Linux Japan Vol.8への掲載原稿です.
26char/46line
#10203040506070809101112131415161718192021222324252627282930

「Linux/Alphaによる数値計算ならびにRISCアーキテクチャのおはなし(第三回)」
-- 21164Aの高性能内部アーキテクチャ その二
清水尚彦 nshimizu@et.u-tokai.ac.jp

LINUX JAPANの読者には当たり前の話ですが
ソフトの第一次の設計資料はソースコードです.
ソースが入手できないソフトを使っているとベンダーのメンテナンスが切れた段階で他がいくら調子が良くても捨てざるをえない事態に追い込まれる可能性があります.
某M社では比較的簡単に古いOSのサポートを打ち切ると宣言していますし,
他のベンダーも似たりよったりです.

例えばうちの奥さんはjxword太郎からのジャストシステムの愛用者ですが, 一太郎はV4.3を使い続けています.
しかしもし一太郎やPC-DOSに不備がでたり不満があるとき(実はたくさんあるようです)にもすでに代替品は存在していません. 結局不満や不備を抱えたまま致命傷になるようなバグがないことを祈ることしかできないのです.
仕事のツールとして依存しているのにこういった状況に置かれるのは私には耐えがたいものがあります.

Linuxをはじめとする各種のフリーソフトは設計の一次資料であるソースコードを開示して配布しています. そこで, 他に誰も使わなくなったとしても不具合があれば最低限自分で何とかする方法も残されるという素晴らしい保険が得られるのです.
さらに自分でできなくてもお金を払ってコンサルタントを雇う道だって考えられるのです.

ベンダーのいいなりになるというのは何もソフトだけの話ではないのです.
我が家ではリースバックの中古を買ったC社のビジネス用のコピー機をもう7年も使っています. さすがにこれだけの年月使うと画質も落ちたし紙送りも失敗することの方が多くなってきたりしてメンテナンスが必要になります.
通常はメンテナンス契約を結んでメンテナンスをしてもらうのでしょうが,
私の機械は中古でメンテナンス契約を断られたので自分でメンテナンスをはじめました.
自分の機械として長く使うためには設計資料(サービスマニュアル)がないとメーカーの言いなりにならざるを得なくなりますが,
日本ではサービスこそ企業収益の要とばかりにマニュアルもパーツも普通には手に入らないので高いお金を出してサービスを受けることのできる機械しか事実上利用できないことになります.
(この機械もアメリカではマニュアルはともかくパーツは入手できるので変な話です)

私の場合たまたま

http://www.copier.net/

で現象を説明したら丁寧に交換や清掃するべき部品を指摘してもらえたので
(パーツが入手できないので交換はできませんが)
動作状況を確認しながら調整することで何とか正常動作になりました.

(私のコピー機の場合はともかくとして)ハードにしてもソフトにしても
ベンダーがサポートをしないのであればそれまでのユーザーに対しては
自助努力が可能なようにできる限りの
資料をオープンにしてもらいたいものですね.

さて, Alphaチップを乗せたマシンは性能が売りものですが実は
これ以外にも64ビットのメリットがあります.
21064Aや21164PCなどの低価格で性能が低いプロセッサであっても64ビットのメリットは享受できます.
商用のソフトウェアで有名なところではオラクルのデータベースソフトがDECのOSでVLM(Very Large Memory)というメモリ上にデータベースを展開して高速化する方式を採り入れています.
また, 64ビットあれば最近の大型の磁気ディスク上に大きなファイルを作っても全部のデータをメモリにマッピングできるのです. 9GBのデ

ータファイルをマッピングして普通の命令でデータの操作ができれば素晴らしいですよね．32ビットのプロセッサだと最大のアドレス空間が4GB止まりなので大きな磁気ディスクに書かれた大きなファイルを64ビットプロセッサ程簡単には使えません．もっともマッピングした時にプロセスの空間が広がってスワップが必要になるのかもしれませんから話半分に読んでください :-p

