第1章　数値計算とPC Cluster

1.1　何故，大規模計算が必要なのか

科学技術における研究開発は様々な方法で行われる。そのうち，数値計算を伴うシミュレーション (コンピュータによる模擬実験) が必要となる場合の研究開発の流れは，おおむね次のようになるだろう。

1. 基本となる自然科学の法則から，シミュレーションが必要なモデルの数学表現を導く。その中の多くは常，偏微分方程式として表現される。

2. シミュレートしたい条件をパラメータとして入力し，その解を得る。次のステップでは解を数値データとして使用するので，解析解が既知であればそれを使用して，それが困難であれば離散化を行って近似的に数値解を得る。

3. シミュレーション結果である数値データの検討を行う。データが大量であったり，数値の羅列を見てもよく理解できない時には，それをコンピュータグラフィックスとして表現し直して検討材料として用いる。

4. 更なる検討材料が必要であれば，パラメータを変更して再度2のステップからシミュレーションを行う。

このうち，モデルが大規模であったり，精密な数値解が必要であったりした場合，2の段階で多大な計算時間が必要となる。この時間を短縮することによって，より多くの検討材料を得ることが出来，結果として本来の目的である科学技術開発へ貢献することが出来る。

しかし，一般に大規模，あるいは精度の高い数値解を得るには多大なコンピュータ資源を必要とする。それを示す一例として，移流拡散方程式を取り上げることにする。

1.2　移流拡散方程式の近似数値計算

x 方向への一様な流速の中で溶解している物質の濃度 $u = u(x)$(未知) は次の線型二階常微分方程式 (1.1) を満足する。これを一次元の移流拡散方程式と呼ぶ。

$$-\nu\frac{d^2u}{dx^2} + \frac{du}{dx} = f(x) \tag{1.1}$$

ここで ν は拡散係数と呼ばれる正定数であり，関数 $f(x)$ は点 x における単位時間・単位面積あたりの物質の発生量であり，どちらも既知であるとする。

この常微分方程式の $x \in [0,1]$ における，境界条件 $u(0) = u(1) = 0$ を満足する解を求める。ごくアラっぽいやり方で，近似的に数値解を求めてみよう。

閉区間 $[0,1]$ を均等に 5 分割したとする。その時，各小区間は $h = (1-0)/5 = 0.2$ の幅を持つ。この小区間の端点を x_0, x_1, x_2, x_3, x_4, x_5 とする。もちろん $x_0 = 0$, $x_5 = 1$ である。それ以外の点は $x_i = x_0 + ih$ $(i = 1,2,3,4)$ と表現できる。この各分割点における解 $u(x_i)$ の近似値を u_i と書くことにする。

(1.1) 式で用いられている各導関数を前進差分で置き換えると，例えば x_1 における一階，二階導関数値は

$$\begin{aligned}\frac{du}{dx}(x_1) &\approx \frac{u_1 - u_0}{h} \\ \frac{d^2u}{dx^2}(x_1) &\approx \frac{u_2 - 2u_1 + u_0}{h^2}\end{aligned}$$

となる。これを用いて (1.1) を各点において置き換えてまとめ，行列とベクトルで表現してみると

$$\left(\frac{\nu}{h^2}\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} + \frac{1}{h}\begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix}\right)\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \end{bmatrix} \tag{1.2}$$

となる。このような問題のすり替えを「離散化」と呼ぶ。これは 4 次元の正則な連立一次方程式であるので，一意な解 $[u_1 \cdots u_4]$ を得ることが出来る。

しかし，問題があることは明白である。

常微分方程式 (1.1) と連立一次方程式 (1.2) は数学的に異なるものであることを認識する必要がある。後者はあくまで前者の「近似」に過ぎないので，果たして後者の解 $u_1, ..., u_4$ はどの程度，$u(x_1), ..., u(x_4)$ に「近い」のかを見極める必要がある。

「微分」を「差分」に置き換えるという，元々は極限値として定義されたものをある一定値に止めたものに差し替えているのだから，常識的にもその置き換えはなるべく「近い」もの，即ち h を極力小さいもので行うべきと考えられる。そう

