CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2004; **16**:61–77 (DOI: 10.1002/cpe.747)

Parallel operation of CartaBlanca on shared and distributed memory computers

N. T. Padial-Collins[*,†], W. B. VanderHeyden, D. Z. Zhang, E. D. Dendy and D. Livescu

Los Alamos National Laboratory Theoretical Division and Los Alamos Computer Science Institute, Los Alamos, NM 87545, U.S.A.

SUMMARY

We describe the parallel performance of the pure Java CartaBlanca code on heat transfer and multiphase fluid flow problems. CartaBlanca is designed for parallel computations on partitioned unstructured meshes. It uses Java's thread facility to manage computations on each of the mesh partitions. Inter-partition communications are handled by two compact objects for node-by-node communication along partition boundaries and for global reduction calculations across the entire mesh. For distributed calculations, the JavaParty package from the University of Karlsruhe is demonstrated to work with CartaBlanca. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: Java; scientific computing; parallel computation

1. INTRODUCTION

Our goal is the development and demonstration of high-performance computing with Java both in serial and parallel modes. It is also highly desirable to have a single code that can run both on shared memory and distributed memory systems. This would greatly simplify the maintenance and modification processes. The rationale for using Java in high-performance computing applications is based on several factors. First, Java is a relatively simplified and streamlined object-oriented language. Threads and networking are built-in features of the language available for parallel computation. Due to its marketplace strength, a wealth of third party software components is available for re-use. One notable example is the Junit facility for testing [1].

Finally, the introduction of dynamic compilers has brought Java performance very close to that of Fortran and C++ for scientific computation. Boisvert *et al.* [2] provide an up-to-date review, which

[*]Correspondence to: N. T. Padial-Collins, Group T3, Fluid Dynamics, MS B251, Los Alamos National Laboratory, Los Alamos, NM 87544, U.S.A.
[†]E-mail: nelylanl@lanl.gov

Received 13 August 2002
Revised 9 January 2003
Accepted 12 January 2003

shows that Java performance has improved significantly over the past several versions of the Java Version Machine (JVM). Remarkably, they show that, for some cases, Java is slightly faster than C when tested back-to-back on a suite of scientific computing benchmark applications called Scimark from the U.S. National Institute of Standards and Technology. In addition, Reinholtz [3], writes that not only is it possible for Java to have better general performance than C++, but also is likely that this will actually occur. He argues that Java performance can exceed that of C++ because dynamic compilation gives the Java compiler access to runtime information not available to a C++ compiler. Furthermore, the rapidly growing market for embedded systems will drive such performance improvements to extend battery life. Considering this information and the important benefits from Java's strong typing, clean design, object-orientation, we have to conclude that Java is a serious alternative language for scientific and engineering computing applications.

1.1. Communication environments for Java

The advent and popularity of clusters of workstations stimulated the development of a large body of methods that allow the execution of a Java program on a distributed memory system. Standard Java provides a standard communication library, the Remote Method Invocation (RMI), which is based on a slow object serialization. Part of the reason for the inefficiency of the standard RMI implementation is the security features for business applications. These security features are, however, not as important for scientific applications. Therefore, researchers have begun to develop non-standard packages that try to improve efficiency and performance of the RMI implementation for scientific applications. These tend to come in two categories, Distributed Shared Memory (DSM) and Message Passing Interface (MPI) environments. DSMs allow the direct application of shared memory programs on distributed processor clusters with little or no modification. The MPI libraries provide wrappers to the standard MPI libraries.

Due to its robustness and relative ease of use, we used JavaParty for the work described in this paper. The following is a brief review of currently available communication packages and libraries for Java. We start by discussing JavaParty [4], which is a DSM system that allows porting of multi-threaded Java programs to distributed environments with minimal modifications. JavaParty allows classes to be declared as remote, which allows them to be visible and accessible anywhere in the JavaParty distributed environment. Consequently, the JavaParty environment acts as a single JVM. Since the access and creation of remote classes is syntactically indistinguishable from regular Java classes, minimal modifications to a multi-threaded Java program are required for use with JavaParty.

For completeness, we provide a brief review of other Java communications libraries discussed in the literature. Java/DSM is a modified JVM implemented on top of ThreadMarks Distributed Shared Memory system [5]. DO! is a system for automatic generation of distributed code [6]. The Reflective Remote Method Invocation (RRMI) is based on an alternative object serialization [7]. Manta is a native Java compiler with highly efficient RMI [8]. Jalapeno is a virtual machine for Java servers written in Java [9]. Cluster Virtual Machine for Java virtualizes the cluster without requiring that any pure Java application be specially written for it [10]. Jackal is a DSM system built upon the Manta environment. It enables the running of unmodified multi-thread Java programs on a cluster of workstations [11]. ProActive Java Library for Parallel, Distributed Concurrent Computing is composed of standard Java classes and uses RMI as the transport layer [11]. JESSICA, 'Java-Enabled Single-System Image Computing Architecture' is an environment that runs on top of standard Unix systems to support parallel execution of multi-thread Java programs on a cluster of computers [12]. MPIJava is an object-oriented Java interface for the standard MPI system [13]. JavaMPI provides an automatic generation of wrappers to MPI libraries [14]. MPIJ is a pure Java implementation of MPI and can run without the need for a native MPI implementation [15]. JMPI is also a 100 percent Java-based implementation of the Message Passage Interface [16]. The Hyperion System [17], combines Java compilation to native code with a run-time library for the distributed execution of Java threads. Finally, JavaSymphony is a high-level Java-based programming paradigm for parallel and distributed systems [18].

