
!       is created to record the function evaluation logs.
!
! TOTAL_MEM is an integer specifying the total amount of memory (in MB) on any
!    node involved in the computation. This implicitly assumes that the host
!    cluster uses a homogeneous memory scheme. If absent, new masters are
!    never spawned.
!
! On output:
!
! X(1:N) is a real vector containing the sampled box center with the
!    minimum objective function value FMIN.
!
! FMIN is the minimum function value.
!
! PROCID is the assigned processor ID that is used for printing out the
!    final results.
!
! STATUS is an array of return status flags for 'N_SUB' subdomains. For
!    the final search result, the units decimal digit of the ith element
!    of the array represents the return status of subdomain i; the units
!    decimal digit tens decimal digit indicates a successful return, or an
!    error condition with the cause of the error condition reported in the
!    units digit. During the search process, the first array element is used
!    for intermediate status on each processor.
!
! Tens digit =
!  0 Normal return.
!    Units digit =
!     1   Stopping rule 1 (iteration limit) satisfied.
!     2   Stopping rule 2 (function evaluation limit) satisfied.
!     3   Stopping rule 3 (minimum diameter reached) satisfied. The
!         minimum diameter corresponds to the box for which X and
!         FMIN are returned.
!     4   Stopping rule 4 (relative change in 'FMIN') satisfied.
!  1 Input data error.
!    Units digit =
```

```
!    0   N < 2.
!    1   Assumed shape array L, U, or X does not have size N.
!    2   Some lower bound is >= the corresponding upper bound.
!    3   MIN_DIA, OBJ_CONV, or EPS is invalid or below the roundoff level.
!    4   None of MAX_EVL, MAX_ITER, MIN_DIA, and OBJ_CONV are specified;
!        there is no stopping rule.
!    5   Invalid SWITCH value.
!    6   SWITCH = 0 and EPS > 0 are incompatible.
!    7   RESTART has an invalid value.
!    8   Problem with the parallel scheme: the requested number of
!        processors is too small, or the worker to master ratio is less
!        than 2, or BOX_SET is specified for the multiple masters case.
!    9   BINSIZE has an invalid value.
!  2 Memory allocation error or failure.
!    Units digit =
!    0   BoxMatrix type allocation.
!    1   BoxLink or BoxLine type allocation.
!    2   int_vector or real_vector type allocation.
!    3   HyperBox type allocation.
!    4   BOX_SET is allocated with a wrong problem dimension.
!  3 Checkpoint file error.
!    Units digit =
!    0   Open error. If RESTART==1, an old checkpoint file may be present
!        and needs to be removed. If RESTART==2, the file may not exist.
!    1   Read error.
!    2   Write error.
!    3   The file header does not match with the current setting.
!    4   A log is missing in the file for recovery.
!  4 MPI error.
!    Units digit =
!    0   Initialization error.
!    1   MPI_COMM_RANK error.
!    2   MPI_COMM_GROUP error.
!    3   MPI_COMM_SIZE error.
!    4   MPI_GROUP_INCL error.
!    5   MPI_COMM_CREATE error.
!    6   MPI_GROUP_RANK error.
!    7   At least one processor aborts.
!
!  For example,
!    03 indicates a normal return (tens digit = 0) with "stopping rule 3
!       satisfied" (units digit = 3), and
!    12 indicates an input error (tens digit = 1) when "some lower bound
!       is >= the corresponding upper bound" (units digit = 2).
!
! Optional arguments:
!
! MAX_ITER (if present) contains the number of iterations.
!
! MAX_EVL (if present) contains the number of function evaluations.
!
! MIN_DIA (if present) contains the diameter of the box associated with
!       X and FMIN.
!
! MIN_SEP (if present) is unchanged if it was a reasonable value on
!       input. Otherwise, it is reset to the default value.
!
! W (if present) is unchanged if it was positive on input. Any
!   non-positive component is reset to one.
!
! BOX_SET (if present) is an array of TYPE (HyperBox) containing the
!        best boxes with centers separated by at least MIN_SEP.
!        The number of returned boxes NUM_BOX <= SIZE(BOX_SET) is as
!        large as possible given the requested separation.
!
! NUM_BOX (if present) is the number of boxes returned in the array
!       BOX_SET(1:).
!
! N_SUB (if present) is the actual number of subdomains.
!
! N_MASTER (if present) is the actual number of masters.
!
INTEGER, INTENT(IN) :: N
REAL(KIND = R8), DIMENSION(:), INTENT(INOUT) :: L
REAL(KIND = R8), DIMENSION(:), INTENT(INOUT) :: U
INTERFACE
  FUNCTION OBJ_FUNC(C, IFLAG) RESULT(F)
    USE REAL_PRECISION, ONLY : R8
    REAL(KIND = R8), DIMENSION(:), INTENT(IN) :: C
    INTEGER, INTENT(OUT) :: IFLAG
    REAL(KIND = R8) :: F
  END FUNCTION OBJ_FUNC
END INTERFACE
REAL(KIND = R8), DIMENSION(:), INTENT(OUT) :: X
REAL(KIND = R8), INTENT(OUT) :: FMIN
INTEGER, INTENT(OUT) :: PROCID
```

```
INTEGER, DIMENSION(:), INTENT(OUT) :: STATUS
INTEGER, INTENT(IN), OPTIONAL :: SWITCH
INTEGER, INTENT(INOUT), OPTIONAL :: MAX_ITER
INTEGER, INTENT(INOUT), OPTIONAL :: MAX_EVL
REAL(KIND = R8), INTENT(INOUT), OPTIONAL :: MIN_DIA
REAL(KIND = R8), INTENT(IN), OPTIONAL :: OBJ_CONV
REAL(KIND = R8), INTENT(IN), OPTIONAL :: EPS
REAL(KIND = R8), INTENT(INOUT), OPTIONAL :: MIN_SEP
REAL(KIND = R8), DIMENSION(:), INTENT(INOUT), OPTIONAL :: W
TYPE(HyperBox), DIMENSION(:), INTENT(INOUT), OPTIONAL :: BOX_SET
INTEGER, INTENT(OUT), OPTIONAL :: NUM_BOX
INTEGER, INTENT(INOUT), OPTIONAL :: N_SUB
INTEGER, INTENT(INOUT), OPTIONAL :: N_MASTER
INTEGER, INTENT(IN), OPTIONAL :: RESTART
INTEGER, INTENT(IN), OPTIONAL :: BINSIZE
INTEGER, INTENT(IN), OPTIONAL :: TOTAL_MEM

! Local variables.
CHARACTER(LEN = 30) :: cpfile, cpfile1 ! Strings for checkpoint file names.
CHARACTER(LEN = 34) :: cpfile2 ! String for a checkpoint file name.
CHARACTER(LEN = 32) :: exec_name ! Name of the executable to be spawned.
CHARACTER(LEN = 9) :: hostfile = "qspawn.hf" ! Name of hostfile for spawning.
CHARACTER(LEN = 15) :: str ! Temporary string for constructing the
  ! checkpoint filename at the run-time.
CHARACTER(LEN = 32), ALLOCATABLE, DIMENSION(:) :: qspawn_proc_list ! Array
  ! containing host names.
INTEGER :: alloc_err ! Allocation error status.
INTEGER :: b_id ! Box matrix identifier.
INTEGER :: BINSIZE_I ! The local copy of 'BINSIZE'.
INTEGER :: bits_sub ! A flag for the result collection from 1 to N_SUB_I-1
  ! subdomain. Bit i represents the subdomain i+1. When bit i is set 1,
  ! the results from subdomain i+1 have been collected.
INTEGER :: box_count ! Counter for the number of boxes stored by a master.
INTEGER :: box_count_global ! The total number of boxes across all masters.
INTEGER :: boxset_ind ! BOX_SET array index counter.
INTEGER :: box_size ! Size of a box in 'lbuffer' and 'gbuffer'.
INTEGER :: bufsize ! Length of 'buffer'.
INTEGER :: c_alldone ! Counter for MYALL_DONE messages to workers.
INTEGER :: c_bits_sub ! Counter for finished subdomains.
INTEGER :: check_t ! Current iteration number for checkpoint recovery.
INTEGER :: child_comm ! Intercommunicator used by initial masters to
  ! communicate with spawned masters.
INTEGER :: chk_n ! N read from a checkpoint log file.
INTEGER :: chk_sd ! N_SUB_I read from a checkpoint log file.
INTEGER :: chk_sm ! N_MASTER_I read from a checkpoint log file.
INTEGER :: chk_sm1 ! N_MASTER_I read from other checkpoint log files.
INTEGER :: chk_switch ! SWITCH read from a checkpoint log file.
INTEGER :: col ! Local column index.
INTEGER :: eval_c ! Function evaluation counter.
INTEGER :: eval_c_i ! Local 'eval_c' on each subdomain master.
INTEGER :: gbuffer_len ! Length of 'gbuffer'.
INTEGER :: gc_convex ! The global convex box counter.
INTEGER :: group_world ! The group world ID for the masters in a subdomain.
INTEGER :: i, j, k ! Loop counters.
INTEGER :: ierr ! Error status for file I/O and MPI function calls.
INTEGER :: iflag ! Error flag for subroutine calls.
INTEGER :: info ! Value of 'info' argument used for spawning new masters.
INTEGER :: i_start ! Records the start index for searching in a node of
  ! 'setInd'.
INTEGER :: lbc ! If 1, LBC (limiting box columns) is enabled (default).
  ! If 0, LBC is disabled when the size of 'BOX_SET' is greater than 1 or
  ! MAX_ITER is not specified.
INTEGER :: lbuffer_len ! Lengths for 'lbuffer'.
INTEGER :: loc_pid = 1, loc_sid = 2, loc_eid = 3, loc_val = 4, loc_c = 5
INTEGER :: loc_side, loc_diam ! When gbuffer and lbuffer are used as buffers
  ! for convex hull boxes, each unit has box_size=2*N + 5 elements. In each
  ! unit:
  ! loc_pid=1 holds the original box owner's Processor ID,
  ! loc_sid=2 holds 'setInd' link ID of the box,
  ! loc_eid=3 holds the 'setInd' link element ID of the box,
  ! loc_val=4 holds function value of the box center,
  ! loc_c=5:N+4 holds point coordinates of the box center,
  ! loc_side=N+5:2*N+4 holds the sides of the box, and
  ! loc_diam=2*N+5 holds the diameter squared of the box.
INTEGER :: max_spawn = 2 ! Maximum number of spawning events.
INTEGER :: N_MASTER_I ! Local copy of argument 'N_MASTER'.
INTEGER :: N_WORKER_I ! Number of workers.
INTEGER :: N_SUB_I ! Local copy of argument 'N_SUB'.
INTEGER :: num_children ! The number of new masters to be spawned at a spawning
  ! event.
INTEGER :: parent_comm ! Intercommunicator used by spawned masters to
  ! communicate with initial masters.
INTEGER :: mygid ! Group ID.
INTEGER :: myid ! Processor ID.
INTEGER :: old_switch ! The value of the aggressive switch before a spawning
  ! event.
```

```
INTEGER :: qspawn_err ! The error code for 'makeHostFile'.
INTEGER :: qspawn_num_procs ! The number of unique host names for a given
  ! communicator.
INTEGER :: recv_tag ! The tag received to indicate the message type.
INTEGER :: RESTART_I ! Local copy of argument RESTART.
INTEGER :: set_size ! Size of a set in 'lbuffer' holding box dividing info.
  ! When lbuffer is used as a buffer to hold 'setI' info (only used in
  ! VTdirect) for all convex hull boxes for further processing in sampleP()
  ! and divide(), each unit has set_size=N+4 elements. In the first unit:
  ! 1: setI%dim, the length of 'setI' holding sampling directions,
  ! 2: the parent box column ID,
  ! 3: the 'setInd' ID,
  ! 4: the global box column ID, and
  ! 5:N+4: setI%elements, the elements of 'setI'.
INTEGER :: size_pointer = 4 ! Size of a pointer in bytes.
INTEGER :: size_r8 ! Size of the R8 real type in bytes.
INTEGER :: spawn_count = 0 ! The current number of spawning events.
INTEGER :: stop_rule ! Bits 0, 1, 2, 3 being set correspond to stopping rules
  ! 1 (iteration limit), 2 (function evaluation limit), 3 (minimum box
  ! diameter), 4 (relative change in 'FMIN') respectively.
INTEGER :: SWITCH_I ! Local copy of argument SWITCH.
INTEGER :: switch_iter ! The iteration at which the last spawning event
  ! occured.
INTEGER :: t ! Iteration counter.
INTEGER :: temp_comm_world ! Temporary communicator containing all processes
  ! after the parent and child intercommunicators are merged.
INTEGER :: temp_group_worker ! Temporary group containing workers.
INTEGER :: temp_group_world ! Temporary group used to make 'temp_comm_world'.
INTEGER :: trans_comm ! Communicator used to transfer state from the lead
  ! master to the spawned master(s).
INTEGER :: trans_group ! Group used to make 'trans_comm'.
INTEGER :: update_counter ! The counter for 'update_array'.
INTEGER :: world_size ! The total number of processes.
INTEGER :: xm_comm ! Handle for extended master communicator.
INTEGER :: xw_comm ! Handle for extended world communicator.
INTEGER :: xm_group ! Handle for extended master group.
INTEGER :: xw_group ! Handle for extended world group.
INTEGER, DIMENSION(:), ALLOCATABLE :: array_masters ! An array of processor
  ! IDs of masters in the same subdomain.
INTEGER, DIMENSION(:), ALLOCATABLE :: b_worker ! An array of blocking
  ! status of workers in multiple subdomain search. b_worker(i) represents
  ! the worker with processor ID = i. When b_worker(i) is 1, the worker has
  ! sent two blocking requests to this master.
INTEGER, DIMENSION(:), ALLOCATABLE :: chk_len ! An array holding the number
  ! of logs for an iteration on all masters.
INTEGER, DIMENSION(1) :: spawnErrcodes ! Array of errcodes for spawning.
INTEGER, DIMENSION(:), ALLOCATABLE :: displs ! An array used in gathering
  ! local convex hull boxes from subdomain masters.
INTEGER, DIMENSION(:), ALLOCATABLE :: lc_convex ! An array of the local
  ! convex box counters for all subdomain masters.
INTEGER, DIMENSION(:), ALLOCATABLE :: m_comm ! An array of sub-communicator
  ! IDs.
INTEGER, DIMENSION(:), ALLOCATABLE :: m_group ! An array of sub-group IDs.
INTEGER, DIMENSION(:), ALLOCATABLE :: master_ranks ! An array containing the
  ! ranks of masters, used for building communicators after spawning.
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: mstatus ! An array of info for MPI_RECV.
INTEGER, DIMENSION(14) :: mySIZE ! Array meant to simulate 'SIZE' primitive for
  ! spawned masters.
INTEGER :: q_counter ! The counter for workers in 'q_worker'.
INTEGER, DIMENSION(:), ALLOCATABLE :: q_worker ! A queue holding processor IDs
  ! of workers that have sent blocking requests to this master.
INTEGER, ALLOCATABLE, DIMENSION(:) :: qspawn_num_cores ! An array containing
  ! the number of cores on the given nodes.
INTEGER, ALLOCATABLE, DIMENSION(:) :: qspawn_num_hosted ! An array containing
  ! the number of processes running on the given nodes.
INTEGER, DIMENSION(49) :: statebuf_integer ! Buffer for transferring INTEGER
  ! variables to the spawned master(s).
INTEGER, DIMENSION(2) :: sub_divisor ! An array of decomposition parameters.
INTEGER, DIMENSION(2) :: sub_index ! An array of dimension indices.
INTEGER, DIMENSION(:), ALLOCATABLE :: trans_ranks
INTEGER, DIMENSION(:), ALLOCATABLE :: update_array ! An array holding processor
  ! IDs of masters that have updated results.
INTEGER, DIMENSION(:), ALLOCATABLE :: worker_ranks
LOGICAL :: do_it ! Sign to process first box in each column of BoxMatrix.
LOGICAL :: spawn_initiated ! Spawn procedure has been initiated.
LOGICAL :: spawn_requested_global ! Spawn procedure has been requested
  ! globally.
LOGICAL :: spawn_requested_local ! Spawn procedure has been requested locally.
LOGICAL :: spawned ! Flag specifying if a master was spawned.
LOGICAL, DIMENSION(1) :: statebuf_logical ! Buffer for transferring LOGICAL
  ! variables to the spawned master(s).
REAL(KIND = R8) :: box_mem ! Memory required to store a single 'HyperBox'
REAL(KIND = R8) :: chk_eps ! EPS read from a checkpoint log file.
REAL(KIND = R8) :: dia ! Diameter squared associated with 'FMIN'.
REAL(KIND = R8) :: dia_i ! The local 'dia' on a subdomain master.
REAL(KIND = R8) :: dia_limit ! Minimum diameter permitted.
```

```
REAL(KIND = R8) :: EPS_I ! Local copy of argument EPS.
REAL(KIND = R8) :: fmin_old ! FMIN backup.
REAL(KIND = R8) :: FMIN_I ! The local 'FMIN' on a subdomain master.
REAL(KIND = R8) :: MIN_SEP_I ! Local copy of argument MIN_SEP.
REAL(KIND = R8) :: size_mb = 1048576.0_R8 ! Size of a megabyte in bytes.
REAL(KIND = R8) :: tmpf ! Temporary real variable for function
  ! values read from the checkpoint log file and swapping operations.
REAL(KIND = R8), DIMENSION(:), ALLOCATABLE :: buffer ! MPI messaging buffer.
  ! When 'buffer' holds an evaluation task to a worker:
  ! buffer(1) holds point starting index,
  ! buffer(2) holds the number of points for this task, and
  ! buffer(3:) holds points.
  ! When 'buffer' holds the result data:
  ! buffer(1) holds FMIN,
  ! buffer(2:N+1) holds X,
  ! buffer(N+2) holds MAX_ITER,
  ! buffer(N+3) holds MAX_EVL,
  ! buffer(N+4) holds MIN_DIA,
  ! buffer(N+5) holds STATUS, and
  ! buffer(N+6) holds c_alldone.
REAL(KIND = R8), DIMENSION(:), ALLOCATABLE :: chk_l, chk_u ! L and U read
  ! from checkpoint log file.
REAL(KIND = R8), DIMENSION(N) :: current_center ! Center coordinates of
  ! the best unmarked box in the current box data structure.
REAL(KIND = R8), DIMENSION(2) :: dim_len  ! An array of dimension ranges.
REAL(KIND = R8), DIMENSION(:), ALLOCATABLE :: gbuffer ! A global buffer holding
  ! convex boxes.
REAL(KIND = R8), DIMENSION(:), ALLOCATABLE :: lbuffer ! A local buffer holding
  ! convex boxes or info for dividing boxes on a master.
REAL(KIND = R8), DIMENSION(11) :: statebuf_real ! Buffer for transferring REAL
  ! variables to the spawned master(s).
REAL(KIND = R8), DIMENSION(N) :: tmp_x ! X normalized to unit hypercube.
REAL(KIND = R8), DIMENSION(N) :: UmL ! An array containing U(:) - L(:).
REAL(KIND = R8), DIMENSION(N) :: unit_x ! X normalized to unit hypercube.
REAL(KIND = R8), DIMENSION(N) :: unit_x_i ! The local 'unit_x' on each SM.
REAL(KIND = R8), DIMENSION(N) :: W_I ! Local copy of weights W.
TYPE(OptResults), DIMENSION(:), ALLOCATABLE :: array_results ! An array of
  ! collected results from subdomain 1 to N_SUB_I-1, not including subdomain 0,
  ! which collects results from other subdomains.
TYPE(BoxMatrix), POINTER :: m_head ! The first box matrix.
TYPE(BoxLink), POINTER :: p_l ! Pointer to the current box link.
TYPE(BoxMatrix), POINTER :: p_b ! Pointer to box matrix.
TYPE(Hyperbox), POINTER :: p_box ! Box for the removed parent box to divide.
TYPE(HyperBox), POINTER :: p_save ! Pointer to the saved best box.
TYPE(int_vector), POINTER :: p_start ! Records the start node for searching
  ! the column with CONVEX_BIT set in 'setInd'.
TYPE(int_vector), POINTER :: p_iset ! Pointer to a node of 'int_vector' type
  ! linked lists, i.e. 'setInd', 'setFcol'.
TYPE(BoxLine) :: setB ! Buffer for newly sampled boxes and resized parent
  ! boxes.
TYPE(int_vector), POINTER :: setFcol ! A linked list. Each node holds free
  ! column indices in BoxMatrices.
TYPE(int_vector), POINTER :: setInd ! A linked list. Each node holds column
  ! indices corresponding to different squared diameters in 'setDia'.
TYPE(real_vector), POINTER :: setDia ! A linked list. Each node holds
  ! current different squared diameters from largest to smallest.
TYPE(ValList) :: setW ! Function values for newly sampled center points.

! Get the name of the executable to be spawned.
CALL GETARG(0, exec_name)

! Initialize box counts.
box_count = 0
box_count_global = 0

! Determine if this process was spawned, and if so obtain parent communicator.
CALL MPI_COMM_GET_PARENT(parent_comm, ierr)
IF (parent_comm == MPI_COMM_NULL) THEN
  spawned = .FALSE.
ELSE
  spawned = .TRUE.
END IF

! Initialize 'child_comm'.
child_comm = MPI_COMM_NULL

! Initialize 'xw_comm', and spawning status variables.
xw_comm = MPI_COMM_WORLD
spawn_requested_global = .FALSE.
spawn_requested_local = .FALSE.
spawn_initiated = .FALSE.

! Initialize 'STATUS'.
STATUS(:) = 0

! Tolerance for REAL number equality tests.
```

```
EPS4N = REAL(4*N, KIND = R8)*EPSILON(1.0_R8)

! Set the MPI_ERRORS_RETURN handler for the initialization phase.
CALL MPI_ERRHANDLER_SET(MPI_COMM_WORLD, MPI_ERRORS_RETURN, ierr)
! Find 'myid', 'group_world', and 'world_size'.
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
IF (ierr /= MPI_SUCCESS) THEN
  STATUS(1) = DMPI_ERROR + 1
  RETURN
END IF
CALL MPI_COMM_GROUP(MPI_COMM_WORLD, group_world, ierr)
IF (ierr /= MPI_SUCCESS) THEN
  STATUS(1) = DMPI_ERROR + 2
  RETURN
END IF
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, world_size, ierr)
IF (ierr /= MPI_SUCCESS) THEN
  STATUS(1) = DMPI_ERROR + 3
  RETURN
END IF
PROCID = myid

! Perform sanity check of input arguments and set local variables derived
! from input arguments.
STATUS(1) = sanitycheck()
IF (STATUS(1) /= 0)  RETURN

! Allocate and initialize buffers. Find the byte size of R8 type and compute
! 'box_mem' and 'bufsize'.
INQUIRE(IOLENGTH=size_r8) 1.0_R8

! Set the amount of memory used by a 'HyperBox'.
box_mem = REAL(2*(N + 1)*size_r8 + 2*size_pointer, R8) ! A 'HyperBox' has
  ! two pointers (4 bytes each), two REAL's (KIND = R8), and two arrays
  ! of length N containing REAL's (KIND = R8).

bufsize = MAX(2 + BINSIZE_I*(N + 2), N + 6)*size_r8 ! 'buffer' is used to hold
  ! an evaluation task to a worker (2+BINSIZE_I*(N+2)) or to hold the result
  ! data (N+6) from a master.
ALLOCATE(buffer(bufsize/size_r8))
buffer(:) = 0
ALLOCATE(array_masters(world_size))
array_masters(:) = 0

! Assign group ID 'mygid'.
IF (world_size == 1) THEN ! Only one processor is used.
  mygid = 0
ELSE ! Multiple processors are used, so allocate 'm_comm' and 'm_group' to
  ! group processors for different subdomains .
  ALLOCATE(m_comm(N_SUB_I))
  ALLOCATE(m_group(N_SUB_I))
  m_comm(:) = 0
  m_group(:) = 0
  ! Create one sub-communicator for each subdomain and store the
  ! group IDs in 'm_group' and communicator IDs in 'm_comm'.
  DO i = 1, N_SUB_I
    DO j = 1, N_MASTER_I
      array_masters(j) = (i - 1)*N_MASTER_I + j - 1
    END DO
    CALL MPI_GROUP_INCL(group_world, N_MASTER_I, array_masters, &
                        m_group(i), ierr)
    IF (ierr /= MPI_SUCCESS) THEN
      STATUS(1) = DMPI_ERROR + 4
      RETURN
    END IF
    CALL MPI_COMM_CREATE(MPI_COMM_WORLD, m_group(i), m_comm(i), ierr)
    IF (ierr /= MPI_SUCCESS) THEN
      STATUS(1) = DMPI_ERROR + 5
      RETURN
    END IF
    ! For a master, obtain the group ID.
    IF (inQueue(array_masters, N_MASTER_I, myid)) THEN
      CALL MPI_GROUP_RANK(m_group(i), mygid, ierr)
      IF (ierr /= MPI_SUCCESS) THEN
        STATUS(1) = DMPI_ERROR + 6
        RETURN
      END IF
    END IF
  END DO
  ! Initialize 'master_ranks' and 'worker_ranks'.
  N_WORKER_I = world_size - (N_MASTER_I*N_SUB_I)
  ALLOCATE(master_ranks(N_MASTER_I))
  ALLOCATE(worker_ranks(N_WORKER_I))
  DO i = 1, N_MASTER_I
    master_ranks(i) = i - 1
  END DO
```

```
  DO i = 1, N_WORKER_I
    worker_ranks(i) = world_size - N_WORKER_I + (i - 1)
  END DO
END IF

! Reset the MPI_ERRORS_ARE_FATAL handler.
CALL MPI_ERRHANDLER_SET(MPI_COMM_WORLD, MPI_ERRORS_ARE_FATAL, ierr)

! Initialize 'bits_sub' and 'c_bits_sub'.
bits_sub = 0
c_bits_sub = 0
! Branch to either a master or a worker depending on 'myid'.
IF (world_size < 3 .OR. ((world_size >= 3) .AND. &
   (myid < N_MASTER_I*N_SUB_I))) THEN ! The processor is a master when fewer
  ! than 3 processors are used, or when more than 3 processors are used and
  ! 'myid' is within the range of master IDs [0, N_MASTER_I*N_SUB_I-1].
  ! Allocate and initialize buffers and counters used by the master.
  ALLOCATE(q_worker(world_size))
  q_worker(:) = 0
  q_counter = 0
  ALLOCATE(b_worker(world_size))
  b_worker(:) = 0
  c_alldone = 0
  IF (N_SUB_I > 1) THEN ! For a multiple subdomain search, divide the
    ! original domain into 'N_SUB_I' parts by subdividing the two longest
    ! scaled dimensions (in dim_len(:)) into sub_divisor(1) and sub_divisor(2)
    ! parts, respectively, where sub_divisor(1)*sub_divisor(2)=N_SUB_I, and
    ! sub_divisor(2)/sub_divisor(1) approximates dim_len(2)/dim_len(1).
    !
    ! Find the index for the longest scaled dimension.
    sub_index(1) = MAXLOC(W_I(1:N)*(U(1:N) - L(1:N)), DIM=1)
    ! Save that longest scaled dimension in dim_len(1).
    dim_len(1) = W_I(sub_index(1))*(U(sub_index(1)) - L(sub_index(1)))
    tmpf = W_I(sub_index(1)) ! Save that dimension's weight.
    ! Zero that weight to find the next longest scaled dimension.
    W_I(sub_index(1)) = 0.0_R8
    sub_index(2) = MAXLOC(W_I(1:N)*(U(1:N) - L(1:N)), DIM=1)
    dim_len(2) = W_I(sub_index(2))*(U(sub_index(2)) - L(sub_index(2)))
    W_I(sub_index(1)) = tmpf ! Recover the zeroed weight.