すると，5 分割よりは 10 分割が望ましく，より正確にするためには 100 分割, 1000 分割，10000 分割... すればよいことになる。しかしその際には分割数 n に比例して連立一次方程式の次元数は増加することになる。10000 分割すれば 9999 次元の連立一次方程式を解かねばならない[1]。

この例は一次元問題なのでこの程度で済むが，もし 2 次元, 3 次元の偏微分方程式であればこの増加率は分割数 d に対して $O(d^2)$, $O(d^3)$ の連立一次方程式を解くことになってしまう。精密な数値データを得るのはかくも大規模な問題を解く必要が出てくるのである。

1.3 PC Cluster へ至る経緯

このような大規模な数値シミュレーションを行うために高速なコンピュータ (Supercomputer) が開発され，使用されてきた。現在最も高速なスーパーコンピュータは Top500[6] にて参照できる。これによれば，近年のハードウェアのトレンドは，多数の PC をネットワークで結合した PC Cluster によるものが多いことがわかる。

その理由をきちんと述べようとすれば，コンピュータの歴史について一くさり語らねばならない。大ざっぱにその流れを示すと

1. ENIAC 開発 (1945 年)
2. 大型計算機の時代
3. Downsizing の流れ→ Workstation(WS)/Personal Computer(PC) へ (1980 年代)
4. IT アーキテクチャの一本化 (1990 年代)

   Hardware DOS/V の登場により，PC/AT Compatible へ

   Network Hardware LAN の普及により，Ethernet(10〜1000BASE) へ

   Network Protocol The Internet の爆発的普及により，TCP/IP へ

   Operating System Windows(TM) と UNIX(TM)(含 UNIX Compatible) による寡占化
5. Node 内並列性能の追求 (2005 年〜)

   CPU SIMD 命令強化，Multi-core 化と電力消費量の抑え込み

   Hardware 演算のハードウェア化 (GPU による並列演算，FPGA による演算)

[1] 実際には行列の 0 成分が多いので，それを利用して計算時間を減らす工夫を行う。

となる。大型計算機中心だった時代から，PC へと移行する中で，ハードウェアから基盤となるソフトウェアが統一されてきたことによって，コンピュータに関するコストも大幅に下がってきた。その上，PC の中核である CPU の速度は劇的に改善され (図 1.1)，大量のリソースを消費する GUI OS が主流となった。

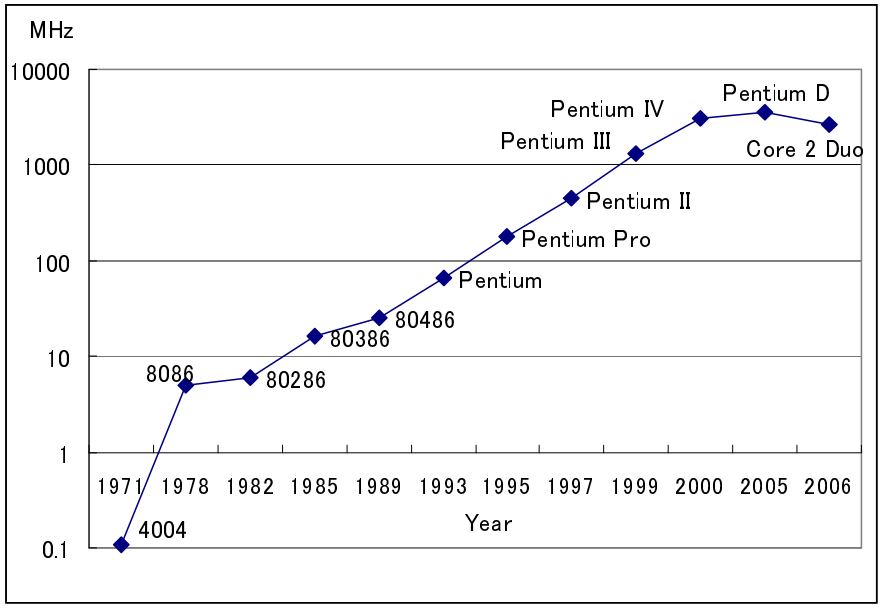

図 1.1: Intel 製 CPU の動作周波数

しかし 21 世紀を越えると，CPU core の動作周波数の向上の限界が見えてくるようになってくる。トランジスタの集積度が上がるにつれて使用電力量が急激に増大し，費用対効果の面から電力量と処理速度の向上との比そのものが問題視されるようになってきた。そこで，並列度を上げて性能向上を図る，という流れになり，2004 年にはコンシューマ向けとしては初の Dual core CPU が AMD, Intel の両者から発売される (図 1.2)。

