PVM による行列演算の並列処理化

-Kernel CG-

東海大学工学部通信工学科 清水研究室

指導教員　清水 尚彦　講師

研究者　20et3111　木戸 純一

提出日　平成 8 年 3 月 8 日

Abstract

単一チップでのプロセッサの性能向上に限界が来つつある現在, 複数のプロセッサの並行動作によりプロセッサ 1 台あたりの計算量を減少させ, 処理全体にかかる時間を短縮する並列処理に期待が高まっている. 並列処理には専用の並列化プログラムを記述しなければならないが, 並列処理向けのプログラムは与えられた問題の分割方法とプロセッサへの割当方法によりその性能が大きく変化するため, 逐次プログラムに比べてより多くの検討を必要とする.

本稿では NAS Parallel Benchmark 内の kernel CG を対象として, kernel CG を並列化するのに最適な分割方法の検討を行った. また, いくつかの高速化の手法をあげ, それぞれについてその有効性を検討し, 有効と思われるものについて通信工学科計算機室と 12 号館のネットワークに実装して実行時間の測定を行った. その結果, 通信工学科計算機室のにおいては並列科後, 直接メッセージ送受信の採用と演算後送信のオーバーラップによって処理を高速化し ,4 台分割実行時に無分割実行時の 3.6 倍, すなわち台数効果の約 90%を得ることに成功した. しかし ,12 号館計算機室においてはマシンと通信路の性能のバランスが et001 ほど良好でなく, 並列実行時の性能が無分割時の性能を下回ってしまった. このことから ,kernel CG の実行を効率良く行えるマシン台数とマシン性能および通信速度の関係を定式化し, 台数効果の条件を明確にすることで 12 号館での実行時間の遅延の検討を行った.

目 次

図 目 次

表 目 次

Chapter 1

はじめに

1946 年に世界で最初のコンピュータ **ENIAC** が開発されて以来, 現在にいたるまでコンピュータは着実に高速化の道をたどってきた. もちろん現在においてもプロセッサの高速化の研究は続いており, 日を追うごとに高速なプロセッサが誕生している.

しかしその反面, 現在では論理素子の速度が物理的限界に近づいており, 単一チップでの高速化にもいずれは歯止めがかかることになる. このため, 更なる高速化を図るための新たなアーキテクチャとして, 並列処理が注目を集めている.

並列処理の最も基本的な概念は, 逐次処理で n ステップ必要な計算を, p 台のプロセッサを使って $\frac{n}{p}$ ステップで計算するというものである. つまり, 分散処理のように一つのプロセッサにタスク[1]の全部を実行させるのではなく, タスク自身をいくつかの要素に分割し, 各プロセッサを一つの目的に向けて協調して動作させる. このため, 価格対性能比や柔軟性, 拡張性, 信頼性に優れたシステムの構築が可能になっている.

しかし, 並列処理には一般的な逐次処理とは根本的に異なった考え方が必要であり, いくつかの新しい手法を用いなければならない. 多くの場合, 問題を分割したことにより各プロセッサでのデータ不足 (必要なデータが分割時に他プロセッサに渡っている.) が発生し, プロセッサ間での通信が必要になる. このプロセッサ間通信を考慮した効果的な分割方法の検討や, 問題のプロセッサへの割り当て方法など, 逐次プログラムには存在しない手法が最大の問題となる.

本研究では,kernel CG にこれらの手法を取り入れ, 基本的な行列の分割方法とそのときの計算内容及び通信内容を示した上で, 並列処理時の最大のネックとなる通信量が最小となる分割方法の検討を行った. そして, 決定した分割方法について, いくつかの高速化の手

[1]問題実行時における独立した処理のまとまり.

法を検討し, 効果があると思われるものについて通信工学科計算機室および 12 号館計算機室に実装し, 結果の評価を行った. また, その測定結果より, 高速化手法の実際の効果の検証と, 希望する性能を得るための条件つまりマシンの計算速度と通信路の通信速度の関係の定式化を行った.

Chapter 2

研究テーマ

2.1 NAS Palallel Benchmark

NAS Parallel Benchmark は,NASA Ames Reserch Center が Numerical Aerodynamic Simulation(航空力学計算シミュレーション) のプログラムをもとに作成した, 並列コンピュータを対象としたベンチマークのセットである. NAS Parallel Benchmark は,5 種類の kernel と, 3 種類の CFD と呼ばれる応用問題の 2 種類の問題から成っている.

この NAS Parallel Benchmark は, また, 浮動小数点演算を 64bit で行うことや, 並列化が可能な Fortran-90 または C 言語のどちらかの言語で記述する等の基本的なルールを守りさえすれば, データの構造, アルゴリズムの選択, プロセッサの割当, メモリの使い方をベンチマークを実装する者が自由に選択できるといった特徴がある.

kernel に含まれる 5 種類の問題を以下に示す.

EP: “Embarrassingly Parallel”. 乱数の生成を行う.

MG: “MultiGrid”. 偏微分方程式をマルチグリッドアルゴリズムにより解く.

CG: “Conjugate Gradient”. 連立一次方程式を, 共役勾配法により解く.

FT: “Fourier Transform”. 偏微分方程式を 3 次元の離散フーリエ変換により解く.

IS: “Integer Sort”. 整数列の並べ変えを行う.

これらの kernel には,probrem size として, 3 種類の異なった入力パラメータのセットが与えられており, その処理の規模により,Sample,Class A,Class B という名前が与えられている. kernel は, 従来の Livermore Loops や Linpack といったベンチマークよりも計算の規

模が大きいため, 並列マシンを評価するのに有用である. また, アルゴリズムが単純であるため, 新しいシステムへの実装が比較的容易であるといった特徴も持っている.

本研究は, この kernel の中の kernel CG を研究の対象とし, バス結合された分散メモリ型並列コンピュータ上で,kernel CG を最も高速に処理することができる並列化の方法を見つけだすことが目的である.

2.2 Karnel CG

2.2.1 概要

kernel CG は, 正値対称行列により構成された連立一次方程式を共役勾配法のアルゴリズムにより解くという処理を指定された回数行い, その処理にかかった時間を測定するというベンチマークテストである. この kernel CG アルゴリズム中の一連の計算の中で, 行列とベクトルの積を求める計算に処理時間のほとんどが費やされてしまう.. したがって, この部分の処理を高速に行うことが kernel CG の高速化の鍵であるといえる.

2.2.2 アルゴリズム

A を n 次の正値対称行列とする.x_jをベクトル x の j番目の要素とし, x^Tはベクトルの転置を表すものとする. また,$\|x\|$ をベクトルの絶対値, すなわち,$\|x\| = \sqrt{x^T x}$とする. これらの定義のもと,kernel CG のアルゴリズムを示すと, 以下のようになる.

$x = [1, 1, \ldots, 1]^T$
(測定開始)
DO $it = 1, niter$
　　連立一次方程式 $Ax = b$ を解く
　　$\|r\| = \|b - Ax\|$
　　$\zeta = \lambda + \dfrac{1}{b^T x}$
　　$it, \|r\|, \zeta$を表示する
END DO
(測定終了)

行列のオーダー n, 全体の反復回数 $niter$, パラメータλは, probrem size により表 2.1のような異なった値が与えられている. “連立一次方程式を解く” 部分は, 共役勾配法 (CG 法) を

Table 2.1: kernel CG の入力パラメータ

Size	n	$niter$	NONZER	λ
Sample	1400	15	7	10
Class A	14000	15	11	20
Class B	75000	75	13	60

使って行う. この部分のアルゴリズムは, 以下のようになる.

$r = b$

$\rho = r^T r$

$p = r$

DO $i = 1, 25$

$q = Ap$

$\alpha = \dfrac{\rho}{p^T q}$

$x = x + \alpha p$

$\rho_0 = \rho$

$r = r - \alpha q$

$\rho = r^T r$

$\beta = \dfrac{\rho}{\rho_0}$

$p = r + \beta p$

END DO

このCGのアルゴリズム中で,Aが行列,b, x, r, p, qがベクトル, それ以外の記号は全てスカラーである.

kernel CGでのベンチマークの正当性の確認は,ζによって行われる. itの各反復ごとに残差$\|r\|$とともにζを表示し,$niter$回の反復の後,ζの値を表2.2に示す値と比較し, $|\zeta - \zeta_{REF}| \leq 10^{-10}$であれば一連の処理が正しく行われたということになる. ただし, 当然のことではあるが, 計算の対象となる行列Aに, 毎回同じものを使わなければ同じ結果を得ることはできない. そのため,kernel CGには行列生成のためのルーチンが提供されている. このルーチンは`makea`と呼ばれ, マシンのアーキテクチャとは無関係に, 常に同じ行列を生成することができる. この`makea`に, 表2.1に示すnおよびNONZERの2つのパラメータを入力する

Table 2.2: kernel CG の出力パラメータ

Size	nonzeors in A	ζ_{REF}
Sample	78148	8.59717750786234
Class A	1853104	17.13023505380784
Class B	13708072	22.712745482078

ことで, 表 2.2の nonzeros in A に示す数の 0 でない要素が行列中に生成される. 表 2.2に示すように行列の要素の総数にくらべ,0 でない要素の数は極めて少ない. そのため, `makea` ではこの 0 でない要素とそのインデックスのみをストアするようにし, 必要なメモリ容量の節約をしている.

2.2.3 共役勾配法 (CG 法)

共役勾配法は, 連立一次方程式

$$Ax = b$$

を解くために用いられる方法のひとつである. ただし,A はオーダー n の係数行列で,x は要素数 n の未知ベクトル, b は要素数 n の定数ベクトルである. この方法は, 解が存在する場合には必ずその解を求めることができる. もともと, 対象とする行列 A は, 正則で正値対象であることが前提であったが, 任意の A に対しても適用できるように容易に拡張することができる.

この共役勾配法は, 最大 n ステップの反復法である. つまり, 計算が完全に行われるならば,$m \leq n$ の m 回の反復でひとつの解に収束する. また, 共役勾配法の一連の処理のなかで, 次のような長所を持っている.

- 計算手続きが単純である.
- 計算途中での係数行列 A に対する変更がない. つまり, 扱う行列がひとつでよい.
- 中間結果を蓄えるのに必要な記憶場所が小さくてすむ.
- 計算のどの段階においても新しくスタートすることができる.
- 各反復ごとの解が, 先行する解に比べて真の解に近い.

これらは, 共役勾配法を他の反復計算法に比べた場合の優れている点でもある.

Chapter 3

CG アルゴリズムの並列化

並列化の方法には, 機能並列とデータ並列がある. 機能並列は, アプリケーションを異なる複数のタスクに分割し, そのタスクを各プロセッサに振り分ける方法である. この方法は, 同じデータに対して各タスクが異なる処理を施す場合に適している. データ並列は, 扱うデータそのものを分割し, 各プロセッサに分散される. 分散メモリ型マルチプロセッサシステムでは, 全マシンで実行する並列プログラムをひとつだけ記述すればよいことや, プロセッサ台数の変化に対する対応が容易であるなどの理由から, データ並列が一般的に用いられている. したがって,kernel CG の並列化はデータ並列を用いて行うこととする.

kernel CG のアルゴリズムは, 主に行列×ベクトル, ベクトルの内積, スカラー×ベクトル, ベクトル+ベクトルの 4 つの計算からなっている. これら 4 種の計算の内容は, それぞれ以下のように行われる.

