計算機アーキテクチャ 137−15
ハイパフォーマンス
コンピューティング 80−15
（2000. 3. 2）

Java による並列 LU の性能評価

長谷川 広和† 松 岡 聡 †† 伊 藤 茂 雄 †

Java で高い可搬性と性能を両立する並列 BLAS ライブラリである AJaPACK を研究開発した。AJaPACK は任意の Java 環境に自己で適合したチューニングを行いその環境に最も適したメモリ最適化を施した BLAS ライブラリとしてインストールされる。性能評価では C 版の ATLAS など、最適化されたネイティブライブラリと比較して 1/2 ～ 1/3 程度の速度を達成し、かつ並列化により SMP 環境ではネイティブライブラリを上回る性能を自動的に得る。これは従来報告された Pure Java による BLAS 性能の数倍から数十倍の性能であり Java が科学技術系の HPC に用いられるために有用な技術であるといえる。一方で Blocked LU 分解においても相応の速度向上を得ているが AJaPACK では部分行列操作のオーバーヘッドにより性能が低下している。これは Java における数値計算ライブラリの構築が未だ容易でなく高性能をあげるために Java 特有のプログラミング技法を必要としている事を示している。

Evaluation of Parallel LU Factorization in Java

Hirokazu Hasegawa,† Satoshi Matsuoka †† and Shigeo Itou†

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. For Blocked LU-decomposition, reasonable speedup had also been reached; on the other hand, the AJaPACK version suffers from high-overhead of subarray manipulation, resulting in loss in performance compared to previous routines such as JLAPACK. This shows that building numerical libraries in Java is still not straightforward, and programming techniques specific to Java should be developed for high-performance.

1. はじめに

近年ではネットワークコンピューティング用の言語として Java 言語が注目されている。しかしながら、現行の Java 言語は, 言語デザインにおいても処理系の実装においても, 高性能な科学技術計算に適切かどうかについて不透明な面が多い[1)]。実際、 Java 処理系の実行環境は, インタプリタ (JVM)・JIT コンパイラ・ネイティブコンパイラと様々であり, ある特定の Java のプラットフォーム向けの最適化は, 仮に CPU や OS やプログラムが同一だとしても, 実行環境の差異によって必ずしも有効になるかどうかは不明である。 これに対し, Fortran, C などでは, コンパイラによる最適化のみならず, ユーザのソースレベルでの最適化技法が知られており, しかもそれらは異なるコンパイラや OS, CPU などでも比較的に普遍的に適用可能である。

ここでネットワーク HPC に必要な性質を鑑みると、それは異なる環境でもダウンロードされたコードのある意味での性能を保証する「性能の可搬性」であると我々は主張する。「性能の可搬性」とは、単なる個々の処理系の絶対性能差を議論するのではなく、与えられたコードが異なる実行環境においてそのマシンのピーク性能に対してある一定のパーセンテージの性能を得ることができることである。例えば、ある最適化を施したとき, それが全ての処理系上で同程度に効果的であれば、その最適化は性能の可搬性を得るために有効な最適化であるといえる。 Fortran や C においてはそのような前提はある程度満たされたが, Java においては阻害要因が多く, それをシステマティックに検証した研究例はない。その観点から Java 言語においても HPC の高速化の手法を検討すると、どの手法も可搬性および性能の可搬性を満たさない:

† 東京工業大学
Tokyo Institute of Technology
†† 東京工業大学 /JST
Tokyo Institute of Technology /JST

- ネイティブライブラリの利用
- 最適化スタティックコンパイラ
- Fortran など、多言語のソースから自動変換
- 手動でのチューニング

そこで、我々は Java 言語において性能の可搬性を確保するための Self Tuning ライブラリおよびコンパイル手法の研究を行っている。これは、実行プラットフォームにあわせてソースレベルおよびバイトコードレベルでポータブルにチューニングしたコードを生成するものである。具体的には、ライブラリ、コンパイラ / コード生成器、実行性能モニタ、などを全て Java で記述し、ある特定のライブラリがダウンロードされて実行される際には、単にライブラリのクラスファイルのみではなく、それらが全てダウンロードされ、自動的に環境に最適化したライブラリを動的に生成するものである。さらに、Java の場合はスレッドによる並列プログラミングを言語レベルでサポートしているので、並列化したライブラリを生成することによって、全環境で自動並列で高性能に動作するライブラリが実現できる。

