分散メモリ計算機におけるプロセッサ間通信の
コンパイラ最適化手法の研究

1995 年 9 月

若谷彰良

論文概要

本論文は、分散メモリ並列計算機における通信コストの削減方法として通信レイテンシの隠蔽について検討し、コンパイラの最適化手法を提案したものである。

次世代のスーパーコンピュータは分散メモリ型マルチコンピュータであるといわれており、既にその範疇に含まれる製品も現れている。しかし、現状では必ずしも普及しているとは言い難く、そのための標準プログラミング環境、特にプログラミング言語の確立が急務であると考える。言語実装の最大のポイントは、性能劣化の原因となるプロセッサ間通信時間の最小化をいかに行うかである。コンパイラが通信レイテンシを削減することにより、計算機の最大性能 (ピーク性能) に近い実効性能を得ることができる。

第 2 章においては、分散メモリ計算機の現状及び問題点を明かにし、分散メモリ並列計算機の標準ソフトウエアの重要性およびその技術背景を述べた。さらにそのコンパイラによる通信レイテンシ最小化手法の限界を示した。

第 3 章では、通信レイテンシを削減するためのハードウエアを持つ並列計算機 ADENA の設計方針について述べ、配列の再分散通信を高速にした分散メモリ計算機であるという特徴を生かしたプログラミング言語 ADETRAN に対するコンパイラの設計思想を示した。ADENA は、通信レイテンシの隠蔽を行なえる S スキームモードをプロセッサに実装しており、コンパイラがこの機能を最大限に利用することが重要である。そのためのプログラム変換手法を述べ、さらに言語の実装手法として仮想プロセッサの物理プロセッサへのマッピング方法及びメモリアクセスの効率化手法について述べた。また、これらの手法の実アプリケーションに対する効果を明かにし、ループサイズが比較的小さい時でも 30% 以上の性能向上が得られたことを述べた。

第 4 章においては、集団通信の中でもコストの大きい配列再分散に注目し、前章で仮定したようなハードウエアを用いずに、一般の分散メモリ計算機において適用可能なソフトウエアによる通信コスト削減方法を検討した。そのために、まず、仮想プロセッサ (概念プロセッサ) を物理プロセッサ (実プロセッサ) に効率良く配置する方法 (螺旋配置) を提案した。これは通信コンフリクトを改善することにより通信レイテンシ削減するもので、その効果についても述べた。例えば、1024×1024 の 2 次元配列の再分散の場合、iPSC/860 において 36%の効率化が得られた。

また、通信レイテンシそのものを計算のかげに隠蔽する方法 (メッセージ掘り下げ法) を提案した。これは、計算に必要な最小の部分に配列を分割し、その単位で計算と通信を分割し、分割された計算と通信をパイプライン的にオーバーラップするものである。本方式が適用可能な配列再分散通信の条件は、通信起動時間と通信レイテンシに依存したパラメータで決定される。さらに、2 次元および 3 次

元の配列再分散における効果を明かにしており、サイズが2048×2048の 2 次元 ADI プログラム例に本方式を適用することにより 1.7 倍の通信時間のスピードアップが得られることを示した。

第 5 章においては、前章で述べたメッセージ掘り下げ法が配列の再分散だけではなく、規則性の高い通信を持つ任意のアプリケーションに対して適用できるものであることを示し、通信をオーバーラップする計算部分の決定方法及び並列化すべき配列の次元と分散配置されている次元の関係についても言及した。また、最小通信単位をメッセージ集団化手法でブロック化する際の最適ブロックサイズ及び通信スピードアップを解析的に決定する手法を示した。さらに、一般化メッセージ掘り下げ法の適用例として、配列の間接アクセスの通信に応用した場合について述べ、iPSC/860 上での実験では、通信時間のスピードアップが 260%を越えることもあることがわかった。最期に、メッセージ掘り下げ法のコンパイラへの実装方法についてその概要を述べ、従来の FORTRAN D コンパイラなどの HPF 系コンパイラへの実装は容易であることを示した。

目 次

1 序論 **5**

1.1 通信最適化手法の問題点 5

1.2 本論文の構成と主な成果 7

2 背景 **9**

2.1 分散メモリ型並列計算機 9

2.2 HPF 言語 12

2.2.1 概要 13

2.2.2 データ分散 (data distribution) 13

2.2.3 実行文 14

2.2.4 第 2 版に向けて 15

2.3 通信レイテンシ最小化手法 16

3 ADETRAN コンパイラ **21**

3.1 ADENA システム 22

3.1.1 HX ネットワーク 23

3.1.2 S スキームモード 24

3.2 ADETRAN の設計方針 25

3.3 ADETRAN の概要 26

3.3.1 基本データ構造 26

3.3.2 region 文 27

3.3.3 pdo 文 27

3.3.4 pass 文 28

3.3.5 ifall/ifany 文 28

3.3.6 サブルーチン 29

3.3.7 pad/pour 文 29

3.3.8 file 配列 30

3.4 HPF との比較 32

3.5 プログラミング例 33

3.5.1 Splitting Up CG 33

3.5.2 疎行列の積演算 . . . 35
3.6 ADETRAN コンパイラの実装方針 . . . 37
3.6.1 実行フロー . . . 38
3.6.2 仮想プロセッサとインデックス解析 . . . 40
3.6.3 pour/pad 文 . . . 45
3.6.4 file 配列 . . . 46
3.7 データロード最適化 . . . 50
3.8 通信最適化 . . . 53
3.8.1 データ通信 . . . 53
3.8.2 通信型変換 . . . 56
3.9 実験 . . . 61
3.10 まとめ . . . 64

4 配列再分散における通信コスト削減 67
4.1 螺旋配置による通信コスト削減 . . . 67
4.1.1 配列再分散 . . . 67
4.1.2 実プロセッサの螺旋配置 . . . 69
4.1.3 関連技術との比較 . . . 76
4.1.4 実験 . . . 77
4.2 メッセージ掘り下げ法 (message strip-mining) . . . 78
4.2.1 ADI における配列再分散 . . . 78
4.2.2 有効性評価 . . . 82
4.2.3 実験 . . . 88
4.2.4 考察 . . . 89
4.3 まとめ . . . 90

5 一般のアプリケーションに対する通信コスト削減 91
5.1 基本構成 . . . 91
5.2 基本構成に対する考察 . . . 93
5.2.1 通信を隠蔽する計算ループの選択 . . . 93
5.2.2 並列化ループとデータ分散 . . . 94

5.3 最適ブロックサイズと理想スピードアップ値 96
5.4 検査官 / 執行官アルゴリズムを用いた間接アクセスへの適用 98
5.5 実験 101
5.5.1 配列の再分散 102
5.5.2 間接アクセスに対する執行官アルゴリズム 103
5.6 コンパイラへの実装 105
5.7 まとめ 109

6 結論 **111**

A 付録 **125**

A.1 通信起動最小化手法 125
A.2 ADAPTOR システムの評価 131
A.2.1 ADAPTOR system の概要 131
A.2.2 ADAPTOR 言語 131
A.2.3 コンパイラ実装 133
A.2.4 実行評価 136
A.2.5 まとめ 141
A.2.6 ADAPTOR の評価に使用したプログラム 141
A.3 APR/xHPF コンパイラ 148

1 序論

1.1 通信最適化手法の問題点

次世代スーパーコンピュータの候補として、分散メモリマルチコンピュータの要求がますます増えている。これを実現するためには、コンパイラの最適化能力がキーとなっている。なかでも、分散メモリマルチコンピュータを使用する上で不可欠となるプロセッサ間通信コストをいかに小さくするかが重要である。しかし、従来からある通信最適化技術には必ずしも汎用性がなく、すべてのアプリケーションにそのまま有効ではなかった。プロセッサ間通信のコストは、通信起動と通信レイテンシからなる。この通信レイテンシを削減する方法として、反復組み替え法 (iteration reordering)[19] やベクタメッセージパイプライン法 (vector message pipelining)[35, 40] などがある。しかし、これらの方法は通信を計算に隠蔽することを基本としているので、通信パターンがある制約を満たす場合にのみ適用される方法である。

既存の通信レイテンシ最小化手法が適用しにくい例として、まず、2 次元の拡散方程式に対する ADI 法 [34] について考える。基礎となる偏微分方程式は次のようなものである。

$$\frac{\partial u}{\partial t} = D(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) \tag{1}$$

ADI の基本的な考えは、時間ステップを$\frac{\Delta t}{2}$の 2 つのステップに分け、それぞれのステップにおいて 1 次元の簡単な 3 重対角行列を解くようにすることである。

$$u_{j,l}^{n+\frac{1}{2}} = u_{j,l}^{n} + \frac{1}{2}\alpha(\delta_x^2 u_{j,l}^{n+\frac{1}{2}} + \delta_y^2 u_{j,l}^{n}) \tag{2}$$

$$u_{j,l}^{n+1} = u_{j,l}^{n+\frac{1}{2}} + \frac{1}{2}\alpha(\delta_x^2 u_{j,l}^{n+\frac{1}{2}} + \delta_y^2 u_{j,l}^{n+1}) \tag{3}$$

$$\alpha \equiv \frac{D\Delta t}{\Delta^2} \tag{4}$$

$$\Delta \equiv \Delta x = \Delta y \tag{5}$$

ADI 法を分散メモリコンピュータに効果的に実装するために、u に対する配列 `u` を 2 種類の方向に分散させる必要がある。最初のステップ $(n + \frac{1}{2}$ step) に対して、`u` は (*, BLOCK) で分散される。というのは、最初のステップは 3 重対角行列を第 1 の次元で解くものであるからである。言い換えると、`u(j,l)` は　`u(k,l)` $(\forall\, k)$ を用いて解かれる。この計算は第 2 の次元において並列に行える。これに反して、第 2 ステップは第 2 の次元に関する 3 重対角行列を解くものなので、配列 `u` は (BLOCK, *)

で分散される。よって、全てのステップにおいて分散された配列の再分散が必要となる。配列の再分散は本質的にはコストの大きい集合通信 (collective communication) であるので、そのような通信のコストを最小化することは重要である。

しかし、反復組み替え法および メッセージパイプライニング法はこのケースには効果的には適用できない。というのは、ループの全ての計算でリモートデータが必要であり、通信を隠すべき計算部分がないからである。よって、大きな通信部分は残る。

次に間接アクセスを含む計算について考える。間接アクセスは実アプリケーションにおいてしばしば用いられる。例えば、有限要素法、分子シミュレーション、疎行列計算などである。間接アクセスの扱いは、これらのアプリケーションにおいて共通なものである。2 種類の配列が使用される。1 つは空間の値を保持するもので、もう一方はその点へのインデックスを保持するものである。

図 1 に示したプログラムは物理データ v の平均値を計算するものである。

```
  do 1 i=1,M
1   v(i)=...
  do 2 i=1,N
   do 3 j=1,le(i)
3    x(i)=x(i)+v(l(i,j))
2  x(i)=x(i)/le(i)
```

図 1: 間接アクセスを用いたプログラム例

分散メモリマルチコンピュータ上で、配列 x,v と l はそれぞれ分散され、配列 x と配列 l の第 1 次元は、align していると仮定する。しかし、配列 v はいずれの配列とも align されない。よって、配列要素 v(l(i,j)) の多くはリモートプロセッサから供給される。

v(l(i,j)) の多くがリモートプロセッサに保持されているので、ループ 3 は通信を隠蔽するのに十分なローカル計算を持たない。また、配列 v はループ 1 で定義されている。よって、この場合も反復組み替え法およびベクタメッセージパイプライニング法のいずれも効果的には適用されない。

このように、既存技術ではカバーしきれない応用分野も多い。しかし、そのための通信最適化技術は、標準プログラミング言語の普及には欠かせないものである。

そこで、本論文では、まず、ハードウエアを用いた通信最適化手法について述べた後、規則性の高い通信には幅広く応用できるソフトウエアによる通信最適化の手法を 3 つ述べる。ここで用いる方法は、通信と計算をオーバラップしてレイテンシを隠蔽するために、通信と計算を複数の単位に分割し、

依存関係にしたがってパイプライン処理する手法を基本としている。本論文で述べる通信レイテンシの削減手法は、規則性の高い集団通信 (collective communication) を含む広い範囲の一般のプログラムに適用可能であり、実効性能の改善が期待できるものである。

1.2 本論文の構成と主な成果

本研究は、分散メモリ並列計算機における通信コストの削減方法として通信レイテンシの隠蔽について検討し、コンパイラの最適化手法を提案し、6 章からなっている。

第 1 章においては、現在の通信最適化手法の問題点、特にレイテンシ削減に関する問題点を明かにし、任意のアプリケーションに適用できる手法が無かったことを述べ、本研究の目的と構成、及び主な成果を述べている。

第 2 章においては、スーパーコンピュータの発展の延長として分散メモリ計算機が必然であることを示し、その状況及び普及のための問題点を明かにしている。特に、分散メモリ並列計算機の標準ソフトウエアとしてプログラミング言語 HPF が重要であることを示し、HPF 言語の技術背景を述べて、さらにそのコンパイラにおける通信レイテンシ最小化手法の限界を示している。

第 3 章では、通信レイテンシを削減するためのハードウエアを持つ並列計算機 ADENA の設計方針について述べ、配列の再分散通信を高速にした分散メモリ計算機であるという特徴を生かしたプログラミング言語 ADETRAN に対するコンパイラの設計思想を示している。ADENA は、S スキームモードをプロセッサに実装し、通信レイテンシの低減化をハードウエアレベルで実現しているが、この通信モードの最適利用のためのプログラム変換手法を述べ、さらに言語の実装手法として仮想プロセッサの物理プロセッサへのマッピング方法及びメモリアクセスの効率化手法についても述べている。また、これらの手法の実アプリケーションにおける効果を明かにし、ループサイズが比較的小さい時でも有効であり、30% 以上の性能向上が得られたことを述べている。

第 4 章においては、集団通信の中でもコストの大きい配列再分散に注目し、前章で仮定したようなハードウエアを用いずに、一般の分散メモリ計算機において適用可能なソフトウエアによる通信コスト削減方法を検討している。そのために、まず、仮想プロセッサ (概念プロセッサ) を物理プロセッサ (実プロセッサ) に効率良く配置する方法 (螺旋配置) を提案し、その通信レイテンシの削減効果について述べている。また、通信レイテンシそのものを計算のかげに隠蔽する方法 (メッセージ掘り下げ法) を提案し、2 次元および 3 次元の配列再分散における効果を明かにしており、サイズが2048×2048の 2 次元 ADI プログラムを例として本方式を適用することにより 1.7 倍のスピードアップが得られることを示している。

第 5 章においては、前章で述べたメッセージ掘り下げ法が配列の再分散だけではなく、規則性の高い通信を持つ任意のアプリケーションに対して適用できるものであることを示し、通信をオーバーラップする計算部分の決定方法及び並列化すべき配列の次元と分散配置されている次元の関係についても言及している。また、最小通信単位をメッセージ集団化手法でブロック化する際の最適ブロックサイズ及び通信スピードアップについて検討している。さらに、一般化メッセージ掘り下げ法の適用例として、配列の間接アクセスの通信に応用した場合について述べ、その効果を明かにしている。最期に、メッセージ掘り下げ法のコンパイラへの実装方法についてその概要を述べ、従来の FORTRAN D コンパイラなどの HPF 系コンパイラへの実装は容易であることを示している。

第 6 章では、本研究で明かにした知見を総括している。

2 背景

この章では、分散メモリ計算機の重要性を述べ、その発展には必要不可欠であるソフトウエアの標準化について述べる。最後に、コンパイラ技術の中心の一つであるプロセッサ間通信の最適化手法及びその問題点について述べる。

2.1 分散メモリ型並列計算機

科学技術計算を中心にしたアプリケーションが必要とする計算能力は、年々増加し、次世代のスーパーコンピューティング技術の重要性は高まる一方である。現在まで、図 2に示すように [61] 、その性能は、５年で約１０倍弱の性能向上の法則によって進歩してきている。

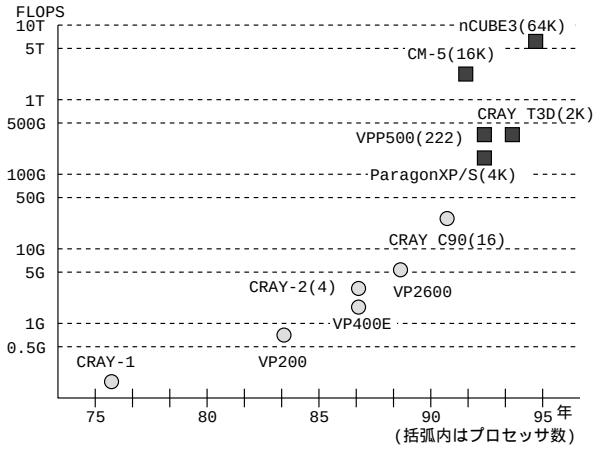


図 2: スーパーコンピュータのピーク性能

スーパーコンピューティングをささえる基礎アーキテクチャは、当初、演算パイプラインを持ったベクトル計算機 [66, 51] であり、その後、マルチプロセッサ型ベクトル計算機 [10, 58] 、分散—共有メモリ型並列計算機、現在は、仮想共有メモリ (実分散メモリ型) 並列計算機 [60, 59] となってきている。

ベクトル計算機に代わって、分散メモリを持つ並列計算機が主流となるのは、次のような理由により自然な流れと考えられる [62, 63] 。

- ベクトルアーキテクチャの持つ限界

メモリ: 前述のとおり、演算系が５年で１０倍弱の性能向上を達成しているにくらべ、メモリの高速性はそれほど達成されてはいない。したがって、複数のメモリユニットに同時にアクセスすることによって高速化することは実際的な解決策である。しかし、同時読み出しとともに、同時書き込みをハード／ソフトのいずれかで解決する必要がある。

演算パイプライン: ベクトル計算機は、一つの演算を複数のユニット (パイプライン) にわけ、そのユニットの処理する時間を最小化し、複数のデータ (ベクトル) を順次パイプラインに投入することによって、高速化を図ってきた。すなわち、どれくらい速くパイプラインにベクトルを投入するか (パイプラインピッチ) でその高速性が制限されていたのである。しかし、パイプラインが短くなるにつれて、それを実現するためのコストは高くなっている。たとえば、窒素やフロンで冷却するためのコストや高速なメモリをベクトルレジスタに使用する必要があるからである。また、パイプラインピッチを 1ns 以下にするのはコスト的に不可能であろうと言われている。

- 並列アーキテクチャとアルゴリズムの親和性

  従来、自然を扱う問題は、そのなかに並列性を持つことが多い。しかし、初期の計算機、フォンノイマン計算機は、そのアーキテクチャが逐次計算のみを行えるものであり、その後、計算機はこの流れに沿ったものが主流であったので、本来の並列性を持っているアルゴリズムも逐次なアルゴリズムに書き直されたのである。

最先端のスーパーコンピュータは、ベクトル計算機から分散メモリ型並列計算機に変わりつつあるが、まだ主流にはなりえていない。さらには、昨今、超並列と呼ばれるシステムを作ってきたメーカーの経営悪化により、並列処理全体の存続の危機が叫ばれている。これらの動向に対し、Rice 大学の Ken Keneddy 教授は次のようなコメントを述べている [18]。”TMC や KSR の脱落などから、業界全体に並列計算の将来を危惧する声が多い。理由の一つは、産業界の流れとしては、メインフレームから高性能ワークステーションで移行している。仮に大規模計算をしたいときは、ネットワークにつながった複数のワークステーションで計算すればいいと考えられている。もう一つの理由はソフトウエアの標準化がなされていないことから、ソフトウエアベンダの参入が遅れていることである。では、本当に並列計算は不要なのか？そう決めつけるには少し早すぎるのでないだろうか。まず、スケーラブルなソフトウエアの研究は重要で、ネットワークのシステムにでも有用なものである。さらに、PVM や HPF のような標準化がすすんでいるので、もう少し長い目で見なければいけない。”

上記の話のように並列処理は将来的には重要な技術である。ここで、並列処理の問題点を整理したい。

1. 実効性能の限界

   ベクトル計算機であれば、パイプラインを効率的に使用するようなプログラミングをした場合には効率的なプログラミングが可能であるが、並列計算機の場合は、分散メモリ間の通信というものが必要不可欠[1] であるので、このための時間が演算以外に必要となる。したがって、ピーク性能に近い性能を達成できるのは極めて稀だと考えられる。

2. ソフトウエア資産の継承性

   ベクトル計算機は、従来のフォンノイマン計算機とは異なるものであるとはいえ、その最適化コンパイラ (ベクトル化コンパイラ) により、従来のプログラムがそのまま使えるケースが多い。しかし、並列計算機ではプログラミングスタイルが大きく異なるので、大幅な書換が必要である。特に、仮想共有メモリ型でない場合は、プログラマがプロセッサ間通信に関するプログラミングもしなければならない。また、仮想共有メモリであってもプロセッサ間のデータ移動を少なくするために、データ分割を意識したプログラミングが必要となる。

3. 標準的なプログラミング環境の欠如

   ソフトウエアの継承が困難な状況に加え、新たにソフトウエアを作る際にも問題がある。つまり、並列計算機間で互換であるプログラミング環境がないので、マシンごとにプログラムを書き換える必要が存在する。

   つまり、プログラムの変更は容易ではなく、プログラマは並列プログラムを行う場合には、”アーキテクチャ”の選択と同時に”プログラミング環境の選択”という危険な行為を迫られるわけである。

ベクトル計算機に対しては、上記の第 2 の課題を克服するために、従来の逐次言語 (たとえば FORTRAN) をベクトル計算機むけにコンパイルする最適化コンパイラ [32, 44] 、すなわち、ベクトライザの研究が進み、自動的にプログラム変換する技術が存在している。

しかし、並列計算機、特に分散メモリ計算機においては、計算負荷の分散がデータ分散と密接な関係にあり、しかもそれがプログラムの詳細な知識や経験などに大きく依存するものである。つまり、最適なデータ分散は自動的には決定しにくいといわれている [2] 。

したがって、現在の研究の動向としては、第 3 の課題と合わせて、コンパイラに何らかの情報を与えられるような言語の拡張を行い、その言語に対するコンパイル技術の研究が中心となっている。このよ

[1] NAS カーネルの EP(Embarassingly Parallel) のような特殊な場合は、通信は不要であるが、一般のアルゴリズムでは通信が必要である。

うな言語として特に有名なのが、FORTRAN90 を元にした HPF(High Performance FORTRAN)[21] である。

言語の拡張の要点は、宣言したデータの分散形態をディレクティブとしてプログラマが与えられることである。コンパイラはこの与えられた情報を元に、プロセッサ間にいかに分散させるかを決定し、さらに必要であればプロセッサ間の通信を生成する。したがって、 HPF コンパイラにおいて重要な技術の一つは通信生成とその最適化である。

2.2 HPF 言語

前節で述べたように、分散メモリ並列計算機の発展にはソフトウエアの標準化が必要不可欠である。この節では、プログラミング環境に関する標準化、特に HPF 言語について紹介する。

分散メモリ計算機におけるプログラミング言語の標準としては 2 つの方向が存在する。一つは、従来の言語を用いて SPMD(Single Program Multiple Data) プログラムを直接書き、プロセッサ間通信などの分散メモリ計算機特有な動作に関する関数を用意しておくものである。もう一方は、従来言語を拡張して、新しい言語を設計するものである。後者は、従来の逐次計算機でもコンパイル可能であるが、前者は不可能であり、プログラミングも複雑になる。しかし、一般に、前者の方式でプログラミングするほうが実行速度は速い。

前者の例としては、本来はヘテロ (Heterogeneous) 環境におけるプログラミング環境であった PVM[11] や、分散メモリ計算機におけるメッセージパッシングの標準となる MPI (Message Passing Interface)[33]、Linda[7] を FORTRAN や C 等の言語とともに用いる方法がある。また、後述する ADETRAN や前述した HPF が後者の代表例である。

Ian Foster[9] によると、前者をライブラリベースアプローチ (library based approach) 、後者を言語ベースアプローチ (language based approach) と呼び、プログラムにおける並列性を基本的なファクタと見るか副次的なファクタと見るかによってによって、いずれのアプローチが適当であるかが判断できると述べている。すなわち、逐次プログラミング言語の場合、データ構造と制御構造は基本要素だと考えられ、言語の中に反映させられてきたが、入出力は副次的なものということで、ライブラリとして扱われていた。本論文の次章以降では、並列性は言語の中に採り入れるべきものだと考え、ADETRAN や HPF を中心とした言語ベースアプローチのみについて議論する。

2.2.1 概要

HPF は、多くの FORTRAN の方言、すなわち、FORTRAN8X, FORTRAN90[1], Message Passing library, CM-FORTRAN[39], FORTRAN-D[15, 13], Vienna FORTRAN[46], Yale extension, DEC proposal などを統合し、分散メモリ計算機の標準プログラミング言語の構築を目指して欧米の研究機関を中心メンバーとして議論された。HPF の特徴は、FORTRAN 言語に対する拡張部分として、コンパイラにデータのローカリティに関する情報を与えていることである。

現在は、第 1 版の仕様が決まり、それに基づいたいくつかの処理系が発表されており、また、第 1 版への追加仕様の検討 (第 2 版) が進んでいる状況である。

以下では、第 1 版の言語仕様の主要な点、及び、第 2 版で問題となっている点について述べる。また、HPF に近い文法を持つ ADAPTOR システム及び最初の商用 HPF システム APR/xHPF の特徴を付録にまとめる。いずれのシステムも比較的初期のものであり、コンパイラの最適化機能は弱い。特に、通信レイテンシ最適化に関してはほとんど実装されていない。

2.2.2 データ分散 (data distribution)

HPF の基本概念であるデータ並列とは、オペランドが同一プロセッサにあれば、計算を 1 つのプロセッサ上でできるようにし、独立な計算はできるだけ異なるプロセッサで行なうようにすることである。

データの分散を指示するためのディレクティブは大きく分けて 4 つである。すなわち、プロセッサ形態を与える PROCESSORS 、仮想プロセッサとの整列化 (align) を指定する ALIGN 、PROCESSORS で指定されたプロセッサへ仮想プロセッサを分散指定する DISTRIBUTE 、メモリ領域を使用しない仮想プロセッサの鋳型としての TEMPLATE である。

- *PROCESSORS* ディレクティブ
  PROCESSORS は、プロセッサのサイズ、形状を指定する。

```
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS()), &
!HPF$            R(8,NUMBER_OF_PROCESSORS()/8)
```

  P は、intrinsic 関数 NUMBER_OF_PROCESSORS() の返す総プロセッサ数からなる一次元配列のプロセッサ形状を指定し、R は、 2 次元を指定する。

- *ALIGN* ディレクティブ

配列を、仮想プロセッサにどう割り当てるかを指定する。仮想プロセッサとは、配列もしくは TEMPLATE である。

```
!HPF$ ALIGN X(:,*) WITH D1(:)
!HPF$ ALIGN Y(:,:) WITH D2(:,*,:)
!HPF$ ALIGN Z(J,K) WITH D3(K-1,J+1)
```

配列 X は縦を D1 と align し、横は非分散 (collapse) とする。配列 Y は第 1 インデックスと第 2 インデックスを D2 と整列化し、D2 の第 2 インデックスに関してコピーを行なう。配列 Z は D3 の転置でかつシフトした位置に整列化される。

- *DISTRIBUTE* ディレクティブ
  仮想プロセッサに対し、実プロセッサにどう割り当てるかを指定する。

```
!HPF$ DISTRIBUTE D3(BLOCK,CYCLIC) ONTO R
!HPF$ DISTRIBUTE D2(BLOCK,*,BLOCK)
```

D3 は、縦をブロック分散し、他方をサイクリック分散する。また、D2 は、第 2 インデックスを分散しない、すなわち、プロセッサ内に非分散 (collapse) する。

これら以外に、実行時に分散形態を変更する、redistribute/realign がある。

2.2.3 実行文

ここで重要なのは、FORALL 文である。HPF は、FORTRAN90 にディレクティブを追加することによって、プログラムのセマンティックスは保存したまま、コンパイラにデータのローカリティを指示するように設計されているが、FORALL に関しては、do 文などとは意味が異なるので、新たな構文追加にしてある。

```
FORALL(I=1:N,J=1:N,A(I,J).NE.0.0) B(I,J)=1.0/A(I,J)
```

上記の例は、配列 A がゼロでない時、配列 B に、その逆数を代入するというものである。演算順序としては、

1. FORALL の実行範囲全部に対するマスク (配列 A) の評価
2. FORALL の実行範囲全部でマスク値が真のものに対して、右辺の評価

3. FORALL の実行範囲全部でマスク値が真のものに対して、左辺の index の評価

4. FORALL の実行範囲全部でマスク値が真のものに対して、代入

となる。すなわち、すべての評価が終るまで、代入は全くされない (Copy-in/Copy-out) 。

```
FORALL(I=2:N-1) A(I)=A(I-1)+A(I)+A(I+1)
```

上記の例は、先ほどの演算順序のからいうと、すべての old 値を用いて計算した後、代入することになる。

また、FORALL の construct もある。

```
FORALL(I=1:N,J=1:N)
 B(I,J)=A(I,J)*C(I,J)
 D(I,J)=B(I,J)+2.0
END FORALL
```

FORALL construct は、単一代入 (single assignement) FORALL が複数列挙されているものの簡略形と考えられる。したがって、上記の例は、

```
FORALL(I=1:N,J=1:N) B(I,J)=A(I,J)*C(I,J)
FORALL(I=1:N,J=1:N) D(I,J)=B(I,J)+2.0
```

と書き換え可能であり、 2 つめの右辺の配列 B は、更新された値を用いる。

2.2.4 第 2 版に向けて

Supercomputing'93 において第 1 版の仕様がほぼ確定した後、第 1 版では含むことのできなかった部分が順次議論されている。項目としては下記の 3 つである。

- 不規則 (irregular) 通信を含む演算

  第 1 版における配列記述 (array notation) や FORALL 文は、規則正しい (regular) 計算の記述は容易であり、コンパイラも効率のよいコードを生成することが可能であった。しかし、下記のような間接アクセスを含むコードは通信発生を伴う場合、コンパイラにおいて必ずしも効率のよいコード生成はできない。

  ```
  !HPF$ DISTRIBUTE (BLOCK) :: x(10),L(10),Y(10)
  ```

```
DO I=1,10
 X(I)=Y(LL(I))
END DO
```

そこで、コンパイラになんらかの情報を与えて通信生成を容易にすることが必要である。

- タスク並列
  第 1 版では、データ並列をもつプログラムの並列記述を主な目的としていた。しかし、マルチメディア関連などを始めとした、異なるモジュール間の協調の元に計算が進んでいくようなアプリケーションを記述するためには、タスク並列 (task parallel) を扱えるような記述方式が必要である。

- 並列入出力
  第 1 版仕様策定時にも議論されていた項目であるが、第 1 版では仕様に入れることができなかった。というのは、従来の FORTRAN 同様、入出力はライブラリ的な扱いになるのであるが、既に各ベンダが独自に並列入出力ライブラリを持っており、それらはハードウエアに依存している部分が多く、限られた期間の間に統一的な仕様を決めることができなかった。しかし、入出力は重要な要素の一つであるので、再度統一仕様策定にのぞんでいる。

2.3 通信レイテンシ最小化手法

前節で述べた HPF 言語の実装において、効率の良い通信生成が特に重要である。つまり、プロセッサ間通信をいかに効率よく行なうかがハードウエア資源の最適利用のキーとなる。この節では最適通信生成の中でも通信起動コスト最小化技術と並んで重要である、通信レイテンシ最小化技術の問題点について述べる。分散メモリ計算機のプロセッサ間の通信コストは大きく分けて 2 つの部分からなる。すなわち、通信起動と通信レイテンシである。集団通信 (collective communication) でない場合、一般に通信は、

1. 送り先 (single destination)
2. メモリの番地 (address)
3. バイト数 (byte)

のパラメータからなるメッセージ毎に送受信される。つまり、送り元の送るデータのアドレスがとびとび (非連続) になっている場合は、別々のメッセージになり、また、送り先が異なるプロセッサの場

合も別のメッセージとなる。なお、昨今の通信ライブラリ、例えば PVM や MPI では、これらのメッセージを 1 つの関数で扱える場合もあるが、ハードウエア実現のレベルでは個別のメッセージに展開されるのが一般的である。

通信起動は、このメッセージの数に比例したコストがかかり、通信レイテンシはバイト数に比例したコストがかかる。したがって、前者のコストを軽減するには、メッセージの数をいかに減らすかがキーとなり、後者のコストを減らすには、冗長なデータの通信を排除し、計算部分に通信を隠蔽させることになる。

以下では、通信レイテンシ最適化に関する、反復組み替え法 (iteration reordering)[19]、ベクタメッセージパイプライン法 (vector message pipelining)[35, 40]、および、ハードウエア支援による通信隠蔽法について説明する。なお、通信起動最適化に関する、メッセージベクトル化法 (message vectorization)[3]、メッセージ合体法 (message coalescing)[40]、メッセージ集団化法 (message aggregation)[40]、検査官/執行官法 (inspector/executor)[20, 23, 22] については付録にまとめる。

反復組み替え法　計算部分がプロセッサ内のデータだけを用いるローカル計算部分と他のプロセッサが保持するデータをも利用するリモート計算部分に容易に分割できる場合に、反復組み替え法 (iteration reordering) は効果的に適用できる。ローカル計算部分は他のプロセッサのデータを必要としないので、リモート計算部分で使用するデータの通信をローカル計算部分と並行して行うことができる。したがって、この通信はローカル計算部分に隠蔽される。

ここで、プロセッサ数が 4 つの場合について、反復組み替え法を例を用いて説明する。図 3の左図の上に示してあるプログラムについて考える。

配列 A と B は (*, BLOCK) で分散されている。よって、プロセッサ 0 以外のプロセッサは、隣接するプロセッサから配列 B の境界値を供給される必要がある。プログラムは左図の下に示す SPMD プログラムにコンパイルされる。

しかし、計算部分がローカル計算部分とリモート計算部分に容易に分割できるので、右図に示すように、配列 B の境界値の通信はローカル計算部分とオーバーラップできる。

ベクタメッセージパイプライン法　ベクタメッセージパイプライン法 (vector message pipelining) では、送信すべきベクタデータがメモリに格納されたすぐ後に送信プロシージャの起動を行い、そのベクタデータが使用される直前に受信プロシージャを起動する。この 2 つのプロシージャの間に十分な計算部分があれば、通信レイテンシはその計算に隠蔽できる。

ベクタメッセージパイプライン法の動作を説明するために、図 4 の左図に示しているプログラムに

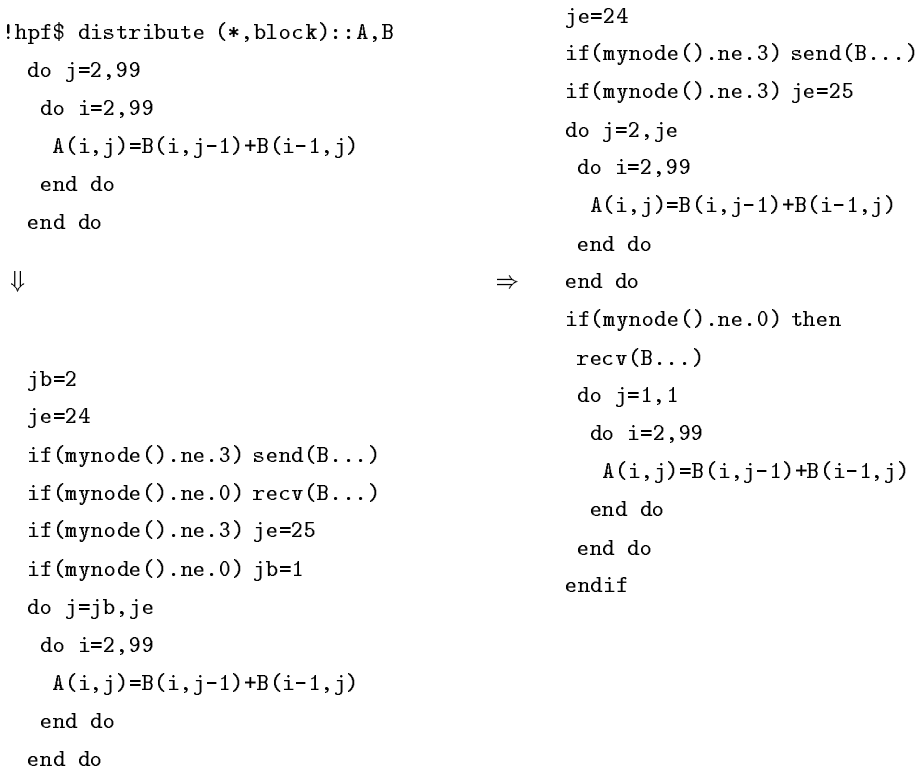


図 3: 反復組み替え法 (Iteration reordering)

ついて考える。

ループ 1 および 2 において、配列 B は、チェッカーボード方式 (Red-Black manner)　で定義される。すなわち、ループ 1 では、j インデックスが奇数となる点が配列 A の値で定義され、ループ 2 では、j インデックスが偶数となる点が配列 C の値で定義される。ループ 3 の計算は j インデックスが偶数となる点で実行される。また、奇数の j 点での値を使用し、これらの点は多くは使用されるのと同じプロセッサ上で定義されているが、境界については隣接プロセッサ上で定義されている。よって、さきほどの例と同様に、配列 B の境界値を通信しなければならない。

配列 B の通信すべき境界値はループ 1 で定義され、ループ 3 で使用されている。したがって、ループ 1 が終了した直後に境界値を送信し、ループ 3 の直前で受信すれば、通信レイテンシはループ 2 の計算部分に隠蔽される。

```
    dimension(80,80)::A,B
!hpf$ distribute (*,block)::A,B
    do 1 j=1,79,2
     do 1 i=1,100
 1    B(i,j)=A(i,j)
    do 2 j=2,80,2
     do 2 i=1,100
 2    B(i,j)=C(i,j)
    do 3 j=2,80,2
     do 3 i=1,100
 3    D(i,j)=B(i-1,j)
```

```
    do 1 j=1,19,2
     do 1 i=1,100
1     B(i,j)=A(i,j)
    if(mynode().ne.3) send(B...)
    do 2 j=2,20,2
     do 2 i=1,100
2     B(i,j)=C(i,j)
    if(mynode().ne.0) recv(B...)
    do 3 j=2,20,2
     do 3 i=1,100
3     D(i,j)=B(i,j-1)
```

図 4: ベクタメッセージパイプライン法 (Vector message pipelining)

ハードウエア支援　通信と計算を行うハードウエアが独立に存在し、それぞれの動作をオーバーラップすれば、通信レイテンシは隠蔽される。ここで、通信は大規模な規則正しいベクタを送るものと仮定し、定義する計算部分および使用する計算部分はベクタ全体に対する計算であると仮定する。このとき、オーバーラップの仕方としては、次の 3 パターンが考えられる。

1. 定義する計算部分と送信部分をオーバーラップする
2. 定義部分と使用部分の間の計算部分とオーバーラップする
3. 使用する計算部分と受信をオーバーラップする。

2 番目の方法は、反復組み替え法やベクタメッセージパイプライン法で用いていたケースであるが、後述するように、いくつかの制約があり、通信を隠蔽するのに必ずしも十分な計算部分が存在しない場合もある。そこで、1 番目か 3 番目の隠蔽方法をとる場合も考えるが、普通のプロセッサ構成では実現が困難である。

1 番目の場合を例にとって、ハードウエアを追加したプロセッサでの実現について述べる。下記のプログラムを考える。

```
!HPF$ DISTRIBUTE(BLOCK)::x(10),y(10)
      FORALL(I=1:10) Y(I)=I*1.0
      FORALL(I=2:10) X(I)=Y(I-1)
```

配列 Y の境界値は 2 番目の FORALL 文で使用され、1 番目の FORALL で定義されている。プロセッサ機能として次の動作を加える。

計算プロセッサ： メモリに格納するとともに、通信プロセッサに信号を送る機能

通信プロセッサ： 計算プロセッサから信号を受けとる毎に、あらかじめ決まった順にメモリから読みだし、送信する機能

つまり、計算プロセッサは 1 番目の FORALL 文を実行する前に、あらかじめ通信プロセッサに決まった手順を伝えておき、分散された配列 Y をメモリに格納する毎に信号を通信プロセッサに送る。通信プロセッサは、計算プロセッサからの信号に従い、境界値となる配列要素が格納される時に通信を 1 ワードだけ行う。このようにすると、計算部分が境界値以外の計算を行っている間に通信を完了することができるので、通信レイテンシは原理的にはすべて隠蔽される。

