計算天文学の現在

牧野淳一郎

October 14, 2009

この文書の PDF 版は こちら[1]

[1] ../on_cs.pdf

Contents

Chapter 1

初めに

ここでは、計算天文学、あるいはシミュレーションを使う科学について、思うことをなんとなく書く。まあ、あんまり話は天文学はなくて、プログラム開発とかデバッグの話が当面続くかも。

Chapter 2

楽をしよう、ということ。(2009/6/8)

シミュレーションに限ったことではないが、研究、開発、といった行為の中では、必ず「同じことを (少しどこかを変更しながら) 繰り返し、その結果を整理、比較する」ということが起こる。

例えば、あるプログラムでマジックナンバーで埋め込まれている (そのこと自体が問題だが) パラメータを、5 個それぞれ 20 種類、但し、1 つを変える時には他は全て初期値固定でよいので、全部で 100 通り (あれ？本当は 96 通り？) のパターンでテストしよう、という時に、ある人は以下のようにしていた。

1. 1 つのパラメータだけをループで変化させてテストするプログラムを書く。
2. それでテストする。テスト終了まで待つ。
3. 他のパラメータのテストの度にプログラムを書き直す。

おそらく、そういうことをしてもプログラム修正を間違えない自信があるのかもしれないが、初めから 2 重ループで全部自動化したほうがよい。一度プログラム書いたらテストは計算機にまかせて寝ててもよいし、他の作業をすることもできるし、また後でテスト全体をやり直すことも容易だからである。

これが、手作業で行なう実験であれば繰り返すのもしかたがないかもしれない。もっとも、これも、手作業でがんばってしまうようでは科学の発展はなく、自動化して大量に均一な処理を行うことで新しいサイエンスができる、というのは天文学では近年の様々な大規模サーベイを見るまでもなく、例えば APM サーベイのころからそうなわけだ。

プログラム開発やデバッグの時の繰り返し作業の自動化、省力化のための有用な道具には、基礎的なものだけでも以下のようなものがある。(以下、Windows のことは無視する。UNIX にある基本的なツールだけを扱う)

- シェルのヒストリー機能
- シェルスクリプト
- make
- エディタのマクロ (emacs や vi の)

これらはそれぞれ極めて奥深いものであるが、全く基礎的なところだけでも知らないと知っている、また、知っていても使うと使わないでは作業能率が桁で変わってくる。桁で、というと大袈裟に聞こえるかもしれないが、これはそうではない。例えば、それほど規模が大きくなく、コンパイルして実

行すれば結果がわかる、というようなプログラムのデバッグサイクルを考えてみる。何もツールを使わないなら、例えば、

```
# cd ../src
# emacs foo.c
# cd ../bin
# cc --some_option --more_options .... -o foo.o ../src.foo.c
# cc --some_option -o xxx xxx.o yyy.o ... foo.o  -lm -lxxx -lyyy   ...
# ./xxx input_parameter1  input_parameter2 ....
```

というような作業を繰り返すことになるであろう。

2.1 エディタは常駐させよう

まず、するべきことはエディタは別ウインドウで開き、その中でセーブするだけで毎回新しく立ち上げたりはしないようにすることである。そんなことは当たり前である、と多くの読者が思ってくれるといいのだが、最近の学生・ポスドクにはそうでない人、つまり、編集操作毎に emacs を起動したり、はては複数のマシンで別に emacs を起動して同じファイルをオープンしたりといったことをする人も見受けられる。

複数のエディタを同時に開くことの問題点は、作業の整合性が失われる可能性があることである。もちろん、エディタ自体に、そういう時に人間が阿呆なことをしないように警告する機能はある。しかし、その警告をオーバーライドしてでも阿呆なことをするのが人間というものである。このような阿呆なことをすると数時間から数日といったオーダーで時間を無駄にすることになる。

エディタを常時開いていれば、上の操作手順から

```
(エディタでファイルを変更、セーブ)
# cc --some_option --more_options .... -o foo.o ../src.foo.c
# cc --some_option -o xxx xxx.o yyy.o ... foo.o  -lm -lxxx -lyyy   ...
# ./xxx input_parameter1  input_parameter2 ....
```

となって、無駄なディレクトリ移動をしなくてもよくなる。また、それ以上に重要なのは、ファイル操作の整合性が維持されやすくなることである。

¡P¿

2.2 make を使おう

次に重要なのは、コンパイル・リンクに make を使うことである。実際の使いかたの解説の例は例えば *make* を使おう[1]といったものをみてみよう。これを使うと

```
(エディタでファイルを変更、セーブ)
# make
# ./xxx input_parameter1  input_parameter2 ....
```

となる。make の重要なところは色々あるが、その一つは、コンパイルに失敗した時にはターゲットとなる実行ファイルは rm されているので、コンパイルに失敗しているのにちゃんと出来ているつもりになって延々時間を無駄にすることがない (そんなのは当たり前だと多くの読者が思うことを望む) ことである。

[1] http://www.notwork.org/~gotoken/mag/softwaredesign/

2.3 ヒストリー機能を使おう

シェルのヒストリー機能は、基本的なことは今の大抵のシェルなら Ctrl-P/N やカーソルキーで前に実行したコマンドを呼び出し、カーソルキー等で編集して実行、とできる、というものである。上の、コマンド 2 つを繰り返す場合だと、それぞれのコマンド実行が Ctrl-P Ctrl-P RET ですむので、1 度の実行が 6 ストロークですむことになる。もちろん、

```
（エディタでファイルを変更、セーブ）
# make;  ./xxx input_parameter1  input_parameter2 ....
```

とした上でヒストリー機能を使えば 2 ストロークである。

ここで重要なのは、単純にストローク数がいくつ、ということではなく、繰り返し作業において間違いが起きる可能性がほぼ 0 になる、ということである。

エディタ操作については次の項に。

Chapter 3

エディタで楽をしよう、ということ。(2009/6/8-6/17)

vi とか他のエディタは良く知らないので、以下は emacs の話。それも基本的なものに限る。これは、私があまりいろんなことを知っているわけではないからである。

3.1 消去と挿入

複数の行からなるエリアを消すのに、バックスペースや DEL キーを延々押し続けるのはよろしくない。

- Ctrl-k で 1 行づつ消す
- Ctrl-SPC で始点をマークし、Ctrl-w でカーソルまで消す

という程度は知っているべき。このどちらの場合にも、Ctrl-y で復活できる、というのも知っているべき。また、 Ctrl-y の直後に ESC-y で、以前に消した (あるいは ESC-w で取り込んだ) ものに置き換えることができる、というのも知っているべき。

3.2 繰り返しとマクロ

エディタにおいても、繰り返し操作が発生することはある。プログラムはそもそもなるべくそういうことがないように書いてあるべきだが、例えば設定ファイルで

```
10.10.0.1 grape001
10.10.0.2 grape002
10.10.0.3 grape003
....
```

と延々と続くファイルがあり、これを

```
10.10.0.1 grape001  grape001.cfca.nao.ac.jp
....
```

と FQDN もいれたくなったとしよう。まあ、これだと正規表現を使った置換で一発でできるが、以下のような編集操作を繰り返すのでもできる。

該当行に移動
先頭に移動
インクリメンタルサーチで gra... 辺りまで移動
g の位置の前まで戻る
ctrl-k で行の後ろまでをヤンク、直後に ctrl-y で復活
スペースの後もう一度 ctrl-y, その後 .cfca.nao.ac.jp をタイプ
次の行の先頭に移動

コマンド列としては以下のようになる

```
"\C-sgra\C-b\C-b\C-b\C-k\C-y \C-y.cfca.nao.ac.jp\C-n\C-a"
```

これを何度も繰り返すのは阿呆だが、一度これを実行してから繰り返し実行するのは「一連のキー操作をそのまま記録する」というキーボードマクロの機能で容易に行うことができる。これには、Ctrl-X (でキーボードマクロ定義モードにはいり、その後に上のコマンド入力をしたあとに Ctrl-X) でマクロ定義を終了する。そうすると、Ctrl-X e で同じコマンド列を再実行できる。さらに、ESC n Ctrl-X e (ここで n は数字。例えば 30 回なら ESC 30 Ctrl-X e) で、指定した回数実行できる。これで、定義ファイルが 1000 行とかあってもマクロで全部一度に変更できる。

ここでも、問題は編集操作自体にかかる時間自体ではない。繰り返し操作を人間が行うと必ず間違えるので、計算機に繰り返しを任せることでそのような間違いを防げることがより重要である。

なお、ここでさらに重要なことは、楽をするために必要なのは知識ではなく、楽をしよう、そのためのは楽をする方法を調べよう、という意志である、ということである。

上の 1000 行ある (かどうか知らないけど) 設定ファイルを編集する時に、

マクロとかよい方法があるかもしれないけど、今は急いでいるから。タイプしたって数分で終わるんだから、マクロとか勉強するよりこっちが速い。

というような理屈で、手作業で 1000 行編集するような方法を正当化し、実際にやってしまう人、というのが世の中には結構いる。このような思考パターンには以下のような問題がある。

- 手作業でやるために実際には間違いが入り、その間違いに気が付いて修正するまでに上の「数分」よりも何桁も長い時間を浪費する
- しかも、同じようなことをする機会がある度に同じような時間の浪費を繰り返す

つまり、「急いだ」ために実際には自分の仕事の効率を何桁も下げているわけである。本来優秀な人であっても、こういう瑣末なことで生産性を何桁も下げていては研究も進まない、ということになってしまう。

3.3 パラグラフ整形、インデント

Emacs でテキストファイルを書く時に、パラグラフ整形、というのがある、というのは、結構知らない人がいるんだろうか？ ESC-q (コマンド名は fill-paragraph) で、現在カーソルがあるパラグラフを、設定された 1 行あたりの文字数 (必要なら ESC 数字 Ctrl-X f で変更可能) になるように整形する。

例えば

Emacs でテキストファイルを書く時に、パラグラフ整形、というのが ある、というのは、結構知らない人がいるんだろうか？
ESC-q（コマンド名は fill-paragraph）で、現在カーソルがあるパラ
グラフを、設定された 1 行あたりの文字数（必要なら ESC 数字 Ctrl-X f で変更可能）になるように整形する。

が

Emacs でテキストファイルを書く時に、パラグラフ整形、というのが
ある、というのは、結構知らない人がいるんだろうか？ESC-q（コマン
ド名は fill-paragraph）で、現在カーソルがあるパラグラフを、設定
された 1 行あたりの文字数（必要なら ESC 数字 Ctrl-X f で変更可能）に
なるように整形する。

となる。エディタで文章を編集する時にはこっちのほうが読みやすいと思う。また、 auto-fill-mode を設定しておけば (あるいは、普通の文書用のメジャーモードをそうしておけば)、書いている時にも勝手に改行してくれる。

プログラムを書く時にも同様の機能があり、言語と、それ用の設定を理解して書いている時の自動インデントとか、領域の再インデントとかをする。コマンドは

Ctrl-i (TAB): 現在の行をインデント
Ctrl-META-\: indent-region, マークからカーソルまでをインデント

あたりが私が知っているものである。なお、もちろん、これは、Emacs がこうしたいと思うインデントになるので、それが自分のしたいものになるようにするためには、例えば c-indent-level とかいった変数を設定する必要がある。私の .emacs (Vine だと .emacs.my.el のような気が) に 20 年以上前に書いた設定はこんな感じである。

```
(setq text-mode-hook 'turn-on-auto-fill)
(setq mail-mode-hook '(lambda () (auto-fill-mode 1)))
(setq default-major-mode 'text-mode)

;; regulate indentation for new cc-mode.el
(defun my-c-mode-common-hook ()
  (setq c-indent-level 4)
  (setq c-basic-offset 4)
  (c-set-offset 'case-label 4)
  (c-set-offset 'statement-case-intro 4)
  (c-set-offset 'statement-block-intro 4)
  (setq c-continued-statement-offset 4)
  (setq c-brace-offset 0)
  (setq c-argdecl-indent 4)
  (setq c-label-offset 0)
  )
(add-hook 'c-mode-common-hook 'my-c-mode-common-hook)
```

hook で設定する必要があるのか、初期設定だけでよいのか不明だが、hook でしておいていけないことはないであろう。なんかいらない設定もいくつもあるような気もするし、現在 (Emacs 20 以降) だともっと簡単に書ける気もする。

Chapter 4

何故楽をしようとしないのか、ということ。(2009/6/10)

ここまでに書いたことは、要するに、

> 計算機のほうが上手くできることは計算機にやってもらって楽をしよう

ということである。こんなことは当たり前だと思いたいわけだが、意外にそうでもないような気がする。前章では、人がわざわざ時間と手間がかかる方法を選ぶ理由 (建前) として、

> よい方法があるかもしれないけど、今は急いでいるから。このやり方でも
> 数分で終わるんだから、勉強するよりこっちが速い。

というものを紹介した。しかし、これは本当に理由なのか、というのは、かなり疑問ではないかと思う。ここで根本にあるのは、

> 難しいことを考えたくない

つまり、「考えないことによって楽をしたい」という考え、あるいは

> 自分には難しいことはわからない、したくない

という、自己評価の低さ、自信のなさ、それと表裏一体の

> 「もう少し賢くやれ」といわれることへの反発

ではないかと思う。まあ、もちろん、単に私の教えかたが学生にとって不愉快なものであるからかもしれない。私の知る限りでは、多くの日本人の大学教師は、学生を相手にする時に

> > こんなことも知らないの？

という調子の、かなり学生から見ると不愉快な態度をとる。これに対して、私が個人的に知っている欧米の大学教師にはあまりそういう態度をとる人はいないように思われる。もちろん、これは単になんらかのセレクション・バイアスのせいかもしれないし、良く知らない欧米の大学教師には結構アレ

な人がいるというのも全く知らないわけではないので、日本人だからどうというわけではなくて大学教師というものがある程度一般的にそういう傾向があるのではないかと思う。

もちろん、研究を進める、特に自力で研究を進める上では、自分の研究成果に対する、多くの場合には理不尽な批判に耐え、さらにはそういう批判に反論して叩きつぶすことがどうしても必要であり、そのための訓練を早い時期からする必要がある、というのは確かであり、また特に日本人にはそのような、他人に対して攻撃的な態度をとることは難しい、ということはあるかもしれない。

が、そういうのは発表練習の時とか、実際に論文を書いた時にすればよいわけで、普段からそういうやり方で指導する必要はない。

これは、(大学) 教師の側だけの問題、というわけでもないと思う。というのは、日本の教育では、「自分で考えて、試行錯誤して問題を解決する」というようなことは特に推奨されないのに対して、研究も、またプログラミングも、「自分で考えて、試行錯誤して問題を解決する」ことそのものだからである。しかし、大学学部までの教育で教えられ、身につくのは、せいぜいが「教師の望むような答を無意識に推測する」能力であって、未知の問題を解決する能力ではない。もちろん、たまたまどういうわけか未知の問題を解決する能力を自力で身につけた学生が、自覚的に教師の望む (あるいは望まないけれど減点しない、あるいは少ししか減点しない) 回答を作ることで定期試験や入試のシステムを突破してくることもある。この場合には、教師への不信感が色々なことの妨げになることもある。自力で問題解決ができる範囲では OK だが、自力で身につけたスキルがあまり高くない場合には、低いレベルでできることが止まってしまうことになる。

つまり、大学教師に側に教えるスキルがないだけでなく、学生の側に何かを教わる経験がない、という問題もある。

まあ、この辺はいわゆる「コーチング」とかそういう技術の問題、という面はもちろんあるが、やりたくないことをやってもらうのはどうしても難しい。いわれたことはしなくてもどこかで読んだことはやってみるとか、その逆とか、人はそれぞれなので、その人に合いそうな色々な方法をとってみて上手くいくものを、というくらいかもしれない。

まあ、その、こんな文章を書いているのは要するにそういうことである。

Chapter 5

楽なやり方は一つではない、ということ。(2009/6/12)

デバッグ、というのは何を作っている時にも必要な作業だが、どういうやり方がよいか、というのは時と場合によって違う、というのは必ずしも理解されているとは限らないような気もする。

例えば、ちょっとしたスクリプトなら、まず書いてみて、実行し、結果が期待と違っていたらそれから考える、というのでさほど問題はない。が、極端な例としてカスタム LSI を作る、という話だと、どこか間違っていたら修正に数ヶ月と、莫大なお金が必要になる。 GRAPE なんかだとそもそもそんな、チップがバグっていた時に作り直すための予算自体がないので、致命的な (こちらのせいの) ミスがあったところでプロジェクトが崩壊する。

こういう場合には、実チップができる前にあらゆるテストをすることになる。

お金がからまなくても、問題の箇所を特定するための実行や修正、修正後のコンパイルといった作業に非常に長い時間がかかる場合には、修正してから数秒で結果がわかる場合と同じやり方をしていてはいけない。例えば、ハードウェア開発をしていて FPGA の中身を変更しないといけない、しかし FPGA の中身のコンパイラは一度のコンパイルに 2 時間かかる、あるいは並列プログラムを開発していて最大コア数の時にだけでるバグがあるが、最大コア数では一日に一度しかジョブが流れない、といった場合である。

私は経験したことはないが、大昔に計算を使う時には普通のジョブでも一日に一度しかジョブが流れないものだったようで、昔からある (言い換えると、昔からあまり改善されていない) 状況がある、という話である。

昔の人がどうしたかというと、結局、

> 良く考えた

ということである。計算機というものが他にないので、使えるものは自分の頭と、1970 年代になれば電卓くらいは使える、ということになる。よく考える、というのはもちろん重要なことだが、FPGA の中身の開発、となると全く違うアプローチもある。それは、 FPGA の中身を書き換えなくても色々な変更ができるようにする、ということである。手軽に中身を変更できるのが FPGA ではなかったか？と思うとこれはなんだか変な気もするが、「手軽」が 2 時間ならもっと違う方法が必要である。

とはいえ、一日に一度しかし実行できない昔の計算機と、コンパイル等に数時間かかるとはいえシミュレーションや実行は何度でもできる FPGA ではデバッグのしかたは根本的に違う。

例えば、通信路のハードウェアエラーをチェックする回路を作ったとしよう。これが正しく動作しているかどうかを調べるためには、人為的にハードウェアエラーを起こして、それが検出されるかど

うかを調べる必要がある。

このチェックをするのに、以下のようなことをしているのをみたことがある。

1. 常にエラーを出すようにハードウェアを書き換える。
2. コンパイル、起動、実行してエラー検出ができることをチェックする。
3. エラーをださない回路に戻す。

コンパイルに数時間かかるのは良いとして、このやり方の問題は、チェックしたエラーパターンが 1 通りしかないということであったことが後で判明した。もちろん、最初からエラー検出が正しく動いたわけではないので、エラーを出す回路、エラーを出さない回路のどちらも何度もコンパイル、ハードウェアに転送、実行のサイクルを繰り返す必要があり、一日ではおわっていなかった、ということはさておき、これで作者が正しく作ったつもりになっていた回路が実は全く間違っていた。

間違っていた理由は簡単で、上のやり方では可能なエラーパターンのうち 1 つだけしかみていなかったが、実際の回路では 15 箇所くらいから出るエラーを 2 ビットに縮約したエラー状態をチェックするようになっていた、ということである。この 15 箇所が全てエラーを出した場合のみのチェックをしていた。

このようなやり方になったのは、ハードウェアを変更してエラーを出す、というやり方では、本来チェックするべき多様なエラーパターンをチェックするのにはあまりに長い時間がかかる、と意識的になり無意識になり思ったからであろう、と想像される。

どのような回路にするべきであったか、というと、外部からのレジスタへの書き込み等で、ハードウェアを変更しなくても任意のエラーパターンを出すことができるような回路にして、それらが正しく検出できるかどうかをテストする、というのが最低限するべきことである。問題の回路はパリティ生成回路なので、単純にパリティビット毎に even か odd かを指定するフラグをつける、というか、回路的にはレジスタと xor 1 つをつけるだけである。

このように、チェックをつける回路をいれても規模・速度的に問題にならないなら、いれておくのが正しいやり方である。あるパターンのエラーを発生させるためだけのために FPGA の回路を変更する、という方法では、エラーパターンを網羅することが不可能なだけでなく、後でテストしたくなったらまたコンパイルからやり直しになるからである。

なお、このように後からまたテストするためには、テスト自体が自動化されていることが必須である。つまり、基本的にスクリプトをパラメータなしで走らせて、スクリプト自体が出力結果があっているか間違っているか判定できないといけない。

この辺の考え方、開発手法をもうちょっと体系化したのが「テスト駆動開発」である。これは沢山書物もあるが、基本的なところは、例えば「テスト駆動開発」はプログラマのストレスを軽減するか？[1]を読めばわかると思う。基本的には、

- モジュール単位で、そのテストを自動的に行うためのテストプログラムと、プログラム実体を並行して開発する。
- 結果的に、テストプログラムが仕様記述になる。
- このテストプログラムは、そのモジュールを「十分に」テストするようにする、つまり、想定する様々な使い方で正しい答がでるかどうか確認できるようにする。
- モジュールを変更したら必ずテストする。これは、 makefile とかに書けばよい。

というような話である。

[1] http://www.atmarkit.co.jp/fdotnet/special/tdd/tdd_01.html

と、こんなわけで、何か変更するのに数時間かかる、というような場合なら、テストやデバッグのためのコードや回路をあらかじめ作りこんでおいて、なるべく少ない変更回数で間違いを修正する、目的の機能を実現する、というのが楽をする方法になる。これに対して、例えば数行のスクリプトで実行したら瞬時に結果がでるなら、大がかりなテスト用コードを書いている間にちょこちょこソースを変更して実行してみるほうが早かもしれない。

Chapter 6

プログラムを楽に書く。(2009/6/13)

プログラムをどういうふうに書くのが楽か、というのは、ある程度楽に書いている人にとってはそうでない人にわかるように説明するのは非常に難しいことである。というのは、「楽でない書き方」は極めて不自然かつ奇妙なものに見えるので、思い付くことも真似をすることも難しいからである。

これは、例えばCプログラミング診断室[1] といった、そのような実例を集めて、何が問題かを解説している本を読んで見るのが、楽な書き方がどんなものか既にわかっている人にも、そうでなく、この本にあるような書き方をしている人にも有益であると思う。

ネット上ではこの本については評価が分裂している。悪い評価は、

- 自分が書いているコードとそっくりなものが例にでていて不快になった
- とても人間が書くものとは思えないような、ありえない例がでているので無意味に感じられる

のどちらかであろう。以下、某君が書いていたプログラムを例に、同じようなことを考えてみる。もう5年くらい前なので時効であろう。今はもうこんなプログラムは書かない立派な研究者になっているし。

```
#include <stdio.h>
#include <math.h>
#include "bh.h"


/* ---------------------------------- */
void get_force_host(int ni, int nj,
                    double *m, double (*x1)[3], double (*v1)[3],
                    double (*xi)[3], double (*vi)[3],
                    double (*ai)[3], double (*ji)[3],
                    double *eps2i, int *indexi, double *pi)
{
  int ii, j, k, idi;
  double rinv, r2, r2inv, r3inv, r5inv;
  double xdotv, xdotvr5inv,r3invdx;
  double r3invdvetc;
  double dx[3], dv[3];
```