1.2. CartaBlanca

To exploit the benefits of Java and the ancillary communication packages described above, we have recently developed a general-purpose nonlinear system solver environment for complex physics computations on unstructured grids. The environment, called CartaBlanca, was reported on at the 2001 Java Grande/ISCOPE Conference [19]. In the following, we review the main features of CartaBlanca and then discuss our progress toward a general parallel computation capability for both shared and distributed memory systems.

The paper is structured as follows. In Section 2 we provide an overview of CartaBlanca. In Section 3 we describe implementation of communication for parallel computations in CartaBlanca. We follow this in Section 4 with some results from example calculations. Finally, in Section 5 we provide a discussion of conclusions and plans.

2. OVERVIEW

The goal of the CartaBlanca project is to produce a modern flexible software environment for prototyping physical models, discretization schemes and solution methods for nonlinear physics problems on unstructured grids. CartaBlanca employs an object-oriented, component-like design using the Java programming language. CartaBlanca uses unstructured grids composed of either triangular or quadrilateral mesh elements in two dimensions, or tetrahedral or hexahedral mesh elements in three dimensions. CartaBlanca employs the finite-volume method [20], to provide flexibility with regard to both meshes and physical effects. Currently, the Jacobian-Free Newton Krylov [21] method is used for the solution of the nonlinear algebraic systems arising from the discretization of the governing partial differential equations. Finally, CartaBlanca uses Java's built-in thread facility for shared-memory parallelization. CartaBlanca has been used to simulate multiphase flow, heat transfer and solidification, free surface flows, for example.

2.1. Status

The CartaBlanca environment has been described in detail elsewhere [19]. We provide here a brief summary description of its current capabilities and features. CartaBlanca accepts unstructured grids in two and three dimensions with triangular, tetrahedral, quadrilateral and hexahedral elements and uses Metis [22] generated mesh partition files for parallel computations. CartaBlanca has a graphical user interface (GUI) for problem specification and uses abstract classes for state, physics and solver objects to impose a uniform component-like interface for developers. Available Krylov solvers include the pre-conditioned Conjugate Gradient and both the standard and flexible variant of the GMRES method [23]. A JFNK nonlinear quasi-Newton solver is also included. Physics objects for high-accuracy scalar advection, interface tracking, heat transfer and multiphase flow, phase change and radiation effects have been implemented. A software-testing facility based on the JUnit framework [21], is integrated into the environment.

As single processor performance, we have reported elsewhere [19], that CartaBlanca has demonstrated roughly half the speed of a mature FORTRAN code on a prototypical multiphase flow problem, the broken dam, which will be discussed further in subsequent sections of this paper.

2.2. Related efforts

It is worthwhile to note relevant examples from outside our laboratory. First, Hauser *et al.* [24] have produced an object-oriented, pure-Java simulation code for aerospace applications. They employ a multi-block structured grid computation scheme and use Java's thread and RMI facilities for parallelization and to enable remote interaction between a GUI and the numerical application.

Another effort worth noting here is that of Hatakeyama *et al.* [25], who describe an object-oriented paradigm for flow simulation software in which the object-oriented concept is used at the computational node level to produce flexible abstractions. They demonstrate their concepts with a C++, structured grid simulation of a wind tunnel with a test object and show that they can easily insert and extract arbitrary-shaped flow obstacles.

Finally, in the work of Riley *et al.* [26], NASA flow solver and adaptive mesh codes were rewritten in object-oriented Java and compared with their FORTRAN counterparts. These workers showed that the performance of their Java codes was about one-half that of similar codes written in FORTRAN. This work shows that full use of object-oriented programming techniques does not necessarily result in poor performance.

3. COMMUNICATION IMPLEMENTATION

Parallelism in CartaBlanca is based on the partitioning of a given computational mesh. Generally speaking, there are two types of communication operations that must be addressed. To explain this consider an example of mesh partitioning as shown in Figure 1. The partitioning was performed using the Metis program [22]. For CartaBlanca, partitions are cut along edges connecting nodes. An example of this is shown in Figure 2. As shown in the figure, the partition boundary runs along node connections following the heavy black line. For each node on the partition boundary, we carry a duplicate on each partition. Thus for the case in which several partitions meet at a corner, we would have more than two duplicates. Each node in the mesh corresponds to a control volume the faces of which correspond to each of the edges connecting neighbor nodes. The dashed line in Figure 2 shows the boundary of the control volume surrounding the center node.

In general, the two types of communication that must occur between these mesh partitions are the exchange of information along partition–partition boundaries and the exchange of global extrema and sums. The former are called boundary communications and the latter are called reduction communications.