3 番目の隠蔽方法についても同様な議論ができるが、1 番目の方法との得失については後述する。また、1 番目の方法でかつ配列の再分配 (redistribution) に限定して実現した例として、並列計算機 ADENA の TCU の例がある。これについても後の章で詳しく述べる。

3 ADETRAN コンパイラ

ここまで述べてきたように、分散メモリ計算機の普及のキーは、標準プログラミング環境とそれを効果的に実装する技術を確立することにある。特に、通信最適化は分散メモリ計算機において最も重要な項目である。しかし、付録でもふれているように、ADAPTOR システムを初めとする既存のインプリメンテーションにおける通信最適化の実装は不十分なものである。それらをふまえ、この章では、並列計算機 ADENA において実現した通信最適化およびメモリ操作の最適化について述べる。

並列計算機 ADENA は、京都大学工学部野木助教授の発案による並列計算機アーキテクチャ[26, 27]であり、1980 年代中ごろより京都大学と松下電器とで試作機を共同開発してきた [16, 29] 。

ADENA システムの設計方針は次のように考えられた。

計算方式 多次元配列を仮想プロセッサ上に分散させ、各仮想プロセッサ上では 1 次元配列として演算し、他の分散形態に再分散することを基本とする。(ADEPS と呼ぶ [30])

ネットワーク 分散された配列を異なる分散形態に高速に再分散するネットワーク。

通信ハードウエア 通信コストを軽減するために、集団通信 (collective communication) をハードウエアレベルでサポートし、通信を計算とオーバーラップさせるためのハードウエアを用意する。

一方、従来の並列計算機上のプログラミング環境として、" ユーザーアプリケーションの記述性が高いこと" と" アーキテクチャの特性を生かすこと" の 2 つの要求を融合させた専用言語 (C*[36],MPC[8]) を用いたものがある。

また、既存の言語 (FORTRAN 等) から並列性を抽出する処理系を用いるものや、既存言語に若干の拡張を行ない、特定のアーキテクチャを目的としない汎用並列言語 (C-LINDA 等) をめざすものがある。

後者は既存のソフトウエアの互換性及び将来へのソフトウエアの継承性の観点から見ると重要な方式であるが、現時点ではその並列抽出効率及び演算実行時の性能を考えると実用には向いていない。従って、ADENA では前者の方法、すなわちアーキテクチャに適した専用言語を用いるプログラミング環境を目指し、京都大学が主となって設計された言語 ADETRAN[28, 57] に対しコンパイラの実装を行なった [45, 54, 52, 53] 。

ADENA は、数値シミュレーションを主なアプリケーションを想定した並列アーキテクチャである。プログラミング言語の条件は次のように考えられた。

FORTRAN タイプの構文: 現在のベクトル計算機のユーザは、FORTRAN でプログラミングをしており、これらのユーザが比較的容易に ADENA を使用するためには必要である。

ADENA のアーキテクチャを自然に取り込む: コンパイラの実装を容易にし、かつ、実行時のオーバーヘッドを削減できる反面、若干ではあるが、プログラマにアーキテクチャを意識したプログラミングをすることを要求する。これは、ADENA が高速計算を第一の目的におく場合不可避な条件である。具体的には、プロセッサ内のデータ構造を示すスラッシュ('/') を用いた配列表現、並列実行を示す pdo 文、プロセッサ間のデータ転送を表現する pass 文等がこれにあたる。

数値シミュレーションに適した構文を持つ: 従来の FORTRAN にはないが、アプリケーションを考えた場合採り入れる必要のある構文である。具体的には、行列計算用の pad 文、及び pour 文、粒子シミュレーション用 file 配列及びそのための実行文である。

そこで、上記の用件を満たす言語 "ADETRAN" を京都大学が中心となって設計され、松下電器において、著者が中心となって、その言語に対するコンパイラを実装した。この章では、ADENA システムの概要、ADETRAN 言語の設計方針及び実装方針、さらにコンパイラで用いた最適化手法 [52, 53] について述べる。

3.1 ADENA システム

ADENA システムは、256 個のプロセッサエレメントとそれらをつなぐネットワーク (HX ネット:Hyper-crossbar) とからなる ADENA と、汎用ワークステーションをホスト計算機とするバックエンド型の計算機構成である (図 5) 。

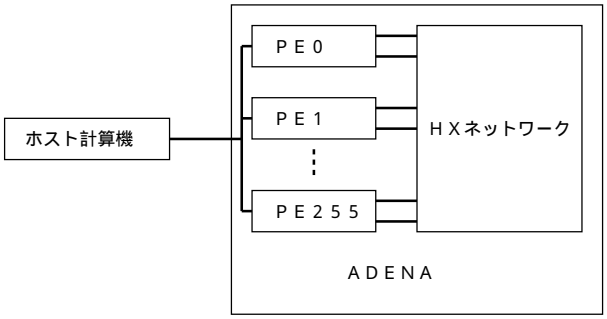


図 5: ADENA システム構成

プロセッサエレメント (以下プロセッサと呼ぶ) は、ピーク性能 20MFLOPS の浮動少数点プロセッサ EPU(Element Processing Unit)[17] とデータ転送用コントローラ TCU(Tramsfer Control Unit)[24] 及

び命令メモリ LIM(Local Instruction Memory:24bit × 64Kwords) とデータメモリ LDM(Local Data Memory: 72bit × 256Kwords) からなる (図 6) 。

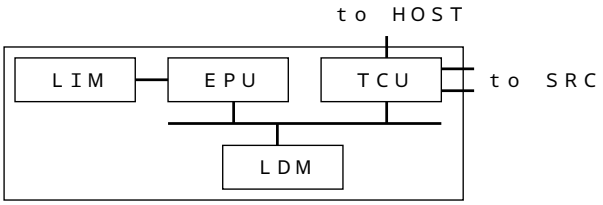


図 6: プロセッサ構成

EPU は内部に 64 ビット浮動少数点加減算器及び乗算器を備え、64 ビット浮動少数点レジスタを 32 本、32 ビット整数レジスタを 32 本持つ。さらに、プロセッサを仮想化するために、多重度制御機構を持つ。

TCU は、1) ホスト計算機とのデータ転送と、2)SRC(Send Receive Controller) と LDM とのデータ転送を行なう。SRC と LDM とのデータ転送は、HX ネットを使うデータ転送の時に使用される。TCU と EPU は、LDM を共有メモリとして相互にアクセスする。

実行時には、EPU の演算実行と並行して TCU のデータ転送が可能である。通信で送信される配列が演算の中で代入されることによって依存関係がある場合には、通信のオーバーラップは不可能であったが、ADENA では S スキームモードにより可能である。S スキームモードとは、EPU が演算結果を配列データとして LDM に格納し、そのデータを TCU がデータ転送する場合、EPU の LDM へのデータ格納と同時に TCU がそのデータを転送開始する。このモードに関しては後で詳述する

3.1.1 HX ネットワーク

プロセッサ間は、HX ネットで接続されている。このネットワークはバースト転送の制御を行なう SRC とクロスポイントの FIFO メモリである BMU(Buffer Memory Unit) よりなる。HX ネットの論理的な接続は、各プロセッサのプロセッサ番号を $/j,k/(j,i = 0 \cdots 15)$ とし、BMU の番号を $/i,j,k/(i,j,k = 0 \cdots 15)$ とする時、

$$PE/j,k/ \rightarrow BMU/i,j,k/ \rightarrow PE/k,i/$$

となっている。この接続は双方向になっており、

$$PE/i,j/ \rightarrow BMU/j,k,i/ \rightarrow PE/k,i/$$

と見ることもできる。

このネットワークの構成は、配列の再分散に適したものである。次の例を考えよう。

```
      REAL*8 D(16,16,16)
!HPF$ DISTRIBUTE D(*,CYCLIC,CYCLIC)
     :
!HPF$ REDISTRIBUTE D(CYCLIC,*,CYCLIC)
```

前述のように ADENA では ADEPS プログラミングモデルに基づき、多次元配列の 1 次元を仮想プロセッサ上に連続に配置 (collapsed) し、他の次元を分散させることを基本とし、演算中に上記のような再分散が必要となる。

初期の分散では D(i,j,k) はプロセッサ/j,k/の i 番目の配列要素として保持されているが、再分散後はプロセッサ/k,i/の j 番目の配列要素となる。一方、HX ネットワークではプロセッサ/j,k/は BMU の/i,j,k/ を介してプロセッサ/k,i/と接続されているので、この通信は高速に処理できる。他の分散パターンも同様に効率的に処理でき、また 2 次元配列においても 3 次元配列の再分散を組み合わせることにより実現できる [49] 。

TCU は、従来の 1 対 1 通信の機能の他に、この再分散通信を集団通信 (collective communication) として処理する機能を持っているので、再分散通信を高速に行うことができる。また、最小限の集団通信に限定したことにより、ハードウエアでの実現が可能となった。

3.1.2 S スキームモード

ADENA において通信を行うのは TCU である。TCU には通信コストを低減する機構を備えている。従来の DMA の機能は、任意の指定したディスティネーションとメモリの間を連続転送するものであったのに対し、TCU には次のような機能が含まれている。

- 複数ディスティネーション間の転送が一回のコマンドで起動できる。
- EPU からのシグナルにしたがって、連続転送をワード毎で同期をとれる。(S スキームモード)

ADENA では通信は基本的に配列の再分散であるので、ほとんどの通信は多対多通信である。したがって、第 1 の機能より配列の再分散を 1 回のコマンド設定で起動可能であり、通信起動コストの軽減が可能である。

また、第 2 の機能 (S スキームモード) により、通信レイテンシの削減も可能である。例えば、下記のようなプログラムを考える。

```
 do 10 i=1,10
  x(i)=....
10 continue
 call send(x(1:10))
```

上記のプログラムにおいて配列 x は do ループ 10 で定義されているので、ループ 10 が終了するまで通信起動は行えない。しかし、実際は x(1) の定義が終了した時点で x(1) の通信は起動できる。そこで、次のような書き換えが可能である。

```
 do 10 i=1,10
  x(i)=....
  if(i.ne.10) call send(x(i:i))
10 continue
 call send(x(10:10))
```

このようにすれば x(i+1) の定義と x(i) の通信がオーバーラップできるので通信レイテンシの隠蔽が可能である。しかし、通信起動が増えそのために通信コストが大きく増大することが考えられる。

TCU のＳスキームモードはこのデメリットを避けたもので、次のような動作になる。

```
 call s_send(x(1:10))
 do 10 i=1,10
  x(i)=.... with signal
10 continue
```

つまり複数ワード分の通信起動を代入文の前に発行しておき、定義においては EPU から TCU をシグナルを送るようにする。TCU はシグナルを受ける毎に 1 ワードずつ通信を行う。したがって、通信起動は最初に 1 回行うだけですみ、また、x(i+1) の定義と x(i) の通信がオーバーラップするので、通信レイテンシの多くは隠蔽することができる。

したがって、ADENA においては配列再分散の処理が高速であるので、コンパイラとしてはできるだけ通信を再分散通信で処理するようにし、また、S スキームモードを利用することが肝要となる。

3.2 ADETRAN の設計方針

ADETRAN は次のような特徴を持つ言語として設計された [28, 57]。

- ADENA では配列の再分散が高速に実行できるので、それを通信の基本とする。
- FORTRAN 言語に近い文法を持つ

そこで、ADETRAN は、数値シミュレーションを主アプリケーションをする FORTRAN ライクな高級言語として定義し、FORTRAN の基本文法に次の構文が付加された [28, 57, 45, 54]。並列動作を記述する構文として pdo 文、プロセッサ間通信のための構文として pass 文、プロセッサ間の同期のための構文として ifall/ifany 文を基本構文とする。また、幅広い数値計算を扱うために、任意長データ構造を表す file 配列、2 次元配列用のマルチキャストを行なう、pad/pour 文が導入された。

以上のように ADETRAN は ADENA との親和性の良い言語であり、ADENA へのコンパイラ実装を容易にする。また、ADENA 向きに書かれた FORTRAN 言語の移植の負荷も低くでき、FORTRAN プログラムを ADETRAN プログラムに変換するトランスレータの実現 [50] を可能とした。

3.3 ADETRAN の概要

3.3.1 基本データ構造

ADENTRAN では、配列データを操作することを基本としているが、その配列のインデックスにスラッシュを挿入し、担当プロセッサを明示する。例えば、 3 次元配列 a は FORTRAN では、

```
a(i,j,k)
```

のように表すが、ADETRAN では、

```
a(i,/j,k/)
```

のように表す。a(i,/j,k/) は x 方向配列といい、プロセッサ/j,k/が持つ配列 a の i 番目のデータを表し、a(i/,j,/k) は y 方向配列といい、プロセッサ/k,i/が持つ配列 a の j 番目のデータを表し、a(/i,j/,k) は z 方向配列といい、プロセッサ/i,j/が持つ配列 a の k 番目のデータを表す。

このように、 3 次元配列は、そのうちの 2 次元をプロセッサアレイにマッピングし、残りの 1 次元を各プロセッサ内で扱う。同じく 2 次元配列は、1 次元をプロセッサアレイにマッピングし、残りの 1 次元を演算で扱う。

2 次元配列の場合もほぼ同様の記述となり、例えば x 方向配列は a(i,/j/) のように表し、2 次元プロセッサ空間を 1 次元空間に写像したプロセッサ空間上のプロセッサ/j/上の 1 次元配列として定義される。同様に y 方向配列は a(/i/,j) と表される。

3.3.2 region 文

ADENA では、プロセッサの物理的個数を越えた並列動作記述が可能であるが、その場合の最大プロセッサ空間のサイズを示すのが region 文である。

```
region(21,32,43)
```

と記述した場合、x 方向配列の場合、最大 32x43 の仮想プロセッサ空間をとることを表す。

つまり、物理的には 16x16 のプロセッサしか持たないシステムに対して、各プロセッサが 2x3 個の仮想プロセッサをエミュレートすることにより、システム全体として 32x43 の仮想プロセッサを持つことになる。

3.3.3 pdo 文

基本データを操作するのが pdo(Parallel DO) 文である。前述の通り、ADETRAN は FORTRAN を基本とした言語であり、その並列処理単位はループである。したがって、並列に処理すべき DO ループを pdo とし、そのループのインデックスは、仮想プロセッサ番号とする。次の例を見よう。

```
      pdo j,k=1,N
       do 10 i=1,N
        a(i,/j,k/)=i*1.0+b(i-1,/j,k/)
10     continue
      pend
```

スラッシュで囲まれたインデックスがプロセッサ番号に相当するので、各プロセッサが並列かつ独立に動作するためには、囲まれたインデックスは揃っていなければならない。この例では、各プロセッサが、 do ループ 10 を並列に実行する。同様に、

```
      pdo k,i=1,N
       do 20 i=1,N
        a(i/,j,/k)=a(i/,j,/k)+j*1.0
20     continue
      pend
```

と記述できる。これは、先ほどの例とは異なり、/k,i/がプロセッサ番号を与え、do ループ 20 を並列に実行する。

3.3.4 pass 文

前節の 2 つの pdo 文を連続して記述した場合、先の pdo 文で定義された配列 a はプロセッサ/j,k/で定義されるが、次の pdo 文で参照される配列 a はプロセッサ /k,i/である。したがって、配列データ a をプロセッサ/j,k/ から /k,i/へ転送する必要がある。これを記述するのが pass 文である。すなわち、

```
pass i,j,k=1,N
 a(i/,j,/k)=a(i,/j,k/)
pend
```

である。

前述の通り、ADENA においては配列の再分散を通信の基本とし、それを高速に処理するためのハードウエア構成を用意している。この通信は、送り元プロセッサは /j,k/ であり、送り先は /k,i/である。これは、HX ネットワークの構成である、

$$PE/j,k/ \rightarrow BMU/i,j,k/ \rightarrow PE/k,i/$$

と同じである。つまり、HX ネットワークによって、3 次元配列の pass 文は効率良く転送できる。 2 次元配列の場合は、 3 次元配列の再分散通信を組み合わせることにより、同様に効率良く再分散通信ができる。

3.3.5 ifall/ifany 文

並列に動作するプロセッサ間でその演算結果に従って制御フローの変更を行なう場合がある。例えば、収束するまで繰り返し計算を行なうプログラムなどで収束判定を行なう場合である。その場合、各プロセッサは論理型配列に収束したか否かの情報を代入し、それの論理積もしくは論理和で制御を変える。これを記述するのが、ifall/ifany 文である。例えば、論理型配列 ok に収束情報を代入し、

```
ifall(ok(1,/j,k/),j,i=1,N) goto 1000
```

と記述すれば、全てのプロセッサで論理値 ok が正であればラベル 1000 へジャンプする。また、

```
ifany(ok(1,/j,k/),j,i=1,N) goto 2000
```

と記述すれば、一つ以上のプロセッサで論理値 ok が正であればラベル 2000 へジャンプする。

3.3.6 サブルーチン

ここまで述べた構文が ADETRAN の基本構文であり、多くのプログラムはこれらの組合せで記述できる。しかし、プログラムのモジュラリティを高めるためには、FORTRAN 同様にサブルーチン構造を持つ必要がある。サブルーチンの階層を図 7に示す。

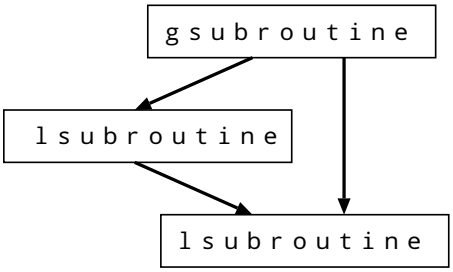


図 7: サブルーチン階層

gsubroutine は ADENA 全体の動作を記述するためのサブプログラムであり、ホスト上で実行されるメインプログラムから呼ばれる。一方、lsubroutine は各プロセッサ内で実行されるプログラムであって、プロセッサ間通信や並列処理は記述しない。lsubroutine は、lsubroutine 及び gsubroutine のいずれかからも呼ばる。並列記述できるのは、gsubroutine だけである。

3.3.7 pad/pour 文

ここまで述べた構文を使うことによってほとんどの計算は記述可能であるが、一部の行列計算を記述するための若干の構文を追加する。

ガウス消去を用いて逆行列計算を行なう時に、各プロセッサに分散しているデータを集めて一つの配列にして全てのプロセッサに再分配する場合 (ピボットの選択) がある。これを表すのが pour 文である。

また、あるプロセッサの持っているデータを全プロセッサに分配する必要がある場合 (ピボット行の分配) に用いられるのが pad 文である。これらを図 8 に示す。図 8 の上のプログラムにおいて、部分配列 `b(/j/,2)` はプロセッサ上で分散されている。この部分配列を全プロセッサにコピーするために pour 文が用いられ、プロセッサ/i/上の配列要素 `a(/i/,j)` にはプロセッサ/j/の保持していた `b(/j/,2)` が代入される。また、下のプログラムにおいては、プロセッサ/2/ の保持している部分配列 `b(/2/,j)` が pad 文により全プロセッサにブロードキャストされ、`a(/i/,j)` に代入される。

pour/pad 文は TCU による通信と EPU による演算の組み合わせによって実現できるので、後述のようにコンパイラによって pass 文と pdo 文の組み合わせに展開される。

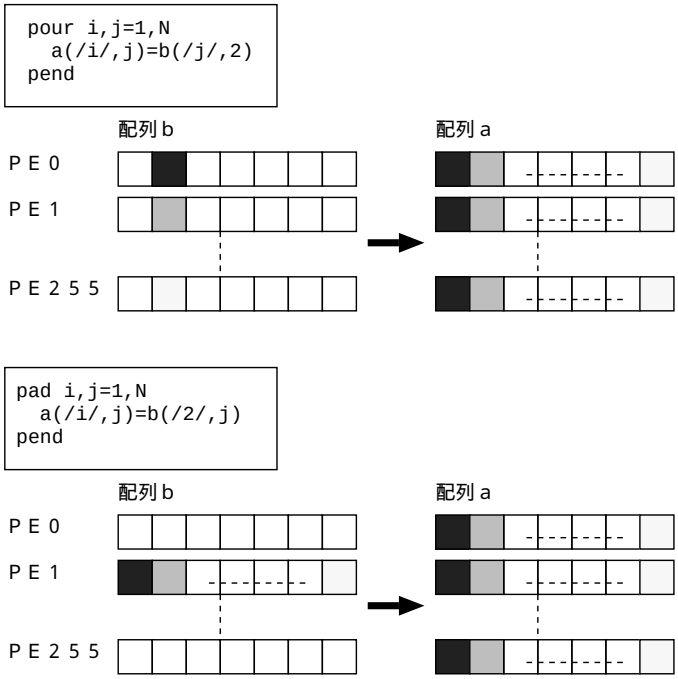


図 8: pour 文及び pad 文

3.3.8 file 配列

規則的な配列 (structured mesh) を用いるアプリケーションに対しては、ここまでの構文によっておおむね記述可能である。しかし、不規則配列を用いるアプリケーションに対しては記述困難なものもある。例として粒子プラズマシミュレーションを用いて説明する。

粒子プラズマモデルを用いたプラズマシミュレーションは、次の 2 つのフェーズを繰り返して計算を進める。

1. プラズマ粒子の位置より電場を計算する。

2. 電場を用いて各粒子にかかる力を求め、粒子の位置を進める。

これを並列計算する場合、空間メッシュの各格子点を並列化の単位とし、その格子点に影響を与えるプラズマ粒子をその格子点を担当するプロセッサに持たせるようにする。従って、各格子点の電場

を計算し、求まった電場を補間することにより、各粒子にかかる力は求まる。それを用いて次のステップでの粒子の位置が求まり、新たにその粒子を担当するプロセッサが決まる。よって、そのプロセッサへ粒子情報を転送することになる。

これらを実現するデータ構造が file 配列である。file 配列は順アクセスのみ可能な任意長のレコードへのポインタの配列である。つまり、複数情報からなる粒子情報をレコードとして持ち、そのレコードは各点に対して、任意個持つことができる。例を下に示す。

```
file/pp(//,)/ii,jj,uu,vv
```

上記の例は、整数フィールド ii,jj 、実数フィールド uu,vv を持つレコードを任意個蓄えられる配列を示す。例えば pp(/1/,1) はプロセッサ/1/が持つ file 配列のポインタであり、このポインタが指すレコードは 4 ワードからなる。また、このポインタからは任意個のレコードにアクセス可能である。図 9に、その概念図を示す。

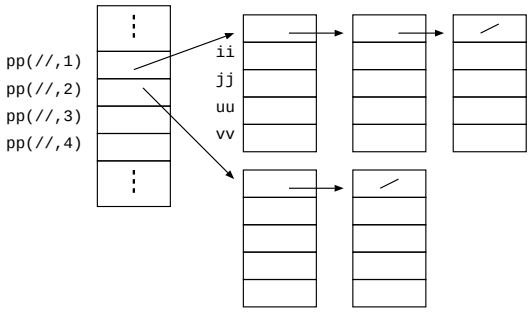


図 9: file 配列の概念図

これらのポインタとレコードを扱う構文として 4 つの実行文と通信文がある。ポインタの内容を無効化し、file 配列を再書き込み可能とするのが rewrite 文であり、file 配列のポインタを先頭のレコードに設定し、レコードの再読みだしを可能とするのが reset 文である。さらに、新たなレコードを file 配列のポインタに追加するのが put 文であり、file 配列のポインタを先に進め、次のレコードの読み出しを行なうのが get 文である。また、 push 文は file 配列を他のプロセッサへ転送するために用いる。

file 配列は粒子シミュレーションを行なうために設けられた構文群であるが、後述の疎行列のリストベクトルのように、粒子シミュレーション以外にも使用できる。

3.4 HPF との比較

ADETRAN が設計された時点では、分散メモリ計算機に対するプログラミング言語の標準化の動きはなく現在の HPF と比較するのは必ずしも適当ではないが、ここではその両者の特徴の違いを簡単にまとめる。

言語仕様拡張とディレクティブ方式 いずれの言語も FORTRAN を基本にしているが、ADETRAN は並列記述を言語の拡張で行い、HPF はディレクティブの形で記述する。既存の FORTRAN 処理系でプログラムを再使用する可能性がある場合は HPF が有利であるが、いずれの方式でも、新たにプログラムの書き直す手間は必要となる。また、ADETRAN の場合は ADENA の専用言語とされており、並列記述を直接表現するのにはディレクティブだけでは十分ではないので FORTRAN 言語の拡張として定義した。[2]

データ分散方式と通信生成 HPF ではプログラマが指示したデータ分散にしたがって処理系が通信を生成するするのに対し、ADETRAN では pass 文で通信をユーザが陽に指定する。HPF のように処理系が通信を自動生成する場合はその分プログラマの負担は軽くなるが、処理系がどのようなコード生成をするかを知っておく必要がある。また、ADETRAN は ADENA の専用言語であるので ADENA での通信パターン、すなわち、配列再分散通信だけを生成する必要がある。よって、プログラマの若干の負担増はあるものの、全ての通信が集団通信として記述されるので、結果的には ADENA では ADETRAN 言語を用いて効率的なアプリケーションが作成できる。

プログラミングスタイル HPF ではできるだけ多くのプログラミングスタイルを扱えるようにするために、プログラミング上の制限は少ない。ADETRAN は ADEPS 方式のプログラミングスタイルに重点をおいているので、他のスタイルでの記述は若干繁雑になる。

しかし、実装するシステムによって得意不得意なプログラミングスタイルは存在するので、自由に記述できるとしても、対象となるシステムを規定した場合はそのプログラミングスタイルも同様に制限される。例えば、ネットワーククラスタシステムでは、データパラレルでかつプロセッサ間通信が頻発するようなスタイルよりもタスクパラレルにして粒度の大きなタスクに分割するほうが実行効率は高いと考えられる。

さらに特定のプログラミングスタルに限定することによりアプリケーションの移植が容易に行える ([64, 47, 65, 48, 31]) と考え、ADENA では ADEPS 方式の記述に適するように ADETRAN は設計された。

[2] 現在設計が進んでいる ADETRAN4[30] は、汎用の言語として定義されている。

3.5 プログラミング例

ADETRAN で記述した代表的なプログラム例を示す。最初のプログラム例は規則的な配列に対する例であり、並列用の構文を用いていること以外はほとんど FORTRAN 構文であり、読み易いプログラムである。2 番目のプログラム例は不規則配列の例として、疎行列の積の記述である。不規則配列の通信を伴う場合は、データ構造として file 配列を用いるのが効果的である。この配列を用いることにより通信の集団化が図れる。

3.5.1 Splitting Up CG

基本的な配列を用いたプログラミング例として、野木及び小田中によって発案されたポワソン方程式の解法のための反復解法の一つである Splitting Up CG 法を用いたアルゴリズムを示す [31, 38]。解くべき方程式を $Ax = b$ のようにおく。この場合のアルゴリズムは下記のようになる。

x_0 : 任意の初期値
$r_0 = b - Ax_0$
$r'_0 = M^{-1} r_0$
$p_0 = r'_0$
$k = 0$
$while(r_k \neq 0)do$
$begin$
$a_k = \frac{(r_k, r'_k)}{(p_k, Ap_k)}$
$x_{k+1} = x_k + a_k p_k$
$r_{k+1} = r_k - a_k A p_k$
$r'_{k+1} = M^{-1} r_{k+1}$
$\beta_k = \frac{(r_{k+1}, r'_{k+1})}{(r_k, r'_k)}$
$p_{k+1} = r'_{k+1} + \beta_k p_k$
$k = k + 1$
end
$x = x_{k+1}$

これは加速するために行列 M を用いている。ADENA 向きの加速法として空間方向に応じた作用素の因子分解型の M を用いた Splitting Up 法があり、それに対するコードを下に示す。

```
      gsubroutine solv
      region(32,32,32)
      real a(*,/*,*/),b(*,/*,*/),c(*,/*,*/),r(*,/*,*/)
```

```
      :
     pdo j,k=1,32
       call trid(r(,/j,k/),a(,/j,k/),b(,/j,k/),c(,/j,k/),r(,/j,k/),
    c                imax)
     pend
     pass i,j,k=1,32
       r(i/,j,/k)=t(i,/j,k/)
     pend
     pdo k,i=1,32
       call trid(r(i/,,/k),a(i/,,/k),b(i/,,/k),c(i/,,/k),r(i/,,/k),
    c                jmax)
      :
     pend
     end

     lsubroutine trid(t,a,b,c,r,imax)
     real r(1),a(1),b(1),c(1),t(1)
     real tl(10000),tm(100000)
     tl(imax)=1/a(imax)
     tm(imax)=tl(imax)*r(imax)
     do 13 i=imax-1,1,-1
      tl(i)=1(a(i)-c(i)*tl(i+1)*b(i+1))
13    tm(i) = tl(i)*(r(i)-c(i)*tm(i+1))
     t(1)=tm(1)
     do 14 i=2,imax
14    t(i)=-b(i)*tl(i)*tl(i-1)+tm(i)
     return
     end
```

最初の pdo 文は、 x 方向の配列に対する計算である 3 次元配列の第 1 次元分が仮想プロセッサ内に連続して保持され、他の次元はプロセッサ上で分散されている。その 1 次元分の部分配列が

lsubroutine に引き渡される。

lsubroutine `trid` の記述は 1 次元配列だけを用いたものであり、従来の FORTRAN と全く同じ構文である。

2 番目の pdo は y 方向の配列に対する記述であり、その演算内容は仮想プロセッサ内に連続して保持されるのが y 方向であるということ以外は、最初の pdo と全く同じである。したがって、中から呼ぶ lsubroutine は最初の pdo と共通のものである。

最初の pdo では x 方向の配列を用い、計算結果が `r(i,/j,k/)` に保持され、2 番目の pdo で `r(i/,j,/k)` として使用される。したがって、この 2 つの配列は分散形態が異なるので、pass 文で配列の再分散を行なっている。

以上のように、 pdo や pass のような並列用構文以外は FORTRAN と同じ記述になっており、FORTRAN から ADETRAN への移植が容易に行なえる。

3.5.2 疎行列の積演算

密行列の演算に対しては通常配列表現と pdo 文及び pass 文、もしくは pad/pour 文を使うことによって記述可能であるが、疎行列の場合は不規則配列の演算となる。ここでは、不規則配列の例として正方疎行列 A と一次元ベクトル x との積を考える。

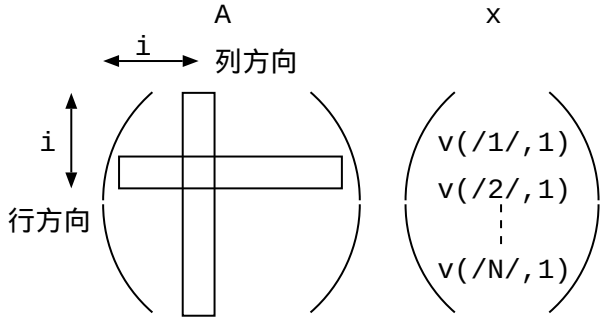


図 10: 行列とベクトルの積

配列 `A` は疎行列なので行方向 `i` の内容は下記のように表すことにより、メモリ使用の効率化を図る。分散方式は疎行列の行方向及び 1 次元ベクトルを仮想プロセッサに分散させる。

`nni(/i/,1)`　非ゼロ要素の総数

nnl(/i/,20)　非ゼロ要素の位置

aav(/i/,20)　非ゼロ要素の値

すなわち、行 i の第 j 番目の非ゼロ要素の位置は nnl(/i/,j) に保持し、その値は aav(/i/,j) とする。なお、非ゼロ要素は 20 以下と仮定している。

また、各仮想プロセッサで積和計算を行なうために、非ゼロ要素に対応した v(/j/,1) を格納する必要がある。それを、nnv(/i/,20) とおく。また、列方向 i の非ゼロ要素位置は、自データを転送すべき転送先であり、それを下記のように表す。

anni(/i/,1)　転送先総数

annl(/i/,20)　転送先位置

なお、対称行列であり、nni と anni 及び nnl と annl は一致している。以上の配列と file 配列を用いて記述する。

```
      dimension pp(/*/,1),qq(/*/,1)
      file /pp,qq/v,iloc,idst,isrc
       :
      pdo i=1,N
        rewrite ($pp(/i/,1))
        do 10 j=1,anni(/i/,1)
         $pp(/i/,1).v    = v(/i/,1)
         $pp(/i/,1).idst = annl(/i/,j)
         $pp(/i/,1).isrc = i
         $pp(/i/,1).iloc = 1
10       put($pp(/i/,1))
      pend
      push(qq(/idst/.iloc)=pp(/i/,j),i=1,N,j=1,1)
      pdo i=1,N
        reset ($qq(/i/,1))
```

```
      do 20 j=1,nni(/i/,1)
       if(eof($qq(/i/,1))) call error
       call jsrc(nnl(//,),$qq(/i/,1).isrc,jj)
       nnv(/i/,jj)=$qq(/i/,1).v
20     get($qq(/i/,1))
    pend
    pdo i=1,N
      v(/i/,1)=0.0
      do j=1,nni(/i/,1)
30     v(/i/,1)=v(/i/,1)+aav(/i/,j)*nnv(/i/,j)
    pend
```

file 配列 pp,qq は分散された 1 次元ベクトル v を送受信するための補助配列であり、通信文 push によって通信される。フィールド v には通信されるべき配列要素の値が格納され、isrc,idst には通信のソース、ディスティネーションとなるべき仮想プロセッサの番号を格納する。

最初の pdo ではソースとなる仮想プロセッサ側で file 配列 pp の定義を行なっている。まず、rewrite 文で file 配列 pp を初期化し、送信すべきレコードを生成し、put 文で登録する。

続く push 文で通信した後、2 番目の pdo 文で受信した file 配列 qq を補助配列 nnl,nnv に代入していく。つまり、reset 文で file 配列 qq のポインタを初期化し、nni(/i/,1) 個のレコードを受信しているので、do 20 で順次レコードを読み出し、書き込むべき nnv(/i/,jj) の位置にそのレコードの v フィールドの値を書き込む。最後の pdo で、実際の積和計算を行なう。

以上のように、従来の FORTRAN にはなかったデータ構造を用いることにより、不規則通信の集団化 (collective communication) が push 文によって行なえた。また、この例のように通信データ数が陽に明確な場合だけでなく、実行時まで確定していないような粒子シミュレーションにおいても有用なデータ構造である。

3.6 ADETRAN コンパイラの実装方針

ADETRAN は ADENA の専用言語であり、そのハードウエアの特徴を反映するように設計された。よって、コンパイラの実装設計としては、最大限にハードウエアを生かせるようにすることが第一である。特に通信に関しては、配列の再分散が効果的に行なえ、かつ通信レイテンシを

削減できるようなアーキテクチャになっているので、十分活用できるようにすることを念頭においた。

この節ではコンパイラ実装に関して、まず全体の実行フローについて説明した後、仮想プロセッサの扱い、さらに pad/pour 文のコンパイリング、最後に ADETRAN に特有な file 配列データ構造の取り扱いについて述べる。

3.6.1 実行フロー

ADETRAN を ADENA に実装するために、ホスト計算機上でコンパイルリンクできるクロスコンパイラ環境を構成した。ほとんどの部分を C 言語で作成し、構文解析部には YACC を使用した。[53]

ADETRAN のロードモジュールは、実行前にホスト計算機より ADENA へダウンロードされる。実行時には、ADETRAN の gsubroutine が FORTRAN から呼ばれ、ADENA 内でそのサブルーチンボディが実行される。

つまり、ホスト上で実行されるモジュールと、ADENA 上で実行されるモジュールの 2 つのモジュールが作成され、前者の中から後者が呼ばれる形式である。gsubroutine は、前述の通り pdo 文、pass 文等から構成されるが、ホスト計算機上の FORTRAN から見た場合は、次の 3 種類のブロックに分けられる。

1. ADENA 上での演算実行
2. ホスト計算機から ADENA へのデータ転送
3. ADENA からホスト計算機へのデータ転送

つまり、連続する pdo 文、ifany/ifall 文、goto 文、ADENA 内でのデータ転送を表す pass 文等は、1 つのブロックにまとめ、ホストからは、その一つの連続した実行文に対して起動をかけるだけである。

つまり、図 12の左に示す gsubroutine は、図 12の右に示す C プログラムに変換され、FORTRAN の call 文で呼ばれることになる。

コンパイル方法は、下記の 3 つの部分からなる。

1. 上位構造及び下位構造それぞれの中間言語を作成する (構文解析)

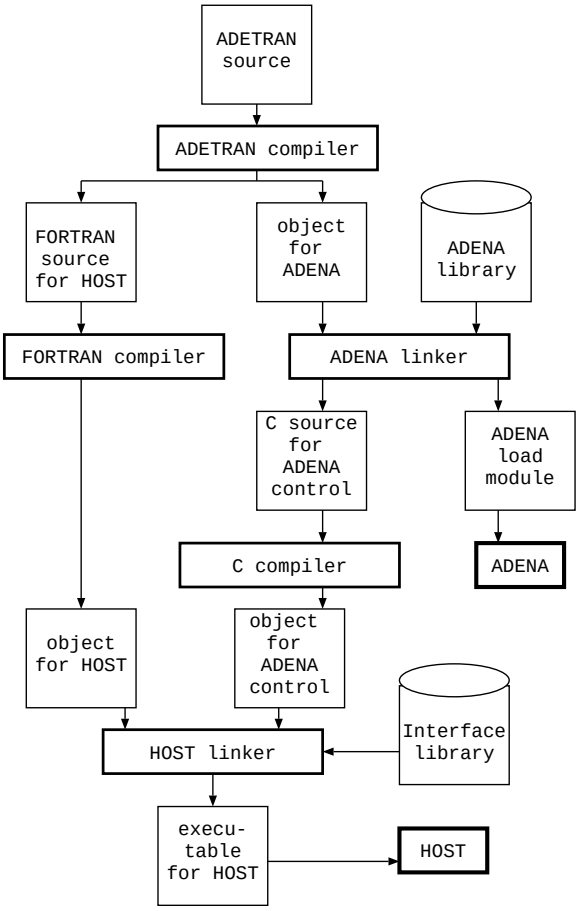


図 11: コンパイルフロー

2. 上位構造の組み換え、及び、下部構造内の最適化を行なう (最適化)

3. コード生成する (コード生成)

中間コードにおける上位構造とは、 gsubroutine を構成する pass 文や pdo を表す木構造である。また、下部構造とは、pdo 内部の実行文を表す 4 つ組みコードである。例を図 13に示す。

上位構造は、第 2 パスにおいて pass の最適化を行なうために組み換えを行なう。また、下部構造は、その内容、スカラ変数及び 1 次元配列を扱う通常の FORTRAN の構文になっているので、ループ最適化などの従来の最適化処理、及び、後述するループアンローリング等の最適化処理を行なう。その後、第 3 パスでジャンプの飛び先アドレスの計算を行なった後、オブジェクトコードの生成を行なう。

```
gsubroutine sub(u,v)
parameter(N=32)
region(N,N,N)
 :
pass i,j,k=1,N
 u(i,/j,k/)=u(i,j,k)
pend
pdo j,k=1,N
 :
pend
pdo j,k=1,N
 :
pend
pass i,j,k=1,N
 v(i,j,k)=v(i,/j,k/)
pend
end
```

```
sub_(u,v)
long *u,*v;{
  snd(...);
  fpustarta(...);
  while(gethalta(...))
    ;
  rcv(...);
}
```

図 12: プログラム例と変換後 C プログラム

3.6.2 仮想プロセッサとインデックス解析

gsubroutine 内の region 文によって、仮想プロセッサ空間が設定される。つまり、

```
region(NX,NY,NZ)
```

の時、 x 方向仮想プロセッサ空間は、/NY,NZ/ であり、y 方向及び z 方向仮想プロセッサ空間は、/NZ,NX/及び/NX,NY/である。

従って、実際のプロセッサ空間 (物理プロセッサ空間) との対応は、図 14 の左図のようになり、各プロセッサを複数回 (多重) 使用する。また、メモリも、多重度毎に持つ。それぞれの多重度に対するメモリにアクセスするためにベースポインタを使い、よって、各多重度に対する実行は同一コードになる。

