

An Adaptive, Fault-tolerant Implementation of BSP for Java-based Volunteer Computing Systems

Luis F. G. Sarmenta

Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
lfgs@cag.lcs.mit.edu
<http://www.cag.lcs.mit.edu/bayanihan/>

Abstract. In recent years, there has been a surge of interest in Java-based *volunteer computing* systems, which aim to make it possible to build very large parallel computing networks very quickly by enabling users to join a parallel computation by simply visiting a web page and running a Java applet on a standard browser. A key research issue in implementing such systems is that of choosing an appropriate programming model. While traditional models such as MPI-like message-passing can and have been ported to Java-based systems, they are not generally well-suited to the heterogeneous and dynamic structure of volunteer computing systems, where nodes can join and leave a computation at any time. In this paper, we present an implementation of the Bulk Synchronous Parallel (BSP) model, which provides programmers with familiar message-passing and remote memory primitives while remaining flexible enough to be used in dynamic environments. We show how we have implemented this model using the Bayanihan software framework to enable programmers to port the growing base of BSP-based parallel applications to Java while achieving adaptive parallelism and protection against both the random faults and intentional sabotage that are possible in volunteer computing systems.

1 Introduction

Volunteer computing is a form of distributed computing that seeks to make it as easy as possible for as many people as possible to donate computer time towards solving a parallel problem. By maximizing potential worker pool size and minimizing setup time, volunteer computing makes it possible to build very large networks of workstations very quickly, and creates many exciting new possibilities, including not only global *true volunteer* systems, such as **distributed.net** [4] (which was used to crack the RC5-56 challenge), but also local *forced volunteer* systems for use within institutions that have so far lacked the necessary expertise and time to pool their existing workstations to provide inexpensive supercomputing facilities for their computational needs [17].

In the past two years, there has been a particular surge of interest in *Java-based* volunteer computing systems, which would allow Internet users to join a parallel computation by simply visiting a web page and letting their standard web browser run a Java applet. Compared to existing network-of-workstations (NOW) software such as PVM [10] or MPI [12], such volunteer systems would be orders-of-magnitude easier to setup. Instead of having to spend weeks setting up accounts and installing software on potentially thousands of machines, one can setup an institution-wide NOW literally overnight by simply asking employees to point their browsers to a particular intranet site and to leave their browsers running before they go home or while they work. Moreover, Java's platform-independence allows programmers to worry less about generating code for many different architectures, and concentrate more on the applications themselves.

While *using* such Java-based systems should be very easy, however, *developing* the infrastructure for them is not as straightforward. A major research problem today is that of choosing an appropriate programming model. By enabling anyone to volunteer any kind of computer at any time, Java-based – and particularly *browser-based* – volunteer systems create a heterogeneous and dynamic environment for which currently popular parallel programming models such as shared memory and MPI-style message-passing are not well-suited. In basic MPI, for example, programs are written assuming that the number of physical processors is known and fixed; there is little or no support for situations where nodes leave or join the system at unexpected times. In general, a good programming model for volunteer computing systems has to be *adaptively parallel* and *fault-tolerant*. It cannot depend on static information on the number of nodes in the system and their individual processing speeds. At the same time, it must be able to tolerate data loss or corruption due not only to random hardware and software faults, but also to malicious *saboteurs* pretending to be volunteers but submitting erroneous results.

In this paper, we present a programming model based on the *Bulk Synchronous Parallel* (BSP) model [23, 20], and show how it can be used in Java-based volunteer computing systems. We begin by presenting the BSP programming model, and then show how we have implemented it using the Bayanihan volunteer computing framework [17–19], taking advantage of Bayanihan's existing adaptive parallelism and fault-tolerance mechanisms. We also describe the new programming interface and show how it can be used to write realistic parallel applications. We conclude by presenting some current results, and discussing related, ongoing, and future work.

2 The BSP Programming Model

In BSP, a parallel program runs across a set of virtual processors (called *processes* to distinguish them from physical processors), and executes as a sequence of parallel *supersteps* separated by barrier synchronizations. Each superstep is composed of three ordered phases, as shown in Fig. 1: 1) a *local computation* phase, where each process can perform computation using local data values and

issue communication requests such as remote memory reads and writes; 2) a *global communication* phase, where data is exchanged between processes according to the requests made during the local computation phase; and 3) a *barrier synchronization*, which waits for all data transfers to complete and makes the transferred data available to the processes for use in the next superstep.

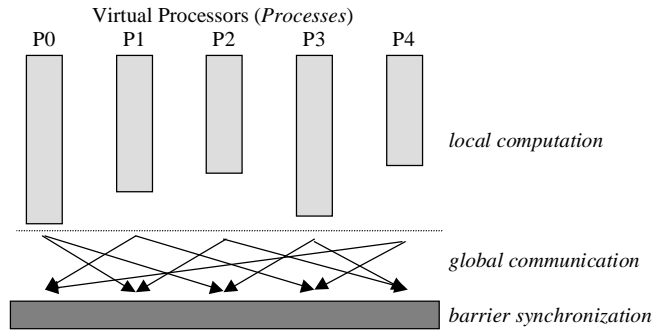


Fig. 1. A BSP superstep.

Figure 2 shows pseudo-code for a simple program where each process passes data to its right neighbor and performs a little local computation. As shown, BSP is very similar to conventional SPMD/MPMD (*single/multiple program, multiple data*)-based programming models such as MPI, and is at least as flexible, having both remote memory (e.g., `bsp_put()`) and message-passing (e.g., `bsp_send()` and `bsp_recv()`) capabilities. The *timing* of communication operations, however, is different – since the global communication phase does not happen until all processes finish the local computation phase, the effects of BSP communication operations are not felt until the *next* superstep. In line 6 of Fig. 2, for example, `s` gets the *unchanged* local value of `d`. Similarly, in line 9, the call to `bsp_recv()` returns `null` or generates an exception.

