北陸先端科学技術大学院大学
情報科学センタ 利用の手引

志田 和人, 河瀬 剛, 宮崎 純, 井口 寧

1994 年 11 月 11 日
IS-TR-94-0002M

北陸先端科学技術大学院大学
情報科学研究科
〒 923-12 石川県能美郡辰口町旭台 15
shida@jaist.ac.jp, kawase@jaist.ac.jp, miyazaki@jaist.ac.jp, inoguchi@jaist.ac.jp

ISSN 0918-7561

巻頭言

本書は北陸先端科学技術大学院大学 (JAIST) 情報科学センターの保有する超並列マシン等の利用方法を書いたユーザー・マニュアルである.

JAIST は 1992 年の発足以来, キャンパス・ネットワーク Frontnet 上の超並列マシンやミニスーパコンピュータを研究・教育のための基本的なインフラストラクチャとして整備してきた. 当初はそれぞれのマシンの利用者が整備を進めることを原則としてきたが, 利用者の関心が高まり, かつ利用者数が漸増するようになった. そこで JAIST 情報環境を実質的に支えている「Frontier 利用者の会」の中の若手有志が立ち上がり, 本書を CM-5,nCUBE-2 および Convex C-3440 の利用の手引としてまとめたわけである. 本書が超並列マシン等の利用の一助となることを期待するとともに, マニュアルとしての改善について利用者から執筆者グループへのフィードバックをお願いしたい.

これからの時代の激変する情報環境の推移を考えると, 最先端のマシンのサポートはそれを実際苦労して使いこなしている利用者グループが, このような利用者の目からみて有用な情報を提供することが必要である. 情報科学センターにある他の超並列マシン等についても, このような情報共有の動きが続くことを期待する.

1994 年 9 月

情報科学センター長

國藤 進

もくじ

1 CM-5 **1**

1.1 コネクションマシン CM5 の概要 2

1.1.1 特徴 2

1.1.2 ハードウェア構成 2

1.1.3 基本的な利用法 3

1.1.4 ソフトウェア 4

1.2 CM-Fortran:簡単なプログラム例 8

1.2.1 ソースの書き方 8

1.2.2 コンパイルのしかた 11

1.2.3 コンパイル 11

1.2.4 実行のしかた 11

1.2.5 並列化するステートメント 12

1.2.6 コンパイラオプション 15

1.2.7 計算機資源 16

1.3 デバッグ 18

1.3.1 デバッグのしかた 18

1.3.2 グラフィカルデバッガ PRISM 19

1.4 行列の乗算 21

1.5 パフォーマンス 25

1.5.1 仮想プロセッサの配置 25

1.5.2 演算子の数 26

1.5.3 不規則演算 26

1.6 失敗例の研究 29

1.6.1 乱数発生 29

1.6.2 配列の大きさ 30

1.6.3 配列の最適化 34

1.7 移植 37

1.8 CMSSL ルーチンのテスト 39

1.8.1 行列積算 39

1.8.2 行列対角化 39

1.8.3 高速フーリエ変換 40

1.9 CMMD 41

1.9.1 基本的な関数 41

1.9.2 簡単なプログラム 42

1.9.3 コンパイル 44

1.9.4 実行例 . . . 45
1.9.5 実行できない時 . . . 45
1.9.6 最後に . . . 46
1.10 その他の機能 . . . 47
1.10.1 SDA . . . 47
1.10.2 チェックポイント機能 . . . 47
1.10.3 CMX11 . . . 47
1.10.4 アセンブラ . . . 47
1.11 CM5 で何をするのか . . . 48

2 NCUBE/2 **50**
2.1 nCUBE/2 の構成 . . . 50
2.1.1 Array Nodes . . . 50
2.1.2 I/O Nodes . . . 50
2.1.3 プログラミング環境 . . . 52
2.2 nCUBE/2 を利用するための準備 . . . 53
2.2.1 パスの追加 . . . 53
2.2.2 有用なコマンド . . . 54
2.3 プログラムのスタイル . . . 55
2.3.1 Identify . . . 55
2.3.2 Message passing . . . 55
2.3.3 Time . . . 58
2.3.4 Barrier synchronization . . . 58
2.3.5 File access . . . 59
2.4 プログラムのコンパイルと実行 . . . 61
2.4.1 コンパイラ . . . 61
2.4.2 make . . . 62
2.4.3 実行 . . . 62
2.5 プログラムのデバッグ . . . 64
2.5.1 ndb デバッガを用いたデバッグ . . . 64
2.5.2 デバッグのテクニック . . . 65
2.6 注意事項 . . . 66

3 convex **67**
3.1 概要 . . . 68
3.1.1 ハードウェア構成 . . . 68
3.1.2 ファイルシステム . . . 68
3.1.3 パス . . . 69
3.1.4 サポートされているソフトウェア . . . 69
3.2 簡易使用法と並列化 . . . 70
3.2.1 簡易使用法 . . . 70
3.2.2 ベクトル化と並列化 . . . 71
3.3 FORTRAN . . . 77
3.3.1 特徴 . . . 77

3.3.2 他のコンパイラとの関連 78
3.3.3 FORTRANプログラムのコンパイラ起動方法 78
3.3.4 FORTRANコンパイラの主なオプション 79
3.4 バッチキューの使用法 82
3.4.1 概要 82
3.4.2 バッチジョブクラス 82
3.4.3 バッチジョブの投入 82
3.4.4 バッチキュー・ジョブの操作 83
3.5 チェックポイント・リスタート 85
3.5.1 概要 85
3.5.2 制限事項 86
3.5.3 コマンドによるチェックポイント 86
3.5.4 バッチジョブのチェックポイント 88

第 1 章

CM-5

志田 和人 : shida@jaist.ac.jp
河瀬 剛 : kawase@jaist.ac.jp

本章では, コネクションマシン CM5 について解説する.

最初は箇条書き風の簡単なものを意図して執筆していたのだが, ある筋からの強い要望によって以前に書いた CM5 利用法のレポートを無理にとり入れた結果, 内容はかなり豊富になったが構成が乱雑で言葉使いのぞんざいさが目立つものになってしまった.

加えて, 以前のレポートは約一年前のものであるため, この一年に新しく分かった内容の記述が不足しているし, 著者の思いちがいや OS のバージョンアップによって生じたくいちがいも, 可能な限りで修整するよう努力したが, 不安の残るところである.

このような事情で分りにくい文章になっていることをこの場でおわびしておくが, 内容は現実に CM5 を利用しようとする人にとっては読んで損はしない程度にはまとまっていると思う.

必要な部分だけ読めばよいように, 以下に各セクションを要約する.

セクション 1 では CM5 のハードウェアとそれでできることのアウトラインを説明する.

セクション 2 からセクション 8 で CM-Fortran を説明する. CM-Fortran はコネクションマシン上の標準言語の一つと考えてよい.
セクション 2 が言語の概要, セクション 3 がデバッギング, 以降の章は CM-Fortran を用いて行なったいくつかの実験についてまとめてある. これらは CM-Fortran 入門編というには極めて不充分だが, あちこちに断片的にソースコードの例が掲載してあるのでそういう風に使用することもできなくもない.

セクション 9 は佐藤研究室の河瀬君が書いてくださったもので,CMMD, つまり MIMD モードでの CM5 の利用法の基本を実例付きで説明する.

セクション 10 は CM5 に装備されているものの, 今回は解説できなかった機能の一覧. セクション 11 は CM5 利用の現状のまとめと, 正統的ではないが読んでみるとおもしろいと思われる, コネクションマシン関係のいくつかの文献の紹介である.

1.1 コネクションマシン CM5 の概要

1.1.1 特徴

本学情報科学センターの超並列計算機 Connection Machine CM5 は,64 プロセッサで構成され, 最大で毎秒 80 億回の浮動小数点演算が可能である. 現状では Frontnet 上で最高速のハードウェアとなっている. 使いこなすのがかなり困難な一面もあるが, 適当な問題においてプログラマが注意深く使用すれば非常に高い性能が得られる. 主な特徴は以下の通りである.

1. 最大で 8192CPU までの拡張性
2. 分散記憶, メッセージパッシングアーキテクチャ
3. 単位プロセッサにベクトル演算機構を付加したことによる高速性
4. 合計 2G バイトの記憶域,23G バイトのディスクアレイ (SDA)
5. 多様なソフトウェア環境

Connection Machine CM5 の利用環境はこれらの複雑な混合物である. 以下, ハードウェア, ソフトウェアそれぞれの構成要素について説明する.

1.1.2 ハードウェア構成

CM5 のハードウェア構成をイメージするために,64 プロセッサの結合ネットワークを図示すると以下のようになる.

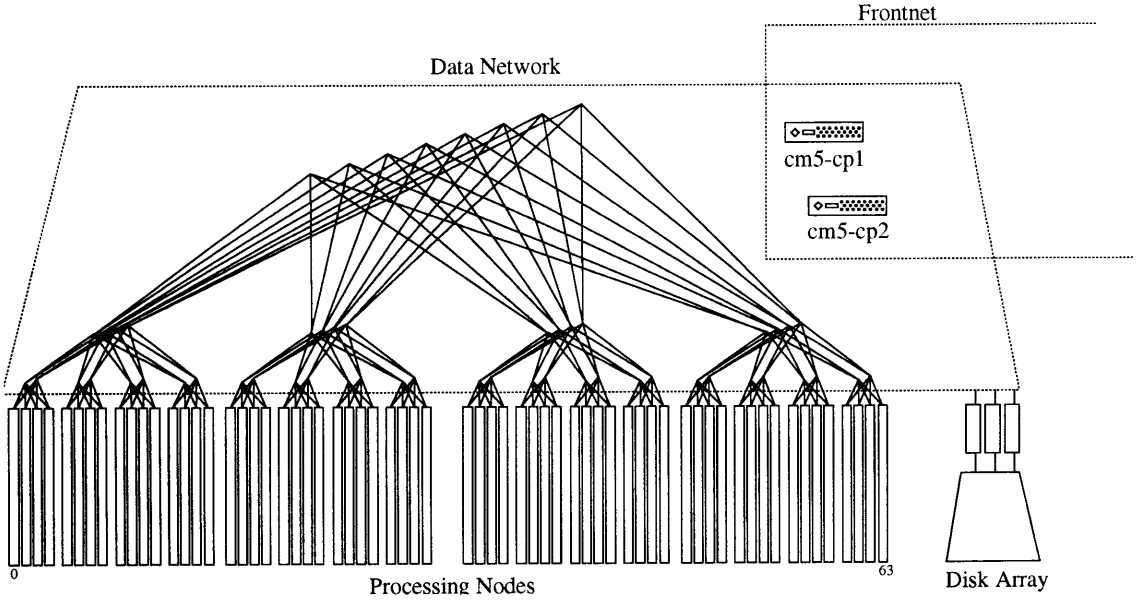


図 1.1: 64 プロセッサの CM5 の結合ネットワーク

単一処理要素 (ノードプロセッサ) は以下のものによって構成される. これは一つの完結したコンピュータであり, オブジェクトコードはそれぞれのローカルメモリにロードされ, 実行される.

- Sparc IU , FPU (33MHz,22Mips,4.2Mflops)
- Memory (32Mbyte)
- Vector Processing Unit (16MHz, 32Mflops) × 4
- Network Interface (20Mbyte/sec) × 2

パケット交換によるメッセージパッシング通信網は, 木結合の一種である FatTree とよばれる構造を採用している. ほとんどの場合, この構造はユーザーに隠蔽されており, 各プロセッサはその ID だけで指定できる.

インターフェースユニットは上記の数値分のパケット発信能力を持っているのだが, ノードプロセッサ自体が低速であるため, ノード対ノードの通信能力は最良でも 10MByte/sec 程度に制限される.

フロントエンド, コンソールとして 5 台の SparcStation2 が使われている. これらはそれぞれ割り当てられた用途がある. 問題を避けるためユーザーは cm5-cp1,cm5-cp2 のみを使用すること.

以上から,64CPU の総計では,

演算速度	1,408Mips, 268.8Mflops (ベクトルユニット未使用時) 8,192Mflops (ベクトルユニット使用時)
主記憶容量	2G バイト
合成転送能力	5120Mbyte/sec
対 SDA 転送能力	33Mbyte/sec

になるはずなのだが, よく知られているように全体が部分の総和以下になってしまうのが超並列計算機の重大な技術課題である. コンパイラのオプティマイズ能力は限られたものであり, 中にはネットワークインターフェースのように, (おそらく意図的に) 他の部品とのかねあいで絶対に使い切られないような無駄な能力を装備している部分もある. しかし利用者はこれらの問題をよく研究することによって膨大な処理を必要とする課題を実際に解くことができ, また並列計算の実際に対する知見をも得られる. 適切な準備, 問題の設定とプログラミングのもとでは,1000Mips/1000Mflops 級の性能が実用可能であると考えられる.

1.1.3 基本的な利用法

- プログラム実行は cm5-cp2
- それ以外 (コンパイルなど) は cm5-cp1

のフロントエンドから使用する.

他にも cmps コマンドによる状況のチェックなど, 並列プロセッサ群にかかわることは cm5-cp2 から行なう. それ以外の操作を全て cm5-cp1 で行なうのは, cm5-cp2 の負担を減少させ, 並列プロセッサ群を高効率で動作させるためである. というのも cm5-cp2 での負荷の上昇は,ts-daemon への CPU 時間の配分を減少させてしまい, これに並列部も影響を受けるため, スーパーコンピュータとしての CM5 の性能をいちじるしく落としてしまうのである.

CM5 の並列部は, 存在するプロセッサを全部使用するだけでなく, 32 個以上の 2 の n 乗個の複数の「パーティション」に分割することができる. そして各「パーティション」が動作するために, 対応するフロントエンド従って cp1,cp2 に 32 個ずつわりあてて両方で使用することも可能だが, そうしていないだけである. この構成の切りかえ自体はオペレーターコンソールから root がソフト的に行い, あっという間にできるのだが, 走っているアプリケーションを止めずに行うことはできない. つまり, アプリケーションとパーティションの関係は固定的である.

これらのマシン上で専用コンパイラ, リンカなどによってオブジェクトを作る.

つまり, どのような通常の計算機用のプログラムでも, そのままではCM5上で使用することはできず, 他人の迷惑になるだけである.

専用オブジェクトができたら, あとは通常のUNIXと同様にこのオブジェクトを起動できる. 例えば,

```
YourWS% ls *.fcm
parallelsort.fcm  並列フォートランのソースコード

YourWS% rlogin cm5-cp1
Password:
cm5-cp1[1]% cc parallelsort.c -c    cmmd ソースのコンパイル
cm5-cp1[2]% cmmd-ld parallelsort.o -comp cc -o PSORT.o リンク

YourWS% rlogin cm5-cp2
Password:
cm5-cp2[1]% PSORT.o &    並列化されたオブジェクトの実行
[1] 863

cm5-cp2[2]% cmps   実行状態のチェック
64 PN System, 28464K mem. free, 6996K VU mem. free, 1 procs


USER      PID   CMPID  TIME   TEXT    ILH     ILS   ...     COMMAND
you      *  863 2      0:00  1304K   488K     48K   ...      PSORT.o

cm5-cp2[3]% ps | grep 863
  863 p4 R    0:01    PSORT.o
```

cmpsコマンドは並列部での実行状況を表示するが, 全ての並列化されたプロセスにはフロントエンドで動作する通常のUNIXプロセスが一つ付属する. 上の例ではそのPIDが863であり, 実際にそのようなプロセスが存在することが示されている. そして, このUNIXプロセスに対して本来のプロセスコントロール機能がほぼそのまま利用可能であり, これに並列実行されているプロセスの方がしたがう.

```
cm5-cp2[1]% PSORT.o < sortdata | head -10 パイプとリダイレクション
[1] 8203

cm5-cp2[2]% kill 8203   強制終了
```

フロントエンドのディレクトリ構成については,

- /usr/cm5の下に各種ソフトウェアのオブジェクトやデモ
- /usr/include/cmの下にCM5関連のヘッダ

などとなっているので, 欠けていて実行できないときは適当にパスにつけくわえること.

1.1.4 ソフトウェア

現在CM5上で利用可能なソフトウェア, ツールをまず列挙してみる. それから, これらの間の同時に使用不可/可能の関係を説明する. さらに, 個々のソフトウェアについて説明を加える.

名称	概要
CM-Fortran	限定的な自動並列化コンパイラ言語, 拡張 Fortran90
C*	限定的な自動並列化ができる拡張 C 言語,
STARLISP	Commonlisp の並列拡張
Prism	並列グラフィカルデバッガ, 兼簡易ビジュアライゼーション
CMMD	メッセージパッシングを提供するライブラリ
CMSSL	並列数値演算ライブラリ
CMX11	低レベルグラフィック出力ライブラリ
pndbx	並列用デバッガ
CMView	ハイパーテキスト風オンラインマニュアル
CMFS	SDA 用並列入出力ライブラリ
CMNA	低レベル通信ライブラリ
CDPEAC	VectorUnit 用アセンブラマクロ

この他にも, パブリックドメインその他で CM5 で使用できるツールなどもあるので, 非常に多様なソフトウェア環境と言っても良いだろう.

ただし, 通常のマシン用の C や Fortran をそのまま並列化できるようなツールはインストールされていないので,CM5 の利用のためには必ずなんらかの形で専用のプログラムを書くか並列計算機用に書きかえる必要がある. つまり, これらを用いて CM5 上でなんらかのアプリケーションを行なう場合, 現状では仕事の手順はつぎのいづれかのようになるだろう.

- CM-Fortran,C* を利用し,
  Prism の支援のもとに,
  並列化手法はある程度コンパイラまかせに,
  CM5 をバーチャル SIMD マシンとして利用するコードを作成する.

- CMMD を採用する.
  CMMD はメッセージパッシングするだけなので,C や f77 とくみ合わせる.
  このコードは各ノードに分散されるが, 互いに協調させるのはプログラマの仕事になる. CM5 をハード本来の MIMD(SPMD) の形で利用できるようになる.

次に, これら複雑な開発環境間の排他的関係を説明する. 次表で縦に並んでいるものは単一のプログラムの中で混ぜて使用できるが, 各表の間で混ぜることはできない.

基本的方式 (a)	CM-Fortran および C* (b)
リンカ	cmld
デバッガ	prism,pndbx.
ランタイム	CMRunTimeSystem を使用する
プロセッサ間通信	言語に内包, 隠蔽
数値演算ライブラリ	CMSSL
ディスクアレイ	CMFS で並列に read/write
グラフィック	CMX11
ベクトルユニット	コンパイル時の指定で利用できる.CDPEAC で機械語レベルでプログラムできる.

基本的方式 (a)	ノード上で個別実行されるプログラム(c)
リンカ	cmmd-ld
デバッガ	prism は利用できない.
ランタイム	CMRunTimeSystem を使用しない
プロセッサ間通信	CMMD メッセージパッシング (d)
数値演算ライブラリ	対応する機能なし
ディスクアレイ	CMMD の UNIX I/O サポート (e)
グラフィック	対応する機能なし(f)
ベクトルユニット	CDPEAC を使って機械語レベルでプログラムする必要がある.

(a) パラダイムと言うべきか?

(b) 多分 STARLISP もここに入れて良い.CM-Fortran と C*の相互コールは可能.

(c) 現状で C と f77 が使用できる. インストールさえすれば C++も利用可とされている.

(d) CMNA で同じようなことができるはずだが, 困難なのでやめた方が良い.

(e) これは Frontnet 上の全ディスクに NFS でアクセスできる. ただし並列入出力は不可能.

(f) 普通の Xlib を無理に使用可か??

「混ぜて」というのは, 他言語でサブルーチンを書いて組みこむような関係のことを指している. 例えば C*で作ったサブルーチンを CM-Fortran から呼べるが,C と CMMD で作ったサブルーチンは無理である.

また,「一個のノードの上で動く」モード (Nodal-mode) の CM-Fortran と C*を使うこともできる. これは説明しにくいが,1CPU の CM5 の CM-Fortran,C*というようなものである. 文法などは通常のものと同じで, ただし他のノードとの通信は CMMD 等で explicit に書いてやらなければならない.

基本的方式	Nodal CM-Fortran および Nodal C*
リンカ	cmmd-ld
デバッガ	Nodal prism(a)
ランタイム	CMRunTimeSystem を使用する
プロセッサ間通信	CMMD メッセージパッシング (b)
数値演算ライブラリ	対応する機能なし
ディスクアレイ	CMMD の UNIX I/O サポート ()
グラフィック	対応する機能なし
ベクトルユニット	自動的に使用される

(a) prism を -node オプション付きで起動する.

(b) 通常の CM-Fortran,C*の言語に内包, 隠蔽された通信, 例えば, 配列の総和やシフトは, そのノード内だけの操作として実現されている. だから 1CPU の CM5 の CM-Fortran,C*と言える.

(c) これは Frontnet 上の全ディスクに NFS でアクセスできる. ただし並列入出力は不可能.

この Nodal-mode のプログラムは通常の CM-Fortran,C*と「混ぜて」使用できる. そのために配列の相互変換関数などが用意されている.

ソフトウェア環境は以上のように複雑なもので, また OS のバージョンアップのたびに細部が変化しつづけているが, オンラインブラウジングツール CMView が利用の際の大きな手助けとなる. 使用法は, 適切な X11 の環境下であれば cm5-cp1 に入り,

```
cm5-cp1[1]%setenv DISPLAY yourWS:0.0

cm5-cp1[2]%cmview &
```

とするだけである. 数秒後に CMView のウインドウが開くが, ここから GUI 操作によって, 現段階における CM5 のマニュアルのほぼ全てを閲覧することができる.

1.2 CM-Fortran:簡単なプログラム例

CM5 に装備されている言語は全て並列処理が可能だが, ここでは主に CM-Fortran を説明する.

CM-Fortran は Fortran の並列拡張である. CM-Fortran は Fortran90 および従来の Fortran と高い親和性を持っており, 高速な演算ライブラリ CMSSL の利用が可能なので, 大量の浮動小数点処理をともなう科学技術演算に適している. 一方 C*は ANSI C の並列拡張とされている. これら二つの文法には大きな相異があるが, 実行時には同じ CMRTS (Connection Machine RunTime System) を利用して実行されるので, C*の方がやや柔軟性に富んでいるものの実際の機能はかなり類似している. そこでここでは CM-Fortran の方にしぼってあらましを解説する.

まず, 非常に簡単な例を使用して CM-Fortran によるプログラミングの要点を説明する. CM-Fortran の言語としての位置づけは,「CM-Fortran Refernce Manual」の冒頭, introduction の章に書かれている. [1] CM-Fortran は, ある程度までの自動並列化機能を有する. 言いかえると,CM-Fortran の流儀にこちらが合わせて書いてやれば, どう並列化するかについてプログラマが考慮する必要はなく,SIMD 的なスタイルで自動並列化する. [2] CM-Fortran の立場は, 従来の Fortran に慣れたプログラマと CM5 とのインターフェースになることと考えられるだろう.

1.2.1 ソースの書き方

CM-Fortran のソースコードは名前が

`*.fcm` もしくは `*.FCM`

でなければならない. 拡張子が FCM の時には, ソースはコンパイラより先に CPP にかけられるため,C のプリプロセッサ機能が利用できる. また,C*のソースコードは

`*.cs`

でなければならない.

エディットはどのマシンでしても同じだが, コンパイルはコンパイラなどを持っているフロントエンドでなければできない. そしてこれらは並列部自体は使用しないが負担が大なので,cp1 の方でやること.

```
      real a(1024)
      real b(1024)

      a = 1.0
      b = 2.0
      b = b + a

      do 100 i=1,1024
      write(6,*) b(i)
  100 continue

      end
```

このプログラムを reptest.fcm として入力したとする. 小文字と大文字はコンパイラディレクティブの指示を例外として混用される. ただし 7 桁目から書かなければならないのはそのままである. また, フォート

[1] 図がまちがえているような気がするのだが.

[2] MIMD 的なスタイルで, つまりはっきりとプロセッサ群に相互に異なったことをやらせるには, f77 で CMMD ライブラリを使用して書く.

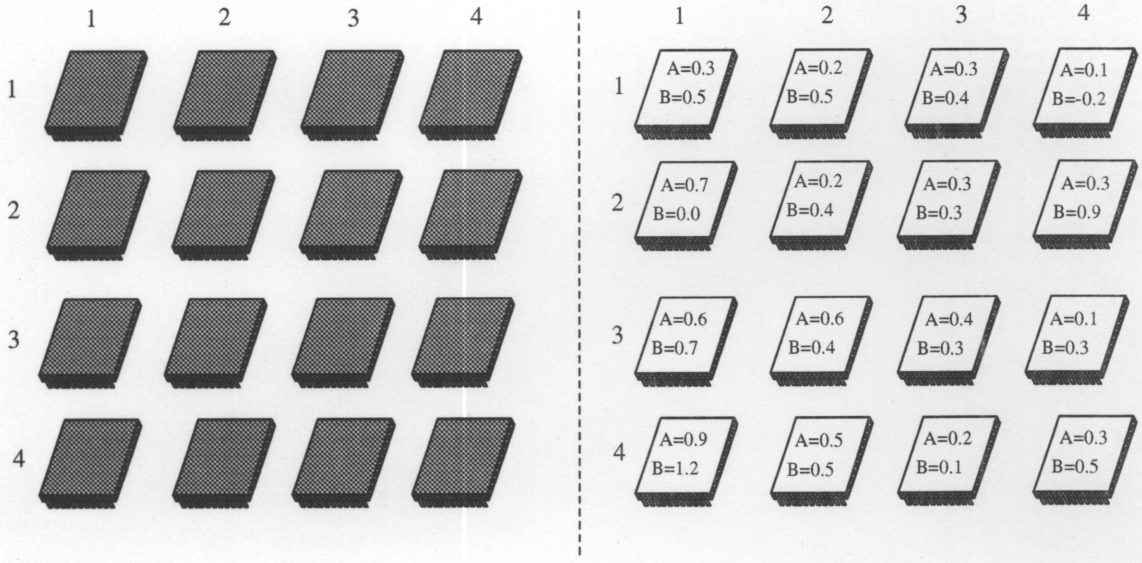


図 1.2: 16 個のプロセッサと, それに分配されているそれに配列変数,A(4,4),B(4,4) が分配されている所の模式図. 二重 DO ループで全ての A_{ij}に対して $A_{ij} = B_{ij} + A_{ij}$ の計算をさせるかわりに, 各プロセッサで独立に $A = B + A$ を実行する. 同時に異なるプロセッサで, 異なるデータに同等なことをするから, Single Instruction Multi Data, もしくは, データ並列と言える. この場合,A(4,4),B(4,4) は同じ形, 同じ大きさの配列である. こういった場合には最も自然な形としてこのように対応する要素がそれぞれのプロセッサで重なるようになる. これを, 二つの配列がコンフォーマルであると言う. コンフォーマルでないとこの場合のような処理は不可能だが, コンフォーマルでない配列が CM のなかに混在すること自体は問題が無い.

