スレッドレベル並列性とプロセッサ性能

Thread-level Parallelism and the Performance of a Program on various Processors

ネットワーク情報学部　石原秀男
School of Network and Information　Hideo ISHIHARA

Keywords: Dual Core、Dual CPUs、Hyperthreading

1. はじめに

Pentium4以降のIntel CPUはクロックの高速化によって性能向上が行われてきた。パイプライン段数を20ステージへと引き上げたNetburstマイクロアーキテクチャ[1]も、180nm、130nm、90nmとシュリンクされてきた製造プロセスも、すべてはクロックを上げるためだったと言ってもよいだろう。しかし、この路線はTejas[2]のキャンセルとともに終わりを告げることとなった。キャンセルの理由については、当初いろいろな憶測が飛び交ったが、今では発熱が原因であったことが知られている。CPUのクロックを上げれば、消費電力もアップし、発熱の面が厳しくなる。発熱の問題は、製造プロセスを微細化すれば動作電圧を引き下げることが可能となり、回避できるというのがこれまでの常識であった。ところが、90nmプロセスで製造されたPrescottの熱消費電力はなんと115Wにも達し、130nmプロセスのNorthwoodよりも格段に上がってしまったのである。犯人は予想をはるかに超えるリーク電流だった。90nmプロセスで5GHz超え、次の65nmプロセスで6GHz超えというIntelのもくろみは、もろくも崩れ去ってしまったのである。リーク電流を減少させるためには、トランジスタのゲートデバイスを吟味することから始めなければならない。当然、数年単位の年月を要することは間違いない。

さて、一方のAMDはクロックとは異なるアプローチによる性能向上を狙っていた。クロックアップは性能向上の王道ではあるが、膨大な投資を要する製造プロセスの微細化が必須であり、弱小企業のAMDにその力はない。そこで彼らが考えたのが、デュアルコアである。AMDはOpteronやAthlon64の設計時に、すでにデュアルコア化を視野に入れた設計をしていたとされており、2004年の秋にはOpteronのデュアルコア版の試作品を公開した。つまり、クロックで攻めるIntelに対してデュアルコアで対抗しようとしていたわけである。

クロックアップ路線が破綻したIntelとしては、64ビット化で遅れを取っていた上に、パフォーマンスでもAMDに置き去りにされるという事態だけは、なんとしても防がなくてはならなかった。そこで急遽、デュアルコアの開発に取り掛かったわけである。その結果、2005年の5月にIntelにとっては急遽、AMDにとっては予定通りにデュアルコアCPUが市場に登場することになったのである。

そういうわけで、2005年は、PCにとってのデュアルコア元年となった。もちろん、ユーザにとって大切なのは、デュアルコアであるか否かではなく性能である。現在のところ、PCのCPUには、デュアルCPU、デュアルコア、擬似デュアルコア（ハイパースレッディング）、シングルコアという選択肢がある。本稿では、筆者が実際にテストすることが可能であった10種のCPUを搭載したシステムについて、自作のベンチマークプログラムで演算性能を調査した結果について述べる。

2. 実験対象システム

実験に使用したCPUはIntel製5種、AMD製5種の計10種である。なお、旧世代の製品も含まれているが、価格については、2006年1月現在の秋葉原での販売価格[3]を記載している。

2. 1 Intel製CPU

(1)Xeon 3.2GHz

サーバ用CPU。コアはPrestonia、FSBは533MHzで動作クロックは3.2GHz。キャッシュはL2の512KBに加えてL3に1MB。価格は¥78000で、テストにはこれを2個搭載したデュアルマシンを使用した。

(2)Pentium4 540

メインストリーム用CPU。コアはPrescott、FSBは800MHzで動作クロックは3.2GHz。キャッシュはL2に512KB。ハイパースレッディングをサポートし、価格は¥24000。テストに使用したのは540であるが、現在はほぼ同価格の541が出回っており、こちらはEM64T[4]に対応している。

(3)PentiumD 820

デュアルコアCPU。コアはSmithfield、FSBは800MHz

---

[1] 最新のものでは31ステージに引き上げられている。
[2] Prescottの次世代コアで4GHzを超える予定のCPUだった。
[3] 新品が入手不可能な旧世代の製品については、中古品の推定価格。
[4] Extended Memory 64 Technology。いわゆる64ビット対応。