[1] http://www.pro.or.jp/~fuji/mybooks/cdiag/

```
  for(ii=0;ii<ni;ii++){
    for (k = 0; k < 3; k++) {
      ai[ii][k] = 0.0;
      ji[ii][k] = 0.0;
    }
    pi[ii] = 0.0;
  }

  //  ALL <== FS
  for(ii=0;ii<ni;ii++){
    idi = indexi[ii];
    for(j=0;j<nj;j++){
      if(idi!=j){
        r2 = eps2i[ii];
        xdotv = 0.0;
        for (k = 0; k < 3; k++) {
          dx[k] = x1[j][k] - xi[ii][k] ;
          dv[k] = v1[j][k] - vi[ii][k] ;
          r2 += dx[k] * dx[k];
          xdotv += dx[k] * dv[k];
        }
        r2inv      =    1.0 / r2 ;
        rinv       =    sqrt(r2inv);
        r3inv      =  rinv * r2inv;
        r5inv      = r3inv * r2inv;
        xdotvr5inv = 3.0 * xdotv * r5inv;
        for (k = 0; k < 3; k++) {
          r3invdx     = r3inv * dx[k];
          r3invdvetc  = r3inv * dv[k] - xdotvr5inv * dx[k];

          if(r3invdx    == 0.0) fprintf(stderr,"idi=%d  ii=%d j=%d ",idi,ii,j);
          if(r3invdvetc == 0.0) fprintf(stderr,"idi=%d  ii=%d j=%d ",idi,ii,j);

          ai[ii][k]  +=  m[j] * r3invdx;
          ji[ii][k]  +=  m[j] * r3invdvetc;
        }
        pi[ii] -= m[j] * rinv;
      }
    }
  }
}

void get_force_from_bh(int ni, int nj,
                       double *m,
                       double (*x1)[3], double (*v1)[3],
                       double (*xi)[3], double (*vi)[3],
                       double (*ai)[3], double (*ji)[3],
                       double *eps2, int *indexi,double *pi,
                       int BHnum, int *BHi, double eps_bb2)
{
  int ii, j, k,idi;
```

```
  double rinv, r2, r2inv, r3inv, r5inv;
  double xdotv, xdotvr5inv,r3invdx;
  double r3invdvetc;
  double dx[3], dv[3];


  for(ii=0;ii<ni;ii++){
    idi = indexi[ii];
    // --- ALL <== BH ---
    for (j = nj; j < (nj+BHnum) ; j++) {
      if(idi!=j){
        if(idi <  nj ) r2 = eps2[j];
        if(idi >= nj ) r2 = eps_bb2;
        xdotv = 0.0;
        for (k = 0; k < 3; k++) {
          dx[k] = x1[j][k] - xi[ii][k] ;
          dv[k] = v1[j][k] - vi[ii][k] ;
          r2 += dx[k] * dx[k];
          xdotv += dx[k] * dv[k];
        }
        r2inv      =    1.0 / r2 ;
        rinv       =    sqrt(r2inv);
        r3inv      =  rinv * r2inv;
        r5inv      = r3inv * r2inv;
        xdotvr5inv = 3.0 * xdotv * r5inv;
        for (k = 0; k < 3; k++) {
          r3invdx       = r3inv * dx[k];
          r3invdvetc    = r3inv * dv[k] - xdotvr5inv * dx[k];

          if(r3invdx    == 0.0) fprintf(stderr,"idi=%d  ii=%d j=%d ",idi,ii,j);
          if(r3invdvetc == 0.0) fprintf(stderr,"idi=%d  ii=%d j=%d ",idi,ii,j);

          ai[ii][k]  +=   m[j] * r3invdx;
          ji[ii][k]  +=   m[j] * r3invdvetc;
        }
        pi[ii] -= m[j] * rinv;
      }
    }
  }
}
/* ---------------------------------------------- */



/*  ---------------------------------------------- */
void get_force_initial(int n, double *eps2, double *m,
                       double (*x0)[3], double (*v0)[3],
                       double (*a0)[3], double (*j0)[3], double *p,
                       int BHnum, int *BHi, double eps_bb2)
{
  int i, j, k, b;
  double rinv, r2, r2inv, r3inv, r5inv;
```

```
double xdotvr5inv,r3invdx, r3invdvetc;
double xdotv, dx[3], dv[3];

for (i = 0; i < n; i++) {
  for (k = 0; k < 3; k++) {
    a0[i][k] = 0.0;
    j0[i][k] = 0.0;
  }
  p[i] = 0.0;
}
/* --- FS <== FS --- */
for (i = 0; i < (n -BHnum); i++) {
  for (j = i+1; j < (n-BHnum); j++) {
    r2 = eps2[i];
    xdotv = 0.0;
    for (k = 0; k < 3; k++) {
      dx[k] = x0[j][k] - x0[i][k] ;
      dv[k] = v0[j][k] - v0[i][k] ;
      r2 += dx[k] * dx[k];
      xdotv += dx[k] * dv[k];
    }
    r2inv      =    1.0 / r2 ;
    rinv       =    sqrt(r2inv);
    r3inv      =  rinv * r2inv;
    r5inv      = r3inv * r2inv;
    xdotvr5inv = 3.0 * xdotv * r5inv;
    for (k = 0; k < 3; k++) {
      r3invdx     = r3inv * dx[k];
      r3invdvetc  = r3inv * dv[k] - xdotvr5inv * dx[k];
      a0[i][k] += m[j] * r3invdx;
      a0[j][k] -= m[i] * r3invdx;
      j0[i][k]  +=   m[j] * r3invdvetc;
      j0[j][k]  += - m[i] * r3invdvetc;
    }
    p[i] -= m[j] * rinv;
    p[j] -= m[i] * rinv;
  }
}

/* --- FS <==> BH --- */
for (b = 0; b < BHnum ; b++) {
  j = BHi[b];
  for (i = 0; i < (n-BHnum) ; i++) {
    r2 = eps2[j];
    xdotv = 0.0;
    for (k = 0; k < 3; k++) {
      dx[k] = x0[j][k] - x0[i][k] ;
      dv[k] = v0[j][k] - v0[i][k] ;
      r2 += dx[k] * dx[k];
      xdotv += dx[k] * dv[k];
    }
    r2inv      =    1.0 / r2 ;
```

```
      rinv       =    sqrt(r2inv);
      r3inv      =  rinv * r2inv;
      r5inv      = r3inv * r2inv;
      xdotvr5inv = 3.0 * xdotv * r5inv;
      for (k = 0; k < 3; k++) {
        r3invdx     = r3inv * dx[k];
        r3invdvetc  = r3inv * dv[k] - xdotvr5inv * dx[k];
        a0[i][k] += m[j] * r3invdx;
        a0[j][k] -= m[i] * r3invdx;
        j0[i][k]  +=   m[j] * r3invdvetc;
        j0[j][k]  += - m[i] * r3invdvetc;
      }
      p[i] -= m[j] * rinv;
      p[j] -= m[i] * rinv;
    }
  }

  /* ---  B1 <=> B2  --- */
  if(BHnum !=0 ){
    for (i = n-BHnum ; i < n; i++) {
      for (j = i+1; j < n; j++) {
        r2 = eps_bb2;
        xdotv = 0.0;
        for (k = 0; k < 3; k++) {
          dx[k] = x0[j][k] - x0[i][k] ;
          dv[k] = v0[j][k] - v0[i][k] ;
          r2 += dx[k] * dx[k];
          xdotv += dx[k] * dv[k];
        }
        r2inv      =    1.0 / r2 ;
        rinv       =    sqrt(r2inv);
        r3inv      =  rinv * r2inv;
        r5inv      = r3inv * r2inv;
        xdotvr5inv = 3.0 * xdotv * r5inv;
        for (k = 0; k < 3; k++) {
          r3invdx     = r3inv * dx[k];
          r3invdvetc  = r3inv * dv[k] - xdotvr5inv * dx[k];
          a0[i][k] += m[j] * r3invdx;
          a0[j][k] -= m[i] * r3invdx;
          j0[i][k]  +=   m[j] * r3invdvetc;
          j0[j][k]  += - m[i] * r3invdvetc;
        }
        p[i] -= m[j] * rinv;
        p[j] -= m[i] * rinv;
      }
    }
  }

}
```

これは、直接重力計算 (全ての粒子からの力を本当に計算する) の N 体プログラムで、粒子に非常

に重いブラックホール (といっても相対論的効果を考えるわけではなくて単に重いニュートン力学、ニュートン重力の質点) と、それ以外の普通の粒子 (星) がある場合を扱おうというものである。で、物理モデルとしては非常に重い銀河中心にあるブラックホール (SMBH, supermassive black hole)、もうちょっと軽い「中間質量ブラックホール」(IMBH, intermediate-mass black hole)、それから普通の星、というものを扱いたいものである。 それぞれの質量は

SMBH $\sim 10^7$ 太陽質量

IMBH $\sim 10^3$ 太陽質量

普通の星 ~ 10 太陽質量

というところである。

で、重力を計算するのに本当に質点でやると、粒子がたまたま近づいた時に計算が発散する。それを防ぐために、重力ポテンシャルエネルギーを

$$\psi_{ij} = -Gm_i m_j \frac{1}{r_{ij}} \tag{6.1}$$

というものから、

$$\psi_{ij} = -Gm_i m_j \frac{1}{\sqrt{r_{ij}^2 + \epsilon^2}} \tag{6.2}$$

というものに変えるのが普通である。これをどのような場合にやってよいか、また、ϵ にどの程度の大きさを取るべきかはここでは議論しないが、ブラックホールの影響を考える場合には、これらについてはこれらの回りの星の軌道が大きく変わってしまうような値にしてはいけない。ちなみに、この ϵ のことを通常ソフトニング長という。面倒なので単にソフトニングということも多い。

これは小さいほど計算は正しいわけだが、何も考えないで小さくすると計算が大変なので、計算がおかしくならない範囲で大きくしたい。 SMBH と IMBH の質量は非常に大きく違うので、このことは、以下の 4 種類の粒子の組合せでソフトニングを変える必要があるということを意味する。

- 普通の星-普通の星
- SMBH-普通の星
- IMBH-普通の星
- SMBH-IMBH

重力ポテンシャルは作用・反作用の法則を満たす必要があるから、 普通の星から例えば SMBH への重力を計算する時と、その逆を計算する時のソフトニングは同じでないといけない。つまり、上の 4 つの場合わけが発生する。

というようなことを考えて某君 (以下 A 君) が書いたのが上のプログラムである。

これは get_force_host、 get_force_host_bh と get_force_initial の 3 つの関数がある、重力計算だけのファイルの中身を示している。

プログラムを見ると、なんとなく繰り返しが多いなあという気がする。

```
r2 = eps2;
xdotv = 0.0;
for (k = 0; k < 3; k++) {
  dx[k] = x1[j][k] - xi[ii][k] ;
```

```
          dv[k] = v1[j][k] - vi[ii][k] ;
          r2 += dx[k] * dx[k];
          xdotv += dx[k] * dv[k];
        }
        このあとはちょっと違うことがある
```

というのが粒子間の重力計算の本体だが、これが都合 5 回でてきていることがわかる。 get_force_host は時間積分の間に重力を計算する関数で、これはある時刻では一部の粒子への力しか計算しない。このため、重力計算自体では対称性をつかわないで 1 つの粒子への力を求めるのに他の全粒子からの力をそのまま計算する。動かす粒子の数が ni 個、全粒子の数は n 個となっている。 get_force_initial のほうは全粒子から全粒子への力を計算するので、対称性を使って一度距離を計算したらそれを使って両方の粒子の加速度を計算することで、計算量が半分とはいかないがある程度減らしている。get_force_host_bh のほうはブラックホール粒子からの力を計算する。

まず、 get_force_host とその _bh がついた版のほうをみてみよう。同じコードが 2 回でてくるならそれを関数にできるので、そうすることを考えてみる。

```
void pairwise_force(double xj[3],
                    double vj[3],
                    double mj,
                    double eps2,
                    double x[3],
                    double v[3],
                    double a[3],
                    double j[3],
                    double *p)
{
    double r2, xdotv, r2inv, rinv, r3inv, r5inv, xdotvr5inv;
    double dx[3], dv[3];
    int k;
        r2 = eps2;
        xdotv = 0.0;
        for (k = 0; k < 3; k++) {
          dx[k] = xj[k] - x[k] ;
          dv[k] = vj[k] - v[k] ;
          r2 += dx[k] * dx[k];
          xdotv += dx[k] * dv[k];
        }
        r2inv      =    1.0 / r2 ;
        rinv       =    sqrt(r2inv);
        r3inv      =  rinv * r2inv;
        r5inv      = r3inv * r2inv;
        xdotvr5inv = 3.0 * xdotv * r5inv;
        for (k = 0; k < 3; k++) {
          a[k]  +=   mj * r3inv * dx[k];
          j[k]  +=   mj * (r3inv * dv[k] - xdotvr5inv * dx[k]);
        }
        *p -= mj * rinv;
}
/* ---------------------------------- */
void get_force_host(int ni, int nj,
                    double *m, double (*x1)[3], double (*v1)[3],
```

```
                    double (*xi)[3], double (*vi)[3],
                    double (*ai)[3], double (*ji)[3],
                    double *eps2i, int *indexi, double *pi)
{
  int ii, j, k, idi;
  for(ii=0;ii<ni;ii++){
    for (k = 0; k < 3; k++) {
      ai[ii][k] = 0.0;
      ji[ii][k] = 0.0;
    }
    pi[ii] = 0.0;
  }
  //  ALL <== FS
  for(ii=0;ii<ni;ii++){
    idi = indexi[ii];
    for(j=0;j<nj;j++){
      if(idi!=j){
          pairwise_force(x1[j],v1[j],m[j],eps2i[ii],
                         xi[ii],vi[ii],ai[ii],ji[ii],pi+ii);
      }
    }
  }
}

void get_force_from_bh(int ni, int nj,
                       double *m,
                       double (*x1)[3], double (*v1)[3],
                       double (*xi)[3], double (*vi)[3],
                       double (*ai)[3], double (*ji)[3],
                       double *eps2, int *indexi,double *pi,
                       int BHnum, int *BHi, double eps_bb2)
{
  int ii, j, k,idi;
  double eps2;
  for(ii=0;ii<ni;ii++){
    idi = indexi[ii];
    // --- ALL <== BH ---
    for (j = nj; j < (nj+BHnum) ; j++) {
      if(idi <  nj ) eps2 = eps2[j];
      if(idi >= nj ) eps2 = eps_bb2;
      if(idi!=j){
          pairwise_force(x1[j],v1[j],m[j],eps2,
                         xi[ii],vi[ii],ai[ii],ji[ii],pi+ii);
      }
    }
  }
}
```

こんなの。元の関数 2 つが 110 行くらいだったのが、関数 3 つ合わせても 80 行になってだいぶ短くなった。しかし、書き直したものを良くみると (良くみなくても元からそうなのだが) 新しく作った pairwise_force を持つ 2 つの 2 重ループ自体が同じ構造をしています。違うのは eps2 の与えかた

だけである。ですから、この eps2 の与えかたのロジックを整理すればもうちょっと簡単になるはずである。

与えかたのロジックを調べてみると、なんだか良くわからない。_bh がつかないほうでは eps2 という配列を使っている。これに対してつくほうでは、自分のインデックスである idi が nj より大きい時は eps_bb2 というものを使い、それ以外の場合には配列を使っている。

ということは、このプログラムでは、ブラックホール粒子は nj より先のところに入っているということである。 このことを素直に表現するなら、

```
#include <stdio.h>
#include <math.h>
#include "bh.h"


/* ---------------------------------- */
void pairwise_force(double xj[3],
                    double vj[3],
                    double mj,
                    double eps2,
                    double x[3],
                    double v[3],
                    double a[3],
                    double j[3],
                    double *p)
{
    double r2, xdotv, r2inv, rinv, r3inv, r5inv, xdotvr5inv;
    double dx[3], dv[3];
    int k;
    r2 = eps2;
    xdotv = 0.0;
    for (k = 0; k < 3; k++) {
        dx[k] = xj[k] - x[k] ;
        dv[k] = vj[k] - v[k] ;
        r2 += dx[k] * dx[k];
        xdotv += dx[k] * dv[k];
    }
    r2inv      =    1.0 / r2 ;
    rinv       =    sqrt(r2inv);
    r3inv      =  rinv * r2inv;
    r5inv      = r3inv * r2inv;
    xdotvr5inv = 3.0 * xdotv * r5inv;
    for (k = 0; k < 3; k++) {
        a[k]  +=   mj * r3inv * dx[k];
        j[k]  +=   mj * (r3inv * dv[k] - xdotvr5inv * dx[k]);
    }
    *p -= mj * rinv;
}
/* ---------------------------------- */
void get_force_host(int ni, int nj,
                    double *m, double (*x1)[3], double (*v1)[3],
                    double (*xi)[3], double (*vi)[3],
                    double (*ai)[3], double (*ji)[3],
```

```
                    double int *indexi, double *pi)
{
    int ii, j, k, idi;
    eps2[3];
    for(ii=0;ii<ni;ii++){
        for (k = 0; k < 3; k++) {
            ai[ii][k] = 0.0;
            ji[ii][k] = 0.0;
        }
        pi[ii] = 0.0;
    }
    //  ALL <== FS
    for(ii=0;ii<ni;ii++){
        idi = indexi[ii];
        if(idi!=j){
            for(j=0;j<nj;j++){
                if (idi<nj){
                    if (j<nj){
                        eps2=eps2_ff;
                    }else if (j==nj){
                        eps2=eps2_bh1;
                    }else{
                        eps2=eps2_bh2;
                    }
                }else{
                    if (j>nj){
                        eps2=eps2_bh;
                    }else if (idi==nj){
                        eps2=eps2_bh1;
                    }else{
                        eps2=eps2_bh2;
                    }
                }
                pairwise_force(x1[j],v1[j],m[j],eps2,
                               xi[ii],vi[ii],ai[ii],ji[ii],pi+ii);
            }
        }
    }
}
```

のように、多少複雑だが条件分岐で与えればいい (このコードはコンパイルできない)。もっとも、もうちょっとましな方法もある。以下をみてみよう。

```
#include <stdio.h>
#include <math.h>
#include "bh.h"


/* ---------------------------------- */
void pairwise_force(double xj[3],
                    double vj[3],
```

```
                    double mj,
                    double eps2,
                    double x[3],
                    double v[3],
                    double a[3],
                    double j[3],
                    double *p)
{
    double r2, xdotv, r2inv, rinv, r3inv, r5inv, xdotvr5inv;
    double dx[3], dv[3];
    int k;
        r2 = eps2;
        xdotv = 0.0;
        for (k = 0; k < 3; k++) {
          dx[k] = xj[k] - x[k] ;
          dv[k] = vj[k] - v[k] ;
          r2 += dx[k] * dx[k];
          xdotv += dx[k] * dv[k];
        }
        r2inv      =    1.0 / r2 ;
        rinv       =    sqrt(r2inv);
        r3inv      =  rinv * r2inv;
        r5inv      = r3inv * r2inv;
        xdotvr5inv = 3.0 * xdotv * r5inv;
        for (k = 0; k < 3; k++) {
          a[k]  +=   mj * r3inv * dx[k];
          j[k]  +=   mj * (r3inv * dv[k] - xdotvr5inv * dx[k]);
        }
        *p -= mj * rinv;
}
void force_from_array(int index,
                      int nj;
                      double xj[][3],
                      double vj[][3],
                      double mj[],
                      double eps2,
                      double x[3],
                      double v[3],
                      double a[3],
                      double jerk[3],
                      double *p)
{
    int j;
    for(j=0;j<nj;j++){
        if (j!=index){
            pairwise_force(xj[j],vj[j],mj[j],eps2,xi,v,a,jerk,p);
        }
    }
}

void get_force_host(int ni, int nj,
                    double *m, double (*x1)[3], double (*v1)[3],
```

```
                    double (*xi)[3], double (*vi)[3],
                    double (*ai)[3], double (*ji)[3],
                    double int *indexi, double *pi)
{
    int ii, j, k, idi;
    for(ii=0;ii<ni;ii++){
        for (k = 0; k < 3; k++) {
            ai[ii][k] = 0.0;
            ji[ii][k] = 0.0;
        }
        pi[ii] = 0.0;
    }
    for(ii=0;ii<ni;ii++){
        double eps2 = eps2_ff;
        int i = indexi[ii];
        if (i<nj){
            pairwise_force(x1[nj],v1[nj],m[nj],eps2_bh1,
                           xi[i],vi[i],ai[i],ji[i],pi+i);
            pairwise_force(x1[nj+1],v1[nj+1],m[nj+1],eps2_bh2,
                           xi[i],vi[i],ai[i],ji[i],pi+i);
        }else{
            eps2 = eps2_bh1;
            jbh=nj+1;
            if (i>nj) {
                eps2 = eps2_bh2;
                jbh=nj;
            }
            pairwise_force(x1[jbh],v1[jbh],m[jbh],eps2_bb,
                           xi[i],vi[i],ai[i],ji[i],pi+i);
        }
        force_from_array(nj,idi,x1,v1,m,eps2,
                         xi[i],vi[i],ai[i],ji[i],pi+i);
    }
}
```

ここでは、「多数の星からの近くを同じソフトニングで計算する」という機能を関数にまとめることで、コードとして見通しがよいものにしている。これにはおまけもあって、このような形で整理されていれば、この部分だけを SIMD 命令とか GRAPE とかいったもので高速化する、というのが容易になる。

初期化の部分では、対称性を利用して計算量を半分強にしているが、問題はそれによって得るもの、失うものがどれくらいあるかである。2 倍弱計算量が増えることを気にしなければ、ここはもちろん普通に重力を計算する関数を全粒子について呼ぶだけですむ。実際のシミュレーションのことを考えると初期化に必要な時間は誤差の範囲であり、この部分を 2 倍弱高速化することにあまり意味はない。また、SIMD 等での高速化を図るなら、対称性の利用のために別ルーチンを書いているのは余計な手間を増やすだけになる。

もしも、常に初期化のほうに時間がかかる (時間積分プログラムに場合には原理的にありえないが) というような場合には、ここでの 2 倍に意味があるかもしれないが、そういう状況であるとわかっている場合でなければここでわざわざ手間を増やすようなことはしてはいけない。

さて、このように順番に見ていくと、何故最初の例のようなプログラムを書くのか？ということが疑問に感じられるかもしれない。というか、疑問に感じないようではちょっと問題だが、でも、実際