    ! Adjust 'sub_divisor' so that the ratio sub_divisor(2)/sub_divisor(1)
    ! approximates dim_len(2)/dim_len(1).
    IF (dim_len(1) >= N_SUB_I*dim_len(2)) THEN
      sub_divisor(1:2) = (/ N_SUB_I, 1 /)
    ELSE
      sub_divisor(1:2) = (/ 1, N_SUB_I /)
      DO
        DO i = sub_divisor(1) + 1, N_SUB_I
          IF (MOD(N_SUB_I, i) == 0) EXIT
        END DO
        IF (dim_len(1)*N_SUB_I < i*i*dim_len(2)) EXIT
        sub_divisor(1:2) = (/ i, N_SUB_I/i /)
      END DO
      IF (ABS(REAL(sub_divisor(2), KIND = R8)/ &
          REAL(sub_divisor(1), KIND = R8) - &
          dim_len(2)/dim_len(1)) > &
          ABS(REAL(N_SUB_I, KIND = R8)/REAL(i*i, KIND = R8) - &
          dim_len(2)/dim_len(1)) ) THEN
        sub_divisor(1:2) = (/ i, N_SUB_I/i /)
      END IF
    END IF

! Decompose the domain along the longest scaled dimensions indexed
! by 'sub_index(1:2)' for the master processor myid.  The subdomain ID for the
! master processor 'myid' is myid/N_MASTER_I.
    SPLITLOOP: DO i = 1, 2
      tmpf = (U(sub_index(i)) - L(sub_index(i)))/ &
        REAL(sub_divisor(i), KIND = R8)
      IF (i == 1) THEN
        L(sub_index(i)) = L(sub_index(i)) + tmpf* &
          ! The subdomain column ID is:
          REAL(MOD(myid/N_MASTER_I, sub_divisor(1)), KIND = R8)
      ELSE
        L(sub_index(i)) = L(sub_index(i)) + tmpf* &
          ! The subdomain row ID is:
          REAL((myid/N_MASTER_I)/sub_divisor(1), KIND = R8)
      END IF
      U(sub_index(i)) = L(sub_index(i)) + tmpf
    END DO SPLITLOOP

    ! The root master allocates 'array_results' for merging final results
    ! from different subdomains.
    IF (myid == 0) THEN
      ALLOCATE(array_results(N_SUB_I - 1))
      DO i = 1, N_SUB_I - 1
        ALLOCATE(array_results(i)%x(N))
```

```
    END DO
  END IF
END IF
! Each root subdomain master allocates 'update_array' for intermediate
! result updates.
IF (mygid == 0) ALLOCATE(update_array(N_MASTER_I))
! Prepare 'array_masters' for calling master().
DO i = 1, N_MASTER_I
  array_masters(i) = INT(myid/N_MASTER_I)*N_MASTER_I + i - 1
END DO
iflag = 0
CALL master(iflag)
IF (iflag /= 0) THEN
  STATUS(1) = iflag
  IF (iflag /= DMPI_ERROR + 7) THEN
    ! Notify all others that it has to abort the program. Use 'b_worker'
    ! as the send buffer and 'q_worker' as the receive buffer.
    b_worker(:) = 0
    CALL MPI_ALLTOALL(b_worker, 1, MPI_INTEGER, q_worker, 1, &
                      MPI_INTEGER, xw_comm, ierr)
    WRITE(*,315) myid, iflag
    315 FORMAT(" Master Processor ",I5, &
              " aborts the program with error flag ", &
              I3,". See comments in spVTdirect to interpret flag.")
  END IF
  RETURN
END IF
! The master releases workers in the queue by sending 'MYALL_DONE'
! messages.
DO
  IF (q_counter == 0) EXIT ! Exit when no more workers in the queue.
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, &
              q_worker(q_counter), MYALL_DONE, xw_comm, ierr)
  q_counter = q_counter - 1
  c_alldone = c_alldone + 1
END DO
! The root master merges subdomain results and initiates the global
! termination to all other masters.
IF (myid == 0) THEN ! The root master.
  IF (c_bits_sub > 0) THEN
    ! At least one subdomain has finished earlier than the root subdomain.
    ! Process the results stored in array_results(:).
    DO i = 1, N_SUB_I - 1
      IF (BTEST(bits_sub, i)) THEN ! Merging results for subdomain i.
        IF (array_results(i)%fmin < FMIN) THEN
          FMIN = array_results(i)%fmin
          X(:) = array_results(i)%x
        END IF
        IF (PRESENT(MAX_ITER)) THEN
          IF (MAX_ITER < array_results(i)%max_iter) MAX_ITER = &
            array_results(i)%max_iter
        END IF
        IF (PRESENT(MAX_EVL)) MAX_EVL = MAX_EVL + array_results(i)%max_evl
        IF (PRESENT(MIN_DIA)) THEN
          IF (MIN_DIA > array_results(i)%min_dia) MIN_DIA = &
            array_results(i)%min_dia
        END IF
        STATUS(i + 1) = array_results(i)%status
      END IF
    END DO
  END IF
  ! Clean up all left-over requests before starting the global termination.
  ! Collect results from subdomains that are finishing.
  DO
    IF ( (c_bits_sub >= N_SUB_I - 1) .AND. &
       (c_alldone >= (world_size - N_SUB_I*N_MASTER_I)*N_SUB_I) ) EXIT
    ! Wait for messages until all subdomain results are collected and
    ! all workers have received MYALL_DONE to terminate themselves.
    CALL MPI_RECV(buffer, bufsize, MPI_BYTE, MPI_ANY_SOURCE, &
                  MPI_ANY_TAG, xw_comm, mstatus, ierr)
    IF (mstatus(MPI_TAG) == RESULT_DATA) THEN
      ! Merge the received results from a root subdomain master.
      IF (buffer(1) < FMIN) THEN
        FMIN = buffer(1)
        X(:) = buffer(2:N + 1)
      END IF
      IF (PRESENT(MAX_ITER)) THEN
        IF (MAX_ITER < INT(buffer(N + 2))) MAX_ITER = INT(buffer(N + 2))
      END IF
      IF (PRESENT(MAX_EVL)) MAX_EVL = MAX_EVL + INT(buffer(N + 3))
      IF (PRESENT(MIN_DIA)) THEN
        IF (MIN_DIA > buffer(N + 4)) MIN_DIA = buffer(N + 4)
      END IF
      STATUS(mstatus(MPI_SOURCE)/N_MASTER_I + 1) = INT(buffer(N + 5))
      ! Update 'c_bits_sub' and 'c_alldone'.
      c_bits_sub = c_bits_sub + 1
```

```
      c_alldone = c_alldone + INT(buffer(N + 6))
    END IF
    ! Process the received left-over request from workers.
    IF ( (mstatus(MPI_TAG) == NONBLOCK_REQ) .OR. (mstatus(MPI_TAG) &
       == BLOCK_REQ) ) THEN
      ! Send 'MYALL_DONE' to remove itself from the worker's master list.
      CALL MPI_SEND(buffer, bufsize, MPI_BYTE, &
                  mstatus(MPI_SOURCE), MYALL_DONE, xw_comm, ierr)
      ! Update 'c_alldone' if the message is not from a master-converted
      ! worker.
      IF (INT(buffer(1)) == 0) c_alldone = c_alldone + 1
    END IF
    ! Process the received message to update 'c_alldone'.
    IF (mstatus(MPI_TAG) == COUNT_DONE) THEN
      c_alldone = c_alldone + INT(buffer(1))
    END IF
  END DO
  ! The root sends termination messages to its left and right processors.
  i = myid*2 + 1
  j = myid*2 + 2
  IF (i <= N_MASTER_I*N_SUB_I - 1) THEN
    CALL MPI_SEND(buffer, bufsize, MPI_BYTE, i, TERMINATE, &
                xw_comm, ierr)
  END IF
  IF (j <= N_MASTER_I*N_SUB_I - 1) THEN
    CALL MPI_SEND(buffer, bufsize, MPI_BYTE, j, TERMINATE, &
                xw_comm, ierr)
  END IF
ELSE ! Nonroot master.
  IF (mygid == 0) THEN ! The root subdomain master.
    ! Send the search results and 'c_alldone' to the root master.
    buffer(1) = FMIN
    buffer(2:N + 1) = X
    buffer(N + 2) = t
    buffer(N + 3) = eval_c
    buffer(N + 4) = SQRT(dia)
    buffer(N + 5) = STATUS(1)
    buffer(N + 6) = c_alldone
    CALL MPI_SEND(buffer, bufsize, MPI_BYTE, 0, RESULT_DATA, &
                xw_comm, ierr)
  END IF
  ! All nonroot masters send the counter 'c_alldone' to the root master.
  buffer(1) = c_alldone
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, 0, COUNT_DONE,&
             xw_comm, ierr)
  ! Wait for the final termination message and pass it down to the binary
  ! tree of master processors.
  TERMLOOP: DO
    CALL MPI_RECV(buffer, bufsize, MPI_BYTE, MPI_ANY_SOURCE, &
               MPI_ANY_TAG, xw_comm, mstatus, ierr)
    IF (mstatus(MPI_TAG) == TERMINATE) THEN
      ! Received a termination message.
      i = myid*2 + 1
      j = myid*2 + 2
      IF (i <= N_MASTER_I*N_SUB_I - 1) THEN
        CALL MPI_SEND(buffer, bufsize, MPI_BYTE, i, &
                    mstatus(MPI_TAG), xw_comm, ierr)
      END IF
      IF (j <= N_MASTER_I*N_SUB_I - 1) THEN
        CALL MPI_SEND(buffer, bufsize, MPI_BYTE, j, &
                    mstatus(MPI_TAG), xw_comm, ierr)
      END IF
      EXIT TERMLOOP
    END IF
    IF (mstatus(MPI_TAG) == BE_WORKER) THEN
      ! Receive a "BE_WORKER" message. To become a worker, obtain
      ! 'bits_sub' (busy status of all subdomains) from 'buffer'
      ! preparing to call worker().
      bits_sub = INT(buffer(1))
      ! The root subdomain master passes this "BE_WORKER" msg to nonroot
      ! subdomain masters (if any).
      IF ( (myid == array_masters(1)) .AND. (N_MASTER_I > 1) ) THEN
        DO i = 2, N_MASTER_I
          CALL MPI_SEND(buffer, bufsize, MPI_BYTE, &
                      array_masters(i), BE_WORKER, xw_comm, ierr)
        END DO
      END IF
      CALL worker(bits_sub)
      EXIT TERMLOOP
    END IF
    IF ( (mstatus(MPI_TAG) == NONBLOCK_REQ) .OR. (mstatus(MPI_TAG) &
       == BLOCK_REQ) ) THEN
      ! Process the received request from a worker. Send "MYALL_DONE"
      ! reply to remove itself from the worker's master list.
      CALL MPI_SEND(buffer, bufsize, MPI_BYTE, &
                  mstatus(MPI_SOURCE), MYALL_DONE, xw_comm, ierr)
```

```
      ! Send the update of 'c_alldone' to the root master if the message
      ! is not from a master-converted worker.
      IF (INT(buffer(1)) == 0) THEN
        buffer(1) = 1
        CALL MPI_SEND(buffer, bufsize, MPI_BYTE, &
                      0, COUNT_DONE, xw_comm, ierr)
      END IF
    END IF
  END DO TERMLOOP
 END IF ! end of IF (myid ==0)

 ! Deallocate buffers.
 DEALLOCATE(q_worker)
 DEALLOCATE(b_worker)
 IF (N_SUB_I > 1) THEN
   ! Deallocate 'array_results', 'buffer', and 'update_array'.
   IF (myid == 0) THEN
     DO i = 1, N_SUB_I - 1
       DEALLOCATE(array_results(i)%x)
     END DO
     DEALLOCATE(array_results)
   END IF
 END IF
 IF (mygid == 0) DEALLOCATE(update_array)
ELSE ! A worker.
 CALL worker(bits_sub)
END IF

! If new masters were spawned, then free parent and child intercommunicators.
IF (spawn_count > 0) THEN
 IF (spawned) THEN
   CALL MPI_COMM_FREE(parent_comm, ierr)
 ELSE
   CALL MPI_COMM_FREE(child_comm, ierr)
 END IF
END IF

! Deallocate common buffers.
DEALLOCATE(array_masters)
DEALLOCATE(buffer)
IF (world_size > 1 .AND. .NOT. spawned) THEN
 DO i = 1, N_SUB_I
   IF (m_comm(i) /= MPI_COMM_NULL) CALL MPI_COMM_FREE(m_comm(i), ierr)
   IF (m_group(i) /= MPI_GROUP_NULL) CALL MPI_GROUP_FREE(m_group(i), ierr)
 END DO
 DEALLOCATE(m_group)
 DEALLOCATE(m_comm)
END IF

RETURN
CONTAINS

SUBROUTINE boxSelection(tmpbox, m_portion, iflag)
IMPLICIT NONE
! Select potentially optimal boxes (convex hull boxes) in parallel when
! multiple masters are used in a subdomain, or in sequential when only one
! master is used in a subdomain.
!
! On input:
! xm_comm - The sub-communicator of the master.
! tmpbox - A pre-allocated temporary buffer for swapping boxes in 'gbuffer'.
!
! On output:
! m_portion - Number of global convex hull boxes for this master.
! tmpbox    - As above under "On input".
! iflag     - Status to return.
!             0    Normal return.
!             >0   Error return.
!
REAL(KIND = R8), DIMENSION(:), INTENT(INOUT) :: tmpbox
INTEGER, INTENT(OUT) :: m_portion
INTEGER, INTENT(INOUT) :: iflag

! Local variables.
DOUBLE PRECISION :: box_avg ! Average box load over masters.
DOUBLE PRECISION :: box_dev ! Percent deviation from average box load.
INTEGER :: eid ! Box element ID extracted from 'gbuffer'.
INTEGER :: endbox ! Index of the last convex hull box for this master.
INTEGER :: i, i1, i2, j ! Loop counters.
INTEGER :: mycol ! Box column index.
INTEGER, DIMENSION(:), ALLOCATABLE :: seed ! Seed for random number generator.
INTEGER :: seed_size ! Size of random seed array.
INTEGER :: sid ! Box set ID extracted from 'gbuffer'.
INTEGER :: startloc1, startloc2 ! Start indices for 'gbuffer' or 'lbuffer'.
INTEGER :: startbox ! Index of the first convex hull box for this master.
INTEGER :: tmp_portion ! Used to store box portion information.
```

```
INTEGER :: tmp_rank ! Used to store process rank.
INTEGER :: top ! Top box index for the box stack.
! contained in the comm buffer.
REAL(KIND = R8) :: harvest ! Stores random real used to distribute excess
! box portion.
REAL(KIND = R8) :: slope, slope_min ! Real variables for slope computation.

! Initialize 'iflag', 'lc_convex', and 'gc_convex'.
iflag = 0
lc_convex(:) = 0
gc_convex = 0
IF (N_MASTER_I > 1) THEN
  ! Using multiple subdomain masters, when convex hull processing is on,
  ! (SWITCH_I==1) convex hull boxes will be processed in parallel as follows:
  ! 1. Each master finds its local convex hull boxes;
  ! 2. The root master gathers all local convex hull boxes and finds
  !    the global convex set of boxes.
  ! 3. Each master obtains its own portion of global convex boxes.
  ! When convex hull processing is off (SWITCH_I==0), the root master collects
  ! all the lowest boxes, removes the ones with smaller diameters than the one
  ! with 'FMIN', and redistributes them to all masters in the same subdomain.

  ! Initially, the master marks the lowest boxes from the one with the
  ! biggest diameter to the one holding the current 'FMIN' as convex
  ! hull boxes.
 p_iset => setInd
 MARKLOOP: DO
   IF (p_iset%dim == 0) THEN
     EXIT MARKLOOP
   END IF
   DO i = 1, p_iset%dim
     ! Mark the box as the convex hull box.
     p_iset%flags(i) = IBSET(p_iset%flags(i), CONVEX_BIT)
     ! Update 'gc_convex'.
     gc_convex = gc_convex + 1
     ! Exit if the column has reached the one with 'FMIN'.
     b_id = (p_iset%elements(i) - 1)/col_w + 1
     col = MOD(p_iset%elements(i) - 1, col_w) + 1
     p_b => m_head
     DO j = 1, b_id -1
       p_b => p_b%child
     END DO
     IF (dia_i == p_b%M(1,col)%diam) EXIT MARKLOOP
   END DO
   ! Go to the next link if any.
   IF (ASSOCIATED(p_iset%next)) THEN
     p_iset => p_iset%next
   ELSE
     EXIT MARKLOOP
   END IF
 END DO MARKLOOP

 IF (SWITCH_I == 1) THEN ! Convex hull processing is on.
   ! Find local convex hull boxes from all the marked boxes.
   CALL findconvex(m_head, p_iset, 0, setInd)
   ! Enlarge 'lbuffer' to hold 'gc_convex' number of local convex boxes
   ! if needed.
   IF (gc_convex*box_size > lbuffer_len) THEN
     DEALLOCATE(lbuffer)
     ALLOCATE(lbuffer(MAX(INT(2*lbuffer_len), gc_convex*box_size)))
     lbuffer_len = SIZE(lbuffer)
   END IF
 END IF
 ! The master puts all marked boxes to 'lbuffer'.
 p_iset => setInd
 PUTLOOP: DO
   IF (p_iset%dim == 0) EXIT PUTLOOP
   DO i = 1, p_iset%dim
     b_id = (p_iset%elements(i) - 1)/col_w + 1
     col = MOD(p_iset%elements(i) - 1, col_w) + 1
     p_b => m_head
     DO j = 1, b_id -1
       p_b => p_b%child
     END DO
     IF (BTEST(p_iset%flags(i), CONVEX_BIT)) THEN
       ! Found a local convex box, so put it in 'lbuffer'.
       startloc1 = lc_convex(mygid + 1)*box_size
       lbuffer(startloc1 + loc_pid) = myid
       lbuffer(startloc1 + loc_sid) = p_iset%id
       lbuffer(startloc1 + loc_eid) = i
       lbuffer(startloc1 + loc_c: startloc1 + loc_c + N - 1) &
         = p_b%M(1,col)%c(:)
       lbuffer(startloc1 + loc_side: startloc1 + loc_side + N - 1) &
         = p_b%M(1,col)%side(:)
       lbuffer(startloc1 + loc_val) = p_b%M(1,col)%val
       lbuffer(startloc1 + loc_diam) = p_b%M(1,col)%diam
```

```
      lc_convex(mygid + 1) = lc_convex(mygid + 1) + 1
      ! Clear the CONVEX_BIT.
      p_iset%flags(i) = IBCLR(p_iset%flags(i), CONVEX_BIT)
    END IF
  END DO
  ! Go to the next link if any.
  IF (ASSOCIATED(p_iset%next)) THEN
    p_iset => p_iset%next
  ELSE
    EXIT PUTLOOP
  END IF
END DO PUTLOOP
! Wait for all subdomain masters to get to this point.
CALL MPI_BARRIER(xm_comm, ierr)
! The root subdomain master gathers the local counters in 'lc_convex'
! from all masters. Then, it collects local convex hull boxes from
! 'lbuffer' on each master, save them in its 'gbuffer', and finds the
! global convex boxes.
CALL MPI_GATHER(lc_convex(mygid + 1), 1, MPI_INTEGER, &
              lc_convex(mygid + 1), 1, MPI_INTEGER, 0, xm_comm, ierr)
IF (mygid == 0) THEN
  IF (gbuffer_len < SUM(lc_convex(:))*box_size) THEN
    ! The global buffer 'gbuffer' is not big enough to hold all local
    ! convex hull boxes. Reallocate 'gbuffer'.
    DEALLOCATE(gbuffer)
    ALLOCATE(gbuffer(MAX(INT(2*gbuffer_len), SUM(lc_convex(:)*box_size))))
    gbuffer_len = SIZE(gbuffer)
  END IF
  ! The root subdomain master prepares 'displs' to gather local convex boxes.
  ! Each entry i specifies the displacement relative to a buffer at which to
  ! place the incoming data from master i.
  displs(1) = 0
  DO i = 2, N_MASTER_I
    displs(i) = displs(i - 1) + lc_convex(i - 1)*box_size*size_r8
  END DO
END IF
! The root subdomain master gathers the boxes. Compute the amount of
! data in unit of MPI_BYTE for gathering in lc_convex(:).
lc_convex(:) = lc_convex(:)*box_size*size_r8
CALL MPI_GATHERV(lbuffer, lc_convex(mygid + 1), MPI_BYTE, &
               gbuffer, lc_convex(:), displs, MPI_BYTE, &
               0, xm_comm, ierr)
! Recover lc_convex with the number of boxes in place of the number of units.
lc_convex(:) = INT(lc_convex(:)/(box_size*size_r8))

! The root subdomain master merges the boxes by size.
IF (mygid == 0) THEN
  ! First, merge box sets according to box diameters in 'gbuffer'.
  ! Initially, the merged portion is the box set at root subdomain master.
  gc_convex = lc_convex(1)
  MERGELOOP1: DO i = 2, N_MASTER_I
    IF (lc_convex(i) <= 0) CYCLE
    i1 = 0 ! Index to the merged portion of gbuffer.
    i2 = 0 ! Index to the portion to be merged from subdomain master 'i'.
    MERGELOOP2: DO
      startloc1 = i1*box_size
      startloc2 = displs(i)/size_r8 + i2*box_size
      IF (gbuffer(startloc2 + loc_diam) > gbuffer(startloc1 + loc_diam)) THEN
        ! The diameter is bigger, so insert this box indexed by 'i2'
        ! before box 'i1' in the merged portion.
        tmpbox(:) = gbuffer(startloc2 + 1:startloc2 + box_size)
        gbuffer(startloc1 + box_size + 1:startloc1 &
          + (gc_convex - i1 + 1)*box_size) = &
          gbuffer(startloc1 + 1:startloc1 + (gc_convex - i1)*box_size)
        gbuffer(startloc1 + 1:startloc1 + box_size) = tmpbox(:)
        ! Update the counter 'gc_convex' for the merged boxes.
        gc_convex = gc_convex + 1
        ! Update 'i1' and 'i2'.
        i1 = i1 + 1
        i2 = i2 + 1
      ELSE ! Smaller or equal diameters.
        IF ( (ABS(gbuffer(startloc1 + loc_diam) - &
           gbuffer(startloc2 + loc_diam))/ &
           gbuffer(startloc2 + loc_diam)) <= EPS4N ) THEN
          ! The diameters are equal, so compare the function values.
          IF (gbuffer(startloc1 + loc_val) &
              > gbuffer(startloc2 + loc_val)) THEN
            ! Smaller function value wins. Override the box 'i1'.
            gbuffer(startloc1 + 1:startloc1 + box_size) = &
              gbuffer(startloc2 + 1:startloc2 + box_size)
          ELSE
            IF (gbuffer(startloc1 + loc_val) &
                == gbuffer(startloc2 + loc_val)) THEN
              ! For equal function values, smaller lex order wins.
              IF (gbuffer(startloc2 + loc_c:startloc2 + loc_c + N - 1) &
                .lexLT. gbuffer(startloc1 + loc_c:startloc1 + loc_c &
```

```
           + N - 1)) THEN
           gbuffer(startloc1 + 1:startloc1 + box_size) = &
             gbuffer(startloc2 + 1:startloc2 + box_size)
         END IF
        END IF
       END IF
       ! Update 'i1' and 'i2'. 'gc_convex' stays the same.
       i1 = i1 + 1
       i2 = i2 + 1
     ELSE ! The diameter is smaller, so 'i2' stays the same and will be
       ! compared with the next box 'i1' in the merged portion.
       i1 = i1 + 1
     END IF
   END IF
   ! When 'i2' reaches the last box for that subdomain master, exit
   ! MERGELOOP2 and move on to local convex hull boxes from next
   ! subdomain master.
   IF (i2 >= lc_convex(i)) EXIT MERGELOOP2
   IF (i1 >= gc_convex) THEN
     ! When 'i1' reaches the end of merged portion, insert the rest of
     ! the local convex hull boxes starting from 'i2' for subdomain 'i'
     ! to the end of merged portion.
     startloc1 = i1*box_size
     startloc2 = displs(i)/size_r8 + i2*box_size
     gbuffer(startloc1 + 1:startloc1 + (lc_convex(i) - i2)*box_size) = &
       gbuffer(startloc2 + 1:startloc2 + (lc_convex(i) - i2)*box_size)
     gc_convex = gc_convex + lc_convex(i) - i2
     EXIT MERGELOOP2
   END IF
  END DO MERGELOOP2
END DO MERGELOOP1
! Throw away boxes after the one with 'FMIN'.
DO i = 0, gc_convex - 1
  IF (dia == gbuffer(i*box_size + loc_diam)) THEN
    ! Update 'gc_convex'.
    gc_convex = i + 1
    EXIT
  END IF
END DO
IF (SWITCH_I == 1) THEN ! Convex hull processing is on.
  ! Identify the global convex boxes in 'gbuffer'.
  i = 0  ! Index to the boxes in 'gbuffer'.
  ! Slightly modified Graham's scan algorithm is used to find a lower
  ! right convex hull instead of a generic convex hull. Scan through the
  ! lowest boxes starting from the box with the biggest diameter using
  ! a stack to backtrack boxes. 'gbuffer' serves as both the buffer for
  ! all boxes and a stack for back-tracking.

  ! Initially, on top of the box stack is the 'b_i' with the biggest
  ! diameter in 'gbuffer'.
  top = i
  ! Find the box 'b_j' after 'b_i', that has a smaller function value.
  DO j = i + 1, gc_convex - 1
    IF (gbuffer(i*box_size + loc_val) > gbuffer(j*box_size + loc_val)) EXIT
  END DO
  ! Remove the boxes between 'b_i' and 'b_j'.
  IF (j <= gc_convex - 1) THEN
    gbuffer((i + 1)*box_size + 1:(i + 1)*box_size &
      + (gc_convex - j)*box_size) = &
      gbuffer(j*box_size + 1:gc_convex*box_size)
    ! Update counter 'gc_convex'.
    gc_convex = gc_convex - (j - i - 1)
    ! Update 'j'.
    j = i + 1
  END IF
  ! Now 'b_j' is at the top of stack.
  top = j
  ! Loop checking if the next box b_k (k=top+1,...) is on the current
  ! convex hull and keep updating b_i and top.
  k = j
  SCANLOOP: DO
    k = k + 1
    IF (k >= gc_convex) EXIT SCANLOOP
    IF (onRight(gbuffer(i*box_size + loc_val), &
              SQRT(gbuffer(i*box_size + loc_diam)), &
              gbuffer(j*box_size + loc_val), &
              SQRT(gbuffer(j*box_size + loc_diam)), &
              gbuffer(k*box_size + loc_val), &
              SQRT(gbuffer(k*box_size + loc_diam)))) THEN
      ! 'b_k' is on the right of the line connecting 'b_i' and 'top'.
      ! Pop 'top' at 'j' position out of stack.
      gbuffer(j*box_size + 1:j*box_size + (gc_convex - j - 1)*box_size) = &
        gbuffer((j + 1)*box_size + 1:gc_convex*box_size)
      gc_convex = gc_convex - 1
      k = k - 1
      ! Update counters.
```

```
  top = top - 1
  i = top - 1
  ! Back-tracking until 'top' can stay on the current convex hull.
  BTLOOP: DO
    IF (top == 0) EXIT BTLOOP
    IF (onRight(gbuffer(i*box_size + loc_val), &
              SQRT(gbuffer(i*box_size + loc_diam)), &
              gbuffer(top*box_size + loc_val), &
              SQRT(gbuffer(top*box_size + loc_diam)), &
              gbuffer(k*box_size + loc_val), &
              SQRT(gbuffer(k*box_size + loc_diam)))) THEN
      ! Pop out 'top'.
      gbuffer(top*box_size + 1:top*box_size+ &
        (gc_convex - top - 1)*box_size) = &
        gbuffer((top + 1)*box_size + 1:gc_convex*box_size)
      ! Update counters.
      gc_convex = gc_convex - 1
      k = k - 1
      top = top - 1
      i = top - 1
      IF (i == -1) EXIT BTLOOP
    ELSE
      EXIT BTLOOP
    END IF
  END DO BTLOOP
  ! Let 'b_k' be 'top', and 'b_i' is the same.
  top = k
  i = top - 1
  j = k
ELSE ! 'b_k' is on the left of the line connecting b_i and b_top.
  IF (gbuffer(k*box_size + loc_val) &
      > gbuffer(j*box_size + loc_val)) THEN
    ! 'b_k' has a bigger function value. Pop it out and move on to
    ! the next one.
    IF (k < gc_convex - 1) THEN
      gbuffer(k*box_size + 1:k*box_size &
        + (gc_convex - k - 1)*box_size) = &
        gbuffer((k + 1)*box_size + 1:gc_convex*box_size)
    END IF
    gc_convex = gc_convex - 1
    k = k - 1
  ELSE ! 'b_k' has a smaller or an equal value.
    ! Let 'top' be 'b_i'.
      i = j
      ! Let b_k be b_top.
      j = k
      top = j
    END IF
  END IF
END DO SCANLOOP
! Check the EPS_I condition on the global convex boxes in 'gbuffer' to
! prevent search from becoming too local.
IF (EPS_I /= 0) THEN
  i = 0
  j = i + 1
  DO
    IF (j >= gc_convex) EXIT
    ! Compute the slope from the convex box 'i' to the imaginary point
    ! with val=FMIN-(ABS(FMIN)+1)*EPS_I and diam=0.
    slope_min = (gbuffer(i*box_size + loc_val) &
      - (FMIN - (ABS(FMIN) + 1)*EPS_I))&
      /SQRT(gbuffer(i*box_size + loc_diam))
    ! Compute the slope from the convex box 'i' to the convex box 'j'.
    slope = (gbuffer(i*box_size + loc_val) &
      - gbuffer(j*box_size + loc_val))&
      /(SQRT(gbuffer(i*box_size + loc_diam)) &
      - SQRT(gbuffer(j*box_size + loc_diam)))
    IF (sLope < slope_min) THEN
      ! When the slope between two convex boxes is smaller than
      ! 'slope_min', remove all boxes after the box 'i' and exit.
      gc_convex = i + 1
      EXIT
    ELSE ! Move down the indices for box 'i' and 'j'.
      i = j
      j = i + 1
    END IF
  END DO
 END IF
END IF
! Broadcast 'gc_convex' and 'gbuffer' to all subdomain masters, each of
! which will take its own portion of boxes to subdivide.
CALL MPI_BCAST(gc_convex, 1, MPI_INTEGER, 0, xm_comm, ierr)
CALL MPI_BCAST(gbuffer, gc_convex*box_size*size_r8, MPI_BYTE, 0, &
            xm_comm, ierr)
! Compute 'm_portion' for this subdomain master.
```

```
box_avg = DBLE(box_count_global) / DBLE(N_MASTER_I)
box_dev = (DBLE(box_count) - box_avg) / DBLE(box_count_global)
IF (gc_convex > N_MASTER_I ) THEN
  m_portion = FLOOR(DBLE(gc_convex)*(1.0 - box_dev) / DBLE(N_MASTER_I))
  ! Make sure that lead master is assigned at least one box.
  IF (N_MASTER_I > 1) THEN
    IF (mygid == 0) THEN
      CALL MPI_SEND(m_portion, 1, MPI_INTEGER, &
                  1, 0, xm_comm, ierr)
      IF (m_portion == 0) m_portion = 1
    ELSE IF (mygid == 1) THEN
      CALL MPI_RECV(tmp_portion, 1, MPI_INTEGER, &
                  0, 0, xm_comm, mstatus, ierr)
      IF (tmp_portion == 0) m_portion = m_portion - 1
    END IF
  END IF
  CALL MPI_ALLREDUCE(m_portion, j, 1, MPI_INTEGER, &
                   MPI_SUM, xm_comm, ierr)
  CALL RANDOM_SEED(SIZE = seed_size)
  ALLOCATE(seed(seed_size))
  seed = t + 101*(/ (i - 1, i = 1, seed_size) /)
  CALL RANDOM_SEED(PUT = seed)
  DO i = 1, gc_convex - j
    CALL RANDOM_NUMBER(harvest)
    tmp_rank = NINT(DBLE(N_MASTER_I - 1)*harvest)
    IF (mygid == tmp_rank) m_portion = m_portion + 1
  END DO
ELSE ! The number of boxes is smaller than N_MASTER_I.
  j = 1
  i = N_MASTER_I
  IF (mygid < gc_convex) THEN
    m_portion = 1
  ELSE
    m_portion = 0
  END IF
END IF
! If needed, reallocate 'lbuffer' to hold output from findsetI().
IF (lbuffer_len < m_portion*set_size) THEN
  DEALLOCATE(lbuffer)
  ALLOCATE(lbuffer(MAX(INT(2*lbuffer_len), m_portion*set_size)))
  lbuffer_len = SIZE(lbuffer)
END IF
IF (N_MASTER_I == 1) THEN ! There is one master.
  startbox = 1
  endbox = m_portion
ELSE IF (N_MASTER_I == 2) THEN !There are two masters.
  IF (mygid == 0) THEN
    startbox = 1
    endbox = m_portion
    CALL MPI_SEND(endbox + 1, 1, MPI_INTEGER, &
                1, 0, xm_comm, ierr)
  ELSE IF (mygid == 1) THEN
    CALL MPI_RECV(startbox, 1, MPI_INTEGER, &
                0, 0, xm_comm, mstatus, ierr)
    endbox = startbox + m_portion - 1
  END IF
ELSE ! There are three or more masters.
  IF (mygid == 0) THEN
    startbox = 1
    endbox = m_portion
    CALL MPI_SEND(endbox + 1, 1, MPI_INTEGER, &
                1, 0, xm_comm, ierr)
  END IF
  DO i = 1, N_MASTER_I - 2
    IF (mygid == i) THEN
      CALL MPI_RECV(startbox, 1, MPI_INTEGER, &
                  i - 1, 0, xm_comm, mstatus, ierr)
      endbox = startbox + m_portion - 1
      CALL MPI_SEND(endbox + 1, 1, MPI_INTEGER, &
                  i + 1, 0, xm_comm, ierr)
    END IF
  END DO
  IF (mygid == N_MASTER_I - 1) THEN
    CALL MPI_RECV(startbox, 1, MPI_INTEGER, &
                N_MASTER_I - 2, 0, xm_comm, mstatus, ierr)
    endbox = startbox + m_portion - 1
  END IF
END IF
IF ( m_portion > 0) THEN
  ! Go through 'gbuffer' three times to post-processing the convex boxes.
  ! First, mark the convex boxes that are originally local, because 'sid'
  ! and 'eid' may be changed in the following two operations.
  DO i = startbox, endbox
    startloc1 = (i - 1)*box_size
    IF (gbuffer(startloc1 + loc_pid) == myid) THEN
      ! This box is originally local, so mark the convex bit.
```

