ハイパフォーマンス
コンピューティング 67－16
（1997. 8. 20）

分散環境を対象とした並列プログラミング環境MC

首藤一幸　菅原健一　浜中征志郎　村岡洋一
compiler@muraoka.info.waseda.ac.jp
早稲田大学 理工学部
〒169 東京都 新宿区 大久保 3-4-1

あらまし　MCは、我々が開発している、逐次、並列計算機群を対象とした並列化コンパイラおよび実行時環境である。実現の前段階として、システムの基盤である中間表現ライブラリ、実行時環境などからプログラミング環境を構成した。本稿では、MCおよびその部分であるプログラミング環境の設計、実装について述べる。

キーワード　並列プログラミング, 並列化コンパイラ, 分散環境, 中間表現

Programming and Execution Environment for A Network of Machines: MC

Kazuyuki SHUDO　Kenichi SUGAHARA　Seishiro HAMANAKA
Yoichi MURAOKA
compiler@muraoka.info.waseda.ac.jp
School of Science and Engineering, Waseda University
3-4-1 Okubo, Shinjuku-ku, Tokyo, 169 Japan

Abstract　The MC system we are developping is a parallelizing compiler and a runtime system for a network of serial and parallel computers. We've constructed a programming environment from bases of the MC system, IR(intermediate representation) library, parser, code generator, dependence analyzer, preliminary version of parallelizer and runtime system. This paper describes the design of the MC system and the programming environment which is part of the system.

key words　Multiprogramming, Parallelizing Compilers, Distributed Environment, Intermediate Representations

1　はじめに

近年、ネットワーク接続をもたない計算機は考えられなくなった。ベクトル、並列計算機においてもそれは例外ではない。ハードウェアレベルでは、高性能計算機群の集合としての並列計算機が出現しているにも関わらず、その資源を活用するシステムソフトウェアはプリミティブなものしか存在しない。我々は、計算機群の有効利用を目的に、並列化コンパイラの提供というアプローチを採り、並列化、分散処理システムMCを開発している。

並列化、分散処理システムMCを実現する前段階として、並列プログラミング環境を構成した。本稿ではまず、目標とするシステム、現在のプログラミングの設計を述べる。続いて、コンパイラ研究の基盤である中間表現、実行時環境の設計、コンパイル手法、を述べる。

2　目標と現在

目標　目標は自動並列化コンパイラであり、現状はプログラミング環境である。

目標は次のような並列化コンパイラである。

- プログラムのタスク集合への分割、スケジュールを自動で行う
- タスク、データ双方の並列性を利用する
- 逐次、並列計算機群からなる非均質な環境を対象とする

現状　現状は次のようなプログラミング環境である。

- 利用者による Remote Procedure Call(RPC) プログラミング
- 関数を単位とした fork-join 型タスク並列処理—Nested Parallelism の利用
- メッセージ通信をコンパイラが解決、タイミングを最適化
- 逐次計算機群を対象とする

差分　今後は次に挙げる差分を埋めていく。

- タスクグラフベースの並列化
  現在はフローグラフに対してタイミングを最適化した通信文を挿入することで並列性を引き出しているところ、タスクグラフを用いたプログラム分割、スケジュール部を実装していく。

- そのためのサブプログラムの実行時間見積もり
- 並列計算機に割り当てられたタスク内でのループ並列処理

3 言語

利用者になるべくシンプルなプログラミングモデルを提供することを目的に、逐次型言語、具体的には Fortran で記述されたプログラムを入力としている。

存在するパーザは Fortran のもののみだが、システムの他の部分は中間表現を含め言語中立なので、パーザを用意することで他の逐次型言語への対応が可能である。

3.1 ディレクティブ

並列処理のためのヒントを利用者から取り入れるためのディレクティブをいくつかサポートしている。

文法は、目的に合う限り High Performance Fortran(HPF)[2] に従う方針である。パーザおよび中間表現は HPF のデータ分散ディレクティブに対応しているが、データ並列性を利用していない現在、その情報は利用されない。