This postponing of communications to the end of a superstep is the key idea in BSP. It removes the need to support *non-barrier* synchronizations between processes, and guarantee that processes within a superstep are *mutually independent*. This makes BSP easier to implement on different architectures than traditional models, and makes BSP programs easier to write and to analyze mathematically. For example, since the timing of BSP communications make circular data dependencies between BSP processes impossible, there is no risk of deadlock or livelock in a BSP program. Also, the separation of the computation, communication, and synchronization phases allows one to compute time bounds and predict performance using relatively simple mathematical equations [20].

Because of these advantages, a growing number of people are now applying BSP to a wide variety of realistic parallel applications [5], ranging from small general-purpose numeric libraries (e.g., for matrix operations), to large computational research applications used in both academia and industry (e.g.,

```

1 void shift2()
2 { right = ( bsp_pid()+1 ) % bsp_nprocs();
3   s = 0; d = bsp_pid();
4   bsp_register( "d",&d ); // register variable "d"
                               // begin superstep 0
5   bsp_put( right,"d",d ); //  put my pid to right's d *at next sync*
6   s = d;                    //  s <- current d (i.e., my pid )
7   bsp_sync();                //  synchronize
                               //  begin superstep 1; d is now left's pid
8   bsp_send( right, d ); //  send d to right * at next sync *
9   s += bsp_recv();         //  error! nothing to receive yet!
10  bsp_sync();                //  synchronize
                               //  begin superstep 2; msg is now in recv q.
11  d = bsp_recv();           //  receive d sent in line 7
12  s += d;                    //  s <- s + 2nd-left's pid
13 }

```

Fig. 2. BSP pseudo-code for a two-step cyclic shift.

computational fluid dynamics, plasma simulation, etc.). By implementing BSP in Java, we hope to enable programmers to port these applications to Java, and make it easier to do realistic research using volunteer computing systems.

3 Implementing the BSP Model

3.1 The Bayanihan Master-Worker Based Implementation

BSP's structure makes BSP relatively easy to implement in Java-based volunteer computing systems. Since processes are independent within supersteps, we can package the work to be done in a superstep as separate work packets and farm them out to volunteer worker nodes using a master-worker system. Figure 3 shows how we have implemented BSP using the existing master-worker model of the Bayanihan framework [19]. As shown, BSP processes are implemented as **BSPWork** objects and placed in a *work pool* on the server. Volunteer *worker clients* each have a *work engine* which runs in a loop, repeatedly making remote method calls via HORB [14] (a distributed object library similar to Sun's RMI [22]) to its corresponding *advocate* on the server side to request more work. Each advocate forwards these calls to the *work manager*, which looks into the work pool, and returns the next available uncompleted work object.

When the work engine receives the new work object, it calls the work object's **bsp_run()** method, which starts the local computation phase of the superstep indicated by the **restorestep** field. As **bsp_run()** executes, communication requests are logged in **MsgQueue** fields, and the values of shared variables are saved in the **bspVars** field, a hash table that stores objects under **String**-type names. Execution continues until **bsp_sync()** is called, at which point **restorestep** is incremented and control returns to the work engine. The work engine then sends

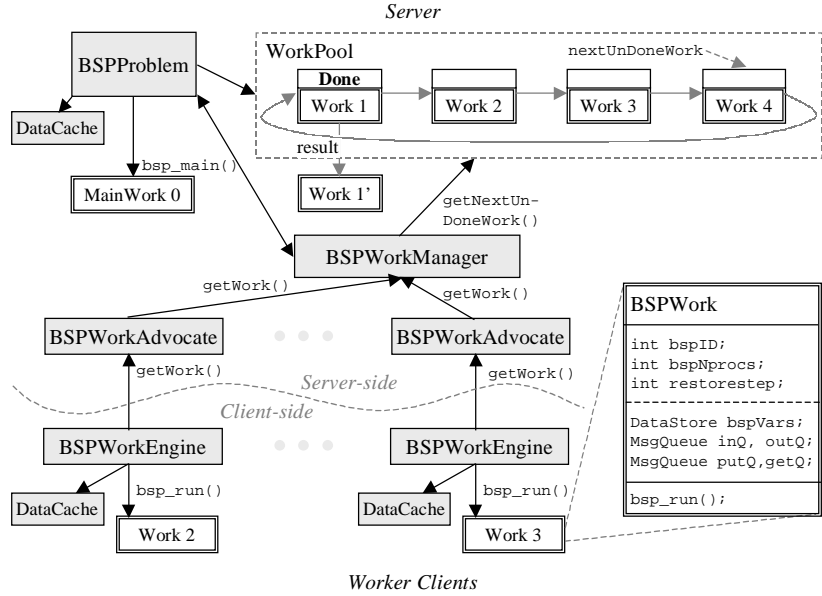


Fig. 3. Implementing BSP on top of the Bayanihan master-worker framework.

the whole work object – including `bspVars`, the `MsgQueue`'s, and `restorestep` – back to the server, which in turn marks the original work object “done”, and stores the returned work object as its *result*.

