Adjusting process count on demand for petascale global optimization⋆

Nicholas R. Radcliffe[a,*], Layne T. Watson[a,b], Masha Sosonkina[c], Rafael T. Haftka[d], Michael W. Trosset[e]

[a] *Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA*
[b] *Department of Mathematics, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA*
[c] *Departments of Computer Science and Electrical and Computer Engineering, Iowa State University, Ames, Iowa, USA*
[d] *Department of Mechanical and Aerospace Engineering, University of Florida, Gaineville, FL, USA*
[e] *Department of Statistics, Indiana University, Bloomington, IN, USA*

ARTICLE INFO

Article history:
This is the history of the article...

Keywords:
petascale
Message passing interface (MPI)
dynamic process count
global optimization

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 paper 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.

1. Introduction and motivation

The ultimate goal of the work presented in this paper 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 paper 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.

The rest of this paper is organized as follows. Section 2 presents a description of the DIRECT algorithm, which forms the basis of the global optimization code `pVTdirect`. Some details of the code `pVTdirect` are also presented. A description of the problem and the main challenges involved are presented in Section 3. Section 4 describes some stress tests performed to evaluate two possible design choices. A dynamic load balancing mechanism is described in Section 5. Performance results are given in Section 6. Section 7 discusses related work. Section 8 finishes with conclusions and lessons-learned.

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

*Corresponding author.
E-mail addresses: nradclif@vt.edu (N.R. Radcliffe), ltw@cs.vt.edu (L.T. Watson), masha@scl.ameslab.gov (M. Sosonkina), haftka@ufl.edu (R.T. Haftka), mtrosset@indiana.edu (M.W. Trosset)

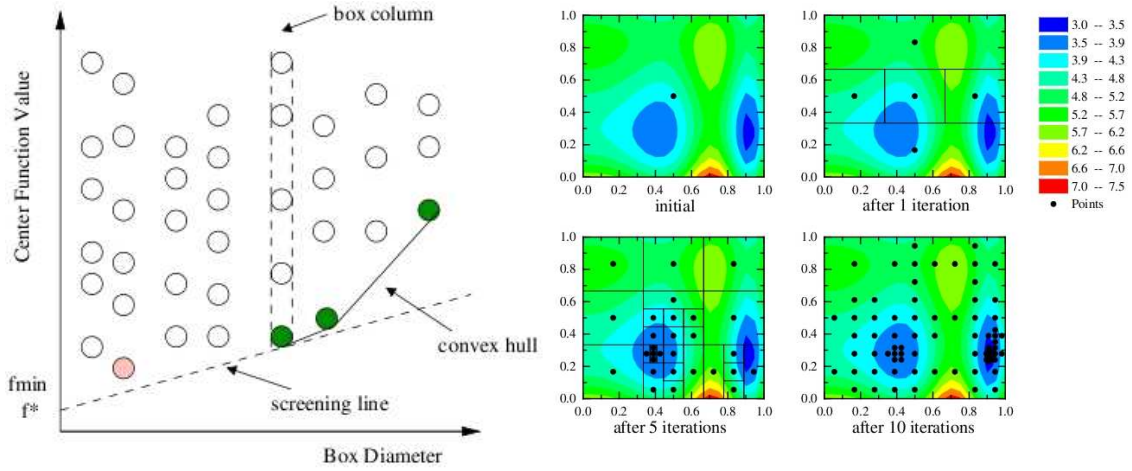

Fig. 1. Illustrations of DIRECT's box columns (left), and VTdirect in action (right).

2. 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. 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.

2.1. Important details of the implementation

pVTdirect, the parallel version of VTdirect and the only version under consideration in this paper, 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 paper.

3. 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.

3.1. 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.

3.2. 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 Section 6 for performance results).

3.2.1. 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 Section 3.4.

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`.

3.3. 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 *i*th communicator is

$$rank_{global} = N_1 + N_2 + ... + N_{i-1} + rank_{local},$$

where N_j is the size of the *j*th communicator for $j = 1, \ldots, i-1$, and $rank_{local}$ is the usual rank of the process within the *i*th 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 "many communicator" and "merge" methods. The results of these tests are presented in Section 4.

3.4. Dealing with inconsistent states

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 *i*th element of an array is the rank of the *i*th 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.

3.4.1. 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 (Section 5).