しかし、この手法には幾つかの技術的問題点がある:
- Java における Self Tuning の有効性
- Self-tuning ライブラリのアーキテクチャ
- Java のマルチスレッド環境において、有効な並列化が可能か否か

以上の問題解決の第一歩として、我々は self-tuning の密行列系の並列数値計算ライブラリである *AJaPACK* を研究開発し、その性能評価を行った。AJaPACK はチューニングの技法としてテネシー大の Whaley と Donngara がの ATLAS[6] を利用し、その Pure Java 版である ATLAS for Java を開発することによって、任意の JVM でポータブルにチューニングが可能になるようにした。実験では、高性能な JIT コンパイラを備えている JVM の場合、ATLAS 流の BLAS の小行列 L1 ブロッキングによる最適化は有効に作用し、C 版の ATLAS と比較すると、約 1/2 〜 1/3 の性能が可搬に得られることが判明した。次に、Java で LAPACK のサブセットを実現した JLAPACK/Harpoon[7] を ATLAS for Java に適合させることによって、従来の JLAPACK の数倍の性能が達成できることを示した。さらに、BLAS を Java のマルチスレッドを用いて並列化した場合、並列なスレッドが高効率にサポートされている JVM の処理系ならば、高い並列化性能を示し、逐次の Harpoon と比較して数十倍の性能向上が実現できた。一方、JVM によっては著しい並列化にるオーバーヘッドの発生、高性能な JIT コンパイラにおける長大なコンパイル時間、プロセッサ数増加時のスレッドシステムの性能の不安定性などの問題も明らかになった。

AJaPACK によって、gemm では Enterprise 10000 においてピーク性能 1.3Gflops 逐次性能の 26 倍、LU 分解では Enterprise 4000 においてピーク性能 248.3 MFlops, 逐次性能の 5.26 倍の性能を得た。一方従来のものと比較して gemm においては大幅な性能向上を見せているが LU 分解では同程度の性能となっており、現在調査中である。

2. AJaPACK: Java で自動チューニングを行う並列 BLAS/LAPACK ライブラリ

2.1 AJaPACK の概要

まず、我々の Downloadable Self-tuning Library の概要 (図 1) を示す。これは以下の 5 つのコンポーネントから構成される。

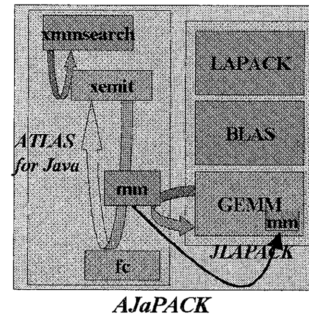

図 1 AJaPACK 概念図

- Code Generator/Compiler: ライブラリの performance critical な部分のコードを生成する。理想的には、我々の OpenJIT[7] のように、一般的なコンパイラフレームワークを用いてそれぞれのライブラリに特化したコード生成器を作成するが、今回の ATLAS for Java の実装ではプロトタイプであるため、一から開発した。
- Generated Kernel: Code Generator が生成したライブラリの Kernel 部分のコード。Driver により性能がテストされ、最適なものが Higher-Level Library Classes に埋め込まれる。
- Driver: Generated Kernel の性能をテスト実行をするためのドライバモジュール。性能を Performance Monitor に報告する。
- Performance Monitor: Driver の性能をモニタリングして、Code Generator に性能を feedback し、新たなコードを生成してもらい、最適な Kernel の探索を行う。
- Higher-Level Library Classes: 上位の API を提供する、性能的には critical でないライブラリの部分。最適化されたカーネルが埋め込まれることにより、高効率に動作する。

今回開発したの AJaPACK は、Downloadable Sefl-tuning Library の proof-of-concept を行う Prototype である。それぞれのコンポーネントは以下のような実装が行われている (これらの詳細は後述する)。
- Code Generator/Compiler: 上記のように、ATLAS を Java に移植した ATLAS for Java の code generator であり、具体的には2の xemit.java の部分と、xmmsearch の一部分。
- Generated Kernel: xemit.java が生成する Java による BLAS の Cache Blocked 小行列カーネル。