When all work objects are done, the work manager notifies the *problem* object, and the global communication and barrier synchronization phases begin. The `BSPProblem` object goes through the result work objects, and performs all the global communication operations by moving data from one work object to another. The remote memory access operations (`bsp_get()` and `bsp_put()`) are done by reading and writing to the work objects' `bspVars` fields. Similarly, the message-passing-style `bsp_send()` operation is performed by enqueueing the message data into the destination node's `inQ` field, so that in the local computation phase of the next superstep, they can be retrieved by calling `bsp_recv()`. After all the communication operations have been performed, each work object in the work pool is replaced by its updated version, and control returns to the work manager. The next superstep begins as the work manager starts giving workers new work objects wherein `restorestep` indicates the new superstep, `bspVars` contains updated variable values, and `inQ` contains new incoming messages.

In addition to the work objects in the work pool, the server also has a special *main work* object, which is like other work objects, except that it is not farmed out to workers, but is run on the server itself. In place of `bsp_run()`, the server calls the `bsp_main()` method. As described in Sect. 4.1, the main work object can be used for such tasks as *spawning* new work objects, coordinating other work objects, collecting and displaying results, and interacting with users.

3.2 Adaptive Parallelism and Fault-Tolerance

Since no communications occur while these work packets are running, we can rerun work packets or run several copies of the same work object at a time, resolving any redundancies safely at the end of the superstep. This makes it possible to implement adaptive parallelism and fault-tolerance mechanisms.

Checkpointing and Process Migration. Previous research in C-based BSP systems has shown that the call to `bsp_sync()` at the end of the superstep provides a convenient place to *checkpoint* the program state and migrate processes to achieve adaptive parallelism and crash-tolerance [15, 13]. In our implementation, checkpointing happens automatically as each work object returns itself to the server at the end of a superstep. Since the returned work object includes all necessary process state in `bspVars`, the `MsgQueue`'s, and other non-transient fields, it can easily be saved and restored later or migrated to another machine.

Eager Scheduling. The underlying Bayanihan master-worker framework provides a simple form of adaptive parallelism called *eager scheduling* [3, 19]. As shown in Fig. 3 (Sect. 3.1), work objects in the work pool are stored in a circular list, with a pointer keeping track of the next available uncompleted work. As workers call `getWork()`, the pointer moves forward, assigning different work objects to different workers. Faster workers will tend to call `getWork()` more often, and naturally get a bigger share of the total work. This gives us a simple but effective form of dynamic load balancing. Moreover, since the list is circular, previously-assigned but uncompleted work can be reassigned to other workers. This “eager” behavior guarantees that slow workers do not cause bottlenecks – fast workers with nothing left to do will simply bypass slow ones, redoing work themselves if necessary. (Redundant results are simply ignored.) It also provides a basic form of crash-tolerance. If a worker crashes or quits, and leaves its work undone, for example, it is alright because the work will eventually be reassigned to another worker. In this way, computation can go on as long as at least one worker is still alive. In fact, even if all the workers crash or quit, the computation can continue as soon as a new worker becomes available.

Fault and Sabotage Tolerance. To be useful on a large-scale, volunteer computing systems must be able to tolerate not only random data faults, but also intentional *sabotage* from malicious volunteers producing erroneous data. While achieving such fault-tolerance is difficult in most parallel programming models, where data can move arbitrarily between processes, it is relatively easy in master-worker systems, where result data are neatly packaged into independent packets that can be checked, compared, and recomputed, if necessary.

At present, we have implemented and are studying two approaches to fault- and sabotage-tolerance for master-worker systems: *majority voting* and *spot-checking*. In *majority voting*, the work manager waits for an agreeing majority of at least m out of r results from different workers to be received before a

work object is considered done. If r results are received without reaching a majority agreement, all the r results are invalidated, and the work object is redone. In *spot-checking*, the work manager precomputes a randomly selected work object before the start of a new superstep, and returns this *spotter work* with probability p at each call to `getWork()`. Then, if a worker is sent a spotter work and does not return the expected result, it is *blacklisted* as untrustable, and the work manager *backtracks* through all the current results, invalidating any results dependent on results from the offending worker. Spot-checking may be less reliable in cases where saboteurs do not *always* submit bad data, but is much more time-efficient, since it only takes an extra p fraction of the time instead of taking m times longer as voting does.

Early experiments have demonstrated the potential effectiveness of these techniques [19]. Spot-checking, tested with backtracking but without blacklisting, was particularly promising, maintaining error rates of only a few percent, even in cases where *over half* of the volunteers were saboteurs. Furthermore, the high rate at which saboteurs were being caught suggests that these error rates will be even smaller with blacklisting enabled. While still not zero, these low error rates may be enough for some applications where a few bad answers may be acceptable or can be screened out. These include *image rendering*, where a few scattered bad pixels would be acceptable, some *statistical computations*, where outliers can be detected and either ignored or double-checked, *genetic algorithms*, where bad results are naturally screened out by the system, and others. For other applications that assume 100% reliability, however, we are exploring ways of *combining* voting, spot-checking, backtracking, and blacklisting, as well as developing new mechanisms, to reduce the error rate as much as possible.

Any progress made in developing mechanisms for master-worker systems in general can be applied automatically to BSP programs by virtue of our master-worker based implementation of BSP. Programmers need only define an appropriate `hasSameValue()` method for each data class used in their work objects so that two work objects can be compared just like ordinary result objects by comparing their `bspVars` and `MsgQueue` fields.

4 Writing BSP programs in Java

To write a Bayesian BSP program, one simply extends the `BSPWork` base class and defines an appropriate `bsp_run()` method containing the BSP program code. No changes to the Java language or Java virtual machine (JVM) are required. A Bayesian program can run on any JDK 1.0-compliant browser or JVM.

