Appears in the Proceedings of the 11th International Workshop on High−Level Parallel Programming Models and Supportive Environments (HIPS 2006), April, 2006.

Babylon v2.0:Middleware for Distributed, Parallel, and Mobile Java Applications

Willem van Heiningen[1], Tim Brecht[2], and Steve MacDonald[2]

[1] Integrative Biology
Hospital for Sick Children
Toronto, ON Canada
willem@sickkids.ca

[2]David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON Canada
{brecht, stevem}@uwaterloo.ca

Abstract

Babylon v2.0 is a collection of tools and services that provide a 100% Java compatible environment for developing, running and managing parallel, distributed and mobile Java applications. It incorporates features like object migration, asynchronous method invocation and remote class loading while providing an easy-to-use interface. Additionally, Babylon v2.0 enables Java applications to seamlessly create and interact with remote objects while protecting those objects from other applications by implementing access restrictions and separate name spaces.

This paper describes the most important programming features of the Babylon v2.0 system, using a heat diffusion example to show how they are used in practice. The potential cluster computing benefits of the system are demonstrated with experimental results which show that sequential Java applications can achieve significant performance benefits from using Babylon v2.0 to parallelize their work across a cluster of workstations.

1 Introduction

The Java language [7] has many features that facilitate distributed systems programming. Java's built-in security, threading and dynamic class loading support can greatly simplify the development of distributed applications. Furthermore, Java applications are compiled into a machine independent representation called bytecodes that can be run on any machine that runs the Java Virtual Machine (JVM). Java also supports Remote Method Invocation (RMI) which can hide much of the complexity of communication with objects residing in other JVMs, possibly on other machines.

Nevertheless, Java and RMI do not include support for many important features for distributed object programming such as dynamic remote object creation, asynchronous remote method invocation, remote object migration and remote object administration facilities.

Babylon v2.0 overcomes these limitations by providing programmers with Java classes and interfaces for remote object creation, interaction and administration. Babylon v2.0 contributes several new features and approaches in the area of Java-based distributed and parallel systems without special language extensions, preprocessors or compilers.

- Babylon v2.0 uses Java's dynamic proxy objects to provide transparent access to remote worker objects. Clients use these dynamic proxies to call methods on workers using standard Java method invocation syntax.
- Babylon v2.0 implements an asynchronous method invocation technique based on asynchronous tickets. Method invocations using asynchronous tickets are syntactically identical to local method invocations but are executed asynchronously.
- Babylon v2.0 provides two forms of remote object creation. Clients can create a new remote object instance based only on a programmer-specified class name or take an existing locally-created object and turn it into a remote object.

Babylon v2.0 provides all of these features while maintaining 100% Java compatibility and can be used on any platform that supports the JVM. We believe that the performance results and the combination of features provided in Babylon v2.0 make it a powerful system for distributed application developers.

2 Related Work

There are currently a large number of commercial and academic Java-based distributed computing projects in various stages of development. The objectives and underlying technologies of each of these projects vary significantly. Some focus on the emerging grid, while others are designed for very specialized groups of computational problems. Babylon v2.0 is designed to handle a variety of parallel, distributed and mobile Java computations.

The original implementation of Babylon [12], referred to here as Babylon v1.0, was an important initiative to make distributed computing resources more widely available to developers. Much of the design for Babylon v1.0 came from experiences with Ajents [13] and ParaWeb [4]. Babylon v1.0 builds on these systems to provide mechanisms for remote object creation, migration and remote I/O, but lacks a flexible mechanism to use and interact with remote objects. Babylon v1.0 suffers from deficiencies in key areas such as remote class loading and object migration. However, the most serious drawback of Babylon v1.0 is its reliance on a non-standard remote method invocation interface that is awkward to use and error-prone because it prevents normal compile-time method invocation checks.

Several other existing systems, like JavaParty [9] and Java/DSM [21], use a modified JVM or special preprocessors and language extensions to implement distributed Java frameworks. Babylon v2.0 provides support for distributed programming without requiring a modified JVM implementation, new keywords or other custom language extensions.

