Bridging the Gap between Distributed Shared Memory and Message Passing

Holger Karl*
karl@cs.nyu.edu
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University

Abstract

Using Java for high-performance distributed computing aggravates a well-known problem: the choice between efficient message passing environments and more convenient Distributed Shared Memory systems which often provide additional functionalities like adaptive parallelism or fault tolerance—with the latter being imperative for Web-based computing.

This paper proposes an extension to the DSM-based *Charlotte* system that incorporates advantages from both approaches. Annotations are used to describe the data dependencies of parallel routines. With this information, the runtime system can improve the communication efficiency while still guaranteeing the correctness of the shared memory semantics. If the correctness of these annotations can be relied upon, additional optimizations are possible, ultimately sharing primitive data types such as `int` across a network, making the overhead associated with accessing and sharing objects unnecessary. In this case, the annotations can be regarded as a compact representation of message passing semantics. Thus, a program's efficiency can be improved by a a step-by-step incorporation of semantic knowledge. The possibility to freely mix and to easily switch between unannotated code, annotated code and shared primitive data types entails a big flexibility for the programmer.

A number of measurements show significant performance improvements for annotations and annotation-based shared primitive types.

*This work was done while the author was on leave from Humboldt-Universität Berlin, Graduiertenkolleg "Kommunikationsbasierte Systeme."

1 Introduction

The Internet has been described as a potential source for an immense computational capacity. A vast number of often idle machines are connected and provide—in principle—enormous resources to tackle large problems. But distributed computing in the Internet is faced with a number of challenges that have prevented this vision from realization so far. Among these obstacles are heterogeneity, security concerns, the need to install programs on remote computers, high communication latencies, and the inherent unreliability of remote machines and the Internet itself.

The Java programming language successfully addresses the heterogeneity and security concerns and removes the need to install programs remotely other than standard execution environments (typically, a Java-enabled browser). This makes Java a prime choice for building environments to execute parallel programs distributed over the Internet or, more precisely, the World Wide Web.

A number of research efforts use Java to build such an environment. Some of them use message-passing interfaces, others provide Distributed Shared Memory (DSM) semantics (cp. Section 2 for details). *Charlotte* [4] is such a DSM system—it uses a reliable parallel machine as programming model and the runtime system implements this model on top of unreliable machines. Charlotte provides an object-based shared memory for objects of certain distributed classes. Main advantages of Charlotte are easy programmability, fault tolerance with regard to crash faults of workers, and adaptive parallelism that makes use of slow machines. Fault tolerance in particular can be regarded as indispensable for a parallel system using the

World Wide Web. Additionally, Charlotte is completely implemented in Pure Java and runs on standard Java virtual machines.

But compared to simpler message-passing systems, Charlotte pays a high overhead for maintaining correct memory semantics. In a message-passing system on the other hand, this overhead is avoided by having the programmer provide precise information which data is transferred where and when—which is automated by a DSM system. Additionally, it is a difficult task for a programmer to provide Charlotte-like capabilities using only message-passing primitives. A programmer finds himself thus faced with the difficult choice between easy programmability and automatic fault tolerance provided by a DSM system like Charlotte and high performance obtainable by using simpler message-passing abstractions. Therefore, a solution is needed that maintains Charlotte's advantages for the programmer and improves its efficiency to be competitive with message-passing systems; of course, such a solution should be implemented in Pure Java as well. The need for such a solution is particularly felt in a Web-based environment using Java: high latencies and the unavailability of hardware support for detecting memory access make low-overhead solutions necessary, the complexity of Web-based network computing calls for higher, fault-tolerant programming models.

This paper presents an annotation-based solution that bridges the gap between Charlotte's DSM semantics and message-passing systems by allowing stepwise refinements of Charlotte programs. In a first step, a Charlotte-program can be enhanced with annotations that describe the data requirements of routines (Charlotte's unit of parallel execution). The runtime system can use this information to improve the communication efficiency; the standard Charlotte distributed classes guarantee the correctness of the computation even if the annotations are wrong. In a second step, after the annotations' correctness has been established, efficiency can be further improved by removing the consistency checks for shared data. In a third step, primitive data types can be used as a basis for sharing, additionally foregoing the overhead of accessing objects. This combines efficient communication and direct data access with Charlotte's advantages without the programmer having to worry about low-level communication issues. This gradual incorporation of semantic knowledge closes the gap between the programmability advantages of DSM and the communication efficiency of message-passing programs. Additionally, it is also conceivable to generate annotations automatically be means of a data flow analysis within a compiler.

