Dynamic Load-Balancing for Data-Parallel MPI Programs

William George
National Institute of Standards and Technology

Abstract— **This paper describes the load-balancing support available in `DParLib`, a library of MPI-based routines that support the data-parallel style of programming in MPI. The basic structure of `DParLib` is described with a focus on the parts of the library needed to support the distribution and re-distribution of arrays.**

Keywords— **MPI, data-parallel, load balancing, library**

I. Introduction

THIS paper describes the load-balancing support available in `DParLib`, a library of MPI-based routines that support the data-parallel style of programming in MPI. Load-balancing in this paper refers to the effort to keep all processors equally busy doing productive work.

The problem of distributing the computational load among the processors of a parallel machine has been with us as long as distributed computing itself. The problem has been studied both in its general sense and in relation to specific algorithms [1] [2] [3] [4] [5] [6] [7]. This paper is less about the specific algorithms used to compute the load balance than it is about the supporting software. Many of the decisions about load balancing are left to the individual application, such as how often to balance and how to determine the optimal redistribution of data and obtain a balanced load. Given the library support for the general redistribution of arrays, as well as some other routines designed to support the computation of a load balancing measure, the majority of the work has been done toward providing each application with dynamic load balancing capability. Application writers can focus on tuning their load balancing algorithm, be it simple or complex, for their particular situation.

The parallel programs we are interested in are those programs that fit into the data-parallel programming model. For our purposes, we define a data-parallel program as one in which the majority of the computation takes place on large arrays of data which can be conveniently distributed among the available processors. Ideally, the computations performed on each element of the array can be accomplished in parallel with the computations on all of the other elements, although this is not strictly necessary.

Load balancing a data-parallel program consists mainly of ensuring that the arrays are distributed among the processors such that each processor has an equal computational load. This does not mean that each processor has the same number of computations to perform, but that each processor can complete its computations in the same amount of time as the other processors. As a simple example, if a program is running on two processors, with the main data array distributed between them, and one processors is twice as fast as the other at performing each array element update, then the fast processor should get twice as many elements as the slower processor. In this way, the overall computation will complete in the shortest amount of time. If the correct distribution cannot be precomputed, or if the correct distribution changes as the program progresses, then the array must be redistributed periodically to avoid suffering from a large load imbalance.

The redistribution of arrays requires knowledge of the current distribution, the desired distribution, and the exact size and shape of the array to be redistributed. This information, along with many routines for creating, deleting, and manipulating distributed arrays, can be managed by `DParLib`.

This paper is structured as follows. Section I describes the basics of array distribution in `DParLib`. Section II describes the specific routines used to support data redistribution. Section III outlines the structure of a program that uses the data redistribution support in `DParLib`. Section IV gives some initial results from an actual application which uses this technique for load-balancing. Finally, Section V gives some conclusions on this effort and describes the future direction of development for `DParLib`.

II. Data-parallel Support

A. Distribution of Arrays

`DParLib` supports the distribution of arrays in the two most common schemes, *block* and *block-cyclic*, as well as *general-block*, a `DParLib` specific distribution scheme designed to support load balancing. Programs that use the load balancing routines in `DParLib` must have at least one axis that is distributed in the general-block scheme.

The library supports these distributions in that, given the extent of the array axis to be distributed, and the number of nodes to distribute the axis between, the library can supply to the user the size of the axis on each of the nodes as well as the global indices owned by each node. The padding of an array axis that cannot be distributed evenly is handled by the library so the user does not need to artificially pad arrays to even sizes. Additionally, array shifting, array reduction, elemental operations, stencil computation, and other data-parallel operations are supported on arrays with these distributions. In general, for each array which the user will use with any of the `DParLib` routines, the user describes the size and shape of the array to be distributed,

The author may be contacted at NIST, 100 Bureau Drive Stop 8951, Gaithersburg MD 20899-8951. E-mail: william.george@nist.gov

identifies how each axis is to be distributed (*local* or not distributed, block distribution, block-cyclic, or general-block distribution), and the library computes the size and shape of each *subarray*, that is, the part of the array that exists on each node. Each array axis is distributed independently so that any number of axes may be local, or distributed as block-cyclic, block, or general-block. From the description of the array distribution given to DParLib, an array *descriptor* is produced by the library to be used (passed as an argument) whenever you need information about the distribution of this array or an operation supported by the library needs to be performed. The library can also allocate and free these arrays if needed.

