数理解析研究所講究録
1514巻 2006年 211-217

数式処理（システム）におけるマルチスレッド化の試み

村尾 裕一
MURAO HIROKAZU*
電気通信大学

兵頭 礼子
HYODO NORIKO†
AlphaOmega Inc.

齋藤 友克
SAITO TOMOKATSU‡
AlphaOmega Inc.

1 はじめに — 背景と動機

本稿は，筆者らが進めている，数式処理における基本演算のための効率のよいソフトウェアの研究開発プロジェクトにおいて，新たに開始したマルチスレッド化の試みに関する，最初の段階のレポートである．

筆者らは，数式処理における計算で計算時間の膨張が顕著な計算のひとつとして行列関連の計算に注目し，その高速化・効率化を実現する計算手法を確立すべく，基本部分からの見直しをしながら研究開発をすすめてきている．数値計算においては，そうした研究は一分野を形成し，長期にわたり研究開発が進められてきており，同様の需要は数式処理においても存在すると思われる．数式処理において行列やベクトルの計算が主要部分である研究例の報告はあまり見られないが，一変数多項式を配列表現した場合等，様々な基本計算に内在することは確実であり，基礎となる機能のソフトウェアの高速化は不可欠であろう．筆者らは，こうした考えに基き，行列演算関係の実験を重ねると同時に，基本演算ルーチンを定義し，実際のソフトウェアの開発をすすめてきた．

一方，PCのCPUの高性能化はベクトル演算命令の導入による並列計算に留まらず，日常的に用いられているPCのCPUとして，ひとつのパッケージ中に二つのCPUコアを登載したデュアルコアのCPUが安価に出回り一般化しようとしている．また，hyper-threadingのように，ひとつのCPUに交互に動作する2つの演算器が存在するように利用する技術は以前から一般化している．更に，高性能計算機は，ベクトル演算器に代わり，もしくはそれに加え，複数のCPUでメモリを共有する密結合の並列機で構成するのが一般的となっている．そういった計算機をひとつのプログラムで有効活用するには，スレッドと呼ばれる実行単位が，複数個同時にひとつのメモリ空間を共有して実行できるようにする必要がある．いわゆるマルチスレッド化である．プログラムをマルチスレッドで動作させるには，threadライブラリを用いてスレッドの生成や制御をプログラム中で明示的に行うのが一般的であり，かつ，汎用性がある．Pthreadのような標準的なライブラリも存在している．これらを用いてマルチスレッドのプログラムを開発，或は，既存のプログラムをマルチスレッド化するには，計算内容そのものに関する知識と，スレッドの制御等の並列処理に関する知識が必要となり，プログラミングが複雑化する．OpenMPでは，並列処理の形態を，単純なループを分割して同時に並列実行するwork-sharingの方法と，個々の実行単位に独立した処理内容を指定して並列実行するタスク並列型の処理法の2つの単純な形に集約し，各々をコンパイラ・ディレクティブにより簡単に指定できるようにしている．前者は，一連の均質なデータに対し同一の処理を行うデータ並列処理を実現し，数値処理における高速化の重要な手法である．OpenMPは共有メモリ並列機向けの並列化プログラミングの事実上の業界標準となっており，数値処理を主体として，既に多方面で実用に用いら

*murao@cs.uec.ac.jp
†noroko@a2z.co.jp
‡saito@a2z.co.jp

れている．OpenMP におけるディレクティブは，基礎となる言語（Fortran や C/C++）にとっては通常は注釈とみなされるため，計算アルゴリズムを実装するプログラムそのものの構造を変更せずに並列化が可能である．我々は，この OpenMP を用いて数式処理に現れる計算の並列化を試みる．

本稿では，OpenMP が我々の用途として利用価値があるかを見極めるための，簡単な実験と実験環境の整備について説明し，今後の展望を述べる．

2 プラットフォームとソフトウェア環境

前述のとおり，マルチスレッドのプログラムを開発するには，Pthread 等の標準的な thread ライブラリを用いる方法と，OpenMP の指示に従ってコンパイラが自動的に並列化する方法とがあるが，ここでは後者を用いる．OpenMP による並列化は，ディレクティブで指定するだけで可能なので，既存のソフトウェアを改造して容易に並列化し実験をすることができる．我々は，実験材料として数式処理システム Risa/Asir [NT92] を用いることを予定している．Risa/Asir は様々なプラットフォーム上で稼働するが，開発は主に x86 系の CPU の PC UNIX（主に FreeBSD）上で，GNU C コンパイラ（GCC）を用いて行われている．OpenMP は商用の多くのコンパイラでサポートされているが，GCC では GOMP プロジェクト [GNU] が進められているが現時点ではリリースもサポートもされていない．言語処理系としては次の選択肢がある．