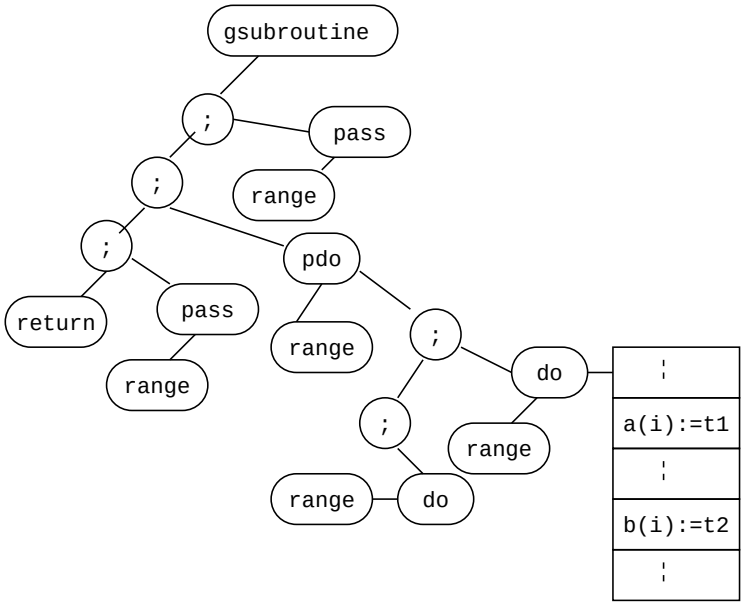


図 13: 中間コード例

仮想プロセッサの制御には、SIMD のようにホストが関与して全プロセッサが同期して行なう方法と各プロセッサで独立に制御する方法の 2 つがある。今回の実装では、仮想プロセッサ動作のために分散制御 (SPMD) で行なった。これの長所として 2 点あげる。

1. ホストから操作しないので、実行時のオーバーヘッドが減る。
2. 実際の並列度が上がり、実行速度が早くなる。

2 について少し説明する。

例えば、region 文で仮想プロセッサ空間を、

```
region(32,32,32)
```

と設定し、pdo 文として、

```
pdo j=16,31,k=16,31
  u(1,/j,k/)=0.0
pend
```

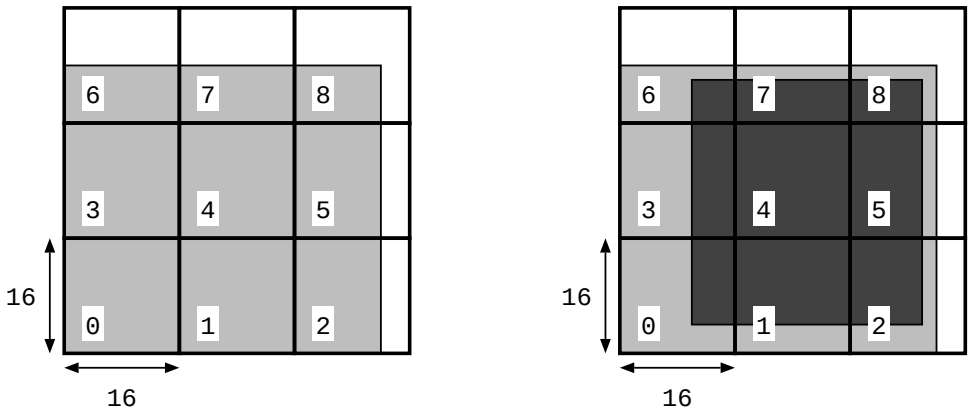


図 14: 仮想プロセッサ空間: region(20,40,44) の x 方向、及び pdo レンジ: pdo j=4,38,k=12,42

を実行する場合を考える。

これをホスト制御すると、図 15のように、 4 多重分の実行時間がかかる。つまり、1 多重目は j=16,16,k=16,16 であり、2 多重目は j=16,16,k=17,31 であり、3 多重目は j=17,31,k=16,16 であり、4 多重目は j=17,31,k=17,31 である。

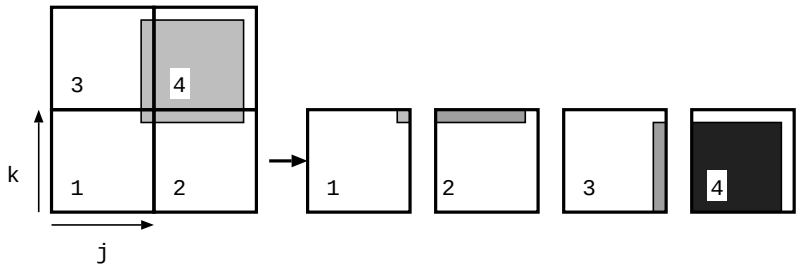


図 15: ホスト制御の場合

しかし、分散制御すると、各プロセッサは、MLTS ルーチンは 4 回通るが、pdo ボディは 1 回しか通らない。つまり、図 16のように、実行時間としてはホスト制御する場合の 1/4 で済むことになる。

したがって、分散制御することにより、実行速度の向上を図ることができる。

次に、pdo 文に対する実際の仮想プロセッサを説明する。仮想プロセッサ動作は、 pdo 文のインデックスの範囲 (pdo レンジ) と pdo 文内の実際の演算実行文 (pdo ボディ) によって決まる。仮想プロセッサ空間と pdo レンジとの関係の例を図 14の右図に示す。

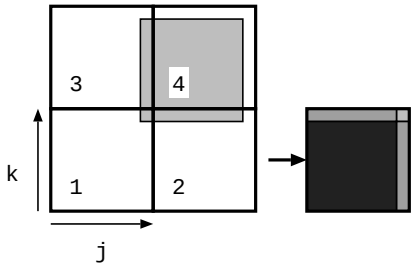


図 16: 分散制御の場合

つまり、(20,40,44) の region の x 方向の演算であるので、仮想プロセッサ全体の大きさは (40,44) である。一方、pdo の実行レンジは (4:38,12:42) であるので、14の右図のように、仮想プロセッサの一部 (色の濃い部分) のみ演算実行する。

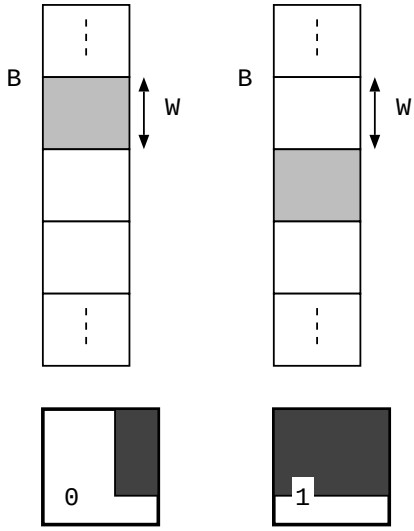


図 17: 第 1 多重と第 2 多重のベースポインタ

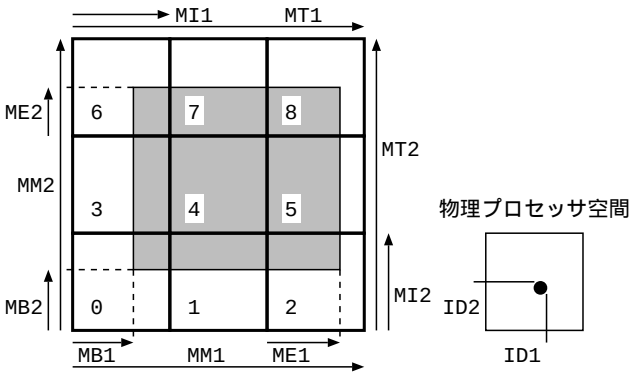
従って、第 1 多重の場合は、図 17の左図のように、ベースポインタは B で、物理プロセッサ空間で動作するのは、灰色の部分である。また、多重度 2 の場合は、図 17の右図のようになり、ベースポインタは B+W である。以下同様の動作となり、この多重度切替えで必要な処理は次のようなものとなる。

1. メモリのベースポインタの切替え

2. 動作プロセッサの選択

これらは物理プロセッサ空間全体に関する動作であるが、先ほど述べたように、ホストが制御するのはオーバーヘッドも多く、また、実行効率の面からもあまり好ましくない。従って、ADENA では、各プロセッサは、自プロセッサ番号を用いて分散制御する。

すなわち、物理空間のプロセッサ番号 (ID1,ID2) と region 文および pdo の実行範囲から抽出されたパラメータ (MMN1, MM2, MI1, MI2, MT1, MT2, MB1, MB2, ME1, ME2) をもちいて、仮想プロセッサ処理 (MLTS) ルーチンを作成する。このルーチンの中で、各物理プロセッサは仮想プロセッサ (多重度) を切替える毎に MLTS ルーチンを実行し、その多重度での実行を行なうかどうかの判断を行なう。




```
Base=B;
for(MC2=0;MC2<MM2;MC2++){
  for(MC1=0;MC1<MM1;MC1++,Base += W){
    if(MC2<MI2 || MC1<MI1 || MC1>MT1 || MC2>MT2)
      ;
    else if(MC2==MI2 && ID2 <MB2)
      ;
    else if(MC1==MI1 && ID1 <MB1)
      ;
    else if(MC2==MT2 && ID2 >ME2)
      ;
    else if(MC1==MT1 && ID1 >ME1)
      ;
    else
      execute();
 }
}
```

図 18: 仮想プロセッサ処理ルーチン (MLTS)

プロセッサ空間のパラメータを図 18の左図のように定義し、図 18の右図の処理ルーチン (MLTS ルーチン) を通し、execute 関数の中で、pdo ボディを実行する。

さらに、pdo ボディに入る前に、ベースポインタの設定をする。pdo 文に対しては、このようなコードを生成し、pdo 内の構文に関しては、通常のコンパイラ処理を行なう。以上のコンパイラ処理をまとめると、図 19のようになる。

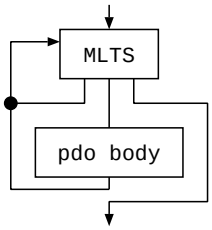


図 19: pdo 文のマクロ構造

3.6.3 pour/pad 文

pour/pad 文は、前述のように一般行列の逆行列計算に主に用いられるものであり、2 次元行列に対するもののみである。ADENA においては pass 文による再分散通信が高速である。したがって、この通信を利用するために、現処理系では、両構文ともに、pdo 及び pass の構文に変換して実行するようにした。

まず、pour 文であるが、これは各仮想プロセッサに分有されている 1 ワードを連結したベクタを全プロセッサに分配するものである。したがって、下記のように全プロセッサ分のコピーを行ない、それを 2 次元転送する。

```
pour i,j=1,N                          pdo i=1,N
 a(/i/,j)=b(/i/,2)                     do 10 j=1,N
pend                              10     t1(/i/,j)=b(/i/,2)
                                      pend
                                      pass i,j=1,N
                          →             t1(i,/j/)=t1(/i/,j)
                                      pend
                                      pdo i=1,N
                                       do 20 j=1,N
                                  20     a(/i/,j)=t1(j,/i/)
                                      pend
```

最後の pdo 文は、一般ユーザでは使用できない構文であるが、同一プロセッサ内での代入であるので、コンパイラのマクロ出力でのみ許しているものである。

次に pad 文であるが、これはあるプロセッサの配列を全部のプロセッサに分配するものである。まず、ソースとなる配列の分散方向を変え、コピーすべき部分を全配列に対してコピーする。さ

らに、再度分散方向を変えることにより実現している。

```
pad i,j=1,N
 a(/i/,j)=b(/2/,j)
pend
```

→

```
      pass i,j=1,N
       t1(i,/j/)=b(/i/,j)
      pend
      pdo j=1,N
       do 10 i=1,N
10      t1(i,/j/)=t1(2,/j/)
      pend
      pass i,j=1,N
        a(/i/,j)=t1(i,/j/)
      pend
```

3.6.4 file 配列

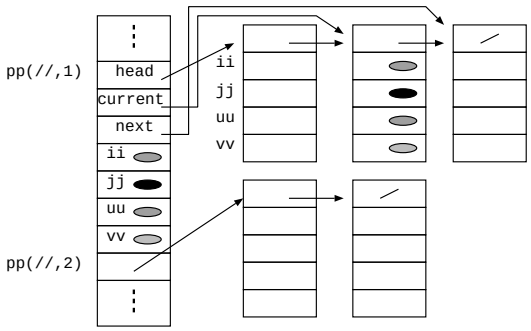


図 20: file 配列構造

file 配列は、順アクセスのみ可能な任意長のレコードへのポインタの配列である。前述の通り、ADENA にはこのデータ構造を高速に送受信するためのハードウエアは用意されていないが、個別通信を集団化することにより、通信起動コストを削減することが可能である。

file 配列を用いた実行は、代入文の右辺における参照等の通常の使用以外に、書き込みを行なうために書き込み用ポインタをリセットする rewrite 文、書き込みを行なう put 文、読み出し用ポインタをリセットする reset 文、そして読み出しを行なう get 文がある。また、通信を行なうには push 文を用いる。

プログラム中の計算には、配列要素毎にウインドウを設定してその中で計算を行なう。したがって、実際の配列構造は次の要素を持つことになる。

- レコードの先頭ポインタ (head)
- ウインドウ上で有効なレコードのポインタ (current)
- 次のレコードのポインタ (next)
- ウインドウ (数ワード)

これの例を図 20に示す。file 配列要素 pp(//,1) には 3 つのレコードが連結されており、current は 2 番目のレコードを指している。

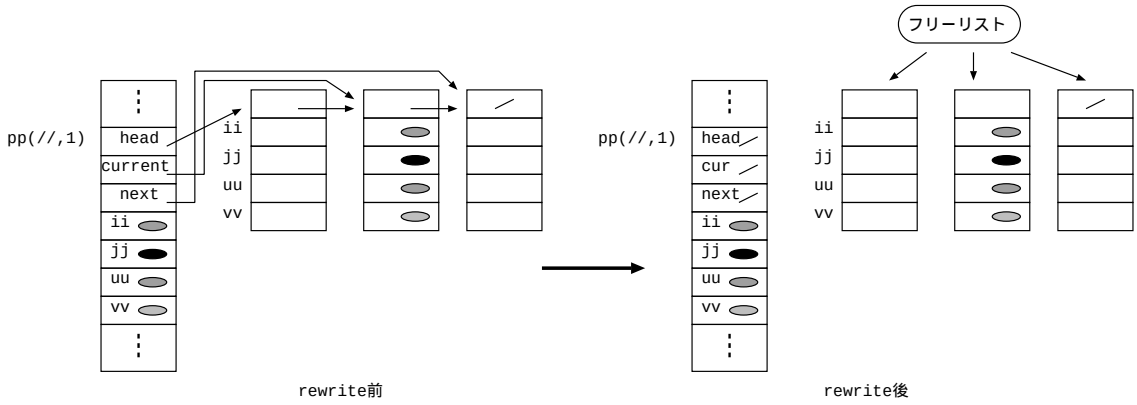


図 21: rewrite 文

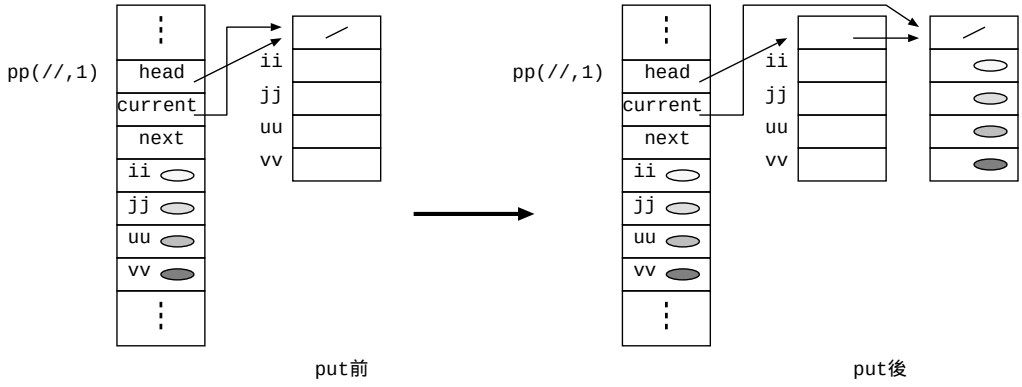


図 22: put 文

まず、rewrite 文の動作について説明する。file 配列を初期化し、新たにレコードを登録し始めるので、現在つながっているレコードを切り離し、それらをフリーリストへ回収する。(図 21)

current ポインタや head ポインタは 無効化 (NULL) する。

put 文は、新たなレコードをメモリ上に割り当て、ウインドウ上に書かれている内容をそれにコピーする。head ポインタが NULL であれば、そのレコードのポインタを書き込み、NULL でなければ更新はしない。current ポインタが NULL でないならば、そのポインタには直前に put されたレコードのポインタが書かれている。したがって、current ポインタが指すレコードの next フィールドに、新たに作成されたレコードのアドレスを設定する。また、current ポインタには、作成したレコードのポインタを書く。(図 22)

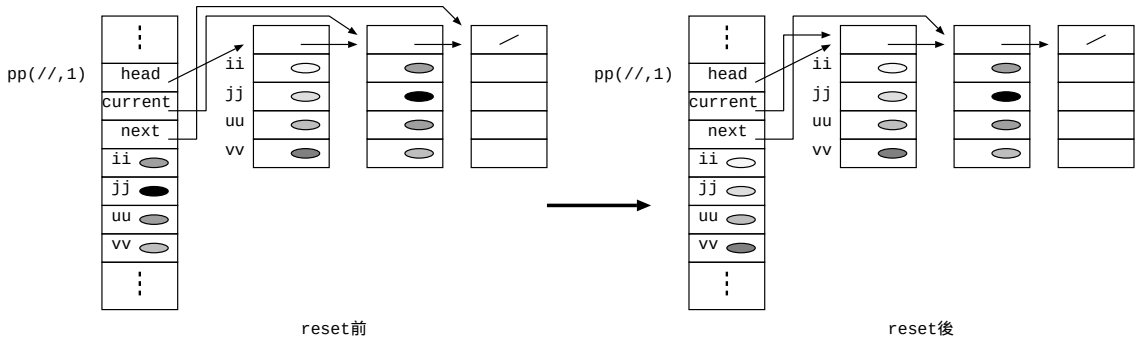


図 23: reset 文

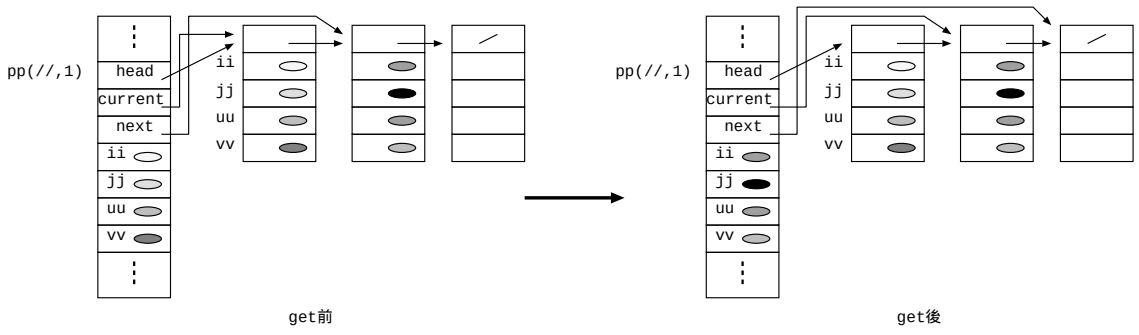


図 24: get 文

reset 文は、読み出しを開始するためのものであるので、head ポインタの内容を current ポインタに設定し、current ポインタの指すレコードの内容をウインドウにコピーする。もし、head が NULL であればコピーは行なわない。これにより、最初のレコードの内容がウインドウ上に存在し、読み出しが可能になる。(図 23)

get 文は、current ポインタが指すレコードの次のレコードを読み出すものである。next ポインタの内容を current ポインタに設定し、current ポインタの指すレコードの内容をウインドウにコピーする。もちろん、next が NULL であればコピーは行なわない (図 24) 。読み出すべきレコードがない場合は、get 文、reset 文ともに戻り値として EOF(End Of File) を返し、任意数のレコードの最終レコードを検出することが可能である。

プラズマ粒子シミュレーションにおいて、file 配列はプラズマ粒子を表すデータ構造である。従って、場の計算が終ると、それぞれの粒子は場の力によって他のプロセッサの管理する空間に移動する。これを表現するのが push 文であり、図 25のようなサブルーチン構成で実現される。

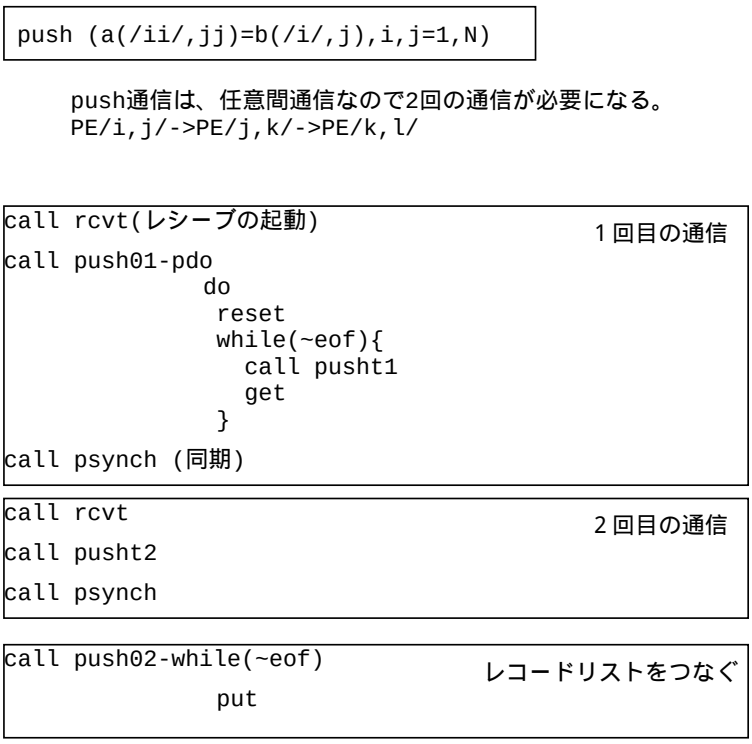


図 25: push 文

push 通信は、任意間の通信なので、一般に 2 回の通信が必要である。つまり、(i,j) 番のプロセッサから (k,l) 番のプロセッサへの通信には (j,k) 番のプロセッサを経由しなければいけないからである。そこで、1 回目の通信においては、まず受信側関数 (rcvt) を起動した後、file 配列要素毎に reset/get を繰り返してレコードを読み出しながら、 (i,j) から (j,k) への通信を行なう (pusht1) 。2 回目の通信では、同様に (j,k) から (k,l) へ送るために受信側関数 (rcvt) と送信側関

数 (pusht2) を起動する。通信同期/エラーチェックを行なった (psynch) 後、最後に通信されたレコードを put 文でリストで連結していく。以上の手続きで、任意の file 配列が任意の file 配列に再構成することができる。

3.7 データロード最適化

FORTRAN において計算量の大きい部分は do ループである。ADENA においても do ループを並列処理して高速化を図っている。したがって、ループ内の計算時間を縮小することは効果がある。

また、偏微分方程式の解法を初めとする数値シミュレーションにおいては、do ループ内の演算は、ループインデックスを用いた差分計算になっていることが多い。これに対する最適手法を検討する。

```
do 1 i=1,21
 u(i,/j,k/)=v(i-1,/j,k/)+v(i,/j,k/)+v(i+1,/j,k/)
1 continue
```

図 26: プログラム例

図 26 に示したプログラム片をコンパイルすると、図 27に示す疑似コードが得られる。

```
_loop_entry i=1,21,1
ENTRY:
_ld  #v(i-1),fr0          ロード命令
_ld  #v(i),fr1
_ld  #v(i+1),fr2
_add fr0,fr1,fr3          浮動小数演算命令
_add fr3,fr2,fr4
_st  fr4,#u(i)            ストア命令
_loop_return ENTRY
```

図 27: 疑似コード 1

このように、ループ内ではロード命令が 3 つ、ストア命令が 1 つ、浮動少数演算命令が 2 つあり、演算効率が悪い。つまり、 2 つの浮動少数演算のために合計 6 命令必要となっている。

そこで、図 26の代入式に着目する。これはループインデックスの差分計算となっており、同じデータを複数回ロードしている。つまり、例えば、i=2 の時、v(1,/j,k/) と v(2,/j,k/) と v(3,/j,k/) をロードし、i=3 の時、v(2,/j,k/) と v(3,/j,k/) と v(4,/j,k/) をロードし、i=4 の時、v(3,/j,k/) と v(4,/j,k/) と v(5,/j,k/) をロードするので、v(3,/j,k/) は 3 回ロードすることになる。よって、同じデータを何度もロードするのではなく、そのデータをレジスタに残しておき、loop return の前にレジスタ間ムーブすることにより無駄なデータのロードは減る。このアプローチを、図 28 に示す。

```
_ld  #v(0),fr0
_ld  #v(1),fr1
_loop_entry i=1,21,1
ENTRY:
_ld  #v(i+1),fr2          ロード命令
_add fr0,fr1,fr3
_add fr3,fr2,fr4          浮動小数演算命令
_st  fr4,#u(i)            ストア命令
_mv  fr1,fr0
_mv  fr2,fr1
_loop_return ENTRY
```

図 28: 疑似コード 2

この方法は、通信最適化における メッセージ合体法 (message coalescing) と同様な発想である。しかし、ロード命令は減ることになるが、ムーブ命令が入ることになり、大幅な時間削減にはならない。

そこで、レジスタ間ムーブをなくすために、コード自体を 3 倍に展開 (アンローリング) し、使用しているレジスタ番号をサイクリックに移動させる。この方法を図 29 に示す。

このように、ロード命令が 3 、ストア命令が 3 、浮動少数点演算が 6 つとなり、演算効率が高くなる。

	最適化前	最適化後
総コード数	6	12
浮動少数演算数	2	6

```
_ld  #v(0),fr0
_ld  #v(1),fr1
_loop_entry i=1,21,3
ENTRY:
_ld  #v(i+1),fr2
_add fr0,fr1,fr3        fr3=fr0+fr1
_add fr3,fr2,fr4        fr4=fr3+fr2
_st  fr4,#u(i)
_ld  #v(i+2),fr0
_add fr1,fr2,fr3        fr3=fr1+fr2
_add fr3,fr0,fr4        fr4=fr3+fr0
_st  fr4,#u(i+1)
_ld  #v(i+3),fr1
_add fr2,fr0,fr3        fr3=fr2+fr0
_add fr3,fr1,fr4        fr4=fr3+fr1
_st  fr4,#u(i+2)
_loop_return ENTRY
```

図 29: 疑似コード 3

FLOPS	1^3	1.5

この手法をまとめる。

1. do インデックスを固定したループ内で使用される配列インデックスの範囲 (i-min,i+max) を求める。
2. i-min から i+max までの配列要素を loop_entry の前でロードしておく。
3. loop_body の先頭で、i+imax の配列要素をロードする。
4. loop_body をコンパイルする。ただし、ストア命令は最小インデックスにのみ生成する。
5. レジスタ割り当てを再クリックに変え、(min+max+1) 回のアンローリングを行なう。
6. loop_return を作成し、その後にループ内のストア命令で最小インデックス以外のコードを書く。

この方法により、数値計算のように do ループ内に配列演算を多く含む場合には、効率良いコードを生成できる。

[3] 最適化前を 1 とする

3.8 通信最適化

ADENA において、並列実行に伴うプロセッサ間の通信は、pass 文によって陽に記述される。よって、コンパイラによるデータ通信の検出は不要である。しかし、プロセッサ間の通信は逐次実行時には存在しないので、オーバーヘッドとなり、できるだけ減らすことが必要である。

ADENA では、通信専用の LSI (TCU) を持っているので、演算実行しながら転送を同時に行なうこと (DMA 転送) ができる。特に、EPU がメモリへのストアと同時に TCU へシグナルを送り、 TCU はそのシグナルを受けた時に、1 ワードの転送を実行するモード (S スキーム通信モード) を備えている [55] 。

よって、このモードを有効に利用するようなコンパイラ最適化処理が重要となる。

3.8.1 データ通信

プログラム中の pass 文と その直前の pdo 文のパラメータの条件により、その pass 文に伴う通信のパターンを 3 つに種類分けできる。

過去 (**past**) 型: pdo 内で更新されない配列の通信

未来 (**future**) 型: 現在型通信以外で、pdo 内で更新される配列の通信

現在 (**now**) 型: pdo 内で更新されるが、更新される範囲と通信される範囲が一致する場合

過去 (**past**) 型　過去型の場合、通信すべき配列はその直前の pdo 文内では更新されないので、この pdo 文の実行と通信をオーバーラップすることが可能である。つまり、pdo 文の開始前に通信の起動を行なっておき、pdo 文の実行終了後に通信終了を確認することになる。

図 30 にこの処理を示す。この中で使用している表記の説明を以下に記述する。

TSET	通信のためのパラメータを TCU にセットし、通信を起動する。
MULTI	仮想プロセッサ処理 (MLTS ルーチン)
EXEC	pdo ボディの実行
SYNC	通信終了及び通信エラーの確認
RSET	通信エラー時に通信の再起動を行なう

図から明らかなように、TSET により起動された通信と MULTI/EXEC の演算実行がオーバーラップできる。

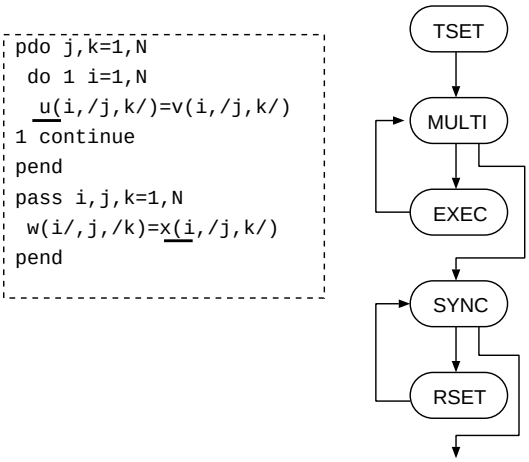


図 30: 過去 (past) 型通信

未来 (**future**) 型　未来型通信では、pdo の演算が全て終了してから初めて通信が開始できる。したがって、通信時間が演算とは独立に発生する。

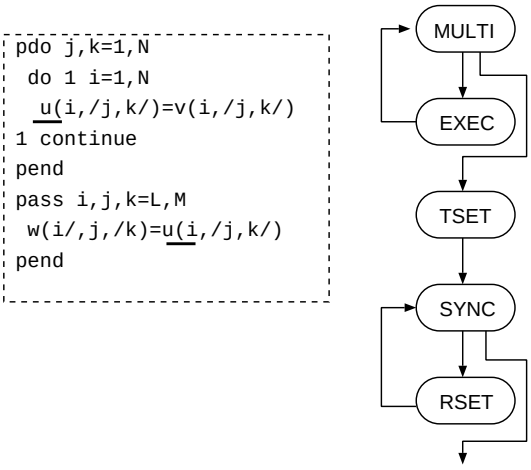


図 31: 未来 (future) 型通信

図 31 に示してあるように、TSET/SYNC の間は通信だけを行なっている。

現在 (**now**) 型　現在型の場合、通信すべき範囲と通信すべき配列が更新される範囲が一致している。したがって、前述の S スキームモードを用いれば、通信と pdo の実行が並行に行なえる。

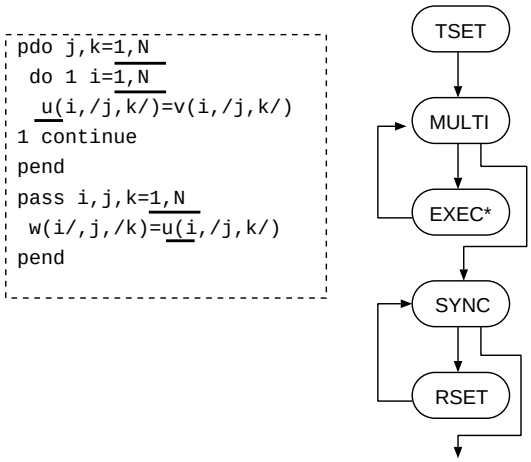


図 32: 現在 (now) 型通信

図 32に示すように、配列要素 u(i,/j,k/) をストアする毎に TCU にシグナルを送り、TCU はそのシグナルを受け取る毎に決められた順に通信を 1 ワード だけ行なえば、通信は pdo 文の実行と並行に行なえる。このとき、EXEC 内のストアは、シグナル付きのストアに変更する。

考察　過去型通信は、バスに空きがあればいつでも通信を行なう。現在型通信は、EPU からのシグナルを受け、かつバスに空きがあれば通信を行なう。未来型通信は、 EPU の処理が終った時点で初めて通信を開始する。

よって、以上より明らかなように、一般に通信時間は

$$\text{過去型通信} < \text{現在型通信} < \text{未来型通信}$$

の順になる。そこで、未来型を現在型に、現在型を過去型に変形することは有効な最適化手法となる。

現在型通信は、EPU が転送対象となる配列要素をストアする毎に通信を行ない、過去型通信は、pdo ボディの実行には関わらず通信を行なう。しかし、pdo ボディの実行中に現在型通信を行なうのに十分なバスの空きがあれば、過去型通信を行なうのにも十分であり、ないのであれば、いずれの通信も効率良くは行なえない。

一方、未来型を過去型に変形するというのは、従来からいくつかの手法がある。例えば、下記のようなプログラムを考える。

```
pdo j=..
 a(i,/j/)=....
pend
pass
 b(/i/,j)=a(i,/j/)
pend
pdo
 x(/i/,j) = b(/i/,j)
pend
pdo
 y(/i/,j) = c(/i/,j)
pend
```

このプログラムは配列 a(i,/j/) 及び b(/i/,j) に関して、第 1 の pdo 文→pass 文→第 2 の pdo 文 というフロー依存性がある。したがって、第 2 の pdo 文と pass 文の関係より、この通信は未来型となる。しかし、第 2 の pdo 文と第 3 の pdo 文には依存関係がないので、この pdo 文を入れ換えることができる。そうすると、新たに 2 番目になった pdo 文と pass 文には依存関係がないので、通信は過去型となり、計算とオーバーラップできる。このように、過去型への変形は、従来からのループ変換技術 [44] によって可能になる。

ここでは従来技術では行なっていなかった未来型通信から現在型通信に注目し、その変換方法について述べる。

3.8.2 通信型変換

未来型通信と現在型通信は、いずれも先行する pdo 文中で変更された配列を通信するものである。未来型通信を現在型通信に変換するためには、次の 2 点が必要である。

- if 文や goto 文の影響をなくして、規則的にストアする
- pdo と pass の range を合わせる

よって、これらの条件を満たすように変換していく方法について以下に述べる。

デフォルト 代入の追加　if 文中の代入は条件判定により、配列への代入 (ストア) が行われたり、行われなかったりする。しかし、現在型通信を行うためには、通信すべき配列要素を連続的にストアしなければならない。そこで、一時変数を用意し、do_body の先頭で代入される配列要素をロードしておき、do_body の最後で、その一時変数を配列要素に代入する。do_body 内のその他の演算は、一時変数を用いて行う。

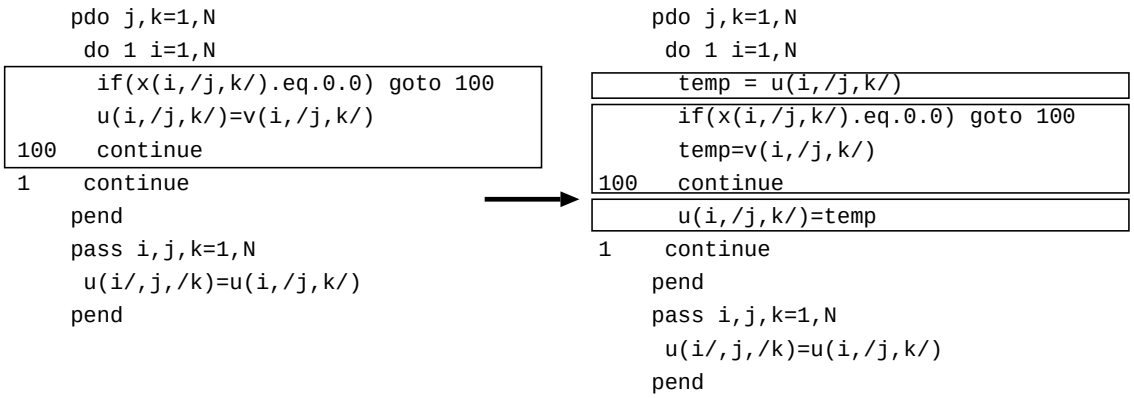


図 33: if 文が存在する (変換 1)

例外として、ブロック if 文の場合は、else 文を追加する。

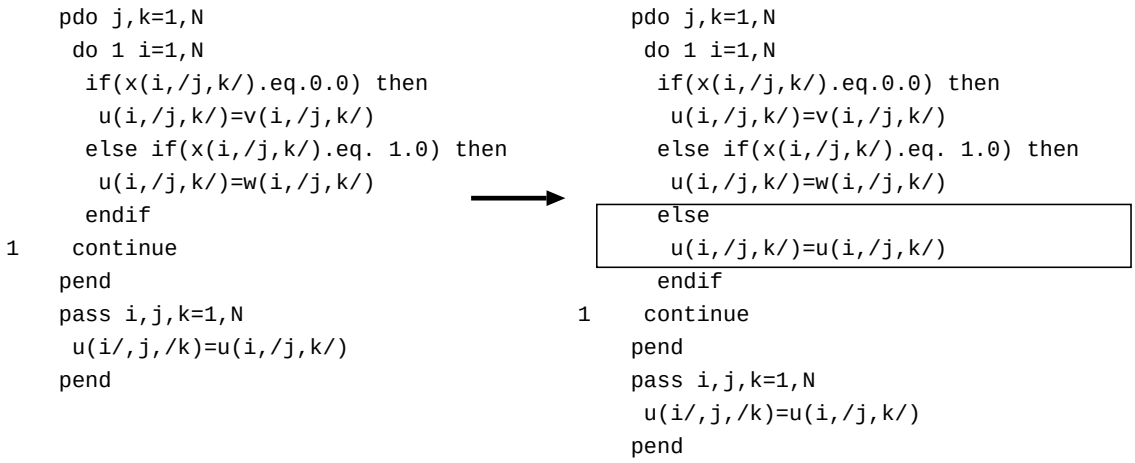


図 34: block if 文が存在 (変換 2)

レンジの一致　pdo レンジと pass レンジが異なる場合、EPU の出すシグナルの数と通信すべきデータの数があわなくなり、現在型通信ができない。そこで、pass レンジにあわせて pdo/do ループの分割/拡張を行う。

まず、pass レンジが狭い場合は、pdo/do ループを pass レンジと一致する pdo/do ループとそうでないものに分割し、前者のストア命令のみをシグナル付に変える。また、pass レンジが広い場合は、pdo/do ループを広い分だけ拡張し、不足分の通信すべき配列要素のロード / シグナル付ストア命令を追加する。

これらの操作により、任意のレンジを持つ未来型通信が現在型通信に変換できる。しかし、複雑に分割/拡張手法を用いるので実行時のオーバーヘッド が増えることになり、有効性が少ない場合も考えられる。ここでは、まず比較的簡単な例を示す。