で動作クロックは2.8GHz。キャッシュはL2に1MB＋1MBの2MB。価格は¥30000。Smithfieldは共有する資源を全く持たない二つのPrescottダイをFSBで接続させたものであり、本格的なマルチコアとは言い難い。なお、Smithfieldコアについては Pentium Extreme Editionを除いてはハイパースレッディングがDisableにされている。

(4)CeleronD 326

ローエンド用CPU。コアはPrescott、FSBは533MHzで動作クロックは2.53GHz。キャッシュはL2に256KB。ハイパースレッディングはサポートされていないが、価格は¥8500と非常に安価である。

(5)PentiumM 1.70MHz

モバイル用CPU。コアはBanias、FSBは400MHzで動作クロックは1.70GHz。キャッシュはL2に1MB。ハイパースレッディングはサポートされていない。Baniasコアは旧世代で、現在はL2が2MBになったDothanコア（FSBは400MHzと533MHz）が主流である。消費電力が低く、クロックの割には高性能であることから、静音PC用の素材として一部のマニアに人気がある。価格は、26000円程度。

2. 2 AMD製CPU

(1)Athlon 64 X2 4200+

デュアルコアCPU。コアはManchester、FSBは1000MHzで動作クロックは2.2GHz。キャッシュはL2に512KB＋512KBの1MB。価格は49000円。二つのコアがシステムリクエストインタフェース経由でCPUクロックと同期して通信できるため、PentiumDのようにFSBによるボトルネックがなく、より本格的なデュアルコアである。既存のシングルコアCPUと互換性があり、従来のSocket939マザーボードでもBIOSアップグレードで使用できるという利点がある。

(2)Athlon MP 2800+

サーバ用CPU。コアはBarton、FSBは333MHzで動作クロックは2.133GHz。キャッシュはL2に512KB。価格は¥25000で、テストにはこれを2個搭載したデュアルマシンを使用した。

(3)Athlon64 3000+

メインストリーム用CPU。コアはClawHammer、FSBは800MHzで動作クロックは2GHz。キャッシュはL2に512KB。価格は16000円。AMDの64ビット対応であるAMD64機能を備えている。

(4)Opteron146

サーバ用CPU。コアはSledgeHammer、FSBは800MHzで動作クロックは2GHz。キャッシュはL2に1MB。価格は25000円。サーバ用CPUではあるが、デュアルCPUには対応していない[5]。AMDの64ビット対応であるAMD64機能を備えている。

(5)Athlon XP-M 2400+

モバイル用CPU。コアはBarton、FSBは266MHzで動作クロックは1.866GHz。キャッシュはL2に1MB。旧世代の製品であるが、探せば10000円程度で入手可能。AMDの現行廉価版CPUであるGeodeは、これとほぼ同じ製品である。デュアル動作が可能なことから一部のマニアに人気がある。

下のTable.1、Table.2は、以上についてその特徴を表にしたものである。

Brand	Xeon	Pentium4	PentiumD	CeleronD	PentiumM
Name	3.2GHz	540	820	326	1.7GHz
Clocks	3.2GHz	3.2GHz	2.8GHz	2.53GHz	1.7GHz
FSB	533MHz	800MHz	800MHz	533MHz	400MHz
Cache	512KB+1MB	512KB	512KB + 512KB	256KB	1MB
Dual Core	×	×	○	×	×
Dual CPU	○	×	×	×	×
HyperThreading	○	○	×	×	×
64bit	×	×	○	○	×
System Costs	¥156,000	¥24,000	¥30,000	¥8,500	¥26,000

Table.1　Intel CPUs

Brand	Athlon 64 X2	Athlon MP	Athlon 64	Opteron	Athlon XP-M
Name	4200+	2800+	3000+	146	2400+
Clocks	2.2GHz	2.133GHz	2GHz	2GHz	1.866GHz
FSB	1000MHz	333MHz	800MHz	800MHz	266MHz
Cache	512KB + 512 KB	512KB	512KB	1MB	512KB
Dual Core	○	×	×	×	×
Dual CPU	×	○	×	×	○
64bit	○	×	○	○	×
System Costs	¥49,000	¥50,000	¥16,000	¥25,000	¥20,000

Table.2　AMD CPUs

[5] 同スペックのOpteron246は2WAYに、846は8WAYに対応している。

3. シングルスレッドに対するパフォーマンス

デュアルコアやデュアルCPUが真価を発揮するのはマルチスレッドアプリケーションであるが、既存のアプリケーションのほとんどはシングルスレッドで作られている。したがって、現状におけるPCの使い勝手については、シングルスレッドに対する処理性能が左右することになる。そこで、まず、シングルスレッドアプリケーションについて、前述の10種のCPUを搭載したシステムの処理速度を測定することとした。

