

# AN MPI IMPLEMENTATION OF A SELF-SUBMITTING PARALLEL JOB QUEUE

JOHN M. NEUBERGER, NÁNDOR SIEBEN, AND JAMES W. SWIFT

**ABSTRACT.** We present a simple and easy to apply methodology for using high-level self-submitting parallel job queues in an MPI environment. Using C++, we implemented a library of functions, *MPQueue*, both for testing our concepts and for use in real applications. In particular, we have applied our ideas toward solving computational combinatorics problems and for finding bifurcation diagrams of solutions of partial differential equations (PDE). Our method is general and can be applied in many situations without a lot of programming effort. The key idea is that workers themselves can easily submit new jobs to the currently running job queue. Our applications involve complicated data structures, so we employ serialization to allow data to be effortlessly passed between nodes. Using our library, one can solve large problems in parallel without being an expert in MPI. We demonstrate our methodology and the features of the library with several example programs, and give some results from our current PDE research. We show that our techniques are efficient and effective via overhead and scaling experiments. For completeness, we include a detailed description of the library functions enabling our methods, along with an overview of their implementation.

## 1. INTRODUCTION

Our methodology is mainly motivated by our work in computational combinatorics [26] and partial differential equations (PDE) [21, 22, 23]. In combinatorial applications, we have the needs and methods to follow trees. The bifurcation diagrams we study in PDE are similar in structure to these trees, and so we can supervise their generation using similar ideas. Traditional approaches to parallel PDE solvers obtain speed up by distributing operations on large matrices across nodes. Our methodology instead uses job queues to solve such problems at a higher level. Our parallel implementation for creating bifurcation diagrams for PDE is suggested by the nature of a bifurcation diagram itself. Such a diagram contains branches and bifurcation points. These objects spawn each other in a tree hierarchy, so we treat the computation of these objects as stand-alone jobs. For PDE, this is a new and effective approach. Processing a single job for one of these applications may create several new jobs, thus a worker node needs to be able to submit new jobs to the boss node. These jobs require and produce a lot of complicated data. For our proof of concept tests and real applications, we have implemented our ideas in a library of C++ functions, in part to take advantage of that language's facility with complicated data structures. In this article, we refer to our library as *MPQueue*.

The Message Passing Interface (MPI) [12] is a well-known portable and efficient communication protocol which has seen extensive use. In spite of its advantages, MPI is difficult to use because it does not have a high-level functionality. *MPQueue* is small, lightweight, and remedies some of these shortcomings. It uses MPI behind the scenes, is easy to use, and is Standard Template Library (STL) friendly. *MPQueue* has a high-level functionality allowing for the rapid development of parallel code using our self-submitting job queue technique. In order to easily pass our data

---

*Date:* September 26, 2010.

2000 *Mathematics Subject Classification.* 65Y05, 35J60, 37G40.

*Key words and phrases.* MPI, job queue, *MPQueue*, bifurcation, non-attacking queens.

This research was supported in part by the National Science Foundation through TeraGrid resources provided by NCSA (TG-DMS090032).

structures between nodes, our library uses the Boost Serialization Library (BSL) [24]. It was a design goal of ours to make everything as simple as possible, so that one does not have to worry about the details of MPI or serialization when applying our methods. MPQueue is freely available from the authors' web site.

Section 2 contains a brief overview of some popular software packages that facilitate parallel programming. These fall into roughly three categories, threaded applications for *shared memory* multi-core processing, message passing for multi-node, *distributed memory* clusters, and single-system image clusters. Our library uses message passing and is intended for use on distributed memory clusters. Sections 3 and 4 explain line-by-line two simple example programs, one for factoring an integer, and the other for squaring a matrix. The purpose of these examples is not to provide efficient solutions for these demonstration applications, but to present our methodology and describe the key features of MPQueue in full detail. In Section 5 we give a more serious example, namely that of finding placements of non-attacking queens on a chessboard. This code is more complicated, and while not optimal, it does solve the problem on a  $20 \times 20$  board, which would not be possible in a timely manner using serial code. We also investigate efficiency, scaling, and speedup for this example. A computationally intensive and mathematically complex example result for PDE can be found in Section 6. The full mathematical details of the particular PDE we are interested in can be found in [23], along with our state of the art numerical results which use MPQueue. Section 7 contains a detailed description of the library interface. In Section 8 we give an overview of the implementation of MPQueue. This section also includes an overhead experiment and summarizes the evidence of our implementation's solid performance. Section 9 contains some concluding remarks, including possible future refinements to MPQueue.

The authors thank the Department of Physics and Astronomy and the College of Engineering, Forestry and Natural Sciences for providing access and support in the use of two Linux clusters located on the Northern Arizona University campus. We also appreciate the TeraGrid resources provided by NCSA, supported in part by the National Science Foundation.

## 2. RELATED WORK

In this section we give a short description of some existing systems offering parallel computing solutions.

**2.1. Shared memory multithreading systems on multicore processors.** Many existing libraries share some features with MPQueue, but unlike MPQueue, use a shared memory model with multithreading. These systems are not designed for use with a distributed memory cluster. They could conceivably be effectively used with some single-system image cluster software (see Section 2.3).

- Intel's Cilk++ language [17] is a linguistic extension of C++. It is built on the MIT Cilk system [5], which is an extension of C. In both of these extensions, programs are ordinary programs with a few additional keywords: `cilk_spawn`, `cilk_synk` and `silk_for`. A program running on a single processor runs like the original code without the additional keywords. The Cilk system contains a work-stealing scheduler.
- Fastflow [1] is based on lock-free queues explicitly designed for programming streaming applications on multi-cores. The authors report that Fastflow exhibits a substantial speedup against the state-of-the-art multi-threaded implementation.
- OpenMP [9] is presented as a set of compiler directives and callable runtime library routines that extend Fortran (and separately, C and C++) to express shared memory parallelism. It can support pointers and allocatables, depending on the chosen underlying language, and extends X3H5 concepts while to support coarse grain parallelism. It also includes a callable runtime library with accompanying environment variables.

- Intel Threading Building Blocks (TBB) [25] is a portable C++ template library for multi-core processors. The library simplifies the use of lower level threading packages like Pthreads. The library allocates tasks to cores dynamically using a run-time engine.
- Pthreads [6] is a POSIX standard for the C programming language that defines a large number of functions to manage threads. It uses mutual exclusion (mutex) algorithms to avoid the simultaneous use of a common resource.

**2.2. Message Passing Interface systems.** There are a few existing libraries that are built on top of MPI. These are C++ interfaces for MPI, without job queues, and mostly without higher level functionality. It would have been possible to implement MPQueue on top of some of these packages, in particular, on top of Boost.MPI.

- The Boost.MPI library [14, 16] is a comprehensive and convenient C++ interface to MPI with some similarity to our library. Like MPQueue, Boost.MPI uses the BSL to serialize messages.
- MPI++ [2] is one of the earliest C++ interfaces to MPI. The interface is consistent with the C interface.
- mpi++ [15] is another C++ interface to MPI. The interface does not try to be consistent with the C++ interface.
- Object Oriented MPI (OOMPI) [19] is a C++ class library for MPI. It provides MPI functionality through member functions of objects.
- Para++ is a generic C++ interface for message passing applications. Like our interface, it is a high level interface built on top of MPI and is meant to allow for the quick design of parallel applications without a significant drop in performance. The article [8] describes the package and includes an example application for PDE. Para++ uses task hierarchies but does not implement job queues.

**2.3. Single-System Image (SSI) Cluster software.** A single-system image cluster [7, 18] appears to be a single system by hiding the distributed nature of resources. Processes can migrate between nodes for resource balancing purposes. In some of its implementations, it may be possible to run shared memory applications.