ラン 77 そのものの部分が混在しており, これはフォートラン 77 そのままの意味として翻訳されるが, これら 77 の文は自動並列化はされない. 並列化動作のためには, かならず何らかのはっきりした並列化するステートメントと一緒に使用しなければならない. 書くほうもそういうつもりで 77 の文法を使用することになる. [3]

ConnectionMachineFortran の並列化の基本的な考え方は, この 3 つの代入文から分かるように, 配列をひとかたまりの物として操作することと, 配列の記憶領域を各プロセッサに分配することによって, 自然にそれらに対する処理の並列化を行うというものである. この分配も配列の宣言自体[4] によって引きおこされるので, 基本的に特別な操作はいらない.

たとえば, b = b + a は,b の全要素に対応する a の要素を加えることを意味している. 並列化するステートメントといったのはこのような書き方のことである. たとえば A(1,1) = 8 のように 77 の書方でこれらの配列になんらかのアクセスをした場合には, わざわざ必要な情報を対象になる配列要素を所持しているプロセッサとの間にやりとりして, パーティションマネージャが処理してやる格好になる. これは非常に効率が低下する.

上の例で, 1024 というのは CM5 のプロセッサの数より多いが, 実際にはこれはコンパイラがどうにかしてくれる. これを 1024 個の「仮想プロセッサ」を用意すると考える. [5] このため, 基本的に配列の大きさは, パー

[3] たとえば入出力や, 実際にある操作を n 回連続して逐次的に行いたい場合, プログラム全体の流れについて影響するような条件判定などである. この場合, フォートラン 77 そのままとはいっても,while や case などの構造化言語風の制御文が拡張されているので, 純 77 よりは使いやすいかもしれない. また, 実例は不明だが, 記憶構造に直接関係するような書式などは並列部への適用などが制限されているらしい. たとえば EQUIVALENCE 文である.

[4] 例としてあげたプログラムで, フォートラン 77 そのままの配列宣言をしているが, もっとフォートラン 90 らしい書き方もある. 全体としての考え方は並列演算などを追加したフォートラン 77 のアッパーコンパチブルである.

[5] 実プロセッサ 1 個が仮想プロセッサ 16 個を模擬することになる. ベクトルユニットを使用しないときは, ノードの Sparc チップが順に 16 個分の計算をすればよいし, 使用するときは長さ 16/4 のベクトル処理になる. 配列の大きさ=仮想プロセッサ数がはんば

ティションのプロセッサ数の倍数である必要もない. 配列の形に対しても同様で, 任意の $n_1 * n_2 * n_3 * \ldots n_7$ の形に対して自動的に分配が行なわれる. 自動的な分配は, 配列をプロセッサ数だけのそれぞれが連続な領域に分割することで行なわれる. [6] なぜなら, 自然なモデル化がプログラミングに採用されていれば配列内で近傍の要素との通信が多いと考えられ, この分配手法はそういった場合の通信の量を最少化するからである. [7]

配列の大きさを 実際的に制限するのは, それに要する各プロセッサでの記憶容量である. ただし, 配列の大きさをパーティションのプロセッサ数の 2^n倍などにした場合, 性能に変化があらわれるのは確かなようである.

文字配列, 並列ステートメントによって操作されていない配列などはそれに関係する記憶領域をパーティションマネージャにとるので, 自然にそれに対する処理はパーティションマネージャがやることになる. 配列が CM におかれるか,PM におかれるか, つまりどこにあるかということを配列の「ホーム」と呼ぶ. 上記のプログラムでの例では,A,B のホームは CM だが, i のホームは PM である.

これをまとめると, 配列に対する操作がかならず並列にできるわけではないし, 並列化の効率も場合によるということである.

また, プロセッサ間の通信のことも考えにいれるべきである. 差分法や行列に対する操作などは, どう考えてみても配列のある要素を値を計算するのに, 自分自身か他の配列の違う場所にあるデータを必要とするし, 配列の値の合計, 積分操作なども同様である. [8] このような場合にプロセッサ間通信が発生するが, プログラマは配列に式を代入したり, 合計を求める関数を CALL したりするだけである. プロセッサ間通信自体をするための命令というのは無い. いずれにせよ, こういった処理が仮想プロセッサが自分で持つデータに対して演算するよりコストが高いのは確実なようである.

配列の分配にたいしてプログラマが明示的にできる唯一の指示は, LAYOUT,ALIGN,COMMON ディレクティブである. コンパイラの自動判断では不十分だと感じた場合に使用する. [9] COMMON ディレクティブは COMMON ブロックのホームを強制的に決める. ALIGN ディレクティブはある配列をずらして他の配列の要素と仮想プロセッサ上で重なるようにする.

ALIGN と LAYOUT ディレクティブは普通の配列のホームを強制的に決定する.

同時に LAYOUT は, その配列が実際にどんな風にプロセッサ上にわりつけられるかを決定する. プロセッサ間通信が無ければこの機能はどうでも良いのだが, 配列はどれでも n 次元メッシュに似た構造になっているのに, 実際のプロセッサ間ネットワークは図 1 の箱の中に示した「Fat Tree」であり, わりつけ方によって得意な通信のパターンとそうでないのが出てくるので,LAYOUT であらかじめ有利なように設定するのである. プロセッサ間通信は数種に分類される. これらがハード上でどう異なるかは不明だが, 後述する Prism の出力によって種類を識別することは可能である. LAYOUT 文で,:NEWS もしくは:SEND とすることで, 配列のある軸をそのタイプのプロセッサ間通信に向いた形に配列させることができ,2 次元以上の場合は各軸についてプロセッサ間通信オプティマイズの重視度を数字で設定することができる. [10] また ,:SERIAL と指定することにより, その軸にそって全要素を同一プロセッサのメモリに積ませることもできる. もしも

な値のときでも, 適当に NOP か待ち時間をすることによって問題を解決できると考えられる. ただしこれはプロセッサの遊休率を高くしていることになるので性能は低下するし, またベクトル処理の場合はその処理速度がベクトル長に大きくかつ非線型的に依存することを考慮すべきである.

[6] ひどくこみいっているので説明しないが, 各 PN 内でのベクトルユニットへのデータ分散も同様な方式を踏襲する.

[7] 後述するコンパイラディレクティブは, 実際はこの分配とプロセッサへのアサインを制御する.

[8] 合計, 最大値のような全部まとめた結果について単一の結果が出るようなものは, プロセッサ間通信ネットワークを通じて, 収集しながら計算するのだそうだ. 特に収集/処理されるデータが Int 型の場合には, その処理はネットワーク・ハードウェア自体で行なわれるらしい.

[9] たとえば, コンパイラにまかせた結果としてプログラマが考えているのと違うことをオブジェクトがやってしまう場合である.「デバッグ」の項参照.

[10] しかし, これでうまく性能を向上できたことがあまり無い. 「失敗例の研究」の項で一つだけうまくいった事例を示す. ホームを変えさせるだけが目的ならば, 普通は全軸に対して:NEWS を指定すれば良い.

全軸に対して:SERIAL を指定すると, 強制的にホームを PM にすることになる.

また,LAYOUT ディレクティブによって, より仮想プロセッサのシステムを介さず, より物理的なプロセッサの構成に近いところで並列部へのわり当てを指定すること, また, パーティションの大きさを変更せずに使用する物理プロセッサ数を変更することが可能のようである. この方法は, 現時点ではなぜかマニュアルには出ていないようだが, 講習会で使用された資料の方には掲載されている. [11]

1.2.2 コンパイルのしかた

1.2.3 コンパイル

- CM-Fortran のコンパイラ名は,cmf
- C*のコンパイラ名は,cs

である.

```
cm5-cp1[1]% cmf ffttest.fcm -o -vu -g PSORT.o
cmf [CM5 VecUnit 2.1.1-2]
Compiling parallelsort.fcm

cm5-cp1[2]% cs -sparc cmprot.cs
C* driver [CM5 SPARC 7.1.1]
Compiling  cmprot.cs
```

これでオブジェクトファイル a.out ができる. 通常のコンパイラと変わりない.

1.2.4 実行のしかた

通常のプログラムと同じで, オブジェクトファイルの名前を入力するだけで良い. リダイレクションなどもそのまま利用できる.

```
CM5(64VU):garage[24]%a.out > res
a.out > res
CM5(64VU):garage[25]%wc res
wc res
    1024    1024   12288 res
CM5(64VU):garage[26]%head -7 res
head res
   3.000000
   3.000000
   3.000000
   3.000000
   3.000000
   3.000000
   3.000000
CM5(64VU):garage[27]%
```

前章で述べたように, これは CM5 の並列プロセス管理手法によっている. あるプログラムはあるパーティション上でプロセスとなり, この時パーティションの全ノードにひとつずつのプロセス, またパーティショ

[11] CM-Fortran の各文の実際の使用例などは「CM-Fortran Reference Manual」に出ている. 特に,LAYOUT,ALIGN,COMMON ディレクティブについては, 「Array allocation」「Appendix:Compiler Directives」の両項を参照.

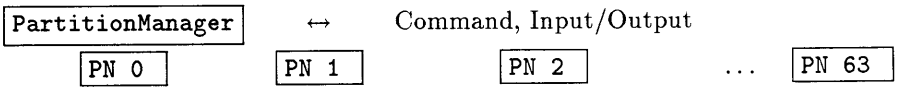


図 1.3: CM5 のプロセスのイメージ

ンマネージャにそれらの「代表プロセス」が一つ生成される. そしてこれらがひとくくりの「並列プロセス」としてあつかわれる.

ややこしいのはこういう構造は CM-Fortran からは見えないという点だが, 頭に入れておくと実際に使用する際に便利である. [12]

1.2.5 並列化するステートメント

並列化するステートメントについてまとめる. あまり複雑なプログラミングをしてみたわけではないので, 詳細はマニュアルを見てもらうことにして, どういう種類の物があってどういう特性を持っているかだけを書く. 後続の章に挙げた例題プログラムの中から書き方の例を探すこともできるだろう.

配列に対する代入文

先のプログラムで使用したものである.

例えば,

$$A = B * C$$

のようなステートメントがこのような配列変数に対して実行されると, 対応する配列の要素のそれぞれについて,

$$for \quad i_0 = 0 \quad to \quad i_{0max}$$

$$for \quad i_1 = 0 \quad to \quad i_{1max}$$

$$for \quad i_2 = 0 \quad to \quad i_{2max}$$

$$A_{i_0,i_1,i_2,...} = B_{i_0,i_1,i_2,...} * C_{i_0,i_1,i_2,...}$$

$$end$$

$$end$$

$$end$$

という操作が, それぞれの仮想プロセッサで, 独立に, かつセマンティクスの上から見て,「同時に」行なわれる.

これが, このような言語で書いたプログラムが多次元メッシュ型のトポロジーを持つ仮想 SIMD 型の超並列計算機として動作することを, また, 通常の逐次型言語では i_0, i_1, i_2 に対する多重ループによって記述するのと同等のことが上記のような単一のステートメントで記述, 実行できることより, コンパイラが限定された範囲での自動並列化能力を持つといえることを意味する.

これは最も単純な形で, マニュアルを調べればこれだけで結構いろいろなことができるのが分かる. 調べきれていないので一番おもしろい例だけを紹介する.

[12] 別に CM-Fortran でなくとも常にこうなる. CMMD Hostless モードでもパーティションマネージャに「代表プロセス」は現れる.

配列 i(64),r(64) があって,CM がホームだったとする. i には,64,63,62,61.....3,2,1 という数列を収容していたとする. ここで,

r = r(i)

というのを実行すると,r の内容はそっくり天地逆転する. 完全にランダムな値の入っているポインタに対する代入命令みたいなのを一遍でできることになる. おもしろいのは, 二つずつの組みにして考えると, データが互いをすりぬけてスワップしてしまうことである. ただしこのようなランダムなプロセッサ間通信は, 規則正しいものに比して低速らしい. [13]

アレイコンストラクタ

この計算, 代入も独立に行なわれるのだろうが, 等差級数程度しか代入できない. 上記の配列に対する代入文の例で出てきた配列 i にあの数列を代入するのは,

```
i = [64:1:-1]
```

とする. つまりカッコの中は DO ループの変数指定と同様である. これも簡単な例で, 実際はもっといろいろなことができる.

FORALL 文

正式な書式を省略するが,

- FORALL (多重 DO ループの制御変数とその範囲指定だけを抜き出したのにそっくりな部分) 代入文

という形で, 代入文は,

- その制御変数でアクセスされている配列 = 「表現」

でなければならない. これは「表現」に, カッコ内で使用した制御変数を書いて良いところが特性である. その「表現」の値を各部分で独立して計算して独立に代入する.

たとえば, 電磁気学的な計算などの際に, ポテンシャルは空間メッシュ上の位置, つまり配列の添え字によって各点で決まるわけで, 表現にはそれを表す関数形を書いてやれば良い.

WHERE 文

- WHERE(条件)
- 文
- ENDWHERE

という形で, 配列のなかで条件を満たす部分だけにある操作を行う.

例えば, ある関数の離散的な各点ごとの値が一次元配列 f におさめられているとして, それが一定値以上の区間のみにおいて, 一次元配列 g におさめられている別な関数に加算するといったようなプログラムは, 通常のプログラム言語の場合, ループ構文を用い, その制御変数をインデックスとして一次元配列 f,g の各内容にアクセスしていくというものになるだろう. これに対して,CM-Fortran では,

```
real*8 F(1024),G(1024), S
integer*4 i
```

[13] この手の処理の例は, 先日行なわれた ThinkingMachine 社による講習会の資料にたくさん出ているので, 見ると良いだろう.

```
where ( F > S)
F = G + F
endwhere

end
```

のようになる. ここで where/endwhere 構文は条件文を満足する仮想プロセッサのみにおいて構文の内部を実行する. つまり同一のサイズをもつ複数の配列の対応する要素, 例えば,

$$\{f(945), g(945)\}$$

が一つの仮想プロセッサと見なされるわけである. これは C*言語では配列のサイズつまり仮想プロセッサの構成を Shape という型の一種と見なして, より厳密に制御される.

f77 の IF 文はプログラムの流れ自体を変更するが, WHERE は条件を満たさなかった仮想プロセッサを,ENDWHERE までの間一時停止させる. そのあいだに問題の仮想プロセッサ群に与えられた命令は一時停止していないプロセッサだけが実行するので, 仮想プロセッサごとに処理の流れを変更できることになる. つまり並列版の IF 文と言っていい. [14]

その他イントリンシックな関数

配列を引数にとったり, 配列を出力するイントリンシック関数として種々のものが用意されている,

これらを用い, 画像処理や偏微分方程式の逐次解法といった, 各要素に比較的一様な操作を要求されるアプリケーションを作成することができると考えられる. 例えば差分方程式が与えられている場合,CM-Fortran の書式はより自然にこれを記述でき, かつかなり効率の良いオブジェクトを生成できる.3 次元ラプラス方程式の場合, その差分法によるプログラムの中心部は,

```
real*8  d( DX , DY , DZ )
real*8 d2( DX , DY , DZ )
integer iota
integer i

d2 = (1.0/6.0)*(
.  cshift(d,1, 1)+cshift(d,1,-1)+
.  cshift(d,2, 1)+cshift(d,2,-1)+
.  cshift(d,3, 1)+cshift(d,3,-1) )

 iota = sum(abs(d2-d))
 d = d2
```

と, ほとんど一つの代入文だけで書ける. ここで,

- cshift 配列の内容をある軸にそってずらす.
- abs ある配列の内容の絶対値が代入された配列を与える.
- sum 配列の内容の合計が代入されたスカラーを与える.

[14] どの仮想プロセッサが活動していてどの仮想プロセッサが活動していないか, ということを TMC では一貫して「コンテクスト」と呼んでいる. そしてたしか, コンテクストはサブルーチンコールのときにデータと一緒にわたされるそうである. これを利用すれば, かなり面白い処理が可能になるだろう.

などはCM-Fortranのイントリンシック関数である.absは単なる通常のabsの拡張に過ぎないが,cshiftとsumなどのはイントリンシック関数は本来なら当然プロセッサ間通信を必要とするのだが,実際に書いてやらなければならないことはこれらの関数やサブルーチンを配列を引数として使うことのみである.

また,例えばd2に関する代入文が各仮想プロセッサについて「同時に」行なわれるように見せることは,本来なら同様に複雑なプロセッサ間の同期操作を必要とする.これらの並列計算機において不可避な問題が,CM-Fortranにおいては仮想プロセッサ,擬似SIMDの考え方によって完全に隠蔽されるという大きな長所がある.

また,特にC*においては仮想プロセッサ間に上記のような規則正しいものの他に不規則なパターンでの通信の記述力も強化されており,これによって数百万個のランダムなコネクションをもった並列計算機としての使用が可能になっている.ただし,いづれの場合も,CSHIFTのような規則正しいパターンでの通信の方が代入によるランダムな転送よりも高速とされている.

ユーティリティサブルーチン

と言う一群のサブルーチンがあって,なぜかヘッダをインクルードし,リンカには特別な指示なしで使用するようになる.

実行効率などは不明だが,これでしかできないような処理があるようなのでぜひ一遍目を通すべきである.

ヘッダは

/usr/include/cm/CMF_defs.h

にある.各ルーチンの説明は,「CM-Fortran User's Guide for the CM-5」に掲載されている.

15

1.2.6 コンパイラオプション

オブジェクトの名前とかリンカへの指令といったオプションは普通のf77やccと何も変わっていないので,CM5の性質をよく示していてcmfに特有な物を紹介する.大部分がC*のフロントエンド,csと共通である.

-list

ソースコードと,変数の一覧,コンパイル時の状況などのリストを作ってくれる.こういったものは普通のコンパイラにもあるが,わざわざここで述べるのはこのリストの中に,配列のホーム,つまり並列部にそれが置かれているかどうかが表示されているからである.

-cmsim

CM5のオブジェクトではなく,フロントエンドでそれをシュミレートするオブジェクトを作る.このオブジェクトは,完全にフロントエンドだけで動く.Ts-Daemonの無いPM上で,これをPrismで観察することさえ可能である.さらには,CM5と関係のない個人のSS2でさえ動作させることができる.また,厳密に計ったことは無いのだが,この時の方がコンパイルは早く済むような気がする.だから文法チェック用に有効かもしれない.ただし,記憶容量,実行速度,さらにアーキテクチャそのものが比較のしようもないほど異なっているので,実行時エラーを予期する能力は0と見て良いだろう.[16]

[15] 並列の入出力というのもあるが,専用のデバイスに対してしか使用できないので,上の例では普通のwrite文にした.専用デバイスとは,たとえば並列磁気ディスク,Data Vaultなどである.Jaistの CM5にも,SDA,Scarable Disk Arrayというのが本体に内臓されている.そしてその入出力命令が,このユーティリティサブルーチンか,CMFS(Connection machine file system)ライブラリからでないとできないようになっている.

[16] 並列計算機による高速化効率を議論する際に,使用するCPUの数をパラメータにして計算速度を考察する例が多い.-cmsimオプションによるオブジェクトは,一つだけのSparcチップで実行されるようコンパイルしたことになるからこの場合に役に立ちそう

-vu

ベクトルユニットを使用する. このベクトルユニットというのが普通の数値演算加速用の付加処理装置というのとかなり性格が異なるため, 数値演算だけではなく, ほとんどすべての場合にこれを使用したオブジェクトの方が高速に実行される. ただし例外もあるようなので, 場合によっては両方で実際に速度を比較されることを推奨する.[17]

-veccode

ベクトルユニットのアセンブラ,dpeac による表記でコンパイルの途中結果を示したファイルを作る. ふつうのコンパイラの -S オプションと同様だが,dpeac による表記はこれでないと得られない. [18] CM5 のアセンブラの研究に役立つだろう. [19]

-g, -cmprofile

-g オプションをつけないと dbx と同様,Prism にかけられない. -cmprofile オプションが無いと Prism での所要時間と実行形態の測定機能が使用不可になる.

-O

オプティマイズする. 具体的に何がどうなるのかは不明なのだが, オプティマイズするとコードの実行結果が変更される場合がある, という注意は普通の計算機より深刻に受けとった方が良いようである. また, このオプションはあるか無いかのみであって,-O2 や-O4 というのは無い.

-fecommon

COMMON ブロックをデフォルトでフロントエンドにもってくる.

1.2.7 計算機資源

ここで, 重要な二種類の資源についてその算出法について述べる. それはメモリ使用量と演算時間である.

つねに論理パーティションの全プロセッサ, つまり 64 個, が使用され, 仮想プロセッサの考えが導入されている以上, 可能な計算の大きさを決定するのは仮想プロセッサの内容が収容されるメモリの大きさである.

メモリ使用量は自動的に取得される作業領域の存在など種々の避けられない事情から, 配列の大きさから単純に割り出すことが困難である. cmps コマンドをフロントエンドに入力することにより実際にプログラムが動いている時のメモリ使用量を知ることはできる. 例としてプログラムが一つも動いていない場合,cmps コマンドは

だが, 未確認だが CM5 の構造を模倣するためのオーバーヘッドがかなりかかっているのではないか? と思われるし, ベクトルユニットを使用すると話が全く異なってきてしまう

[17] ベクトルユニットを実際にどう使用するかはコンパイラが考える. 結果としてどう使っているかは実行時に cmps コマンドを入力して VUS/VUH の欄を見れば良い.0K とか 4k とかでなければ使用されている. ベクトルユニットは PE 当り四つあるので, この値の四倍が 1PE でベクトルユニットが消費している容量となる. ベクトルユニットの処理効率を直接判定できるベクトライザのようなツールは今の所存在していない.

[18] ベクトルユニットは自分で主記憶から命令フェッチせず, Sparc チップが特定の番地に書きこんだデータをインストラクションとして実行する, ようになっているらしい. したがって-S オプションによる普通のアセンブラの中にもこれらのインストラクションは書きこみ命令のオペランドとしてはっきりと出ている. ただ,dpeac は Sparc アセンブラを拡張したマクロ言語であり, これらを単一のベクトル命令として表記するので分かりやすいし, それで書いたものを専用アセンブラの dpas によってコンパイルできるらしい.

[19] 役に立ったので, アセンブラのマクロ群である CDPEAC でかなり複雑でしかも高速なプログラムが作れるようになった. これには,C*もしくは CM-Fortran で書いた親プログラムと連携させるか, CMMD のノードプログラムに埋めこむのと二つの方法がある. 今の所この両者の混在使用は不可能である. サンプルプログラムは,"DPEAC Reference Manual" の末尾に掲載されている.

```
cm5-cp2(36)%cmps
64 PN System, 30304K mem. free, 6996K VU mem. free, 0 procs ..........
```

と回答するが, 最初の二カラムから 64 プロセッサのパーティションであること, またメモリが一つのプロセッサについてあと 30304K バイト空いていることをしめす.

物理メモリは 32768KByte あるので, これと 30304K バイトの差の分は PN のオペレーティングシステムが使用している. ここで何かプログラムを実行させると, まずそのオブジェクトコード[20] が各 PN にロードされる. これで各 PN において, 例えば 200Kbyte ずつ均等にメモリが消費される.

その後, ヒープ, スタックなどにメモリがアロケートされ, 消費されていく. この時, 特に CMMD モードの場合, 各 PE ごとに消費量が異なってくることが考えられるが, こういう場合に cmps が返す値はどう代表されるのかは不明である.

ここで,-vu モードでコンパイルしたコードの場合はヒープ, スタックなどが「ベクトルヒープ (VUH)」「ベクトルスタック (VUS)」などとして使用されるが, これらは四基のベクトルユニットに対して均等に確保され, その量が表示される. つまり,cmps の VUH 使用量の欄に 1000KByte と表示されたら, 実際には 4000KByte 使用されているのである.

なお, こういったことは PN のオペレーティングシステムの malloc システムコールに対応するものによって実現しているわけだが, 同一プロセスの実行が続いている間にこれを開放して再利用するといった非常に動的なメモリの使用能力については未知である.

第 2 に, 処理に要する時間の測定について, 現時点で CM-Fortran のタイマにはいくつかの問題がある. 時間測定の方法として,

1. そのプログラムの外部の正確と思われる時計を使う. 例えば, 正確なタイマをもつどこかのプログラムにプロセス間通信でキューを出す, または, 画面を見ていてストップウォッチではかる.

2. プログラム全体の所要時間を教えてくれるような機構を用いる. 例えば,time a.out とする.

3. 後述する prism の時間計測機能を使用する.

4. cm_timer_print 関数で, 画面にタイマのデータを出力する.

5. cm_timer_read 関数で,Fortran の変数にタイマのデータを代入する.

きちんとしたカテゴライズになっていないと感じるだろうが, プログラム自身が自身の所要時間データを知る手段が cm_timer_read だけであることに注目してほしい. そして現時点ではこの関数は時々全く理屈に合わないようなデータを返してくることがある.

また, 処理時間の測定においては,CM5 の並列部で動くプロセスは常にそのパーティションマネージャで動くプロセスをともなうことを思い出してほしい. 何らかの原因で cm5-cp2 に過負荷をかけると,CM5 全体が影響を受けるので, 並列部のパフォーマンスも変化してしまう.

適切な方法は場合によって異なるので何とも言えないが, プログラムの性能評価のさいには種々の問題があるので自分が何を測定しようとしているのかはっきりさせておき, 念のため極端な条件下や繰りかえし測定での性能が矛盾したものでないことを確認することをすすめたい.

---

[20] の,PN で実行される部分.cmmd-ld,cmjoin などのマニュアルを見よ.

1.3 デバッグ

1.3.1 デバッグのしかた

前述のように, 並列部のプロセスは必ずPMの通常のUNIXのプロセスに付随していて, 基本的な処理の流れはこの通常のUNIXのプロセスに従う. したがって, ソースに変数の書き出しポイントをつけるような古典的なデバッグの方法も可能だし,CM-Fortranが強力な処理の枠組みを用意しているため,PEごとにハードの物理的な構造を心配するようなことは滅多にないが,[21] それでも並列機のデバッグは困難である.

多少の経験から, 役に立ちそうなことを挙げてみよう.

プログラムが実行時エラーによって終了したとする. この時にはつぎの二通りのうちどちらかの事態が起こる. 前者の例は

```
Segmentation Fault (core dumped)
```

といった感じで止まる. もうひとつは,

```
*** RTS-FATAL-UNIX: Floating point operand error
Traceback follows:
(Only the first location is guaranteed to be valid.)
   pc = 0x1361c       _mont_, line 899
   pc = 0x135fc       _mont_, line 899
   pc = 0x3460        _MAIN_, line 268
   pc = ox22f9c       _main_
Abort (core dumped)
1470.2u 8.9s 28:31 86% 0+968k 35+446io 114pf+0w
```

といった風に止まる. この場合には実際にMAINのline 268からコールされたサブルーチンmontのline 899で落ちていた.[22] 後者の場合がほとんどだが, 止まった場所をこのように指示してくれるので, デバッグに大変有効である.[23]

また, 最初の行のFloating point operand errorなど, 実行時エラーとして具体的に何が起こったのかの情報も提供されるが, 一度, 記録を残していないのだが, 配列の添え字の範囲を超過した場所にアクセスしようとした場合に,

```
Bad PN address.
```

というようなメッセージが返ってきた.[24] この問題になった配列のホームはCMだったので, 配列の範囲の以上が結果としてPNのアドレスの以上として表れたのだろう. というわけで, 言っていることはあてにできるのだが, メッセージが通常のUNIXマシンと多少変わった表現を使っていることがある.

