

データ並列言語における通信と計算のオーバーラップの効果

李 曉傑 原田 賢一

慶應義塾大学大学院 計算機科学専攻
横浜市港北区日吉 3-14-1

概 要

データ並列プログラムの実行においては、データ転送にかかるコストが大きく、これがプログラムの実行効率を低下させる要因の1つになっている。データ転送最適化の手法の1つとして、通信と計算をオーバーラップさせる方法が提案されている。計算とデータ転送を可能な限りオーバーラップさせるために、本稿では、従来の一重オーバーラップとは異なり、多重オーバーラップを導入する。このため、まず、 N レベルメッセージキューと呼ぶオーバーラップ機構を提案する。次に、データフロー解析の結果に基づき、このキューを利用したオーバーラップのコンパイル手法を述べる。続いて、オーバーラップ可能なプログラムを用いて、本手法によるオーバーラップ、PVMによるオーバーラップ、およびオーバーラップしない場合の実行を試みる。この実験から、本稿で提案しているオーバーラップ機構の有効性が示された。

Evaluating the Efficiency of Overlapping Communication and Computation in Data Parallel Languages

Xiaojie LI Ken'ichi HARADA

*Department of Computer Science, Keio University
3-14-1, Hiyoshi, Kouhoku-ku, Yokohama, 223, Japan*

Abstract

In data parallel program executions, reducing the overhead of data transmissions is crucial to harnessing the potential of distributed memory multiprocessors. Overlapping communication and computation in programs can be used to hide the overhead of data transmissions. In this paper, we propose a new overlapping mechanism called as N -level message queue for nested overlapping in data parallel languages. Based on an exact data-flow analysis on individual array element accesses, we show the compile techniques using our message queue to overlap communication and computation. We evaluated our overlapping mechanism, and compared it with PVM, all results of experiments showed that our overlapping mechanism is efficient.

1 はじめに

データ並列言語では、データを配置するために、分割配置の仕方を指示するディレクティブを、逐次プログラムの中に挿入した形式が主に用いられる。一般に、データ並列言語のコンパイラでは、Owner Computes Ruleに従って、代入文の実行を担当する各プロセッサを決め、SPMD形式の目的プログラムを生成している。分散メモリマシンの場合、データ転送はメッセージパッシングによって実現され、そのコストが大きいため、データ転送がプログラムの実行効率を低下させる原因となる。通信コードの最適化として、通信と計算をオーバーラップさせる方法が提案されている[5]。文献[6]では、High Performance Fortran (HPF)の代入文を評価する時に、通信と計算をオーバーラップするためのアルゴリズムが示されている。計算とデータ転送を可能な限りオーバーラップさせるために、本稿では、従来の一重オーバーラップとは異なり、多重オーバーラップを導入し、その効果を示す。

プロセッサ P_i 上の計算点 n において、データ x が参照されるとする。 x が n の先行節点 m で更新され、 m から n までの経路上で x の値が更新されることがなければ、 x を m の出口で転送することによって、節点 m から n までの経路上での計算と m の出口での転送とをオーバーラップさせることができる。このような転送をプログラムの静的解析によって検出するには、グローバルな範囲での配列要素の参照と代入の関係を解析する必要がある。また、節点 m から n までの経路上で新たなデータ転送が必要となるときに、 x の転送とこの転送とに対して、それぞれの優先順位を指定できることが望ましい。そのためには、優先順位付きでデータを転送するためのオーバーラップ機構が必要である。

本稿では、 N レベルメッセージキューと呼ぶオーバーラップ機構を提案する。このキューは、プログラム呼出し(以下、 N レベルキュー呼出しという)によってデータと優先順位を受け取り、優先順位に従って目的プロセッサに転送する。プロセッサ P_i が P_j へデータを転送する時、 P_i は、そのデータを N レベルキューに送るだけで、受信側のリプレイを待たずに、計算を開始することができる。優先順位はレベルによって表し、最上位レベルを N 、最下位レベルを 1 とする。

例 1 図1のサンプルプログラムについてを考える。Owner Computes Ruleに従って、代入文の右辺で参照される配列要素は左辺の配列要素を所有するプロセッサに転送される必要がある。このプログラムの実行でもっとも時間がかかると思われる部分は、ループ s25 と s26 である。 $A(I, J+1)$ が内側のループで参照されるので、 $A(I, J+1)$ を最優先で転送する必要がある(レベル3)。 $B(I+3)$ は外側のループで参照されるので、その転送の優先度はレベル2でよい。s34 で $C(I+3)$ が参照されているが、s21 から s32 までの経路上には $C(I+3)$ への更新がないので、 $C(I+3)$ は s20 で転送の要求を出しておくことができる(レベル1)。