The next section presents related work. Section 3 introduces the annotation-based extensions for Charlotte and Section 4 gives experimental results for the various extensions—both sections use matrix multiplication as a common example. Section 5 discusses possible extensions of this work and Section 6 gives some conclusions.

2 Related Work

Two areas of distributed computing are of interest: Work related to using Java for parallel or distributed computing and more general research into aspects of DSM systems, in particular the use of annotations for increased efficiency.

A number of recent projects use Java as an implementation platform for distributed computing. A PVM-like interface for Java is described in [8]. ATLAS [2] extends Cilk's technologies, e.g., hierarchical work stealing, and integrates them into Java. Unlike Charlotte, ATLAS needs daemon processes as compute servers and also makes use of native code. JavaParty [13] transparently adds remote objects to Java by introducing a new keyword `remote` that is handled by a preprocessor. JavaParty mainly targets clusters of workstations. The programmer or compiler generate code to guide data distribution and migration. JavaParty programs are very similar to Java programs; their efficiency is comparable to RMI-based implementations. The ParaWeb project [5] is concerned with providing an infrastructure for seamless access to heterogeneous computing resources. A library can be used for explicit message-passing programs or, with a modified Java Virtual Machine, threads can run remotely and are presented with the illusion of a single, shared memory. Similarly, Java/DSM [16] suggests implementing a modified Java Virtual Machine on top of TreadMarks using a distributed garbage collector. In [9] it is argued that integrating the global pointer and asynchronous remote method invocation concept (constituting a global name space) from the Nexus library is beneficial for web-based supercomputing. Javelin [6], similar to Charlotte in that it allows standard Web-browsers to be used, provides brokering functionalities for computational resources and adds a layer supporting the implementation of parallel programming models in Java.

Research into DSM in general can be broadly classified as using hardware support or being an all-software approach. Among the former, Munin [7] is interesting as it uses annotations to express expected access patterns for data (e.g., "producer-consumer"), thereby allowing the run-time system to choose an appropriate consistency protocol. Since Java does not allow access to low-level system concepts such as memory pages and memory protection, all-software solutions are more relevant. A number of such systems have been proposed; examples include object-based systems like Orca [1] and Aurora [11] or C-based systems like CRL [10] and Cid [12] or Jade [14].

Aurora is an object-based system where the programmer can select different consistency models for a shared object dynamically at runtime by annotating the C++ source code with corresponding function calls. Orca on the other hand defines an own language for a shared data-objects model. In Orca, the compiler generates information about the usage pattern of shared objects which is then used by the run-time system to implement efficient data distribution. CRL is a library of C functions that implement a DSM system. The code must be annotated with calls explicitly mapping shared data into local memory. Cid is quite similar to CRL, extending C with source code annotations to identify global objects. Cid is more general than CRL due to better multithreading support and potential for the programmer to influence data placement. Jade's approach is closest to the annotations suggested here: units of code are identified as tasks and the programmer provides data access information for these tasks; Jade's runtime system dynamically extracts the parallel execution from this.

3 The Bridge

This section describes the original Charlotte system and extensions to it that constitute a possible bridge between Charlotte's DSM semantics and message-passing semantics.

3.1 Standard Charlotte

A Charlotte program consists of alternating sequential and parallel steps. A manager application (running as a standalone process) executes the sequential steps and administers the parallel steps. In such a parallel step (delimited by `parBegin()` and `parEnd()`), a number of *routines* are defined and picked up by any number of workers (which are applets running in a browser). The end of a parallel step is a barrier synchronization for all the routines in this step. The current Charlotte implementation does not allow nested parallel steps. The memory is logically partitioned in private (local to a routine) and shared segments. The shared memory has *concurrent read, exclusive write* semantics.[1] Charlotte deliberately chooses this conservative semantics to make the programming model as simple as possible.

The shared memory is implemented at the data type level. For a Java primitive type like, e.g., `int`, there is a corresponding Charlotte class `Dint` (distributed int) that provides the correct distributed semantics. The actual data access happens via member functions `get()` and `set()`, since Java does not allow implicit type conversion. If, upon a read-access, a data item is detected to be invalid at a worker,[2] this worker sends a request for this data item to the manager and declares it valid upon reception. During a write access, the object is marked as modified; all modified objects are sent back to the manager at the end of a routine. Since the Java applet security restrictions impose a star-like topology on communication, this master-worker structure fits especially well.