4. Comparison between 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 dynamic communication scheme that can adjust to increases in the number of masters.

4.1. Experimental setup

For the "many communicator" method, the experimental setup for the broadcast and gather tests was the same, but differed from the setup for the send and receive tests. 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. The timing data was averaged over a large number of tests for each message length because unusually noisy data was obtained in previous tests.

4.1.1. 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.

4.1.2. 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.

4.1.3. 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.

4.1.4. 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.

4.2. 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 and below the mean, respectively. The three curves for each method form a band that likely contains the "true" runtime curve for the method.

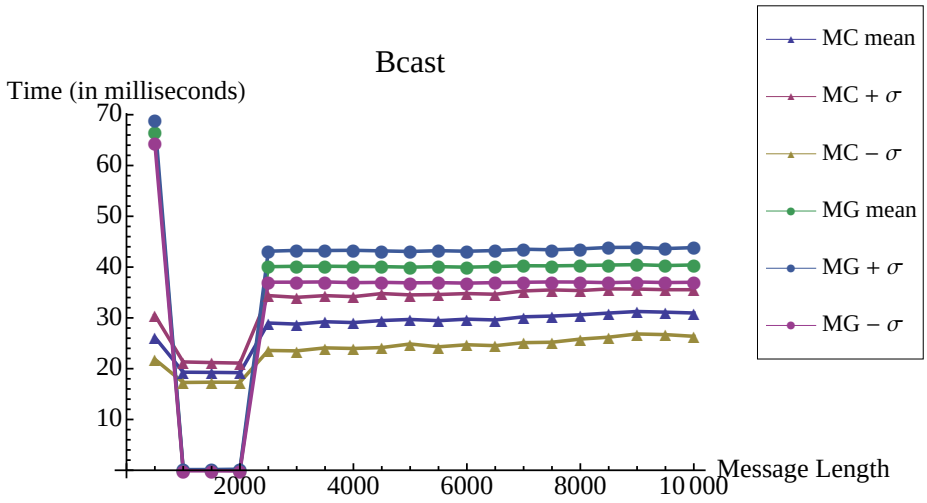

Fig. 2. "Many communicator" (triangles) and "merge" method (circles) broadcasts on System G.

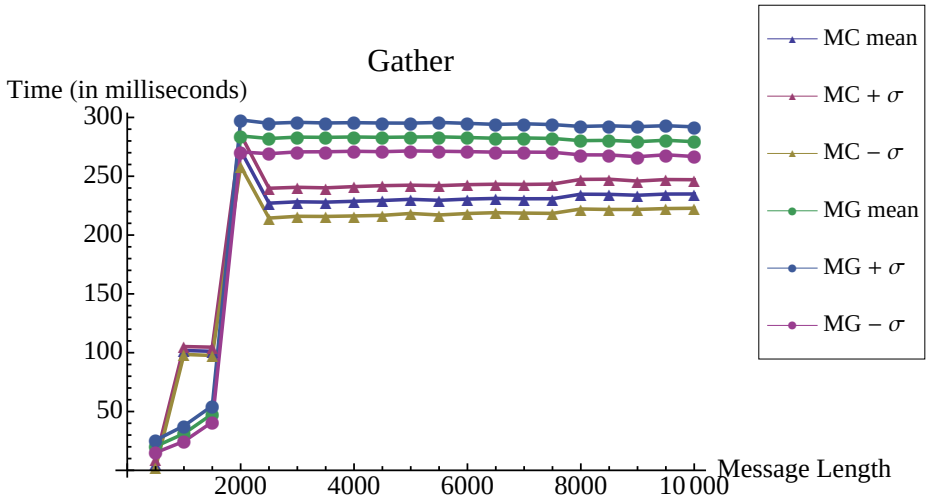

Fig. 3. "Many communicator" (triangles) and "merge" method (circles) gathers on System G.

4.2.1. Broadcast and gather

For the broadcast and gather tests (Figures 2 and 3), with a few exceptions, the "many communicator" method generally performed better. For message lengths of 1000 to 2000 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.