N レベルキュー呼出しを SPMD のどの節点に挿入したらよいか、そしてそのレベルをいくつに設定したらよいかを決定するためには、プログラムの流れに沿って配列に対するアクセス状況を解析する必要がある。

本稿の構成は次のとおりである。まず、2章で、 N レベルキューの操作規則、およびアルゴリズムを示す。3章では、区間データ

```

DIMENSION A(100,100), B(100), C(100), D(100)
!HPF$ ALIGN (I) WITH TEMPLATE(I,1) :: B,C,D
!HPF$ ALIGN (I,J) WITH TEMPLATE(I,J) :: A
!HPF$ DISTRIBUTE TEMPLATE(CYCLIC,CYCLIC)

.....
s20: # get C(4:53) at level 1 of the queue
s21: DO I= 1, 100
s22:   B(I) = ...
s23: ENDDO
s24: # get B(4:53) at level 2 of the queue
s25: DO I = 1, 50
s26:   DO J = 1, 99
s27:     # get A(I,J+1) at level 3 of the queue
s28:     A(I, J) = ... A(I, J+1) ...
s29:   ENDDO
s30:   D(I) = ... B(I+3) ...
s31: ENDDO
s32: .....
s33: DO I = 1, 50
s34:   D(I) = 3*C(I+3)
s35: ENDDO

```

図 1: サンプル プログラム

フロー解析法による配列参照情報を求める手順を示し、 N レベルキュー呼出しの挿入規則を与える。4章では、実プログラムを用いて、オーバーラップの効果と本手法の有効性を示す。

2 N レベルキュー機構

プロセッサ間でのデータ転送のために、ソフトウェアによって実現される N レベルメッセージキュー(以下、 Q_n と呼ぶ)を導入する。このキューは N 個のチャネルからなる。高いレベルにあるデータが低いレベルにあるデータよりも先に転送され、同一レベルにあるデータは FIFO で転送される。 k 個のデータが $m_1, m_2, \dots, m_{k-1}, m_k$ の順序で N レベルキューに送られてきたとき、キューの状態を $\langle m_k, m_{k-1}, \dots, m_2, m_1 \rangle$ で表す。これらのデータはそれぞれ指定されたレベルに従って転送されるので、キューの入口での状態と出口での状態は必ずしも同じとは限らない。例えば、プロセッサ P_i が 2 つのデータ m_1, m_2 をプロセッサ P_j に送るときに、キューの入口と出口での状態をそれぞれ $Entry, Exist$ とすると、 m_2 が m_1 よりも優先順位が高ければ、 $Entry = \langle m_2, m_1 \rangle$ 、 $Exist = \langle m_1, m_2 \rangle$ となる。

N レベルキューの機能を実現するアルゴリズムを示すために、まずキューの操作規則を示す。演算規則 G は、同一レベルにあるデータに対する規則 G_n と異なるレベルにあるデータに対する規則 G_h からなる。 $Entry$ に G を適用することによって、 $Exist$ が得られる。これを $Entry \xrightarrow{G} Exist$ で表す。 S_1, S_2 、および S_3 をデータ列とし、キューの入口での状態を $\langle S_1 m S_2 \rangle$ とする。このとき、 m の優先順位がもっとも高ければ、 $\langle S_1 m S_2 \rangle \xrightarrow{G} \langle S_3 m \rangle$ となり、 m

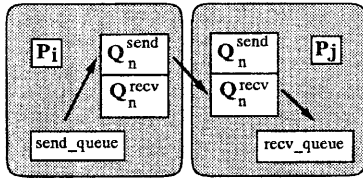


図 2: N レベルメッセージキュー

は最優先で転送される。演算規則 G は次のとおりである。

$$\begin{aligned}
 G &= G_u \cup G_h \\
 G_u : \langle S_2 m_i m_j S_1 \rangle &\rightarrow \langle S_2 m_j m_i S_1 \rangle \\
 &\text{iff } m_i = m_j \\
 G_h : \langle S_2 m_i m_j S_1 \rangle &\rightarrow \langle S_2 m_j m_i S_1 \rangle \\
 &\text{iff } m_i.\text{level} > m_j.\text{level}
 \end{aligned}$$