- 行列×ベクトル

$$\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0n} \\ a_{10} & a_{11} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n0} & a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} a_{00} \times y_0 + a_{01} \times y_1 \cdots a_{0n} \times y_n \\ a_{10} \times y_0 + a_{11} \times y_1 \cdots a_{1n} \times y_n \\ \vdots \\ a_{n0} \times y_0 + a_{n1} \times y_1 \cdots a_{nn} \times y_n \end{pmatrix}$$

- ベクトルの内積

$$c = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix}^T \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix} = (x_0 \times y_0 + x_1 \times y_1 \cdots x_n \times y_n)$$

- スカラー×ベクトル

$$\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} = c \times \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} c \times y_0 \\ c \times y_0 \\ \vdots \\ c \times y_n \end{pmatrix}$$

- ベクトル+ベクトル

$$\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_0 + y_0 \\ x_1 + y_1 \\ \vdots \\ x_n + y_n \end{pmatrix}$$

配列を分割するということは, 上記の計算を複数のプロセッサに分散させて行うということになる.kernel CG の実装対象は分散メモリ型の並列コンピュータであるため, 処理を進めていくうちに自分の持たない配列が指定された場合にはプロセッサ間通信[1]が必要になる. 上記 4 種類の計算のうち, 行列×ベクトルとベクトルの内積の計算にはプロセッサ間通信が必要になる. 以下, 配列のマッピング方法として, 行分割, 列分割, 格子状分割の 3 種類の分割法について, 分割時の各プロセッサで行われる処理の内容と, 必要になるプロセッサ間通信の内容について検討する. そして, その結果から kernel CG に最適な配列の分割方法を決定する.

3.1 配列のマッピング方法

3.1.1 行分割

行分割は, 図 3.1のように, 行列とベクトルの各要素を行方向に分割し, 各プロセッサにストアさせる方法である. 図 3.1は,6 行 6 列の正方行列と, 要素数 6 のベクトルを 3 プロセッサで行分割した場合であり, a_{ij},b_iはそれぞれ行列 A, ベクトル b の要素である. 図 3.1中では, ベクトルは b となっているが, b のみが図 3.1のように分割されるのではなく, 計算過程で登場する全てのベクトルが b と同様に分割され, ストアされる. また,p_iは割り当てられるプロセッサの番号を表している. 各プロセッサは, 参照する配列のインデックスが異なるだけで, 基本的には全て同じ内容の処理を行い, 必要とあればプロセッサ間通信を行い, 計算を進める. ここで,x,yというベクトルと,c というスカラーを仮定し, 各プロセッサでの動作を説明する.

[1]プロセッサ間のデータのやりとり

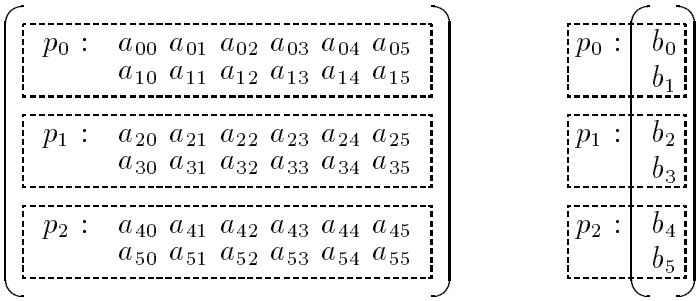

Figure 3.1: 行列·ベクトルの行分割

スカラー×ベクトルの計算は, スカラー c が, あらかじめ各プロセッサに分配されていれば, 参照するベクトル yは全て自らがストアしているので, プロセッサ間の通信無しに計算を行うことができる. 各プロセッサでの処理を, 図 3.2に示す.

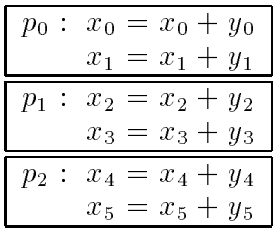

Figure 3.2: スカラー×ベクトル計算時の各プロセッサでの演算 (行分割)

ベクトル+ベクトルの計算も同様で, 扱う行列 x,yは全て自らがストアしているものであるため, 各プロセッサが独自に行うことができる. ベクトル+ベクトルの計算時の各プロセッサでの処理を, 図 3.3に示す.

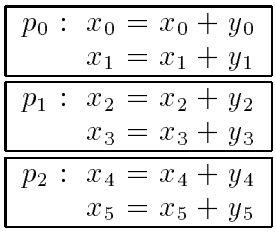

Figure 3.3: ベクトル+ベクトル計算時の各プロセッサでの演算 (行分割)

一方, ベクトルの内積計算と行列×ベクトルの計算を完全に終えるためには, プロセッサ間通信が必要になる. ベクトルの内積に関しては, 図 3.4に示すように, 各プロセッサが自ら

がストアしている分のベクトルの内積をとり, その計算結果を全て足し合わせることによってベクトルの内積を計算することができるが, そのときにプロセッサ間通信が必要になる. また, その結果を各プロセッサが必要としているときは, 結果を集計したプロセッサが全プロセッサに対し, 結果をブロードキャスト[2]送信しなければならないため, このときにもプロセッサ間通信が必要になる. 集計役のプロセッサをプロセッサ 0 とした場合のベクトルの内積計算時に発生する通信イメージを図 3.5に示す. ただし, 図 3.5中の c_1,c_2はそれぞれ各プロセッサでの計算結果であり, c はベクトルの内積計算の解である.

$$c = x^T y = \boxed{\begin{array}{c} p_0 \\ x_0y_0 + x_1y_1 \end{array}} + \boxed{\begin{array}{c} p_1 \\ x_2y_2 + x_3y_3 \end{array}} + \boxed{\begin{array}{c} p_2 \\ x_4y_4 + x_5y_5 \end{array}}$$

Figure 3.4: ベクトルの内積計算時の各プロセッサでの演算

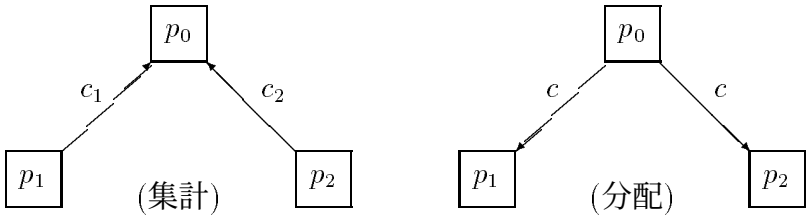

Figure 3.5: ベクトルの内積計算時に発生するプロセッサ間通信

行列×ベクトルの処理は,kernel CG のアルゴリズムの中で計算ステップ数, 通信量ともに非常に大きく時間がかかるため, 熟慮すべき部分である. 行列×ベクトルの計算時の各プロセッサでの処理を図 3.6に示す.

図 3.6中のプロセッサ 0 の動作において, ベクトルの要素 y_0, y_1はプロセッサ 0 がストアしているが, それ以外の y_2, y_3, y_4, y_5は, 他のプロセッサがストアしているため, プロセッサ間通信により $y_2 \sim y_5$を手にいれなければ, 計算を完了することができない. プロセッサ $1, 2$ についても同様で, プロセッサ 1 では y_2, y_3以外, プロセッサ 2 では y_4, y_5以外をプロセッサ間通信により手にいれないと, 計算を行うことができない. つまり, 計算を始める前に, 各プロセッサは, 自分以外のプロセッサに向けて自分の持つベクトルの要素をブロードキャスト送信する必要がある. このときの通信イメージを図 3.7に示す.

[2]同一のデータを複数の相手に向かって送信すること.

$$
\begin{array}{|ll|}
\hline
p_0: & x_0 = a_{00}y_0 + a_{01}y_1 + a_{02}y_2 + a_{03}y_3 + a_{04} + y_4 + a_{05}y_5 \\
 & x_1 = a_{10}y_0 + a_{11}y_1 + a_{12}y_2 + a_{13}y_3 + a_{14} + y_4 + a_{15}y_5 \\
\hline
p_1: & x_2 = a_{20}y_0 + a_{21}y_1 + a_{22}y_2 + a_{23}y_3 + a_{24} + y_4 + a_{25}y_5 \\
 & x_3 = a_{30}y_0 + a_{31}y_1 + a_{32}y_2 + a_{33}y_3 + a_{34} + y_4 + a_{35}y_5 \\
\hline
p_2: & x_4 = a_{40}y_0 + a_{41}y_1 + a_{42}y_2 + a_{43}y_3 + a_{44} + y_4 + a_{45}y_5 \\
 & x_5 = a_{50}y_0 + a_{51}y_1 + a_{52}y_2 + a_{53}y_3 + a_{54} + y_4 + a_{55}y_5 \\
\hline
\end{array}
$$

Figure 3.6: 行列×ベクトル計算時の各プロセッサでの演算 (行分割)

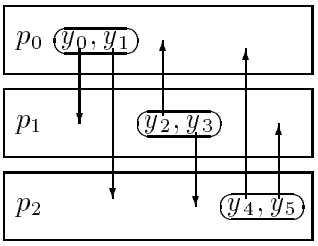

Figure 3.7: 行列×ベクトル計算時に発生するプロセッサ間通信 (行分割)

3.1.2 列分割

列分割は, 図 3.8のように, 行列を列方向に分割する方法である. 6 行 6 列の行列を 3 プロセッサで列分割した場合を図 3.8に示す. ベクトルは, 行分割と同じ くベクトルの要素を p 等分し, 各プロセッサにストアする.

行列の分割方法は異なるものの, ベクトルの分割方法が同じであるためにベクトルの内積, スカラー×ベクトル, ベクトル+ベクトルの計算時の各プロセッサでの処理および通信は, 行分割の場合と同じになる.

行列×ベクトルの計算時の各プロセッサの処理は, 図 3.9に示すようになる. 行分割と異なり, 計算開始前のプロセッサ間通信は不必要だが計算を完全に終えるためには各プロセッサでの計算結果を全て足し合わせる必要がある. つまり, 各プロセッサは他のプロセッサがストアすべき部分のベクトルをそのプロセッサに向けて送信しなければならず, プロセッサ間通信が必要になる. このときの通信イメージを図 3.10に示す.

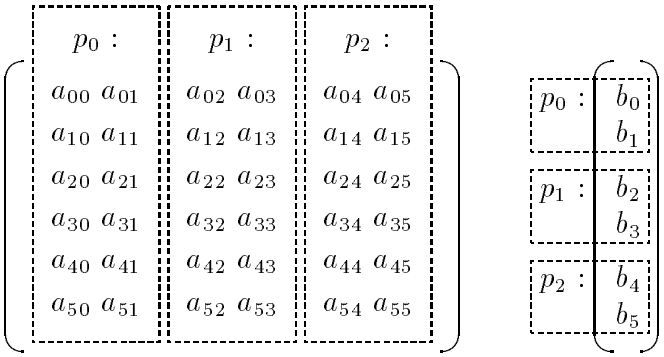

Figure 3.8: 行列·ベクトルの列分割

	p_0		p_1		p_2
$x_0 =$	$a_{00}y_0 + a_{01}y_1$	$+$	$a_{02}y_2 + a_{03}y_3$	$+$	$a_{04}y_4 + a_{05}y_5$
$x_1 =$	$a_{10}y_0 + a_{11}y_1$	$+$	$a_{12}y_2 + a_{13}y_3$	$+$	$a_{14}y_4 + a_{15}y_5$
$x_2 =$	$a_{20}y_0 + a_{21}y_1$	$+$	$a_{22}y_2 + a_{23}y_3$	$+$	$a_{24}y_4 + a_{25}y_5$
$x_3 =$	$a_{30}y_0 + a_{31}y_1$	$+$	$a_{32}y_2 + a_{33}y_3$	$+$	$a_{34}y_4 + a_{35}y_5$
$x_4 =$	$a_{40}y_0 + a_{41}y_1$	$+$	$a_{42}y_2 + a_{43}y_3$	$+$	$a_{44}y_4 + a_{45}y_5$
$x_5 =$	$a_{50}y_0 + a_{51}y_1$	$+$	$a_{52}y_2 + a_{53}y_3$	$+$	$a_{54}y_4 + a_{55}y_5$

Figure 3.9: 行列×ベクトル計算時の各プロセッサでの演算 (列分割)

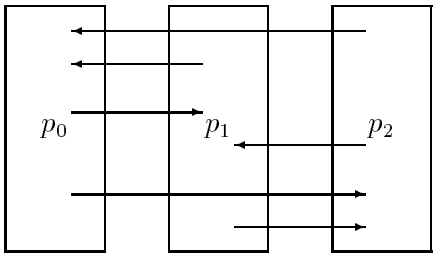

Figure 3.10: 行列×ベクトル計算時に発生するプロセッサ間通信 (列分割)

3.1.3 格子状分割

格子状分割は, 図 3.11のように, 行列を格子[3] 状に分割する方法である. 図 3.11は,6 行 6 列の行列を 9 プロセッサで分割した場合であり, ベクトルは行列の左上から右下に向かって引いた対角線上にあるプロセッサ (以後, このプロセッサを対角プロセッサと呼ぶ) に$\sqrt{p}$(p はプロセッサ台数) 等分され, ストアされる.