```
    sid = gbuffer(startloc1 + loc_sid)
    eid = gbuffer(startloc1 + loc_eid)
    p_iset => setInd
    DO j = 1, sid - 1
      p_iset => p_iset%next
    END DO
    p_iset%flags(eid) = IBSET(p_iset%flags(eid), CONVEX_BIT)
  END IF
END DO
! Second, remove boxes that have been assigned to other subdomain masters.
DO i = gc_convex, 1, -1
  startloc1 = (i - 1)*box_size
  IF ( (i < startbox) .OR. (i > endbox) ) THEN
    IF (gbuffer(startloc1 + loc_pid) == myid) THEN
      ! Delete the box that is assigned to another subdomain master.
      sid = gbuffer(startloc1 + loc_sid)
      eid = gbuffer(startloc1 + loc_eid)
      p_iset => setInd
      DO j = 1, sid - 1
        p_iset => p_iset%next
      END DO
      b_id = (p_iset%elements(eid) - 1)/col_w + 1
      mycol = MOD(p_iset%elements(eid) - 1, col_w) + 1
      CALL rmMat(b_id, mycol, eid, p_iset)
    END IF
  END IF
END DO
! Last, add boxes that are assigned from other subdomain masters.
DO i = startbox, endbox
  startloc1 = (i - 1)*box_size
  IF (gbuffer(startloc1 + loc_pid) /= myid) THEN
    tempbox%val = gbuffer(startloc1 + loc_val)
    tempbox%c(:) = gbuffer(startloc1 + loc_c:startloc1 + loc_c + N - 1)
    tempbox%side(:) = gbuffer(startloc1 + loc_side:startloc1 + &
      loc_side + N - 1)
    tempbox%diam = gbuffer(startloc1 + loc_diam)
    ! Since this box is from other master, check if need to update
    ! 'FMIN_I' and 'unit_x_i'.
    IF (tempbox%val < FMIN_I) THEN
      FMIN_I = tempbox%val
      unit_x_i(:) = tempbox%c(:)
      dia_i = tempbox%diam
    END IF
      IF (tempbox%val == FMIN_I) THEN
        IF ( (tempbox%diam < dia_i) .OR. ((tempbox%diam == dia_i) &
           .AND. (tempbox%c(:) .lexLT. unit_x_i)) ) THEN
          dia_i = tempbox%diam
          unit_x_i(:) = tempbox%c(:)
        END IF
      END IF
      ! Insert 'tempbox' to the box data structure.
      CALL insMat(tempbox, m_head, setDia, setInd, setFcol, iflag, 1)
      IF (iflag /= 0) RETURN
    END IF
  END DO
ELSE ! m_portion == 0, no convex hull boxes have been assigned.
  ! Remove all boxes that have been assigned to other subdomain masters.
  DO i = gc_convex, 1, -1
    startloc1 = (i - 1)*box_size
    IF (gbuffer(startloc1 + loc_pid) == myid) THEN
      ! Delete the box that is on another master.
      sid = gbuffer(startloc1 + loc_sid)
      eid = gbuffer(startloc1 + loc_eid)
      p_iset => setInd
      DO j = 1, sid - 1
        p_iset => p_iset%next
      END DO
      b_id = (p_iset%elements(eid) - 1)/col_w + 1
      mycol = MOD(p_iset%elements(eid) - 1, col_w) + 1
      CALL rmMat(b_id, mycol, eid, p_iset)
    END IF
  END DO
 END IF
END IF

IF (N_MASTER_I == 1) THEN ! Only one subdomain master.
 ! Preprocess for identifying potentially optimal hyperboxes of Step
 ! 3a for the next iteration. Find and process the hyperboxes which
 ! are on the convex hull if SWITCH_I== 1; otherwise, process the first
 ! box of each column until it reaches the one with 'FMIN'.
 p_iset => setInd
 gc_convex = 0
 OUTER: DO
  DO i = 1, p_iset%dim
    p_iset%flags(i) = IBSET(p_iset%flags(i), CONVEX_BIT)
    gc_convex = gc_convex + 1
```

```
      ! Check if the column has reached the one with 'FMIN'.
      b_id = (p_iset%elements(i) - 1)/col_w + 1
      col = MOD(p_iset%elements(i) - 1, col_w) + 1
      p_b => m_head
      DO j = 1, b_id -1
        p_b => p_b%child
      END DO
      IF (dia_i == p_b%M(1,col)%diam) EXIT OUTER
    END DO
    IF (ASSOCIATED(p_iset%next)) THEN
      p_iset => p_iset%next
    ELSE
      EXIT OUTER
    END IF
  END DO OUTER

  IF (SWITCH_I == 1) CALL findconvex(m_head, p_iset, i, setInd)
  ! Assign 'm_portion'.
  m_portion = gc_convex
  ! If needed, reallocate lbuffer to hold info from findsetI().
  IF (lbuffer_len < gc_convex*set_size) THEN
    DEALLOCATE(lbuffer)
    ALLOCATE(lbuffer(MAX(INT(2*lbuffer_len), gc_convex*set_size)))
    lbuffer_len = SIZE(lbuffer)
  END IF
END IF ! End of IF ( N_MASTER_I== 1 )

RETURN
END SUBROUTINE boxSelection

SUBROUTINE cleanup()
IMPLICIT NONE
! Clean up all data structures allocated to prevent a memory leak.
!
! On input: None.
!
! On output: None.
!
! Local variables.
INTEGER :: i, j  ! Loop counters.
TYPE(BoxMatrix), POINTER :: p_b, p_bm
TYPE(BoxLink), POINTER :: p_l
TYPE(int_vector), POINTER :: p_seti
TYPE(real_vector), POINTER :: p_setr

! Deallocate box links and box matrices starting from the first box matrix.
! First deallocate all box links associated with each box matrix, and
! finally deallocate the box matrix.
p_b => m_head
! Check all columns with box links which will be deallocated one by one
! starting from the last box link.
DO WHILE(ASSOCIATED(p_b))
  ! Check all the columns in 'p_b'.
  DO i = 1, col_w
    IF (p_b%ind(i) > row_w) THEN
      ! There must be box link(s). Chase to the last one and start
      ! deallocating them one by one.
      p_l => p_b%sibling(i)%p
      DO WHILE(ASSOCIATED(p_l%next))
       p_l => p_l%next
      END DO
      ! Found the last box link 'p_l'. Trace back and deallocate all links.
      DO WHILE(ASSOCIATED(p_l))
        IF (ASSOCIATED(p_l%prev)) THEN
          ! Its previous link is still a box link.
          p_l => p_l%prev
        ELSE
          ! There is no box link before it. This is the first box link of
          ! this column.
          DO j = 1, row_w
            DEALLOCATE(p_l%Line(j)%c)
            DEALLOCATE(p_l%Line(j)%side)
          END DO
          DEALLOCATE(p_l%Line)
          DEALLOCATE(p_l)
          EXIT
        END IF
        DO j = 1, row_w
          DEALLOCATE(p_l%next%Line(j)%c)
          DEALLOCATE(p_l%next%Line(j)%side)
        END DO
        DEALLOCATE(p_l%next%Line)
        DEALLOCATE(p_l%next)
      END DO
    END IF
  END DO
```

```
  ! Save the pointer of this box matrix for deallocation.
  p_bm => p_b
  ! Before it's deallocated, move to the next box matrix.
  p_b => p_b%child
  ! Deallocate this box matrix with all box links cleaned up.
  DEALLOCATE(p_bm%ind)
  DEALLOCATE(p_bm%sibling)
  DO i = 1, row_w
    DO j = 1, col_w
      DEALLOCATE(p_bm%M(i,j)%c)
      DEALLOCATE(p_bm%M(i,j)%side)
    END DO
  END DO
  DEALLOCATE(p_bm%M)
  DEALLOCATE(p_bm)
END DO

! Deallocate 'setB' and 'setW'.
DO i = 1, SIZE(setB%Line)
  DEALLOCATE(setB%Line(i)%c)
  DEALLOCATE(setB%Line(i)%side)
END DO
DEALLOCATE(setB%Line)
DEALLOCATE(setB%dir)
DEALLOCATE(setW%val)
DEALLOCATE(setW%dir)

! Deallocate nodes of 'setDia', 'setInd' and 'setFcol' starting from
! the last node.
p_setr => setDia
DO WHILE(ASSOCIATED(p_setr%next))
  p_setr => p_setr%next
END DO
! Found the last link pointed to by 'p_setr' of 'setDia', so deallocate
! links one by one until reaching the head node which has a null 'prev'.
DO
  DEALLOCATE(p_setr%elements)
  IF (p_setr%id /= 1) THEN
    p_setr => p_setr%prev
    DEALLOCATE(p_setr%next)
  ELSE
    DEALLOCATE(setDia)
    EXIT
  END IF
END DO
p_seti => setInd
DO WHILE(ASSOCIATED(p_seti%next))
  p_seti => p_seti%next
END DO
! Found the last link pointed to by 'p_seti' of 'setInd', so deallocate
! links one by one until reaching the head node which has a null 'prev'.
DO
  DEALLOCATE(p_seti%elements)
  DEALLOCATE(p_seti%flags)
  IF (p_seti%id /= 1) THEN
    p_seti => p_seti%prev
    DEALLOCATE(p_seti%next)
  ELSE
    DEALLOCATE(setInd)
    EXIT
  END IF
END DO
p_seti => setFcol
DO WHILE(ASSOCIATED(p_seti%next))
  p_seti => p_seti%next
END DO
! Found the last link pointed to by 'p_seti' of 'setFcol', so deallocate
! links one by one until reaching the head node that has a null 'prev'.
DO
  DEALLOCATE(p_seti%elements)
  IF (p_seti%id /= 1) THEN
    p_seti => p_seti%prev
    DEALLOCATE(p_seti%next)
  ELSE
    DEALLOCATE(setFcol)
    EXIT
  END IF
END DO

! Deallocate p_box.
DEALLOCATE(p_box%c)
DEALLOCATE(p_box%side)
DEALLOCATE(p_box)

! Deallocate tempbox
DEALLOCATE(tempbox%c)
```

```
DEALLOCATE(tempbox%side)
DEALLOCATE(tempbox)

RETURN
END SUBROUTINE cleanup

SUBROUTINE divide(b, setB, setDia, setInd, setFcol, p_box, setW, iflag)
IMPLICIT NONE
! Divide all boxes in 'setB'. Each box subdivision starts from
! the dimension with minimum w to the one with maximum w, where w is
! minf(c+delta), f(c-delta).
!
! On input:
! b       - The head link of box matrices.
! setB    - A set of 'HyperBox' type structures, each with newly sampled
!           center point coordinates and the corresponding function value.
!           After dividing, it contains complete boxes with associated side
!           lengths and the squared diameters.
! setDia  - A linked list of current different squared diameters of box
!           matrices. It's sorted from the biggest to the smallest.
! setInd  - A linked list of column indices corresponding to the different
!           squared diameters in 'setDia'.
! setFcol - A linked list of free columns in box matrices.
! p_box   - A 'HyperBox' type structure to hold removed parent box to
!           subdivide.
! setW    - A set of type 'ValList' used to sort wi's, where wi is defined as
!           minf(c+delta*ei), f(c-delta*ei), the minimum of function values
!           at the two newly sampled points.
!
! On output:
! b      - 'b' has the parent box removed and contains the newly formed boxes
!          after dividing the parent box.
! setB   - Cleared set of type 'BoxLine'. All newly formed boxes have been
!          inserted into 'b'.
! setDia - Updated linked list 'setDia' with new squared diameters of boxes,
!          if any.
! setInd - Updated linked list 'setInd' with new column indices corresponding
!          to newly added squared diameters in 'setDia'.
! setFcol - Updated linked list 'setFcol' with current free columns in 'b'.
! p_box  - A 'HyperBox' structure holding removed parent box to subdivide.
! setW   - 'setW' becomes empty after dividing.
! iflag  - status to return.
!          0    Normal return.
!          1    Allocation failures.
!
TYPE(BoxMatrix), INTENT(INOUT),TARGET :: b
TYPE(BoxLine), INTENT(INOUT) :: setB
TYPE(real_vector),INTENT(INOUT), TARGET :: setDia
TYPE(int_vector), INTENT(INOUT), TARGET :: setInd
TYPE(int_vector), INTENT(INOUT) :: setFcol
TYPE(HyperBox), INTENT(INOUT) :: p_box
TYPE(ValList), INTENT(INOUT) :: setW
INTEGER, INTENT(OUT) :: iflag

! Local variables.
INTEGER :: b_id ! Box matrix ID.
INTEGER :: id ! 'setInd' link ID.
INTEGER :: i, i1, j, j1, k, m ! Loop counters.
INTEGER :: leaf, maxid, maxpos ! Heap operation variables.
INTEGER :: mycol ! Box column ID.
INTEGER :: newbox_c ! Counter for new boxes.
INTEGER :: parent_i ! Parent element ID in a 'setInd' link.
INTEGER :: setlen ! Number of total new boxes.
INTEGER, DIMENSION(2*n) :: sortInd ! An array for sorting.
REAL(KIND = R8), DIMENSION(N) :: maxc ! Point coordinates with 'maxf'.
REAL(KIND = R8) :: maxf ! Maximum function value in a heap.
REAL(KIND = R8) :: temp ! Temporary variable.
TYPE(BoxMatrix), POINTER :: p_b ! Pointer to a box matrix.
TYPE(int_vector), POINTER :: p_iset ! Pointer to 'setInd' or 'setFcol' links.
TYPE(BoxLink), POINTER :: p_l, p_l1 ! Pointers to box links.

! Initialize 'iflag' for a normal return.
iflag = 0
! Loop subdividing each convex box in 'setB'.
newbox_c = 0
DO m = 0, gc_convex - 1
  setlen = 2*INT(lbuffer(m*set_size + 1))
  parent_i = INT(lbuffer(m*set_size + 2))
  id = INT(lbuffer(m*set_size + 3))
  ! Find the desired node of 'setInd'.
  p_iset => setInd
  DO i = 1, id - 1
    p_iset => p_iset%next
  END DO
  IF (p_iset%elements(parent_i) <= col_w) THEN
    ! This column is in the head link of box matrices.
```

```
  p_b => b
  mycol = p_iset%elements(parent_i)
ELSE
  ! Find the box matrix that contains this column.
  b_id = (p_iset%elements(parent_i) - 1)/col_w + 1
  mycol = MOD(p_iset%elements(parent_i) - 1, col_w) + 1
  p_b => b
  DO i = 1, b_id - 1
    p_b => p_b%child
  END DO
END IF
! Fill out 'setW'.
DO i = newbox_c + 1, newbox_c + setlen, 2
  ! Add minimum 'val' of a pair of newly sampled center points
  ! into 'setW'.
  setW%val((i - newbox_c + 1)/2) = MIN(setB%Line(i)%val, &
                                  setB%Line(i + 1)%val)
  setW%dir((i - newbox_c + 1)/2) = setB%dir(i)
END  DO
setW%dim = setlen/2

! Find the order of dimensions for further dividing by insertion
! sorting wi's in 'setW'.
DO i = 2, setW%dim
  DO j = i, 2, -1
    IF (setW%val(j) < setW%val(j - 1)) THEN
      ! Element j is smaller than element j-1, so swap 'val' and 'dir'.
      temp = setW%val(j)
      k = setW%dir(j)
      setW%val(j) = setW%val(j - 1)
      setW%dir(j) = setW%dir(j - 1)
      setW%val(j - 1) = temp
      setW%dir(j - 1) = k
    ELSE
      EXIT
    END IF
  END DO
END DO

! Sort the indices of boxes in 'setB' according to the dividing order in
! 'setW%dir'. Record the sorted indices in 'sortInd'.
DO i = 1, setW%dim
  DO j = newbox_c + 1, newbox_c + setlen, 2
    IF (setB%dir(j) == setW%dir(i)) THEN
      sortInd(2*i - 1) = j - newbox_c
      sortInd(2*i) = j - newbox_c + 1
    END IF
  END DO
END DO
! 'setW%dir' contains the order of dimensions to divide the parent box.
! Loop dividing on all dimensions in 'setW%dir' by setting up the new
! side lengths as 1/3 of parent box side lengths for each newly
! sampled box center.
DO i = 1, setW%dim
  temp = p_b%M(1, mycol)%side(setW%dir(i))/3.0_R8
  DO j = i, setW%dim
    setB%Line(newbox_c + sortInd(2*j - 1))%side(setW%dir(i)) = temp
    setB%Line(newbox_c + sortInd(2*j))%side(setW%dir(i)) = temp
  END DO
  ! Modify the parent's side lengths.
  p_b%M(1,mycol)%side(setW%dir(i))= temp
END DO
! Clear 'setW' for next time.
setW%dim = 0
! Move the parent box from box matrix 'p_b' to 'setB'.
p_box = p_b%M(1, mycol)
! Move the last box to the first position.
IF (p_b%ind(mycol) <= row_w) THEN
  ! There are no box links.
  IF (p_b%ind(mycol)>1) p_b%M(1, mycol) = p_b%M(p_b%ind(mycol), mycol)
ELSE
  ! There are box links. Chase to the last box link.
  p_l => p_b%sibling(mycol)%p
  DO i = 1, (p_b%ind(mycol) - 1)/row_w - 1
    p_l => p_l%next
  END DO
  p_b%M(1, mycol) = p_l%Line(p_l%ind)
  p_l%ind = p_l%ind - 1
END IF
! Update counters.
p_b%ind(mycol) = p_b%ind(mycol) - 1
! Adjust this box column to a heap by calling siftdown.
CALL siftdown(p_b, mycol, 1)
! Modify the diameter squared for the parent box temporarily saved in
! 'p_box', which will be stored at the end of 'setB' for insertion later.
p_box%diam = DOT_PRODUCT(p_box%side*UmL, p_box%side*UmL)
```

```
  setB%ind = setB%ind + 1
  setB%Line(setB%ind) = p_box
  ! Update 'dia' associated with 'FMIN' which has coordinates in 'unit_x'.
  IF (ALL(unit_x_i == p_box%c)) dia_i = p_box%diam
  ! Compute squared diameters for all new boxes in 'setB'.
  DO i = newbox_c + 1, newbox_c + setlen
    setB%Line(i)%diam = DOT_PRODUCT(setB%Line(i)%side*UmL, &
                                    setB%Line(i)%side*UmL)
    ! Update 'dia' if needed.
    IF (ALL(unit_x_i == setB%Line(i)%c)) THEN
      dia_i = setB%Line(i)%diam
    END IF
  END DO
  ! Update counter.
  newbox_c = newbox_c + setlen
END DO
! Update local box count.
box_count = box_count + newbox_c

! Add all new boxes and parent boxes in 'setB' to 'b'
! according to their squared diameters and function values.
DO i = 1, setB%ind
  CALL insMat(setB%Line(i), b, setDia, setInd, setFcol, iflag, 0)
  IF (iflag /= 0) RETURN
END DO
setB%ind = 0

! Scan through box columns to remove empty box columns, and squeeze box
! column lengths to MAX_ITER-t+1 if LBC (limiting box columns) is enabled.
p_iset => setInd
DO WHILE(ASSOCIATED(p_iset))
  i = 1
  DO
    IF (i > p_iset%dim) EXIT
    IF (p_iset%elements(i) <= col_w) THEN
      ! This column is in the head link of box matrices.
      p_b => b
      mycol = p_iset%elements(i)
    ELSE
      ! Find the box matrix that contains this column.
      b_id = (p_iset%elements(i) - 1)/col_w + 1
      mycol = MOD(p_iset%elements(i) - 1, col_w) + 1
      p_b => b
      DO j = 1, b_id - 1
        p_b => p_b%child
      END DO
    END IF
    IF (p_b%ind(mycol) == 0)  THEN
      ! This column is empty. Remove this diameter squared from a
      ! corresponding node of 'setDia'.
      CALL rmNode(col_w, p_iset%id - 1, i, setDia)
      ! Push the released column back to top of 'setFcol'.
      IF (setFcol%dim < col_w) THEN
        ! The head node of 'setFcol' is not full.
        CALL insNode(col_w, p_iset%elements(i), &
                     setFcol%dim + 1, setFcol)
      ELSE
        ! The head node is full. There must be at least one more node
        ! for 'setFcol'. Find the last non-full node of 'setFcol' to
        ! insert the released column.
        p_iset => setFcol%next
        DO
          IF (p_iset%dim < col_w) THEN
            ! Found it.
            CALL insNode(col_w, p_iset%elements(i), &
                         p_iset%dim + 1, p_iset)
            EXIT
          END IF
          ! Go to the next node.
          p_iset => p_iset%next
        END DO
      END IF
      ! Remove the column index from a corresponding node of 'setInd'.
      CALL rmNode(col_w, 0, i, p_iset)
      ! Decrease 'i' by 1 to compensate for this removal.
      i = i - 1
    ELSE ! This box column is not empty.
      IF (lbc == 1) THEN
        ! LBC is used, so adjust the box column length to MAX_ITER-t+1.
        IF (p_b%ind(mycol) > MAX_ITER - t + 1) THEN
          ! This box column has more boxes than needed for the remaining
          ! iterations. Loop removing the largest elements until the box
          ! column length equals MAX_ITER-t+1.
          DO
            IF (p_b%ind(mycol) == MAX_ITER - t + 1) EXIT
            ! Look for the largest element starting from the first leaf.
```

```
leaf = p_b%ind(mycol)/2 + 1
IF (leaf <= row_w) THEN
  ! The first leaf node starts inside M.
  maxid = 0
  maxpos = leaf
  maxf = p_b%M(leaf, mycol)%val
  maxc(:) = p_b%M(leaf, mycol)%c(:)
ELSE ! The first leaf node starts from a box link.
  maxid = (leaf - 1)/row_w
  maxpos = MOD(leaf - 1, row_w) + 1
  p_l => p_b%sibling(mycol)%p
  DO k = 1, maxid - 1
    p_l => p_l%next
  END DO
  maxf = p_l%Line(maxpos)%val
  maxc(:) = p_l%Line(maxpos)%c(:)
END IF
IF (maxid == 0) THEN
  ! Search starts from M.
  DO k = maxpos + 1, row_w
    IF (k > p_b%ind(mycol)) EXIT
    IF ((p_b%M(k, mycol)%val > maxf) .OR. &
      ((p_b%M(k, mycol)%val == maxf) .AND. &
      (maxc .lexLT. p_b%M(k, mycol)%c))) THEN
      maxf = p_b%M(k, mycol)%val
      maxc(:) = p_b%M(k, mycol)%c(:)
      maxpos = k
    END IF
  END DO
  p_l => p_b%sibling(mycol)%p
  k = 0
  j = 0
  i1 = 0
ELSE
  ! Start from a box link.
  k = maxid - 1
  j = maxpos
  i1 = maxid - 1
END IF
! Search through box links.
DO j1 = 1, (p_b%ind(mycol) - 1)/row_w - i1
  IF (.NOT. ASSOCIATED(p_l)) EXIT
  IF (p_l%ind == 0) THEN
    DEALLOCATE(p_l)
    EXIT
  END IF
  k = k + 1
  DO
    j = j + 1
    IF (j > p_l%ind) THEN
      ! Reset 'j' if it reached the end of the box link.
      j = 0
      EXIT
    END IF
    IF ((p_l%Line(j)%val > maxf) .OR. &
       ((p_l%Line(j)%val == maxf) .AND. &
       (maxc .lexLT. p_l%Line(j)%c))) THEN
      maxf = p_l%Line(j)%val
      maxc = p_l%Line(j)%c
      maxid = k
      maxpos = j
    END IF
  END DO
  p_l => p_l%next
END DO
! Found the largest fval box at maxid link and maxpos position.
! If maxpos is not the last element in the heap, replace it
! with the last element of the heap and sift it up.
IF (maxpos + maxid*row_w < p_b%ind(mycol)) THEN
  ! Locate the last heap element.
  IF (p_b%ind(mycol) <= row_w) THEN
    ! The last element is inside M.
    p_b%M(maxpos,mycol) = p_b%M(p_b%ind(mycol), mycol)
    ! Update the box counter for the box column.
    p_b%ind(mycol) = p_b%ind(mycol) - 1
    call siftup(p_b, mycol, maxpos)
  ELSE ! The last element is inside a box link.
    IF (maxid == 0) THEN
      ! The largest element is inside M.
      p_l1 => p_b%sibling(mycol)%p
      DO k = 1, (p_b%ind(mycol) - 1)/row_w - 1
        p_l1 => p_l1%next
      END DO
      p_b%M(maxpos,mycol) = &
        p_l1%Line(MOD(p_b%ind(mycol) - 1, row_w) + 1)
    ELSE ! The largest element is inside a box link.
```

```
            p_l => p_b%sibling(mycol)%p
            DO k = 1, maxid - 1
              p_l => p_l%next
            END DO
            p_l1 => p_l
            DO k = 1, (p_b%ind(mycol) - 1)/row_w - maxid
              p_l1 => p_l1%next
            END DO
            p_l%Line(maxpos) = &
              p_l1%Line(MOD(p_b%ind(mycol) - 1, row_w) + 1)
          END IF
          ! Update counters for the box link and the box column.
          p_l1%ind = p_l1%ind - 1
          p_b%ind(mycol) = p_b%ind(mycol) - 1
          ! Siftup the box that replaced the largest fval box.
          call siftup(p_b,mycol,maxid*row_w + maxpos)
        END IF
      ELSE ! The largest element is the last element.
        IF (maxid > 0) THEN
          ! Locate the box link that holds it.
          p_l => p_b%sibling(mycol)%p
          DO k =1, (p_b%ind(mycol) - 1)/row_w - 1
            p_l => p_l%next
          END DO
          p_l%ind = p_l%ind - 1
        END IF
        ! Update the counter.
        p_b%ind(mycol) = p_b%ind(mycol) - 1
      END IF
    END DO
  END IF
 END IF
 ! Go to the next box column.
 i = i + 1
END DO
! Go to the next 'setInd' link.
p_iset => p_iset%next
END DO

RETURN
END SUBROUTINE divide
```

```
FUNCTION findcol(i_start, p_start, index, do_it) RESULT(p_iset)
IMPLICIT NONE
! Find the rightmost column (setInd%elements(index)), in the plot of
! (box diameter, function value) points, with CONVEX_BIT of 'flags' set
! in linked list 'setInd', which indicates a potentially optimal box
! to be subdivided.
!
! On input:
! i_start - The index to start searching in node 'p_start'.
! p_start - The pointer to the node at which to start searching.
!
! On output:
! index    - The found index in node 'p_iset' of the linked list 'setInd'.
! do_it    - The returned sign to continue processing or not.
! i_start  - The index at which to resume searching in node 'p_start'
!            next time.
! p_start  - The pointer to the node at which to resume searching next time.
! p_iset   - The returned node, which contains the next box to subdivide.
!
INTEGER, INTENT(INOUT) :: i_start
TYPE(int_vector), POINTER :: p_start
INTEGER, INTENT(OUT) :: index
LOGICAL, INTENT(OUT) :: do_it
TYPE(int_vector), POINTER :: p_iset

! Local variables.
INTEGER :: i, start
TYPE(int_vector), POINTER :: p_set

do_it = .FALSE.
start = i_start
p_set => p_start
DO WHILE(ASSOCIATED(p_set))
  DO i = start, p_set%dim
    ! Find the first column with CONVEX_BIT set in 'flags' of 'p_set'.
    IF (BTEST(p_set%flags(i), CONVEX_BIT)) THEN
      ! Clear the CONVEX_BIT of 'flags' as being processed.
      p_set%flags(i) = IBCLR(p_set%flags(i), CONVEX_BIT)
      do_it = .TRUE.
      index = i
      p_iset => p_set
      ! Save them to i_start and p_start for resuming searching
      ! next time.
```

```
      i_start = i
      p_start => p_set
      EXIT
    END IF
  END DO
  ! There are no more box column with CONVEX_BIT set in 'flags' in
  ! this node. Go to the next one.
  IF (.NOT. do_it) THEN
    p_set => p_set%next
    ! Reset 'start' to be 1 for all the following iterations except the
    ! first one which resumed from 'i_start'.
    start = 1
  ELSE
    EXIT
  END IF
END DO

RETURN
END FUNCTION findcol

SUBROUTINE findconvex(b, p_fmin, i_fmin, setInd)
IMPLICIT NONE
! In 'setInd', clear CONVEX_BIT of columns if the first boxes on these
! columns are not on convex hull. Bit CONVEX_BIT with value 0 indicates
! the first box on the column is not one of potentially optimal boxes.
! This is determined by comparing slopes. When a single master is used with
! EPS_I==0 or when multiple masters are used in a subdomain, starting from
! the first column, find the maximum slope from the first box on that column
! to the first boxes on all other columns until reaching the box with 'FMIN'.
! Then, starting from the next column with the first box on convex hull,
! repeat the procedure until no more columns before the column with 'FMIN'
! to check. In case of a single master, if EPS_I is greater than 0, the outer
! loop breaks out when the maximum slope is less than the value:
! (val - (FMIN - EPS_I))/diam.
!
! On input:
! b     - The head link of box matrices.
! p_fmin - Pointer to the node holding the column index of the box with
!         'FMIN' (a single master only).
! i_fmin - Index of the column in the node 'p_fmin' (a single master only).
! setInd - A linked list holding column indices of box matrices.
!
! On output:
! setInd - 'setInd' has the modified column indices.
!
TYPE(BoxMatrix), INTENT(IN), TARGET :: b
TYPE(int_vector), POINTER :: p_fmin
INTEGER, INTENT(IN) :: i_fmin
TYPE(int_vector), INTENT(INOUT), TARGET :: setInd

! Local variables.
INTEGER :: b_id1, b_id2, col1, col2, i, j, k, target_i
LOGICAL :: stop_fmin
REAL(KIND = R8) :: slope, slope_max
TYPE(BoxMatrix), POINTER :: p_b1, p_b2
TYPE(int_vector), POINTER :: p_iset1, p_iset2, target_set

! Initialize the first node pointer.
p_iset1 => setInd
IF (p_iset1%dim == 0) RETURN ! Exit when 'setInd' is empty.