Q_n は 1 つのタスクとして、各プロセッサでの SPMD プログラムと同時に実行される。SPMD がキュー呼出しを用いて、データを Q_n へ送り、 Q_n の操作によって、そのデータが目的のプロセッサに転送される。 N レベルキュー呼出しの形式は次のとおりとする。

send_queue(sender_ID, receiver_ID, level, message_length, element_list)
recv_queue(sender_ID, receiver_ID, level, message_length, element_list)

sender_ID と *receiver_ID* は、それぞれ送信側と受信側のプロセッサ番号であり、*level* はキューのレベルを示す。*element_list* はデータ列そのものを表し、*message_length* はデータ列のバイト数を表す。*send_queue* はデータを Q_n に送って、その呼出しが終了するのに対し、*recv_queue* は Q_n にデータ受信を要求した後、要求が満たされたまで、その呼出しの終了は保留される。

代入文を実行するとき、同期操作を簡単にするために、代入文の右辺に現れるすべての配列要素は受け取ってから、右辺の評価が始められるものとする。 N レベルキュー呼出しの挿入位置を示すために、本稿では、コメント “# get Array($n:m$) at level j of the queue” を用いている。実際の呼出しとその挿入位置は、後述するように、コンパイラによって決定される。“# get Array($n:m$) at level j of the queue” では、Array($n:m$) の転送が複数のプロセッサの間で行われるので、SPMD には次のような通信コードを挿入しなければならない。

send_queue(p_h, p_i, j, message_length, owner_element_list.1)
send_queue(p_h, p_j, j, message_length, owner_element_list.2)

***SENDING MESSAGE PART

m : message which is received by Q_n^{send} ;
 m_s : message which is sent to Q_n^{recv} ;

```

WHILE()
  IF(  $m$  is received ) THEN
     $S_i \leftarrow mS_i$ ;
    IF( NoneHigherAhead( $m, S_i$ ) ) THEN
       $H = H \cup m$ 
    ENDIF;
    IF( LevelAhead( $(m, S_i)$ ) ) THEN
       $F = F \cup m$ 
    ENDIF;
  ENDIF;
  send( $m_s, Q_n^{\text{recv}}$ ),  $m_s \in H \cap F$ ;
  discard  $m_s$  from  $S_i$ ,  $H$ , and  $F$ ;
   $H \leftarrow \{ m' \mid \text{NoneHigherAhead}(m', S_i) \}$ ;
   $F \leftarrow \{ m' \mid \text{LevelAhead}(m', S_i) \}$ ;
ENDDO

```

***RECEIVING MESSAGE PART

m : message which is received by Q_n^{recv} ;
 m_s : message which is sent to *recv_queue*;
 v : message requested by *recv_queue*;

```

WHILE()
  IF(  $m$  is received ) THEN
     $S_j \leftarrow mS_j$ ;
    IF( NoneHigherAhead( $m, S_j$ ) ) THEN
       $H = H \cup m$ 
    ENDIF;
  ENDIF;
  IF(  $v$  is received ) THEN
     $S_v \leftarrow vS_v$ ;
    IF( NoneHigherAhead( $v, S_v$ ) ) THEN
       $H_v = H_v \cup v$ 
    ENDIF;
  ENDIF;
  IF( ( $H_v \neq \text{null}$ )  $\wedge$  ( $H \neq \text{null}$ ) ) THEN
    send( $m_s, \text{recv\_queue}$ ),  $m_s \in H \cap H_v$ ;
    discard  $m_s$  from  $S_j$ ,  $S_v$ ,  $H$ , and  $H_v$ ;
     $H \leftarrow \{ m' \mid \text{NoneHigherAhead}(m', S_j) \}$ ;
     $H_v \leftarrow \{ m' \mid \text{NoneHigherAhead}(m', S_v) \}$ ;
  ENDIF;
ENDDO

```

図 3: N レベルキューアルゴリズム

recv_queue(p'_i, p_h, j, message_length, nonowner_element_list.1)
recv_queue(p'_j, p_h, j, message_length, nonowner_element_list.2)

図2にレベルキューの構造を示す。\$Q_n\$は送信部\$Q_n^{send}\$と受信部\$Q_n^{recv}\$とからなり、1台のプロセッサに対して、\$Q_n^{send}\$と\$Q_n^{recv}\$のペアが割り当てられる。