実際はjavacによりコンパイルされクラスファイルとなるが直接バイトコードを出力するのも可能。

- Driver: ATLAS for Java の fc.java に相当。
- Performance Monitor: ATLAS for Java の xmmsearch の一部。
- Higher-Level Library Classes: Chatterjee らによる JLAPACK/Harpoon[7] の上位ライブラリ部分を ATLAS for Java 用に移植し、かつ後述の BLAS の並列化を施したもの。

2.2 オリジナルの ATLAS の概要

オリジナルのATLAS(Automatically Tuned Linear Algebra Software)[6] は, Tennessee 大の R. C. Whaley, Jack Dongarra らによって開発されたソースレベルの自動最適化ツールである. ATLAS は与えられたハードウェアプラットフォームに対し, ブロック数やアンロール数等の最適なパラメータを探索し, C による行列積 (BLAS Level 3) プログラムを生成する.

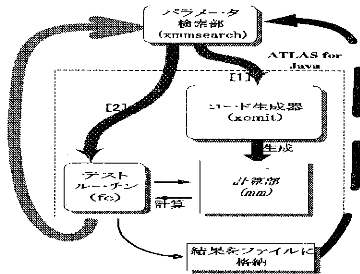

図2 ATLAS のパラメータ検索部

2.3 ATLAS for Java

我々は, AJaPACK のプロトタイプを作成するために、 ATLAS を Java に適用できるように変更した ATLAS for Java[1] を開発した. Java の介在により, C 版の ATLAS のようにそのアーキテクチャにおける最高性能を得るわけではないが, 生成されたプログラムは少なくともその Java 処理系上で実行できる Java プログラムによるピーク性能である可能性は非常に高い, といえる. のみならず, 生成された Java プログラムを観測することで, Java における有効な数値計算の最適化や可搬性の指針ともなりうる.

2.4 AJaPACK の High-Level Library Class — JLAPACK/Harpoon

AJaPACK の High-level Library Class の API 部分としては、二つの Java 版の LAPACK のシステムを検討した。また、単なる適合のみではなく、ライブラリの並列化を行い、 Java の実行環境が並列な場合、自動的に並列化が行われるようにした。

LAPACK の Java 版パッケージは既に 2 つ存在する。一つは Fortran 2 Java コンパイラプロジェクトによる LAPACK で、 Fortran の LAPACK のコードを機械的に変換するものである[3]。もう一つが North Carolina 大の Chatterjee らによる JLAPACK/Harpoon で、 Java 言語によるモジュール化および情報隠蔽を活かすように手でコーディングしている[7]。前者は機械的に変換したため実際に正しく実行されない部分が多い; 一方後者はごく一部の LAPACK のルーチンしか実装されていない (gesv, getrf, getrs, getf2, laswp)。 2 つを比較した結果、本研究でのプロトタイプという性格上性能が上でオブジェクト指向としての設計がきちんとされいる後者を採用した。

AJaPACK の具体的な実装は、この JLAPACK が Kernel routime を呼び出す部分を実装することで行われる; 具体的には Package JBLAS 内の *BLAS クラス (* には D,C などが入る) のメソッド gemm を変更し、かつ ATLAS が生成する Kernel 小行列 BLAS に対して JLAPACK が要求する配列の Descriptor Object を生成するように glue のコードを生成する。

さらに、 gemm() の実装では、行列サイズが小行列サイズ倍でない場合のために、余剰部分に 0 があるとして、小行列積を計算する。この場合、実際に与えられた行列を先に変換するのと、小行列にコピーするときに動的に判断する方法がある。計測の結果では、前者の方が一割ほど性能が高いが、余分なメモリを消費する (ブロックは 数十バイト^2 程度の大きさ。) よって、これはユーザの嗜好によって切り替えられるようにした。

3. AJaPACK の並列化

Java における一つの利点は言語が並列性を内在していることである。これは単純に言語仕様がそうなっているだけではなく、ライブラリや JVM が全て reentrant になっているなど言語システム全体の本質的な性質であり、その点が C や Fortran + スレッドのようなライブラリレベルでの並列性と本質的に違う点である。