! Initialization for outer loop which processes all columns before
! the column containing 'FMIN' in order to find a convex hull curve.
stop_fmin = .FALSE.
i = 1
k = 1
OUTLOOP: DO WHILE((.NOT. stop_fmin) .AND. ASSOCIATED(p_iset1))
  ! Initialization for inner loop, which computes the slope from the first
  ! box on the fixed column 'i' to the first boxes on all the other columns,
  ! before reaching the column containing a box with 'FMIN', to locate the
  ! target column with maximum slope. Mark off any columns in between the
  ! fixed first column and the target column.
  NULLIFY(target_set)
  slope_max = -HUGE(slope)
  p_iset2 => p_iset1
  ! Fix the first convex hull column as column 'i' in 'p_iset1'.
  ! The second column used to calculate the slope has index 'k' in
  ! 'p_iset2'.  'k' is incremented up to the column index corresponding
  ! to 'FMIN'.  Find the box matrix 'p_b1' and the local column index
  ! 'col1'.
  b_id1 =(p_iset1%elements(i) - 1)/col_w + 1
  col1 = MOD(p_iset1%elements(i) - 1, col_w) + 1
  p_b1 => b
  DO j = 1, b_id1 - 1
    p_b1 => p_b1%child
  END DO
```

```
! Check if the first column has reached the column with 'FMIN'. If
! so, break out of the outer loop.
IF (dia_i == p_b1%M(1, col1)%diam) EXIT OUTLOOP
k = i + 1
INLOOP: DO
  IF (k > p_iset2%dim) THEN
    ! Move to the next node as k increments beyond the maximum
    ! length for each node of 'setInd'.
    p_iset2 => p_iset2%next
    IF (.NOT.ASSOCIATED(p_iset2)) EXIT INLOOP
    IF (p_iset2%dim == 0) EXIT INLOOP
    ! Reset 'k' for the next box link.
    k = 1
  END IF
  ! To compute the slope from the first box on column 'i' of 'p_iset1' to
  ! the first box on column 'k' of 'p_iset2', find the local column index
  ! 'col2' and the corresponding box matrix 'p_b2'.
  b_id2 = (p_iset2%elements(k) - 1)/col_w + 1
  col2 = MOD(p_iset2%elements(k) - 1, col_w) + 1
  p_b2 => b
  DO j = 1, b_id2 -1
    p_b2 => p_b2%child
  END DO
  ! Use the slope formula (f1 - f2)/(d1 - d2), where f1 and f2 are the
  ! function values at the centers of the two boxes with diameters
  ! d1 and d2.
  slope = (p_b1%M(1, col1)%val - p_b2%M(1, col2)%val) / &
          (SQRT(p_b1%M(1, col1)%diam) - SQRT(p_b2%M(1, col2)%diam))
  ! Compare the new slope with the current maximum slope. Keep track
  ! of the target column index and the target node.
  IF (slope > slope_max) THEN
    slope_max = slope
    target_i = k
    target_set => p_iset2
    ! Check if this target column contains 'FMIN'.
    IF (dia_i == p_b2%M(1, col2)%diam) stop_fmin = .TRUE.
  END IF
  ! IF the target column contains 'FMIN', break out the inner loop.
  IF (dia_i == p_b2%M(1, col2)%diam) EXIT INLOOP
  ! Move on to the next column.
  k = k + 1
END DO INLOOP
! Mark off boxes in between.
  IF (ASSOCIATED(target_set)) CALL markoff(i, target_i, p_iset1, target_set)
  IF (N_MASTER_I == 1) THEN
    ! When a single master is used for convex hull processing,
    ! check if EPS_I /= 0. If so, it stops if the found 'slope_max' from
    ! the first box is less than the desired accuracy of the solution.
    IF ((EPS_I /= 0) .AND. ASSOCIATED(target_set)) THEN
      IF ((p_b1%M(1, col1)%val - (FMIN - (ABS(FMIN) + 1)*EPS_I))/ &
         SQRT(p_b1%M(1, col1)%diam) > slope_max ) THEN
        ! Mark off the first boxes on the columns from the column target_i to
        ! the one with 'FMIN'.
        target_set%flags(target_i) = IBCLR(target_set%flags(target_i), &
                                           CONVEX_BIT)
        gc_convex = gc_convex - 1
        CALL markoff(target_i, i_fmin, target_set, p_fmin)
        IF (BTEST(p_fmin%flags(i_fmin), CONVEX_BIT)) THEN
          ! Mark off the first box on the column with 'FMIN' if it is not
          ! marked off yet as the target_set.
          p_fmin%flags(i_fmin) = IBCLR(p_fmin%flags(i_fmin), CONVEX_BIT)
          gc_convex = gc_convex - 1
        END IF
        EXIT OUTLOOP
      END IF
    END IF
  END IF
  ! To start the next pass, the first fixed column jumps to the target column
  ! just found which is the next column on convex hull.
  i = target_i
  p_iset1 => target_set
END DO OUTLOOP

RETURN
END SUBROUTINE findconvex

SUBROUTINE findsetI(b, col)
IMPLICIT NONE
! Fill out 'lbuffer', holding dimensions with the maximum side length
! of the first boxes on 'col' in box matrix links 'b'.
!
! On input:
! b   - The head link of box matrices.
! col - The global column index of box matrix links.
!
! On output: None.
```

```
!
TYPE(BoxMatrix), INTENT(IN), TARGET :: b
INTEGER, INTENT(IN) :: col

! Local variables.
INTEGER :: b_id, i, j, pos
REAL(KIND = R8)  :: temp
TYPE(BoxMatrix), POINTER :: p_b

! Find the box matrix link that 'col' is associated with.
IF (col <= col_w) THEN
  p_b => b
  j = col
ELSE
  b_id = (col - 1)/col_w + 1
  j = MOD(col - 1, col_w) + 1
  p_b => b
  DO i = 1, b_id - 1
    p_b => p_b%child
  END DO
END IF

! Search for the maximum side length.
temp = MAXVAL(p_b%M(1, j)%side(:))
! Find all the dimensions with the maximum side length.
DO i = 1, N
  pos = lbuffer(gc_convex*set_size + 1)
  IF ((ABS(p_b%M(1, j)%side(i) - temp)/temp) <= EPS4N) THEN
    ! Add it to 'lbuffer'.
    lbuffer(gc_convex*set_size + 5 + pos) = i
    lbuffer(gc_convex*set_size + 1) =  pos + 1
  END IF
END DO

RETURN
END SUBROUTINE findsetI

SUBROUTINE init(b, status, role)
IMPLICIT NONE
! Allocate the arrays and initializes the first center point.
! Evaluate the function value at the center point and initializes
! 'FMIN' and 'unit_x'.
!
! On input:
! role   - The role of initialization. If 0, it initializes as the root
!          subdomain master. If 1, it is a nonroot subdomain master.
!
! On output
! b      - The first box matrix to initialize.
! status - Status of return.
!          =0    Successful.
!          >0    Allocation or restart related error.
!
TYPE(BoxMatrix), INTENT(OUT), TARGET :: b
INTEGER, INTENT(OUT):: status
INTEGER, INTENT(IN) :: role

! Local variables.
INTEGER :: i, iflag

! Normal status.
status = 0
iflag = 0

! Allocate arrays.
ALLOCATE(b%M(row_w, col_w), STAT = status)
IF (status /= 0) THEN; status=ALLOC_ERROR; RETURN; END IF
ALLOCATE(b%ind(col_w), STAT = status)
IF (status /= 0) THEN; status=ALLOC_ERROR; RETURN; END IF
! Clear the box counter for each column.
b%ind(:) = 0
! Nullify the child link to the next box matrix.
NULLIFY(b%child)
ALLOCATE(b%sibling(col_w), STAT = status)
IF (status /= 0) THEN; status=ALLOC_ERROR; RETURN; END IF
DO i = 1, col_w
  NULLIFY(b%sibling(i)%p)
END DO
DO i = 1, row_w
  DO j = 1, col_w
    ALLOCATE(b%M(i, j)%c(N), STAT = status)
    IF (status /= 0) THEN; status=ALLOC_ERROR; RETURN; END IF
    ALLOCATE(b%M(i, j)%side(N), STAT = status)
    IF (status /= 0) THEN; status=ALLOC_ERROR; RETURN; END IF
  END DO
END DO
```

```
! Initialize the center of the first unit hypercube in box matrix 'b'
! and 'unit_x' in the normalized coordinate system.
IF (role == 0 .AND. .NOT. spawned) THEN ! The root subdomain master.
  ! Initialize 'setFcol' starting from the last column, push free columns
  ! to 'setFcol'.
  DO i = 1, col_w - 1
    setFcol%elements(i) = col_w - (i - 1)
  END DO
  setFcol%dim = col_w - 1
  b%M(1, 1)%c(:) = 0.5_R8
  b%M(1, 1)%side(:) = 1.0_R8
  unit_x_i(:) = 0.5_R8
  unit_x(:) = unit_x_i
  IF (RESTART_I == 2) THEN ! Recover from the checkpoint logs.
    IF (chk_sm /= N_MASTER_I) THEN ! Recover from the root's checkpoint file.
      ! Read the sub-header.
      READ(UNIT=CHKSUNIT + 1, IOSTAT=ierr) check_t, chk_len(1)
      IF (ierr /= 0) THEN; status=FILE_ERROR + 4; RETURN; END IF
      ! Recover the first log.
      READ(UNIT=CHKSUNIT + 1, IOSTAT=ierr) X, tmpf
      IF (ierr /= 0) THEN; status=FILE_ERROR + 4; RETURN; END IF
      IF (ALL(X == b%M(1, 1)%c)) THEN
        FMIN_I = tmpf
      ELSE
        status = FILE_ERROR + 4
        RETURN
      END IF
      ! Open a new checkpoint file to append the new sequence of evaluations.
      OPEN(UNIT=CHKRUNIT + chk_sm, FILE=cpfile2, FORM="UNFORMATTED", &
          STATUS="UNKNOWN", POSITION="APPEND", IOSTAT=ierr)
      IF (ierr /= 0) THEN; status=FILE_ERROR; RETURN; END IF
      WRITE(UNIT=CHKRUNIT + chk_sm, IOSTAT=ierr) t, 1
      IF (ierr /= 0) THEN; status=FILE_ERROR + 3; RETURN; END IF
      WRITE(UNIT=CHKRUNIT + chk_sm, IOSTAT=ierr) b%M(1,1)%c, FMIN_I
      IF (ierr /= 0) THEN; status=FILE_ERROR + 3; RETURN; END IF
      CLOSE(CHKRUNIT + chk_sm)
    ELSE ! Recover from the list of logs.
      IF (.NOT. inList(b%M(1, 1)%c, FMIN_I)) THEN
        status = FILE_ERROR + 4
      END IF
    END IF
  ELSE ! Evaluate objective function at 'c'.
    ! Store the function value and initialize 'FMIN_I' at root.
    iflag = 0
    FMIN_I = OBJ_FUNC(L + b%M(1, 1)%c(:)*UmL, iflag)
    ! Check the iflag to deal with undefined function values.
    IF (iflag /= 0) THEN
      ! It is evaluated at an undefined point, so assign it a huge value.
      FMIN_I = HUGE(1.0_R8)
    END IF
    IF (RESTART_I == 1) THEN
      OPEN(UNIT=CHKSUNIT, FILE=cpfile, FORM="UNFORMATTED", STATUS="UNKNOWN",&
          POSITION="APPEND", IOSTAT=ierr)
      IF (ierr /= 0) THEN; iflag = FILE_ERROR; RETURN; END IF
      ! Write a function evaluation log to the checkpoint file and
      ! the sub-header consisting of t and setB size.
      WRITE(UNIT=CHKSUNIT, IOSTAT=ierr) t, 1
      IF (ierr /= 0) THEN; iflag = FILE_ERROR + 2; RETURN; END IF
      WRITE(UNIT=CHKSUNIT, IOSTAT=ierr) b%M(1,1)%c, FMIN_I
      IF (ierr /= 0) THEN; iflag = FILE_ERROR + 2; RETURN; END IF
      CLOSE(CHKSUNIT)
    END IF
  END IF
  b%M(1, 1)%val = FMIN_I
  FMIN = FMIN_I
  eval_c_i = 1
  eval_c = 0
  ! Initialize the diameter squared for this box and 'dia',
  ! the diameter squared associated with 'FMIN_I'.
  dia_i = DOT_PRODUCT(b%M(1, 1)%side*UmL, b%M(1, 1)%side*UmL)
  b%M(1, 1)%diam = dia_i
  dia = dia_i
  ! Initialize the 'ind' for the first column and 'id' for this box matrix.
  b%ind(1) = 1
  b%id = 1
  ! Initialize 'setDia' and 'setInd'.
  setDia%dim = 1
  setDia%elements(1) = m_head%M(1,1)%diam
  ! Set the first box as being on convex hull by setting the CONVEX_BIT
  ! of the 'flags'.
  setInd%dim = 1
  setInd%elements(1) = 1
  setInd%flags(1) = IBSET(setInd%flags(1), CONVEX_BIT)
ELSE
! A nonroot subdomain master has no box in the data structure and will
```

```
! receive convex boxes from the root subdomain master.
  IF ( (RESTART_I == 2) .AND. (chk_sm /= N_MASTER_I) ) THEN
    ! Read the first point to bypass it for MAIN_LOOP.
    READ(UNIT=CHKSUNIT + 1, IOSTAT=ierr) check_t, chk_len(1)
    IF (ierr /= 0) THEN; status=FILE_ERROR + 4; RETURN; END IF
    ! Recover the first log.
    READ(UNIT=CHKSUNIT + 1, IOSTAT=ierr) X, tmpf
    IF (ierr /= 0) THEN; status=FILE_ERROR + 4; RETURN; END IF
  END IF
  b%id = 1
  eval_c_i = 0
  unit_x_i(:) = 0.0_R8
  unit_x(:) = unit_x_i(:)
  FMIN_I = HUGE(1.0_R8)
  dia_i = HUGE(1.0_R8)
  DO i = 1, col_w
    setFcol%elements(i) = col_w - (i - 1)
  END DO
  setFcol%dim = col_w
END IF

RETURN
END SUBROUTINE init

FUNCTION inQueue(list, s, e) RESULT(ans)
IMPLICIT NONE
! Return TRUE if 'e' is in 'list' with length 's'. Otherwise, return
! FALSE.
!
! On input:
! list - A list of integers.
! s    - The length of 'list' under consideration.
! e    - The element to be located in 'list'.
!
! On output:
! ans   - Answer to return.
!         .TRUE.    'e' is found in the range [1,s] of 'list'.
!         .FALSE.   Otherwise.
!
INTEGER, DIMENSION(:), INTENT(IN) :: list
INTEGER, INTENT(IN) :: s
INTEGER, INTENT(IN) :: e
LOGICAL :: ans

! Local variable
INTEGER :: i

ans = .FALSE.
DO i = 1, s
  IF (list(i) == e) THEN; ans = .TRUE.; EXIT; END IF
END DO

RETURN
END FUNCTION inQueue

SUBROUTINE markoff(i, target_i, p_iset1, target_set)
IMPLICIT NONE
! Mark off columns in between the column 'i' of 'p_iset1' and column
! 'target_i' of 'target_set' by clearing the CONVEX_BIT in 'flags'.
!
! On input:
! i         - Column index of the first box for computing slope in findconvex.
! target_i   - Column index of the second box for computing slope in
!              findconvex.
! p_iset1  - The node of 'setInd' holding the column 'i'.
! target_set - The node of 'setInd' holding the column 'target_i'.
!
! On output:
! p_iset1  - 'p_iset1' has changed column indices.
! target_set - 'target_set' has changed column indices.
!
INTEGER, INTENT(IN) :: i
INTEGER, INTENT(IN) :: target_i
TYPE(int_vector), INTENT(INOUT), TARGET :: p_iset1
TYPE(int_vector), POINTER :: target_set

! Local variables.
INTEGER :: j
TYPE(int_vector), POINTER :: p_set

! Check if any columns in between.
IF (ASSOCIATED(target_set, p_iset1)) THEN
  ! If 'target_i' is next to column 'i' or no columns in between, return.
  IF ((i == target_i) .OR. (i + 1 == target_i)) RETURN
END IF
```

```
! Clear all CONVEX_BITs in 'flags' in between.
j = i
p_set => p_iset1
DO
  j = j + 1
  IF (j > p_set%dim) THEN
    p_set => p_set%next
    IF (.NOT. ASSOCIATED(p_set)) EXIT
    IF (p_set%dim == 0) EXIT
    j = 1
  END IF
  ! Check if at the target node.
  IF (ASSOCIATED(target_set, p_set)) THEN
    ! If 'j' has reached 'target_i', exit.
    IF (j == target_i) EXIT
  END IF
  ! Clear the CONVEX_BIT of 'flags' for column 'j' of 'p_set'.
  p_set%flags(j) = IBCLR(p_set%flags(j), CONVEX_BIT)
  gc_convex = gc_convex - 1
END DO
END SUBROUTINE markoff

SUBROUTINE master(iflag)
IMPLICIT NONE
! A master is responsible for box SELECTION and DIVISION. During
! SELECTION, it selects convex hull boxes to be subdivided. When more
! than one subdomain masters exist, they collaborate with each other
! to select a set of global convex hull boxes in parallel. Then
! the master samples new points around convex hull boxes and obtain
! their function values either by local computation when no workers
! are present or by distributed computation among workers (SAMPLING).
! Lastly, it subdivides the convex hull boxes according to function
! values obtained (DIVISION). These three steps repeat until the
! stopping condition is satisfied.
!
! On input: None.
!
! On output:
! iflag        - Status to return.
!                0    Normal return.
!                >0   Error return.
!
INTEGER, INTENT(OUT) :: iflag

! Local variables.
INTEGER :: i, j ! Loop counters.
INTEGER :: m_portion ! Number of global convex hull boxes for this master.
INTEGER :: setB_c ! Counter for elements that will be stored in 'setB'.
INTEGER :: setB_len ! Length of 'setB' buffer.
LOGICAL :: good_ratio ! True if the ratio of masters to workers will be
  ! acceptable after spawning new masters.
LOGICAL :: order ! Used to determine the ordering of ranks in the merged
  ! parent-child intracommunicator.
LOGICAL :: spawn_if_lowmem ! Flag specifying if new masters should be spawned
  ! when the amount of avialable memory falls below a certain threshold.
REAL(KIND = R8) :: mem_threshold ! If available memory for a master falls below
  ! this value, then a local spawn request flag is set.
REAL(KIND = R8), DIMENSION(:), ALLOCATABLE :: tmpbox ! Temporary buffer with
  ! length 'box_size'.
REAL(KIND = R8) :: tmpf ! Temporary function value for recovering checkpoints.
REAL(KIND = R8) :: totl_mem ! The total amount of memory (in MB) available on a
  ! processor.
TYPE(int_vector), POINTER :: p_iset ! Pointer to a node of 'int_vector'
  ! type data structures, i.e. 'setInd' and 'setFcol

! Initialize 'iflag'.
iflag = 0

! Initialize 'totl_mem', 'mem_threshold', and 'switch_iter'.
spawn_if_lowmem = PRESENT(TOTAL_MEM)
IF (spawn_if_lowmem) THEN
  totl_mem = REAL(TOTAL_MEM, R8)
  mem_threshold = totl_mem / 2.0_R8
END IF
switch_iter = 1

! Set 'xm_comm'.
IF (.NOT. spawned) THEN
  IF (world_size == 1) THEN
    xm_comm = MPI_COMM_WORLD
  ELSE
    xm_comm = m_comm(INT(myid/N_MASTER_I) + 1)
  END IF
END IF

! Assign 'row_w' and 'col_w' in terms of N.
```

```
IF (N <= 10) THEN
  row_w = MAX(10, 2*N)
ELSE
  row_w = 17 + CEILING(LOG(REAL(N))/LOG(2.0))
END IF
IF (lbc == 1) THEN
  ! Limit 'row_w' to be not greater than 'MAX_ITER'.
  IF (row_w > MAX(MAX_ITER,2)) row_w = MAX(MAX_ITER,2)
END IF
col_w = 35*N
! Assign parameter values and allocate buffers.
box_size = 5 + N*2
set_size = 4 + N
loc_side = N + loc_c
loc_diam = 2*N + loc_c

IF (N_MASTER_I > 1) THEN
  ! Only with multiple subdomain masters, 'gbuffer' and 'displs' are
  ! needed. Initially, allocate 100*N_MASTER_I slots for boxes in 'gbuffer'.
  gbuffer_len = 100*N_MASTER_I*box_size
  IF (ALLOCATED(gbuffer)) DEALLOCATE(gbuffer)
  ALLOCATE(gbuffer(gbuffer_len), STAT = alloc_err)
  IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 3; RETURN; END IF
  gbuffer(:) = 0
  IF (ALLOCATED(displs)) DEALLOCATE(displs)
  ALLOCATE(displs(N_MASTER_I), STAT = alloc_err)
  IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 3; RETURN; END IF
END IF
lbuffer_len = 100*box_size
IF (ALLOCATED(lbuffer)) DEALLOCATE(lbuffer)
ALLOCATE(lbuffer(lbuffer_len))
lbuffer(:) = 0
! Initially allocate 'setB' with at least 1000 elements to reduce
! the number of reallocations.
setB_len = 1000
IF (ASSOCIATED(setB%Line)) NULLIFY(setB%Line)
ALLOCATE(setB%Line(setB_len), STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 1; RETURN; END IF
IF (ASSOCIATED(setB%dir)) NULLIFY(setB%dir)
ALLOCATE(setB%dir(setB_len),STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 1; RETURN; END IF
DO i = 1, setB_len
  IF (ASSOCIATED(setB%Line(i)%c)) NULLIFY(setB%Line(i)%c)
  ALLOCATE(setB%Line(i)%c(N), STAT = alloc_err)
  IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 1; RETURN; END IF
    IF (ASSOCIATED(setB%Line(i)%side)) NULLIFY(setB%Line(i)%side)
    ALLOCATE(setB%Line(i)%side(N), STAT = alloc_err)
  IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 1; RETURN; END IF
END DO
! Allocate 'tmpbox'.
IF (ASSOCIATED(tempbox)) NULLIFY(tempbox)
ALLOCATE(tmpbox(box_size), STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 3; RETURN; END IF
! Allocate 'lc_convex'.
IF (ALLOCATED(lc_convex)) DEALLOCATE(lc_convex)
ALLOCATE(lc_convex(N_MASTER_I), STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 3; RETURN; END IF
lc_convex(:) = 0

! Allocate 'setW'.
IF (ASSOCIATED(setW%val)) NULLIFY(setW%val)
ALLOCATE(setW%val(N), STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 2; RETURN; END IF
ALLOCATE(setW%dir(N), STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 2; RETURN; END IF
setW%dim = 0

! Allocate 'setDia', 'setInd', and 'setFcol' for the first box matrix.
ALLOCATE(setDia)
ALLOCATE(setDia%elements(col_w), STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 2; RETURN; END IF
NULLIFY(setDia%next)
NULLIFY(setDia%prev)
setDia%id = 1
setDia%dim = 0
ALLOCATE(setInd)
ALLOCATE(setInd%elements(col_w), STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 2; RETURN; END IF
ALLOCATE(setInd%flags(col_w), STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 2; RETURN; END IF
setInd%flags(:) = 0
NULLIFY(setInd%next)
NULLIFY(setInd%prev)
setInd%id = 1
setInd%dim = 0
ALLOCATE(setFcol)
```

```
ALLOCATE(setFcol%elements(col_w), STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 2; RETURN; END IF
NULLIFY(setFcol%next)
NULLIFY(setFcol%prev)
NULLIFY(setFcol%flags)
setFcol%id = 1
setFcol%dim = 0

! Allocate p_box.
ALLOCATE(p_box)
ALLOCATE(p_box%c(N), STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 3; RETURN; END IF
ALLOCATE(p_box%side(N), STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 3; RETURN; END IF

! Allocate tempbox.
ALLOCATE(tempbox)
ALLOCATE(tempbox%c(N),STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 3; RETURN; END IF
ALLOCATE(tempbox%side(N), STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 3; RETURN; END IF

! Initialize 'm_portion' and 't'.
IF (.NOT. spawned) THEN
  m_portion = 0
  t = 0
END IF

! Step 1: Checkpoint file/log management, normalization of the search
!        space, and initialization of first hyperbox.
ALLOCATE(m_head, STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR; RETURN; END IF

! Initialize the checkpointing mechanism.
CALL initCheckpoint()

! Initialization for main loop.
CALL init(m_head, iflag, mygid)
IF (iflag /= 0) RETURN

! Notify all others that it has passed the initialization. Use 'b_worker'
! as the send buffer and 'q_worker' as the receive buffer.
b_worker(:) = 1
IF (.NOT. spawned) THEN
  CALL MPI_ALLTOALL(b_worker, 1, MPI_INTEGER, q_worker, 1, &
                    MPI_INTEGER, MPI_COMM_WORLD, ierr)

  IF (SUM(q_worker) /= world_size) THEN
    ! At least one processor did not pass the initialization.
    iflag = DMPI_ERROR + 7
    RETURN
  END IF
END IF
b_worker(:) = 0
q_worker(:) = 0

! The root subdomain master has one box initially.
IF (.NOT. spawned) THEN
  IF (mygid == 0) m_portion = 1
  t = 1
END IF

! Spawned masters update state.
IF (spawned) THEN
  ! Initialize state transfer. In particular, receive values for 'N_MASTER_I'
  ! and 'world_size'.
  CALL initStateTransfer()
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, num_children, ierr)
  ! Determine 'N_WORKER_I' allocate rank arrays used for merging the parent and
  ! child intercommunicators.
  N_WORKER_I = world_size - N_MASTER_I
  IF (ALLOCATED(master_ranks)) THEN
    DEALLOCATE(master_ranks)
  END IF
  ALLOCATE(master_ranks(N_MASTER_I))
  IF (ALLOCATED(worker_ranks)) THEN
    DEALLOCATE(worker_ranks)
  END IF
  ALLOCATE(worker_ranks(N_WORKER_I))
  IF (ALLOCATED(trans_ranks)) THEN
    DEALLOCATE(trans_ranks)
  END IF
  ALLOCATE(trans_ranks(1 + num_children))
  ! Assign ranks to the old masters.
  DO i = 1, N_MASTER_I - num_children
    master_ranks(i) = i - 1
```

```
END DO
! Assign ranks to the new masters beginning at 'world_size - 1'.
DO i = 1, num_children
  master_ranks((N_MASTER_I - num_children) + i) &
    = (world_size - num_children) + (i - 1)
END DO
! Assign 'worker_ranks'.
DO i = 1, N_WORKER_I
  worker_ranks(i) = (N_MASTER_I - num_children) + (i - 1)
END DO
! Assign state transfer ranks.
trans_ranks(1) = 0
DO i = 1, num_children
  trans_ranks(i + 1) = (world_size - num_children) + (i - 1)
END DO
! Merge parent and child intercommunicators.
order = .TRUE.
CALL MPI_INTERCOMM_MERGE(parent_comm, order, temp_comm_world, ierr)
! Create expanded communicator for new set of masters.
CALL MPI_COMM_GROUP(temp_comm_world, temp_group_world, ierr)
CALL MPI_GROUP_INCL(temp_group_world, N_MASTER_I, master_ranks, &
                xm_group, ierr)
CALL MPI_COMM_CREATE(temp_comm_world, xm_group, xm_comm, ierr)
! Create expanded communicator for new world (= old world + spawned masters).
CALL MPI_GROUP_INCL(temp_group_world, N_WORKER_I, worker_ranks, &
                temp_group_worker, ierr)
CALL MPI_GROUP_UNION(xm_group, temp_group_worker, xw_group, ierr)
CALL MPI_COMM_CREATE(temp_comm_world, xw_group, xw_comm, ierr)
! Create communicator for state transfer.
CALL MPI_GROUP_INCL(temp_group_world, 1 + num_children, trans_ranks, &
                trans_group, ierr)
CALL MPI_COMM_CREATE(temp_comm_world, trans_group, trans_comm, ierr)
! Get new ranks.
CALL MPI_COMM_RANK(xw_comm, myid, ierr)
CALL MPI_COMM_RANK(xm_comm, mygid, ierr)
! Update 'PROCID' and 'N_MASTER'.
PROCID = myid
N_MASTER = N_MASTER_I
! Call the state update subroutine.
CALL bcastState()
! Set the iteration at which the last spawning event occured.
switch_iter = t
! Wait for all masters to synchronize.
  CALL MPI_BARRIER(xw_comm, ierr)
END IF

MAIN_LOOP: DO
  ! If there is a global spawn request, then prepare for spawning.
  IF (spawn_requested_global) THEN
    ! Send spawn request messages to workers.
    IF (mygid == 0) THEN
      DO i = N_SUB_I*N_MASTER_I, N_SUB_I*N_MASTER_I + (N_WORKER_I - 1)
        CALL MPI_SEND(buffer, bufsize, MPI_BYTE, i, SPAWN_REQ, &
                    xw_comm, ierr)
      END DO
    END IF
    ! Initiate the spawning procedure.
    CALL spawnInit(ierr)
    IF (ierr /= 0) THEN
      ! Abort the spawning procedure.
      spawn_initiated = .FALSE.
    END IF
  END IF
  ! If span initialization succeeded, then finalize the spawning procedure by
  ! updating state to account for the spawned masters.
  IF (spawn_initiated) THEN
    CALL spawnFinalize()
    ! Wait for all masters to synchronize.
    CALL MPI_BARRIER(xw_comm, ierr)
  END IF
  ! If available memory for this master has fallen below the specified
  ! threshold, then set the local spawn request flag.
  IF (spawn_if_lowmem) THEN
    ! Determine the memory threshold based on total memory and spawn count.
    mem_threshold = totl_mem*(1.0_R8 - 1.0_R8 / 2.0_R8**(1 + spawn_count))
    ! Determine if the ratio of masters to workers will be good after spawning.
    good_ratio = .FALSE.
    IF (2*N_MASTER_I <= (world_size + N_MASTER_I)/3) THEN
      good_ratio = .TRUE.
    END IF
    ! Set flag for a local spawn request if spawning conditions hold.
    IF (REAL(box_count, R8)*box_mem / size_mb > mem_threshold .AND. &
        t - switch_iter > 2 .AND. spawn_count < max_spawn .AND. &
        N_SUB_I == 1 .AND. good_ratio) THEN
      spawn_requested_local = .TRUE.
    END IF
```

```
END IF
! Determine the total box count across masters in a subdomain.
CALL MPI_ALLREDUCE(box_count, box_count_global, 1, MPI_INTEGER, &
                MPI_SUM, xm_comm, ierr)
! If there is a local spawn request, notify all other masters.
CALL MPI_ALLREDUCE(spawn_requested_local, spawn_requested_global, 1, &
                MPI_LOGICAL, MPI_LOR, xm_comm, ierr)

! Reset counters and clear buffers.
update_counter = 0
IF (mygid == 0) update_array(:) = 0
b_worker(:) = 0
IF (t > 1) eval_c_i = 0
IF (RESTART_I == 1) THEN
  ! Open the checkpoint log file(s) for writing at the beginning
  ! of each iteration.
  OPEN(UNIT=CHKSUNIT, FILE=cpfile, FORM='UNFORMATTED', &
       STATUS='UNKNOWN', POSITION='APPEND', IOSTAT=ierr)
  IF (ierr /= 0) THEN; iflag = FILE_ERROR; RETURN; END IF
END IF
IF (RESTART_I == 2) THEN
  IF ( (chk_sm /= N_MASTER_I) .AND. moreList() ) THEN
    ! Read the number of logs for the current iteration from
    ! 'chk_sm' number of checkpoint files.
    DO j = 1, chk_sm
      READ(UNIT=CHKSUNIT + j, IOSTAT=ierr) check_t, chk_len(j)
      IF (ierr < 0) EXIT ! Exit when no more logs to recover.
    END DO
    IF (ierr == 0) THEN
      ! Sum up the total number of logs for this iteration.
      i = SUM(chk_len)
      IF (i > sizeList()) THEN
        ! Enlarge the list to hold all logs for this iteration.
        CALL enlargeList(MAX(i,2*sizeList()), .FALSE.)
      ELSE ! Reset the counter to start loading logs from the files.
        CALL resetList()
      END IF
      ! Read all logs for this iteration to the list.
      DO j = 1, chk_sm
        DO i = 1, chk_len(j)
          READ(UNIT=CHKSUNIT + j, IOSTAT=ierr) X, tmpf
          IF (ierr /= 0) THEN; iflag = FILE_ERROR + 4; RETURN; END IF
          CALL insList(X, tmpf)
        END DO
      END DO
      ! Sort the list in ascending lexicographical order of 'X'.
      CALL sortList()
    ELSE ! No more logs in the checkpoint file. Release the list of logs.
      CALL cleanList()
      ! Close all units of checkpoint files.
      DO i = 1, chk_sm
        CLOSE(CHKSUNIT + i)
      END DO
      DEALLOCATE(chk_len)
    END IF
    ! Open a new checkpoint file to record the new sequence of
    ! function evaluations.
    OPEN(UNIT=CHKRUNIT + chk_sm, FILE=cpfile2, FORM="UNFORMATTED", &
         STATUS="UNKNOWN", POSITION="APPEND", IOSTAT=ierr)
    IF (ierr /= 0) THEN; iflag = FILE_ERROR; RETURN; END IF
  END IF
  IF (chk_sm == N_MASTER_I) THEN ! Recover directly from its own list.
    ! When all function values in the list have been recovered
    ! (t >= check_t + 1), open the checkpoint file for appending new logs.
    IF (t >= check_t + 1) THEN
      IF (t == check_t + 1) CALL cleanList() ! Release the list of logs.
      OPEN(UNIT=CHKSUNIT, FILE=cpfile, FORM="UNFORMATTED", &
           STATUS="UNKNOWN", POSITION="APPEND", IOSTAT=ierr)
      IF (ierr /= 0) THEN; iflag = FILE_ERROR; RETURN; END IF
    END IF
  END IF
END IF

!Step 2: Identify the set of potentially optimal boxes.
!       They are the first boxes of all columns with CONVEX_BIT set
!       in 'flags' in 'setInd'.
IF (t > 1) THEN
  ! Except for the first iteration, the loop starts with convex
  ! hull box selection that marks the CONVEX_BIT in 'setInd'.
  CALL boxSelection(tmpbox, m_portion, iflag)
  IF (iflag /= 0) RETURN
END IF

! Initialize 'i_start' and 'p_start' in order to search such
! columns in 'setInd'. The first boxes on columns with CONVEX_BIT set
! in 'flags' are potentially optimal.
```