に自分が書いたプログラムのことをふりかえってみた時に最初の例のようなことをしていない、と言い切ることができる人はあまりいないのではないかと思う。

最初の例のようなプログラムができるのは、ブラックホールとかないプログラムから、場当り的にプログラムを変更していってできたものをそのまま放置するからである。つまり、元々は

```
#include <stdio.h>
#include <math.h>
#include "bh.h"

#include <stdio.h>
#include <math.h>
#include "bh.h"

void get_force(int ni, int nj,
               double *m, double (*x1)[3], double (*v1)[3],
               double (*xi)[3], double (*vi)[3],
               double (*ai)[3], double (*ji)[3],
               double eps2, int *indexi, double *pi)
{
    int ii, j, k, idi;
    double rinv, r2, r2inv, r3inv, r5inv;
    double xdotv, xdotvr5inv,r3invdx;
    double r3invdvetc;
    double dx[3], dv[3];

    for(ii=0;ii<ni;ii++){
        for (k = 0; k < 3; k++) {
            ai[ii][k] = 0.0;
            ji[ii][k] = 0.0;
        }
        pi[ii] = 0.0;
    }

    for(ii=0;ii<ni;ii++){
        idi = indexi[ii];
        for(j=0;j<nj;j++){
            if(idi!=j){
                r2 = eps2;
                xdotv = 0.0;
                for (k = 0; k < 3; k++) {
                    dx[k] = x1[j][k] - xi[ii][k] ;
                    dv[k] = v1[j][k] - vi[ii][k] ;
                    r2 += dx[k] * dx[k];
                    xdotv += dx[k] * dv[k];
                }
                r2inv      =    1.0 / r2 ;
                rinv       =    sqrt(r2inv);
                r3inv      =  rinv * r2inv;
                r5inv      = r3inv * r2inv;
                xdotvr5inv = 3.0 * xdotv * r5inv;
                for (k = 0; k < 3; k++) {
```

```
                    r3invdx     = r3inv * dx[k];
                    r3invdvetc  = r3inv * dv[k] - xdotvr5inv * dx[k];
                    ai[ii][k]  +=  m[j] * r3invdx;
                    ji[ii][k]  +=  m[j] * r3invdvetc;
                }
                pi[ii] -= m[j] * rinv;
            }
        }
    }
}
void get_force_initial(int n, double eps2, double *m,
                       double (*x0)[3], double (*v0)[3],
                       double (*a0)[3], double (*j0)[3], double *p)
{
    int i, j, k, b;
    double rinv, r2, r2inv, r3inv, r5inv;
    double xdotvr5inv,r3invdx, r3invdvetc;
    double xdotv, dx[3], dv[3];

    for (i = 0; i < n; i++) {
        for (k = 0; k < 3; k++) {
            a0[i][k] = 0.0;
            j0[i][k] = 0.0;
        }
        p[i] = 0.0;
    }
    for (i = 0; i < (n -BHnum); i++) {
        for (j = i+1; j < (n-BHnum); j++) {
            r2 = eps2[i];
            xdotv = 0.0;
            for (k = 0; k < 3; k++) {
                dx[k] = x0[j][k] - x0[i][k] ;
                dv[k] = v0[j][k] - v0[i][k] ;
                r2 += dx[k] * dx[k];
                xdotv += dx[k] * dv[k];
            }
            r2inv      =    1.0 / r2 ;
            rinv       =    sqrt(r2inv);
            r3inv      =  rinv * r2inv;
            r5inv      = r3inv * r2inv;
            xdotvr5inv = 3.0 * xdotv * r5inv;
            for (k = 0; k < 3; k++) {
                r3invdx     = r3inv * dx[k];
                r3invdvetc  = r3inv * dv[k] - xdotvr5inv * dx[k];
                a0[i][k] += m[j] * r3invdx;
                a0[j][k] -= m[i] * r3invdx;
                j0[i][k]  +=   m[j] * r3invdvetc;
                j0[j][k]  += - m[i] * r3invdvetc;
            }
            p[i] -= m[j] * rinv;
            p[j] -= m[i] * rinv;
        }
```

```
    }
}
```

というような感じの、まあ、力の計算が重複しているけど２つくらいで初期化と時間積分で違うからまあいいか、というプログラムだったものに、特殊な場合としてブラックホール１つ、もう一つ、と付け加えていった時に、さらにソフトニングを配列にすればよいかと思って変更したけどよく考えるとそれでは駄目だったのでもうちょっといじって動くようにした、というのが最初のコードである。

問題は、どのようにすれば最初のコードのようなものを書かないですむか、ということだが、個人のレベルとしては、これは、

　　書いてしまったと
気が付いたら、あるいは
指摘されたら
　　直す

というだけである。最初のコードが駄目なものであるのはみればわかるし、どうなっているべきかの判断もそれほど難しいことではない。問題は、あくまでも、実際に直すかどうか、である。直したほうが後で楽だが、「直す」という作業は常に大変なことにように思え、また現在なんとか動いているものに手をつけたくない、とか、「動いているものを変更するな」と偉い人がいっていたとか、そういう、直さないための理由や口実を見つけてくることは簡単だからである。

これは、色々な人がいっているように、意志、あるいは習慣の問題で、本人が努力して身につけるしかない。が、ここでのポイントは、最初のコードのようなものを長年にわたって書き続ける、ということは、それだけの間研究者なりプログラマなりとしての人生をドブに捨てている、ということである。よりましなやり方をとっていれば短い時間ででき、より進歩できるはずなのに、それができない、ということになるからである。

Chapter 7

文献データベースの利用 (2009/6/18)

25 年くらい前には、自分のやっていることに関係する論文を探す、というのは簡単ではなかった。新しいテーマを始めよう、という時には、*Astronomy and Astrophysics Abstracts*[1] という電話帳みたいなもので、関係ありそうな論文を過去何年にも渡って探す、とかするしかなかった。

しかし、ADS と arXiv.org によって、少なくとも物理・天文学の分野ではこの状況は根本的に変わり、あらゆる論文が Web 上で検索でき、また雑誌は有料であっても著者が arXiv.org に登録していれば無料でダウンロードして読めるようになった。

ADS だと論文に、

- その中で引用されている論文
- その論文を引用している論文

のリストがついてくるので、その辺を辿っていくと、1 つの論文から、その分野でどんなことが研究されているかが大体わかる。

ADS で色々な意味で重要なのは、上の

- その論文を引用している論文

のリスト、というより論文の数である。これは、専門用語でいうと citation count ないし citation index (引用度) というもので、極めて大雑把にいうと、その論文の重要性を数値として表しているといえなくもない。もちろん、リニアに対応するわけではないが、非常に重要な論文であり、そのことが多くの研究者から認識されていれば、通常はその論文は多くの人が引用するので引用度は高くなるはずである。

まあ、そう単純ではない、ということを議論した論文は一杯あるが、無関係ではない、ということである。なので、大雑把には、引用度が高い論文は重要なものである可能性が高い。

これを読んでいる読者が大学院生や、あるいは学部生だとしたら、 ADS のようなデータベースを使ってできる極めて重要なことは、指導教員、あるいはその候補の人がどういう研究者かをある程度知ることである。

人の名前をいれて検索すれば、その人がどういう論文を書いたかが概ね全部でる。もちろん、イニシャルが同じ人のがまざっているかもしれない。まず重要なことは、最近数年の間に査読付き雑誌にちゃんと論文を書いているかどうかである。これは、 検索ページ[2]の下のほうにある、FILTERS の

[1] http://www.ari.uni-heidelberg.de/publikationen/aaa/
[2] http://adsabs.harvard.edu/abstract_service.html

ところの、Select References From: で All refereed articles をチェックする。これで過去数年間に論文がない人の場合には、今後数年間にも論文がない可能性が高い。言い換えると、その人のところで修士論文や博士論文のためにやる研究も、論文にならない可能性が高い、ということである。

もちろん、なんらかの事情でたまたま数年間論文がなかったとか、そもそも天文の研究者でないとか、色々な理由で ADS では論文がでてこない、ということもあるので、そうでないかどうかは確認が必要である。

さて、ちゃんと論文を書いている人である、とわかったら、次はその中身である。これは、

- 自分の興味に近いかどうか
- まともな研究かどうか
- さらには、良い研究かどうか

といったことである。自分の興味と近くかどうかは中身をみないとわからない。まともな研究かどうかは、例えば上の citation count で、ある程度の数がでていれば普通に業界の中で評価されている論文であることがわかる。出版から 10 年たっても全く引用がないとかいった論文があってもそれはまあそういうこともあるが、ある人の論文が全てそうだったりするとそれはちょっと問題である可能性もある。

まあ、もちろん、沢山論文を書いていて、それが沢山引用もされていても、重要かどうかはまた違う話なので、だからどうというわけではない。が、全く論文を書いていない、あるいは全く引用されていない、というのは、研究者として存在していないのと同じである。

ある人の全論文について、それらがどれくらい引用されているかを見るには、FILTERS のさらに下にある SORTING のオプションで、 citation count, あるいは normalized citation count をチェックする。後者は、 citation count を著者数で割ったものである。

装置開発だけしていて観測結果とかの論文はない、という人もいるかもしれないが、装置開発として新しいことをしているならそれ自体が論文になっているはずであり、そういう論文もない、というのは何か理由がある。

Chapter 8

再帰と並列化、ベクトル化 (2009/8/30)

再帰というのは、ここを読んでいる人が知らない、ということはないと思いますが、ある関数なりサブルーチンが自分自身を呼び出すことです。例えば、ツリーコードのような、データ構造自体が自分と同じ構造をもつ、という意味で再帰的なものでは、その処理も再帰を使って書くのが自然です。

ツリーコードの基本的なステップは

ツリーのデータ構造を作る
ツリーノードの物理量 (位置、質量等) を計算する
ツリー構築を使って各粒子への力を計算する

です。この中でもっとも単純な、物理量を計算するところを再帰を使って書いた例を以下に示します。

```
void  bhnode::set_cm_quantities()
{
    int i;
    cmpos = 0.0;
    cmmass = 0.0;
    if (isleaf){
        bhparticle * bp = bpfirst;
        for(i = 0; i < nparticle; i++){
            real mchild = (bp+i)->get_rp()->get_mass();
            cmpos += mchild*(bp+i)->get_rp()->get_pos();
            cmmass += mchild;
        }
    }else{
        for(i=0;i<8;i++){
            if (child[i] != NULL){
                child[i]->set_cm_quantities();
                real mchild = child[i]->cmmass;
                cmpos += mchild*child[i]->cmpos;
                cmmass += mchild;
            }
        }
    }
```

```
    cmpos /= cmmass;
}
```

これは C++ を使っていて、3 次元ベクトルをクラスにしているので、重心の計算のために質量掛ける位置を積算するのが

```
            cmpos += mchild*(bp+i)->get_rp()->get_pos();
```

ですむ、といったことがあって単純ですが、基本的な構造が

ノードの物理量の計算:
子供が全部粒子の時
    粒子の位置、質量を計算
子供がノードの時
    各子供について
物理量を計算 <---- ここが再帰
位置、質量を積算
位置と質量の積の積算を質量でわって、重心を求める

というもので、要するに、ノードの重心はその下のノードなり粒子の重心の重心である、という定義を、素直にプログラムに表現しています。これを C なのですが再帰でなくて書いてあるのが以下の例です。

```
void calc_com_tree (prm_struct *pprm, ptcl_struct *pptcl, tree_struct *ptree)
{
        int nnl, nnd, nst, ilev, inl, jnl, ind, ipn, ip;
        int *lhp, *lnp, *lnd, *lhn, *ltn, *lnl, *lst, *lpn, *lfg;
        real invn, x0, y0, z0, x1, y1, z1, ds, l0, lbox, invtheta, rc;
        real *ppos, *com, *rc2, *minpos, lln[MAXTREELEV], lil[MAXTREELEV];
        ppos = pptcl->ppos;
        lbox = ptree->lbox;
        minpos=ptree->minpos;
        lhp = ptree->lhp;
        lnp = ptree->lnp;
        lnd = ptree->lnd;
        lhn = ptree->lhn;
        ltn = ptree->ltn;
        nnl = ptree->nnl;
        nnd = ptree->nnd;
        invtheta = 1.0 / pprm->theta;
        MALLOC (com, real, nnd * NDIM); ptree->com = com;
        for (ind=0; ind<nnd*NDIM; ind++) com[ind] = 0.0;
        MALLOC (rc2, real, nnd); ptree->rc2 = rc2;
        MALLOC (lnl, int,  nnd);
        MALLOC (lst, int, nnl * 3);
        lfg = lst + nnl;
        lpn = lfg + nnl;

        /* push the root node to the stack */
        nst = 0;
```

```
lst[nst] = 0;
lfg[nst] = FALSE;
lpn[nst] = NULLNODE;
nst++;
lnl[0] = 0;

/* main loop */
while (nst){
        /* pop from stack */
        nst--;
        inl = lst[nst];
        ipn = lpn[nst];
        ind = lnd[inl];
        if (lfg[nst]){
                /* ascend tree */
                if (ipn != NULLNODE){
                        com[ipn*NDIM]   += com[ind*NDIM];
                        com[ipn*NDIM+1] += com[ind*NDIM+1];
                        com[ipn*NDIM+2] += com[ind*NDIM+2];
                }
        } else {
                /* descend tree */
                if (ind == NULLNODE){
                        ip = lhp[inl];
                        com[ipn*NDIM]   += ppos[ip*NDIM];
                        com[ipn*NDIM+1] += ppos[ip*NDIM+1];
                        com[ipn*NDIM+2] += ppos[ip*NDIM+2];
                } else {
                        if (ipn != NULLNODE) lnl[ind] = lnl[ipn] + 1;
                        lfg[nst++] = TRUE;
                        for (jnl=ltn[ind]; jnl>=lhn[ind]; jnl--){
                                lst[nst] = jnl;
                                lfg[nst] = FALSE;
                                lpn[nst] = ind;
                                nst++;
                        }
                }
        }
}
FREE_MEM (lst);

/* calculate cell size */
for (ilev=0; ilev<MAXTREELEV; ilev++){
        l0 = lbox / ((real)( 1 << ilev));
        lln[ilev] = l0;
        lil[ilev] = 1.0 / l0;
}

/* calculate center of mass */
for (ind=0; ind<nnd; ind++){
        invn = 1.0 / (real) lnp[ind];
        com[ind*NDIM]   *= invn;
```

```
                com[ind*NDIM+1] *= invn;
                com[ind*NDIM+2] *= invn;
        }

        /* calculate and center of node and critical radius */
        for (ind=0; ind<nnd; ind++){
                /* center of mass */
                x0 = com[ind*NDIM]   - minpos[0];
                y0 = com[ind*NDIM+1] - minpos[1];
                z0 = com[ind*NDIM+2] - minpos[2];

                /* center of node */
                ilev=lnl[ind];
                x1 = lln[ilev]* (0.5+ (real)((int) (x0 * lil[ilev])));
                y1 = lln[ilev]* (0.5+ (real)((int) (y0 * lil[ilev])));
                z1 = lln[ilev]* (0.5+ (real)((int) (z0 * lil[ilev])));

                /* check con is consistent with com of children (see tree.h)  */
                CHECK_COMCON;

                /* distance between center of mass and center of node */
                x1 -= x0;
                y1 -= y0;
                z1 -= z0;

                ds = sqrt(x1 * x1 + y1 * y1 + z1 * z1);
                rc = lln[ilev] * invtheta + ds;
                rc2[ind] = rc * rc;
        }
        FREE_MEM (lnl);
}
```

すみません、何をしているかよくわからないので、解説とか変更の試みは今回省略します。nst という変数があって、これがスタックポインタになっていて、色々なものがスタックに積まれたり取り出されたりしているようですが、私の理解力を超えています。再帰ができない Fortran で、ベクトル化できるように書いたものの例を以下に示します。

```
        subroutine hacCF2

#include "tdefs.F"
        integer top(nbits), last(nbits), nlevel
        integer i, j, k, r, lastnd, level
        REAL dterm, mr, dx1, dx2, dx3
        lastnd=nxtnode+ncell-1
        do 200 i=nxtnode,lastnd
           mass(i)=0.0
           do 210 k=1,ndim
              pos(i,k)=0.0
 210       continue
 200    continue
        top(1)=nxtnode
        last(1)=nxtnode
```

```
      nlevel=1
      do 230 i=nxtnode+1, lastnd
         if(sizetol2(i) .lt. sizetol2(i-1)*0.9) then
            nlevel=nlevel+1
            top(nlevel)=i
            last(nlevel-1)=i-1
         endif
230   continue
      last(nlevel)=lastnd
      do 240 level=nlevel, 1, -1
         do 250 j=1,nsubcell
            do 260 i=top(level), last(level)
               r=subp(i,j)
               if(r .ne. null) then
                  mr=mass(r)
                  mass(i)=mass(i)+mr
                  pos(i,1)=pos(i,1)+pos(r,1)*mr
                  pos(i,2)=pos(i,2)+pos(r,2)*mr
                  pos(i,3)=pos(i,3)+pos(r,3)*mr
               endif
260         continue
250      continue
         do 280 i=top(level), last(level)
            do 270 k=1,ndim
               pos(i,k)=pos(i,k)/mass(i)
270         continue
280      continue
240   continue
     end
```

ここでは、再帰によってツリーを辿るのをスタックを使ってエミュレーションする、といった面倒なことはしないで、ツリーの階層の下から順番に作る、同じレベルのノードの間でベクトル化する、という方法をとっています。ここでは、そもそもノードがレベルの順番に並んでいて、上のレベルのノードは添字が小さい、と仮定しています。このことは、ツリーの作り方によって保証されています。

230 のループでは、レベルの境界を捜しています。これは無能なコードで、$O(N)$ の計算量で全ノードをみていますが、本当は 2 分探索を繰り返すことで $O(\log N^2)$ でできるし、そもそもツリーを作った時に憶えておけば計算する必要もないのですが、まあ、20 年以上前に書いたコードなので、、、

レベルの境界が決まれば、後は下から作るだけで、レベルのループが 240 です。250、260 は奇妙な順番で、 250 で 8 個ある子供についてのループを回し、その内側の 260 で同じレベルにある全ノードを回るループにしていますが、これは大抵のベクトル化 fortran コンパイラが最内側ループしかベクトル化しないので自然な順番と入れ替えているものです。

再帰はアルゴリズムを簡潔で、従って間違いが少なくデバッグも容易な形に表現する優れた方法ですが、並列化やベクトル化とは必ずしも適合しないこともあります。特に、コンパイラの能力として最内側ループだけしか並列化 (ベクトル化) しないベクトル化コンパイラの場合には、仮に再帰が扱える言語やコンパイラであったとし、また再帰の中に自然な並列性が表現されていたとしても、再帰の形に書いたものがハードウェアで効率良く走る可能性は殆どありません。

例えば、上の再帰の例では、8 個の子供について重心を計算するのは完全に並列に行えますし、さらにその 8 個の子供の 64 個くらいの子供についても並列に行えるわけです。従って、MIMD で共有メモリ計算機用の良くできた並列化コンパイラなら勝手に並列化してくれてもよいし、実際にデー

タフローマシン用のコンパイラなら自然に並列化されます。

なお、例えば OpenMP でも、task や taskwait プラグマを使うことで並列実行できるようです。但し、なにも考えずにこの辺を使うと、ノード数だけのタスクが生成されてしまうので実行効率は落ちるかもしれません。途中から並列化しない再帰ルーチンを呼ぶといった工夫が必要になる可能性もあります。

ベクトル化と例えば OpenMP を使った並列化は、「どのように並列化されるべきか」という高レベルの概念は同じであっても、「どうやってコーディングするべきか」という観点では全く違ってきてしまうことが普通です。例えば、上のベクトル化できる fortran ルーチンは、もちろん最内側ループを omp parallel で並列化できますが、あまり効率は良くありません。これは、このループが、ツリーのレベル数× 8 回起動されるもので回数が多いので、起動のオーバーヘッドが大きくなるからです。もちろん、この場合には 250 と 260 のループを入れ替えるだけで回数が 1/8 になるのでだいぶましで、十分に粒子数が多い時にはその程度でよいかもしれません。

Chapter 9

HPL 書き直し (2009/9/2 書きかけ)

別のところに書いたような事情で、 HPL を書き直してます。とりあえず MPI とか並列化は後回しで、どういうメモリレイアウトやチューニングをするか、を 1 ノードの短いプログラムで調べています。

普通に Top500 で使われている HPL は、行列のデータ格納の順序が Column Major (列優先) というもので、 a_{ij} の i が変わるものが連続アドレスに入ります。これは、ピボット探索ではメモリフットプリントが最小になるし、また昔からある普通の縦ブロックガウスでキャッシュ(昔ならディスクでない主記憶自体)) を有効に使うという話であれば、これを逆転させて Row Major (行優先) にする、というのは全くありえない話です。

が、行列乗算にアクセラレータを使い、また そのためのブロック化に Gustavson の再帰的ブロッキングとかを使うと話が変わってきます。まず、行列乗算の効率を上げるために right-looking にするので、ここに関しては縦長の行列と横長の行の積となり、行優先でも列優先でも同じです。再帰的パネル分解では縦長の行列と小さな正方行列の積 (結果は縦長) となるので、これは列優先のほうが若干得ですが、行列サイズが極端に小さい (幅がキャッシュラインサイズ以下になる) 場合でなければ大きな性能差はでないでしょう。

right-looking の場合には、右側に残っている部分正方行列全体に対して、部分ピボットの結果でてくる行交換を行う必要があります。この部分は列優先では非常に不利で、どうやってもキャッシュラインの殆どが無駄になります。

さて、では、ピボット探索と行交換の両方を高速にするのは不可能か、というと、縦方向に再帰分解をしているわけなので、適当なサイズのところで転置すればよいはずです。転置のための余計なメモリアクセスが発生するわけですが、

- 転置後の列優先行列はキャッシュに入る
- 転置前に行列の幅はキャッシュラインサイズ以上

というのを同時に満足できれば、メインメモリからは行列半分を読み出して半分を格納、となります。また、転置はその領域を更新する行列乗算の直後に起こるので、行列乗算ルーチンが結果をキャッシュに残していれば (これは、moventpd を使わない、ということでもありますが)、高い性能が期待できます。ということで、まだ GRAPE-DR を使っていないしシングルコアのコードですが、転置を行うものを書いてみました。これは、8 列になったところで転置します。また、最大のブロックサイズは 64 です。N=2K で、 Athlon 64 X2 4050e でのプロファイルの結果が以下です。実際の実行時間は 1.8 秒だったので、下の時間速度はあまり正確ではありませんが、相対値には意味があるかな？という程度です。

```
Each sample counts as 0.01 seconds.
```

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
86.41	1.27	1.27				dgemm_kernel
2.72	1.31	0.04	256	0.16	0.16	transpose_coltorow
2.04	1.34	0.03				dgemm_otcopy
1.36	1.36	0.02	8192	0.00	0.00	swaprows
1.36	1.38	0.02	1	20.00	20.00	copymats
0.68	1.39	0.01	4096	0.00	0.00	vvmulandsub
0.68	1.40	0.01	1024	0.01	0.01	column_decomposition
0.68	1.41	0.01	256	0.04	0.04	solve_triangle
0.68	1.42	0.01	256	0.04	0.04	transpose_rowtocol
0.68	1.43	0.01	193	0.05	0.05	matmul_for_small_nk
0.68	1.44	0.01	1	10.00	10.00	backward_sub
0.68	1.45	0.01				dgemm_beta
0.68	1.46	0.01				dgemm_nn
0.68	1.47	0.01				dgemm_oncopy

copymats は初期の行列をコピーする関数で、ソルバの時間にははいりません。dgemm 以外で無視できない時間を占めるのは

transpose_coltorow	列優先から行優先に戻す
swaprows	行交換
vvmulandsub	ベクトル同士の外積
column_decomposition	再帰の最後のパネル分解
solve_triangle	3 角行列の求解
transpose_rowtocol	行優先から列優先に転置
matmul_for_small_nk	行優先の状態での縦長な行列と小さな行列の乗算

といった辺りです。参考のために、 single core での STREAM の実行結果は

Function	Rate (MB/s)	RMS time	Min time	Max time
Copy:	4096.5704	0.0500	0.0500	0.0504
Scale:	2506.7073	0.0818	0.0817	0.0829
Add:	3410.3011	0.0901	0.0901	0.0901
Triad:	3247.1131	0.0946	0.0946	0.0947

というところです。これは、moventpd イントリンシックを使ったもので、Triad では

```
        for (j=0; j<N/2; j++)
            //      a[j] = b[j]+scalar*c[j];
            __builtin_ia32_movntpd((double*)&av[j], bv[j]+ss*cv[j]);
```

というようなコードです。Scale が極端に遅いですが、まあ 3-4GB/s がメモリの実力です。

まず、一番上の行列転置 transpose_coltorow ですが、行列全体は 32MB で、そのうち半分が転置に関係するので 16MB、これを 0.04 sec でなんとかしているので速度は 400MB/s で、話にならないほど遅いので明らかに工夫が必要です。swaprows は交換のためにには 2 回読んで書く必要があり、それでも半分の時間でおわっているのでまあよいような気がします。

上のテストで転置に使ったのは

```
void transpose_coltorow(int n, double a[][RDIM],int m, double at[][n],
int istart)
{
    int i,j;
    for(i=istart;i<n;i++)
for(j=0;j<m;j++)
    a[i][j] = at[j][i];
}
```