- Kerrighed [20] is an extension of the Linux operating system. It allows for applications using OpenMP and Posix multithreading, presenting an alternative to using MPI on a cluster with distributed resources. This approach is seemingly simpler than message passing, but the memory states of the separate nodes must be synchronized during the execution of a shared memory program. This probably is not as efficient as a carefully designed message passing solution.
- OpenSSI is another SSI clustering system based on Linux. It facilitates process migration and provides a single process space. OpenSSI can be used to build a robust, high availability cluster with no single point of failure. It is a general purpose system that is not specifically designed for parallel computing.
- The Linux Process Migration Infrastructure (LinuxPMI) is a Linux Kernel extension for single-system image clustering. The project is a continuation of the abandoned openMosix clustering project which is a fork of MOSIX [3].

**2.4. Summary of Alternatives to MPQueue.** As researchers in PDE and combinatorics, we have a need for the computational power only obtainable through parallel programming. To be practical, we need a method that uses the distributed memory clusters available to us. Our key, proven effective programming idea is to use self-submitting job queues. We did not find an existing package that completely satisfied all of our requirements, and so wrote our own, MPQueue. Our main design goal was to make a simple library that is easy to use, allowing effortless, rapid development of scientific experiments requiring parallel execution, without sacrificing performance.

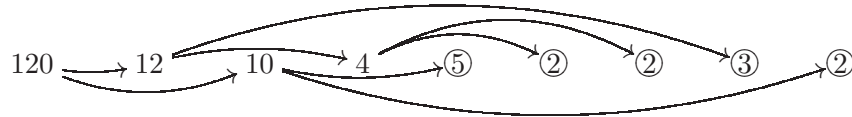


FIGURE 3.1. A possible order of jobs in the job queue during the factorization of 120. The arrows represent the job submission process. The circled jobs are the prime number outputs and so they do not produce new submissions.

Existing shared memory model software for multicore systems have some similar functionality as MPQueue, but they do not suit all needs because of the limitations on the number of cores on currently available hardware. It may be a possibility to use some such systems on top of SSI cluster software to implement techniques similar to ours over a distributed memory cluster, but we do not believe that this would be any simpler than or have as good performance as our simple, more direct message passing approach.

Other message passing systems are mainly concerned with lower level tasks. They offer a large variety of ways to send and receive messages efficiently, but generally do not offer higher level constructs. It would probably be a reasonable programming alternative to implement our key idea of the self-submitting job queue on top of some of these existing systems, but we do not think the result would be as simple to use or have better performance than our implementation. In the sequel, we demonstrate with examples how our simple message passing system handles low level details automatically and offers an easy to use yet powerful and efficient programming methodology based on the self-submitting job queue.

### 3. FACTORING EXAMPLE

Listing 3.1 shows an example program using the MPQueue library for factoring an integer. This example demonstrates many of the features of the library. In particular, it shows the general structure of a program, including the creation, supervision, and post-processing of job queues. It also shows the mechanism by which workers themselves can submit new jobs.

The main idea of the algorithm is to split the input as a product of two integers that are as large as possible, and then submit these factors to the job queue for further splitting. These submissions are done by the workers. The job submission process is visualized in Figure 3.1. Note that the order of job submissions is not fully determined.

We now present the detailed explanation of the code in Listing 3.1

- line 1: We load the MPQueue library. This automatically loads all the libraries needed to use MPI. It also loads the required parts of the BSL [24].
- line 5: Every job type has a positive integer identifier. In this simple example we only have one type of job.

MPQswitch:

- line 8: This function is called every time a new job needs to be done. In this example, the function is only executed by worker nodes. The *job.type* variable determines the type of the job, although in this case there is only one type. Typically a program has several kinds of jobs and so this function contains a switch statement. The *job.data* variable contains the serialized input of the job at the time the function is called. This variable is replaced by the serialized output of the job. The output is the same as the input if the input is a prime, otherwise the output is empty.
- lines 9–10: The input is deserialized into the variable *x*.

```

1  #include "MPQueue.h"
2  #include "math.h"
3  using namespace std;
4
5  const int SPLIT = 1;           // only one job type, so no
6                                // cases in MPQswitch
7
8  void MPQswitch (Tjob & job) {
9      int x;
10     from_string (x, job.data);
11     int sqt = int (sqrt (double (x)));
12     for (int y = sqt; y > 1; y--) // search for divisors
13         if (0 == x % y) { // found a divisor
14             MPQsubmit (Tjob(SPLIT, x/y)); // submit two new jobs
15             MPQsubmit (Tjob(SPLIT, y));
16             job.data = ""; // no output to return
17             return;
18         }
19 }
20
21 int main (int argc, char *argv[]) {
22     MPQinit (argc, argv);
23     MPQstart (); // only the boss returns
24     Tjobqueue inqueue, outqueue;
25     int x = 1120581000; // factor this number
26     inqueue.push(Tjob(SPLIT,x)); // add one job to the job queue
27     MPQrunjobs (inqueue, outqueue); // supervise queue processing
28     cout << x << "_factors_as:\n";
29     while (!outqueue.empty ()) { // get the results
30         int factor; // one factor
31         from_string(factor, outqueue.front().data);
32         cout << factor << "_";
33         outqueue.pop();
34     }
35     MPQstop ();
36 }

```

LISTING 3.1. Prime factorization example program.

- line 11: The worker calculates the square root of  $x$ . We hope to split  $x$  into the product of two integers that are as large as possible. Thus, the search for these factors starts at the square root of  $x$ .
- line 12: The loop tries to find a factor  $y$  of  $x$ .
- line 13: Check if we have found a factor.
- lines 14–15: The worker has found a factor  $y$ . The worker submits two new jobs to the current job queue to split the two factors  $y$  and  $y/x$ .
- line 16: The split produces no output since the further splitting of the found factors is going to be done by other workers. Hence the output is set to empty.
- line 17: The job is done.
- line 19: We did not find any divisors so  $x$  is a prime. Thus the unchanged *job.data* containing  $x$  is returned to the boss node as the result of the job.

main:

- line 21: The main function is executed by every node.
- line 22: The MPI is initialized.
- line 23: The nodes are split into one boss and several workers. Only the boss returns from this function call. The workers are ready to accept jobs.

- line 24: The boss creates two job queues, one for storing jobs to do, and another for storing results.
- line 25: The goal is to find the prime factorization of this number.
- line 26: The boss places one splitting job into the job queue.
- line 27: The boss starts the supervision of the workers. The workers split numbers into factors and submit these factors to *inqueue* for further splitting. The workers return the prime factors, which the boss collects in the *outqueue*. The boss node spends the vast majority of its running time in this function.
- line 29: The boss retrieves all the prime factors from the output queue.
- line 30–33: One of the prime factors is retrieved and printed.
- line 35: The boss halts all the workers.

#### 4. MATRIX SQUARE EXAMPLE

Listing 4.1 shows an example program for calculating the square of a matrix. The example demonstrates several new features of the MPQueue library. Namely, it shows how to efficiently share a large amount of data with all the workers using MPQsharedata, how to serialize struct data types using a template function, and how to use MPQtask to return results to the boss for immediate processing rather than using the output job queue. Unlike the first example, this program uses several job types so there is an actual switch statement in the MPQswitch function.

We now present the detailed explanation of the code in Listing 4.1

- line 4: There are three types of jobs in this example. The job types need to be positive integers, so *NONE* is added to take the unused value of zero.
- line 6: The variable *matrix* contains the input matrix. The variable *square* contains the square of the input matrix, which is the result of the program.
- line 7–10: This data type is used to send back one row of the result matrix, together with the position of this row.
- line 12–16: The BSL requires a simple template function for the serialization of each structure.