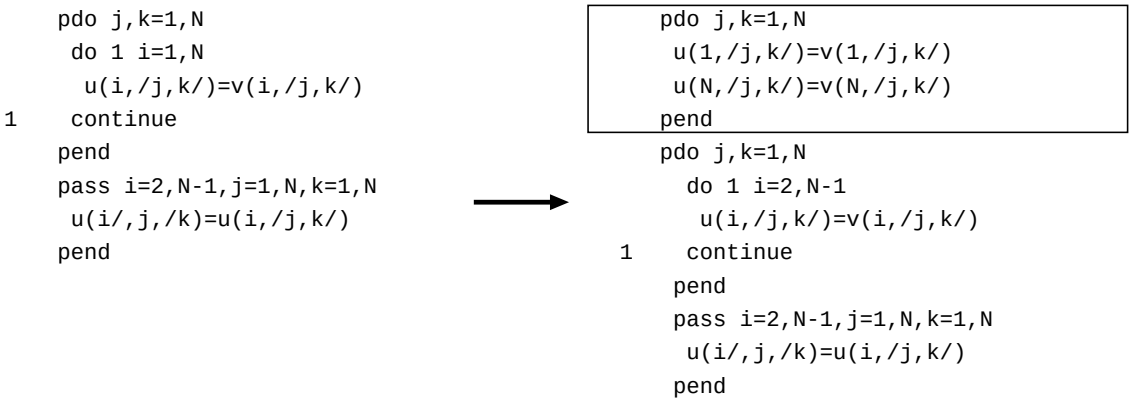


図 35: 通信レンジが小さい (変換 3)

図 35の例は、代入される do ループの再内側のレンジが pass レンジよりも大きい場合である。この時、はみ出した pdo 内の代入を pdo の前で行っておく。

図 36は、図 35の場合とは異なり、代入される do ループの再内側のレンジが pass レンジよりも小さい場合、不足する代入を追加する。

次に、より複雑な、再内側以外のレンジも異なる場合について検討する。

図 37の中の左に記述しているプログラムの pdo レンジと pass レンジの関係を右に示す。つまり、pdo 文で更新される部分もあるが、されずに通信される部分もある。このレンジの共通集合を pdo レンジ 1 、それ以外の部分を pdo レンジ 2 とおく。これらの部分に対し、次のような処理を行なう。

```
    pdo j,k=1,N
     do 1 i=2,N-1
      u(i,/j,k/)=v(i,/j,k/)
1    continue
    pend
    pass i,j,k=1,N
     u(i/,j,/k)=u(i,/j,k/)
    pend
```

→

```
    pdo j,k=1,N
      u(1,/j,k/)=u(1,/j,k/)
      do 1 i=2,N-1
       u(i,/j,k/)=v(i,/j,k/)
1     continue
      u(N,/j,k/)=u(N,/j,k/)
    pend
    pass i,j,k=1,N
     u(i/,j,/k)=u(i,/j,k/)
    pend
```

図 36: 通信レンジが大きい (変換 4)

```
    pdo j,k=10,100
     do 1 i=10,100
      u(i,/j,k/)=f(v(i,/j,k/))
1    continue
    pend
    pass i,j,k=1,30
     u(i/,j,/k)=u(i,/j,k/)
    pend
```

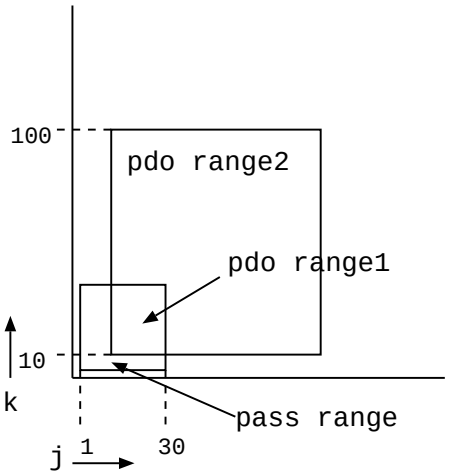


図 37: 演算と通信のレンジが大きく異なる場合

1. pdo レンジ に対してマスクデータを作成する。マスクデータとは、pdo を実行する範囲を示すフラグであり、このプログラム例に対しては、下記のようなコード を付加する。

```
pdo j,k=1,100
 mask(1,/j,k/)=.false.
pend
pdo j,k=10,100
 mask(1,/j,k/)=.true.
pend
```

2. pdo レンジ 2 に対するコードを生成する。このコードでのストアはシグナル無しである。

```
      pdo j=31,100,k=10,100
       do 10 i=10,100
10      u(i,/j,k/)=f(v(i,/j,k/))
      pend
      pdo j=10,30,k=31,100
       do 11 i=10,100
11      u(i,/j,k/)=f(v(i,/j,k/))
      pend
```

3. pdo レンジ 1 および pdo レンジ外の部分の通信に対応するデフォルト代入文を生成する。この場合、f(v(i,/j,k/)) を計算すべき部分とデフォルト代入 u(i,/j,k/)=u(i,/j,k/) との切替えのためにマスクデータ (mask(1,/j,k/)) を用いる。

```
      pdo j,k=1,30
       do 20 i=1,9
20      u(i,/j,k/)=u(i,/j,k/)      (シグナル有り)
       do 21 i=10,30
        if(mask(1,/j,k/))then
         u(i,/j,k/)=f(v(i,/j,k/)) (シグナル有り)
        else
         u(i,/j,k/)=u(i,/j,k/)    (シグナル有り)
21      continue
       do 22 i=31,100
        if(mask(1,/j,k/))then
         u(i,/j,k/)=f(v(i,/j,k/)) (シグナル無し)
        else
         u(i,/j,k/)=u(i,/j,k/)    (シグナル無し)
22      continue
      pend
```

再内側ループの範囲が i=1,30 の場合は、pass レンジと重なっているので、全ての代入は

シグナル付きに行なわれる。また、それ以降の部分はシグナル無しで行なわれる。

以上のように pdo 文を変換することにより、最後の pdo 文 (`pdo j,k=1,30`) は、pass 文と S スキームモード通信可能となり、通信がオーバーラップできる。つまり、未来型通信を現在型通信に変換できる。

なお、この例のように、図 37の元プログラムよりも変換後のプログラムは長くなるので、マスクデータの生成や、デフォルト代入文の付加などによる、演算処理時間の増大の危険性がある。しかし、通常のプログラムの場合は、pdo レンジと pass レンジの差は、$\pm 1 \sim 3$ であることが多い。したがって、一般に本方式が有効なケースは多いと考えられる。

3.9 実験

ここまでに述べた最適化手法の有効性を評価するために、偏微分方程式の反復解法によくでてくるものとして、do ループを用いた差分計算に、本方式を適用してみる。

まず、使用するプログラムを下に示す。

```
      do 1 i=2,N-1
      do 1 j=2,N-1
      do 1 k=2,N-1
       u(i,j,k)=-v(i,j+1,k)-v(i,j-1,k)+(2-r)*v(i,j,k)
    c           -v(i,j,k+1)-v(i,j,k-1)+(2-r)*v(i,j,k)
```

これを ADETRAN で表現すると、次のようになる。

```
      pdo  k,i=2,N-1
      do 10 j=2,N-1
10     u(i/,j,/k)=-v(i/,j+1,/k)-v(i/,j-1,/k)+2*(2-r)*v(i/,j,/k)
      pend
      pass i,j,k=1,N
       u(/i,j/,k)=u(i/,j,/k)
       v(/i,j/,k)=v(i/,j,/k)
      pend
```

```
      pdo  i,j=2,N-1
      do 20 k=2,N-1
20     u(/i,j/,k)=u(/i,j/,k)-v(/i,j/,k+1)-v(/i,j/,k-1)
      pend
```

このプログラムに関して、通信最適化及びデータロード最適化を適用し、256 プロセッサの ADENA(ピーク性能 5GFLOPS) で性能を測定した。

1. 1 個のプロセッサで全計算をする。グラフ中には記していないが、1.85MFLOPS である。
2. 256 個のプロセッサを持つ ADENA で、未来型通信で、データロード最適化をしない場合。
3. 256 個のプロセッサを持つ ADENA で、未来型通信で、データロード最適化をする場合。
4. 256 個のプロセッサを持つ ADENA で、現在型通信で、データロード最適化をしない場合。
5. 256 個のプロセッサを持つ ADENA で、現在型通信で、データロード最適化をする場合。

上記の 2-5 の場合の実験結果を図 38に示す。

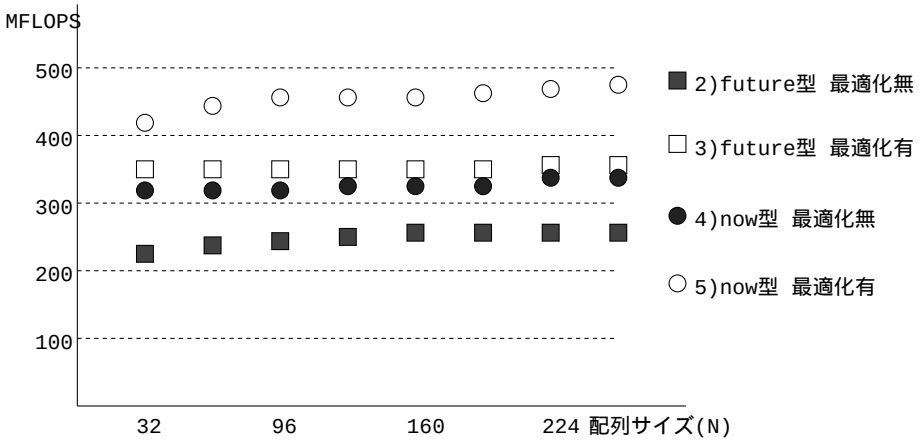


図 38: 実験結果 1

性能に関しては、$5 > 3, 4 > 2 >> 1$ と予想されるが、実験結果はそれに一致している。特に、配列サイズ (N) が 256 の場合、5 の実測値 (480MFLOPS) は 2 の実測値 (240MFLOPS) よりも 2 倍性能がいいことが分かる。また、サイズが小さい場合、例えば 32 のような場合でも、すでに 30% の効率の増加を得ている。

次に、このプログラムの pdo レンジをやや一般化し、その効果を確かめるために、下のプログラムを考える。

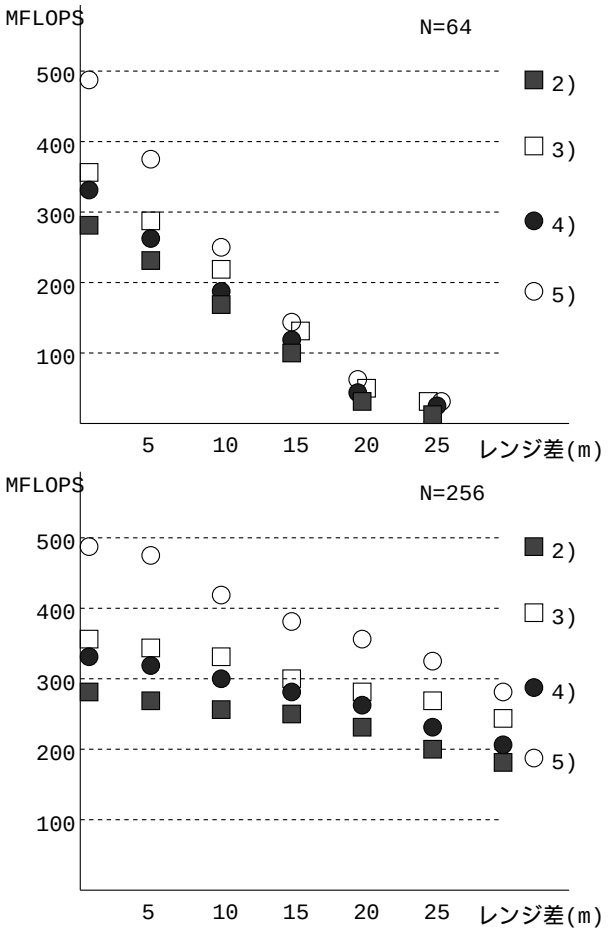


図 39: 実験結果 2

```
      pdo  k,i=m,N-m
      do 10 j=m,N-m
10     u(i/,j,/k)=-v(i/,j+1,/k)-v(i/,j-1,/k)+2*(2-r)*v(i/,j,/k)
      pend
      pass i,j,k=1,N
       u(/i,j/,k)=u(i/,j,/k)
       v(/i,j/,k)=v(i/,j,/k)
      pend
```

先ほどのプログラムとの違いは、pdo の範囲が pass 文よりも狭くなっているので、デフォルト代入の手法によって、未来型通信を現在型に変換する必要があることである。これの実験結果を図 39に示す。

左の結果は、配列サイズが 64 の場合を示し、右の結果は 256 の場合を示す。図から明らかなように、レンジ差が大きくなるにつれて、性能の低下が生じる。しかし、最適化を適用した方の性能 (5) が適用しない方 (2) よりも悪くなるという結果は出ていないので、任意のループに対して本方式の最適化を適用することは効果があると考えられる。

3.10 まとめ

ADENA は、配列の再分散通信を高速にした分散メモリ計算機である。通信パターンを効率的に実装できる HX ネットワークをプロセッサ間結合網として採用し、プロセッサ内の通信プロセッサである TCU は複数ディスティネーションに対する通信起動を低コストで行なえる。また、S スキームモードをプロセッサに実装し、通信レイテンシの低減化をハードウエアレベルで実現している。

ADENA に対するプログラミング言語として、上記の通信モードを有効に活用し、かつ FORTRAN をベースとしたプログラミング言語 ADETRAN が設計された。この ADETRAN 言語のコンパイラ設計を行ない、効果的な仮想プロセッサ処理及び通信モードの最適利用の手法を考案し実装した。

コンパイラ最適化として、ループアンローリングを使ったデータロード最適化手法を適用した。この方式は既存のベクトル演算方式とは異なり、ループサイズが比較的小さい時でも有効であり、30% 以上の性能向上が見られた。この方法は ADENA 以外の計算機でも有効である。

また、未来型通信を現在型通信に変換することは、pdo レンジと pass レンジの差が小さい場合に特に効果があるが、差が大きくなっても性能の逆転は見られない。また、通常はこの差は比較的小さい (1-5) ので、本方式も有効な手法であると考えられる。

4　配列再分散における通信コスト削減

前章では、ハードウエアを用いた通信最適化手法について述べた。しかし、一般の分散メモリ計算機においては、必ずしもそのようなハードウエアを仮定できない。ここでは、集団通信の中でもコストの大きい配列再分散に注目し、ソフトウエアによる通信コスト削減方法について考える。

4.1　螺旋配置による通信コスト削減

前述の通り、ADI 等のアプリケーションでは配列の再分散は必須なものであり、その通信コストが大きいものの一つである。この再分散のコストを削減するために、仮想プロセッサ (概念プロセッサ) を物理プロセッサ (実プロセッサ) に効率良く配置する方法を提案する [43]。

4.1.1　配列再分散

前章での述べたように、ADI をはじめとするいくつかのアプリケーションは実行時の配列の再分散が必要である。

以下の説明に用いるいくつかの用語を定義する。

- 概念プロセッサ (**Conceptual processor**)：ある部分配列を含む領域は、ある概念プロセッサに対応する。80×80 の 2 次元配列が 4×4 の部分領域に (BLOCK, BLOCK) で分割されるとき、0 番目の概念プロセッサは [1:20,1:20] の部分配列を含む (0,0) 番目の部分領域を持ち、4 番目の概念プロセッサは [1:20,21:40] の部分配列を含む (0,1) 番目の部分領域を持つ。
- 実プロセッサ (**Real processor**)：実プロセッサとは物理プロセッサである。4×4 の 2 次元トーラスネットワークをもつマルチコンピュータシステムにおいて、0 番目の実プロセッサは 1, 3, 4 および 12 番目の実プロセッサと接続されている。
- サブプロセッサグループ：隣接するプロセッサからなるプロセッサの部分集合。実サブプロセッサグループは、いくつかの実プロセッサからなる。概念サブプロセッサグループは、いくつかの概念プロセッサからなる。
- ローカル / グローバル通信 (**Local/Global communication**)：もし、サブプロセッサグループ内のプロセッサが同じサブプロセッサグループ内の他のプロセッサと通信する場合、

その通信はローカルと呼ぶ。

一般にローカル通信はグローバル通信よりも効率的である。というのは、ローカル通信は他のサブプロセッサグループ内の通信を邪魔しないからである。

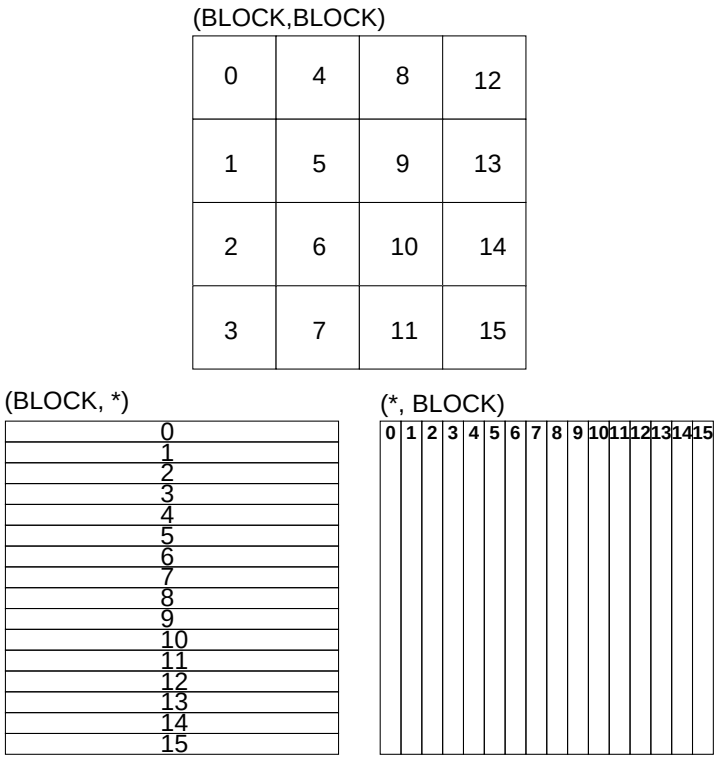


図 40: 2 次元配列における概念プロセッサ配置

2 次元のトーラスネットワークによって互いに連結された 16 個のプロセッサを考える。図 40に示すように、2 次元配列は (BLOCK,BLOCK), (BLOCK,*) および (*, BLOCK) の 3 種類の方法で概念プロセッサ上に配置される。

例えば、2 次元配列 が 80×80 の時、 0 番目の概念プロセッサは、(BLOCK, BLOCK) 分散では [1:20,1:20] の部分配列を持ち、(BLOCK, *) 分散では、 [1:5,1:80] の部分配列を持ち、(*, BLOCK) 分散では、 [1:80,1:5] の部分配列を持つ。

通常、概念プロセッサ番号と実プロセッサ番号を一致させて配列するが、この通常の配置を用いた配列再分散はグローバルな通信を引き起こす。

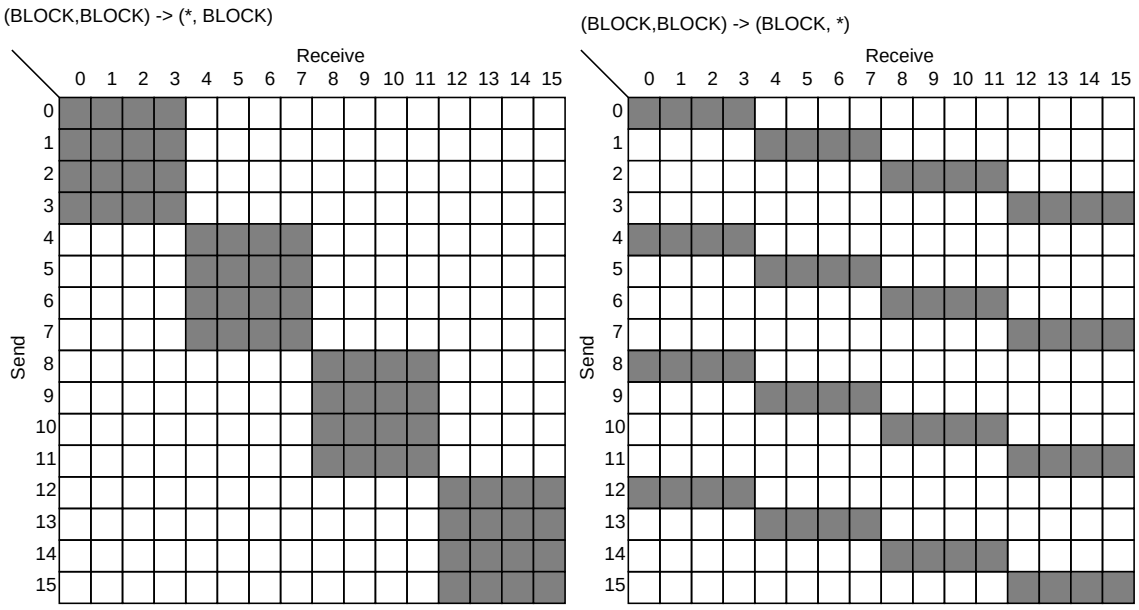


図 41: 2 次元配列再分散の通信パターン

これを図 41を用いて説明する。(BLOCK, BLOCK) と (*, BLOCK) 間の配列再分散は、ローカル通信のみ生成する。言い換えると、受信データの送信元の実プロセッサの集合は、送信先の実プロセッサの集合と同じである。しかし、(BLOCK, BLOCK) と (BLOCK,*) 間の再分散はローカル通信ではない。例えば、(BLOCK, BLOCK) から (BLOCK,*) に再分散をする場合、0 番目の実プロセッサは 1,2,3 番目の実プロセッサに送信するが、4,8,12 番目の実プロセッサから受信する。

一般には、配列再分散は通信パターンに局所性がないので、システム内での通信コンフリクトを増加させ、アプリケーションの性能劣化を引き起こす。

4.1.2 実プロセッサの螺旋配置

前節で紹介した通常配置は、再分散のための通信の局所性を妨害する場合がある。そこで、通信コンフリクトを軽減するために、他の実プロセッサ配置方法を検討しなければならない。効率的な配置のために、いくつかの制約を満たす必要がある。

- 任意の再分散の通信は実サブプロセッサグループ内で閉じているべきである。
  この制約は通信コンフリクトを減少させる。これを満たしていない場合、例えば、通常配置での (BLOCK, BLOCK) から (BLOCK, *) の再分散はグローバル通信を必要とする。
- 隣接する概念サブプロセッサグループ間の任意の境界プロセッサは、隣接する実プロセッサであるべきである。
  異なる概念サブプロセッサグループ間の通信はローカルではないが、この制約が満たされている限り、コストは小さくなる。
- 概念サブプロセッサグループ内の任意の 2 つの概念プロセッサ間の通信は、必ずしも、隣接する実プロセッサ間の通信ではない。
  この通信は、実サブプロセッサグループ内で閉じた通信である。よって、隣接実プロセッサ間通信である必要はない。

図 42 は上記の条件を説明したものである。

上記の条件を満たすために、螺旋上に畳み込んだ配置方法 (簡単に、螺旋配置と呼ぶ) を提案する。

2 次元配列における螺旋配置　まず、2 次元配列に対する螺旋配置を定義する。$N \times M$ 概念プロセッサ (実プロセッサも $N \times M$ である) を考える。

- (BLOCK, BLOCK) 分散している配列に対して、概念プロセッサは実プロセッサにそのまま対応する。例えば、0 番目の実プロセッサは 0 番目の概念プロセッサに一致する。
- (*, BLOCK) 分散している配列に対して、概念プロセッサは、N 個の概念プロセッサからなる M 個の概念サブプロセッサグループに分割される。偶数である i 番目の概念サブプロセッサグループにおいて、j番目の概念プロセッサは、$i \times N + j$番目の実プロセッサである。また、奇数である i 番目の概念サブプロセッサグループにおいて、j番目の概念プロセッサは、$i \times N + N - 1 - j$番目の実プロセッサである。
- (BLOCK,*) 分散している配列に対して、概念プロセッサは、M 個の概念プロセッサからなる N 個の概念サブプロセッサグループに分割される。偶数である i 番目の概念サブプロセッサグループにおいて、j番目の概念プロセッサは、$j \times N + i$ 番目の実プロセッサである。また、奇数である i 番目の概念サブプロセッサグループにおいて、j番目の概念プロセッサは、$(M - 1 - j) \times N + i$ 番目の実プロセッサである。

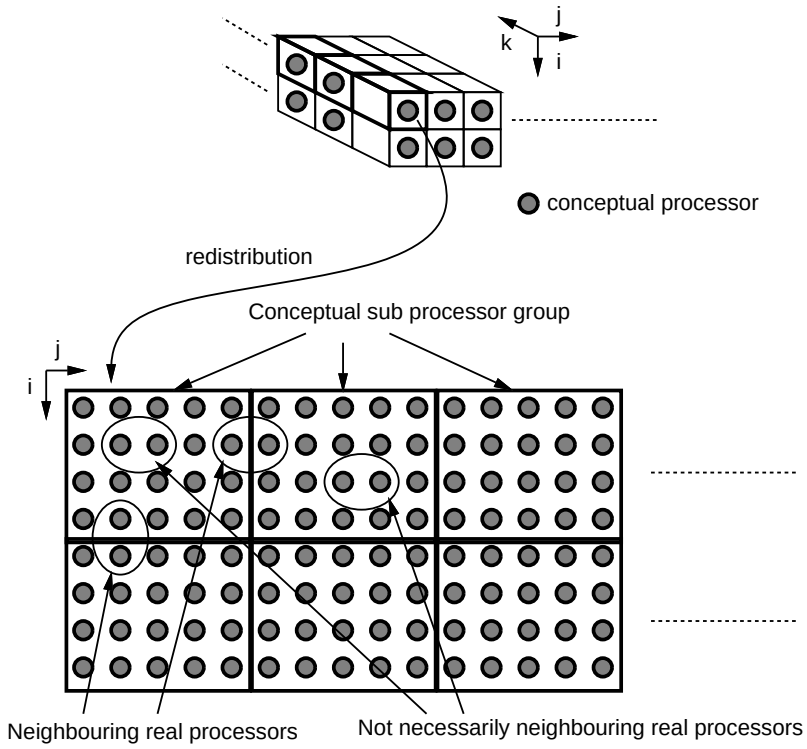


図 42: 概念プロセッサを実プロセッサに割り当てるための制約条件

図 43 は、16 実プロセッサを用いた配置方法を示している。

図 44に示してあるように、螺旋配置を用いた場合、いかなる再分散パターンに対しても通信は、概念サブプロセッサグループ内で閉じたものになる。

(BLOCK, BLOCK) と (*, BLOCK) 間の配列再分散において、0 番目の概念サブプロセッサグループは、0,1,2,3 番目の実プロセッサからなり、そのサブプロセッサグループ内だけで通信する。0 番目と 1 番目の概念サブプロセッサグループの境界プロセッサは、3,4 番目の実プロセッサであり、隣接プロセッサである。

一方、(BLOCK, BLOCK) と (BLOCK,*) 間の配列再分散において、0 番目の概念サブプロセッサグループは、0,4,8,12 番目の実プロセッサからなり、そのサブプロセッサグループ内だけで通信する。0 番目と 1 番目の概念サブプロセッサグループの境界プロセッサは、12,13 番目の実プロセッサであり、さきほどと同様に、これらは隣接プロセッサである。

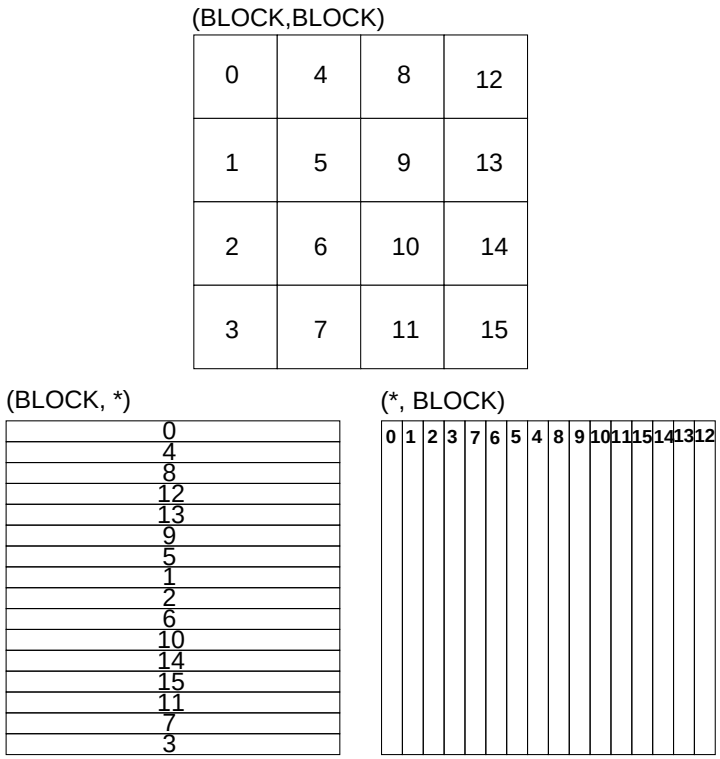


図 43: 2 次元配列に対する螺旋配置

3 次元配列に対する螺旋配置　次に、3 次元配列に対する螺旋配置を定義する。$N \times M \times L$ の概念プロセッサ ($N \times M \times L$ 実プロセッサ) を考える。

- (BLOCK, BLOCK, BLOCK) 分散している配列に対して、概念プロセッサは、そのまま実プロセッサに対応する。例えば、0 番目の実プロセッサは、0 番目の概念プロセッサである。
- (*, BLOCK, BLOCK) 分散している配列に対して、概念プロセッサは、$N1 \times N2 (= N)$ 個の概念プロセッサからなる $M \times L$ 個の概念サブプロセッサグループに分割される。

  (j, k) 番目 (j は偶数 k は偶数) の概念サブプロセッサグループにおいて、$j \times N + k \times N \times M + i$ 番目の実プロセッサ $(\forall i)$ は、$(0, 0)$ 番目の概念プロセッサから反時計回りに螺旋上に配置されていく。

  (j, k) 番目 (j は奇数 k は偶数) の概念サブプロセッサグループにおいて、$j \times N + k \times N \times M + i$ 番目の実プロセッサ $(\forall i)$ は、$(N1 - 1, 0)$ 番目の概念プロセッサから時計回りに螺旋上に配置されていく。

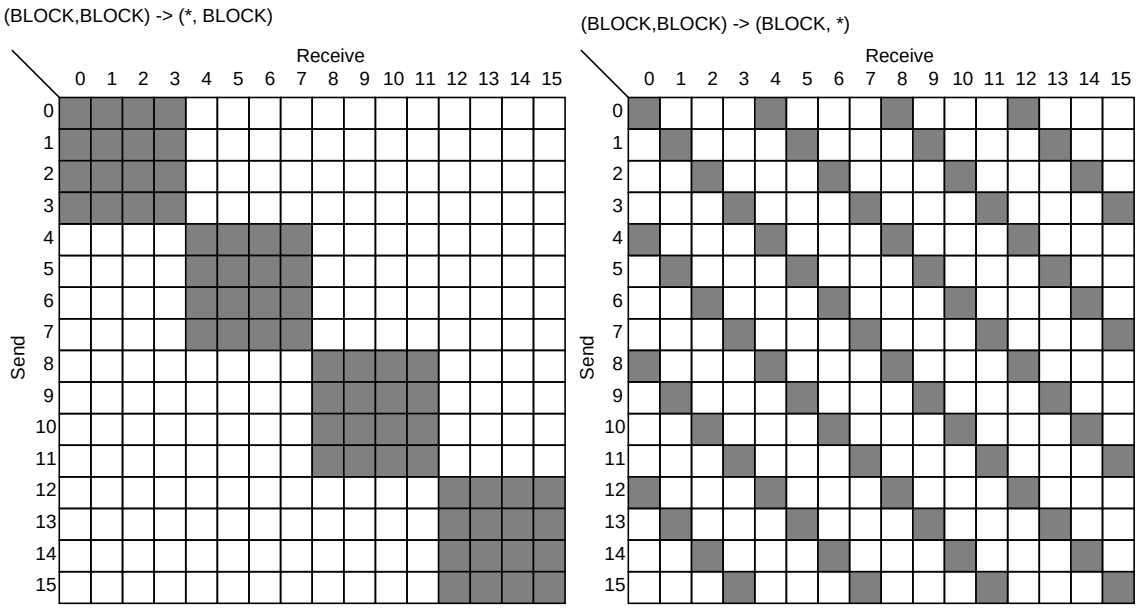


図 44: 2 次元配列再分散の螺旋配置を用いた通信パターン

(j,k) 番目 (j は偶数 k は奇数) の概念サブプロセッサグループにおいて、$j \times N + k \times N \times M + i$ 番目の実プロセッサ $(\forall i)$ は、$(0, N2-1)$ 番目の概念プロセッサから時計回りに螺旋上に配置されていく。

(j,k) 番目 (j は奇数 k は奇数) の概念サブプロセッサグループにおいて、$j \times N + k \times N \times M + i$ 番目の実プロセッサ $(\forall i)$ は、$(N1-1, N2-1)$ 番目の概念プロセッサから反時計回りに螺旋上に配置されていく。

- $(\mathrm{BLOCK},^* ,\mathrm{BLOCK})$ に分散されている配列に対して、概念プロセッサは、$M1 \times M2 (= M)$ 個の概念プロセッサからなる $L \times N$ 個の概念サブプロセッサグループに分割される。$(^*, \mathrm{BLOCK}, \mathrm{BLOCK})$ 分散の場合と同様に、$j \times N + k \times N \times M + i$ 番目の実プロセッサ $(\forall j)$ は、螺旋上に配置される。
- $(\mathrm{BLOCK},\mathrm{BLOCK},^*)$ に分散されている配列に対して、概念プロセッサは、$L1 \times L2 (= L)$ 個の概念プロセッサからなる $N \times M$ 個の概念サブプロセッサグループに分割される。$(^*, \mathrm{BLOCK}, \mathrm{BLOCK})$ 分散の場合と同様に、$j \times N + k \times N \times M + i$ 番目の実プロセッサ $(\forall k)$ は、螺旋上に配置される。
- 分散している次元が 1 の場合は、上記の配置方法と 2 次元の場合との組み合わせとなる。

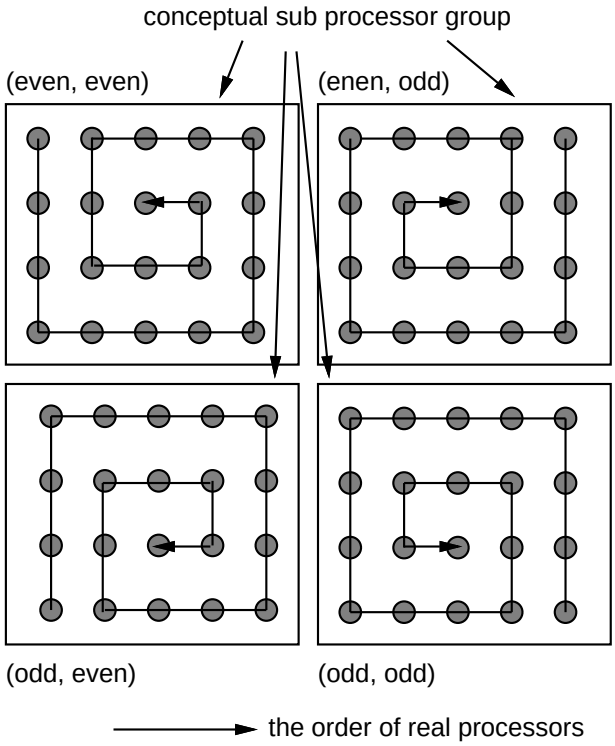


図 45: 3 次元配列に対する螺旋配置

図 45は、4 つのケースの概念サブプロセッサグループの配置方法を示している。

なお、(偶数, 偶数) 概念サブプロセッサグループの右側の境界の概念プロセッサと (偶数, 奇数) 概念サブプロセッサグループの左側の境界の概念プロセッサは、隣接する実プロセッサであり、他の境界の概念プロセッサは隣接する実プロセッサである。

図 46は、$4 \times 4 \times 4$ プロセッサでの螺旋配置を示す。

(BLOCK, BLOCK, BLOCK) と (*, BLOCK, BLOCK) 間の再分散において、(0,0) 番目の概念サブプロセッサグループは、プロセッサ 0, 1, 2 および 3 からなり、そのサブグループ内でのみ通信する。(0,1) 番目の概念サブプロセッサグループは、プロセッサ 16,17,18,19 からなりそのサブグループ内でのみ通信する。(0,0) 番目および (0,1) 番目の概念サブプロセッサグループの境界プロセッサは、3 番目および 19 番目の実プロセッサであり、隣接している。また、同様に、2 番目および 18 番目の実プロセッサも隣接している。

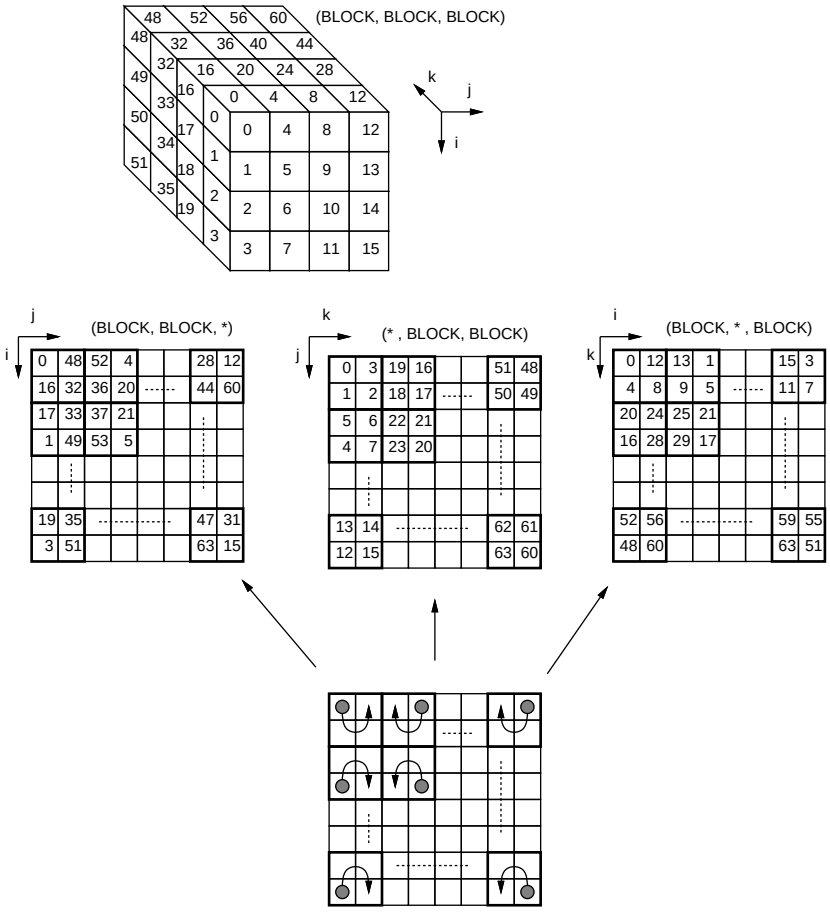


図 46: 64 プロセッサにおける 3 次元配列の螺旋配置

(BLOCK, BLOCK, BLOCK) と (BLOCK, BLOCK,*) 間の再分散において、(0,0) 番目の概念サブプロセッサグループは、プロセッサ 0,16,32 および 48 からなり、そのサブグループ内でのみ通信する。(0,1) 番目の概念サブプロセッサグループは、プロセッサ 4,20,36,52 からなりそのサブグループ内でのみ通信する。(0,0) 番目および (0,1) 番目の概念サブプロセッサグループの境界プロセッサは、48 番目および 52 番目の実プロセッサであり、隣接している。また、同様に、32 番目および 36 番目の実プロセッサも隣接している。