今までは 1CPU 内部に一つだけだった Core を複数搭載したものが Multi-core CPU である。Dual core は二つの core を搭載したものを指す。実質的には複数 CPU を搭載した SMP(Symmetric Multi-Processor) と同じ効果をもたらすものであるが，Dual-core CPU は従来の Single core CPU と同じレンジの価格帯で売り出され，コストは劇的に下がった。

しかし，SMP も Multi-core CPU も，複数の CPU/Core を目一杯使いこなすにはソフトウェアで対応する必要があり，ソースコードレベルの書き換えがいる。つまり並列処理を行えるような機構を備えたソフトウェアに仕立て直す必要があるのだ。1 node(1PC) 内部のみで小規模な並列計算 (処理) を行うのであれば。メモリ

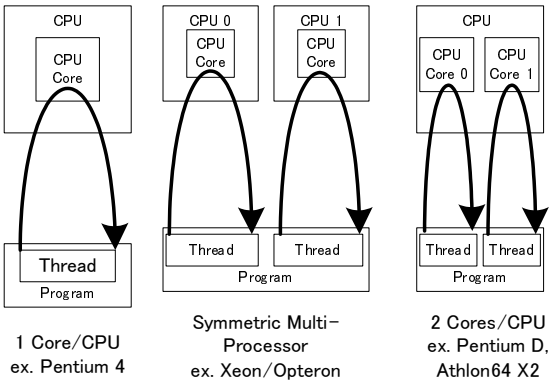

図 1.2: Single core, SMP, Dual core CPU

(RAM) は共有しつつ，内部処理を複数 Thread(スレッド) に分割して実行する，共有メモリモデルの並列化を行えばよい。この代表として Pthread(POSIX Thread) や，コンピュータ言語のマクロレベルの指定を行う OpenMP 等がある。

このような SMP/Multi-core CPU を搭載した強力な並列計算機である Commodity PC を更に複数並べて使いこなすことが出来れば，高速だが恐ろしく導入及び維持コストのかかる大型のスーパーコンピュータより身近な，しかも安価な高速計算機が実現できることになる。これを PC cluster(Commodity PC cluster) と呼ぶ。

PC cluster は使いこなすのは厄介であるが (後述)，多少のパフォーマンス低下を覚悟すれば，様々な目的に使用することも可能である。ネットワークに Ethernet を使った環境は，近年ごく当たり前のものとなっており，多くの事業所や学校，企業に見られるから，業務に差し支えのない時間帯にあらかじめ登録されたバッチジョブを流す，といったことも不可能ではない。

このような PC Cluster を使って大規模な並列計算を行うためには，ネットワークプログラミングの知識が必要となる。しかし，ユーザが個別に BSD ソケットを扱うプログラミングを行うのは非効率的であり，コンピュータアーキテクチャの統一化という歴史の流れにも反する。そこで，ユーザの利便性を図るためのソフトウェアが提供されてる。その代表的なものの一つが次に述べる，分散メモリ型の並列処理機構を手助けしてくれる MPI である。

1.4 MPI とは？

大規模な科学技術計算を実行するためには大量のメモリと高速な浮動小数点演算を備えたスーパーコンピュータが不可欠であるが，一言で「スーパーコンピュー

タ」と言ってもさまざまなタイプがある。M.J.Flynn は 1966 年，コンピュータアーキテクチャ(コンピュータの内部構成) を，命令の流れ (Instruction Stream) とデータの流れ (Data Stream) によって次の 4 つに分類した。現在では CPU を処理する命令等にも使用される用語になっている。