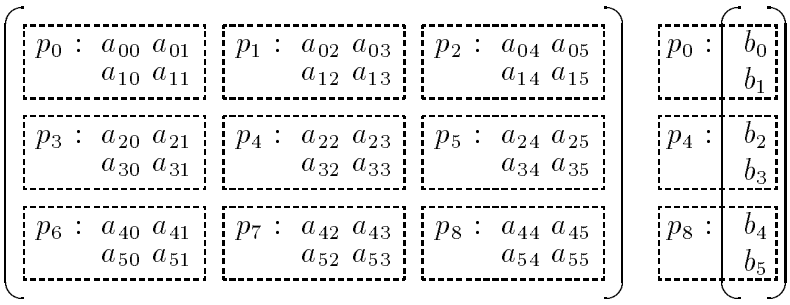

Figure 3.11: 行列·ベクトルの格子状分割

ベクトルは, 対角プロセッサにしかストアされないが, その分割方法自体は行分割·列分割と何等変わりがないので, ベクトルの内積, スカラー×ベクトル, ベクトル+ベクトルの計算での各プロセッサでの処理と通信は, 対角プロセッサにおいて行分割·列分割と同様に行われる.

一方, 行列×ベクトルの計算時の各プロセッサの処理は, 図 3.12に示すようになる. 図 3.12において, 対角プロセッサでの計算に使う行列およびベクトルの要素は対角プロセッサ自身が持っているのでそのまま行うことができるが, それ以外のプロセッサは, ベクトルの要素を手にいれてからでないと計算を進めることができない. つまり, 計算を始める前に, 対角プロセッサは同列上の自分以外のプロセッサに向けて自分の持つベクトル要素をブロードキャスト送信する. このときの通信イメージを図 3.13(a) に示す. その後, 各プロセッサは自分の担当する計算を行うが, ここまでででは行列×ベクトルの処理は完了しない. 処理を完了させるには, 同行上のプロセッサが持つ解のベクトルを全て加算する必要がある. この作業を, 各行の対角プロセッサが行うとすると, このときの通信イメージは図 3.13(b) のようになる. ただし, 図 3.13(b) 中の v_nは, 各プロセッサでの計算結果のベクトルである.

[3] メッシュ分割·ブロック分割とも呼ばれる.

$$
\begin{array}{rccccc}
 & \boxed{\begin{array}{c} p_0 \\ a_{00}b_0 + a_{01}b_1 \end{array}} & + & \boxed{\begin{array}{c} p_1 \\ a_{02}b_2 + a_{03}b_3 \end{array}} & + & \boxed{\begin{array}{c} p_2 \\ a_{04}b_4 + a_{05}b_5 \end{array}} \\
x_0 = & & & & & \\
x_1 = & a_{10}b_0 + a_{11}b_1 & + & a_{12}b_2 + a_{13}b_3 & + & a_{14}b_4 + a_{15}b_5 \\
 & \boxed{\begin{array}{c} p_3 \\ a_{20}b_0 + a_{21}b_1 \end{array}} & + & \boxed{\begin{array}{c} p_4 \\ a_{22}b_2 + a_{23}b_3 \end{array}} & + & \boxed{\begin{array}{c} p_5 \\ a_{24}b_4 + a_{25}b_5 \end{array}} \\
x_2 = & & & & & \\
x_3 = & a_{30}b_0 + a_{31}b_1 & + & a_{32}b_2 + a_{33}b_3 & + & a_{34}b_4 + a_{35}b_5 \\
 & \boxed{\begin{array}{c} p_6 \\ a_{40}b_0 + a_{41}b_1 \end{array}} & + & \boxed{\begin{array}{c} p_7 \\ a_{42}b_2 + a_{43}b_3 \end{array}} & + & \boxed{\begin{array}{c} p_8 \\ a_{44}b_4 + a_{45}b_5 \end{array}} \\
x_4 = & & & & & \\
x_5 = & a_{50}b_0 + a_{51}b_1 & + & a_{52}b_2 + a_{53}b_3 & + & a_{54}b_4 + a_{55}b_5
\end{array}
$$

Figure 3.12: 行列×ベクトル計算時の各プロセッサでの演算 (格子状分割)

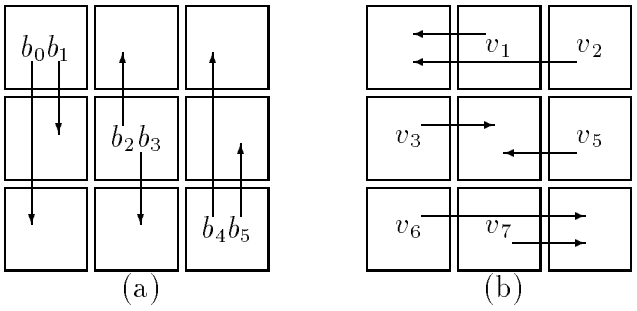

Figure 3.13: 行列×ベクトル計算時に発生するプロセッサ間通信 (格子状分割)

3.2 最適な分割方法

kernel CG を効果的に並列化するには, 分割時の各プロセッサでの演算ステップ数[4]や, ネットワークの通信形態･通信速度を考慮した上で, 最適な分割方法を選ばなければならない. 以下, 演算ステップと通信量の両面から, 最適と思われる配列の分割方法を検討する.

3.2.1 演算ステップ数による検討

どのような分割方法をとるにせよ, 行列およびベクトルのオーダーを n, 分割するプロセッサ数を p 個と決めれば, ストアする行列要素は同じ個数であるために行列×ベクトルの演算ステップ数はどの分割方法でも同じになる. 一方, ベクトルについては, 行分割および列分割ではプロセッサ 1 台あたり $\frac{n}{p}$ 個の要素をストアするのに対し, 格子上分割では $\frac{n}{\sqrt{p}}$ 個の要素をストアしなければならない. 当然,

$$\frac{n}{p} < \frac{n}{\sqrt{p}}$$

であるから, プロセッサあたりの演算ステップ数は格子上分割の方が多くなる. しかも, ベクトル演算 (内積･加算･積算) は対角プロセッサのみで行われてしまうため, ベクトル演算中は対角以外のプロセッサはアイドル状態[5]になってしまう.

しかし, シングルプロセッサ版 kernel CG のコード[6]をもとに, 各 Size における行列×ベクトルと, 全ベクトル演算にかかる演算ステップ数を計算したところ, 表 3.1の結果を得た. 表 3.1に示すように, 全演算中に占めるベクトル演算の割合はたいへん小さい. つまり, 格子状分割の場合のベクトル演算中の対角以外のプロセッサのアイドル時間は非常に短いということである. したがって,kernel CG では, 行列×ベクトルの計算時の演算ステップ数が実行時間に直接影響を与えると考えてよい.

ところが, 前述の通り行列×ベクトル計算の演算ステップ数は, プロセッサ台数が同じ場合に分割方法に関係なく同じになってしまう. つまり, 演算ステップ数だけを見た場合, 分割方法の違いによる実行時間の差はほとんどなくなってしまうのである. このため, 演算ステップ数による最適な行列の分割方法の決定は困難である.

---

[4]命令サイクル数ではなく, 計算回数を表すものとする. 例えば,$y = a * x + b$ という計算の場合, 演算ステップ数は 2 となる.

[5]何もしていない状態

[6]普通に C 言語で書いた kernel CG のコードのこと.

Table 3.1: 行列×ベクトルとベクトル演算の演算ステップ数の比較

Size	行列×ベクトル	全ベクトル演算	処理全体における ベクトル演算の割合 [%]
1400	60955440	5481000	9.00
14000	1445421120	54810000	3.79
75000	53461480800	1468125000	2.75

3.2.2 通信量による検討

本実験で使用する通信形態はバス結合であるから, 複数のプロセッサが同時にメッセージ[7] を送ろうとした場合, 通信路の競合が発生する. そのため, 通信路の使用権のないプロセッサは通信路が空くのを待ってからメッセージの送信を開始することになる. 当然, 通信路が空くのを待つ間プロセッサはアイドル状態になるので, 処理中の通信量が多くなるほど CG の性能は落ちてしまう. したがって, 最も通信量が少なくなるような分割方法を選ぶことが kernel CG の高速化につながることになる.

前述のように, 処理中にプロセッサ間通信が必要な処理は, 行列×ベクトルと, ベクトルの内積の 2 種類の計算である. このうち, ベクトルの内積の計算時に発生する通信は, 各プロセッサから結果であるスカラーを集めるときに発生する. 演算を 64bit(8byte) で行った場合, スカラーは 8byte の大きさであるからプロセッサの台数を p 台とすると, その総通信量は $8 * (p-1)$byte である. これは, 後述する行列×ベクトルの処理時に発生する通信量に比べると, 微々たるもので, この通信によって通信路が busy になることはないと考えて差し支えない.

一方, 行列×ベクトルの処理時に発生する通信量は膨大である. いつ, どのような通信が必要になるかはすでに述べたので, ここではその通信量を各分割ごとに定式化して示す. ただし, 行列のオーダーとベクトルの要素数を n, プロセッサ台数を p とする.

- 行分割

  行分割における通信は, 図 3.7に示すように, 各プロセッサがストアしているベクトルを自分以外のプロセッサに分配するときに生じる. このとき,1 台のプロセッサは, 自分以外のプロセッサに $\frac{n}{p}$個のベクトル要素を送る. これを全プロセッサで行い, また演算は全て 64bit(8byte) で行うことになっているので, 行分割における行列×ベクト

[7] 通信路を移動するデータのこと.

ルの処理時の総通信量 c_v^{row}は, 次式で表される.

$$\begin{aligned} c_v^{row} &= 8 \times \frac{n}{p} \times p \times (p-1) \\ &= 8n(p-1) \qquad [byte] \end{aligned} \tag{3.1}$$

- 列分割

  列分割における通信は, 図 3.10に示すように, 各プロセッサでの結果の集計を行うときに発生する. このときに各プロセッサは,$\frac{n}{p}$個の要素を自分以外の全てのプロセッサに送信する. したがって, 列分割における行列×ベクトルの処理時の総通信量 c_v^{colmun}は, 次式で表される.

$$\begin{aligned} c_v^{column} &= 8 \times \frac{n}{p} \times p \times (p-1) \\ &= 8n(p-1) \qquad [byte] \end{aligned} \tag{3.2}$$

- 格子状分割

  格子状分割における通信は, 図 3.13(a)·(b) に示すように, 対角プロセッサが計算用のベクトルを分配するときと, 非対角プロセッサの計算結果を集計するときに生じる. 分配·集計時の各通信におけるメッセージは, 要素数$\frac{n}{\sqrt{p}}$個のベクトルである. この大きさのメッセージを, 各対角プロセッサが$\sqrt{p}-1$ 回送信し, $\sqrt{p}-1$ 回受信することになる. したがって, 格子状分割における行列×ベクトルの処理時の総通信量 c_v^{block}は, 次式で表される.

$$\begin{aligned} c_v^{block} &= 8 \times \frac{n}{\sqrt{p}} \times (\sqrt{p}-1) \times \sqrt{p} \times 2 \\ &= 16n(\sqrt{p}-1) \qquad [byte] \end{aligned} \tag{3.3}$$

これら 3 つの式を Class A のときにあてはめ, 行列×ベクトルの処理時のプロセッサ台数に対する総通信量を計算すると, 表 3.2に示す結果が得られる. 表 3.2中の通信量の欄で, 通信量が—となっているのは, 行列のオーダー n が,p あるいは$\sqrt{p}$で割り切れないためである.

