AJaPACK: Experiments in Performance Portable Parallel Java Numerical Libraries

Shigeo Itou, Satoshi Matsuoka, Hirokazu Hasegawa
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku, Tokyo, Japan
{itou,matsu,b962220}@is.titech.ac.jp

ABSTRACT

Although Java promises platform portability amongst diverses et of systems, for most Java platforms today, it is not clear if they are appropriate for high-performance numerical computing. In fact, most previous attempts at utilizing Java for HPC sacrificed Java's portability, or did not achieve necessary performance required for HPC. Instead, we propose an alternative methodology based on *Downloadale Self-tuning Library*, and constructed an experimental prototype called AJaPACK, which is a portable and high-performance parallel BLAS library for Java which "tunes" itself to the environment to which it is installed upon. Once AJaPACK is downloaded and executed, the Java version of ATLAS (ATLAS for Java) and the parallelized version of JLAPACK combine to achieve optimized pure Java execution for the given environment. Benchmarks have shown that AJaPACK achieves approximately 1/2 to 1/5 of the speed of optimized C-ATLAS and vendor supplied BLAS libraries, and with portable parallelization in SMP environments, achieves superior performance to single-threaded C-based native libraries. This is an order of magnitude superior w.r.t. performance compared to previous pure Java BLAS libraries, and opens up further possibilities of employing Java in HPC settings, but still shows that JIT compilers with optimizations expecting numerical code highly-tuned at the source- or bytecode level would be highly desirable.

1. INTRODUCTION

Distributed and high-performance computing ares becoming much more synergetic thanks to the widespread availability of high-performance networks and inexpensive computing nodes such as PC clusters. In such an environment, HPC applications and libraries highly-portable across a variety of highly-divergent execution platforms are in absolute need. Traditional HPC languages such as C or Fortran do not completely standardize the language, the library, the machine binary, the parallel machine architecture, nor the execution environment. As a result, it is quite difficult to have portable HPC applications that work ubiquitously in a distributed environment. Although efforts such as HPF and OpenMP strive to achieve some portability of parallel code, a single binary working across different parallel machines is still unavailable.

Java has received considerable attention in the HPC community, thanks to its good object-oriented language design, as well as support of standard language features for parallel and distributed computing such as threads and RMI. Moreover, standardized Java bytecode supports portable execution via the Java Virtual Machine. On the other hand, it is not clear whether Java AS IS is entirely appropriate for portable distributed HPC programming, as pointed out by various efforts including those by JavaGrande. In particular, Java execution environment is quite divergent, ranging from pure interpretation, dynamic JIT compilation, to static compilation; as a result, an optimization scheme for a particular Java platform may not be well-applicable for others. This includes not only the compiler, but also the run-time system.

We claim that the important characteristics for HPC in an network environment is "performance portability" for code that are downloaded and move across the network. By performance portability we do not mean that a given code must perform the same across divergent machines; rather, it is a property whereby the same code will perform within some certain fraction of the peak performances of different machines. For example, when a certain optimization is performed on the code, if the code exhibits measurable performance improvement over different machines, then the optimization is performance portable. For Fortran and C, this is typically the case, but for Java, whether this is the case is not clear, and in fact several counter cases have been reported [1].

From this perspective, we categorize several previous approaches employed for HPC computing in Java:

- **Employing Native Libraries:** A HPC native library is called, using interfaces such as JNI. Kernel numerical performance obviously matches that of C or Fortran, but portability naturally suffers, and moreover, maintaining multiple native code for downloading would be quite cumbersome. Examples include JCI(Java-to-C interface generator)[2] and the frontend wrapper in IceT[3].

- **Optimizing (Static) Compiler:** Traditional optimizing compiler as is with C and Fortran. Examples include IBM High Performance Compiler for Java[4] and Fujitsu HBC[6]. In this case, one obtains utmost

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Java 2000 San Francisco CA USA
Copyright ACM 2000 1-58113-288-3/00/6...$5.00

efficiency, sometimes almost matching that of optimizing Fortran[4]. On the other hand, not only portability suffers, because it only speeds up program on a particular platform, but it becomes more difficult to implement the dynamic aspects of Java, such as dynamic loading, security checks, reflection, etc. In fact, as far as we know both compilers do not support the official full Java spec in this respect.

- **Automated translation from other programming languages to Java:** There are several tools that translate existing code in traditional HPC languages, such as Fortran, into Java. An example is f2j[7], where Fortran LAPACK is automatically translated into Java in a source-to-source fashion. However, in order to implement Fortran semantics, the translated source embodies various artifacts such as "pseudo-gotos", and as a result, becomes rather difficult to understand for further tuning. Moreover, as it does not take into account specific Java features, performance for pure Java code is low compared to the original native Fortran.

- **Manual Tuning:** We tune a particular Java HPC code at either source- or bytecode level by hand, leaving the low-level optimization to the JIT compiler. Examples include the evaluation of Daxpy and other BLAS implementations by Pozo[8]. Although portability is quite high with this approach, there is no guarantee that performance portability is achieved on different platforms, due to the divergent nature of Java platform implementations as mentioned above.

In order to achieve performance portability of Java numerical code, we are investigating the construction of a self-tuning library and compilation framework for Java. Basically, we obtain best performance on each Java execution platform by automatically tuning for that particular platform at the source- and bytecode level. This allows leveraging of existing JVMs and JIT compilers, while still obtaining the best speed achievable for that particular platform. More concretely, we implement the compiler/code-generator, performance monitor, code tester, and the high-level glue library, etc. entirely in Java, and when a particular library is downloaded over the network for the first time, not only the library class file itself but the entire self-tuning framework is downloaded, and optimization will occur either on the spot or off-line when the machine utilization is low. Moreover, we define or generate parallel multithreaded code when possible, thanks to the portability of Java threads. Although a similar strategy has been attempted in various settings, more recently in efforts such as ATLAS and PhiPAC we describe below, with Java it would be easy to make the entire process automated, without user intervention. Furthermore, self-tuning library is more significant for Java, again due to relatively divergent performance characteristics of Java execution environment.