図 47 $4 \times 4 \times 2$ 個のプロセッサでの螺旋配置を示している。

先程の例のように、隣接する概念プロセッサは隣接する実プロセッサであり、再分散のための通信は、概念サブプロセッサグループ内で閉じている。

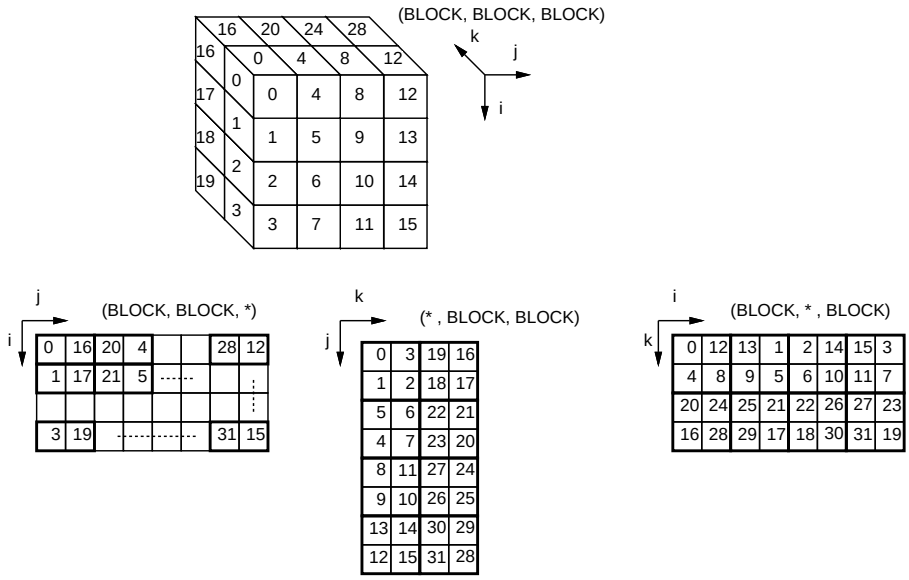


図 47: 32 プロセッサにおける 3 次元配列の螺旋配置

(BLOCK, BLOCK, *) と (*, BLOCK, BLOCK) 間の再分散は、2 通りの方法で実行できる。一つは、直接的にグローバル通信を用いる方法で、もう一方は、再分散を 2 回繰り返すものである。すなわち、後者の方法は、(BLOCK, BLOCK, *) から (BLOCK, BLOCK,BLOCK) への再分散を行った後、再度、(*, BLOCK,BLOCK) へ再分散するものである。この方法は、概念サブプロセッサグループ内で閉じた通信だけで構成できるものの、ストアアンドフォワード通信となり、コストが高くなる場合もある。いずれの方法を取るべきかは、使用するハードウエアのパラメータに依存する。

4.1.3 関連技術との比較

実プロセッサへの配置方法の工夫により実効性能向上をめざしたものとして、進藤らはデータスキューイングを分散メモリ計算機に拡張したデータ分散法 Twisted Data Layout[56] を提案している。twisted data layout では、多次元配列の分散において、次元毎にずらして 1 次元実プロセッサに割り当てる。これによって、任意の次元に対してループ並列を有効化できる。したがって、Eispack の ELMEHES ルーチンのように、配列の再分散を行なわないと並列化できなかったようなアプリケーションに対しても、twisted data layout によって並列化可能である。

twisted data layout は、仮想プロセッサの実プロセッサへの割り当て方法を工夫することによって実効パフォーマンスの向上を図る点では、本螺旋配置とは同一である。しかし、ADI の基本ルーチンのように配列の再分散が避けられないようなアプリケーションに対しては、螺旋配置を用いることによる通信コスト低減が必要となる。

4.1.4 実験

この節では、通常配置と螺旋配置を用いて再分散の実行時間測定結果を示す。

次の簡単な HPF プログラム片を考える。

```
      real*8,dimension(n,n) ::A
c$hpf distribute (BLOCK,BLOCK),dynamic :: A
        :
c$hpf redistribute (BLOCK,*) :: A
        :
```

まず、配列 A は、(BLOCK, BLOCK) に分散されているとする。redistributeデイレクテイブは、元の分散形態から (BLOCK, *) に再分散させる。

この再分散ライブラリを 16 ノードの iPSC/860 上にインプリメントし、速度を計測した。それぞれの時間は、再分散を 100 回繰り返した時の経過時間である。結果を表 1に示す。

matrix size	(BLOCK,BLOCK)→(BLOCK,*)		(BLOCK,BLOCK)→(*,BLOCK)	
	normal	spiral	normal	spiral
16 × 16	6.77E-002	6.33E-002	5.45E-002	5.89E-002
64 × 64	0.244	0.161	0.158	0.155
256 × 256	1.65	1.17	1.12	1.25
1024 × 1024	26.3	19.3	19.3	19.3

表 1: 16 ノードの iPSC/860 における配列再分散 100 回の実行時間 (秒)

予想されるように、(BLOCK, BLOCK) と (*,BLOCK) 間の再分散に関しては、実プロセッサの配置の違いが実行時間の大きな違いにはなっていない。というのは、通常配置も螺旋配置もほと

んど同じで、概念サブプロセッサグループの境界が螺旋配置では隣接した実プロセッサであるのに対し、通常配置では隣接していない点が異なるだけである。また、概念サブプロセッサグループのメンバーはいずれの配置でも同一であるので、この違いは再分散の実行時間に影響をあたえることはない。

一方、(BLOCK,BLOCK) から (BLOCK,*) への再分散時間の差は顕著である。1024×1024 の配列に対して、通常配置での再分散時間は、螺旋配置に比べて 1.4 倍になっている。送信受信のメッセージの数は同一であるので、実行時間の差は通信の局所性の差に起因していると考えられる。

4.2 メッセージ掘り下げ法 (message strip-mining)

前節で述べたように、概念プロセッサを物理プロセッサにマッピングする手法 (spiral mapping) によって、再分散が必要とする大域通信を近接通信に変換できた。しかし、通信時間自体は陽に残っている。

本節で述べる手法は、計算および通信において多次元配列の 1 次元部分を最小単位として分割し、その最小単位の通信と計算部分とがオーバーラップできるアプリケーションに対しては、その配列再分散の通信レイテンシを隠蔽できるものである。これをメッセージ掘り下げ法 (message strip-mining) と呼ぶ

次節で、通常の再分散方式 (massive redistribution) とメッセージ掘り下げ法を用いた再分散方式 (strip mining redistribution) との違いについて述べる [42]。

4.2.1 ADI における配列再分散

従来の再分散方式 (Massive Redistribution) 4 プロセッサ上における 2 次元 100×100 サイズの配列を用いた ADI プログラムを考える。ADI プログラムの部分を 図 48 に示す。配列は再分散される前は (Block, *) で分散されており、(*, Block) に再分散される。

(*, Block) への 配列 `uy` の再分散の後に、その配列を用いた計算が続いている。つまり、その再分散は計算が始まる前に終了していなければならない。

再分散において、それぞれのプロセッサは、分散されている配列の 1/4 を残りの 3 つのプロセッサに送り、再分散された配列を構成するために 1/4 ずつを受け取る。

```
      real ux(100,100),uy(100,100)
!hpf$ processors q(100)
!hpf$ distribute(Block,*)
     +           onto q, dynamic::uy
!hpf$ distribute(*,Block)
     +           onto q, dynamic::ux
      real a(n),b(n)
        :
!hpf$ redistribute(*,Block) onto q::uy
      do j=1,100
       a(1)=0.0
       b(1)=0.0
       do i=2,100
        x=2.0-a(i-1)
        a(i)=1/x
        b(i)=(b(i-1)-uy(i,j))/x
       end do
       ux(100,j)=1.0
       do i=99,1,-1
        ux(i,j)=a(i)*ux(i+1,j)+b(i)
       end do
      end do
      end
```

図 48: ADI プログラム (部分)

以下の部分で、この再分散方式を 従来方式 (massive redistribution) と呼ぶ。

再分散する前のそれぞれの物理プロセッサは、2500 個の配列要素をもっている。従来方式でのメッセージのサイズは、それぞれ、$625(= \frac{25\times100}{4})$ である。

メッセージ掘り下げ法による再分散方式 (Strip Mining Redistribution) 本方式は、2 つの部分からなる。すなわち ブロッキング通信部分及び非ブロッキング通信部分である。計算部分は、分散されている j のループである。それぞれのプロセッサは、j ループの分散された一部分のみ計算する。この局所化されたループにおいて、物理プロセッサは、いくつかの仮想プロセッサ (template) をエミュレートする。それぞれの仮想プロセッサは、他の仮想プロセッサから独立な配列部分を使用する。ある物理プロセッサが第一の仮想プロセッサをエミュレートしている場合、つまり、ローカル変数 j が 1 の場合、その物理プロセッサは第一の仮想プロセッサが使用

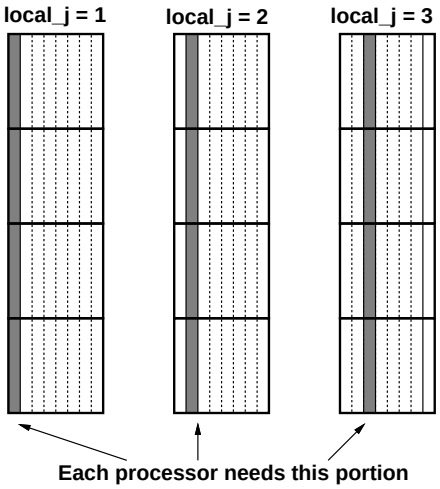
する第一の部分配列が必要となる。第二の仮想プロセッサは、第二の部分配列のみを使用する。以下同様である。

よって、本方式のタイムチャートは、表 2 に示す通りである。

Time slot	local_j	Computation	Redistribution
0	none	none	The 1st portion (blocking)
1	1	The 1st portion	The 2nd portion (non-blocking)
2	2	The 2nd portion	The 3rd portion (non-blocking)
:	:	:	:

表 2: 本方式における計算と配列再分散のタイミング

再分散の 2 番目以降の部分は非ブロッキング通信なので、通信起動および通信終了同期の部分を除いて、通信のほとんどは計算と並行に処理できる。本方式を用いた ADI の疑似プログラムを図 49に示す。




```
 call blocking_redist
&         (uy,1st portion)
 do local_j = 1, 24
  call non_blocking_redist(uy, j+1-th portion)
  computation using array uy (j-th portion)
  call wait
 end do
 computation using array uy (25-th portion)
```

図 49: 並列化された j ループとメッセージ掘り下げ法を用いた疑似コード

物理プロセッサが他のプロセッサに送るべき分割された配列のサイズは 25 であり、時間スロット 1 の間に送るべきプロセッサの総数は 3 である。したがって、ブロッキング通信に対しては、それぞれの物理プロセッサは (3×25) 個の配列要素を送信しなければいけない。

これに反して、非ブロッキング通信の多くは、計算に隠蔽されると期待される。したがって、この部分の通信時間は無視できる。それぞれの物理プロセッサがエミュレートする総数は$25(=\frac{100}{4})$である。

従来方式の再分散が (3×625) 個のメッセージを必要としていたのに対し、本方式の場合は、ブロッキング部分を除いて多くの通信が隠れるが、従来方式では 3 個であった、ライブラリ起動の回数が (25×3) 個となり、通信起動オーバーヘッドが増加する。

次に、メッセージ掘り下げ再分散方式が、従来方式に対して優位性を保持するための条件を見つけよう。

これに関する議論の前に、いくつかの言葉を定義する。

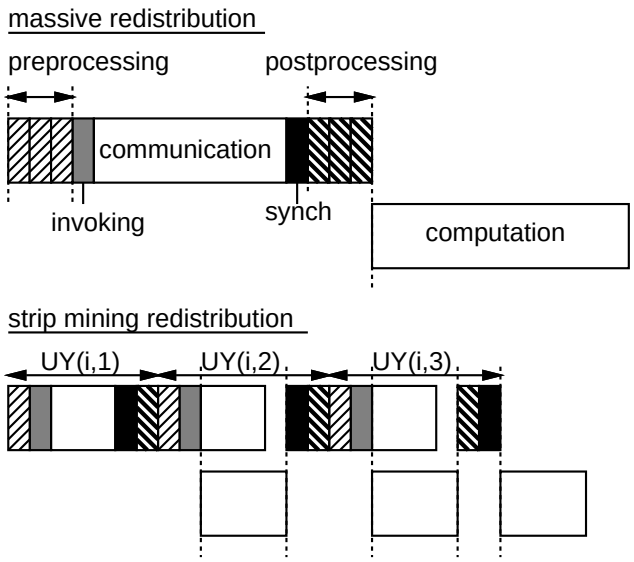


図 50: 従来の再分散方式とメッセージ掘り下げ法再分散方式

図 50 が示すように、再分散のモデルは、次の 5 つのパートからなる。

前処理 *(preprocessing):* 物理プロセッサは特定のプロセッサに送るべき配列要素をひとつのメッセージバッファに連続的に並べ直さなければならない。この前処理に要する時間は配列要素の数に比例する。

通信起動 *(invoking):* 物理プロセッサは通信を起動するためにライブラリを起動する。ライブラリ起動のための時間は、送信すべき相手先プロセッサの総数 (メッセージ数) に比例する。

通信 *(communication):* メッセージルータが送信し、中継して、受信するのにかかる時間である。この時間は、メッセージの長さに比例する。

同期 *(synch):* 物理プロセッサは通信終了を確認するためにを同期処理を行なう。このための時間は、送信すべき相手先プロセッサの総数に比例する。

後処理 *(postprocessing):* 物理プロセッサは、受信メッセージバッファから受信用配列にコピーする必要がある。この後処理に要する時間は配列要素の数に比例する。

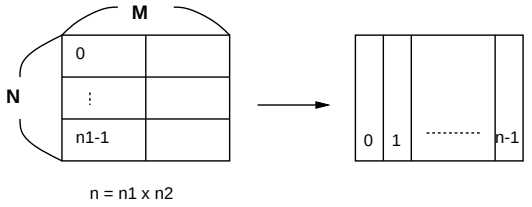
1 ワードの通信を完了するのに要する平均時間を u と表す。すなわち u は、各配列要素に対する前処理、通信、後処理時間の総和である。さらに、p は各配列要素に対する前処理と後処理時間の総和、ST は各メッセージに対する起動と同期のための時間の総和を表す。

4.2.2 有効性評価

ADI のように、再分散に引き続く計算部分がいくつかの最小単位に分割できるアプリケーションに対しては、その最小単位に合わせて通信を分割し、この分割された通信と計算をスキューイングすることによって通信レイテンシが隠蔽できる。

このように、ある条件のもとでは、メッセージ掘り下げ法再分散方式は、従来方式に対して優位性を持っている。この節では、様々な配列のタイプ、分散の方法に対する条件を示す。本方式は分散されたループを使用することを仮定しており、そこで、従来方式で用いられていたメッセージが、テンプレート (template) に依存したサイズの小さなメッセージ群に分割される。例えば、2 次元配列が (Block, Block) に分割されている時、各テンプレートのサイズは 1×1(=スカラー) である。したがって、分割されたメッセージのサイズは 1 である。明らかに、この場合、メッセージ掘り下げ法再分散方式は非効率である。というのは、送るべき全ての配列要素毎に通信起動のコストがかかるからである。一方、2 次元配列が (*, Block) に再分散される時、それぞれのテンプレートに対するサイズは $1 \times row\_size$ である。それぞれのメッセージには先程よりもより多くの要素が含まれているので、本方式は効果的に機能する。それゆえ、再分散される配列の少なくとも 1 つの次元は、分散されていない (collapsed) ことが必要である。

これ以降の部分では、この条件を満たすもののみを扱う。また、プロセッサは、一度には一つの特定プロセッサに対してのみメッセージが送出できると仮定する。すなわち、プロセッサは複数受信先を指定できないとする。




```
      parameter(N=..., M=...)
      parameter(n1=..., n2=...)
      parameter(n=n1*n2)
      real uy(N,M)
!hpf$ processors p(n1,n2)
!hpf$ processors q(n)
!hpf$ distribute(block block)
     +     onto p, dynamic::uy
        :
!hpf$ redistribute(*,block) onto q::uy
      <computation using uy>
```

図 51: 2 次元配列の再分散の例

2 次元配列　サイズが $N \times M$ で、再分散前は $(\mathrm{BLOCK}, \mathrm{BLOCK})$ に分散されており再分散後は $(\mathrm{BLOCK}, *)$ に分散されるような 2 次元配列を考える。この仮定は容易に $(*, \mathrm{BLOCK})$ の分散形態に拡張できる。

総プロセッサ数は n と仮定する。配列は、再分散前は $n_1 \times n_2$ 個のプロセッサ上で分散されており、再分散後は $n\,(= n_1 \times n_2)$ 個のプロセッサ上に分散される。ここで、 n_1 は、再分散後は分散されない次元の再分散前の分割数である。分割された配列の構成図を図 51に示す。

なお、 n_2 が 1 の時は、再分散前の分散形態は $(\mathrm{BLOCK},*)$ と同じである。

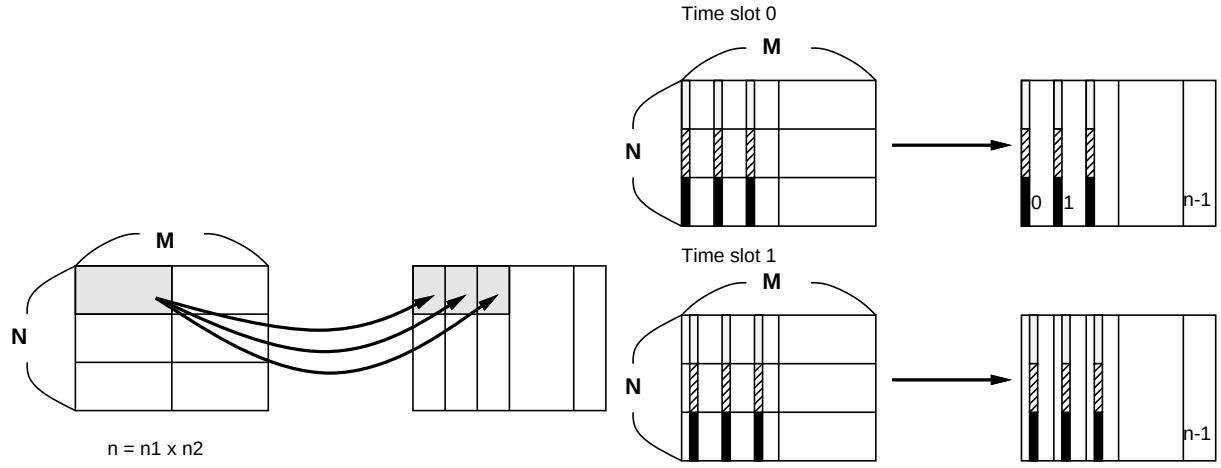
従来方式が要する実行時間は次のようになる。再分散前、それぞれの物理プロセッサは$\frac{N \times M}{n}$ 個の配列要素を持っている。この配列要素は n_1個に分割され、それぞれ $(n_1 - 1)$ 個のプロセッサ[4]に送信されるはずである。

$$massive\_redistribution = (ST + \frac{N \times M \times u}{n \times n_1}) \times (n_1 - 1) \tag{6}$$

これに反して、前の章で述べたように、メッセージ掘り下げ法による再分散は 2 つの部分、すなわち、ブロッキング通信と非ブロッキング通信からなる。

ある物理プロセッサが他のプロセッサに送るべき部分配列のサイズは$\frac{N}{n_1}$であり、時間スロット 1 に送るべき相手先プロセッサの数は $(n_1 - 1)$ である。よって、メッセージ掘り下げ再分散のブロッキング通信の時間は次のようになる。

[4] 分割された部分配列の 1 つは送信されない。それは局所的に使用されるからである。



(a) 従来方式　　(b) メッセージ掘り下げ法

図 52: 従来方式とメッセージ掘り下げ法再分散方式

$$blocking = (ST + \frac{N}{n_1} \times u) \times (n_1 - 1) \tag{7}$$

図 50に示すように、非ブロッキング通信の多くは計算に隠蔽されると考えられる。この通信に対しては、前処理、後処理、通信起動、同期の時間は考慮しなくてはいけないが、通信自体の時間は無視できる。よって、非ブロッキング通信の時間は次のようになる。

$$non\_blocking = (ST + \frac{N}{n_1} \times p) \times (\frac{M}{n} - 1) \times (n_1 - 1) \tag{8}$$

つまり、式 (6) 、(7) 及び (8) から、次の条件を満たすときに、メッセージ掘り下げ再分散方式は従来方式よりも効率がよくなる。

$$N > \frac{ST \times n_1}{u - p} \tag{9}$$

なお、$u - p$ は、各配列要素を通信するのに必要な時間である。言い換えると、メッセージ掘り下げ法による再分散の優位性は配列サイズ (N) 及び再分散後には分散されていない方の次元の分割数 (n_1) に依存する。

3 次元配列　3 次元配列に対しては 2 つのケースに対して考察する必要がある。第一に、再分散後に 1 次元だけが分散されていない (collapsed) 場合で、第二には 2 次元が分散されていない場合である。

1 次元だけが分散されていない場合　再分散前は (BLOCK, BLOCK, BLOCK) に分散されており、再分散後に (*, BLOCK, BLOCK) に分散されるようなサイズ $N \times M \times L$ である 3 次元配列を考えよう。なお、(BLOCK,* ,BLOCK) 及び (BLOCK, BLOCK, *) の分散形態でも同様の議論ができる。

総プロセッサ数は n と仮定する。最初、その配列は $n_1 \times n_2 \times n_3$個のプロセッサ上に分散されており、$m_1 \times m_2$ $(m_1 = n_{11} \times n_2, m_2 = n_{12} \times n_3, n_1 = n_{11} \times n_{12})$ 個のプロセッサに再分散される。n_1は再分散後には分割されていないが、再分散前に分割されていた次元の分割数である。

なお、n_2 が 1 の場合、再分散前の分散形態は (BLOCK, *, BLOCK) と一致する。同様に、n_2 が 1 で n_3 が 1 の場合は、分散は (BLOCK, *, *) と一致する。

12 個のプロセッサを用いた 3 次元問題の構成は図 53に示してある。ここでは通信競合を防ぐために前章で紹介した螺旋配置を使用している。

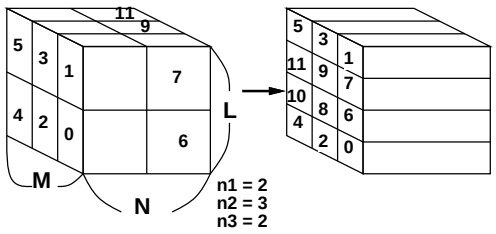
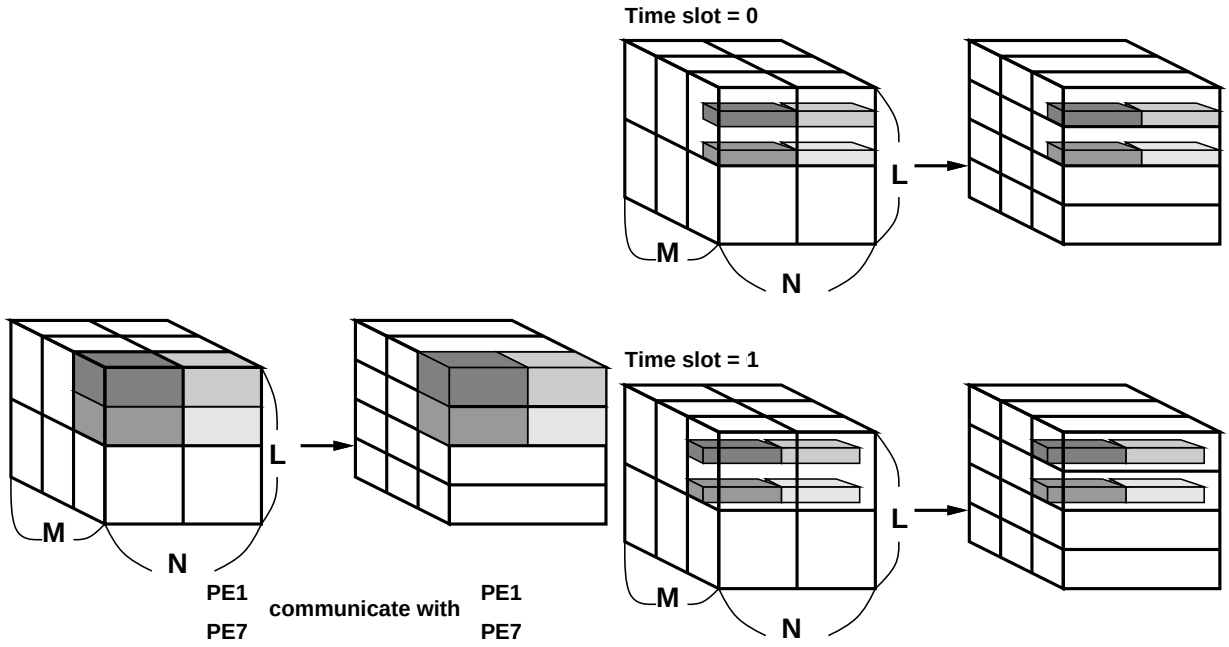


図 53: 12 プロセッサを用いた 3 次元配列再分散

図 54に示しているように、プロセッサ 1 はプロセッサ 7 と、プロセッサ 7 はプロセッサ 1 と通信する。2 次元の場合と同様に、従来方式の分散時間は次のように見積もれる。再分散前、それぞれの物理プロセッサは$\frac{N \times M \times L}{n}$ 個の配列要素を持っており、それらは n_1個の部分配列に分割されて $(n_1 - 1)$ 個のプロセッサに送られる。

$$massive\_redistribution = (\mathrm{ST} + \frac{\mathrm{N} \times \mathrm{M} \times \mathrm{L}}{\mathrm{n} \times \mathrm{n}_1} \times \mathrm{u}) \times (\mathrm{n}_1 - 1) \tag{10}$$



(a) 従来方式　　(b) メッセージ掘り下げ法

図 54: 従来方式とメッセージ掘り下げ法再分散方式

前節で書いたのと同様に、メッセージ掘り下げ法による再分散は 2 つの部分から構成されている。

ある物理プロセッサから別のプロセッサに転送される部分配列のサイズは$\frac{N}{n_1}$であり、時間スロット 1 に送るべき相手先プロセッサの総数は $(n_1 - 1)$ である。よって、メッセージ掘り下げ再分散方式のブロッキング通信部分の時間は次のようになる。

$$blocking = (ST + \frac{N}{n_1} \times u) \times (n_1 - 1) \tag{11}$$

非ブロッキング通信の多くは計算に隠蔽されると考えられる。また、各物理プロセッサに対する仮想プロセッサの数は$\frac{M \times L}{n}$ である。よって、非ブロッキング部分の時間は前処理、後処理、通信起動及び同期から成り、非ブロッキングメッセージを送るべきプロセッサの数は $(n_1 - 1)$ なので、通信時間は次のように表わされる。

$$non\_blocking = (n_1 - 1) \times (ST + \frac{N}{n_1} \times p) \times (\frac{M \times L}{n} - 1) \tag{12}$$

よって、式 (10)、(11) 及び (8) から、メッセージ掘り下げ法による再分散は、次の条件を満たす場合に、従来方式よりも効率的になる。

$$N > \frac{ST \times n_1}{u - p}. \tag{13}$$

言い換えると、メッセージ掘り下げ再分散方式の優位性は、プロセッサ数、再分散後に 分散されない次元に対する分割数およびそのサイズに依存する。なお、この条件は、本質的に 2 次元の場合と同様なものである。

2 次元が分散されていない場合　次に、2 次元が分散されない 3 次元配列の場合を考察しよう。従来方式及びメッセージ掘り下げ再分散方式の通信パターンを図 55に示す。

プロセッサ 1 はプロセッサ 3、5、7 及び 11 と通信する。各物理プロセッサがエミュレートする仮想プロセッサの数は$\frac{L}{n}$である。従来方式及びメッセージ掘り下げ法による再分散方式の実行時間は次のようになる。

$$\begin{aligned}
massive_redistribution &= (ST + \frac{N \times M \times L \times u}{n \times n_1 \times n_2}) \times (n_1 \times n_2 - 1) && (14)\\
blocking &= (ST + \frac{N \times M}{n_1 \times n_2} \times u) \times (n_1 \times n_2 - 1) && (15)\\
non_blocking &= (ST + \frac{N \times M}{n_1 \times n_2} \times p) \times (\frac{L}{n} - 1) \times (n_1 \times n_2 - 1) && (16)
\end{aligned}$$

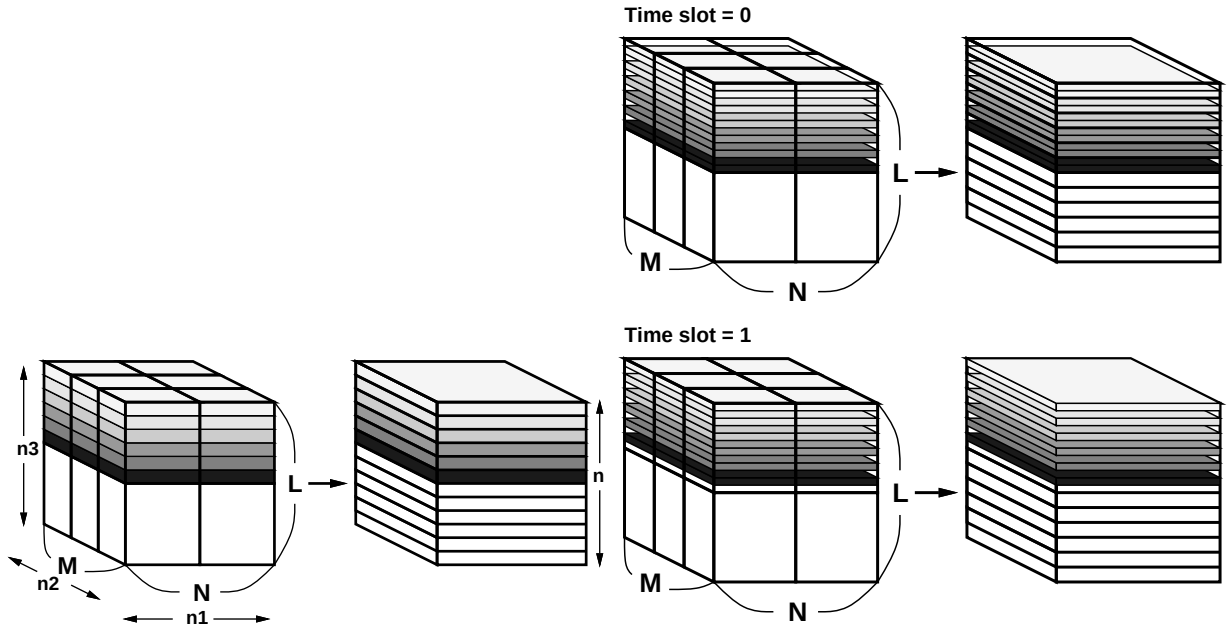
よって、メッセージ掘り下げ再分散方式の優位性の条件は次のようになる。

$$M \times N > \frac{ST \times n_1 \times n_2}{u - p} \tag{17}$$

なお、これまで示した不等式から、メッセージ掘り下げ再分散の優位性を保証する条件は、次のように一般化できる。

$$collapsed\_dimensions > \frac{\text{startup\_time} \times \text{num\_sub\_processors}}{\text{communication\_unit\_time}} \tag{18}$$

上の不等式では $collapsed\_dimensions$ は再分散後に分散していない次元のサイズを表わし、$num\_sub\_processors$ は $collapsed\_dimensions$ が再分散前にその上に分散していたようなプロセッサ数を表わす。



(a) 従来方式　　(b) メッセージ掘り下げ法

図 55: 従来方式とメッセージ掘り下げ法再分散方式

4.2.3 実験

前の章で導出した評価の妥当性を検証するために、従来方式とメッセージ掘り下げ再分散方式を用いた実行評価実験を行う。そのために 16 プロセッサの iPSC/860 上に 2 次元の ADI プログラムを実装した。配列の再分散を 10 回繰り返す毎に時間計測を行った。

図 56に配列サイズを変えた場合の従来方式及びメッセージ掘り下げ方式の再分散時間を示してある。配列サイズが 1050 以下の場合は、メッセージ掘り下げ再分散は、従来方式よりも多くの実行時間が要するが、それ以上になると事情は反転する。例えば、配列サイズが 2048×2048 の時、従来方式の再分散が $16.9sec$ であるのに対し、メッセージ掘り下げ再分散は $9.8sec$ である。すなわち、1.7 倍のスピードアップが得られていることがわかる。

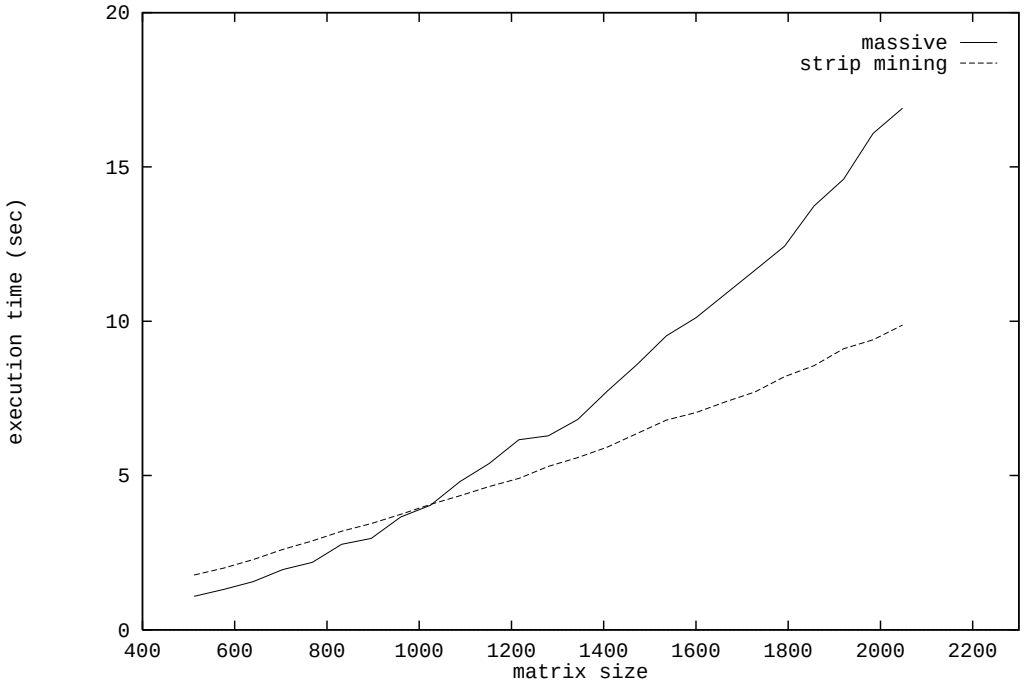


図 56: 16 ノードの iPSC/860 における配列サイズと配列再分散に要する時間比較 (秒)

4.2.4 考察

メッセージ掘り下げアプローチは、配列サイズに関する条件を満たしている限り、再分散の性能を向上させる。しかし、ここまでの解析及び実験が示すように、通信起動コストの起因する性能低下のために、本方式は比較的大きな配列にのみ適用できる。

本方式をより効果的にする方法がいくつか考えられる。現在のメッセージパッシング型マルチコンピュータの中には、J-machine[25] のように、比較的小さな通信起動コストをもったものがある。よって、そのようなシステムに対しては条件 (9) の $startup\_time$ は減少すると考えられる。この場合、本方式はより小さいサイズの配列の再分散にも適用できる。

2 番目の方法は集団通信を扱うハードウエアを利用するものである。本論文ではプロセッサは複数送信先を一つのメッセージの中に指定できないものを仮定していた。しかし、複数の送信先にメッセージを送出できるハードウエアがプロセッサ上にあれば通信起動コストは激減できる。その場合、条件 (9) は、不等式 (19) となる。

$$N > \frac{ST}{u-p} \times \frac{n_1}{n_1 - 1} \tag{19}$$

上記の不等式の右辺は不等式 (9) の右辺よりも小さいので、本方式はより小さい配列に適用できる。また、大きな配列に対しても実行性能はそのようなハードウエアを用いない場合より向上する。

最後の方法はメッセージ集団化 (message aggregation) である。つまり、同じ転送先に送るメッセージを 1 つのメッセージに融合することにより、通信起動コストは軽減できる。しかし、メッセージを融合するにつれて、メッセージ掘り下げ法による再分散の効果は低下する。これに関する議論は次章で考察する。

4.3 まとめ

本章では、まず配列再分散の通信コストを軽減するために実プロセッサの配置方法として螺旋配置について述べた。

従来の配置方法では、配列再分散は全プロセッサ空間内での多対多通信であったが、螺旋配置を用いると、配列再分散は分割されたサブプロセッサ空間内での多対多通信にリダクションでき、その通信コストを減少させることができる。例として、通常配置及び螺旋配置を用いた 2 次元配列の再分散ライブラリをインプリメントした。1024×1024 の配列に対し、螺旋配置を用いた再分散は、通常配置に比べて 36%速くなった。

次に通信レイテンシを軽減するために、通信と計算を分割し、それらをオーバーラップする手法、すなわち、メッセージ掘り下げ法について述べた。使用される配列のサイズに関する条件を満たす限り、メッセージ掘り下げ手法は配列の再分散の実行性能を向上させる。例えば、サイズが(2048×2048)の 2 次元 ADI プログラムの場合、本方式によって 1.7 倍のスピードアップが得られた。また、メッセージ集団化法やハードウエア支援を用いることにより、本方式をより効果的にすることが可能である。

5　一般のアプリケーションに対する通信コスト削減

掘り下げによってメッセージを分割すると通信レイテンシが隠蔽される一方、通信起動のオーバーヘッドが増加することにより性能劣化を引き起こす場合がある。本章では、メッセージ集団化法を用いてメッセージを再組み立てすることにより、適用できる問題範囲の拡大を図れることを示す。

また、前章では配列の再分散に対するメッセージ掘り下げ法を説明したが、本最適化はそれだけではなく、規則性の高い通信を持つ任意のアプリケーションに対して適用できるものである。ここでは、一般的なアプリケーションに対するメッセージ掘り下げ法の適用に関して述べ、適用例として配列の間接アクセスの通信に応用した場合を紹介する。間接アクセスは、本来、不規則通信であるが、検査官 / 執行官 (inspector/executor) 法を用いると、執行官フェーズにおける通信は規則性の高い通信に変更可能であり、メッセージ掘り下げ法を適用することが可能となる [41]。

5.1　基本構成

この節では、メッセージ掘り下げ法 (message strip-mining) にメッセージ集団化法を適用する場合の概略について説明する。

まず計算部分は、いくつかの最小部分に分割できるとする。前章で扱った ADI では、多次元配列の連続する 1 次元部分が最小単位として扱われた。この最小単位に対応する通信単位を 1 つのメッセージと考える。

受信したデータはその後の計算において使用されると仮定する。すなわち、i 番目のメッセージで受信されたデータは、i 番目の計算部分で使用されるとする。さらに、受信する通信単位[5] の数及びループの繰り返し回数を N とする。

図 57の上に、典型的な通信及び計算のタイムチャートを示してある。

通信モデルは前章の場合と同様に、前処理 (preprocessing) 、通信起動 (invoking) 、通信 (traveling) 、同期 (synch) 、後処理 (post-processing) の 5 つの部分からなる。

なお、これらのフェイズのうちで、通信フェイズだけが計算とオーバーラップできる。というのは他のフェイズは通信プロセッサではなく計算プロセッサによって実行されるからである。しか

[5] 1 つの通信単位は 1 つ以上のデータからなる。

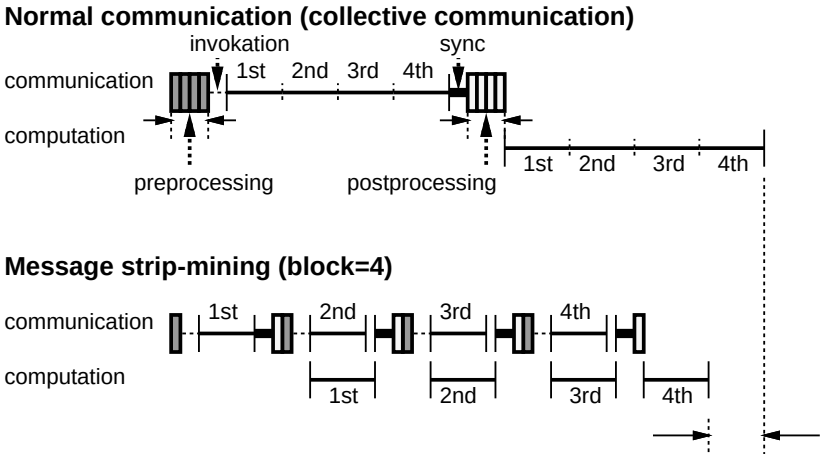


図 57: 従来の (collective) communication と メッセージ掘り下げ法のタイムチャート

し、通信フェイズは他のフェイズに比べてかなり大きいと考えられる。

前章では、計算とオーバーラップさせるために、通信を最小の単位に分割した。配列の再分散の場合、多次元配列の 1 次元のデータを最小の通信単位としていた。

本章では、その最小単位をいくつか集団化 (aggregation) する。したがって、掘り下げ (strip-mining) のために、通信全体を複数の通信単位からなる B 個のブロックに分けるとする。すなわち、それぞれ通信ブロックには N/B個の個別の通信を含んでいるので、i 番目の通信ブロックには$(i-1) \times \frac{N}{B} + j$ 番の個別通信が含まれている $(0 \leq j \leq \frac{N}{B} - 1)$。同様に、ループも B個のブロックに分ける。それぞれの計算ブロックには対応する通信ブロックで受信したデータを参照するような計算が含まれている。なお、B が 1 の場合は、それは従来方式の通信である。

i 番目の計算ブロックを開始する前に、i 番目の通信ブロックは完了していなければならない。しかし、$(i-1)$ 番目の計算ブロックは i 番目の通信ブロックと並列に実行できる。つまり、i 番目の通信ブロックを $(i-1)$ 番目の計算ブロックとオーバーラップさせることにより、通信オーバーヘッドはほとんど隠蔽できる。