4.1 The Bayesian BSP Methods

The `BSPWork` class provides several methods which subclasses inherit and can call within `bsp_run()` in the same way one would call library functions in C. These are based on `BSPLib` [5], a programming library for C and Fortran currently being used in most BSP applications, but have been modified to account for

language and style differences between Java and C. These methods are shown in Table 1, subdivided by function into the following categories (in order): *inquiry*, *message passing*, *direct remote memory access*, and *synchronization*.

Table 1. Basic Bayanihan BSP methods

method	meaning
<code>int bsp_pid()</code>	my BSP process identifier
<code>int bsp_nprocs()</code>	number of virtual processes
<code>double bsp_time()</code>	local time in seconds
<code>void bsp_send(BSPMessage msg)</code>	send prepared message
<code>void bsp_send(int dest,int tag, Object data)</code>	create tagged message, and send to <i>dest</i>
<code>BSPMessage bsp_recv()</code>	receive message
<code>int bsp_qsize()</code>	number of incoming messages
<code>BSPMessage bsp_peek()</code>	preview next message
<code>int bsp_peekTag()</code>	preview next message tag
<code>void bsp_save(String name,Object data)</code>	save data under <i>name</i>
<code>Object bsp_restore(String name)</code>	restore data under <i>name</i>
<code>Object bsp_unregister(String name)</code>	remove data under <i>name</i>
<code>void bsp_put(int dest,String destName, Object data)</code>	write data to <i>destName</i> variable of <i>dest</i>
<code>Object bsp_get(int src,String srcName, String destName)</code>	read <i>srcName</i> variable of <i>src</i> to local <i>destName</i>
<code>boolean bsp_sync()</code>	sync; possibly checkpoint and migrate
<code>boolean bsp_step()</code>	mark beginning of superstep
<code>boolean bsp_cont()</code>	mark continuation of superstep

Message-Passing. The message-passing methods are similar in functionality to those found in systems such as MPI [12] or PVM [10], except that, as noted earlier, messages sent during a superstep can only be received by the recipient in the *next* superstep. In the present implementation, messages are queued at the recipient in arbitrary order, but can be identified by tag using the `bsp_peekTag()` method, or by calling the `getBSPTag()` method of the `BSPMessage` object returned by `bsp_recv()`. The data itself can be retrieved as an object by calling the `getData()` method of the `BSPMessage`. Any data object that implements the `java.io.Serializable` interface or has a HORB proxy (generated by the HORB compiler) can be passed in a message. Note that Java’s object-oriented features and platform-independence make sending and receiving messages much easier than in C-based systems such as MPI or PVM, where one must explicitly decompose complex messages into its primitive components.

Direct Remote Memory Access. Bayanihan BSP also provides the direct remote memory access (DRMA) methods `bsp_put()` and `bsp_get()`, which allow

processes to write to and read from remote variables by specifying the owner's process ID and the `String`-type name of the desired variable.

The `bsp_save()` method registers a variable into the `bspVars` table. This allows it to be saved at a checkpoint, and to be read remotely using `bsp_get()`. A variable remains registered until it is removed with `bsp_unregister()`, and its contents can be modified freely by its owner using accessor methods. If the object itself is replaced, however, then `bsp_save()` must be called again to update the reference stored in `bspVars`. A variable can also be registered by processes other than its owner by using the `bsp_put()` method. This technique can be used by main work objects to initialize worker variables.

The `bsp_restore()` method restores the value of a variable so it can be used within a superstep. Variables that are used in all or most supersteps can be restored in the `initVars()` method, which is always called before `bsp_run()`. This guarantees that the variables are always restored and available at the beginning of a superstep. If an attempt is made to restore an unregistered name, `bsp_restore()` returns `null`. Typically, variables restored with `bsp_restore()` are declared as `transient` fields since their values are already stored in `bspVars` and do not need to be serialized independently.

Like the message-passing methods, the DRMA methods do not take effect until the next superstep. Thus, for example, a `bsp_get()` called during a superstep will only update the destination variable in the next superstep. The value received by `bsp_get()` is the value of the source variable at the end of the local computation phase, before any puts or gets are done. Similarly, the value sent by `bsp_put()` is the value of the data object at the end of the local computation phase. This may be different from the value of the data object at the time `bsp_put()` was called, unless the data is first *cloned* before `bsp_put()` is called.

Synchronization and Checkpointing. Calls to `bsp_sync()` not only indicate barrier synchronizations, but also mark potential checkpoint locations where a virtual process may be paused, saved, and moved to a different physical processor. While pausing and moving processes is easily done as described in Sect. 3.1, resuming execution is harder. In C, resuming execution can be done relatively easily by either using system-specific low-level mechanisms (as done in [13]), or using a pre-compiler to annotate the source code with a jump table using `switch` and `goto` statements [21]. Unfortunately, these techniques are impossible in Java, where there are no explicit `goto`'s – not even at the virtual machine level.

Thus, instead of using direct jumps, we use `if` statements to *skip* over already-executed code. Figures 4 and 5 show an example. In general, the code for a superstep should be enclosed in an `if (bsp_step())` block and ended with a call to `bsp_sync()`, as shown in step 0. As a syntactic shortcut, an `if (bsp_sync())` statement can be used to end a superstep and start the next one at the same time, as shown at the end of step 1. In cases where function calls, `for`, `while`, `if-else`, or other special constructs prevent a superstep's code from being enclosed in one block, one can use an `if (bsp_cont())` statement to continue the superstep on the other side of the offending statement as shown.

```

void bsp_run()
{ step 0;
  bsp_sync();
  step 1;
  bsp_sync();
  step 2, part a;
  for ( int i=0; i<n; i++ )
  { step 2+i, part b;
    bsp_sync();
    step 2+i+1, part a;
  }
  // code continued in next col. ...
}

step 2+n, part b;
foo(); // func with sync
step 3+n, part b;
bsp_sync();
} // end bsp_run()

void foo()
{ // cont. current step
  step 2+n, part c;
  bsp_sync();
  step 3+n, part a;
} // end foo();