HPC で性能を得るという観点からは、当然 Downloadable Self-tuning Library においても、 SMP など複数の CPU が存在するときは、それを活用して自動的にライブラリが並列に動作してほしい。一方、先に述べたように、 Java におけるマルチスレッドによるマルチプログラミングの効率化の研究は近年多くなされているものの、本格的な並列プログラミングの効率や有効性はまだきちんと検証されていない。

そこで本研究では AJaPACK の並列化を Java の native threads を用いて試みた。 BLAS の並列化は良く知られており、特に本研究では各行列がブロック化されているので A × B の演算を $A_{ij} \times B_{jk}$ の小行列要素に分割した積形式で表した場合各 $A_{ij} \times B_{jk}$ は並列に計算できる。そこで以下の 3 つの並列化の手法を試みた:

Fine-grained Master-Worker (FMW) 一定数の Worker スレッドを生成し、各 Worker スレッドは Master から仕事をとってきて仕事が終わったらそれを Master に報告し、新たな仕事を Master から仕事が無くなるまでとり続けることを繰り返す。 Fine-Grained では 1 つの小行列積を仕事の単位とする。ただし、 C, Fortran と違い、 Java は完全部分行列の aliasing を許さないため、元の A, B 行列から小行列 A′, B′ をコピーし、 ATLAS for Java が生成した最適化小行列積を行った後、結果の C 行列に書きこむ (この際には C を synchronize して相互排除する必要がある)。

Coarse-Grained Master-Worker (CMW) FMW

とほぼ同じだが、Worker の一度の仕事の小行列積の数を増やすことにより、粒度を上げる。行う積の数はパラメタで指定する。Worker がとった仕事はステージ状に処理される; つまり小行列生成、ATLAS for Java 小行列起動、結果の書き戻しを各小行列積ごとではなく、ステージングしてまとめて行う。これにより、仕事を獲得するオーバーヘッドが減少し、かつキャッシュに有利にブロック (特に, A の小行列) を獲得できるが、ワーキングセットとプログラムの複雑度が増え、かえってオーバーヘッドとなる可能性がある。

Statically-Decomposed Fork-Join (SFJ) 行列積は deterministic なプログラムなので、最外ループ (IJK 行列積では I ループ) で静的に分割を行い、Fork した各 Worker に均等に割り当てる。各ループごとのコード上のオーバーヘッドは最小である。また、$C_{ik} = \sigma_j A_{ij} \times B_{jk}$ の C_{ik} の書きこみは同期を取る必要がない。

Master-Worker による並列化のほうがオーバーヘッドは大きくなる可能性があるが、任意の問題分割に対して load balancing が自然に行え、柔軟性が高い。本来行列積では SFJ で十分であるが、本研究では Master-Worker のプログラミングの容易性、および SFJ と比較して Java の処理系上でのオーバーヘッドを見るために、3 方式を比較した。

4. Java による LU 分解とその並列化

Java においても LU 分解を行うに際して Block 分割法は有効である. 特に AJaPACK を使う場合はキャッシュブロッキングが有効であったので LU 分解においてもブロッキングアルゴリズムは有効である. ブロッキングのアルゴリズムには以下のようなものが考えられる. (実際のプログラムではピボット選択を行っている)

Blocked LU (getrf) 処理中の部分行列の左上の NB×NB である正方行列 A をとり, その下を L, 右を U, 右下を B とする. A と L 全体 ($m \times$ NB) を基本の LU 分解 (getf2) をしたのち, U と B の部分をアップデートし (trsm, gemm), B を部分行列として処理を繰り返す.

Recursive LU [8] (getf2) ポイントは行列の中央で 2 つに分割して左右の部分行列についてさらに分割操作をし, それを 1 列になるまで続けることである. 具体的には, A のサイズをを $n/2 \times n/2$ として, A と L 全体 ($m \times$ NB) を部分行列として分割操作, $n = 1$ の場合は L の各要素を 1/A 倍, これが終了したのち, U, B をアップデートし (trsm, gemm), B を部分行列として分割操作 ($n = 1$ の時は終了) をする.