表 3.2に示すように, 通信量が最も少なくなるのは, 行列を格子上に分割した場合であり, その通信量は, 他の分割方法に比べて極めて小さい. プロセッサ台数が多くなるほどその差は顕著になり, プロセッサ 100 台の場合では, 格子状分割の通信量は行分割·列分割の$\frac{1}{5}$程度にまで小さくなる. 同時に 1 つのメッセージしか存在できないバス結合という形態を

Table 3.2: 行列×ベクトル処理時のプロセッサ台数に対する通信量

p	通信量 [kbyte]		
	c_v^{row}	c_v^{column}	c_v^{block}
2	112	112	—
4	336	336	224
10	1008	1008	—
16	1680	1680	672
20	2128	2128	—
25	2688	2688	896
40	4368	4368	—
64	—	—	1568
80	8848	8848	—
100	11088	11088	2016

もつ通信路上で, この通信が Sample,Class A では $(25+1)\times 15 = 390$ 回, Class B では $(25+1)\times 75 = 1950$ 回反復されることを考えると, 通信量が大きくなる行分割·列分割は実用的でないといわざるを得ない. したがって, 最適な分割方法は, 通信量の最小となる格子状分割と決定する.

ただし, 格子状分割の場合には行列のオーダーが$\sqrt{p}$で割り切れなければならない[8]ため, 実装できるプロセッサ台数が限られてしまうという欠点もあるが, これは行列を$\sqrt{p}$で割り切れるように拡張する等の方法により, pが整数の平方根をもつというところまで条件を緩和できる.

[8]行·列分割では,n が p で割り切れればよい.

Chapter 4

PVM の実装

4.1 PVM

PVM は,**Parallel Virtual Machine** の略で, ネットワークに接続された UNIX ベースのコンピュータ群を単一の並列コンピュータと見なし, 利用することを可能にするソフトウェアである. コンピュータ群は, 全てが同じ機種である必要はない. したがって, 低レベルの端末から高性能のベクトルコンピュータに至るまでを, 同一の線上において並列処理を行える, というところに,PVM の特徴がある.

PVM のもとでは, コンピュータ群を 1 台の分散メモリ型の並列コンピュータと見なされ, この分散メモリ型並列コンピュータをバーチャルマシン呼ぶ. PVM は, このバーチャルマシン上にタスクを生成し, タスク間の通信と同期をサポートする. したがって, これまでプロセッサ間通信と呼んでいたものは, PVM によるタスク間通信に他ならない. 実装者は,C 言語または Fortran でアプリケーションを記述し, PVM が提供するタスク間通信のためのメッセージパッシングライブラリを利用して並列化を行う.

4.2 並列化 kernel CG の構成法

データ並列モデルにおけるアプリケーションの構成法は, 大きく 2 種類に分けられる. マスター/スレーブ方式と, 単一プログラム多重データ (SPMD:Single Program Multiple Data) 方式である.

マスター/スレーブ方式では, マスタープログラムが複数のスレーブプログラムを生成･制御し, 各スレーブプログラムが計算を行う. したがって, 実装者は複数のプログラムを記述する必要がある.

SPMD 方式では, プログラムはひとつである. 実行された単一のプログラムは自分自身のコピーをタスクとして複数生成し, 各プロセッサ上で実行する. したがって, 各タスクは対等であり, 他のプロセスと協調しながらデータを分割して処理する.

kernel CG のコードの並列化は, このどちらを用いても可能である. マスター/スレーブ方式をとる場合には, マスターを対角プロセッサ, スレーブを非対角プロセッサと位置づけ, 各々の動作を別々に記述することにより並列化できる. SPMD 方式では自分がマスターかそれともコピー (マスターにより生成されたタスク) であるかを判断させ, それぞれに対応する処理を実行させる形になる. したがって, マスターとコピーの少なくとも 2 種類の動作をひとつのプログラム上に記述することになる. どちらの方式を用いても結果は同じだが, 複数のプログラムを書くマスター/スレーブ方式では, コンパイルやデバッグを各プログラムに対して行わなければならない. 一方,SPMD 方式では, コンパイルやデバッグは唯一つのプログラムに対してのみ行えばよいので, 作業は楽である. したがって,kernel CG の並列化コードは SPMD 方式により記述することとした.

4.3 実装

シングルプロセッサのコードとマルチプロセッサのコードの決定的な違いは, コード中にメッセージの送受信を行う命令を含んでいることである. したがって, シングルプロセッサのコードにメッセージの送受信を指示する命令を付加していことが並列化の主な作業になるが, そのほかにも, 子タスク (自分のコピー) の生成や, 行列およびベクトルの分割を行う部分も必要である.

まず, タスクの生成だが, これは PVM のライブラリに含まれるルーチンにより容易に実現できる. その後, マスタープロセスは生成された各タスクへ, 自分を含めた全タスクの ID とプロセッサ番号 (各プロセッサの行·列のインデックス) を送信する. タスクの ID は, プロセッサ間通信のときの送信先·受信先を指定するのに使い, プロセッサ番号は, 行列のストアする部分の決定や, 各プロセッサが対角プロセッサか非対角プロセッサかを判断するのに使う.

次に, 行列およびベクトルの分割だが, ベクトルに関しては初期値が決まっているので各プロセッサが独自に配列を定義し, 初期値を与えてやればよい. 一方, 行列は `makea` で作ることが決められている. そのため, マスタープロセスが一度 `makea` で完全な形の行列を作った後, 各プロセッサ番号に対応する部分行列の抽出と, その部分行列の送信をそれぞれの子

タスクに対して行わなければならない. つまり, マスタープロセスは, 部分行列を抽出し, その結果を対応する番号のプロセッサに送る, という作業を子タスクの個数回だけ繰り返すということである. 各プロセスが独自に行列を生成し, 不必要な部分を捨てるという方法も考えられるが, もしもバーチャルマシン中のマシンの中に行列の生成に必要なだけのメモリが確保できないものがあった場合, そのマシンでのみ行列の生成が不可能になってしまう[1]. kernel CG の測定時間の中には行列生成に要する時間は含まれていないので, この部分にはどれだけ時間をかけてもかまわない. したがって, 時間は少々多めにかかるものの, 多くのマシンにプログラムを対応させるには, マスタープロセスのみで部分行列の抽出を行い, 各プロセスに送信を行う方法が確実である.

そして, プロセッサ間通信命令の実装だが, これも PVM のライブラリに含まれているので実装そのものは容易である. しかし, 容易であるが故に誤りも発生しやすく, プロセッサ間通信の順番を考慮せずにやみくもに送受信命令を記述したりすると, 最悪の場合には実行時に全てのプロセッサが受信待ちの状態になり, 全プロセスが停止する “デッドロック” 状態に陥ってしまうようなコードになってしまわないとも限らない. 一度こういったエラーが発生すれば, どこに誤りがあるかはコードを 1 行ずつトレースしていく他なく, デバッグの手間も増えることにつながる. したがって, マスターと子, あるいは対角と非対角のそれぞれの場合について, 送信と受信を一つ一つ対応させながらコードを記述していくのが各プロセスに誤動作をさせない確実な方法である.

4.4 高速化

これまで説明した方法は, 単に kernel CG のコードを並列化するための方法であり, 通信によるプロセッサのアイドル時間の発生などの並列化に特有の問題を考慮してはいない. つまり, このコードにさらに手を加えることで, 実行時間を短縮できる可能性がある. 以下, いくつか考えられる高速化の手法と, その方法によって本当に高速化が可能なのかどうかを検討する.

4.4.1 ベクトル演算の高速化可能性

kernel CG 並列化のための行列分割に格子状分割を採用したとで, 各種のベクトル演算は対角プロセッサでのみ行われる. その間, 非対角プロセッサはアイドル状態になることはす

[1] 通信工学科計算機室のマシンがこれにあたる. et011〜et015 では, Class A の行列生成に必要なメモリが確保できないが, et001 には行列生成が可能なだけのメモリが装備されている.

でに述べたが, 各プロセッサでのベクトル演算をアイドル状態となっているプロセッサにさらに分散させることは可能である. しかし, そのためにはベクトル要素の分配と, 結果の収集のためのプロセッサ間通信が必要になる. 結果の収集とは, 非対角プロセッサで処理の終了した部分ベクトルを対角プロセッサが収集し, 要素数$\frac{n}{\sqrt{p}}$のベクトルに戻す作業である. もし, 負荷分散により短縮された実行時間が分配·収集のためのプロセッサ間通信よりも長ければ, 高速化が可能ということになる.

ベクトル演算中, 各対角プロセッサにおいて, 行方向あるいは列方向に$\sqrt{p}-1$ 個のプロセッサ (p はプロセッサ台数) がアイドル状態になっている. したがって,1 プロセッサあたりのベクトル演算のステップ数を$\frac{1}{\sqrt{p}}$にすることができる. 一方, プロセッサ間通信は,$\frac{n}{p}$の長さのベクトルを非対角プロセッサに送信する作業が全部で $p \times (p-1)$ 回行われる. また, 非対角プロセッサからの結果の収集も $p \times (p-1)$ 回行われるので, その通信量を c'とすると,c'は次式で表される.

$$\begin{aligned} c' &= 2 \times \frac{n}{P} \times 8 \times \sqrt{p} \times (\sqrt{p}-1) \\ &= 16(n - \frac{n}{\sqrt{p}}) \qquad [byte] \end{aligned} \tag{4.1}$$

これらの式をもとに,CG 法のアルゴリズム pp すなわち 25 回の反復部分において, 反復 1 回あたりの短縮される実行時間と, 通信にかかる時間を計算する.

実行時間を計算するとはいっても, 変数のレジスタへのリード/ライトや, 加減乗除の計算にかかる時間 (サイクル) が明確になっていないと, 実行時間を計算することはできない. したがって, 計算に使用したリード/ライトおよび加減乗除にかかるサイクル数を, 表 4.1に示す.

Table 4.1: リード/ライト·加減乗除にかかるサイクル数

操作·演算	サイクル数
$read \cdot write$	10
$read(cachehit)$	2
$+ \cdot -$	5
$*$	5
$/$	40

表 4.1の値を用いて, 無分割 (シングルプロセッサ) 時の 5 個のベクトル演算にかかるサイクル数を計算したところ, Class A で 4424000 サイクルという結果を得た. 無分割時で

4424000 サイクルであるから,p 台のプロセッサで実行した場合にはプロセッサ 1 台 (対角プロセッサ) あたりのサイクル数は$\frac{4424000}{\sqrt{p}}$になる. ここで検討している方法は, これをさらに分散させ, サイクル数をさらに$\frac{1}{\sqrt{p}}$にさせようとしているのであるから, プロセッサ 1 台あたりのサイクル数は$\frac{4424000}{p}$となり, 短縮されるサイクル数は, $\frac{4424000}{\sqrt{p}} - \frac{4424000}{p}$サイクルとなる.

これらより, プロセッサ台数に対する短縮される時間と増加する通信量を計算したものを表 4.2に示す. ただし,CPU のクロック周波数は 200[MHz], ネットワークの通信速度は 1[Mbyte/sec] とした.

Table 4.2: ベクトル演算分散時のプロセッサ台数に対する短縮時間と通信量

p	通信時間 [s]	短縮時間 [s]
4	0.112	0.005530
16	0.168	0.004148
25	0.179	0.003539
64	0.196	0.002419
100	0.202	0.001991

表 4.2に示すように, 新たにベクトル演算を分散処理させると, そのためにかかる通信時間が分散化により短縮される実行時間よりもはるかに大きくなってしまう. しかも, プロセッサ台数の増加にしたがい通信時間が増加しているのに対し, 短縮される演算時間は減少している. つまり, 実行時間短縮のための手法であるはずが, 逆に性能を大きく落としてしまう. したがって, ベクトル演算の分散化による高速化の可能性は無い.

4.4.2 直接メッセージ送受信

通常,PVM ではメッセージの送信･受信は送信･受信のためのバッファを介して行われる. つまり, 送信側で送信する配列内のデータを一度送信バッファに置き, その送信バッファデータを送り出す. そして, 受信側では送られてきたデータを受信バッファに置き, 受信バッファからデータを取り出し配列に格納する (4.1). この標準の送受信命令は `send`,`recv` と名付けられており, この標準送受信命令には, 送信バッファを利用することにより複数のデータをまとめて単一のメッセージとして送信できることや, 配列のストライド指定[2] ができる等の特徴がある. しかし, 受信側から見れば送信側のバッファリングの作業の分メッセージが自分のところにつくまでの時間が長くかかることになる. また, メッセージ到着前に受信命令

[2]配列の要素を指定された間隔で取り出す. 例えば, ある配列に対してストライドを 5 と指定した場合, 配列から 0,5,10,15,･･･番目の要素が取り出されてバッファに置かれる.