5.2 基本構成に対する考察

5.2.1 通信を隠蔽する計算ループの選択

通信を計算とオーバーラップする方法として 2 つ考えられる。1 つは受信データを使用している計算部分と通信をオーバーラップする方法で、もう 1 つは送信データを定義する計算部分と通信をオーバーラップする方法である。それぞれの長所短所を次に示す。

- 通信をデータの使用部分とオーバーラップする場合、他の通信と集団化 (aggregation) できる可能性がある。それによりメッセージの数を減らすことができる。しかし、十分な計算が無い場合、通信オーバーヘッドは残る。
- 一方、定義側と通信をオーバーラップする場合は、使用側に十分な計算は必要ない。しかし、受信データをバッファに保持している時間 (lifetime) は長くなり、より多くのメモリを必要とする。

これらを図 58の例を用いて説明する。

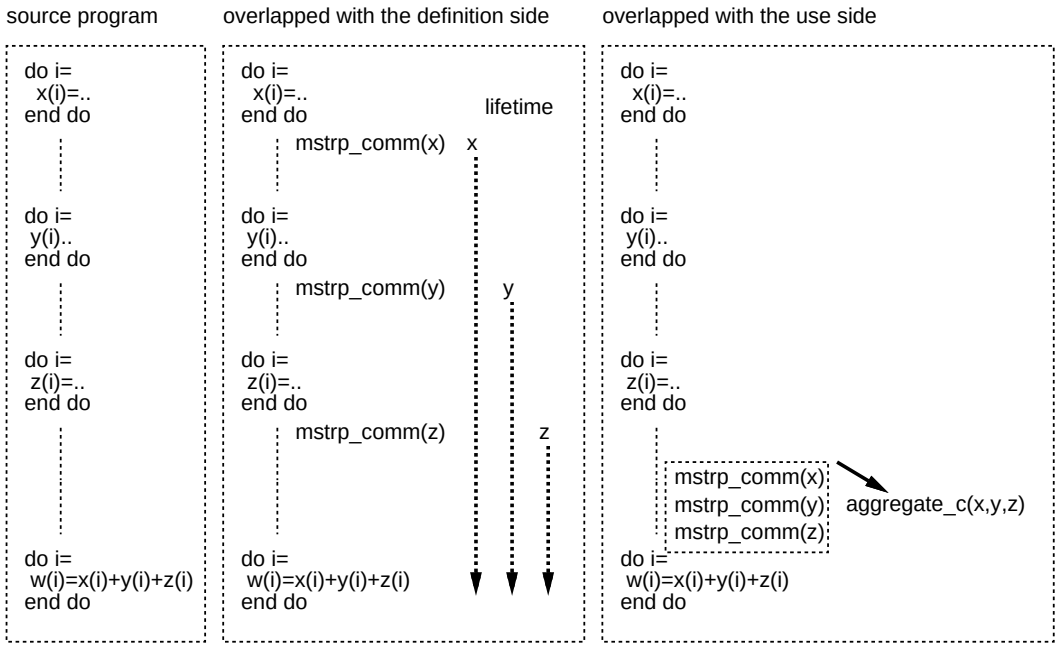


図 58: メッセージ掘り下げ (strip-mining) の実現方法

配列 x，y，z は最後のループで使用されている。これらは配列 w と割り付け (align) されていないので、通信を必要とする。定義側でオーバーラップした場合、配列 x は第 2 ，第 3 のループで使用されることなくバッファに格納したままにしなければならない。これに対して、使用側でオーバーラップした場合は、使用される直前で通信されるので、バッファのためのメモリを少なくすることができる。さらに、配列 x，y，z がお互いに割り付け (align) されていたら、これらのメッセージは 1 つのメッセージに集団化 (aggregation) でき、通信起動コストを低減できる。

これ以降の議論においては、常に使用側に十分な計算量があると仮定し、使用側でオーバーラップするとする。

5.2.2 並列化ループとデータ分散

ここまでは、再外側ループが並列化され、かつ、そのループの次元が配列の分散している次元に一致しているような場合だけを扱ってきた。しかし、他のケースも実アプリケーションにおいては存在する。つまり、外側ループが並列化されない、もしくは、分散次元と一致しない場合である。そのような場合に対処するために、ループ置換 (loop interchanging)[44] および遅延通信 (*delayed communication*) を用いる。表 3に示されるような 2 次元配列を用いたプログラムを考えよう。配列 A は当初 (BLOCK,*) で分散されていると仮定する。

プログラム 1 はこの論文でここまで扱ってきた典型的なループであるので、メッセージ掘り下げ法は容易に適用できる。

プログラム 2 では、内側ループに関して依存関係 (true dependence) があるので、外側ループが並列化される。つまり、配列 A の第 2 の次元が分散される。しかし、第 1 の次元が並列化されるであろう。4 プロセッサの場合、本プログラムから並列性を抽出するためには次のようなスキューイングをする必要がある。まず、プロセッサ 0 がインデックス (i=1,j=1:25) に関する計算を行い、配列 A の部分配列 (i=1,j=26:50) をプロセッサ 1 に送る。その後、プロセッサ 0 は、境界値 (i=1,j=25) をプロセッサ 1 に送る。次の時間スロットで、プロセッサ 0 はインデックス (i=2,j=1:25) の計算を行い、部分配列 (i=1,j=51:76) をプロセッサ 2 に送り、部分配列 (i=2,j=26:50) をプロセッサ 1 に送る。同時にプロセッサ 1 はインデックス (i=1,j=26:50) の計算を行う。

しかしながら、この計算スキームはプログラム 1 よりも多くの通信を必要とする。というのは、それぞれの通信は多対多ではなく 1 対多通信であり、また、隣接プロセッサへの境界値の通信のために追加の点対点通信が必要である。さらに、同期をとらなければいけないポイントも増え

Outer loop is parallelized		Inner loop is parallelized	
Outer loop is distributed	Inner loop is distributed	Outer loop is distributed	Inner loop is distributed
Program 1 !hpf redistribute + (*,block):: A do j=1,100 do i=1,100 A(i,j)= + A(i-1,j)+A(i,j) end do end do	**Program 2** !hpf redistribute + (*,block):: A do i=1,100 do j=1,100 A(i,j)= + A(i,j-1)+A(i,j) end do end do	**Program 3** !hpf redistribute + (*,block):: A do j=1,100 do i=1,100 A(i,j)= + A(i,j-1)+A(i,j) end do end do	**Program 4** !hpf redistribute + (*,block):: A do i=1,100 do j=1,100 A(i,j)= + A(i-1,j)+A(i,j) end do end do

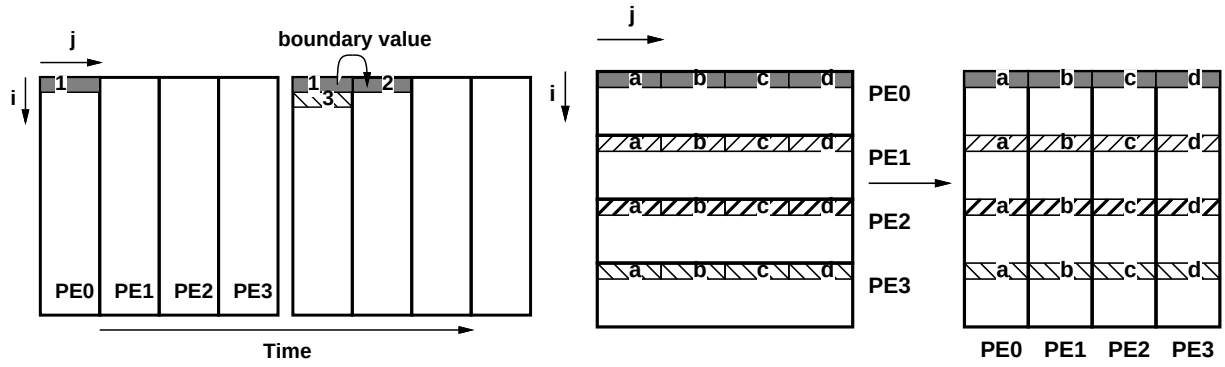
表 3: 分散次元と並列化次元

る。それゆえ、プログラム 2 に、直接、メッセージ掘り下げ法を適用すれば、性能劣化を招く。この通信を図 59の左図で示す。

通信オーバーヘッドおよび同期ポイントを軽減するために、通信と計算の順序を入れ替える方法 (delayed communication) を用いる。すなわち、計算は分散前の位置で行い、その後配列を分散するのである。なお、ここでは、再分散ディレクティブの意味合いを若干拡大している。

前の処理と同様に、4 プロセッサを用いた実行シーケンスについて説明する。p と q を プロセッサ番号 (p,q=0..3) とする。すべてのプロセッサ p は、第 1 の部分計算 (i=1+p*25,j=1:100) を通信なしに行う。その後、第 2 の部分計算 (i=2+p*25,j=1:100) を行いながら、それぞれのプロセッサから部分配列 (i=1+p*25,j=1+25*q:25*q) をプロセッサ q に送信する。この通信は多対多である。この通信を図 59の右図で示す。このスキームは計算のあとに通信起動を行っている点を除けばプログラム 1 に適応したメッセージ掘り下げ法に似ている。よって、プログラム 2(delayed communication) に適用したメッセージ掘り下げ法の有効性はいままでの議論から容易に引き出せる。

ループ置換を用いるとプログラム 3 および 4 は容易にプログラム 1 および 2 に変換できるので、すべてのケースでメッセージ掘り下げ法再分散が効果的に適用できることがわかる。



(a) 従来方式　　(b) 遅延メッセージ掘り下げ法

図 59: 並列化される次元 (i) が分散されている次元 (j) と一致しない場合の通信生成

5.3 最適ブロックサイズと理想スピードアップ値

この節では、通信単位の最適粒度の観点より通信コストをもっとも小さくするブロックサイズを求め、そのサイズのメッセージ掘り下げ法によって得られる通信時間のスピードアップについて考察する。

$nproc$ をメッセージを交換しあうプロセッサの数とする。この値は通常 (全プロセッサ数-1) である。また、α をメッセージ当たりの通信起動と同期処理の時間の合計とし、β を通信単位当たりの通信レイテンシとする。さらに、δ を通信単位当たりの前処理及び後処理の時間とし、γ を通信単位当たりの計算時間とする。従来の通信と計算のコストは次のように見積もれる。

$$T_{comm} = nproc \times \alpha + N \times (\beta + \delta) \quad (20)$$

$$T_{comp} = N \times \gamma \quad (21)$$

$$T_{total} = nproc \times \alpha + N \times (\beta + \delta + \gamma) \quad (22)$$

メッセージ掘り下げ法を適用した場合、トータルコストは、1 番目の通信ブロックの時間、$i(2 \leq i \leq N)$ 番目の通信起動、同期の時間、$i(1 \leq i \leq N)$ 番目の前処理、後処理、及び計算の時間で

ある。

$$\begin{aligned} T_{strp} &= B \times nproc \times \alpha + \frac{N \times \beta}{B} \\ &\quad + N \times (\delta + \gamma) \end{aligned} \tag{23}$$

式 (22) 及び (23) から、もし、次の条件を満たす整数 $B(>1)$ が存在すれば、T_{strp} は、 T_{total} よりも小さい。

$$B < \frac{N \times \beta}{nproc \times \alpha} \tag{24}$$

つまり、メッセージ集団化法を用いた掘り下げ法は、十分大きい N に対して有効である。

次に、掘り下げ法の通信コストを最小にする集団化の最適なブロック数 B_{opt}は、T_{strp} を Bに関して微分することによって、次のように表される。

$$\begin{aligned} \frac{dT_{strp}}{dB} &= nproc \times \alpha - \frac{N \times \beta}{B^2} = 0 \\ \Rightarrow B_{real} &= (\frac{N \times \beta}{nproc \times \alpha})^{\frac{1}{2}} \end{aligned} \tag{25}$$

B_{up} を B_{real} よりも大きいか等しい最小の整数とし、B_{low} を B_{real}よりも小さいか等しい最大の整数とする。

$$B_{opt} = \begin{cases} B_{low} & \text{if } (T_{strp}|_{B=B_{up}}) > (T_{strp}|_{B=B_{low}}) \\ B_{up} & \text{otherwise} \end{cases} \tag{26}$$

さらに、B_{opt} が B_{real}に等しく、δ が無視でき、B_{opt}個のブロックに通信を分けた場合、通信理想スピードアップは次のように容易に計算できる。

$$T_{comm\_strp} \equiv T_{strp} - T_{comp} \tag{27}$$

$$\begin{aligned} \Rightarrow ideal\_speedup &= \frac{T_{comm}}{T_{comm\_strp}}|_{B=B_{opt}} \\ &= \frac{1 + B_{opt}^2}{2B_{opt}} \end{aligned} \tag{28}$$

ここで、T_{comm_strp} は計算に隠れなかった通信時間である。この評価によると、B_{opt} が 2 の場合、スピードアップは 125% であり、B_{opt}が 4 の場合は、スピードアップは 212.5% であり、B_{opt}が 8 の場合は、スピードアップは 406.3% である。

5.4 検査官 / 執行官アルゴリズムを用いた間接アクセスへの適用

ここでは、メッセージ掘り下げ法が間接アクセスに対してどのように適用されるかを示し、メッセージ集団化法を用いることによりいかに効果的になるかを示す。

```
!hpf$ distribute (block):x,l,v
   do itr=1,MAXITR
    do i=1,M
     v(i)=...
    end do
    do i=1,N
     x(i)=x(i)+v(l(i))
    end do
   end do
```

図 60: 間接アクセスを用いたプログラム例

間接アクセスを含む計算の例として、簡単のために、図 60に示すプログラムを使う。

前述の通り、分散メモリシステムにおいては、分散された配列への間接アクセスは、しばしば、検査官 / 執行官 *(inspector/executor)* 法で実現されている。配列の間接アクセスは、アクセスすべき配列要素が必ずしも規則正しい部分配列となっていないので、要素毎に通信を起動する必要があり、そのコストが大きく不効率なものとなっている。したがって、実行時に不規則な通信起動を規則的な通信に変換し、通信起動を削減することによってコスト削減をおこなうのが本手法である。

この手法は 2 つの部分からなる。前半 (inspector) は、間接アクセスが通信を生成するかどうかを判定し、さらに通信すべき相手先、受け取り先のプロセッサを決定する。また、バッファエリアを獲得し、グローバルインデックスをローカルインデックスに変換し、executor の為の通信スケジュールを生成する。後半 (executor) は、そのスケジュールにしたがって、他のプロセッ

```
call localize(l,m,M/nproc,sched)
   do itr=1,MAXITR
    do i=1,M/nproc
     v(i)=...
    end do
    call dgather(sched)
    do i=1,N/nproc
     x(i)=x(i)+v(m(i))
    end do
   end do
```

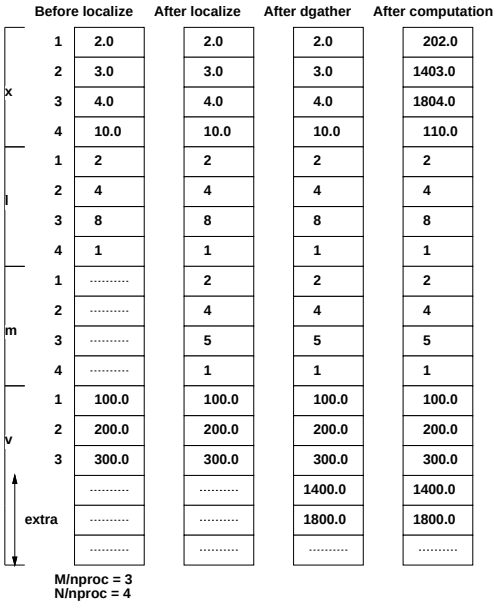


図 61: SPMD コードと 検査官 / 執行官 (inspector/executor) を用いた間接アクセスに対するメモリ構成

サとメッセージを交換するものである。アルゴリズムに関するより詳細な説明を付録に記述しておく。

検査官 / 執行官法を用いることにより、図 60のプログラムは、図 61に示すように、*PARTI*[37] ライブラリを用いた SPMD プログラムにコンパイルされる。PARTI は、Maryland 大学で開発された不規則 (irregular) 通信のためのランタイムライブラリである。なお、`nproc` はプロセッサ数を表す。

プロシージャ `localize` は、 検査官であり、グローバルインデックスを保持している配列 `l` をローカルインデックス `m` に変換し、他のプロセッサから供給されたデータを保持するために付加メモリ領域を確保する。さらに、このプロシージャは通信スケジュールを生成する。プロシージャ `dgather` は 執行官ある。

これらのプロシージャの動作を詳細に説明するために、プロセッサ 0 のメモリ構成を図 61の右図に示す。ここで、説明を簡単にするために、`M` は 12 、 `N` は 16 、 `nproc` は 4 とする。

最初の図にあるように、プロセッサ 0 はグローバルインデックス 2，4，8，1 にアクセスすると仮定する。グローバルインデックス 2，1 に対応する配列要素は、プロセッサ 0 が保持するもの

である。というのは、プロセッサ 0 が持つものは、1，2，3 だからである。しかし、グローバルインデックス 4 に対応する配列要素はプロセッサ 1 上に配置されており、グローバルインデックス 8 に対応する配列要素はプロセッサ 2 上に配置されている。よって、配列 v の隣に新たに 2 ワードの付加メモリ領域を確保しなければいけない。なお、付加メモリ領域のサイズはそのプロセッサはアクセスするグローバルインデックスの内容に依存するが、せいぜい配列 l のサイズである。

2 番目の図にあるように、2 と 1 に対するローカルインデックスはグローバルインデックスと同じであるが、4 と 8 に対するローカルインデックスは付加メモリ領域、すなわち 4 と 5 が割り当てられる。プロシージャdgather はプロセッサ 1,2 からデータを収集し、それらを付加メモリ領域に書き込む。なお、ここで、localize のコストは無視できることに注意しよう。というのは、それは外側ループのさらに外側に存在するからである。

```
call localize (l,m,M/nproc/2,sched1)
call mlocalize(l,m,M/nproc/2,sched2)
do itr=1,MAXITR
 do i=1,M/nproc
  v(i)=...
 end do
 call dgather(sched1)
 NN = N/nproc
 do ii=1,2
  if(ii.eq.1) call dgather_invoke(sched2)
  do i=(ii-1)*NN/2+1,ii*NN/2
   x(i)=x(i)+v(m(i))
  end do
  if(ii.eq.1) call dgather_sync(sched2)
 end do
end do
```

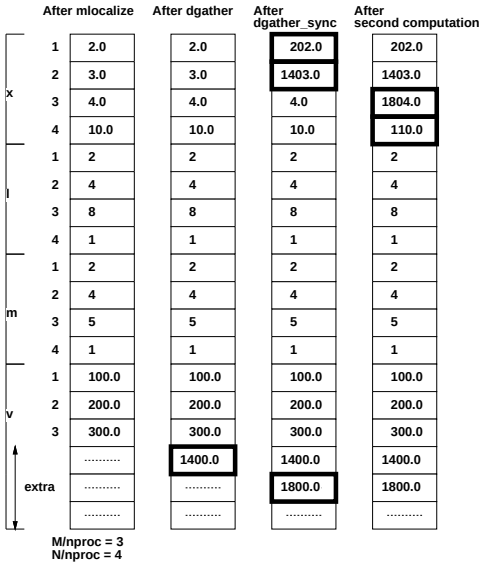


図 62: SPMD コードとメッセージ掘り下げ法を用いた間接アクセスのメモリ構成

図 62 はブロック数が 2 のメッセージ掘り下げ法による SPMD コードである。このコードは第 1 及び第 2 の通信ブロックに対するスケジュールを生成するために、それぞれプロシージャ localize と mlocalize を使用している。プロシージャmlocalize は本質的に localize と同じであるが、前者は既に送ったデータを再度送らないようにして無駄なメッセージを減らすような工夫がされている。一般に、mlocalize は最初以外の通信ブロックに対して随時呼び出され

るルーチンである。

さらに注意すべき点は、dgather_invoke が非ブロッキング通信であり、本質的に dgather と等価である。dgather_sync は dgather_invoke によって起動された通信の完了を確認するためのプロシージャである。

計算 (2 番目の内側ループ) と通信 (dgather) が 2 つの部分に分割されたので、それぞれの通信に対してスケジュールを生成しないといけないので、localize も 2 つに分割される必要がある。

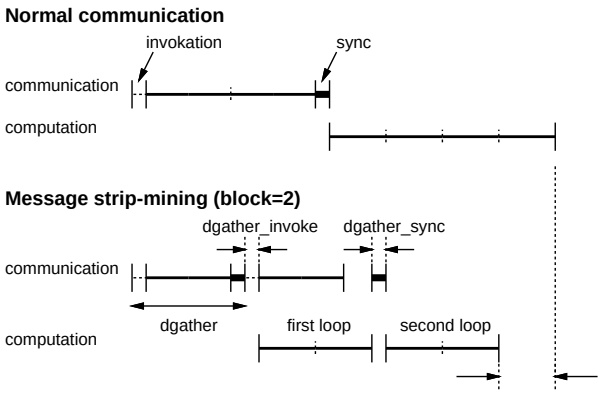


図 63: ブロック数 2 のメッセージ掘り下げ法を用いた執行官の実行タイミング

メッセージ掘り下げ法を用いた SPMD コードの実行タイミングを図 63に示してある。第 1 のループ (ii = 1) に対するリモートデータはそのループを開始する前に、dgather によって通信されている。その後、第 2 のループ (ii = 2) に対するリモートデータをフェッチするために dgather_invoke が非ブロッキング通信を起動する。その通信と同時に、第 1 のループの計算が実行される。そして、dgather_sync によって、その非ブロッキング通信の完了が確認されたら、第 2 のループを開始する。このように、ほぼ半分の通信が計算時間に隠蔽される。

5.5 実験

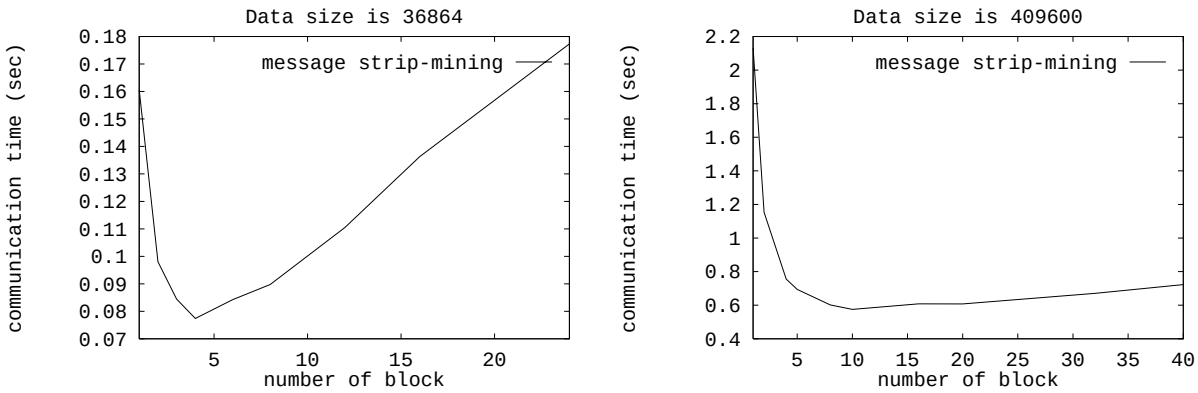
本方式の効果を例証するために、iPSC/860 上に実装した。特に、ブロック数が通信時間にどう影響するかについて調べる。前述した評価からわかるように、ブロック数の最適値が存在し、最適値より大きいもしくは小さいブロック数の場合は通信時間が増加すると予想される。

ここでは配列の再分散及び間接アクセスに対する実験を取り上げる。アプリケーションは特に限

定していないが、通信コストを隠蔽するのに十分な演算量がある計算とし、ノンブロッキング通信の通信本体は計算にオーバーラップできていると仮定する。

5.5.1 配列の再分散

16 ノードの iPSC/860 上で 2 次元配列の再分散をブロッキング通信と非ブロッキング通信を用いて実装した。配列は (Block,*) から (*,Block) へ再分散する。タイミングは dclock 組み込み関数で計測する。



(a) データサイズ=36864　　(b) データサイズ=409600

図 64: 16 ノード iPSC/860 上での再分散の通信時間 (sec)

実験の結果を図 64 に示す。ここで、data size は分散配列のローカルなサイズを示す。つまり、配列のトータルサイズは data size の 16 倍である。なお、ブロック数が 1 のメッセージ掘り下げ法とは従来方法の再分散方式のことである。

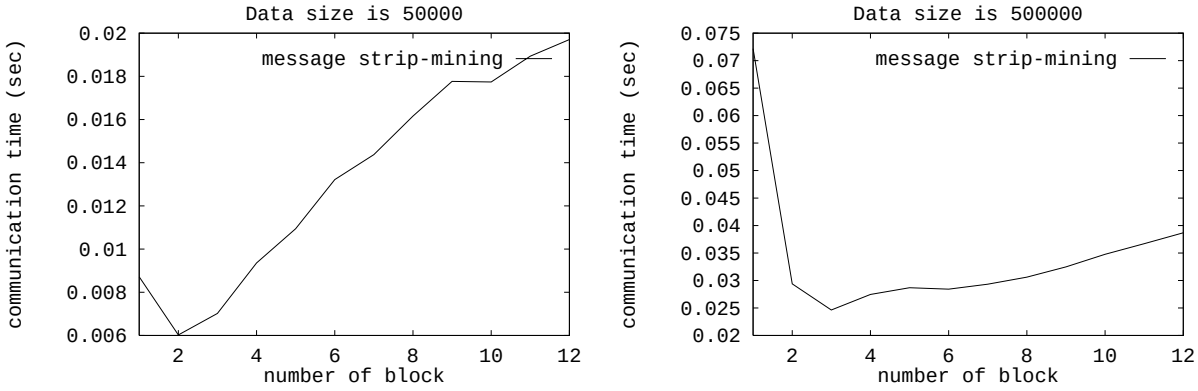
予想されるように、それぞれの場合に対してブロック数の最適値が存在する。データサイズ 409600 の場合はブロック数 8 、データサイズ 36864 の場合はブロック数 4 が最適である。スピードアップはそれぞれ 354%、 219.1% である。理想スピードアップは、ブロック数が 4 の時は 212.5% と予想されたが、計測されたスピードアップは理想スピードアップよりも高い。その理由は再分散の従来方法がより多くのネットワークコンテンションを産み出していることに加えて、キャッ

シュメモリのヒット率が向上しているからであると考えられる。というのは、メッセージ掘り下げ法で使用されているメッセージの長さが従来方法よりも短いからである。

5.5.2 間接アクセスに対する執行官アルゴリズム

本方式の間接アクセスに対する効果を示すために、2 つの実験を行なう。まず、ラフな評価を行なうために、複雑なバッファ処理を省いた簡易化した執行官プログラムの性能を測定する。さらに、実用の検査官 / 執行官プログラムの評価のために、PARTI ライブラリに非ブロッキング通信ルーチンを追加し、メッセージ掘り下げ法を用いた執行官アルゴリズムの性能を測定した。

第 1 の実験　8 ノードの iPSC/860 上で複雑なバッファ処理を省いた簡易版執行官プログラムを NX-2 ライブラリを用いて実現した。通信するデータサイズとしては、5×10^4 と 5×10^5 を用いた。なお、それぞれのメッセージの送信先は実行時にランダムに与えた。よって、全てのデータが通信されたわけではない。



(a) データサイズ=50000　　(b) データサイズ=500000

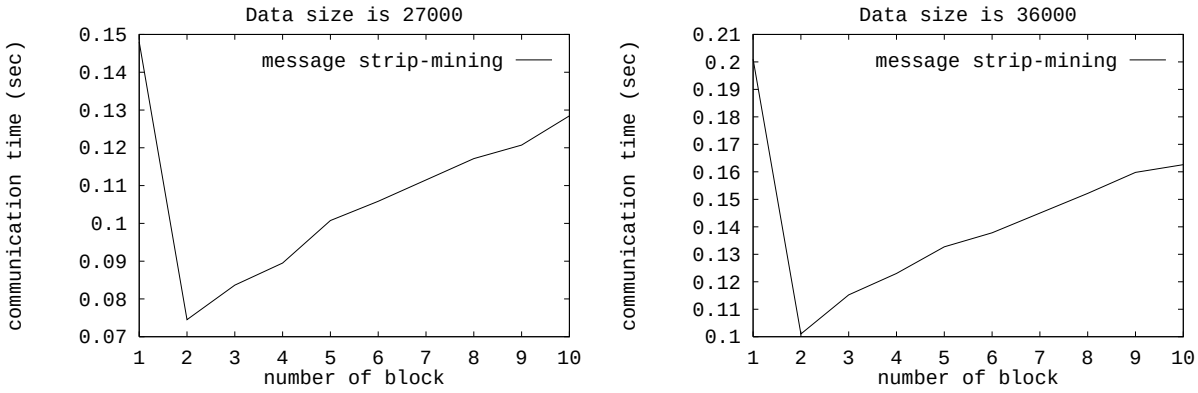
図 65: 8 ノード iPSC/860 上での執行官の通信時間 (sec)

実験結果を図 65に示す。なお、前の実験と同様に、ブロック数が 1 のメッセージ掘り下げ法は従来の執行官と同じである。

図 65 が示すように、最小の通信時間は、データサイズが 5×10^4、5×10^5の時、それぞれ、ブロック数が 2 、4 の時に得られる。また、スピードアップは、144.9% 及び 262.9%で、これらは前の実験と同様に、式 (28) で得られる理想スピードアップよりも高い。理由も同様で、メッセージの長さが短くなったことによる効果と考えられる。

さらに、通信時間のプロット図は、再分散の場合よりもギザギザになっている。これは、執行官の通信が不規則性を含んでおり、その通信パターンの変化が規則通信 (regular communication) よりも激しいからである。

第 2 の実験　実用の検査官 / 執行官プログラムの評価のために、PARTI ライブラリ Revision 2 に非ブロッキング通信ルーチンを追加し、8 ノードの iPSC/860 上でメッセージ掘り下げ法用いた執行官プログラムを実装した。通信するデータサイズを 36000 と 27000 とし、ブロック数を変化させて通信時間を測定した。また、今回の実験では `localize` と `mlocalize` のコストは無視した。というのは、これらは外側ループのより外に位置しているからである。



(a) データサイズ=27000　　(b) データサイズ=36000

図 66: 8 ノード iPSC/860 上での PARTI を用いた執行官の通信時間 (sec)

実験結果を図 66に示す。

第 1 の実験のように、メッセージ掘り下げ法を用いた執行官の通信時間は従来方法よりも確かに短くなってはいるが、メッセージ掘り下げ法を用いた執行官の性能はブロック数が 2 を越えた

後はブロック数とともに低下している。その理由は、PARTI ライブラリが通信スケジュールや分散に関する情報保持するために、通信するデータ数に比例したメモリ容量を必要とするので、十分大きなデータに対して執行官プログラムを実装できなかったからである。

5.6 コンパイラへの実装

ここまで述べてきたように、メッセージ掘り下げ法による通信コストの削減の効果が確かめられた。ここでは、実際のコンパイラにおいてメッセージ掘り下げ法を組み込むための概要をプログラム例を用いて述べた後、一般的な適用方法を表す。

下記の HPF プログラムを 4 プロセッサの並列コンピュータ用 SPMD FORTRAN プログラムにコンパイルする場合を考える。

```
!hpf$ distribute a(block)
!hpf$ distribute b(cyclic)
        :
      do i=1,6400
       a(i)=func(b(i))
      end do
```

このプログラムにおいて配列 a,b は align されていないので do ループの計算においては通信が必要となる。

FORTRAN D コンパイラ [40] 等で用いられている通信メッセージ生成手法により、このプログラムは集団通信 (collective communication) を用いた下記に示す SPMD プログラムにコンパイル可能である。

```
      send (b,1:1600)
      receive (tmp,1:1600)
      do i=1,1600
       a(i)=func(tmp(i))
      end do
```

FORTRAN D コンパイラでは、ループに関する依存性にしたがって生成するメッセージを 3 種類に分類している。ループに関して依存性のないメッセージを independent と呼び、ループの前

にメッセージを配置する。ループに依存するメッセージの内、その依存するループが全プロセッサで実行される場合、そのメッセージを carried-all と呼び、依存のあるループのループヘッダ内にメッセージを配置する。また、依存するループがプロセッサ間に分散されている場合は、そのメッセージを carried-part と呼び、ループの前後に send と receive のメッセージを配置する。上記のプログラムの通信はループ `i` に関してデータ依存がないので、independent メッセージとなり、ループの前に通信が配置されている。なお、`send`，`receive` は集団通信を示すマクロ命令である。

この independent メッセージにメッセージ掘り下げ法を適用するために、最適ブロックサイズを式 (25) を用いて決定する。

$nproc$ は 4 、N は 1600 である。α, β は ハードウエアに依存するが、ここでは$\alpha = 25 \times \beta$ と仮定する。

$$\begin{aligned} B &= irint((\frac{N \times \beta}{\alpha \times nproc})^{\frac{1}{2}}) \\ &= irint((\frac{1600}{25 \times 4})^{\frac{1}{2}}) \\ &= 4 \end{aligned} \tag{29}$$

よって、通信および計算を 4 つのブロックに分割 (strip mining) する。

```
do iB=1,4
 send (b,1+(iB-1)*400:iB*400)
 receive (tmp,1+(iB-1)*400:iB*400)
end do
do iB=1,4
 do i=1+(iB-1)*400,iB*400
  a(i)=func(tmp(i))
 end do
end do
```

4 個の通信の内、最初の通信はブロッキング通信とする。2 番目以降の通信をノンブロッキング通信とし、計算とスキューイングする。

```
send (b,1:400)
```

```
receive (tmp,1:400)
do iB=1,4
 if(iB.not.4) send (b,1+iB*400:(iB+1)*400)
 do i=1+(iB-1)*400,iB*400
  a(i)=func(tmp(i))
 end do
 if(iB.not.4) receive (tmp,1+iB*400:(iB+1)*400)
end do
```

なお、実プロセッサ等パラメータの一部がコンパイル時に不明の場合も、最適ブロックサイズを実行時に計算することにより、同様にメッセージ掘り下げ法を用いてコンパイル可能である。

以上のように、メッセージ掘り下げ法により通信時間の削減を図るコンパイリングができる。この効果は式 (28) により、

$$\frac{1+B_{opt}^2}{2\times B_{opt}} = \frac{1+4^2}{2\times 4} = 2.13$$

となり、約 2.13 倍の通信のスピードアップが得られると予想される。

この例以外のプログラムの場合も同様に適用可能である。すなわち、集団通信のような規則的な通信が発生するプログラムに対して、上記の方法でコンパイラで処理ができる。生成されるメッセージタイプの違いによるメッセージ掘り下げ法の一般的適用手順を図 67に示す。

まず、通信が陽に集団通信である場合 (1) 、例えば配列の再分散通信のような場合、メッセージをオーバラップできる do ループの直前に移動し、その計算にメッセージ掘り下げ法を適用する。また、通信が陽に現れていないが、分散されている配列を参照するなどしてコンパイラによってメッセージが生成される場合 (2) には、そのメッセージのタイプによって処理を変える。

independent の場合 (3) は依存するループが無いので、最外側のループの外側に通信が配置されている。したがって、全計算時間から見た場合、コストが必ずしも大きくないので、まず予想される通信コストを計算し、それがある一定値以上の場合 (4) のみメッセージ掘り下げ法を行うことを考える。その理由は、メッセージ掘り下げ法を適用するために通信を内側のループ内に移動する場合があり、各種のオーバーヘッドを増加させてしまう可能性があるからである。移動した通信に対して掘り下げ法を適用するが、内側のループに対しては依存関係がないメッセージなので、通信起動が 1 回だけになるように通信起動を if 文で囲まねばならない。

```
if(explicitな集団通信がある){/* (1) */
     - receive される配列が最初に使用される do ループの直前に通信を移動する
     - メッセージ掘り下げ法を適用
}
else{
  if(参照代入関係より通信が生成される){/* (2) */
    switch(メッセージタイプ){
      case independent:/* (3) */
        if(予想される通信コスト<一定値){/* (4) */
           - 最初に使用される do ループの前に通信を移動し、そのループよ
             り外側のループが初期値の時のみ通信起動されるように if 文で囲む
           - メッセージ掘り下げ法を適用
        }
        break;
      case carried-all:/* (5) */
        - メッセージ掘り下げ法を適用
        break;
      case carried-part:/* (6) */
        loop interchange や配列の再分散を行い、
        carried-part部を解消し、最初から適用し直す。
        break;
    }
  }
}
```

図 67: メッセージ掘り下げ法の一般的適用手順

carried-all の場合 (5) 、メッセージ掘り下げ法を適用すべき do ループは、通信が配置された位置の直後の do ループであるので、その位置のまま適用する。最後に、carried-part の場合 (6) 、そのままではメッセージ掘り下げ法は適用できない。したがって、loop interchange や配列再分散を行って、分散すべき do ループを変更したり実行順序を変更して carried-part 部分を解消しなければいけない。しかし、このような変更は他の do ループにも影響を及ぼすので、全体の通信コストを鑑みて合理的な処理判断が必要である。

なお、メッセージ掘り下げ法自体の適用方法は前述の通り、1) 最適ブロックサイズを決定し、2) オーバーラップ先の計算部分及び通信を分割し、3) 最後に通信をブロッキング通信/ノンブロッキング通信に分けて、2 で分割した計算部分とオーバーラップさせることによって行なう。

5.7 まとめ

通信時間を計算時間に隠蔽すること、及び、短いメッセージを用いることによりキャッシュメモリのヒット率が向上することにより、メッセージ掘り下げ法はプロセッサ間通信のコストを大幅に減少できる。

本章では、メッセージ掘り下げ法が配列の再分散 (regular communication) だけでなく、間接アクセス時の執行官アルゴリズム (irregular communication) をはじめ規則性の高い通信を持つ任意のアプリケーションにも適用できることを示した。

また、メッセージ集団化することにより、幅広い問題に対してメッセージ掘り下げ法が適用でき、そのブロック数の最適値も解析的に決定できることが分かった。

さらに、本方式の有効性を実験的に確かめた。例えば、メッセージ集団化のブロック数が 8 のメッセージ掘り下げ法をサイズ 2560×2560 の配列の再分散に適用したとき、通信のスピードアップは 354% となった。また、ブロック数を 4 とした時に、サイズ 5×10^5 のデータを収集するような執行官アルゴリズムは、最適なメッセージ掘り下げ法により、260%を越える通信のスピードアップを得られた。

最後に、メッセージ掘り下げ法のコンパイラへの実装方法についてその概要を述べた。従来の FORTRAN D コンパイラなどの HPF 系コンパイラへの実装は容易であると考えられる。

6 結論

次世代のスーパーコンピュタは分散メモリ型マルチコンピュータであるといわれており、既にその範疇に含まれる製品も現れている。しかし、現状では必ずしも多数派とはなっておらず、そのための標準プログラミング環境の確立が急務であると考える。つまり、分散メモリ型計算機の普及のためには