```

Fig. 4. BSP pseudo-code without code-skipping.

```

void bsp_run()
  throws MigrateMeException
{ if ( bsp_step() )
  { step 0
  }
  bsp_sync();
  if ( bsp_step() ) // explicit
  { step 1
  }
  if ( bsp_sync() ) // shortcut
  { step 2, part a
  }
  for ( int i=0; i<n; i++ )
  { if ( bsp_cont() )
    { step 2+i, part b
    }
    if ( bsp_sync() )
    { step 2+i+1, part a
    }
  }
  // code continued in next col.
}

if ( bsp_cont() )
{ step 2+n, part b
}
foo(); // func with sync
if ( bsp_cont() )
{ step 3+n, part b
}
bsp_sync();
} // end bsp_run()

void foo() throws
  MigrateMeException
{ if ( bsp_cont() )
  { // cont. current step
    step 2+n, part c
  }
  if ( bsp_sync() )
  { step 3+n, part a
  }
} // end foo()

```

Fig. 5. Transformed BSP pseudo-code with code skipping.

Figure 6 shows how these methods are implemented. When restarting a checkpointed `BSPWork` object, `bsp_run()` is called with `curstep` initialized to 0, and `restorestep` set to the step to be resumed. The `bsp_step()` and `bsp_cont()` functions, called at the beginning of each code block, return `false` while `curstep` has not yet reached `restorestep`, allowing blocks that have already been executed to be skipped. At the end of each skipped superstep, the call to `bsp_sync()` increments `curstep` so that `curstep` eventually reaches `restorestep`, and the desired superstep's block is allowed to execute. The next call to `bsp_sync()` then ends the superstep by updating `restorestep` and throwing an exception which causes `bsp_run()` to return control to the work engine.

```

protected boolean bsp_sync()                protected boolean bsp_step()
    throws MigrateMeException              { return ( curstep>=restorestep );
{ if ( curstep++ >= restorestep )          }
    { restorestep = curstep;
      throw new MigrateMeException();
    }
    return bsp_step();                    protected boolean bsp_cont()
}                                           { return bsp_step();
                                           }

```

Fig. 6. Code for `bsp_sync()`, `bsp_step()`, `bsp_cont()`.

As demonstrated in Fig. 5, these methods can be used even when `bsp_sync()` calls occur in loops, `if-else` blocks, and function bodies. Moreover, the code transformation rules are simple enough to be applied manually without using pre-compilers. In many cases, all a programmer needs to do is enclose the superstep code in braces, and add an `if` in front of `bsp_sync()`. As shown here, and more clearly in the realistic sample code in Sect. 4.2, the resulting code is still reasonably readable. In fact, the indented blocks may even help emphasize the divisions between supersteps. This simplicity offers a significant advantage in convenience over other programming interfaces that require language changes and sophisticated pre-compilers, since it does not require programmers to generate and keep track of extra files, and allows them to more easily use off-the-shelf integrated development and build environments. Furthermore, if more readability is desired, one can implement a parallel language with nothing more than a C-style preprocessor by using `#define` macros to replace `if (bsp_step())` with `parbegin, bsp_sync()`; with `parend`, and `if (bsp_cont())` with `parcont`.

As far as we know, the only significant disadvantage of this code-skipping technique is that resuming execution inside or after an unbounded or very long loop may take unnecessary extra time because the skipping has to go through all previous iterations of the loop before reaching the current superstep. We are currently working on a solution to this problem, which may take the form of a special version of `bsp_cont()` which updates loop variables and `curstep` without having to iterate through the loop.

BSPMainWork. A Bayesian BSP program is started by giving the name of a subclass of **BSPMainWork** to the **BSPProblem** object. **BSPMainWork** is a subclass of **BSPWork** that provides extra methods for server-side control of a computation, as shown in Table 2. The **BSPProblem** object assigns the **BSPMainWork** the reserved ID of 0, and calls its **bsp_main()** method, which is then expected to spawn work objects of the desired class. As the program runs, the main work object runs together with the other processes, and can communicate with them using BSP methods. In addition to spawning new work, the main work object can be used for coordinating other work objects, for collecting and sending results to interested *watcher clients* [19], and for receiving and reacting to user requests. This is similar to the common programming practice in SPMD/MPMD systems like PVM, MPI, and BSPLib of having a “master” or “console” process that handles data distribution and coordinates the operation of all the other processes.

Table 2. Basic **BSPMainWork** methods

method	meaning
<code>int bsp_spawn(String class, int n)</code>	spawn <i>n</i> BSPWork 's of class <i>class</i>
<code>int bsp_spawn(BSPWork w, int n)</code>	spawn <i>n</i> BSPWork 's of same class as <i>w</i>
<code>void bsp_postResult(Result res)</code>	post result for watchers
<code>Object bsp_getRequest()</code>	get user input from watchers
<code>boolean isMain()</code>	am I the main work?

To write an application, one can write separate **BSPMainWork** and **BSPWork** classes, or a single **BSPMainWork** class with separate **bsp_main()** and **bsp_run()** methods. The latter is useful when the main work object has the same fields as the other work objects, and has the added advantage of keeping all the code in one file. Since **bsp_main()** defaults to calling **bsp_run()**, one can also write a single **BSPMainWork** class with a single **bsp_run()** method containing code for *both* the main work and the other processes. This allows SPMD-style programming, and is useful in cases where the main work code is very similar to the worker code, or where it is useful to see the master and worker codes for each superstep in the same place. The **isMain()** function can be used within **bsp_run()** to let the main work process differentiate itself.