Boundary communications arise, for example, when gradients and divergences of a given scalar, vector or tensor field are computed using Gauss' theorem. Discretization of the surface integral

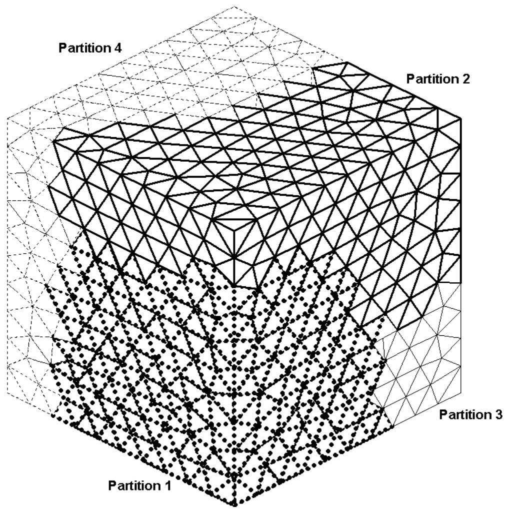

Figure 1. A four-partition mesh example.

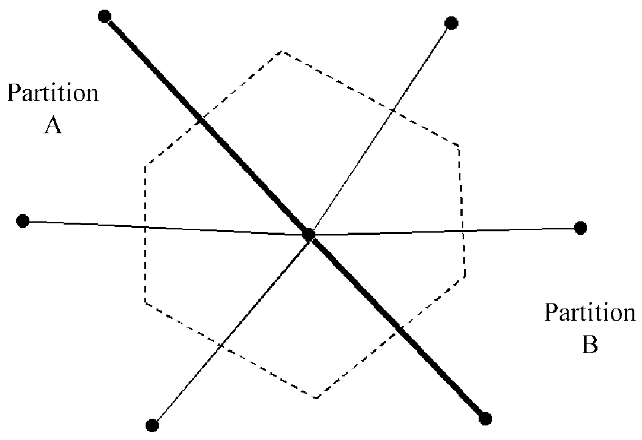

Figure 2. Example of partition boundaries in CartaBlanca.

introduces a sum over control volume face areas

$$\nabla q \approx \frac{1}{V}\int_V \nabla q \, \mathrm{d}V = \frac{1}{V}\int_A q\vec{n}\, \mathrm{d}A \approx \frac{1}{V}\sum_f q_{\mathrm{f}}\vec{n}_{\mathrm{f}}A_{\mathrm{f}} \tag{1}$$

where q is the field under consideration, V denotes the control volume, A denotes the surface area of the control volume, and $\vec{n}$ is the surface outward normal vector. The subscript f denotes one of the faces corresponding to the edge connections to the neighbor nodes. For the nodes that lie on the partition boundary, the surface integral spans the adjacent partitions and thus requires inter-thread communication.

In the CartaBlanca software, mesh partition objects contain information on the geometrical aspects of the mesh partitions. The information includes physical node coordinates and connectivity between node neighbors on the partition. In addition, each mesh partition object stores connectivity information for nodes on the boundary of the partition regarding neighbors on other partitions. This information is accessed for communication operations such as in the case of gradient and divergence calculations.

The second class of communication in CartaBlanca includes the reduction operations such as global extrema and global sums. For these operations, each mesh partition must contribute a single numerical result such as a maximum. This type of operation is required, for example, when computing an error norm across the entire mesh for use in stopping criteria in the iterative solvers.

Both of the communication operations—boundary and reduction–are handled in a single communication object as depicted in Figure 3. Starting with the main method at the top level, the program spawns threads for each of the mesh partitions. In the example shown in Figure 3, there are four partitions. Each thread has a cycle driver object, a solver object, a physics object and finally a discrete operator object. The cycle driver object invokes the solver object each time cycle increment. The solver object calls the physics object repeatedly while it iteratively adjusts the solution vector to bring the physics object's equation residual norms to within tolerance of zero. The physics object accepts the latest guess of the solution vector from the solver object and in turn evaluates the equation residuals. The physics object uses the discrete operator object to evaluate quantities such as gradients, and divergences that are required to form the residuals.

As shown in Figure 3, two distinct types of communication occur among the threads of control. The solver objects communicate reduction information for mesh-wide data such as global maxima, minima. Communication in the discrete operator objects consists of cross-boundary information amongst the partitions for the computation of gradients and divergences as discussed previously. To facilitate these two modes of communication, CartaBlanca has a communication class that provides the bridge between the various threads as two services. The reduction communication service is used by the solver objects to compute global extrema. The boundary communication service is used to compute surface integrals for the duplicate nodes on the inter-partition boundaries. In each of these, the basic mode of operation is for each of the partition threads to compute their own contribution and then send it to the communication object. Once this is done, then the partitions can retrieve from the communication objects the data from the other partitions. Barriers are used to ensure the synchronization of these processes.

Communication between the various objects is handled by simple get and set methods. The program, as written, works directly on shared memory computers. For distributed memory clusters, a DSM facility can be used. With some additional modifications, MPI-based systems may also be used to compute on cluster computers.