4.2.2. 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 4). 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

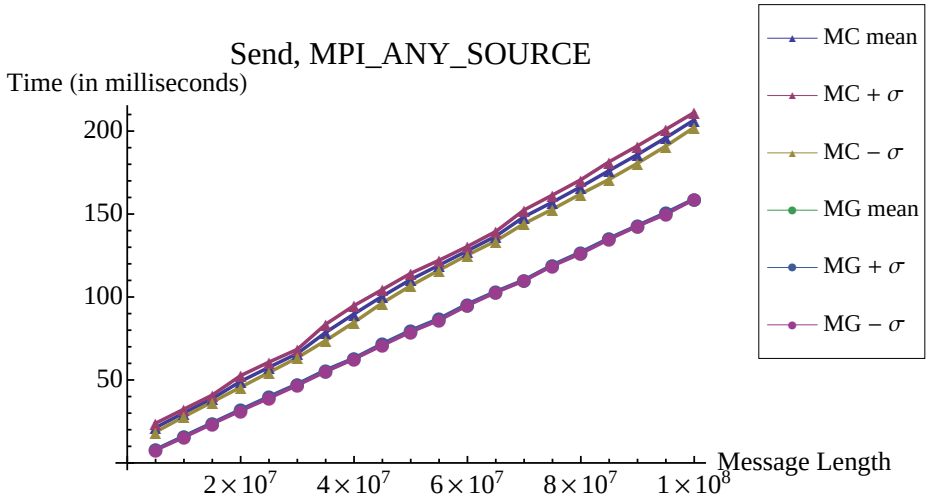

Fig. 4. "Many communicator" (triangles) and "merge" method (circles) gathers on System G.

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 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 child programs spawned for the global optimization application is generally small.

5. 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 $1 - 1/2^s$, where s is the current spawn count, 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 Section 6.3, it is shown that this method is effective in dynamically balancing box load after new masters are spawned.

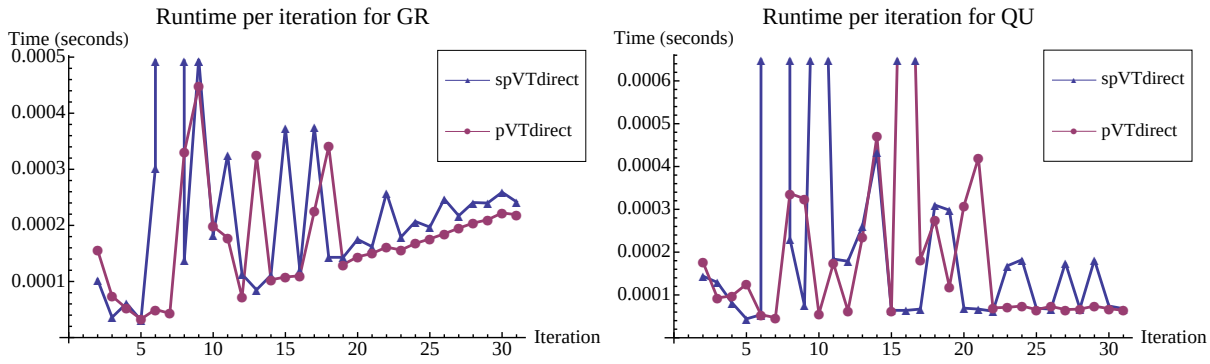

Fig. 5. Comparison of runtimes per iteration for the GR (left) and QU (right) objective functions.

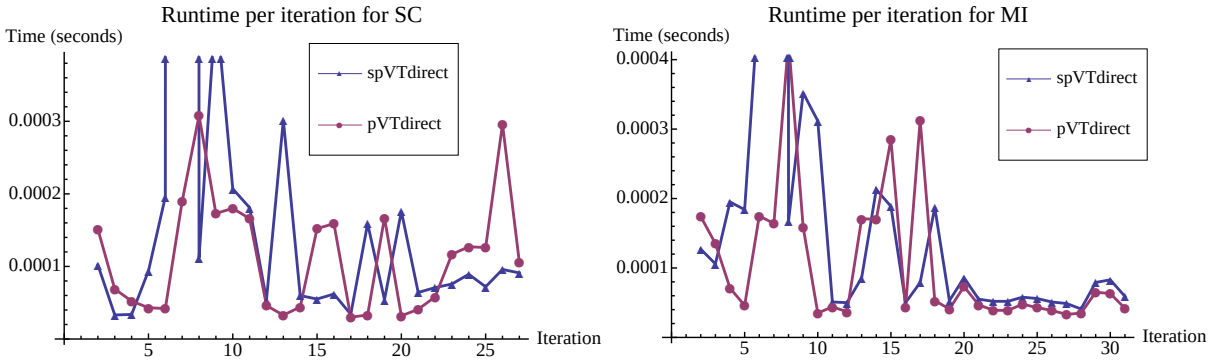

Fig. 6. Comparison of runtimes per iteration for the SC (left) and MI (right) objective functions.