On the other hand, there are several technical challenges that are open questions, mainly regarding the feasibility of the approach:

- **Ubiquitous Effectiveness of Self-tuning on Various Java Platforms:**

  Although previous work on adaptive compilation, as well as more aggressive self-tuning libraries have shown success for Fortran or C such as PhiPAC[9] and ATLAS[10], often achieving the performance of vendor-tuned libraries, it is not clear whether the same strategy would be effective for Java platforms. In fact, there are various possibilities where traditional optimization strategies applicable to Fortran and C would not be effective, we must investigate whether self-tuning will (or will not) perform well under different Java platforms. A partial study for the L1-blocked BLAS core has been done in [1], but a study using a larger-grained library with parallel execution is required.

- **Effective Architecture for Self-tuning Libraries:** Since libraries are somewhat persistent and used multiple times over different applications, it could afford longer time durations for tuning. However, when libraries are huge, tuning time may still outweigh the gain in execution time of the application utilizing the library. Thus, as is with traditional libraries, the tuning architecture would ideally identify the kernel routines that would be most beneficial, and leave the non-kernel, higher-level routines to standard JIT compiler optimization. It is not clear, however, how much the slower execution of higher-level routines due to Java (such as non-aliasable, non-contiguous array semantics often requiring array copies) would penalize the overall library performance.

- **Use of Java Threads for Large-scale HPC:**

  Also, although Java is multithreaded at the programming language level, it is not clear how much parallel multithreading will scale, especially w.r.t. numerical computation. Early versions of Java only supported "green threads" which were merely coroutines. Many recent versions of Java are truly multithreaded (native threads), but various research has shown the substantial overhead associated with multithreading in Java, and proposed various solutions (Six papers on Java thread synchronization appeared in recent OOPSLA'1999[11]). All such research we know to date, however, optimize cases where the threads do NOT synchronize, eliminating or minimizing the cost of synchronization. It is not clear how scalable Java performance is for highly-tuned numerical code, especially for code which DO synchronize, and will suffer from other overheads such as thread scheduling.

In order to investigate whether downloadable self-tuning libraries are feasible for attaining performance portability, we are constructing AJaPACK, a prototype self-tuning linear algebra package for dense matrices, as a proof-of-concept experimentation. AJaPACK employs our pure-Java port of ATLAS[10], called *ATLAS for Java*. ATLAS for Java, as is with the original ATLAS, allows generation of optimized L1-blocked small-matrix kernel Level 3 BLAS code for each Java platform. Experiments show that, on various platforms, ATLAS for Java benefits from the L1 blocking optimization, and exhibits 1/2 to 1/4 the speed of C-version of ATLAS on the best JIT compilers for the particular hardware.

In order to implement a higher-level linear-algebra routine on top of ATLAS for Java, we modified and extended JLAPACK/Harpoon[12] significantly, as well as parallelizing some of the routines, notably gemm(). This allowed several factors of performance increase compared to the original JLAPACK/Harpoon implementation. Parallelization of

gemm() was done in several ways, including static and dynamic job partitioning and thread allocation, and tested its scalability on several different large SMP platforms. Benchmark results show that, with static parallelization, we obtain high scalability of gemm() code on some platforms; however, dynamic job allocation did not scale as well. Moreover, we found that some platforms have relatively poor support for multithreading, resulting in performance loss due to parallelization. We believe this is because Java in general is still in development stage for scalable parallel programming—as such, support for low-overhead context switching, effective memory management in the context of parallel processing including parallel garbage collection, etc. will need ubiquitous deployment on all Java platforms.

2. AJAPACK: AUTOMATIC PARALLEL TUNING LIBRARY FOR PARALLEL BLAS IN JAVA

2.1 Overview of Downloadable Self-Tuning Library and AJaPACK

We first overview the general architecture of downloadable self-tuning library; as seen in Figure 1 this is composed of five components, all written in Java.

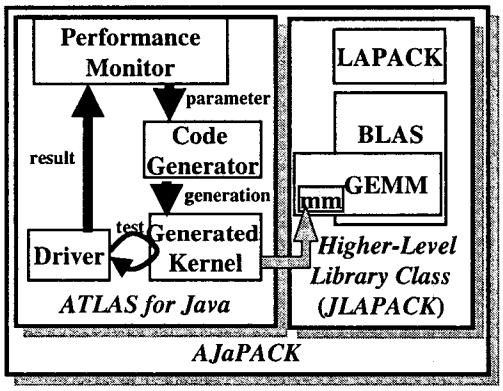

Figure 1: Overview of Downloadable Self-tuning Library in Java

- Code Generator/Compiler: Generates performance critical portion of the library. Ideally, we employ general compiler framework such as our OpenJIT[13] for this purpose as a toolkit framework to easily construct a customized code generator for each library. Since AJaPACK is a proof-of-concept prototype, however, we directly targeted dense linear algebra computation, using the code generator of ATLAS for Java as well as additional code for feasibility experimentation.
- Generated Kernel: The numerical kernel generated by the code generator. The driver tests the performance and the optimal one is automatically embedded into the higher-level library classes.
- Driver: The driver module that tests the performance of the Generated Kernel. Reports the measured performance to the performance monitor.
- Performance Monitor: Receives report from the driver on performance of each generated kernel routine, and feeds back performance to the code generator, guiding generation of additional test kernel code. As such the performance monitor is parameterized by a search heuristics, and guides the pruned search of possibly optimal kernel code.
- Higher-Level Library Classes: Parts of libraries that Provide higher-level APIs, but are not on the performance-critical path of the library. By embedding the generated kernel tuned to the particular Java platform, the library executes at the optimal performance possible.

We owe much of the higher-level architecture as a generalization of ATLAS. Again, we do feel that automated code tuning has more than the importance that ATLAS had for C/Fortran for several reasons including those mentioned earlier, namely:

1. Automated downloading and execution of the entire self-tuning library is easy because of Java portability.
2. Java platform is much more divergent, and performance portability is more difficult to achieve with mere static code optimization.
3. As mentioned earlier, manual tuning of Java numeric code is difficult due to language design and implementation features, and it is better to resort to automated means.
4. As the tuning occurs at source-code or bytecode level, it can readily adapt to cases where the underlying architecture changes, for with Java it is easy to maintain the downloaded code in portable format.
5. Since the validity of the generated kernel code is checked by the Java security checker, the user will have better confidence that whatever code dynamically generated will be safer than Fortran or C code.