Additionally, Charlotte's memory semantics allows the use of *eager scheduling*. A routine can be given to multiple workers without jeopardizing the correctness of the computation. Thus, crashed or slow workers can be compensated for by re-scheduling their routines on other workers while still guaranteeing "exactly once" execution (for implementation details, please cp. [4]). This entails Charlotte's fault tolerance and adaptive parallelism properties. As an example, Figure 1 shows the skeleton of a matrix multiplication program in Charlotte; `drun` is the actual implementation of a routine.

It is important to point out that Charlotte only uses standard Java mechanisms and does not require a modified Java Virtual Machine or low-level libraries. In particular, the distributed classes like `Dint` are standard Java classes.

3.2 Annotating Routines

Requesting data from the manager upon read access can be very time-consuming, particularly in a high-latency environment. Charlotte tries to amortize this overhead by copying, for each request, a set of objects (a "page"—not to be confused with virtual memory pages) from the manager to the worker. Choosing page sizes is difficult: large

[1] *Concurrent read, concurrent write* semantics is possible with somewhat higher overhead in the manager.

[2] Workers have no valid data at the beginning of a parallel step.

```
// multiply two SizexSize matrices: C = A*B
public class Matrix extends Droutine {
  // this is executed by the workers:
  // compute row 'myId' of C
  public void drun (int numRoutines, int myId) {
    int sum;
    for(int col=0; col<Size; col++)  {
      sum = 0;
      for(int k=0; k<Size; k++)
        sum += A[myId][k].get() * B[k][col].get();
      C[myId][col].set(sum);
    }
  }
  ...
  // this is executed by the manager:
  public void run () {
    ...
    // a parallel step with Size routines
    // (one for each row):
    parBegin();
    addRoutine (this, Size);
    parEnd();
    ...
  }
}
```

Figure 1: Matrix Multiplication in Charlotte (abbreviated)

pages reduce the frequency of data requests, small page sizes reduce redundant communication. Since Charlotte has no way of predicting which data is going to be used, any page size is merely a heuristic.

If, on the other hand, the programmer gave Charlotte some hints which data is actually going to be used by a routine, Charlotte could send this "read set" along with the routine itself. The advantage of doing this is two-fold: there is no latency wasted for data requests and no communication bandwidth is wasted (if the hints given are correct). If the hints turn out to be wrong, the correctness of the program is still guaranteed since a read-access to data which was not sent in advance is still detected and served by standard Charlotte mechanisms. In this sense, these hints are correctness-insensitive. Additionally, it is possible to generate a warning if hints turn out to be redundant or incomplete.

Hints are given by annotating a routine: a method `dloc_read` is defined which is called by the runtime system to obtain the read set for a given routine (see Figure 2 for an example).[3] Since both `drun` and `dloc_read` are methods of the same object, the association between a routine and its annotations poses no problem. Similar hints can be given for the data written by a routine.

```
public class Matrix extends Droutine {
  ...
  public Locations dloc_read (int numRoutines, int myId)
  {
    // compute the read set and store it in loc
    Locations loc = new Locations();

    // all of B:
    loc.add (B);

    // row ``myId'' of A:
    loc.add (A[myId]);

    return loc;
  }
  ...
}
```

Figure 2: Annotating a Charlotte routine with its read set.

The manager keeps track of which data is currently valid at which worker. Thus, if two routines with overlapping read sets are given to a worker, only the missing data is sent with the second routine.

3.3 Relying on Annotations

Assuring the local availability of data at a worker is costly: essentially, an if-statement has to be executed for every read access to a shared object and a flag has to be set for a write access. If the programmer is sure that the annotations describe the read and write behavior of a routine correctly (e.g., after sufficient testing), or if they are generated by a compiler, there is no longer any reason for this overhead since the manager takes care of sending the required data and the worker knows which data to return to the manager.

[3] In the base class `Droutine`, `dloc_read` just returns `null`. The class `Locations` handles descriptions of data sets. In the current implementation, individual objects, arrays, subarrays and matrixes of primitive types and Charlotte's distributed classes can be added directly. Objects that implement `Serializable` can be treated in a similar fashion. It is also possible to extend `Locations` to handle other, application-specific classes.