Others systems, such as Javelin [17], Charlotte [3] and Ninflet [18], only support distributed applications that can be formulated as master-worker or branch-and-bound computational problems. Babylon v2.0 does not limit itself to a particular computational model and provides support for general remote object interaction using standard Java method invocation syntax. As a result, distributed applications that require more complex object interactions, such as the mesh computation-based heat diffusion application described in Section 3 and used in our experiments, can be written using Babylon v2.0. In contrast, this program could not have been implemented easily using these systems.

ProActive [2] provides similar services to Babylon v2.0, using similar techniques that do not require JVM changes or external tools. As well, it provides a group communication abstraction. ProActive generates proxies for remote objects at runtime using bytecode engineering libraries as it predates Java's dynamic proxies. This capability is used to create new remote objects and export local ones. ProActive can execute remote methods synchronously or asynchronously, but the choice is made automatically by the system based on the method return type and presence of checked exceptions. Asynchronous calls return future objects [8], which are proxies that resolve to a final value on a method call and transparently implement wait-by-necessity. All proxy classes generated by ProActive are subclasses of the originals, overriding public methods. This provides polymorphism between local and remote objects and obviates the need for interfaces, but limits the classes that can be used to create remote objects or used as return types in methods. Final classes and methods, including arrays, cannot be used as they cannot be subclassed or overridden. As well, a remote object can only run one method at a time, which can make some applications difficult to implement efficiently.

Another important feature of Babylon v2.0 is the use of separate name spaces on servers for clients. This is a crucial feature in a multi-user environment where several different clients may be running remote objects on a single server at the same time. Many existing systems such as ProActive [2], JavaParty [9] and the initial version of Babylon [12] do not provide separate class name spaces for clients. In these systems, servers must be restarted every time a client changes any of his/her classes and no two clients can ever use classes with the same name. In addition to supporting multiple class name spaces, Babylon v2.0 also provides access restrictions for remote objects based on context information transmitted with remote method invocations.

3 Programming in Babylon v2.0

To demonstrate some of the important features of Babylon v2.0, we briefly describe the implementation of a heat diffusion application.

The heat diffusion application simulates heat transfer across a two-dimensional surface over time. In this simulation, the surface is discretized into an $M \times M$ 2D array, T, and the Jacobi iterative method [11] is used to compute the final temperature distribution of the surface. The array is initialized to an even temperature distribution and a constant heat source is applied to each edge of the array. Each iteration simulates heat diffusion across the surface over a small period of time. At each iteration, i, the value of every cell in the array is recomputed to be the average value of its four neighbouring cells using data from iteration $i - 1$.

In our implementation, N worker objects are used to compute the temperature of a surface after 100 iterations. We use a decomposition strategy to partition the original array into N blocks each consisting of $\frac{M}{N}$ contiguous rows. Each block is transmitted over the network to a worker object running on a remote Babylon server. Two block edges must be exchanged between neighbouring worker objects at each iteration. Once these data points have been exchanged, each worker can compute the updated temperature for all the cells in its block for the current iteration.

Selected portions of the heat diffusion code appear in Figures 1 and 2. Figure 1 creates worker objects, initializes them, and performs the computation. The code starts by creating the remote worker objects of type `GridDiffuserImpl`, one per processor, using the `remoteNew()` method in line 9. The first argument is the class of the worker implementation, followed by the class exporting the client interface. The third argument is a user-defined name that is registered with the Babylon lookup service so clients can find remote objects. The last argument is a Java Archive (JAR) file containing the code and other files needed for the `GridDiffuserImpl` since we do not assume that remote servers share a file system. The archive is sent to the machine on which the new remote object is created. This process can be optimized by specifying a JAR file as the second argument to the call to `Babylon.-initApplication()` on line 3. This forwards the JAR file to Babylon servers at application startup, so `remote-New()` need not supply the archive. The JAR file is used as a convenient packaging mechanism; future versions of Babylon may use the remote class loading capabilities in Java, and may optimize communication by caching class information sent to other Babylon servers so it need not be transmitted multiple times [10].