測定には、Appendix.1に記した自作プログラムを用いた。このプログラムは、3から10000までの間に存在する素数の数を数えるというものであり、ある奇数nが素数であるか否かの判定は、3からn/2までの数でnを割り、すべての数で割り切れなければ素数とカウントするという素朴な方法を用いている。割る数としては、n/2まででではなくnの平方根までで充分なのであるが、浮動小数点パッケージを使わずに純粋な整数演算性能を調べるためにこのようにしている。また、count_func()において

```
for(j=3;j<=upper;j+=2){
    if(i%j==0){
            is_prime=0;
        }
}
```

ではなく

```
for(j=3;j<=upper;j+=2){
    if(i%j==0){
            is_prime=0;
        break;
        }
}
```

とすべきであるが、後述するマルチスレッド版との比較のためループの回数を固定化する目的であえてこのようにしている。計算速度については、この計算を10回行った時間を測定した。

時間の測定法は、Windowsのマルチメディアタイマを使用し、timeBeginPeriod(1)で精度を1ミリ秒とした上で、timeGetTime()で行っている。また、Windowsそのものがマルチプロセスの OSであるために、常に数十ものプロセスが同時に動いている。そこで、SetPriorityClass APIを用いて、このプロセスの優先度を上げることによって優先的に実行[6]させている。タイマの誤差やマルチプロセスによる誤差の影響を低減するために、各CPUについて10回の測定を行い、最大値と最小値を除いた上で平均値を求めた結果がTable.3である。

[6] 優先度を高めてはいるが、Windowsはリアルタイム OSではないので、占有的にCPUを使用しているわけではない。

Brand	Time (msec)	Score	CPI
Xeon	1095	75.7	5.2
Pentium 4	829	100	44.6
Pentium D	891	93	33.2
Celeron D	1045	79.3	100
Pentium M	1398	59.2	24.4
Athlon 64 X2	1225	67.6	14.8
Athlon MP	1524	54.4	11.7
Athlon 64	1512	54.8	36.7
Opteron	1362	60.9	26.1
Athlon XP-M	1335	62.1	33.3

Table.3　Results of Single Thread Application

Table.3において、Timeは測定された時間、ScoreはTimeの逆数を規格化したもので、値が大きいほど高速であることを示している。また、CPIはCPUの価格（デュアルCPUの場合は2個分）をScoreで割って規格化したもので、やはり値が大きいほどコストパフォーマンスが良いことを示している。Fig.1はTable.1のTimeをグラフにしたものである。

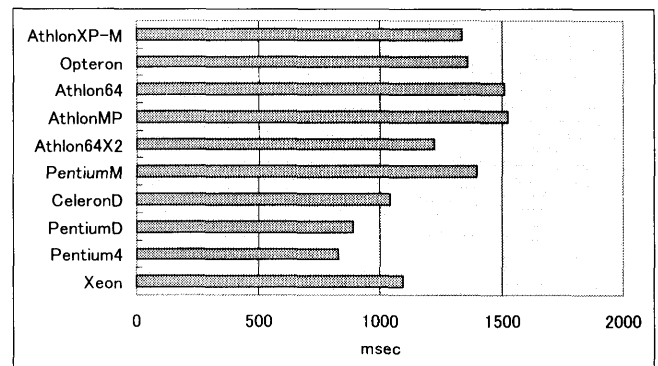

Fig.1 Times of Single Thread Application

Table.3やFig.1からわかるように、単純なシングルスレッドアプリケーションにおいてはPentium4が最速で、PentiumDがそれに続くという結果になった。一方、AMDのCPUは全般にあまり振るわない。これについては、コンパイラにVisualC++6.0を用いているためデフォルトて適用されるPentium最適化オプションの影響が考えられるが、このように作られているアプリケーションは少なくないので、素直に結果として受け止めておくべきだろう。AMD同士で比較した場合、64 X2の性能は際立っている。64に比べて実クロックは1割の上昇に過ぎないが、20%以上の性能アップである。既存のCPUと差し替えるだけでこの効果が得られるわけであるから、AMDユーザはX2の導入を考えても間違いではないだろう。また、OpteronやXP-Mもクロックを考えればなかなかの性能を示している。これに対して、Pentium4とPentiumDについては、クロック相応の差しかない。シングルスレッドアプリケー