This can be accomplished by using unchecked counterparts of Charlotte's distributed classes (`Uint` instead of `Dint`, etc.), retaining the exact same interface as the correctness-guaranteeing classes. Moving from checked to unchecked classes is a mere syntactic change of class name and constructor.

3.4 Sharing Primitive Types

For these unchecked classes, the `get()` and `set()` methods for such unchecked data are completely trivial—there is no longer any reason to pay the overhead for their invocation. As a matter of fact, primitive data types like `int` can be used directly—the runtime system uses the annotations to move data back and forth between manager and worker as needed. Thus, the shared memory semantics of Charlotte can be implemented on top of primitive data types allowing direct access without Java's high method invocation overhead (much as it would be done in a message passing program).

Unlike the unchecked classes, using primitive data types does change the interface, e.g., the `get()` and `set()` method invocations have to be removed. In particular, objects are passed by reference, primitive types by value. This can make the transition to primitive types awkward for single variables. But since the overhead for single variables is small in either case, the main advantages lie in the use of annotations for arrays. And arrays of objects or arrays of primitive types do have the same passing semantics. Nevertheless, this step requires careful consideration.

Note that it is of course possible to mix objects of the original Charlotte classes (with or without annotations) and shared primitive types at will. This allows a programmer to use Charlotte's distributed classes for data with complicated access patterns and primitive types for more straightforward data.

3.5 Additional Optimizations

Since the manager keeps track of which data is currently valid at a worker, it is possible to use this information for two additional optimizations.

First, the manager can use the difference between a worker's valid data set and a routine's read set as a criterion for choosing which routine to give to a worker. Choosing the routine which minimizes this difference also minimizes the amount of data the manager has to sent.[4] In a sense, a routine is given to a worker that already has "co-located" data for this routine (hence "colocation" as a short term for this heuristic).

Second, for the unchecked shared objects, data movement to the workers is solely the managers responsibility. It is therefore possible to leave all the workers' local data intact at the end of a parallel step (instead of declaring them invalid as in standard Charlotte) and to overwrite them with new values only if necessary. If a program declares shared data as unchanged at the beginning of a parallel step, the manager will not remove this data from the workers' valid data set and therefore not send it again. This mechanism constitutes inter-step caching. It also allows colocation to take advantage of data send in a previous step and not to be restricted to overlapping data within one step.

3.6 Discussion

The most important thing to note for these extensions is that they allow a gradual improvement of a program: From Charlotte's pure DSM to DSM plus hints to shared objects without correctness checks to sharing primitive data types the correctness of which is completely based on annotations (cp. Figure 3). Access to these primitives types is direct without any method invocations and therefore equivalent to what is commonly used in message-passing systems.

The annotations for read and write data sets of a routine do look a little like reading and writing data from and to the network. But since only the data sets are described, the programmer does not have to worry about streams, I/O-exceptions, etc. Additionally, only one description is necessary as opposed to code for sending and receiving data—by comparison, a message-passing program actually over-specifies the communication. It is possible to transform these descriptions into direct send/receive calls, but the gain should be minimal; e.g., it is difficult to avoid redundant data transmission with pure message-passing calls. While this approach does generate some overhead for the runtime system, the following section will show that this overhead is well invested.

All Charlotte's initial advantages like fault tolerance and adaptive parallelism are maintained—capabilities that

[4] Assuming the routine is correctly described by the annotations. Otherwise, this is the best guess the manager can make regarding the amount of communication for a given worker/routine combination.

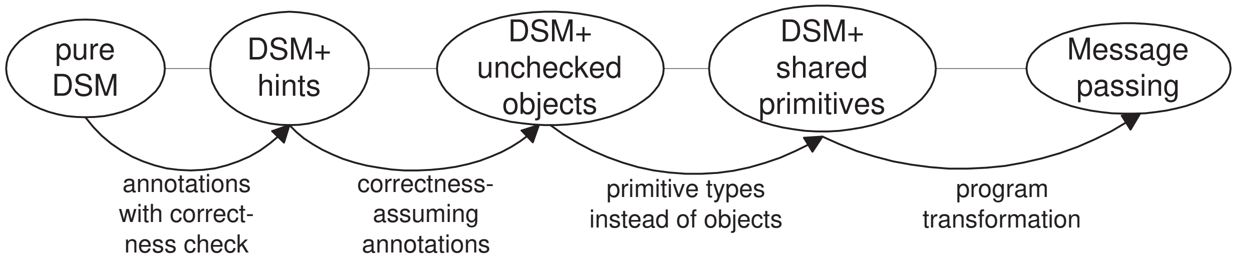

Figure 3: Steps between Charlotte's DSM and a message-passing system.

would be laborious to implement using message-passing primitives alone. Additionally, a flexible mixture of purely DSM-based and annotation-supported objects is possible.