また, コアダンプがデバッグに利用できるのは並列部での実行時エラーでも同様である. 並列部でのエラーではPNの番号を名前に含んだコアが, それも特定のいくつかのPNに対して生成される. これは実際にエラーがそのPNで起こったことを表すと考えられる. 解析するのにはpndbxを使用するのだが, その方法はよく実験していない. また. これらコアのうち一つは各ノードからの「エラーメッセージ」例えば,PN0では異常なし,PN1では異常なし..... PN33でセグメント・バイオレーション..... というような全PN分

[21] ただしこれが役に立たなくて, とにかくどうしても落ちてしまうという場合には, 配列のどこかの内容に変な値が入っていないか, また宣言した範囲外の変な場所をアクセスしていないか念入りに注意すること, よく調査してないが, これらは不可解な異常終了の最大の原因になるようである.

[22] ただしこういうのの通例として, 実際にはエラーでありえないような行を指示してくれることも一度あったので, 注意が必要である.

[23] この両者の違いが, エラーのあるタイプを指摘するのではないかと考えて再現性があるようなエラーを含んだモデルプログラムを作ろうとしたが, うまくいかなかった.

[24] この場合も, 配列のアクセスの仕方における特定のパターンの誤りを指示するのではないかと考え, 再現性があるようなモデルプログラムを作ろうとしたが, 失敗している.

の情報をテキストファイルとして記録している.[25] 別の一つには, 異常終了の原因になった全 PN のベクトルユニットのベクトルレジスタと全内部フラグのその瞬間の内容の一覧表が表示される.[26] しかしこれらは抽象度が低すぎてあまり役には立たないと思われる.

CM-Fortran でデバッグの基本は, 各仮想プロセッサつまり配列の各要素に妙な値が代入されていないかどうかソースをチェックすることである. これは後述の prism が与えるデータよりも基本的である.

また, 次に,CM-Fortran を使用して初めて出くわしたタイプのエラーについて述べる. これらは, 主に配列のホームに関するものである. とくに何も指示しなければ, 配列のホームはコンパイラによって自動決定されるが, その判断のもとにされるのはそれがどのような操作をされているかということである. 非並列ステートメントだけによってアクセスされているなら, ホームを CM にわざわざすることは無いと判断される. そしてこの判断はメインルーチンについて, また各サブルーチンごとについて独立に行われる. これでコンパイルは無事に済むのだが, 実行時にサブルーチンとメインルーチンの間を行き来すると, ホームが FE のものから CM のものに代入されるような場面に遭遇する. そして, ある種の並列ステートメントはこれに対応できないのである. 対策としては,LAYOUT ディレクティブを用いて強制的にホームの矛盾を正させるのが一番簡単だが, その配列に対して処理の種類を変えておなじ効果を得たり問題になった並列ステートメントを等価なものに置きかえても良い.

また, これはエラーでもクラッシュでもないのだが, プログラマが気がつかない所で, 配列のホームや処理の実質をフロントエンドに持ってこられるというのはそれだけで問題になりうる. 規模が小さいうちは良いのだが,PM は並列部に対して理論演算性能でも 2000 倍ものひらきがある. 遅すぎて全然進まないため, クラッシュ同然の状態になってしまうのだ. 記憶領域の大きさについても同様な傾向がある. 本当に大な配列は CM にしかおさめることができない. その上この大な配列にデータを置いただけで, 実際の処理はフロントエンドにデータを転送しながら行なっていたのでは大変な処理時間の無駄だし, これが有限な CM メモリを使用するために他のプログラムが走らせられないかもしれない. 要するに, 並列部をフルに活用するようにしなければならないということである.

1.3.2 グラフィカルデバッガ PRISM

並列計算機の場合, 単にバグを除くためではなく, 計算機が実際のところ何をやっているのか把握するのにもデバッガが重要となる. CM5 の場合, 並列部にアクセスできるデバッガは, dbx の並列拡張としての pndbx と, グラフィカルなデバッガ,PRISM というのがある. ここでは PRISM の機能の概略を説明する. ブレークポイントや処理のトレースといったことを並列プログラムに対してできるのだが, 最も重要な機能は,

- 並列部の変数の内容.
- あるステートメントによって引きおこされた処理の時間的な量とその種類.

をグラフィカルに表示する機能である.

前述のように対象となるコードは必要なオプションをつけてコンパイルされていなければならない. X ウインドウ関係の設定の後,

---

[25]「抽象度が低い」という意味は,CM5 が並列計算機であることを強調することによって明らかになる. 一般には上のような場合,PN33 はたまたま一番最初にバグに激突したにすぎない. 頻繁に同期がとられているとはいうものの, MIMD ハードウェアは各部分で基本的には勝手なことをしているだけだから, ランダムな時間的ずれが PE 間につねに存在する. 何度も同じことをすれば PN33 はは PN46 だったり, また PN55 だったりするだろう. つまりこの情報を利用するためには, バグがその PN の担当領域だけなのかどうかということを考えなければならない.

[26]これも役には立たない. ベクトルユニットを制御している SPARC CPU のレジスタ内容の方が優先課題であるし, それが分らなければベクトルレジスタの内容が配列のどの部分にあたるのかも分らないであろう. ただし開発しているプログラムが極めて低レベルの場合にはこれらのコアファイルは充分役に立つ. あらかじめ狙っている PN が先に落ちるようにタイミングを調整したり, プログラムの任意の場所で異常終了するようにしたりできるからだ.

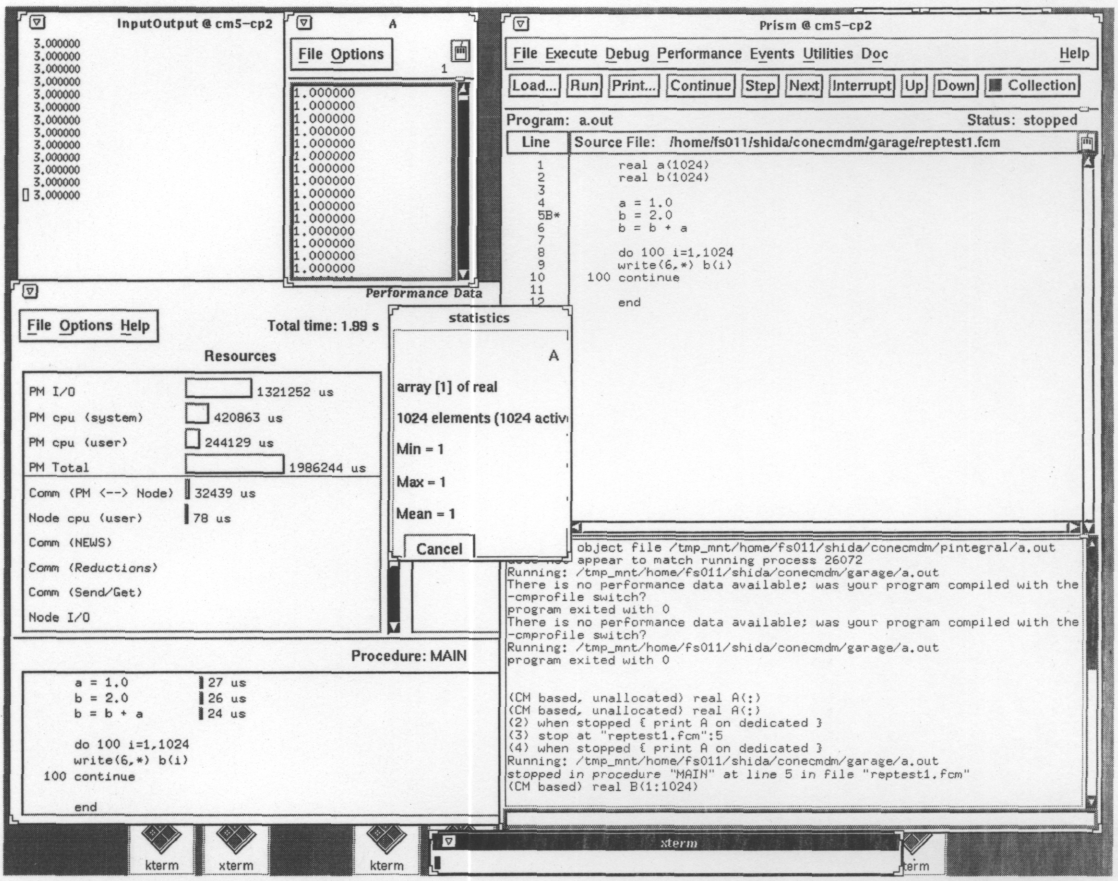


図 1.4: prism 使用中の例. ソース表示とメイン操作用のウインドウ, 変数 A の内容や性質をしめすウインドウ, 実行時の処理の内容を表示するウインドウなどが分かる.

```
prism &
```

として,prism を起動する.

例として挙げた画面では数値データをキャラクター表示しているが, 折れ線グラフ, その集合としての,2 次元配列の内容を起伏で表現したサーフェース, カラーかモノクロのグラデーションで表現するディザなどの簡単なグラフィックス表示のモードがあり, 変数の状態を直観的に把握するのに非常に有用である. prism は, ディスプレイ関係の設定をしておけばもちろん自分の WS や X 端末でもグラフィカルに実行できる. ただ PM から送出されてくる描画用データが大量なために反応に非常に時間がかかる場合があるので, 注意されたい.

prism の操作は,GUI なのでやっているうちに理解できるだろう. またマニュアル「CM5 Technical Summary」には,prism の概要が画面のコピーとともに掲載されているので, おおまかにどんなものか知りたい人には良い.

また,94 年度春期に行なわれた OS バージョンアップにともない,「Nodal-prism」が導入された. まだ使用の経験は蓄積されていないが, これで MIMD スタイルのプログラムのデバッグもある程度可能になった. ただし機能の複雑化により従来の prism 以上に重くなったことを指摘しておく.

1.4 行列の乗算

並列プログラミングのもっと高度な例として, 比較的簡単で応用上も重要な物として, 行列の乗算をとりあげる.

行列の次数を n として加算, 乗算がそれぞれ n^3 として flops 値を単純に評価できるというのも利点である.

行列の乗算には以下の方法がある.

1. CM-Fortran-Intrinsic の,MATMUL 関数を使用する.A,B,C が二次の配列だとして, C = MATMUL(A,B) とするだけである.
2. CM-Fortran の他の Intrinsic な機能, 例えば DO ループ, を利用して, 普通の計算機に対するような行列乗算のコードを書く. これをやると, 全く並列計算が行われない.
3. CM-Fortran の Intrinsic な機能のうち, 並列化される構文を使用して, 行列乗算のコードを書く.
4. 科学計算ライブラリー CMSSL の中の行列乗算ルーチンを使用する.

このうち三番目だけが, 自分でプログラムを書いてみたいという目的にかなっている. また MATMUL(A,B) のスピードは, 単精度, ベクトルユニット使用の場合,

- n=1024 において,246.8Mflops.
- n=4096 において,442.2Mflops.

倍精度, ベクトルユニット使用の場合,

- n=1024 において,233.4Mflops.

[27] という程度だったので, もう少々なんとかならないかという気もおこってくる.

そこで今回作成したプログラムを以下に示す. おそらく最良の方法ではないと思うし, 並列アルゴリズムだなどと自慢できるものでは無いのだが, それでも過剰なくらい細く説明するのは, 並列計算機ではアルゴリズムに対する考え方を変える必要があるという認識を強調するためである.

CM-Fortran の枠のなかで, 多数のプロセッサに処理を分割するわけだが, まず単純に配列に対する二項演算だけでは行列乗算が不可能なのはあきらかである. 必要とされるスカラー乗算の全ての組みについて巨大な配列に展開すれば, $C = A * B$ だけで済むかもしれないが, これでは記憶領域から考えて小さな行列しかあつかえなくなるし, どうせ処理の前のデータ分配と後の部分的合計操作の時にはもっと複雑な文が要る. そしてそういうのは例外なくプロセッサ通信を起こす.

そこで, 行列は 2 次元配列で自然に表現できるのだから, 処理の間ずっと対象が 2 次元配列のままでいるようなシンプルな方法にすること, プロセッサ間通信が不可避なのなら効率の良い一様な通信のみに限定すること, などを目標にして, 次のようなものを考えた.

```
DIMENSION A(N,N),B(N,N),C(N,N)

(LAYOUT 文)

DO 100 K=1,N
FORALL (I=1:N,J=1:N) C(I,J)=C(I,J)+A(I,J)*B(I,J)
```

[27] 組みこみ関数だから当然かもしれないが, オプティマイズの効果はほとんど無い. それから,cmps による VU メモリの使用量と速度について, 相関があるような気がするといったが,MATMUL はそれが成立しない顕著な例となっている.n=4096 において, 倍精度のデータが無いのは, VU メモリ 領域が不足して実行できなかったためである.

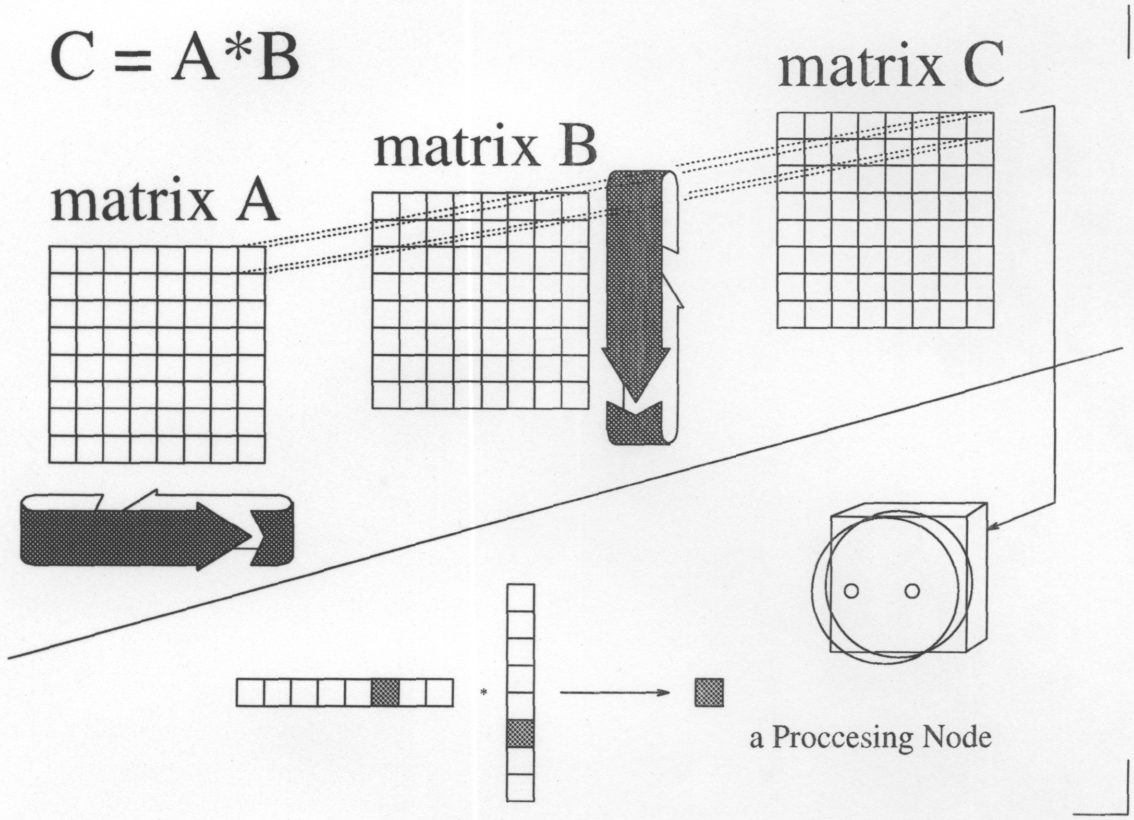


図 1.5: 行列乗算アルゴリズムの図解. 矢印で示される仮想プロセッサは点線で結ばれた部分を記憶している. この自分の記憶域内のデータ同士の乗算のあと, プロセッサ間通信を 2 回行う. 帯状の矢印のようにサイクリックなシフトをおこなうのである. 仮想プロセッサのレベルでは, 常に互いに同等なことをしていることになる. 一つ一つのプロセッサは図の下に示すような単純な内積演算をしている. ただしある瞬間には陰影のついている記憶領域しか持っていない. 下図を上図にあてはめると, 行ベクトル, 列ベクトルの次要素が隣接プロセッサから送られてくることが分かる.

```
      A=CSHIFT(A,AXIS1,1)
      B=CSHIFT(B,AXIS2,1)
100   CONTINUE
```

各仮想プロセッサは, それぞれ一つの内積を一貫して計算する. f77 の DO ループによってシリアルな流れが与えられるが, その一回ごとにベクトルの成分一組ずつの演算と, 合計値への加算が行なわれる. そして次のベクトルの成分を得るために, 規則的プロセッサ通信を起こしている. これは CSHIFT によっておこなわれるが, 配列 A と配列 B では直交する方向にサイクリックシフトするため, 正しく行ベクトルと列ベクトルの次要素が隣接する仮想プロセッサから送られてくる. 以上を行列の次数回だけ繰り返えす. 結果が入る配列 C のうち, 通常の計算の時の順序で内積を求めているのは A(1,1) だけだが, 他の仮想プロセッサも前後にずれた順序で正しく内積を計算しているので心配はない. 配列 C の配置は不変だからループ終了時には結果がそのままの形でえられ, 配列 A と配列 B は一回回転してもとの配置にもどっている. 仮想プロセッサは配列の大きさそのものだけ必要である. シリアルなループの間, プロセッサ間通信によって始終処理が中断されているが, 仮想プロセッサ数が大ならベクトル長は十分長くなるので, 心配ない.

次にこのプログラムの計算速度を前述の考え方で評価する. 結果は, 単精度の場合, n=4096 において,778.3Mflops, n=8192 においては,1008Mflops となって,MATMUL ルーチンよりは高速であることが分かり, 一応の成功と言える. が, 理論最大値には遠くおよばない値である.

そこで, より一層の性能向上と, その過程を通じてオプティマイズの方法をさぐるために, 以下のような方法をこころみた.

1. LAYOUT ディレクティブを使い, 行列の次数によっては半分かそれ以上の処理時間を消費していた CSHIFT が高速化するよう配列を最適化する. [28]

2. ある種のベクトル型スーパーコンピュータのコンパイラで有効な様に, 処理をもっと複雑にする. 最内側ループ内の演算子の数を増加させたり, 独立したデータの流れの数を増加させることによって, それまで使用されていなかった演算ユニットを使用する余地をコンパイラに与えるやり方である. [29]

3. ベクトル長を延長する. たとえば行列を, 自分と同じ大きさの配列四つにコピーする. これらでそれぞれ次数の 1/4 ずつずらして内積を計算し, あとで合計すれば, ベクトル長は理論上四倍になり, 途中中断のあるシリアルな繰り返し長は 1/4 になるので, 変換のためのオーバーヘッドはとりかえせる可能性がある.

   またより簡単な実験手段として, かわりに行列の次数を限界まで大にしてみた. [30]

これらについて, 網羅しつくしたとは言えないけれども, 随分たくさんの変形版を作成して実験した.

その結果は, ほとんど失敗であった. プロセッサ間通信はおそらく最初の段階でそれ以上の速度にはできなかった. 他の方法でも, 書き方の変更によって予期してなかったところにプロセッサ間通信がおきたり, 前処理のオーバーヘッドが大になるなども低速化の原因になっていたが, それにプラスの効果が打ち消されたというよりは, はじめからこれらの手段の効果が希薄なようだった. 処理速度を規定しているのは何か別の要因のようである.

最終的に,(2) と (3) を組み合わせたような方法

```
c
      DO 110 KC=1,SZ4
        C1=C1 + A1*B1 + A2*B2 + A3*B3 + A4*B4
c
        A1 = CSHIFT(A1,1,1)
        B1 = CSHIFT(B1,2,1)
        A2 = CSHIFT(A2,1,1)
        B2 = CSHIFT(B2,2,1)
        A3 = CSHIFT(A3,1,1)
        B3 = CSHIFT(B3,2,1)
        A4 = CSHIFT(A4,1,1)
        B4 = CSHIFT(B4,2,1)
```

[28] それはそうと, より複雑な計算の場合には, たとえば前出のコードの乗算の部分と CSHIFT の間に他の並列ステートメントがはさまっていて, しかもそれが処理する対象のデータがその前後のステートメントと関係ない. ということもありうる. こういう時に, プロセッサ間通信とそれとデータの独立な演算とをパラレルにやってくれるのか? というのは重要な問題であった. ハードに関する知識の増加によって,CM5 の PN はネットワークインターフェースとの間の I/O の際に, 必ずメモリマップド I/O でワードを読み書きしていることが明らかになった. つまり, 通信と計算はミクロレベルでパラレルには行なえないと見なせる. マルチタスキングによって見かけ上並行に行なえるのは確かだが. ただし, もし I/O 専用プロセッサの装備によって通信と計算をパラレルには行なえるようになったとしても, 他の部分とのかね合いがあるからそれで飛躍的に性能が向上するとは思えない.

[29] ベクトルユニットは PN ごとに 4 基あるので, この方法が有効なのではと思えた. 思えたが, あとでハードに関する知識が増えた時に, ベクトルユニット間での演算器のチェイニングは不可能と分かったので, こういうことは無い.4 基のベクトルユニットもまた小規模のデータ並列マシンとして動くのである. ベクトルユニット内でのチェイニングは命令が乗算+ロード+その他である場合などに可能である. つまり演算子の数が増加すると高速化するのは本当である.

[30] いずれの場合も, それを実現するための書きかえによってプロセッサ間通信が変化することを本当は考慮しなければならないのだろうが, プロセッサ間のパターンや回数, 内容から時間を見立てる一般論が不明である.

```
110  CONTINUE
```

で, n = 1024 で一割程度高速化することに成功した, しかしこれは何でこうすると高速化するのかという理由づけがうまくできなかった.[31]

さらに詳細に分析しようとした結果, もっと困った可能性が出てきたのだが, それは「失敗例の研究」の項で述べる.

[31] 演算子の増加によるベクトル化率の改善が原因である.

1.5 パフォーマンス

パフォーマンス向上については一般的原則を述べるのが非常に困難だが, いくつかのの基本的な項目を追加してみる. ベクトルユニットの性質に間する考察は行列乗算と失敗例の章でも述べている.

この項でも問題は, 主にプロセッサ内での処理の高速化に関係している.

1.5.1 仮想プロセッサの配置

ある人が,[32] 3 次元空間内の多数の点の運動をシミュレートするプログラムを CM-Fortran で作った. つまり,

$$(X_0, Y_0, Z_0)$$
$$(X_1, Y_1, Z_1)$$
$$(X_2, Y_2, Z_2)$$
$$(X_3, Y_3, Z_3)$$
$$\vdots$$

というのを基本的なデータ構造にしてあつかいたかったわけである.

このデータを Fortran や C の一次元配列に格納するとする.[33] するといくつか方法があって, これらは普通等価であると見なされている.

A 三つ配列をとって,X,Y,Z 座標のそれぞれに割り当てる.

X_0	X_1	X_2	...
Y_0	Y_1	Y_2	...
Z_0	Z_1	Z_2	...

B 一つ配列をとって,X,Y,Z 座標に交互に割り当てる.

X_0	Y_0	Z_0	X_1	Y_1	Z_1	...

C 一つ配列をとって三分割し,X,Y,Z 座標のそれぞれに割り当てる.

X_0	X_1	X_2	...	Y_0	Y_1	Y_2	...	Z_0	Z_1	Z_2	...

で, 普通ワークステーションやメインフレームなどのプログラミングではこれらは等価であると見なされているので, この人は手法 B をそのまま CM-Fortran にひきうつして使用した結果, パフォーマンスが非常に悪くなった. そこで手法 A に変更したところ, パフォーマンスは大きく改善された.

これにはいくつかの複合した理由があると考えられる. このような配列について, それぞれの点で X,Y,Z の値の関数になるような値を求めようとするさい, 手法 B ではいくつかのデータが, 実プロセッサ間の分割の境界上をまたぐことになる. そこで低速な実プロセッサ間通信が必要になり, これが性能を悪化させる.

また実プロセッサ内でのデータ移動も, 手法 B のような一つの配列内の移動といったパターンでは比較的低速である.

---

[32] この例は堀口研究室の林さん:ryoko@jaist.ac.jp による.

[33] C で構造体に入れるのが良いという話はないことにする. それを言うと C*の方でも構造体に入れればいいことになって本題から外れるから.

これで分かるように, CM-Fortran で可能な最も高速なデータ移動は, 組になっている仮想プロセッサ間の, つまり, コンフォーマルな配列間の対応する場所の間の代入である. むろん演算もこれらの間のみで行なわれるものが一番効率が良い.

もし, 純粋にこのような演算だけで可能な処理があったとすれば, それは完全な並列化が可能な処理であるということであり, そのような問題はほとんどないので, 結局コンフォーマルな配列間の演算だけで記述できるようなアプリケーションはほとんどないということになる. しかしパフォーマンスのためにはこれを最大限に利用し, 最大の計算量を要求する部分がより完全に近くこのような配列間の演算だけで記述できるよう記述することが重要である.

1.5.2 演算子の数

今, 簡単のために, コンフォーマルな配列間の一つの演算・代入文しか無いようなプログラムを作って, そのパフォーマンスを測定してみる. アルファベットは全て配列名である.

$$(1)\ A = B * C$$

$$(2)\ R = X * X + Y * Y + Z * Z$$

実際に測定してみると,

- (1) A = B * C , 1 から 1.3 ギガフロップス
- (2) R = X * X + Y * Y + Z * Z , 4 ギガフロップス程度

になるはずである. つまり, 演算子の数が増加しても処理時間は比例して増加はせず, 複雑な計算の方が向いている.

これは多くのベクトル型スーパーコンピュータに見られる性質であり, CM5 のノードに装備されているベクトルユニットがその性質を受けついでいることを示している. 最終的には, 現在の CM5 のベクトルユニットは一度ベクトルレジスタに格納したデータ再利用できるかどうかには極めて敏感である. [34] このため, 連続した処理の中での演算子の数を増加させることがパフォーマンスには重要であり, このためにはより複雑な問題が適している.

先の例で大きく性能が向上したのは, 手法 A に (2) のような形の演算が含まれていたからである.

いくつかの式が連続している場合, これらのあいだの連続性を考慮し, ベクトルレジスタへのデータ移動を最適化する能力が CM-Fortran にはそなわっている. これは上の例のようないくつかの場合には非常に強力である. [35]

1.5.3 不規則演算

次に計算が全てのデータに対して均等に行なわれるのでなく, 配列の一部のみで演算が行なわれるとする.