(a) gcc の GOMP プロジェクト

(b) OMNI OpenMP Compiler[OMN]: 複数のプラットフォーム対応のトランスレータ．国内の有志により開発・サポートされているため利用し易い．リリースされているコンパイラの開発は停止したままで，対応しているというプラットフォームが古い．

(c) Intel C++ Compiler for Linux[Cor]: 本来は有償だが，非商用ソフトウェア開発向けには無料で配布される版がある

(d) その他の有償のもの：PGI(The Portrand Group)[The]．PathScale EKOPATH Compiler[Pat].

今回は，OpenMP の利用可能性を探ることが目的であり，また，将来有効性を示すことができた場合に多くの利用者に無料でその成果を活用してもらえるであろうことと，高いコンパイル性能を有することから Intel コンパイラ (icc) を用いることとした．[1]尚，最新の Intel コンパイラは gcc の上位完全互換を唱っているが，メモリ管理を行う GC（gc6.5）[Boe] はプラットフォームへの依存度が極めて高く，icc での正常動作には到っていない．現状では，GC のみ gcc で作成したオブジェクトを利用している．並列処理を試みる，Risa/Asir の他の部分については icc でコンパイルおよび動作が可能なので，支障はない．

併せて，OpenMP の汎用性を確認し，大規模な並列処理の効果と利用可能性を探るために，大型の並列処理計算機上にも実験環境を構築することとした．その試みとして，国立大学法人の全国共同利用施設に導入されている並列計算機の内の一種である HITAC SR11000 上への環境の構築を試みている．この計算機はノード当たり 8× dual Power5 がメモリを共有する並列機で，OS は IBM の AIX が稼動し，C 言語処理系は日立製と AIX 標準の xlc の 2 種類が利用可能である．先述のとおりプラットフォームへの依存性が非常に高い GC は幸い AIX に対応しているが，日立製の C 言語処理系を用いる場合，オブジェクトの dynamic loading 時のメモリの利用法が異なるためか static にリンクする必要があった．総じて，xlc を利用した方が安定感がある．[2]

---

[1] FreeBSD には Linux のバイナリが実行可能であり，実際，icc も実行可能と思われる．しかし，我々が実験環境として用意した，CPU に AMD Athlon X2 を用いアドレス空間を 64bit とした FreeBSD 上では，64bit の icc を稼働させるには到らなかった．本稿執筆時には，同機の OS を Linux (Fedora Core 4) に変更して利用している．

[2] xlc では，char 型が符号付きであることを明示してコンパイルする必要がある．

こうして，複数の異種のプラットフォームで OpenMP が利用可能で数式処理システム Risa/Asir を可能する環境が整備できた．今後，マルチスレッド処理が安定して動作し，その有効性の見通しが立てば，実験環境を更に多様なプラットフォームへと拡大し，OpenMP の有用性を試す予定である．

3 OpenMP について

OpenMP の機能の概要を，実例を用いて簡単に説明する．Lawrence-Livermore 国立研究所の Web ページ[3]には次のような簡単な例題があるが，これらに関連付けて説明する．

ファイル名	概　要
omp_hello.c	複数スレッドの生成と実行
omp_workshare1.c	for ループの分割と dynamic な scheduling
omp_workshare2.c	複数の section
omp_reduction.c	配列要素の和の reduction
omp_orphan.c	omp parallel と omp for の scope. reduction による内積計算
omp_mm.c	行列乗算