In AJaPACK, each component has the following implementation with respect to the systems we have ported or developed (to be described in detail later):

- Code Generator/Compiler: The code generator portion ATLAS for Java, namely parts of xemit.java and xmmsearch we see in Figure 2.
- Generated Kernel: The small L1 cache-blocked Level 3 BLAS in Java which xemit.java generates. Currently, we emit the source file which is compiled by javac, although in principle direct generation of bytecode is also possible.
- Driver: Corresponds to fc.java of ATLAS for Java.
- Performance Monitor: Corresponds to parts of xmmsearch.java in ATLAS for Java.
- Higher-Level Library Classes: As mentioned earlier, we employed JLAPACK/Harpoon[12] by Chatterjee et al., to work with ATLAS for Java, and furthermore parallelized the BLAS routine. We describe the details of parallelization and their implications in later sections.

2.2 Overview of the Original ATLAS

The original ATLAS(Automatically Tuned Linear Algebra Software) [10] is a source-level automated tuning tool for BLAS, being developed by Whaley and Dongarra at University of Tennessee. ATLAS searches for and finds the most appropriate L1 cache blocking factor, number of unrolls, software pipelining latencies, and generates the best C-based BLAS Level 3 code for a given platform.

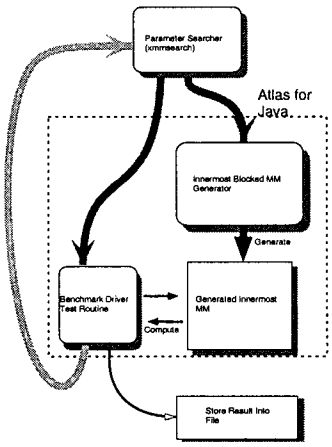

Figure 2: Architecture of ATLAS (v.1.0)

The parameters employed in ATLAS v.1.0[1] are basically muladd, NB, MU, NU, KU, and LAT. Muladd indicates where fused multiply-add is available. NB is the L1 cache blocking size, and MU, NU, and KU are number of unrolls for each loop, and LAT is the latency factor for software pipelining. Given such parameter space, ATLAS generates the kernel cache-blocked code for each parameter, tests the performance, and finally outputs the one with the best performance. In order to prune the search space, ATLAS employs a prescribed search strategy[10].

Although the search space is considerably pruned, ATLAS execution is still quite expensive, and such extensive optimization is difficult to embed in a compiler. However, for libraries, such cost could be amortized over multiple execution of the library, and in fact it is reported that ATLAS generated code matches the best results by the vendor-supplied optimized BLAS library.

We also note here that ATLAS represents all matrices in one-dimensional form, as required by LAPACK. This is an advantage for Java, as it avoids several overhead issues relevant to performance of array accesses, including non-rectangular representation of multi-dimensional arrays, and requirement to check array bounds for each dimension.

2.3 ATLAS for Java

ATLAS for Java is a port of ATLAS v.1.0 onto Java. Because we have the Java VM intervene between the native CPU and ATLAS, what we obtain by automated tuning is the best possible performance for the particular Java platform, and not for that particular hardware architecture. Still, this in a sense satisfies performance portability, as it is the best performance that the particular Java platform can produce without resorting to non-portable means such as native methods.

[1]The current version of ATLAS is 3.0.

```
class mm{
void dNBmm(myarray ma, int ldc){
      do /* N-loop */{
         do /* M-loop */{
            indexb = indexB;
            c00_0 = 0.0;
            b0 = ma.B[indexb];
            a0 = ma.A[indexA];
/* Start floating point pipe */
            m0 = a0 * b0;
            b0 = ma.B[indexb + 1];
            a0 = ma.A[indexA + 1];
/* easy loop to unroll */
            for (k=0; k > 0; k--) {
               indexA++;
               indexb++;}
/* Do last iteration of K-loop, and drain the pipe */
            c00_0 += m0;
            m0 = a0 * b0;
            c00_0 += m0;
            indexA += 2;
            ma.C0[C0index] += c00_0;
            C0index ++;}
         while(indexA != stM);
         indexA = ma.indexa;
         C0index += incC;
         indexB += 2;}
      while(indexB != stN);}
```

Figure 3: Sample code generated by ATLAS for Java (Not Optimal)

ATLAS for Java is entirely written in Java to work portably across all standard JDK environments that facilitate standards tools such as javac. It basically employs the same search algorithm as the original ATLAS, employing cache blocking, loop unrolling and software pipelining.

The original ATLAS makes extended usage of pointer passing for arrays and scalars. Because Java does not have pointers to scalars or array interiors, we defined a wrapper class which embodies the array indices as well as the contents of the array in one-dimensional form. As objects are passed by reference, we can pass all the arguments at once. We also performed initial analysis to verify that use of object field access was being compiled away and not causing overhead. The same applies for normal scalars, where the argument is being used as an in-out parameter; again, we defined wrapper classes, and made sure that no overhead will occur due to this indirection.

Figure 3 illustrates the sample code generated by ATLAS for Java. We note that arrays are referenced via fields, but otherwise local variables are employed exclusively in hopes that it will be mapped to registers, just as is with C and Fortran. We also note that the parameters for this code is far from the optimal actually found; on modern-day processors with deep pipelining and large L1 caches, inner loops are unrolled over 30 times, and the generated classfile typically exceeds 200 Kilobytes.

2.4 AJaPACK High-Level Library Class — JLAPACK/Harpoon

For AJaPACK's high-level library class API, we considered the available LAPACKs in Java. In order to achieve portable performance, we also undertook parallelizing the code so that multiprocessing platforms can automatically take advantage of parallel Java threads.

We found two good Java LAPACKS available; one me-

chanically converts the LAPACK written in Fortran into equivalent Java code[7]. Thus, the resulting code is not object-oriented, but rather, Java is employed as an intermediate target language. We considered this to be somewhat inappropriate, as it is difficult to have clean API between the optimized kernel versus the higher-level classes. Moreover, as far as we found not all of LAPACK routines execute correctly, likely due to mechanical translation.

The other one we considered is the JLAPACK/Harpoon. Its architecture is well-designed with appropriate object encapsulation of array types and operations. The problem is that only a very few LAPACK routines are actually implemented (gesv, getrf, getrs, getf2, and laswp; by contrast, f2j implements all 288 LAPACK routines). For the purpose of our research on producing an prototype proof-of-concept, we decided to use JLAPACK, due to its clean object-oriented API, plus stable and correct operations.