An example of a call to generate an array descriptor for an array of floats of size 200×100 looks like this:

```
extent[0] = 200; extent[1] = 100;
dist.dist_type[0] = DP_general_block_dist;
dist.dist_type[1] = DP_block_dist;
A = dp_make_array_desc(comm, MPI_FLOAT, 2,
    extents, false, dist);
axis=0;
dp_compute_general_block_dist_x(A, axis, bsizes)
dp_init_general_dist(A, axis, bsizes);
```

where comm is an MPI communicator with a Cartesian topology (which in this case it must have 2 axes), extent gives the size of the array axes, dist is a DParLib defined structure used to describe the distribution of each of the array axes, and false (a DParLib defined boolean value) indicates that the subarrays are to be stored in row-major order (true would mean column-major). The number of processors each axis is distribute over is determined by the MPI communicator passed to dp_make_array_desc(). The returned value, A, is the array descriptor. The last two lines are needed to set the initial distribution block sizes for axis 0. The dp_compute_general_block_dist_x() routine simply returns in the integer vector bsizes a set of block sizes that will distribute the array as evenly as possible. The call to dp_init_general_dist() then adjusts the array descriptor A to include this required information. This must be done for each array axis that uses the general-block distribution. Other routines are planned that will perform an initial load balance based on the speeds of the processors or on some other appropriate criteria.

A separate DParLib call can be used to allocate the subarrays, or the program can obtain them in any way suitable. This array is shown in Fig. 1 distributed over a 5×4 mesh of processors. The block sizes along axis 0 are shown as b_n since these can vary, although the sum of the b_n must, in this example, be 200.

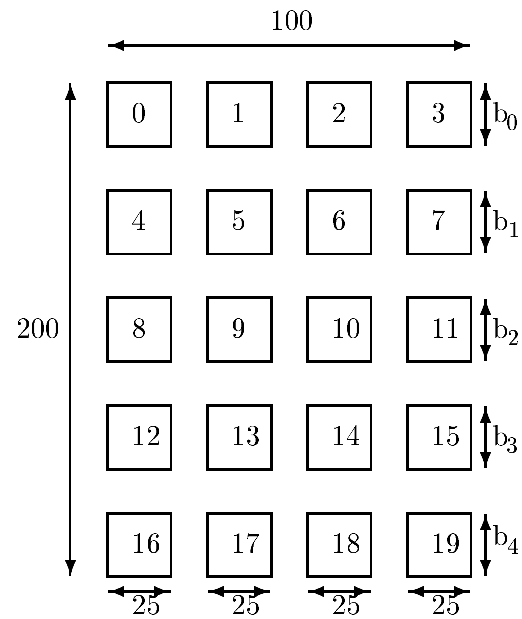

Fig. 1. A 100×200 array distributed over a 5×4 mesh of processors. Each box is one processor, labeled with its MPI rank. The rows of the array are distributed in the general-block distribution scheme and the columns are distributed in the block distribution scheme. The subarrays are of size $25 \times b_n$ where the sum of the b_n must equal 200.

As mentioned previously, the *general-block* distribution scheme was added to DParLib specifically to support load balancing. Unlike the standard block distribution in which a single block size is used on all nodes (except for the block in the last node in the case of an array extent that does not divide evenly into the block size), the general-block distribution allows each node to have a different block size. Initially this distribution was used to allocate and distribute arrays based on a performance rating for each of the available nodes. When a program started, a short timing test was run to determine the relative compute speeds of the nodes using a sample computation from the application. With this information, the faster nodes were assigned larger blocks than the slower nodes. This was a static, one-time, computation completed at the start of the program only, before the arrays were distributed. We have since expanded this support by allowing for the redistribution of arrays.

III. Implementation

A. Measuring Load Balance

Periodically, the state of the application must be determined with respect to the load balance. I will refer to this measurement, along with any required array redistribution, as a load balancing phase of the application. A load-balancing phase is necessarily a collective operation and therefore all nodes will be synchronized. It is expected that load balancing will be performed periodically at some point in the main iteration loop of the program, with all nodes executing the same number of iterations between load balancing phases.