MPQswitch:

- line 18: This function is executed by the workers with the *job.type* variable set to *DATA* or *MULTIPLY*. It is also executed by the boss with *job.type* set to *RESULT*.
- line 19: The variable *data* contains one row of the result produced by a worker.
- line 20: Decide the type of the current job.
- lines 21–23: The input matrix is shared by all the workers. Each worker receives the matrix as a *DATA* job. The workers deserialize it and store it locally in the variable *matrix*.
- line 24–29: A worker calculates one row of the goal matrix.
- line 25: The input is deserialized into *data.pos*. It tells the worker which row to compute.
- line 26: The calculated row is stored in *data.result*.
- lines 27–29: The row is calculated using standard matrix multiplication.
- lines 30–31: The worker sends the calculated row to the boss.
- line 33–36: The boss receives a row and puts in into the result matrix *square*.
- line 34: The row is deserialized.
- line 35: The row is placed at the appropriate location.

main:

- line 41: The MPI is initialized.
- line 42: The nodes are split into one boss and several workers.
- line 43: The example input matrix contains rows of length 10 containing a 1 at each position.
- line 44: The input variable *matrix* is initialized.

```

1  #include "MPQueue.h"
2  using namespace std;
3
4  enum { NONE, DATA, MULTIPLY, RESULT };
5  typedef vector < int > Trow;
6  vector < Trow > matrix, square;
7  typedef struct {
8      int pos;                // row index
9      Trow result;           // output row
10 } Tdata;
11
12 template < class Archive > void // needed to serialize Tdata
13 serialize (Archive & ar, Tdata & data, const unsigned int version) {
14     ar & data.pos;
15     ar & data.result;
16 }
17
18 void MPQswitch (Tjob & job) {
19     Tdata data;
20     switch (job.type) {
21     case DATA:              // receive the input matrix
22         from_string (matrix, job.data);
23         break;
24     case MULTIPLY:
25         from_string (data.pos, job.data); // get the row position
26         data.result = Trow (matrix.size (), 0);
27         for (int j = 0; j < matrix.size (); j++) // calculate one row
28             for (int k = 0; k < matrix.size (); k++)
29                 data.result[j] += matrix[data.pos][k] * matrix[k][j];
30         job = Tjob(RESULT, data); // prepare the result
31         MPQtask (job); // send result to boss
32         break;
33     case RESULT:            // receive one output row
34         from_string (data, job.data);
35         square[data.pos] = data.result;
36         job.data = ""; // nothing to return
37     }
38 }
39
40 int main (int argc, char *argv[]) {
41     MPQinit (argc, argv);
42     MPQstart ();
43     Trow row (10, 1); // input matrix containing
44     vector < Trow > mat (row.size (), row); // 1 at every entry
45     square.resize (row.size ()); // container for the output
46     MPQsharedata (Tjob(DATA, mat)); // input matrix sent to workers
47     Tjobqueue inqueue, outqueue;
48     for (int i = 0; i < mat.size (); i++) // every row is a separate job
49         inqueue.push (Tjob(MULTIPLY, i));
50     MPQrunjobs (inqueue, outqueue); // run the jobs
51     for (int i = 0; i < row.size (); i++) { // print the output matrix
52         for (int j = 0; j < row.size (); j++)
53             cout << square[i][j] << "_";
54         cout << "\n";
55     }
56     MPQstop ();
57 }

```

LISTING 4.1. Matrix square example program.

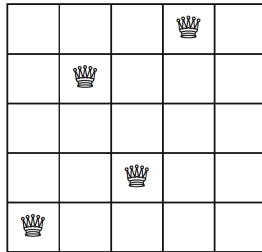


FIGURE 5.1. A partial placement of queens on a  $5 \times 5$  board. In the example code, this placement is encoded as the vector  $(0, 3, 1, 4)$ . This partial placement can be extended to the full placement  $(0, 3, 1, 4, 2)$ . There are 10 distinct placements on the  $5 \times 5$  board.

- line 45: The output variable *square* is resized to the correct dimensions.
- line 46: The input matrix is sent to all the workers
- lines 48–49: The job queue is filled with *MULTIPLY* jobs, each requesting the calculation of one row of the goal matrix.
- line 50: The boss starts the supervision of the workers. At the end of this work, the goal matrix *square* will have been calculated. No other result is created, so *outqueue* is empty.
- lines 51–55: The goal matrix is printed.
- line 56: The boss halts all the workers.

## 5. NON-ATTACKING QUEENS EXAMPLE

We now present an application that illustrates how to avoid the *too many jobs* obstacle to scalability. The example uses local job queues to avoid excessive communication costs. This technique enhances the flexibility of our methodology, allowing for the efficient use of the library in somewhat unexpected situations.

The  $n$ -queen puzzle is a well-known problem in mathematics. It concerns the placement of  $n$  non-attacking queens on an  $n \times n$  chess board. Figure 5.1 shows a valid partial placement with one missing queen. For a survey of results on the generalizations of this problem see [4]. Listing 5.1 shows an example program which counts the number of solutions on an  $n \times n$  board. Figure 5.2 shows the serial pseudo code. We were able to run this code for  $n \leq 20$  on a cluster containing 24 Intel(R) Xeon(TM) 2.80GHz dual CPUs with hyper-threading. For  $n = 20$ , it took 9.0 hours (see Figure 5.3) using the  $96 = 24 \times 2 \times 2$  cores to find the 39,029,188,884 solutions. Note that the state of the art is  $n = 26$ , that is, the number of solutions for  $n = 27$  is not known.

Our code uses a parallel version of a standard backtracking algorithm. A worker keeps a local job queue containing possible search branches, that is valid partial placements. If this queue grows large enough, then the worker submits one of the partial placements to the boss node as a job so that another worker can process it. Submitting jobs to the boss node too early may result in a very large global job queue and a lot of unnecessary communication between the nodes. Submitting jobs too late can starve the workers for jobs. The decision depends on the number of nodes, the speed of the nodes, and the branching factor of the search tree. Our code uses a simple heuristic that depends on an overflow value that we have adjusted experimentally. Figures 5.3, 5.4, and 5.5 show the effect of different choices of the value. This very simple submission process could be fine tuned using the `MPQinfo` command listed in Section 7. Many of the task distribution schemes discussed in the survey paper [13] could be implemented using the `MPQinfo` command.

Table 5.1 and Figures 5.4 and 5.5 show that our approach to the problem scales well. For the size 15 board, we could have used more than the available 96 nodes; to attempt the unknown  $n = 27$  case we could make good use of a very large number of nodes.

```

1  #include "MPQueue.h"
2  #include <deque>
3  using namespace std;
4
5  enum { NONE, PLACE, RESULT };
6  typedef vector < int > Trow;
7  unsigned long int allsolutions = 0;           // total number solutions
8  int overflow = 70;                          // controlls local queue size
9  const int size = 20;                        // size of the board
10
11 bool inline fits (const Trow & row) {
12     int j = row.size () - 1;
13     for (int i = 0; i < j; i++)
14         if ( ( row[i] == row.back () ) || ( abs (row[i] - row[j]) == j - i ) )
15             return false;                   // queens interfere
16     return true;                             // last queen fits
17 }
18
19 void MPQswitch (Tjob & job) {
20     Trow row;                                // a partial placement
21     deque < Trow > rows;                       // local job queue
22     unsigned int solutions = 0;              // local number of solutions
23     switch (job.type) {
24     case PLACE:                               // add queen in next column
25         from_string (row, job.data);
26         rows.push_back (row);                 // populate a local job queue
27         while (!rows.empty ()) {              // still local jobs to do
28             row = rows.back ();
29             rows.pop_back ();
30             for (row.push_back (0); row.back () < size; row.back ()++)
31                 if (fits (row))               // does the new queen fit?
32                     if (row.size () == size) // all queens added
33                         solutions++;          // found a new solution
34                 else {
35                     rows.push_back (row);     // add to local job queue
36                     if (rows.size () > overflow) { // if too many local jobs
37                         MPQsubmit (Tjob(PLACE, rows.front ()); // send one to the boss
38                         rows.pop_front ();
39                     }
40                 }
41             }
42     case RESULT:
43         job = Tjob(RESULT, solutions);        // prepare the result
44         MPQtask (job);                        // send result to the boss
45         break;
46     case RESULT:
47         from_string (solutions, job.data);    // update total solutions
48         allsolutions += solutions;           // with new result
49         job.data = "";                       // no result to return
50     }
51 }
52
53 int main (int argc, char *argv[]) {
54     MPQinit (argc, argv);
55     MPQstart ();
56     Tjobqueue inqueue, outqueue;
57     inqueue.push (Tjob(PLACE, Trow ());       // initial empty placement
58     MPQrunjobs (inqueue, outqueue);
59     cout << allsolutions << "_solutions\n";
60     MPQstop ();
61 }

```

LISTING 5.1. Non-attacking queens example program.