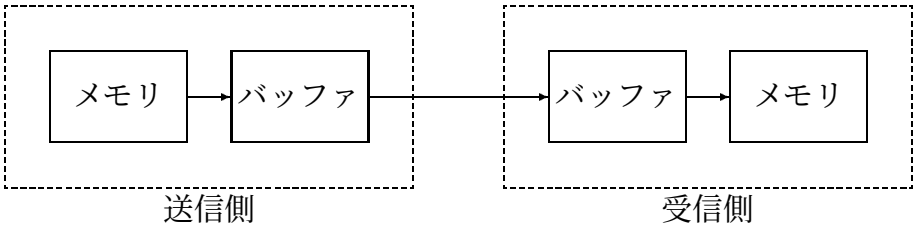

Figure 4.1: バッファを介したメッセージの送受信

が発効されていても, 到着したメッセージは強制的にバッファに入れられてしまうので, そこからデータを取り出すのに必要な時間がオーバーヘッドになってしまう.

そこで, もう PVM にはバッファを介さずにメッセージを送信·受信する命令も用意されており, これらは psend,precv と名付けられている. この直接送受信命令を用いると, 送信側ではメモリから読み込んだデータをそのままメッセージにパックして送信する 4.2. 一方, 受信側では送られてきたメッセージをそのまま指定されたアドレスに書き込む. ただし,

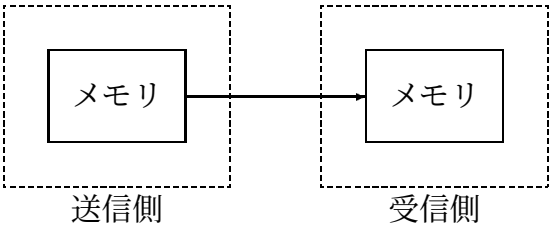

Figure 4.2: 直接メッセージ送受信

メッセージが到着した時点で受信命令が発効されていない場合には受信バッファが生成され, メッセージはそのバッファに入れられてしまうため, recv を用いた場合と変わらなくなってしまう. この直接メッセージ送信を使った場合, 受信側から見ると, 送信側でデータのバッファリングという作業を行わない分だけメッセージが到着するまでの時間は短くなる. また, メッセージ到着前に受信命令が発効された場合には, 自分自身がバッファリングを行わないために受信命令の発効から処理再開までの時間は短くてすみ, 全体の処理時間の短縮につながる. したがって, メッセージの送受信を直接メッセージ送受信で行うことで, 処理時間の短縮ができる可能性がある.

送信側で, データ送信時にバッファが必要であると考えられるのは, スカラーや, 要素数

の少ない配列等の独立した小さなデータを一度に数多く送信する場合である. 小さなデータを別々に通信路に送り出すと, その度にメッセージの起動オーバーヘッドがかかる上, 一つのメッセージの大きさが通信路のパケットの大きさよりもずっと小さい場合にはスループットが大きく減少してしまう. そのため, 一つ一つのデータを全てバッファに置いた上でそれら全てを単一の大きなメッセージとして送ることで, 通信路のスループットの低下を防ぐことができる. しかし, 行列を格子状に分割した場合に発生する通信において, 1 回の送信に含まれるのは$\frac{n}{\sqrt{p}} \times 8[\mathrm{byte}]$ の大きさの配列 (ベクトル) であり, そのまま送信してもスループットの低下を引き起こすほど小さくはないので, バッファリングの必要はないと考えられる. したがって, 処理中に発生する送信を全て psend,precv による直接メッセージ送信にすることで, バッファリングにかかる時間の分だけ一連の通信作業にかかる時間を減少させることができる.

受信側においては直接メッセージ受信による効果は大きいと考えられる. 直接メッセージ受信失敗時にはバッファリングが行われてしまうものの, 受信側でメッセージ到着時に受信命令が発効されていない状況では, まだ何らかの処理 (おそらく計算) が続いているわけで, その計算終了時にメッセージが到着していれば, たとえメッセージがバッファに入っていてもメッセージの “待ち” 時間を覆い隠すことができ, オーバーヘッドはバッファからデータを取り出す時間だけになる. ただし, この場合には直接メッセージ受信の効果は表れない. 一方, メッセージ到着前に受信命令が発効されたときは, その “待ち” 時間そのものはあきらめる他ないが, メッセージ到着からそのメッセージがメモリに格納されるまでの間にバッファを通過しない分, 標準の受信命令よりも処理再開までの時間はずっと短くてすむ. したがって, 直接メッセージ受信の失敗時にはバッファリングが自動的に行われ, 直接メッセージ受信成功時にはバッファリングによるオーバーヘッドを除去できるという理由から, 全ての受信を直接メッセージ受信で行うことが有効である.

4.4.3 通信と演算のオーバーラップ

オーバーラップとは, 通信と演算を部分的に重ね合わせて処理することで, オーバーヘッドの除去を行う手法である. 通信と演算のオーバーラップには, 送信と演算·受信と演算の 2 通りあるが, 受信と演算のオーバーラップは, 直接メッセージ受信失敗時に受信バッファにメッセージが入ることにより自動的に達成される. したがって, オーバーラップの処理を施すのは送信と演算に関してということになる.

PVM により提供されているメッセージ送信ルーチンはバッファ使用の有無に関わらずブ

ロック送信であるので, 送信作業中には他の一切の処理を行うことはできない. ブロックが解除されるのは, send では送信バッファの再使用が可能になった時点であり, psend ではメモリへの書き込みが許可された時点である. よって, ブロックが解除されるのはバッファあるいはメモリ中のデータを全て通信路上に送り出すまでの間であると解釈できる. これは, 受信側にメッセージが全て到着する時間よりも前にブロックが解除されることを意味しており, オーバーラップに関する対応を何も施さなくてもある程度のオーバーラップは自動的になされることになる. ここでは, 通信路が空かないうちから演算を開始してしまう方法について検討する.

送信と演算のオーバーラップを行うことのできる状況には 2 通りある. 一つは, 処理の順番が

送信 → 演算

である場合で, もう一つは

演算 → 送信

である場合である.

まず, 送信 → 演算 の場合についてその方法を検討する. 前述の通り, 通信路にメッセージを乗せ終わるまでは送信側の処理はブロックされる. もし, 何らかの方法でこのブロックされる時間を減らすことができれば, 送信側での演算開始を早めることができる. このときの様子を図 4.3に示す. ただし, 図中の演算時間と通信時間の比は適当であり, 通信の部分はメッセージを通信路に乗せるのに必要な時間である. kernel CG のアルゴリズム中で, この処理に対応するのが, 図 3.13 (a) の, 分配フェイズにおける対角プロセッサでの送信である. 分配フェイズでの対角プロセッサは, 列方向の $\sqrt{p}-1$ 個のプロセッサに対し, 自分の持つベクトル要素を送信する. このとき, 自分の担当する計算はブロックが解除されるまで実行できない. この計算を送信開始と同時に行うことができれば, 計算後に他プロセッサから送られてくるメッセージの直接受信に成功する可能性が高くなり, 全体の処理時間の短縮につながると考えられる.

では, その具体的なオーバーラップ方法だが, ノンブロッキングでメッセージを送信できるルーチンが存在しない以上, 単独のプロセスでオーバーラップを行うことは不可能である. したがって, マルチタスクを利用し, 同じプロセッサ上にもう一つプロセスをバックグラウンドで走らせてそのプロセス (以後, このプロセスをサブプロセスと呼ぶ) に送信を全て任せることで, もとのプロセスは送信を行わずにすぐに担当する計算を開始することができる.

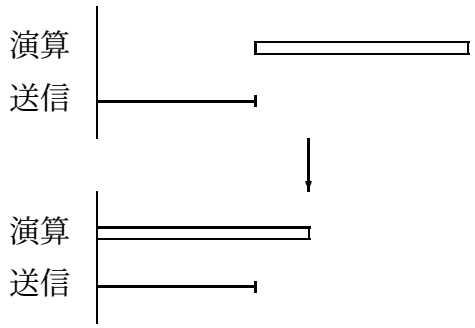

Figure 4.3: 送信と演算のオーバーラップ

このサブプロセスの生成方法だが, C 言語の `fork()` というルーチンによって自分のコピーを生成して, それをサブプロセスとする方法と, PVM のタスク生成機能 (`pvm_spawn()`) を使って生成したタスクをサブプロセスとする方法が考えられる. このうち,`fork()` を使った場合では, 処理を進めるうちにオーバーラップをさせたい処理が表れた場合にだけ `fork()` を呼び出してサブプロセスを生成する. サブプロセスの持つデータ (配列の内容) は, マスターがコピーを生成した時点で持っていた配列の内容に等しいため, サブプロセスは自分が生成された瞬間から送信を行うことが可能である. そして, 送信が終了した後はサブプロセスを消滅させて, 元のプロセスに合流させる. この繰り返しでオーバーラップが達成できる. ところが, 試験段階で `fork()` を実装して実行したところ, 数回に 1 回, 処理が停止してしまうという事態が発生した. この原因がサブプロセスの付加にあることは明かである. 原因追求の結果, 以下のような結論に達した.

マスタープロセス (この場合は, 対角プロセッサに割り当てられたプロセス) とサブプロセス (`fork` によるコピー) は, OS から見れば異なる ID を持つ別々のプロセスと認識されるが, PVM の管理のもとで `fork()` を使うと, タスク ID[3] までもがコピーされ,PVM からはどちらがマスターであるかの判別は不可能になってしまう. PVM では, 送信相手の指定は相手のタスクの ID を指定することで行うので, あるプロセスがマスタープロセスに向けて送信を行った時点でサブプロセスがまだ存在していた場合には, メッセージはマスターとコピーのどちらに届くかはわからない. もしコピーに届いた場合, メッセージはサブプロセスと共に消滅してしまうので, マスターにはメッセージは永遠に届かない. 全タスク中で, ひ

[3] タスク ID とは,PVM が自分が生成したタスクに付ける ID で, OS がプロセスに付けるプロセス ID とは全くの別物である.

とつでもタスクが停止してしまえば, その他のタスクもやがては受信待ちの状態になってしまうので, デッドロックが発生し, 全体の処理が停止してしまうのである. デッドロックの発生を防止するには, 少なくともサブプロセスが存在している間はどのプロセッサもマスタープロセスに対しては送信を行わないことを保証する必要がある. これは, 非対角プロセッサでの受信が全て終了した時点で非対角プロセッサがバリア同期を取ることで実現できるのだが, このときに処理が終了し, 解を返すことのできる状態にあるプロセスもブロックされるので, その分の遅延が発生してしまい, オーバーラップの効果を打ち消してしまう. そのため,`fork()` によりオーバーラップを実現しても, 全体の処理時間の短縮にはならなくなってしまうので, `fork()` による送信と演算のオーバーラップは実用的ではない.

したがって, オーバーラップの実現は PVM のタスク生成機能を利用して行うことになる. 具体的には,kernel CG のアルゴリズムに入る前に PVM によりサブプロセスを生成し, それをマスタープロセスと同じプロセッサ上に割り当てておく. `fork()` の場合と異なり, 始めから終わりまでずっとサブプロセスが存在し続けるため, 生成されたタスクに常にマスターと同じ内容の配列を持たせることはできない. このため, マスタープロセスの配列をサブプロセスに渡すという動作が必要である. これは,PVM のタスク間通信を使って行うほかないが, マスタープロセスとサブプロセスは同じプロセッサ上に存在するためにメッセージが通進路に出ることはない. つまり, 送信側のプロセスが “送信” という作業を行うことに変わりはないものの, この時の通信は非常に高速に行われるために送信側のプロセスがブロックされる時間は, 配列をそのまま送信した場合に比べて少なくなると考えられる. これにより, 送信 → 演算 の場合の送信と演算のオーバーラップが達成できる.