The source of a load imbalance can be either internal, that is, directly a result of the algorithm used in the application, or external. External causes can include a varying system load, network load, memory available, or the raw processing speed of each node. This suggests two different methods for computing the current load imbalance during any load balancing phase.

The first method assumes that the cause of any load imbalance is due entirely to the algorithm. If this is the case, then the amount of the load imbalance should be directly computable from the parameters of the current run. For example, assume an application works on a one dimensional array and that it is known that the algorithm starts perfectly load balanced and slowly, and linearly, becomes imbalanced with the majority of the computations taking place at the ends of the array. This knowledge could be used to compute, based on an iteration count, or some other measure of algorithm progress, the current load imbalance. The time to update each element could be expressed as a function of its index and the load on each node would be the sum of this function over all of the elements it owns. A perfectly balanced system would result in an equal sum on each node.

The second method does not rely on any knowledge of the cause of the load imbalance, although any such knowledge can be factored in to the computation. In this method, the application is instrumented to gather timing data. Assuming the application is structured with a main loop in which the there is computation phases and communication phases, the timings should indicate the amount of time spent on each node for the computations and the amount of time spent for communications. If the processors are force to be synchronized at some point, then timing this synchronization could indicate the level of load imbalance. Processors that have blocked for long periods of time on the synchronization points may have too little computation assigned to them Similarly, comparing total computation times between nodes can indicate the state of the load balance.

This second method has the advantage that it can accommodate load imbalances regardless of the cause. Any load imbalance due to a combination of changing system load, differing processor speeds, algorithm effects, or any other reason will be detected and taken into account.

In general, the results of this measurement of load balance must be some type of performance rating, S_n, for each distribution block size, b_n, that must be computed. For the array shown in Fig. 1, there are 5 block sizes to compute, one for each of the 5 rows of the processor mesh. The next section describes a simple algorithm for computing new distribution block sizes based on these ratings.

How these performance ratings, S_n, are computed is strictly up to the application, although there are `DParLib` routines that can assist in this computation. In particular, one performance rating we have used is a simple *time per subarray slice*, T_{ss}, where a subarray slice is a $(D-1)$-dimensional section of the D-dimensional subarray. Referring again to the array in Fig. 1, if processor mesh row n has a distribution block size of b_n, then each node in that row will have b_n of these *subarray slices*. This measurement, T_{ss}, is the compute time (no communication time or synchronization time is included) used during one iteration of the main loop divided by the node's current distribution block size. For the example in Fig. 1, each subarray slice would be a 1-dimensional array of length 25. Nodes 0–3 would each have b_0 slices, nodes 4–7 would have b_1 slices, and so on. Because each processor row contains 4 nodes, S_n is computed as the *average* of the $T_s s$ values for the processors in that row. Ultimately, all nodes must have the complete set of 5 performance ratings, S_0 through S_4, so that they can all independently compute the new set of block sizes. A single `DParLib` routine is used to compute the complete set of S_ns and broadcast them to all of the nodes. All that is needed is the local T_{ss} for each node. The call for this example is:

```
dp_average_by_slice (A, Tss, 0, Sn);
```

where `A` is the array descriptor, `Tss` is the local time per slice (T_{ss}), `0` is the array axis that is to be used for load balancing, and `Sn` is the result, a vector of average T_{ss}'s.

It is these types of global operations that make parallel programming difficult, error prone, and time consuming and are perfect for encapsulation in a general purpose library such as `DParLib`.

B. *Computing a New Distribution*

If you know that the source of the load imbalance is strictly a function of the algorithm, then computing the best redistribution can also be based strictly on the algorithm.

Otherwise, choosing the best redistribution may not be simple. There are several obvious possibilities to consider.
1. Assume the balance now is stable and correct the distribution to obtain a perfect balance based on the latest S_n values.
2. Assume the balance is moving in one direction and estimate the best average distribution to use from now until the next rebalancing phase.
3. Assume the balance may vary wildly and quickly, due, for example, to external system loads, and only adjust the balance in small increments based on a trend rather than on the actual current measurement alone.

In each case, the algorithm may also impose a restriction on the size of the subarrays. In particular, I expect that some algorithms require a minimum block size of 1 or 2 for each node. A more robust algorithm may handle block sizes of 0 enabling the complete exclusion of a node should it become too slow. After we gain more experience with actual applications, we may implement one or more of these rules in `DParLib` routines rather that requiring the application developer to implement them. A reasonable interface to such a routine is not yet obvious.