\$N\$レベルキュー アルゴリズムを図3に示す。このアルゴリズムでは、2つのバッファ\$S_i, S_j\$を使用する。\$Q_n^{send}\$は、\$send_queue\$から受け取ったデータをバッファ\$S_i\$に格納し、\$Q_n^{recv}\$は他のプロセッサの\$Q_n^{send}\$から送られてきたデータをバッファ\$S_j\$に格納する。\$Q_n\$の状態を表すために、送信部と受信部それぞれの集合\$H\$と\$F\$を導入している。\$H\$は最上位レベルのデータの集まりであり、\$F\$は各レベルにおける最初のデータの集まりである。\$H\$と\$F\$は次のように定義される。

$$\begin{aligned} H &= \{ m \mid (S = S'mS'') \wedge (\forall m' \in S'' : m'.level \leq m.level) \} \\ F_u &= \{ m \mid (m.level = u) \wedge (ExtractSame(m, S) \stackrel{\Delta}{=} S'mS'') \wedge (\forall m' \in S'' : m'.level > m.level) \} \\ F &= \bigcup_{u \geq 0} F_u \end{aligned}$$

関数\$ExtractSame(m, S)\$は、データ間の到着順序を保ちながら、同じレベルのデータをバッファ\$S\$から取り出し、そのデータ列を関数の値として返す。関数\$NoneHigherAhead(m, S)\$は、\$(S = S'mS'') \wedge (\forall m' \in S'' : m'.level \leq m.level)\$が成り立つとき、真を関数の値として返す。関数\$LevelHead(m, S)\$は、\$(ExtractSame(m, S) \stackrel{\Delta}{=} S'mS'') \wedge (\forall m' \in S'' : m'.level > m.level)\$が成り立つとき、真を関数の値として返す。

\$Q_n^{send}\$は、最上位レベルでのデータ列の先頭要素を最優先で転送する。すなわち、\$H \cap F\$に属するデータ\$m\$は、もともと優先順位が高いので、\$m\$を\$S_i\$から取り出し、指定されるプロセッサの\$Q_n^{recv}\$へ転送する。\$Q_n^{recv}\$は複数のプロセッサからデータを受け取るために、あるレベルでは、到着順序で受け取る。

3 Nレベルキュー呼出しの挿入

\$N\$レベルキュー呼出しを次のように分類する。

ループ\$L\$の実行に必要なデータののために、ループ\$L\$内に挿入する\$N\$レベルキュー呼出しを**内側呼出し**、\$L\$のすぐ外側に挿入する呼出しを**外側呼出し**、それ以外の呼出しを**オーバーラップ呼出し**と呼ぶ。

オーバーラップ呼出し節点から、そのデータが参照される節点までの経路上に、他のデータ転送がなければ、**一重オーバーラップ**と呼び、ほかにデータ転送がある場合には、**多重オーバーラップ**と呼ぶ。

図1は多重オーバーラッププログラムの例である。\$s34\$で\$C(4:35)\$が参照される。\$s21\$から\$s34\$までの間で、このセクションは更新されることがないので、\$C(4:35)\$を転送する呼出しを\$s20\$の直後に挿入することができる。これはオーバーラップ呼出しである。\$B(4:53)\$において、ループ内にデータ依存関係がないので、このセクションをベクトル化し、\$s25\$の直前に挿入することができる。これは外側呼出しである。ループ\$s26\$には逆依存があるので、\$A(I, J+1)\$はベクトル化することができず、\$A(I, J+1)\$を転送するための呼出しは、\$s28\$の直前に挿入しなければならない。これは内側呼出しである。

呼出しの挿入位置は、プログラムの流れに沿って配列に対するアクセス状況を解析することによって決定できる。解析対象となるプログラムのフローグラフを考え、その各節点でのセクションアクセス情報を表すデータフロー方程式を示す。フローグラフの開始節点から、ある節点\$n\$への経路上で参照され、その後\$n\$までに代入が行われない配列要素によって構成されるセクションを\$n\$における**参照セクション**とよぶ。また、開始節点から\$n\$への経路上で代入が行われ、その後\$n\$までに参照されることがない配列要素によって構成されるセクションを、\$n\$における**無効セクション**とよぶ。