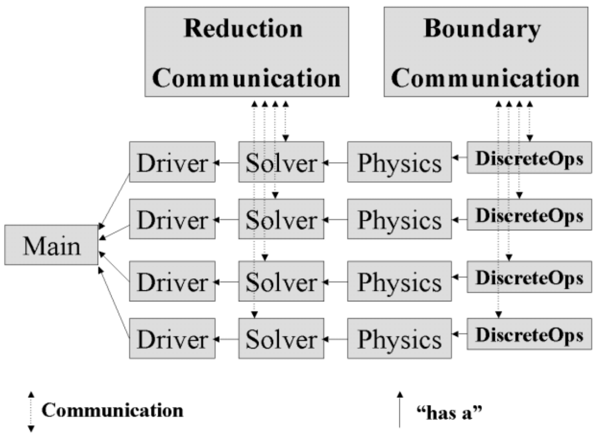

Figure 3. Overview of the parallel structure in CartaBlanca.

We have successfully used JavaParty [4] in conjunction with CartaBlanca for parallel operations on distributed systems. As previously explained, JavaParty allows the declaration of classes as remote and the simultaneous creation of more than one virtual machine, each of which is able to access remote classes and their instances. In addition, in the JavaParty environment, the access to a remote class is syntactically identical to the access to a regular Java class as if we had only a virtual machine distributed over several computers. Thus, the modification of the base CartaBlanca code is minimal. Only two changes had to be implemented to our code. First, we had to declare the communication object as remote. Second, we used JavaParty's remote threads instead of using Java's native threads [4] for the parallel threads of control depicted in Figure 3.

4. RESULTS

We demonstrate CartaBlanca's parallel performance on two prototypical physics problems: heat transfer and multiphase flow. These problems are described more fully in [19]. Briefly, the heat transfer problem solves the transient heat equation on a square domain. In the examples, the mesh is a uniform, two-dimensional quadrilateral grid. The initial temperature distribution has run from zero to one in the x direction and has no gradient component in the y direction. The solution to the problem is that the temperature relaxes to a steady state with a uniform temperature of one-half. The multiphase flow problem simulates a broken dam flow wherein a fluid such as water is initially confined to the right half of a square domain with another fluid such as air in the left half. At zero time, gravity is 'turned on' and the water slumps and flows to fill the bottom half of the domain. The broken dams simulated were two-dimensional and the meshes were Cartesian quadrilateral element grids.

We performed both shared memory and distributed memory parallel scaling tests. For the shared memory tests, we used an 8-processor Intel SMP machine with 900 MHz Pentium-3 chips.

Table I. Intel shared memory, broken dam, 201 × 201 grid.

Number of processors	Grind time, ms/cycle/node	Efficiency
1	6.991	1.00
2	3.953	0.88
3	2.632	0.89
4	2.604	0.67
5	2.811	0.50
6	2.808	0.41
7	2.857	0.35
8	3.030	0.29

The operating system was RedHat Linux-2.4.10. For the distributed memory tests, we used a Linux cluster distributed memory machine. Each node was a Compaq DL360 with two Intel Pentium-3, 1 GHz processors. The network was GigE and the operating system was RedHat Linux 2.4.10. In the runs discussed here, we used one processor per node since this mode of operation provided a pure test of distributed memory operation. The Sun JDK version 1.3.1_02 was used on both the shared and distributed memory clusters. We used the JavaParty package with the University of Karlsruhe fast Remote Method Invocation package (KaRMI).

Parallel efficiency is defined as

$$\frac{\text{Single processor time}}{\text{Multi-processor time} \times \text{Number of processors}}$$

for both the shared and distributed memory tests. Thus, perfect scaling results in an efficiency of unity. The following sections discuss the results of our tests.

4.1. Shared memory results

The results for the shared memory tests are shown in Tables I–V. For proper interpretation, it is important to point out that the timings in the tables are on a per node, per cycle basis. That is they are not raw wall clock but wall clock time per node per cycle. We call this quantity the grind time.

In general, the scaling is reasonable and is similar to what we have seen with similar FORTRAN codes on this class of problems. We are not prepared at this point, however, to make any quantitative comparisons of scaling with FORTRAN codes. This is left for future work.

4.2. Distributed memory results

We now discuss the performance of CartaBlanca on the distributed memory cluster.

Table II. Intel shared memory, broken dam, 301 × 301 grid.

Number of processors	Grind time, ms/cycle/node	Efficiency
1	8.351	1.00
2	5.015	0.83
3	3.532	0.79
4	3.116	0.67
5	3.021	0.55
6	3.101	0.45
7	3.000	0.40
8	3.125	0.33

Table III. Intel shared memory, broken dam, 401 × 401 grid.