By default, the Babylon scheduler places the new object on the first available server, though the user can request a specific machine. The Babylon scheduler is a distinguished process that assigns objects to servers and provides a worker object directory that can be used to look up references to live worker objects. Details on the scheduler and Babylon architecture can be found in [20].

The `remoteNew()` call returns a new dynamic Java proxy object that can be used to call client methods on the new remote worker. Method calls made using this proxy are executed synchronously. The proxy supports the methods declared in the interface supplied as the second argument. The behaviour of these proxies is identical to their `rmic`-generated equivalents. Later, we will show how we exploit dynamic proxies to support other Babylon v2.0 features.

Once the worker objects have been created, each must have a reference to neighbouring workers so they can exchange edges in each iteration. These remote references are set in the `setAdjacencies()` method (line 15) using a remote mutator method defined for the diffusion workers.

Starting the computation requires each worker object to start executing the `diffuse()` method, which causes each worker to iteratively execute the heat diffusion computation on its portion of the surface. This code is shown in the loop starting at line 20 in Figure 1. The `diffuse()` method must be called asynchronously to allow each worker to compute independently, but there must also be a way to determine when the workers have completed so results can be gathered. Babylon provides an asynchronous method invocation mechanism called an *asynchronous ticket*. An asynchronous ticket is a special dynamic Java proxy that can be used to execute a single remote method in a separate thread. By using dynamic Java proxies, the asynchronous method call still looks like a normal Java method call, as shown on line 25. No special syntax or library calls are needed for asynchronous invocations. The loop on line 20 partitions the surface, creates one ticket for each worker, and uses that ticket to call the `diffuse()` method. At the end of the loop, all workers are executing the computation. Any additional code that can run asynchronously with this computation can be placed after the loop.

Synchronization with an asynchronous ticket is accomplished using the `getResult()` method in the `Asynch-Ticket` class in Babylon v2.0 (line 32). After being used as a proxy for making an asynchronous method call, the ticket becomes a form of *future* representing the outstanding method results [8]; the `getResult()` method blocks until the worker finishes executing the method invoked with the ticket and returns its results. Any exceptions thrown during the execution of the method are raised at this point, and should be caught and handled. The worker results are combined into the global result.

The code in Figure 2 shows the main execution loop for each worker in the heat diffusion computation. One important characteristic of Java RMI is that each method invoked on a remote object is executed in a new thread at the server. This feature is used in the implementation of the exchange of remote edges in line 11. This method uses the synchronous RMI feature in Babylon to request edges from the data held by adjacent workers. Synchronous RMI in Babylon is built using Java RMI but with dynamic proxies rather than the proxies that would normally be generated by the `rmic` stub compiler. The need for dynamic proxies will be explained later in this section.

One difficulty with the exchange of edges stems from the use of Jacobi iteration in the computation. The values used to compute the diffusion in iteration i are those obtained from iteration $i-1$. A worker requesting remote edges from an adjacent worker must wait until that worker completes its current iteration. This synchronization is accomplished using the thread synchronization facilities in Java. The worker calls an accessor to retrieve the edge values. This accessor blocks the calling thread until the edges are updated. Since the method call is synchronous, the remote client is also blocked. Once the current iteration is complete, these blocked threads are woken (on line 22 of Figure 2) and return the edge data, allowing adjacent workers to proceed with their next iteration.

This application shows many features of Babylon v2.0. In particular, asynchronous tickets are a novel feature supporting asynchronous method invocations. Asynchronous tickets permit synchronous and asynchronous RMI to coexist in the same program, where both forms of *remote* invocation are indistinguishable from *local* method invocations. The only difference is in the type of dynamic proxy used in the call. In contrast, many other systems require additional keywords to distinguish asynchronous calls. The creation of asynchronous tickets is enabled by the addition of dynamic proxies in Java; it is possible to create a new proxy object for an interface at runtime rather than having to generate proxies statically using stub compilers like `rmic`. Different dynamic proxies can handle method invocations differently, but the use of these proxies still provides the basic RMI abstraction of hiding remote calls behind a local object.