まず、プログラムのフローグラフについて、各節点における参照セクションと無効セクションの一般的な関係を示す。各節点\$n_i\$について、次の集合を定義する。

- \$IN_i\$: \$n_i\$の入口における参照セクションの集合
- \$OUT_i\$: \$n_i\$の出口における参照セクションの集合
- \$K_i\$: \$n_i\$の出口における無効セクションの集合
- \$Gen_i\$: \$n_i\$の実行によって、参照されるセクションの集合
- \$Kill_i\$: \$n_i\$の実行によって、無効になるセクションの集合

節点\$n_i\$についてのデータフロー方程式は、次に示すとおり、従来のデータフロー問題における方程式と同じ形の式で表すことができる[1]。方程式中の\$p\$は\$n_i\$の直接先行節を表す。データフロー方程式は次のとおりである。

$$IN_i = \bigcap_p OUT_p \quad (1)$$

$$K_i = (\bigcup_p K_p) \cup Kill_i \quad (2)$$

$$OUT_i = (IN_i - Kill_i) \cup Gen_i \quad (3)$$

ここでは、区間解析法をもとに、フローグラフの各節点\$n_i\$における参照セクションの集合を求めることを考える。区間解析法は、簡約フェーズと伝播フェーズの2つのフェーズからなるが、本稿では、簡約フェーズで得られる結果の利用だけで十分である。1つの区間は節点の集合からなり、各区間にはヘッダ節点とよばれる節点が必要1つ存在する。ループは1つの区間を構成する。その場合、ループの最後の文に当る節点を最終節点と呼ぶ。

簡約フェーズの各ステップでは、与えられたフローグラフを区間に分割し、各区間を要約節点とよぶ1つの節点で表す。そして、もとのグラフの節点間の辺を区間どうしの辺で表した高次のフローグラフを作成する。グラフ全体が単一の節点になるまでこのステップを繰り返す。簡約ステップでは、各区間について、そこに含まれる節点を前向きに(制御の流れに沿って)たどりながら、各節点\$n_i\$におけるローカルな無効セクション\$K'_i\$と参照セクション\$OUT'_i\$を求めていく。1つの区間についての簡約が終了したら、最終節点における\$K'_i\$と\$OUT'_i\$から、それぞれその区間の要約節点に対する\$Gen_i, Kill_i\$を設定する。

ループを構成する1つの区間\$I_h\$について、ローカルなセクションアクセス情報の求め方を示す。そのヘッダ節点\$n_h\$の出口では、参照セクションも無効セクションもないものとして、次の初期設定を行う。

$$K'_h = \phi, \quad OUT'_h = \phi$$

このあと、\$I_h\$内の各節点\$n_i\$を前向きに走査し、データフロー方程式(1)~(3)に従って、\$n_i\$での\$Gen_i\$と\$Kill_i\$とから、\$K'_i, IN'_i\$

および OUT'_i を表す式を生成する。その最終節点を n_i とすると、 K'_i と OUT'_i には、区間内でのアクセス情報を表す式が得られる。

OUT'_i と K'_i を、 I_h の要約節点 n_s の Gen_s 、 $Kill_s$ にそれぞれ与える際には、 I_h に対するループ制御変数を k とすると、セクション記述子の中で、 k によって表される添字の部分に k の制御パラメタでの表現 (範囲表現) に置き換える必要がある。この操作は、 k によって制御されるループの実行全体を通じての参照および無効セクションを求めることを意味する。そのために、次の関数 *iterations* を定義する。

関数 *iterations*($S, k, low : high : step$) は、与えられたデータ集合 S に対して、 S 中の各データセクションの添字 $\alpha \times k + \beta$ を次のように書き換えたデータセクションを返す関数とする。

$$\alpha \times low + \beta : \alpha \times high + \beta : \alpha \times step$$

区間 I_h の要約節点 n_s における Gen_s と $Kill_s$ は次の式で与えられる。ここで、そのループ制御変数を k とし、 k の制御パラメタを $low : high : step$ とする。

$$Gen_s = iterations(OUT'_i, k, low : high : step) -$$

$$(Union iterations(S_{def}, k, low : high : step)) \quad (4)$$

$$Kill_s = iterations(K'_i, k, low : high : step) \quad (5)$$

ここで、*def* は区間内の逆依存を表し、 S_{def} は逆依存の定義側における配列要素のデータセクションを表す。逆依存が存在する場合、それまでの繰返しで参照していた要素は、次の繰返しで値の更新が行われるため、その要素は無効になる。したがって、 Gen_s の計算においては、逆依存によって無効になる要素を除かなければならない。