次に, 演算 → 送信 の場合のオーバーラップだが, 計算部分をいくつかに分け, その部分の計算が終了したらそこまでの結果をひとまとめにして送る, という作業を繰り返すことで, 演算と送信のオーバーラップができる. これは送信するデータが揃うのが演算かすべて終了した後であることから, 演算終了前に送信を完了することが不可能であるためで, 送信終了は演算終了よりも必ず後に来る. 演算と送信のオーバーラップの様子を図 4.4に示す. ただし, 図 4.4は演算を 4 回に分けた場合の例である. また, 図 4.3と同様に図中の通信時間はデータを通信路に送り出すまでの時間であり,, 演算時間と通信時間の比は適当である.

kernel CG のアルゴリズム中で, この処理に対応するのが図 3.13 (b) の集計フェイズの非対角プロセッサでの送信である. 非対角プロセッサは, 列方向の対角プロセッサよりベクトルを受け取って自分の担当する計算を終了した後, その結果を行方向の対角プロセッサに向けて送信する. このとき, 何も手を加えない場合には対角プロセッサは少なくとも非対

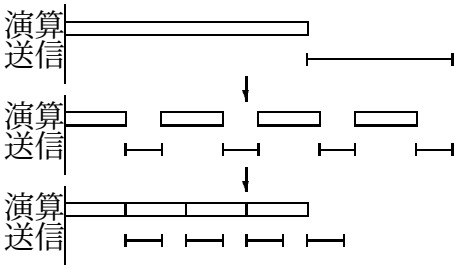

Figure 4.4: 演算と送信のオーバーラップ

角プロセッサの演算が終了し, 結果を送り出すまでの間ブロックされ, 受信待ちの状態になる. したがって, 非対角プロセッサにおいて早いうちから結果の送信を開始することができれば, 対角プロセッサでの受信待ちの時間を削減でき, 処理の高速化につながる.

送信→演算の場合は, 送信が終了しないうちに結果が帰ってくる可能性があるという理由から, `fork` によるオーバーラップは不可能であったが, 演算→送信の場合には送信が完了する前に他のプロセッサからメッセージが送られてくることはないので, オーバーラップのためのサブプロセス生成方法は, `fork` による方法でも PVM によるタスク生成でも構わない. このため, どちらか早い方を選ぶことになる. こればかりは実際に実行してみないとわからないため, `fork` 版と PVM 版の 2 種類のプログラムを記述し, 通信工学科計算機室の Sun WarkStation を 4 台使い, 行列の 4 分割時の実行時間を測定した. その結果を表 4.3に示す. ただし, 実行時の Probrem Size は Sample である.

Table 4.3: 演算と送信のオーバーラップ実装時の実行時間 (Sample)

生成方法	実行時間 [sec.]
`fork()`	110
`pvm_spawn()`	70

4.3より明らかなように, `fork()` を使った場合にはオーバーラップの効果どころか実行時間が長くなってしまう. これは,`fork()` により子プロセスを生成した場合, 親プロセスがそのとき持っていた配列の内容も全てコピーしなければならないため, 大量のメモリアクセスが発生する. しかも, これが送信 1 回ごとに繰り返されるため, その遅延は膨大なものとなる. 一方,PVM によるサブプロセス生成の場合は, 送信 1 回あたりのメモリアクセスはそ

のときに送信する配列の大きさの分だけである. 実行時間の差は, このあたりに原因があると考えられる.

原因が何であれ, 実行速度が遅くなる手法を使うわけにはいかない. したがって, 演算·送信の場合のサブプロセス生成方法は, PVM のタスク生成機能を使って行う.

4.4.4 不必要な同期の削除

プログラム中のある場所で同期を取った場合, 全プロセッサ (格子状分割の場合は対角プロセッサ) の処理は最も処理の遅れているプロセッサに揃えられる. したがって, 同期を取るポイントに早く到達したプロセッサには “待ち” の時間が発生してしまう. kernel CG のプログラムには同期を取る命令は実装していないが, ベクトルの内積計算時にほんの 8[byte] のメッセージの集計·分配があるために, 強制的に同期が取られる形になっている. ベクトルの内積の計算結果は, 全対角プロセッサで必要になるためにこの処理を省くことはできないが, kernel CG の 1 ループ中に 2 回 (全体では 2×15=30 回) だけ内積の結果を分配しなくてよい場合がある. $resid$ とζの計算を行う部分である. $resid$ とζは, マスタープロセスが表示するだけで, その後の処理に使われることはない. したがって, この 2 回のみ集計だけ行い, 分配を行わないことで若干ではあるが次ループの行列×ベクトルの送信開始を早めることができる. このとき, 同期を取っていないから各プロセッサでの処理が少しずつずれ, 通信路が競合する時間を減少させることができる.

Chapter 5

評価

本章では, これまで述べた手法を実装した場合の実行時間を示し, その結果より, 台数効果と kernel CG を高速に実行できるようなネットワーク形態の検討を行う. Probrem Size には Class A を用いた. 測定した実行時間は, 複数回測定した中で最も短いものを結果とした. ただし, たとえ実行時間が最短であっても, その 1 回だけが他の実行結果に比べて飛び抜けて短いような場合にはそれを除外した中で最短のものを結果とした.

5.1 実装環境

実行時間を測定するにあたり, 以下の 2 箇所に PVM を実装した.

- 通信工学科計算機室 (9 号館 4F)·····Sun Warkstation×6
- 12 号館計算機室 (12 号館 5F)·····NEC EWS4800/320VX×64

これらのマシンは全て Ether net で結合されている. Ether net では通信路中には常に単一のメッセージの存在しか許されないため, PVM を実装すると, バス結合型並列計算機 (図 5.1) が構成される.

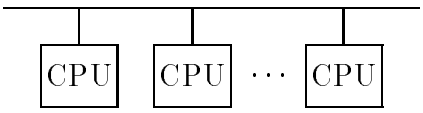

Figure 5.1: バス結合型並列計算機

ただし,12 号館計算機室のネットワークは, バス結合ながら 4 本にパーティーションが分かれている (図 5.2). この 1 パーティーションに 16 台ずつのマシンが接続されており, 全体で 64 台という構成になっている.

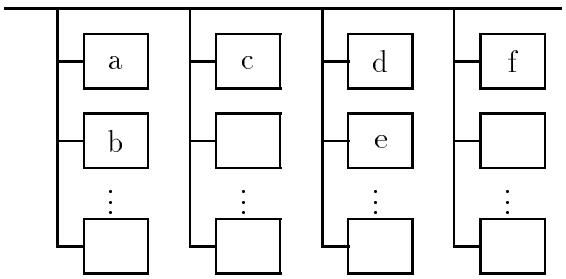

Figure 5.2: 12 号館計算機室のネットワーク

図 5.2のような接続において, 各パーティーションが, そのパーティーション内で通信を行う分には他のパーティーションには影響を与えず通信路競合は起こらないが, パーティーションの外に出ての通信が起こると, 各パーティーションをつなぐ 通信路上での競合が起きてしまう. つまり,a と b が通信を行っている最中であっても, d と e での通信が可能であり, a と c が通信を行っている間は,d と f での通信は不可能である. また,c と f が通信を行っていても a と b,d と e での通信は可能である. したがって, この通信路の特性をうまく利用すればマシンが全てダイレクトに接続されている場合よりも通信路の競合を減少させることができ, 全体性能を向上させることができる.

5.2 実行前の検討

実行結果の検討を行うには, そのもとになるようなデータがなければ不可能である. したがって, 実行前に検討にに必要と思われるデータを測定した. 測定したのは, 以下の事項である.

- シングルプロセッサ版 kernel CG の実行時間
  これにより測定した時間は,kernel CG において純粋に “計算” のみにかかる時間である. したがって, この結果より並列化プログラム実行時の行列×ベクトルの計算にかかるおおよその時間を求めることができる.

- プロセッサ間の通信速度
  これは, 並列化時の実行時間を決定する重要な要素である.

- 同じプロセッサに割り当てられたタスク間の通信速度
  オーバーラップ時にサブプロセスにデータを渡さねばならないが, このときにかかる時間を決定するのがこれにかかる時間である.

これらを, 通信計算機室 (et001) と 12 号館の両方で測定した. その実行結果を表 5.1に示す. ただし, 通信量の測定は, ある大きさのメッセージのやりとりを繰り返し, かかった時間を測

Table 5.1: 実装環境における基礎データ

実装環境	Single CG 実行時間 [sec.]	通信速度	
		プロセッサ間 [kbyte/sec.]	タスク間 [Mbyte/sec.]
et001	2244	475	1.66
12lab.	353	325	2.8

定したものである. 1 回あたりのメッセージの大きさは, 要素数 1750 個の倍精度型の配列, つまり 14[kbyte] である. 1750 というのは,Class A を 64 台分割したときに発生する通信 1 回あたりの配列の個数である. ただし, ここにあげた通信速度は送信開始から受信完了までの時間であるから, 送信側のメモリリードと受信側のメモリライトの時間も含まれている. したがって, 実際の “通信” のみの速さは表 5.1の値よりも大きい. また, オーバーラップ時には 1 回あたりの通信量はもっと小さくなるので, パケットサイズの関係から通信速度が変化する可能性はある. これらより, 並列化した場合の実行時間を, 非常にいい加減ではあるが, ある程度予測できる. 例えば,et001 において,CG を 4 分割した場合の予想実行時間は以下のように計算できる.

無分割時の実行時間は 2244[sec.] であるから, 4 台に分割した場合の 1 台あたりの計算量は

$$\frac{2244}{4} = 561 \quad [sec.]$$

となる.Class A においては, このうち 96%が行列×ベクトルであり, 全体でその処理が 390 回あることから, 各プロセッサが 1 回の行列×ベクトルにかける時間は,

$$\frac{561 \times 0.96}{390} = 1.381 \quad [sec.]$$

となる. また, 発生する通信 1 回あたりの通信量は,

$$\frac{14000}{\sqrt{4}} \times 8 = 56000 \quad [byte]$$

であるから, その通信にかかる時間は,

$$\frac{54000}{470 \times 10^3} = 0.1149 \quad [sec.]$$

となる. これより, 各プロセッサでの動作は図 5.3のようになる. 図中のプロセッサ間を結ぶ線が通信であり, 長方形が計算である. ただし, プロセッサ 0,3 では, 受信の後に集計の処理があるが, それにかかる時間はこのタイムチャートに描けないくらい短い時間でなされるので, 省略してある. また, 送信後演算の場合には, 送信側は受信側の受信終了まで待たされることはないが, ブロックされる時間は不明のため, 受信終了時を計算開始可能時刻とした.

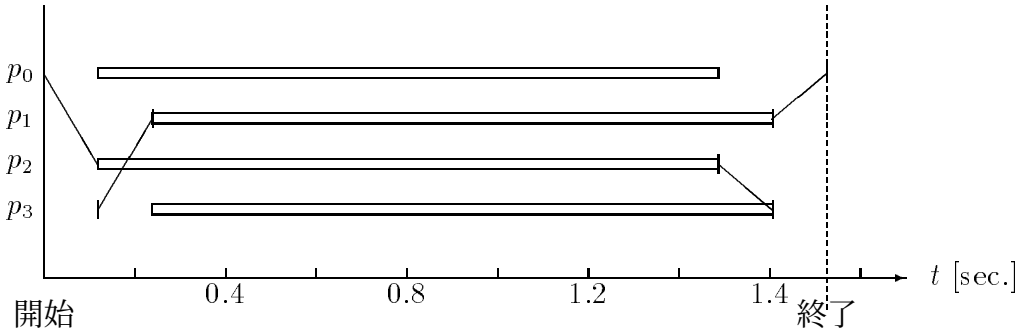

Figure 5.3: 行列×ベクトル計算時のタイムチャート

図 5.3より, 行列×ベクトル計算 1 回あたりにかかるおおよその時間は, 次のように求められる.

$$0.1149 \times 3 + 1.381 = 1.7257[sec.]$$

したがって, 通信工学科計算機室において, CG を 4 台分割して実行したときの実行時間は, 次のようになる.

$$1.7257 \times 390 + \frac{2244 \times 0.04}{\sqrt{4}} = 717.9 \quad [sec.]$$

ここで計算した実行時間は, あくまでおおよその値であり, 無分割実行時の結果を信頼できる唯一の値としているため, これと実際の実行時間を比較しても全く意味がない. しかし, 図 5.3は, オーバーラップの効果を検証する材料になる.