The original JLAPACK consists of the 3 packages below:

- JLAPACK
- JBLAS
- JLASTRUCT

Packages JLAPACK and JBLAS are port of LAPACK into Java, and the BLAS into Java, respectively. Package JLASTRUCT adds various helper and glue methods where a mere port of the Fortran LAPACK is insufficient.

For AJaPACK, we reimplemented JBLAS so that JLAPACK calls the optimized kernel generated by ATLAS for Java. More concretely, we changed the xBLAS classes (where x is D, C, etc.), in particular the xgemm() methods so that it performs optimized blocking, and calls the kernel routines. Moreover, we wrote glue code so that the kernel routines produce appropriate the descriptor object for arrays employed by JLAPACK. This turned out to be not trivial, as considerable modifications were required for ATLAS for Java, as well as requiring good amount of glue code for interfaces.

We also are writing additional routines available in LAPACK, such as LU decomposition as well as parallelizing gemm() and additional algorithms. For LU, we are implementing both sequential and parallel versions of blocked LU (dgetrf) and recursive algorithm by Gustavson[14] (rgetf2). Both employ gemm() for blocked operations.

There are several minor notes in the implementation. First, the the blocked subarrays always need to be copied, as no aliasing is possible with Java. Instead of generating new submatrices every time (typically about 36 by 36), we try to reuse the submatrices avoiding object allocation and deallocation [2]. Another minor note is when arrays do not exactly match the block size. In order to always employ the fast kernel code, we fill the non-used portion with 0. A simple strategy is to dynamically determine this, but instead we generate the boundary blocks once, and cache them. Although this uses $O(n)$ memory for $n \times n$ arrays, in our benchmarks this strategy was 10–30% faster, but in some cases the inverse was seen. We do provide a switch so that dynamic filling can be used for memory-tight situations.

3. PARALLELIZING AJAPACK

The advantage of Java for performance portability is multithreading at the language level. In fact, it is not just the language specification itself, but also that the entire system, including the run-time system, the libraries, profilers and debuggers, etc. assume that the language is multithreaded. This is important for ubiquitous multi-threaded parallel execution e.g., the libraries and JVMs are designed to be reentrant. Inherent support for parallelism is what distinguishes Java from C or Fortran, where parallel execution is added as an afterthought, and its safety is not at all guaranteed across all systems, libraries, compilers, etc.

In order to achieve good performance in a portable way, downloadable self-tuning libraries should exploit the availability of multiple CPUs on SMPs wherever possible. On the other hand, characteristics of truly parallel execution on all Java platforms, especially their scalability on intensive scientific code, have not been well-established.

We explored whether parallelization of AJaPACK would be feasible using Java threads across all platforms. Parallelization for BLAS is well known, especially with subarray blocking, since if A × B is performed as summation of $A_{ij} \times B_{jk}$, then each $A_{ij} \times B_{jk}$ can be executed independently in parallel. The question is rather, what *style* or a *method* of parallel programming would be appropriate for achieving proper scalability. So, for BLAS we implemented three most likely styles for multithreaded parallelism, and compared their performance on SMPs ranging from 2 to 60 processors on various Java platforms:

Fine-grained Master-Worker (FMW)

We generate a prescribed number of worker threads (typically number of processors + 1–2), and each worker thread on each iteration requests work from the master worker queue, computes the product, stores the result, and repeats the sequence until all the work are exhausted. For fine-grained we allocate a single small blocked matrix multiply per each worker. Although this seems too fine-grained, it eliminates various overhead, as in Java the arrays must be copied up-front as there is no aliasing. Synchronization occurs on fetching work from the master queue, and when the result is written back to the product matrix.

Coarse-Grained Master-Worker (CMW)

Similar to FMW, but the worker acquires multiple tasks at once, increasing granularity. In order to reduce synchronization costs, we stage the execution into three phases, namely array copying, matrix multiply, and writing back of all the results. Although this may seem more efficient than FMW, in practice it could add overhead due to the extra complexity involved.

Statically-Decomposed Fork-Join (SFJ)

Since matrix multiply is deterministic, we decompose the outermost I-loop, and allocate tasks to each forked worker in a balanced fashion. The program needs no synchronization for blocked distribution except for the outermost fork-join, as writebacks into the product array are independent.

More sophisticated strategies such as workstealing amongst multiple work queues, are possible. Of the current three, Master-Worker parallelization may incur more overhead, especially for BLAS where it is relatively easy to perform static decomposition, but for arbitrary problems load balancing occurs naturally, and is thus more flexible. Since it was obvious that SFJ would win out, we compared FMW and

[2]Note that this does preclude the exploitation of Java semantics for the 0 filling, as newly allocated arrays are guaranteed to contain 0

CMW against SFJ to see if they would scale equivalently or not, exhibiting the cost of Java multithreading overhead. These will become apparent in the next section.

4. PERFORMANCE BENCHMARK

We measure the performance of AJaPACK, how the kernel tuned by ATLAS for Java compare against the C kernels tuned by the original ATLAS (v.2.0), and also explore how much overhead the higher-level library APIs will sacrifice performance, for each platform. We also investigate how much of the performance could be recovered by parallel execution, and how they scale. We also explore what style of parallelism is effective for implementing BLAS, as well as LU factorization.

4.1 Evaluation Environment and Methodologies

Our previous work[1] showed that x86 platforms exhibited the best sequential performance for Java. We re-tested several JIT compilers, and chose the IBM JDK 1.1.8 which includes a tuned and heavily modified version of the Sun's original JVM, and also a high-performance JIT compiler, which exhibited the best performance in our initial tests.

For Sparcs, we employed Sun's Research VM (Solaris 7 Production Release JVM), and turned on the flag to enable the optimizing JIT compiler, which is claimed to be faster than the default JIT (`java -Xoptimize`). According to our measurements, Sun's Hotspot was much slower with respect to numerical code, and in fact the adaptive compilation in the new release failed to get turned on in ATLAS for Java, resulting in performance around 1MFlops.