SISD ･･･ Single Instruction stream+Single Data stream

SIMD ･･･ Single Instruction stream + Multiple Data stream ･･･ SSE, SSE2 命令等

MISD ･･･ Multiple Instruction stream + Single Data stream

MIMD ･･･ Multiple Instruction stream + Multiple Data stream ･･･ PC Cluster

PC Cluster を Ethernet を用いて構築した時の模式図は図 1.3 のようになる。

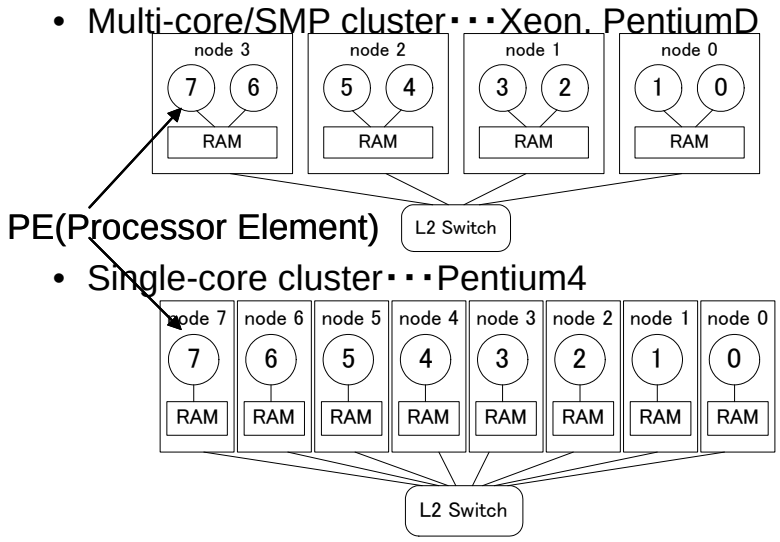

図 1.3: PC Cluster の例

それぞれの PC をノード (node) と呼ぶ。各ノードは独立したメモリと CPU を持つため，これらを統一的に扱うには，プログラムを同時に流して実行させる仕組みが必要である。また，計算に使用するデータも各ノードに分散させて運んだり，やりとりしたりする必要もある。Flynn による並列コンピュータの分類法によれば，このような複数のプログラムを流し，データも各ノード間を行き来するようなアーキテクチャを，MIMD(Multiple Instruction stream, Multiple Data stream) と呼ぶ。

このような MIMD タイプの PC Cluster を統一的に扱う API をまとめた規格が MPI(Message Passing Interface) であり，これに忠実な実装系は mpich[5]，LAM[2]

がある。名前の通り，各ノード間でメッセージをやりとりして処理を行うための関数群で，C/C++/Fortran から利用できる関数のセットが提供されている。

この MPI は 1993 年から 1994 年にかけて，40 を越える組織から研究者が集まって仕様を検討して出来上がったものである。1994 年 6 月に MPI 1.0 がリリースされ，次の年に MPI 1.1 が登場した。

その後,1997 年に動的プロセス生成，片側通信，並列 I/O をサポートした MPI 2.0 がリリースされたが，大部分の数値計算プログラムはそこまでの機能を必要としないため，現在でも MPI 1.1 の機能のみを用いるのが普通である。本書のプログラミングスタイルも，MPI 1.1 の，更にサブセットのみを使ったものになっている。

実際の MPI プログラミングは後述するソースコードを見て頂きたいが，基本は SPMD(Sigle Program, Multiple Data) というスタイルで実行される。図 1.4 の通り，大本のノードを PE0(Processor Element #0) とし，ユーザはそこで自分のプログラムをコンパイルして実行すればよい。そこで生成されたプログラムは各ノードにほぼ同時に流されて，自分のプロセス番号 (ランク番号) に合致する部分のみを実行する。標準入出力は特に指定しない限り PE0 において行われる。

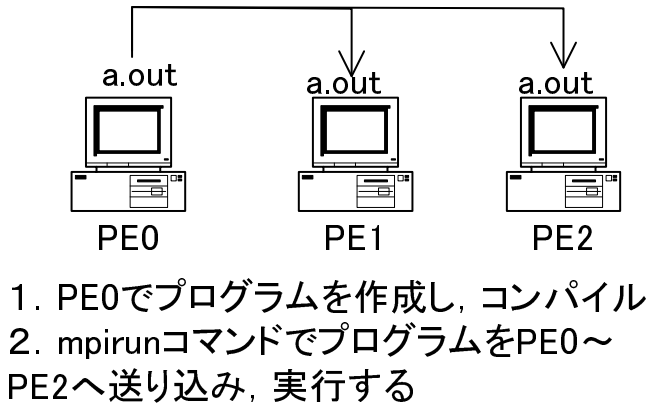

図 1.4: SPMD アプローチ

このように，ユーザは「複数のノードで実行される一本のプログラム」を書けばよい，という仕組みが SPMD である

しかし，そのためには各ノードにおけるプログラムの挙動をイメージしながらプログラミングを行う必要がある。個人差はあるものの，これに慣れるにはある程度の時間が掛かる。また，MPI は各ノード間のデータの流れを細かく制御できるよう，種々の関数が提供されているが，そのためにかえってどれを選んで使ってよいのか判断が難しい，と言われている。