Babylon v2.0 explicitly distinguishes between synchronous and asynchronous method calls using these tick-

```
public void startHeatDiffusion() {
  try {
    Babylon.initApplication("schedulerHostName", null, null);
  } catch (Exception e) { /* handle exception */ }
  GridDiffuser gd[] = new GridDiffuser[nprocs];
  Grid grid = new Grid(matrix_size, matrix_size);
  try {
     for (i = 0; i < nprocs; i++) {
       gd[i] = (GridDiffuser) Babylon.remoteNew(GridDiffuserImpl.class,  GridDiffuser.class,
                                                "GridSection" + i, "GridDiffuser.jar");
     }
  } catch (Exception e) { /* handle exception */ }

  // Provide references to adjacent grid workers for edge exchange.
  setAdjacencies(gd, nprocs);

  // Start diffusing... first prepare space for the asynchronous tickets.
  GridDiffuser gd_asynch[] = new GridDiffuser[nprocs];

  for (i = 0; i < nprocs; i++) {
    grid_section[i] = grid.getMyRows(i, nprocs);

    // Calling diffuse() asynchronously on each worker object.
    gd_asynch[i] = (GridDiffuser) AsynchTicket.newTicket(gd[i]);
    gd_asynch[i].diffuse(grid_section[i], 0, iterations);
  }

  // Now retrieve the results from each worker.
  float sub_result[][][] = new float[nprocs][][];
  for (i = 0; i < nprocs; i++) {
    try {
      sub_result[i] = (float[][])AsynchTicket.getResult(gd_asynch[i]);
    } catch (RemoteExecException t) { /* handle exception */ }
  }
}
```

Figure 1. Babylon v2.0 Grid Diffusion Example - Starting the Computation.

```
public void diffuse(float[][] partition, int endIteration) {
  float[][] temp;
  float[][] srcGrid = initLocalGrid(partition);
  float[][] targetGrid = new float[srcGrid.length][srcGrid[0].length];
  int current_Iteration = 0;
  while (current_iteration < end_iteration) {

    // Make sure we have the updated values for the remote edges
    // before we compute the diffusion for this iteration.
    // Block until other worker completes last iteration if necessary.
    getRemoteEdgesFromAdjacentWorkers(srcGrid);

    // Compute the temperature diffusion for this iteration.
    diffuse(targetGrid, srcGrid);

    // We’re done. Swap the source and target grid.
    float[][] temp = srcGrid;
    srcGrid = targetGrid;
    targetGrid = temp;

    // Update the local edge array and the iteration.  Wake any waiting threads.
    setLocalEdges(++current_iteration);
  }
}
```

Figure 2. Babylon v2.0 Grid Diffusion Example - Main Worker Code.

```
gd[i] = (GridDiffuser)
        Babylon.export(new GridDiffuserImpl(),
                       GridDiffuser.class,
                       "GridDiffuser.jar");
```

Figure 3. Exporting Local Objects.

ets. This feature provides the user with control over how methods are invoked, rather than using implicit rules like ProActive. Further, it provides control over synchronization with asynchronous methods. Specifically, ProActive asynchronously executes methods with a return type of `void` (unless they throw a checked exception), but these methods produce no future and so synchronization is not possible. For methods with side effects on the state of a remote object, these semantics may not be desired. Although additional code is needed, Babylon v2.0 allows this synchronization.

The remote object creation and method invocation facilities in Babylon v2.0 are a significant improvement over those in Babylon v1.0 and Java RMI. Babylon v1.0 uses strings for class names, method names and object types, preventing compile-time checks for class existence, and the number and types of parameters used in remote method invocations. Java RMI cannot create remote objects directly. Instead the object servers must be created with factory objects (already running on the remote host), which must then be used to create new remote objects. Since Java 1.2, it is possible to create `Activatable` objects, which permit remote objects to be started on demand. However, creating such objects requires the application developer take several additional steps including the creation of a setup program to create information about the activatable object and register it with the RMI registry.