```
i_start = 1
p_start => setInd
! If OBJ_CONV is present and not zero, save 'FMIN' in 'fmin_old' to be
! compared with the updated 'FMIN' later.
IF (PRESENT(OBJ_CONV)) THEN
  IF (OBJ_CONV /= 0.0_R8) fmin_old = FMIN
END IF
! Loop processing any boxes in the columns with CONVEX_BIT set in 'flags'
! in 'setInd'.
! Initialize counters and buffers.
gc_convex = 0
lbuffer(:) = 0
setB_c = 0
IF (m_portion == 0) THEN
  ! No convex hull boxes have been assigned to this master, so release
  ! blocked workers (if any).
  DO WHILE (q_counter > 0)
    CALL MPI_SEND(buffer, bufsize, MPI_BYTE, q_worker(q_counter), &
                NO_POINT, xw_comm, ierr)
    q_counter = q_counter - 1
  END DO

  IF (RESTART_I == 1) THEN
    ! Record a log with an empty setB size in the checkpoint file.
    WRITE(UNIT=CHKSUNIT, IOSTAT=ierr) t, 0
  END IF
  IF (RESTART_I == 2) THEN
    IF (chk_sm /= N_MASTER_I) THEN
      ! Record a log with an empty setB size in the new checkpoint file.
      WRITE(UNIT=CHKRUNIT + chk_sm, IOSTAT=ierr) t, 0
    ELSE
      IF (t >= check_t + 1) THEN ! All iterations have been recovered.
      ! Append a log with an empty setB size in the old checkpoint file.
      WRITE(UNIT=CHKSUNIT, IOSTAT=ierr) t, 0
      END IF
    END IF
  END IF
ELSE  ! Start looking for convex hull boxes in the data structures.
  INNER:  DO
    do_it = .FALSE.
    ! Find such a box column in linked list 'setInd' starting
    ! from position 'i_start' in the node 'p_start'. If found,
    ! 'do_it' will be set TRUE and index 'i' and node 'p_iset'
    ! will be returned.
    p_iset => findcol(i_start, p_start, i, do_it)
    IF (do_it) THEN
      ! Step 3:
      ! Step 3a: Obtain the 'setI' of dimensions with the maximum
      !         side length for the first box on column
      !         'p_iset%elements(i)', where 'i' is the index
      !         in 'setInd' for the column holding the hyperbox to
      !         subdivide.
      CALL findsetI(m_head, p_iset%elements(i))
      ! Save information used in sampleP() and divide() in 'lbuffer'.
      lbuffer(gc_convex*set_size + 2) = i ! 'setInd' element ID.
      lbuffer(gc_convex*set_size + 3) = p_iset%id ! 'setInd' link ID.
      lbuffer(gc_convex*set_size + 4) = p_iset%elements(i) ! Box column ID.
      ! Update 'setB_c' and 'gc_convex'.
      setB_c = setB_c + 2*lbuffer(gc_convex*set_size + 1)
      gc_convex = gc_convex + 1
    ELSE
      ! There are no more columns of boxes to divide for this iteration.
      EXIT INNER
    END IF
  END DO INNER
  ! Allocate or reallocate 'setB' with a reasonable size. 'setB' holds
  ! all newly generated boxes from divide() and their subdivided parent
  ! boxes.
  setB_c = setB_c + gc_convex + 1
  IF (setB_len < setB_c) THEN
    ! Reallocate a bigger 'setB'.
    setB_len = MAX(setB_c, 2*setB_len)
    DO i = 1, SIZE(setB%Line)
      DEALLOCATE(setB%Line(i)%c)
      DEALLOCATE(setB%Line(i)%side)
    END DO
    DEALLOCATE(setB%Line)
    DEALLOCATE(setB%dir)
    ALLOCATE(setB%Line(setB_len), STAT = alloc_err)
    IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 1; RETURN; END IF
    ALLOCATE(setB%dir(setB_len), STAT = alloc_err)
    IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 1; RETURN; END IF
    DO i = 1, setB_len
      ALLOCATE(setB%Line(i)%c(N), STAT = alloc_err)
      IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 1; RETURN; END IF
      ALLOCATE(setB%Line(i)%side(N), STAT = alloc_err)
```

```
    IF (alloc_err /= 0) THEN; iflag = ALLOC_ERROR + 1; RETURN; END IF
  END DO
END IF
! Initialize the buffer counter 'setB%ind'.
setB%ind = 0
! Sample new points around all convex boxes and store them in 'setB'.
! Step 3b: Sample new center points at c + delta*e_i and
!         c - delta*e_i for all dimensions in 'lbuffer', where
!         c is the center of the parent box being processed,
!         and e_i is the ith unit vector. Evaluate the objective
!         function at new center points and keep track of current
!         global minimum 'FMIN' and its associated 'unit_x'.
CALL sampleP(m_head, setB)
! For all the newly sampled points in 'setB', evaluate their
! function values or obtain the values via checkpoint logs.
IF ((world_size >= 3) .AND. (N_MASTER_I*N_SUB_I /= world_size)) THEN
  ! When more than 3 processors are available and some of them are
  ! workers, call sampleF to distribute function evaluation tasks to
  ! workers. If checkpointing is on for recovery, sampleF_local is
  ! called instead.
  IF ((RESTART_I == 2) .AND. moreList()) THEN
    ! Not all checkpoint logs are recovered, so obtain the function
    ! values from the local list or table.
    CALL sampleF_local(setB, eval_c_i, iflag)
    IF (iflag /= 0) EXIT MAIN_LOOP
  ELSE
    CALL sampleF(setB, eval_c_i)
  END IF
ELSE ! When less than 3 processors are available, or all of them
  ! are masters (no workers), call sampleF_local to evaluate function
  ! values locally.
  CALL sampleF_local(setB, eval_c_i, iflag)
  IF (iflag /= 0) EXIT MAIN_LOOP
END IF
! Step 3c: Divide the hyperbox containing c into thirds along the
!         dimensions in 'lbuffer', starting with the dimension with
!         the lowest function value of f(c +- delta*e_i) and
!         continuing to the dimension with the highest function
!         value f(c +- delta*e_i).
CALL divide(m_head, setB, setDia, setInd, setFcol, p_box, setW, iflag)
IF (iflag /= 0) THEN
  STATUS(1) = ALLOC_ERROR + iflag - 1
  EXIT MAIN_LOOP
  END IF
END IF
IF (N_MASTER_I > 1) THEN
  ! When more than one masters per subdomain, update intermediate results
  ! from different masters at every iteration.
  IF (mygid == 0) THEN ! The root subdomain master.
    IF (FMIN_I < FMIN) THEN
      FMIN = FMIN_I
      unit_x = unit_x_i
      dia = dia_i
    END IF
    IF (FMIN_I == FMIN) THEN
      IF (dia_i < dia) THEN
        dia = dia_i
        unit_x = unit_x_i
      ELSE
        IF (dia_i == dia) THEN
          IF (unit_x_i .lexLT. unit_x) THEN
            dia = dia_i
            unit_x = unit_x_i
          END IF
        END IF
      END IF
    END IF
    eval_c = eval_c + eval_c_i
    ! Update the counter and buffer for updating.
    update_counter = update_counter + 1
    update_array(1) = 1
    DO WHILE (update_counter < N_MASTER_I)
      ! Loop receiving the updates from all nonroot subdomain masters.
      CALL MPI_RECV(buffer, bufsize, MPI_BYTE, MPI_ANY_SOURCE, MPI_ANY_TAG, &
                    xw_comm, mstatus, ierr)

      ! For a message 'UPDATES', merge the results to the global copy.
      IF (mstatus(MPI_TAG) == UPDATES) THEN
        IF (buffer(1) < FMIN) THEN
          FMIN = buffer(1)
          unit_x = buffer(2:N + 1)
          dia = buffer(N + 3)
        END IF
        IF (buffer(1) == FMIN) THEN
          IF (ALL(buffer(2:N + 1) == unit_x)) THEN
            ! It's the same box with smaller diameter.
```

```
      IF (buffer(N + 3) < dia) THEN
        dia = buffer(N + 3)
        unit_x = buffer(2:N + 1)
      END IF
    ELSE
      IF (buffer(2:N + 1) .lexLT. unit_x) THEN
        dia = buffer(N + 3)
        unit_x = buffer(2:N + 1)
      END IF
    END IF
  END IF
  eval_c = eval_c + INT(buffer(N + 2))
  ! Update the counter and buffer for updating.
  update_counter = update_counter + 1
  ! Find 'mygid' for this subdomain master.
  DO k = 2, N_MASTER_I
    IF (mstatus(MPI_SOURCE) == array_masters(k)) EXIT
  END DO
  update_array(k) = 1
  ! Exit when it has received all updates.
  IF (update_counter == N_MASTER_I) EXIT
END IF
! For results from a root subdomain master, store the results.
IF (mstatus(MPI_TAG) == RESULT_DATA) THEN
  ! Find the subdomain ID.
  k = mstatus(MPI_SOURCE)/N_MASTER_I
  ! Set bit k in 'bits_sub' for the finished subdomain.
  bits_sub = IBSET(bits_sub, k)
  c_bits_sub = c_bits_sub + 1
  array_results(k)%fmin = buffer(1)
  array_results(k)%x = buffer(2:N + 1)
  array_results(k)%max_iter = INT(buffer(N + 2))
  array_results(k)%max_evl = INT(buffer(N + 3))
  array_results(k)%min_dia = buffer(N + 4)
  array_results(k)%status = INT(buffer(N + 5))
  c_alldone = c_alldone + INT(buffer(N + 6))
  ! Send a 'BE_WORKER' message and 'bits_sub' to this master.
  buffer(1) = bits_sub
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, mstatus(MPI_SOURCE), &
              BE_WORKER, xw_comm, ierr)
END IF
! For a delayed nonblocking request from a worker, reply a
! 'NO_POINT' message.
IF (mstatus(MPI_TAG) == NONBLOCK_REQ ) THEN
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, mstatus(MPI_SOURCE), &
              NO_POINT, xw_comm, ierr)
END IF
! Process a delayed blocking request from a worker.
IF (mstatus(MPI_TAG) == BLOCK_REQ) THEN
  IF (N_SUB_I > 1) THEN
    ! For a multiple subdomain search, let the worker try to seek work
    ! from other masters if this is its first blocking request.
    IF (b_worker(mstatus(MPI_SOURCE) + 1) == 0) THEN
      ! Assign 'b_worker' to remember the first blocking request for
      ! this worker.
      b_worker(mstatus(MPI_SOURCE) + 1) = 1
      CALL MPI_SEND(buffer, bufsize, MPI_BYTE, mstatus(MPI_SOURCE), &
                  NO_POINT, xw_comm, ierr)
    ELSE ! This is the second blocking request from this worker, so
      ! block this worker in the queue 'q_worker'.
      IF (.NOT. inQueue(q_worker, q_counter, mstatus(MPI_SOURCE))) THEN
        q_counter = q_counter + 1
        q_worker(q_counter) = mstatus(MPI_SOURCE)
      END IF
    END IF
  ELSE ! For a single subdomain search, block the worker in 'q_worker'.
    IF (.NOT. inQueue(q_worker, q_counter, mstatus(MPI_SOURCE))) THEN
      q_counter = q_counter + 1
      q_worker(q_counter) = mstatus(MPI_SOURCE)
    END IF
  END IF
END IF
! Update 'c_alldone'.
IF (mstatus(MPI_TAG) == COUNT_DONE) THEN
  c_alldone = c_alldone + buffer(1)
END IF
END DO
! Send the updates to all other masters.
buffer(1) = FMIN
buffer(2:N + 1) = unit_x
buffer(N + 2) = eval_c
buffer(N + 3) = dia
i = 1
DO
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, i, UPDATES, &
              xw_comm, ierr)
```

```
    i = i + 1
    IF (i == N_MASTER_I) EXIT
  END DO
ELSE ! Nonroot subdomain masters.
  ! Send updates to the root subdomain master.
  buffer(1) = FMIN_I
  buffer(2:N + 1) = unit_x_i
  buffer(N + 2) = eval_c_i
  buffer(N + 3) = dia_i
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, array_masters(1), UPDATES, &
              xw_comm, ierr)
  ! Wait for the updated results from the root master.
  DO
    CALL MPI_RECV(buffer, bufsize, MPI_BYTE, MPI_ANY_SOURCE, &
                MPI_ANY_TAG, xw_comm, mstatus, ierr)
    IF (mstatus(MPI_TAG) == UPDATES) THEN
      ! Update the global copy of results.
      FMIN = buffer(1)
      unit_x = buffer(2:N + 1)
      eval_c = INT(buffer(N + 2))
      dia = buffer(N + 3)
      EXIT
    END IF
    ! For a delayed requests from a worker, reply a "NO_POINT" message.
    IF (mstatus(MPI_TAG) == NONBLOCK_REQ) THEN
      CALL MPI_SEND(buffer, bufsize, MPI_BYTE, mstatus(MPI_SOURCE), &
                  NO_POINT, xw_comm, ierr)
    END IF
    IF (mstatus(MPI_TAG) == BLOCK_REQ) THEN
      IF (N_SUB_I > 1) THEN
        IF (b_worker(mstatus(MPI_SOURCE) + 1) == 0) THEN
          b_worker(mstatus(MPI_SOURCE) + 1) = 1
          CALL MPI_SEND(buffer, bufsize, MPI_BYTE, mstatus(MPI_SOURCE), &
                      NO_POINT, xw_comm, ierr)
        ELSE
          IF (.NOT. inQueue(q_worker, q_counter, mstatus(MPI_SOURCE))) THEN
            q_counter = q_counter + 1
            q_worker(q_counter) = mstatus(MPI_SOURCE)
          END IF
        END IF
      ELSE
        IF (.NOT. inQueue(q_worker, q_counter, mstatus(MPI_SOURCE))) THEN
          q_counter = q_counter + 1
          q_worker(q_counter) = mstatus(MPI_SOURCE)
        END IF
      END IF
    END IF
  END DO
 END IF ! End of IF (mygid==0)
 ! Update the local copy of results.
 FMIN_I = FMIN
 unit_x_i = unit_x
 dia_i = dia
ELSE ! Only one subdomain master.
 FMIN = FMIN_I
 unit_x = unit_x_i
 dia = dia_i
 eval_c = eval_c + eval_c_i
END IF
! Check stop rules:
! Stop rule 1: maximum iterations.
IF (BTEST(stop_rule, 0)) THEN
 IF (t >= MAX_ITER) THEN; STATUS(1) = 1; EXIT MAIN_LOOP; END IF
END IF
! Stop rule 2: maximum evaluations.
IF (BTEST(stop_rule, 1)) THEN
 IF (eval_c >= MAX_EVL) THEN; STATUS(1) = 2; EXIT MAIN_LOOP; END IF
END IF
! Stop rule 3: minimum diameter.
! Check if minimum diameter has been reached regardless of whether
! MIN_DIA was specified.
IF (sqrt(dia) <= dia_limit) THEN
 STATUS(1) = 3
 EXIT MAIN_LOOP
END IF
! Stop rule 4: objective function convergence.
! If the optional argument OBJ_CONV is present, save 'FMIN' to be
! compared with the updated 'FMIN'.
IF (PRESENT(OBJ_CONV)) THEN
 IF ((OBJ_CONV /= 0.0_R8) .AND. (fmin_old /= FMIN)) THEN
  ! 'FMIN' has been updated.
  IF (fmin_old - FMIN < (1.0_R8 + ABS(fmin_old))*OBJ_CONV) THEN
   STATUS = 4
   EXIT MAIN_LOOP
  END IF
 END IF
```

```
  END IF

  ! Close logical units for writing checkpoint logs at the end of each
  ! iteration.
  IF (RESTART_I == 1) CLOSE(CHKSUNIT)
  IF (RESTART_I == 2) THEN
    IF (chk_sm /= N_MASTER_I) THEN
      ! Close the logical unit for the new checkpoint file.
      CLOSE(CHKRUNIT + chk_sm)
    ELSE
      ! Close the logical unit for appending new logs to the checkpoint file
      ! after all iterations have been recovered.
      IF (t >= check_t + 1) CLOSE(CHKSUNIT)
    END IF
  END IF
  ! Update iteration counter.
  t = t + 1
END DO MAIN_LOOP

IF (iflag /= 0) RETURN
IF (RESTART_I == 2) THEN
  IF ((chk_sm /= N_MASTER_I) .AND. moreList()) THEN
    ! Release the list of logs.
    CALL cleanList()
    ! Close all units of checkpoint files.
    DO i = 1, chk_sm
      CLOSE(CHKSUNIT + i)
    END DO
    DEALLOCATE(chk_len)
  END IF
  IF ( (chk_sm == N_MASTER_I) .AND. (t <= check_t) ) THEN
    CALL cleanList()
    DEALLOCATE(chk_len)
  END IF
END IF

IF (mygid == 0) THEN
  ! The root subdomain master prepares to return the results.
  ! Scale 'unit_x' back to 'X' in original coordinates.
  X = L + unit_x*UmL
  ! Return current diameter of the box with 'FMIN'.
  IF (PRESENT(MIN_DIA)) MIN_DIA = SQRT(dia)
  ! Return the total iterations and evaluations.
  IF (PRESENT(MAX_ITER)) MAX_ITER = t
  IF (PRESENT(MAX_EVL)) MAX_EVL = eval_c
END IF

! Find as many as SIZE(BOX_SET) best boxes.
IF (PRESENT(BOX_SET)) THEN
  ! Put the function value and the center coordinates of the best box into
  ! 'BOX_SET'. Initialize the 'current_center' as the best box.
  current_center = unit_x
  boxset_ind = boxset_ind + 1
  BOX_SET(boxset_ind)%val= FMIN
  ! Scale the center coordinates to the original coordinate system.
  BOX_SET(boxset_ind)%c = L + unit_x*UmL
  ! Loop to find SIZE(BOX_SET) - 1 best boxes.
  OUTER1: DO k = 1, SIZE(BOX_SET) - 1
    ! Set an initial value to be compared with box function values.
    ! Reuse the real variable 'fmin_old' (FMIN backup).
    fmin_old = HUGE(0.0_R8)
    ! Loop over the entire data structure starting from the box matrix
    ! 'm_head'.
    p_b => m_head
    ! Initialize the number of marked boxes (with zero diameter) 't' to be
    ! 0. Reuse the integer variable 't' (loop counter for main loop).
    t = 0
    INNER1: DO WHILE(ASSOCIATED(p_b))
      ! Check all the columns in 'p_b'.
      INNER2: DO i = 1, col_w
        DO j = 1, MOD(p_b%ind(i) - 1, row_w) + 1
          ! Only in the first pass, locate the box with 'FMIN' and mark it.
          IF (k == 1) THEN
            IF (ALL(unit_x == p_b%M(j,i)%c)) THEN
              ! Fill in the 'side' and 'diam' of the first best box in
              ! BOX_SET. Scale them back to the original coordinate
              ! system. Mark this box by setting its diameter zero and update
              ! 't'.
              BOX_SET(1)%side = p_b%M(j, i)%side*UmL
              BOX_SET(1)%diam = SUM(BOX_SET(1)%side**2)
              p_b%M(j,i)%diam = 0
              t = t + 1
              CYCLE
            END IF
          END IF
          ! Process unmarked boxes (with non-zero diameter) and update 't'.
```

```
  IF (p_b%M(j,i)%diam /= 0) THEN
    ! Compute the weighted separation between 'current_center' and the
    ! center of this box 'p_b%M(j,i)'.
    tmp_x= (current_center - p_b%M(j,i)%c)*UmL
    IF (DOT_PRODUCT(tmp_x*W_I, tmp_x) < MIN_SEP_I) THEN
      ! If the separation is less than 'MIN_SEP_I', mark this box by
      ! setting its diameter zero.
      p_b%M(j,i)%diam = 0
      t = t + 1
    ELSE
      ! If the separation >= MIN_SEP_I, compare the function value
      ! of the box with the current best value saved in 'fmin_old'.
      IF (p_b%M(j,i)%val < fmin_old) THEN
        fmin_old = p_b%M(j,i)%val
        ! Backup the pointer to the current best box.
        p_save => p_b%M(j,i)
      END IF
    END IF
  ELSE
    t = t + 1
  END IF
END DO
! Repeat the above steps if any box link exists.
IF (p_b%ind(i) > row_w) THEN
! There must be box link(s).
  p_l => p_b%sibling(i)%p
  DO WHILE(ASSOCIATED(p_l))
    DO j = 1, p_l%ind
      IF (p_l%Line(j)%diam /= 0) THEN
        tmp_x = (current_center - p_l%Line(j)%c)*UmL
        IF (DOT_PRODUCT(tmp_x*W_I, tmp_x)< MIN_SEP_I) THEN
          p_l%Line(j)%diam = 0
          t = t + 1
        ELSE
          IF (p_l%Line(j)%val < fmin_old) THEN
            fmin_old = p_l%Line(j)%val
            p_save => p_l%Line(j)
          END IF
        END IF
      ELSE
        t = t + 1
      END IF
    END DO
    ! Move to the next box link, if any.
    p_l => p_l%next
  END DO
 END IF
 END DO INNER2
 ! Move to the next box matrix, if any.
 p_b => p_b%child
END DO INNER1
IF (ASSOCIATED(p_save)) THEN
  IF (p_save%diam /= 0) THEN
    ! Found the next best box. Put it into BOX_SET and mark it.
    boxset_ind = boxset_ind + 1
    BOX_SET(boxset_ind) = p_save
    ! Scale the coordinates back to the original coordinate system.
    BOX_SET(boxset_ind)%c = L + &
                    BOX_SET(boxset_ind)%c*UmL
    BOX_SET(boxset_ind)%side = BOX_SET(boxset_ind)%side*UmL
    BOX_SET(boxset_ind)%diam = SUM(BOX_SET(boxset_ind)%side**2)
    p_save%diam = 0
    t = t + 1
    ! Update 'current_center'.
    current_center = p_save%c
  END IF
ELSE ! Exit when the next best box satisfying MIN_SEP is not available.
  EXIT
END IF
 ! If all boxes in the data structure are marked, exit.
 IF (eval_c <= t) EXIT
END DO OUTER1
! If the NUM_BOX or MIN_SEP is specified as an optional input
! argument, assign the value to it.
IF (PRESENT(NUM_BOX)) NUM_BOX = boxset_ind
IF (PRESENT(MIN_SEP)) MIN_SEP = SQRT(MIN_SEP_I)
END IF

! Deallocate all the data structures explicitly allocated, including
! box matrices, box links, setB, setW, setInd, setFcol, setDia,
! and p_box.
CALL cleanup()

IF (N_MASTER_I > 1) THEN
  DEALLOCATE(gbuffer)
  DEALLOCATE(displs)
```

```
END IF
DEALLOCATE(lbuffer)
DEALLOCATE(tmpbox)


RETURN
END SUBROUTINE master

FUNCTION onRight (box1v, box1d, box2v, box2d, box3v, box3d) RESULT (ans)
IMPLICIT NONE
! Return .TRUE. if line box1-box3 is on the right side of box1-box2.
!
! On input:
! box1v  - The function value of box1.
! box1d  - The diameter of box1.
! box2v  - The function value of box2.
! box2d  - The diameter of box2.
! box3v  - The function value of box3.
! box3d  - The diameter of box3.
!
! On output:
! ans    - The result of checking "on-right" condition.
!
REAL(KIND = R8), INTENT(IN) :: box1v, box1d, box2v, box2d, box3v, box3d
LOGICAL :: ans

ans = .FALSE.
IF ((box1v - box3v)*(box1d - box2d) - &
    (box1d - box3d)*(box1v - box2v) > 0) ans = .TRUE.

RETURN
END FUNCTION onRight

SUBROUTINE rmMat(b_id, mycol, eid, p_iset)
IMPLICIT NONE
! Remove the first box in the box column 'mycol' in 'b_id' box matrix
! and adjust setDia, setInd, and setFcol accordingly.
!
! On input:
! b_id   - Box matrix ID of the box column.
! mycol  - Box column ID.
! eid    - Element ID in a node of linked list 'setInd'.
! p_iset - Pointer to a node in linked list 'setInd'.
!
! On output:
! p_iset - Pointer to the changed setFcol.
!
INTEGER, INTENT(IN) :: b_id
INTEGER, INTENT(IN) :: mycol
INTEGER, INTENT(IN) :: eid
TYPE(int_vector), POINTER :: p_iset

! Local variables.
INTEGER :: i

p_b => m_head
DO i = 1, b_id -1
  p_b => p_b%child
END DO
! Move the last box to the first position in heap.
IF (p_b%ind(mycol) <= row_w) THEN
  ! There are no box links.
  IF (p_b%ind(mycol) > 1) p_b%M(1,mycol) = p_b%M(p_b%ind(mycol),mycol)
ELSE
  ! There are box links. Chase to the last box link.
  p_l => p_b%sibling(mycol)%p
  DO i = 1, (p_b%ind(mycol) - 1)/row_w - 1
    p_l => p_l%next
  END DO
  p_b%M(1, mycol) = p_l%Line(p_l%ind)
  p_l%ind = p_l%ind - 1
END IF
p_b%ind(mycol) = p_b%ind(mycol) - 1
! Update 'setDia', 'setInd' and 'setFcol' if this column is empty.
! Find which node 'setInd' is associated with, by checking
! 'setInd%id'. Find the link in setInd.
IF (p_b%ind(mycol) == 0 )THEN
  ! This column is empty. Remove this diameter squared from a
  ! corresponding node of 'setDia'.
  CALL rmNode(col_w, p_iset%id - 1, eid, setDia)
  ! Push the released column back to top of 'setFcol'.
  IF (setFcol%dim < col_w) THEN
    ! The head node of 'setFcol' is not full.
    CALL insNode(col_w, p_iset%elements(eid), setFcol%dim + 1, setFcol)
  ELSE
    ! The head node is full. There must be at least one more node
```

```
    ! for 'setFcol'. Find the last non-full node of 'setFcol' to
    ! insert the released column.
    p_iset => setFcol%next
    DO
      IF (p_iset%dim < col_w) THEN
        ! Found it.
        CALL insNode(col_w, p_iset%elements(eid), p_iset%dim + 1, p_iset)
        EXIT
      END IF
      ! Go to the next node.
      p_iset => p_iset%next
    END DO
  END IF
  ! Remove the column index from a corresponding node of 'setInd'.
  CALL rmNode(col_w, 0, eid, p_iset)
ELSE
  ! Call siftdown to reorder the heap.
  CALL siftdown(p_b, mycol, 1)
END IF

RETURN
END SUBROUTINE rmMat

SUBROUTINE sampleF(setB, eval_c_i)
IMPLICIT NONE
! Evaluate the objective function at each newly sampled center point.
! Keep updating 'FMIN' and 'unit_x'. Function evaluations are distributed
! to workers.
!
! On input:
! setB    - The set of newly sampled boxes with their center points'
!           coordinates.
! eval_c_i - The local counter of evaluations on this master.
!
! On output:
! setB    - The set of newly sampled boxes with added function values
!           at center points.
! eval_c_i - The updated local counter of evaluations on this master.
!
TYPE(BoxLine), INTENT(INOUT) :: setB
INTEGER, INTENT(INOUT) :: eval_c_i

! Local variables.
INTEGER :: c_completed ! Counter for completed evaluations.
INTEGER :: i, j, k ! Counters.