ションを使用する限りにおいては、PentiumDを使う意味はないと言えよう。

一方、コストパフォーマンスに関して言えば、CeleronDの良さが際立っている。価格で10数倍も違うXeonを上回る性能を示しているのであるから素晴らしい。CeleronDを除けば、Pentium4やPentiumDはまずまずだが、PentiumMについてはそれらよりも一段劣っている。静音などにこだわるのでないならば、デスクトップマシンで使う理由はないだろう。AMDについては、Athlon64とXP-Mあたりがコスト的には許せる下限で、X2はAMDの中でこそトップの性能を誇るものの、IntelのCPUに比べればコストが高すぎる。

さて、Fig.2はTable.3のTimeと実クロックを図にしたものである。これを見ると、Opteron、Athlon64 X2、CeleronD、PentiumDがクロック相応の性能、PentiumM、Athlon XP-Mがクロックに対して高性能、Athlon64、Athlon MP、Pentium4がクロックに対して若干遅く、Xeonが大幅に遅いことがわかる。

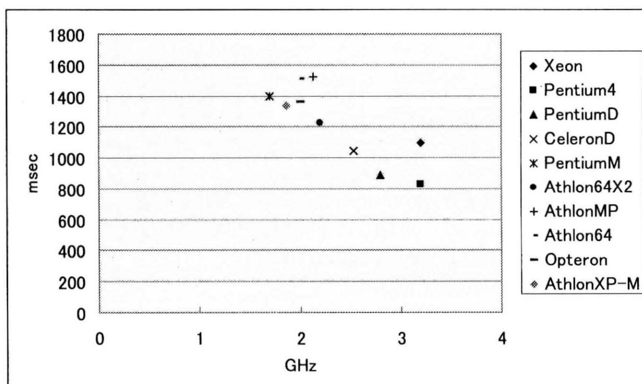

Fig.2 Times and Clocks

結局、シングルスレッドアプリケーションに限るならば、Pentium4が最もバランスが取れた選択ということになり、コストを重視するならCeleronDもあり得るだろう。いずれにしろ、PentiumD、Athlon64 X2などのデュアルコアやXeon、Athlon MPなどのデュアルCPUは全くコストに見合わず、既存のアプリケーションを使うのなら、使う意味はなさそうに思える。デュアル系で唯一理性的な選択は、入手性には難があるもののAthlon XP-Mなのかも知れない。

4. マルチスレッドに対するパフォーマンス

シングルスレッドアプリケーションでは全く効果がなかったデュアルコアやデュアルCPUであるが、マルチスレッドに対してはどうであろうか。ここでは上の素数の数え上げをマルチスレッド化することによって、各システムの性能を測定することとした。たとえば、3から10000までの素数を数える場合、4スレッド化するためには

3, 11, 19, ・・・, 9979, 9987, 9995
5, 13, 21, ・・・, 9981, 9989, 9997
7, 15, 23, ・・・, 9983, 9991, 9999
9, 17, 25, ・・・, 9985, 9993

というように4つの系列に分けて素数判定を行うこととし、各系列を異なるスレッドとして処理することにすれば全体の計算量はほぼ同じでスレッドの並列化ができる。これを踏まえて作成したプログラムがAppendix.2である。なお、mainについては

```
RunMutex=CreateMutex(NULL,FALSE,NULL);
```

と排他制御に用いるミューテックスを作成する部分と、それを閉じる

```
CloseHandle(RunMutex);
```

が加えられていること、count_primeを呼び出すときに引数としてスレッド数を渡すこと以外は同じなのでソースコードは省略した。マルチスレッドのプログラミング手法については文献1)を参照して欲しい。ここではスレッドの作成にCreateThread APIを使用しているが、マイクロソフトによるとCの標準関数とCreateThreadを同時に使用した場合、若干のメモリリークが生じるらしい。実際、このアプリケーションを連続して実行した場合、原因不明のハングアップが発生するケースがあった。これについては、CreateThreadのかわりにbeginthreadを使用することによって回避できるようである。