Alphaを載せたマシンは低価格でもかなりの性能を持っているので数値計算には大変コストパフォーマンスがよいと思っていますが，他社が次々とワークステーションのマルチプロセッサを出しているのにAlphaのマルチプロセッサはサーバータイプのみという時代が続いていました．
性能の低いマシンをマルチプロセッサにしても得られる性能はたかが知れていますが高性能なプロセッサこそマルチプロセッサで他では得られない性能を提供するべきだと私は思っているのでこの製品戦略は納得がいきませんでした．
もちろんDECにも言い分はあると思います．考えられる戦略上の話としては，マルチプロセッサの性能は単体性能が高くなればなるほどメモリ性能によって律速されてしまうのでワークステーションで高性能なマルチプロセッサマシンを作るのは困難だということが考えられます．でも，アプリケーションによってはキャッシュミスが少なくプロセッサ台数に比例した性能が出しやすいものもあり，選択肢としてマルチプロセッサマシンがないというのはつまらないものです．ある日電子ニュースを見ていたらマルチプロセッサのAlphaマシンについて書いてある記事がありました．Alphaの世界にもようやく待ちに待ったマルチプロセッサのワークステーションが出るようです．

http://www.workstation.digital.com/announce/uw_an.html

このマシンは2つのプロセッサのそれぞれ独立して4MBの外部キャッシュを持つぜいたくな構成を取っており性能的にもSPECfp95が30を越えるなど期待の持てる内容になっています．
日本DECの営業の方に聞いた限りではまだまだ私など手が届くような値段ではないのでとりあえずは私には関係ない代物ですが，DECがようやくワークステーションのマルチプロセッサ市場に本腰を入れたという気がするので今後の展開が楽しみです．

楽しみといえばAMDがDECの21264のバスを使ってx86の高性能プロセッサを作るという話が伝わっています．これが実現するとAMD用のマザーボードを作るボードメーカーは必ず現れるだろうから21164Aまでの時代と異なり市販のボードを買ってBIOSだけAlpha用に変更すればAlphaのPCが入手できるという時代になるかもしれません．(もちろん誰がBIOSを書くのかという問題が残っていますしAlphaではEEPROMの容量が余分に必要になるので簡単にはいかないことも考えられます)
さらに同一バスを使うのであればAMDのx86とAlphaを混在搭載してマルチアーキテクチャの機械などできるともっと楽しいかもしれません．
Linux/MA(Multi Architecture)なんてプロジェクトを立ち上げてプログラム毎に適切なプロセッサに割り振って動作するマシンを作るなんて考えただけでワクワクしませんか?

さて今回は前回に引続き21164の内部アーキテクチャのおはなしからはじめます．
それと，Alphaアーキテクチャと数値計算の続きならびに
編集部からRedHat 5.0をお借りして試してみたのでその話も少し書きます．
誌面の都合で(いい言葉ですね)Octaveの話は今回はお休みします．

1．21164内部アーキテクチャにおける工夫(その2)

さて，前回誌面の関係で書き切れなかった21164Aの内部アーキテクチャのおはなしの続きから始めたいと思います．
今回は21164の特徴の中でも私が特に気に入っているミスアドレスファイルについて詳しく書きたいと思います．この機能はあまり耳にしたことのない人も多いかと思います．80年代のRISCアーキテクチャの多くはキャッシュヒット時にのみ性能を出せるような仕組みだったのですが実際のアプリケーションでは設計者が考える以上にキャッシュミスは多発していて性能上の大きな隘路になっていました．そのことに対する見直しがかかったからか90年代のアーキテクチャではメモリモデルの見直しのみならずメモリの読みだし遅延が性能を低下させることをなんとか防ごうという努力の後が見られます．

1) ライトバッファ

Alphaの命令ではメモリへ書き込みを行うストア命令の最大書き込み単位は8バイトです.
これは浮動小数点の倍語長と整数のlongに対応しています.
パイプラインの項で説明したようにストアは1クロックに一つ
ずつ発行することができますが, 実は2次キャッシュは1クロックに16バ
イトずつ書き込みもしくは読み出しができるのでバッファなんかなくても間に
合いそうに見えますね.
ところが2次キャッシュのミスが発生して主記憶もしくは3次キャッシュへの転
送要求が出る時には読み出し/書き込みの時間がかかるため転送性能は大きく
低下します.
一方, 多くのプログラムでは連続したストアは連続もしくは
近傍のアドレスにデータを書き込むことが多いのです.
そこで, 近傍のストアを一旦バッファにまとめて適当なきっかけでバッファの
内容を2次キャッシュに書き出すようにすればキャッシュメモリの
待ち時間は短くなって性能が上がるという訳です.
そのための仕組みがライトバッファになります. Alphaでは6個のライトバッフ
ァがあるのですが, それぞれは32バイトのバッファを持っています.
あるバッファにデータが書き込まれたらその32バイトの境界内のデータへの後
続のストアは同じバッファに書き込まれるので6個のバッファで8バイトのデータを
最大24個置いておけます.