! Initialization.
c_completed = 0
i = 0 ! Counter for sending boxes.
! Record the total number 'setB%ind' of logs for this iteration to the
! checkpoint file or the new checkpoint file.
IF (RESTART_I == 1) THEN
   WRITE(UNIT=CHKSUNIT) t, setB%ind
ELSE IF (RESTART_I == 2) THEN
  IF ( (chk_sm == N_MASTER_I) .AND. (t >= check_t + 1) ) THEN
    WRITE(UNIT=CHKSUNIT) t, setB%ind
  ELSE IF (chk_sm /= N_MASTER_I) THEN
    WRITE(UNIT=CHKRUNIT + chk_sm) t, setB%ind
  END IF
END IF
LOOP1: DO WHILE (c_completed < setB%ind)
  ! There is at least one point to evaluate, so send to workers.
  LOOP2: DO WHILE ((q_counter > 0) .AND. (i < setB%ind))
    ! There is at least one worker in 'q_worker', so send tasks
    ! and update q_counter.
    IF (i + BINSIZE_I <= setB%ind) THEN
      ! The first slot holds the point starting index.
      buffer(1) = i + 1
      ! The second slot holds the number of points for this task.
      buffer(2) = BINSIZE_I
      ! The remaining slots hold the set(s) of point coordinates.
      DO j = 1, BINSIZE_I
        buffer(3 + (j - 1)*N:(j - 1)*N + N + 2) = L + &
                                    setB%Line(i + j)%c*UmL
      END DO
      ! Update 'i'.
      i = i + BINSIZE_I
    ELSE
      buffer(1) = i + 1
      buffer(2) = setB%ind - i
      DO j = 1, INT(buffer(2))
        buffer(3 + (j - 1)*N:(j - 1)*N + N + 2) = L + &
                                    setB%Line(i + j)%c*UmL
      END DO
      ! Update 'i'.
      i = i + INT(buffer(2))
```

```
  END IF
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, q_worker(q_counter), &
              POINT_BREQ, xw_comm, ierr)
  q_counter = q_counter - 1
END DO LOOP2
IF (N_SUB_I > 1) THEN
  ! For a multiple subdomain search, release the remaining blocked workers
  ! after sending all tasks to blocked workers.
  DO WHILE (q_counter > 0)
    CALL MPI_SEND(buffer, bufsize, MPI_BYTE, q_worker(q_counter), &
                NO_POINT, xw_comm, ierr)
    q_counter = q_counter - 1
  END DO
END IF
! Wait for messages from workers.
CALL MPI_RECV(buffer, bufsize, MPI_BYTE, MPI_ANY_SOURCE, &
            MPI_ANY_TAG, xw_comm, mstatus, ierr)
recv_tag = mstatus(MPI_TAG)

SELECT CASE (recv_tag)
  CASE (NONBLOCK_REQ) ! A nonblocking request from a worker.
    ! If there is at least one point, send a task to this worker. Otherwise,
    ! send 'NO_POINT'.
    IF (i < setB%ind) THEN
      IF (i + BINSIZE_I <= setB%ind) THEN
        buffer(1) = i + 1
        buffer(2) = BINSIZE_I
        DO j =1, BINSIZE_I
          buffer(3 + (j - 1)*N:(j - 1)*N + N + 2) = L + &
                                    setB%Line(i + j)%c*UmL
        END DO
        i = i + BINSIZE_I
      ELSE
        buffer(1) = i + 1
        buffer(2) = setB%ind - i
        DO j = 1, INT(buffer(2))
          buffer(3 + (j - 1)*N:(j - 1)*N + N + 2) = L + &
                                    setB%Line(i + j)%c*UmL
        END DO
        i = i + INT(buffer(2))
      END IF
      CALL MPI_SEND(buffer, bufsize, MPI_BYTE, mstatus(MPI_SOURCE), &
                  POINT_NBREQ, xw_comm, ierr)
    ELSE
      CALL MPI_SEND(buffer, bufsize, MPI_BYTE, mstatus(MPI_SOURCE), &
                  NO_POINT, xw_comm, ierr)
    END IF
  CASE (FUNCVAL) ! Returned function value(s) from a worker.
    c_completed = c_completed + INT(buffer(2))
    ! Put the value(s) back to 'setB'. 'buffer(2)' holds the number of
    ! function values in this returned buffer.
    DO j = 1, INT(buffer(2))
      ! 'buffer(1)' holds the point starting index. Function values
      ! are stored starting from 'buffer(3)'.
      setB%Line(INT(buffer(1)) + j - 1)%val = buffer(3 + 2*(j - 1))
      IF (INT(buffer(4 + 2*(j - 1))) /= 0) THEN
        ! 'iflag' returned for the function evaluation is not 0, so
        ! it was evaluated in an undefined point. Assign a huge value.
        setB%Line(INT(buffer(1)) + j - 1)%val = HUGE(1.0_R8)
      END IF
      ! Update evaluation counter.
      eval_c_i = eval_c_i + 1
    END DO
    ! Send another task if at least one point is available.
    IF (i < setB%ind) THEN
      IF (i + BINSIZE_I <= setB%ind) THEN
        buffer(1) = i + 1
        buffer(2) = BINSIZE_I
        DO j =1, BINSIZE_I
          buffer(3 + (j - 1)*N:(j - 1)*N + N + 2) = L + &
                                    setB%Line(i + j)%c*UmL
        END DO
        i = i + BINSIZE_I
      ELSE
        buffer(1) = i + 1
        buffer(2) = setB%ind - i
        DO j = 1, INT(buffer(2))
          buffer(3 + (j - 1)*N:(j - 1)*N + N + 2) = L + &
                                    setB%Line(i + j)%c*UmL
        END DO
        i = i + INT(buffer(2))
      END IF
      CALL MPI_SEND(buffer, bufsize, MPI_BYTE, mstatus(MPI_SOURCE), &
                  POINT_NBREQ, xw_comm, ierr)
    ELSE ! Send 'NO_POINT' to this worker.
      CALL MPI_SEND(buffer, bufsize, MPI_BYTE, mstatus(MPI_SOURCE), &
```

```
              NO_POINT, xw_comm, ierr)
  END IF
CASE (BLOCK_REQ) ! A blocking request from a worker.
  ! If it has at least one point to evaluate, send a task.
  IF (i < setB%ind) THEN
    IF (i + BINSIZE_I <= setB%ind) THEN
      buffer(1) = i + 1
      buffer(2) = BINSIZE_I
      DO j = 1, BINSIZE_I
        buffer(3 + (j - 1)*N:(j - 1)*N + N + 2) = L + &
                                  setB%Line(i + j)%c*UmL
      END DO
      i = i + BINSIZE_I
    ELSE
      buffer(1) = i + 1
      buffer(2) = setB%ind - i
      DO j = 1, INT(buffer(2))
        buffer(3 + (j - 1)*N:(j - 1)*N + N + 2) = L + &
                                  setB%Line(i + j)%c*UmL
      END DO
      i = i + INT(buffer(2))
    END IF
    CALL MPI_SEND(buffer, bufsize, MPI_BYTE, mstatus(MPI_SOURCE), &
                POINT_BREQ, xw_comm, ierr)
  ELSE ! No point to evaluate.
    IF (N_SUB_I > 1) THEN
      ! For a multiple subdomain search, if 'b_worker' for this worker
      ! is not set 1 (the worker has not sent any blocking request to
      ! this master in this iteration), set the bit to 1 and send
      ! 'NO_POINT'; if 'b_worker' for this worker is set 1 block it
      ! in 'q_worker'.
      IF (b_worker(mstatus(MPI_SOURCE) + 1) == 0) THEN
        b_worker(mstatus(MPI_SOURCE) + 1) = 1
        CALL MPI_SEND(buffer, bufsize, MPI_BYTE, mstatus(MPI_SOURCE), &
                    NO_POINT, xw_comm, ierr)
      ELSE
        IF (.NOT. inQueue(q_worker, q_counter, mstatus(MPI_SOURCE))) THEN
          q_counter = q_counter + 1
          q_worker(q_counter) = mstatus(MPI_SOURCE)
        END IF
      END IF
    ELSE ! For a single domain search, always queue up the worker
      ! that sent a blocking request.
      IF (.NOT. inQueue(q_worker, q_counter, mstatus(MPI_SOURCE))) THEN
        q_counter = q_counter + 1
        q_worker(q_counter) = mstatus(MPI_SOURCE)
      END IF
    END IF
  END IF
CASE (RESULT_DATA) ! For a root master only: results from a finished
  ! subdomain root master. Save the results.
  k = mstatus(MPI_SOURCE)/N_MASTER_I
  bits_sub = IBSET(bits_sub, k)
  c_bits_sub = c_bits_sub + 1
  array_results(k)%fmin = buffer(1)
  array_results(k)%x = buffer(2:N + 1)
  array_results(k)%max_iter = INT(buffer(N + 2))
  array_results(k)%max_evl = INT(buffer(N + 3))
  array_results(k)%min_dia = buffer(N + 4)
  array_results(k)%status = INT(buffer(N + 5))
  c_alldone = c_alldone + INT(buffer(N + 6))
  ! Send 'BE_WORKER' message and 'bits_sub' to this master.
  buffer(1) = bits_sub
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, mstatus(MPI_SOURCE), &
              BE_WORKER, xw_comm, ierr)
CASE (UPDATES) ! The root subdomain master received updates from a
  ! nonroot subdomain master. Merge the updated results.
  IF (buffer(1) < FMIN) THEN
    FMIN = buffer(1)
    unit_x = buffer(2:N + 1)
    dia = buffer(N + 3)
  END IF
  IF (buffer(1) == FMIN) THEN
    IF (ALL(buffer(2:N + 1) == unit_x)) THEN
      ! It's the same box with smaller diameter.
      IF (buffer(N + 3) < dia) THEN
        dia = buffer(N + 3)
        unit_x = buffer(2:N + 1)
      END IF
    ELSE
      IF (buffer(2:N + 1) .lexLT. unit_x) THEN
        dia = buffer(N + 3)
        unit_x = buffer(2:N + 1)
      END IF
    END IF
  END IF
```

```
      eval_c = eval_c + INT(buffer(N + 2))
      update_counter = update_counter + 1
      ! Find 'mygid' for the sender and set the update status.
      DO k = 1, N_MASTER_I
        IF (mstatus(MPI_SOURCE) == array_masters(k)) EXIT
      END DO
      update_array(k) = 1
    CASE (COUNT_DONE) ! The root master updates 'c_alldone'.
      c_alldone = c_alldone + INT(buffer(1))
  END SELECT
END DO LOOP1
! Update 'FMIN_I' and 'unit_x_i'. If needed, record all evaluations in 'setB'
! to the checkpoint file.
DO i = 1, setB%ind
  IF (FMIN_I > setB%Line(i)%val) THEN
    FMIN_I = setB%Line(i)%val
    unit_x_i(:) = setB%Line(i)%c
  END IF
  IF (FMIN_I == setB%Line(i)%val) THEN
    IF (setB%Line(i)%c .lexLT. unit_x_i) THEN
      ! The new point has a smaller lexicographical order than 'unit_x'.
      unit_x_i(:) = setB%Line(i)%c
    END IF
  END IF
  ! Log the point (in the scaled frame) and its function value.
  IF (RESTART_I == 1) THEN
    WRITE(UNIT=CHKSUNIT) setB%Line(i)%c, setB%Line(i)%val
  ELSE IF (RESTART_I == 2) THEN
    IF (chk_sm == N_MASTER_I) THEN
      WRITE(UNIT=CHKSUNIT) setB%Line(i)%c, setB%Line(i)%val
    ELSE
      WRITE(UNIT=CHKRUNIT + chk_sm) setB%Line(i)%c, setB%Line(i)%val
    END IF
  END IF
END DO

RETURN
END SUBROUTINE sampleF

SUBROUTINE sampleF_local(setB, eval_c_i, ierr)
IMPLICIT NONE
! Evaluate the objective function locally at each newly sampled center point.
! Keeps updating 'FMIN' and 'unit_x'.
!
! On input:
! setB   - The set of newly sampled boxes with their center points'
!          coordinates.
! eval_c - The counter of evaluations.
!
! On output:
! setB   - The set of newly sampled boxes with added function values
!          at center points.
! eval_c - The updated counter of evaluations.
! ierr   - Status to return.
!          =0    Normal return.
!          >0    Error return.
!
TYPE(BoxLine), INTENT(INOUT) :: setB
INTEGER, INTENT(INOUT) :: eval_c_i
INTEGER, INTENT(OUT) :: ierr

! Local variables.
INTEGER :: i

! Loop evaluating all the new center points of boxes in 'setB'.
ierr = 0
! Record 't' and 'setB' size to the checkpoint file either when RESTART_I==1
! or RESTART_I==2 when all logs have been recovered.
IF (RESTART_I == 1) THEN
  WRITE(UNIT=CHKSUNIT) t, setB%ind
ELSE IF (RESTART_I == 2) THEN
  IF ( (chk_sm == N_MASTER_I) .AND. (t >= check_t + 1) ) THEN
    WRITE(UNIT=CHKSUNIT) t, setB%ind
  ELSE IF (chk_sm /= N_MASTER_I) THEN
    ! Record 't' and 'setB' size to the new checkpoint file.
    WRITE(UNIT=CHKRUNIT + chk_sm) t, setB%ind
  END IF
END IF
DO i = 1, setB%ind
  ! Evaluate the function in the original coordinate system.
  IF (RESTART_I == 2) THEN
    IF ( ((chk_sm /= N_MASTER_I) .AND. moreList()) .OR. &
      ((chk_sm == N_MASTER_I) .AND. (t < check_t + 1)) ) THEN
      ! Recover from logs.
      IF ( (chk_sm /= N_MASTER_I) .AND. moreList() ) THEN
        ! Recover from the merged list sorted in lexicographical order.
```

```
      IF (.NOT. lookupTab(setB%Line(i)%c, setB%Line(i)%val)) THEN
        ! The function value cannot be found in the checkpoint log file.
        ierr = FILE_ERROR + 4
        EXIT
      END IF
      ! Record it to the new checkpoint file.
      WRITE(UNIT=CHKRUNIT + chk_sm) setB%Line(i)%c, setB%Line(i)%val
    END IF
    IF ( (chk_sm == N_MASTER_I) .AND. (t < check_t + 1) ) THEN
      ! Recover from the list.
      IF (.NOT. inList(setB%Line(i)%c, setB%Line(i)%val)) THEN
        ! The function value cannot be found in the checkpoint log file.
        ierr = FILE_ERROR + 4
        EXIT
      END IF
    END IF
  END IF
ELSE ! A normal evaluation.
  setB%Line(i)%val = OBJ_FUNC(L + setB%Line(i)%c*UmL, iflag)

  ! Check 'iflag'.
  IF (iflag /= 0) THEN
    ! It is evaluated at an undefined point, so assign it a huge value.
    setB%Line(i)%val = HUGE(1.0_R8)
  END IF
  ! Record it to the checkpoint file or the new checkpoint file.
  IF (RESTART_I == 1) THEN
    WRITE(UNIT=CHKSUNIT) setB%Line(i)%c, setB%Line(i)%val
  ELSE IF (RESTART_I == 2) THEN
    IF (chk_sm == N_MASTER_I) THEN
      WRITE(UNIT=CHKSUNIT) setB%Line(i)%c, setB%Line(i)%val
    ELSE
      WRITE(UNIT=CHKRUNIT + chk_sm) setB%Line(i)%c, setB%Line(i)%val
    END IF
  END IF
END IF
! Update evaluation counter.
eval_c_i = eval_c_i + 1

IF (FMIN_I > setB%Line(i)%val) THEN
  ! Update 'FMIN_I' and 'unit_x_i'.
  FMIN_I = setB%Line(i)%val
  unit_x_i(:) = setB%Line(i)%c
  END IF
  ! Update 'FMIN_I' and 'unit_x_i' in Lexicographical order.
  IF (FMIN_I == setB%Line(i)%val) THEN
    IF (setB%Line(i)%c .lexLT. unit_x_i) THEN
      ! The new point has a smaller lexicographical order than 'unit_x_i'.
      unit_x_i(:) = setB%Line(i)%c
    END IF
  END IF
END DO

RETURN
END SUBROUTINE sampleF_local

SUBROUTINE sampleP(b, setB)
IMPLICIT NONE
! On each dimension in 'lbuffer', samples two center points at the c+delta*e_i
! and c-delta*e_i, where e_i is the ith unit vector. In 'setB',
! records all the new points as the centers of boxes which will be formed
! completely through subroutines sampleF and divide.
!
! On input:
! b    - The head link of box matrices.
! setB - The empty box set (type 'HyperBox') which will hold newly sampled
!        points as the centers of new boxes.
!
! On output:
! b    - The head link of box matrices.
! setB - The box set holding the newly sampled center points.
!
TYPE(BoxMatrix), INTENT(INOUT), TARGET :: b
TYPE(BoxLine), INTENT(INOUT) :: setB

! Local variables.
INTEGER :: bc ! Box counter.
INTEGER :: b_id ! Box matrix ID.
INTEGER :: col ! Box column index
INTEGER :: i ! Loop counter.
INTEGER :: j ! Local column index converted from the global one 'col'.
INTEGER :: new_i ! Index of new points in setB.
REAL (KIND = R8) :: delta ! 1/3 of the maximum side length.
TYPE (BoxMatrix), POINTER :: p_b ! Pointer to the associated box matrix.

DO bc = 0, gc_convex - 1
```

```
col = lbuffer(bc*set_size + 4)
! Find the box matrix that 'col' is associated with. Store the pointer
! to box matrix in 'p_b'. The local column index 'j' will be converted from
! 'col'.
IF (col <= col_w) THEN
  p_b => b
  j = col
ELSE
  b_id = (col - 1)/col_w + 1
  j = MOD(col - 1, col_w) + 1
  p_b => b
  DO i = 1, b_id - 1
    p_b => p_b%child
  END DO
END IF

! Find the maximum side length by obtaining the dimension in 'setI'.
! Then, extract the maximum side length from the first box on column 'j' of
! box matrix 'p_b'. Calculate 'delta', 1/3 of the maximum side length.
delta = p_b%M(1, j)%side(INT(lbuffer(bc*set_size + 5)))/3

! Loop sampling two new points of all dimensions in 'setI'.
! c + delta*e_i => newpt_1; c - delta*e_i => newpt_2, where e_i is
! the ith unit vector.
DO i = 1, INT(lbuffer(bc*set_size + 1))
  new_i = setB%ind + 1
  ! Copy the coordinates of parent box to the two new boxes
  setB%Line(new_i)%c(:) = p_b%M(1, j)%c(:)
  setB%Line(new_i + 1)%c(:) = p_b%M(1, j)%c(:)
  ! Assign changed coordinates to the two new points in 'setB'.
  setB%Line(new_i)%c(INT(lbuffer(bc*set_size + 4 + i))) = &
     p_b%M(1, j)%c(INT(lbuffer(bc*set_size + 4 + i))) + delta
  setB%Line(new_i + 1)%c(INT(lbuffer(bc*set_size + 4 + i))) = &
     p_b%M(1, j)%c(INT(lbuffer(bc*set_size + 4 + i))) - delta

 ! Record the directions with changes in 'setB%dir' for further
  ! processing to find the dividing order of dimensions.
  setB%dir(new_i) = INT(lbuffer(bc*set_size + 4 + i))
  setB%dir(new_i + 1) = INT(lbuffer(bc*set_size + 4 + i))

  ! Update 'ind' of 'setB'.
  setB%ind = setB%ind + 2
    ! Initialize side lengths of new points for further dividing
    ! by copying the sides from the parent box.
    setB%Line(new_i)%side(:) = p_b%M(1, j)%side(:)
    setB%Line(new_i + 1)%side(:) = p_b%M(1, j)%side(:)
  END DO
END DO

RETURN
END SUBROUTINE sampleP

FUNCTION sanitycheck() RESULT(iflag)
IMPLICIT NONE
! Check the sanity of the input parameters, and set all local variables
! derived from input arguments.
!
! On input: None.
! On output:
! iflag  - The sanity check result.
!
INTEGER :: iflag

! Initialize 'iflag'.
iflag = 0
!
! Check the required arguments.
!
IF (N < 2) THEN
  iflag = INPUT_ERROR
  RETURN
ELSE
  N_I = N
END IF

IF ((SIZE(X) /= N) .OR. (SIZE(L) /= N) .OR. (SIZE(U) /= N)) THEN
  iflag = INPUT_ERROR + 1
  RETURN
END IF
IF (ANY(L >= U)) THEN
  iflag = INPUT_ERROR + 2
  RETURN
END IF
!
! Check the optional arguments.
```

```
!
IF (PRESENT(W)) THEN
  IF (SIZE(W) /= N) THEN
    iflag = INPUT_ERROR + 1
    RETURN
  END IF
END IF
! Default: processing boxes only on convex hull.
SWITCH_I = 1
IF (PRESENT(SWITCH)) THEN
  IF ((SWITCH < 0) .OR. (SWITCH > 1)) THEN
    iflag = INPUT_ERROR + 5
    RETURN
  END IF
  IF (SWITCH == 0) THEN
    IF (PRESENT(EPS)) THEN
      IF (EPS > 0.0_R8) THEN
        iflag = INPUT_ERROR + 6
        RETURN
      END IF
    END IF
  END IF
  ! Assign the local copy 'SWITCH_I'.
  SWITCH_I = SWITCH
END IF
old_switch = SWITCH_I
! Enable LBC (limiting box column) by default.
lbc = 1
stop_rule = 0
! When MAX_ITER <=0, the number of iterations will be returned on exit.
IF (PRESENT(MAX_ITER)) THEN
  IF (MAX_ITER > 0) THEN
    ! Set bit 0 of stop_rule.
    stop_rule = IBSET(stop_rule, STOP_RULE1)
  ELSE ! Disable LBC (limiting box columns).
     lbc = 0
  END IF
ELSE
  lbc = 0  ! Disable LBC (limiting box columns).
END IF
! When MAX_EVL <=0, the number of evaluations will be returned on exit.
IF (PRESENT(MAX_EVL)) THEN
  IF (MAX_EVL > 0) THEN
    ! Set bit 1 of stop_rule.
    stop_rule = IBSET(stop_rule, STOP_RULE2)
    ! When 'MAX_ITER' is positive and 'MAX_EVL' is
    ! sufficiently small, disable LBC (limiting box columns).
    IF (PRESENT(MAX_ITER)) THEN
      IF ((MAX_ITER > 0) .AND. (MAX_EVL*(2*N + 2) < 2D+6)) lbc = 0
    END IF
  END IF
END IF

! Even if user doesn't specify 'MIN_DIA', a diameter smaller than
! SQRT(SUM(UmL*UmL))*EPSILON(1.0_R8)*N is not permitted to occur.
! When MIN_DIA <=0, the diameter associated with X and FMIN will be
! returned on exit. Assign 'UmL'.
UmL(:) = U(:) - L(:)
dia_limit =  SQRT(SUM(UmL*UmL))*EPSILON(1.0_R8)
IF (PRESENT(MIN_DIA)) THEN
  IF (MIN_DIA > 0.0_R8) THEN
    IF (MIN_DIA < dia_limit) THEN
      iflag = INPUT_ERROR + 3
      RETURN
    ELSE
      dia_limit = MIN_DIA
      ! Set bit 2 of stop_rule.
      stop_rule = IBSET(stop_rule, STOP_RULE3)
    END IF
  END IF
END IF

! When OBJ_CONV is present a minimum relative change in the minimum
! objective function value will be enforced.
IF (PRESENT(OBJ_CONV)) THEN
  IF (OBJ_CONV /= 0.0_R8) THEN
    IF ((OBJ_CONV < EPSILON(1.0_R8)*REAL(N, KIND = R8)) .OR. &
        (OBJ_CONV >= 1.0_R8)) THEN
      iflag = INPUT_ERROR + 3
      RETURN
    ELSE
      ! Set bit 3 of stop_rule.
      stop_rule = IBSET(stop_rule, STOP_RULE4)
    END IF
  END IF
END IF
```

```
! When EPS is present a test involving EPS is used to define potentially
! optimal boxes.  The absence of this test is equivalent to EPS=0.
! Initialize the local copy 'EPS_I'.
EPS_I = 0.0_R8
IF (PRESENT(EPS)) THEN
  IF (EPS /= 0.0_R8) THEN
    IF ((EPS < EPSILON(1.0_R8)) .OR. (EPS > 1.0_R8)) THEN
      iflag = INPUT_ERROR + 3
      RETURN
    ELSE
      EPS_I = EPS
    END IF
  END IF
END IF

! Check if stop_rule has at least at 1 bit set. Otherwise no stopping rule
! has been given.
IF (stop_rule == 0) THEN
  iflag = INPUT_ERROR + 4
  RETURN
END IF

! Set default weights for distance definition or dimension scaling.
W_I(1:N) = 1.0_R8
! Verify W.
IF (PRESENT(W)) THEN
  WHERE (W > 0)
    W_I = W
  ELSEWHERE
    W = 1.0_R8
  END WHERE
END IF

! Check if MIN_SEP and BOX_SET are correctly set.
IF (PRESENT(BOX_SET)) THEN
  ! Compute the weighted diameter of the original design space. Reuse
  ! the variable 'dia' (the diameter squared associated with 'FMIN').
  dia = SQRT(SUM(UmL*W_I*UmL))
  ! Check the optional argument MIN_SEP. Set a default value if it is not
  ! present or correctly assigned.
  MIN_SEP_I = (0.5_R8*dia)**2
  IF (PRESENT(MIN_SEP)) THEN
    IF ((MIN_SEP < dia*EPSILON(1.0_R8)) .OR. (MIN_SEP > dia)) THEN
      MIN_SEP = 0.5_R8*dia
    END IF
    MIN_SEP_I = MIN_SEP**2
  END IF
  ! Initialize BOX_SET by allocating its component arrays c(:) and
  ! side(:).
  DO i = 1, SIZE(BOX_SET)
    IF (ASSOCIATED(BOX_SET(i)%c)) THEN
      IF (SIZE(BOX_SET(i)%c) /= N) THEN
        iflag = ALLOC_ERROR + 4
        RETURN
      END IF
    ELSE ! Allocate component 'c'.
      ALLOCATE(BOX_SET(i)%c(N))
    END IF
    IF (ASSOCIATED(BOX_SET(i)%side)) THEN
      IF (SIZE(BOX_SET(i)%side) /= N) THEN
        iflag = ALLOC_ERROR + 4
        RETURN
      END IF
    ELSE ! Allocate component 'side'.
      ALLOCATE(BOX_SET(i)%side(N))
    END IF
  END DO
  ! Initialize the index counter 'boxset_ind'.
  boxset_ind = 0
  ! Disable LBC (limiting box columns) that removes boxes needed for
  ! finding 'BOX_SET'.
  lbc = 0
END IF

! Assign default values to 'N_SUB_I' and 'N_MASTER_I'.
N_SUB_I = 1
N_MASTER_I = 1
! Check if reasonable values for N_SUB and N_MASTER are given.
IF (PRESENT(N_SUB)) N_SUB_I = N_SUB
IF ((N_SUB_I > 32) .OR. (N_SUB_I < 1)) THEN
  N_SUB_I = 1
  N_SUB = N_SUB_I
END IF
IF (PRESENT(N_MASTER)) N_MASTER_I = N_MASTER
IF (N_MASTER_I < 1) THEN
```

```
  N_MASTER_I = 1
  N_MASTER = N_MASTER_I
END IF
IF (N_MASTER_I*N_SUB_I > world_size) THEN
  iflag = INPUT_ERROR + 8
  RETURN
END IF
IF (PRESENT(BOX_SET)) THEN
  ! 'BOX_SET' should not be set for multiple masters.
  IF (N_MASTER_I*N_SUB_I > 1) THEN
    iflag = INPUT_ERROR + 8
  END IF
END IF

! When workers are used and the worker to master ratio is less than
! 2, that is (world_size-N_MASTER_I*N_SUB_I)/(N_SUB_I*N_MASTER_I)<2, return
! an error status. Notice that this sanity check is essentially performed for
! spawned masters before the spawning occurs.
IF (.NOT. spawned) THEN
  IF ( (N_MASTER_I*N_SUB_I /= world_size) .AND. &
     (N_MASTER_I*N_SUB_I > world_size/3) ) THEN
    iflag = INPUT_ERROR + 8
    RETURN
  END IF
END IF

! Initialize 'RESTART_I', the local copy of 'RESTART'.
RESTART_I = 0
IF (PRESENT(RESTART)) THEN
  IF ( (RESTART < 0) .OR. (RESTART > 2) ) THEN
    iflag = INPUT_ERROR + 7
    RETURN
  END IF
  ! Copy RESTART to 'RESTART_I'.
  RESTART_I = RESTART
END IF

BINSIZE_I = 1
IF (PRESENT(BINSIZE)) THEN
  IF (BINSIZE > 0) THEN
    BINSIZE_I = BINSIZE
  ELSE
    iflag = INPUT_ERROR + 9
    RETURN
  END IF
END IF

RETURN
END FUNCTION sanitycheck

SUBROUTINE verifyHeader(fileunit, sm, iflag)
IMPLICIT NONE
! Verify the file header of the checkpoint file specified by 'fileunit'.
!
! On input:
! fileunit - File unit number.
!
! On output:
! sm      - Number of masters in the checkpoint file.
! iflag    - Status to return.
!           0    Successfully verified.
!           >0   Error return.
!
INTEGER, INTENT(IN) :: fileunit
INTEGER, INTENT(OUT) :: sm
INTEGER, INTENT(OUT) :: iflag

! Verify 'N'.
READ(UNIT=fileunit, IOSTAT=ierr) chk_n
IF (ierr /= 0) THEN; iflag = FILE_ERROR + 1; RETURN; END IF
IF (chk_n /= N) THEN; iflag = FILE_ERROR + 3; RETURN; END IF
READ(UNIT=fileunit, IOSTAT=ierr) chk_l, chk_u, chk_eps, chk_switch, &
                                 chk_sd, sm
IF (ierr /= 0) THEN; iflag = FILE_ERROR + 1; RETURN; END IF
! Verify 'L' and 'U'.
IF ( (.NOT. ALL(chk_l == L)) .OR. .NOT. ALL(chk_u == U) ) THEN
  iflag = FILE_ERROR + 3
  RETURN
END IF
IF ( (chk_eps /= EPS_I) .OR. (chk_switch /= SWITCH_I) .OR. (chk_sd /= &
   N_SUB_I) ) THEN
  iflag = FILE_ERROR + 3
  RETURN
END IF

RETURN
```

```
END SUBROUTINE verifyHeader

SUBROUTINE worker(sub_status)
IMPLICIT NONE
! Perform the tasks of a worker who obtains points to be evaluated from
! master(s) and returns the function values.
!
! On Input:
! sub_status - An integer containing bit statuses of subdomains. If bit i is
!              set, subdomain i+1 has finished. Then, the worker will not
!              send requests to masters in the finished subdomain.
!
! On Output: None.
!
INTEGER, INTENT(IN) :: sub_status

! Local variables.
INTEGER :: c_active ! Counter for active masters that have not been done.
INTEGER :: c_idle ! Counter for idle active masters.
INTEGER, DIMENSION(N_SUB_I) :: c_masters ! An array of subdomain master
  ! counters. c_masters(i) counts the masters in subdomain i.
INTEGER :: i, j ! Counters.
INTEGER, ALLOCATABLE, DIMENSION(:,:) :: l_masters ! A 2-D array of
  ! master IDs. Row i holds master IDs for subdomain i. l_masters(i,j)
  ! holds the master ID for master j in subdomain i.
INTEGER :: loop_s ! Loop status. 0 is nonblocking and 1 is blocking.
INTEGER :: master_id ! Master ID.
INTEGER :: recv_tag ! The tag value received to indicate the message type.
INTEGER :: r_int ! The random integer converted from 'r_real'.
INTEGER, ALLOCATABLE, DIMENSION(:) :: seed
  ! The random seed for selecting masters.
INTEGER :: seed_size ! Compiler dependent size of array 'seed'.
INTEGER, ALLOCATABLE, DIMENSION(:,:) :: s_masters ! A 2-D array holding
  ! master busy statuses. Element (i,j) holds the status for the master whose
  ! ID is in l_masters(i,j). 0 is busy (with work), 1 is idle (no work), 2 is
  ! all done.
INTEGER :: sub_id ! Subdomain ID.
LOGICAL :: m_converted ! A flag indicating the worker is converted from a
  ! master.
REAL(KIND = R8) :: r_real ! The random number between 0 and 1.
REAL(KIND = R8), DIMENSION(N) :: tmpc ! Holding point coordinates.

! Allocate 'l_masters' and 's_masters'
ALLOCATE(l_masters(N_SUB_I, world_size))
ALLOCATE(s_masters(N_SUB_I, world_size))

! Initialization.
loop_s = 0
l_masters(:,:) = 0
c_masters(:) = 0
c_active = 0
m_converted = .FALSE.
! Fill in 'l_masters'.
DO i = 1, N_SUB_I
  IF (i == 1) THEN
    ! Root subdomain is always active.
    DO j = 1, N_MASTER_I
      l_masters(i, j) = (i - 1)*N_MASTER_I + j - 1
    END DO
    c_masters(i) = N_MASTER_I
    c_active = c_active + c_masters(i)
  ELSE ! Check 'sub_status' to determine if subdomain i is still active.
    IF (.NOT. BTEST(sub_status, i - 1)) THEN
      DO j = 1, N_MASTER_I
        l_masters(i, j) = (i - 1)*N_MASTER_I + j - 1
      END DO
      c_masters(i) = N_MASTER_I
      c_active = c_active + c_masters(i)
    END IF
  END IF
END DO
! The worker was converted from a master if not all masters are active.
IF (c_active < N_MASTER_I*N_SUB_I) m_converted = .TRUE.