---

**Input:** *size*  
**Output:** *solutions*

---

```

1 add empty row to the queue of partial placements           // no queens placed yet
2 while queue is not empty do
3   move the last job from queue into row
4   increase the size of row                                 // add room for the next column
5   for  $i \in \{0, \dots, size\}$  do
6     set the last coordinate of row to  $i$                  // place the next queen into the  $i$ -th row
7     if row is a good partial placement then
8       if row is a full placement then
9         increment solutions
10      else
11        add row to queue

```

---

FIGURE 5.2. Pseudo code for the serial non-attacking queens placement program. The algorithm uses simple back tracking.

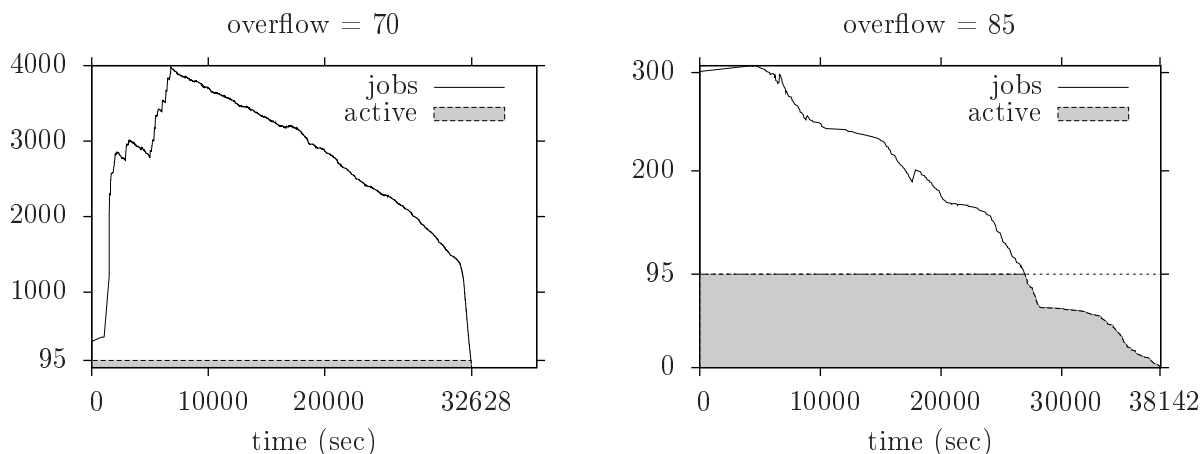


FIGURE 5.3. Load diagrams of the 20-queen placement example code for two different values of the *overflow* variable. In both cases there were 95 available workers. The shaded region shows the number of workers actively working on jobs. The solid curve shows the number of jobs available in the global job queue together with the number of active workers. The unshaded region below 95 represents idle workers. The pictures show that the choice of 85 for *overflow* is too large, since with this choice there are not enough available jobs and so the program takes longer to finish. The optimal value of 70 was determined experimentally. From Figures 5.4 and 5.5 it appears that the optimal *overflow* value is somewhat predictable and depends only on the size of the problem, not the size of the cluster.

We did not take advantage of the symmetry of the problem, which would have immediately resulted in a four-fold speedup. This could be done with minimal work, but our goal here is to present a meaningful yet simple example demonstrating the use of our library and methodology. A

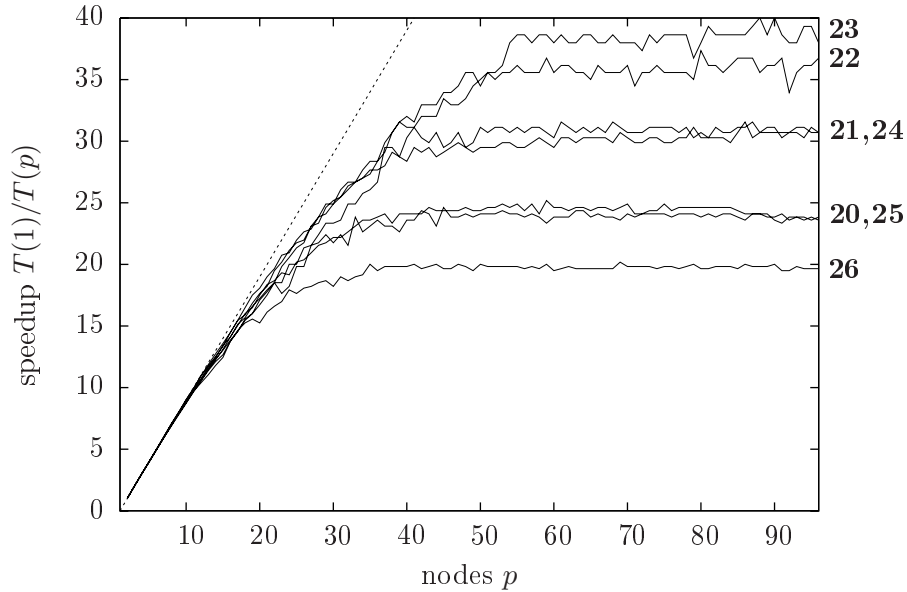


FIGURE 5.4. Speedup as a function of nodes for the 12-queen placement problem. The bold numbers indicate the overflow values used for a given curve. The dashed line is the theoretical maximum speedup of  $p - 1$ , the number of workers. It appears that adding more than 55 nodes does not increase the speedup. This limitation is the result of the small board size. The number of solutions is only 14,200.

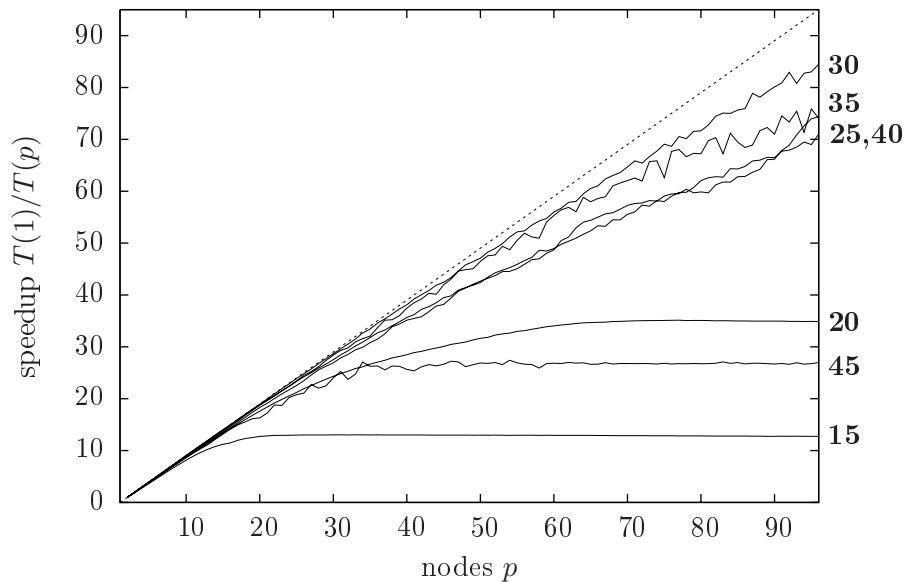


FIGURE 5.5. Speedup as a function of nodes for the 15-queen placement problem. The bold numbers indicate the overflow values used for a given curve. It appears that the optimal overflow value does not depend on the number of nodes, only on the size of the problem. The dashed line again corresponds to the theoretical maximum speedup. The number of solutions for this board size is 2,279,184.