2) ミスアドレスファイル

キャッシュミスを起こした場合にはプロセッサはキャッシュにそのデータが
到着するまではデータを使えません. メモリにデータを要求してからそのデー
タが使えるようになるまでの時間をメモリレイテンシと呼んでいますが, この
間にいかにプロセッサを止めずに処理を続けられるかが性能の向上の鍵となり
ます.
キャッシュミスを起こした場合にはそのデータを使わなくても
処理できる演算が他にあればデータを待っているより先にその演算をやった方が
性能が上がることは分かりますね. この仕組みを一般化してどんな命令でも
できるものからやるという仕組みを順不同実行もしくはアウトオブオーダー実行と
呼んでいて多くの最近のプロセッサで採用されています.

ところがAlphaの21164まではこの仕組みは導入されていません. それでは
キャッシュミスを起こした時には待っているしかないのでしょうか?
実は21164ではミスアドレスファイルという巧妙な仕組みでキャッシュミスに対しては
アウトオブオーダーと同等の効果を得ています.
21164以前にも同様の狙いを持った機構としてヒットアンダーミスと呼ばれる仕組みが
ありましたが, これはキャッシュミスの後のキャッシュヒットのロード命令は
先に実行できるというものでした. ヒットアンダーミスでは二回目の
キャッシュミスが起きるとそこで止まってしまいましたが, 21164ではミスア
ドレスファイルに記録できる限りのキャッシュミスは後に続く命令の実行を
妨げないように作られています. もちろん, 後に続く命令がミスを起こしたロ
ード命令の結果を必要とする場合には原理的に先に進めないので止まってしま
いますが, ロード命令だけ早めにまとめて発行しておけばキャッシュミスによ
る実行の停止は最低限に抑えることができます.
この仕組みは, 21164で私がもっとも気に入っている特徴です.

ミスアドレスファイルはキャッシュミスによって命令の実行が止まる
ロード命令だけに対して順不同実行をサポートするための仕組みです.
21164/21164Aにはそれぞれが32バイトのキャッシュのラインに対応する
6個のミスアドレスファイルが備えられています.
ミスアドレスファイルのうちの5個はそれぞれ4個のロード命令の要求を
受け付けます. また, 残りの1個は1個だけ受け付けるので最大21命令の
ロード命令が順不同で実行できることになります.
ミスアドレスファイルが記憶するアドレスはそれぞれ1個だけで, 一つの
ミスアドレスファイルは同じ32バイト境界の中の4つのロード命令を記憶する
ようになっています. そこで
バラバラのアドレスのロード命令が21個扱えるわけではないのですが
数値計算では連続したベクトル要素などを大量に読みだしたりするので
この仕組みはたいへん便利です.

例えばつぎの様なFORTRANのプログラムを考えてみましょう.

```
do i=1,n
  a = a + x(i)*y(i)
enddo
```

このプログラムではx(i),y(i)のベクトル要素を次々に読みだして来ます．例えば
このベクトルがちょうどキャッシュのラインの先頭から始まっていたと考えま
しょう．するとx(1)がキャッシュミスを起こしているならx(2),x(3),x(4)もや
っぱりキャッシュミスになってしまうでしょう． y(i)も同じです．
前回やったアンローリングのテクニックをつかってループを展開すると

```
do i=1,n,4
  a = a + x(i)*y(i)
  a = a + x(i+1)*y(i+1)
  a = a + x(i+2)*y(i+2)
  a = a + x(i+3)*y(i+3)
enddo
```