IF (.NOT. m_converted) THEN
  ! Notify others that it has passed the initialization.
  array_masters(:) = 1
  CALL MPI_ALLTOALL(array_masters, 1, MPI_INTEGER, array_masters, 1, &
                    MPI_INTEGER, MPI_COMM_WORLD, ierr)
  IF (SUM(array_masters) /= world_size) THEN
    iflag = DMPI_ERROR + 7
    RETURN
  END IF
END IF

! Assume all masters busy initially.
```

```
s_masters(:,:) = 0
c_idle = 0
! Assign 'seed' for random number generation.
CALL RANDOM_SEED(SIZE = seed_size)
ALLOCATE(SEED(seed_size))
seed(1:seed_size) = 32749 + 128*myid
CALL RANDOM_SEED(PUT = seed(1:seed_size))
DEALLOCATE(SEED)

! Receive messages from masters.
OUTER: DO
  IF (spawn_requested_global) THEN
    CALL spawnInit(ierr)
    IF (ierr /= 0) THEN
      ! Abort spawning.
      spawn_initiated = .FALSE.
    END IF
  END IF
  IF (spawn_initiated) THEN
    CALL spawnFinalize()
    DEALLOCATE(l_masters)
    DEALLOCATE(s_masters)
    ALLOCATE(l_masters(N_SUB_I, world_size))
    ALLOCATE(s_masters(N_SUB_I, world_size))
    CALL updateWorker(c_masters, c_active, l_masters, s_masters, sub_status)
    c_idle = 0
    m_converted = .FALSE.
    CALL MPI_BARRIER(xw_comm, ierr)
  END IF
  IF (loop_s == 0) THEN ! Nonblocking loop.
    ! Send a nonblocking request to a randomly selected busy master.
    CALL RANDOM_NUMBER(HARVEST = r_real)
    r_int = INT(r_real*(c_active - c_idle)) + 1
    ! Find the master ID from 'l_masters' with status = 0 (busy)
    ! in 's_masters'.
    master_id = 0
    INNER2: DO i = 1, N_SUB_I
      DO j = 1, N_MASTER_I
        IF (s_masters(i,j) == 0) THEN
          r_int = r_int - 1
          IF (r_int == 0) THEN
            master_id = l_masters(i,j)
            EXIT INNER2
          END IF
        END IF
      END DO
    END DO INNER2
    ! Mark the message if sent from a master-converted worker.
    IF (m_converted) THEN; buffer(1) = 1; ELSE; buffer(1) = 0; END IF
    CALL MPI_SEND(buffer, bufsize, MPI_BYTE, master_id, &
                  NONBLOCK_REQ, xw_comm, ierr)
  ELSE ! Since loop_s == 1, send a blocking request.
    ! Assign this worker to one of the active masters.
    r_int = MOD(myid - c_active, c_active) + 1
    ! Find the master ID from 'l_masters' with status = 0 (busy)
    ! in 's_masters'.
    INNER3: DO i = 1, N_SUB_I
      DO j = 1, N_MASTER_I
        IF (s_masters(i,j) == 0) THEN
          r_int = r_int - 1
          IF (r_int == 0) THEN
            master_id = l_masters(i, j)
            EXIT INNER3
          END IF
        END IF
      END DO
    END DO INNER3
    ! Mark the message if sent from a master-converted worker.
    IF (m_converted) THEN; buffer(1) = 1; ELSE; buffer(1) = 0; END IF
    CALL MPI_SEND(buffer, bufsize, MPI_BYTE, master_id, &
                  BLOCK_REQ, xw_comm, ierr)
  END IF
  ! Loop checking responses from master(s).
  INNER: DO
    CALL MPI_RECV(buffer, bufsize, MPI_BYTE, MPI_ANY_SOURCE, &
                  MPI_ANY_TAG, xw_comm, mstatus, ierr)
    recv_tag = mstatus(MPI_TAG)
    SELECT CASE (recv_tag)
      CASE (POINT_NBREQ) ! A function evaluation task.
        ! Loop evaluating all the points in the task. 'buffer(2)' holds
        ! the number of points to be evaluated.
        DO i = 1, INT(buffer(2))
          ! Extract a set of point coordinates.
          tmpc(:) = buffer(3 + (i - 1)*N:(i - 1)*N + N + 2)
          ! Put the function value to the buffer.
          buffer(3 + (i - 1)*2) = OBJ_FUNC(tmpc, iflag)
```

```
    ! 'iflag' is stored after the function value.
    buffer(4 + (i - 1)*2) = iflag
  END DO
  ! Return the function value(s).
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, &
              mstatus(MPI_SOURCE), FUNCVAL, xw_comm, ierr)
CASE (POINT_BREQ) ! An evaluation task responding to a blocking request.
  IF (s_masters(mstatus(MPI_SOURCE)/N_MASTER_I + 1, &
     MOD(mstatus(MPI_SOURCE), N_MASTER_I) + 1) == 1) THEN
    ! The master has changed from idle to busy. Mark it back to be busy.
    s_masters(mstatus(MPI_SOURCE)/N_MASTER_I + 1, &
      MOD(mstatus(MPI_SOURCE), N_MASTER_I) + 1) = 0
    ! Update 'c_idle'.
    c_idle = c_idle - 1
  END IF
  ! Loop changes to be nonblocking since at least one master is busy.
  loop_s = 0
  ! Evaluate the points in the task.
  DO i = 1, INT(buffer(2))
    tmpc(:) = buffer(3 + (i - 1)*N:(i - 1)*N + N + 2)
    buffer(3 + (i - 1)*2) = OBJ_FUNC(tmpc, iflag)
    buffer(4 + (i - 1)*2) = iflag
  END DO
  ! Return the function value(s).
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, &
              mstatus(MPI_SOURCE), FUNCVAL, xw_comm, ierr)
CASE (NO_POINT) ! No more points to evaluate from this master for this
  ! iteration.
  IF (s_masters(mstatus(MPI_SOURCE)/N_MASTER_I + 1, &
     MOD(mstatus(MPI_SOURCE), N_MASTER_I) + 1) == 0) THEN
    ! This master has changed from busy to idle. Mark it as idle.
    s_masters(mstatus(MPI_SOURCE)/N_MASTER_I + 1, &
      MOD(mstatus(MPI_SOURCE), N_MASTER_I) + 1) = 1
    ! Update 'c_idle'.
    c_idle = c_idle + 1
  END IF
  EXIT INNER ! Exit INNER loop to request work from other masters.
CASE (MYALL_DONE) ! This master has finished all iterations.
  ! Find the subdomain ID 'sub_id'. Mark off all the master(s) in this
  ! subdomain.
  sub_id = INT(mstatus(MPI_SOURCE)/N_MASTER_I) + 1
  IF (c_masters(sub_id) /= 0) THEN
    ! Update 'c_active' and 'c_masters'.
    c_active = c_active - c_masters(sub_id)
    c_masters(sub_id) = 0
    ! Mark the status of all masters in this subdomain to be 2 (alldone)
    ! and update 'c_idle'.
    DO i = 1, N_MASTER_I
      IF (s_masters(sub_id,i) == 1) c_idle = c_idle - 1
      s_masters(sub_id,i) = 2
    END DO
  END IF
  IF (c_active == 0) THEN
    ! All masters have done all the work. The worker terminates itself
    ! if it was not converted from a master; otherwise, it waits for the
    ! final termination message if it was converted from a master to a
    ! worker.
    IF (.NOT. m_converted) THEN
      EXIT OUTER
    ELSE
      CYCLE INNER
    END IF
  END IF
  EXIT INNER  ! Exit to request work from other masters.
CASE (BLOCK_REQ)
  ! This worker was converted from a master, so there might be some
  ! delayed requests from workers. Send 'MYALL_DONE' to let them update
  ! their master lists.
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, &
              mstatus(MPI_SOURCE), MYALL_DONE, xw_comm, ierr)
  ! Send a message to update 'c_alldone' at the root master.
  buffer(1) = 1
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, &
              0, COUNT_DONE, xw_comm, ierr)
CASE (NONBLOCK_REQ)
  ! React similarly as for 'BLOCK_REQ'.
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, &
              mstatus(MPI_SOURCE), MYALL_DONE, xw_comm, ierr)
  ! Update 'c_alldone' at the root master.
  buffer(1) = 1
  CALL MPI_SEND(buffer, bufsize, MPI_BYTE, &
              0, COUNT_DONE, xw_comm, ierr)
CASE (SPAWN_REQ)
  ! There is a pending spawn request.
  spawn_requested_global = .TRUE.
  EXIT INNER
```

```
    CASE (TERMINATE)
      ! Received a termination message and pass it to its neighbors.
      i = myid*2 + 1
      j = myid*2 + 2
      IF (i <= N_MASTER_I*N_SUB_I - 1) THEN
        CALL MPI_SEND(buffer, bufsize, MPI_BYTE, i, &
                      TERMINATE, xw_comm, ierr)
      END IF
      IF (j <= N_MASTER_I*N_SUB_I - 1) THEN
        CALL MPI_SEND(buffer, bufsize, MPI_BYTE, j, &
                      TERMINATE, xw_comm, ierr)
      END IF
      EXIT OUTER
  END SELECT
 END DO INNER
 IF (c_active == c_idle) THEN
   ! There is no busy master, so set the loop status to be blocking.
   loop_s = 1
   ! Before move to the blocking loop, assume all active masters are busy.
   DO i = 1, N_SUB_I
     DO j = 1, N_MASTER_I
       IF (s_masters(i, j) == 1) s_masters(i, j) = 0
     END DO
   END DO
   c_idle = 0
 ElSE ! There is at least one busy master, set the loop status to be
   ! nonblocking.
   loop_s = 0
 END IF
END DO OUTER

RETURN
END SUBROUTINE worker

SUBROUTINE spawnInit(spawnErr)
IMPLICIT NONE
! Initialize a spawning event by spawning the new master(s). However, state
! update is handled later.
!
! On input: None.
!
! On output:
! spawnErr - An error flag for the subroutine. It is set to 0 if the spawning
!            is executed without error, and 1 otherwise.
!
INTEGER, INTENT(OUT) :: spawnErr

! Local variables.
INTEGER :: root_proc ! Root process for executing 'MPI_COMM_SPAWN'.
INTEGER :: size_hostfile ! The number of hosts specified in the host file.

! Inialize 'spawnErr'.
spawnErr = 0

! Initialize 'num_children'.
num_children = N_MASTER_I

! Update spawning status.
spawn_requested_local = .FALSE.
spawn_requested_global = .FALSE.
spawn_initiated = .TRUE.

! Prepare to spawn.
root_proc = 0
CALL initProcList(xw_comm)
IF (myid == 0) THEN
  CALL makeHostFile(num_children, hostfile, len(hostfile), size_hostfile)
  IF (size_hostfile < num_children) THEN
    spawnErr = 1
    RETURN
  END IF
  CALL MPI_INFO_CREATE(info, ierr)
  CALL MPI_INFO_SET(info, "hostfile", hostfile, ierr)
END IF
! Free old 'child_comm' before spawning.
IF (child_comm /= MPI_COMM_NULL) THEN
  CALL MPI_COMM_FREE(child_comm, ierr)
END IF
! Spawn the master.
CALL MPI_COMM_SPAWN(exec_name, MPI_ARGV_NULL, num_children, info, &
                    root_proc, xw_comm, child_comm, spawnErrcodes, ierr)

RETURN
END SUBROUTINE spawnInit

SUBROUTINE spawnFinalize()
```

```
! Finalize a spawning event. This consists in updating state for initial
! masters and workers. Spawned masters update state separately.
!
! On input: None.
!
! On output: None.
!
! Local variables.
INTEGER :: tempint
LOGICAL :: order

! Update spawning status and count.
spawn_initiated = .FALSE.
spawn_count = spawn_count + num_children
! Update other variables to account for the spawned master(s).
N_MASTER_I = N_MASTER_I + num_children
world_size = world_size + num_children
gbuffer_len = 100*N_MASTER_I*box_size
q_counter = 0
! Allocate memory for arrays used to manipulate communicators.
IF (ALLOCATED(master_ranks)) DEALLOCATE(master_ranks)
ALLOCATE(master_ranks(N_MASTER_I))
IF (ALLOCATED(worker_ranks)) DEALLOCATE(worker_ranks)
ALLOCATE(worker_ranks(N_WORKER_I))
IF (ALLOCATED(trans_ranks)) DEALLOCATE(trans_ranks)
ALLOCATE(trans_ranks(1 + num_children))
! Update arrays to account for spawned masters.
IF (ALLOCATED(gbuffer)) THEN ! Update 'gbuffer'.
  DEALLOCATE(gbuffer)
END IF
ALLOCATE(gbuffer(gbuffer_len))
gbuffer(:) = 0
IF (ALLOCATED(lc_convex)) THEN ! Update 'lc_convex'.
  tempint = SIZE(lc_convex)
  DEALLOCATE(lc_convex)
  ALLOCATE(lc_convex(tempint + num_children))
  ! 'lc_convex' is recomputed every iteration, so it can be reset to all zeros.
  lc_convex(:) = 0
END IF
IF (ALLOCATED(displs)) THEN ! Update 'displs'.
  DEALLOCATE(displs)
END IF
ALLOCATE(displs(N_MASTER_I))
! 'displs' is recomputed every iteration, so it can be reset to all zeros.
displs(:) = 0
IF (ALLOCATED(update_array)) THEN ! Update 'update_array'.
  tempint = SIZE(update_array)
  DEALLOCATE(update_array)
  ALLOCATE(update_array(tempint + num_children))
  ! Progress was interrupted by spawn request, so we assume no updates.
  update_array(:) = 0
END IF
IF (ALLOCATED(array_masters)) THEN ! Update 'array_masters'.
  tempint = SIZE(array_masters)
  DEALLOCATE(array_masters)
  ALLOCATE(array_masters(tempint + num_children))
  array_masters(:) = 0
  DO i = 1, N_MASTER_I
    array_masters(i) = INT(myid/N_MASTER_I)*N_MASTER_I + i - 1
  END DO
END IF
IF (ALLOCATED(q_worker)) THEN ! Update 'q_worker'.
  tempint = SIZE(q_worker)
  DEALLOCATE(q_worker)
  ALLOCATE(q_worker(tempint + num_children))
  q_worker(:) = 0
END IF
IF (ALLOCATED(b_worker)) THEN ! Update 'b_worker'.
  tempint = SIZE(b_worker)
  DEALLOCATE(b_worker)
  ALLOCATE(b_worker(tempint + num_children))
  b_worker(:) = 0
END IF
! Initialize the state transfer to the spawned master(s).
IF (myid == 0) THEN
  CALL initStateTransfer()
END IF
! Assign ranks to the old masters.
DO i = 1, N_MASTER_I - num_children
  master_ranks(i) = i - 1
END DO
! The ranks of the new masters begin at 'world_size - 1'.
DO i = 1, num_children
  master_ranks((N_MASTER_I - num_children) + i) &
    = (world_size - num_children) + (i - 1)
END DO
```

```
! Update 'worker_ranks'.
DO i = 1, N_WORKER_I
  worker_ranks(i) = (N_MASTER_I - num_children) + (i - 1)
END DO
! Assign state transfer ranks.
trans_ranks(1) = 0
DO i = 1, num_children
  trans_ranks(i + 1) = (world_size - num_children) + (i - 1)
END DO
! Merge intercomm.
order = .FALSE.
CALL MPI_INTERCOMM_MERGE(child_comm, order, temp_comm_world, ierr)
! Create expanded communicator for new set of masters.
CALL MPI_COMM_GROUP(temp_comm_world, temp_group_world, ierr)
CALL MPI_GROUP_INCL(temp_group_world, N_MASTER_I, master_ranks, &
                xm_group, ierr)
CALL MPI_COMM_CREATE(temp_comm_world, xm_group, xm_comm, ierr)
! Create expanded communicator for new world (= old world + spawned masters).
CALL MPI_GROUP_INCL(temp_group_world, N_WORKER_I, worker_ranks, &
                temp_group_worker, ierr)
CALL MPI_GROUP_UNION(xm_group, temp_group_worker, xw_group, ierr)
CALL MPI_COMM_CREATE(temp_comm_world, xw_group, xw_comm, ierr)
! Create communicator for state transfer.
CALL MPI_GROUP_INCL(temp_group_world, 1 + num_children, trans_ranks, &
                trans_group, ierr)
CALL MPI_COMM_CREATE(temp_comm_world, trans_group, trans_comm, ierr)
! Get new ranks.
CALL MPI_COMM_RANK(xw_comm, myid, ierr)
IF (myid < N_MASTER_I) THEN
  CALL MPI_COMM_RANK(xm_comm, mygid, ierr)
END IF
! Update 'PROCID' and 'N_MASTER'.
PROCID = myid
N_MASTER = N_MASTER_I
! Transfer state to spawned master(s).
IF (myid == 0) THEN
  CALL bcastState()
END IF
! Set the iteration at which the last spawning event occured.
switch_iter = t

RETURN
END SUBROUTINE spawnFinalize
```

```
SUBROUTINE updateWorker(c_mast, c_act, l_mast, s_mast, sub_status)
IMPLICIT NONE
! Update certain aspects of the state of a worker that are not updated by the
! subroutine 'spawnFinalize'. These aspects are variables and arrays that are
! defined within the 'worker' subroutine itself, and their update is taken
! directly from the 'worker' subroutine.
!
! On input:
! c_mast    - An array of subdomain master counters. c_masters(i) counts the
!             masters in subdomain i.
! c_act     - Counter for idle active masters.
! l_mast    - A 2-D array of master IDs. Row i holds master IDs for subdomain
!             i. l_masters(i,j) holds the master ID for master j in subdomain
!             i.
! s_mast    - A 2-D array holding master busy statuses. Element (i,j) holds
!             the status for the master whose ID is in l_masters(i,j). 0 is
!             busy (with work), 1 is idle (no work), 2 is all done.
! sub_status - An integer containing bit statuses of subdomains. If bit i is
!             set, subdomain i+1 has finished. Then, the worker will not
!             send requests to masters in the finished subdomain.
!
! On output:
! c_mast    - Same as upon input, but recomputed with new 'N_MASTER_I'.
! c_act     - Same as upon input, but recomputed with new 'N_MASTER_I'.
! l_mast    - Same as upon input, but recomputed with new 'N_MASTER_I'.
! s_mast    - Same as upon input, but recomputed with new 'N_MASTER_I'.
!
INTEGER, DIMENSION(:), INTENT(INOUT) :: c_mast
INTEGER, INTENT(INOUT) :: c_act
INTEGER, DIMENSION(:,:), INTENT(INOUT) :: l_mast
INTEGER, DIMENSION(:,:), INTENT(INOUT) :: s_mast
INTEGER, INTENT(IN) :: sub_status

! Local variables.
INTEGER :: i, j ! Loop counters.

! Initializations.
c_act = 0
c_mast(:) = 0
l_mast(:, :) = 0
s_mast(:, :) = 0
! Update variables and arrays.
```

```
DO i = 1, N_SUB_I
  IF (i == 1) THEN
    ! Root subdomain is always active.
    DO j = 1, N_MASTER_I
      l_mast(i,j) = (i - 1)*N_MASTER_I + j - 1
    END DO
    c_mast(i) = N_MASTER_I
    c_act = c_act + c_mast(i)
  ELSE ! Check 'sub_status' to determine if subdomain i is still active.
    IF (.NOT. BTEST(sub_status, i - 1)) THEN
      DO j = 1, N_MASTER_I
        l_mast(i,j) = (i - 1)*N_MASTER_I + j - 1
      END DO
      c_mast(i) = N_MASTER_I
      c_act = c_act + c_mast(i)
    END IF
  END IF
END DO

RETURN
END SUBROUTINE updateWorker

SUBROUTINE bcastState()
IMPLICIT NONE
! Broadcast state of lead master to the spawned master(s). The exact state of
! the lead master is not sent, because certain aspects of state need to be
! taylored to the spawned master(s).
!
! On input: None.
!
! On output: None.
!
! Local variables.
INTEGER :: comm ! Communicator used for state transfer.
INTEGER :: myarray_masters = 1 ! Indices for 'mySIZE' array.
INTEGER :: mybuffer = 2
INTEGER :: myb_worker = 3
INTEGER :: mychk_l = 4
INTEGER :: mychk_len = 5
INTEGER :: mychk_u = 6
INTEGER :: mydispls = 7
INTEGER :: mygbuffer = 8
INTEGER :: mylbuffer = 9
INTEGER :: mylc_convex = 10
INTEGER :: mym_comm =11
INTEGER :: mym_group = 12
INTEGER :: myq_worker = 13
INTEGER :: myupdate_array = 14
INTEGER :: root ! Root value for broadcasting state.

! Set values for 'comm' and 'root'.
comm = trans_comm
root = 0
! Set buffer for global integers.
IF (.NOT. spawned) THEN
  statebuf_integer(1:5) = (/ alloc_err, b_id, BINSIZE_I, bits_sub, &
                           boxset_ind /)
  statebuf_integer(6:10) = (/ box_size, bufsize, c_alldone, c_bits_sub, &
                            check_t /)
  statebuf_integer(11:15) = (/ chk_n, chk_sd, chk_sm, chk_sm1, chk_switch /)
  statebuf_integer(16:20) = (/ col, eval_c, eval_c_i, gbuffer_len, gc_convex /)
  statebuf_integer(21:25) = (/ group_world, iflag, i_start, lbc, lbuffer_len /)
  statebuf_integer(26:30) = (/ loc_pid, loc_sid, loc_eid, loc_val, loc_c /)
  statebuf_integer(31:35) = (/ loc_side, loc_diam, N_MASTER_I, N_SUB_I, &
                            spawn_count /)
  statebuf_integer(36:40) = (/ q_counter, recv_tag, RESTART_I, set_size, &
                           size_r8 /)
  statebuf_integer(41:45) = (/ stop_rule, SWITCH_I, t, update_counter, &
                            world_size /)
  statebuf_integer(46:49) = (/ col_w, row_w, N_I, switch_iter /)
END IF
! Broadcast integer's.
CALL MPI_BCAST(statebuf_integer, SIZE(statebuf_integer), MPI_INTEGER, root, &
               comm, ierr)
IF (spawned) THEN
  CALL processStatebufInteger()
END IF
! Set buffer for global real's.
IF (.NOT. spawned) THEN
  statebuf_real(1:5) = (/ chk_eps, dia, dia_i, dia_limit, EPS_I /)
  statebuf_real(6:10) = (/ fmin_old, FMIN_I, MIN_SEP_I, tmpf, FMIN /)
  statebuf_real(11:11) = (/ EPS4N /)
END IF
! Broadcast real's.
CALL MPI_BCAST(statebuf_real, SIZE(statebuf_real)*size_r8, MPI_BYTE, root, &
               comm, ierr)
```

```
IF (spawned) THEN
  CALL processStatebufReal()
END IF
! Set buffer for global logical's.
IF (.NOT. spawned) THEN
  statebuf_logical(1) = do_it
END IF
! Broadcast logical's.
CALL MPI_BCAST(statebuf_logical, SIZE(statebuf_logical), MPI_LOGICAL, root, &
             comm, ierr)
IF (spawned) THEN
  CALL processStatebufLogical()
END IF
! Broadcast optional parameters.
IF (PRESENT(SWITCH)) THEN
  CALL MPI_BCAST(SWITCH, 1, MPI_INTEGER, root, comm, ierr)
END IF
IF (PRESENT(MAX_ITER)) THEN
  CALL MPI_BCAST(MAX_ITER, 1, MPI_INTEGER, root, comm, ierr)
END IF
IF (PRESENT(MAX_EVL)) THEN
  CALL MPI_BCAST(MAX_EVL, 1, MPI_INTEGER, root, comm, ierr)
END IF
IF (PRESENT(NUM_BOX)) THEN
  CALL MPI_BCAST(NUM_BOX, 1, MPI_INTEGER, root, comm, ierr)
END IF
IF (PRESENT(N_SUB)) THEN
  CALL MPI_BCAST(N_SUB, 1, MPI_INTEGER, root, comm, ierr)
END IF
IF (PRESENT(N_MASTER)) THEN
  CALL MPI_BCAST(N_MASTER, 1, MPI_INTEGER, root, comm, ierr)
END IF
IF (PRESENT(RESTART)) THEN
  CALL MPI_BCAST(RESTART, 1, MPI_INTEGER, root, comm, ierr)
END IF
IF (PRESENT(BINSIZE)) THEN
  CALL MPI_BCAST(BINSIZE, 1, MPI_INTEGER, root, comm, ierr)
END IF
IF (PRESENT(MIN_DIA)) THEN
  CALL MPI_BCAST(MIN_DIA, 1, MPI_REAL, root, comm, ierr)
END IF
IF (PRESENT(OBJ_CONV)) THEN
  CALL MPI_BCAST(OBJ_CONV, 1, MPI_REAL, root, comm, ierr)
END IF
IF (PRESENT(EPS)) THEN
  CALL MPI_BCAST(EPS, 1, MPI_REAL, root, comm, ierr)
END IF
IF (PRESENT(MIN_SEP)) THEN
  CALL MPI_BCAST(MIN_SEP, 1, MPI_REAL, root, comm, ierr)
END IF
IF (PRESENT(W)) THEN
  CALL MPI_BCAST(W, N*size_r8, MPI_BYTE, root, comm, ierr)
END IF
! Broadcast global character arrays.
CALL MPI_BCAST(cpfile, 30, MPI_CHARACTER, root, comm, ierr)
CALL MPI_BCAST(cpfile1, 30, MPI_CHARACTER, root, comm, ierr)
CALL MPI_BCAST(cpfile2, 34, MPI_CHARACTER, root, comm, ierr)
CALL MPI_BCAST(str, 30, MPI_CHARACTER, root, comm, ierr)
! Broadcast global integer arrays.
IF (mySIZE(myarray_masters) /= -1) THEN
  CALL MPI_BCAST(array_masters, mySIZE(myarray_masters), MPI_INTEGER, root, &
               comm, ierr)
END IF
IF (mySIZE(myb_worker) /= -1) THEN
  CALL MPI_BCAST(b_worker, mySIZE(myb_worker), MPI_INTEGER, root, &
               comm, ierr)
END IF
IF (mySIZE(mychk_len) /= -1) THEN
  CALL MPI_BCAST(chk_len, mySIZE(mychk_len), MPI_INTEGER, root, &
               comm, ierr)
END IF
IF (mySIZE(mydispls) /= -1) THEN
  CALL MPI_BCAST(displs, mySIZE(mydispls), MPI_INTEGER, root, &
               comm, ierr)
END IF
IF (mySIZE(mylc_convex) /= -1) THEN
  CALL MPI_BCAST(lc_convex, mySIZE(mylc_convex), MPI_INTEGER, root, &
               comm, ierr)
END IF
IF (mySIZE(mym_comm) /= -1) THEN
  CALL MPI_BCAST(m_comm, mySIZE(mym_comm), MPI_INTEGER, root, &
               comm, ierr)
END IF
IF (mySIZE(mym_group) /= -1) THEN
  CALL MPI_BCAST(m_group, mySIZE(mym_group), MPI_INTEGER, root, &
               comm, ierr)
```

```
END IF
IF (mySIZE(myupdate_array) /= -1) THEN
  CALL MPI_BCAST(update_array, mySIZE(myupdate_array), MPI_INTEGER, root, &
                 comm, ierr)
END IF
IF (mySIZE(myq_worker) /= -1) THEN
  CALL MPI_BCAST(q_worker, mySIZE(myq_worker), MPI_INTEGER, root, &
                 comm, ierr)
END IF
CALL MPI_BCAST(sub_divisor, 2, MPI_INTEGER, root, comm, ierr)
CALL MPI_BCAST(sub_index, 2, MPI_INTEGER, root, comm, ierr)
CALL MPI_BCAST(STATUS, N_SUB_I, MPI_INTEGER, root, comm, ierr)
! Broadcast global real arrays.
IF (mySIZE(mychk_l) /= -1) THEN
  CALL MPI_BCAST(chk_l, mySIZE(mychk_l)*size_r8, MPI_BYTE, root, &
                 comm, ierr)
END IF
IF (mySIZE(mychk_u) /= -1) THEN
  CALL MPI_BCAST(chk_u, mySIZE(mychk_u)*size_r8, MPI_BYTE, root, &
                 comm, ierr)
END IF
IF (mySIZE(mybuffer) /= -1) THEN
  CALL MPI_BCAST(buffer, mySIZE(mybuffer)*size_r8, MPI_BYTE, root, &
                 comm, ierr)
END IF
CALL MPI_BCAST(dim_len, 2*size_r8, MPI_BYTE, root, comm, ierr)
IF (mySIZE(mygbuffer) /= -1) THEN
  CALL MPI_BCAST(gbuffer, mySIZE(mygbuffer)*size_r8, MPI_BYTE, root, &
                 comm, ierr)
END IF
IF (mySIZE(mylbuffer) /= -1) THEN
  CALL MPI_BCAST(lbuffer, mySIZE(mylbuffer)*size_r8, MPI_BYTE, root, &
                 comm, ierr)
END IF
CALL MPI_BCAST(tmp_x, N*size_r8, MPI_BYTE, root, comm, ierr)
CALL MPI_BCAST(UmL, N*size_r8, MPI_BYTE, root, comm, ierr)
CALL MPI_BCAST(unit_x, N*size_r8, MPI_BYTE, root, comm, ierr)
CALL MPI_BCAST(unit_x_i, N*size_r8, MPI_BYTE, root, comm, ierr)
CALL MPI_BCAST(W_I, N*size_r8, MPI_BYTE, root, comm, ierr)
CALL MPI_BCAST(L, N*size_r8, MPI_BYTE, root, comm, ierr)
CALL MPI_BCAST(U, N*size_r8, MPI_BYTE, root, comm, ierr)
CALL MPI_BCAST(X, N*size_r8, MPI_BYTE, root, comm, ierr)
RETURN
END SUBROUTINE bcastState

SUBROUTINE initCheckpoint()
IMPLICIT NONE
! Initialize the checkpoint mechanism.
!
! On input: None.
!
! On output: None.
!
! Check the user-specified restart condition.
IF (RESTART_I > 0) THEN
  ! Form the checkpoint filename with subdomain ID and master ID.
  WRITE(str, 3000) (INT(myid/N_MASTER_I) + 1)/1000.0, mygid/1000.0
  3000 FORMAT(f4.3, f4.3)
  cpfile = 'pvtdirchkpt.dat'//str
  IF (RESTART_I == 1) THEN ! Record checkpoint logs.
    ! Each master writes N, L, U, EPS_I, SWITCH_I, and N_SUB_I,
    ! and N_MASTER_I as the checkpoint file header.
    OPEN(UNIT=CHKSUNIT, FILE=cpfile, FORM='UNFORMATTED', STATUS='NEW', &
        POSITION='REWIND', IOSTAT=ierr)
    IF (ierr /= 0) THEN
      iflag = FILE_ERROR
      RETURN
    END IF
    WRITE(UNIT=CHKSUNIT, IOSTAT=ierr) N
    IF (ierr /= 0) THEN
      iflag = FILE_ERROR + 2
      CLOSE(CHKSUNIT)
      RETURN
    END IF
    WRITE(UNIT=CHKSUNIT, IOSTAT=ierr) L, U, EPS_I, SWITCH_I, &
      N_SUB_I, N_MASTER_I
    IF (ierr /= 0) THEN
      iflag = FILE_ERROR + 2
      CLOSE(CHKSUNIT)
      RETURN
    END IF
    CLOSE(CHKSUNIT)
  END IF
  IF (RESTART_I == 2) THEN ! Restart from checkpoint logs.
    ! Initialize the list to hold INITLEN (defined in VTDIRECT_CHKPT) logs.
```

```
CALL initList(INITLEN)
IF (mygid == 0) THEN ! Root subdomain master.
  ! Each root subdomain verifies if the checkpoint log file matches with
  ! the current setting.
  OPEN(UNIT=CHKSUNIT, FILE=cpfile, FORM='UNFORMATTED', STATUS='OLD', &
      POSITION='REWIND', IOSTAT=ierr)
  IF (ierr /= 0) THEN
    iflag = FILE_ERROR + 1
    CLOSE(CHKSUNIT)
    ! Notify other masters (if any).
    CALL MPI_BCAST(0, 1, MPI_INTEGER, 0, xm_comm, ierr)
    RETURN
  END IF
  ! Allocate 'chk_l' and 'chk_u' to verify 'L' and 'U' bounds.
  ALLOCATE(chk_l(N))
  ALLOCATE(chk_u(N))
  CALL verifyHeader(CHKSUNIT, chk_sm, iflag)
  IF (iflag /= 0) THEN
    CLOSE(CHKSUNIT)
    DEALLOCATE(chk_l)
    DEALLOCATE(chk_u)
    ! Notify other masters (if any).
    CALL MPI_BCAST(0, 1, MPI_INTEGER, 0, xm_comm, ierr)
    RETURN
  END IF
  ! Free 'chk_l' and 'chk_u'.
  DEALLOCATE(chk_l)
  DEALLOCATE(chk_u)
  IF (chk_sm /= N_MASTER_I) THEN
    ! Close unit.
    CLOSE(CHKSUNIT)
    ! Different number of masters were used for the checkpointing logs.
    ! Verify each checkpointing file header with the current setting.
    ! Allocate 'chk_l' and 'chk_u' to verify 'L' and 'U' bounds.
    ALLOCATE(chk_l(N))
    ALLOCATE(chk_u(N))
    DO i = 1, chk_sm - 1
      ! Form the checkpoint filename with other master IDs.
      WRITE(str, 1000) (INT(myid/N_MASTER_I) + 1)/1000.0, mygid + i/1000.0
      cpfile1 = 'pvtdirchkpt.dat'//str
      OPEN(UNIT=CHKSUNIT, FILE=cpfile1, FORM='UNFORMATTED', STATUS='OLD', &
          POSITION='REWIND', IOSTAT=ierr)
      IF (ierr /= 0) THEN
        iflag = FILE_ERROR + 1;
        CLOSE(CHKSUNIT)
        ! Notify other masters (if any).
        CALL MPI_BCAST(0, 1, MPI_INTEGER, 0, xm_comm, &
                  ierr)
        RETURN
      END IF
      CALL verifyHeader(CHKSUNIT, chk_sm1, iflag)
      IF (iflag /= 0) THEN
        CLOSE(CHKSUNIT)
        DEALLOCATE(chk_l)
        DEALLOCATE(chk_u)
        ! Notify other masters (if any).
        CALL MPI_BCAST(0, 1, MPI_INTEGER, 0, xm_comm, &
                  ierr)
        RETURN
      END IF
      ! Verify if 'chk_sm1' is same as 'chk_sm' in the checkpoint file of
      ! the root subdomain master.
      IF (chk_sm1 /= chk_sm) THEN
        iflag = FILE_ERROR + 3
        CLOSE(CHKSUNIT)
        DEALLOCATE(chk_l)
        DEALLOCATE(chk_u)
        ! Notify other masters (if any).
        CALL MPI_BCAST(0, 1, MPI_INTEGER, 0, xm_comm, &
                  ierr)
        RETURN
      END IF
      CLOSE(CHKSUNIT)
    END DO
    ! Broadcast the number of existing masters 'chk_sm' read from the
    ! checkpoint file to all subdomain masters so that each master
    ! reads the logs from all checkpointing files.
    CALL MPI_BCAST(chk_sm, 1, MPI_INTEGER, 0, xm_comm, &
              ierr)
    ! Allocate 'chk_len'.
    ALLOCATE(chk_len(chk_sm))
    ! Open each checkpoint file, read the header, and keep it open.
    DO j = 1, chk_sm
      WRITE(str, 1000) (INT(myid/N_MASTER_I) + 1)/1000.0, (j - 1)/1000.0
      cpfile1 = 'pvtdirchkpt.dat'//str
      OPEN(UNIT=CHKSUNIT + j, FILE=cpfile1, FORM="UNFORMATTED", &
```