呼出しの挿入位置を決めるとき、各区間で参照されるセクション Gen_s と無効になるセクション $Kill_s$ の情報が必要となる。これらのローカルな情報は簡約フェーズだけで求めることができる。

例 2 図 1 に示しているプログラムについて、各区間における Gen_s および $Kill_s$ を求める。 n_1, n_2, n_3 、および n_4 をループ s21, s25, s26, および s33 の要約節点とする。

ループ s21 において、 Gen_{s1} および $Kill_{s1}$ はそれぞれ ϕ 、 $B(1 : 100)$ となる。ループ s33 において、 Gen_{s4} および $Kill_{s4}$ はそれぞれ $C(4 : 53)$ 、 $D(1 : 50)$ となる。

節点 s28 では、 $Kill_{28}$ は $A(I, J)$ であり、 Gen_{28} は $A(I, J + 1)$ である。最終的に要約節点における Gen_{s3} は次のように得られる。(1) OUT_{28} を拡張して、 $A(I, 2 : 100)$ が得られる。(2) s28 によって、 $A(I, 1 : 99)$ が無効になる。(3) 逆依存による無効になった部分を $A(I, 2 : 100)$ から除いて、 Gen_{s3} が得られる。簡約の結果を表 1 に示す。

簡約フェーズで得られた結果に基づいて、 N レベルキュー呼出しの挿入位置を決定する規則を示す。

規則 1 (内側呼出し規則) Gen_i を節点 n_i での参照セクション、 Gen_s を n_i を含む要約節点の参照セクションとする。 $iterations(Gen_i) \in Gen_s$ が成り立たない場合、節点 n_i における N レベルキュー呼出しは内側呼出しとする。

規則 2 (外側呼出し規則) Gen_i を節点 n_i での参照セクション、 Gen_s を n_i を含む要約節点の参照セクションとする。 $iterations$

表 1: 簡約フェーズの例

Step	Set	Array Section
Elimination Step 1	OUT_{22}	ϕ
	K_{22}	$B(I)$
	Gen_{s1}	ϕ
	$Kill_{s1}$	$B(1 : 100)$
Elimination Step 2	OUT_{28}	$A(I, J + 1)$
	K_{28}	$A(I, J)$
	Gen_{s3}	$A(I, 100)$
	$Kill_{s3}$	$A(I, 1 : 99)$
Elimination Step 3	OUT_{29}	$A(I, 100)$
	K_{29}	$A(I, 1 : 99)$
	OUT_{30}	$A(I, 100), B(I + 3)$
	K_{30}	$A(I, 1 : 99), D(I)$
	Gen_{s2}	$A(1 : 100, 100), B(4 : 53)$
	$Kill_{s2}$	$A(1 : 100, 1 : 99), D(1 : 50)$
Elimination Step 4	OUT_{34}	$C(I + 3)$
	K_{34}	$D(I)$
	Gen_{s4}	$C(4 : 53)$
	$Kill_{s4}$	$D(1 : 50)$

(Gen_i) $\in Gen_s$ が成り立つ場合、節点 n_i における N レベルキュー呼出しは外側呼出しとする。

規則 3 (オーバーラップ呼出し規則) 要約節点 n_s を節点 n_i の先行節点とし、 $Kill_s$ を n_s における無効になるセクションの集合、 Gen_i を n_i における参照セクションとする。 n_i での転送が外側呼出しであって、かつ次の条件を満たす場合、 N レベルキュー呼出しはオーバーラップ呼出しとする。 $iterations(Gen_i) \in Kill_s$ 、および $iterations(Gen_i) \notin Kill_i$ (n_i は n_s から n_i までの経路上のすべての節点を指す)、オーバーラップ呼出しを n_s の出口に挿入する。

規則 4 (呼出しレベル規則) 同じ深さのループでは、同じレベルでデータを転送する。深さ n のループについて、最外側のループにおける転送レベルが i のとき、もっとも内側のループにおける転送レベルを $n + i$ とする。多重オーバーラップの場合、オーバーラップ呼出しはレベル j ($j < i$) で行われる。

4 評価

オーバーラップの効果を調べるために、AP1000 で、*slalom* (Light-Dispersed) および *hydro2d* (Navier-Stokes) の 2 つのプログラムの実行を試み、 Q_n によるオーバーラップ、PVM によるオーバーラップ、およびオーバーラップを行わない場合の実行時間を比較した。