となります．最適化コンパイラでは大抵ロード命令を先に実行するような命令
コードを出すのでx(i)からy(i+3)までのベクトルの読み出しがまず行われま
す．これを順番に実行するとx(i)でキャッシュミスが起きるとx(i+1)もやっぱ
りキャッシュミスになることが多いのでヒットアンダーミスではこの時点で止
まって最初のキャッシュミスのデータが到着するのを待つことになります．と
ころが21164ではx(i)とx(i+1)がキャッシュミスを起こしてもこれをミスアド
レスファイルに記録するだけでつぎの命令に進みますので，x(i+2)のロード命
令が発行できます．このようにキャッシュミスが起きても次々と命令発行をす
ることができるので一連のロード命令が発行し終ったころに最初のロード命令
のデータを使いにいくことができます．前に書いたように21164の2次キャッシュ
は8クロックでデータを取ってこられるし，パイプライン構造になっていて次々と
データの要求を発行できるので31個しか浮動小数点レジスタのない21164でも
命令の並びを工夫することで2次キャッシュに入っているデータについては命
令の実行を止めることなく利用できるようになります．
そこでちょっと工夫してプログラムを作ると遅延なしで読み出せるキャッシュを
96KBも持っていることになると言えます．

3) プリフェッチ
後で使うデータをあらかじめキャッシュまで持ってくることをプリフェッチと呼びます．
Alphaのアーキテクチャにはデータをプリフェッチするための専用の命令が
備えられています．それはFETCH, FETCHMという2つの命令ですが，命令の動作
は実装依存と定義されていて普通に我々が入手できるマシンでは何もしていな
いようです．この命令が発行された時プロセッサ自身はプロセッサの外に特別
の要求を発行するのですが，システムの制御チップセットがこの要求を扱うこ
とができなければ何も起こらないのです．クレイのT3Eなどはこれらの命令を
特別な用途に使っていると思われます．

さて，FETCH, FETCHMはアーキテクチャ上に定義された命令だった訳ですが
実装依存としてしまったためにコンパラは振舞を予想できなくなってしまい
かえって使いにくい命令になってしまいました．

21164以降のプロセッサではこれらの命令に加えてもう二つのプリフェッチ命
令がアーキテクチャに加えられました．
といっても命令コードが増えたわけではなく常に0である2つのレジスタへのロ
ード命令をこの目的に使用しています．

・固定小数点レジスタ $31 へのロード命令：この命令はデータを一次キャッ
シュまで持ってくるプリフェッチとして定義されています．

・浮動小数点レジスタ $f31 へのロード命令：この命令も同様に一次キャッシ
ュまでデータをロードしますが，固定小数点レジスタへのロード命令と異なり
もし一次キャッシュが使用中であればロードをあきらめて二次キャッシュまで
取ってきたところで命令の動作を止めます．

一次キャッシュがいつ使われるかは予測できないのですが十分な性能を出した
い時には前出ミスアドレスファイルによってメモリレイテンシを低減する対策
と併せて浮動小数点レジスタへのロード命令をプリフェッチとして使うべきでしょう．
というのは前に書いたように二次キャッシュまでデータが到着していればミス
アドレスファイルの働きで命令の停止をせずに計算できるようなプログラムは
簡単に書けるからです．
無理に一次キャッシュにデータを移動して動作中のプログラムの邪魔をするこ
とはないでしょう?

これらの工夫によって21164はプログラムのチューニングしだいで
たいへん高い性能を発揮するF1カーのようなプロセッサになりました.

次回は外部キャッシュやバスなどプロセッサの外回りの話をしたいと思います.

2. Alphaアーキテクチャと数値計算(その2)

前回はループアンローリングによって命令のレイテンシの影響を低減すると
いう話をしましたが, 単純にアンローリングしただけでは隠蔽できない
レイテンシもありました.
今回はそんな場合に有効なソフトウェアパイプラインや変数の変換の話をしたいと
思います.
前回の話を読んでいない方もいるでしょうから復習も兼ねて
内積演算の例題でアンローリングの説明をします.

```
subroutine diprd(n, dx, dy)
  real*8 dx(*), dy(*), z
  integer i,n
  z=0.0
  do i=1,n
   z=z+dx(i)*dy(i)
  enddo
  return
  end
```