In addition to offering remote object creation, Babylon v2.0 offers a second method for creating remote objects called *exporting*. Exporting takes a local object and uses it to create and initialize a new remote object that is moved to an available server. Figure 3 shows an example of exporting. This is the object creation code from line 9 in Figure 1, rewritten to export a locally-created instance of `GridDiffuserImpl`. The `export()` method takes three parameters: the local object to be exported, an interface implemented by the class of that object (used to construct a dynamic Java proxy for the new remote object), and the JAR file containing the required class files. Like `remoteNew()`, the JAR file can instead be specified in the call to `Babylon.initApplication()` and removed from calls to `export()`.

The only requirement for exporting a local object is that its class must implement a Java interface that declares client methods and the `Serializable` interface. In contrast, to create remote objects using Java RMI a worker interface must extend the `Remote` interface, and the worker class usually extends the `UnicastRemoteObject` class or must call its `exportObject()` method for all new objects. In Babylon v2.0 neither of these requirements are necessary to export a local object or create a new remote object because both use dynamic Java proxies.

Exporting a local object creates a dynamic proxy based on the interface implemented by the class, which serves as the RMI stub. Babylon v2.0 must use dynamic proxies (as opposed to `rmic`-generated proxy stubs) for two reasons. First, the `rmic` compiler cannot generate correct stubs for worker object classes since these classes do not adhere to the Java RMI conventions. Second, the stub compiler cannot know which classes will be exported in advance and would have to generate stubs for every class, which is not feasible. In particular, it is possible to export objects from classes in the standard Java class library, so the set of stubs that may be needed is simply too large. Remote object creation uses dynamic proxies for the same reasons.

4 Additional Babylon Features

We now briefly describe some of the interesting features of Babylon v2.0 that were not used in the example program.

4.1 Remote Class Loading and Name Spaces

Unlike most programming languages, Java provides a very flexible class loading mechanism that only finds and loads classes when they are accessed [14]. Normally, the virtual machine looks for class data in the form of “class files” that reside in the file system. However, developers can customize how the virtual machine finds and loads classes by implementing their own custom class loaders.

Babylon v2.0 uses custom class loaders to load worker object classes over the network. The class files for a worker object must be placed in a JAR file whose location is specified as an argument to the `Babylon.remoteNew()` or `Babylon.export()` methods when the client creates the remote worker object. The JAR file is transmitted to the target server along with the remote object creation request. As a result, Babylon v2.0 servers do not need local file system access to the class files of the worker object and clients can run their worker objects on remote servers without requiring login access to the server machine.

Babylon v2.0 servers create a new instance of a custom class loader for each client. Each class loader manages its own name space. Providing separate name spaces for each client solves many of the class loading issues experienced by systems such as ProActive, JavaParty and Babylon v1.0. Some of the advantages of Babylon v2.0 class loading are:

- Different clients can safely use identical names for their classes without causing naming conflicts.

- To load and use the new class definitions, clients can change the classes and simply restart their application.
- When a client application finishes, the class loader and classes loaded by the client can be garbage collected.
- Clients no longer share a single class name space and, consequently, no longer have access to each other's classes (unless permissions are set to permit sharing).

4.2 Object Migration

Another key feature of Babylon v2.0 is the ability to freely move remote objects from one Babylon server to another. Object mobility can be used to support dynamic load balancing (i.e., move a remote object from a heavily loaded server to a lightly loaded server), fault-tolerance (i.e., move a remote object from a faulty server to a stable server) or to exploit server locality (i.e., move a remote object to a server with lower communication latency).

Worker object migration takes a snapshot of a worker object's data state, known as a checkpoint, and transmits this checkpoint to a new Babylon v2.0 server. Consequently, only objects that are serializable can be migrated.

Babylon v2.0 stores the most current worker object location information in a worker object registry in the Babylon scheduler. In addition, *location forwarding* is used [6]. Forwarding information for a mobile object is stored at each of the object's former locations, essentially creating a chain leading from the object's original location to its current location. Stale worker references are updated transparently using this information. The combination of these two mechanisms ensures calls to a migrated object cannot be lost.

Three types of migration are supported by Babylon v2.0. Idle migration takes an idle worker object (one that is not executing any methods) and moves it to a new server. If a worker object is actively executing one or more methods, either delayed or safe-point migration can be used. Delayed migration prevents new method invocations from starting while allowing in-progress methods to finish executing. Once all the in-progress methods finish executing the object becomes idle and migration can occur.