N レベルキューアルゴリズムは、AP1000 専用ライブラリを使って実装されている。 N レベルキューを利用してデータ転送を行うものと、PVM の `pvm_sendsig()`、`pvm_nrecv()` など標準コールを利用してデータ転送を行うものとの 2 種類の SPMD コードを生

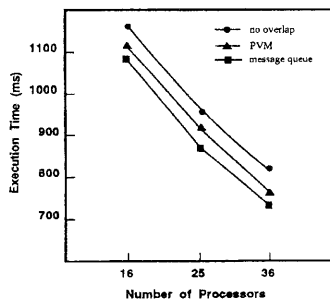


図 4: *slalom* におけるオーバーラップ効果の比較

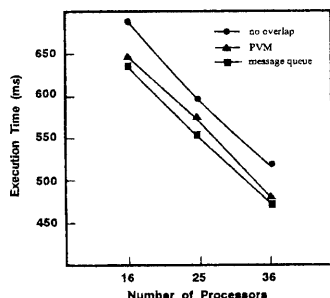


図 5: *hydro2d* におけるオーバーラップ効果の比較

成し、それぞれの実行時間を測定した。 Q_n と PVM の主な違いは、 Q_n で送信側が優先順位に従ってデータを送るのに対して、PVM は優先順位に従わずにデータを送ることである。一般に、多重オーバーラップの場合は、 Q_n が PVM よりも有利と考えられる。プロセッサ数をそれぞれ 16, 25, 36 として、 Q_n によるオーバーラップ、PVM によるオーバーラップ、およびオーバーラップしない場合の実行時間を図 4 と図 5 に示す。

分散メモリマシンでは、メッセージ通信によるプロセッサ間データ転送に時間がかかる。通信と計算をオーバーラップさせることによってプログラム全体の実行効率が改善できる。実験では、多重オーバーラップのために提案されている N レベルキューの有効性を示した。

5 結論

分散メモリ型のマシンでは、ハードウェアのピーク性能に対して、実際のプログラムで達成できる性能が低い、ということが指摘されている。その一因として、プロセッサ間でのデータ転送によるオーバーヘッドが並列処理に大きく影響することがあげられる。本稿では、通信と計算のオーバーラップ手法の 1 つ、 N レベルメッセージキューと呼ばれているオーバーラップ機構を提案した。続いて、オ

ーバーラップ可能なプログラムを用いて、本手法によるオーバーラップ、PVM によるオーバーラップ、およびオーバーラップしない場合とについて、実行を試みた結果を示した。これらを通じて、オーバーラップ最適化手法の有効性、および本稿で提案しているオーバーラップ機構の有効性が示された。

参考文献

- [1] A. V. Aho, R. Sethi, and J. D. Ullman: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
- [2] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," in *Proc. of 3rd Workshop on Compilers for Parallel Computer*, pp.121-160, July 1992.
- [3] *High Performance Fortran Forum. High Performance Fortran Language Specification*, Technical Report CRPC-TR9 2225, Rice University, January 1993.
- [4] M. Burke, "An Interval-based Approach to Exhaustive and Incremental Interprocedural Data-flow Analysis," *ACM Trans. Programming Languages and Systems*, Vol.12, No.3, (1990), pp.341-395.
- [5] S. Hiranandani, K. Kennedy, and C. Tseng, "Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines," in *Proc. of 1992 ACM Int. Conference on Supercomputing*, pp.1-14, July 1992.
- [6] V. Bouchitte, P. Boulet, A. Darte and Y. Robert, "Evaluating Array Expressions on Massively Parallel Machines with Communication/Computation Overlap," in *Proc. of 3rd Joint International Conference on Vector and Parallel Processing*, pp.713-724, September 1994.
- [7] X. Li and K. Harada, "An Efficient Asynchronous Data Transmission Mechanism for Data Parallel Languages," in *Proc. of 1996 IPSJ/IEEE International Conference on Parallel and Distributed Systems*, pp.238-245, IEEE Computer Society Press, June, 1996.
- [8] X. Li and K. Harada, "Evaluation of an Asynchronous Data Transmission Mechanism for High Performance Fortran Compilers," in *Proc. of IEEE Second International Conference on Algorithms and Architectures for Parallel Processing*, pp.430-437, IEEE Press, June, 1996.