4.1 LU 分解の並列化

AJaPACK では並列版 gemm の実装も行っており, このライブラリを使えば LU 分解のプログラムの書き換えなしに容易に並列実行できる. だが gemm や trsm を個別に並列化してもそこそこの性能は出るが, LU の処理全体で性能を出すには一体として並列化するのがよいと思われる. さらに trsm のチューニングルーチンはまだ実装していない. そこで並列 LU 分解は独自に実装したものも用意する.

並列化は建部[2] を参考にした. 本研究で行った並列化は, laswp, trsm, gemm の部分でデータ分散をする. 逆を述べると getf2 の部分は並列化不可能であり, NB の大きさが台数効果を左右する. Blocked LU と Recursive LU に関しては U, B をアップデートする部分で行列積を行う. この部分は並列化が容易であり, 列分割 (縦切り) で並列化を行った. この実装の利点は trsm と gemm の間でデータ分散を再び行うことなしに同じプロセッサ数で実行でき, さらに U のデータ領域を trsm, gemm 間で共有できることである. つまり laswp, trsm, gemm が非同期に実行できる. 逆に欠点はスレッド数が多くなると未処理の部分行列が小さくなったときに全プロセッサを効果的に使えなくなることである.

Blocked LU backpivot は Blocked LU (dgetrf) ではその次のブロックの 2. が始まるまでに実行が終了していればよい. そこで backpivot は並列実行不可能な 1. と並列に行うことにする. なお, 並列 LU を動かすために JLAPACK は (HARPOON, AJaPACK 共に) クラス変数をインスタンス変数にする等若干の変更を施している.

Recursive LU Recursive LU の場合は backpivot の処理の移動は不可能である. また recursive call 自身も並列化はできない.

5. 性能評価

5.1 gemm と LU 分解のピーク性能

まず以下の表1に xmmsearch と GEMM と LU に関して、各処理系でのピーク性能をまとめたものを示す。

AJaPACK によって、gemm では Enterprise 10000 においてピーク性能 1.3Gflops 逐次性能の 26 倍、LU 分解では Enterprise 4000 においてピーク性能 248.3 MFlops, 逐次性能の 5.26 倍の性能を得た。また、この測定により 3 つの並列化の実装の中で全ての処理系で最も高い性能は SFJ であり、それより 1-2 割低い性能で FMW、CMW は最も性能が低かった。FMW と CMW は O2K に関しては台数効果が現れなかった。従来の JLAPACK と比較すると gemm に関してはかなり速くなるが LU では同程度の性能である.(図 4) これにはいくつかの理由が考えられる.

- 列分割のみであることの限界
- AJaPACK の能力を引き出すブロックサイズの設定による逐次処理部分の増大
- 小行列生成のオーバーヘッド

原因は現在調査中である.

5.2 AJaPACK での LU 分解の性能

LU 分解は BLAS ライブラリとして

- HARPOON project JLAPACK
- AJaPACK

の 2 種類を用いて比較する.

ブロックサイズによる変化： Blocked LU (dgetrf) の性能はブロックのサイズに大きく依存する. ブロックのサイズが大きくなると並列処理が不可能である部分 (dgetf2) の処理量が大きくなり, 台数効果が落ちる. 一方で, ブロックサイズが小さくなると AJaPACK の小

ピーク性能 (MFlops)	E4K C	E4K EVM04	PIII C	PIII IBM	Athlon C	Athlon IBM	E10K C	E10K EVM04	O2K C	O2K SGI
xmmsearch	401.9	132.9	375.6	171.7	570.5	296.0	281.2	110.4	—	81.95
GEMM 逐次	321.5	52.10	325.0	102.6	555.7	165.1	286.0	52.60	—	43.04
GEMM 並列	—	349.9	—	77.3	—	112.5	—	1365.4	—	487.3
LU b 逐次	216.2	47.25	216.6	87.40	298.5	140.9				
LU b 並列	—	248.3	—	98.0	—	—				
LU re 逐次	250.8	34.23	273.1	36.70	399.8	86.40				
LU re 並列	—	201.3	—	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)
Athlon = Athlon PC (Athlon 600MHz x 1)
O2K = Origin 2000 (R10000 250Mhz x 16)
LU b = Blocked LU
LU re = Recursive LU