という全くなんの芸もないコードです。こんなのでは駄目に決まっているので、色々改良してみます。まずはブロック化です。一旦、 m x m の正方行列に転置しながらコピーして、それを連続アクセスでコピー、としてみます。

```
void transpose_coltorow(int n, double a[][RDIM],int m, double at[][n],
int istart)
{
    int i,j,k;
    double atmp[m][m];
    for(i=istart;i<n;i+=m){
for(j=0;j<m;j++){
    for(k=0;k<m;k++){
atmp[k][j]  =at[j][i+k];
    }
}
for(k=0;k<m;k++){
    for(j=0;j<m;j++){
a[i+k][j] = atmp[k][j];
    }
}
    }
}
```

これの結果は

```
 0.65      1.52     0.01      256     0.04     0.04  transpose_coltorow
```

というところで、測定精度があまりありませんが遅くなってはいません。後半のコピーで、 moventpd を使ってみると

```
            v2df * atp = (v2df*) atmp[k];
    for(j=0;j<m;j+=2)
__builtin_ia32_movntpd(a[i+k]+j, atp[j/2]);
```

という感じです。これは速いか、というと、

```
 0.00      1.51     0.00      256     0.00     0.00  transpose_coltorow
```

となって確かに速いのですが、全体の実行時間は必ずしも短くはなりません。これは、moventpd では代入した結果がキャッシュに残らないのですが、実際には代入した結果はそのあとすぐに使うの

で、どうせまたキャッシュにはいってくるからです。これは行列優先から列優先への転置でも同じで、movntpd を使うとこの関数自体は速くなったように見えるのですが、プログラム全体としてはあまり変化しません。 rdtsc 命令を使って実行時間を直接測定して見ると、実は movntpd 命令を使わないほうが 30% 程度短く、転送速度 (上り下り合計) では 2.7GB/s 程度でていました。 (0.09words/clk 程度)

部分的にはキャッシュ内の移動ですんでいるはず、と思うとまだ遅いので、内側のループを 4 重にアンロールしてみます。

```
void transpose_coltorow(int n, double a[][RDIM],int m, double at[][n],
int istart)
{
    int i,j,k;
    double atmp[m][m];
    for(i=istart;i<n;i+=m){
for(j=0;j<m;j++){
    double * atj = at[j]+i;
    for(k=0;k<m;k+=4){
atmp[k][j]  =atj[k];
atmp[k+1][j]  =atj[k+1];
atmp[k+2][j]  =atj[k+2];
atmp[k+3][j]  =atj[k+3];
    }
}
for(k=0;k<m;k++){
    double * aik = a[i+k];
    for(j=0;j<m;j+=4){
aik[j] = atmp[k][j];
aik[j+1] = atmp[k][j+1];
aik[j+2] = atmp[k][j+2];
aik[j+3] = atmp[k][j+3];
    }
}
    }
}
```

これで 0.115words/clk 程度まで。アンロール、と考えてみると、この関数は呼ぶ時の m の値はいつも同じなので、最内側は展開するべきでしょう。そうすると、

```
void transpose_coltorow8(int n, double a[][RDIM], double at[][n],
int istart)
{
    int i,j,k;
    const int m=8;
    double atmp[m][m];
    for(i=istart;i<n;i+=m){
for(j=0;j<m;j++){
    double * atj = at[j]+i;
    atmp[0][j]  =atj[0];
    atmp[1][j]  =atj[1];
    atmp[2][j]  =atj[2];
```

```
    atmp[3][j]  =atj[3];
    atmp[4][j]  =atj[4];
    atmp[5][j]  =atj[5];
    atmp[6][j]  =atj[6];
    atmp[7][j]  =atj[7];
}
for(k=0;k<m;k++){
    v2df * atp = (v2df*) atmp[k];
    v2df * ap = (v2df*) a[i+k];
    *(ap)=*(atp);
    *(ap+1)=*(atp+1);
    *(ap+2)=*(atp+2);
    *(ap+3)=*(atp+3);
}
    }
}
```

とより簡潔で高速なコードになり、0.13word/clk、4.4GB/s となってまあよいかも、というところです。キャッシュにはいっているはずの割には遅いのが気になります。

次は行交換です。

```
void swaprows(int n, double a[n][RDIM], int row1, int row2,
      int cstart, int cend)
{
    /* WARNING: works only for row1 % 4 = 0 and RDIM >= n+4*/

    int j;
    if (row1 != row2){
int jmax = (cend+1-cstart)/2;
v2df *a1, *a2, tmp, tmp1;
a1 = (v2df*)(a[row1]+cstart);
a2 = (v2df*)(a[row2]+cstart);
for(j=0;j<jmax;j+=2){
    tmp = a1[j];
    tmp1 = a1[j+1];
    a1[j]=a2[j];
    a1[j+1]=a2[j+1];
    a2[j]=tmp;
    a2[j+1]=tmp1;
    __builtin_prefetch(a1+j+16,1,0);
    __builtin_prefetch(a2+j+16,1,0);
}
    }
}
```

2 重にアンロールし、 SSE を使い、プリフェッチもいれています。プリフェッチをいれると、大体 0.07words/clock、2 語の上り下りの合計としては 4.7GB/s となって、まあよいところです。プリフェッチなしだと 0.55words/clock 程度ですが、それなりにプリフェッチの効果はあります。

swaprows の後で正規化をする scalerow もみておきます。

```
void scalerow( int n, double a[n][RDIM], double scale,
       int row, int cstart, int cend)
{
    int j;
    for(j=cstart;j<cend;j++) a[row][j]*= scale;
}
```

こんなのですね。

いじった後は

```
void scalerow( int n, double a[n][RDIM], double scale,
       int row, int cstart, int cend)
{
    int j;
    int jmax = (cend+1-cstart)/2;
    v2df *a1 = (v2df*)(a[row]+cstart);
    v2df ss = (v2df){scale,scale};
    for(j=0;j<jmax;j+=2){
__builtin_prefetch(a1+j+16,1,0);
a1[j] *= ss;
a1[j+1]*= ss;
    }
}
```

こんな感じで、 0.16words/cycle、5.4GB/s です。 これは、いじる前は 0.08words/cycle 程度だったので、SSE2 とプリフェッチの利用に非常に大きな効果があります。

但し、scale と swap を一体化すれば余計な主記憶アクセスを減らせるので、そっちも検討するべきですね。

GRAPE-DR のようなアクセラレータを使うシステムでは、Linpack Benchmark では行列乗算を速くするだけでは十分ではなく、主記憶全体をなめることになる、つまり $O(n^2)$ のあらゆる操作について、主記憶の理論バンド幅に可能な限り近い速度がでる形にコードを書き直す必要があります。

ここまでで、行列乗算や 3 角行列の求解、つまり BLAS3 の用語でいうと DGEMM と DTRSM 以外についてはかなり高速なコードになっています。 DTRSM については DGEMM に変換し、変換できないところも再帰的に DGEMM を取り出すアクセラレータ専用のアルゴリズムを使うとして、DGEMM 自体はどうか、というのが以下の表です。シングルコアの K8 アーキテクチャなので 2 演算/クロックが最大で、 K=64 だと十分な性能になっていますが K=8 ではかなり低いことがわかります。

M	N	K	ops/clock
2040	8	8	0.42988
2032	16	16	0.815346
2024	8	8	0.435265
2016	32	32	1.20451
2008	8	8	0.428314
2000	16	16	0.822711
1992	8	8	0.434964
1984	1985	64	1.70924

メモリバンド幅としてはどういうことになるかというと、 N=K=8 では 2 語読み出して 1 語格納するのに対して 16 演算ですから、 0.08 語、1.35GB/s となって、同じようなアクセスである STREAM

ADD の 4 割程度しかでていません。これはかなり大きな問題です。というのは、再帰的ブロッキングでは、各ブロックサイズでメモリアクセスは基本的には行列全体 (細かいことをいうと 1/4) になり、階層のレベル数、つまり最大ブロックサイズの対数回全体をなめるからです。もちろん、ここで K が大きいところではアクセラレータを使うので、そっちのチューニングが問題ですが、 K が小さいところで主記憶のバンド幅がでていない、ということは、そこに大きなオーバーヘッドがある、ということになります。

今回 GotoBLAS 1.26 を使っていますが、中身をいじるのはやりたくないのでこのような小さい行列の時だけ書き下してみます。今までの例の中で同じようなアクセスパターンなのは行列転置ですが、この場合には単純に、転置しながら正方行列に書いて、それを連続アクセスでコピー、というのでよい性能がでました。従って、同じようなアプローチを考えてみます。

```
void matmul_for_nk8(int n1, double a[][n1],
 int n2, double b[][n2],
 int n3, double c[][n3],
 int m,
 int n)
{
    int i,j,k;
    const int kk = 8;
    const int kh = kk/2;
    int nh = n/2;
    v2df bcopy[nh][kk];
    v2df acopy2[kk][kk];
    for(k=0;k<kk;k++)
for(j=0;j<nh;j++)
bcopy[j][k] = *((v2df*)(b[k]+j+j));
    for(i=0;i<m;i+=kk){
for(k=0;k<kk;k++){
    __builtin_prefetch(a+i+k+16,0,0);
    __builtin_prefetch(c+i+k+16,1,0);
}
for (j=0;j<n;j++){
    double * ap = (double*)( a[i+j]);
    for (k=0;k<kk;k++){
acopy2[j][k]=(v2df){ap[k],ap[k]};
    }
}
for(k=0;k<kk;k++){
    v2df * cp = (v2df*) (c[i+k]);
    v2df * ap = acopy2[k];
    for(j=0;j<nh;j++){
v2df * bp = bcopy[j];
cp[j] -= ap[0]*bp[0] + ap[1]*bp[1]
    + ap[2]*bp[2]    + ap[3]*bp[3]
    + ap[4]*bp[4]    + ap[5]*bp[5]
    + ap[6]*bp[6]    + ap[7]*bp[7];
    }
}
    }
}
```

これ、 N=K=8 でないと動かない代物ですが、

```
M     N   K           ops/clock
2040    8   8         0.727034
2008    8   8         0.730609
1992    8   8         0.782853
```

で、元々の BLAS に比べると 1.7 倍ほど速くなり、メモリバンド幅の 7 割程度は有効利用できていることがわかります。このコードでは、 b は転置しながらローカルにコピーします。a は、 SSE2 の 2 要素ベクトルに同じものがはいる形でふくらませたローカルコピーを作ります。これは、 SSE2 命令が無能でスカラとベクタの掛け算ができないことに対応するためです。 a は大きな行列で、b は小さいので、b を膨らませて a はそのまま、ですめば無駄が少ないのですが、行列の格納のしかたがそれはできないような順番になっています。

で、 a, c についてはプリフェッチをかけています。今回高速になっているのは主にこの効果ですね。実際の演算では、a の 2 要素 (同じものがはいっている) と b の 2 要素をいっぺんに積算して、 c の 2 要素を求める、という形になっています。

a を膨らまさないで、SSE の 2 要素に同じ行のデータが入るようにして DOTD 命令で加算するほうが無駄が少なくて速いような気がしますが、 gcc 4.1.2 のイントリンシックには dotd はなさそうなので今回はそっちは使っていません。

ここは、さらにいじましい高速化を考えると、どうせ次に転置するんだから転置してから乗算するべきです。そうすると、もっと容易に高速化ができます。

転置したあとの行列乗算も、幅が狭いだけに性能が重要です。この場合、基本的にキャッシュに載っているはずなので、命令スケジューリングによるはずです。 GotoBLAS の性能は、 n=k=2、 4 で 0.4、 0.7 ops/clock となっていて、キャッシュだと思うとどうにも遅い、という感じです。メモリバンド幅としては 3 語で 4、 8 演算ですから、5GB/s 程度です。L2 でももっとでるはずなので、工夫の余地があります。

N=2048 の行列で、計算時間が大体 1.75 秒ですが、そのうち n=k=2 の行列の計算時間が 5ms 程度です。これは小さいように見えるのですが、大きな行列乗算が相対的に 100 倍程度速くなると、行列サイズを 10 倍程度大きくしても無視できないわけです。

というわけで、ここまでで、

- 転置状態での細い行列乗算
- 3 角行列の処理
- swap/scale の一体化

以外は大体済んだ、ということになります。

Ci7 の機械を組んだので、そっちで実験 (A64 と比較) をしています。

```
Athlon, N=2K
swaprows time=3.2801e+07
scalerow time=1.42839e+07
trans rtoc8 time=3.25526e+07
trans ctor8 time=1.63445e+07
Ci7 920 (3GHz), N=2K
swaprows time=9.78e+06
scalerow time=3.29607e+06
trans rtoc8 time=1.42135e+07
```

```
trans ctor8 time=6.05478e+06
Ci7 920 (3GHz), N=8K
swaprows time=1.79584e+08
scalerow time=5.39146e+07
trans rtoc8 time=2.53306e+08
trans ctor8 time=1.1879e+08
trans dgemm time=2.07928e+08
trans nonrec cdec time=1.77051e+08
```

メモリアクセスの総量はそれぞれ 64MB, 32 MB, 32 MB, 32MB なので、速度は

```
            A64 2.1GHz   Ci7 3GHz  Ci7 (N=8K)

swaprows      4.1           19.6    18.1
scalerow      4.7           29.1    37.9
trans rtoc8   2.07          6.7     8.1
trans ctor8   4.1           15.9    17.2
```

です。32K の場合、このへんで 2 秒くらいはまだかかる計算になります。主記憶リミットであるので並列化には殆ど意味がないかな、という気もしますが、まあ、 OMP でやってみると:

```
N=8K, Ci7
trans rtoc8 time=1.74734e+08
N=4k
trans rtoc8 time=4.94381e+07
N=2k
trans rtoc8 time=1.77795e+07
```

意外に効果があります。N=2k では若干遅くなりますが、 8K では 1.5 倍近く速くなります。スレッドで書いたらもうちょっといいかもしれません。こういうものの並列化の場合、起動オーバーヘッドとメモリバンド幅の極限性能、キャッシュ効率の 3 つの要因があるので性能予測が困難ですが、今キャッシュのことは無視すると、N が無限に大きい時の性能は 8k の時から 10-15% 程度の向上で、11GB/s 程度と推測できます。 col2row については、

```
N=8K trans ctor8 time=8.79587e+07
N=4k trans ctor8 time=3.00645e+07
```

で、これもそれなりに効果があることがわかります。特に N がもっと大きいところを考えるので Open MP の起動オーバーヘッドは十分小さくなり、結構よくなることが期待できます。まあ、この場合、OpenMP の並列化ループは 1000 回くらいしか呼ばれていなくて、実行が 10 マイクロ秒程度ですから OpenMP の起動オーバーヘッドはその程度あるわけです。ベクトル化に比べるとはるかにオーバーヘッドが大きく、利用が難しいもの、ということがわかります。

Ci7 (というか、Core 2 以降？) は、デフォルトでは勝手にに L2 に 2 ライン (128 バイト) もってくる、というオプションがあるんだそうで、これは確かに Bios で有効になってたので、 1 ライン単位での転置が遅いのはこれが原因と思われます。

とはいえ、このオプションを BIOS で切ると

```
Nswap=0 cpsec =  46.3909 wsec=14.1913
swaprows time=1.80358e+08
scalerow time=5.39611e+07
```

```
trans rtoc8 time=1.93491e+08
trans ctor8 time=1.01146e+08
trans dgemm time=2.42105e+08
trans nonrec cdec time=1.65904e+08
trans vvmul time=5.65689e+07
trans findp time=1.06555e+08

Nswap=0 cpsec =  46.4709 wsec=13.9176
swaprows time=1.81601e+08
scalerow time=5.47325e+07
trans rtoc8 time=1.86051e+08
trans ctor8 time=8.86901e+07
trans dgemm time=2.48892e+08
trans nonrec cdec time=1.66319e+08
trans vvmul time=5.68656e+07
trans findp time=1.06808e+08
```

ON にすると

```
Nswap=0 cpsec =  46.4349 wsec=46.4116
swaprows time=1.79092e+08
scalerow time=5.39498e+07
trans rtoc8 time=1.90267e+08
trans ctor8 time=1.01477e+08
trans dgemm time=2.3238e+08
trans nonrec cdec time=1.66845e+08
trans vvmul time=5.69506e+07
trans findp time=1.07196e+08
```

Nswap=0 cpsec = 46.7269 wsec=13.9467 swaprows time=1.80993e+08 scalerow time=5.44132e+07 trans rtoc8 time=1.88233e+08 trans ctor8 time=9.26843e+07 trans dgemm time=2.40648e+08 trans nonrec cdec time=1.65907e+08 trans vvmul time=5.64983e+07 trans findp time=1.06597e+08

全然変わらない。これは、プリフェッチが十分効いてない、ということだと思われ、、、というか、コードをちゃんとみたら、あ、プリフェッチの入れかたが間違ってました、、、まあ、でも、直しても普通は変わらないと、、、Core i7 のメモリレイテンシは 200 サイクル程度らしいので、20GB/s = 7B/cycle とすれば大雑把には 200 語程度先読みをしていないといけないことになります。。

以下は細かい作業記録なのであんまり読んでも役に立たないです。

```
Nswap=0 cpsec =  46.4469 wsec=14.2066
swaprows time=1.82082e+08
scalerow time=5.46414e+07
trans rtoc8 time=1.89987e+08
trans ctor8 time=9.16793e+07
trans dgemm time=2.45399e+08
trans nonrec cdec time=1.68357e+08
trans vvmul time=5.64953e+07
trans findp time=1.08689e+08
solve tri u time=9.15278e+07
solve tri   time=1.05368e+09
```

```
matmul nk8  time=2.3478e+08
matmul snk  time=3.00549e+08
```

nk8 は、256 MB アクセスのはず。snk も同じ。

16 で転置すると

```
Nswap=0 cpsec =  46.5109 wsec=13.6755
swaprows time=1.80648e+08
scalerow time=5.29178e+07
trans rtoc8 time=0
trans ctor8 time=9.17586e+07
trans dgemm time=3.54391e+08
trans nonrec cdec time=1.66416e+08
trans vvmul time=5.67401e+07
trans findp time=1.07049e+08
solve tri u time=9.06005e+07
solve tri   time=9.15348e+08
matmul nk8  time=0
matmul snk  time=2.97462e+08
```

転置のメモリアクセス: 512MB。これを 0.028 秒だと、16GB/s で結構速い。

matmul snk は 256MB を 0.1 秒で遅い。演算速度は 8k x 8k /4 * 32 =0.54e9 くらいなので、5Gflops

```
OMP_NUM_THREADS=2, snk を OMP で。

Emax=  4.007e-10
Nswap=0 cpsec =  46.0989 wsec=23.3147
swaprows time=1.80596e+08
scalerow time=5.2993e+07
trans rtoc8 time=0
trans ctor8 time=8.32039e+07
trans dgemm time=1.84467e+19
trans nonrec cdec time=1.66061e+08
trans vvmul time=5.74618e+07
trans findp time=1.06753e+08
solve tri u time=9.12501e+07
solve tri   time=1.06944e+09
matmul nk8  time=0
matmul snk  time=1.55114e+08
```

なんだか素晴らしい効果がある。

m=8 で転置に戻すと

```
Emax=  8.745e-10
Nswap=0 cpsec =  46.1349 wsec=23.2356
swaprows time=1.79216e+08
scalerow time=5.33178e+07
trans rtoc8 time=1.6885e+08
```

```
trans ctor8 time=8.19126e+07
trans dgemm time=1.95553e+08
trans nonrec cdec time=1.66091e+08
trans vvmul time=5.65165e+07
trans findp time=1.06811e+08
solve tri u time=1.84467e+19
solve tri   time=1.07682e+09
matmul nk8  time=1.27066e+08
matmul snk  time=1.52946e+08
```

Nthreads=1, N=16384

```
Emax=  1.076e-06
Nswap=0 cpsec =  417.914 wsec=160.704
swaprows time=7.40931e+08 ops/cycle=0.181147
scalerow time=3.47836e+08 ops/cycle=0.385864
trans rtoc8 time=1.84467e+19 ops/cycle=7.27596e-12
trans ctor8 time=1.84467e+19 ops/cycle=7.27596e-12
trans mmul  time=1.35821e+09 ops/cycle=1.48229
trans nonrec cdec time=1.84467e+19 ops/cycle=7.27596e-12
trans vvmul time=2.23035e+08 ops/cycle=0.60178
trans findp time=1.84467e+19 ops/cycle=7.27596e-12
solve tri u time=1.81516e+08 ops/cycle=9.02621e-05
solve tri   time=1.84467e+19 ops/cycle=8.88178e-16
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=1.19488e+09 ops/cycle=1.79724
trans mmul8 time=7.07553e+08 ops/cycle=1.51754
trans mmul4 time=4.08479e+08 ops/cycle=1.31432

Nt=4
Nswap=0 cpsec =  420.382 wsec=118.056
swaprows time=1.84467e+19 ops/cycle=7.27596e-12
scalerow time=3.48263e+08 ops/cycle=0.385392
trans rtoc8 time=6.80027e+08 ops/cycle=0.197371
trans ctor8 time=3.0702e+08 ops/cycle=0.437162
trans mmul  time=1.37626e+09 ops/cycle=1.46285
trans nonrec cdec time=6.41575e+08 ops/cycle=0.2092
trans vvmul time=2.21756e+08 ops/cycle=0.605251
trans findp time=4.15484e+08 ops/cycle=0.323039
solve tri u time=1.81714e+08 ops/cycle=9.01635e-05
solve tri   time=5.17148e+09 ops/cycle=3.16814e-06
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=6.14184e+08 ops/cycle=3.49649
trans mmul8 time=7.25474e+08 ops/cycle=1.48006
trans mmul4 time=4.0895e+08 ops/cycle=1.3128
```

Nt=2, N=16384, NB=2048 相当いじったあと

```
Emax=  1.471e-07
```

```
Nswap=0 cpsec =  440.624 wsec=222.468
swaprows time=8.39751e+08 ops/cycle=0.15983
scalerow time=4.17531e+08 ops/cycle=0.321456
trans rtoc8 time=1.02016e+09 ops/cycle=0.131565
trans ctor8 time=5.5054e+08 ops/cycle=0.243793
trans mmul  time=1.35954e+09 ops/cycle=1.48085
trans nonrec cdec time=6.41812e+08 ops/cycle=0.209123
trans vvmul time=2.22362e+08 ops/cycle=0.603601
trans findp time=4.15344e+08 ops/cycle=0.323148
solve tri u time=1.41841e+10 ops/cycle=1.1551e-06
solve tri   time=8.92199e+10 ops/cycle=1.83636e-07
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=1.19747e+09 ops/cycle=1.79335
trans mmul8 time=7.08126e+08 ops/cycle=1.51631
trans mmul4 time=4.09517e+08 ops/cycle=1.31099
DGEMM       time=4.85794e+11 ops/cycle=6.03555
```

再帰的な DTRSM は実装した。

GotoBLAS 2 1.01 に変更

```
Enter n, seed, nb:N=20480 Seed=1 NB=512
read/set mat end
copy mat end
Emax=  3.087e-07
Nswap=0 cpsec =  585.725 wsec=146.726
swaprows time=1.1845e+09 ops/cycle=0.17705
scalerow time=6.05163e+08 ops/cycle=0.346543
trans rtoc8 time=1.3443e+09 ops/cycle=0.156003
trans ctor8 time=8.0741e+08 ops/cycle=0.259738
trans mmul  time=2.41859e+09 ops/cycle=1.30065
trans nonrec cdec time=1.02126e+09 ops/cycle=0.205349
trans vvmul time=3.63753e+08 ops/cycle=0.576532
trans findp time=6.51132e+08 ops/cycle=0.322078
solve tri u time=7.77699e+08 ops/cycle=2.63341e-05
solve tri   time=1.6045e+10 ops/cycle=1.27641e-06
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=1.94126e+09 ops/cycle=1.72849
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=6.3223e+08 ops/cycle=1.32683
DGEMM       time=3.67084e+11 ops/cycle=15.6003
```