We also tested several SMP and CC-NUMA platforms, namely the Ultra Enterprise Servers, and the Origin 2000s. As mentioned above, these are shown not to have the best sequential numerical execution speed, especially in comparison to the C counterparts. On the other hand, they claim to support high-performance native threads, which the JVMs should be exploiting for server-style applications.

- PC Platforms
  - Dual PIII PC (Pentium III 450MHz x 2) procs + Linux Redhat 6.0
    - L1 Cache Inst-16KB + Data-16KB
    - L2 Cache 512KB
    - IBM JDK-1.1.8 JVM with optimizing JIT compiler
  - Athlon PC (Athlon 600MHz) + Linux Redhat 6.0
    - L1 Cache Inst-64KB + Data-64KB
    - L2 Cache 512KB
    - IBM JDK-1.1.8 JVM with optimizing JIT compiler
- SMP Platforms
  - Sun Enterprise4000(UltraSPARC 300MHz x 8 procs) + Solaris 2.6
    - L1 Cache Inst-16KB + Data-16KB
    - L2 Cache 1MB
    - Solaris Production Release JVM 1.2 with optimizing JIT compiler (JBE)
  - Sun Enterprise 10000(StarFire))(UltraSparc 250Mhz x 60 procs) + Solaris 2.6
    - L1 Cache Inst-16KB + Data-16KB
    - L2 Cache 4MB
    - Solaris Production Release JVM 1.2 with optimizing JIT compiler (JBE)
  - SGI Origin2000(R10000 250Mhz x 16 procs) + IRIX
    - L1 Cache Inst-32KB + Data-32KB
    - L2 Cache 4MB
    - SGI JDK 1.2.1

For all the platforms, we performed the benchmarks with the following methodologies:

- We tested the basic kernel performance as reported by xmmsearch, the gemm() performance and the performances of blocked and blocked recursive LU.
- As a reference, we tested C performance for the original ATLAS and the ATLAS-enhanced C-LAPACK against AJaPACK.
- For Java gemm() and LU factorizations, we tested sequential and parallel versions. For gemm(), we test the three methodologies for parallelization, namely SFJ, FMW, and CMW. (The parallel LU factorization is in early development, and is not fully optimized as we see in the benchmarks.)
- For all benchmarks, we vary the sizes of the problem matrices. For parallel benchmarks, we also vary the number of threads. The reported performance are the best scores achieved for respective parameters. For example, when we vary the matrix size for the parallel versions, the reported score is the best amongst different number of threads benchmarked.

4.2 Overview of Results

Table 1 shows the peak performance achieved by each library. As we see, sequential performance differs for the type of CPU, and the Java platform employed. On Athlon, we obtain the best score for xmmsearch.java, where we record nearly 300MFlops, which is approximately about 1/2 of C performance. The same is true for Pentium III, but the performance is lower, although still being 1/2 of C. However, for SPARCs, the Java score is lower relative to C, being approximately 1/3 of C performance. Similar phenomenon was seen for the Origin 2000.

We also see a sharp decline in performance when we move to higher-level class libraries, in contrast to C. In fact, we observe approximately 40–60% performance drop compared to xmmsearch performance, whereas for C, we observe much smaller or very little performance penalty. Preliminary investigation for the cause revealed that, this is attributable to overhead imposed by the the object-oriented nature of the library. Because gemm() generally accepts various forms of underlying data representations of the matrix (such as being transposed), copying of the portions of the matrix to L1-blocked subarray requires sophisticated translations. In fact, the current version of AJaPACK employs element-wise copying by specifying higher-level matrix (not array) indices, incurring invocation of the accessor method for each element copied. We initially assessed that the copying cost would be considerably smaller compared to actual computation of the L1-blocked subarray, and that the compiler would compile away the method call and and the overhead, but this was not the case. For future versions we plan to add methods to copy the subarray all at once, according to each underlying matrix data representation.

For blocked LU, we see that for C the Gustavson's recursive algorithm is generally superior to standard, non-recursive algorithm. On the other hand, for Java, the re-

Table 1: Summary of Peak Performances Achieved by Each Library

Peak(MFlops)	E4KC	E4KJ	E10KC	E10KJ	PIIIC	PIIIJ	AthlC	AthlJ	O2KC	O2KJ
xmmsearch	401.9	132.9	281.2	110.4	375.6	171.7	570.5	296.0	330.5	81.95
GEMM seq.	321.5	52.10	286.0	52.6	325.0	102.6	555.7	165.1	340.9	43.04
GEMM par.	—	349.9	—	1365.4	—	77.3	—	112.5	—	487.3
LU block seq.	216.2	47.25	—	—	216.6	87.40	298.5	140.9	—	—
LU block par.	—	152.9	—	—	—	98.0	—	—	—	—
LU recur. seq.	250.8	34.23	—	—	273.1	36.70	399.8	86.40	—	—
LU recur par.	—	58.85	—	—	—	79.4	—	—	—	—

E4K = Sun Enterprise 4000 (UltraSparc 300MHz x 8)
E10K = Sun Enterprise 10000 (UltraSparc 250MHz x 60)
PIII = Dual Pentium III PC (Pentium III 450MHz x 2)
Athl = Athlon PC (Athlon 600MHz x 1)
O2K = Origin 2000 (R10000 250Mhz x 16)
C and J denote C and Java, respectively.
"—" indicates benchmark not yet performed due to time restrictions.

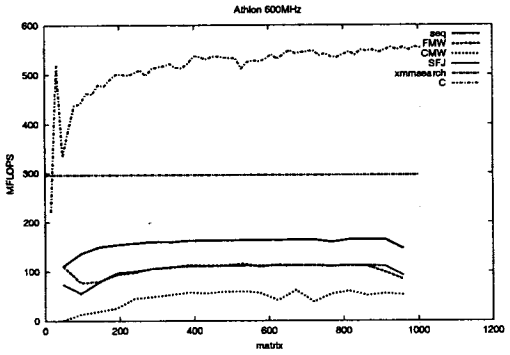

Figure 4: Athlon PC (600MHz x 1) Gemm Performance with Varying Matrix Sizes

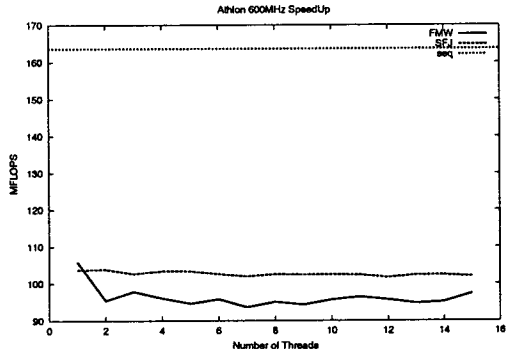

Figure 5: Athlon PC (600MHz x 1) Gemm Performance with Varying Num. of Threads

cursive algorithm is slower; our preliminary profiling analysis has revealed that this is probably due to the BLAS L1 and L2 operations not being appropriately cache-blocked. In fact, as recursion becomes deeper, the blocksize of the ATLAS for Java-generated multiply routine turned out to be excessively large. Still, more investigation is needed.