In comparison to related work, CRL and Cid are close relatives. But Charlotte's simpler programming model and the direct use of objects make these annotations easier to use for a programmer than having to worry about mapping and locking memory regions, plus having the additional possibility to use pure DSM objects. Jade's annotation technique is also very similar to the approach proposed here, but it lacks the capability to mix different levels of correctness guarantees; Jade completely relies on the correctness of the given annotations. The flexibility to mix different levels of guarantees and switch back and forth between them is the particular strength of this approach.

4 Measurements

This section illustrates the differences between and advantages of the various approaches with matrix multiplication as an example for measurements. Matrix multiplication was deliberately chosen as a problem with only moderate granularity. Problems with very high computation/communication (e.g., computing prime numbers), which are often used as examples for Web-based computing, suffer from the problems addressed by the extensions proposed here only to a much smaller degree, but such a focus also limits the application scope.

The environment used for experiments consists of a number of PentiumPro 200 machines at the Distributed Systems Laboratory of New York University connected by a 100 MBit/sec Ethernet and two Pentium 90 at Humboldt University Berlin. A ping between these two sites typically takes about 130 msec. All machines were running Linux 2.0. Sun's Java Development Kit Version 1.1.3 and the Kaffe Virtual Machine Version 0.92 [15] (a Java Just-In-Time (JIT) compiler) were used to run the programs. Runtimes for a purely sequential implementation are shown in Table 1.

	JDK	Kaffe
Pentium 90	16.5	8.1
PentiumPro 200	9.0	2.3

Table 1: Sequential runtimes (in sec.) for multiplying two 200x200 integer matrices.

The first question asked of an enhancement for a parallel system if of course the one regarding improvements in runtime. Figure 4 shows the runtime for a 200x200 matrix multiplication with up to four workers, measured on the local network at NYU (all numbers are averaged over 10 runs) using JDK, Figure 5 shows the same times using the Kaffe JIT-compiler. In both figures, as in all the following ones, `Dint` refers to standard Charlotte, `Dint+A` to annotated Charlotte with correctness check, `Uint` indicates the use of the unchecked class and `int` sharing primitive integers instead of objects.[5] Times for a message passing implementation of matrix multiplication (implemented directly on top of Java IOStreams) are given in both figures.

Both for the interpreted and the compiled version the improvements in runtime using annotations are striking. Since the JIT-compiler gives considerably better results, and JIT-compiler are more and more commonly available in most browsers, only these times will be discussed in the following. Nevertheless, it is worthwhile to point out that the interpreted case behaves in general very similar to

[5] The actual programs are slightly more complicated than shown in Figure 1. Using one routine per matrix row is too inefficient—grain sizes of 5, 10, 25 and 50 rows per routines were used in the experiments and the best of these results is reported. As is to be expected, for more workers, optimal grain sizes are decreasing.

Figure 4: Run time of Matrix Multiplication on local network (NYU) with JDK.

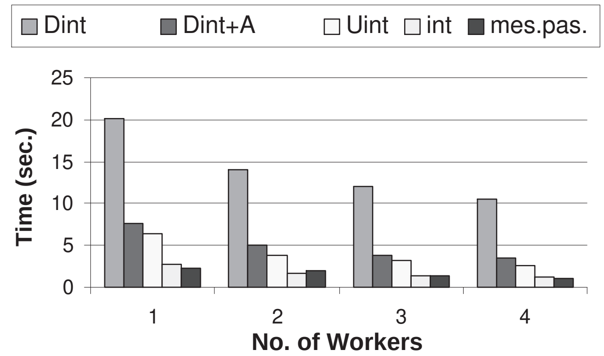

Figure 5: Run time of Matrix Multiplication on local network (NYU) with JIT-compiler.

the compiled case. Note that with `int`, one worker executes almost as fast as the sequential version and shows actual speedup with two or more workers in the interpreted version—the message passing implementation suffers from only negligible overhead with one worker. The times for message passing and the Charlotte program with primitive data types are practically identical,[6] proving the claim that with annotating Charlotte near-message passing efficiency[7] is possible while still maintaining advantages like fault tolerance. Figure 6 shows the absolute speedup of the compiled case compared with the sequential execution time (thus taking Charlotte's overhead into account).