Nodes $p$	Run time $T(p)$	Worker usage	Total usage	Efficiency $\frac{T(1)}{pT(p)}$
1	94.08 sec		94.08 CPU sec	
3	47.05 sec	94.10 CPU sec	141.15 CPU sec	0.67
6	18.77 sec	93.85 CPU sec	112.62 CPU sec	0.84
24	4.11 sec	94.53 CPU sec	98.64 CPU sec	0.95
48	2.05 sec	96.35 CPU sec	98.40 CPU sec	0.96
72	1.44 sec	102.24 CPU sec	103.68 CPU sec	0.91
96	1.11 sec	105.45 CPU sec	106.56 CPU sec	0.88

TABLE 5.1. Processing times for the 15-queen placement problem using the overflow value 30. The *Worker usage* column is the number of workers multiplied by the run time, while *Total usage* includes the boss as well. A constant worker usage would indicate perfect scaling. For larger board sizes, the efficiency increases up to and beyond our maximum available 96 nodes.

possible future extension to the MPQueue library that could help reduce the number of idle workers for problems like this one would be the implementation of priority job queues.

We now present the detailed explanation of the code in Listing 5.1.

- line 6: A placement of queens is stored in a vector. In a valid placement, every column contains exactly one queen. The  $i$ -th entry of the vector contains the location of the queen in the  $i$ -th column.
- line 7: The total number of solutions is stored in this global variable.
- line 9: We work on a  $20 \times 20$  board.

fits:

- lines 11–17: This function checks if the queen in the last column of a placement interferes with the other queens.

MPQswitch:

- line 21: Each worker keeps a local job queue of partial placements.
- line 22: Solutions found by a worker are counted locally.
- line 24: A worker receives a partial placement and tries to extend it.
- line 25: The input is deserialized.
- line 26: The initial job received from the boss is stored as the first job in the local job queue.
- line 27: The worker finishes all the jobs in the local job queue.
- lines 28–29: The worker receives the first job in the local job queue.
- line 30: The job contains a partial placement of queens. The worker tries to add a queen in every possible position of the next column.
- line 31: If this particular placement of the new queen does not interfere with the other queens, then this new placement is a possible extension of the original placement.
- lines 32–33: If the extended placement is a full placement, then this is a new solution.
- line 35: The extended placement is not yet a full placement, so it is stored as a new job in the local job queue.
- lines 36–39: If the local job queue size is large enough, then it is time to submit one of the local jobs to the global job queue.
- lines 42–43: When the local job queue is empty, the worker sends the number of solutions it found to the boss node. This method is more efficient than the use of the output queue would be, since there is a large number of results and we are only interested in the sum of them.

- lines 45–48: The boss receives a result from a worker and updates the total number of solutions.

main:

- line 56: The job queue originally contains an empty placement containing no queens.

## 6. APPLICATION TO NONLINEAR ELLIPTIC PDE

Our library has provided us with an easy way to port to a parallel environment our serial code for solving PDE. The resulting increase in computational power has enabled us to solve the computationally intensive problem

$$(6.1) \quad \begin{aligned} \Delta u + f_s(u) &= 0 && \text{in } \Omega \\ u &= 0 && \text{on } \partial\Omega, \end{aligned}$$

where  $\Delta$  is the Laplacian,  $\Omega$  is the square or cube, and  $f_s$  is the family of nonlinearities  $f_s(u) = su + u^3$ , parameterized by  $s \in \mathbf{R}$ . Other regions and nonlinearities can be handled as well. We had previously developed serial C++ code for obtaining good approximations of solutions to parameterized PDE such as (6.1), provided that the dimensions involved were not too big. The reader can refer to [21, 22] (serial) and [23] (parallel) for the mathematical details of our PDE algorithms, which are based on Newton’s method operating in a coefficient space corresponding to eigenfunction expansions of solution approximations. Generally, our C++ code follows branches of solutions by starting with an initial point on a branch and an approximate tangent vector to the branch, applying Newton’s method to find a next point and corresponding tangent, and repeating until a window is exited. Each time a new bifurcation point is identified on a branch, our code seeks points on new, bifurcating branches, which can then also be followed. Processing a bifurcation point can be very quick but it can also take more time than following a branch. Thus, it is natural that following a branch and analyzing a bifurcation point are the two job types we use.

The required input for both job types is complicated. Each input is a structure with 18 different fields, containing such data as the discretized solution approximation at  $N$  grid points, the corresponding  $M$  eigenfunction expansion coefficients, the parameter  $s$ , the current tangent vector, the branch history, and C++ parameter values that lead to the current state (speed, tolerances, maximum iteration counts, etc). For large problems, the embedded vectors may be very large. The automatic serialization feature of our library makes it easy to pass such data between nodes.

Figure 6.1 shows a partial bifurcation diagram when  $\Omega$  is the square  $(0, \pi)^2$ . The horizontal axis is the parameter  $s$  and the vertical axis is the value of the solution at a generic point  $(x^*, y^*)$  [22]. The diagram demonstrates how branch following creates a tree structure whose growth is unpredictable. Details of the size and speed for this simple example are shown in the first row of Table 6.1. Using two nodes is essentially a simulation of our serial code, since there is only one worker.

The second row of Table 6.1 shows the same summary for a problem where  $\Omega$  is the cube  $(0, \pi)^3$  and we search for the branches connected to the bifurcation point on the trivial branch at  $s = 18$ . Parallelization is essential since it takes about a minute to compute each of the thousands of solutions.

We conclude the section with a few details of our implementation for creating bifurcation diagrams. Solution approximations lie in a subspace spanned by the first  $M$  eigenfunctions of the Laplacian, which themselves have been previously and independently approximated by  $M$  vectors in  $R^N$ , obtained via calls to ARPACK if not known in closed form. For the example results included in this section, these bases are well known in terms of sine functions. The construction of the Jacobian of the object function for Newton’s method requires order  $M^2$  numerical integrations, each requiring order  $N$  arithmetic operations. The number of integrations is reduced somewhat in the presence of symmetry. The search direction system is solved via a standard LAPACK subroutine. Each approximated point requires roughly 4 iterations of Newton’s method. Finding bifurcation