If the object is not doing any computation at the time of the migration call the object is immediately migrated. If the object is actively computing at the time of the call the migration will be delayed until the method being invoked has completed. Finally, safe-point migration uses a checkpointing and rollback protocol to perform migration at programmer specified safe migration points. Safe-point migration is only supported for worker objects running in *safe mode*. Firstly, safe mode enforces single threaded execution for a worker object and forces concurrent method invocation requests to be executed sequentially in the order of their arrival. Secondly, a checkpoint of the worker object state is created prior to the start of each method invocation. This ensures that the worker object always has a recent checkpoint which can be used if safe-point migration is requested.

Worker objects that require the ability to be immediately migrated using safe-point migration must include calls to `Babylon.setMigrationPoint()` in their worker object code at points where safe-point migration can safely be performed. Normally, `Babylon.setMigrationPoint()` does nothing and simply returns. However, if safe-point migration is pending, this method will throw a `babylon.core.BabylonThreadDeath` object as an exception. Unless caught by the worker object code, this exception will propagate up to the server. When the server catches the exception, it knows that the worker thread has been stopped and that the worker object can safely be migrated. A worker object can also catch the `BabylonThreadDeath` exception if it needs to perform any cleanup tasks before migration occurs, provided the exception is rethrown when cleanup tasks complete. Another approach would be to allow a cleanup object to be provided in the call to `setMigrationPoint()`, which could provide a method to perform the required cleanup if migration is pending. This would eliminate the possibility of incorrectly handling the `BabylonThreadDeath` exception.

This approach provides a 100% Java compatible and thread safe mechanism for stopping worker threads and gives workers an opportunity to perform cleanup recovery tasks to defend against the checkpoint consistency problem [5] before migration occurs. For instance, a worker could catch the `BabylonThreadDeath` exception after setting a safe migration point and use the handler to close I/O connections or undo an operation that affects an external component. The drawback is that the source code of the worker object must be available so that calls to `setMigrationPoint()` can be added at safe migration points.

4.3 Remote I/O

Babylon v2.0 includes a mechanism for performing I/O operations with worker objects using server-side callbacks. This technique works by creating an I/O server inside the client which can be remotely referenced and used by a worker object using RMI. Babylon v2.0 provides wrapper classes for many standard Java I/O classes. These wrapper classes are essentially RMI servers with interfaces that more or less match their standard Java I/O counterparts. A worker object that needs to perform console, file or socket I/O can obtain a remote reference to the appropriate wrapper object and use it instead of the normal Java I/O class.

5 System Evaluation

In this section, we examine the performance of parallel matrix multiplication and heat diffusion benchmarks. The experiments were run on a cluster of 8 dual CPU (500 MHz Pentium III) PCs running Red Hat Linux 7.1 connected via a 100 Mbps switch. Each of these machines contains 256 MB of RAM and is running the J2SE platform, version 1.3.1_02. Babylon v2.0 also works on newer JVMs since it does not require additional tools or extensions to the language or runtime. Babylon v2.0 only requires JVM support for dynamic proxies, which is standard starting from version 1.3.

Matrix multiplication is often used as a test application for distributed systems because it can be implemented using the master-worker design pattern. In other words, a matrix multiplication problem can be divided into sub problems that can be solved independently by worker objects. The participating workers do not need to communicate with each other to synchronize or share data. A distributed version of the matrix multiplication benchmark is used to evaluate the performance of a typical master-worker computation using Babylon v2.0. The sequential baseline is a simple, standalone matrix multiplication application (without RMI or domain partitioning) that uses the same core matrix multiplication algorithm as the distributed implementation.

The heat diffusion benchmark was used to evaluate the performance of a communication-intensive Babylon v2.0 application that could not have been efficiently realized using the master-worker computation model. As a result, it could not have been easily written or executed using many existing distributed systems (e.g., Babylon v1.0, Charlotte, Javelin, and Ninflet) since they do not support the more general programming model required for this type of application. A ProActive version would suffer from the limitation that only one method can be running in a remote object at a given time. This means the edges cannot be exchanged as shown in Figure 2, since the remote accessor method cannot execute as long as the `diffuse()` method runs. Instead, each iteration of the diffusion must be a separate remote method call, making the code awkward and increasing communication costs. The sequential baseline is a simple, standalone diffusion application (without RMI or domain partitioning) that uses the Jacobi iterative algorithm to compute the final temperature distribution across the surface.