4.2 Sample Code.

Figure 7 shows a matrix multiplication example demonstrating the use of the Bayesian BSP programming interface. Here, we use a single **BSPMainWork** subclass with separate **bsp_main()** and **bsp_run()** methods. The **initVars()** method is called in both the worker and main work objects before running **bsp_run()** and **bsp_main()**. The algorithm used is based on one of the MPI example programs in [12], where each process *i* from 1 to *n* is given a copy of the matrix *B* and row *i* – 1 of *A*, and computes row *i* – 1 of the product *C*.

```

public class BSPMatMultMain extends BSPMainWork
{
    transient Matrix A, B, C;
    // ... some code omitted ...

    // called before bsp_main() and bsp_run()
    public void initVars() // initialize B and C
    {
        B = (Matrix)bsp_restore( "B" );
        C = (Matrix)bsp_restore( "C" );
    }

    // run by main work on server
    public void bsp_main() throws TaskMigratedException
    {
        if ( bsp_step() ) // *** superstep 0 ***
        {
            // create n-by-n matrices, spawn, send a and B
            A = createSampleMatrix( n ); B = createSampleMatrix( n );
            bsp_spawn( this, n ); // spawn n workers of this type
            for ( int i=1; i <= n; i++ )
            {
                bsp_put( i, "B", B ); // write B
                bsp_send( i, i-1, A.getRow(i-1) ); // send row of A
            }
        }
        if ( bsp_sync() ) // *** superstep 1 ***
        {
            // workers compute C; main does nothing
        }
        if ( bsp_sync() ) // *** superstep 2 ***
        {
            // collect and print results on server
            BSPMessage msg;
            C = new Matrix( n );
            while( (msg = bsp_recv()) != null ) // get result rows
            {
                C.setRow( msg.getBSPTag(), (Row)msg.getData() );
            }
            trace( "C = \n" + C ); // print C
            bsp_save( "C", C ); // allow workers to get C
        }
        if ( bsp_sync() ) // *** superstep 3 ***
        {
            // workers print C
        }
    }

    // run by worker clients
    public void bsp_run() throws TaskMigratedException
    {
        if ( bsp_step() ) // *** superstep 1 ***
        {
            // restored local var B contains matrix B put
            BSPMessage msg = bsp_recv(); // get row
            if ( msg != null )
            {
                int i = msg.getBSPTag(); // get row #
                Row a = (Row)msg.getData(); // get row data
                Row c = Matrix.rowMatMult( a, B ); // compute
                bsp_send( MAIN, i, c ); // send row back
            }
            bsp_unregister( "B" ); // we don't need B anymore
        }
        if ( bsp_sync() ) // *** superstep 2 ***
        {
            // main collects results, get it
            bsp_get( MAIN, "C", "C" ); // get C for next step
        }
        if ( bsp_sync() ) // *** superstep 3 ***
        {
            // local var C now contains result
            trace( "C = \n" + C ) // print C
        }
    }
}

```

Fig. 7. Bayanihan BSP code for matrix multiplication.

5 Results

By hiding the previously-exposed implementation details of barrier synchronization and process communication from the programmer, the new BSP programming interface makes it much easier to write Bayanihan applications than before. At present, we have successfully used it not only to rewrite existing applications in a more readable and maintainable SPMD style, but also to implement new programs and algorithms such as *Jacobi iteration*, whose more complex communication patterns were difficult to express in the old model.

Unfortunately, while it is now possible to *write* a much wider variety of parallel programs with Bayanihan, such programs may not necessarily yield useful speedups yet since the underlying implementation is still based on the master-worker model. For example, even though one can now use peer-to-peer style communication patterns in one's code, these communications are still sequentialized at the implementation level since they are all performed by the central server. Also, since all work objects are currently checkpointed at the end of each superstep (i.e., workers send back their *entire* process state to the server at each `bsp_sync()`), algorithms that keep large amounts of local state between supersteps will perform poorly as this state would be unnecessarily transmitted back and forth between the server and the worker machines.

Figure 8 shows speedup results from some tests using 11 200 MHz Pentium PCs running Windows NT 4.0 on a 10Mbit Ethernet network, with Sun's JDK 1.1.7 JVM on the server and Netscape 4.03 on the 10 clients. The Mandelbrot test is a BSP version of the original Bayanihan Mandelbrot demo running on a 400x400 pixel array divided into 256 square blocks, with an average depth of 2048 iterations/pixel. The matrix multiplication test is a variation of the sample code in Sect. 4.2 where the worker processes get 10 rows of A at a time to improve granularity, and do not get C from the main work. It was run using 700x700 `float` matrices. The Jacobi iteration test solves Poisson's equation on a 500x500 `double` array decomposed into a 10x10 2D grid, with an exchange of border cells and a barrier synchronization done at the end of each iteration.

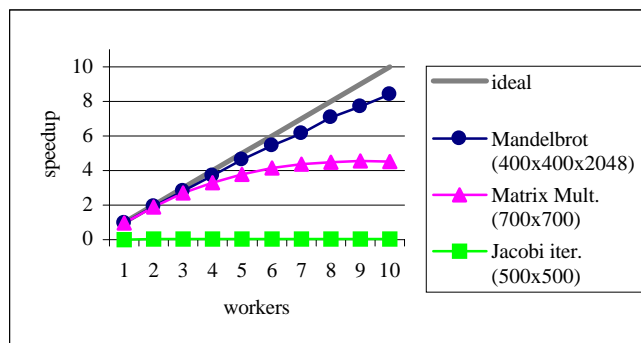


Fig. 8. Speedup measurements (relative to sequential version).