```
      STATUS="OLD", POSITION="REWIND", IOSTAT=ierr)
  IF (ierr /= 0) THEN; iflag = FILE_ERROR; EXIT; END IF
  ! Bypass N, L, U, EPS_I, SWITCH_I, N_SUB_I, and N_MASTER_I.
  READ(UNIT=CHKSUNIT + j, IOSTAT=ierr) chk_n
  IF (ierr /= 0) THEN; iflag = FILE_ERROR + 1; EXIT; END IF
  READ(UNIT=CHKSUNIT + j, IOSTAT=ierr) chk_l, chk_u, chk_eps, &
                              chk_switch, chk_sd, chk_sm
  IF (ierr /= 0) THEN; iflag = FILE_ERROR + 1; EXIT; END IF
 END DO
 ! Release the memory for 'chk_l' and 'chk_u'.
 DEALLOCATE(chk_l)
 DEALLOCATE(chk_u)
 IF (iflag /= 0) THEN; CLOSE(CHKSUNIT); RETURN; END IF
 ! Open a new checkpoint file to record the new sequence of
 ! function evaluations with the changed number of masters.
 WRITE(str, 2000) (INT(myid/N_MASTER_I) + 1)/1000.0, mygid/1000.0
 2000 FORMAT(f4.3, f4.3)
 cpfile2 = 'pvtdirchkpt.dat.new'//str
 OPEN(UNIT=CHKRUNIT + chk_sm, FILE=cpfile2, FORM="UNFORMATTED", &
     STATUS="NEW", IOSTAT=ierr)
 IF (ierr /= 0) THEN; iflag = FILE_ERROR; RETURN; END IF
 ! Write the new header.
 WRITE(UNIT=CHKRUNIT + chk_sm, IOSTAT=ierr) N
 IF (ierr /= 0) THEN; iflag = FILE_ERROR + 2; RETURN; END IF
 WRITE(UNIT=CHKRUNIT + chk_sm, IOSTAT=ierr) L, U, EPS_I, SWITCH_I, &
                                  N_SUB_I, N_MASTER_I
 IF (ierr /= 0) THEN; iflag = FILE_ERROR + 2; RETURN; END IF
 CLOSE(CHKRUNIT + chk_sm)
ELSE
 ! Same number of masters were used. Broadcast 'chk_sm' to all
 ! subdomain masters so that each master reads its own file to
 ! recover the logs by the order that they were written.
 CALL MPI_BCAST(chk_sm, 1, MPI_INTEGER, 0, xm_comm, &
             ierr)
 ! Allocate 'chk_len'.
 ALLOCATE(chk_len(1))
 ! Read the logs from its own checkpoint file that is open.
 OUTER: DO
   ! Read the sub-header consisting of iteration number and the number
   ! of logs for that iteration.
   READ(UNIT=CHKSUNIT, IOSTAT=ierr) check_t, chk_len(1)
   IF (ierr < 0) EXIT OUTER
   ! Check if the list has enough space.
   IF (chk_len(1) > (sizeList() - idxList())) THEN
     ! Keep the stored logs in the list and enlarge it.
     CALL enlargeList(MAX(chk_len(1) + idxList(), 2*sizeList()), .TRUE.)
   END IF
   DO i = 1, chk_len(1)
     READ(UNIT=CHKSUNIT, IOSTAT=ierr) X, tmpf
     IF (ierr /= 0) THEN; iflag = FILE_ERROR + 4; RETURN; END IF
     ! Insert the function value to the list.
     CALL insList(X, tmpf)
   END DO
 END DO OUTER
 CLOSE(CHKSUNIT)
 ! Reset the index of function value list to 0 preparing for recovery.
 CALL resetList()
 END IF
ELSE ! Nonroot subdomain masters.
 ! Receive the broadcast message to recover the checkpoint logs.
 chk_sm = 0
 CALL MPI_BCAST(chk_sm, 1, MPI_INTEGER, 0, xm_comm, &
             ierr)
 IF (chk_sm == 0) THEN
   ! Exit because error occurred at the root subdomain master.
   iflag = FILE_ERROR
   RETURN
 END IF
 ! Allocate 'chk_len'.
 ALLOCATE(chk_len(chk_sm))
 IF (chk_sm == N_MASTER_I) THEN ! Recover from its own checkpoint file.
   WRITE(str, 1000) (INT(myid/N_MASTER_I) + 1)/1000.0, mygid/1000.0
   cpfile1 = 'pvtdirchkpt.dat'//str
   OPEN(UNIT=CHKSUNIT, FILE=cpfile1, FORM="UNFORMATTED", STATUS="OLD", &
       POSITION="REWIND", IOSTAT=ierr)
   IF (ierr /= 0) THEN; iflag = FILE_ERROR; RETURN; END IF
   ! Allocate 'chk_l' and 'chk_u' for verifying the file header.
   ALLOCATE(chk_l(N))
   ALLOCATE(chk_u(N))
   CALL verifyHeader(CHKSUNIT, chk_sm1, iflag)
   IF (iflag /= 0) THEN
     CLOSE(CHKSUNIT)
     DEALLOCATE(chk_l)
     DEALLOCATE(chk_u)
     RETURN
   END IF
```

```
  ! Free 'chk_l' and 'chk_u'.
  DEALLOCATE(chk_l)
  DEALLOCATE(chk_u)
  ! Verify if 'chk_sm' is the same as N_MASTER_I.
  IF (chk_sm1 /= N_MASTER_I) THEN
    iflag = FILE_ERROR + 3
    CLOSE(CHKSUNIT)
    RETURN
  END IF
  ! Load the logs from the checkpoint file.
  DO
    READ(UNIT=CHKSUNIT, IOSTAT=ierr) check_t, chk_len(1)
    IF (ierr < 0) EXIT
    ! Check the list size.
    IF (chk_len(1) > (sizelist() - idxList())) THEN
      ! Keep the stored logs in the list and enlarge it.
      CALL enlargeList(MAX(chk_len(1) + idxList(), 2*sizeList()), .TRUE.)
    END IF
    IF (ierr < 0) EXIT
    DO i = 1, chk_len(1)
      READ(UNIT=CHKSUNIT, IOSTAT=ierr) X, tmpf
      IF (ierr /= 0) THEN; iflag = FILE_ERROR + 4; RETURN; END IF
      CALL insList(X, tmpf)
    END DO
  END DO
  CLOSE(CHKSUNIT)
  ! Reset the index of function value list to 0 preparing for recovery.
  CALL resetList()
ELSE ! Read the logs to bypass headers for all checkpoint files, which
  ! are kept open until all logs have been recovered.
  ! Allocate 'chk_l' and 'chk_u' for holding the file header.
  ALLOCATE(chk_l(N))
  ALLOCATE(chk_u(N))
  DO j = 1, chk_sm
    WRITE(str, 2003) (INT(myid/N_MASTER_I) + 1)/1000.0, (j-1)/1000.0
    2003 FORMAT(f4.3, f4.3)
    cpfile1 = 'pvtdirchkpt.dat'//str
    OPEN(UNIT=CHKSUNIT + j, FILE=cpfile1, FORM="UNFORMATTED", &
        STATUS="OLD", POSITION="REWIND", IOSTAT=ierr)
    IF (ierr /= 0) THEN; iflag = FILE_ERROR; EXIT; END IF
    ! Bypass N, L, U, EPS_I, SWITCH_I, N_SUB_I, and N_MASTER_I
    READ(UNIT=CHKSUNIT + j, IOSTAT=ierr) chk_n
    IF (ierr /= 0) THEN; iflag = FILE_ERROR + 1; EXIT; END IF
    READ(UNIT=CHKSUNIT + j, IOSTAT=ierr) chk_l, chk_u, chk_eps, &
                                  chk_switch, chk_sd, chk_sm
    IF (ierr /= 0) THEN; iflag = FILE_ERROR + 1; EXIT; END IF
  END DO
  ! Release the memory for 'chk_l' and 'chk_u'.
  DEALLOCATE(chk_l)
  DEALLOCATE(chk_u)
  IF (iflag /= 0) THEN; CLOSE(CHKSUNIT); RETURN; END IF

  ! Open a new checkpoint file to record the new sequence of
  ! function evaluations with the changed number of masters.
  WRITE(str, 1000) (INT(myid/N_MASTER_I) + 1)/1000.0, mygid/1000.0
  1000 FORMAT(f4.3, f4.3)
  cpfile2 = 'pvtdirchkpt.dat.new'//str
  OPEN(UNIT=CHKRUNIT + chk_sm, FILE=cpfile2, FORM="UNFORMATTED", &
      STATUS="NEW", IOSTAT=ierr)
  IF (ierr /= 0) THEN; iflag = FILE_ERROR; RETURN; END IF
  ! Write the new header.
  WRITE(UNIT=CHKRUNIT + chk_sm, IOSTAT=ierr) N
  IF (ierr /= 0) THEN; iflag = FILE_ERROR + 2; RETURN; END IF
  WRITE(UNIT=CHKRUNIT + chk_sm, IOSTAT=ierr) L, U, EPS_I, SWITCH_I, &
                                      N_SUB_I, N_MASTER_I
  IF (ierr /= 0) THEN; iflag = FILE_ERROR + 2; RETURN; END IF
  CLOSE(CHKRUNIT + chk_sm)
 END IF
 END IF
 END IF
END IF

RETURN
END SUBROUTINE initCheckpoint

SUBROUTINE processStatebufInteger()
IMPLICIT NONE
! Set values of global variables from the integer buffer for state transfer.
!
! On input: None.
!
! On output: None.
!

! Set 'alloc_err', 'b_id', 'BINSIZE_I', 'bits_sub', 'boxset_ind',
! 'box_size', 'bufsize', 'c_alldone', 'c_bits_sub', and 'check_t'.
```

```
alloc_err = statebuf_integer(1)
b_id = statebuf_integer(2)
BINSIZE_I = statebuf_integer(3)
bits_sub = statebuf_integer(4)
boxset_ind = statebuf_integer(5)
box_size = statebuf_integer(6)
bufsize = statebuf_integer(7)
c_alldone = statebuf_integer(8)
c_bits_sub = statebuf_integer(9)
check_t = statebuf_integer(10)

! Set 'chk_n', 'chk_sd', 'chk_sm', 'chk_sm1', 'chk_switch',
! 'col', 'eval_c', 'eval_c_i', 'gbuffer_len', and 'gc_convex'.
chk_n = statebuf_integer(11)
chk_sd = statebuf_integer(12)
chk_sm = statebuf_integer(13)
chk_sm1 = statebuf_integer(14)
chk_switch = statebuf_integer(15)
col = statebuf_integer(16)
eval_c = statebuf_integer(17)
eval_c_i = statebuf_integer(18)
gbuffer_len = statebuf_integer(19)
gc_convex = statebuf_integer(20)

! Set 'group_world', 'iflag', 'i_start', 'lbc', 'lbuffer_len',
! 'loc_pid', 'loc_sid', 'loc_eid', 'loc_val', and 'loc_c'.
group_world = statebuf_integer(21)
iflag = statebuf_integer(22)
i_start = statebuf_integer(23)
lbc = statebuf_integer(24)
lbuffer_len = statebuf_integer(25)
loc_pid = statebuf_integer(26)
loc_sid = statebuf_integer(27)
loc_eid = statebuf_integer(28)
loc_val = statebuf_integer(29)
loc_c = statebuf_integer(30)

! Set 'loc_side', 'loc_diam', 'N_MASTER_I', 'N_SUB_I', 'spawn_count',
! 'q_counter', 'recv_tag', 'RESTART_I', 'set_size', and 'size_r8'.
loc_side = statebuf_integer(31)
loc_diam = statebuf_integer(32)
N_MASTER_I = statebuf_integer(33)
N_SUB_I = statebuf_integer(34)
spawn_count = statebuf_integer(35)
q_counter = statebuf_integer(36)
recv_tag = statebuf_integer(37)
RESTART_I = statebuf_integer(38)
set_size = statebuf_integer(39)
size_r8 = statebuf_integer(40)

! Set 'stop_rule', 'SWITCH_I', 't', 'update_counter', 'world_size',
! 'N', 'PROCID', 'col_w', 'row_w', and 'N_I'.
stop_rule = statebuf_integer(41)
SWITCH_I = statebuf_integer(42)
t = statebuf_integer(43)
update_counter = statebuf_integer(44)
world_size = statebuf_integer(45)
col_w = statebuf_integer(46)
row_w = statebuf_integer(47)
N_I = statebuf_integer(48)
switch_iter = statebuf_integer(49)

RETURN
END SUBROUTINE processStatebufInteger

SUBROUTINE processStatebufReal
IMPLICIT NONE
! Set values of global variables from the real buffer for state transfer.
!
! On input: None.
!
! On output: None.
!

! Set 'chk_eps', 'dia', 'dia_i', 'dia_limit', and 'EPS_I'.
chk_eps = statebuf_real(1)
dia = statebuf_real(2)
dia_i = statebuf_real(3)
dia_limit = statebuf_real(4)
EPS_I = statebuf_real(5)

! Set 'fmin_old', 'FMIN_I', 'MIN_SEP_I', 'tmpf', and 'FMIN'.
fmin_old = statebuf_real(6)
FMIN_I = statebuf_real(7)
MIN_SEP_I = statebuf_real(8)
tmpf = statebuf_real(9)
```

```
FMIN = statebuf_real(10)

! Set 'EPS4N'.
EPS4N = statebuf_real(11)

RETURN
END SUBROUTINE processStatebufReal

SUBROUTINE processStatebufLogical
IMPLICIT NONE
! Set values of global variables from the logical buffer for state transfer.
!
! On input: None.
!
! On output: None.
!

! do_it
do_it = statebuf_logical(1)

RETURN
END SUBROUTINE processStatebufLogical

SUBROUTINE initStateTransfer()
IMPLICIT NONE
! Initialize state transfer. First, send the number of masters per subdomain
! and total number of processes to the spawned masters. This allows for the
! parent and child intercommunicators to be merged before the state transfer
! subroutine, simplifying state transfer. Next, send the sizes of all arrays
! defined in the 'spVTdirect' subroutine.
!
! On input: None.
!
! On output: None.
!
! Local variables.
INTEGER :: comm ! Communicator handle for state transfer.
INTEGER :: myarray_masters = 1 ! Indices for 'mySIZE' array.
INTEGER :: mybuffer = 2
INTEGER :: myb_worker = 3
INTEGER :: mychk_l = 4
INTEGER :: mychk_len = 5
INTEGER :: mychk_u = 6
INTEGER :: mydispls = 7
INTEGER :: mygbuffer = 8
INTEGER :: mylbuffer = 9
INTEGER :: mylc_convex = 10
INTEGER :: mym_comm =11
INTEGER :: mym_group = 12
INTEGER :: myq_worker = 13
INTEGER :: myupdate_array = 14
INTEGER, DIMENSION(2) :: proc_info ! Number of masters for this subdomain, as
  ! well as the total number of processes.
INTEGER :: root ! Root value for broadcasting state.
INTEGER :: tag ! Tag for state transfer
INTEGER :: tmp_size ! Size of array to be transferred.

! Set 'tag'.
tag = STATE_TRANSFER_TAG

! Initial masters send information to spawned masters.
IF (.NOT. spawned) THEN
  ! Set values for the communicator and the root process for broadcasts.
  comm = child_comm
  root = MPI_ROOT
  ! Send the number of masters and the total number of processes to the spawned
  ! masters.
  proc_info = (/ N_MASTER_I, world_size /)
  CALL MPI_SEND(proc_info, 2, MPI_INTEGER, 0, STATE_TRANSFER_TAG, comm, &
                ierr)
  ! Send the sizes of arrays defined in the 'spVTdirect' subroutine.
  IF (ALLOCATED(array_masters)) THEN
    tmp_size = SIZE(array_masters)
  ELSE
    tmp_size = -1
  END IF
  mySIZE(myarray_masters) = tmp_size
  IF (ALLOCATED(b_worker)) THEN
    tmp_size = SIZE(b_worker)
  ELSE
    tmp_size = -1
  END IF
  mySIZE(myb_worker) = tmp_size
  IF (ALLOCATED(chk_len)) THEN
    tmp_size = SIZE(chk_len)
  ELSE
```

```
  tmp_size = -1
END IF
mySIZE(mychk_len) = tmp_size
IF (ALLOCATED(displs)) THEN
  tmp_size = SIZE(displs)
ELSE
  tmp_size = -1
END IF
mySIZE(mydispls) = tmp_size
IF (ALLOCATED(lc_convex)) THEN
  tmp_size = SIZE(lc_convex)
ELSE
  tmp_size = -1
END IF
mySIZE(mylc_convex) = tmp_size
IF (ALLOCATED(m_comm)) THEN
  tmp_size = SIZE(m_comm)
ELSE
  tmp_size = -1
END IF
mySIZE(mym_comm) = tmp_size
IF (ALLOCATED(m_group)) THEN
  tmp_size = SIZE(m_group)
ELSE
  tmp_size = -1
END IF
mySIZE(mym_group) = tmp_size
IF (ALLOCATED(update_array)) THEN
  tmp_size = SIZE(update_array)
ELSE
  tmp_size = -1
END IF
mySIZE(myupdate_array) = tmp_size
IF (ALLOCATED(q_worker)) THEN
  tmp_size = SIZE(q_worker)
ELSE
  tmp_size = -1
END IF
mySIZE(myq_worker) = tmp_size
IF (ALLOCATED(chk_l)) THEN
  tmp_size = SIZE(chk_l)
ELSE
  tmp_size = -1
END IF
mySIZE(mychk_l) = tmp_size
IF (ALLOCATED(chk_u)) THEN
  tmp_size = SIZE(chk_u)
ELSE
  tmp_size = -1
END IF
mySIZE(mychk_u) = tmp_size
IF (ALLOCATED(buffer)) THEN
  tmp_size = SIZE(buffer)
ELSE
  tmp_size = -1
END IF
mySIZE(mybuffer) = tmp_size
IF (ALLOCATED(gbuffer)) THEN
  tmp_size = SIZE(gbuffer)
ELSE
  tmp_size = -1
END IF
mySIZE(mygbuffer) = tmp_size
IF (ALLOCATED(lbuffer)) THEN
  tmp_size = SIZE(lbuffer)
ELSE
  tmp_size = -1
END IF
mySIZE(mylbuffer) = tmp_size
CALL MPI_SEND(mySIZE, SIZE(mySIZE), MPI_INTEGER, 0, tag, comm, ierr)
! Spawned masters receive information from the initial masters.
ELSE
  ! Set values for the communicator and the root process for broadcasts.
  comm = parent_comm
  root = 0
  ! Receive the number of masters and the total number of processes, and
  ! broadcast this information to all other spawned masters.
  IF (myid == 0) THEN
    CALL MPI_RECV(proc_info, 2, MPI_INTEGER, 0, tag, comm, &
                  MPI_STATUS_IGNORE, ierr)
  END IF
  CALL MPI_BCAST(proc_info, 2, MPI_INTEGER, root, MPI_COMM_WORLD, ierr)
  N_MASTER_I = proc_info(1)
  world_size = proc_info(2)
  ! Receive the sizes of arrays defined in the 'spVTdirect' subroutine, and
  ! broadcast this information to all other spawned masters.
```

```
IF (myid == 0) THEN
  CALL MPI_RECV(mySIZE, SIZE(mySIZE), MPI_INTEGER, 0, tag, comm, &
              MPI_STATUS_IGNORE, ierr)
END IF
CALL MPI_BCAST(mySIZE, SIZE(mySIZE), MPI_INTEGER, root, MPI_COMM_WORLD, ierr)
tmp_size = mySIZE(myarray_masters)
IF (tmp_size /= -1) THEN
  IF (ALLOCATED(array_masters)) DEALLOCATE(array_masters)
  ALLOCATE(array_masters(tmp_size))
END IF
tmp_size = mySIZE(myb_worker)
IF (tmp_size /= -1) THEN
  IF (ALLOCATED(b_worker)) DEALLOCATE(b_worker)
  ALLOCATE(b_worker(tmp_size))
END IF
tmp_size = mySIZE(mychk_len)
IF (tmp_size /= -1) THEN
  IF (ALLOCATED(chk_len)) DEALLOCATE(chk_len)
  ALLOCATE(chk_len(tmp_size))
END IF
tmp_size = mySIZE(mydispls)
IF (tmp_size /= -1) THEN
  IF (ALLOCATED(displs)) DEALLOCATE(displs)
  ALLOCATE(displs(tmp_size))
END IF
tmp_size = mySIZE(mylc_convex)
IF (tmp_size /= -1) THEN
  IF (ALLOCATED(lc_convex)) DEALLOCATE(lc_convex)
  ALLOCATE(lc_convex(tmp_size))
END IF
tmp_size = mySIZE(mym_comm)
IF (tmp_size /= -1) THEN
  IF (ALLOCATED(m_comm)) DEALLOCATE(m_comm)
  ALLOCATE(m_comm(tmp_size))
END IF
tmp_size = mySIZE(mym_group)
IF (tmp_size /= -1) THEN
  IF (ALLOCATED(m_group)) DEALLOCATE(m_group)
  ALLOCATE(m_group(tmp_size))
END IF
tmp_size = mySIZE(myupdate_array)
IF (tmp_size /= -1) THEN
  IF (ALLOCATED(update_array)) DEALLOCATE(update_array)
  ALLOCATE(update_array(tmp_size))
 END IF
 tmp_size = mySIZE(myq_worker)
 IF (tmp_size /= -1) THEN
  IF (ALLOCATED(q_worker)) DEALLOCATE(q_worker)
  ALLOCATE(q_worker(tmp_size))
 END IF
 tmp_size = mySIZE(mychk_l)
 IF (tmp_size /= -1) THEN
  IF (ALLOCATED(chk_l)) DEALLOCATE(chk_l)
  ALLOCATE(chk_l(tmp_size))
 END IF
 tmp_size = mySIZE(mychk_u)
 IF (tmp_size /= -1) THEN
  IF (ALLOCATED(chk_u)) DEALLOCATE(chk_u)
  ALLOCATE(chk_u(tmp_size))
 END IF
 tmp_size = mySIZE(mybuffer)
 IF (tmp_size /= -1) THEN
  IF (ALLOCATED(buffer)) DEALLOCATE(buffer)
  ALLOCATE(buffer(tmp_size))
 END IF
 tmp_size = mySIZE(mygbuffer)
 IF (tmp_size /= -1) THEN
  IF (ALLOCATED(gbuffer)) DEALLOCATE(gbuffer)
  ALLOCATE(gbuffer(tmp_size))
 END IF
 tmp_size = mySIZE(mylbuffer)
 IF (tmp_size /= -1) THEN
  IF (ALLOCATED(lbuffer)) DEALLOCATE(lbuffer)
  ALLOCATE(lbuffer(tmp_size))
 END IF
END IF

RETURN
END SUBROUTINE initStateTransfer

SUBROUTINE checkProcList(not_in_list, length, host_name, list)
IMPLICIT NONE
! Check if a host name is already contained in a list of host names.
!
! On input:
! length    - The length of the list to be checked.
```

```
! host_name - The name of the host whose membership is being checked.
! list      - An array of host names possibly containing 'host_name'.
!
! On output:
! not_in_list - A boolean that specifies the membership of 'host_name' in
!               the array 'list'.
!
LOGICAL, INTENT(OUT) :: not_in_list
INTEGER, INTENT(IN) :: length
CHARACTER(LEN = 32), INTENT(IN) :: host_name
CHARACTER(LEN = 32), DIMENSION(length), INTENT(IN) :: list

! Local variables.
INTEGER :: i

! Set 'not_in_list' to '.TRUE.' if 'host_name' is in not in 'list', or
! '.FALSE.' otherwise.
not_in_list = .TRUE.
DO i = 1, length
  IF (list(i) == host_name) THEN
    not_in_list = .FALSE.
    EXIT
  END IF
END DO

RETURN
END SUBROUTINE checkProcList

SUBROUTINE initProcList(comm)
IMPLICIT NONE
! Initialize the list of host names that will be used to create a host file.
!
! On input:
! comm - The handle for the communicator whose member processes will contribute
!        their host names to the list of host names.
!
! On output: None.
!
INTEGER, INTENT(IN) :: comm

! Local variables.
CHARACTER(LEN = 32), ALLOCATABLE, DIMENSION(:) :: tmp_proc_list ! Initial list
  ! of host names used to produce final list.
LOGICAL, ALLOCATABLE, DIMENSION(:) :: tmp_mark_list ! List of boolean values
  ! used to mark which hosts have already been added to the final host list.
INTEGER :: comm_size ! The number of member processes in 'comm'.
INTEGER, ALLOCATABLE, DIMENSION(:) :: hname_length ! The length of a host name.
INTEGER, ALLOCATABLE, DIMENSION(:) :: tmp_num_cores ! Initial list of core
  ! counts for each processor used to produce the final list of core counts.
INTEGER :: i, j, k ! Loop counters.
INTEGER :: pid ! The rank of a process in 'comm'.

! Determine the ranks of all processes and the size of 'comm'.
CALL MPI_COMM_RANK(comm, pid, qspawn_err)
CALL MPI_COMM_SIZE(comm, comm_size, qspawn_err)
ALLOCATE(tmp_proc_list(comm_size))
ALLOCATE(tmp_num_cores(comm_size))
ALLOCATE(tmp_mark_list(comm_size))
ALLOCATE(hname_length(comm_size))

! Initialize the initial list of host names, and mark all host names as not in
! the list.
DO i = 1, comm_size
  tmp_proc_list(i) = ""
  tmp_mark_list(i) = .FALSE.
END DO

! Get processor names for all participating processes and broadcast them to
! all processes in 'comm'.
CALL MPI_GET_PROCESSOR_NAME(tmp_proc_list(1 + pid), hname_length(1 + pid), &
                            qspawn_err)
DO i = 1, comm_size
  CALL MPI_BCAST(hname_length(i), 1, MPI_INTEGER, &
                 i - 1, comm, qspawn_err)
END DO
DO i = 1, comm_size
  CALL MPI_BCAST(tmp_proc_list(i), hname_length(i), MPI_CHARACTER, &
                 i - 1, comm, qspawn_err)
END DO
! Find complete set of unique host names.
tmp_mark_list(1) = .TRUE.
qspawn_num_procs = 1
IF (pid == 0) THEN
  tmp_num_cores(1) = OMP_GET_MAX_THREADS()
END IF
CALL MPI_BCAST(tmp_num_cores(1), 1, MPI_INTEGER, &
```

```
            0, comm, qspawn_err)
DO i = 2, comm_size
  ! Check if the host name has already been added to the list.
  CALL checkProcList(tmp_mark_list(i), i - 1, tmp_proc_list(i), &
                     tmp_proc_list)
  IF (tmp_mark_list(i)) THEN
    ! Update the number of unique host names.
    qspawn_num_procs = qspawn_num_procs + 1
    ! Determine the number of cores for the node running this process.
    IF (pid == i - 1) THEN
      tmp_num_cores(i) = OMP_GET_MAX_THREADS()
    END IF
    CALL MPI_BCAST(tmp_num_cores(i), 1, MPI_INTEGER, &
                   i - 1, comm, qspawn_err)
  END IF
END DO
! Create final lists of host names and core counts, and determine the number of
! cores already hosting a  process for each node.
IF (ALLOCATED(qspawn_proc_list)) DEALLOCATE(qspawn_proc_list)
ALLOCATE(qspawn_proc_list(qspawn_num_procs))
IF (ALLOCATED(qspawn_num_cores)) DEALLOCATE(qspawn_num_cores)
ALLOCATE(qspawn_num_cores(qspawn_num_procs))
IF (ALLOCATED(qspawn_num_hosted)) DEALLOCATE(qspawn_num_hosted)
ALLOCATE(qspawn_num_hosted(qspawn_num_procs))
j = 1
DO i = 1, comm_size
  IF (tmp_mark_list(i)) THEN
    qspawn_proc_list(j) = tmp_proc_list(i)
    qspawn_num_cores(j) = tmp_num_cores(i)
    ! Determine the number of processes hosted on this node.
    qspawn_num_hosted(j) = 1
    DO k = i + 1, comm_size
      IF (tmp_proc_list(i) == tmp_proc_list(k)) THEN
        qspawn_num_hosted(j) = qspawn_num_hosted(j) + 1
      END IF
    END DO
    j = j + 1
  ELSE
    qspawn_num_hosted(k) = qspawn_num_hosted(k) + 1
  END IF
END DO
! Deallocate local arrays that are no longer needed.
DEALLOCATE(tmp_proc_list)
DEALLOCATE(tmp_num_cores)
DEALLOCATE(tmp_mark_list)
DEALLOCATE(hname_length)

RETURN
END SUBROUTINE initProcList

SUBROUTINE makeHostFile(num_hosts, hostfile, hfname_length, counter)
IMPLICIT NONE
! Make a host file from the hosts in 'qspawn_proc_list'. Note that this must
! only be called after 'initProcList' has been used to create the host list.
!
! On input:
! num_hosts    - The desired number of hosts in the host file.
! hostfile     - The name of the host file.
! hfname_length - The length of the host file name.
!
! On output:
! counter - The actual number of hosts specified in the host file.
!
INTEGER, INTENT(IN) :: num_hosts
CHARACTER(LEN = hfname_length), INTENT(IN) :: hostfile
INTEGER, INTENT(IN) ::hfname_length
INTEGER, INTENT(OUT) :: counter

! Local variables.
INTEGER :: i ! Loop counter.
INTEGER :: marker ! A marker used to determine if all cores have been assigned.
LOGICAL :: flag ! Flag used to specify if the marker has been set.

! Write host names to host file in a round-robin fashion.
OPEN(UNIT=QSPAWN_UNIT, FILE=hostfile)
flag = .TRUE.
marker = 0
i = 1
counter = 0
DO WHILE (.TRUE.)
  ! Exit loop if we have returned to the marker, or if 'num_hosts' hosts have
  ! been assigned.
  IF (i == marker .OR. counter == num_hosts) EXIT
  IF (qspawn_num_hosted(i) < qspawn_num_cores(i)) THEN
    WRITE(QSPAWN_UNIT, *) qspawn_proc_list(i)
    qspawn_num_hosted(i) = qspawn_num_hosted(i) + 1
```

```
  counter = counter + 1
  flag = .TRUE.
! If the marker has not been set, then mark 'i' as the first node with all
! cores assigned.
ELSE IF (flag) THEN
  marker = i
  flag = .FALSE.
END IF
i = i + 1
IF (i > qspawn_num_procs) THEN
   i = 1
  END IF
END DO
CLOSE(UNIT=QSPAWN_UNIT)

RETURN
END SUBROUTINE makeHostFile
END SUBROUTINE spVTdirect
END MODULE spVTdirect_MOD
```