標準プログラミング言語の確立: プログラムに移植性及び継承性を与える。

コンパイラ通信最適化: コンパイラ実装の最大のポイントは、性能劣化の原因となるプロセッサ間通信時間の最小化をいかに行うかである。通信レイテンシを削減することにより、計算機の最大性能 (ピーク性能) に近い実効性能を得ることができる。

が重要である。

本論文では、まず、分散メモリ計算機に対するプログラミング言語として並列計算機 ADENA におけるプログラミング環境設計を述べ、HPF とは異なるアプローチによる言語 ADETRAN のコンパイラ設計方針を述べた。さらに、通信レイテンシ削減に関して ADENA で用いているハードウエアを利用した解決法を述べた後、プロセッサ配置による配列再分散の高速化、および、通信と計算を分割しオーバーラップする手法 (メッセージ掘り下げ法) を述べた。さらに、ADENA および iPSC/860 を用いた実験も行った。

本論文の第 3 章においては、並列計算機 ADENA において開発したプログラミング言語 ADETRAN の設計思想及びコンパイラ最適化について述べた。主な結果は次のようなものであった。

- ADENA の専用プログラミング言語として ADETRAN に対するコンパイラを開発した。ADETRAN は、1)FORTRAN 言語に近い構文にする、2)ADENA に対して効率の良いコードを生成するために配列再分散通信を中心とした通信文を用意する、3) 数値計算で必要となる新規構文を導入する、などの特徴を持つ言語である。
- ADENA には通信レイテンシ削減のためのハードウエアを備えている。この機能は、計算プロセッサが送信データをメモリに書き込むと同時に通信プロセッサに送信可能シグナルを送るもので、計算プロセッサが次のデータの処理をしている間に前のデータを送信することにより、通信レイテンシが隠蔽される。このハードウエアが利用できる通信パターンは限られたものであるが、コンパイラによってプログラムを若干変換することにより、本

手法の適用できる範囲は広くなる。本手法に加え、メモリアクセスの最適化手法を組み合わせることにより、ADENA での実験では 30%の性能向上が見られた。

本論文の第 4 章においては、ADI 等のアプリケーションで必須となる配列再分散通信に注目し、特殊なハードウエアを仮定せずにソフトウエアによってその通信レイテンシを削減する方法について検討した。主な結果を次に示す。

- 仮想プロセッサをプロセッサグループの局所化を考慮して効率良く物理プロセッサに配置する手法 (螺旋配置) を提案した。通信の混雑度を削減することにより、通信時間の削減が図られた。例えば、1024×1024 の 2 次元配列の再分散の場合、iPSC/860 において 36%の効率化が得られた。
- 通信を計算にオーバーラップさせることによって通信レイテンシを削減する方法として、メッセージ掘り下げ法を提案した。本方式は、計算に必要な最小の部分に配列を分割し、その単位で計算と通信を分割し、分割された計算と通信をパイプライン的にオーバーラップするものである。本方式が適用可能なアプリケーションの条件は、通信起動時間と通信レイテンシに依存したパラメータで決定される。2048×2048 の 2 次元配列を用いた ADI プログラムの場合、iPSC/860 で 1.7 倍のスピードアップが得られた。

本論文の第 5 章においては、配列再分散通信だけでなく規則性の大きい通信を持つ任意のアプリケーションにメッセージ掘り下げ法を適用できることを示した。主な結果を次に示す。

- メッセージ掘り下げ法にメッセージ集団化法を用いることにより、メッセージ掘り下げ法の適用可能条件が緩和され、より広い配列の通信に適用できた。また、通信時間を最小化する最適なブロックサイズを解析的に決定する手法を示し、その場合の理想スピードアップ値も示した。
- 任意のアプリケーション例として、配列の間接アクセスにメッセージ集団化を利用した掘り下げ法を適用した場合、実験では、通信時間のスピードアップが 260%を越えることもあることがわかった。
- メッセージ掘り下げ法のコンパイラへの実装方法についてその概要を述べ、従来の FORTRAN D コンパイラなどの HPF 系コンパイラへの実装は容易であること考えられる。

以上のように、本論文で述べた通信レイテンシの削減手法は、広い範囲の一般のプログラムに適用可能であり、実効性能の改善が期待できる。今後はこのような手法を実用コンパイラ上に実現

し、実アプリケーションによる評価が重要となる。

謝辞

本論文をまとめるにあたり、終始御懇切な御指導と御鞭撻を賜りました京都大学工学部教授 富田眞治博士に深甚なる謝意を表します。

また本論文につき数々な有益な御教示と御助言を頂いた京都大学工学部教授 津田孝夫博士、京都大学工学部教授 布川昊博士に心より感謝いたします。

並列計算機 ADENA の共同開発などを通じ、日頃からの多大なる御指導と本論文作成に対する有益な御助言を頂いた京都大学工学部助教授 野木達夫博士に深く感謝の意を表します。

また本論文をまとめるにあたり、御助言をいただきました京都大学工学部助教授 中島浩博士に感謝いたします。

米国留学中の研究指導及び本論文後半の研究において有益なご助言を頂きました米国オレゴン科学技術大学大学院 (OGI) 教授 マイケルウルフ博士、ポートランド州立大学助教授 ジンケリー博士に心より感謝の意を表します。

また本研究を行うに当たり有益な御助言を頂きました山梨大学工学部助教授 本多弘樹博士、米国オレゴン科学技術大学大学院 マイケルガーレック氏、プリヤダルシャンコルテ氏をはじめとする OGI Sparse グループの方々及び英国ブリストル大学大井仁氏、に深く謝意を表します。

本研究の機会を与えられ、直接、間接に御指導いただいた松下電器産業株式会社常務取締役堀内司朗博士、同社研究本部長 取締役 新田恒治博士、同社半導体研究センター所長 竹本豊樹博士に深く感謝の意を表します。

松下電器株式会社中央研究所 松島宏司主担当、廉田浩博士、半導体研究センター 間野洋治郎主担当には本研究の遂行に当たって御指導と御鞭撻を頂きました。ここに深く感謝いたします。

また本研究を行うに当たり御尽力頂いた松下電器産業株式会社中央研究所 金子克幸リーダー、岡林一郎主任研究員、岡本理主任研究員、森康浩研究員、西川順二研究員、佐々木真司研究員、尾林善正氏、右田学氏、古屋晋二氏、同社半導体研究センター 中島雅逸主任技師、中倉康浩主任技師、貴志哲司技師、三村忠昭技師、材木幸治技師、松下電子工業株式会社 谷川裕二主任技師、株式会社松下ソフトリサーチ 冨田貞文技師、黒田茂技師、松下電子部品株式会社 伊藤守主席技師、さらに有益な御助言を頂いた松下電器中央研究所 栄藤稔博士、前田利之研究員、同社半導体研究センター 小田中伸二博士、同社映像音響研究所 清原督三主任技師、木村浩三主任技師、同社情報通信研究所 宮崎雅也主任技師、に厚く御礼申し上げます。

研究業績一覧

発表論文

1) H. Kadota, K. Kaneko, I. Okabayashi, T. Okamoto, T. Mimura, Y. Nakakura, A. Wakatani, M. Nakajima, J. Nishikawa, K. Zaiki and T. Nogi, "Parallel Computer ADENART - its Architecture and Application -", International Conference on Supercomputing, pp. 1-8, 1991

2) 三村忠昭, 岡林一郎, 岡本理, 若谷彰良, 右田学, 廉田浩, "超高速並列計算機 ADENA", National Technical Report, Vol.36, No.5, pp. 20-27, Oct., 1990

3) 材木幸治, 若谷彰良, 黒田茂, 岡本理, "APARC: Parallelizing Compiler for Parallel Computer ADENART", Joint Symposium of on Parallel Processing 91, pp. 293-300, May, 1991

4) Akiyoshi Wakatani and Michael Wolfe, "A New Approach to Array Redistribution: Strip Mining Redistribution", International Conference on Parallel Architectures and Languages Europe, pp.323-335, 1994

5) Akiyoshi Wakatani and Michael Wolfe, "Effectiveness of Message Strip-Mining for Regular and Irregular Communication", 7th International Conference on Parallel and Distributed Computing Systems, pp. 426-433, 1994

6) Hitoshi Oi and Akiyoshi Wakatani, "Virtual Tree: A Remedy for Hot Spot Contention in the Data Diffusion Machine", International Symposium on Parallel Architectures, Algorithms and Networks, pp. 221-228, 1994

7) Akiyoshi Wakatani and Michael Wolfe, "Optimization of the Redistribution of Arrays for Distributed Memory Multicomputers", Parallel Computing, to appear, 1995

8) Koji Zaiki, Akiyoshi Wakatani, Tadashi Okamoto, Katsuyuki Kaneko and Tatsuo Nogi, "Parallel Programming Language ADETRAN", Parallel Language and Compiler Research in Japan, Kluwer, 1995

9) 若谷彰良, "分散メモリコンピュータにおける通信レイテンシ最小化手法", 情報処理学会論文誌, to appear, 1996

国内学会発表・研究会

1) 若谷彰良, 野木達夫, “並列計算機用言語 ADETRAN 処理系の開発”, 第 36 回情報処理学会全国大会, pp. 153-154, 1988

2) 若谷彰良, 材木幸治, 岡本理, “並列計算機 ADENA に対する最適化コンパイルの一手法”, 情報処理学会アーキテクチャ研究会, 89-ARC-79, pp. 57-64, 1989

3) 若谷彰良, 森康浩, 佐々木真司, 岡本理, “並列処理言語 ADETRAN の実装”, 情報処理学会プログラミング言語研究会, 90-PL-27, pp. 1-10, 1990

4) 狩野信吾, 若谷彰良, 岡林一郎, 金子克幸, “並列計算機 ADENA におけるプロセッサ間通信の高効率化”, 電子情報通信学会全国大会, page. 6-110, 1990

5) 佐々木真司, 若谷彰良, 森康浩, 岡本理 “並列計算機 ADENA システムのプログラミング環境”, 第 40 回情報処理学会全国大会, pp. 1243-1244, 1990

6) 冨田貞文, 若谷彰良, 廉田浩, 岡本理, “並列計算機 ADENA システムの性能評価”. 第 40 回情報処理学会全国大会, pp. 1241-1242, 1990

7) 黒田茂, 材木幸治, 若谷彰良, 岡本理, 廉田浩, “並列計算機 ADENA 用自動並列化コンパイラ”, 第 42 回情報処理全国大会、pp. 5/119-5/120, 1991

参考文献

[1] J.C. Adams, et al. *Fortran 90 Handbook*. Intertext-McGraw Hill, 1992.

[2] Vikram Adve, Alan Carle, Elana Granston, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, John Mellor-Crummey, Scott Warren, and Chau-Wen Tseng. Requirement for data-parallel programming environments. *IEEE Parallel & Distributed Technology*, 1994.

[3] V. Bakasundaram, et al. An interactive environment for data partitioning and distribution. In *5th DMCC*, 1990.

[4] T Brandes. *ADAPTOR Language Reference Manual Version 1.0*. GMD, 1993.

[5] T Brandes. *ADAPTOR The Disrtibuted Array Library Version 1.0*. GMD, 1993.

[6] T Brandes. *ADAPTOR User Guide Version 1.0*. GMD, 1993.

[7] Nicholas Carriero and David Gelernter. Linda in context. *Communication of ACM*, Vol. 32, No. 4, pp. 444–458, 4 1989.

[8] P. Christy. Software to support massively parallel computing on the MasPar MP-1. In *IEEE Compcon Spring 1990*, 1990.

[9] Ian Foster. Task parallelism and high performance languages. *IEEE Parallel & Distributed Technology*, 1994.

[10] D. Gajski, et al. Cedar:-a large scale multiprocessor. In *International Conferece on Parallel Processing*, 1983.

[11] Al Geist, et al. *PVM*. The MIT Press, 1994.

[12] G. Geist and V. Sunderam. Network based concurrent computing on the PVM system. Technical Report TM-11760, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, June 1991.

[13] R. Hanxleden, Ken Kennedy, et al. Compiler analysis for irregular problems in Fortran D. In *5th workshop on Lang. and Compiler for parallel computing*, 1992.

[14] M T Heath. Visual animation of parallel algorithms for matrix computations. In *5th DMCC*, 1990.

[15] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for distributed-memory machines. *Communication of ACM*, Vol. 35, No. 8, 8 1992.

[16] H. Kadota, A. Wakatani, et al. Parallel computer ADENART - its architecture and application -. In *International Conference on Supercomputing*, 6 1991.

[17] K. Kaneko, et al. A VLSI RISC with 20 MFLOPS peak 64bit floating point unit. *IEEE Journal of Solid-State Circuit*, Vol. 24, No. 5, 1989.

[18] Ken Kennedy. Is parallel computing dead? *Parallel Computing Research*, Vol. 2, No. 4, 1994.

[19] C. Koelbel and P. Mehrotra. Programming data parallel algorithms on distributed memory machines using kali. In *International Conference on Supercomputing*, 1991.

[20] Charles Koelbel. Compile-time generation of regular communications patterns. In *Supercomputing'91*, 1991.

[21] Charles H. Koelbel, et al. *The High Performamce FORTRAN Handbook*. The MIT Press, 1994.

[22] Charles Koelbel, Piyash Mehrotra, Joel Salts, and Harry Berryman. Parallel loops on distributed machines. In *DMCC-5*, 1990.

[23] Charles Koelbel and Piyush Mehrotra. Compiling global name-space parallel loop for distributed execution. *IEEE Tran. Parallel and Distributed Systems*, Vol. 2, No. 4, 1991.

[24] Y. Nakakura, et al. A versatile data transfer control unit for a parallel processor system. In *Digest 1989 Symposium on VLSI Circuits*, 1989.

[25] Michaell Noakes, Deborah Wallach, and William Dally. The J-Machine multicomputer: An architectural evaluation. In *Intl. Symp. on Computer Architecture*, pp. 224–235, 1993.

[26] T. Nogi. Parallel machine ADINA. In *Computing Methods in Applied Sciences and Engineering*, pp. 103–122. North-Holland, 1982.

[27] T. Nogi. Parallel computation. In *Pattern and Waves - Studies in Mathematics and its Application 18*. North-Holland, 1986.

[28] T. Nogi. Parallel programming language ADETRAN. *Memoirs of the Faculty of Engineering*, Vol. 51, No. 4, pp. 235–290, 1989. Kyoto University.

[29] T. Nogi. Parallel computation on ADENA. In *Parallel Computing 91*, 1992.

[30] Tatsuo Nogi. Promising data parallel environment-ADEPS, ADETRAN & ADENA-. In *Proceedings of the first Aizu international symposium on parallel algorithms/architecture synthesis*, 3 1995.

[31] S. Odanaka, et al. Massively parallel computation using a splitting-up operator method for a three-dimensional device simulation. In *IEICE Technical Report*, 10 1992.

[32] David A. Padua and Michael Wolfe. Advanced compiler optimizations for supercomputers. *Communications ACM*, Vol. 29, No. 12, pp. 1184–1201, December 1986.

[33] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report TR-CS-94-230, University of Tennessee, 4 1994.

[34] William Press. *Numerical Recipes in C*, pp. 665–666. Cambridge, 1988.

[35] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conf. on Program Language Design and Implementation*, 1989.

[36] John Rose, et al. C*: An extended c language for data parallel programming. Technical Report PL97-5, Thinking Machines, 1987.

[37] J. Saltz, R. Das, R Ponnusamy, and D Mavriplis. PARTI procedures for realistic loops. In *Proceedings of the 6th Distributed Memory Computing Conference*, 1991.

[38] S.Odanaka and T. Nogi. Massively parallel computation using a splitting-up operator method for three-dimensional device simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 15, No. 7, pp. 824–832, 7 1995.

[39] The Thinking Machine Corporation. *CM Fortran User's Guide*, 1990.

[40] Chau-Wen Tseng. An optimizing Fortran D compiler for MIMD distributed-memory machines. PhD dissertation, Rice University, January 1993.

[41] Akiyoshi Wakatani and Michael Wolfe. Effectiveness of message strip-mining for regular and irregular communication. In *Proceedings of 7th Int'l Conf. on Parallel and Distributed Computing Systems*, 1994.

[42] Akiyoshi Wakatani and Michael Wolfe. A new approach to array redistribution: Strip mining redistribution. In *Proceedings of Int'l Conf. on Parallel Architectures and Languages Europe*, 1994.

[43] Akiyoshi Wakatani and Michael Wolfe. Optimization of the redistribution of arrays for distributed memory multicomputers. *Parallel Computing*, 1995.

[44] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, 1989.

[45] Koji Zaiki, Akiyoshi Wakatani, Tadashi Okamoto, Katsuyuki Kaneko, and Tatsuo Nogi. Parallel programming language ADETRAN. In *Parallel language and compiler research in Japan*. Kluwer, 1995.

[46] Hans Zima, et al. Vienna Fortran -a language specification. Technical report, ICASE NASA Langley Research Center, 3 1992.

[47] 岡本理, 材木幸治, 冨田貞文, 小田中紳二, 廉田浩. 並列計算機 ADENA の性能評価 (大規模シミュレーションへの並列計算機の応用). 日本機械学会全国大会, 1990.

[48] 岡本理, 三谷真人, 崎山一幸. 並列計算機 ADENART のシミュレーション分野での応用. *National Technical Report*, Vol. 39, No. 1, 2 1993.

[49] 岡林一郎ほか. 並列計算機 ADENA におけるネットワークの構成と vlsi による実現. Technical Report ICD89-152, 電子情報通信学会集積回路研究会, 1989.

[50] 材木幸治, 若谷彰良, 黒田茂, 岡本理. APARC: Parallelizing compiler for parallel computer ADENART. In *Joint Symposium of on Parallel Processing 91*, 5 1991.

[51] 寺西和也. 主記憶 2G バイトで液浸冷却方式の CRAY-2 スーパーコンピュータ. 日経エレクトロニクス, No. 384, 1985.

[52] 若谷彰良, 材木幸治, 岡本理. 並列計算機 ADENA に対する最適化コンパイルの一手法. Technical Report 89-ARC-79, 情報処理学会アーキテクチャ研究会, 1989.

[53] 若谷彰良, 森康浩, 佐々木真司, 岡本理. 並列処理言語 ADETRAN の実装. Technical Report 90-PL-27, 情報処理学会プログラミング言語研究会, 12 1990.

[54] 若谷彰良, 野木達夫. 並列計算機用言語 ADETRAN 処理系の開発. 情報処理学会全国大会, 1988.

[55] 狩野信吾, 若谷彰良, 岡林一郎, 金子克幸. 並列計算機 ADENA におけるプロセッサ間通信の高効率化. 電子情報通信学会全国大会, 1990.

[56] 進藤達也ほか. Twisted data layout. In *Joint Symposium of on Parallel Processing 94*, 1994.

[57] 杉山和徳. ADETRAN 言語とそのコンパイラ設計. Master's thesis, 京都大学工学研究科数理工学専攻, 1987.

[58] 星野力. PAX コンピュータ. オーム社, 1985.

[59] 浅見. 米 CRAY が超並列マシンを製品化、2048 プロセッサで 307GFLOPS. 日経エレクトロニクス, No. 592, 1993.

[60] 浅見他. 355.2GFLOPS のスーパーコンピュータを富士通が開発. 日経エレクトロニクス, No. 564, 1992.

[61] 鶴見敬之. スーパーコンピュータが超並列型へ、分散共有メモリ方式 を採用. 日経エレクトロニクス, No. 603, 1994.

[62] 富田眞治. 並列計算機構成論. 昭晃堂, 1986.

[63] 富田眞治, 末吉敏則. 並列処理マシン. オーム社, 1989.

[64] 冨田貞文, 若谷彰良, 廉田浩, 岡本理. 並列計算機 ADENA システムの性能評価. 情報処理学会全国大会, 1990.

[65] 兵賀幸夫, 金子克幸, 岡林一郎, 西川順二, 岡本理. 並列計算機 ADENA による有限要素解析大域行列の作成. 電子情報通信学会春期全国大会, 1991.

[66] 平栗他. マシンサイクル 7.5ns を達成した並列パイプライン処理方式のスーパーコンピュータ FACOM VP. 日経エレクトロニクス, No. 314, 1983.

A　付録

この章では本論文で行なってきた議論の補足を行なう。まず、第 2 章で説明した通信レイテンシ最小化手法と並んで重要な通信最適化技法である、通信起動最小化手法の既存技術について述べる。その中の検査官 / 執行官法 (inspector/executor) は、配列の間接アクセスを高速に行なうアルゴリズムで、第 5 章でメッセージ掘り下げ法を適用したものである。

さらに、同じく第 2 章で説明した HPF 言語の実現例として、ADAPTOR システムと APR 社の xHPF コンパイラの特徴について述べる。サンプルプログラムの実行結果より、両システムともに比較的初期のものでもあり、通信最適化があまりなされていないことが分かった。

A.1　通信起動最小化手法

プロセッサ通信の最適化には、第 2 章で述べた通信レイテンシ最小化の他にも、通信起動コストを最小化する手法がある。この節では、メッセージベクトル化法 (message vectorization)[3] 、メッセージ合体法 (message coalescing)[40] 、メッセージ集団化法 (message aggregation)[40] 、検査官 / 執行官法 (inspector/executor)[20, 23, 22] について述べる。

メッセージベクトル化法 (message vectorization)　メッセージベクトル化法は、通信起動最適化においては基本的な手法である。ネストされたループにおいて、通信すべきデータどのループに依存しているかもしくは依存していないかを調べ、挿入すべきループレベルにおいてベクトル化する。図 68の左図のプログラムを考える。このプログラムの右辺のオペランドはすべて境界の部分で通信が必要となる。

コンパイルされた SPMD プログラムを右に示している。配列要素はループに依存しないので、k ループの外で通信できる。したがって、配列要素は `B(i+1,1:N)` のようにベクトル化されている。また、配列要素 `A(i+1,j)` は、全体が k ループで依存しているので、k ループの先頭でベクトル化されて通信される。しかし、配列要素 `A(i-1,j)` は i ループで依存しているので、その前後に個別メッセージとして通信される。以上のように、3 つのオペランドのうち 2 つに関してはベクトル化され、ループの外で通信が行えているので、通信起動のコストは軽減できる。

メッセージ合体法 (message coalescing)　メッセージ合体法は、同じ範囲の配列要素を複数回送ることのないように、通信範囲を比較し合体する手法である。図 69の左に示すプログラムを考

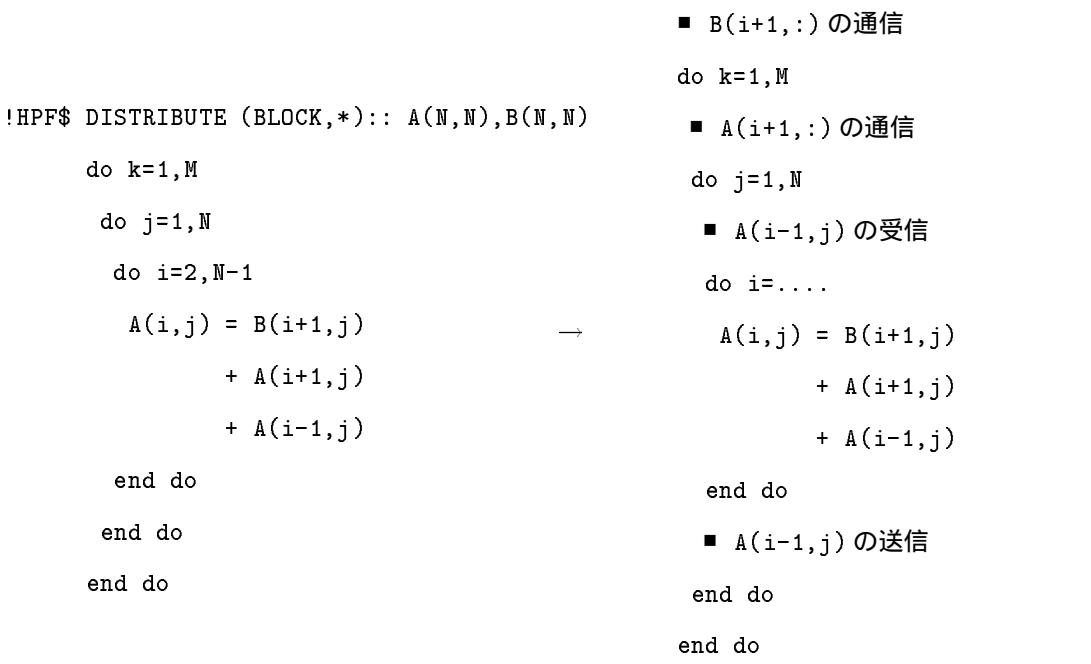


図 68: メッセージベクトル化法 (message vectorization)

える。このプログラム中の計算式の右辺のオペランドは境界を越える分に関して通信が必要である。すなわち、プロセッサ数を４、N を１００とした場合、B(i+1,j) に対して、メッセージベクトル化法を用いると、B(26,:) を隣接プロセッサから供給されることになる。また、B(i+2,j) に対して、tt B(26,:) と B(27,:) を隣接プロセッサから供給されねばならない。以下、B(i+3,j) および B(i+4,j) に対しても同様に通信が必要となる。

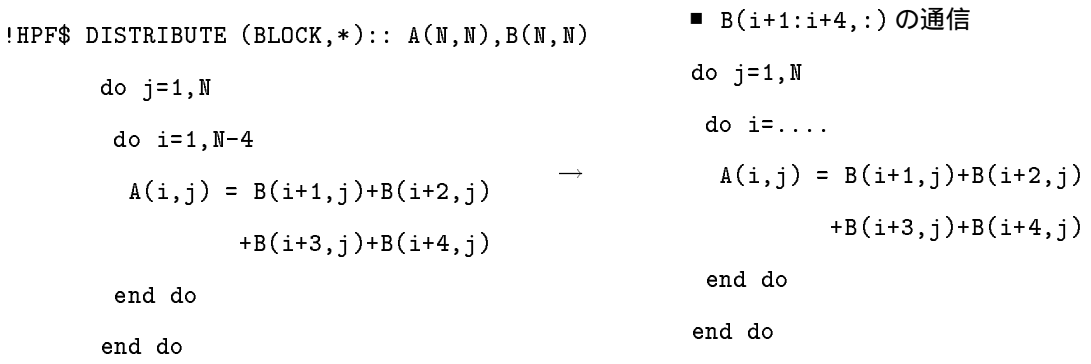


図 69: メッセージ合体法 (message coalescing)

このとき、ベクトル化された通信 B(26,:) は、４回起動されることになるが、実際は同じデータの通信であるので冗長である。また、冗長度を考慮すると、tt B(26,:)、tt B(27,:)、tt B(28,:)、

tt B(29,:) の４つの通信が残ることになるが、これらは同一送り元のメッセージとなるので、合体する (coalescing) ことができるので、実際必要となる通信メッセージは tt B(26:29,:) である (図 69の右図) 。メッセージ合体法により、メッセージ数は当初１０個必要であったが、１個に減少できた。なお、送るべき配列要素はメモリ中では非連続であるので、バッファエリアにコピーする必要がある。

メッセージ集団化法 (message aggregation)　メッセージ集団化法は、メッセージベクトル化法など最適化を施したあとのコード生成直前に行われる。生成された通信のパターンを比較し、同じ通信範囲を持つ複数のメッセージを１つのメッセージにあつめて、通信起動のコストを軽減するものである。

しかし、メッセージ集団化法を用いるためには、１つのメッセージにするために送信の際にバッファへのコピー動作が必要であり、受信の際にも同様にバッファから各配列要素へのコピーが必要となる。したがって、メッセージ集団化を適用する場合には、集団化することによる通信起動のコスト減とコピー動作によるコスト増とのトレードオフを考慮しなければいけない。

図 70の左図のプログラムに関してメッセージ集団化がどのように適用されるかを考える。

```
!HPF$ DISTRIBUTE (BLOCK,*):: A(N,N),B(N,N),C(N,N)
      do j=1,N
       do i=1,N-1
        A(i,j) = B(i+1,j)
        B(i,j) = C(i+1,j)
       end do
      end do
```

→

```
■ B(i+1,:),C(i+1,:)の通信
do j=1,N
 do i=....
  A(i,j) = B(i+1,j)
  B(i,j) = C(i+1,j)
 end do
end do
```

図 70: メッセージ集団化法 (message aggregation)

第１の計算式のオペランドとして配列要素 B(i+1,j) の境界値の通信が必要であり、メッセージベクトル化すると B(i+1,:) の通信となる。同様に、配列要素 C(i+1,j) も通信が必要で、通信範囲、送り元プロセッサなどは同一である。したがって、これらのメッセージを一つのメッセージとするために、送り元プロセッサの適当なバッファエリアに配列要素 B(i+1,j) と配列要素 C(i+1,j) をコピーして送信する。受信プロセッサでは、このメッセージを分解して、それぞれ、配列要素 B(i+1,j) と配列要素 C(i+1,j) に格納する (図 70の右図) 。

検査官 / 執行官法 (inspector/executor)　前の節までに紹介した手法は、コンパイル時に通信範囲や通信先プロセッサがある程度確定しているような場合に適用できるものであるが、間接アドレスをもった配列アクセスのような場合は、実行時になって初めて通信先などが決定される。

そこで、このような通信を 2 フェーズに分けて行う手法を検査官 / 執行官法 (inspector/executor) という。まず、検査官フェーズで、通信が発生する計算部分の通信パターンの解析を行い、ローカルデータだけで計算できる部分とそうでない部分に分け、通信が必要な部分に関しては、メッセージベクトル化法およびメッセージ集団化法によって、少ないメッセージに通信を詰め込むように通信スケジュール作成する。

つづく執行官フェーズでは、反復組み替え法 (後述する通信レイテンシ最適化手法) を用い、通信スケジュールにしたがって通信起動を行い、それとともにローカル計算を行う。さらに、受信データの確認を行った後に、リモート計算部分を行う。つまり、本手法によると、検査官フェーズで、個別発生するメッセージの通信起動コスト削減のための最適化を行うことができる。図 71 の左図のプログラムに検査官 / 執行官法を適用すると、右図のような SPMD プログラムが得られる。

```
!HPF$ DISTRIBUTE (BLOCK,*):: L(N,N),C(N,N)
!HPF$ DISTRIBUTE (BLOCK):: OLDA(N),A(N),IC(N)
      do itr=1,IMAX
       OLDA = A
       do i=1,N
        x=0.0
        do j=1,IC(i)
         x = x+C(i,j)*OLDA(L(i,j))
        end do
        if(IC(i).gt.0) A(i)=x
       end do
      end do
```

→

```
do itr=1,IMAX
 if(itr .eq. 1) then
  (検査官フェーズ)
  L(i,j) がローカルだけの計算部分を lset
  そうでないのを nlset と分ける。
  通信に必要なスケジュールを作成する。
 end if
 OLDA = A
 (執行官フェーズ)
 ■ 送信開始
 do i in lset
  (計算部分)
 end do
 ■ 受信確認
 do i in nlset
  (計算部分)
 end do
end do
```

図 71: 検査官 / 執行官法 (inspector/executor)

つまり、再外側ループの 1 回目の最初で検査官を起動して、各種の解析行い、2 回目以降は執行

官だけが実行される。しかし、検査官の処理は比較的重いので、通信パターンが頻繁に変わるようなプログラムの場合は、本手法は適用しづらい。Koelbel の論文によると、検査官及び執行官のコストに関して下記のような値が示されている。図 72に検査官のコストとベスト実行時間との比を表わしてある。ベスト実行時間とは、下記で表わされるものである。

$$\text{ベスト実行時間} = \frac{\text{逐次プログラムでの実行時間}}{\text{使用プロセッサ数}}$$

たとえば、逐次プログラムで 10 秒で実行されたプログラムは、4 プロセッサあった場合、ベストとしては 2.5 秒で実行されるものであると考える。

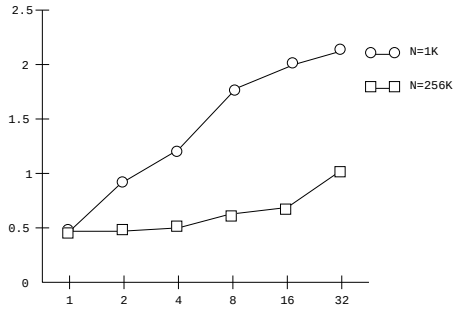


図 72: 検査官のコスト

グラフは、このベスト実行時間と検査官にかかった時間の比であって、たとえば、比が 2 の場合、検査官の実行時間はループ 2 回分の時間と同じだということであり、ループが 98 回繰り返された場合は、検査官のオーバーヘッドはせいぜい 2 あることを示している。

この図から明らかなように、配列サイズが大きいほど検査官のコストは見かけ上小さくなっている。したがって、十分な繰り返しを行うアプリケーションに対しては検査官のコストは無視できるほど小さいといえる。

一方、執行官を用いることによるオーバーヘッドに関して、図 73に、執行官を用いた時のスピードアップとリニアスピードアップとの比を表わしてある。

図から明らかなように配列サイズが小さい (N=1K) 場合、プロセッサ数が 32 であれば比が 3 倍以上になっているのでオーバーヘッドは無視できないが、十分大きい配列のとき (N=256K) は、せいぜい 20%くらいのオーバーヘッドであり十分実用になりうる。

以上のように、検査官 / 執行官法は、対象とする配列が十分大きく、繰り返し回数がある程度大

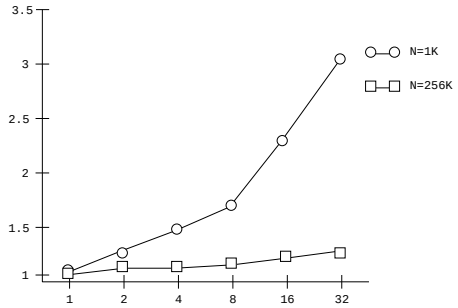


図 73: 執行官のオーバーヘッド

きいような場合には有効な手法であるが、これらの条件を満たさない場合は他の方法をとるほうが通信コストを小さくできる。

A.2 ADAPTOR システムの評価

ADAPTOR システムは、GMD で開発された HPF 言語に近い ADAPTOR 言語を処理するシステムである。容易に入手可能な HPF ライクな言語の処理系であるので、処理系の全体像を掴むには適したシステムである。本章では ADAPTOR システムの処理内容を評価する。

A.2.1 ADAPTOR system の概要

ADAPTOR (version 1.0, June, 1993) は、ADAPTOR プログラムを マシン native な SPMD FORTRAN プログラムに変換するトランスレータ (fadapt)[4, 6] と distributed array のための message passing ライブラリ (DALIB)[5] からなる。現在のところ、iPSC/860, PVM system, KSR1, CM5, Alliant などに移植されている。プログラミング支援として、Alliant 上では、trace が可能であり、その結果は、PARAGRAPH [14] で表示可能な形式で出力される。

以下のセクションで、ADAPTOR で使われる言語の概要、コンパイラ実装の要点、実行の評価を述べる。

A.2.2 ADAPTOR 言語

ADAPTOR (automatic DAta Parallelism TranslatOR) は、array extension, parallel loop, distributed array を指示するための directive などが付加された FORTRAN プログラムを入力とする。すなわち、FORTRAN 90, CM-FORTRAN, HPF (High Performance FORTRAN) などを、組み合わせたものであるが、必ずしも包含しているわけではない。以下で、言語の概要を示す。

まず、FORTRAN 90 からは、次の構文を用いている。

- array extensions

  配列演算の記述の簡易化及びコンパイラがそれを容易に認識できるために、array extension が許される。左の記述の代わりに、右の記述が可能である。

```
do j=1,N                          a=b+c
 a(j)=b(j)+c(j)
end do
```

- intrinsic function for arrays

  配列に関するいくつかの組み込み関数をサポートしている。

- dynamic arrays

  allocate 構文により、実行時に任意サイズの配列が生成できる。ただし、マシンによってはそのような関数を許していないものがある。その場合は、疑似動作になる。

- 新しい宣言文の記述方法

  宣言文の簡素化のために、左の記述の代わりに、右の記述が可能である。

```
real a(n,n),b(n,n),c(n,n)        real, dimension(n,n)::a,b,c
```

また、HPF からは、次の構文を借用している。

- distribution directive

  distribute directive を用いて分散形態を指定する。align 及び processor 構文は用いない。現バージョンでは、多次元配列で分散できる次元は、一次元で、BLOCK 分散だけである。distribute directive で指定されない配列に関しては、システムが自動的に分散指定する。また、分散させない (replicated) ことも指定できる。

- FORALL statement

  Copy-in Copy-out で並列実行される構文である。

```
forall(i=2:N,j=1:N)a(i,j)=a(i-1,j)
```

  上記の例では、flow dependence はなく、右辺の a(i-1,j) は、全て、更新前の値が用いられる。

- PURE

  PURE は、関数、サブルーチンに対する qualification であり、PURE なプロシージャは、並列に起動できる。引数となれるのは、distributed array element (distributed array ではない) である。

CM-FORTRAN からの借用部分は、省略する。最後に、ADAPTOR 独自の構文をあげる。

- independent loop

  do ループの実行が、確実に local access だけであることをシステムに知らせる。

- overlapping array

  分散配列が、アクセスのパターンにより重なりを持って分散させた方が良い場合がある。例えば、下の左のプログラムの場合、配列 a,b の境界部分で通信が必要になり、また、テンポラリ配列も必要になる (後述) 。したがって、配列 b にオーバーラッピング領域を指定しておくことにより、テンポラリ配列の生成が抑制できる。

```
real a(N,N)                        real a(N,N)
real b(N,N)                        real b(N[1:1],N[1:1])
do i=2,N-1                         do i=2,N-1
do j=2,N-1                         do j=2,N-1
 a(i,j)=b(i-1,j+1)+b(i+1,j-1)       a(i,j)=b(i-1,j+1)+b(i+1,j-1)
end do                             end do
end do                             end do
```

以上をまとめると、HPF の 公式 subset から、align を取り除けば、ADAPTOR がカバーしている範囲になるといえる。

A.2.3 コンパイラ実装

実際にプログラムを書き、コンパイラ実装上の特徴を調べる。

1. do より forall を使う方が高速。

   do だと runtime resolution を用いているようだが、forall の場合 compile time resolution になっている。

```
       real a(N,N)                        real a(N,N)
       real b(N,N)                        real b(N,N)
!hpf$ distribute(block,*)::a       !hpf$ distribute(block,*)::a
!hpf$ distribute(block,*)::b       !hpf$ distribute(block,*)::b
       do i=2,N                           forall(i=2:N,j=1:N)a(i,j)=i+j*N
       do j=1,N                           forall(i=2:N,j=1:N)b(i,j)=a(i-1,j)
        a(i,j)=i+j*N                      end
        b(i,j)=a(i-1,j)
```

```
      end do
      end do
      end
```

また、forall の場合、通信は、1 重ループに対して vectorization されている。約 10 倍程速くなる。

2. 複数次元の分散や cyclic の指定などを行なった場合は、default の分散になる。
3. 総和計算などは、reduce intrinsic を用いると、1000 倍以上速い。

```
      parameter(N=400)                  parameter(N=400)
      real a(N,N)                       real a(N,N)
      real b(N,N)                       real b(N,N)
!hpf$ distribute(block,*)::a      !hpf$ distribute(block,*)::a
!hpf$ distribute(block,*)::b      !hpf$ distribute(block,*)::b
      s=0.0
!hpf$ independent, local_access   !hpf$ independent, local_access
      do i=1,N                          do i=1,N
!hpf$ independent, local_access   !hpf$ independent, local_access
      do j=2,N                          do j=2,N
       s=b(i,j)+s                        reduce(SUM,s,b(i,j))
      end do                            end do
      end do                            end do
      end                               end
```

4. common の中の配列が distribute しているときは、storage sequence は保たない。一方、全ての配列が、replicate の時は、sequence association を保証する。
5. global_send/get は、do などで書いた場合より、約 2 倍速くなる。

```
      parameter(N=40)                   parameter(N=40)
      real a(N),b(N)                    real a(N),b(N)
      integer l1(N)                     integer l1(N)
```

```
!hpf$ distribute(block)::a,b      !hpf$ distribute(block)::a,b
!hpf$ distribute(block)::l1       !hpf$ distribute(*    )::l1
      forall(i=1:N) l1(i)=N+1-i         forall(i=1:N) l1(i)=N+1-i
      forall(i=1:N) a (i)=i             forall(i=1:N) a (i)=i
      call global_send(b,l1,a)
                                         do i=1,N
                                          b(l1(i))=a(i)
                                         end do
      end                                end
```