Enter n, seed, nb:N=16384 Seed=1 NB=2048

```
read/set mat end
copy mat end
Emax=  3.090e-08
Nswap=0 cpsec =  329.265 wsec=82.7602
swaprows time=8.37005e+08 ops/cycle=0.160355
scalerow time=4.17547e+08 ops/cycle=0.321443
```

```
trans rtoc8 time=7.71399e+08 ops/cycle=0.173993
trans ctor8 time=5.27207e+08 ops/cycle=0.254582
trans mmul  time=1.59757e+09 ops/cycle=1.2602
trans nonrec cdec time=6.5203e+08 ops/cycle=0.205846
trans vvmul time=2.29239e+08 ops/cycle=0.585492
trans findp time=4.16039e+08 ops/cycle=0.322609
solve tri u time=5.49025e+09 ops/cycle=2.9842e-06
solve tri   time=3.42907e+10 ops/cycle=4.77797e-07
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=1.23869e+09 ops/cycle=1.73368
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=4.10097e+08 ops/cycle=1.30913
DGEMM       time=1.76038e+11 ops/cycle=16.6556
```

swaprows and scalerow OMP 化

```
<cfcafs03:/home/makino/src/linsol>
Enter n, seed, nb:N=16384 Seed=1 NB=128
read/set mat end
copy mat end
Emax=  4.767e-09
Nswap=0 cpsec =  306.351 wsec=76.7172
swaprows time=8.07664e+08 ops/cycle=0.16618
scalerow time=3.58689e+08 ops/cycle=0.37419
trans rtoc8 time=7.92825e+08 ops/cycle=0.16929
trans ctor8 time=5.24754e+08 ops/cycle=0.255773
trans mmul  time=1.59751e+09 ops/cycle=1.26025
trans nonrec cdec time=6.50838e+08 ops/cycle=0.206223
trans vvmul time=2.30045e+08 ops/cycle=0.583441
trans findp time=4.15841e+08 ops/cycle=0.322762
solve tri u time=1.21474e+08 ops/cycle=0.000134877
solve tri   time=3.71787e+09 ops/cycle=4.40683e-06
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=1.25042e+09 ops/cycle=1.71741
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=4.09244e+08 ops/cycle=1.31186
DGEMM       time=1.95923e+11 ops/cycle=14.9652
```

Core i7 に最適化された GotoBLAS は結構速いので、色々普通にやってあまり問題ない。

```
Enter n, seed, nb:N=20480 Seed=1 NB=2048
read/set mat end
copy mat end
Emax=  6.227e-08
Nswap=0 cpsec =  623.027 wsec=156.475
swaprows time=1.34135e+09 ops/cycle=0.156346
scalerow time=6.46645e+08 ops/cycle=0.324313
trans rtoc8 time=1.26259e+09 ops/cycle=0.1661
trans ctor8 time=7.75399e+08 ops/cycle=0.270461
```

```
trans mmul  time=2.43097e+09 ops/cycle=1.29402
trans nonrec cdec time=9.99599e+08 ops/cycle=0.209799
trans vvmul time=3.49642e+08 ops/cycle=0.5998
trans findp time=6.44818e+08 ops/cycle=0.325232
solve tri u time=6.86921e+09 ops/cycle=2.98142e-06
solve tri   time=5.42023e+10 ops/cycle=3.77844e-07
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=2.04036e+09 ops/cycle=1.64453
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=6.36945e+08 ops/cycle=1.31701
DGEMM       time=3.48804e+11 ops/cycle=16.4179
```

20k の時:

```
swap 0.5sec
scale 0.24sec
trans (r->c) 0.47sec
trans (c->r) 0.29sec
large DGEMM  131.2 sec
```

DGEMM の時間の測定のしかたが間違っていたので修正。 cblas_dgemm を全部合計にする。

```
Enter n, seed, nb:N=16384 Seed=1 NB=256
read/set mat end
copy mat end
Emax=  3.875e-09
Nswap=0 cpsec =  300.335 wsec=75.2046
swaprows time=8.10285e+08 ops/cycle=0.165643
scalerow time=3.71966e+08 ops/cycle=0.360833
trans rtoc8 time=7.6822e+08 ops/cycle=0.174713
trans ctor8 time=5.02146e+08 ops/cycle=0.267288
trans mmul  time=1.56957e+09 ops/cycle=1.28269
trans nonrec cdec time=6.43971e+08 ops/cycle=0.208422
trans vvmul time=2.26705e+08 ops/cycle=0.592036
trans findp time=4.13498e+08 ops/cycle=0.324591
solve tri u time=2.58505e+08 ops/cycle=6.33798e-05
solve tri   time=5.89804e+09 ops/cycle=2.77787e-06
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=1.24804e+09 ops/cycle=1.72069
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=4.10276e+08 ops/cycle=1.30856
DGEMM       time=1.89628e+11 ops/cycle=15.462
```

DGEMM の分が 71.2 秒なのでそれ以外が 4 秒

```
swap 0.3sec
scale 0.15sec
trans (r->c) 0.3sec
trans (c->r) 0.2sec
mmul in trans 0.6sec
```

```
nb=2          0.3sec
small mmul r  0.5sec


Enter n, seed, nb:N=20480 Seed=1 NB=256
read/set mat end
copy mat end
Emax=  1.686e-07
Nswap=0 cpsec =  582.388 wsec=145.805
swaprows time=1.25323e+09 ops/cycle=0.167339
scalerow time=5.8671e+08 ops/cycle=0.357443
trans rtoc8 time=1.2708e+09 ops/cycle=0.165026
trans ctor8 time=7.75856e+08 ops/cycle=0.270302
trans mmul  time=2.42934e+09 ops/cycle=1.29489
trans nonrec cdec time=1.00069e+09 ops/cycle=0.20957
trans vvmul time=3.49502e+08 ops/cycle=0.60004
trans findp time=6.44612e+08 ops/cycle=0.325335
solve tri u time=3.28611e+08 ops/cycle=6.23229e-05
solve tri   time=9.28552e+09 ops/cycle=2.20559e-06
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=2.02897e+09 ops/cycle=1.65377
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=6.36088e+08 ops/cycle=1.31878
DGEMM       time=3.71893e+11 ops/cycle=15.3986
```

TIMETEST を外しても合計時間は殆ど (0.1 秒も) 変わらない。

まだ間違っていたのでもう一度修正。

```
Enter n, seed, nb:N=16384 Seed=1 NB=256
read/set mat end
copy mat end
Emax=  3.875e-09
Nswap=0 cpsec =  300.427 wsec=75.2283
swaprows time=8.10422e+08 ops/cycle=0.165615
scalerow time=3.75172e+08 ops/cycle=0.35775
trans rtoc8 time=7.66713e+08 ops/cycle=0.175056
trans ctor8 time=5.03349e+08 ops/cycle=0.266649
trans mmul  time=1.57006e+09 ops/cycle=1.28229
trans nonrec cdec time=6.97766e+08 ops/cycle=0.192354
trans vvmul time=2.26423e+08 ops/cycle=0.592773
trans findp time=4.67333e+08 ops/cycle=0.287199
solve tri u time=2.66089e+08 ops/cycle=6.15733e-05
solve tri   time=5.89679e+09 ops/cycle=2.77846e-06
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=1.24246e+09 ops/cycle=1.72842
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=4.10345e+08 ops/cycle=1.30834
DGEMM       time=1.93737e+11 ops/cycle=15.1341
```

DGEMM 72.6 sec.

残りは 2.5 sec なので、説明はできている。

```
Enter n, seed, nb:N=32768 Seed=1 NB=2048
a, awork offset size= 7f223c800000 7f243d000000 200800000 200800000 200800000
size of size_t and off_t long= 8 8 8
read/set mat end
copy mat end
Emax=  4.229e-07
Nswap=0 cpsec =  2441.84 wsec=612.648
swaprows time=3.30799e+09 ops/cycle=0.162295
scalerow time=1.60583e+09 ops/cycle=0.334325
trans rtoc8 time=4.15807e+09 ops/cycle=0.129115
trans ctor8 time=2.23611e+09 ops/cycle=0.240091
trans mmul  time=6.2654e+09 ops/cycle=1.28532
trans nonrec cdec time=2.62081e+09 ops/cycle=0.204849
trans vvmul time=9.41764e+08 ops/cycle=0.57007
trans findp time=1.66965e+09 ops/cycle=0.321548
solve tri u time=1.10225e+10 ops/cycle=2.97282e-06
solve tri   time=1.42092e+11 ops/cycle=2.30611e-07
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=5.46641e+09 ops/cycle=1.5714
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=1.62096e+09 ops/cycle=1.32482
DGEMM       time=1.60535e+12 ops/cycle=14.6113
```

DGEMM 603.4 sec 残り 9 sec

34 K なら 10 秒ちょっと。

34K の時のパネル分解の計算量。 N, K で考える。行列要素数は $N^2/2$ で、要素当りの計算量は基本的に内積演算の平均の長さが $K/2$ なので、$KN^2/2$

42.56Gflops で 29 秒。

```
Enter n, seed, nb:N=34816 Seed=1 NB=2048
a, awork offset size= 7fb84f200000 7fba91a00000 242800000 242800000 242800000
size of size_t and off_t long= 8 8 8
read/set mat end
copy mat end
Emax=  5.044e-07
Nswap=0 cpsec =  2912.46 wsec=730.613
swaprows time=3.72683e+09 ops/cycle=0.162625
scalerow time=1.96941e+09 ops/cycle=0.307745
trans rtoc8 time=4.34275e+09 ops/cycle=0.139561
trans ctor8 time=2.38512e+09 ops/cycle=0.254108
trans mmul  time=6.94446e+09 ops/cycle=1.30912
trans nonrec cdec time=2.903e+09 ops/cycle=0.208776
trans vvmul time=1.02328e+09 ops/cycle=0.592288
trans findp time=1.86832e+09 ops/cycle=0.324396
solve tri u time=1.16829e+10 ops/cycle=2.98007e-06
solve tri   time=1.60738e+11 ops/cycle=2.16601e-07
matmul nk8  time=0 ops/cycle=inf
```

```
matmul snk  time=6.39244e+09 ops/cycle=1.51698
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=1.82715e+09 ops/cycle=1.32683
DGEMM       time=1.91709e+12 ops/cycle=14.6758
```

どれだけ短くできるか、を考える。

```
  計算量の半分: K=1024  600Gflops 14 倍
  計算量の 1/4 : K=512  300       7
  計算量の 1/8 : K=256  150       3.5
  計算量の 1/16 : K=128 75        1.75
```

```
合計は: 0.21
```

```
Enter n, seed, nb:N=34816 Seed=1 NB=2048
a, awork offset size= 7fae21c00000 7fb064400000 242800000 242800000 242800000
size of size_t and off_t long= 8 8 8
read/set mat end
copy mat end
Emax=  5.044e-07
Nswap=0 cpsec =  2910.57 wsec=730.139
swaprows time=3.72784e+09 ops/cycle=0.162581
scalerow time=1.79725e+09 ops/cycle=0.337224
trans rtoc8 time=4.34249e+09 ops/cycle=0.139569
trans ctor8 time=2.40224e+09 ops/cycle=0.252296
trans mmul  time=5.83755e+09 ops/cycle=1.55736
trans nonrec cdec time=2.9062e+09 ops/cycle=0.208546
trans vvmul time=1.02325e+09 ops/cycle=0.592304
trans findp time=1.86805e+09 ops/cycle=0.324444
solve tri u time=1.16613e+10 ops/cycle=2.9856e-06
solve tri   time=1.6063e+11 ops/cycle=2.16747e-07
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=6.36856e+09 ops/cycle=1.52267
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=1.82159e+09 ops/cycle=1.33087
DGEMM       time=1.91704e+12 ops/cycle=14.6762
```

NB=256 では

```
Enter n, seed, nb:N=34816 Seed=1 NB=256
a, awork offset size= 7fad35600000 7faf77e00000 242800000 242800000 242800000
size of size_t and off_t long= 8 8 8
read/set mat end
copy mat end
Emax=  9.244e-08
Nswap=0 cpsec =  2797.93 wsec=700.362
swaprows time=3.57559e+09 ops/cycle=0.169504
scalerow time=1.71234e+09 ops/cycle=0.353947
```

```
trans rtoc8 time=4.35866e+09 ops/cycle=0.139051
trans ctor8 time=2.40053e+09 ops/cycle=0.252476
trans mmul  time=5.83701e+09 ops/cycle=1.5575
trans nonrec cdec time=2.90349e+09 ops/cycle=0.208741
trans vvmul time=1.02339e+09 ops/cycle=0.592227
trans findp time=1.8687e+09 ops/cycle=0.324331
solve tri u time=5.74246e+08 ops/cycle=6.06291e-05
solve tri   time=2.66881e+10 ops/cycle=1.30455e-06
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=6.3678e+09 ops/cycle=1.52286
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=1.8209e+09 ops/cycle=1.33138
DGEMM       time=1.83881e+12 ops/cycle=15.3006

swaprows time=3.57559e+09 ops/cycle=0.169504
scalerow time=1.71234e+09 ops/cycle=0.353947
trans rtoc8 time=4.35866e+09 ops/cycle=0.139051
trans ctor8 time=2.40053e+09 ops/cycle=0.252476
trans mmul  time=5.83701e+09 ops/cycle=1.5575
trans nonrec cdec time=2.90349e+09 ops/cycle=0.208741
trans vvmul time=1.02339e+09 ops/cycle=0.592227
trans findp time=1.8687e+09 ops/cycle=0.324331
matmul snk  time=6.3678e+09 ops/cycle=1.52286
trans mmul4 time=1.8209e+09 ops/cycle=1.33138
```

このへんの細かいところを全部もうちょっとつめたい、というのが残ったチューニングになる。

行列データ: 9.7GB (not GiB)。交換は read+write 18GB, scale 9.7GB、trans rtoc, ctor それぞれ 9.7GB

trans mmul: K=16 の下。演算数としては、要素当り 16 演算なので、9.7G 演算、時間 2 秒なので 4Gflops 程度しかでていない。メモリアクセスは、K= 2, 4, 8 で 3 回、それぞれ行の下半分を半分読んで半分書く。なので、15GB。2 秒なので、 7.5GB/s。ここはもうちょっと速くならないか、という気がする。そもそも、データサイズとして 34k の場合でも 16 行、4MB なので、L3 キャッシュに全部はいっているんだと思うんだが、、、

あと、見た目でできそうなチューニング箇所は以下の通り。

現状の転置方式は、行列消去が進んでも n x 16 の配列にコピーする、という形、つまり、上のほうに空きがある形にコピーするようになっている。これはコードが簡単だが、明らかに効率は悪い。配列のディメンションを変えて、(n-i) x 16 の領域を連続アクセスできるようにすればいくらかは速くなるはず。

あと、転置前の nx16 の行列乗算がちょっとありえないほど遅い。32k で、転置とこの乗算で合わせて 4 秒程度かかっていて、時間の大半である。ここでは、実は乗算の結果を一旦メモリにストアしてから転置しているが、これは本当は無駄なので、乗算した結果を転置しながら直接ストアすればこのメモリアクセスを 60% にできるだけでなく、不連続ストアを減らすこともできる。

が、それ以前にここまでを GotoBLAS にすると、

```
Enter n, seed, nb:N=8192 Seed=1 NB=128
read/set mat end
copy mat end
Emax=  6.833e-10
```

```
Nswap=0 cpsec =  39.4825 wsec=9.88793 37.0658 Gflops
swaprows time=2.04298e+08 ops/cycle=0.164242
scalerow time=7.06527e+07 ops/cycle=0.474921
trans rtoc8 time=1.43615e+08 ops/cycle=0.233642
trans ctor8 time=1.37003e+08 ops/cycle=0.244917
trans mmul  time=2.94734e+08 ops/cycle=1.7077
trans nonrec cdec time=1.69648e+08 ops/cycle=0.197789
trans vvmul time=5.64071e+07 ops/cycle=0.594862
trans findp time=1.13391e+08 ops/cycle=0.295919
solve tri u time=5.94506e+07 ops/cycle=0.000137795
solve tri   time=9.13853e+08 ops/cycle=9.39969
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=54240 ops/cycle=9898.06
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=8.26731e+07 ops/cycle=1.62348
trans mmul2 time=4.56105e+07 ops/cycle=1.47135
DGEMM       time=2.50872e+10 ops/cycle=14.9516
```

変更前:

```
N=8192 Seed=1 NB=128
Emax=  1.902e-09
Nswap=0 cpsec =  39.7945 wsec=9.97176 36.7542 Gflops
swaprows time=2.03733e+08 ops/cycle=0.164698
scalerow time=7.06972e+07 ops/cycle=0.474622
trans rtoc8 time=1.4062e+08 ops/cycle=0.238617
trans ctor8 time=1.36213e+08 ops/cycle=0.246337
trans mmul  time=2.95006e+08 ops/cycle=1.70612
trans nonrec cdec time=1.6697e+08 ops/cycle=0.20096
trans vvmul time=5.64536e+07 ops/cycle=0.594372
trans findp time=1.10669e+08 ops/cycle=0.303197
solve tri u time=5.95494e+07 ops/cycle=0.000137566
solve tri   time=9.09796e+08 ops/cycle=9.44161
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=3.05898e+08 ops/cycle=1.75506
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=8.29545e+07 ops/cycle=1.61797
trans mmul2 time=4.55855e+07 ops/cycle=1.47215
DGEMM       time=2.50133e+10 ops/cycle=14.9958
```

元々は nx16 に 3.05898e+08 サイクル。DGEMM の差はずっと小さいので、これはこっちを使えば OK と思われる。

```
DGEMM       time=2.50872e+10 ops/cycle=14.9516
DGEMM       time=2.50133e+10 ops/cycle=14.9958
```

HPL の速度は

```
================================================================================
T/V                N    NB     P     Q               Time             Gflops
--------------------------------------------------------------------------------
```

```
WR00L2C2        4096   128     1     1              1.36          3.362e+01
WR00L2C4        4096   128     1     1              1.35          3.401e+01
WR00L2C2        8192   128     1     1             10.18          3.601e+01
WR00L2C4        8192   128     1     1             10.10          3.631e+01
================================================================================
```

なので、 N=8192 の場合に、計算量が 1.5% 程度増加している (dtrsm の処理のため) が時間は 2% 程度減少していることがわかる。

Chapter 10

HPL 書き直しその2 (2009/9/12 書きかけ)

ここでは、主に自分の中での整理、という意味で現在の書き直し版コード (名前は lu2 ということにします) の概要をまとめておきます。

lu2 の特色は、

- 行列の格納順序を途中で転置することで、行スワップとピボットサーチの両方を連続アクセスにする
- パネル分解は普通の再帰。(Gustavson のアルゴリズム)
- 右上側の処理 (通常なら DTRSM) に、DGEMM と再帰的 DTRSM を組み合わせた新しいアルゴリズムを開発

というところです。最後のところは、普通の計算機では若干計算量が増えてしまうのですが、GRAPE-DR のような行列積だけが速くなる計算機では重要になります。また、普通の計算機でも DGEMM のチューニングは DTRSM よりも容易だったりするので、場合によってはメリットがあります。

転置の部分の関数を以下に示すます。これは 16 行の転置専用の関数です。

```
void transpose_rowtocol16(int n, double a[][RDIM], double at[][n],
                          int istart)
{
    int i,j,k;
    const int m=16;
    double atmp[m][m];
    int mend;
#ifdef TIMETEST
    BEGIN_TSC;
#endif

    //#pragma omp parallel for private(i,j,k,atmp)
    for(i=istart;i<n;i+=m){
        for(k=0;k<m;k++){
            v2df * ak = (v2df*) a[i+k];
            v2df * akk = (v2df*) atmp[k];
```

```
            akk[0]  =ak[0];
            akk[1]  =ak[1];
            akk[2]  =ak[2];
            akk[3]  =ak[3];
            akk[4]  =ak[4];
            akk[5]  =ak[5];
            akk[6]  =ak[6];
            akk[7]  =ak[7];
        }
        for(j=0;j<m;j++){
            v2df * atk= (v2df*)(at[j]+i);
            atk[0]=(v2df){atmp[0][j],atmp[1][j]};
            atk[1]=(v2df){atmp[2][j],atmp[3][j]};
            atk[2]=(v2df){atmp[4][j],atmp[5][j]};
            atk[3]=(v2df){atmp[6][j],atmp[7][j]};
            atk[4]=(v2df){atmp[8][j],atmp[9][j]};
            atk[5]=(v2df){atmp[10][j],atmp[11][j]};
            atk[6]=(v2df){atmp[12][j],atmp[13][j]};
            atk[7]=(v2df){atmp[14][j],atmp[15][j]};
        }
    }
#ifdef TIMETEST
    END_TSC(t,2);
#endif
}
```

これは特にこったことはしていなくて、行優先で格納された行列を 16 行読み出して連続アドレス領域に一旦格納し、それを転置しながら格納します。16 行 (128 バイト) を単位にしているのは、 Core i7 ではキャッシュを 2 ラインづつ勝手にに読んでくる機能があるので、その辺が性能的には具合がよいからです。現在のコードでは、転置前の行列を連続アドレスの小行列にコピーしてから、いきなり転置行列に書き込んでいますが、一旦連続アドレスでの転置した小行列に書き込んでから最終的な縦長の行列に書くほうが具合がよいかもしれません。

この部分の転送速度は、現在のところ Core i7 920 で 4GB read+write (合計 8GB) に 1.4 秒、5.6GB/s というところです。転置前の行列サイズは最大で 4.25MB、平均ではずっと小さくて殆どの場合に L3 にいるはずであることを考えるともうちょっと速くてもよいはずで、まだかなり工夫の余地があります。L3 のバンド幅自体は 50GB/s 程度あるのに対してその 1/10 程度の速度しかでていないからです。

もっとも、L3 レイテンシが 50 サイクル程度なので、16 ワード、つまり 128 バイトアクセス毎に 50 サイクルかかるのかもしれません。それでは話にならないので、16x16 の行列全部を L1 までもってくるようなプリフェッチをかけることができれば改善しそうです。GCC の __builtin_prefetch では今一つ細かい制御ができないので、

```
  asm("prefetcht0 %0"::"m"(a[i+m][0]):"memory");
```

(prefetchnta を使うべき？) のように、実際に SSE プリフェッチを手で入れるべきかもしれません。Ci7 は 32KB もの巨大な L1 をもつのですが、1 度に読み込む行列は 2KB しかありません。なので、かなり先までプリフェッチをだしておいてもよいはずです。

逆方向の転置は以下のようにしています。

```
void transpose_coltorow16(int n, double a[][RDIM], double at[][n],
```

```
                        int istart)
{
    int i,j,k;
    const int m=16;
    double atmp[m][m];
#ifdef TIMETEST
    BEGIN_TSC;
#endif
#pragma omp parallel for private(i,j,k,atmp)
    for(i=istart;i<n;i+=m){
        for(k=0;k<m;k++){
            v2df * ak = (v2df*) (at[k]+i);
            v2df * akk = (v2df*) atmp[k];
            akk[0]  =ak[0];
            akk[1]  =ak[1];
            akk[2]  =ak[2];
            akk[3]  =ak[3];
            akk[4]  =ak[4];
            akk[5]  =ak[5];
            akk[6]  =ak[6];
            akk[7]  =ak[7];
        }
        for(j=0;j<m;j++){
            v2df * atk= (v2df*)(a[i+j]);
            atk[0]=(v2df){atmp[0][j],atmp[1][j]};
            atk[1]=(v2df){atmp[2][j],atmp[3][j]};
            atk[2]=(v2df){atmp[4][j],atmp[5][j]};
            atk[3]=(v2df){atmp[6][j],atmp[7][j]};
            atk[4]=(v2df){atmp[8][j],atmp[9][j]};
            atk[5]=(v2df){atmp[10][j],atmp[11][j]};
            atk[6]=(v2df){atmp[12][j],atmp[13][j]};
            atk[7]=(v2df){atmp[14][j],atmp[15][j]};
        }
    }
#ifdef TIMETEST
    END_TSC(t,3);
#endif
}
```