Number of processors	Grind time, ms/cycle/node	Efficiency
1	11.422	1.00
2	7.381	0.77
3	5.400	0.71
4	4.588	0.62
5	4.365	0.52
6	4.299	0.44
7	4.254	0.38
8	3.125	0.46

4.2.1. Prototype code

Before examining Cartablanca parallel runs with JavaParty in distributed memory service, we tested our communication modules on a small prototype communication code that consists of cycles in which a for-loop calculates polynomials, with a barrier before and after the for-loop. For multiple processors, this for-loop can be divided in two, three, four or eight parts if calculated on, respectively, two, three, four or eight processors. At the end of the for-loop on each processor, we calculate the minimum and maximum value of the last polynomial across all loop instances. Thus, for the multiple processor cases, this operation required inter-processor communication. This set of operations was done for a several cycles. Consequently, we have two barriers, a global maximum and a global minimum per cycle. The total number of polynomials calculated is the same in every test. This small code is a nearly ideal parallel application since the size of the calculation (4 800 000 polynomials per cycle) is large

Table IV. Intel shared memory, heat transfer, 201 × 201 grid.

Number of processors	Grind time, ms/cycle/node	Efficiency
1	2.833	1.00
2	1.483	0.96
3	0.959	0.98
4	0.966	0.73
5	1.097	0.52
6	1.039	0.45
7	0.885	0.46
8	0.748	0.47

Table V. Intel shared memory, heat transfer, 301 × 301 grid.

Number of processors	Grind time, ms/cycle/node	Efficiency
1	8.121	1.00
2	4.840	0.84
3	2.809	0.96
4	2.312	0.88
5	2.529	0.64
6	2.228	0.61
7	1.958	0.59
8	1.751	0.58

compared with the size of the communication. As such, it should scale well in both shared memory and distributed memory service. The scaling results of these tests are shown in Table VI.

The results show very good scaling for the case of the distributed cluster using JavaParty. In contract, the shared memory machine, somewhat surprisingly, shows less than ideal scaling. We believe this to be an artifact of the hardware. In both cases, the efficiencies fall off for the 8-processor case. We do not understand why this occurs and the effect should be studied further. These results provide a context for the interpretation of the CartaBlanca results in the following sections.

4.2.2. CartaBlanca-JavaParty

We have been able to run all the problems in the shared memory examples using JavaParty as a DSM facility with very little code modification (as explained in Section 3) with times shown in Tables VII–XI.

Table VI. Grind times for the prototype communication program. (Note time units in this table are nanoseconds per node.)

Number of processors	Grind time, ns/node, shared memory	Efficiency	Grind time, ns/node, cluster	Efficiency
1	18.12	1.00	22.37	1.00
2	11.46	0.79	9.92	1.13
3	7.70	0.78	7.05	1.06
4	5.88	0.77	5.42	1.03
8	3.35	0.68	3.25	0.86

Table VII. Linux cluster, broken dam problem, 201 × 201 grid.

Number of processors	Grind time, ms/cycle/node	Efficiency
1	6.900	1.00
2	4.349	0.79
3	3.693	0.62
4	3.215	0.54
5	3.174	0.43
6	3.148	0.37
7	3.149	0.31
8	3.065	0.28

Table VIII. Linux cluster, broken dam problem, 301 × 301 grid.

Number of processors	Grind time, ms/cycle/node	Efficiency
1	8.832	1.00
2	4.666	0.95
3	3.449	0.85
4	2.919	0.76
5	2.727	0.65
6	2.505	0.59
7	2.408	0.52
8	2.368	0.47

Table IX. Linux cluster, broken dam problem, 401 × 401 grid.

Number of processors	Grind time, ms/cycle/node	Efficiency
1	11.848	1.00
2	6.105	0.97
3	4.419	0.89
4	3.580	0.83
5	3.288	0.72
6	2.952	0.67
7	2.657	0.64
8	2.587	0.57

Table X. Linux cluster, heat transfer problem, 201 × 201 grid.

Number of processors	Grind time, ms/cycle/node	Efficiency
1	3.679	1.00
2	2.102	0.88
3	1.808	0.68
4	1.719	0.54
5	1.686	0.44
6	1.696	0.36
7	1.636	0.32
8	1.723	0.27

Table XI. Linux cluster, heat transfer problem, 301 × 301 grid.

Number of processors	Grind time ms/cycle/node	Efficiency
1	8.939	1.00
2	4.904	0.91
3	3.766	0.79
4	3.341	0.67
5	3.120	0.57
6	2.881	0.52
7	2.672	0.48
8	2.637	0.42

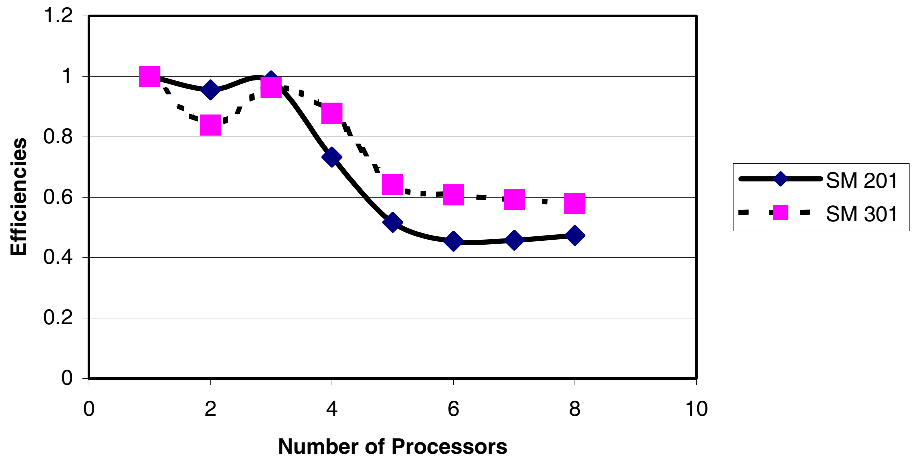

Figure 4. Shared memory efficiencies for the heat transfer problem. SM 201, 201 × 201 grid; SM301, 301 × 301 grid.