E4K and E10K : Solaris Production Release JVM 1.2 with optimizing JIT compiler (JBE)
PIII and Athlon : IBM JDK-1.1.8 JVM with optimizing JIT compiler
O2K : SGI JDK 1.2.1

表1 ピーク性能一覧

行列生成のオーバーヘッドが大きくなる．そのため，行列サイズを 1000 に固定してブロックサイズを変化させて最も効果が大きい を測定した．その結果が図 5である．HARPOON ではブロックサイズが 2 から 4 でピーク性能を出すが AJaPACK では小行列のサイズ（グラフ中の縦線部分）である 36 でピークとなる．そのため，台数効果が落ちるので HARPOON と同等の MFlops 値となる．

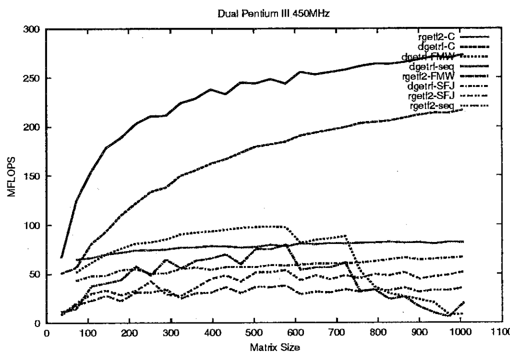

図3 Dual Pentium III PC LU Performance (450Mhz x 2) with Varying Matrix Sizes

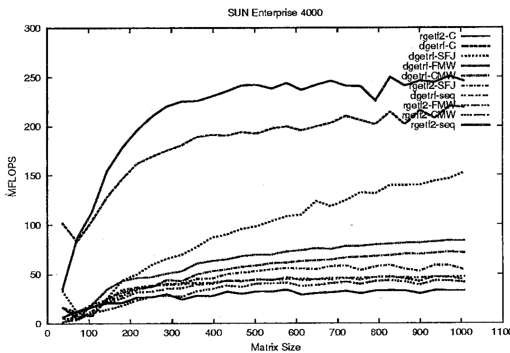

図4 Enterprise 4000 LU Performance with Varying Matrix Sizes

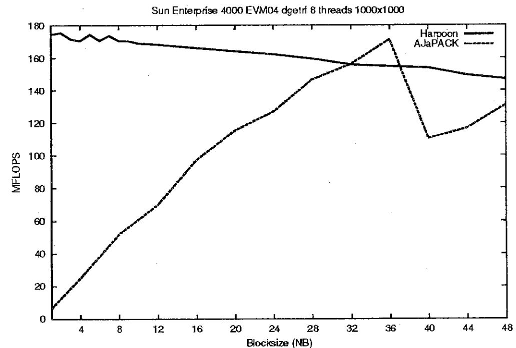

図5 Enterprise 4000 におけるブロックサイズを変化させた場合の性能

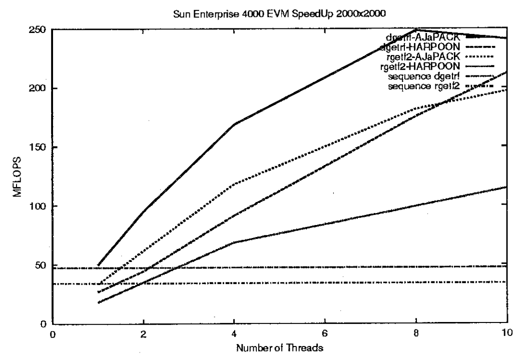

図6 Enterprise 4000 における SpeedUp

LU 分解の性能図 6 はスレッド数に対する性能向上を表したグラフである．AJaPACK での Blocked LU (dgetrf) の 8 スレッドから 10 スレッドにかけ