Parallel execution on the SMP platforms scaled well, but not so on a PC, where practically no benefit and even performance loss is incurred with parallelization.

4.3 Detailed Results for Each Platform

Figures 4 through 16 describe detailed performance measurements for each platform.

Athlon PC

Athlon is a uniprocessor machine, as there is no current chipsets that support a multiprocessor configuration. We nevertheless tested Athlon under multiprocessing setting to investigate whether the parallelized code would penalize performance and if so, by how much.

As mentioned earlier, the performance of Athlon is quite impressive, with xmmsearch reaching nearly 300MFlops. However, whereas the C ATLAS incurs very little penalty for gemm(), we are impacted with nearly 45% overhead for Java, likely due to subarray coping overhead. For blocked LU, we again only reach approximately 50% of C performance due to similar reasons. Parallelization also penalizes performance, as we see in Table and Figures 4–6, especially for master-worker parallelism. This shows that the portable library must judge the number of CPUs available, and employ the sequential version if only a single CPU is available.

Dual Pentium III PC

The dual processor should give us twice the performance, since there should be little sequential overhead for such a low-parallel machine. However, this is not necessarily the case—here, the sequential speed outclasses all parallel versions. Close examination of the graph in Figure 8 reveals that, the performance saturates at threads = 2. Another anomaly is seen Figure 7, where the master-worker performance suddenly drops at matrix size = 500. These suggest that, although IBM JDK 1.1.8 is a native threads implementation, there seem to be anomalies which precludes smooth parallel operations, especially on Linux. We plan to investigate the phenomenon on a larger Pentium Xeon machines.

Enterprise 4000 and 10000

Here we observe the parallel speedup Figures 10–14. For relatively lower number of processors (For Enterprise 4000 and Also Enterprise 10000 with number of threads < 20),

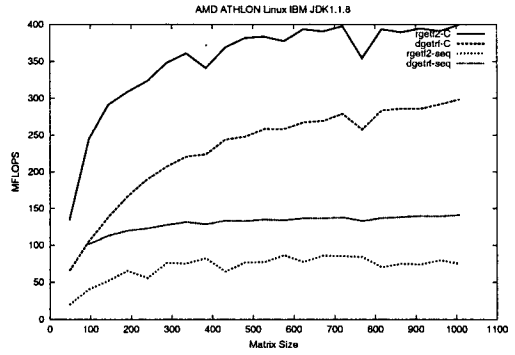

Figure 6: Athlon PC (600MHz x 1) LU Performance with Varying Matrix Sizes

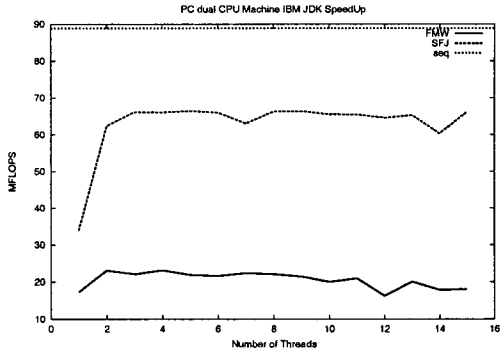

Figure 8: Dual Pentium III PC Gemm Performance (450Mhz x 2) with Varying Num. of Threads

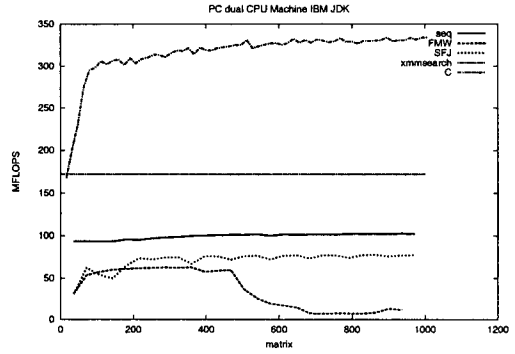

Figure 7: Dual Pentium III PC Gemm Performance (450Mhz x 2) with Varying Matrix Sizes

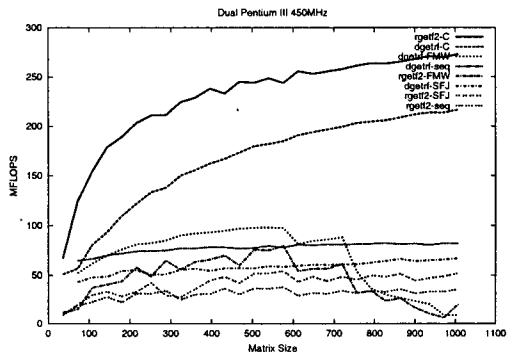

Figure 9: Dual Pentium III PC LU Performance (450Mhz x 2) with Varying Matrix Sizes

gemm() does scale well for static fork-join, whereas master-worker seems to incur some overhead. However, when we increase the number of threads beyond 20 (Figure 14), we start observing the dropoff in scalability even for static decomposition. Still, we reach maximum performance when number of threads reaches the number of processors for each machine (Figures 11 and 14). This suggests that, with better JIT compilers tuned for better numerical performance, we could obtain significant performance with SMPs for Java numerical computing.

Origin 2000

We observe similar behavior to Enterprise server for Origin 2000 (Figure 15–16). At 16 processors, it does not seem to have reached the limits of scalable Java parallel execution. The difference is, however, that the fine-grain multithreading has very little scalability, and exhibits extremely poor performance compared to static fork-join. This could be attributed to the difference in thread scheduling and mutual exclusion in the operating system, the JVM, or both.