6. WHERE の mask と statement の lhs を align すると高速になる。
   但し、align しておく方が約 20 倍ほど速い。下記の例では、左の方が速い。

```
      parameter(N=40)                    parameter(N=40)
      real a(N,N),b(N,N)                 real a(N,N),b(N,N)
!hpf$ distribute(*,block)::a,b    !hpf$ distribute(block,*)::a,b
      where (b(1,:).eq.0.0)              where (b(1,:).eq.0.0)
       a(2,:) = 1.0                       a(2,:) = 1.0
      else where                         else where
       a(2,:) = b(1,:)                    a(2,:) = b(1,:)
      endwhere                           endwhere
      end                                end
```

7. サブルーチンの 実引数には、expression が許されていない
8. サブルーチンの 仮引数には、assumed-shape が許されていない
9. 実引数と仮引数は、同じ分散でなければいけない
10. pure procedure は、indepedent な do もしくは forall 内に置かないといけない。

```
      parameter(N=40)
      real a(N)
!hpf$ distribute(block)::a
```

```
!hpf$ independent, local_access        <---- pure の場合は、この行が必須
      do i=1,N
       a(i)=a(i)
       call pp(a,i)
      end do
      end
      pure subroutine pp(a,i)
      parameter(N=40)
      real a(N)
!hpf$ distribute(block)::a
      a(i)=a(i)+1.0
      end
```

A.2.4 実行評価

5 点差分 ADAPTOR でのプログラムの実行評価を行なうために、PVM[12], ADAPTOR 及びマシン独自の通信ライブラリで、5 点差分を計算するプログラムをを書き、速度を比較した[6]。PVM は、version 3.1 のものを用いた。使用したマシンは、16 プロセッサの iPSC/860 で、通信ライブラリには NX-2 を使用した。また、比較のために SUN-IPX (33MHz, 48Mbyte memory) の値も示す。

```
      parameter(N=512)
      real*8 u(N,N),v(N,N)
!hpf$ distribute(*,block)::u,v
      forall(i=2:N-1,j=2:N-1) u(i,j)=v(i-1,j)+v(i+1,j)+v(i,j-1)+v(i,j+1)
      end
```

このプログラムは、2 次元配列 u,v が、2 つめの次元に関して block 分散しているので、v(i,j-1), v(i,j+1) をフェッチする際に、 block 境界に当たる部分については、通信が生じる。

各々の結果の wallclock 時間 (単位は、10^{-2}秒) を示す。

[6] 付録 1 に、PVM, NX-2 のプログラムを示す。

	ADAPTOR	PVM	NX
計算	7.51	3.17	3.16
通信	5.48	1.21	0.61
合計	12.99	4.38	3.77

NX 及び PVM を使った時に比べて、 ADAPTOR は性能が落ちる。詳細は、次のセクションで述べるが、計算に関しては、次のような原因が考えられる。

- distributed array の分散情報を実行時に計算する
- 補助配列の使用に起因するキャッシュミス
- 配列アクセスのために offset 計算をしている

通信に関しては、通信範囲の決定を実行時に計算することなどに起因する速度低下が見られる。

一方、PVM の通信 の中には、 pack/unpack[7] する時間が含まれるので、NX よりもやや遅くなるが、計算は全く同じである。

実行の詳細　ADAPTOR で、コンパイルされた SPMD プログラムは、次の各部分からなる。

- 分散配列のサイズを確定する (サイズ確定)
- `v(i,j-1)`, `v(i,j+1)` を他のプロセッサと通信する (通信)
- 5 点差分を計算する (計算)

コンパイラ出力の SPMD プログラム、及び比較に使用した PVM プログラム及び NX-2 のプログラムを付録に示す。

サイズ確定

配列 `u`, `v` は、各プロセッサで分散されて持たれる。各々のプロセッサが、各々の配列のどの部分を保持しているかを計算する。

`dalib_array_pardim (512,U_LOW,U_HIGH)` は、配列 `u` の分散している次元の配列サイズが 512 の時、そのプロセッサが持つ範囲を `U_LOW` と `U_HIGH` に返す関数である。具体的には、次のようになる。

[7] heterogeneous な環境に対応するための型変換

```
U_HIGH = (pcb.i * 512 / pcb.p)
U_LOW  = (pcb.i * 512 / pcb.p) + 1
```

上記で、pcb.i は、そのプロセッサのプロセッサ番号、pcb.p は、プロセッサの総数を表す。

一方、U_DIM1 は、分散された次元の大きさを表し、U_OFS は、lower bound までのオフセットを表す。オフセットが必要な理由は、ADAPTOR で出力された SPMD は、全体として、global index のままで計算されている。したがって、適宜必要な時に、 local index に変換するので、このオフセットが前もって計算されている。

配列 v に関しても同様であり、また、通信された配列を保持するために、配列 u_tmp1, t_tmp2 が導入されており、これらも同様の処理がなされる。

このフェーズにかかる時間は、$5.77 \times 10^{-3} sec$ である。

通信

一部の v(i,j-1), v(i,j+1) は、local ではなく remote に保持されているので、計算に入る前にそれを通信する必要がある。通信を行なうために 3 つの関数を呼ぶ。

まず、dalib_move_define (512,2,N-1,512,2-1,N-1-1)は、各々のプロセッサの通信範囲を決定する関数である。最初の 3 引数が target 側のパラメータ、後の 3 つが source 側である。target 側は、512 の大きさを持つ次元の内、2 から N-1 を受信することを示す。この関数の中で、各々のプロセッサが受けとる範囲を決定し、 global 変数 target_size, target_low, target_up に値をセットする。 source 側も同様である。

dalib_move_source2 (V,8,512,2,N-1,512,2-1,N-1-1)は、送信を行なうもので、次の 2 つの関数が重要な役割を持っている。dalib_setup_section2は、転送すべき次元の local な情報、例えば、開始インデックスや増分値などを、sc_... というデータ構造にまとめる。dalib_send_section2 は、dalib_setup_section2 で作られた情報を元に、必要であれば、dalib_create_buffer 及び dalib_fill_buffer でバッファを作り、dalib_send_buffer もしくは asend で送信を行なう。

dalib_move_target2 (V,8,512,2,N-1,512,2-1,N-1-1) は、受信側であり、source とは、逆向きでほぼ同じ動作をする。

このフェーズにかかる時間を表にまとめる。初めの 3 つが v(i,j-1) の通信時間、後の 3 つが、v(i,j-1), v(i,j+1) の通信時間である。

dalib_move_define ()	$2.55 \times 10^{-5} sec$
dalib_move_source2 (V,..)	$1.67 \times 10^{-2} sec$
dalib_move_target2 (U_TMP1,..)	$1.24 \times 10^{-2} sec$
dalib_move_define ()	$3.18 \times 10^{-5} sec$
dalib_move_source2 (V,..)	$1.50 \times 10^{-2} sec$
dalib_move_target2 (U_TMP2,..)	$1.07 \times 10^{-2} sec$

主要なのは、dalib_move_source2 及び dalib_move_target2 であり、この両者を合わせて、$5.48 \times 10^{-2} sec$ である。

計算

計算部分は、PVM 及び NX で書いた時よりも約 2.5 倍遅くなっている。ループ前に、ループの local インデックスを決定するために、dalib_local_slice を呼んでいる。ループの内容は、本プログラムとほぼ同じであるが、v(i,j-1), v(i,j+1) の代わりに、u_tmp1(i,j),u_tmp2(i,j) をオペランドにしていることが相違点である。

このフェーズにかかる時間を表にまとめる。

dalib_local_slice ()	$2.20 \times 10^{-3} sec$
ループ本体	$7.51 \times 10^{-2} sec$

主要なのは、ループ本体であり、その演算時間は、$7.51 \times 10^{-2} sec$ である。

考察

ADAPTOR の実行結果は、通信、計算ともに、PVM, NX よりも遅い。

通信の遅さの原因は、実行時に、通信範囲を決定しており、そのための演算時間がかかっていることがあげられる。すなわち、通信すべきデータ量の多少に関わらず、関数毎にかかっている一定時間が大きいので、データを細切れにして送るような場合は、通信が大きな障害になると思われる。

例えば、元のプログラムを次のように書き換えた場合を考えてみる。

```
      parameter(N=512)
      real*8 u(N,N),v(N,N)
!hpf$ distribute(*,block)::u,v
```

```
      do 10 i=2,N-1
      do 10 j=2,N-1
10      u(i,j)=v(i-1,j)+v(i+1,j)+v(i,j-1)+v(i,j+1)
      end
```

元のプログラムと意味は同じであるが、現在のバージョンの ADAPTOR では、通信が combine されずに、ループ内に展開される。この場合、実行時間を示す。

通信	$15.9 \times 10^{0} sec$
計算	$7.51 \times 10^{-2} sec$

となり、通信時間が極めて大きくなる。

また、通信のための関数において、source 及び target が別々の関数であるが、機能を見てみると共通している部分が多く、同じことを 2 回しているものもあるようである。したがって、source_target を combine することにより、若干の速度の改善が可能と思われる。

一方、計算時間も、PVM などの約 2.5 倍になっている。この原因として次のものがあげられる。

- distributed array の分散情報を実行時に計算する
  これは、dalib_array_pardim によるもので、$0.58 \times 10^{-2} sec$ 程度の増加をもたらす。
- 補助配列の使用に起因するキャッシュミス
  通信のバッファとして、配列 u_tmp1，u_tmp2 を使用しているが、これにより、計算ループ内のアクセスが、キャッシュミスを多発させているようである。これによって計算時間が、2 倍程度増えている。
- 配列アクセスのために offset 計算をしている
  global index から local index への変換をループ内で行なっているが、このための OFFSET の加算が若干の速度低下をもたらせている。約 10 % の増加である。

以上の要因を加味すると、

$$(0.58 \times 10^{-2}) + (3.17 \times 10^{-2}) \times 2 \times 1.1 = (7.55 \times 10^{-2}) \simeq (7.51 \times 10^{-2})$$

となる。

A.2.5 まとめ

ここでは、ADAPTOR システムのあらまし、コンパイラ実装の特徴、実行性能について述べた。ADAPTOR は、HPF と CM-FORTRAN のサブセットに 若干の ADAPTOR 独自な構文を加えた FORTRAN 言語を入力としている。したがって、FORTRAN を使う多くのユーザにとっては、使用しやすいものになっている。

しかし、実行性能に関しては、標準的な通信ライブラリ (PVM) 及びマシン独自の通信ライブラリ (NX-2) で記述した場合と比較したが、数倍の遅さがあり不十分である。

また、通信に関する最適化をほとんど行なっておらず、おおむね runtime resolution[8] で処理されているので、現状のバージョンでは実用レベルのアプリケーションの実行には適さない。

A.2.6 ADAPTOR の評価に使用したプログラム

前節で行なった ADAPTOR システムの評価には、ADAPTOR 言語、PVM 及び NX-2 を通信ライブラリとして利用した C 言語の合計 3 つの言語で記述した 5 点差分プログラムを用いた。ここでは、PVM 及び NX-2 を用いたプログラム、及び ADAPTOR 言語で記述したプログラムのコンパイラ出力結果を示す。

ADAPTOR のコンパイラ出力

```
SUBROUTINE NODEMODULE ()
PARAMETER ( N = 512 )
INTEGER*4 U_LOW,U_HIGH,U_START,U_STOP,U_INC,U_DIM1,U_OFS
REAL*8 U (1:N,1:32)
INTEGER*4 V_LOW,V_HIGH,V_START,V_STOP,V_INC,V_DIM1,V_OFS
REAL*8 V (1:N,1:32)
REAL*8 T01
REAL*8 T02
REAL*8 DCLOCK
INTEGER*4 U_TMP2_LOW,U_TMP2_HIGH,U_TMP2_START,U_TMP2_STOP
INTEGER*4 U_TMP2_INC,U_TMP2_DIM1,U_TMP2_OFS
```

[8] 通信相手先、及びメモリアド レスの計算を実行時に行なう。

```
      REAL*8 U_TMP2 (1:N,1:32)
      INTEGER*4 U_TMP1_LOW,U_TMP1_HIGH,U_TMP1_START,U_TMP1_STOP
      INTEGER*4 U_TMP1_INC,U_TMP1_DIM1,U_TMP1_OFS
      REAL*8 U_TMP1 (1:N,1:32)
      LOGICAL*4 dalib_have_i
      EXTERNAL dalib_have_i
      INTEGER*4 dalib_pid
      EXTERNAL dalib_pid
      call dalib_array_pardim (512,U_LOW,U_HIGH)
      U_DIM1 = U_HIGH-U_LOW+1
      U_OFS = 1-U_LOW
      call dalib_array_pardim (512,V_LOW,V_HIGH)
      V_DIM1 = V_HIGH-V_LOW+1
      V_OFS = 1-V_LOW
      call dalib_array_pardim (512,U_TMP2_LOW,U_TMP2_HIGH)
      U_TMP2_DIM1 = U_TMP2_HIGH-U_TMP2_LOW+1
      U_TMP2_OFS = 1-U_TMP2_LOW
      call dalib_array_pardim (512,U_TMP1_LOW,U_TMP1_HIGH)
      U_TMP1_DIM1 = U_TMP1_HIGH-U_TMP1_LOW+1
      U_TMP1_OFS = 1-U_TMP1_LOW
      call dalib_move_define (512,2,N-1,512,2-1,N-1-1)
      call dalib_move_source2 (V,8,512,2,N-1,512,2-1,N-1-1)
      call dalib_move_target2 (U_TMP1,8,512,2,N-1,512,2,N-1)
      call dalib_move_define (512,2,N-1,512,2+1,N-1+1)
      call dalib_move_source2 (V,8,512,2,N-1,512,2+1,N-1+1)
      call dalib_move_target2 (U_TMP2,8,512,2,N-1,512,2,N-1)
cdir$ ivdep
      DO I=2,N-1
         call dalib_local_slice (512,2,N-1,U_START,U_STOP)
cdir$ ivdep
```

```
         DO J=U_START,U_STOP
            U(I,U_OFS+J) = V(I-1,V_OFS+J)+V(I+1,V_OFS+J)+
     $                     U_TMP1(I,U_TMP1_OFS+J)+U_TMP2(I,U_TMP2_OFS+J)
         END DO
       END DO
       END
```

PVM

```
/*
#define ARRAYSIZE 512
*/
#include <stdio.h>
#include "/projects/dsrg/pvm/release/include/pvm3.h"

main()
{
  /* computation variable */
  double *u,*v;
  int itr,j,k,klow,khigh;
  /* process control variable */
int mytid, ptid,i,me,NumProcs,Iteration,ARRAYSIZE,upk;
int tids[1024];
        double  tim1,tim2,dclock();

mytid = pvm_mytid();
ptid = pvm_parent();



  pvm_recv(ptid,0);
  pvm_upkint(&NumProcs,1,1);
  pvm_upkint(&Iteration,1,1);
```

```
pvm_upkint(&ARRAYSIZE,1,1);
pvm_upkint(&upk,1,1);
pvm_recv(ptid,0);
pvm_upkint(tids,NumProcs,1);
for(i=0;i<NumProcs;i++)
  if(mytid == tids[i]){me=i;break;}
u = (double *)malloc(sizeof(double)*ARRAYSIZE*(ARRAYSIZE/NumProcs+2));
v = (double *)malloc(sizeof(double)*ARRAYSIZE*(ARRAYSIZE/NumProcs+2));
klow = 1;
khigh=ARRAYSIZE/NumProcs;
if(me==0)        klow=1;
if(me==NumProcs-1) khigh=ARRAYSIZE/NumProcs-1;
tim1 = dclock();
for(itr=0;itr<Iteration;itr++){
  for(j=1;j<ARRAYSIZE-1;j++)
    for(k=klow;k<khigh;k++)
      u[j+ARRAYSIZE*k]= v[(j-1)+ARRAYSIZE*k]+ v[(j+1)+ARRAYSIZE*k]+
                v[j+ARRAYSIZE*(k-1)]+ v[j+ARRAYSIZE*(k+1)];

  if(me!=NumProcs-1){
    pvm_initsend( PvmDataRaw    );
    pvm_pkdouble(u+ARRAYSIZE*(ARRAYSIZE/NumProcs-2),ARRAYSIZE,1);
    pvm_send(tids[me+1],0);
  }
  if(me!=0){
    pvm_recv(tids[me-1],0);
    if(upk) pvm_upkdouble(u,ARRAYSIZE,1);
  }
  if(me!=0){
    pvm_initsend( PvmDataRaw    );
```

```
      pvm_pkdouble(u+ARRAYSIZE,ARRAYSIZE,1);
      pvm_send(tids[me-1],0);
    }
    if(me!=NumProcs-1){
      pvm_recv(tids[me+1],0);
      if(upk) pvm_upkdouble(u+ARRAYSIZE*(ARRAYSIZE/NumProcs-1),ARRAYSIZE,1);
    }
  }
  tim2 = dclock();
  tim2 = tim2-tim1;
  gdhigh(&tim2,1,&tim1);
  if(me==0) printf("Dclock is  %e.\n", tim2);

        pvm_initsend( PvmDataDefault);
        pvm_pkint(&NumProcs,1,1);
        pvm_send(ptid,0);

pvm_exit();
exit(0);

}
```

NX-2

```
#define ARRAYSIZE 512
#include <stdio.h>

main(argc,argv)
int argc;
char **argv;
{
  /* computation variable */
```

```
  double *u,*v;
  double tim1, tim2, dclock_();
  int itr,j,k,klow,khigh;

  /* process control variable */
int mytid, ptid,i,me,NumProcs,Iteration;
int tids[1024];

  sscanf(argv[1],"%d",&Iteration);
  mytid = mynode();
  me    = mynode();
  NumProcs = numnodes();
  u = (double *)malloc(sizeof(double)*ARRAYSIZE*(ARRAYSIZE/NumProcs+2));
  v = (double *)malloc(sizeof(double)*ARRAYSIZE*(ARRAYSIZE/NumProcs+2));

  klow = 1;
  khigh=ARRAYSIZE/NumProcs;
  if(me==0)        klow=1;
  if(me==NumProcs-1) khigh=ARRAYSIZE/NumProcs-1;
  tim1 = dclock_();
  for(itr=0;itr<Iteration;itr++){
    for(j=1;j<ARRAYSIZE-1;j++)
      for(k=klow;k<khigh;k++)
        u[j+ARRAYSIZE*k]= v[(j-1)+ARRAYSIZE*k]+ v[(j+1)+ARRAYSIZE*k]+
                  v[j+ARRAYSIZE*(k-1)]+ v[j+ARRAYSIZE*(k+1)];

    if(me!=NumProcs-1){
      csend(0,u+ARRAYSIZE*(ARRAYSIZE/NumProcs-2),ARRAYSIZE*8,mytid+1,0);
    }
    if(me!=0){
```

```
      crecv(0,u,ARRAYSIZE*8);
    }
    if(me!=0){
      csend(5,u+ARRAYSIZE,ARRAYSIZE*8,mytid-1,0);
    }
    if(me!=NumProcs-1){
      crecv(5,u+ARRAYSIZE*(ARRAYSIZE/NumProcs-1),ARRAYSIZE*8);
    }

  }
  gsync();
  tim2 = dclock_();
  tim2 = tim2-tim1;
  gdhigh(&tim2,1,&tim1);
  if(me==0) printf("Dclock is  %e.\n", tim2);

}
```

A.3 APR/xHPF コンパイラ

HPF 言語の最初の商用システムが Applied Parallel Research から発表されている。ここでは、いくつかの HPF プログラムとそのコンパイラ出力 (可読性の問題より中間言語をここでは扱う) を示し、その特徴を述べる。
まず、中間言語の例を示す。

```
    1: C
    2:        parameter (N = 100)
    3:        doubleprecision W1(N,N), W2(N,N)
    4: !HPF$ distribute(*,BLOCK)::w1,w2
  ---> partition W1 (*,SHRUNKBLOCK)
  ---> partition W2 (*,SHRUNKBLOCK)
  ---> Distribute the loop on W1<:, 2~1>
  ---> Use of  W1<2:99, 2~1>
  ---> Preloop  communication of  W2<2:99, 1~1>
  ---> Set of  W1<2:99, 2~1>
   11:        do   J=2,N-1
   12:          do   I=2,N-1
   13:            W1(I,J) = W2(I,J-1) + W1(I,J)
   14:          end do
   15:        end do
*  16:        write (6, *) W1(1,1)
   17:       end
```

最初の 2 つの annotation "partition" は、配列 W1 と W2 を "(*,SHRUNKBLOCK)" に分散させることを指示しており、次の "Distribute the loop on W1<:, 2 1>" は、引き続くループを、W1 の 2 番めの次元で分散するようにすることを表す。つまり、行番号 11 のループを分散するのである。"<:, 2 1>" は、1 番めの次元は分散せずに、2 番めの次元を 2 で始まり、ステップを 1 にするように、分散することを示している。また、annotation "use" と "set" は、ここではあまり重要な意味を持っていない。Preloop communication は、配列 W2 を ループ前に通信することを示している。

なお、行番号 16 の write 文は、あまり意味がないが、コンパイラにとっては必要である。というのは、プログラム中で使用 (use) されていない配列は、コンパイラは正確には解析を行なわない (行なう必要がない) ので、コンパイラの挙動を確かめるには、必要である。

基本差分計算 1　基本的な配列計算をどのように扱うかを見るために、簡単な差分計算を含むプログラムをコンパイルした。

```
      parameter (N = 5)
      double precision X(N,N), Y(N,N), Z(N,N)
      integer LL(N,N)
!HPF$ distribute(BLOCK,*)::x,y,z,ll
      do   I=1,N
        do   J=2,N-1
          X(I,J) = Y(I,J-1) + Z(I,J+1)
        end do
      end do
      do   J=1,N
        do   I=2,N-1
          X(I,J) = X(I+1,J) + Z(I-1,J)
        end do
      end do
      write (6, *) X(5,5), LL(4,4), 99
      end
```

```
  1:       parameter (N = 5)
  2:       doubleprecision X(N,N), Y(N,N), Z(N,N)
  3:       integer LL(N,N)
  4: !HPF$ distribute(BLOCK,*)::x,y,z,ll
—> partition X (SHRUNKBLOCK,*)
—> partition Y (SHRUNKBLOCK,*)
—> partition Z (SHRUNKBLOCK,*)
—> partition LL (SHRUNKBLOCK,*)
—> Distribute the loop on X<1~1, :>
—> Use of  Y<1~1, 1:3>
—> Use of  Z<1~1, 3:5>
—> Set of  X<1~1, 2:4>
 13:       do   I=1,N
 14:         do   J=2,N-1
 15:           X(I,J) = Y(I,J-1) + Z(I,J+1)
 16:         end do
 17:       end do
 18:       do   J=1,N
—> Distribute the loop on X<2~1, :>
—> Preloop  communication of  X<3~1, j>
—> Preloop  communication of  Z<1~1, j>
—> Set of  X<2~1, j>
 23:         do   I=2,N-1
 24:           X(I,J) = X(I+1,J) + Z(I-1,J)
 25:         end do
 26:       end do
*27:       write (6, *) X(5,5), LL(4,4), 99
 28:      end
```

最初のループであるが、配列 X と Y と Z は、お互いが align されており、通信は不要である。外側ループの index とそれらの配列の分散されている次元が一致するので、問題なく外側ループが、並列化される。なお、このループでは、外側、内側いずれも並列化可能であるが、一般的には、 synchronization の回数を減らすために、外側ループを並列化される。

しかし、2 番めのループでは、内側ループが配列の分散している次元と一致するので、内側ループが並列化される。この場合、配列要素 X(I+1,J) と Z(I-1,J) は、1 番めの次元が分散されているので、通信する必要がある。

基本差分計算 2　別の基本的な計算についてみる。

```
      parameter (N=100)
      double precision,dimension(N,N)::x,y,z
!hpf$ distribute(BLOCK,*)::x,z
!hpf$ distribute(*,BLOCK)::y
      do j=2,N
        do i=2,N
          x(i,j)=x(i-1,j)+x(i,j-1)
        end do
      end do
      do j=2,N
        do i=2,N
          x(i,j)=x(i-1,j)
        end do
      end do
      do j=2,N
        do i=2,N
          x(i,j)=x(i-1,j-1)
        end do
      end do
      z = x
      y = x
      write(6,*)z(1,1),y(1,1)
      end
```

```
  1:       parameter (N = 100)
  2:       doubleprecision X(N,N), Y(N,N), Z(N,N)
  3:       integer II1, II2
  4: !HPF$ distribute(BLOCK,*)::x,z
  5: !HPF$ distribute(*,BLOCK)::y
—> partition X (SHRUNKBLOCK,*)
—> partition Z (SHRUNKBLOCK,*)
—> partition Y (*,SHRUNKBLOCK)
  9:       do  J=2,N
 10:         do  I=2,N
*11:           X(I,J) = X(I-1,J) + X(I,J-1)
 12:         end do
 13:       end do
—> Distribute the loop on X<:, 2~1>
—> Preloop  communication of  X<1:99, 2~1>
—> Postloop communication of  X<2:n, 2~1>
 17:       do  J=2,N
 18:         do  I=2,N
 19:           X(I,J) = X(I-1,J)
 20:         end do
 21:       end do
 22:       do  J=2,N
—> Distribute the loop on X<2~1, :>
—> Preloop  communication of  X<1~1, j-1>
—> Set of  X<2~1, j>
 26:         do  I=2,N
 27:           X(I,J) = X(I-1,J-1)
 28:         end do
 29:       end do
—> Distribute the loop on Y<:, 1~1>
—> Preloop  communication of  X<1:100, 1~1>
—> Postloop communication of  Z<1:100, 1~1>
—> Set of  Y<1:100, 1~1>
 34:       do  II1=1,100
 35:         do  II2=1,100
 36:           Z(II2,II1) = X(II2,II1)
 37:           Y(II2,II1) = X(II2,II1)
 38:         end do
 39:       end do
*40:       write (6, *) Z(1,1), Y(1,1)
 41:     end
```

1 番めのループは、配列 X が、ループ境界を越えたフロー依存があるので、並列化ができない。

2 番めのループは、内側ループにフロー依存があるため、外側ループが並列化される。しかし、外側ループの index と配列 X の分散されている次元が異なるので、その配列は ループの前で (BLOCK,*) 分散から (*,BLOCK) に移動され、ループ終了後に反対方向に移動される。これは、実質的な、再分散 (redistribution) である。

Array assignment[9] は、トランスレータにより、tightly nested なループに展開され、1 番めの次元が内側ループになるようになっている[10]。Array assignment “z=x” に対し、それぞれの配列は 1 番めの次元上で分散され、その次元は内側ループの index に一致しているので、外側ループの前後に再分散するための通信が必要となる。しかし、この通信は避けることができる。まず、ループ全体を loop distribution して、 2 つのループに分ける。さらに、“z=x” に対応する方のループの loop interchange を行なえば、外側ループの index と分散されている次元が一致

[9] FORTRAN 90 で定義されている、copy-in copy-out に基づく配列の一斉代入構文である。

[10] キャッシュ局所性を上げるためであると推測する

するようにできるので、通信は不要となる。

次元の異なる配列　1 次元の配列と 2 次元の配列が混合された場合について調べる。

```
      parameter (N=100)
      double precision,dimension(N,N)::w1,w2
      double precision,dimension(N,N)::x1,x2
      double precision,dimension(N)::y1,y2,y3,y4
!hpf$ distribute(*,BLOCK)::w1,w2
!hpf$ distribute(BLOCK,*)::x1,x2
!hpf$ distribute(BLOCK)::y1,y2,y3,y4
      do j=1,N
        do i=1,N
          y1(j)=y1(j)+w1(i,j)*w2(i,j)
        end do
      end do
      do i=1,N
        do j=1,N
          y2(j)=y2(j)+w1(i,j)*w2(i,j)
        end do
      end do
      do j=1,N
        do i=1,N
          y3(j)=y3(j)+x1(i,j)*x2(i,j)
        end do
      end do
      do i=1,N
        do j=1,N
          y4(j)=y4(j)+x1(i,j)*x2(i,j)
        end do
      end do
      write(6,*)y1(1),y2(1),y3(1),y4(1)
      end
```

```
  5:       parameter (N = 100)
  6:       doubleprecision W1(N,N), W2(N,N)
  7:       doubleprecision X1(N,N), X2(N,N)
  8:       doubleprecision Y1(N), Y2(N), Y3(N), Y4(N)
  9: !HPF$ distribute(*,BLOCK)::w1,w2
 10: !HPF$ distribute(BLOCK,*)::x1,x2
 11: !HPF$ distribute(BLOCK)::y1,y2,y3,y4
—> partition W1 (*,SHRUNKBLOCK)
—> partition W2 (*,SHRUNKBLOCK)
—> partition X1 (SHRUNKBLOCK,*)
—> partition X2 (SHRUNKBLOCK,*)
—> partition Y1 (SHRUNKBLOCK)
—> partition Y2 (SHRUNKBLOCK)
—> partition Y3 (SHRUNKBLOCK)
—> partition Y4 (SHRUNKBLOCK)
—> Distribute the loop on Y1<1^1>
—> Use of  W1<1:n, 1^1>
—> Use of  W2<1:n, 1^1>
—> Use of  Y1<1^1>
—> Set of  Y1<1^1>
 25:       do   J=1,N
 26:         do   I=1,N
 27:           Y1(J) = Y1(J) + W1(I,J) * W2(I,J)
 28:         end do
 29:       end do
 30:       do   I=1,N
—> Distribute the loop on Y2<1^1>
—> Use of  W1<i, 1^1>
—> Use of  W2<i, 1^1>
—> Use of  Y2<1^1>
—> Set of  Y2<1^1>
 36:         do   J=1,N
 37:           Y2(J) = Y2(J) + W1(I,J) * W2(I,J)
 38:         end do
 39:       end do
—> Distribute the loop on Y3<1^1>
—> Use of  Y3<1^1>
—> Preloop  communication of  X1<1:n, 1^1>
—> Preloop  communication of  X2<1:n, 1^1>
—> Set of  Y3<1^1>
 45:       do   J=1,N
 46:         do   I=1,N
 47:           Y3(J) = Y3(J) + X1(I,J) * X2(I,J)
 48:         end do
 49:       end do
—> Distribute the loop on X1<1^1, :>
—> Reduction function ADD on  Y4<1:n>
—> Use of  X1<1^1, 1:n>
—> Use of  X2<1^1, 1:n>
 54:       do   I=1,N
 55:         do   J=1,N
 56:           Y4(J) = Y4(J) + X1(I,J) * X2(I,J)
 57:         end do
 58:       end do
*  59:       write (6, *) Y1(1), Y2(1), Y3(1), Y4(1)
 60:     end
```

1 つめのループは、配列 `Y1` と `W1` と `W2` の分散されている次元が外側ループの index に一致するので、容易に並列化できる。

2 番めのループも容易に並列化できるが、内側ループが並列化されるので、1 番めのループに比べて、synchronization point が増え、性能が落ちる可能性がある。しかし、この場合も、コンパイラが、loop interchange を柔軟に適用し、内側と外側を入れ換え、外側ループが並列化でき

るようにすることは可能である。

配列 X1 及び X2 は、Y3 には align されていない。したがって、3 つめのループは、ループの前後で通信が必要とされる。

4 番めのループは、 2 番めに似ているが、コンパイラは別のアプローチを行なっている。考えられるアプローチは、次の 2 つである。

- 2 つめのループ同様、内側ループを並列化する。配列 X1 と X2 の通信が必要 (redistribution/realign) 。
- 外側ループの index が 配列 X1 及び X2 の分散されている次元に一致するので、外側ループが並列化される。配列 Y4 の加算計算は、reduction 計算と認識され、ループ後に、reduction 通信が行なわれる。

コンパイラは、通信量の少ない 2 つめの方法を採用している。なお、4 つめのループの計算は、owner computes rule にはなっていない。

FORALL 文　HPF の中心的な並列構文である FORALL について調べる。

```
      parameter (N=500)
      double precision,dimension(N,N)::x
!hpf$ distribute(*,BLOCK)::x
      forall(i=1:N,j=1:N)x(i,j)=x(i-1,j)
      forall(j=1:N,i=1:N)x(i,j)=x(i-1,j)
      write(6,*)x(1,1)
      end
```

```
  7:       parameter (N = 500)
  8:       doubleprecision X(N,N)
  9: !HPF$ distribute(*,BLOCK)::x
 10:       doubleprecision DD1(500,500), DD2(500,500)
—> partition X (*,SHRUNKBLOCK)
—> partition DD2 (*,SHRUNKBLOCK)
—> partition DD1 (SHRUNKBLOCK,*)
 14:       do  I=1,N
—> Distribute the loop on X<:, 1~1>
—> Use of  X<i-1, 1~1>
—> Set of  DD1<1~1, i>
 18:         do  J=1,N
 19:           DD1(J,I) = X(I-1,J)
 20:         end do
 21:       end do
 22:       do  I=1,N
—> Distribute the loop on X<:, 1~1>
—> Use of  DD1<1~1, i>
—> Set of  X<i, 1~1>
 26:         do  J=1,N
 27:           X(I,J) = DD1(J,I)
 28:         end do
 29:       end do
—> Distribute the loop on X<:, 1~1>
—> Use of  X<0:499, 1~1>
—> Set of  DD2<1:n, 1~1>
 33:       do  J=1,N
 34:         do  I=1,N
 35:           DD2(I,J) = X(I-1,J)
 36:         end do
 37:       end do
—> Distribute the loop on X<:, 1~1>
—> Use of  DD2<1:n, 1~1>
—> Set of  X<1:n, 1~1>
 41:       do  J=1,N
 42:         do  I=1,N
 43:           X(I,J) = DD2(I,J)
 44:         end do
 45:       end do
*  46:       write (6, *) X(1,1)
 47:     end
```

FORALL 文は、tightly nested ループに変換され、1 番めの triplet-spec[11]が、最外側ループ、2 番めの triple-spec が、2 番めの外側ループのようになる。したがって、最初の FORALL 文では、`i` ループが外側になる。しかし、配列 `X` の 2 番めの次元が分散されているので、内側ループが並列化される。なお、FORALL 文は、array assignment 同様、copy-in copy-out[12] で実行される。しかがって、いかなるデータ依存関係も生じない。

一時配列 `DD1` は、配列要素 `X(I-1,J)` を保持するために使用される。コンパイラは、その配列要素を一時配列に、“`DD1(I,J)=X(I-1,J)`” ではなく、“`DD1(J,I)=X(I-1,J)`” となるように、代入している。これは、配列 `DD1` のアクセスが、線形になるようにしているものであり、メモリもしくはキャッシュのアクセスの空間的局所性を高められるようにするためである。

サブルーチンコール 1　サブルーチンコールを使った次のプログラムを考える。

[11] “`forall(i=1:10:2) x(i)=1.0`” の中の”`i=1:10:2`” に対応する部分。

[12] 概念上は、assignment の右辺は全てループ前に fetch され、左辺は、ループ後に store されること。

```
      parameter (N=100)
      double precision,dimension(N)::y1,y2,y3,y4
!hpf$ distribute(BLOCK)::y1,y2,y3,y4
      do j=1,N
        call subx(y1(j))
      end do
      do j=1,N
        call sub2(y2(j))
      end do
      write(6,*)y1(1),y2(1)
      end

      subroutine subx(x)
      double precision x
      x=1.0
      end

      subroutine sub2(x)
      double precision x(1)
      x(1)=2.0
      end
```

```
  6:       parameter (N = 100)
  7:       doubleprecision Y1(N), Y2(N), Y3(N), Y4(N)
  8: !HPF$ distribute(BLOCK)::y1,y2,y3,y4
—> partition Y2 (SHRUNKBLOCK)
—> partition Y3 (SHRUNKBLOCK)
—> partition Y4 (SHRUNKBLOCK)
—> partition Y1 (FULLBLOCK)
—> Distribute the loop on Y1<1^1>
—> Set of  Y1<1^1>
 15:       do   J=1,N
 16:         call SUBX (Y1(J))
 17:       end do
—> Distribute the loop on Y2<1^1>
—> Set of  Y2<1^1>
 26:       do   J=1,N
 27:         call SUB2 (Y2(J))
 28:       end do
*  29:       write (6, *) Y1(1), Y2(1)
 30:      end

  2:       subroutine SUBX (X )
  3:         doubleprecision X
  4:         X = 1.0
  5:       end

  2:       subroutine SUB2 (X )
  3:         doubleprecision X(1)
*  4:         X(1) = 2.0
  5:       end
```

本質的に、サブルーチン subx と sub2 は同じものである。しかし、サブルーチン subx は並列化されているが、サブルーチン sub2 はされない。一般に、サブルーチンの仮変数に配列があれば、その配列を用いた計算は並列化されない。というのは、そのサブルーチンコールを行なったプロセッサが、その配列要素を持っているかどうかはコンパイル時には分からないからである。しかし、コンパイラが interprocedual 解析を十分やっておれば、サブルーチン sub2 と subx は、全く同じものであることが分かり、両方とも並列化できるはずである。

サブルーチンコール 2　ここでは、プロシージャ境界を越えた分散情報の扱いについて調べる。

```
      parameter (N=100)
      double precision,dimension(N,N)::x,y
!hpf$ distribute(BLOCK,*)::x
!hpf$ distribute(*,BLOCK)::y
      call sub4(x,y,N)
      call sub4(y,x,N)
      call sub7(x,y,N)
      write(6,*)x(1,1),y(1,1)
      end

      subroutine sub4(x,y,N)
      double precision,dimension(N,N)::x,y
!hpf$ inherit ::x,y
      forall(i=1:N,j=1:N)x(i,j)=y(i,j)
      end

      subroutine sub7(x,y,N)
      double precision,dimension(N,N)::x,y
!hpf$ inherit ::x,y
      forall(i=1:N,j=1:N)x(i,j)=y(i,j)
      end
```

```
  1:       parameter (N = 100)
  2:       doubleprecision X(N,N), Y(N,N)
  3: !HPF$ distribute(BLOCK,*)::x
  4: !HPF$ distribute(*,BLOCK)::y
 —> partition X (SHRUNKBLOCK,*)
 —> partition Y (*,SHRUNKBLOCK)
*  9:       call SUB4 (X,Y,N)
* 10:       call SUB4 (Y,X,N)
* 11:       call SUB7 (X,Y,N)
* 16:       write (6, *) X(1,1), Y(1,1)
 17:      end

  2:       subroutine SUB4 (X, Y, N )
  3:         doubleprecision X(N,N), Y(N,N)
  4: !HPF$ inherit ::x,y
 —> Distribute the loop on X<1~1, :>
 —> Preloop  communication of  Y<1~1, 1:n>
 —> Postloop communication of  X<1~1, 1:n>
  8:         do  I=1,N
  9:           do  J=1,N
 10:             X(I,J) = Y(I,J)
 11:           end do
 12:         end do
 13:       end

  2:       subroutine SUB7 (X, Y, N )
  3:         doubleprecision X(N,N), Y(N,N)
  4: !HPF$ inherit ::x,y
 —> Distribute the loop on X<1~1, :>
 —> Preloop  communication of  Y<1~1, 1:n>
 —> Set of  X<1~1, 1:n>
  8:         do  I=1,N
  9:           do  J=1,N
 10:             X(I,J) = Y(I,J)
 11:           end do
 12:         end do
 13:       end
```

配列 X 及び Y は、異なった分散形態を持ち、サブルーチン sub4 と sub7 の仮変数は、実変数の分散情報を継承 (inherit) する。

サブルーチン sub4 は、sub7 とほとんど同じであるが、 sub4 が、異なった分散形態を持つ実変数を持って、 2 回呼ばれていることが、sub7 と異なる (行番号 9 及び 10 参照) 。sub4 及び sub7 のいずれにおいても、外側ループが並列化される。

サブルーチン sub4 において、仮変数の分散形態はコンパイル時には決定されない (定数でない) ので、仮変数の分散状態を実行時に検索し、必要であれば、通信を行なう関数が、ループの前後に挿入される。しかし、sub7 では、仮変数の分散がコンパイル時に決定されているので、必要な通信だけが挿入される。仮変数 X の分散されている次元は、外側ループの index と同じである。