After a new distribution has been computed, by whatever means, the choice must be made whether to redistribute or not. If the new distribution is nearly identical to the current distribution, it may not be worth the cost of redistributing. This decision must be made based on the system in use since redistribution costs will vary. After some experience with this library, we may be able to determine some general guidelines that can be applied automatically.

C. *Frequency of Measuring*

Once you have determined how to measure the level of load imbalance, the question remains as to how often this measurement should be taken. Practically, once per iteration of the main loop of the update algorithm would be the upper limit. However, this could be too costly for some algorithms. How often to rebalance can depend on the number of iterations needed to complete the algorithm, the time required per iteration, the expected source of the load imbalances, and the expected behavior of the algorithm with respect to load balancing.

The application described in section IV uses a measure of execution time to set the interval between load balancing phases. For example, the code can be set to check the load balance approximately once an hour. Rather than using a test against a raw timer to trigger a load balancing phase, the program estimates the number of iterations of the main loop that can be completed in the allotted time based on the timings used in the previous load balancing phase. Using an iteration count is preferred over a raw time since each node's timings will vary slightly and so using a local timer will not be sufficient to guarantee that all nodes will start a load balancing phase on the same iteration of the main loop. Using a raw time therefore requires the broadcasting of a single time value to all nodes on each iteration. This would ensure each node makes the same decision to load-balance or not. Using an iteration count avoids this constant broadcasting.

D. An Example Redistribution Scheme

The question now is, how to compute the new block sizes given the set of performance ratings (S_n). For simplicity, assume the array we are using is distributed along a single axis. In this case, we have a 1-dimensional processor mesh. This means that each S_n is associated with a single node. In general, if node i has a performance rating of S_i, and node j has a rating of S_j, and $S_i < S_j$ then node j is slower than node i and so should be assigned less data than node i if they are to finish their work for each iteration at the same time. The slowest node will have a rating of S_{max}.

The goal is to compute the new block sizes such that the product of the performance rating (in seconds/slice/iteration) and the block size (which is the number of slices the node will be assigned) is the same for all nodes, or as close as possible to the same.

A relative performance rating can be computed by first determining S_{max}, the largest S_n, which will be will be the performance rating for the slowest node. Then all of the performance ratings are divided into S_{max} to obtain the relative performance rating $S_n^r = S_{max}/S_n$. This assigns a relative rating of 1 to the slowest node, and a rating greater than or equal to 1 for all the rest of the nodes. The block size to assign to the slowest node can be computed as:

$$b_{min} = \frac{\text{NumSlices}}{\sum_{n=0}^{P-1} S_k^r}$$

where *NumSlices* is the total number of array slices to be distributed (that is, the array extent along the axis that is being redistributed), and P is the number of nodes the array axis is being distributed across. For the remaining nodes, the block size for node i is computed as $b_i = b_{min} \times S_i^r$.

Note that we are computing these new block sizes using the first assumption listed in section III-B, that is, we are adjusting the block sizes to obtain a perfect load balance assuming the current performance ratings are stable and will not change.

With the new set of block sizes just computed, the b_is, we can call a `DParLib` routine which will generate a new array descriptor and another to generate a *communications schedule* for this redistribution. The communications schedule describes all of the communications and other data movement needed to complete the redistribuion of any array that is in the original distribution. Typically more than one array will need to be redistributed so this communications schedule can be reused as needed. The `DParLib` routine which performs the redistribution accepts two array descriptors and the communications schedule so this can be called with each of the arrays to be redistributed. Once all of the arrays have been moved to their new distribution, the communications schedule should be freed.

Assuming that the newly computed block sizes are in the integer vector `bsizes`, the current array is `A`, and we are redistributing along axis 0, the following code will redistribute `A`:

```
axis=0;
A_new = dp_dup_array_desc(A, MPI_FLOAT, false);
dp_init_general_dist(A_new, axis, bsizes);
redist_sched =
    dp_compute_redist_comm(A, A_new, axis)
dp_redist(A, A_new, redist_sched, axis)
dp_free_array(A); /* no longer needed */
A=A_new;
dp_free_redist_comm(&redist_sched);
```

IV. Initial Results

We have used this load balancing support from `DParLib` to improve the performance of a large simulation which we typically run on our IBM SP2.