5. RELATED WORK

There have been recent surge of efforts of implementing numerical libraries in pure Java. A notable example the Java Array Package in Java by IBM[5], where a flexible Java array class is defined so that it could be run as pure Java code or subject to optimization by a special optimizing compiler. Another is Java Numerical computing in Java[8], by where it supports standard linear algebra operations such as BLAS, LU-decomposition, QR-decomposition, etc. B. Blount [12] and f2j are the efforts of porting JLAPACK to Java. Pozo et. al. have proposed SciMark as a benchmark for Java in numerical computing [3].

These efforts and others, especially those by the Java-Grande Numerics Working Group are quite significant in attempting to make Java applicable to hard-core numerical computing traditionally dominated by C and Fortran. However, although optimizing individual Java compilers have been investigated, especially in the context of the Java array package and the IBM HPCJ[4], achieving performance portable numerical code, and their implications especially for parallel machines, have not been well investigated.

Both PHiPAC[9] and ATLAS are numerical kernel generators for optimized blocked GEMM. Although they have very similar objectives, while ATLAS aims to tune the BLAS kernel for each platform, PHiPAC aims to be more general; however, search time on PHiPAC takes considerably longer, reportedly requiring several days.

MTTL[15] and BLITZ[16] are portable matrix and linear algebra libraries in C++. By extensive use of C++

[3] `http://gams.nist.gov/javanumerics/`

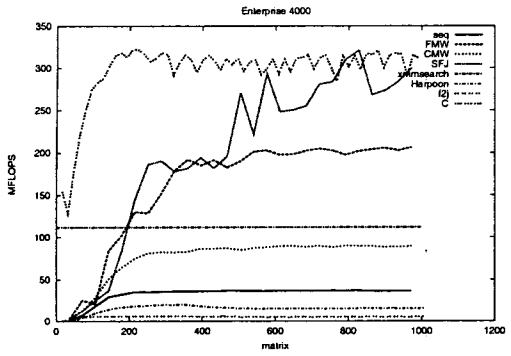

Figure 10: Enterprise 4000 Gemm Performance with Varying Matrix Sizes

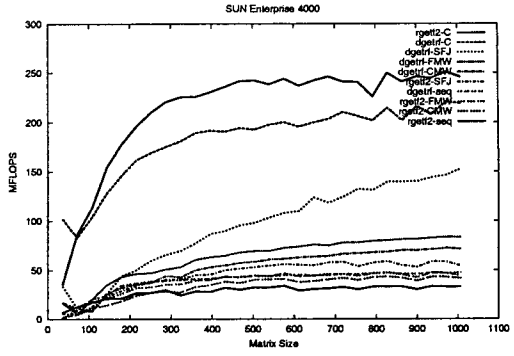

Figure 12: Enterprise 4000 LU Performance with Varying Matrix Sizes

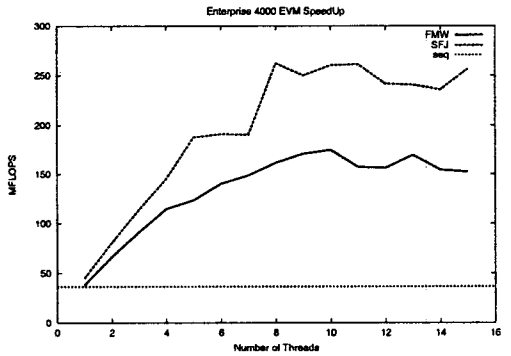

Figure 11: Enterprise 4000 Gemm Performance with Varying Num. of Threads

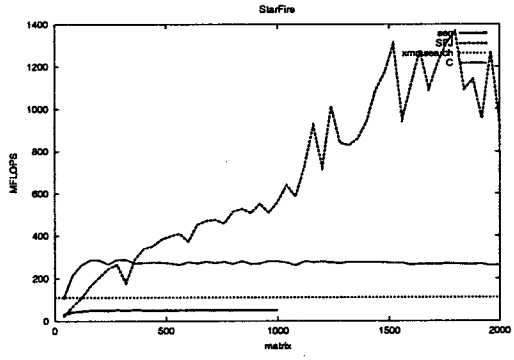

Figure 13: Enterprise 10000 Gemm Performance with Varying Matrix Sizes

templates allowing *template metaprogramming* technique, both allow extensive optimizations such as loop unrolls and some loop restructuring that traditionally compilers had performed for C and Fortran. This allows performance nearing or matching that of Fortran-optimized code or even vendor libraries, just as is with PHiPAC or ATLAS. However, they do not offer any support of "tuning" the parameters for loop unrolls etc; thus, they need to integrate the tuning techniques of, say, ATLAS to be truly portable. For Java, it becomes more difficult due to the lack of compile-time template support; rather, source-code generation tools in Java such as EPP[17] or OpenJIT could be used in place of templates.

What we really propose is to not only to follow the footsteps of Fortran and C and create "static" libraries, but exploit the characteristics of Java, such as dynamic compilation, portable code, automatic downloading, security checks, etc. to aggressively optimize Java libraries automatically for each Java platform. This is more difficult with non-portable languages such as C or Fortran, where such infrastructures are not provided.

6. CONCLUSION

We have introduced AJaPACK, a self-tuning parallel linear algebra package for dense matrices for Java. AJaPACK tunes itself to respective Java platforms using the ATLAS technology, and utilizes the parallel threads which is natively provided by Java platform running on SMPs. Benchmarks show that AJaPACK reaches approximately 50%-25% performance of ATLAS-based C library, and with parallelization, exceeds that performance. It is substantially superior to tested figures for f2j and JLAPACK/Harpoon on the same platforms.

On the other hand, AJaPACK is still somewhat reliant on the quality of both the JVM and the JIT compiler, the former for multithreading performance, and the latter for code quality. We see substantial overhead for higher-level libraries; this is due to object-oriented data encapsulation and also that it is difficult to alias Java subarrays, requiring element-wise array copies to incur method invocation on every copy. We need to add considerable code to AJaPACK to cope with optimal copying of subarrays, and is a subject of immediate future work.

Thread parallelization scaled well for Solaris and O2K, both server platforms; this was somewhat negated by relatively underperforming JIT compiler for numerical computing purposes. On the other hand, for PCs/Linux dual processor, although the JIT compiler was superb, it did not have proper multithreading support.