要するに、構造は同じです。構造は同じで、メモリアクセスの量も同じなのですが、こちらのほうが 2 倍近く速度が速くなっています。これは、読出しアドレスの連続性が高い、つまり、 row-to-col の場合には長さ n のベクトルから 16 ワードを読む、というのを、 n 個の別のベクトルに順番に適用するのに対して、 col-to-row では 16 本の長さ n のベクトルから 16 語ずつ読むので、フットプリントが小さい、というのが 1 つの理由でしょう。

但し、2 倍速いだけで、十分に速い、というわけでは決してないので、改善の余地がないわけではありません。

```
    asm("prefetcht0 %0"::"m"(at[k][i+m*3]):"memory");
    asm("prefetcht0 %0"::"m"(at[k][i+m*3+8]):"memory");
```

をいれてみた場合のバンド幅変化は以下のようでした

```
ctor            N=8192   16384
no prefetch     0.2443   0.2613
prefetchnta     0.3177
prefetcht0      0.3180
prefetcht1      0.3235
prefetcht2      0.3239   0.3235

rtoc            N=8192   16384
no prefetch     0.2327   0.2043
prefetchnta     0.2813   0.2506
prefetcht0      0.2789
prefetcht1      0.2608
prefetcht2      0.2605
builtin(0,0)    0.2778
```

で、 prefetch で 2-3 割は上がりますが、5 倍とかには到底いかないです。試しに、rtoc のケースで小行列へのコピーと小行列からのコピーを別々に測定して見ると、

```
小行列へのコピー     0.5746
小行列からのコピー 0.6643
```

で、小行列「から」に意外に時間がかかっています。これを、転置を小行列の範囲でもう一度やるようにすると、最後の連続アドレスでのコピーは 0.8 語/ サイクル程度まで向上しますが、転置自体に 1 語/サイクル程度の時間がかかって総合的にはかえって速度低下します。

通常の計算機で実行する分には、例えば 40Gflops で 34k の行列だと 700 秒程度になり、1 秒を問題にする意味はあまりありません。が。 1Tflops 出すとこれが 25 秒で終わってしまうので、行列乗算でないところを 5 秒程度まで抑え込む必要があります。このためには、つめられるものは極限までつめる必要があるわけです。現在のコード (2009/9/13 現在) では、34k での実行時間が

```
行交換              1.3
行スケール          0.6
行->列転置          1.4
列->行転置          0.8
転置後の行列乗算  2.0
2 列でのソルバ      1.5
合計                7.6 (秒)
```

という程度ですが、これをあと 2-3 秒はつめたいところです。これらの処理は殆ど L3 にあるデータに対する操作なので、全体にもう数倍は速くできてもよいはずなので、これは不可能ではないと思います。

と、話がずれてますね。再帰のところは、実際に書いてみると

```
void column_decomposition_recursive(int n,
                                    double a[n][RDIM],
                                    int m,
                                    double awork[][n],
                                    int pv[],
                                    int i)
{
    int  j, k;
```

```
    int ip,ii;
    double ainv;
    if (m <= 16){
        column_decomposition_with_transpose(n, a, m,awork, pv,i);
    }else{
        column_decomposition_recursive(n, a, m/2, awork, pv,i);
        process_right_part(n,a,m/2,awork, pv,i,i+m);
        column_decomposition_recursive(n, a, m/2, awork, pv+m/2,i+m/2);
        for(ii=i+m/2;ii<i+m;ii++){
            swaprows(n,a,pv[ii-i],ii,i,i+m/2);
        }
        for(ii=i+m/2;ii<i+m;ii++){
            scalerow(n,a,1.0/a[ii][ii] ,ii,i,i+m/2);
        }
    }
}
```

と単純な構造です。日本語での疑似コードにして見ると

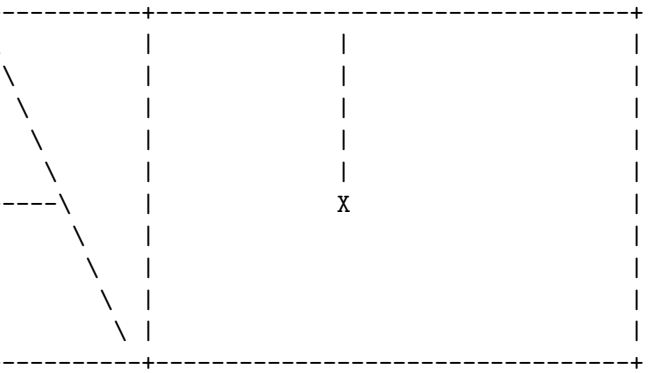
```
    幅が 16 以下になったら
        転置してパネル分解する
    でなければ
         左半分をパネル分解する
 左半分の分解結果を使って右側半分を更新する
 右側半分を分解する
 右側半分の結果を使って、左半分の行交換とスケーリングをする
```

です。

右上側の処理 (通常なら DTRSM) に、DGEMM と再帰的 DTRSM を組み合わせた新しいアルゴリズムですが、本来するべき処理は

```
    for(ii=i;ii<i+m;ii++)
        for(j=ii+1;j<i+m;j++)
            for (k=i+m;k<n;k++)
        a[j][k] -= a[j][ii]*a[ii][k];
```

というものです。図で書くと以下のような感じです。



X の位置のデータから、左の下三角行列の 「—」と、 X の上の列との内積を引くわけですが、単純な行列積ではなくて、1 つ上の行の要素は変形してからの操作になります。わかりにくいので、右側の行列の列 1 行についての操作にしてみると、

```
    for(ii=i;ii<i+m;ii++)
        for(j=ii+1;j<i+m;j++)
        b[j] -= a[j][ii]*b[ii];
```

ですね。この形式だと b の計算はいわゆる recurrent なもので、 b[i] が求まらないと b[i+1] が計算できないですが、数学的には、新しい b[k] は b[0] から b[k] までの線形結合ですから、a を変形しておけば

```
    for(j=i+1;j<i+m;j++){
        c[j]=0;
        for(ii=i;ii<=j;ii++)
      c[j] += anew[j][ii]*b[ii];
    }
```

という形にできます。これは、anew も上半分がない、ということを別にすれば単純なベクトル行列積なので、k の方向も考えると行列積になって、効率が半分になるのをいいことにすれば GRAPE-DR や、他の高速な行列計算ライブラリで実行できます。

anew は、単位行列を上の処理と同じ計算で変形すればよい、つまり、

```
    for(ii=0;ii<m;ii++)

for(j=ii+1;j<m;j++)
    for (k=0;k<m;k++)
b[j][k] -= a[j][ii]*b[ii][k];
```

というような計算をすればよいわけですが、ここでも DGEMM を引っ張り出すことを考えてみます。書いたコードを示します。

```
 static void solve_triangle_for_unit_mat_recursive(int n,
                                  double a[][RDIM],
                                  int nb,
                                  double b[][nb],
                                  int m)
 {
     int i,ii,j,k;
     if (m < 16){
         solve_triangle_for_unit_mat_internal(n, a, nb, b,m);
         return;
     }
     const int mhalf = m/2;
     solve_triangle_for_unit_mat_recursive(n, a, nb, b,mhalf);

     dgemm( mhalf, mhalf, mhalf, -1.0, &(a[mhalf][0]), RDIM,
                 &(b[0][0]), nb, 1.0, &(b[mhalf][0]),nb );

     double bwork[mhalf][mhalf];
```

```
    double bwork2[mhalf][mhalf];
    for (j=0;j<mhalf;j++)
        for (k=0;k<mhalf;k++)bwork[j][k]=0.0;
    for (j=0;j<mhalf;j++)bwork[j][j]=1.0;
    solve_triangle_for_unit_mat_recursive(n, (double(*)[])(&a[mhalf][mhalf]),
                                mhalf, bwork,mhalf);
    for(i=0;i<mhalf;i++)
        for(j=0;j<mhalf;j++)
            bwork2[i][j]=b[i+mhalf][j];
    dgemm(mhalf, mhalf, mhalf, 1.0, (double*)bwork,mhalf,
                (double*)bwork2, mhalf, 0.0, &(b[mhalf][0]),nb );

    for (j=0;j<mhalf;j++)
for (k=0;k<j+1;k++)b[mhalf+j][mhalf+k]=bwork[j][k];
}
```

このコードに表現した手順は、b に単位行列がはいっているとして、

まず左上 1/4 だけを変形する
変形した結果と a の左下 1/4 との積を b の左下 1/4 から引く
右下 1/4 を変形する
変形した結果と b の左下 1/4 との積を b の左下 1/4 から引く

というものです。これにでてくる行列積に単純に DGEMM を使うと、計算量が 2 倍になって若干損ですが、 GRAPE-DR 等が使えるならそれでもそのほうがメリットがあるわけです。普通の計算機なら、3 角行列と正方行列の乗算専用のルーチンを書けば問題はありません。

Chapter 11

HPL 書き直しその3 (2009/9/14 書きかけ)

lu2 では、基本データ構造は単純な row major としましたが、実際にコードを書いて性能を測ってみるとやはりこれは問題があるように見えます。それは、例えば hugetlb を使うにしても、細い幅で縦にアクセスする時にどうも遅い、ということ、もうひとつは、横にアクセスする時にも、並列化効率は低い、ということです。

例えば行をスケールする時には、複数の行を並列に処理でき、この時には複数のコアが全く違うところをアクセスするので高速にできます。ところが、行交換では、1 行づつやっていくので並列化は行内ですることになり、連続アドレスのベクトルを分割してアクセスすることになるのであまり並列化効率が上がりません。

これを改善する方法ですが、以下の 2 つがありえるように思います。

1. 転置のサイズをずっと大きくする
2. 単純な row major でなく、ブロック化する

現在は幅 16 くらいで転置していて、これは基本的には転置した前後に行列が l3 に入ることを期待しているからです。しかし、はいっているにもかかわらず速度が 0.3words/cycle, つまり 2.67ghz では 14gb/s 程度しかでないなら、主記憶を使っても速度はあまり変わりません。この場合には、幅が 16 とか 32 とか、もっと広い幅でのアクセスをしたほうがよいような気もします。

しかし、これは実際にやってみると速度が 1/4 程度になってしまって全く話になりません。

ブロック化は、例えば 2048 くらいの長さに行を切って、それが縦に並ぶ、という形にする、というものです。この、n x 2048 のブロックを横に並べて行列を作るわけです。 そうすると、行交換等は幅 2048 のブロック単位に並列化すればアドレス空間は完全に独立になります。また、転置したものも同様にブロック化しておくと、行列乗算の処理をアドレス空間を分離して並列化できます。転置についても有利になるはずです。

この場合に効率の観点から 1 つだけ問題なのは、DGEMM を呼ぶのが煩雑になる、つまり、スレッド並列されている DGEMM を呼ぶためには横長の行列に代入しなおして、結果も戻す必要があることです。

但し、コードはかなり煩雑になるので、あまりやりたくない、という気もします。

行交換については、もうちょっと賢いやり方はないか？というのが問題です。例えば、行交換とスケールで、広い領域を 2 度アクセスするのはいかにも非効率です。また、行交換には相当程度並列性

があるはずです。つまり、ほとんどの交換は独立なはずです。もちろん、問題は、独立でない行交換があったらどうするかです。

独立でない交換とは、例えば 2 行目を 8 行目と交換して、次に 3 行目を交換したらそこに前の 2 行目がはいってきてしまった、というものです。ブロックサイズが大きくなると、起こる確率はもちろんあがるので、無視して処理するわけにはいきません。

まあ、そうはいっても割合は低いはずなので、単純に、独立でないものは分離してシリアルに処理して、それ以外を並列に処理、というのでよいように思います。

但し、 swaprows については、現在でも 15GB/s ていどの速度なので改善の余地はそれほどないかもしれません。であれば、単純に、 swaprows で上側に代入する時にスケールもする、というので十分でしょう。これは実装しています。

```
void swaprows_simple_with_scale(int n, double a[n][RDIM],
                                int row1, int row2,
                                 int cstart, int cend, double scale)
{
    int j;
    if (row1 != row2){
        int jmax = (cend+1-cstart)/2;
        v2df *a1, *a2, tmp, tmp1;
        v2df ss = (v2df){scale,scale};
        a1 = (v2df*)(a[row1]+cstart);
        a2 = (v2df*)(a[row2]+cstart);
        for(j=0;j<(jmax & (0xfffffffe));j+=2){
            __builtin_prefetch(a1+j+32,1,0);
            __builtin_prefetch(a2+j+32,1,0);
            tmp = a1[j];
            a1[j]=a2[j];
            a2[j]=tmp*ss;
            tmp1 = a1[j+1];
            a1[j+1]=a2[j+1];
            a2[j+1]=tmp1*ss;
        }
        if (jmax &1){
            tmp = a1[jmax-1];
            a1[jmax-1]=a2[jmax-1];
            a2[jmax-1]=tmp*ss;
        }
    }else{
        scalerow(n,a,scale ,row2,cstart,cend);
    }
}
```

まあ、どうして初めからそうしてなかったんだ？というような話です。これで、 34 k で交換+スケーリングが大体 1.3 秒で、スケーリングにかかっていた 0.6 秒が減りました。

2 列にまで再帰したところでの変形では、BLAS1 でいう DAXPY な操作がでてきます。これについては sse を使ってさらに 4 重にアンロールすると、バンド幅として 20GB/s 程度になり、まあいいかな、という程度です。

この程度までチューニングすると、ピボットのための最大値探索が結構遅いものとして見えてきます。

```
Enter n, seed, nb:N=34816 Seed=1 NB=256
read/set mat end
copy mat end
Emax=  1.136e-07
Nswap=0 cpsec =  2784.11 wsec=696.654 40.3857 Gflops
swaprows time=3.43235e+09 ops/cycle=0.176578
scalerow time=1.56005e+07 ops/cycle=38.8497
trans rtoc  time=3.14886e+09 ops/cycle=0.192475
trans ctor  time=1.89551e+09 ops/cycle=0.319744
trans mmul  time=5.14042e+09 ops/cycle=1.76856
trans nonrec cdec time=2.94967e+09 ops/cycle=0.205472
trans vvmul time=5.03337e+08 ops/cycle=1.20412
trans findp time=2.44507e+09 ops/cycle=0.247877
solve tri u time=3.80914e+08 ops/cycle=9.14013e-05
solve tri   time=2.67092e+10 ops/cycle=11.6182
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=67816 ops/cycle=142993
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=1.39435e+09 ops/cycle=1.73866
trans mmul2 time=8.13527e+08 ops/cycle=1.49
DGEMM       time=1.83959e+12 ops/cycle=15.4628
```

これは、小細工ではどうしようもないのですが、*Intel* の技術メモ[1] に詳しく議論したものがあって SSE2 を使ってオールアセンブラでやれば 2-3 倍速くなりそうです。これは、自力で書くものでもなくて GotoBLAS の idamax を使うので OK でした。

```
Enter n, seed, nb:N=34816 Seed=1 NB=256
Emax=  1.136e-07
Nswap=0 cpsec =  2782.51 wsec=696.245 40.4095 Gflops
swaprows time=3.43022e+09 ops/cycle=0.176688
scalerow time=1.56175e+07 ops/cycle=38.8077
trans rtoc  time=3.14043e+09 ops/cycle=0.192992
trans ctor  time=1.89603e+09 ops/cycle=0.319656
trans mmul  time=5.14364e+09 ops/cycle=1.76745
trans nonrec cdec time=1.40592e+09 ops/cycle=0.431089
trans vvmul time=4.97509e+08 ops/cycle=1.21822
trans findp time=9.07189e+08 ops/cycle=0.668083
solve tri u time=3.85358e+08 ops/cycle=9.03471e-05
solve tri   time=2.66933e+10 ops/cycle=11.6251
matmul nk8  time=0 ops/cycle=inf
matmul snk  time=59096 ops/cycle=164093
trans mmul8 time=0 ops/cycle=inf
trans mmul4 time=1.39377e+09 ops/cycle=1.73939
trans mmul2 time=8.11773e+08 ops/cycle=1.49322
DGEMM       time=1.84005e+12 ops/cycle=15.459
```

現時点で、大きく残っているのは k=8 の行列乗算で、k=4 の場合に対して演算量は 2 倍。メモリアクセスは半分なんだが時間は 2 倍でほぼ 1 秒かかっている。k=8 の時には GotoBLAS を使っている

[1] http://download.intel.com/jp/developer/jpdoc/w_max_app_j.pdf

が、素晴らしく速い、というわけでもありません。それ以前にそもそも問題なのは k=4 のケースです。k=2 の場合にサイクル当り 1.5 演算です。これはメモリバンド幅としては、アクセス 3 語あたりに 4 演算するので、9/8 語/サイクルとなり、主記憶一杯一杯です。キャッシュにはいってるはずなのにどうして遅いのかはともかく、そんな悪い数字ではありません。で、希望としては、k=4 になったらメモリアクセスに対する演算数は倍になるので、3 演算くらいしてくれ、と思うわけです。ここはシングルコア実行なので、4 演算が理論限界ですが、それでもまあ 3 くらいはいってくれてもよさそうなものです。現在使っている k=4 のコードは以下です。

```
static void matmul_for_nk4(int n1, double a[][n1],
                           int n2, double b[][n2],
                           int n3, double c[][n3],
                           int n)
{
    register v2df b00 = (v2df){b[0][0],b[0][0]};
    register v2df b01 = (v2df){b[0][1],b[0][1]};
    register v2df b02 = (v2df){b[0][2],b[0][2]};
    register v2df b03 = (v2df){b[0][3],b[0][3]};
    register v2df b10 = (v2df){b[1][0],b[1][0]};
    register v2df b11 = (v2df){b[1][1],b[1][1]};
    register v2df b12 = (v2df){b[1][2],b[1][2]};
    register v2df b13 = (v2df){b[1][3],b[1][3]};
    register v2df b20 = (v2df){b[2][0],b[2][0]};
    register v2df b21 = (v2df){b[2][1],b[2][1]};
    register v2df b22 = (v2df){b[2][2],b[2][2]};
    register v2df b23 = (v2df){b[2][3],b[2][3]};
    register v2df b30 = (v2df){b[3][0],b[3][0]};
    register v2df b31 = (v2df){b[3][1],b[3][1]};
    register v2df b32 = (v2df){b[3][2],b[3][2]};
    register v2df b33 = (v2df){b[3][3],b[3][3]};

    v2df * a0 = (v2df*) a[0];
    v2df * a1 = (v2df*) a[1];
    v2df * a2 = (v2df*) a[2];
    v2df * a3 = (v2df*) a[3];
    v2df * c0 = (v2df*) c[0];
    v2df * c1 = (v2df*) c[1];
    v2df * c2 = (v2df*) c[2];
    v2df * c3 = (v2df*) c[3];
    int nh = n>>1;

    int j;
    if (nh & 1){
        j=nh-1;
        c0[j] -= a0[j]*b00+a1[j]*b01+a2[j]*b02+a3[j]*b03;
        c1[j] -= a0[j]*b10+a1[j]*b11+a2[j]*b12+a3[j]*b13;
        c2[j] -= a0[j]*b20+a1[j]*b21+a2[j]*b22+a3[j]*b23;
        c3[j] -= a0[j]*b30+a1[j]*b31+a2[j]*b32+a3[j]*b33;
        nh--;
    }

    for(j=0;j<nh;j+=2){
        c0[j] -= a0[j]*b00+a1[j]*b01+a2[j]*b02+a3[j]*b03;
```

```
        c0[j+1] -= a0[j+1]*b00+a1[j+1]*b01+a2[j+1]*b02+a3[j+1]*b03;
        c1[j] -= a0[j]*b10+a1[j]*b11+a2[j]*b12+a3[j]*b13;
        c1[j+1] -= a0[j+1]*b10+a1[j+1]*b11+a2[j+1]*b12+a3[j+1]*b13;
    }

    for(j=0;j<nh;j+=2){
        c2[j] -= a0[j]*b20+a1[j]*b21+a2[j]*b22+a3[j]*b23;
        c2[j+1] -= a0[j+1]*b20+a1[j+1]*b21+a2[j+1]*b22+a3[j+1]*b23;
        c3[j] -= a0[j]*b30+a1[j]*b31+a2[j]*b32+a3[j]*b33;
        c3[j+1] -= a0[j+1]*b30+a1[j+1]*b31+a2[j+1]*b32+a3[j+1]*b33;

    }
}
```

ループを 2 重にアンロールするので、最初に端数の処理をします。それから、まず c の 0、1 行目を計算してから独立のループで 2, 3 行目を処理します。

大原さんの解析[2]によると、レジスタと L1 の転送速度限界は read/write それぞれ 2 語/Cycle (L2, L3 はその半分、 1/4) とのことなので、L3 にきているだけのデータだと上の形のループでは 7+2 語アクセスで 16 演算なので、 L3 からデータがくるなら 14 サイクルで 16 演算、L1 からきたら 3.5 サイクルで 16 演算になってピークがでるかも、という感じです。ただ、L1 に入るデータサイズにしたとしても、レジスタの本数が足りないような気がします。 xmm レジスタは 16 本なので、基本的に入らないわけです。そうすると、b が全部溢れているとすれば 32 演算に対して 22 リードで、L1 に全部はいってスケジューリングも理想的として 3 演算です。なので、ここはちゃんと b がレジスタに止まるようなコードを書くことが必須です。出力アセンブラをみながら色々やってみたところでは、以下のコードは b を完全にレジスタに置くようです (gcc 4.1.2, -O2 の場合)

```
static void matmul_for_nk4(int n1, double a[][n1],
                           int n2, double b[][n2],
                           int n3, double c[][n3],
                           int n)
{
    v2df * a0 = (v2df*) a[0];
    v2df * a1 = (v2df*) a[1];
    v2df * a2 = (v2df*) a[2];
    v2df * a3 = (v2df*) a[3];
    int nh = n>>1;
    int j;

    int k;
    v2df * cv;
    v2df * cvv;
    register v2df b0;
    register v2df b1;
    register v2df b2;
    register v2df b3;
    register v2df b4;
    register v2df b5;
    register v2df b6;
    register v2df b7;
```

[2] http://journal.mycom.co.jp/special/2008/nehalem02/011.html

```
    for(k=0;k<4;k+=2){
        cv = (v2df*) c[k];
        cvv = (v2df*) c[k+1];
        b0 = (v2df){b[k][0],b[k][0]};
        b1 = (v2df){b[k][1],b[k][1]};
        b2 = (v2df){b[k][2],b[k][2]};
        b3 = (v2df){b[k][3],b[k][3]};
        b4 = (v2df){b[k+1][0],b[k+1][0]};
        b5 = (v2df){b[k+1][1],b[k+1][1]};
        b6 = (v2df){b[k+1][2],b[k+1][2]};
        b7 = (v2df){b[k+1][3],b[k+1][3]};
        for(j=0;j<nh;j++){
            register v2df aa0 = a0[j];
            register v2df aa1 = a1[j];
            register v2df aa2 = a2[j];
            register v2df aa3 = a3[j];
            register v2df x = aa0*b0;
            x+= aa1*b1;
            x+= aa2*b2;
            x+= aa3*b3;
            cv[j]-=x;
            x = aa0*b4;
            x+= aa1*b5;
            x+= aa2*b6;
            x+= aa3*b7;
            cvv[j] -= x;
        }
    }
}
```