これらの理由により，MPI プログラミングは難しい，アセンブラのようだ，とも言われる。よって，本書では，まずごく簡単な MPI プログラミングを大量に示し，イメージトレーニングの手助けを行う。その上で，数値計算で頻繁に使用する MPI の機能のみを抜粋して紹介する。その後はなるべく MPI の機能を陽的には使用せず，MPIBNCpack の手助けを借りて並列分散型の数値計算を行っていく。

1.5　ネットワークの性能について

コンピュータネットワーク (以下ネットワーク) は OSI の Reference model による 7 階層の機能を持っており，現実に使用されている TCP/IP+Ethernet の組み合わせの場合は図 1.5 のような機能分担になる。

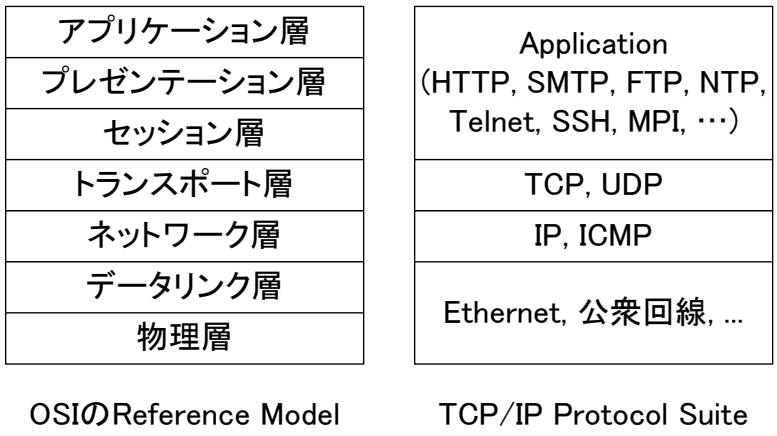

図 1.5: ネットワークの階層構造

一番上層部のアプリケーション層から送られたデータ (ペイロード) は下部の階層に渡され，必要な制御情報 (ヘッダ) が次々に加えられ，最終的には電気信号として Ethernet を渡って相手に伝えられる (図 1.6)。

電気信号は 2 値デジタルになるため，1bit 単位で送信される。従って，回線のスピード（帯域）は bit 単位 (bit/sec, bps) で表現される。Ethernet では，10BASE, 100BASE, 1000BASE(Gigabit Ethernet, GbE), 10G BASE という表現が使用されるが，これはそれぞれ最大到達速度が 10 M bps, 100 M bps, 1000 M bps, 10000 M bps ということを意味する。信号が媒体を通過する物理的な速度は光速を越えないので，実際には圧縮を加えたり，同時に送信する bit 数を増やすことで速度を上げることになるため，速さ，というよりは「太さ」という方が正確かもしれない。