Future work includes identifying other overhead of higher-level libraries. Also, for AJaPACK in particular, we need to improved the search strategy; for this purpose, we are

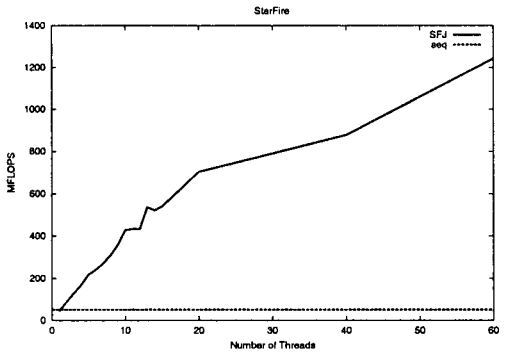

Figure 14: Enterprise 10000 Gemm Performance with Varying Num. of Threads

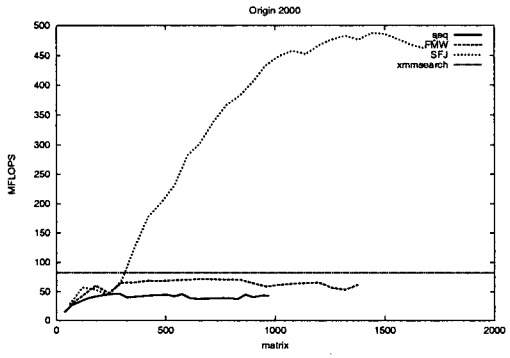

Figure 15: Origin 2000 Gemm Performance with Varying Matrix Sizes

looking at the newly released Atlas 3.0. We would like to, however, generalize our framework by not directly relying on the ATLAS for Java, but rather coming up with a more generalized toolkit for code transformation and generation based on OpenJIT. This will allow us to expand the domain of applicability to other numerical algorithms, as well as allowing other people to construct their own libraries. Ideally, coding a library plus a small effort, such as annotating the core elements and/or following a certain design pattern, based on such a framework, will allow users to code a wide range of performance portable parallel libraries for Java.

Acknowledgments

We deeply thank Jack Dongarra, Clint Whaley, and Sid Chatterjee for the development of their respective libraries which we have substantially employed as a basis, and also their valuable feedback on our work. We also thank the Department of Information Science, the University of Tokyo, and Electrotechnical Laboratory, especially Kenjro Taura and Osamu Tatebe, who have made our usage of E10000 and Origin 2000 possible.

REFERENCES

[1] Satoshi Matsuoka and Shigeo Itou. Towards Performance Evaluation of High-Performance Computing on Multiple Java Platforms. Proceedings of ICS'99 workshop on Java for High-Performance Computing, Rhodes, Greece, July, 1999. (`http://www.csrd.uiuc.edu/ics99/workshops.html`)

[2] Sava Mintchev and Vladimir Getov. Automatic Binding of Native Scientific Libraries to Java. Proceedings of ISCOPE'97, Springer LNCS 1343, pp. 129–136.

[3] Paul Gray and Vaidy Sunderam. The IceT Environment for Parallel and Distributed Computing. Proceedings of ISCOPE'97, Springer LNCS 1343, pp. 275–282.

[4] J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to Megaflops: Java for technical computing. In Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing, LCPC '98, 1998. IBM Research Report 21166.

[5] J. E. Moreira S. P. Midkiff M. Gupta R. Lawrence. High Performance Computing with the Array Package for Java: A Case Study using Data Mining. Proceedings of Supercomputing'99, Portland, Oregon, 1999 (CD-ROM proceedings).

[6] The J-Accelerator and HBC (High-Speed Bytecode Compiler). `http://www.fujitsu.co.jp/hypertext/softinfo/product/use/jac/`.

[7] David M. Doolin, Jack Dongarra, and Keith Seymour. JLAPACK-Compiling LAPACK FORTRAN to Java. Technical Report ut-cs-98-390, University of Tennessee, 1998.

[8] Ronald F.Boisvert, Jack Dongarra, Roldan Pozo, Karin Remington, and G. W. Stewart. Developing Numerical Libraries in Java. Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing, 1998.

[9] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing Matrix Multiply Using PhiPAC: a Portable, High-Performance, ANSI C Coding Methodology. Proceedings of ACM International Conference on Supercomputing, July 1997.

[10] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra software. Proceeding of IEEE/ACM Supercomputing '98, Nov. 1998.

[11] Proceedings of OOPSLA'99, Denver, Colorado, The ACM Press, November 1999. p

[12] Brian Blount and Sid Chatterjee. An evaluation of Java for numerical computing. Proceedings of ISCOPE'98, Springe LNCS 1505, 1998, pp. 35–46.

[13] Satoshi Matsuoka et. al. The OpenJIT Project, `http://www.openjit.org`.

[14] Fred G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms IBM Journal of Research and Development Vol.41, No.6, 1997 p.737

[15] Jeremy Siek and Andrew Lumsdaine. The Matrix Template Library: A Generic Programming Approach to High-Performance Numerical Linear Algebra. Proceedings of ISCOPE'98, Springer LNCS 1505, 1998, pp. 59–70.

[16] Todd Veldhuizen. Arrays in Blitz++. Proceedings of ISCOPE'98, Springer LNCS 1505, 1998, pp. 223–230.

[17] Yuuji Ichisugi and and Yves Roudier. Extensible Java Preprocessor Kit and Tiny Data-Parallel Java. Proceedings of ISCOPE'97, Springer LNCS 1343, 1997, pp. 153–160.

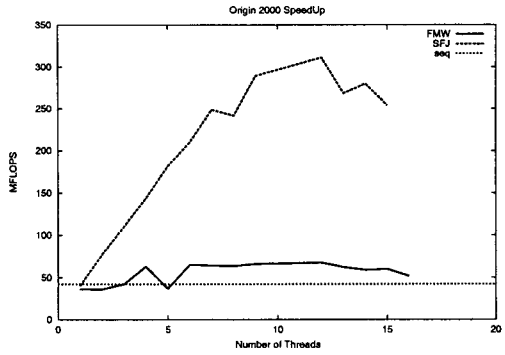

Figure 16: Origin 2000 Gemm Performance with Varying Num. of Threads