この時に、性能は 2.6 演算/サイクル程度でした。アセンブラ出力を見ると

```
.L19:
        movapd  (%rax,%rsi), %xmm6
        addl    $1, %edi
        movapd  (%rax,%r12), %xmm3
        movapd  %xmm6, %xmm1
        mulpd   %xmm10, %xmm6
        movapd  %xmm3, %xmm0
        mulpd   %xmm14, %xmm1
        movapd  (%rax,%rbp), %xmm4
        mulpd   %xmm13, %xmm0
        movapd  (%rax,%rbx), %xmm5
        mulpd   %xmm9, %xmm3
        movapd  %xmm5, %xmm2
        mulpd   %xmm7, %xmm5
        addpd   %xmm1, %xmm0
        movapd  %xmm4, %xmm1
        mulpd   %xmm11, %xmm2
        addpd   %xmm6, %xmm3
        mulpd   %xmm12, %xmm1
        mulpd   %xmm8, %xmm4
```

```
addpd   %xmm0, %xmm1
movapd  (%rax,%r10), %xmm0
addpd   %xmm3, %xmm4
addpd   %xmm1, %xmm2
addpd   %xmm4, %xmm5
subpd   %xmm2, %xmm0
movapd  %xmm0, (%rax,%r10)
movapd  (%rax,%r8), %xmm0
subpd   %xmm5, %xmm0
movapd  %xmm0, (%rax,%r8)
addq    $16, %rax
cmpl    %r11d, %edi
jne     .L19
```

という感じです。 %xmm[n] は SSE2 命令のレジスタなので、その間の演算が殆どで、メモリからのロード (movapd (%rax,%r12), %xmm3 とか) が 6、ストア (movapd %xmm0, (%rax,%r8) とか) が 2 で、メモリアクセスの観点からはベストなコードです。ループ内の演算数はあまり多くないし、依存関係が多いので、スケジューリングは上手くいっていないかもしれません。ループアンロールをするともうちょっと改善する可能性はあります。

また、このループでは L1 に入る範囲に切る必要があり、さらに少なくとも k=8 の場合には並列化して速度を稼ぐ必要もあるので、そういうチューニングが残っています。

Chapter 12

eASIC と次世代 GRAPE(2009/9/14 書きかけ)

さて、 eASIC です。色々見積もると、 Nextreme-2 で低精度の GRAPE ならチップ当り数 Tflops は容易です。 GPU に比べたメリットは、

- 消費電力
- 実効性能

というところです。但し、低精度、ということで当然 treecode 用になるので、性能が通信リミットになるかどうかが問題です。例えば 10TF というのは 300G interactions/sec, 300MHz のクロックで 1000 本のパイプライン、という話でそれほど無理ではありません。

NX21000 で 100 万 eCell ですが、まあボード 1 つに 2-4 チップだと上の数字くらいになると思われます。

相互作用数を 1000 とすれば、これは 300M 粒子/秒、つまり、3 億粒子が 1 秒で計算できてしまいます。

こうなると通信が大問題で、結果の加速度をなんとかして 4 バイトに収めても 12 バイトで、それだけで 3.6 GB/s と普通では不可能な速度が必要です。また、粒子を送るほうはその数倍必要で、20GB/s とかそういった全く不可能な速度が必要になってしまいます。現実問題としては高い FPGA を使って Gen2 8 レーン、実力 2GB/s 双方向、といった辺りがボード 1 枚での限界です。

そうすると、色々邪道なことを考えないといけません。まず、粒子やツリーの座標データについては、相互作用の精度自体が低いものでよい時にどうやって送るのが最適か、というのが問題です。例えば、ツリーの大きなセルについては、どうせ遠くからしかみないんだから位置が高い精度で決まっている必要はありません。

例えば、相互作用リストをモートンオーダーで作ったら、それに対応するツリー構造が存在していますから、そのツリーのセルの中での相対座標だけを指定すれば例えば力の精度に 10 ビットとるとして、セル中座標にそれくらいあれば十分なはずです。 モートンキー自体については前の粒子との差をとって、これはしょうがないので可変長にします。原理的に、ほとんどの場合に下のほうだけが変わるので 1-2 バイトでよいはずです。そうすると、結局 40 ビット程度で 1 粒子が表現でき、普通の場合の 1/3 程度ですみます。

が、この 3 倍は大きいとはいえ、それでも極限的な転送速度が必要なことに変わりはありません。

ハードウェアの極限性能も重要ですが、競争相手がどれくらい、というのも考えておく必要があるでしょう。現在、 CPU を使った並列化された treecode (TreePM も含めて) でおそらく地球上最高

速なのは石山君の GreeM で、Barcelona 2.2GHz 1 コアで 5-70000 粒子/(秒・コア)、つまりソケット当りでは 20-30 万 (これおよび以下では単位は秒・ソケットあたりの粒子数) となります。GRAPE 等の時の基本として、「構想段階で 100 倍の性能向上」というのがあるのですが、そのためには 20M 粒子でよいことになります。

もっとも、価格の問題はあり、ボードの量産コストが 20 万だとすると PC 1 ソケットの 2 倍するので、さらに速度が 3 倍くらいは本当は欲しい、となって 100M 粒子です。そうすると、上の見積もりの 20GB/s から圧縮して 1/3、速度が 1/3 でよいことにすることでさらに 1/3 となり、2GB/s 実力でででるならまあなんとか、となります。

GPU と比べるとどうか？というのが一つの問題ですが、例えば Hamada et al. (2009) によると Core 2 Quad 6600+nVIdia GF8800GTS では、ノード当り (ノードは 1 ソケット 1 カード)35 万粒子、となっています。これは TreePM ではなくて pure tree なので計算量は多いですが、それにしても、まあ、これに比べるなら新しくハードウェアを作った時に 100 倍 にするのは決して不可能でもなさそうです。もっとも、問題になりそうなのは treecode の場合ホストでのツリーの構築や treewalk です。ツリー構築 (通信も合わせて) は GreeM では全体の 10% 以下なので、数年たってホストが速くなっていることを期待すればまあなんとか、となります。 treewalk は遅いですが、これは逆に ng を大きくするしかありません。つまり、おそらくは 10Tflops/card くらいで、2GB/s も通信速度があれば十分に演算速度を有効に使える、ということになりそうです。これはまあそれほど不思議な話でもなくて、 GRAPE-6 の PCI インターフェースに比べると 100 倍近く、 GRAPE-7 の PCI-X に比べても 10 倍近く加速しよう、という話なので、 10 倍程度性能を積み上げても全く問題はありません。

GPU の場合は現在のところ名目ピーク 700Gflops のボードで 160Gflops 程度で、ハードウェアの計算速度限界に達しているようには見えないです。とすれば、通信リミットなはずです。相互作用リストが 12000, ng が 500 (? 250?) とすれば、i 粒子 1 つについて 24 -50 個 j 粒子を送っています。これを 16 バイトとすれば 400-800 バイトです。 GPU で PCIe の理論性能の 8GB/s が出れば 10M 粒子の性能のはずです。なので、GPU は通信については 30 倍くらい謎な部分があることになります。

GPU が今後数年のうちに謎な部分が直るとも思えないので、結局ツリーコードでの GPU の性能は CPU の 1-3 倍に留まると思ってよいでしょう。新 GRAPE では、現在の CPU, GPU のほぼ 100 倍の性能を数年後に実現することで、従来の GRAPE と同様な「圧倒的なメリット」を目指せそうです。

サイエンス、という観点からは、低精度でよいのか？という問題はあるわけですが、これについては最近藤井さんや押野君が開発してきたハイブリッドスキーム、つまり、遠距離相互作用については tree を使うけれど近距離力には direct scheme を使う、という方法で基本的には対応できるはずです。これらのスキームではツリーの部分ではシンプレクティックにすることで、長時間積分についても十分な精度を維持すると同時に近距離相互作用には高い精度を実現します。

このスキームを使うためには、近距離側での重力相互作用に適当な形のカットオフをいれる必要があり、また TreePM と一緒に使うためには遠距離側にもカットオフが必要です。さらにもうひとつ考えるべきことは、多重極モーメントもいれるかどうかです。例えば 4 重極は書き下すと以下のようなものです。

```
                  r5inv = r2inv*r2inv*rinv
                  phiq = (-.5*((dx*dx-dz*dz)*
     &                 quad(p,1)+(dy*dy-dz*dz)*
     &                 quad(p,4))-(dx*dy*quad(p,2)+
     &                 dx*dz*quad(p,3)+dy*dz*
     &                 quad(p,5)))*r5inv
                  phi(pbody) = phi(pbody)+phiq
                  phiq = 5.*phiq*r2inv
                  acc(pbody,1) = acc(pbody,1)+dx*phiq+(dx*
```

```
&                    quad(p,1)+dy*quad(p,2)+dz*
&                    quad(p,3))*r5inv
                acc(pbody,2) = acc(pbody,2)+dy*phiq+(dy*
&                    quad(p,4)+dx*quad(p,2)+dz*
&                    quad(p,5))*r5inv
                acc(pbody,3) = acc(pbody,3)+dz*phiq+(dz*
&                    (-quad(p,1)-quad(p,4))+dx*quad(p,3)
&                    +dy*quad(p,5))*r5inv
```

これは基本的に固定小数点で行うことができるはずです。乗算の精度は 3-4 ビット、加算はもうちょっと必要ですが 6 ビット程度でしょう。そうすると、乗算 31 個、加算 18 個です。乗算器、加算器ともに 100 ゲート程度ですから、全体で 5000 ゲートになって、演算器のゲート効率は素晴らしく高いものになります。単精度の乗算器が 6000 ゲート程度ですから、普通なら乗算器 1 つしか入らない回路規模で 50 演算くらいできるわけです。これはもちろん、消費電力削減にもそれだけ貢献することになります。

上式は純粋な $1/r$ ポテンシャルの場合で、カットオフが入るともうちょっと式が複雑になりますが、データ量はあまり変わらないので計算速度と通信速度の関係からはさらに有利になるわけです。

GRAPE のような完全に専用化した回路のメリットは、このように極限的にデータ長をつめることで普通のプログラマブルな計算機に比べて 1-2 桁回路規模を小さくできることにあります。プログラマブルな GRAPE-DR の場合はこのようなメリットは失われており、ASIC ゲート数に対する演算器の数は低精度 GRAPE に比べると 2 桁以上低いものになります。もっとも、普通の計算機ではこれは 3-4 桁低いので、 GRAPE-DR にメリットがないわけではないのですが、GRAPE のメリットは他のやり方が無意味に思えるほど大きいものです。

もちろん、 eASIC のような構造化 ASIC を使うと、回路規模での 3 桁の差のある程度は失われます。 eASIC の 45nm プロセスの製品は、順調に開発・検証が進んだとして 2000 万ゲート規模、すなわち ASIC ならば 1 億トランジスタ程度と 130nm で実現可能であったものがせいいっぱいですから、3 世代、つまり大体 10 倍の回路規模の違いがあります。速度については、差は 10 倍もなくなっており、開発期間があまり長くなければチップ当りの性能で 1-2 桁程度の優位、消費電力では 2-3 桁の優位を実現できるわけです。

この、速度差があまり大きくない、というのは、昔からいわれていたゲート遅延より配線遅延が大きくなる、という要因と、カスタム ASIC やマイクロプロセッサの動作速度は既に電力リミットになって久しい、という要因の両方によっています。

Chapter 13

HPL 書き直しその4 (2009/9/16 書きかけ)

lu2 を実際に GRAPE-DR で動かしてみてもらったところ、結構良い性能がでました。そうすると、もっと抜本的な改善の方法はないかというのが問題です。というわけで、 GDR でのテストをしてくれた小池君と色々議論ひたら、いくつかよさそうなアイディアがでてきました。以下は、そういうわけで私のオリジナルアイディアというわけではなくて、議論の中でででてきたものです。

これまでみてきたところでは、

$$\text{HPL の性能} = \frac{\text{演算数}}{(\text{DGEMM 時間}) + (\text{それ以外時間})} \tag{13.1}$$

という式で物事を考えてきました。そうすると、計算時間の内訳はこんな感じです。

```
    GDR DGEMM 時間
      計算時間
      計算時間に隠蔽できない通信時間
        K=2048 の時の時間
K<=1024 の時の時間
    GDR DGEMM 以外
      ホストでの DGEMM
      ホストでの、行交換、ピボット探索等
```

Core i7 で 12GB のメモリの時に、 GDR DGEMM 時間は GDR の枚数、速度によりますが 20 から 50 秒になります。これに対して、ホストでの DGEMM は k をどこまでするかに依存しますが 5-10 秒、それ以外の残りが 5-7 秒なので、例えば GDR での計算時間が 20 秒になっても、残りで 20 秒近くかかっては効率が半分で、たとえ全体が 40 秒なら実効 700Gflops 程度となり、逆に実行 1 Tflops にするためには全体を 28 秒にする必要があり、 DGEMM のところを 10 秒程度、つまり 3TF 程度必要、ということになってしまいます。

しかし、 GRAPE での伝統的なチューニングでは、ここまできた時の定石があります。それは、「ホストでの計算と GRAPE での計算をオーバーラップさせる」というものです。上の場合では、DGEMM の計算をしている間に他の処理を、というわけです。

ブロック化したアルゴリズムでは

```
    縦ブロック変形    (PUPDATE)
```

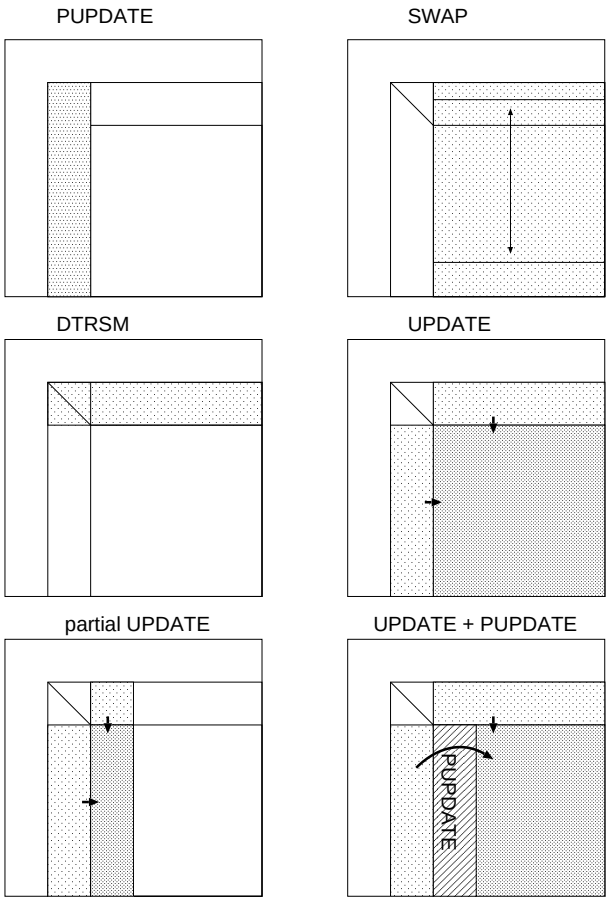


Figure 13.1: 普通の UPDATE と、 PUPDATE と UPDATE の並列実行

残り部分の行交換 (SWAP)
右上の変形 (DTRSM)
残り全体の変形 (UPDATE)

という順番で計算が進むのですが、次の縦ブロック変形は別に UPDATE 全部が終わるまで待つ必要はなくて、縦ブロックの変形をするのに必要なところまで UPDATE が終わっていればよいつまり、ブロックの行数分がおわっていれば十分です。さらにいえば、縦ブロックの変形自体が左から進んでいくので、これが UPDATE 処理を追い越してしまわなければよいわけで原理的には DGEMM が 1 行終われば PUPDATE を開始できます。

もっとも、 PUPDATE 自体で、ブロックサイズが大きいところでは GDR DGEMM を使うので、この時には並行実行はできません。なので、これらをスレッド実行するなら、PUPDATE で GDR を使いたくなったら、UPDATE を中断して PUPDATE の処理をして、それが終わったら UPDATE を再開、という少し面倒な処理が必要になります。が、これは時間とか決めうちでもよいので、プログラムとしてはそれほど複雑ではありません。

GDR での DGEMM の場合には、 GRAPE の通常の処理と違って計算中にずーっと通信もしており、そのために CPU コアも使っています。これは、 DMA read よりも write combined 領域への PIO write のほうがずっと高速だから、という分と、行列の引き算自体に CPU を使うというのがあります。Core i7 の場合カード 1 枚なら 2 コア使えば性能は十分なので、残りの 2 コアは使えます。これでは、 DGEMM のピーク性能は半分になりますが、基本的に k が非常に小さいところの話な

のでそもそも DGEMM の性能があまり高くないところが時間の大半を占めており、2 コアになることによる時間の増加はあるにしてもそれほど大きなものではありません。

もっと問題なのは、主記憶のアクセス競合ですが、これはやってみないとわかりません。アクセス競合については原理的には column-major のほうが有利ですが、ブロックサイズが 2048 もあれば row-major でもそれほど問題にならないといいなあ、という感じです。

スレッドで書くとこれは大変そうですが、 omp parallel sections での並行実行が有効に使えそうです。

原理的には、この方法により GRAPE-DR を使わないホストでの計算は全て GRAPE-DR を使う計算と並行に行うことができます。なので、理想的には時間の内訳は以下のようになります。

```
    GDR DGEMM 時間
      計算時間
      計算時間に隠蔽できない通信時間
        K=2048 の時の時間
K<=1024 の時の時間
```

K=2048 の時の隠蔽できない通信は、行列の下半分を送るものであり N=34k なら 5GB 分です。もっと小さい場合は、 K=1024 で 1.25GB、 K=512 で 1.875GB、 K=256 で 2.1875 で、合計 10GB です。ちょっと楽観的ですが通信に 2GB/s でるなら 5 秒ですから、例えば 1Tflops の実行性能をだすには計算が 23 秒でよいことになり、DGEMM の性能が 1.2Tflops あればよい、ということになります。

このように、ホストでの計算を隠蔽できることにはもうひとつ大きな意味があり、それは複数カードでも性能がでる可能性がある、ということです。ホストが計算していて GDR が休んでいる時間がたとえ 15 秒あれば、 GDR の計算が 30 秒から速度が倍の 15 秒になっても、全体の計算速度は 1.5 倍にしかなりません。しかし、GDR が休んでいる時間がなければ、これはもともと 30 秒だったものがそのまま半分の 15 秒になります。

普通は、通信速度がリミットになってそういううまい話はないのですが、GRAPE-DR の場合には通信速度をリミットしているのは GDR カードの PICe インターフェース自体であり、そのためにカードが 2 枚あると通信は基本的に 2 倍速くなります。隠蔽できない通信の部分も、ほとんどが 2 倍高速化されます。このため、大きな性能向上が期待できます。

Chapter 14

HPL 書き直しその5 (2009/9/16 書きかけ)

ここでは、並列化について考えます。HPL と同様なサイクリック分割をして、分割の単位としてはとりあえず NB=2048 を想定します。但し、通信を考えた時にこれがベストなサイズかどうか、つまり、縦パネルのサイズと通信の単位は同じがよいのかどうかは検討の余地があります。

基本的な手順としては、

1. PUPDATE
2. SWAP
3. DTRSM
4. UPDATE

なのは変わらないのですが、問題は何をどこまで並列にできるかです。

一つのキーになるのは、前のパネルの update から次のパネルの update までにどれだけ空き時間がはいってしまうか、ということです。次のパネルについての処理は

1. SWAP
2. DTRSM
3. UPDATE

なわけですから、まず UPDATE に必要な上側の正方形パネルについて考えます。

1. まず SWAP に必要なピボット配列を転送し
2. これと並行して逆行列作成を元パネル側で行ない、
3. できた逆行列を送り
4. 受け取った側では行列積をやって 1 パネル分だけ DTRSM をすませる

というのが最速でしょう。逆行列作成は 0.1 秒程度、転送や行列積は (転送は 1GB/s 程度を仮定すれば) ずっと短いので、その程度です。

問題は左パネルです。これは最大で 2k x 32k =500MB、平均でその半分のサイズがあります。転送は 16 x P(Q?) 回で、計算時間に対する通信時間の比は P や Q によりません。なので、 5GB 受け取って 20 秒計算、という感じの割合になり、かなり大きなオーバーヘッドになります。1GB/s でたとしても、平均で 0.3 秒です。

この部分をつめるには、PUPDATE をしながら、転送できるところは転送を始めればよいことはすぐにわかると思います。パネル分解自体には平均 1 秒程度の時間がかかるはずなので、オーバーラップできれば隠蔽できます。但し、例えば左半分の処理が終わったあとでも、計算の最後でこの部分にもう一度行交換をかける必要がありますから、仮にこの部分を先に送ったとすると、ピボット配列も送って行交換とスケーリングもする必要があります。また、この場合には対角パネルも送り先で処理することが必須になります。従って、手順としては、

1. (A)PUPDATE をしながら送れるところは隣に送る
2. (A) ピボット配列を転送し、受け取った側 (B) では 1 パネル分行交換をする
3. (A) 対角パネルを送る
4. (B) 逆行列を作り、1 パネル分 DTRSM する
5. (B) 行列積をする
6. (B)PUPDATE を始める

ということになります。いま一つよくわからないので、プロセッサについて、

a) 対角パネルをもつ
b) 対角パネルと同じ列
c) 対角パネルと同じ行
d) 対角パネルの隣
e) 次の対角パネルと同じ列 (対角パネルを含み、それより下)
f) 上のどれでもない

ケースについて、することを記述してみます。

a) 対角パネルをもつ時。

1. PUPDATE をしながら、できる左帯行列 (L) の送れるものは隣に送る (U を待ちながら 前の UPDATE もしている)
2. ピボット配列を送る
3. 対角パネルを送る
4. 残りの DTRSM をしながら、右上帯行列 (U) を下に放送する
5. UPDATE する

b) 対角パネルと同じ列

1. PUPDATE をしながら、L の送れるものは隣に送る
2. 対角パネルのプロセッサから U がくるのと並行して UPDATE する

c) 対角パネルと同じ行

1. ピボット配列を受け取り、行交換する
2. 対角パネルを受け取る
3. 逆行列を作り、しながら下に放送する
4. UPDATE する

d) 対角パネルの隣

1. ピボット配列を受け取り、1 パネル分だけ行交換する
2. 対角パネルを受け取り、逆行列を求める
3. 1 パネル分 DTRSM し、下に放送する
4. 残りの行交換をする
5. 残りの DTRSM をする
6. 残りの UPDATE をする

e) 次の対角パネルと同じ列 (対角パネルを含み、それより下)

1. L がくるのを受け取る
2. ピボット配列を受けて、1 パネル分行交換
3. 1 パネル分 U がくるのを待って、UPDATE
4. PUPDATE、L の転送、残りの U がくるのを待って UPDATE を並行に行う

f) 上のどれでもないつまり、対角パネルの隣の右、対角パネルの下

1. ピボット配列を受け取り、行交換する
2. L, U がくるのを待って、UPDATE する

今回の計算方法の通常の方式との違いは、UPDATE と PUPDATE の並列動作が前提になっていることで、これが暗黙に lookahead をしていることになります。

HPL では、「行交換と U 放送を混ぜる」という複雑な方式を採用しています。これのメリットは通信時間を短縮できることのはずです。例えば、 Long メソッドでは、まず交換前の U を放送し、それからそれぞれの行でどう交換されたかという情報をリングでぐるっと回して全部のところでつじつまがあうようにします。

ぐるっと回る情報の量は結局 U 全部と同じで、これは基本的に、ネットワークがトーラスで隣としかつながっていない、といった場合にどうやって改善するか、というアルゴリズムなので、必ずしも通信量が減るわけではないように思います。なので、今回はこの方法にこだわらず、単純な実装を使います。