- `#pragma omp parallel` — 複数のスレッドの生成が生成され，直後の文または { ... } ブロック内がすべてのスレッドで実行される．下の例は omp_hello.c の概要だが，{ ... } 内が複数のスレッドで並列に実行され，各スレッドからの「Hello ...」メッセージが順不同で出力される．但し，関数 omp_get_thread_num() および omp_get_num_threads() は OpenMP の関数で，各々，実行しているスレッドの識別番号および実行中のスレッドの個数を返す．

```
int tid;

#pragma omp parallel private(tid)
{
   tid = omp_get_thread_num();
   printf("Hello from thread = %d\n", tid);
   if (tid == 0)
      printf("Number of threads = %d\n",
             omp_get_num_threads());
}
```

生成されたスレッド毎に異なる処理を指定するには，「omp parallel」が指定されたスコープ中で，ディレクティブ「omp for」か，「omp sections」と「omp section」の組合せを指定する．このスコープは動的であり，ディレクティブの指定はマルチスレッドで実行中の状況において有効である（例題 omp_orphan.c 参照）．

- `#pragma omp for`（ループの work-sharing）：単純な for 文による繰返しをいくつかに分割し，各々をスレッドが処理する．分割は，繰返しの開始直前に各スレッドに割当てる回数を決めて固定して実行する標準的な方法と，繰返しの各ステップを暇なスレッドに割当てる動的スケジュールの方法とが可能である．

omp_reduction.c では

```
#pragma omp parallel
{
   #pragma omp for reduction(+:sum)
   for (i=0; i < VECLEN; i++)
      sum = sum + (a[i]*b[i]);
}
```

（注）`#pragma omp parallel for`：この例のように，omp parallel 下に omp for しかない場合は，この２つの指定をひとつにまとめることができる．

```
#pragma omp parallel for ...
for (i=0; i < N; i++) ...
```

[3] http://www.llnl.gov/computing/tutorials/openMP/samples/

上の例は均等に分割し固定して実行する例だが，右図は動的なスケジュールの例である．また，右図では，共有変数の sum への加算（総和の計算）は本来逐次的に行われるが，reduction() の指定により総和計算は自動的に reduction による並列計算が行われる．

omp_workshare1.c では

```
#pragma omp parallel
{
   ...
   #pragma omp for schedule(dynamic,chunk)
   for (i=0; i < N; i++) {
      c[i] = a[i] + b[i];
      printf( ... );
}
```

- `#pragma omp sections` と `#pragma omp section`: 各スレッドが処理内容を別々に記述する方法（例題 omp_workshare2 参照）．

```
#pragma omp sections
{
   #pragma omp section
   { ... (ひとつのスレッドで処理する内容) ... }
   #pragma omp section
   { ... (ひとつのスレッドで処理する内容) ... }
   ...
}
```

上記以外に，複数のスレッドが同時には実行できないプログラム部分を指定する critical，同期をとる barrier 等の指定の他，データの属性の指定法，実行時ライブラリ，環境変数等が OpenMP で規定されている．

より具体的な例として行列乗算の例を見ると右図のとおり．我々の実験における高速化の目安とすべく，この数値処理の典型的な例の実測を試みたが，行列が小さい，hyper-threading を用いたコマンドレベルでの時間計測などの条件がよくないのか，殆ど性能向上が見られなかった．

omp_mm.c では

```
#pragma omp parallel private(i,j,k)
{
   ...
   #pragma omp for schedule(static, chunk)
   for (i=0; i < NRA; i++)
      for (j=0; j < NCB; j++)
         for(c[i][j] = 0,  k=0; k < NCA; k++)
            c[i][j] += a[i][k] * b[k][j];
}
```

4 数式処理における題材

単に並列処理という観点では，数式の計算には，いちいち例を挙げるまでもなく，様々な並列性が存在し，分散処理による高速化例も多種多様である．そのした大規模な並列処理も我々の将来の課題ではあるが，当面の目標は OpenMP の記述法と記述能力の理解，先述の CPU を対象とした性能向上の割合等の基本的な知識を獲得し，数式計算に如何にいかせるかを検討することにある．本稿では，先ず，性能向上の期待できるデータ並列処理の計算を扱う．

数式処理において，一連の均一なデータの処理といえば，多倍長数の演算と各種のモジュラー算法に見られる $\mathbb{Z}_p$ の演算がある．前者は一般には逐次性があるため，効率の良く並列処理を行うのは難しい．一方，後者は，既存の算法の単純な並列化の手法として知られており，算法の構成にも関係して代数的にも重要であり高い需要もある．ここでは，$\mathbb{Z}_p$ を要素とするベクトルや行列，あるいは，係数にもつ一変数多項式の配列表現を扱い，数値処理と同等のデータ並列処理においてどの程度の性能が得られるかを調べる．