実際に

```
forall (i=n:512,j=n:512) a(i,j) = a(i,j)+.....
```

や,

```
a(512:1024,512:1024) = a(512:1024,512:1024)*3.0
```

[34] ベクトルレジスタが小さく, それと主記憶とを結合する経路がロードもしくはストアが可能なパイプライン一本だけであるため.

[35] 乗算と加算のチェイニングも考慮されるので.

のように書いて, または where 構文で配列のある一部だけで演算を行なうことが可能である.

この時, 非常に重要なことは, 演算するべき部分の大きさが変動しても演算の所要時間が変化しないということである. これは実際に SIMD であるハードウェアと同様の性質である. [36]

そのため, 演算の効率を向上させるためには, 宣言した配列のうちなるべく多くの要素が利用されるようにすることである. しかしこれは問題によっては困難なことである.

図のような部分アクセスのパタンは, 例えば2次元配列に行列をそのまま代入してしまってその連立1次方程式を消去法で解こうとすると発生する. これはたとえば列方向に順に消去が行なわれ, 一度作業の済んだところはもう使用されないからである. そのルーチンは,CM-Fortran で,

```
C-------------------------------------------------------
      subroutine splitsolve1(tl,n,m,sp,ip)
      integer n,i,j,k,l,m,o,i1,i2
      integer sp,ip
      real tl(n,m),ol(n,m),sl(n,m),ql(n,m)
      real pv

CMF$ LAYOUT tl(:NEWS,:NEWS)
CMF$ ALIGN  ol(i,j) with tl(i,j)
CMF$ ALIGN  sl(i,j) with tl(i,j)

      sp = sp - 1
      pv = 0.0

      do 100 i=1,ip
      pv = tl(i+sp,i)

      forall (j=1:n,k=1:m) ql(j,k) = tl(j,i)
      forall (j=1:n,k=1:m) sl(j,k) = tl(i+sp,k)  / pv
      forall (k=1:m)       sl(i,k) = 0.0
      forall (j=1:n,k=1:m) tl(j,k) = tl(j,k) - sl(j,k) * ql(j,k)

  100 continue


      return
      end
C -------------------------------------------------------
```

のようになる. このルーチンだけでは, 例えば枢軸選択を行なわないので実際のアプリケーションの再現とは言えないが, 途中で配列全体のサイズ m を変更してこのルーチンを2回コールすること, つまり, 列消去

[36] 実際にはこれに加えて,「一部分だけに特定の操作をしようとするとそれだけでこの操作がいちじるしく低速になる」場合がある. なぜなら, はじめにこの「一部分」に対応する部分のみが真になっているベクトルマスクを作り, そののちにベクトル化された処理を行なうので, ベクトルマスクを作成する分だけ時間が余計にかかるのである. これを解決するためには, 無駄だが有害ではない (落ちたりしない) データを埋め草にして配列の全部を操作するようにコードを書き直す. もしくは同じベクトルマスクを再利用する, つまり一つの Where 構文のなかになるべく演算命令をつめこむ, といった対策が考えられる.

が半分まで進行して効率が半分以下になった時点でより小さい配列に乗りかえることによって, 行列のサイズ 1024 の時に 3 割程度の性能の向上が得られた.

つまり, 適切なサイズの配列と適切なデータの格納方法を考え, 効率が低すぎる場合には処理の途中で適応することが, CM-Fortran と C*のパフォーマンス向上には重要である.

1.6 失敗例の研究

この項での問題は, 主にプロセッサ間通信に関係している.

ここでとり上げるプログラムは, 簡単な物理学のシュミレーションを行うための物であり, ある本[37]に掲載されていた数式をほぼそのまま実装したものである, だが, 出力される結果は矛盾していたし, そもそもよく考えてみると, 並列計算を効率よく利用しようとして案出した実装方法が正しく物理学の法則を写像していたかどうかあやしくなってきたのであった. [38] それでもこのプログラムをここでとり上げるのは, その実行速度が計測の結果あまりにも低速であると判明したため, なにがその原因なのか分析することがより高速なプログラムの作成に役だつと考えたためである. まず, このプログラムの本質とは関係が無いが, 乱数のことについて述べておきたい. それからこのプログラム自体の問題について述べる.

1.6.1 乱数発生

CM-Fortran は, 一度に複数の乱数を並列に発生させることができる. もちろん, それらをある配列に代入するようになっているのが前提である. CMFRANDMIZE というのが初期化で, CMFRANDOM というのが実際の配列への代入を行なう. ちなみにこれらのルーチンで使用されるアルゴリズムは「ウォルフラムの 30x30 オートマタ」と呼ばれるものである. ここでは乱数の正規性の検証などは一切行なわなかったため, 一応使用アルゴリズムの名前だけはマニュアルから転記しておく, これらのルーチンには欠点が二つあった.

- 低速である.(何と比較して低速かという問題があるが.)
- これがやっている処理の内容と所要時間を, Prism が把握できない.

低速だと感じたのは, 実際は後者のためにプログラムが低速な理由が全てここにあると誤解していたからである. 客観的な評価基準が用意できるなら, それほどでもないのかもしれない. この時点で CMSSL に乱数発生ルーチンがあるのは分かっていたが, CMSSL は全部後者の欠点を持っている. だから, このプログラムでは, 丁度うまい具合に雑誌[39]に乱数の記事が出ていたので, それに掲載されていた方式をそのまま並列演算するようにさしかえてしまった. これは合同法を実数演算で行うものだったので, その処理の内容と所要時間を, Prism で正確に割り出せるようになった.

しかしこの後で重大なことが分かった. 並列乱数発生に限っては, どこも書きかえなくともコンパイル時に CMSSL をリンクするだけで,CMFRANDOM の中身がオーバーライドされるのである. 試みに次のようなプログラムを作成してみた.

```
      DIMENSION R(1024,1024)

CMF$  LAYOUT R(:NEWS,:NEWS)

      INCLUDE '/usr/include/cm/CMF_defs.h'

      CALL CMF_RANDOMIZE(1092)
      do 100 i=1,16
```

[37] 日本物理学学会編「計算機物理学」

[38] 何があやしいかというと, マルコフ過程が熱平衡を再現するまで過程をたどっていってから平均値をとるというのが問題の本質だったら, たくさんの過程を並行してたどっていってもそれぞれが熱平衡になるまでのステップ数はちっとも変わらないんじゃないかという点に, プログラムが出来てから気づいたのだった. 並列計算機の応用に際してはこのようなことも問題になりうるということを指摘しておく.

[39] 「Bit」93 年 4 月号,「乱数生成系で良質のものはほとんどない,1」

```
      LDSEED = LTSEED - RANDM * INT (LTSEED / RANDM)
      DV = LDSEED / RANDM
      DV = DV - 0.5
      NX(:,I2) = X(:,I2) + DV*SC2

      FORALL(I=1:MPY,J=1:TROT) DX(I,J)=(NX(I,MOD(J+1,TROT)+1)-NX(I,J))**2
      FORALL(I=1:MPY,J=1:TROT)DP(I,J)=NX(I,J)**2
      DSUM  = SUM(DX,DIM=2)
      DPSUM = SUM(DP,DIM=2)
      W1 = exp(CONST2*DSUM+CONST3*DPSUM - W0)

      LTSEED = RANDA * LDSEED
      LDSEED = LTSEED - RANDM * INT (LTSEED / RANDM)
      RANDSH = LDSEED / RANDM

      METROB = W1.GT.RANDSH
      BX2 = BX2 + DBLE(COUNT(METROB))

      WHERE(METROB)
      X(:,I2) = NX(:,I2)
      ENDWHERE

1001  CONTINUE

      SU1 = SU1 + AU1 / DBLE (TROT)
      SU2 = SU2 + AU2 / DBLE (TROT)
      SUC1 = SUC1 + AUC1 / DBLE (TROT)
      SUC2 = SUC2 + AUC2 / DBLE (TROT)

1000  CONTINUE
```

この後に出力部があるのだが, 省略する. このままの条件で,-g,-cmprofile オプションをつけてコンパイルし, PRISM で状況を見ると, 最も計算時間のかかっているのは, 最初の FORALL 文であることが分かる. しかし, 似たようなコストのものがいくつかあるし, 他のものとの差も 10 倍までは行っていないので, この文一つを集中して改善するのが得策では無い. 全体が一様に低速と言える.

次に, ベクトル長の問題と同時にオプティマイズの効果を検証してみる.

- -vu オプションをつけない.-O オプション無し. 6.0 秒
- -vu オプションをつけない.-O オプション有り. 4.9 秒
- -vu オプションをつける. -O オプション無し. 6.0 秒
- -vu オプションをつける. -O オプション有り. 4.9 秒

となり, ほとんどベクトル化の効果が出ていない. [40] これから, ベクトル長が問題の一つと考えられるから,

[40] CM-Fortran には, 配列の設定に関するもの以外のコンパイラディレクティブは無い. ということは, ベクトル型スーパーコンピュータのコンパイラにあるような特定のステートメントだけベクトル化の対象から除外させるようなディレクティブも無い.

```
      REAL MU1,MU2,MUC

      REAL SUBX1(MPY,TROT),SUBX2(MPY,TROT)
      REAL BX1,BX2
      REAL LDSEED(MPY),LTSEED(MPY)
      REAL PRVVAR,PRVTMP,TIME,VCVU

CMF$  LAYOUT X(:NEWS,:NEWS)
CMF$  ALIGN DX(I,J)      WITH X(I,J)
CMF$  ALIGN DP(I,J)      WITH X(I,J)
CMF$  ALIGN NX(I,J)      WITH X(I,J)
      PRVTMP = 0.0
      PRVVAR = 0.0

      CALL CMF_RANDOMIZE(80107)
      CALL CMF_RANDOM(LDSEED(:),1)

      LDSEED = INT ( LDSEED * RANDM )
```

変数の初期設定部は長いので省略する. 内側ループでどのような処理を行なっているかだけ分れば十分である. 非並列の二重ループの内側に並列化される部分を囲っているが, 外側のはランダムウォークの回数に対応し, 内側のは並列部を条件を変えながら TROT 回行うことに対応する. この条件というのがループの制御変数によるので, これを変更することができない.

```
      DO 1000 I1=1,ITAE

      AU1  = 0.0
      AU2  = 0.0
      AUC1 = 0.0
      AUC2 = 0.0

      DO 1001 I2=1,TROT

      IF (I1.GT.ITAE2) THEN
      U1  = const4 - DSUM /BETA
      U2  = const5 * DPSUM
      UC2 = U1 + U2
      UC1 = UC2**2 + CONST6*U1 + CONST7
      AU1  = AU1 + U1
      AU2  = AU2 + U2
      AUC1 = AUC1 + UC1
      AUC2 = AUC2 + UC2
      ENDIF

      LTSEED = RANDA * LDSEED
```

```
      CALL CMF_RANDOM(R,1)
      write (6,*) R(12,234)
 100  continue

      end
```

この実行時間をシステム時間による簡便な方法で測定すると,

- -vu オプションをつけない.10.9 秒
- -vu オプションをつける.14.2 秒
- -vu オプションと,-lcmsslcm5vu をつける.0.35 秒

となり, この方法によって圧倒的に高速化することと, この方法を使用しないとき, ベクトルユニット無しの方が高速であるという実例であることが分かった. なお,CMSSL ルーチンの乱数発生アルゴリズムは, フィボナッチ法の一種である.

1.6.2 配列の大きさ

このプログラムの速度評価のために加減乗除の回数を算出した. これには, 条件判断, 整数化, 指数関数などの計算量を素直に評価しがたい要素が含まれていたし, 加減乗除算の計算だけで正確な評価が可能なものでもないはずだが, これによって得られた値は,SparcStation2 の 64 台分よりも十分に低速なものだった. 大々的にアルゴリズムを変更するようなこと無しに, これをもう少し改良する方法を考慮する. そのため, 計算の内容はおいておくとして, 以下ではこのプログラムの総システム実行時間を UNIX の TIME コマンドによって評価することにする.

プログラム

```
      INCLUDE '/usr/include/cm/CMF_defs.h'
      INTEGER MPY,ITAE,ITAE2,TROT
C      PARAMETER ( MPY = 512 , ITAE = 36000  ,ITAE2 = 12000 , TROT= 32 )
      PARAMETER ( MPY = 512, ITAE = 160 ,ITAE2 = 80 , TROT= 8 )
      REAL RANDA,RANDM
      PARAMETER ( RANDA = 16807.0 , RANDM = 214783647.0 )
      REAL    MASS    , PIE   , BETA   , M  , T, K  , HB
      REAL    CONST1,CONST2,CONST3,CONST4,CONST5,CONST6,CONST7
      REAL    SC,SC2

      REAL  X(MPY,TROT),KE(MPY,TROT)
      REAL DX(MPY,TROT), DP(MPY,TROT)
      REAL DSUM(MPY), DPSUM(MPY)
      REAL W0(MPY), W1(MPY)
      REAL DV(MPY),NX(MPY,TROT),NR(MPY),RANDSH(MPY)
      LOGICAL METROB(MPY)

      REAL U1(MPY),U2(MPY),UC1(MPY),UC2(MPY)
      REAL AU1(MPY),AU2(MPY),AUC1(MPY),AUC2(MPY)
      REAL SU1(MPY),SU2(MPY),SUC1(MPY),SUC2(MPY)
```

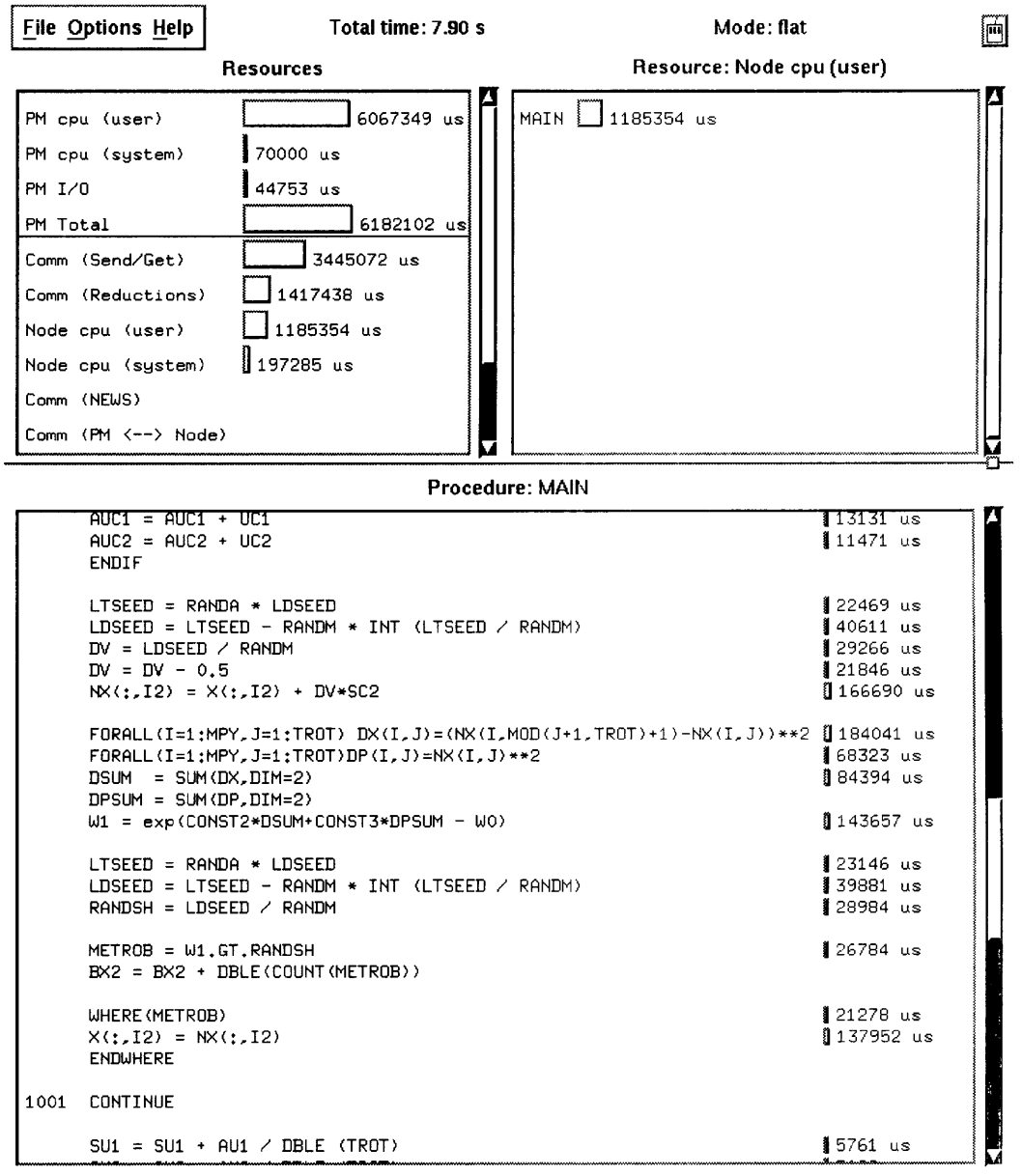


図 1.6: PRISM で処理の内容とステートメントごとの分配を見たところ. 適当なオプションをつけてコンパイルしたオブジェクトをロードし, collection スイッチを入れて一回最後まで走らせると,Performance メニューからこのウインドウを出せる. 最も時間を消費しているのは (Send/Get) タイプのプロセッサ間通信であることが, 左上の領域に表示されている. 下の領域には中枢部での所要時間が表示されているが, 一つだけ極端に突出しているというのは無い. なお右上の領域には各処理の所要時間がサブルーチンごとに表示されるので最も平均速度に影響しているサブルーチンを発見するのに役立つ.

それを単純に大にしてみる. 具体的には, あつかう配列を大にすることとし, このプログラムでは, 並列部での殆どのステートメントの計算量が MPY に比例することから, MPY の大きさを変動パラメータとする. コンパイルオプションはベクトルユニット使用/未使用の他は何もつけないことにする. 時間の数値は数回のうちの平均値である.

	Sparc チップのみ	ベクトルユニット使用
MPY = 64	3.6 秒	5.4 秒
MPY = 128	3.9 秒	5.8 秒
MPY = 256	4.5 秒	5.8 秒
MPY = 512	6.0 秒	5.6 秒
MPY = 1024	9.1 秒	6.3 秒
MPY = 2048	15.8 秒	7.3 秒
MPY = 4096	30.9 秒	10.3 秒

これにはもちろん初期設定と出力の時間も含まれている. どれだけ信頼できるかは分からないが, この影響を残くために外側メインループ繰り返し数を 1 にしてみた. この場合の TIME コマンドの結果は 0.1 秒である. パイプライン型の計算機の特徴が良くあらわれて, ある点を超えると計算量が多いほど計算速度は高速になっているのが分かる. ベクトル長を 64 倍にしても計算時間は二倍程度にしかなっていない.

ところで, パイプラインを使用しない場合でも MPY が大きくなると, ある点までは高速化効果がある. これはどういうことだろうか? PRISM をつかってしらべたところ, 実際に処理の数の増加に比して最も時間の延び率が小だったのは Comm Send/Get だが, 他の処理も速度としてはわずかに向上を示していた. これはどう解釈して良いのか分からない. [41] いずれにせよ, 多くの場合で,CM5 と CM-Fortran はより大きな問題の方が得意なようである. TROT 側の軸を拡大するのも, やってみるとたしかに有効のようであった. [42]

ただし, これでアルゴリズムの組み換えによって問題の大きさまたは計算量を変えずにベクトル長拡大する時のための指針を与えられたわけではないし, [43] ベクトル長がどうしても延長できないような問題が対象かもしれない. [44] 絶対的にどの程度の速度で計算しているかもまた別問題である.

また, 浮動小数点変数を全て倍精度にしてもテストしてみた, 多くの在来型スーパーコンピュータ, 例えば jaist の VP1100 では 64bit 演算が基本なので, これは有効な高速化手段である. しかし CM5 では, 系統的に調査したとはいいがたいのだが, 殆ど変化しないか, 二分の一程度までに速度が低下した. [45] このプログラムの場合は, あまり変化がなかった. 結果の数字は変わったが, それはむしろ当然だろう.

1.6.3 配列の最適化

そこで,MPY は 512,TROT は 8 のままにして, 他の方法をためしてみる. LAYOUT ディレクティブを使用して配列のセッティングを変更するのである. プログラムリストから主要部では第二軸方向の通信しか行なっていないことがわかるし, 図 5 のように PRISM で主要な通信の種類が (Send/Get) と表示されるので, これからどのように変更すれば良いか判断する. コンパイラオプションは何もつけない.

[41] アセンブラのソースを調べればなにか分かるかも. しかしそのためにはルータや PE のアーキテクチャの細部までソフト, ハードの知識が要求されるだろう.

[42] このとき計算速度を算出するのに使用したルーチンがあとで別なテストで変な値を返したことがあったので, 結果を廃棄. ただし配列が大なほど速くなっていたのはまちがいが無いと思う.

[43] ベクトル長は十分長いものとして, その場合にどういう計算のさせかたが有利が分かったわけでは無い.

[44] そのような規則性の低い問題のために CMMD があるのかもしれない. これと CDPEAC などを組み合わせれば場合によって非常に高い性能が獲得できるが, この場合には単一のノード, 単一のベクトルユニットの内部が不規則演算に弱いという問題がある.

[45] 間に種々のソフトウェア階層がはさまっているのでこういうことも起こりうるとは思うが,CM5 のベクトルユニットはハードウェアレベルでは殆どの演算で倍精度単精度の速度差は無いと思われる.

LAYOUT 文の指定	所要時間
(:NEWS,:NEWS)	6.0 秒
(1:NEWS,10:NEWS)	3.1 秒
(1:NEWS,20:NEWS)	3.1 秒
(:NEWS,:SEND)	5.8 秒
(1:NEWS,10:SEND)	2.9 秒
(1:NEWS,20:SEND)	2.9 秒
(:NEWS,:SERIAL)	2.9 秒

となって著しく高速化されたことが分かる. なお, 結果は前述のように物理的意味が曖昧で全過程に対する統計量のみが出力されるのだが, どのモードでも全く同じ値が出力された.

そこで, (:NEWS,:NEWS),(1:NEWS,10:NEWS),(1:NEWS,10:SEND),(:NEWS,:SERIAL) の各場合について, ベクトル長も MPY によって拡大して計測を行なってみる. 配列のとり方によるベクトル化への影響が見られると予想されるが, ここで最良の方法でも他の問題の場合にはそうではないかもしれない.

LAYOUT 文の指定	MPY = 512	MPY = 1024	MPY = 2048
(:NEWS,:NEWS)	6.0 sec	9.1 sec	15.8 sec
(1:NEWS,10:NEWS)	3.1 sec	4.5 sec	7.2 sec
(1:NEWS,10:SEND)	2.9 sec	4.1 sec	7.1 sec
(:NEWS,:SERIAL)	2.9 sec	4.1 sec	7.1 sec

ベクトルユニットを使用した場合は,

LAYOUT 文の指定	MPY = 512	MPY = 1024	MPY = 2048
(:NEWS,:NEWS)	5.6 sec	6.3 sec	7.3 sec
(1:NEWS,10:NEWS)	5.5 sec	5.7 sec	5.8 sec
(1:NEWS,10:SEND)	5.4 sec	5.6 sec	5.8 sec
(:NEWS,:SERIAL)	5.7 sec	5.9 sec	5.7 sec

となった. このうち最も計算速度が高いと考えられる (1:NEWS,10:SEND) MPY = 2048 のケースについて -O オプションをつけてみたところ, 結果は変動なしで 4.6 秒で終了した. 最初の段階から見て 34 倍程度高速になったことになる. [46]

ところで, 計算とプロセッサ通信に処理がはっきり二分できるものなら, [47] プログラム全体の性能という見地から見て, 計算速度の面からは, いつも瞬間的最大速度に, プロセッサ間通信の分だけマイナスのゲタをはいた値が出てくることになる.

そして, PRISM によれば各ステートメントごとの実行に必要だった時間を知ることができる. これからプログラム中の各ステートメント毎にそれを処理していた時の平均速度を出すことができる. いくつかのプログラムについて, このステートメント毎の速度を算出してみた.

たとえば先の行列計算のプログラムでは, 中心になっている FORALL に限定した Mflops 値は,

[46] もともと欠点が多すぎたと言えばそれまでだが.

他に書く場所が無いような漠然としたことなのでここに書くが, 同じオブジェクトを同じデータで何回か動作させた時の所要時間のバラツキが, ベクトルユニットを使用した場合に非常に大きいような気がする. このリポートのなかの速度実測値には, 書く時になって再度実験すると一割ぐらいのずれが出てくるのがあったが, 測定がいいかげんだっただけでもないのかもしれない.

[47] 前述の通り, できると思われる. Lin 以下ミネソタの人々が書いたレポートを見よ.

行列の次数	中枢部の使用時間	中枢部の実行速度
n=128	0.005Sec	838.1Mflops
n=256	0.032Sec	1048.5Mflops
n=512	0.565Sec	475.1Mflops
n=1024	2.135Sec	1005.8Mflops
n=2048	17.080Sec	1005.8Mflops
n=4096	137.188Sec	1001.8Mflops

となっていた. つまり, この場合は, 行列の次数を上げると速くなるように見えたのは, 単にFORALL以外のゲタの部分が変動していた結果だということになる. [48]

さらに, これらのステートメント毎速度の最高値はいままでの所,2000Mflops強である. それは最終的に前述の一割のオプティマイズに成功した行列乗算プログラムの中枢部でのデータであるが, あれもそうなった理由が分からないからこれ以上高速にできるとしてその方法論が分からないし, 現在まで2000Mflopsを大きく超える結果は現に出せていない. [49] つまり, どんな場合でもそれ以下の速度しか出ないことになる. [50]

何度も言うようにFortranについて全ての場合を試験したのではないからもっとうまいやり方があるのかもしれないが, さしあたって自前でCM-Fortranを使用して理論性能に匹敵する計算パワーを獲得する方法は無い.

しかしそれでも充分速いので, より重要と思われるポイントを指摘しておく, これはCM5のハード, ソフトの内部深くまで調査してえられた中で多分最も重要なポイントであり,CM-Fortran利用者はその結果に対する考察の際に常に留意していただきたいことである.

- CM5上のアプリケーションの性能が予想よりも低い場合, 並列計算機だからそうなっているとは言えない場合もある.

なぜなら,footnoteなどで触れているように, 速度の低下要因が各ノードプロセッサ内の出来事で全部説明できてしまったりするからである. この場合, 明らかに並列化効率と実際の性能とは何の関係もない. より複雑な問題においてもノードプロセッサ内とノードプロセッサの相互作用とのどちらがどれだけパフォーマンスに影響するかは複雑な問題であり, prismなどで得られたデータをよく吟味して考察しなければ意味が無い.

次は高速化を期待できるCMSSLのテスト結果を報告するが, その前にCM-Fortranの別な面での有用性について考察する.