アセンブラだと分かりにくいのでソースコードで示すとアンローリングは
次のようになります(ここでは話を簡単にするために端数の処理を除いて
ありますのでこのままでは特殊ケースでしか使えません, ご注意下さい).

```
subroutine diprd_ul(n, dx, dy)
  real*8 dx(*), dy(*), z
  integer i,n
  z=0.0
  do i=1,n,4
   z=z+dx(i)*dy(i)
   z=z+dx(i+1)*dy(i+1)
   z=z+dx(i+2)*dy(i+2)
   z=z+dx(i+3)*dy(i+3)
  enddo
  return
  end
```

アンローリングによって性能が上がるのは 1)分岐の回数が減るので
分岐命令の数だけ性能が上がる 2)書かれた通りに実行しなくても計算結果に差が出ない
処理が増えるので他の命令の実行を待つ間に別の命令を実行するようなコードを
出すことが可能となるということによります.

ところが書かれた通りに実行しなくても結果に差が出ないかどうかは
一文ずつ検証するので上の例の変数zのように前の計算結果を使うような
書き方をするとこの部分の命令の入れ換えは難しくなります.
コンパイラから見るとdoループの中の文はすべて前のzの値を使うように見えます.
ではzの変数の扱いを少し変えてみたらどうでしょうか?

```
subroutine diprd_ul(n, dx, dy)
  real*8 dx(*), dy(*), z,z1,z2,z3
  integer i,n
  z=0.0
  z1=0.0
  z2=0.0
  z3=0.0
  do i=1,n,4
   z=z+dx(i)*dy(i)
   z1=z1+dx(i+1)*dy(i+1)
   z2=z2+dx(i+2)*dy(i+2)
   z3=z3+dx(i+3)*dy(i+3)
  enddo
  z=z+z1+z2+z3
  return
```

```
  end
```

どうでしょうコンパイラになったつもりでプログラムを眺めてください.
こんどはdoループの中はお互いに全然関係のない文が並んでいます.
この4つの文はどれを先に実行しても結果には全く影響がでないのです.
プログラムの中にこのような自由度の高い部分があるとコンパイラは
命令を並べ直すことも簡単にできます.

さて, たびたび述べているように最近の高クロックプロセッサの性能を
制限する一番の原因はプロセッサとメモリの性能の差です. これは
つまりキャッシュミスが起きた時に大きく性能が低下するという現象に
見えます.
Alphaの特に21164以降のプロセッサに備えられたミスアドレスファイルは
キャッシュミスが起きてもとりあえず命令を先に進めてくれますので実際に
キャッシュミスを起こした命令を使わない限りは性能も低下することはない
のですが, ロードしたデータは使うためにロードしているのですからずっと
使わない訳にはいきません.
どれだけ間を空けて使うべきかは実はデータがどこにあるかとかバスの利用
状況とか主記憶の利用率などによって変わります. でもプログラムを
そのように柔軟には作れないので結局できるだけロード命令と使う命令を
離して使うという対策しか取れません.
ところが並べ直そうと思っても元もとループの中にはロード命令が8個と
加算と乗算がそれぞれ4個ずつしかないので余り離しようがないですね.
ちょっと冗長ですがロード命令とその結果を使う命令(use命令と呼びます)
の関係を明示的に分かるようにFORTRANで書くと次のようになります.

```
subroutine diprd_ul2(n, dx, dy)
  real*8 dx(*), dy(*), z,z1,z2,z3
  real*8 dx0,dx1,dx2,dx3,dy0,dy1,dy2,dy3
  integer i,n
  z=0.0
  z1=0.0
  z2=0.0
  z3=0.0
  do i=1,n,4
   dx0 = dx(i)
   dy0 = dy(i)
   dx1 = dx(i+1)
   dy1 = dy(i+1)
   dx2 = dx(i+2)
   dy2 = dy(i+2)
   dx3 = dx(i+3)
   dy3 = dy(i+3)
   z=z+dx0*dy0
   z1=z1+dx1*dy1
   z2=z2+dx2*dy2
   z3=z3+dx3*dy3
  enddo
  z=z+z1+z2+z3
  return
  end
```