$\mathbb{Z}_p$ 演算： p は一語長に収まるとし，$\mathbb{Z}_p$ 要素は整数型で表すものとする．$\mathbb{Z}_p$ 要素どうしの演算については，定番の方法が確立している [DGP02], [村川 04]．加減算では，mod p の計算に乗除算は不要である．また，

乗算については，(積 $\bmod p$) の計算で，予め $1/p$ を十分な精度の実数で求めておけば，その商は乗算と切捨てで求めることができ，除算が不要となる．この除算を乗算で置き換える方法は数値計算でも高速化の手法として知られている．数値計算では誤差に気をつけて用いる必要があるが，$\bmod p$ の場合は本質的には整数演算なので誤差とは無縁である．

$\mathbb{Z}_p$ 要素のベクトルの内積・行列乗算： 上記のとおり，浮動小数による数値計算の場合に比べ，算術演算は複雑になり，行列演算における各要素の演算の粒度は大きくなるため，並列処理の効果が現れ易くなると期待できる．行列乗算は，互いに独立な内積の計算を積行列の要素分だけ繰り返すので，並列処理の格好の題材である．三重のループをどの順番に実行すべきかはメモリアクセスに依存し，数値計算の分野で十分に議論が行われた．$\mathbb{Z}_p$ 要素の場合，内積計算についても $\bmod p$ 演算を減らす方法が確立している．

$\mathbb{Z}_p$ の要素 a_i, b_j は $[0,p)$ の値をとるとし，符号無し整数で表されているものとする．$S_k \stackrel{\text{def}}{=} \sum_{i=0}^{k} a_i b_i \bmod p$ とし，内積 S_n を $S_k = a_k b_k + S_{k-1} \bmod p$ の繰返しで計算する．容易に思い付くように，S_k の値を保持する変数を十分な精度をもつ整数型（$p \le 2^{16}$ ならば 32bit の unsigned int, $2^{16} < p \le 2^{32}$ ならば 64bit の unsigned long long．以下，この型を「S_k-型」と呼ぶ）とし，毎回 $a_k b_k$ に対し $\bmod p$ の計算をするのではなく，必要に応じて $\bmod p$ をとるようにすれば，(除算に代わる）乗算の回数を減らすことができ，高速化につながる．更に，次の工夫により，$\bmod p$ の計算は，総和 S_n に対する 1 回だけで済ませられる．

- X を，(S_k-型で表現される最大の整数値の $\bmod p$) $+1$ とする．即ち，S_k-型が m ビットの時 $X := (2^m - 1 \bmod p) + 1$.
- $t := a_k b_k$ を S_k-型で求めた後 $S_k := t + S_{k-1}$ を計算し，S_k のオーバーフローが検知された場合（32/64bit の符号なし整数値として $t > S_k$ や $S_{k-1} > S_k$ となった場合）は，$S_k := S_k + X$ とする．
- 総和 S_n が $\ge p$ ならば，予め double 型で計算しておいた $R = 1.0/p$ を用いて $S_n \bmod p$ を計算する：
  - S_k-型が 64bit ($2^{16} \le p < 2^{32}$) の場合，$q := \lfloor (S_n \texttt{>>} 16)\, R \rfloor$ ($< 2^{32}$) を unsigned int (32bit) で求めた後，$S_n := S_n - ((pq) \texttt{<<} 16)$ ($< 2^{32}$).
  - $q := \lfloor S_n R \rfloor$ を unsigned int (32bit) で求めた後 $S_n := S_n - pq$ を計算する．

応用計算： ベクトル演算器では，上のような内積計算における算術演算を並列処理すれば効果が得られるが，スレッドによる並列処理では少なくとも内積計算程度の粒度でスレッドとすべきであろう（例題 `omp_mm.c` 参照）．内積計算の繰返しとなる計算の例を挙げる．