---

[48] プロセッサ間通信の実行パターンをコードから予想して所要時間のモデルを立てることができない. ただこの場合は, たとえば何らかの形で演算と通信が並行して行なわれると仮定し, また計算の量はn^3に比例するわけだから, プロセッサ間通信の方が, 対象が2次元配列なんだからn^2に比例していたのだ, とでもすれば, つじつまが合う. しかしハードの構造を考えると, プロセッサ間通信時間の主要項はn^1になっていないとおかしい. だからこういうことをきちんと議論したい人はCMMDを使用するべきである.

[49] その後, 前章で述べたようにある数式の計算で4000Mflops程度まで可能なことが分った. しかしいつでもこれが可能とは思えない.

[50] この速度は同時に加算と乗算を行なう命令に翻訳されそうなステートメントが記録したものである. これは三項演算なので, 三組の変数をベクトルレジスタにロードしなければならず, 単独で行なった時の実行をアセンブラのマニュアルから予想すると, どう速くとも2000Mflops以上は出ないことになり, この実験での低速な結果はむしろ自然である. これ以上の高速性を獲得するには, ステートメントを増加してベクトル長を増加し, またベクトルレジスタに一度ロードしたデータを放さないようにする必要がある.

1.7 移植

機械語でなく高級言語を使用するのは, プログラムが書きやすくなるのが最大の理由だろうが, ハードウェアの差異を吸収するという効果もある. たとえば科学技術用計算に Fortran が広汎に使用されるのは, そういう計算でパフォーマンスを向上させるのに有利だからでもあるが, 既に存在する, 場合によっては全くことなった構造の計算機で開発されたプログラムが, 新規にプログラムを開発することなく利用可能となる環境が Fortran の枠組みによって可能になったからである.

現在超並列計算機はどういった形態が主流とも言えない状況にある. お互いに全く構造が異なっているのが当然と思われているから, 並列機同士でプログラムを融通しあうような話は実際のところきいたことが無い. [51]

今回, あるところから,TMC の製品でない,CM5 と全く関係の無い別の超並列計算機用に開発された,Fortran 90 のソースコードを一つ提供していただいた. 内容はある種の物理系シュミレーションである.

このプログラムはまだ CM5 への移植作業が終了していないし, 実を言えば完全に理解しているわけでもない. 研究目的に限定するということでいただいてきたので, あまり詳細なことはここでは述べないことにするが, [52]

これを対象にして, あまりにも時期尚早で的外れという感じもするのだが, Fortran 90 規格を基礎とした並列機同士の互換性について検討してみたい.

結論から言うと, いちおう動作するようにするのは非常に簡単だった. CM-Fortran で翻訳しかねるようなものは, 二千行以上の中に五種しかなかった. これらは, 並列乱数初期化, 並列乱数発生, 計算時間評価用のタイマー開始, タイマー終了のサブルーチンで, まあ普通どの計算機環境でも共通してないことの方が多いものなので, すぐに CM-Fortran での対応物に移しかえることができた. 乱数の並列化など考え方がまるで一緒である.

最後の一つはある書法で, 注意深く見ると, CM-Fortran のある文とまったく同じことをやっているのだということが分かった. プロトタイプとして作ったらしい完全に逐次型 Fortran 77 のコードもいただいていたので, これと比較してこの考えが正しいことを確認することができた. 先にシリアルのプログラムを完全に作っておいてパラレルに書き直し, 成功したら問題規模を拡大するというのは大変うまいやり方だと思われる. [53] この時点で, プログラムがどんな処理をするのかといった全体的な知識もないままで, コンパイルには成功した. その後現在にいたるまでシュミレーションとして有用なように動作したことはまだないが, それは内部で, 物理学的に異常な状況に入りこんでいることを警告する目的らしいルーチンにひっかかってしまうからで, 移植は一応できたことになるのかもしれない. [54]

さて, これがアーキテクチャの差をどれだけ埋めるかという問題だが, 配列に対する多数の操作を一命令で書けるようにし, プロセッサ群への配列のマッピングが具体的な並列計算のやり方を与えてくれるという考え方は両者で同じである. 両者ともアーキテクチャにスケーラビリティを持っているため, 絶対的な性能は評価のしようが無いが, 開発元の機械とここの CM5 を比較するなら, 最大理論演算性能のちがいは 2 けたにならないから, 似たようなものと言って良いと思う.

ただ, 移植元の機械には仮想プロセッサの機能が無い. プロセッサ数はソースにハードコードである. [55]

---

[51] どこまで移植性が確保されているのか知れないが,PVM のように他機種で一応動作する共通環境は存在する. また, 最近の情報によると, メッセージパッシング型並列計算環境のみではそのメッセージ型式を統一できる標準仕様 MPI が策定中である.

[52] あとで許可が出たら, コードと移植元の機械についても細かく書くことにしよう.

[53] ただし, 配列を一まとめにしてあつかえる書法がたくさんあるので, 慣れればいわゆるプログラム生産性は CM-Fortran の方がもしかすると高い.

[54] これをやっている途中で見つけたことなのだが,CONVEX C3430 の FORTRAN コンパイラ, FC は,Fortran 90 をコンパイルできるらしい. ただ, 配列操作のイントリシック関数の一部か, 全てが装備されていない. しかし, これらを用意できれば, CONVEX C3430 のピーク性能と CM5 での効率が低いときの性能には膨大な差は無いし, 主記憶容量だって 1/4 もある.CONVEX をバックアップに回させることで CM5 の孤立性を低めることもあながち不可能では無いわけだ.

[55] 実際に存在している PE の数より多く使おうとするとコンパイル時にエラーになって, 少く使おうとすると, 実際にそれだけしか

そのプロセッサ群は一つ一つの記憶容量も処理能力も比較的小であるかわり, 数が非常に大である.jaist に導入されているどの並列機よりも処理細粒性が高いかもしれない. プロセッサ間接続/通信のトポロジーは,2 次元メッシュ+ α といったようなものである. だからと言って 2 次元以上の配列が使用できないものでも無いが, CM5 の代入命令のうち特殊性の高いものは実装できないのかもしれない. [56] 普通の配列宣言のあと,CM-Fortran のによく似たコンパイラディレクティブによって, 配列の軸をプロセッサ間にばらまくかローカルメモリに積むかを指定する. コンフォーマリティはこれによって自動的に決まってしまう.

CM5 は仮想プロセッサ機能により PE 数には影響されないし, 通信のトポロジーの冗長性や柔軟度が高いので, こうした異システムのやり方をとりこんでしまうことができ, かんたんに動作することができた. 逆に CM5 から他のものに移行させるのは大変だろう. ただし, 厳密に評価すれば, 行列乗算の項で述べたような規則正しい転送パターンの場合には, シンプルなメッシュの方が優越しているのかもしれない.

まとめると, 性能効率を追求するのには問題があるが, Fortran 90 などを使用して超並列機のプログラムを互換するのは, かなり期待のもてる手法であると言えるだろう. [57]

使用しないようなオブジェクトができて, その数と演算性能から並列化効率を求められるようなシステムではないか?と憶測している.

[56] 利用できそうなものを利用しないで他の手段の組み合わせによって実現している個所がいくつかあった. こういうのは CM5 風の書法に変更したバージョンを作成中だが, どうも作業中に変なところをいじって重大なバグをかかえこんでしまったらしい.

[57] 互換性に関して別の重大な問題を指摘しておく, それは CM5 と CM2/CM200 の互換性である. これらは CMSSL での一部の機能が片方に無かったりする以外は高い互換性があるし,jaist には実際 CM2 は無いのだから関係が無さそうだが, 両者で, アーキテクチャのせいなどでパフォーマンス最適化のセオリーが異なる部分があり, ここでの CM5 利用者にとっても身近な問題として, マニュアルのかなりの部分に CM2 を対象とした記述が混じっているのである. もちろん大概の場合 CM2 でのことであると書いてあるのだが, あやしい場合はそれが TMC のどの機械について述べているのか照合した方が良い. 93 年の夏秋以降の新しいマニュアルには表紙に CM5 と明記してあり, こうした問題は無い. また, 未確認だが,SuperSparc を採用し, 約 25 互換性は高いものと考えられる.

1.8 CMSSL ルーチンのテスト

CMSSL のルーチンのうち基本的でテストが簡単なものを選択して計算速度を計測した. 例えば初期データの性質に問題があるなどの理由で実用時にはより低下すると考えられるものもあるが, それでも一応の参考にはなると思われる. プログラムは, いずれも CMSSL のサンプルとしてインストールされていたものを小改造したものなので, ここにはあげない.

CMSSL の使用にはサブルーチンを利用してプログラムを書き, ライブラリをリンクする. 一部には操作簡略かの為にマクロ変数を定義したヘッダをインクルードするのもある. ライブラリにはベクトルユニットを使用するのとしないのがあるので注意する, CMSSL とベクトルユニットを両方使用する場合, またよりメモリ消費が少なくて済むベクトルユニット無しの場合それぞれ,

```
cmf *****.fcm -vu -lcmsslcm5vu

cmf *****.fcm     -lcmsslcm5
```

のようにライブラリをリンクする.

1.8.1 行列積算

CM-Fortran によるコーディングとの比較のために, まず行列乗算を利用してみた. 単精度のみの実験だが,

行列の大きさ	計算速度	備考
n=256	475.4Mflops	
n=512	1.014Gflops	
n=1024	2.147Gflops	
n=2048	3.887Gflops	
n=4096	7.815Gflops	(これはいくらなんでも速すぎると思う. 時間の測定ミスか?
n=8192	7.484Gflops	

という値がえられ, 十分に高速であることが分かった. 以下の行列対角化は計算量が不明なため, 高速フーリエ変換はプロセッサ間通信効率による影響が高くて比較対象が無いため, CM5 の高性能の直接的な例は今回このデータが唯一である.

1.8.2 行列対角化

対角化の計算量は行列の性質によって変動し確定的な値があるわけではないし, 大きなほうからの数個の固有値が分かれば良いといった特殊な要求もあるので具体的な問題からえられた行列を使用するのが望ましいのだが, 時間が無いので乱数行列を使用した. 最良でこの程度という目安と, 現状でとりあつかうことのできる最大の行列次数の見立てには有益だろう. ここでは最もあつかいが単純なルーチンに対して実行してみた.

他の場合にも何度かあったことだが, メモリの使用量は対角化の最中にも動的に増加しているようである. またこのテストは限界までたしかめたわけではないので,2304 次というのが上限では無い.2500 程度までは可能だと思う.

倍精度, 対称,0 1 擬似一様乱数, 密行列. 全対角化. アルゴリズムは不明. 結果はコード自体に検証ルーチンあり.

行列の大きさ	所要時間	備考
1500x1500	400sec	success.
2048x2048	832sec	success.
2304x2304	1264sec	success.
3000x3000	*	crash. (これは事故の可能性あり.CM5 自体が不調の時にテストしていた.)

数値演算ライブラリには一般的に言えるのかもしれないが, 作動する時に大量の作業領域を消費する.CMSSL の場合, これを開放するのにそのためのルーチンを実行してやらなければならないが, 今回はどのテストでもこれを使用しなかったためにこのような結果となった.

ただし, 逆に容量が許せば作業領域を維持したままで連続してを処理を行なうことができ, おそらくこの方が開放と再取得に必要な時間だけ高速である.

1.8.3 高速フーリエ変換

高速フーリエ変換は, 複素数対複素数のものしかない, ただし変換対象の次元は任意である. 今回は 3 次元でしかテストしてないが[58], 他の次元でも同じルーチンでできる.

対象のサイズ	所要時間	備考
64*64*64	1.45sec	(2 回の平均)
128*128*128	12.55sec	(2 回の平均)
128*128*128	19.55sec	(2 回の平均, これは倍精度)
256*256*256	*	CM メモリがパンク

これは配列を初期化して試験用の正弦波を代入し, FFT の高速化用構造体 (と仮に呼んでおく) を CM に作ってフーリエ変換し, それを逆フーリエ変換して残差をとる, というプログラムのシステム使用時間である. 変換するデータによって変動すると考えられるので, 残差は極めて小だとだけ述べておく. 連続に実行するのであれば一回はこの二分の一かそれ以下の時間でできるはずである.

パフォーマンス向上については CMSSL のディレクトリに入っているオンラインマニュアルなどに記載されている. 結果をビット反転したままで返させるなら速いとか, 配列に対するディレクティブのつけ方などである. これらについてはやってみたがあまり効果が上がらなかった. だがあまりよく読まずに簡単に済ましてしまったので, 詳しく調べれば, また, 変換対象の次元など場合によっては, これらの忠告が有効かもしれない.

限られたものだけに対する試験では,CMSSL ルーチンは非常に高効率である. これだけでも有用だし, たとえば行列に対する操作の場合, 直接実行したのではメモリが不足する操作でも問題を小行列に分割するなどの手段で処理が可能だろう. また,CM-Fortran Intrinsics と重複した処理を行うルーチンもあるので, これらを CM-Fortran プログラムの低速と感じた部分に差し換えるのも実験する価値があるだろう.

しかし, 組み合わせによってもともと CMSSL になかったアルゴリズムを作成するための部品にするのには, あたりまえなもの, たとえばベクトルのスカラー倍のようなのが欠けているので, ここに Fortran Intrinsics を使用することになり, 「失敗例の研究」で述べた壁にあたってしまう問題がある.

究極的に性能を向上するためには, 結局問題適用例に応じてそのたびに種々の手段を工夫するよりないようである.

[58] おそかった.

1.9 CMMD

(CMMD に関するこのセクションは佐藤研の河瀬君に書いていただいたものである.)

プロセッサエレメント (Processor Element, 以下ではプロセッサと略する) 間でやりとりされるデータのことをメッセージといい, この通信のことをメッセージパッシングという. メッセージパッシングを CM-5 上で使用するためのライブラリが CMMD である. この章では CMMD の使い方を実際のプログラムを用いて簡単に説明する. いくつかの言語で CMMD を呼び出すことができるが, ここでは,C 言語を使用し,C 言語で用いられる用語で統一する.

このライブラリを用いたプログラムは, Host/Node 型と Hostless 型に区別することができる. Host/Node 型のプログラムは, フロントエンドで実行するプログラムとプロセッサで実行するプログラムとを作成する必要がある. Hostless 型のプログラムは, プロセッサで実行するプログラムだけを作成すればよい.

1.9.1 基本的な関数

ここでは,1.9.2に示す題材で, Host/Node 型,Hostless 型の両プログラムで使用する関数 (プロセッサ間のメッセージパッシング, プロセッサ情報, 入出力) を中心に説明していく.

メッセージパッシング

次の 2 つの関数を用いて, メッセージのやりとりを行う.

```
int CMMD_send(int dest, int tag, void *buf, int len_buf)
```

```
    dest     送信先のプロセッサ ID
    tag      タグ, メッセージのラベル (正整数か CMMD_DEFAULT_TAG)
    *buf     送信するメッセージが入ったバッファへのポインタ
    len_buf  バッファの長さ (Byte)
     返値    0  長さ len_buf のメッセージを送信した時
             1  len_buf より短いメッセージを送信した時
```

```
 int CMMD_receive(int src, int tag, void *buf, int len_buf)
```

```
    src      送信元のプロセッサ ID
             送信元を指定しない時は CMMD_ANY_NODE
    tag      タグ, メッセージのラベル (正整数)
             タグを指定しない時は CMMD_ANY_TAG
    *buf     受信したメッセージを入れるバッファへのポインタ
    len_buf  バッファの長さ (Byte)
     返値    0  長さ len_buf のメッセージを受信した時
             1  それ以外
```

`CMMD_receive()` はメッセージを受けとるまでは, 次の処理へは進めない.

プロセッサ情報

プロセッサに関する情報を獲得する時は次の関数を用いる.

`int CMMD_partition_size()`	現在のパーティション内にあるプロセッサ数を返す.
`int CMMD_self_address()`	この関数を呼び出したプロセッサの ID を返す.

入出力

プロセッサの入出力に関する変更を行う時, 次の関数を用いる.

```
int CMMD_fset_io_mode(FILE *fp, CMMD_mode_t io_mode)
```

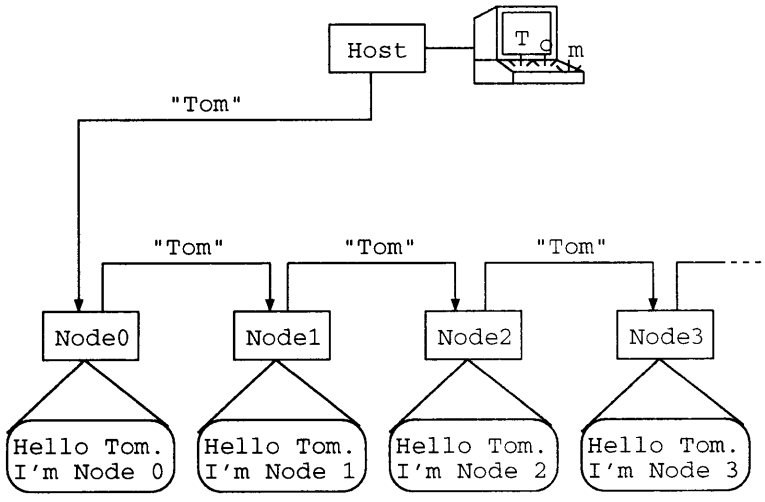


図 1.7: 設定した題材

```
*fp      ファイルポインタ
io_mode ファイルモード.
  CMMD_local          異なるファイルに各プロセッサが
                      独立にアクセスできる.
                      変更を行わない時はこのモード.
  CMMD_independent    一つのファイルに各プロセッサが
                      独立にアクセスできる.
  その他,CMMD_sync_bc, CMMD_sync_seq がある.
  しかし, この関数で指定できるモードは
  CMMD_independent, CMMD_sync_bc,
  CMMD_sync_seq の 3 つである.
```

返値は, 成功すれば 0, 失敗すれば−1 である.

1.9.2 簡単なプログラム

ここでは, 簡単な題材を用いて, CMMD のプログラムがどのような形をしているか説明する. 設定した題材は次のようなものである.

標準入力から文字列を読み込み, プロセッサ 0 へ文字列を送信する. 次にプロセッサ 0 は, 自分のプロセッサ ID とその文字列を標準出力に出力し, プロセッサ 1 へこの文字列を送信する. 同様のことをプロセッサ 1,2,3,... が順番に行う. 最後のプロセッサは標準出力への出力だけとする.

Hostless 型

まず,Hostless 型のプログラムを紹介する. 次のプログラムではプロセッサ 0 がホストの処理を兼ねている.

```
#include <stdio.h>
#include <cm/cmmd.h> /* CMMD で用いるマクロの読み込み */

#define NN 50
main(argc, argv)
int argc;
char *argv[];
{
  int pn_number;
  char name[NN];

  pn_number = CMMD_self_address();

  CMMD_fset_io_mode(stdout, CMMD_independent);
```

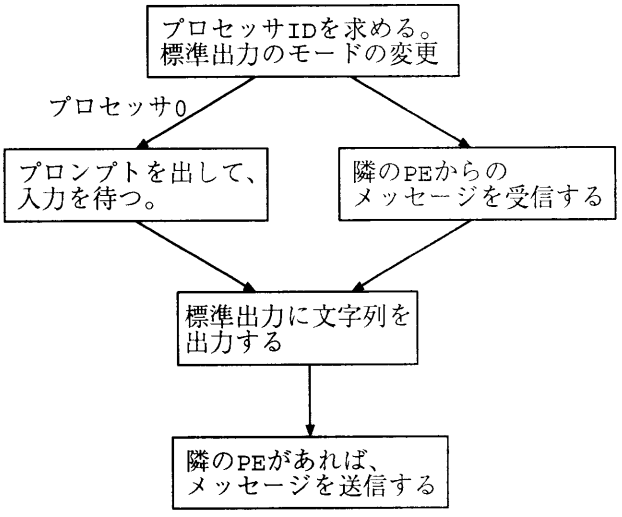


図 1.8: Hostless 型プログラムの処理

```
    if(pn_number == 0) fprintf(stdout, "Input>");
    fscanf(stdin, "%s", name);

    if(pn_number != 0)
      CMMD_receive(pn_number-1, CMMD_ANY_TAG,
                   name, NN*sizeof(char));

    fprintf(stdout, "Hello %s. I'm node %d\n", name, pn_number);

    if(pn_number != CMMD_partition_size()-1)
      CMMD_send(pn_number+1, CMMD_DEFAULT_TAG,
                name, NN*sizeof(char));
}
```

処理の流れは図1.8のようになっている. ID が0以外のプロセッサは, メッセージを受信するまで次の処理へ進めない. つまり, まずプロセッサ0が出力を終え, プロセッサ1へメッセージを送信する. メッセージの受信待ちをしていたプロセッサ1は, メッセージを受信することで次の処理へ進むことができる. 同様のことがすべてのプロセッサで起こる.

標準入力のモードは CMMD_local である. もし, 各プロセッサが異なるファイルポインタに対して CMMD_local モードであるときは, 各プロセッサは独立にアクセスを行うことが可能である. しかし, ひとつしか存在しない標準入力に対しては, プロセッサ0とアクセスを行う. 逆に, 標準出力を CMMD_independent モードにしておかないと, 全プロセッサの出力はできないことになる.

Host/Node 型

次に Host/Node 型のプログラムについて考える. まず, この Host/Node 型プログラム内で有効な関数を紹介する. まず, メッセージパッシングの初期化, Node Program のダウンロードをする. `CMMD_enable()` と `CMMD_enable_host()` を呼び出す必要がある. これらを呼び出す前で, メッセージパッシングを行うことは不可能である. 次に,Node Program で使用されている `CMMD_host_node()` はホストの ID を求める関数である.

Host Program

```
#include <stdio.h>
#include <cm/cmmd.h>
```

```
#define NN 50

main(argc, argv)
int argc;
char *argv[];
{
  int i, nprocs;
  char name[NN];

  CMMD_enable();

  nprocs = CMMD_partition_size();
  fprintf(stdout, "INPUT>");
  fscanf(stdin, "%s", name);

  CMMD_send(0, CMMD_DEFAULT_TAG, name, NN*sizeof(char));

  i = 0;
  while(i < nprocs){
    if(CMMD_poll_for_services()) i++;
  }
}
```

Node Program

```
#include <stdio.h>
#include <cm/cmmd.h>

#define NN 50

main(argc, argv)
int argc;
char *argv[];
{
  int pn_number;
  char name[NN];

  pn_number = CMMD_self_address();
  CMMD_fset_io_mode(stdout, CMMD_independent);

  CMMD_enable_host();

  if(pn_number == 0)
    CMMD_receive(CMMD_host_node(), CMMD_DEFAULT_TAG,
                 name, NN*sizeof(char));
  else
    CMMD_receive(pn_number-1, CMMD_DEFAULT_TAG,
                 name, NN*sizeof(char));

  fprintf(stdout, "Hello %s. I'm node %d\n", name, pn_number);

  if(pn_number != CMMD_partition_size()-1)
    CMMD_send(pn_number+1, CMMD_DEFAULT_TAG,
              name, NN*sizeof(char));
}
```

処理の流れは図1.9のようになっている.

しかしHostで行われる処理は, プロセッサ0へメッセージを送信した後にも続いている. この部分はプロセッサの出力に関する処理を行っている. ここで使われている `CMMD_poll_for_services()` は, 入出力のサービス要請があれば1, なければ0を返す. すなわち最後のwhile文は, 全プロセッサの出力が終るまで続けられる.

1.9.3 コンパイル

プログラムができたらコンパイルに通すのであるが, サンプルプログラム[59]についているMakefileを手直しすれば十分であろう.

[59] cm5-cp1:/usr/cm5/CMMD_3.2/examples/cmmd/{hostnode|hostless}/c にある.

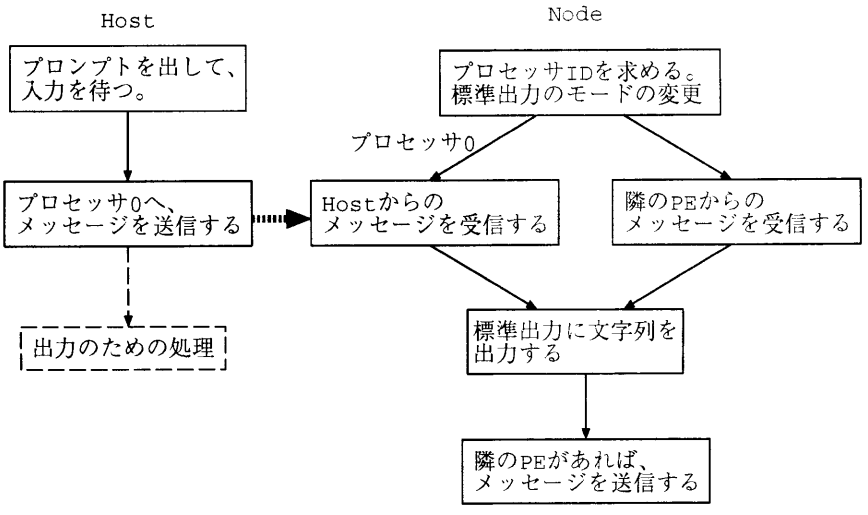


図 1.9: Host/Node 型プログラムの処理

1. Makefile をワーキングディレクトリにコピーする.
2. ソースファイル名の定義を変更する.
   Hostless 型ならば SRCS を,Host/Node 型ならば HOST_SRCS,NODE_SRCS を変更する.
3. コンパイル後の実行ファイル名の定義 (TARGET から始まる 1 行) を変更する.

以上が済めば, ただ make とタイプすればよい.

1.9.4 実行例

入力文を Tom とすると, 上のプログラムは次のように出力する.

```
cm5-cp2 % hello
Input> Tom
Hello Tom. I'm node 0
Hello Tom. I'm node 1
Hello Tom. I'm node 2

        <省略>

Hello Tom. I'm node 61
Hello Tom. I'm node 62
Hello Tom. I'm node 63
cm5-cp2 %
```

1.9.5 実行できない時

プログラムを作成し, コンパイルも通った. しかし, 必ずしも思い通り実行するとは限らない. 逐次型のプログラムを実行した時に, メッセージもプロンプトも返さない症状とはっきりと core をはくという症状とが現れる. CMMD を用いたプログラムでも, これらとよく似た症状が現れる.

まず, メッセージもプロンプトも返さない症状の場合, CMMD を用いたプログラムでは, メッセージの受渡しがうまくいっていないことがあげられる. つまり, メッセージは送信されたが, 受信側が何らかの原因で受信できないという状態が起こる.

フロントエンドで実行しているプログラムに矛盾が生じた時は core をはくが, プロセッサ側のプログラムで矛盾が起きた時は, CMTSD_errors.*pid*, CMTSD_heap.*pid*, CMTSD_stack.*pid* というファイルが作ら

れる (*pid* はプロセッサ ID). この症状が起こる時は, ほぼ逐次型のプログラムで core をはく時を考えてもらえばよい. しかし, その原因に CMMD の関数が関わっている場合もよくある.