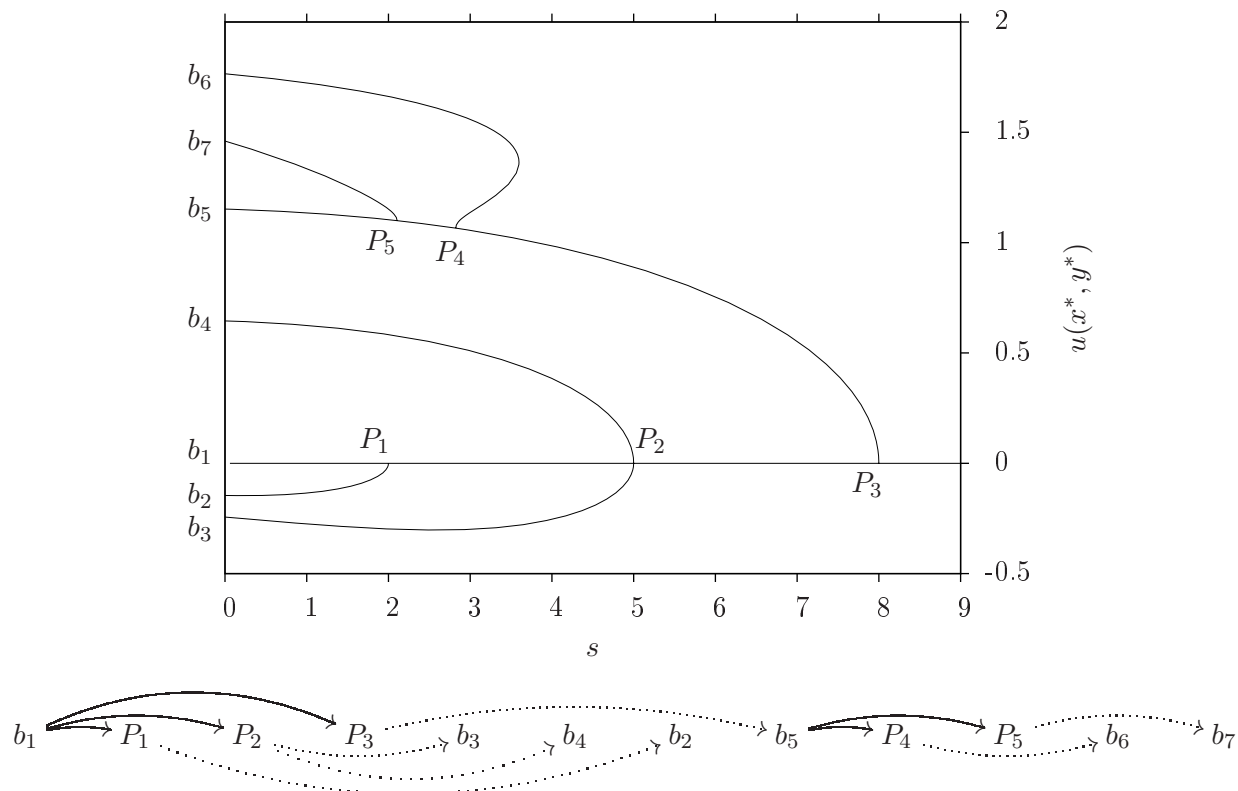


FIGURE 6.1. The first four primary branches and all the branches connected to them for PDE (6.1) on a square region. For this example, a total of 12 jobs are placed on the job queue. The lower diagram represents a possible creation order of these jobs in the job queue during the generation of the bifurcation diagram presented in the upper diagram. A solid arrow represents the submission of a bifurcation analysis job, while a dotted arrow represents a branch following job. The trivial branch  $u = 0$  lying on the horizontal axis is the first branch to be followed. This job, labeled  $b_1$ , is the first and only job to be placed in the job queue by the boss. One of the workers follows this trivial branch and encounters the 3 primary bifurcation points and submits 3 bifurcation jobs  $P_1$ ,  $P_2$ , and  $P_3$ . These three bifurcation points are each processed by a different worker, which leads to the submission of 4 branch jobs. This process continues until the job queue is empty. The workers do not send results to the boss using the outqueue; they store all their results in separate files which are post processed by other scripts after the termination of the program.

points requires the computation of the eigenvalues of the Jacobian, computed by another LAPACK routine.

## 7. DETAILED DESCRIPTION OF THE LIBRARY INTERFACE

In this section, we give a detailed description of the library interface.

- *Job data type*. Every job has one of several possible job types stored in the `type` field. The `data` field contains the serialized input for unfinished jobs and the serialized output for finished jobs.

```

struct Tjob {
    int type;
    string data;
};

```

$\Omega$	$M$	$N$	Branches	Bifurcations	Solutions	Nodes	Run time	Worker usage
$(0, \pi)^2$	323	$41^2$	7	5	165	2	8.17 min	8.17 CPU min
						4	3.73 min	11.19 CPU min
$(0, \pi)^3$	564	$21^3$	385	364	19724	26	22.4 hour	560.0 CPU hour
						50	11.5 hour	563.5 CPU hour
						76	8.98 hour	673.5 CPU hour

TABLE 6.1. Processing times for PDE examples. These results are using NCSA’s IA-64 TeraGrid Linux Cluster (Mercury) with 1.3 GHz nodes [11]. The data for the square region corresponds to the diagrams found in Figure 6.1. The entries for the cube region are typical of the results found in [23]. Note that the cube problem scales well since there are many jobs.

There is a template constructor that takes care of the automatic serialization of the data.

```
template <class T>
Tjob(int type, T data);
```

- *Jobqueue data type.* This type is used to declare the input queue of unassigned jobs, as well as the output queue. The output queue might remain empty if the workers produce empty outputs.

```
typedef queue < Tjob > Tjobqueue;
```

- *Initialize MPI.* This is usually the first function called by the main function. The global variables *MPIrank* and *MPIsize* are set. Note that *MPIrank* ranges from 0 to *MPIsize*−1; the boss node always has *MPIrank* set to 0.

```
void MPQinit (int argc, char *argv []);
```

- *Split the workers from the boss process.* Only the boss process returns from a call to *MPQstart*; the workers start waiting for jobs.

```
void MPQstart ();
```

- *Submit a job into the currently running job queue.* Only a worker can call this function.

```
void inline MPQsubmit (const Tjob & job);
```

- *Ask the boss to run a task.* A task is a short job that is immediately run by the boss and not placed on the job queue. Only a worker can call this function. The *job.data* variable contains the serialized input at invocation and the serialized output upon completion. The function *MPQtask* can be used, for example, to ask for the value of a counter to create a unique file name. It can also be used to return the result of a job as shown in Listing 4.1.

```
template < class T >
void inline MPQtask (Tjob & job);
```

- *Get the number of jobs in the running job queue and the number of workers currently working from the boss.* The workers execute this function, for example, when they need to decide if a new job should be submitted to the currently running job queue.

```
void MPQinfo (int &queuesize, int &sent);
```

- *Assign the jobs in inqueue to the workers and collect the results in outqueue.* Only the nonempty results are collected in the output queue. During the execution of this function, the boss acts as

a supervisor. It sends out jobs, collects the results, accepts new jobs submitted by workers via `MPQsubmit`, and executes tasks requested by the workers via `MPQtask`.

```
void MPQrunjobs ( Tjobqueue & inqueue , Tjobqueue & outqueue );
```

- *Send data to all the workers.* The implementation uses a broadcast mechanism as an efficient way to share a large amount of data, enlisting workers to help in the distribution after they receive the data.

```
template < class T >
void inline MPQsharedata ( const Tjob & job );
```

- *Execute a job.* This function must be implemented by every program. The workers execute this function when they receive a job to run. The boss executes this function when it receives a job request via `MPQtask`. The `job.data` variable contains the serialized input at invocation and the serialized output upon completion.

```
void MPQswitch ( Tjob & job );
```

- *Release the workers and stop MPI gracefully.* This is usually the last line of the program.

```
void MPQstop ();
```

- *Serialize any variable.* That is, turn it into a string. The above listed functions which take as input data structures use this function behind the scene. Standard STL variables are automatically serialized. Serializing a class requires a bit more effort; the creation of a simple template function is required. Consult the BSL for details.

```
template < class T >
string to_string ( const T & in );
```

- *Deserialize a variable.* That is, decode the serialized variable `out` stored in the string `str`.

```
template < class T >
void from_string ( T & out , const string & str );
```

- *Generate load data.* Turning on the compiler directive results in the creation of a data file used to produce a load diagram. See Figure 5.3.

```
#define LOAD_DIAGRAM
```

## 8. IMPLEMENTATION

**8.1. Design.** Passing complicated data structures between nodes requires a significant amount of work in MPI. To avoid this difficulty, we rely on the BSL. This library encodes every standard STL data structure automatically as a string, and more complicated data types can be encoded with a minimal amount of work. The serialization is done mostly automatically using template functions. A serialized data string is sent between the nodes in two steps. In the first step, a preliminary message is sent containing the size of the string together with some additional information like the job type. In the second step, the string itself is sent. Most of this is done using point-to-point communication with `MPI_Send` and `MPI_Recv`. The exception is the `MPQsharedata` command, which uses point-to-point communication for the preliminary message and uses `MPI_Bcast` for the data string.