- 行列の乗算．その繰返しによる巾乗計算
- 有限体上の線形方程式を解く Wiedemann の算法：$n \times n$ 行列 A と n 元列ベクトル $\vec{b}$ が与えられた時，$A\vec{x} = \vec{b}$ を満たす $\vec{x}$ を求める．具体的には，ベクトル列 $A^i\vec{b}, i = 0, ..., 2n-1$ を計算し，この列に対する最小多項式 $f(z)$ を求め，$f(A)\vec{b} = \vec{0}$ を変形して解 $\vec{x}$ を得る．並列処理すべきはベクトル列の計算だが，$A^{i+1} = A(A^i\vec{b})$ の繰返しは逐次的である．並列化には次のような手法が考えられる．
  (a) 行列とベクトルの乗算は独立な内積計算の繰返しなので，各々を並列に計算
  (b) $n >> 1$ の時（粒度の大きい，並列度 2 の並列性）：ベクトル列 $A\vec{b}, ..., A^n\vec{b}$ の逐次計算全体と A^2 の計算を並列処理した後，以降は A^2 を用いて $\begin{cases} A^{n-1}\vec{b} \to A^{n+1}\vec{b} \to A^{n+3}\vec{b} \to \cdots \\ A^n\vec{b} \to A^{n+2}\vec{b} \to A^{n+4}\vec{b} \to \cdots \end{cases}$

- polynomial composition: 一変数多項式 $P(z)=\sum_i p_i z^i$ 及び $Q(z)=\sum_i q_i z^i$ に対し，$Q(P(z)) \bmod z^{n+1}$ を求める．Brent & Kung の高速算法 [BK78] は，一変数多項式の巾乗の列の計算と行列乗算からなる．(n 次までの計算を，$n=k\times k$ と折り畳み k 毎に繰り返すという手法を用いる．特殊化して 2 毎に折り畳んだのが前項の方法である．)

並列度を 2 に限定すれば，巾乗の計算において，更に粒度を大きくして並列処理を行う方法が考えられる（右図）．$e=2^s+\sum_{i=0}^{s-1} 2^i b_i$, $b_i\in\{0,1\}$ として，f^e を計算する．Z には f^{2^i} を計算していき，Y には対応する b_i が $\neq 0$ の時に Z を掛けていく．ここで，2 つの積 $Z*Y$ と $Z*Z$ の計算は並行して実行可能であり，計算量も同等である．f が行列の場合，この計算の粒度は大きいので有効と期待できる．

```
Z = f;
for (i=0; b_i==0; i++) Z = Z*Z;
Y = Z;   Z = Z*Z;
for (i++; i < s; i++, Z = W)
  if (b_i ≠ 0) { Y = Z*Y;   W = Z*Z; }
  else W = Z*Z;
Y = Y*Z;
```

5 計算機実験

本稿では，最初に行った最も簡単な計算機実験の結果を報告する[4]．計算は行列の巾乗 A^e である．

- 行列の巾乗（メモリは可能な限り節約して利用する）
- Pentium 4@3GHz, L2 キャッシュ1MB, メモリ 512MB. OS: Fedora Core 4 linux, hyper-threading. コンパイラ:Intel C++ compiler for Linux, Ver 9.0
- 時間計測は，time コマンドによる経過時間

p としては半語長の 911 と 65533 を用い，行列の要素は乱数的に生成した．また，行列の次数については 64×64, 128×128, 256×256 等を試みた．プログラムは，特にキャッシュを意識したりせず，最適化はコンパイラの -O3 に任せた．紙幅の都合により，実測データは大部分を省略し，特徴的な事柄を述べる．

先ず，前節の最後の方法を，その全体を *#omp parallel* 下に置き，並列実行を行った．この実験は，数値計算の例題で効果があまり見られなかったことから，粒度を大きくして試したものである．しかし，e が偶数の場合の無駄な重複計算を含むだけでなく，キャッシュミスやメモリアクセスの競合を容易に招く稚拙な方法であった．下表に，$p=65533$, A が 128×128 の行列の場合の A^e を 10 回計算した時の計算時間（単位は秒）と比率をまとめる．この表から定量的な傾向を見出すことは難しいが，右に行く（繰返しの回数が 1 回ずつ増える）に従って大きくなる計算時間の伸びが，概ね 2 threads の方が小さく（2/3 程度）並列処理の効果が現われていることがわかる．64×64 では演算量が少な過ぎるためか比較が困難であり，また，256×256 ではキャッシュミスを起こす（メモリは作業領域を含め行列 5 個分が必要）ためか並列処理の効果（計算時間の比）は弱まる．また $p=911$ とした場合も，演算量が減りオーバーヘッドの割合が大きくなるためか効果は薄れる．