1.9.6 最後に

今回, 紹介した関数は CMMD ライブラリ中の一部の関数を紹介しただけである. これらの他にも,Broadcast, 非同期通信,Reduction などの関数がある. これらのことは,CMview の CMMD Reference Manual で参照することができる.

1.10 その他の機能

その他の機能について, 概説する.

1.10.1 SDA

CMFS ライブラリによってディスクアレイへのアクセスができる. CMFS は, 無理に要約すると, 通常の UNIX が逐次的なファイルへのバイトを基本データとしたアクセスを提供しているのと同様に, ディスクアレイ上での特殊ファイルの配列を基本データとした並列アクセスを提供している. つまり比較的高速に, ある配列をまるごと write したり, その内容を別な配列にまるごと read したりできる. つまり配列を一度に書いたり, 読んだりといった形で並列処理が行なわれることは CM-Fortran, C*の他の機能と同様である.

これでディスクに書かれた内容は, SDA 自体が PM からは UNIX のディスクに見えるので, 外部から見れば UNIX のシーケンシャルなファイルとしてあつかえる.

問題は, この場合, 配列を一挙にあつかう以外の並列入出力機能が欠けていることである.

1.10.2 チェックポイント機能

これは「CM5 Technical Summary」の The Program Execution Environment の項に出ている. 好きな時に実行中のプログラムを停止させ, パーティションマネージャ, 並列部の全メモリイメージを二次記憶に移してしまい, それをもとにして好きな時に最起動できるような機能である. これは極めて大規模な計算をするためには絶対に不可欠な機能である. CM5 の次期オペレーティングシステムで実用化する見込みである.

1.10.3 CMX11

CMX11 のライブラリで作れば, X ウインドウのクライアントプログラムを CM5 の並列部で動作させられるらしい. 大規模なシュミレーションなども, 計算しながら結果をビジュアライゼーションできるということになる. 今でもインストールされているのだが, 使用していないのは時間が無かったのと Frontnet がパンクしそうな気がしたからである. また使ってみたいのはやまやまだが, もしも Xlib に準拠したライブラリが用意されているのみなら, それで高度なビジュアライゼーションのプログラムを組むのは非常に困難な仕事と言える. サンプルプログラムがあるので走らせて見た. 実際に描画する SparcStation2 が遅いだけかもしれないが, 画面の出力が非常に低速なような印象を受けた.

1.10.4 アセンブラ

究極的に高速化するためには理屈上アセンブラ以外の手段は無いだろう. 具体例は挙げられないが, 種々の CM5 に関する種々の報告や論文でも, 最もパフォーマンスに影響のある部分に限定して人力でアセンブラコーディングをしている模様である. 困難だが, アセンブラをどうにか使用できるようにしておくのは必要不可欠と考えたからそうした. いくつかの簡単なサンプルプログラムと修士研究のプログラムに CDPEAC を使用したが, よく考えて使えば非常に有効であると感じられた.

1.11 CM5 で何をするのか

本稿の大部分を書いた1993年6月には, あとがきに次のように書いた.

> 科学技術計算用スーパーコンピュータとして見た場合に,CM5 は極めて高速だし, 非常にユーザーフレンドリーでもある. 集中した研究が行なわれている分野なので近い将来そうでもなくなるかもしれないが, 超並列スーパーコンピュータの中でこれほど多様な, また完成度の高い環境を提供しているものは現時点では他には無いのではないだろうか. ただし今回の性能評価の数字を見ると, 場合によってはこの両者が共存できないと言える.
>
> ただし CMSSL に前面的に依存してもおそらく研究用の計算に使用するのには何の問題もないだろう, しかしこの場合には, 簡単に記憶領域の半分やそこらを使用してしまうことが問題である. 問題の大きさに制限がある.
>
> 記憶領域は,SDA にデータを一時退避したりしてそのプログラム自体ではどうにかなるだろうが, プロセス間に並列部の記憶域を分割して使用するようになっているので, もし他のプログラム, たとえば,LISP のように記憶域を動的に使用するようなプログラムと同時に走らせれば, どちらか落ちてしまうか, 双方ともに能力をフルに利用できないようなことになってしまうだろう. これはチェックポイント機能で巨大プログラムの方を場合に応じて一時停止, 再開するようにして, インタラクティブで動的なプログラムとすみわければ良いはずである.
>
> 結局, 巨大な計算能力があるのだから, それに見合った巨大な計算をする必要がある. しかしその実現のためには, 現状の知識では, 記憶容量, 使用時間などに種々の制限が付いてしまう. プログラミングも場合によっては非常に困難なので, 問題を機械が実際に解くための時間とそのためにユーザーが働く時間とのトレードオフさえ問題になりうる. 問題意識のもちようによっては, 同じ問題のためには CM5 よりも普通のワークステーションを数か月にわたって連続使用した方が良いということも考えられるのである. これからもどのような問題を解かせたら有意義かという問題をサイエンスの見方から検討すると同時に, それを現実に実行するための問題を一つ一つ解決していく必要があるだろう.

それから一年以上経過した現在, 超並列スーパーコンピュータは機種も増加しとくに商業利用の分野においてより実用に近いものとなった. 科学技術演算の分野でも, この一年間に登場した新機種の多くがその能力において CM5 をしのいでいる.

しかし, そのソフトウェア技術においては, いまだに CM5 の環境を大きく凌駕するものは見当らない. 超並列向けプログラミング技術はいまだ未開拓の分野である.

我々もこの一年間のあいだにいくつかの問題点を解決し, 多くの応用を CM5 上に実現してきた. それらは並列計算の基礎技術の試行から人工知能, データベース, 流体力学, 分子動力学までにおよんでおり, これらの研究の過程によってその有効性を実感することができた.

CM5 についてはこれからも今までの研究の延長, 問題点の解決などやるべきことは山積しているし, 特定用途, つまり非常に大量の規則的演算用の大規模サーバとしての利用も有用であろう.

最後に, コネクションマシンに関するマニュアル以外の参考になりそうな文献を挙げておく.

- われわれが書いた修士論文その他
  いく人かの人は CM5 利用の具体的な面についてかなり詳細に記述しているはずである.

- 「Science on the ConnectionMachine」
  「Proceeding of the Conference on Scientific Application of the ConnectionMachine」
  これらは図書館にある.CM5 以前の機種に関する記述が多いが,CM-Fortran のソースコードがついている論文もふくまれているため, 非常に有用である.

- bit 誌 94 年 7 月号「CM5 式ピザの作り方」
  これを書いた Guy L.Steele Jr. 博士は ConnectionMachine の言語開発に深くかかわっている. 本論文の内容の大部分はキッチンにおけるピザの作り方についてであるが, のこりの部分には CM5 の並列処理と結合ネットワークのデザインにおける基本的な考え方が述べられている.

- 「コネクションマシン」パーソナルメディア
  W.Daniel Hillis が最初のコネクションマシン CM1 について述べている. CM5 はそれ以前のモデルと構造的に全く似ていないのでなにも直接には参考にはならないが, そもそもどのような考えからこの超並列マシンが生まれてきたかという解説は興味深いものであろう. 同様の理由で,
  「さようならファインマンさん」丸善
  を挙げておく.1988 年に亡くなった著名な物理学者の Richard P.Feynman は, 実は最初のコネクションマシンの設計に一枚かんでいた. この本の中で Hillis はその際の出来事について回想しているのだが, これを読むと Frontier というのが実際にはどういうものなのか分ってくるのではないだろうか?

第 2 章

NCUBE/2

宮崎 純: miyazaki@jaist.ac.jp

2.1 nCUBE/2 の構成

本学に設置されたnCUBE/2は，MIMDのメッセージパッシング型並列計算機である．nCUBE/2のネットワークトポロジはハイパーキューブであり，ルーティングはworm hole routing方式を用いている．前世代のstep and forwardルーティングに比較して，通信レイテンシは非常に小さくなったが，現在の技術からすれば，nCUBE/2の通信レイテンシは必ずしも小さいとは言えない．

以下では，nCUBE/2の概要を示す．

2.1.1 Array Nodes

Array Node(Processor Element:PE)は，ユーザーが自由に使えるプロセッサである．本学に納入されたnCUBE/2は，256台のPEを持つ．すなわち8次元のハイパーキューブを構成している．各PEは，VAXコンパチの64bitのCPU，FPU，メモリ，およびネットワークインターフェースを持つが，本学のnCUBE/2のPE構成では，物理PEのIDが0～15は16MB，16～255は4MBのメモリを持つ．ネットワークインターフェースは，inputとoutputの2本1組の通信チャネルからなり，13組のArray Nodeへの通信チャネルと1組のI/O Nodeへの通信チャネルの合計14組の通信チャネルを持つ．各PEは，SVR4に準拠したVertexと呼ばれるOSが搭載されており(図2.1)，マルチプログラミングは勿論のこと，通常のUNIXの標準的なシステムコールも使用できる．諸元は表2.1の通りである．

ユーザーは，この8次元のハイパーキューブから任意の次元を使用できる(subcubing)．しかし，どの次元のsubcubeが割り当てられるか分からない[1]ため，全てのメモリが4MBであると仮定してプログラミングした方が良いであろう．占有したsubcube同士は干渉しないため，複数のユーザーが異なるsubcube上で同時に利用できる．また，占有したsubcube中の各PEは論理番号0番から始まる．

2.1.2 I/O Nodes

I/O Nodeは，ディスクサーバとグラフィックサーバの2種類が存在する．I/O Nodeは，その性質上ユーザーが自由に使用することは出来ない．

Disk

I/O Nodeの0x8002~0x8009の合計8台のPEがディスクサーバである．これらの各PEはSCSI-2で接続された1GBのディスクを2組持っている．すなわち合計16GBのディスク容量がある．通常，各PE

[1]強制的に任意のハイパーキューブを指定することも可能である

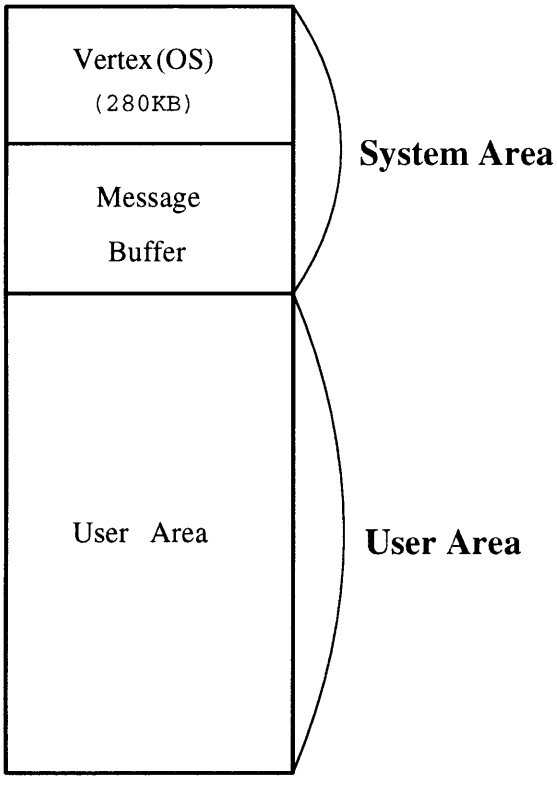


図 2.1: PE のメモリマップ

には nsdisk と呼ばれるディスクサーバが搭載されている．Array Node からの I/O Node へのファイルアクセスは，Array Node のシステムコールで実現されるため，あまり意識する必要がない．

ファイルのアクセスモードには 2 通りあり，一つは通常の Unix と同様に，各々の I/O Node のディスクにシーケンシャルアクセスするモードと，8 台の I/O Node を利用したストライピングアクセスモード[2]である (図 2.2).

[2]一般に，striping disk は高速であるが，nCUBE/2 の場合はそうとは限らない

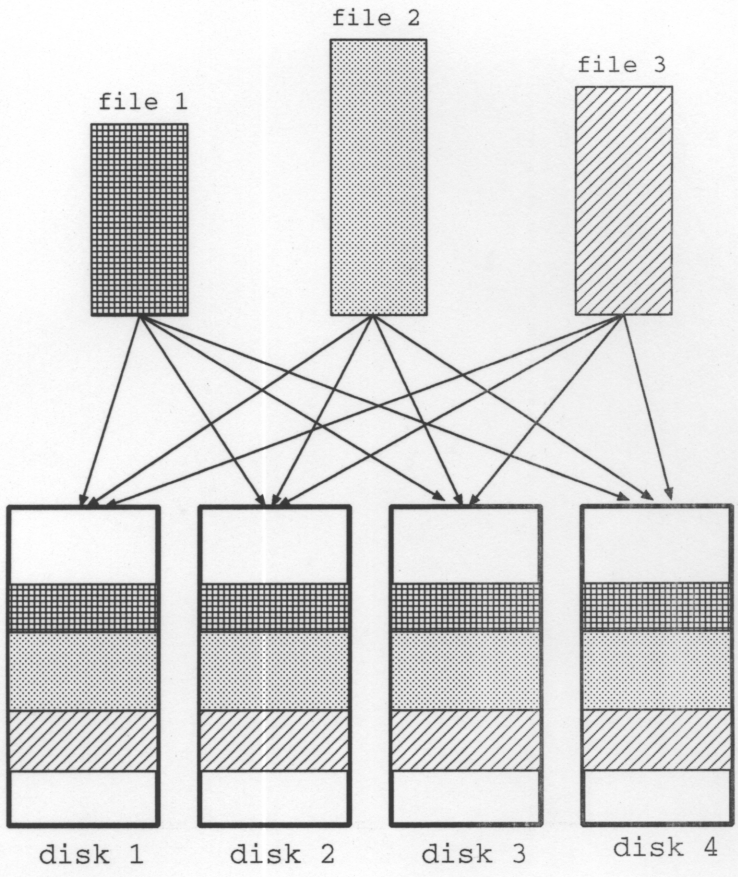


図 2.2: ストライプファイル

Graphics

I/O Node の 0x8070~0x807f の 16 台の PE には ngraph と呼ばれるグラフィックサーバが搭載されている[3].

2.1.3 プログラミング環境

nCUBE/2 では，標準で ANSI C，Fortran 77 をサポートしている．またライブラリとして，ncube(ホストプログラム用ライブラリ)，npara(nCUBE/2 パラレルライブラリ)，m(数学関数ライブラリ)，ngraph(nGRAPHICS サブシステムライブラリ)，ntv(nTV サブシステムライブラリ) 等をはじめ，util,n_lib(バックコンパチビリティの維持のためのライブラリ) がある．

ncube は，ホストプログラム用の通信，subcube のマッピング等のライブラリである．npara は，FFT，各種リダクション関数，ハイパーキューブから 2 次元配列へのマッピング等のライブラリである．

さらに，execution, communication, event の 3 つのプロファイラを備えている．プロファイラはユーザーのヒープを消費するため，プロファイルされるプログラムの実行時間は制限される．

[3]使用したことがないので，sample プログラムを参照して欲しい

表 2.1: nCUBE/2 諸元

Array Node	256PE
I/O Node	8PE (nsdisk) 16PE (ngraph)
CPU	64bit カスタム，10MIPS
FPU	32bit 3.3MFLOPS 64bit 2.4MFLOPS
メモリ	0 ～ 15: 16MB 16 ～ 255: 4MB
通信チャネル	シリアル 2.2MB/s
通信オーバヘッド	send 140(μsec) receive 55(μsec)
ディスク	16GB，SCSI-2 striping
言語	C,Fortran,Assembler
ライブラリ	パラレルライブラリ 数学関数ライブラリ等
デバッガ	ndb, ngdb
使用環境	マルチタスク (Node) マルチユーザー (Subcube)

2.2 nCUBE/2 を利用するための準備

nCUBE/2 はバックエンド型であり，フロントエンドは Sun4(MP600) を用いている．フロントエンドの名前は，ncube-1 である．

2.2.1 パスの追加

nCUBE/2 を使うにあたり，/usr/ncube/current/bin/sun4,
/usr/ncube/current/bin をサーチパスに加える．csh 系の shell を使用する人ならば，

```
ncube-1% set path = ( /usr/ncube/current/bin/sun4 \
                      /usr/ncube/current/bin $path)
```

sh 系の shell を使用する人ならば，

```
ncube-1% PATH=/usr/ncube/current/bin/sun4:\
             /usr/ncube/current/bin:$PATH
ncube-1% export PATH
```

等を shell のスタートアップファイルに書いておくと良い．

ncube-1 では /usr/local をマウントしていない[4]ため，エディタは vi(ed) 等を使うことになる．emacs 等の高級なエディタを使用したい場合，自分のマシン上で使用することとなる．

[4]emacs 等の重いプログラムをフロントエンド上で動かして欲しくないため

2.2.2 有用なコマンド

通常の on-line マニュアルは man で引くが，nCUBE/2 関係の on-line マニュアルは nman コマンドを使用する．

```
ncube-1% nman xnc
```

nwho コマンドで，誰がどの Array Node を使用しているか分かる．

```
ncube-1% nwho
miyazaki [0000, 0063]   Jun 15 18:18
root     [0064, 0127]   Jun 15 18:20
ncube-1%
```

2.3 プログラムのスタイル

nCUBE/2 上での基本的なプログラミングの例を C 言語を用いて示す．フロントエンド上でのプログラムをホストプログラムと呼び，Array Node 上のプログラムをノードプログラムと呼ぶ．なお，ホストプログラムはなくても良い．

個人の趣味によるが，基本的にメッセージパッシング型の並列計算機では，message driven となるよう設計した方がプログラミングしやすい．すなわち，受信したあるメッセージに対応するプロシジャをコールするように設計する．

2.3.1 Identify

プログラムは，フロントエンドから nCUBE/2 の各 PE にロードされる．各 PE にロードされたプログラムは，自分の PE の位置を関数 whoami で得ることができる．whoami を用いて，自分の PE の ID，プロセス ID，ホストプログラム ID，次元を得る方法を以下で示す．

```
int mynode, myproc, host, dims, nodes;
....
whoami(&mynode, &myproc, &host, &dims);
nodes = 1 << dims;
```

mynode には PE の論理 ID，**myproc** の上位 16bit にはプロセス ID が入り下位 16bit には PE の論理 ID が入る．**host** にはホストプログラムの ID が入り，host-node プログラミングを行なった時に，host と node プログラムとのメッセージの送受信に用いる．**dims** には，占有した subcube の次元数が入る．1 を **dims**(bit) 左シフトすることにより，占有した PE 数が得られる (**nodes**).

Fortran では，**whoami** の型は以下の通りである．

```
subroutine whoami(pid, proc, host, dims)
integer pid, proc, host, dims
```

2.3.2 Message passing

プロセッサ間およびプロセス間通信は，通常 nwrite, nread を用いる (図 2.3).

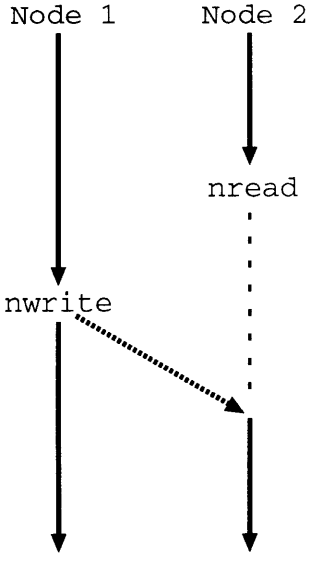


図 2.3: メッセージの送受信

メッセージ送信

nwrite は，PE の ID およびプロセス ID で特定される PE のプロセスへ任意のメッセージ長を送信できる．各メッセージの区別のためにメッセージタイプをつけることが出来る．なお，nwrite が発行されると，ユーザープログラムエリアからその PE のメッセージバッファにコピーされるため，メッセージ長はメッセージバッファのサイズを越えることは出来ない．また，メッセージタイプは 0 ～ 32767 の範囲で任意に使用できる．

```
char message[10];
int nbytes, dest, type, flag;
...
nbytes = sizeof message;
dest = (2 << 16) | 3;
type = 1;
nwrite(message, nbytes, dest, type, &flag);
```

message は送信したいメッセージへのポインタ，nbytes はメッセージ長，dest は，上位 16bit が送信先のプロセス ID，下位 16bit は送信先 PE の論理 ID を指定する．なお，**下位 16bit に 0xffff を指定した場合は，占有した全 PE の上位 16bit で指定されるプロセスにブロードキャストされる．**whoami で得たホスト ID を指定することによりホストプログラムに送信することも出来る．type は，メッセージを区別するためのメッセージタイプを指定する．flag は現在の OS では使用されていない．

Fortran では，nwrite の型は次のようになる．

```
integer function nwrite(buffer, nbytes, dest, type, flag)
dimension buffer(*)
integer nbytes, dest, type, flag
```

メッセージ受信

内容の理解のため，言葉の定義をしておく．

- **メッセージの到着:** その PE のシステムのメッセージバッファにメッセージが来ているが，まだユーザープログラムはそのメッセージを受けとっていない状態．
- **メッセージの受信:** ユーザープログラムが，システムのメッセージバッファからメッセージを受けとった状態．

nread は指定した PE の論理 ID，プロセス ID から，指定した長さのメッセージを受信する．nread は blocking の受信であり，指定したメッセージが到着するまで suspend する．

```
char message[10];
int nbytes, src, type, flag;
...
nbytes = sizeof message;
src = (2 << 16) | 255;
type = 1;
nread(message, nbytes, &src, &type, &flag);
```

message は受信したメッセージを入れるべき場所へのポインタである．すなわち，nread が完了するとシステムのメッセージバッファから，message で示されるユーザーエリアにメッセージがコピーされる．そのため，受信するメッセージ長に十分なメッセージエリアを確保しなければならない．nbytes は受信すべきメッセージの長さを指定する．src は受信すべきメッセージの送信元プロセス ID，PE の ID を指定する．上位 16bit は送信元のプロセス ID，下位 16bit は送信元の PE の論理 ID を示す．type は受信すべきメッセージタイプを指定する．ある決められたプロトコルにより，受信すべきメッセージの送信元のプロセス ID，PE の ID，メッセージタイプが既知の場合はよいが，通常，どこから，どのようなタイプのメッセージが送られてくるか分からないことが多い．このようなとき，src，type に−1 を指定することにより，任意の送信元，任意のメッセージタイプを受信することが可能である．このとき，src には受信したメッセージの送信元プロセス ID，PE の論理 ID が入り，type には受信したメッセージのタイプが入る．また，flag は現在使用されていない．

Fortran では，nread の型は以下のようになる．

```
integer function nread(buffer, nbytes, source, type, flag)
dimension buffer(*)
integer nbytes, source, type, flag
```

受信のテスト

nread は blocking の受信のため，CPU の使用効率を下げることが多い．そのため，ntest を用いて non-blocking のメッセージ受信を行なうことにより，CPU の使用効率を上げることができる．ntest は指定したメッセージがすでに到着したかどうかを判定するための関数である．

```
int size, src, type;
...
size = ntest(&src, &type);
```

src は，上位 16bit に送信元のプロセス ID，下位 16bit に送信元 PE の論理 ID を指定する．type は，到着すべきメッセージのタイプを指定する．src,type は nread の時と同様に，−1 を指定することにより，任意の送信元から，任意のタイプのメッセージを到着したかどうかを判定できる．もし，メッセージが到着していれば，ntest はそのメッセージのサイズを返す．これと同時に，src,type に−1 が指定されていれば，src にはそのメッセージの送信元のプロセス ID，PE の論理 ID が入り，type にはそのメッセージのタイプが入る．もし，メッセージが到着していなければ負のメッセージサイズを返す．これにより，nread では，メッセージサイズが既知でなければならなかったが，ntest を用いることにより，任意のメッセージ長を扱えることができるようになる．

以下に，その具体例を示す．任意のメッセージの到着をテストし，もしメッセージが到着していれば，そのメッセージを適切なサイズのエリアに受信し，work1 を実行する．もし，いかなるメッセージも到着していなければ work2 を実行する．

```
char *message;
int size, src, type;
...
/* any source, type message */
src = type = -1;
if ((size = ntest(&src, &type)) < 0) {
    /* no message arrived */
    work2();
} else {
    /* a message arrived */
    message = (char *)malloc(size);
    nread(message, size, &src, &type, NULL);
    work1();
}
...
```

Fortran では，ntest の型は以下のようになる．

```
integer function ntest(source, type)
integer source, type
```

その他

メッセージバッファへのコピー，メッセージバッファからのコピーを行なわない，多少高速な nwritep, nreadp ファミリがある．これには ngetp, nrelp を用いてメッセージバッファを allocate/release する必要がある．また，この方法はバグの原因となりやすいので，十分な注意が必要である．

2.3.3 Time

性能測定の際，時間の計測がしばしば必要となる．精密な時間測定には micclk を用いた方が良い[5]．なお，micclk の返す型は 64bit 整数であり，すなわち long long であることに注意する必要がある．double に型変換して使用した方が便利と思われる．返す値は，その PE が前回 boot されてからの経過時間をμsecで表したものである．

2.3.4 Barrier synchronization

メッセージパッシング型の並列計算機では，同期をメッセージパッシングのみを用いて行なうのことができる．しかし，各 PE の物理時計を合わせるときなど，高速なバリア同期が必要な場合がしばしばある．このようなとき，nsync を用いれば 20 clock cycle 以内でバリア同期がとれる (図 2.4)．nsync は，同期要因を区別するための引数を取る．

[5]何種類か時間測定関数が用意されているが，amicclk などは 1 tick = 128(μsec) といった荒い精度である

```
#define ADJUST 0
...
long long micclk();
double base_time;
...
nsync(ADJUST);
base_time = (double)micclk();
```

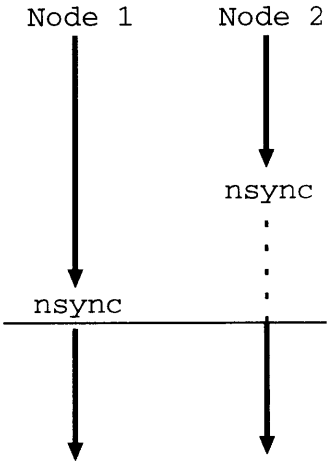


図 2.4: バリア同期

2.3.5 File access

ファイルへのアクセスは，NFS でファイルサーバ等のファイルをアクセスできるが，そのアクセス速度は非常に制限される．NFS の場合は，通常の C のプログラミングと全く同様である．

高速なファイルアクセスを行なうためには，I/O Node のディスク (ndisk) を用いる．以下では ndisk の使用方法について述べる．

Striped file

ファイルは ndisk 上だけでなく，NFS を併用して，任意にストライピングできる．通常は，NFS を併用した場合，速度上問題があるので ndisk 上に限定して設定してある．現在の設定では，8-stripe を主軸に，各種のブロッキングファクタ別に設定してある．ブロッキングファクタは，16, 32, 64, 128, 256, 1k, 2k, 4k, 16k, 32k, 64k, 128k, 256k, 1m である．

1KB のブロッキングファクタのストライプファイルをアクセスする場合の例を示す．他のブロッキングファクタでのストライプファイルは，ブロッキングファクタを BF としたとき，/nsdiskBF である[6]．

```
int fd;
...
fd = open("/nsdisk1k/test", O_WRONLY|O_CREAT, 0644);
write(fd, "0123456789", 10);
close(fd);
```