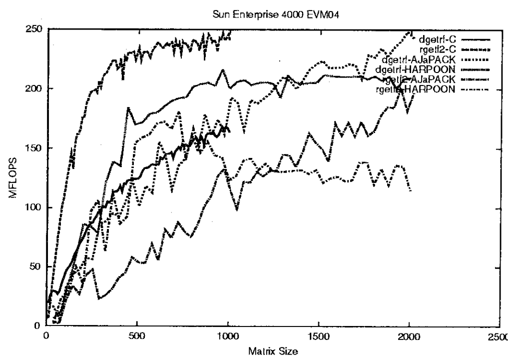

図7 Enterprise 4000 における LU 分解の性能

て MFlops 値が逆に落ちている．この現象は行列積の測定においても現れている．AJaPACK の, Blocked LU (dgetrf) では最大で逐次の 5.26 倍, Recursive LU (rgetf2) では 5.75 倍となっている．よって逐次の Recursive LU の性能が向上すれば並列においても Blocked LU を上回ることが可能と言える．AJaPACK は現在は gemm のみの高速化であるので trsm の高速化を行って確認する．図 7 はそれぞれ最高性能を出しているスレッド数等の設定で行列サイズ毎の MFlops 値を測定したものである．参考として C の逐次性能も載せてある．行列サイズが大きくなると AJaPACK の性能が向上し, サイズが 1300 から 1400 で同関数の HARPOON を上回る．

6. 関連研究

Java Numerical computing in Java[4] は Java での数値計算のライブラリである．初期バージョンでは, BLAS, LU や QR 分解などをサポートしている．B. Blount[7] では, LAPACK を Java で実装した JLAPACK を用いて性能評価している．他にも NIST の Pozo らが Java における BLAS の実装や, SciMark などのベンチマークの結果を報告しているが☆, 性能の可搬性や, 処理系のピーク性能に言及した研究はまだない．

また, ソース上での最適行列積生成ツールの関連研究として, PHiPAC がある．PHiPAC[5] は, ポータブルで高性能な ANSI C の数値ライブラリを開発するための GEMM 生成ルーチンである．ATLAS が各アーキテクチャ毎に最適化ソースを供給するのとは異なり, 様々なアーキテクチャで使用可能なように分岐を挿入した GEMM を供給する．しかし, ATLAS と比べると, あらゆるルーチンに同種の測定を繰り返すため, パラメータ発見にかかる時間が長くかかる．

7. まとめ

本稿では、Java が異なる処理系において性能の可搬性を維持できるかどうか検証するために、C の最適化 BLAS コード生成システムである ATLAS の Java 版 (ATLAS for Java) を開発し、複数の Java 処理系の BLAS のピーク性能を調べ, ソースレベルで可能な最適化と性能比較をすることによって, ソースレベル最適化の可搬性を検証した．また、ATLAS for Java を用いた Java 用 LAPACK パッケージ AJaPACK を開発し、並列化にも対応した高速な Java 数値計算パッケージを提供する。その結果、逐次の C の性能を越える性能を得てソースレベル最適化の有効性を実証した。

AJaPACK では性能の大きいマシンほど小行列サイズが大きくなってしまうため (Enterprise 4000 で 36, Athlon で 48, R12K で 60) キャッシュヒット率が上がる代わりに並列化の効果が犠牲になってしまう．このため, 小行列サイズより一部が小さい長方行列などに対しても性能を出せるよう改善することが必要である．

LU の並列化については行のみの分割でそれなりの性能向上を見せているがさらにプロセッサ数が増え, スレッドの数を増やすと縦長の行列になってしまい AJaPACK の最適化の効果もなくなってしまう．そのため行分割を加えたものを用意して測定する予定である．

参考文献

1) Satoshi Matsuoka and Shigeo Itou. Towards Performance Evaluation of High-Performance Computing on Multiple Java Platforms, to appear in Future Generation Computer Systems, Elsevier Science, North-Holland.
2) 建部 修見,「分散メモリ型並列計算機による LU 分解」, 情報処理学会研究報告, 95-HPC-57, SWoPP'95 別府, pp.55-60, 1995 年 8 月
3) David M. Doolin, Jack Dongarra, and Keith Seymour. JLAPACK-Compiling LAPACK FORTRAN to Java. Technical Report ut-cs-98-390, University of Tennessee, 1998.
4) 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.
5) 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.
6) R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra software. Proceeding of IEEE/ACM Supercomputing '98, Nov. 1998.
7) Brian Blount and Sid Chatterjee. An evaluation of Java for numerical computing. Proceedings of ISCOPE'98, Springe LNCS 1505, 1998, pp. 35–46.
8) 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.

☆ http://gams.nist.gov/javanumerics/