As a result of calling `MPQstart`, all the nodes become workers except node zero, which becomes the boss. The workers go into a waiting loop, as shown in Figure 8.1, and the boss continues its own work. When the boss requires help, it builds a queue of jobs and becomes a supervisor using

---

**Input:** none  
**Output:** none

---

```

1 repeat
2   wait for a message from the boss
3   switch the message is a
4     case data share request
5       accept the broadcast message from the boss
6       call MPQswitch to handle the broadcast message
7     case job
8       call MPQswitch to run the job
9       send the result back to the boss
10    case stop command
11      stop

```

---

FIGURE 8.1. Pseudo code for the main loop of the workers. The workers get to this loop from the MPQstart function.

---

**Input:** input queue  
**Output:** output queue

---

```

1 while input queue is not empty or workers are working do
2   if input queue is not empty and there are idle workers then
3     assign a job to a worker
4     remove the job from the input queue
5   else
6     wait for a message from a worker
7     switch the message is a
8       case job submission
9         add the job to the input queue
10      case job result
11        if result is not empty then
12          add the result to the output queue
13      case task
14        call MPQswitch to run the task
15        send the result back to the worker
16      case info request
17        send the info back to the worker

```

---

FIGURE 8.2. Pseudo code for the MPQrunjobs function. This function is run by the boss.

`MPQrunjobs`, as shown in Figure 8.2. The boss then assigns jobs in the job queue to available workers and removes these jobs from the job queue. The workers execute `MPQswitch` with the job.

When a worker finishes a job, it sends a message to the boss, letting the boss know that the result is available. The boss accepts the result from the worker and stores it in the output queue. The boss keeps track of the number of jobs sent out. If the job queue is empty or every worker is working already, then it only accepts messages and does not try to assign jobs. The supervision phase ends when the job queue is empty and the result of every assigned job is received.

During the supervision phase, the workers can create new jobs by sending a job to the boss using `MPQsubmit`. The boss puts these jobs into the currently executing job queue.

The workers can use `MPQtask` to ask the boss to execute a task immediately and return the result to the same worker. To satisfy this request, the boss executes `MPQswitch`. This feature can be used, for example, to set or get the value of a global variable like a counter. This allows for rudimentary communication between the workers. The feature can also be used for returning results to the boss. This allows the boss to post-process the results while the workers are busy with their jobs. A sequence diagram of the supervision phase is shown in Figure 8.3. At the end of the program, the boss uses `MPQstop` to tell the workers to quit.

**8.2. Scaling and overhead.** In the non-attacking queens example we presented evidence suggesting that our methodology scales well. In particular, we show with load diagrams that all workers can be kept effectively busy if the problem has very many small jobs that can be managed via a local job queue to avoid excessive communication. We show that the efficiency of our parallel solution to this typical application is very good, up to a certain number of nodes, and that this number of nodes is very large if the problem is sufficiently big. We show that the speedup for this example can be close to ideal when using the optimal overflow, that this optimal overflow value appears to be independent of the number of nodes and only dependent on the size of the problem, and that for a sufficiently large problem, an arbitrary number of nodes can be effectively used. For our PDE application where there are a lesser number of much larger jobs and local job queues are not needed, we have observed similarly good performance indicators (see Table 6.1 and [23]).

In a final test, we demonstrate that the overhead of our implementation is reasonable. To observe the cost of bookkeeping, data serialization, and communication, we ran two experiments where each job was an instruction to the workers to sleep for one second. In the first experiment, we sent no additional input or output data, only the type of the job. In the second experiment, each job was sent with a vector containing 1000 doubles as input, and then the same vector was returned as output. The addition of data causes some unavoidable delay on our distributed memory system due to the serialization time and inherent communication speed of the network. The results are summarized in Table 8.1. It shows that the correspondence between the overhead and the number of jobs is linear in both experiments.

## 9. CONCLUSIONS

We have demonstrated that parallel self-submitting job queues are a powerful computational paradigm. Job queues are high-level constructs not available in other MPI libraries. The ability of workers to submit jobs to the job queue is the key component that gives our method its flexibility and power. We implemented our ideas in a light-weight and easy to use library based on MPI. The library, `MPQueue`, has been used successfully in real research applications. Using our interface saves effort by avoiding low-level coding; scientists without expertise in parallel programming can rapidly develop code to obtain good results for serious research problems. Experts in parallel programming can also take advantage of our methodology for applications that can be broken into relatively large pieces. In applications such as our PDE example found in Section 6, this decomposition is obvious. We also find the approach to be effective in less obvious situations, as demonstrated in the queens example in Section 5 which uses local queues on top of the global job queue to avoid the excessive