The speedup results for the matrix multiplication and heat diffusion benchmarks are summarized in Figure 4. The straight dotted line represents perfect speedup and measured speedups are relative to a sequential implementation.

The speedups obtained for the 2048×2048 matrix multiplication experiments are quite good until 16 worker objects are used. With a 2048×2048 matrix, the granularity of the computation is large enough to produce significant speedups with four or eight workers. For instance, using four workers we were able to achieve a speedup of 3.8 and with eight workers, a speedup of 7.1. However, as the number of workers increases, so does the cost of distributing the matrix data to the workers. This overhead becomes especially

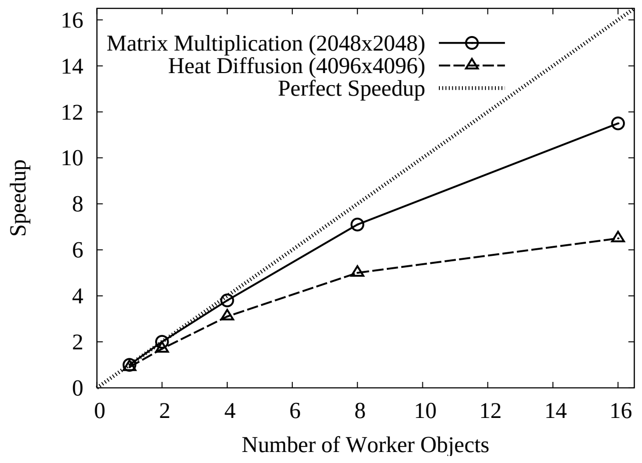

Figure 4. System Evaluation Results

evident in configurations that use 16 workers. The largest speedup we were able to obtain with 16 workers was 11.5.

To determine why the speedup is not better for 16 worker objects, we conducted a detailed analysis of data distribution costs incurred by the multiplication of two 2048×2048 matrices. The communication costs of distributing the matrix data to the worker objects increases proportionally with N. This is primarily because the entire matrix B must be transmitted to each worker object participating in the computation. For example, if 16 workers are participating in the computation, the master program will need to send all of matrix B and a portion of matrix A a total of 16 times at the start of the computation so that each worker has the required data. For large matrices, this data transmission can have a considerable impact on the resulting speedup.

The total amount of network traffic generated with N worker objects for a b byte matrix (b = 16.0 MB for a 2048×2048 matrix) is $(A + B + C) \times N$, where $A = b/N$, $B = b$ and $C = b/N$. In the case of 16 worker objects 288 MB of data is transmitted. The process of distributing this large amount of data from the master to the workers causes the algorithm efficiency to drop significantly for configurations using 16 workers. Because the transmission of these matrices is done sequentially, the speedups obtained can be explained using Amdahl's law [1]. In this case an upper bound on the speedup is 12.3 which is quite close to the actual speedup of 11.5. A more detailed analysis can be found in [20].

Although the speedup values obtained in the heat diffusion benchmark are smaller than those obtained in the matrix multiplication benchmark, the results are still promising and indicate that speedup can be achieved despite the communication-intensive nature of the problem. In fact, similar experiments in [16], [15] and [19] yield speedups in the range of 2.5 to 4.9 on eight processors and 3.5 to 6.5 on 16 processors. Speedups obtained using Babylon v2.0 are as high as 5.0 using 8 worker objects and 6.5 using 16 worker objects. This suggests that Babylon v2.0 can be just as effective at running communication-intensive applications as other systems.

By analyzing the heat diffusion experiments based on the amount of data communicated, the time required to sequentially transmit that data (in this case some communication occurs in parallel and we only count the sequential portion of the communication), and the sequential execution time, we are again able to obtain a bound on the speedup using Amdahl's law. In this case the bound is 7.3 which again is not much higher than the actual speedup of 6.5. A more detailed analysis can be found in [20].