6. Performance results

A variety of optimization problems were used to test the modifications made to `pVTdirect`. Several toy problems, as well as a realistic nulcear 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 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 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.

6.1. 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 5 and 6 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 6, right) for a nice illustration of this effect.

The toy problems with negligible runtimes were useful 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

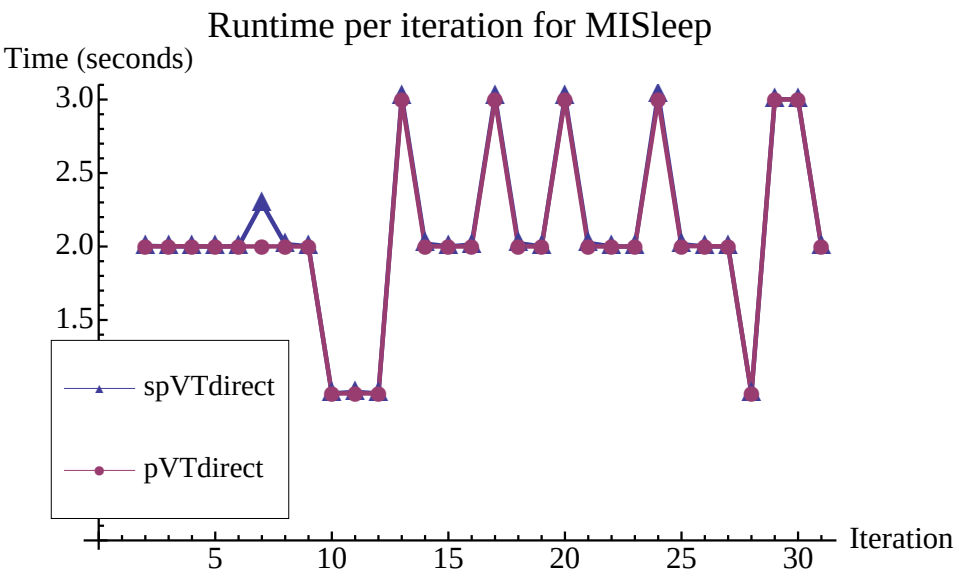

Fig. 7. Comparison of runtimes per iteration for the MISleep objective function.

`pVTdirect`. Although there are very minor differences in performance, the boxes examined at every iteration are identical for `pVTdirect` and `spVTdirect`.

In Figure 7, 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 7, 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 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).

6.2. 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 Schrodinger equation with a large, sparse, and irregularly structured Hamiltonian matrix [10]. One reason for this 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 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 it must be run 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

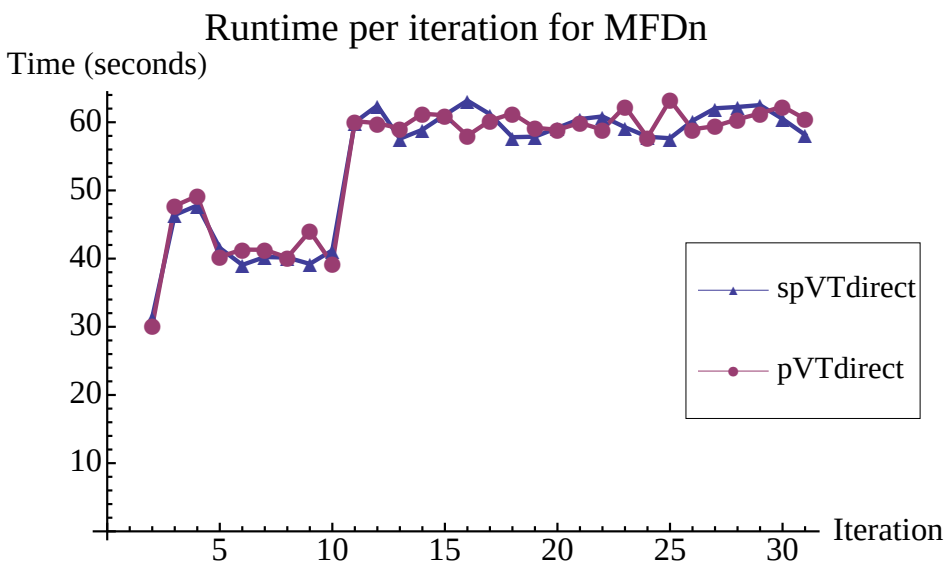

Fig. 8. Comparison of runtimes per iteration for the MFDn objective function.