まず, 送信後演算の場合のオーバーラップだが, これに効果がないことは一目瞭然である. なぜなら, 送信後演算の場合のオーバーラップは, プロセッサ 0 とプロセッサ 3 での演算開

始時刻を早めるものであるが, 図 5.3において, プロセッサ 0 とプロセッサ 3 の演算を左にシフトしても終了時刻が変わることはない. それどころか, プロセッサ 1, プロセッサ 2 にデータが届くのが遅れ, 終了時刻を送らせてしまうことさえ予想される. これは, 通信時間と演算時間の比や, プロセッサ台数に関係なく言えることであるので, 送信後演算を実装した場合の CG の実行時間の測定は行わないこととした. 一方, 演算後送信の場合には小量ながらある程度の効果が期待できる. つまり, 演算を 2 分割してオーバーラップをかけた場合, 最後の送信 (図 5.3の場合では, プロセッサ 1 から 0 への送信) にかかる時間を$\frac{1}{2}$にすることができる. したがって, オーバーラップのための分割数 (以後, これを深さと呼ぶ) を 2 とした場合には, その短縮時間は,

$$\frac{0.12}{2} \times 390 = 23.4 \quad [sec.]$$

となる. ただし, 送信側 (プロセッサ 1·2) ではオーバーラップのために送信データをサブプロセスに送る作業があり, その伝達時間のために送信開始が遅れるため, 実際に短縮できる時間は最後の送信の$\frac{1}{2}$よりも短くなる.

これらの結果より, 測定は以下の 4 種類のプログラムに関して行うこととした.

- `cgblkdiv.c`···標準送受信命令により並列化
- `cgb_psnd.c`···直接メッセージ送受信により並列化
- `cgb_fwd.c`···演算後送信をいくつかに分割
- `cgb_fwds.c`···サブプロセスを生成し, 演算後送信をオーバーラップ

この中で,`cgb_fwd.c` には効果がないように見えるが, 4.4.3で述べたように, 演算後送信の場合は何もしなくても多少のオーバーラップが達成できるので, その効果を確認するために測定項目に加えた.

5.3 実行結果

本節では,PVM を実装したネットワーク上でプログラムを実行し, 測定した結果を示す. ただし, その測定時にはネットワーク上にユーザーがいないことが望ましい[1]. しかし,12 号

[1]たとえマシンを使っていなくてもユーザーがネットワークに負荷をかけている場合は, 通信回数の多い kernel CG には致命的である.

館計算機室にユーザーがほとんどいない時間は限られるので, まず et001 で測定を行い, その中で最も実行時間の短いものを 12 号館計算機室で実行するという手順で測定を行った. et001 における測定は, ユーザーが皆無であろう午後 11 時から翌日の午前 5 時の間で行った. この時間帯に各プログラムを複数回実行し, その中で最速のものを測定結果とした. ただし, その数値が同プログラムのその他の測定結果と比べて飛び抜けて早いような場合には, 一時的に通信路を過度に占有できた[2]可能性があり, 他の結果との釣合がとれなくなるのでそれを除外した上で最速なものを結果とした.

5.3.1 通信工学科計算機室 (et001)

表 5.2に,et001 での各プログラムの測定結果を示す. ただし, 行列の分割数は 4 とし, 4 台のマシンを仮想並列計算機に加えて実行したものである. また, 表 5.2の右の高速化の項目は, シングルプロセッサでの実行に比べて何倍の早さで実行できたかを表す. ただし,`cgb_fwd.c`

Table 5.2: Sun Warkstation 4 台での各プログラムの実行結果

プログラム名	実行時間 [sec.]	高速化 [倍]
Single CG	2244	—
`cgblkdiv.c`	640	3.506
`cgb_psnd.c`	630	3.562
`cgb_fwd.c`	613	3.661
`cgb_fwds.c`	643	3.490

と `cgb_fwds.c` のオーバーラップの深さは 5, すなわち 1 回あたりの通信量を 112[kbyte] とした. これは,et001 において表 5.1に示す通信速度を確保できる最少サイズのメッセージが 112[kbyte] であったためである[3]. つまり,et001 においてオーバーラップの深さを 5 よりも深くすると通信速度が低下し, かえって性能を落としてしまう恐れがあるため, そのぎりぎりである 5 を深さとしたのである.

5.3.2 検討 (1) ～オーバーラップの効果～

通信工学科計算機室での結果だが, 結果はどれもシングルプロセッサでの実行時間を大きく短縮し, 並列化の効果は一目瞭然である. しかし, 並列化プログラムだけの結果を見比べ

[2]Ether net の通信速度は 1[Mbyte/sec.] であるが, ユーザーが皆無であっても普通に占有できるのは 400～500[kbyte/sec.] 程度である (表 5.1参照). したがって, 一時的であっても 1[Mbyte/sec.] 近い通信速度が達成されれば, CG の実行時間は大きく短縮される.

[3]ここには結果は載せていないが, 実際に測定を行って調査したものである.

た場合, これはいささか意外な結果と言わざるを得ない. なぜなら, 単に並列化しただけの cgblkdiv.c よりも直接メッセージ送受信を行った cgb_psnd.c が高速であり, さらにそれよりも軽くオーバーラップをかけた cgb_fwd.c が高速である. ここまでは 4.4.2,4.4.3での検討通りなのだが, 本来ならば実行時間が最短でなければならないはずの cgb_fwds の実行時間が最も長い. したがって, 実行時間が最も短いのは集計フェイズの通信を分割し, 命令に任せてオーバーラップをかけた cgb_fwd.c となった. もし, 送信時にプロセッサがブロックされる時間が全くないとすると, オーバーラップにより短縮できる時間は,

$$\frac{\frac{7000\times 8}{5}\times 5-15}{475\times 10^3}=36.78\ \ [sec.]$$

である. オーバーラップにより短縮された時間は 17[sec.] であるから, 直接メッセージ送信時にプロセッサがブロックされていた時間は, 通信を行っている時間の約 50%と計算できる.

cgb_fwds.c は, 非対角プロセッサにサブプロセスを持たせ, それが送信に係わる処理の全てを行うことで集計フェイズ (図 3.13(b)) の非対角プロセッサの演算に遅延をもたらすことなく送信開始を早めることが目的であった. しかし, 実際の実行時間がオーバーラップをかけないときよりも落ちている以上, 一連の処理の中に実行時間を遅らせる要素が含まれていることは事実である. オーバーラップの効果が cgb_fwd.c には表れ, cgb_fwds.c には表れないことから, cgb_fwd.c では行っておらず, cgb_fwds.c では行っている処理が実行遅延の原因であることは明かである.

この 2 つのプログラムの相違点は,

- サブプロセスの存在
- サブプロセスへのデータの送信

の 2 箇所である. このうち, サブプロセスはメインプロセスとマルチタスクで実行されるため, プロセス一つあたりの実行速度は低下する. しかも, サブプロセスにおいて送信命令が実行されるのは, メインプロセスが演算を行っている真裏であるから, 演算速度そのものが低下していることは間違いない. ただし, その処理のステップ数では, メインの演算の方が遥かに大きいことは容易に想像できる. したがって, 演算速度の低下よりもサブプロセスへのデータの送信時に遅延発生の原因の多くがあることになる.

サブプロセスを置いたのはデータ送信時のブロック時間の削減のためであるが, この部分でブロック時間の削減が行われていない可能性が大きい. 仮に, データをデータをサブプロセスに渡す場合であろうと他プロセッサに渡す場合であろうと, その通信速度に関係なく

ブロックされる時間が同じであった場合, つまり, ブロックされる時間というのが, これまで考えていた “命令発効から通信路にメッセージを送り終わるまでの時間” ではなく, “命令発効後, 送るべき要素を確認してハードウェアに送信要求を出すまでの時間” であったとすると, サブプロセスへのデータの送信時間がそのままオーバーヘッドになってしまう. 結果として, サブプロセスの存在自体が単に処理を送らせるだけの荷物になってしまうのである. 単に集計フェイズをいくつかに分割した cgb_fwd.c にオーバーラップによる効果が表れていることからも, これが実行時間を送らせた原因の大きな要素と見てまず間違いない.

プロセスのマルチタスクでの実行による速度低下はある程度予想はしていたが, これはブロック時間の短縮により覆い隠され, 全体の実行時間は短縮されるものとばかり考えていたが, ブロック時間が変わらないというのは盲点だった. これはあくまでも仮定にすぎないのだが, そう考えると全てのつじつまが合う. ただし, 送信時のブロック時間が予想よりも短かったことで, cgb_fwd.c の効果が思った以上にあったのは収穫であった.

5.3.3 12 号館計算機室

et001 での測定により, 複数のプログラムの中で最も実行時間が短かった cgb_fwd.c を 12 号館計算機室の EWS(最大 64 台) に実装し, 実行時間を測定した. ただし, 前述の通り 12 号館計算機室のネットワークは 4 つのパーティーションに分かれているため, プロセッサの割当にバリエーションがでてくる. とはいっても,kernel CG では通信は基本的には図 3.13 のように行方向と列方向にしか行われない[4]ので, 分配か集計のどちらの通信を優先するかという二者択一になる. 分配時と集計時の通信量は等しいので, どちらを優先しても通信に必要な時間そのものは変化しないが, 非対角プロセッサに早くベクトル要素を渡すことで, 対角プロセッサに解が帰ってくるのが早くなる[5]という観点から, 分配を優先した割当を行うこととした. 測定に使用したプロセッサ台数は, 整数の平方根を持つ 4 台,16 台,64 台である. ただし,4 台の場合は, パーティーション別にプロセッサの割当を行っても効果が期待できないことは図 5.3から明らかであるため, 4 台の場合は 1 パーティーションでの実行のみとした. 同様に整数の平方根を持つ 9 台,36 台,49 台のときは行列のオーダーが $\sqrt{p}$ で割り切れず行列の分割が不可能なため, 測定は行っていない. また,25 台の場合行列の分割は可能だが, プロセッサをパーティーションごとに公平にマッピングできないためにこれも測定は行っていない.

[4] ベクトルの内積計算時には対角プロセッサ間で通信が発生するが, そのときの通信量は 1 プロセッサあたり 8[byte] であるので, 通信路競合など問題にならない.

[5] 通信路がブロックされていても, 送信要求の発行は早くなる.

12 号完計算機室での kernel CG の実行時間を表 5.3に示す. この実行において, 集計フェイズでの 1 回の通信量は,
64 台で 7[kbyte](要素数 875 個·深さ 2),
16 台で 14[kbyte](要素数 1750 個·深さ 2),
4 台で 11.2[kbyte](要素数 1400 個·深さ 5)
である. ただし, 実行時にネットワーク上には何人かのユーザーがいたため,100%信頼できる値ではない.

Table 5.3: 12 号館計算機室での実行結果

プロセッサ数	実行時間 [sec.]	高速化 [倍]
1(Single)	353	—
4	523	0.675
16	661	0.534
64	941	0.375

5.3.4 検討 (2) ～台数効果～

表 5.3に示すように, 残念ながら,12 号館計算機室の結果では並列化による高速化を実現することはできなかった. それどころか, 並列化時の性能はどれも 1 台で実行した場合を下回り, 台数の増加にしたがって実行時間はさらに増加してしまっている. プロセッサ台数を増やした場合, プロセッサ 1 台あたりの計算量は減るものの, 表 3.2に示すように台数の増加にしたがって通信量が増えることから, 実行時間の増加が通信量の増加によるものであることは明らかである. つまり, 通信量の増加が計算量の減少を多い隠すほど大きくなってしまっているのである. 当然のことながら, 並列化においてプロセッサ台数を増やした場合には増やした台数に見合うだけの性能向上がなければならない. 問題を p 台で分割して処理した場合, その性能向上の上限は p 倍[6]である. 今回 PVM を実装した 2 箇所のネットワークにおいて, et001 では 1 台での実行の $\frac{2244}{613} = 3.66$ 倍の性能を得ることができた. et001 での性能向上の限界は 4 倍であるから, 記述したプログラムは性能限界の 91.5%を達成できたことになる. しかし,12 号館計算機室においては, 個々のマシン性能が et001 をはるかに上回る[7]にもかかわらず, 性能向上は 1 倍を下回るという無様な結果となった. これは全て

[6] まれに, 分割して小さくなったデータが全てプロセッサのキャッシュに納まってしまい, p 倍以上の性能向上がある場合がある. これを, スーパーリニアという.
[7] CG の実行速度では約 6.3 倍の開きがある.