This is a simulation which models the casting of a bimetal alloy in three dimensions. There are 2 main data arrays, one containing the percent concentration of each metal at each point in the volume (`CON`), and the other containing the current phase of the alloy (solid/liquid) at each point in the volume (`PHI`). Details of this algorithm for a 2-dimensional simulation can be found in the papers by Warren and Boettinger [8] [9]. In addition to these two main data arrays, there are 3 other support arrays of the same size and shape and 2 more that hold the contents of the `CON` and `PHI` from the previous iteration of the update algorithm. This simulation uses a typical finite difference algorithm to update the grid points in the `CON` and `PHI` on each iteration using only values from the adjacent grid points.

We have run this simulation on our SP2 on computational grids of sizes up to 400^3 using up to 32 processing nodes and requiring up to 72 hours of execution time for the largest simulation. Many of these runs have been instrumented to identify performance bottlenecks for possible tuning. The performance data we have collected show that there existed a chronic load imbalance on most runs. There are two main causes for the load imbalance we observed, one algorithmic and the other processor performance.

The algorithmic aspect of the load imbalance is related to the progression of the solidification of the alloy. The volume initially is a super-cooled liquid with a small *seed* of solid placed in one corner of the volume. The process of solidification results in the volume changing from mostly liquid to all solid with a complicated leading surface of freezing liquid (like water freezing into a snowflake). The load imbalance is due to the different computational load required to update a point in the volume depending on whether it is currently in the solid phase, liquid phase, or is transitioning from liquid to solid. Points that are currently solid require slightly more computations.

The arrays are distributed along one of the axes only with each node assigned an equal number of volume points. Initially, assuming that the seed is small, only one node contains a part of the volume that is solid. All other factors being equal, the first node should then be assigned slightly fewer volume points than the rest of the nodes. As the volume begins to freeze (solidify), the load balance can be recomputed based on the relative number of solid and liquid points each node currently contains.

The second and potentially more dramatic cause for the load imbalances we observed was due to the varying computational power of the nodes that comprise our SP2. Updates to the machine and the addition of new compute nodes periodically has resulted in a collection of heterogeneous compute nodes. Without enforcing the assignment of nodes to each run of the simulation (through the batch queuing system), the program must adapt to the collection of nodes it has been assigned each time it is executed.

A third cause for the load imbalances appears when the program is run on one or more of the SP2 nodes that also allow interactive users. Only on these nodes is there ever more than one user program running. This is an unpredictable source for of load imbalance since users normally run short tests on these nodes before submitting them as long running production jobs. If one or more of these nodes are used in the simulation, their performance can degrade quickly at any time and for an unpredictable amount of time.

Dealing with a collection of heterogeneous nodes is not a problem unique to the SP2. We also have collections of PCs and workstations that can be used to run simulations such as this one. Maintaining a reasonable load balance when running on these clusters will also be simplified by using this technique. In general, any parallel machine that can be expanded will eventually have a collection of nodes of varying speed.

Because of the unpredictable nature of the source of possible load-imbalances in this simulation, we have used the second method for measuring the level of load imbalance as described in section III-A. We have instrumented the application to obtain a measure of the per-iteration time spent performing the update of the `CON` and `PHI` arrays.

Two separate performance results will be given. The first is the overall performance improvement that this load balancing achieved on two test runs of the alloy solidification simulation. These results are highly specific to this application, our approach to the frequency and method for computing distribution block sizes, and to the particular machine we used. Therefore only very general conclusions can be drawn from this data. The second performance result relates to the actual cost of each load balancing phase, which consists mostly of computing the new block sizes. This is also highly dependent on the specific application and machine used but is discussed separately since this determines the frequency at which load balancing phases can occur without significantly increasing the total execution time. The cost for redistributing the arrays at any load balancing phase is determined mostly by the number and size of the arrays that must be redistributed and the communications speed of the machine. This can be used to determine a threshold, or minimum block size change, that must be meet before a redistribution is actually performed. These parameters must all be considered when adding this type of load balancing to an application.