6 Conclusions

Babylon v2.0 incorporates features like remote object migration, remote object access restrictions, separate name spaces for clients, and dynamic class loading while providing an easy-to-use interface that works seamlessly with existing Java code. Worker objects created using Babylon v2.0 can be accessed transparently using dynamic proxy objects. Babylon v2.0 also introduces an asynchronous method invocation technique based on proxy objects called asynchronous tickets. The result is a powerful system that gives developers the necessary tools and services for building powerful cluster computing applications.

Performance evaluation results are also promising and indicate that applications can use Babylon v2.0 to distribute objects over multiple hosts to execute efficiently in parallel. Experiments show that reasonable speedups can be obtained for simple master-worker applications (e.g., matrix multiplication) and for more complicated and communication-intensive applications (e.g., heat diffusion). Our experiments demonstrate that Babylon v2.0 can be used to effectively build and run clustered computing applications.

Acknowledgments

We gratefully acknowledge Morgan Stanley Dean Witter, the Ontario Research and Development Challenge Fund, and the National Sciences and Engineering Research Council of Canada for financial support for portions of this project. This paper has also benefited from the comments and suggestions of the anonymous reviewers.

References

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. *Proc. AFIPS 1967 Joint Computer Conference*, 30:483–485, 1967.

[2] I. Attali *et al*. A step toward automatic distribution of Java programs. In *4th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems*, pages 141-161, 2000.

[3] A. Baratloo *et al.*Charlotte: Metacomputing on the Web. *Future Generation Computer Systems*, 15(5):559–570, 1999.

[4] T. Brecht *et al.*ParaWeb: Towards world-wide supercomputing. In *Proc. 7th ACM SIGOPS European Workshop*, pages 181–188, 1996.

[5] O. P. Damani and V. K. Garg. How to recover efficiently and asynchronously when optimism fails. In *IEEE Intl Conf. on Distributed Computing Systems*, pages 108–115, 1996.

[6] R. J. Fowler. *Decentralized Object Finding Using Forwarding Addresses*. PhD thesis, University of Washington, Seattle, Washington, December 1985. (Department of Computer Science Technical Report TR85-12-1).

[7] J. Gosling *et al*. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

[8] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[9] B. Haumacher and M. Philippsen. Exploiting object locality in JavaParty, a distributed computing environment for workstation clusters. In *Proc. 9th Workshop on Compilers for Parallel Computers*, pages 83–94, June 2001.

[10] F. Huet *et al.*A high performance java middleware with a real application. In *Proc. Supercomputing Conference*, 2004.

[11] F. P. Incropera and D. P. DeWitt. *Introduction to Heat Transfer, Fourth Edition*. John Wiley and Sons, Inc., August 2001.

[12] M. Izatt. Babylon: A Java-based distributed object environment. Master's thesis, York University, Toronto, July 2000.

[13] M. Izatt *et al*. Ajents: Towards an environment for parallel, distributed and mobile Java applications. *Concurrency: Practice and Experience*, 12(8):667–685, 2000.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison Wesley, 1999.

[15] S. MacDonald *et al.* From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1683, 2002.

[16] S. Markus *et al*. Performance evaluation of MPI implementations using the parallel ELLPACK PSE. In *The Second MPI Developer's Conference*, pages 162–169, 1996.

[17] M. O. Neary *et al.*Javelin 2.0: Java-based parallel computing on the Internet. In *6th Intl. European Parallel Computing Conf.*, volume 1900 of *LNCS*, pages 1231–1238, 2000.

[18] H. Takagi *et al.*Ninflet: A migratable parallel objects framework using Java. In *1998 Workshop on Java for High-Performance Network Computing*, pages 151–159, 1998.

[19] K. Tan *et al*. Using generative design patterns to generate parallel code for a distributed memory environment. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 203–215, June 2003.

[20] W. van Heiningen. Babylon v2.0: Support for distributed parallel and mobile Java applications. Master's thesis, University of Waterloo, Waterloo, Ontario, August 2003.

[21] W. M. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997.