Figure 6: Absolute speedups (compared with sequential program) for the different parallelized versions using JIT-compiler.

It is particularly interesting to compare the runtimes needed by the different extensions of Charlotte that have been introduced in this paper and the message passing version. Figure 7 shows the ratios of runtimes when comparing standard Charlotte (`Dint`) with annotated Charlotte (`Dint`+A), Charlotte with unchecked distributed objects (`Uint`) and primitive types (`int`). The annotations make data requests unnecessary and send all the data needed for a routine in one batch—this improves runtime by about a factor of three (`Dint` vs. `Dint`+A). The `Uint` version shows another slight improvement, but the ability to forego the overhead associates with objects and to share primitive types adds another factor of two—resulting in an

[6]Actually, in some circumstances the Charlotte version is faster than the message version. While this may be due to statistical fluctuations, Charlotte's load balancing capabilities give it an advantage over a straightforward message passing program.

[7]Of course, the message passing program also communicates only along a star-like topology to reflect the constraints imposed by the Java security model.

overall improvement of about a factor of nine over standard Charlotte.

Figure 7: Ratios of run times of different optimizations (compiled case).

The runtime using connections with high latencies was tested with two workers running at Humboldt University Berlin; runtime and ratios between various methods for this setup are shown in Figure 8 and Figure 9 respectively.[8] Again it is obvious that the shared primitives version attains a performance comparable to the message passing implementation. Unfortunately, since these machines are considerably slower than the local machines, the numbers are not directly comparable and no direct conclusions concerning the respective gains for low- and high-latency environments are possible.

Figure 8: Times with master at NYU, workers at Berlin for various versions of Charlotte and message passing.

The optimization in Subsection 3.5 where also proposed with long latencies in mind. For the example of multiplying a matrix A with two matrixes B_1 and B_2 in two consecutive parallel steps, Figure 10 shows the communication time using `Dint` plus annotations, additionally caching A between the two steps, and both caching A and taking the distribution of A among the workers into account for the second parallel step (colocation). While in a LAN environment the impact of colocation is only small, for high-latency connections colocation can save up to 25% of runtime (in this particular example). Perhaps even more important is the fact that the standard deviation of colocation is roughly a factor of three smaller than the other methods.'

Figure 9: Ratios with master at NYU, workers at Berlin for various versions of Charlotte and message passing.

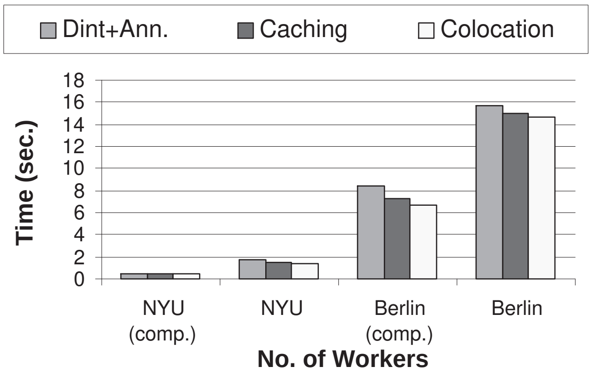

Figure 10: Communication times for matrix multiplication with `Dint` plus annotations, `Dint` plus annotation and caching, and `Dint` plus annotation and caching and colocation (averaged over 1000 runs).

[8] `Uint` is not shown since, as seen above, the major improvements stem from annotations and primitive data types.

5 Future Work

There are a number of possible extensions to this work. The annotations described here suffer from some limitation imposed by the Charlotte `parbegin()`/`parend()` structure. In particular, it is awkward to use more than one routine per `Droutine` object and, hence, annotating such routines is difficult, too. We are currently investigating a different syntactic approach for both Charlotte and Calypso (a page-based DSM system [3]). Studying the applicability of such annotations to Calypso is under way.

Also, the annotation techniques proposed here should be studied with some more elaborate examples to establish the difficulty of writing the annotations. The biggest improvement is to be expected for programs that show regular data-access behavior. But since annotated and pure DSM objects can be freely mixed, even programs with more difficult structures should be manageable (e.g., with speculative annotations).

This can be extended by studying the impact of problem size and communication/computation ratio on the relative performance of these optimizations. Generating the annotations by a compiler-based data-flow analysis would also be most interesting.