To demonstrate the range of possible results, two test cases are presented. The first case uses a computational grid of size 100^3 and 4 processing nodes. The results for this case are shown in Fig. 2. The second case uses a computational grid of size 200^3 and 8 processing nodes. These results are shown in Fig. 3. In each of these figures, the iteration number at which a load balancing phase occured is indicated as well as whether the arrays were redistributed as a result of that load balancing phase.

The dramatic performance increase for the 4-node test case was due to 1 of 4 nodes being heavily loaded at the time of the tests. For the initial distribution, each node had subarrays of size $(25 \times 100 \times 100)$. Focusing only on the distributed axis, the 4 block sizes were (25,25,25,25). At the first load balancing phase (hard coded to occur at iteration 5), the arrays were redistributed so that the heavily loaded node had a block size of 4. The exact distribution obtained was (4,31,32,33). A second load balancing, at iteration 20, changed this slightly to (3,32,32,33). The bottom graph in Fig. 2 shows the execution time for each iteration. The periodic glitches in this graph are due to the periodic snapshots taken of the `CON` and `PHI` arrays which required additional computation and some file I/O. This test clearly shows the large effect that a single slow node can have on the performance of this program.

Figure 3 shows the results of the 8-node test. In this test, 1 of the 8 nodes was significantly faster than the other nodes. Specifically, the computation of the relative performance ratings resulted in 7 nodes receiving a rating of approximately 1.0 and the fast node receiving a rating of approximately 1.6. The initial array distribution in this test assigned subarrays of size $(25 \times 200 \times 200)$ to each node. Focusing only on the distributed axis, the 8 nodes had block sizes of (25,25,25,25,25,25,25,25). At the

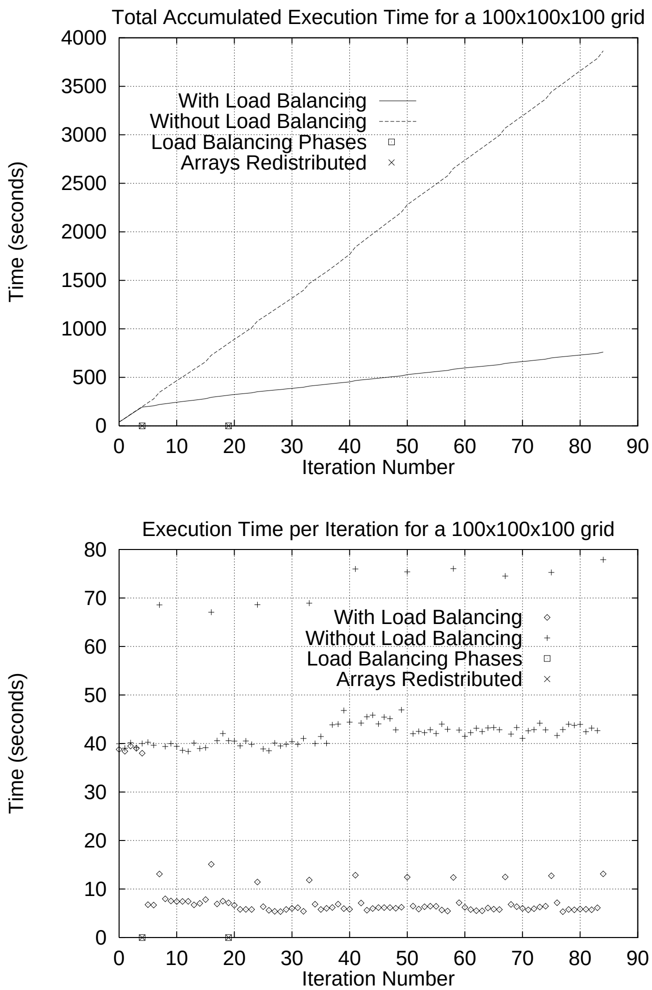

Fig. 2. The top plot shows the total accumulated execution time as a function of the iteration number. The bottom plot shows the time for each iteration. This test used 4 SP2 nodes and a grid of size 100^3. One of the 4 nodes was heavily loaded and therefore much slower.