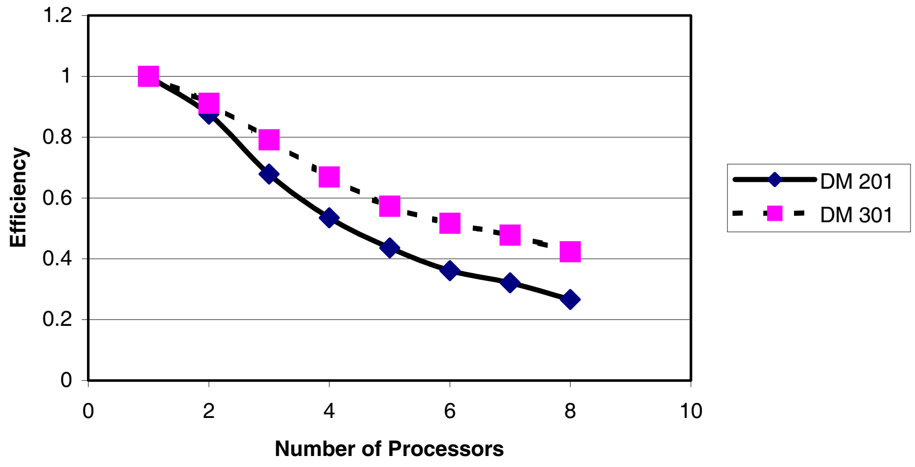

Figure 5. Distributed memory efficiencies for the heat transfer problem. SM 201, 201 × 201 grid; SM301, 301 × 301 grid.

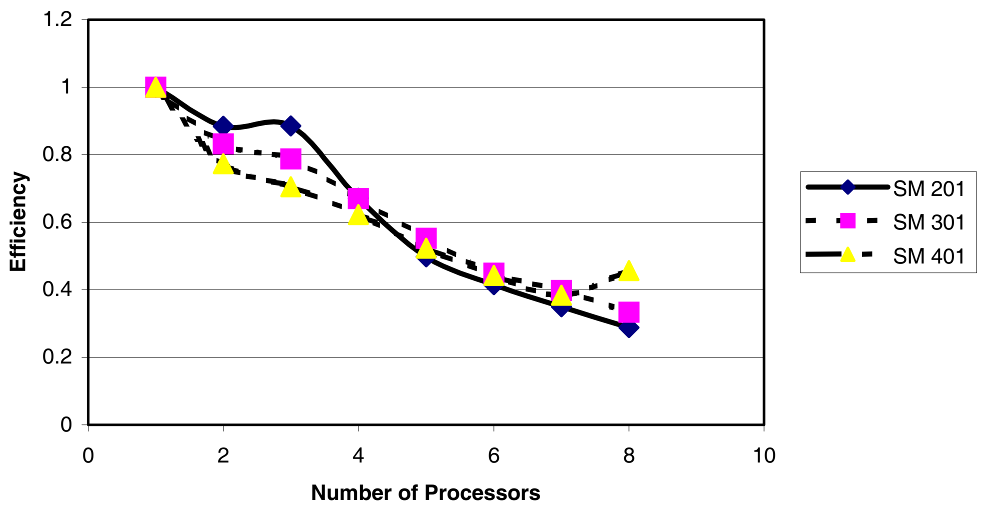

Figure 6. Shared memory efficiencies for the broken dam problem. SM 201, 201 × 201 grid; SM301, 301 × 301 grid; SM401, 401 × 401 grid.

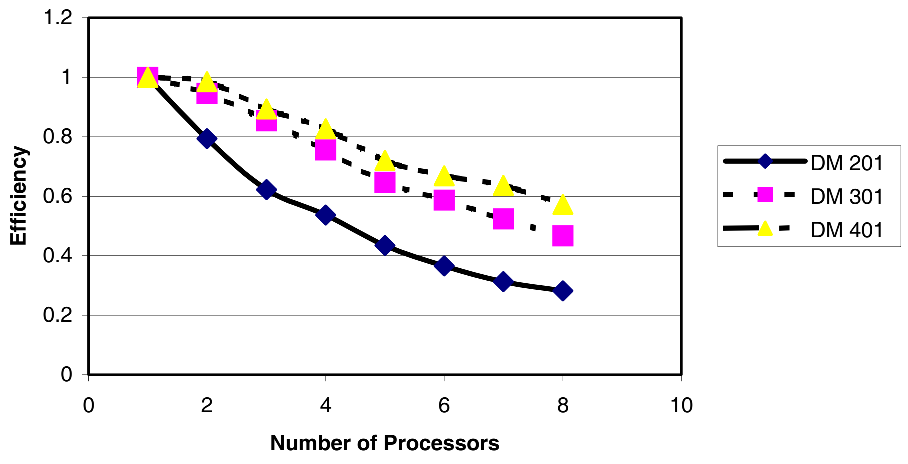

Figure 7. Distributed memory efficiencies for the broken dam problem. SM 201, 201 × 201 grid; SM301, 301 × 301 grid; SM401, 401 × 401 grid.