e	31	63	127	255	511	1023	2047	4095	8191	16383	32767	65535
1 thread	1.47	1.78	2.20	2.51	2.82	3.24	3.55	3.87	4.28	4.59	4.91	5.32
2 threads	1.37	1.57	1.80	2.01	2.32	2.40	2.71	2.93	3.24	3.46	3.67	3.99
比	.932	.882	.818	.801	.823	.741	.763	.757	.757	.754	.747	.750

その後，行列乗算を通常どおりに並列化する方法も試みた．即ち，行列乗算の三重ループの一番外側で並列化をするものである．勿論，前節の最後に述べた 2 つの乗算はひとつのループに統合した．下表に実

[4] 本報告の執筆時点では，まとめで述べるいくつかの課題について大きく進展している．

測時間を示す．マルチスレッド化の十分な効果とが見られ，256 × 256 と行列を大きくしても順にアクセスするため性能の劣化は見られない．また，実測値は省略するが，行列乗算毎に並列処理を行っているため，並列処理の効果は e の値に依存しなくなっている．$p = 911$ の場合についても前と同様の傾向が見られ，実行時間の比率は 0.8 台までおちる．

	128 × 128									256 × 256			
e	31	63	127	255	511	1023	2047	4095	8191	31	63	127	255
1 thread	1.44	1.75	2.05	2.46	2.77	3.09	3.38	3.79	4.10	11.81	14.63	17.28	20.16
2 threads	0.94	1.25	1.45	1.75	1.95	2.16	2.46	2.67	2.97	8.43	10.47	12.61	14.56
比	.653	.714	.707	.711	.704	.699	.728	.704	.724	.714	.716	.730	.722

6 まとめ

本稿で示したとおり，数式処理においてもマルチスレッド化の有効な様々な計算があり，OpenMP を用いれば容易に並列化することが可能でかつ効果をあげることができる．本稿では，hyper-threading でのみ実測を行ったが，複数コアの CPU ではより大きな効果があると期待できる．数学的には単純な計算であっても，アルゴリズムの並列化や数式処理において有用な並列処理プログラミング技術の開発は研究課題であり，この点では数値処理に比べて大きく遅れている．数値処理で蓄積された技術（微細な最適化，キャッシュサイズを意識したプログラミング（ブロッキング，stripmining, ...）など）を活用していくことは次の課題である．併せて，異なるプラットフォーム上での実証実験と開発を進めることは急務である．また，処理（計算量）の均一性を望めない数式の計算を，如何にマルチスレッド化するかは今後の検討課題である．

参 考 文 献

[BK78] R. P. Brent and H. T. Kung. Fast algorithms for manipulationg formal power series. *Journal of ACM*, 25(4):581–595, 1978.

[Boe] B. Boehm. A garbage collector for C and C++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[Cor] Intel Corp. *Intel C++ Compilers for Linux.* `http://www.intel.com/cd/software/products/asmo-na/eng/compilers/clin`.

[DGP02] J. G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In *Proc. ISSAC '02*, pages 63–74. ACM Press, 2002.

[GNU] Free Software Foundataion GNU Project. GOMP – an OpenMP implementation for GCC. `http://gcc.gnu.org/projects/gomp`.

[NT92] M. Noro and T. Takeshima. Risa/Asir — a computer algebra system. In P. S. Wang, editor, *Proceedings of ISSAC '92*, pages 387–396, Berkeley, CA, July 27–29 1992.

[OMN] Omni: OpenMP compiler project. `http://phase.hpcc.jp/Omni/home.ja.html`.

[Pat] PathScale, Inc. PathScale – Compiler Suite. `http://www.pathscale.com/ekopath.html`.

[The] The Portland Group. PGI High-Performance Compilers and Tools. `http://www.pgroup.com/`.

[村川 04] 村尾 and 川目. 数式処理のための基本線形演算プログラム集 MBLAS の開発. In *SACSIS 2004 - 先進的計算基盤システムシンポジウム. ポスター*, pages 139–140, 2004.