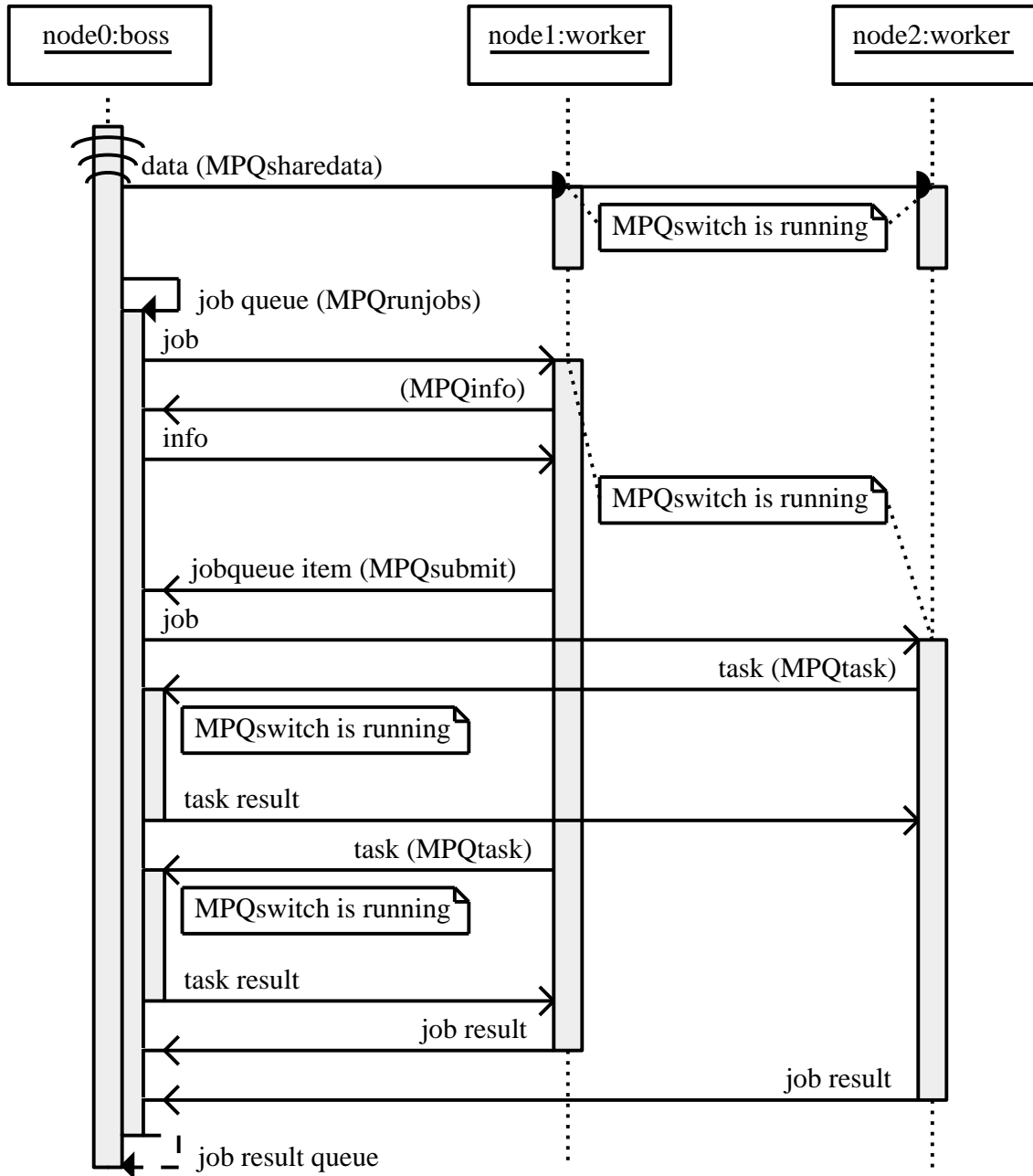


FIGURE 8.3. An example sequence diagram for the communication between the boss and worker nodes. The boss sends out some global data to the workers using `MPQsharedata`. The boss creates a job queue with a single job and then becomes the supervisor by calling `MPQrunjobs`. The first worker takes this job and realizes that an extra job should be created. It uses `MPQinfo` to check if there are available workers to take this additional job. The second worker is not busy, so the first worker submits the new job. The new job is assigned to the second worker. Both workers need the value of a global variable so they use `MPQtask` to get it from the boss. When the workers are finished, they send the results to the boss and the supervision phase stops.

Jobs $n$	without data		with data	
	Run time $T(p)$	Overhead $T(p) - n/95$	Run time $T(p)$	Overhead $T(p) - n/95$
760	8.02	0.02	8.73	0.73
3040	32.07	0.07	24.88	2.88
12160	128.19	0.19	139.50	11.50
48640	512.76	0.76	557.90	45.90
194560	2051.09	3.09	2231.28	183.28
778240	8204.51	12.51	8964.35	772.35

TABLE 8.1. The time in seconds to execute sleep(1) jobs on 96 nodes. We used the same cluster specified in Section 5. Since there are 95 workers, it takes at least  $n/95$  seconds to do  $n$  such jobs. The rest of the time is the overhead. The overhead is approximately 16 microseconds per job with no data sent, and 98 milliseconds per job with a vector of 1000 doubles serialized, deserialized and sent both ways for every job.

communication that would result from transmitting too many small jobs across nodes. We have supplied evidence in this same example that our approach is scalable. The overhead associated with our implementation has been demonstrated to be reasonable. Possibilities for applications are wide; we have also used MPQueue to find winning strategies in combinatorial game theory [26] and in simulating a critical branching random walk [10].

Further improvements to MPQueue could include developing algorithms for automatically adjusting the local queue size (i.e., overflow), or other more sophisticated control procedures for distributing workload, such as the implementation of priority job queues and the capability of workers to send jobs directly to workers. We could easily add to the library the ability for a subset of workers to act as sub-supervisors, each with their own pool of workers. We believe that the currently implemented features of MPQueue will suffice for most applications, and that its simplicity is in many ways an advantage.

In some communication intensive applications, efficiency could be increased by avoiding the serialization of fixed-length data. The Boost.MPI library sends fixed length data more efficiently, without sacrificing convenience. It would be possible to implement MPQueue on top of Boost.MPI or other MPI interfaces in order to take advantage of the optimizations offered by those libraries.

## REFERENCES

- [1] Marco Aldinucci, Marco Danelutto, Massimiliano Meneghin, Peter Kilpatrick, and Massimo Torquati. Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed. In Barbara Chapman, Frédéric Desprez, Gerhard R. Joubert, Alain Lichnewsky, and Frans Peters and Thierry Priol, editors, *Parallel Computing: From Multicores and GPU's to Petascale (Proc. of PARCO 2009, Lyon, France)*, volume 19 of *Advances in Parallel Computing*, pages 273–280, Lyon, France, September 2009. IOS press.
- [2] Purushotham Bangalore, Nathan E. Doss, Anthony Skjellum, and Department Of Computer Science. Mpi++: Issues and features. In *In Proceedings of OONSKI '94*, page page, 1994.
- [3] Amnon Barak, Shai Geday, and Richard G. Wheeler. *The MOSIX Distributed Operating System - Load Balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer, 1993.
- [4] Jordan Bell and Brett Stevens. A survey of known results and research areas for  $n$ -queens. *Discrete Math.*, 309(1):1–31, 2009.
- [5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.
- [6] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [7] Rajkumar Buyya, Toni Cortes, and Hai Jin. Single system image. *IJHPCA*, 15(2):124–135, 2001.

- [8] O. Coulaud and E. Dillon. Para++: A high level C++ interface for message passing. *Journal of Parallel and Distributed Computing*, 51(1):46–62, 1998.
- [9] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.
- [10] János Engländer and Nándor Sieben. Critical branching random walk in an iid environment. (preprint).
- [11] C. Catlett et al. *TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications*. HPC, Amsterdam, 2007. IOS Press 'Advances in Parallel Computing' series.
- [12] MPI Forum. MPI: A message-passing interface standard. version 1.x. <http://www.mpi-forum.org>.
- [13] Ananth Y. Grama and Vipin Kumar. A survey of parallel search algorithms for discrete optimization problems. *ORSA Journal on Computing*, 7, 1993.
- [14] Douglas Gregor and Matthias Troyer. Boost.MPI. <http://www.boost.org>.
- [15] Dennis Kafura and Liya Huang. Collective communication and communicators in mpi++. *MPI Developers Conference*, 0:0079, 1996.
- [16] Prabhanjan Kambadur, Douglas Gregor, Andrew Lumsdaine, and Amey Dharurkar. Modernizing the c++ interface to mpi. In Bernd Mohr, Jesper Trøff, Joachim Worringer, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 266–274. Springer Berlin / Heidelberg, 2006.
- [17] Charles E. Leiserson. The cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA, 2009. ACM.
- [18] Renaud Lottiaux, Benoit Boissinot, Pascal Gallard, Geoffroy Vallée, and Christine Morin. OpenMosix, OpenSSI and Kerrighed: A Comparative Study. Research Report RR-5399, INRIA, 2004.
- [19] B.C. McCandless, J.M. Squyres, and A. Lumsdaine. Object-oriented mpi (oompi): A class library for the message passing interface. *MPI Developers Conference*, 0:0087, 1996.
- [20] C. Morin, R. Lottiaux, G. Vallee, P. Gallard, D. Margery, J.-Y. Berthou, and I. D. Scherson. Kerrighed and data parallelism: cluster computing on single system image operating systems. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 277–286, Washington, DC, USA, 2004. IEEE Computer Society.
- [21] John M. Neuberger, Nándor Sieben, and James W. Swift. Symmetry and automated branch following for a semilinear elliptic PDE on a fractal region. *SIAM J. Appl. Dyn. Syst.*, 5(3):476–507, 2006.
- [22] John M. Neuberger, Nándor Sieben, and James W. Swift. Automated bifurcation analysis for nonlinear elliptic partial difference equations on graphs. *Internat. J. Bifur. Chaos Appl. Sci. Engrg.*, 19(8):2531–2556, 2009.
- [23] John M. Neuberger, Nándor Sieben, and James W. Swift. A parallel implementation of automated bifurcation analysis for nonlinear elliptic partial differential equations in higher dimensions; work in progress. 2010.
- [24] Robert Ramey. Boost Serialization. <http://www.boost.org>.
- [25] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.
- [26] Nándor Sieben. Proof trees for weak achievement games. *Integers*, 8:G07, 18, 2008.

DEPARTMENT OF MATHEMATICS AND STATISTICS, NORTHERN ARIZONA UNIVERSITY, FLAGSTAFF, AZ 86011-5717, USA

*E-mail address:* john.neuberger@nau.edu, nandor.sieben@nau.edu, jim.swift@nau.edu