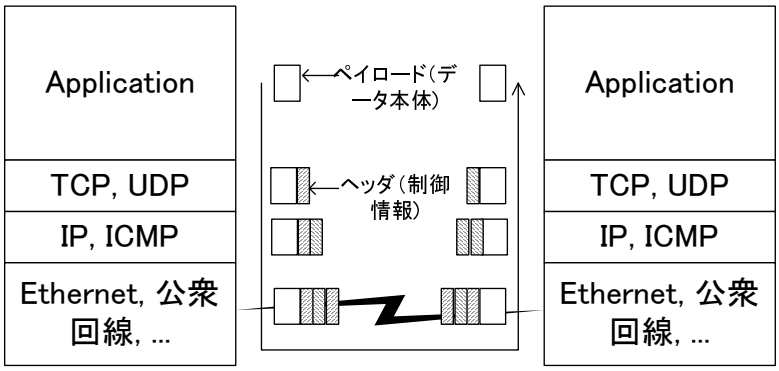

図 1.6: データがネットワークを通過する模式図

ネットワークは階層ごとにそれを接続する機器が必要になる。その名称を図 1.7 に示す。

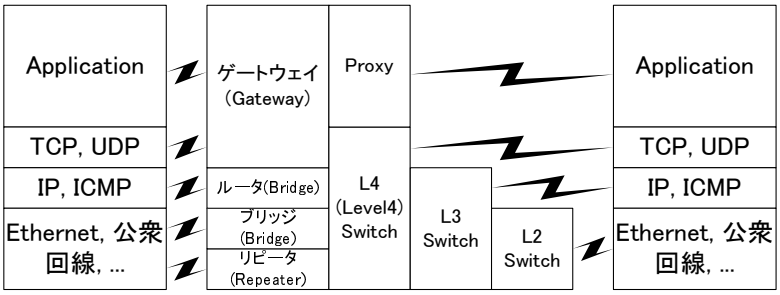

図 1.7: ネットワーク階層を接続する機器の名称

PC cluster を接続する回線の速度は，使用する Ethernet の速度と，それを接続する機器 (Switch) の性能，そして各 Node の CPU，NIC(Network Interface Card)chip の性能に依存して決定されるため，実際に計測してみないとよく分からないものなのである。そこで NetPIPE というツールを使ってネットワーク性能を計測してみることにする。

NetPIPE[15] は同一マシン内におけるメモリ転送 (memcpy)，TCP, PVM, MPI などの様々な関数，プロトコルに対応した転送性能を計測するベンチマークテストツールである。ネットワーク転送性能を計測するツールは他にも存在するが，NetPIPE は，転送するバイト数を指数関数的に増加させ，各転送バイト数ごとに通信速度を Mbps(Mega bits/sec) で出力するという特徴を持っている。そのアルゴリズムは図 1.8 ようになっている。

指数関数的に c を増加させ，更にデフォルトではバイト数を c − 3, c, c + 3 と

```
T = MAXTIME
For i = 1 to NTRIALS
    t0 = Time()
    For j = 1 to NREPEAT
        if 自分が送信元であれば
            c byte のデータを送信
            c byte のデータを受信
        else
            c byte のデータを受信
            c byte のデータを送信
        endif
    endFor
    (略)
endFor
T = T / (2 * NREPEAT)
```

図 1.8: NetPIPE のアルゴリズム

前後 3 バイトずらした場合も計測するようになっている。これにより，微妙なバイト数における cache ミスヒットや通信性能の劣化の検出が可能になっている。

使用した PC cluster は Pentium 4(cs-pccluster2), VTPCC(Xeon SMP), Pentium D である。スペックの詳細は次章を参照されたい。全て，計測時において最新バージョンである 3.6.2 をソースコードからコンパイルして使用した。

今回使用した PC cluster は全て，各 Node 間を GbE で繋いでいる。では実際にはどの程度の速度が出るのか。その結果を図 1.9 に示す。

共通しているのは，送信するデータ長が短いうちはかなり遅く，長くするにつれてロジスティック曲線を描くようにして最大の速度に漸近的に近づいている，ということである。このうち VTPCC と Pentium D は共に SMP(Dual CPU)/Dual-core の CPU を持っており，CPU 性能に余裕があるためかほぼ 900Mbps の最大性能を叩き出している。逆に Pentium 4 ではせいぜい 500〜600Mbps 程度の速度しか得られていない。

次に，MPI 通信において，GbE と Node 内通信の両方を伴う VTPCC と Pentium D cluster の，No 内通信の速度を計測してみる。この場合，実際には Node 内においては共通の RAM を読み出すことになるから通信を行う必要はないため，事実上の overhead がここで生じることになる。その結果を図 1.10 に示す。