"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 8 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.

6.3. 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 9).

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 10). 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 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 11), 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).

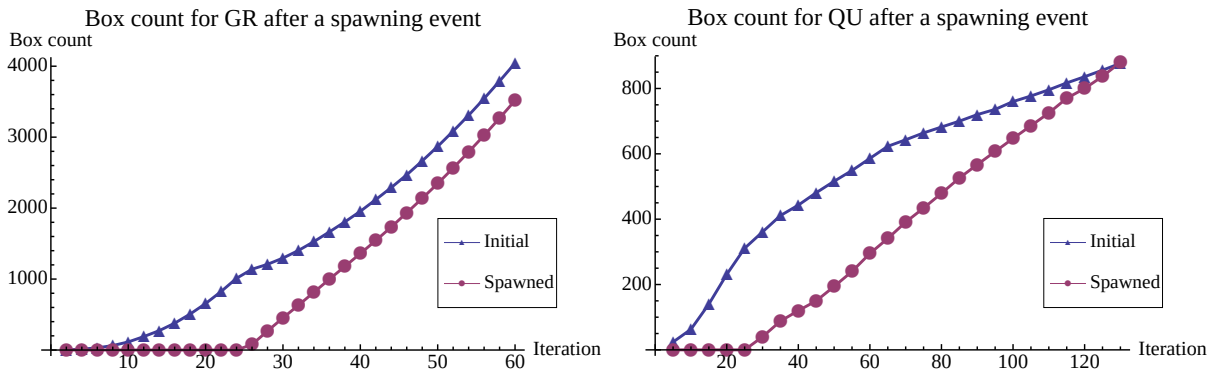

Fig. 9. Box count versus iteration for GR (left) and QU (right).

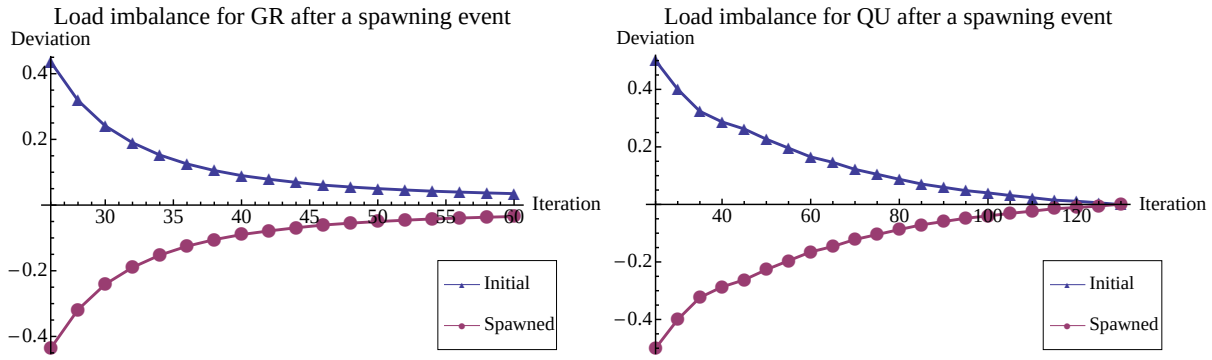

Fig. 10. Load deviation versus iteration for GR (left) and QU (right).

7. Related work

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 Section 3.2.1). 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.