さて、同時に実行できるスレッドの数はCPUによって決まっている。今回の場合、XeonについてはデュアルCPUとHyperThreadingで4個、Athlon MPとAthlon XP-MはデュアルCPUで2個、PentiumDとAthlon64 X2はデュアルコアで2個、Pentium4はHyperThreadingで2個、その他のCPUは1個である。理屈の上では、最高のパフォーマンスを得るためには、それぞれについて同時実行可能なスレッドの数だけスレッドを用意すれば良いということになるが、現実はそれほど単純ではない。WindowsがマルチタスクOSであるからである。つまりWindows上では、常に複数のタスクが動作しており、ラウンドロビンスケジューリングで切り替わっている。アプリケーションを多くのスレッドに分割すれば、実際に実行されるタスクの中でそのアプリケーションが占める割合が上がるために、結果として純粋な実行時間が増える可能性があるのである。もちろん、スレッドの切り替えには、プロセスほどではないにしろ、多少のオーバヘッドを伴うために、スレッドをやたらと増やせばよいというものでもない。そこで、ここでは、各システムについて1スレッドから50スレッドまでにアプリケーションを分割して、実行時間を測定することとした。測定については、シングルスレッドと同様に10回行い、最大値と最小値を除いた上で平均値を求めた。

Fig.3とFig.4がその結果である。グラフを二つに分けたことからも明らかなように、マルチスレッドに対する結果

は二つのグループに分けられた。すなわち、スレッド数を増やすと実行時間が短縮されるかあるいはほとんど増えないものと、スレッド数を増やすと実行時間が爆発的に増加してしまうものである。前者に属するのは、Xeon、Pentium4、PentiumD、Athlon64 X2、Athlon MP、Athlon XP-Mであり、後者に属するのはCeleronD、PentiumM、Athlon64、Opteronである。このグループが、2つ以上のスレッドを同時実行できるCPUと、そうではないCPUに一致しているのは偶然ではないだろう。前述したように、Windowsのタスクスケージューリングの関係上、たとえ同時に実行できるスレッドが一つであったとしても、マルチスレッド化によって現実的な実行速度が速くなる可能性があるし、筆者は実際にそのような例を経験している。しかし、今回のテストではマルチスレッド化が悪影響を与えている。この理由として、本来マルチスレッド対応ではないシステムにおいては、スレッド切り替えのオーバヘッドが相当に大きいという可能性が考えられる。スレッドの数を比較的少なめに抑えておけば弊害は少ないものの、このことは注意しておいた方が良いだろう。

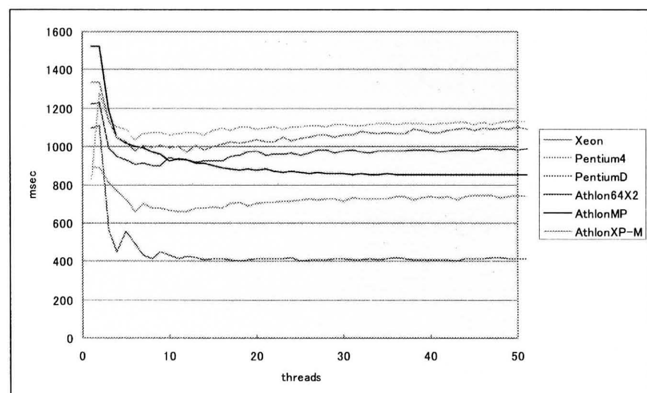

Fig.3 Results of Multi Threads Application(1)

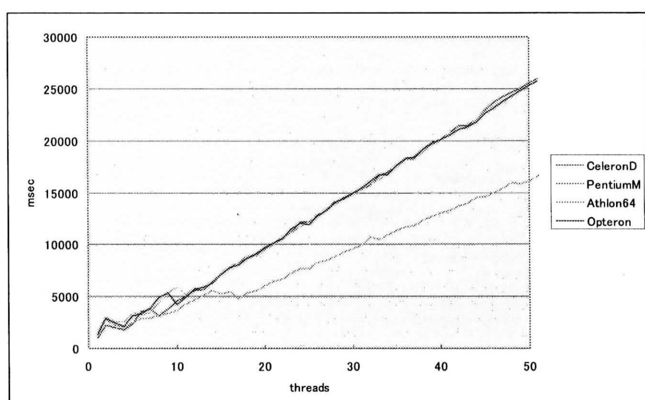

Fig.4 Results of Multi Threads Application

これに対してマルチスレッド対応のシステムの場合には、10スレッドくらいまでは急激に実行時間が短縮化され、それ以降はスレッドを増やすことによって若干の速度低下を生じるケースもあるものの、その増え方は非常に緩やかである。単一のアプリケーションが数十ものスレッドを持つというのは常識的ではないものの、複数のマルチスレッドアプリケーションが同時実行される可能性を考えれば、今後はこの種のCPUが力を発揮するケースは増えて行くに違いない。