The trends in efficiency are similar to those obtained with the shared memory machines. For the heat transfer problems, the cluster efficiencies trend better for shared memory, while for the broken dam problem the efficiencies are better for the cluster. In both cases the scaling improves as the problem size increases as would be expected.

We have plotted the efficiencies for both the shared memory and cluster cases in Figures 4–7. As can be seen the shared memory results display some unusual non-monotonic behavior. This is not completely understood, although we have reason to believe that it is an artifact of our shared memory machine and not the CartaBlanca program.

While we have not performed side-by-side a comparison of scaling behavior seen with CartaBlanca to, say, a parallel FORTRAN code, we can say that the trends reported here are qualitatively similar to results we have seen in the past with such FORTRAN codes. That is, in general, the efficiencies improve, as the problem size is increased for fixed number of partitions. This is expected since as problem size increases, the ratio of communication operations to on-processor work decreases. That is, for a fixed number of mesh partitions and processors, the ratio of the number of nodes on the inter-partition boundaries to the number of nodes interior to the partitions decreases and thus the ratio of communication to work done solely on each of the processors decreases. We thus see an improvement in efficiencies for a fixed number of processors as the problem size increases.

5. CONCLUSIONS AND FUTURE PLANS

We conclude that for both shared memory computers and distributed memory clusters we achieve similar scalability with CartaBlanca on the test problems. For the case of distributed memory clusters, only minor modifications were necessary to allow CartaBlanca to run in parallel. What is very encouraging is that the simplicity of the shared memory communication algorithm can be retained in going to distributed-memory computations. This simplicity keeps the burden of the development and maintenance of the computation software from the physics developer in keeping with the advantages of the Java paradigm. We plan to continue to pursue modifications to improve scalability. This will be followed by testing the ability of the CartaBlanca–JavaParty combination on a parallel problem of a much larger scale. We will also make quantitative comparisons to scaling from traditional FORTRAN codes [27].

ACKNOWLEDGEMENTS

We gratefully acknowledge the support for this work from the Department of Energy and the ASCI Institutes program. We also thank Bernhard Haumacher from the University of Karlsruhe for his help with JavaParty.

REFERENCES

1. JUnit. http://www.JUnit.org/.
2. Boisvert RF, Moreira J, Phillipsen M, Pozo R. Java and numerical computing. *Computing in Science & Engineering* 2001; **3**(2):18–24.
3. Reinholtz K. Java will be faster than C++. *ACM Sigplan Notices* 2000; **35**(2):25–28.