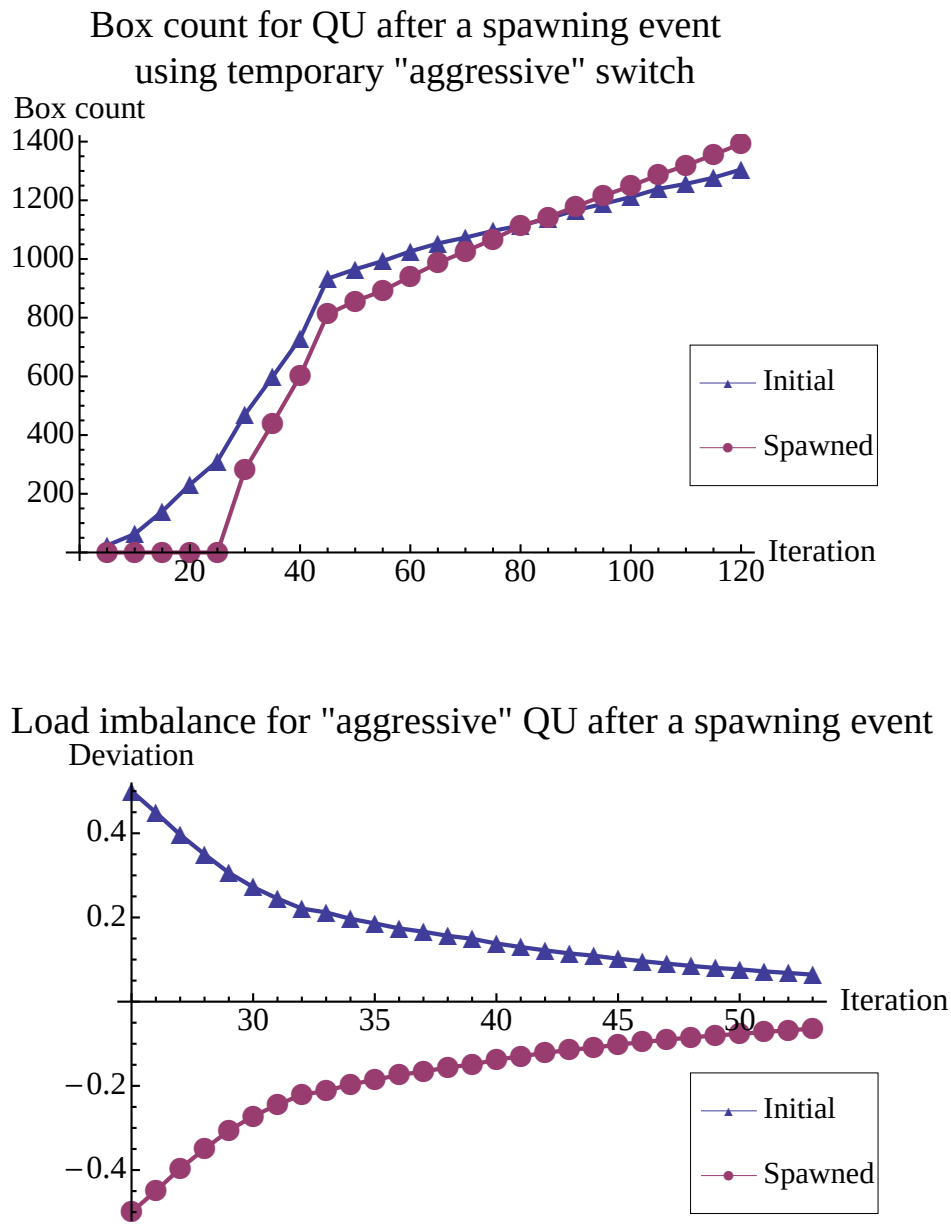

Fig. 11. Box count (top) and deviation (bottom) versus iteration for QU using temporary "aggressive" switch.

8. Conclusions and discussion

This work shows that it is possible to dynamically adjust the number of masters in the global optimization code `pVTdirect`, and hence prevent 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 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, but it might create 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 much 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.

Acknowledgements

The authors 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.

References

[1] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, J. Larsson, 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, M. Ropo, J. Westerholm, J. Dongarra (Eds.), Springer-Verlag, Berlin, Heidelberg, (2009), 20–30.

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

[3] J. He, L. T. Watson, M. Sosonkina, Algorithm 897: VTDIRECT95: serial and parallel codes for the global optimization algorithm DIRECT, ACM Transactions on Mathematical Software 26 (2009) 1–24.

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

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

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

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

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

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

[11] A. Scherer, H. Lu, T. Gross, W. Zwaenepoel, 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, New York, NY, USA, (1999), 96–106.

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

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

306–322 [14] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, MPI—the Complete Reference, Vol. 1: the MPI Core (2nd. ed.), MIT Press, Cambridge, MA, USA, 2000.

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