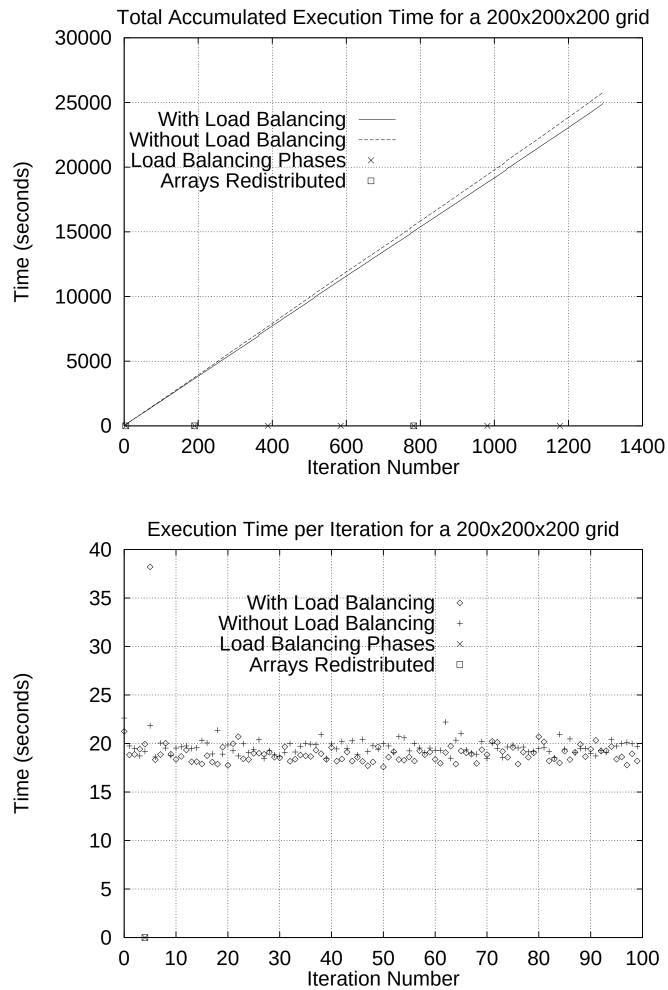

Fig. 3. The top plot shows the total accumulated execution time as a function of the iteration number. The bottom plot shows the time for each iteration from 0 to 100. This test used 8 SP2 nodes and a grid of size 200^3. One of the 8 nodes had a newer, and faster, processor.

first load balancing phase (at iteration 5), the block size for the fast node was increased to 40. The exact distribution after this load balancing was (22,23,23,23,23,23,23,40). At iteration 191, a load balancing phase adjusted this to (21,22,22,23,22,23,23,44). At iteration 783, this was again adjusted to (18,20,21,22,23,24,24,48). Four other load balancing phases resulted in no change in the distribution. A minimum threshold of a 10% change in any nodes block size was required before a redistribution was allowed to occur. The last two small adjustments in the data distribution were probably due to the algorithmic causes discussed previously. The improvement in performance in this test was noticable but not nearly as dramatic as the for the 4-node test. A single fast node out of 8 nodes only changed the block sizes on the slower nodes by a small amount so the decrease in execution time was also small.

The other measurement of performance relates to the cost of each load balancing phase. For the load-balanced 4-node test, the entire test run took 747 seconds, of which the 2 load balancing phases took a total of 1.7 seconds, including one redistribution of the arrays. Because the data in 5 of the 7 arrays in this code are not reused from iteration to iteration, for any redistribution these arrays were simply deallocated and then recreated in the new distribution. This is possible only because the load balancing phase was placed at the end of the main iteration loop, after the data in these arrays is no longer needed. To make sure no additional memory space was needed for the redistribution of the remaining 2 arrays, the redistribution of those arrays took place after deallocating the 5 temporary arrays and before allocating the temporary arrays in the new distribution.

The load-balanced 8-node test case took approximately 7 hours (25,000 seconds). The program was setup to perform a load balancing phase once per hour. This resulted in 7 load balancing phases, of which 3 included a redistribution of the arrays. All 7 load balancing phases, including the 3 redistributions, accounted for approximately 30 seconds of this execution time.

The 4-node test run was short, relative to the 8-node test case, because it was setup to terminate very early in the simulation. Allowing that test case to run to completion would not have added any significant information to this study. Another run of the 4-node test on 4 equal, and unloaded nodes, resulted in no redistributions and a total execution time of about 425 seconds. In all tests, the overhead cost of this load balancing was insignificant.