4. Philippsen M, Zenger M. JavaParty: transparent remote objects in Java. *Concurrency Practice and Experience* 1997; **9**:1225–1242.
5. YU W, Cox A. Java/DSM: a platform for heterogeneous computing. *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997. *Concurrency Practice and Experience*, 1997; **9**(11). http://www.cs.rice.edu/∼weimin/papers/java97.ps.
6. Launay P, Pazat J-L. A framework for parallel programming in Java. *Publication Interne No. 1154*. http://www.irirsa.fr/bibli/publi/pi/1997/1154/1154.html.
7. Thiruvathukal GK, Thomas LS, Korczynksi AT. Reflective remote method invocation. *Concurrency: Practice and Experience* 1998; **10**(11–13):911–926.
8. Veldema R, vanNieuwpoort R, Maassen J, Bal HE, Plaat A. Efficient remote method invocation. *Technical Report IR-450*, 14, September 1998. http://www.cs.vu.nl/manta.
9. Alpern B *et al*. The Jalapeno Virtual Machine. *IBM Systems Journal* 2000; **39**(1).
10. Aridor Y, Factor M, Teperman A, Eilam T, Schuster A. Tranparently obtaining scalability for Java applications on a cluster. *Journal of Parallel and Distributed Computing (Special Issue Java on Clusters)* October 2000. http://www.haifa.il.ibm.com/projects/systems/cjvm/papers.html.
11. Veldema R, Hofman RFH, Bhoedjang RAF, Jacobs CJH, Bal HE. Jackal: a compiler-supported distributed shared memory implementation of Java. *Proceedings of Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, 18–19, June 2001. http://www.cs.vu.nl.manta.
12. Ma MJM, Wang Cho-li, Lau FCM. Jessica:Java-enabled single-system-image computing architecture. *Journal of Parallel and Distributed Computing* 2000; **60**(10):1194–1222.
13. Baker M, Carpenter B, Fox G, Ko SungHoon, Lim S. MPIJava: an object-oriented Java interface to MPI. *At International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999*, San Juan, Puerto Rico, April 1999. http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html.
14. Ma RKK, Wang Cho-Li, Lau FCM. M-JavaMPI: a Java-MPI binding with process migration support. *The Second IEEE/ACM Internationl Symposium on Cluster Computing and the Grid, CCGrid* 2002. http://www.csis.hku.hk/∼clwang/projects/M-JavaMPI.html.
15. Northeast Parallel Architectures Center, Syracuse University. http://www.npac.syr.edu/users/gcf/pdptajune99/foilsephtmldir/072HTML.html [1999].
16. Dincer K. JMPI and a performance instrumentation analysis and visualization tool for jmpi. *First UK Workshop on Java for High Performance Network Computing, EUROPAR-98*, UK, 2–3, September 1998. http://www.cs.cf.ac.uk/hpjworkshop/
17. Antoniu G, Bouge L, Hatcher P, MacBeth M, McGuigan K, Namyst R. The Hyperion system: compiling multithreaded Java bytecode for distributed execution. *Parallel Computing* 2001; **27**:1279–1297.
18. http://www.par.univie.ac.at/project/javasymphony.
19. VanderHeyden WB, Dendy ED, Padial-Collins NT. CartaBlanca- A Pure-Java, Component-based systems simulation tool for coupled nonlinear physics on unstructured grids. *Proceedings for the ACM 2001 Java Grande/ISCOPE Conference*. ACM Press: New York, 2001.
20. Ferziger JH, Peric M. *Computational Methods for Fluid Dynamics*. Springer: New York, 1999.
21. Brown PN, Saad Y. Hybrid Krylov methods for nonlinear systems of equations. *SIAM Journal of Scientific and Statistical Computing* 1990; **11**(3):450–481.
22. Karypis G, Kumar V. METIS a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, version 4.0. University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN. http://www-users.cs.umn.edu/∼karypis/metis/index.html.
23. Saad Y. *Iterative Methods for Sparse Linear Systems*. PWS Publishing: San Fransisco, 1995.
24. Hauser J, Ludewig T, Gollnick T, Winkelman R, Williams R, Muylaert J, Spel M. A pure Java parallel flow solver. *AIAA paper 99-0549*.
25. Hatakeyama M, Watanabe M, Suzuki T. Object-oriented fluid flow simulation system. *Computers & Fluids* 1998; **27**(5):581–597.
26. Riley C, Chatterjee S, Biswas R. High performance Java codes for computational fluid dynamics. *Proceedings of the ACM 2001 Java Grande/ISCOPE Conference*, Palo Alto, 2001. ACM Press: New York, 2001.
27. Kashiwa BA, Padial NT, Rauenzahn RM, VanderHeyden WB. A cell-centered ICE method for multiphase flow simulations. *Numerical Methods in Multiphase Flows* (*FED*, vol. 185), Crowe CT, Johnson R, Prosperetti A, Sommerfeld M, Tsuji Y (eds.). ASME: New York, 1994.
28. Dendy ED, Padial-Collins NT, VanderHeyden WB. A general purpose, finite-volume advection scheme for continuous and discontinuous fields on unstructured grids. *Journal of Computational Physics* 2002; **180**(2):559–583.
29. Drew DA, Passman SL. *Theory of Multicomponent Fluids*. Springer: New York, 1999.
30. Kashiwa B, Padial NT, Rauenzahn RM, VanderHeyden WB. A cell-centered ICE method for multiphase flow simulations. *Numerical Methods in Multiphase Flows, ASME* 1994; **185**:159–176.
31. Kelly CT. *Iterative Methods for Linear and Nonlinear Equations*. SIAM: Philadelphia, 1995.

32. Martin JC, Moyce WJ. An experimental study of the collapse of liquid columns on a rigid horizontal plane. *Philosophical Transactions of Royal Society* 1952; **A244**:312–324.
33. Oaks S, Wong H. *Java Threads*. O'Reilly: Cambridge, 1999.
34. O'Rourke PJ, Sahota MS. CHAD: a parallel, 3-D. implicit, unstructured-grid, multimaterial, hydrodynamics code for all flow speeds. *Los Alamos National Laboratory Report LA-UR-98-5663*, October 1998.
35. ProActive, a Java library for parallel, distributed and concurrent computing with security and mobility. http://www-sop.inria.fr/oasis/ProActive [2002].
36. Schatzman J, Donehower R. High-performance Java software development. *Java Report* 2001; **6**(2):24–41.
37. Selmin V. The node-centered finite volume approach: bridge between finite differences and finite elements. *Computing Methods in Applied Mechanical Engineering*, 1993; **102**(1):107–138.
38. Shirazi J. *Java Performance Tuning*. O'Reilly: Cambridge, 2000.
39. Veldema R, Hofman RFH, Bhoedjang RAF, Jacobs C, Bal HE. Source-level global optimizations for fine-grain distributed shared memory systems. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 18–20 June 2001, Snowbird, UT.
40. Veldema R, Hofman RFH, Bhoedjang RAF, Bal HE. Runtime optimizations for a Java DSM implementation. *Proceedings for the ACM 2001 Java Grande/ISCOPE Conference*. ACM Press: New York, 2001.
41. http://gcc.gnu.org/Java/.