[6] /usr/ncube/etc/stab を参照

ストライピングすることにより，ファイルの並列アクセスが可能となりファイルアクセスの高速化できるはずであるが，実測の結果，現在のバージョンでの nsdisk(ディスクサーバ) では，ストライピングにより高速ファイルアクセスは望めないことが分かっている．ストライピングすることの利点は，長大なファイルを生成できる程度である．

Non striped file

I/O Node の各ディスクにアクセスする方法である．前述の通り，I/O Node のノード ID は，0x8002~0x8009 である．各 I/O Node は，SCSI-ID が 0 と 1 の 2 つの 1GB のディスクを持っている．各々のディスクは，次のような規則のディレクトリ名を持っている．I/O Node のノード ID の下 1 桁を N，SCSI-ID を S とすれば，//d0NSの prefix から始まるディレクトリである．

以下では，I/O Node 0x8005 の SCSI の ID が 0 ディスクのアクセスの例を示す．

```
FILE *fp;
char buf[80];
...
fp = fopen("//d050/test", "r");
fgets(buf, 80, fp);
fclose(fp);
```

2.4 プログラムのコンパイルと実行

2.4.1 コンパイラ

C コンパイラ

フロントエンド ncube-1 上で ncc によってクロスコンパイルを行なう．ncc は ANSI C 準拠である．なお，現在の Ver3.2 では，gcc と同様に-g と-O の併用が可能となった．以下では，主要なオプションを表2.2に挙げる．通信バッファが溢れるような時，-Ncomm を用いて通信バッファのサイズを大きくする．auto

表 2.2: コンパイル時の主要なオプション

オプション	機能
-d dim	使用する subcube の次元 (dim) を指定する．(default: 0)
-Ncomm size	システムのメッセージバッファサイズをバイト単位で変更する．(default: 65536)
-Nheap size	ユーザーの heap サイズをバイト単位で変更する．(default: 32768)
-Nstack size	ユーザーの stack サイズをバイト単位で変更する．(default: 65536)
-Nfile size	file descripter table のサイズを変更する．特に，striped file を使う時は大きくしなければならない．(default: 8)
-g	デバッガのためのシンボルテーブルを生成する．
-O	コードの最適化を行なう．

変数のサイズが大きい時や，深い再帰呼びだしがある場合には，スタックが溢れる時がある．このような時，-Nstack を用いてスタックのサイズを大きくする．大きなヒープが必要なときは，-Nheap を用いてヒープのサイズを大きくする．通信バッファ，スタック，ヒープを大きくすれば，ユーザープログラムを格納する領域が小さくなるので，そのアプリケーションに応じてこの 3 つのサイズをうまく増減させる必要がある．

なお，高速にプログラムを動作させたい場合は，-O オプション (コード最適化) をつけるべきである．

ncc の使用例は，次の通りである．

```
ncube-1% ncc -d 3 -Nheap 65536 -g -O test.c -o test
```

Fortran コンパイラ

ncc 同様，ncube-1 上で nf77 によってクロスコンパイルを行なう．オプションは，ncc のものとほぼ共通である．また，ncc の用いて Fortran のソースをコンパイルすることもできる[7]．また，ループの最適化，整数型のサイズの指定など Fortran 特有のコンパイラオプションも存在する[8]．

Fortran で高速にプログラムを実行させたい場合は，-O オプション (コード最適化) のみならず-BX オプション (ループの最適化) を併用した方が良いであろう．

nf77 の使用例は以下の通りである．

[7] ソースのサフィックス (.f) を見て，Fortran のソースと解釈しているようである．

[8] 詳しくはマニュアルで ncc を引き，FORTRAN OPTIONS の辺りを見よ．

```
ncube-1% nf77 -d 1 -O -BX test.f -o test
```

2.4.2 make

デフォルトの make を使うと implicit なルールを適用するので，次のようなルールを用いると良い．

```
CC       = ncc
CFLAGS   =
CPPFLAGS =

.SUFFIXES: .o .c
.c:;     $(CC) $(CFLAGS) $(CPPFLAGS) -o $* $<
.c.o:;   $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $@
```

2.4.3 実行

直接実行する方法

Unix のように，直接バイナリを実行する．占有する次元は，コンパイル時に-d オプションで指定した値である．また，この方法は SPMD モデルの実行しかできないことに注意が必要である．すなわち，各 PE に同一のプログラムしかロードできない．

```
ncube-1% cat test.c
#include <stdio.h>
main()
{
    int mynode, myproc, host, dim;

    whoami(&mynode, &myproc, &host, &dim);
    printf("Hello World! My node=%d\n", mynode);
}
ncube-1% ncc -d 1 test.c -o test
ncube-1% test
Hello World! My node=1
Hello World! My node=0
ncube-1%
```

xnc ローダを用いる方法

次元数や，ヒープサイズ等を実行時に変更したい場合に用いる．直接実行する方法よりも一般的である．先ほどの test.c を例にとると，以下のようになる．

```
ncube-1% ncc test.c -o test
ncube-1% xnc -d 1 test
Hello World! My node=0
Hello World! My node=1
ncube-1%
```

また，MPMD モデルでの実行時は xnc ローダを使わなければ，異なるプログラムを各 PE にロードできない．この時，どの PE にどのプログラムをロードするのかを指定するマップファイルが必要である．以下にその例を示す

```
ncube-1% cat sample.c
#include <stdio.h>
main()
{
    int mynode, myproc, host, dim;

    whoami(&mynode, &myproc, &host, &dim);
    printf("Hi, World! My node=%d\n", mynode);
}
ncube-1% ncc sample.c -o sample
ncube-1% cat mapfile
0-1 test
2-3 sample
ncube-1% xnc -d 2 -l mapfile
Hi, World! My node=3
Hi, World! My node=2
Hello World! My node=0
Hello World! My node=1
ncube-1%
```

2.5 プログラムのデバッグ

2.5.1 ndb デバッガを用いたデバッグ

nCUBE/2 のサポートしているデバッガは，現在の Ver3.2 では dbx スタイルの ndb と，gdb スタイルの ngdb がある．しかしながら，ngdb は gdb を単に移植したもので，専用設計されていないため使い心地が悪く，さらにはまだバグが多く存在するようで，Segmentation fault 等がたまに起きる．そのため，ngdb の使用を控えた方が良いかと思われる．

以下では，ndb について説明する．ndb のほとんどの命令，例えば next, step, stop, where 等，dbx の命令と共通である．拡張されたのは，ノード集合の概念，ノード間の移動が主である．

先ほどの test.c を例に，ndb の簡単な使い方を示す．

```
ncube-1% ndb test
Ndb Version 3.2.5 -- Copyright (C) 1991,1992 ParaSoft
Pre-scanning global symbols.
Reading symbol table "test": 100%
symbol table: 416 public, 4 local.
ndb> run xnc -d 1 -P test
/* test を 1 次元で実行. start point で停止 (-P オプション) */
Process has 2 nodes. (Origin at 0)
Node    0; 1> on all stop in main
/* 全てのノードの main に breakpoint を設定 */
Inserting breakpoint at test.c, line 7 ....  ..... done
Node    0; 1> on 0-1 cont
/* 0-1 番のノードについて continue */
Node    0; 1> pick 1
/* 1 番のノードに制御を移動する */
Node    1; 1> where
/* 1 番のノードはどこを実行しているかを調べる */
main+0x2  [test.c,7] () at 0xc0010fb4
_kickoff+0x203  [?,?] ???( 1, 65537, 94142464 ....) at 0xc0010fcc
start+0x29  [crt.s,?] () at 0xc001100c
Node    1; 1> next
/* 1 番のノードを next */
   8:          printf("TEST pid=%d\n",pid);
Node    1; 1> pick 0
/* 0 番のノードに制御を移動する */
Node    0; 1> next
/* 0 番のノードについて next */
   8:          printf("TEST pid=%d\n",pid);
Node    0; 1> print pid
/* 変数 pid の内容を表示する */
                    pid = 0
ndb> quit
ncube-1%
```

MPMD のデバッグは，複数の ndb を上げて行なう．まず，1 つめの ndb をあげ，-P オプションで start point で停止しておく．

```
ncube-1% cat map
0 test
1 sample
ncube-1% ndb -x 0 test
ndb> run xnc -d 1 -l map -P
Node      0; 1>
```

次に，他の tty で nps でデバッグしようとするプログラムの PID を調べる (*印のプロセスを探す)．その後，ndb を-p オプションを用いて PID を指定し，立ち上げる．

```
ncube-1% nps -au
USER       PID CHAN  SPID STAT NODES NMIN NMAX  TMIN  TMAX START
miyazaki  1449    3     -    1     2    0    1     0 65535 16:11
miyazaki  1448    3  1449    0     2*   0    1 32009 32009 16:11
ncube-1% ndb -p 1448 -x 1 sample
Node      1; 1>
```

2.5.2 デバッグのテクニック

1. nCUBE/2 のノードは，abort()，segmentation fault 等で core を吐かない．このようなトラップにかかった場合，自動的に ndb が上がる[9]ようになっているが，そのノードに十分な空きメモリ空間がない場合は ndb は上がらず，実行は終了してしまう．このような場合に備えて，エラートラップのために次のような関数[10]を使うと良い．

```
ErrorTrap(char *s, int mynode)
{
    printf("error: %s in Node=%d\n", s, mynode);
    fflush(stdout);
    while(1);
}
```

これで，`while(1)` の部分で exit せずに止まっているので，MPMD プログラムのデバッグ方法の時と同様にして，プロセス ID を ndb 立ち上げ時に指定することにより，後からスタックトレース，変数値の調査等ができる[11]．

2. int, short, unsigned などの C のプリミティブデータ型を明示せず，typedef を用いて抽象的なデータ型を用いてプログラムする方が良い．なぜなら，各 PE はメモリの制限が厳しいため，例えば int から short にすることによりプログラムサイズや，メッセージサイズを小さくしたい時に簡単に変更可能とするためである．

3. ndb での予約語は，プログラム中の変数として使用しない方が良い．ndb は予約語と，プログラムの変数を区別できないからである．特に次のデータオブジェクトを指すためのポインタに `next` という名前を付けることが多いが，ndb は，`next` コマンドと混同してしまうので要注意である．

---

[9] この場合の ndb はノード上のメモリ空間上に上がるようになっている

[10] マクロの方がいいかも知れない

[11] この場合の ndb はホスト上のメモリ空間を使用する

2.6 注意事項

program size

256PE の内，16PE(物理 PE0 ～ 15) は 16MB，残りの 240PE(物理 PE16 ～ 255) は 4MB である．複数のユーザーが使用するため，いつでも物理 PE0 ～ 15 が割り当てられるとは限らない．CPU が CISC であるため，バイナリは SPARC に比べて小さいが，4MB に収まるよう[12]に各サイズを調整しなければならない．それでも駄目な時は，nstrip を用いて symbol table を削るという最終手段がある．

byte order

ホスト (SPARC) は big endian，ノード (VAX 系) は little endian のため，host-node プログラミングをする場合は，特に，メッセージを交換する時は byte order の変換をしなければならない．

printf dump

printf は，全てフロントエンドを経由して表示される．そのため，多数の PE からの printf はフロントエンドの負荷を上げてしまい，他の人に迷惑をかける[13]．できるだけ printf での dump を避け，ndisk を使うようにする．こちらの方が快適である．

---

[12] Vertex(各ノードの OS) のサイズは約 280KB である

[13] フロントエンドが重大なボトルネックとなるだけでなく，ファイルサーバに書き込みを行なう場合などは，ネットワークにも大きな負担をかけてしまう

第 3 章

convex

井口 寧 : inoguchi@jaist.ac.jp

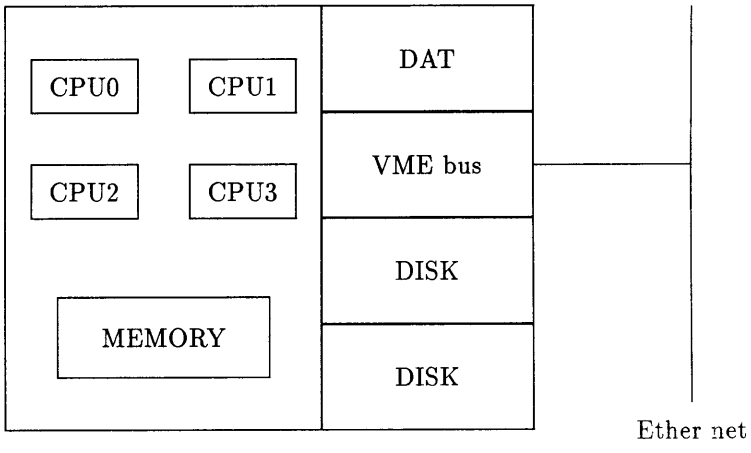
本章では, 密結合型ベクトル計算機である CONVEX C3440 について解説する.

最初に本学に導入された計算機の概要, 次に実際に使う際に役に立つように, 簡易的な使用法とベクトル化及び並列化について述べる. その後, より高度な使い方として, コンパイラの諸機能, バッチジョブ制御, チェックポイント・リスタートなどの機能について述べる.

3.1 概要

convex (ホスト名:convex) は, 本学の小規模計算サーバーシステムの 1 つであり, 米国コンベックス社の CONVEX C3440 (倍精度浮動少数点演算 200MFLOPS [1] , 主記憶 512MB, 磁気ディスク容量 5GB) を用いている. このシステムは, 64bit スカラー演算とベクトル演算[2] に加え, 4 プロセッサ構造の並列演算を行なう機能を有し, CONVEX UNIX オペレーティングシステムの下で, 高度な会話型システムの機能が提供可能である. さらに自動ベクトル化・並列化が行なわれる Fortran, C コンパイラのサポート, 大容量実メモリなどの機能を備えている.

3.1.1 ハードウェア構成



CPU	50MFLOPS × 4
Memory	512MByte
Disk	5GByte (2.5 × 2 台)
Network	Ethernet
MT	DAT (DDS format)

3.1.2 ファイルシステム

/work1	1.5GByte	作業領域
/work	1.12GByte	checkpoint restart 保存領域
/tmp	250MByte	一時作業領域
swap	1GByte	スワップ領域

このうち, /work1 が作業領域として一般ユーザーに解放されている. /work1 に計算の中間結果などを格納すると, ジョブ全体の処理速度が改善され, ネットワークの負荷も低くなる. 但し, /work1 は convex を利用する全てのユーザーで共有するので, 格納するファイルは, 中間結果などの一時的に置くファイルのみとし, 長期間占有することは避けるようにする. また, /work1 上のファイルは, 長期間の占有 (約 1 週間前後), システムエラー時, および領域が一杯になった時など, 予告なく消去されることがあるので, 重要なファイルや恒久的なファイルは置かないようにする.

[1] 単精度では更にこの 2 倍弱 (400MFLOPS 弱) となる.

[2] 最近ではワークステーションでも理論性能が 200MFLOPS を越えるものもある. しかしワークステーションではキャッシュなどの関係上, 高速演算が可能な条件が非常に制限される. 一方, convex などのベクトル計算機では, 大規模な問題も扱えるように設計されている. 両者の「速さの質」の違いを理解した上で利用されたい.

3.1.3 パス

convex のソフトウェアを利用するためには, /usr/convex にパスが通っている必要がある. 具体的には, ログオン時に以下のコマンドを実行すると良い.

```
% set path=($path /usr/convex)
```

また, このようなコマンドを自動的に実行するように, 次のようなスクリプトを .cshrc ファイルに書いておくのも便利である.

```
if ( -d /usr/convex ) then
        set     path=(/usr/convex $path)
endif
```

3.1.4 サポートされているソフトウェア

convex では主に次のようなソフトウェアが利用可能である.

CONVEX OS	ご存知 convex の OS
FC	FORTRAN コンパイラ
C	C コンパイラ
CXwindow	motif ベースの X-clients
CXbatch	バッチ処理システム
CXdb	デバッガ
profs	プロファイラ (性能評価ツール)
CXpa	パフォーマンス アナライザ

3.2 簡易使用法と並列化

3.2.1 簡易使用法

ここでは, ログオン後, test.f というプログラムをコンパイルし, 実行する過程を簡単に説明する.

パスの設定

ログオン後, 次のコマンドを実行する.

```
% set path=(/usr/convex $path)
```

この手続きは, .cshrc によってパスが自動設定される場合は不要である.

プログラムのコンパイル

```
% fc -O3 -o test.out test.f
```

この例では, ソースプログラム test.f を, 最適化オプション -O3 (ベクトル化+並列化) でコンパイルし, 実行形式のファイルを test.out という名前で得る.

JCL の作成

次に, 実行プログラムを実行するための JCL ファイルを作成する. JCL ファイルは, 実行すべきコマンド列をつらつらと書き連ねたファイルである.

```
% cat go.jcl
#!/bin/sh
./test.out
```

1 行目は実行されるシェルを指定している. 標準では sh が起動される. 2 行目に実行するコマンドを書く. ここで示した例では, 先に得た test.out を実行する 1 行だけのスクリプトとする. 1 つのジョブで複数のコマンドを実行したい場合は, 実行する順に 1 行に 1 命令ずつ書く.

バッチキューへ投入 (ジョブの実行)

JCL をバッチキューに投入する[3] . バッチキューは, ジョブの大きさ (CPU 時間, 使用メモリなど) の制限により, 複数のクラスに分かれてる. ここでは short ジョブ (CPU 時間 120 分×4) として投入する. ジョブクラスについては, バッチキューの使用法で述べる.

```
% qsub -q short go.jcl
```

先に作った go.jcl を投入する.

この後, しばらくするとジョブが開始される. ジョブの実行状況は, qstat コマンドで確認できる. また, ジョブ投入時に me オプションを指定すれば, ジョブ終了時にメールで通知される.

実行するプログラムが複数ある場合は, 「プログラムのコンパイル」から再度入力する.

[3] ジョブは, バッチを使わずに会話型でも実行できるが, 長時間実行するとジョブの優先順位が低下し, 自分のジョブがちっとも終らないという結果となる.

3.2.2 ベクトル化と並列化

convex では, vector 型 CPU が並列に (密結合) 接続されている.

最適化オプション -O0 又は -O1 は, FORTRAN プログラムに対し, スカラーの最適化を行なう. 最適化オプション -O2 を指定すると, コンパイラはベクトル化を行なう. また, -O3 を指定すると, ベクトル化に加えて並列化も行う.

並列化を行なわない場合, ジョブは convex 内に複数ある CPU の内のどれか 1 つで実行される. CPU が複数あるので, 複数のジョブがかかっている場合, CPU の数までのジョブ数ならば, それぞれのジョブの実行時間は他のジョブがかかっているか否かにかかわらず一定である.

並列化を行なった場合, 1 つのジョブが複数の CPU を並列に使用する. 従って, 効率の良い並列化が達成できたジョブは, もし他のジョブがかかっていなければ, 並列化を行なわない場合に比べ数倍高速に実行できる. 並列化を行なったジョブが複数かかっている場合は, 複数の CPU をあたかも 1 つの CPU のように扱い, それを各ジョブが分け合うことになる.

vector 化

最適化オプション -O2 を指定すると, ベクトル化による最適化を行なう. ベクトル化は, ベクトル化対象のループ回数がおよそ 4 回以上の時, スカラーよりも高速になる. ベクトル化したループの回数が少ない場合は, -O1 オプションによってスカラーのまま実行した方が速い場合がある.

ベクトル化は, 最初に DO ループの入れ換えが行なわれ, 次に最内周 DO ループがベクトル化される.

次に最内周 DO ループのベクトル化例を示す.

```
convex% cat test.f
        subroutine      init(x)
        dimension       x(512,512,512)

        do 10 k=1,512
          do 20 j=1,512
            do 30 i=1,512
              x(i,j,k) = 1.0
   30       continue
   20     continue
   10   continue

        return
        end
convex% fc -O2 -c test.f


            Optimization for Procedure INIT

Line     Iter.   Reordering            Optimizing / Special
Exec.
Num.     Var.    Transformation         Transformation
 Mode
-----------------------------------------------------------------
   4     K       Scalar

   5     J       Scalar

   6     I       FULL VECTOR

convex%
```

i が最内周 DO ループの制御変数であり, これがベクトル化されている.

最適化の結果, DO ループの入れ換えが自動的に行なわれる場合もある.

例えば, 3 次元ベクトル $x(i,j,k)$ はメモリ上に,

$$x(1,1,1), x(2,1,1), x(3,1,1), \cdots, x(1,2,1), x(2,2,1), x(3,2,1),$$
$$\cdots, x(1,1,2), x(2,1,2), x(3,1,2), \cdots, x(n-1,n,n), x(n,n,n)$$

のように格納されるので, 一番左の添字 (i) に関してベクトル化すると, データが連続にアクセスでき, 実行が高速になることが期待できる. ループを入れ換えても実行結果に影響がないと判断される場合, コンパイラにより自動的にループの入れ換えが行なわれる.

例として, 先のプログラムで i と k の DO ループを入れ換え, ベクトル化する.

```
convex% cat test.f
        subroutine      init(x)
        dimension       x(512,512,512)

        do 10 i=1,512
          do 20 j=1,512
            do 30 k=1,512
              x(i,j,k) = 1.0
   30       continue
   20     continue
   10   continue

        return
        end

convex% fc -O2 -c test.f


            Optimization for Procedure INIT

Line     Iter.   Reordering            Optimizing / Special
Exec.
Num.     Var.    Transformation         Transformation
 Mode
-----------------------------------------------------------------
   4     I       FULL VECTOR Inter

   5     J       Scalar

   6     K       Scalar

Line     Iter.   Analysis
Num.     Var.
-----------------------------------------------------------------
   4     I       Interchanged to innermost

convex%
```

コンパイラからのメッセージの最後に, 制御変数 i の DO ループを最内周に移動したことが示されている. このように, DO ループを入れ換え, 添字 i を制御している DO ループを最内周としてベクトル化すると, データをメモリ上で連続にアクセスで, 高速実行が可能になる.

並列化

最適化オプション -O3 を指定すると, ベクトル化による最適化を行なった上で, 更に並列化を行なう.

並列化を行なった場合, CPU 時間は実行した CPU それぞれの CPU 時間の合計として計算される. 例えば, 1 つの CPU で 8 時間かかるジョブは, 4CPU で効率良く並列実行できれば, 実際のターンアラウンド時間は 2 時間となる. しかし, time コマンドなどで返される CPU 時間は 8 時間 (2 時間×4) となる. 逆に言えば, time コマンドで CPU 時間が 8 時間と返されても, 実際には 2 時間で結果が得られたことになる. バッチジョブのジョブクラスでは, CPU 時間の見積りが重要なので, 特に注意する必要がある.

多重 DO ループの場合, 先に述べたようにベクトル化は最内周 DO ループに対して行なわれる. これに対し, 並列化は最外周 DO ループに対して行なわれる. 次に例を示す.

```
convex% cat test.f
        subroutine      init(x)
        dimension       x(512,512,512)

        do 10 k=1,512
          do 20 j=1,512
            do 30 i=1,512
              x(i,j,k) = 1.0
   30       continue
   20     continue
   10   continue

        return
        end
convex% fc -O3 -c test.f


            Optimization for Procedure INIT

Line     Iter.   Reordering            Optimizing / Special
Exec.
Num.     Var.    Transformation         Transformation
 Mode
-----------------------------------------------------------------
   4     K       PARALLEL

   5     J       Scalar

   6     I       FULL VECTOR


convex%
```

この例では, 最外周 DO ループ制御変数 k が並列化され, 最内周 DO ループ制御変数 i がベクトル化されている. 中間のループは並列化された各プロセッサで, それぞれスカラーで実行される.

一方, ベクトル化に伴う DO ループの入れ換えが行なわれた後で並列化が行なわれる場合, 最初に DO ループの入れ換えが行なわれ, 次に入れ換えられた DO ループがベクトル化され, 最後に残りの DO ループのうち最外周の DO ループが並列化される.

```
convex% cat test.f
        subroutine      init(x)
        dimension       x(512,512,512)

        do 10 i=1,512
          do 20 j=1,512
            do 30 k=1,512
              x(i,j,k) = 1.0
   30       continue
   20     continue
   10   continue

        return
        end
convex% fc -O3 -c test.f


            Optimization for Procedure INIT

Line      Iter.   Reordering            Optimizing / Special
Exec.
Num.      Var.    Transformation         Transformation
 Mode
-------------------------------------------------------------------
   4      I       FULL VECTOR Inter

   5      J       PARALLEL

   6      K       Scalar


Line      Iter.   Analysis
Num.      Var.
-------------------------------------------------------------------
   4      I       Interchanged to innermost

convex%
```

上の例では, 最初にiを制御変数とするDOループが最内周に移され, ベクトル化される. 次に, 残りのDOループ(j,k)のうち, 最外周DOループであるjのDOループが並列化される. kに関するDOループは, 各プロセッサでそれぞれ並列に, スカラーで実行される.

多重DOループの並列化は, ベクトル化の為のループの入れ換えを行なった後, 最外周のDOループに対して行なわれる. 従って, 次の様な特性を考慮し, DOループの順番を決めるべきである.

- 最外周DOループの回数が非常に小さいと, プロセッサに均等に負荷分散ができず, 遊休プロセッサができる.
- メモリ上の配列の格納状態を考慮し, 離れている要素が並列化された方がメモリアクセス競合が起こりにくく性能が高い.

先の例では, 配列要素x(i,j,k)で, iが最も要素間が近くkが最も要素間が離れるので, iに関してベクトル化, kに関して並列化を行なうと効率が良くなる. また, kの要素数も十分にあるので, 負荷が偏る可能性も少ない. 従って, jに関して並列化を行なっている後者の例よりも, kに関して並列化を行なっている前者の例の方が, より高速な実行が期待できる.

コンパイルの結果, 意図した並列化が行なわれないかもしれない. この場合, コンパイラの診断メッセー

ジを検討し, 可能な限り効率の良い並列化が行なわれるようにする.

単一 DO ループの場合, -O3 オプションが指定されると, コンパイラは DO ループを複数の区間に分割し, 区間を各プロセッサに割り当てる. 各プロセッサは, 割り当てられた DO ループの一部の区間をベクトル化することにより, 実行する.

```
convex% cat test2.f
        subroutine      init(x)
        dimension       x(512)

        do 10 i=1,512
          x(i) = 1.0
   10   continue

        return
        end
convex% fc -O3 -c test2.f


           Optimization for Procedure INIT

Line     Iter.   Reordering            Optimizing / Special
Exec.
Num.     Var.    Transformation         Transformation
 Mode
-----------------------------------------------------------------
   4     I       PARA/VECTOR


Line     Iter.   Analysis
Num.     Var.
-----------------------------------------------------------------
   4     I       Parallel outer strip mine loop

convex%
```

この例では, i を

$$1 \sim 128, 129 \sim 256, 257 \sim 384, 385 \sim 512$$

の様な複数の区間に分割し, それぞれをプロセッサに割り当てる. プロセッサは各々の区間をベクトル化し実行する.