さて、筆者にとって個人的に最も興味があるのはピーク性能なのであるが、それを表にしたものが下のTable.4である。

Brand	Time(msec)(threads)	Up	Score	CPI
Xeon	401(17)	273.1	100.0	31.0
Pentium 4	1038(5)	79.9	38.6	77.8
Pentium D	657(11)	135.6	61.0	98.3
Athlon 64 X2	920(14)	133.2	43.6	43.0
Athlon MP	848(41)	179.7	47.3	45.7
Athlon XP-M	969(11)	156.0	41.4	100.0

Table.4 Peak Performance of Multi Threads Application

Timeは1から50スレッドの中で最速であった実行時間であり、カッコ内はそのときのスレッド数である。Upはシングルスレッドアプリケーションに対する実行時間の向上率であり、値が大きいほどマルチスレッド化のメリットが大きいこと意味している。ScoreはTimeの逆数を規格化したもので、値が大きいほど高速であることを示している。また、CPIはCPUの価格（デュアルCPUの場合は2個分）をScoreで割り規格化したもので、やはり値が大きいほどコストパフォーマンスが良いことを示している。

この結果で注目すべきはPentium4のUpである。Upが79.9ということは、マルチスレッド化により、速度がシングルスレッドの79.9%になったということで、残念ながらマルチスレッドの効果は得られていない。すなわち、Fig.4に示したグループのように、マルチスレッド化による大幅な性能ダウンは見られないものの、性能はダウンしてしまうことを示している。つまりHyperThreadingの効果が限定的であることを示している。性能面で秀でているのはデュアルCPUプラスHyperThreadingで実質4CPUのXeonであり、次点のPentiumDの1.5倍以上の速度と他を圧倒している。全般的な傾向としては、デュアルCPUが良好なパフォーマンスを示しており、特にAthlonMPでは1.8倍近いパフォーマンスアップが得られている。それに対してデュアルコアCPUであるPentiumDやAthlon64 X2の性能向上は1.3倍程度に留まっている。また、PentiumDよりもより本格的なデュアルコアであるはずのAthlon64 X2が残念な結果に終わっているが、これは絶対的なクロックが低いことによるものであろう。ゲーマーの間では、Athlon系のCPUは高性能であるとされているが、ここで扱っている例のように非常に単純な整数演算のような素の能力においてはPentium系に及ばない部分もあるようである。一方、コストパフォーマンスの面を見ると、

Athlon XP-Mが一番であり、PentiumD、Pentium4なども比較的良好であることがわかる。

5. おわりに

本稿では、10種のCPUを搭載したシステムについて、シングルスレッド、マルチスレッドの性能を調べた。整数演算という限られた条件で、しかも、OSをクリーンインストールせずに既存の環境上でベンチマークを走らせているために、正確な測定が行われているわけではないが、ある程度の傾向はつかめているものと思う。結果を踏まえて、筆者のお薦めCPUについてまとめておこう。

CPUのことなど知りたくもない、性能など気にしないという人ならば、CeleronDという選択もあるかも知れない。マルチスレッドや負荷の高い環境には向きそうもないが、コストパフォーマンスだけは抜群である。しかしながら、メールとブラウザが使えれば充分というならともかく、ネットワーク情報学部の学生に薦められるようなものではない。現状で万人に薦められるのは、やはりPentium4ということになるだろう。クロックの高さのおかげでシングルスレッドの性能は良好であるし、マルチスレッドや高負荷で急激に性能が低下するというわけでもない。少し予算があるのなら、PentiumDも悪くない。但し、シングルスレッド、すなわち既存のアプリケーションでPentium4並みの性能を出すためには、Pentium4並みのクロックの製品が必要になる。但し3.2GHzのPentiumDとなると、現状では60000円を超えており、コストに見合うものとは言い難い。AMDについては、今回の結果からは積極的に採用する理由が見つからないが、コンパイラの設定などによって結果が大きく異なる可能性がある。それについてはいずれ機会を改めて調べてみたい。しかしながら、既存のAthlon XPをリプレースするのなら、Athlon64 X2という選択はあるだろう。なんと言ってもCPU代だけでアップグレードできるのは、Intelにはないメリットである。Xeon、Athlon MPなどのデュアルシステムは、性能は良いもののコストが高過ぎる。高性能のものがコストパフォーマンスで劣るのはこの世の常ではあるが、自分でマルチスレッドのプログラムを書くような人でないと、使いこなすことはできないだろう。かなりマニアックな選択にはなるが、そういうことに興味があって資金に余裕がない学生ならば、Athlon XP-Mを使ってみるのも面白そうである。