キャッシュミスの影響の大きさを考えるともっともっとロード命令とuse命令を
離したいと思ってもループの中の命令数が多くなければ難しいですね.
ところがここに面白いアイデアとしてアンローリングよりも大きな可能性を持った
ソフトウェアパイプラインというものがあります.
論より証拠
上のプログラムを元にソフトウェアパイプラインを組み込んでみましょう.
すでにアンローリングしたところからはじめるのでパイプライン化は
ロード命令だけを対象にします.

```
subroutine diprd_sp(n, dx, dy)
  real*8 dx(*), dy(*), z,z1,z2,z3
  real*8 dx0,dx1,dx2,dx3,dy0,dy1,dy2,dy3
  integer i,n
  z=0.0
  z1=0.0
  z2=0.0
```

```
  z3=0.0
  i = 1
  dx0 = dx(i)
  dy0 = dy(i)
  dx1 = dx(i+1)
  dy1 = dy(i+1)
  dx2 = dx(i+2)
  dy2 = dy(i+2)
  dx3 = dx(i+3)
  dy3 = dy(i+3)
  do i=1,n,4
   z=z+dx0*dy0
   dx0 = dx(i+4)
   dy0 = dy(i+4)
   z1=z1+dx1*dy1
   dx1 = dx(i+4+1)
   dy1 = dy(i+4+1)
   z2=z2+dx2*dy2
   dx2 = dx(i+4+2)
   dy2 = dy(i+4+2)
   z3=z3+dx3*dy3
   dx3 = dx(i+4+3)
   dy3 = dy(i+4+3)
  enddo
   z=z+dx0*dy0
   z1=z1+dx1*dy1
   z2=z2+dx2*dy2
   z3=z3+dx3*dy3
  z=z+z1+z2+z3
  return
  end
```

do ループの中を見てください.

```
   z=z+dx0*dy0
   dx0 = dx(i+4)
   dy0 = dy(i+4)
```

ちょっと分かりにくいですが内積の加算と乗算をするとすぐその後にその変数に
新しくメモリから配列データをロードするようなプログラムになっています.
そこでこのロードしたデータは do ループをぐるっと一周回ってから
次のループではじめて演算に使われることになります.
アンローリングではループの中だけで処理していたのに対してループを
跨いでデータの待ち合わせができるのでデータが到着するまでの待ち時間を
有効に使うことができます.

次回はキャッシュを有効に使うための方法を検討してみましょう.

3. ちょっとだけ RedHat 5.0 のおはなし

編集部から RedHat 5.0 のことを書いて欲しいとの依頼がありました. 普段なら
評価のためにインストールをやり直すなんてとてもできないのですが, たまた
ま部屋のマシンの共通バックアップのためにハードディスクを入手したところ
だったので簡単に OK の返事をしてしまいました.
ところがこのハードディスクが大失敗でどのマシンにつなげてもまともに動作
してくれないのです.
というわけでインストールができないのでまた今度の機会ということで..
と思ったのですが, せっかく CDROM まで送っていただいたので少しはやってみ
ようかと思い現在の環境を壊してインストールする事にしました.

私のマシンは AlphaPC64 というマザーボードを使っていてディスクは当時大枚
をはたいて 1GB という大容量 :-) の SCSI が接続されていますので 2 つの OS を
両方インストールしてブート時に切替えるなんて簡単なはずでした.

ところが, RedHat の標準の CDROM では NIS すら動きませんでした.
マニュアルには NIS のことは何も記載されていません.
ypbind を rpm からインストールし, NIS のドメインの設定をしても
まともに動かないではないですか! メイリングリストの方で RedHat のサイトか

ら修正差分を取ってきてはというアドバイスをいただき見に行ったところ
どうやらglibcを更新しないといけないようでした.
glibcをrpmを使って更新して(これも結構綱渡りで動作中のライブラリを
無理に更新するのでその後のコマンドはリブートするまでほとんどが
segmentation faultを起こしてしまいます)リブートしてypbindを動かすと
めでたく立ち上がるようになりました.

RedHatにはFORTRANコンパイラのg77は付いて来ませんでした.
その替わりにFORTRANからCへの変換コンパイラであるf2cが入っています.
RedHatに附属するf2cは INTEGERが 8バイトになっていて気持悪いのですが
一応LINPACKベンチマークを動作させる
ことはできました. そこで, StataboWare-1.2β のg77とRedHat 5.0のf2cで
LINPACKベンチマークを動作させて見ました.
StataboWareでは普通のFORTRANとして使えるf2cとg77が入っていますが,
RedHatのf2cを普通のFORTRANとして使いたい人はヘッダファイルに
必要な修正を加えてlibf2cを再構築する必要があります.
ただし, 今回の記事では時間が計れればOKなのでそのまま使います.