1 ノード用のコードでは、再帰的パネル分解の途中で row major と column major を入れ替えることでそれなりの性能向上を実現できましたが、並列版ではどうするべきか、ということを考えてみます。まず、単純に考えると、行交換の速度は所詮通信速度にリミットされるのであまり意味がないようにも思えます。しかし、実際の計算を考えると、

- 交換行の受け取り
- 実際の行交換
- update のための U 行列の受け取り
- update のための L 行列の受け取り

が全て並行して進んでいる「かもしれない」わけで、メモリアクセスの負荷を大きく下げることができる row major はかなり魅力的です。もっとも、プロセッサグリッドが大きい場合には、 それぞれのプロセッサが受け取るデータ量があまり大きくなく、平均的にはあまり意味がないような気がします。

行交換の問題はもっと基本的なところにあります。基本的なアルゴリズムでは 1 行づつ交換していて、これはもちろん、原理的にはここに依存関係があるからです。つまり、例えば 2 行目と 10 行目を交換したあとに、3 行目と 10 行目を交換したら、元々の 2 行目が最終的には 3 行目に戻ってくるわけです。この時に、10 行目には 3 行目が入ります。つまり、 2, 3, 10 が 10, 2, 3 となるわけです。

もうちょっと違うケースとして、 2 行目と 3 行目を交換した後に 3 行目と 10 行目を交換したら、2, 3, 10 行目にはそれぞれ 3, 10, 2 行目が入ることになります。

もちろん、これは、まず移動元を全部バッファに移し、それからそのバッファから移動先に一括して移動するようにすれば済む話です。もっとも、これを本当にやると、少なくとも 1 ノードの場合、メモリアクセスが 4/3 倍に増えてしまうことになって面白くありません。

といっても自分でも良くわからないので、以下にアルゴリズムを書いてみます。

ピボット配列 pv はブロックサイズを K として、 0:K-1 の各行をどれと交換するべきか、がはいっています。これを使って、まず行き先の配列を作ることを考えます。これは、0:N-1 までが順番にはいった配列 id を作っておいて、それにピボット配列によって与えられた交換を適用すれば作成できます。(ここでは N は現在のブロックを含めた残りの行列のサイズで、元々の行列のサイズではないとします) ここで、 i 行目に j と書いてあれば、そこにデータをもってくればよいわけです。

まず、 k 行目以降については、行き先は必ず k-1 より前になります。これは、一度交換されたらもうそれ以降はピボット探索に入らないからです。なので、転送はピボットをもつプロセッサに送るだけでよいことがわかります。ピボットをもつプロセッサが持つ 0:k-1 行については、自分の中ですむところもあるしそうでないところもありますが、まあ、行き先に送る、というだけです。つまり、ピボットのところについては、どうせ殆どの行が入れ替わるので、

行き先の新しい配列を用意する
自分の中に行き先がある行列は、この新しい配列にコピー
それ以外は他のプロセッサに送る
他のプロセッサからくるデータを新しい配列にコピーする

が手順になり、それ以外は

ピボットのプロセッサに送るべきものを送る
ピボットから受け取ったものを格納する

というだけ、ということになります。両方のプロセッサでバッファがちゃんとあれば、順序がおかしくてもデータが壊れたりはしません。

まだ今一つイメージがわかないので、なにが並列に起こるか、という観点から、最初のステップから順番に書いてみます。

対角パネルプロセッサ　　対角の隣　　次の対角

```
1  最初の縦パネルの消去
1  L の送出                    L の GRAPE-DR への転送   L の GRAPE-DR への転送
2  ピボットの送出              ピボット受け取り          ピボット受け取り
2  対角パネル右に送出          1 パネル分交換、          1 パネル分交換
3  行交換 DTRSM       DTRSM、下に放送          1 パネル分アップデート
3  U 放送                      残りの DTRSM
4  アップデート                U 放送、アップデート      縦パネル消去開始
4                                                       残りアップデート
   対角の下
1  最初の縦パネルの消去
1  L の送出
2  ピボット送出
3  行交換
3  U 受け取り
4  アップデート
```

このように見ると、最初のパネル消去においては、結局 DTRSM をしないといけない分だけ対角パネルをもつプロセッサの仕事は同じ列のプロセッサより多く、同じ列でもその分下のプロセッサは遊んでいることがわかります。

次の対角パネルについて考えると、これは L を受け取ったあと、上から 1 パネル受け取れば次の対角パネルのアップデートができるので、これがすめば縦消去が始められます。但し、この時には残りのアップデートもあります。

計算のクリティカルパスがどこにあるか、ということを考えると、1 ノードの場合のように縦パネル消去とアップデートを並行処理するのがよいのか、それとも縦パネル消去だけを可能な限り早く終わらせるのがよいのか、という問題があります。縦パネル処理が終わっていれば、次の次の縦パネル処理を原理的には始められるからです。もちろん、縦パネル処理を極限まで速くしても隣ではまだ前のアップデートがおわっていないはずです。前のアップデートが終わる前に次のパネル消去を始める、というのは、GRAPE-DR 上のデータを再利用できないケースが発生するので望ましいことではありません。なので、縦パネル処理は極限まで速く終わる必要は実はなく、そこそこでよいわけです。

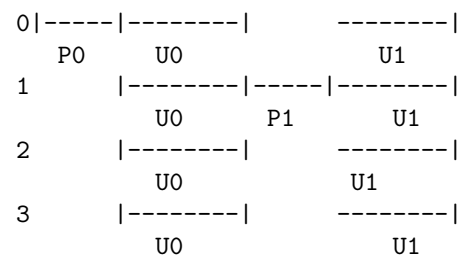
並行処理しないとどうなるかというと、対角パネルを含む列のアップデートは次の列よりもかなり遅れて終わることになります。

```
0|-----|--------|--------|--------|-----|--------|--------|--------|
   P0     U0       U1        U2     P4     U3       U4        U5
1      |-----|--------|--------|--------|--------|-----|--------|
         P1     U0       U1       U2       U3      P5     U4
2      |--------|-----|--------|--------|--------|--------|-----|--------|
          U0      P2     U1       U2       U3       U4       P6    U5
3      |--------|--------|-----|--------|--------|--------|--------|
          U0       U1      P3     U2       U3       U4        U5
```

この状況を示したのが上のタイムシーケンスです。ここでは、4 つプロセッサ列があって、順番にピボットパネルがくることを想定します (ブロックサイクリックにすれば当然こうなります)。そうすると、依存関係は

```
Pn -> Un
Pn -> Pn+1
```

だけなので、 遅れて終わることで穴があくわけではなく、また P が隠蔽されないことのインパクトはプロセッサ数が多いとあまり大きくないことがわかります。ちなみに、普通に Ui が終わった後で Pi+1 を始めてみると



とやたら空きができて全く話になりません。従って、実装の単純さを優先するなら、並列版では Pupdate と update の並列動作にこだわらず、

- Pupdate 自体の高効率化 (といっても、それほど重要ではない)
- update 自体の高効率化 (極めて重要)
- 違う列間でのこれらの非同期動作 (極めて重要)

を実現することが必要、ということになります。

この場合に、パネル消去と通信を並列にする必要は多分ありますが、これはあとからでも実装可能と思います。

さて、update です。GRAPE-DR の仕様や、今回の通信方式から、L は前のステップのうちに送られてきていることを想定します。

データ量として、最大 500MB、平均 250MB 程度であり、これが update をしている間、ないしは、前の列で縦パネル分解がおわってから、自分の update が終わるまでの間に転送できればよいので、これが秒程度の時間であることを考えると通信速度がそれほど速い必要はありません。

U については話ははるかに面倒です。というのは、ここでは

- 行交換
- DTRSM
- U 放送
- アップデート

が並行して進む必要があるからです。うーん、このことを考えると、行交換と放送を混ぜるアルゴリズムはやはり魅力的ですね。

行交換と放送を混ぜるなら、やることは MPI_SCATTERV と MPI_ALLGATHERV でできるわけで、 HPL に実装されている面倒な通信アルゴリズムを使うまでもない可能性が高いです。これは、これらを実質非同期通信として使って大丈夫かどうかに依存しますが。この場合には、全プロセッサで

- SCATTERV+ALLGATHERV
- DTRSM のあとアップデート

という同一の動作をすればよいことになり、話がだいぶ簡単になります。

通信量ですが、ピボットの行からは SCATTERV で U が全部でていきます。ALLGATHERV では全部はいってくるので、結局クリティカルパスになるのはピボット行のプロセッサのはずで、これは

並行動作しないで全部送り、全部受け取るわけです。データ量は最大 1 GB、平均 500MB です。計算時間は最大 5 秒で、計算時間は残りサイズの 2 乗に比例するのに対して通信は 1 乗という問題はありますが、最大の時に間に合っていれば効率低下は 33%、半分の時に間に合っていれば 10%程度になるので、半分の時に間に合う、というのが目安です。これは 500MB/s 程度の実効通信速度、ということになります。

パネル分解については、対角パネルとその下で同じように再帰をすることになり、再帰の最後のステップ以外は、基本的に普通の処理と同じなので特別なことはありません。左側の行交換についてはもう放送や DTRSM がいらないので、単純な処理をするべき、というのが違うところです。

最後のステップでは 2 列の処理の形にします。まず、ピボット探索で最大値を探索します。これは MPI_MAXLOC を使えば MPI_ALLREDUCE で一発で計算できます。それからこれをピボット行と交換し、スケールし、放送、という手順ですが、通信のステップ数を減らすためにはもうちょっと考えておかないといけません。

列の数だけ以下を繰り返す
    MPI_ALLREDUCE（最大値とそれがあるプロセッサがわかる）
    問題のプロセッサでその行をスケール
    その行を放送
    問題のプロセッサが対角パネルでなければ、対角パネルから交換される
    べき行を受け取る
    対角パネル、および最大値のソースプロセッサでは、行を更新
    右側（があれば）を update

この部分の通信がボトルネックにならないか、というのが問題ですが、1 行あたりの処理にミリ秒程度のはずなので、 ALLREDUCE と SEND/RECEIVE のレイテンシがあわせて 20 マイクロ秒程度なら問題ないはずで、 IB ならさすがにもうちょっと小さいと思うのでおそらく致命的な問題にはならないはずです。但し、かなりクリティカルであることはわかると思います。

では、各プロセッサのプログラムの構造をもう一度整理します。基本的には、以下のようなものであればよいはずです。

処理するパネルが残っていたら以下を繰り返す。
    自分がピボット列であれば列パネル消去

アップデート処理

で、列パネル消去は

最初のパネルでなければ
    左から L をもらう（通信はずっと前に起動している）
    1 パネル分について update 処理・送れる L は送る
本当の列パネル処理

で、本当の列パネル処理は再帰で、アップデート処理は

L がまだきてなければくるまで待つ
次の L を待ち始める
適当にブロック化して、
    行交換
    DTRSM

UPDATE
を並行して行う

です。ここで一つ問題なのは、次の L を待つのと他の処理をスレッドで並行処理した時にコアの数が足りなくなったりしないか、ということですが、これはやってみないとわかりません。

さて、あとは、サイクリック・ブロッキングをどうプログラムで表現するかです。すぐにわかることは、ピボット処理以外では、「自分の左側にもデータを送る必要がある」ということ以外に特にサイクリックだからといって考える必要はない、ということです。

単純に、 4x4 にブロックした行列を 2x2 のプロセッサグリッドで処理することを考えてみます。もとの行列が

```
A00 A01 A02 A03
A10 A11 A12 A13
A20 A21 A22 A23
A30 A31 A32 A33
```

で、これを

```
A00 A02   A01 A03
A20 A22   A21 A23

A10 A12   A11 A13
A30 A32   A31 A33
```

というふうに格納するわけです。そうすると、まず Ax0 列の処理では、 A20 と A10 が入れ替わっていますがそれを処理するプログラムが意識する必要はありません。

また、その後の swap, update についても、別に意識するところはありません。問題は、 Ax1 列の処理にはいるところです。行列は

```
A22   A21 A23

A12   A11 A13
A32   A31 A33
```

という状態になっています。 Ax1 列の処理では、通常の場合と同じですが、

- 自分より上のプロセッサにもデータが残っていること
- 但し、それらは自分および下のプロセッサより 1 ブロック少ないこと

を意識する必要があります。行入れ替え等の判定のためには、(プロセッサ番号, 行位置) や (プロセッサ番号, 列位置) とグローバルな行位置、列位置の相互変換の関数ないしマクロを作っておいて基本的にはグローバルな側で処理するのが安全そうです。とはいえ、アップデートやスワップ処理では、結局ローカルに考えたプロセッサ内部ではサイクリックブロッキングを意識するところは特になく、意識する必要があるのは次のブロックがどこか、というのの判定だけのように思われます。

プロセッサの数が p^2 でない時はどうでしょう？もっとも簡単なケースとして 2 プロセッサで縦にわけてみます。

```
A00 A02   A01 A03
A20 A22   A21 A23
A10 A12   A11 A13
A30 A32   A31 A33
```

この場合には、

```
    A22   A21 A23
    A12   A11 A13
    A32   A31 A33


    A22       A23

    A32       A33
```

と、穴があいてしまいそうです、、、あれ、これはおかしいですね。

```
A00 A02    A01 A03
A10 A12    A11 A13
A20 A22    A21 A23
A30 A32    A31 A33
```

と、中のブロックを

```
1  2
```

の形にすれば

```
    A12    A11 A13
    A22    A21 A23
    A32    A31 A33


    A22        A23
    A32        A33
```

となって穴があくことはない、ということがわかります。割り当ての一般式を導出しておきましょう。
行列サイズを N、 ブロックサイズを K、 プロセッサの行、列数をそれぞれ P, Q とします。

i 行は、ブロック i/K の i%K 行にあります。
逆に、 ブロック l の i' 行は l*K+i' 行です。
ブロック l はプロセッサ行 l%P の l/P 個めのブロックになります
逆に、プロセッサ行 p のブロック l' は、p+Pl' ブロックです。

という感じですね。

この辺から 10/9 です。で、ぼちぼち並列版を書いています。まず、ブロック化しないものを書いて、それからブロック化し、それから最終的に縦パネル再帰をやります。

アルゴリズムとして意外に面倒な感じがするのは後退代入のところになります。ベクトル b (途中までは a の最終列として処理される) は下から順番に処理されて、その処理は下の要素全部からなるベクトルと a の 1 行の積になります。

まず、ブロック化しない場合を考えてみます。内積型の処理であれば、 a は分散してもたれているとしても、 b は小さいので放送するとします。そうすると、b の要素が 1 つ求まる度にそれを放送して、次の要素の行をもつプロセッサで内積を計算、合計もして、引き算もすればいい、ということになるでしょう。

SAXPY 型の処理にすると、 求まった b を縦に放送した後で、対応する a の列と掛けて引き算、となります。これを a をもっているプロセッサ列でやれば、1 行の処理毎には横方向の通信が発生しない、というのはメリットであるはずです。ブロックが切り替わる時には b を全部放送するのが簡単でしょう。

なお、b を縦に放送するのも、もっとも単純なアルゴリズムでは 1 要素毎にするわけですが、1 ブロックの要素が全部求まってから放送でもあまり問題はないはずです。遅延は発生しますが、通信回数を劇的に減らすことができます。

計算時間として問題かどうかをあらかじめ見つもっておきます。非同期とか難しいことを考えないと、メモリアクセスは行列全体ですが、これは縦方向のプロセッサしか使いません。例えば、 256 ノードの Core i7 クラスタで 16x16 のプロセッサグリッドの時に、メモリ総量は大体 3TB になるわけですが、16 ノードでの並列処理だと思うとメモリバンド幅は 300GB/s 程度しかなく、ベストでも 10 秒程度かかってしまいます。これに対して LU 分解自体は 500-1000 秒なのであまり無視できません。従って、もうちょっと並列度を横方向に増やす必要があります。

全ノードを使うようにするのは別に難しいことではなくて、 1 ブロック単位ではなくて、横方向のプロセッサ数だけのブロックをまとめて処理すればよいわけです。まとめるために、最初はやはり少数のプロセッサしか働かない処理がはいりますが、その分の計算量は小さいので常時 1 列しか動かない場合よりは大きく改善されます。

この方法では、基本的にはまだ処理がおわっていない行列のうち、下の 16(プロセスグリッドの横の数) x 16 の領域をまず処理します。これには、一番右下の処理をして、それからこれで求まった b を全ノードに放送し、同じ列のプロセッサでこの b の寄与を計算し、 b を左の列に移します。そうすると、あとは最右下のプロセッサの斜め上が自分の中の処理をして、下に放送して、という繰り返しになり、対角位置のプロセッサに対応する行の b が求まることになります。

通信は、主要項は b の横向きの通信 (放送) と、内積のやはり横方向の総和のはずです。これは、1 プロセッサの 1 行が 32k ワード = 256kb くらいで、これを 16 回とかの通信ですから数 MB となりあまり問題になるような量ではありません。レイテンシも、通信回数が非常に少ないため、大きな問題にはならないはずです。

この処理のブロックサイズは本当は GRAPE-DR の要請から決まるものよりもずっと小さいものであるべきですが、コーディングが煩雑になりそうなのでそこを変更するのはできれば避けたいかもしれません。

というわけで、ブロック化しない場合でも SAXPY 型の処理で書いてみるのがよさそうです。

とりあえず、メモリ配置はブロック化しているけどアルゴリズムは単純な BLAS1 レベルのものを書いてみた。

以下、デバッグ中のメモ

have_currnt_col の結果が間違っている。

Procid=1, i, nb, bid, npcol, procid = 0 2 0 1 0 1: hav_current_col = 0

p=2 なので processors in row = 2

have_currnt_col = 0 になるのはあっていて、行列の寸法とかのほうがおかしい。

というか、そもそも p, q の使いかたがおかしかった。修正する。

P=2、 Q=1 のときに、プロセッサグリッドは

```
0 1
```

であって

```
0
1
```

ではない。これは、 npcol が「コラムの数」と思うこと。

rank_in_row がおかしい。

ものすごく沢山のバグを直した。

答は、、、とりあえず、 nan ではないものがでるようになった

元行列

```
 0:   5.688e-01  9.527e-01  7.854e-01  3.638e-01  1.000e+00
 1:   5.050e-01  9.333e-01  2.201e-01  1.810e-01  1.000e+00

 0:   5.366e-02  8.510e-01  5.746e-01  8.587e-01  1.000e+00
 1:   6.317e-01  8.170e-01  3.353e-01  3.798e-01  1.000e+00
```

答

```
 0:   1.000e+00  1.293e+00  5.308e-01  6.013e-01  3.168e-01
 1:   2.622e+00  1.000e+00  2.228e+00  1.000e-01  8.658e-01
```

Printmat_MP: Proc 1, loc = 0 1

```
 0:  -7.510e-01 -4.168e-01  1.000e+00  2.241e-01 -2.017e-01
 1:   5.281e-02  7.692e-01 -1.177e+00  1.000e+00  4.217e-01
```

ちなみに、 q=1, p=2 ではまだ全然おかしい

検算をどうするか、ということも考える必要があります。ブロック化されていいて、また縦と横はブロックの周期が全然違うかもしれないので、 b を縦方向に放送して全部もたせるのが簡単なような気がします。それから、b の必要なところを切り出して、 a と乗算し、その結果を横方向に積算すればよいはずです。

```
make lu2_mpi ; mpirun -n 4 lu2_mpi -n 32 -q 4 -p 1
```

でも動いた。縦方向は一応大丈夫なはず？

q=3, n=3 も OK (n=6,12,18)

次は横方向

動いているような気がする

縦横: 2x2 なら動く

もうちょっと検証が必要な気もするが、ブロック化のほうを始めてみよう。

ブロック化。

まず、再帰にはしないのをする。基本的アルゴリズムは

```
    for(i=0;i<n;i+=m){
        column_decomposition(n, a, m, pv,i);

process_right_part(n,a,m,awork, pv,i,n+1);
    }
```

column_decomposition、process_right_part の中身は以下の通り。

- swaprows で前も入れ換えているはず
- scalrow でも前もスケールする
- pv を作る

という辺りに注意が必要。

```
 void column_decomposition(int n, double a[n][RDIM],  int m, int pv[], int i)
 {
     int  j, k;
     int ip,ii;
     double ainv;
     for(ip=0;ip<m;ip++){
 ii=i+ip;
 int p = findpivot(n,a,ii);
 if (fabs(a[p][ii]) > 2* fabs(a[ii][ii])){
     pv[ip]=p;
     swaprows(n,a,p,ii,i,i+m);
     nswap++;
 }else{
     pv[ip]=ii;
 }
 // normalize row ii
 ainv = 1.0/a[ii][ii];
 scalerow(n,a,ainv,ii,i,ii);
 scalerow(n,a,ainv,ii,ii+1,i+m);
 // subtract row ii from all lower rows
 vvmulandsub(n,  a, ii, ii+1, i+m, ii+1, n);
     }
 }



 void process_right_part(int n,
 double a[n][RDIM],
 int m,
 double awork[][n],
 int pv[],
 int i,
 int iend)
 {
     int ii;
     for(ii=i;ii<i+m;ii++){
```

```
swaprows_simple_with_scale(n,a,pv[ii-i],ii,i+m,iend,
   1.0/a[ii][ii] );
    }
    solve_triangle(n,a,m,awork, i,iend);
    mmmulandsub(n, a, i,i+m, i+m, iend, i+m, n);
}
```

global index で colum l1-l2 が与えられた時に、自分のインデックスで関係するのはどこからどこまでかを判定する関数が必要。

ブロックについて、 l1 のブロック番号、 l2-1 のブロック番号を判定して、その範囲内に自分のインデックスがひっかかるかどうか。ひっかかるならどこからどこまでか。

あるレンジの整数の中に p でわって q あまる数があるか？

まあ、実際には一般の場合が必要なわけではないので、、、あ、でも、update の時には起こるのか。

bl1 = bl2 なら判定しておしまい

bl1 != bl2 の時: bl1 がかかっていればかかっている

`bl2` がかかっていればかかっている

そうでないとき: `bl1` のあとの最初の自分のブロックが `bl2` より手前ならば
かかっている

こんな感じかな。

```
00001111222233330000111122223333000011112222333300001111222233330
0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
                X                   Y
```

X: 18, Y: 38

```
    C1    C2
```

0: 6 11 1: 4 10 2: 4 7 3: 4 7

ブロック化の基本はできた。但し、コードはまだ行列積が見えてないので、見える形に変更必要。

現在のコード: 単純に 1 行更新のループを回しているだけ。

これをまず、 nb だけ更新と下も更新に分離。これをやって行列積をだすのはできた。

$$C- = LU \tag{14.1}$$

の実現方法について、今回は、 U はそもそも

$$U = D^{-1} * U_0 \tag{14.2}$$

の形

Chapter 15

HPL 書き直しその5 (2009/10/3 書きかけ)

ここでは、前の節の議論から、 HPL にちょっとだけ手をいれて速くする、という方針を検討します。
まず、普通にやると

- 行交換
- DTRSM
- U 放送
- アップデート

が並行してむ必要がある、というふうに書きましたが、良く考えるともうちょっと簡単にできます。我々の方法では、 DTRSM を DGEMM で置き換えます。つまり、行交換と放送でできた U に、対角部分の逆行列 P^{-1} を左から掛けて、それと左から放送されてきた L を掛ける、つまり、 $L(P^{-1}U)$ という計算をするわけです。行列乗算には結合則がなりたつので、これは¡($L(Pˆ{-1})U$¿ という順序でやる、つまり、 L と P^{-1} を先に掛けてもよいわけです。この場合は DTRSM はここでしなくてもよいので、話がだいぶ簡単になります。