Overlapping computation with computation is an orthogonal issue. Coordinated execution of multiple workers within one browser is an obvious approach to this problem.

6 Conclusions

This paper set out to provide a means to bridge the gap between the DSM-semantics behind Charlotte and simpler, yet more efficient message-passing systems. An annotation-based method has been proposed to augment a Charlotte program with information about the data dependencies of routines executing in parallel.

These annotations can have the character of hints, allowing the run time system to improve communication efficiency while guaranteeing the correctness of the program. They can also be used as a precise description of read and write sets which allows the sharing of primitive types like `int` across multiple machines. The stepwise nature of this concept allows a programmer to gradually incorporate knowledge about a program's behavior and to freely mix pure DSM objects with annotation-based objects or shared primitive types.

Sharing primitive types results in data access efficiency usually found only in message-passing systems or hardware-supported DSM systems. By building on top of Charlotte, this efficiency is now available for Java-based Web-computing without putting the burden of low-level communication or locking primitives on a programmer while properties that are crucial for Web-computing, e.g., fault tolerance, are maintained. In this sense, advantages from DSM systems and message passing are incorporated in this concept.

The practicability and ease-of-use of this approach has been shown with matrix multiplication as an example. A number of measurements substantiate the claim to vastly improved performance—run time improvements of up to a factor of nine over standard Charlotte and competitive with a pure message passing implementation were observed.

This shows that with modest overhead for programmer and runtime system, even problems of only moderate granularity can be efficiently solved in a Java-based DSM programming environment.

Acknowledgements

Acknowledgements are due to Arash Baratloo for valuable discussions and to the anonymous referees for pointing out related work (in particular, Jade) and helpful comments.

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; by the National Science Foundation under grant number CCR-94-11590; by the Intel Corporation; and by Deutsche Forschungsgemeinschaft (German Research Council).

References

[1] H. E. Bal and M. F. Kaashoek. Object-Distribution in Orca using Compile-Time and Run-Time Techniques. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93)*, pages 162–177, Washington, D.C., 1993.

[2] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proc. of the 7th ACM SIGOPS European Workshop: Systems support for Worldwide Applications*, Connemara, Ireland, September 1996.

[3] A. Baratloo, P. Dasgupta, and Z. M. Kedem. CALYPSO: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms. In *Proc. of the 4th IEEE Intl. Symp. on High-Performance Distributed Computing*, Washington, D.C., August 1995.

[4] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proc. of the 9th Intl. Conf. on Parallel and Distributed Computing Systems*, Dijon, France, September 1996.

[5] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards World-Wide Supercomputing. In *7th ACM SIGOPS European Workshop*, pages 181–188, Connemara, Ireland, September 1996.

[6] P. Cappello, B. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. In *ACM Workshop on Java for Science and Engineering Computation*, 1997.

[7] J. B. Carter, J. K. Bennet, and W Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating System Principles*, pages 152–164, October 1991.

[8] A. J. Ferrari. JPVM: Network Parallel Computing in Java. In *Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 245–250, Palo Alto, CA, February 1998.

[9] I. Foster and S. Tuecke. Enabling Technologies for Web-Based Ubiquitous Supercomputing. In *Proc. of the 5th IEEE Symp. on High Performance Distributed Computing*, pages 112–110, 1996.

[10] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performacne All-Software Distributed Shared Memory. In *Proc. of the Fifteenth Symposium on Operating Systems Principles*, March 1995.

[11] P. Lu. Aurora: Scoped Behaviour for Per-Context Optimized Distributed Data Sharing. In *Proc. of the 11th Intl. Parallel Processing Symposium*, pages 467–473, Geneva, Switzerland, 1997.

[12] R. S. Nikhil. Cid: A Parallel "Shared-Memory" C for Distributed Memory Machines. In *Proc. of the 7th Ann. Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *LNCS*, pages 376–390, Ithaca, NY, August 1994. Springer-Verlag.

[13] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. In *ACM 1997 PPoPP Workshop on Java for Science and Engineering Computation*, Las Vegas, NV, June 1997.

[14] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer*, 1993.

[15] T. Wilkinson. Kaffe—A free virtual machine to run Java code. http://www.kaffe.org/.

[16] A. Yu, W. Cox. Java/DSM: A Platform for Heterogeneous Computing. In *Proc. of ACM 1997 Workshop on Java for Science and Engineering Computation*, Las Vegas, NV, June 1997.