通信速度と演算速度のバランスが悪いために生じている.

繰り返すようだが,CG の演算のほとんどが行列×ベクトルで占められる. よって, この計算で要求する性能が得られれば, 全体の性能も要求するものにほぼ等しくなる. 台数効果を得るのに必要な条件を求めるために, 行列×ベクトルの計算にかかる時間を定式化する. ただし, 簡単のためオーバーラップをかけない場合を対象とする. また, 各プロセッサはバス上に直列に接続されており, パーティーションに分けられていることもないとする.

行列×ベクトルの計算時に発生する通信 1 回あたりにかかる時間 t_cは, 通信速度を v_cとすると,

$$t_c = \frac{\frac{8n}{\sqrt{p}}}{v_c} = \frac{8n}{\sqrt{p}v_c} \quad [sec.] \tag{5.1}$$

であるから, 行列×ベクトルの計算 1 回あたりにかかる時間 t_{mv}は式 5.1を通信回数倍し, 通信路の空き時間 (分配終了から集計開始までの時間) を加えることによって求められる. 通信路の空き時間は, 分配フェイズに一番最後発生した通信が終了した時点から, 一番最初にデータを受け取ったプロセッサが計算を終えるまでの間である (図 5.3参照). この時間 t_oは, 次式で与えられる.

$$t_o = \begin{cases} t_p - (\sqrt{p}(\sqrt{p}-1)-1)t_c & (t_p \geq (p-\sqrt{p}-1)t_c) \\ 0 & (t_p < (p-\sqrt{p}-1)t_c) \end{cases} \tag{5.2}$$

ただし,t_pは 1 プロセッサあたりの計算時間である. 式 5.2は,t_oには通信路に空きが生じる状態と演算が全て通信時間に飲み込まれる 2 つの場合が存在することを表している. 以下に示す式は, 通信路に空きが生じる状態のみに関してのものである.

式 5.2より,t_{mv}は,

$$\begin{aligned} t_{mv} &= (p-\sqrt{p}) \times t_c \times 2 + t_o \\ &= (p-\sqrt{p}) \times t_c \times 2 + t_p - (p-\sqrt{p}-1) \times t_c \\ &= (p-\sqrt{p}+1)t_c + t_p \\ &= \frac{8n(p-\sqrt{p}+1)}{v_c\sqrt{p}} + t_p \end{aligned} \tag{5.3}$$

となる.

この演算が分割前の行列×ベクトルの演算にかかる時間の$\frac{1}{x}$になれば,x 倍の効果が得られたということである. したがって, 分割前の行列×ベクトルにかかる時間を t_{mv0} とすると,p

台分割時に仮に wp 倍 $(w < 1)$ 以上の効果を得たい場合の条件は, 以下のようになる.

$$\begin{aligned} t_{mv} &> \frac{t_{mv0}}{wp} \\ \frac{8n(p-\sqrt{p}+1)}{v_c\sqrt{p}} + t_p &> \frac{t_{mv0}}{wp} \\ v_c &> \frac{8n(p-\sqrt{p}+1)}{(\frac{t_{mv0}}{wp} - t_p)\sqrt{p}} \end{aligned} \tag{5.4}$$

つまり, ある問題と分割数が与えられた場合, 任意の性能を得ようとすると, 式 5.4で求められるだけの性能を持った通信路が必要である. ただし, 式 5.2において, 通信路にわずかでも空きができることを仮定しているので, wには, 次のような制限ができてしまう.

$$\begin{aligned} t_c \times 2 + \frac{t_c}{p-\sqrt{p}-1} \times 2 &> \frac{t_{mv0}}{wp} \\ w &> \frac{1}{2}\left(\frac{p-\sqrt{p}-1}{p-\sqrt{p}}\right) \end{aligned} \tag{5.5}$$

一方,$w \leq \frac{1}{2}\left(\frac{p-\sqrt{p}-1}{p-\sqrt{p}}\right)$ の場合には,t_{mv}は,

$$t_{mv} = (p-\sqrt{p}) \times t_c \times 2 \tag{5.6}$$

となるので, 同様の計算手順により v_cは次式で与えられる.

$$v_c = \frac{16npw(\sqrt{p}-1)}{t_{mv0}} \tag{5.7}$$

したがって,p 台分割時に $wp(w < 1)$ 倍の台数効果を得たい場合に必要となる通信路性能の下限値は, 以下のように示される.

$$v_c = \begin{cases} \frac{8n(p-\sqrt{p}+1)}{(\frac{t_{mv0}}{wp} - t_p)\sqrt{p}} & \left(w > \frac{1}{2}\left(\frac{p-\sqrt{p}-1}{p-\sqrt{p}}\right)\right) \quad \cdots\cdots (a) \\ \frac{16npw(\sqrt{p}-1)}{t_{mv0}} & \left(w \leq \frac{1}{2}\left(\frac{p-\sqrt{p}-1}{p-\sqrt{p}}\right)\right) \quad \cdots\cdots (b) \end{cases} \tag{5.8}$$

台数効果が全く出なかった 12 号館のネットワークにおいて, 16 台分割時に仮に $0.5p$ 倍の性能を得たいとする. 式の変わる境界は

$$\frac{1}{2}\left(\frac{16-4-1}{16-4}\right) = 0.458$$

であり,

$$w > 0.458$$

であるから, 式 5.8(a) より,

$$\frac{8 \times 14000(16 - \sqrt{16} + 1)}{\sqrt{16}\left(\frac{\frac{353 \times 0.96}{390}}{0.5 \times 16} - \frac{\frac{353 \times 0.96}{390}}{16}\right)} = 6.77 \times 10^6 \quad [byte/sec.]$$

の通信路性能が必要だとわかる. ただし,t_pは,Single CG の実行時間の 96%を反復回数 390 で割った値とした. 12 号館の (占有可能な) ネットワークの性能は, せいぜい 300[kbyte/sec.] であるから, 台数による性能向上がないのもうなずける. これが 64 台となれば必要な通信路性能はさらに増加するので, 現状では台数効果は絶対に得られない.

一方,et001 において,4 台分割時に 50%の効果を得たい場合には,

$$\frac{8 \times 14000(4 - \sqrt{4} + 1)}{\sqrt{4}\left(\frac{\frac{2244 \times 0.96}{390}}{0.5 \times 4} - \frac{\frac{2244 \times 0.96}{390}}{4}\right)} = 123 \times 10^3 \quad [byte/sec.]$$

の通信性能でよい.et001 において確保できる通信速度はこれを上回っているので, 実行時に $0.5p$ を超える $0.9p$ 倍の性能をあげることができたのである. つまり,kernel CG では, ある台数のマシンが与えられた場合, マシン 1 台あたりの速さと通信路の通信速度の比が実行時間に大きな影響を与える. この比が適当であってはじめてオーバーラップの効果が表れるのである.

ただし, 別の考え方をすれば, 通信路が遅いのではなく, 各プロセッサでの演算量が極端に少ない, あるいはマシンが通信路性能に比べて不釣り合いなくらい高速な場合にも台数効果は表れなくなる[8]. したがって, 通信路の性能が固定されている場合には演算量を増やすことで通信時間との比を大きくできるので, 必要な性能を得ることが可能である.

[8] et001 でも, 演算量の少ない Sample では 1.5 倍の性能しか得られなかった. また, 12 号館で性能向上がないのは明らかに後者の理由である.

Chapter 6

おわりに

本研究では, バス結合型並列計算機上で kernel CG を実行するにあたり, 配列の最適な分割方法と, 実行時間の短縮すなわち高速化についての検討を行った. その結果, 分割方法としては格子状分割が最適であり, 高速化手法としては, 少なくとも直接メッセージ送受信と演算と送信のオーバーラップの 2 種類については実装が効果的という結論に至った.

格子状分割は, 同時に検討した行分割·列分割に比べてプロセッサ台数が同じ場合にプロセッサ間でやりとりするデータの大きさがずっと小さくて済む. 同時にひとつしかメッセージが存在できないバスという通信形態において, 通信量が少ない分割というのは実に効果的である. 直接メッセージ送受信は, プロセッサ間の通信をバッファを介さずに行う方法で, ある程度の大きさの単独の配列をやりとりする場合に効果を示す. ただし, 時間的に見て受信命令が先に発行されていないとメッセージはバッファリングされてしまうので効果がなくなってしまう. 演算と送信のオーバーラップは, 順次計算される結果を計算終了に先行して送信する手法で, これによって受信側のメッセージ待ち時間を削減することができる.

これらを実装して結果を測定したところ, 演算速度と通信速度の比が十分大きく, kernel CG に適していた通信工学科計算機室では良好な結果を得たが,12 号館計算機室では演算速度に見合うだけの通信速度を得られなかったため, 並列化による高速化を実現することができなかった. このため, バス結合のネットワーク上で, 問題とマシンが与えられた場合に, 期待する効果を得るのに必要な通信速度を定式化して示した.

kernel CG では,Class A においてその処理の 96%が行列とベクトルの積を求める計算であり, 研究においては並列化および高速化の主眼をそこに置いた結果, 議論が行列ベクトル積に終始してしまった. 残るベクトル演算の高速化可能性についてより追求していけば, 若干でもさらなる高速化が可能であったかもしれない. 加えて, 実行可能なプロセッサ台数の

バリエーションがあまりにも少なく (64 台までで 4 通り), 汎用性に欠けているなど, 課題として残る部分もまだまだ多い. また,12 号館において台数効果が出ないのを全て通信路が遅いせいにしているようなところもあり, 全体的に検討不足が目立つ部分が多くなってしまったのが何より残念である.

謝辞

本研究の遂行にあたり, いろいろと有益な指導をいただいた指導教員の清水尚彦先生をはじめ, 数々の助言や提案を下さった同輩の大瀧由章氏および松崎将紀氏に深く感謝いたします. また, 実験の環境を提供してくれた 12 号館計算機室の職員の皆様にも感謝いたします.

Bibliography

[1] D.Bailey,E.Barszcz,L.Dagum,P.Frederickson,R.Schreiber,H.Simon:“The NAS Parallel Benchmarks” RNR Technical Report RNR-94-007,March 1994

[2] Ansul Gupta,Vipin Kumar,Ahmed Sameh:“Performance and Scalability of Preconditioned Conjugate Gradient Methods on Parallel Computers” IEEE Transactions on Parallel and Distributed Systems,Vol.6,No.5,May 1995

[3] S.White,A.Ålund,V.S.Sunderam:“Perfoemance of the NAS Parallel Benchmarks on PVM based Networks” RNR-94-008 May 1994

[4] 湯川高志, 石川勉:“連続行列演算の最適並列化条件とそれに基づく最適 BP 並列化方式” 電子情報通信学会論文誌 D-I Vol.J78-D-I No.5 pp.445-454 May 1995

[5] 萩原純一, 安江俊明, 金子正教, 山名早人, 村岡洋一:“分散メモリ型並列計算機における DO ループ処理方式の提案” 信学技報 CPSY92-15 pp.47-54

[6] 堀江健志, 小柳洋一, 今村信貴, 林憲一, 清水俊幸, 石畑宏明:“メッセージ通信の分散メモリ型並列計算機への影響” 情報処理学会論文誌 Vol.35 No4 Apr 1994

[7] F.S.Beckman:“共約勾配法による連立 1 次方程式の解法” IBM

[8] Al Geist,Adam Beguelin,Jack Dongarra,Weicheng Jiang,Robert Manchek,Vaidy Sunderam:“PVM3 USER'S GUIDE AND REFERENCE MANUAL” May 1993

[9] 奥川峻史:“並列計算機アーキテクチャ” コロナ社

[10] S. ラグスデイル, 訳 吉川正孝:“並列処理とオブジェクト指向プログラミング” マグロウヒル

[11] 木下是雄:“理科系の作文技術” 中公新書