実行したのはLINPACK 100元と呼ばれるベンチマークです.
このベンチマークはDongarra reportという報告に主要な計算
機における性能指標が発表されますので得られた値を他の計算機と比較するの
に便利です.
短いプログラムのベンチマークなので数値計算を売りものにしているメーカーは
研究し尽くしていて大抵の商用コンパイラではLINPACK専用の特殊処理があり
普段のプログラムの性能を確認する役には立たないのが残念ではあります.
プログラム自体は密行列を係数行列とする連立一次方程式の解を
求めるものです. 表の数値の単位はMFLOPSですが
私のマシンはAlpha 21064Aという一世代古いチップなので
egcsなどの最適化の対象外の可能性が高く
得られた数値自身はあまり気にしないで下さい.

また, 参考データのDEC Cでの値はf2cの結果をコンパイルしたものですが,
timeコマンドでプログラムの実行速度を測定すると明らかに異なるのですが,
clock関数の分解能が悪いため -O 以上のオプションで結果に差が出ませんで
した. 問題が小さすぎるのですが, LINAPACK 100元は問題に手を入れてはいけ
ない決まりなのでそのままにしています.
もっと解像度の高いクロック関数を使うとさらに正確な比較ができます.
ここで, gccやegcsのチューニングのターゲットは21164なので私のマシンでは
必ずしも最適なコンパイル結果が得られるとは限らないのでこの結果で
DEC Cの方が優秀であるという結論にはすぐにはなりません.
(しかも商用のコンパイラはLINPACKベンチマークなどの有名なベンチマークの
ソースコードには特別な扱い(注)がある場合があるので余計割り引いて考える
必要があります.)

(注)特定のソースコードのパターンと一致した場合に専用の命令コードを出力
させるような工夫はコンパイラには簡単に入れられます. そこで, 簡単で著名
なベンチマークを商用コンパイラでコンパイルした結果は似たような振舞をす
る他のパターンのプログラムよりたいへん優れた結果を出す場合があります.
LINPACKではdaxpyというサブルーチンが実行時間の90%を占めることが分かっ
ていますのでDEC Cでもdaxpyを狙いうちでインラインの展開をしたりdaxpyの特定パター
ンのソースコードに対して手で並べた命令列を展開するようなことをやってい
る可能性があります.

OS ＼ オプション	-O0	-O	-O2	-O2 -funroll-loops
RedHat 5.0	5.72	21.99	27.06	28.14
StataboWare-1.2β	9.25	21.31	28.13	31.46
DEC C + f2c(参考)	13.73	41.20	-	-
性能比 SW12/RH5	1.62	0.97	1.04	1.12

今回はハードディスクの不良というアクシデントがあったのであまり
評価ができませんでした. 次号ではDECのコンパイラとの比較も含めて性能の
評価をもう少し真面目にやってみたいと思います.

(とはいっても私が新しいマシンを買える訳ではないので21064Aでの
評価となることは御承知おきください.
文句がある人はマシンを寄付して下さい.. なんてね, 冗談です :-)

4. おわりに

フリーなOSにおける数値計算環境としてLinux/Alphaは極めて優れた
性能を持っています.
たびたび言及する後藤さんのdgemm互換ルーチンはAlphaServer 8400/625で
700MFLOPSを記録したそうです. こうなると本当にスーパーコンピュータの領域に
近付いたと言えるのではないでしょうか?

さらにStataboWareのβ版では私と後藤さんの
関数ライブラリのチューニングの結果弱いと言われてきた関数性能に
関してもかなり実用になっていると思います.
これらの成果は本来は逐一英語で世界に向けて情報発信をするべきと思うの
ですが, 残念ながら忙しくてさぼっています.

さて, 最初に述べたように
マルチプロセッサのマシンはまだまだ高値の花ですが単一プロセッサであるなら
Alphaを載せたマシンも多くの人の手の届く値段になりつつあります. そこで,
これらのマシンを並列につないで性能を出すことも視野に入れて欲しいと思います.
単一プロセッサの話が一段落したらLinux/AlphaでMPIを動作させて
クラスタ接続の話をしたいと思います.