V. Conclusions

Our results have shown that the dynamic load balancing support in `DParLib` can improve performance in environments that contain heterogeneous compute nodes. The use of dynamic load balancing in a program requires some method of measuring the current level of load imbalance. This measurement can be either static, based directly on the specific algorithm used in the program, or dynamic, based on a run-time timing measurement. The `DParLib` library supports either method, leaving the details of this measurement up to the programmer.

What the `DParLib` offers to the scientific application programmer is a set of routines that can greatly simplify the construction of a data-parallel program in MPI which can maintain a reasonable load balance by redistributing arrays as needed.

This is a recent addition to `DParLib` and additional support routines will likely be added as we gain more experience in designing load balancing schemes for applications.

The benefits of using the data-parallel programming model have been studied for many years. Both languages, such as HPF [10] and SISAL [11], and machines, such as the Thinking Machines CM-2, CM-200, and CM-5, the MasPar MPP, and more recently the Cray T3E, have been designed specifically support this style of parallel programming. One of the lessons learned over the years has been that although this programming model is well suited to many scientific algorithms, it is not so well suited for general purpose computations. The result has been that machines and languages that support only the data-parallel style of programming have not proliferated. We find that in order to use the data-parallel model, we must instead rely on the capabilities of the available, non-data-parallel, languages and machines. We have found that C and Fortran supplemented with an MPI library can support the data-parallel programming style efficiently.

Our data-parallel support library for MPI has been developed to help us design and implement algorithms, as well as port serial applications to parallel, without duplicating all the bookkeeping routines needed to manage the distribution of the arrays. We expand the library with additional data-parallel operations when they have proven to be generally useful.

The source to the library is available (contact the author) and has been written with the intent that it be easily modified, tuned, and extended. Also, the use of this library does not preclude the use of raw MPI routines in any way. This allows for maximum flexibility in implementing parallel algorithms.

Many of the details of data distribution in a data-parallel style MPI program can be transparently handled within a general purpose library. Adopting the data-parallel programming model, with distributed arrays as the main data structure, allows for the construction of many general purpose routines such as array shifting, array reduction, elemental operations, and general array redistribution for dynamic load balancing.

Disclaimer

References

[1] Mounir Hamdi and Chi-Kin Lee, "Dynamic load balancing of data parallel applications on a distributed network," in *Proceedings of the 9th ACM International Conference on Supercomputing*, 1995, pp. 170–179.

[2] Bhaskar Ghosh and S. Muthukrishnan, "Dynamic load balancing in parallel and distributed networks by random matchings," in *SPAA '94. Proceedings of the 6th annual ACM symposium on Parallel algorithms and architectures*, 1994, pp. 226–235.

[3] K. D. Devine and J. E. Flaherty, "Dynamic load balancing for parallel finite element methods with adaptive h- and p-refinement," in *Proceedings 7th SIAM Conference on Parallel Processing for Scientific Computing*, Philadelphia, 1995, pp. 593–598, SIAM.

[4] B. Hendrickson and R. Leland, "The chaco user's guide: Version 2.0," Technical Report SAND94-2692, Sandia National Laboratory, 1994.

[5] Peter Christen, "A parallel iterative linear system solver with dynamic load balancing," in *Conference proceedings of the 1998 International Conference on Supercomputing*, 1998, pp. 7–12.

[6] Petra Berenbrink, Tom Friedetzky, and Ernst W. Mayr, "Parallel continuous randomized load balancing," in *SPAA '98. Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, 1998, pp. 192–201.

[7] Azzedine Boukerche and Sajal K.Das, "Parallel continuous randomized load balancing," in *Proceedings of the 1997 workshop on Parallel and distributed simulation*, 1997, pp. 20–28.

[8] James A. Warren and William J. Boettinger, "Prediction of dendritic growth and microsegregation patterns in a binary alloy using the phase-field method," *Acta Metall. Mater*, vol. 43, no. 2, pp. 689–703, 1995.

[9] James A. Warren, "How does a metal freeze?," *Computational Science & Engineering*, vol. 2, no. 2, pp. 38–49, 1995.

[10] D. B. Loveman, "High performance fortran," *IEEE Parallel and Distributed Technology*, February 1993.

[11] J. T. Feo, D.C. Cann, and R. R. Oldehoeft, "A report on the sisal language project," *Journal of Parallel and Distributed Computing*, December 1990.