なお今回の実験においては、ネットワーク情報学部4年の須藤洋平君にOpteronとAthlon XP-Mのデータを、ネットワーク情報学部3年の中西勇貴君にAthlon64 X2のデータを提供してもらった。二人の協力に感謝の意を表しておきたい。

参考文献

1) マルチプロセッサとハイパースレッディングのパフォーマンス, 石原, ネットワーク&インフォメーション, 専修大学, 2004

Appendix.1 Single thread program

```
#include <stdio.h>
#include <windows.h>
#include <mmsystem.h>
#define MAX      10000
#define MIN      3

typedef struct{
    int lower;
    int upper;
    int step;
}DATA;

int count_func(DATA *p){
    int i,j,upper;
    int num_prime=0,is_prime;

    for( i=p->lower ; i<=p->upper ; i+=p->step){
       upper=i/2;
       is_prime=1;
       for(j=3;j<=upper;j+=2){
           if(i%j==0){
               is_prime=0;
           }
       }
       if(is_prime==1)
           num_prime++;
    }

    return num_prime;
}

int count_prime(){
    DATA data;

    SetPriorityClass(GetCurrentProcess(),HIGH_PRIORITY_CLASS);

    data.lower = MIN;
    data.upper = MAX;
    data.step = 2;

    return count_func(&data);
}

int main(void){
    int num_primes,turn,rec;
    int i,j;
    DWORD start_time,finish_time;
```

```
    timeBeginPeriod(1);

    start_time=timeGetTime();
    for(i=1;i<=10;i++)
        num_primes = count_prime();
    finish_time=timeGetTime();
    printf("single(%d) %d",num_primes,finish_time-start_time);

        timeEndPeriod(1);

    return 0;
}
```

Appendix.2 Multi threads program

```
#include <stdio.h>
#include <windows.h>
#include <mmsystem.h>

#define MAX_THREADS       50
#define MAX           10000
#define MIN           3

typedef struct{
    int lower;
    int upper;
    int step;
    intnum;
}DATA;

volatile int finished_threads;
volatile int primes_each_thread[MAX_THREADS+1];

DATA data[MAX_THREADS+1];

HANDLE RunMutex;

DWORD WINAPI count_thread(LPVOID ptr){
    int i,j,upper;
    int num_prime=0,is_prime;
    DATA *p;

    p=(DATA *)ptr;

    for( i=p->lower ; i<=p->upper ; i+=p->step){
        upper=i/2;
        is_prime=1;
        for(j=3;j<=upper;j+=2){
            if(i%j==0){
                is_prime=0;
            }
        }
        if(is_prime==1)
            num_prime++;
    }

    primes_each_thread[p->num]=num_prime;
    WaitForSingleObject(RunMutex,INFINITE);
        finished_threads++;
    ReleaseMutex(RunMutex);
    return 0;
}

int count_prime(int num_threads){
    int i;
    int step,lower,num_primes;
    DWORD id[MAX_THREADS+1];

    SetPriorityClass(GetCurrentProcess(),HIGH_PRIORITY_CLASS);

    finished_threads = 0;
    for( i=1 ; i<= num_threads ; i++)
        primes_each_thread[i] = 0;

    step =  num_threads*2;
    lower=MIN;

    for( i=1 ; i<=num_threads ; i++){
        data[i].lower=lower;
        data[i].upper=MAX;
        data[i].step=step;
        data[i].num=i;
        lower+=2;
    }

    for( i=1 ; i<= num_threads ; i++){
        CreateThread(NULL,0,count_thread,&data[i],0,&id[i]);
        SetThreadPriority(&id[i],THREAD_PRIORITY_TIME_CRITICAL);
    }

    while(finished_threads!=num_threads);

    SetPriorityClass(GetCurrentProcess(),NORMAL_PRIORITY_CLASS);

    num_primes = 0;
    for( i=1 ; i<= num_threads ; i++)
        num_primes += primes_each_thread[i];

    return num_primes;
}
```