As shown, the “embarrassingly parallel” Mandelbrot demo performed and scaled reasonably well. In contrast, the much finer-grain Jacobi iteration test performed very poorly, being at best about 40 times slower than a pure sequential version. This is not so surprising, though, given that it takes less time for the server to compute an element than to send it to another processor. Although speedups are theoretically still possible if the array is very large and workers are allowed to keep “dirty” local state for a long time, we cannot expect to get any speedup in this case since work objects send all their elements back to the server at the end of each iteration. On a more positive note, the matrix multiplication test, which employs a master-worker-like communication pattern (although written as a message-passing program), performed much better than the Jacobi iteration test, though not as well as the Mandelbrot test. The limiting factor in this case was the time needed to send the matrix B to the worker clients. Since we currently do not support multicast protocols, the server has to send B to workers individually. Thus, the broadcast time grows linearly with the number of worker clients, and eventually limits speedup. In the future, it may be possible to improve performance by using a more communication-efficient algorithm. However, most such algorithms keep local state between supersteps, and will thus probably be inefficient in the current implementation.

These results show that at present, Bayanihan BSP is still best used with coarse-grain applications having master-worker style communication patterns, and more work needs to be done to improve performance and scalability in other types of applications. Meanwhile, however, simply allowing programmers to write much more complex parallel programs than before is already a major result in itself, and represents a significant step towards making Java-based volunteer computing systems useful for realistic applications.

6 Current Issues and Future Work

At present, we are looking into ways of improving the performance of Bayanihan BSP applications. Some ideas we are studying include:

Caching. Implementing a form of software-based caching can improve performance and scalability in several ways. In the matrix multiplication program in Sect. 4.2, for example, the matrix B is sent to all *virtual* processes. Without caching, this would cause B to be sent over the communication network potentially thousands of times. By adding a cache to each work engine as shown in Fig. 3 in Sect. 3.1, we only need to send B once for each *physical* processor. We have successfully used such a caching scheme in the experiment presented in Sect. 5, and are currently generalizing it to apply to other situations as well.

Less frequent checkpointing and lazy migration. As noted, checkpointing and migrating processes at *all* calls to `bsp_sync()` can lead to unnecessarily poor performance in programs where large amounts of local state persist between supersteps. We can improve efficiency by having `bsp_sync()` normally send back

communication requests only, and only occasionally send back the entire process state. We can also implement a *lazy* migration scheme that does not migrate a virtual process unless it is causing the system to slow down unnecessarily.

Peer-to-peer communication. Using the server for the global communication phase makes it easy for worker applets to communicate with each other despite the Java applet security restrictions that limit an applet to communicating only with its source host. Unfortunately, it can also be a serious and unnecessary sequential bottleneck in cases where such restrictions can be removed, such as within a trusted intranet. For these situations, we can implement peer-to-peer communication by modifying the work engine to send and accept communication requests to and from other work engines directly. We can still distribute the resulting worker code as an applet by either *signing* it, or asking volunteers to manually turn off their browser's applet network restrictions (Microsoft Internet Explorer 4.0, Sun's Hot Java browser, and Sun's Java plug-in [22], which works with Netscape, allow this). Implementing peer-to-peer communications should greatly improve the performance of parallel algorithms that depend on parallelizing communications as well as computation.

Fault-tolerance across supersteps. At present, our crash-tolerance and fault-tolerance mechanisms only work within a superstep. It is not clear yet how they can be extended if checkpoints are not performed each superstep. In such cases, a crash or fault may force a rollback to the previous checkpoint, possibly wasting work unnecessarily. It would be useful to develop a way to recover from a fault without having to do a full rollback.

Programming interface improvements. Aside from performance improvements, we are also studying some programming interface improvements such as the loop version of `bsp_cont()` mentioned in Sect. 4.1, tag-based retrieval of incoming messages, and saving and restoration of BSP variables in recursive functions.

7 Related Work

In recent years, an increasingly growing number of Java-based parallel computing systems have emerged [8, 1, 2]. Among these are systems with interfaces based on PVM, such as JPVM [9], and MPI, such as DOGMA [7]. Some of these, such as DOGMA and JavaParty [16], have already been successfully used in realistic parallel applications. Unfortunately, however, most of these systems use command-line Java *applications*, and thus still require some time and user expertise to set up. Making them use Java *applets* that can be run in a browser would be difficult or impossible not only because they require peer-to-peer communication not normally available to applets, but also because the semantics of the conventional message-passing models they use make it difficult to implement the adaptive load-balancing and fault-tolerance mechanisms required in the dynamic environment of applet-based volunteer systems.

A number of applet-based systems have also emerged, including Charlotte [3], Javelin [6], DAMPP [24], some RC5 cracking applications [11], and the original Bayanihan [17]. These are much easier to use than application-based systems since they allow a volunteer to join a computation by simply visiting a web page. Unfortunately, however, applet security restrictions and the need for adaptive parallelism and crash-tolerance seem to have so far limited applet-based systems to using ad hoc master-worker-based programming models that drastically limit the types of programs that one can write. Among applet-based systems, the most general and programmable one so far seems to be Charlotte [3]. Charlotte has a clean programming interface that provides transparent cache-coherent distributed shared memory between browser-based worker applets, while supporting adaptive parallelism in the form of eager scheduling. Its execution model is very similar to BSP in that computation is also divided by barrier synchronizations into superstep-like parallel blocks, and changes to distributed memory are not felt by other processors until after the next barrier. In addition, Charlotte has a fully-developed distributed caching mechanism that is still lacking in Bayanihan. Charlotte, however, requires defining a separate class for each superstep, and does not have message-passing functions.