Xeon に比べて Pentium D はメモリ転送能力や Cache メモリサイズが大きいこと

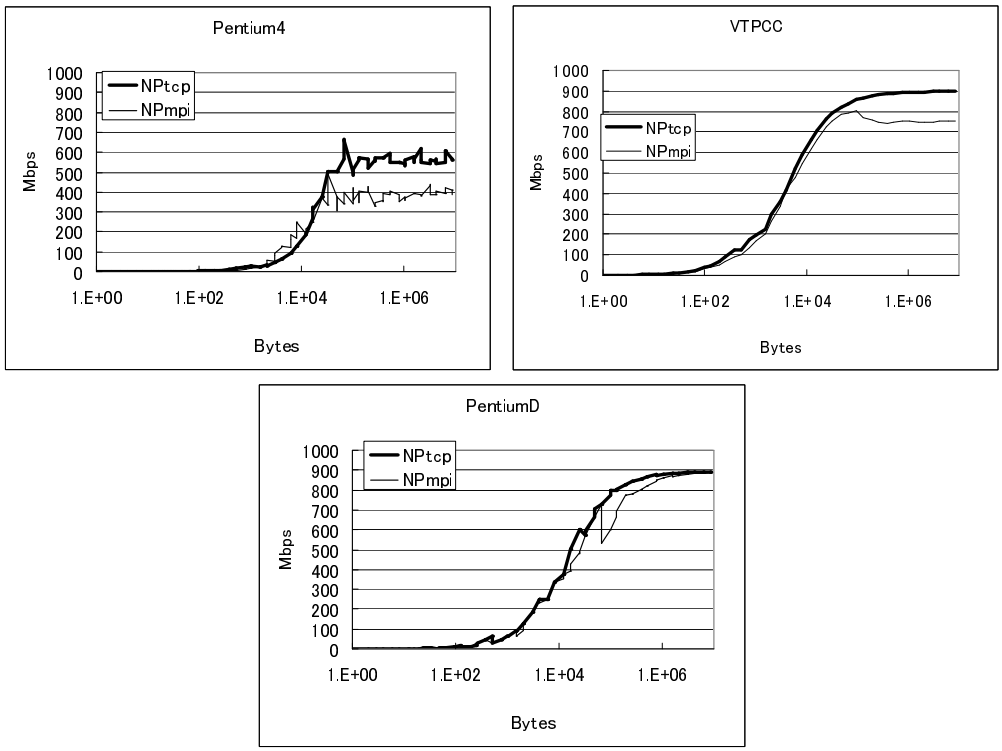

図 1.9: GbE の性能 (Pentium 4, Xeon(VTPCC), Pentium D)

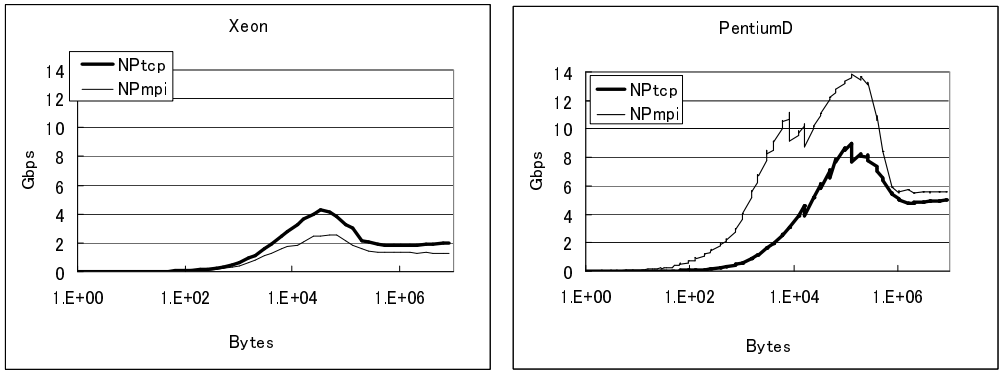

図 1.10: SMP/Multi-core のノード内転送性能 (Xeon(VTPCC), Pentium D)

が寄与し，相当な差をつけていることが分かる。しかしどちらも図 1.9 と比べて数倍以上も高速であるから，GbE を使用するようなデータ転送を大量に伴う並列計算を行う時には，この Node 内通信性能差はあまり意味を持たないことも予想できる。

演習問題

1. 移流拡散方程式を離散化した連立一次方程式 (1.2) を実際に解け。例として $f(x) = 1, \nu = 1$ とせよ。
2. 実際に NetPIPE を使用して自分の PC やそれに繋がるネットワーク性能を計測し，その結果についての考察をまとめよ。