3.3 FORTRAN

3.3.1 特徴

CONVEX FORTRAN COMPILER では以下のような特徴がある.

1. 他のマシンのコンパイラとの互換性

   (a) FORTRAN 77,90 に準拠

   (b) CRAY の FORTRAN COMPILER のソースコードのコンパイルが可能

   (c) DEC の FORTRAN COMPILER のソースコードのコンパイルが可能

   (d) SUN の FORTRAN COMPILER のソースコードのコンパイルが可能

   グローバルなスカラーオプティマイゼーションが可能

2. パフォーマンス

   (a) 自動ベクトル化

   (b) 自動並列化

   (c) インライン機能による最適化

3. 他のユーティリティとの接続

   (a) プロファイラ prof,bprof,gprof が利用可能

   (b) プログラム解析ツール CONVEX Performance Analizer (CXpa) が利用可能

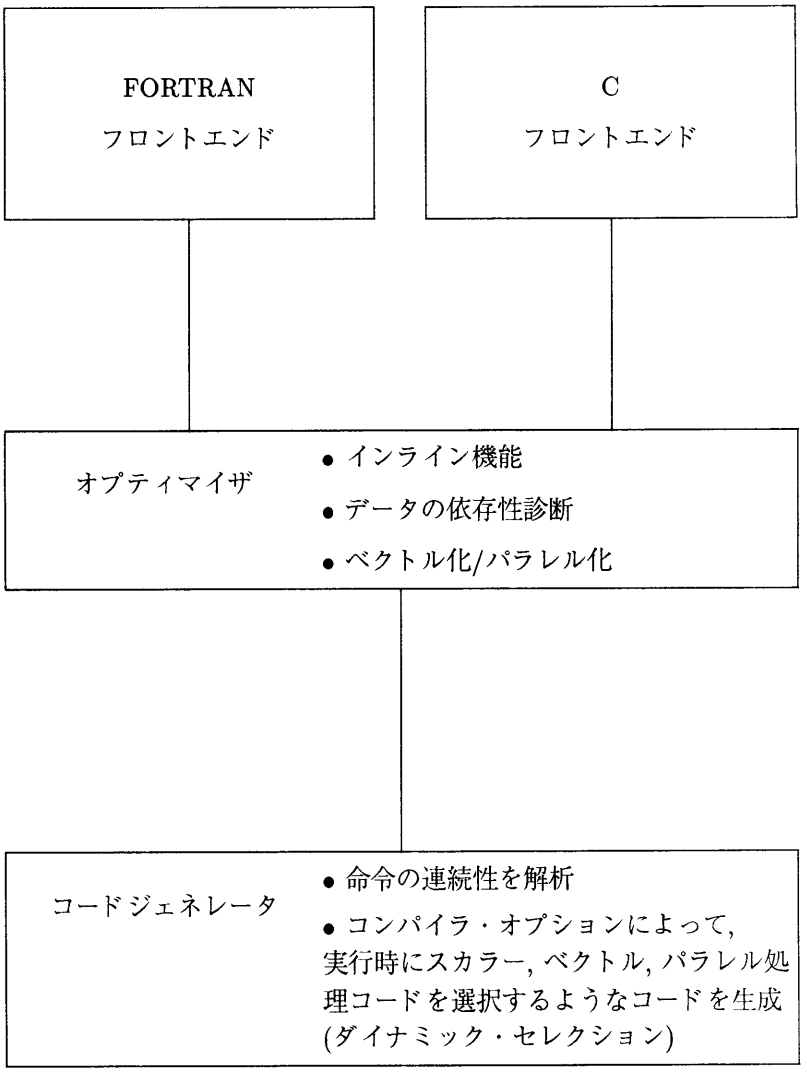
   (c) ウィンドウデバッガ CXdb が利用可能

4. 実行モジュールのフォーマットの変換

   (a) IEEE 浮動小数点フォーマット出力可能. 他機種とのバイナリデータ共有可能.

   (b) ソースの変更なく単精度・倍精度の実行モジュールが作成可能

3.3.2 他のコンパイラとの関連



3.3.3 FORTRAN プログラムのコンパイラ起動方法

実行モジュール作成方法

```
% fc [option] <source-file-name>
```

例 (基本):

```
% fc -O3 test.f
```

test.f というソースプログラムをコンパイルする. ベクトル化並列化による最適化 (-O3 オプション) が行なわれる.

オブジェクトファイル作成方法

```
% fc -c [option] <source-file-name>
```

例:

```
% fc -c -O3 test.f
```

ソースファイル test.f を -O3 最適化しコンパイルする. -c オプションが付いているので, オブジェクトファイルのみの出力となる.

リンクの方法

```
% fc [option] <object-file-name> [<source-file-name>]
```

例:

```
% fc -o test.exe test1.f test2.o test3.f
```

test1.f, test3.f をコンパイルし, 出力オブジェクトを test2.o とリンクし, 実行形式ファイル test.exe に出力する.

3.3.4 FORTRAN コンパイラの主なオプション

CONVEX FORTRAN の主なオプションは次の表の通りである. 詳細はオンラインマニュアル (man fc で引ける) を参照すると良い.

これは便利! 最も高速に実行するコードを生成するオプションは `-O3` である.

例:

```
% fc -O3 test.f
```

test.f を最適化オプション -O3 でコンパイルし, a.out ファイル (デフォルト) に出力する.

言語互換オプション

option	概要
-F66	FORTRAN66 表記を受け付ける
-f90	FORTRAN90 表記を受け付ける
-cfc	Cray FORTRAN compiler (cft77) 表記を選択
-sfc	Sun FORTRAN compiler (f77) 拡張表記を選択
-vfc	VAX FORTRAN compiler 拡張表記を選択 (参考 : -dfc)
-72	72 カラムまでを処理

最適化関連オプション

option	概要
-no	コンパイラの最適化を行なわない
-O0	基本ブロックの計算機非依存スカラー最適化
-O1	-O0 + プログラムユニット内の計算機非依存スカラー最適化
-O2	-O1 + ベクトル化 (DO 繰り返し 4 回以上で効果あり)
-O3	-O2 + パラレル化
-ep *n*	プロセッサ数を n と仮定. システムスループットを犠牲にして高速化を達成する
-il	インラインの中間ファイル (拡張子 .fil) を作成
-is *dir*	実際に中間ファイルが存在するディレクトリを指定し インラインを可能にする

最適化オプション -On (n=0,1,2,3) が指定されない場合, コンパイラは最適化を行わない. また, この他にも最適化関連のオプションがある.

コード生成オプション

option	概要
-S	アセンブラコードの出力
-c	オブジェクトファイルのみ出力
-i[n]	INTEGER, LOGICAL 宣言でバイト数を明示していない時, n で指定されたバイト数でコンパイルされる. n=2,4,8
-r[n]	REAL 宣言でバイト数を明示していない時, n で指定されたバイト数でコンパイルされる. n=4,8

デバッグ関連オプション

option	概要
-cxdb	CXdb を使用するためのオプション
-db or -g	標準デバッガ csd を使用するためのオプション
-p	プロファイラ (prof) 用実行モジュールの作成
-pa	CXpa を使用するためのオプション
-pb	bprof 用実行モジュールの作成
-pg	gprof 用実行モジュールの作成

コンパイラ出力オプション

option	概要
-LST	リスト出力
-na	診断勧告メッセージを出力しない
-nw	診断警告メッセージを出力しない
-or *table*	最適化に関してのメッセージを出力する *table* = all,none,loop,array
-xr	クロスリファレンス出力を行なう

その他

option	概要
-I[x]	インクルードファイルが格納されているディレクトリの指定
-o [file]	出力ファイル名の指定 (デフォルトは a.out)
-vn	コンパイラのバージョン表示
-l[x]	リンク・ライブラリの指定, /usr/lib/lib$x$ をリンク

3.4 バッチキューの使用法

3.4.1 概要

本学の convex では, short, long, very long, memory という 4 つのバッチキュー (ジョブを入れる待ち行列) が用意されている. バッチジョブによる実行と, 会話型処理によるフォアグランド・ジョブやバックグラウンド・ジョブの実行では, 以下に述べるような差がある.

- ジョブの開始から終了まで優先順位が変化しない. (csh から投入したジョブは, 実行時間が長くなると優先順位が低くなる)
- バックグラウンド・ジョブでは, ジョブが投入された時点で他のジョブが実行中であるか否かに関わらず即座に実行されるが, バッチキューによる投入では, 決まった数 (1 つ) ずつ実行され, 前のジョブが終了するまで次のジョブが実行されない. 従ってシステム資源の競合が避けられ, 効率的な実行が可能である.

このため, 主に長時間ジョブ, メモリを多く必要とするジョブを実行する場合, バッチジョブによる実行を行なうと, 効率的にジョブを実行することができる.

3.4.2 バッチジョブクラス

バッチジョブクラスは以下のように設定されている.

queue name	short	long	very long	memory
キューの別名	s	l	v	m
priority	48	32	16	16
CPU 時間制限	2 時間 ×4	12 時間 ×4	無制限	無制限
データサイズ制限	128MB	256MB	128MB	無制限
同時実行ジョブ数	1	1	2	1

短時間で終わるジョブは short, 非常に長時間必要なジョブは very long クラスを利用する. また, memory クラスは, 非常に大きなメモリを必要とするジョブ用である.

CPU 時間制限は, 4 つの CPU 時間の合計で制限される. 従って, short ジョブクラスは, 並列化を行わないプログラム (最適化オプション -O2) を 1CPU だけを用いて 8 時間実行できる. 並列化を行うようにコンパイルした (最適化オプション -O3) プログラムは 4 つの CPU を使用し合計 8 時間, すなわち 4 つの CPU をそれぞれ 2 時間分ずつ占有できる. 同時実行ジョブ数は, そのジョブクラスで同時に run 状態に入れるジョブの数である. 現在 very long ジョブクラスが 2 になっており, 同時に very long ジョブを 2 つ並行実行できる. 但し, 1 ユーザー当たりの同時実行可能なジョブ数は, 1 つだけである.

現在のジョブクラスの設定は, 以下のコマンドで調べることができる.

```
% qmgr show long queues
```

将来ジョブクラスの制限が変更された場合でも, qmgr コマンドによって, その時点でのジョブクラスの制限について知ることができる.

3.4.3 バッチジョブの投入

バッチキューにジョブを投入するためには, qsub コマンドを使用する.

```
% qsub [-q <queue-name>] <script-file>
```

<queue-name> でジョブクラス (s, l, v, m のいずれか) を指定する.

<script-file> は実行するシェルスクリプトである.[4]簡単には, 実行するコマンドを順に記述する.

以下に例を示す.

```
% qsub -q v start.jcl
```

start.jcl の内容をジョブクラス very long で実行する. start.jcl は, 例えば次のようなファイルである.

```
% cat start.jcl
fc -O3 -o test.out test.f
./test.out
```

この例では, ソースファイル test.f を最適化コンパイルし, test.out という出力ファイル名で出力する. この後 test.out を実行する.

次に qsub の主なオプションを示す.

option	概要
-q [queue]	キュー (ジョブクラス) の指定.
-o path	標準出力を書き込むファイルの指定
-eo	標準エラー出力を標準出力ファイルに書き出す.
-me	ジョブ終了時にユーザー宛にメールを送信
-s [shell]	スクリプトを実行するシェルを指定

3.4.4 バッチキュー・ジョブの操作

キューの状態を調べる

```
% qstat [-a] [-l] [-m] [-x] [-u <user-name>] [<queue-name>]
```

queue-name: ジョブクラス (s, l, v, m のいずれか) の指定

-a: 全ユーザーのジョブを表示

-l: 詳細に表示

-m: 中くらいに表示

ジョブの停止

```
% qmgr hold request <request-id>
```

request-id は qstat コマンドで調べることができる. 自分のジョブに対してのみ停止できる.

[4]このファイルは必ずテキストファイルを指定する. 実行形式のバイナリファイルは受け付けられない

停止したジョブの再開

```
% qmgr release request <request-id>
```

注:オペレータによって停止されたジョブは一般ユーザーが再開することはできない.

ジョブクラスの変更

```
% qmgr move my_request <request-id> <queue-name>
```

キューの移動はジョブがまだ実行されていない時にのみ可能である.

ジョブの消去

```
% qdel [-k] <request-id>
```

指定した **<request-id>**のジョブを消去する. もし, 実行中のジョブを途中でとり止める時は, -k オプションをつける.

3.5 チェックポイント・リスタート

3.5.1 概要

Check Point Restart は, プロセスの状態をファイルとしてディスクに保存しておき, 後で保存されたファイルの情報によって, チェックポイントされた時点からプロセスを再び実行することができる機能である.

適当な時間が経過するごとにチェックポイントすることにより, 予期しないシステムダウンなどからジョブを復旧させることが可能になる.

親子関係のある全てのプロセスは, 親子関係を 1 つのグループとして自動的にチェックポイントするために, 全体のプロセスの階層構造の情報がストアされる.

チェックポイント機能は, コマンド, バッチコマンド, 及び C, FORTRAN ライブラリ関数とで構成されている.

コマンド

chkpnt プロセスのチェックポイントを行なう

restart プロセスのリスタート

ライブラリ関数

chkpnt() プロセスのチェックポイントを行なう

restart() プロセスのリスタート

バッチ用コマンド

プロセスが CXBatch によって実行されている場合は, 以下のコマンドを使用する.

qchkpnt バッチプロセスのチェックポイントを行なう

qrestart バッチプロセスのリスタート

階層プロセスのチェックポイント

階層プロセスがチェックポイントされると, 階層中の全プロセスを 1 つのグループとしてリスタートできる.

階層プロセスがチェックポイントされると, 階層中の全プロセスは親プロセスがチェックポイントされる前に, pattach によって停止される. 全プロセスが停止した時に, 各々のプロセス毎にチェックポイント・ファイルを作成する.

チェックポイント・ファイル

チェックポイント・ファイルには, プロセスがチェックポイントされた時にプロセスのリスタートに必要な情報が書き込まれる. チェックポイント・ファイルは, chkpnt コマンドによってデフォルト名, またはユーザー指定の名前が付けられる. デフォルトのチェックポイント・ファイル名は, コマンドとプロセス id から付けられる. 例えば pid 12345 の csh プロセスのチェックポイント・ファイル名は csh.12345 となる.

階層プロセスをチェックポイントした場合, 階層的なコマンド名とプロセス id によりチェックポイント・ファイル名が決定される. 例えば, vi がプロセス id 12346 で, 先の csh の子プロセスとして動いている時に, csh がチェックポイントされた場合, vi のチェックポイント・ファイル名は csh.vi.12346 になる.

リスタート

リスタートはチェックポイントされたプロセスを復帰させる. ルートプロセスから始まって, チェックポイントされた全てのプロセスのチェックポイント・ファイルが読み込まれ, プロセスが復帰する.

階層プロセスにおける全てのプロセスの状態は, チェックポイント操作によって, プロセスが停止した時と同じ状態で主記憶にストアされる. この操作によって, プロセスの pid, スレッド, レジスタ, ファイルディスクリプタ, 仮想空間の領域, text, data がストアされる.

リスタート操作が始まった時に, 階層プロセスのスタートを行なう restart プロセスが作成される. リスタート操作が階層プロセスのリストアを終了した時, 実行権はプロセスがチェックポイントされた時に実行権を持っていたプロセスに渡される. 階層中のどれか 1 つでもリスタートに失敗すれば, プロセス全体のリスタートは失敗する.

3.5.2 制限事項

ある条件のもとでは, プロセスをチェックポイントすることがができない. チェックポイント不可能な条件はいろいろあるが, 詳細は man chkpnt 調べると良い.

注意!! chkpnt コマンドまたはライブラリ関数では, 既に存在するチェックポイント・ファイルを上書きすることはできない

注意!! チェックポイントされたプロセスがオープンしているファイルの内容は, 自動的にはセーブされない. (chkpnt でオプションを指定すると可能) 通常, ファイルのパス名とオープンしたファイルのポジションがセーブされるが, データはセーブされない.

注意!! チェックポイントしたプロセスが, 一般のファイルをリード/ライトしている場合, プロセスがリスタートまで, ファイルに関して操作を行なってはいけない. 即ち, ファイルのパスや名前を変更してはいけないし, ファイルのタイムスタンプがチェックポイントをかけたときより新しくなっていれば, 警告が発生する.

3.5.3 コマンドによるチェックポイント

chkpnt コマンド

```
chkpnt [ -CFinqrvX ] [ -j|-p ] [ -d <chkpnt-directory> ]
     [ -f <chkpnt-file> ] [ -I <logfile> ] [ -L <logfile> ]
     [ -k|-K <signo> ] -P <fd>|<pid>
```

pid にはプロセス ID を指定する. また, pid の代わりに, P fd 形式を用いてファイルディスクリプタを指定することができる.

chkpnt コマンドの有用なオプションは以下のようなものがある.

-C 使用中のファイルをチェックポイント・ディレクトリにコピー

-F チェックポイント不可能な状態でも強制的に行なう

-i 会話モード

-n チェックポイント可能かどうかのテスト. 実行はしない

-q メッセージ出力を行なわない

-v 実行過程表示

-X デバッグ出力

-d chkpnt-directory チェックポイント・ファイルを作るディレクトリの指定. デフォルトはカレントディレクトリ

-f chkpnt-file チェックポイント・ファイル名の指定

-L logfile 会話モードでのログをとる

-k signo チェックポイント終了後, 階層のルートプロセスにシグナルを送る

-K signo チェックポイント終了後, 階層の全プロセスにシグナルを送る

pid チェックポイントをかける pid の指定

基本的なチェックポイントのかけ方

一番簡単な使い方は以下の通りである.

```
% chkpnt 9121
```

pid=9121 のプロセスにチェックポイントをかける.

チェックポイント・ディレクトリを指定しなければ, カレントディレクトリにチェックポイント・ファイルが作られる. チェックポイント・ファイルは restart コマンドを実行する際に使用される.

注意!! chkpnt コマンドはチェックポイント・ファイルを上書きしない. 従って同じプロセスを複数回チェックポイントを行なう際は, 以前のファイルを消去, チェックポイント・ファイル名の変更 (-f オプション), またはチェックポイント・ファイルを作るディレクトリを変更する (-d オプション) 操作を行なわなくてはならない. また, -F オプションを用いると, 強制的に上書きを行なう.

チェックポイント・ファイル格納ディレクトリの指定

[-d chkpnt-directory] オプションを指定すると, チェックポイント・ファイルを格納するディレクトリを変更できる. chkpnt-directory は相対パスまたは絶対パスで指定する.

例:

```
% chkpnt -d $HOME/mychkpnt 9121
```

ホームディレクトリの mychkpnt に プロセス 9121 のチェックポイント・ファイルを作ることを指定.

restart コマンド

```
restart [ -CFiqtvwWXz ] [ -k|-K <signo> ]  <checkpoint-file>
```

<checkpoint-file> に, chkpnt コマンドで出力されたチェックポイント・ファイル (「チェックポイント・ファイル」の項参照) を指定する.

restart コマンドで有用なオプションは以下のようなものがある.

-C chkpnt-directory から元のディレクトリにファイルを戻す.

-F リスタート条件が揃っていなくても強制的にリスタートする.

-i 会話モード

-q メッセージ出力を行なわない

-t トレースステートでリスタート. デバッガで使用される.

-X デバッグ出力

-z ストップ状態のプロセスをリスタート.

-k signo リスタート終了後, 階層のルートプロセスにシグナルを送る

-K signo リスタート終了後, 階層の全プロセスにシグナルを送る

基本的なリスタートのやり方

一番簡単な使い方は以下の通りである.

```
% restart ./a.out.9121
```

pid=9121 のプロセスにチェックポイントをかける. (チェックポイントをかけた pid=9121 のコマンド名が a.out の時, チェックポイント・ファイルとして a.out.9121 がカレントディレクトリに出力される)

wait/no wait

w オプションをつけると, リスタートしたプロセスの終了を待つ. restart コマンドは, リスタートしたプロセスの終了コードを, 自身の終了コードとして返す. このオプションはデフォルトである.

W オプションをつけると, 目的のプロセスの終了を待たない. このオプションでは, restart は fork せずに自分の pid がリスタートしたプロセスの pid に変更されるので, シェルモードでは使用しない.

3.5.4 バッチジョブのチェックポイント

バッチジョブとして動いているプロセスをチェックポイントするためには, qchkpnt 及び qrestart コマンドを使用する. また, pid を指定する代わりにジョブ投入時の requestid を指定する.

バッチジョブがチェックポイントされると, チェックポイント・ファイルが生成され, 実行はそのまま続けられる. システムのダウンなどでバッチシステムが停止し, その後バッチシステムが復旧した場合に, チェックポイント・ファイルが読み込まれ, チェックポイントされたジョブがチェックポイントされたところから再開される. **従って, 通常一般ユーザーは qrestart コマンドを発行する必要はない.** バッチシステムの復旧後, 自分のジョブが停止している場合に qrestart コマンドを発行する.

チェックポイントされたバッチジョブが再開されるためには, キューにそのまま残っている必要がある. チェックポイント後, ジョブをバッチキューから削除 (`qdel -k <request-id>`) すると, チェックポイント・ファイルも削除され, 再び実行を再開することができなくなる.

実行中のバッチジョブを停止, 再開するためには, `qmgr SUspend Request <request-id>` 及び, `qmgr RESume Resuest <request-id>` を使用する. qmgr suspend コマンドは, ジョブをチェックポイントし, 終了する. qmgr resume コマンドは, qmgr suspend コマンドによって中断されたジョブを再開する.

チェックポイント・ファイルはカレントディレクトリではなく, バッチジョブ用のチェックポイント・ディレクトリに格納される.

qchkpnt コマンド

```
qchkpnt [ -e <freq> ] [-f] <request-id>
```

requestid には, バッチジョブ投入時のリクエスト ID を指定する. この ID は qstat コマンドを用いても見ることができる.

qchkpnt コマンドの有用なオプションは以下のようなものがある.

-e freq 周期的にチェックポイントをかける. 周期は freq で指定し, この書式は, <number-unit> の形式. unit には, Hours | Days | Weeks のいずれかが指定できる. また, number に 0 を指定すると, 周期的チェックポイントの解除となる.

-f チェックポイント要求が出されたプロセス群が, チェックポイント不可能な資源を保持していた場合でも強制的にチェックポイントを行なう.

qrestart コマンド

```
qrestart [-f] [ <request-id> ]
```

qrestart コマンドは, qchkpnt コマンドでチェックポイントされたジョブの実行を再開する. requestid には, バッチジョブ投入時のリクエスト ID を指定する. この ID は qstat コマンドを用いても見ることができる.

f オプションを指定すると, チェックポイント要求が出されたプロセス群がチェックポイント不可能な状態にあっても強制的に復帰を試みる.

使用例

ジョブの投入

```
convex% cat go.jcl
#!/bin/sh
fc -O3 test1.f
a.out

convex% qsub -q short go.jcl
Request 1544.convex submitted to queue: short.
```

jcl ファイル go.jcl の内容を確認し, ジョブとして投入する.

チェックポイント操作

```
convex% qstat
Queue for short jobs.
short@convex;  type=BATCH;  [ENABLED, RUNNING];  pri=48
aliases: s, short_queue, S, SHORT
  0 exit;   1 run;   0 stage;   0 queued;   0 wait;   0 hold;   0
arrive;

        REQUEST NAME        REQUEST ID            USER  PRI    STATE
   PGRP
    1:        R2d.jcl      1544.convex          inoguchi  31  RUNNING
  19272

Queue for long jobs.
long@convex;  type=BATCH;  [ENABLED, INACTIVE];  pri=32
aliases: l, long_queue, L, LONG
  0 exit;   0 run;   0 stage;   0 queued;   0 wait;   0 hold;   0
arrive;


Queue for very long jobs.
verylong@convex;  type=BATCH;  [ENABLED, INACTIVE];  pri=16
aliases: v, verylong_queue, V, VERYLONG
  0 exit;   0 run;   0 stage;   0 queued;   0 wait;   0 hold;   0
arrive;


memory@convex;  type=BATCH;  [ENABLED, INACTIVE];  pri=16
aliases: m, memory_queue, M, MEMORY
  0 exit;   0 run;   0 stage;   0 queued;   0 wait;   0 hold;   0
arrive;


convex% qchkpnt 1544
Request 1544 checkpointed.
```

最初に requestid を確認し, 自分のジョブにチェックポイントをかけている.

チェックポイントされたファイルの確認.
(super user でないと確認できない)

```
convex% qmgr show parameters

  Debug level =  0
  Default batch_request priority = 31
  Default batch_request queue = long
  Default destination_retry time = 72 hours
  Default destination_retry wait = 5 minutes
  Global per-user runlimit = NONE
  Checkpoint directory = /work/chkpnt
  Accounting log file = /dev/null
  Activity ID mask = None
  Mail account = root
  Next available sequence number = 1545
  Batch request shell choice strategy = FREE


convex% su
Password:
# cd /work/chkpnt
#ls
long      memory    short     verylong
# ls short
1545.convex.0
# ls -l short
total 4
drwxr-xr-x  2 root          512 Jul 21 23:31 1545.convex.0
# ls -l short/1545.convex.0/
total 1892
-r-------- 1 inoguchi  663778 Jul 21 23:31 R2d.jcl.19365
-r-------- 1 inoguchi  196830 Jul 21 23:31 R2d.jcl.19365.sh.19368
-r-------- 1 inoguchi 1298645 Jul 21 23:31 R2d.jcl.19365.sh.a.out.19372
# exit
convex%
```

バッチジョブのチェックポイント・ファイルが作成されるディレクトリは, 'qmgr show parameters' コマンドの応答中の 'Checkpoint directory =' 欄に出力される. この例では, /work/chkpnt に指定されている. 次にスーパーユーザーになって, このファイルができているか確かめる. qchkpnt コマンドでチェックポイントをかけた時エラーが返されなければ, チェックポイント・ファイルはきちんと生成されるので, 確認の必要はない.(一般ユーザーは確認できない)

著者紹介

志田 和人　北陸先端科学技術大学院大学 情報科学研究科 博士後期課程
shida@jaist.ac.jp

河瀬 剛　北陸先端科学技術大学院大学 情報科学研究科 博士前期課程
kawase@jaist.ac.jp

宮崎 純　北陸先端科学技術大学院大学 情報科学研究科 博士後期課程
miyazaki@jaist.ac.jp

井口 寧　北陸先端科学技術大学院大学 情報科学研究科 博士後期課程
inoguchi@jaist.ac.jp

北陸先端科学技術大学院大学
情報科学センター 利用の手引

1994 年 9 月 20 日 初版 第 1 刷

著作者　志田 和人
河瀬 剛
宮崎 純
井口 寧

発行　北陸先端科学技術大学院大学 情報科学センター
〒 923-12 石川県能美郡辰口町旭台 15
TEL 0761-51-1300, FAX 0761-51-1305

Printed in Japan