The Bayanihan BSP interface bridges the two worlds of application- and applet-based parallel Java systems by allowing programmers to write programs using the flexible and easy-to-use remote memory and message-passing style found in peer-to-peer systems, while at the same time allowing these programs to be run in much more volunteer-friendly applet-based environments.

8 Summary and Conclusion

In this paper, we have shown how the BSP programming model can be implemented and used in Java-based volunteer computing systems. Our implementation makes use of existing adaptive parallelism and fault-tolerance mechanisms, such as eager scheduling, majority voting, and spot-checking, already developed in our Bayanihan volunteer computing framework, while providing a new programming interface with familiar and powerful methods for remote memory and message passing operations. At present, the performance and scalability of our implementation is limited in some applications, but is expected to improve with future work on mechanisms for lazy checkpointing and migration, caching, and peer-to-peer communication. Meanwhile, the BSP programming interface we have developed represents a significant improvement in programmability and flexibility over current programming interfaces for volunteer-based systems, and enables programmers to start porting and writing more realistic research applications for use with volunteer computing systems.

References

1. *Proc. ACM 1997 Workshop on Java for Science and Engineering Computation*. Las Vegas (1997) <http://www.npac.syr.edu/users/gcf/03/javaforcse/acmspecissue/latestpapers.html>

2. *Proc. ACM 1998 Workshop on Java for High-Performance Network Computing*. Palo Alto (1998). <http://www.cs.ucsb.edu/conferences/java98/>
3. Baratloo, A., Karaul, M., Kedem, Z., Wyckoff, P.: Charlotte: Metacomputing on the Web. *Proc. of the 9th International Conference on Parallel and Distributed Computing Systems*. (1996) <http://cs.nyu.edu/milan/charlotte/>
4. Beberg, A. L., Lawson, J., McNett, D.: <http://www.distributed.net/>
5. BSP Worldwide Home Page. <http://www.bsp-worldwide.org/>
6. Cappello, P., Christiansen, B. O., Ionescu, M. F., Neary, M. O., Schausser, K. E., Wu, D.: Javelin: Internet-Based Parallel Computing Using Java. *ACM Workshop on Java for Science and Engineering Computation*. (1997) <http://www.cs.ucsb.edu/research/superweb/>
7. DOGMA Home Page. <http://zodiac.cs.byu.edu/DOGMA/>
8. Fox, G.C. ed.: Special Issue on Java for Computational Science and Engineering – Simulation and Modeling. *Concurrency: Practice and Experience*. **9**(6) (1997).
9. Ferrari, A.: JPVM: Network Parallel Computing in Java. *Proc. ACM 1998 Workshop on Java for High-Performance Network Computing*. Palo Alto (1998). <http://www.cs.virginia.edu/~ajf2j/jpvm.html>
10. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.: PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallelism. MIT Press. (1994) <http://www.netlib.org/pvm3/book/pvm-book.html>
11. Gladychew, P., Patel, A., O'Mahony, D.: Cracking RC5 with Java applets. *Proc. ACM Workshop on Java for High-Performance Network Computing* (1998).
12. Gropp, W., Lusk, E., Skjellum, A.: Using MPI. MIT Press. (1994)
13. Hill, J.M.D., Donaldson, S.R., Lanfear, T.: Process Migration and Fault Tolerance of BSPLib Programs Running on Networks of Workstations. *Proc. Euro-Par'98. LNCS*, Vol. 1470. Springer, Berlin (1998) 80-91
14. Hirano, S.: HORB: Distributed Execution of Java Programs. *Proc. WWCA'97. LNCS*, Vol. 1274. Springer, Berlin (1997) 29-42 <http://www.horb.org/>
15. Nibhanupudi, M.V., Szymanski, B.K.: Adaptive Parallelism in the Bulk Synchronous Parallel model. *Proc. Euro-Par'96. LNCS*, Vol. 1124. Springer (1996)
16. Philippsen, M., Zenger, M.: JavaParty – Transparent Remote Objects in Java, in: *Concurrency: Practice and Experience*. **9**(11)(1997) 1125-1242
17. Sarmenta, L.F.G.: Bayanihan: Web-Based Volunteer Computing Using Java. *Proc. WWCA'98. LNCS*, Vol. 1368. Springer, Berlin (1998) 444-461
18. Sarmenta, L.F.G., Hirano, S., Ward, S.A.: Towards Bayanihan: Building an Extensible Framework for Volunteer Computing Using Java. *Proc. ACM Workshop on Java for High-Performance Network Computing* (1998).
19. Sarmenta, L.F.G., Hirano, S.: Building and Studying Web-Based Volunteer Computing Systems Using Java. *Future Generation Computer Systems*, Special Issue on Metacomputing (1999) <http://www.cag.lcs.mit.edu/bayanihan/>
20. Skillicorn, D., Hill, J.M.D., McColl, W.F.: Questions and answers about BSP. *Scientific Programming*. **6**(3) (1997) 249-274
21. Strumpfen, V., Ramkumar, B.: Portable Checkpointing for Heterogeneous Architectures. *Fault-Tolerant Parallel and Distributed Systems*. Kluwer Academic Press (1998) 73-92
22. Sun Microsystems: Java Home Page. <http://java.sun.com/>
23. Valiant, L.G.: A bridging model for parallel computation. *Communications of the ACM*. **33**(8) (1997) 103-111
24. Vanhelsuwe, L.: Create your own supercomputer with Java. *JavaWorld*. (Jan. 1997) <http://www.javaworld.com/jw-01-1997/jw-01-dampp.ibd.html>