また、HPF2.0 では Approved Extension として、タスク並列性を利用するための構文が用意されている。しかし HPF は非均質な環境への対応を目標としていないので HPF2.0 の構文は分散処理には適さない。よって MC プログラミング環境では独自ディレクティブを用意している。今後タスクグラフベースの並列化に移行する際は、自動並列化へのパスとしてタスクを構成するディレクティブを用意する。

3.2 拡張

RPC 並列プログラミングで fork-join 型並列処理を行うために Fortran を拡張した。

入力プログラム中のある関数呼び出しが RPC になる条件は次の 1.、2. 双方が満たされることである。

1. callee に **TASK_FUNCTION** ディレクティブが記述してあること。

   ```
   function 関数名 (引数並び)
   !HDS$ TASK_FUNCTION
   ```

   関数の頭に記述する。リモート呼び出しに適す、という表示。

2. 呼び出し文側に **PROCESS_ON** ディレクティブで呼び出し先計算機の指示があること。

   ```
   !HDS$ 関数名 PROCESS_ON 計算機名
   関数呼び出しを含む文
   ```

   関数呼び出しの前の行に記述する。関数の呼び出し先計算機を指示する。

入出力先資源の URL 指定　タスクをネットワーク上のどの計算機へ割り当てても、入出力先資源のプログラム中での表現は変えたくない。入出力先ファイルを URL で指定、HTTP の上で入出力を行う構文を追加した。

4 設計

4.1 システム構成

MC は次のソフトウェアから成る。

- プリプロセッサ
- コンパイラ
- 資源管理サーバ
- 実行時環境

実行時環境は C 言語、他は Java で作成している。

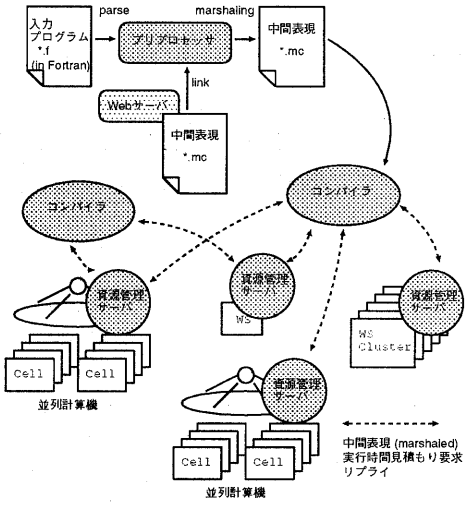

図 1: システム構成

また、周辺ツールとして次のものがある。

- 中間表現ビューア (図 2)
  開発中の中間表現エディタ。中間表現の構造、属性の可視化ツール。

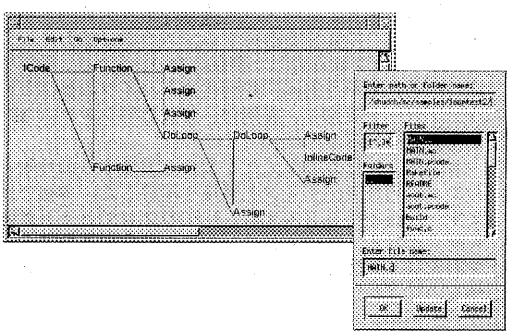

図 2: 中間表現ビューア

続いて、それぞれの仕事を述べる。

プリプロセッサ

Fortran の入力プログラムを中間表現に変換し、プログラム各部分の処理コストを算出。続いて、中間表現を構成する Java のオブジェクト群を整列 (marshaling)、ファイルに保存する。

処理コストは、式中の演算子数、ループの繰り返し回数に基づいて求める。

また、中間表現中の未解決関数群を中間表現としてリンクする。

資源管理サーバ

計算機ごとに起動しておくデーモンプロセス。その計算機に密接した仕事を受け持つ。仕事は次の通り。

- 中間表現から C のコードを生成、その計算機に応じた実行形式を作成。

並列化コンパイラに仕立てるために以下を検討している。

- 仮実行に基づくタスクの実行時間見積もり
- ループ並列処理
  並列計算機上の資源管理サーバは、割り当てられたタスクに対してループ並列化を試みる。また、実行時間の見積もり時にループ並列処理を見込んでおく。

コンパイラ

利用者から中間表現を受け取り、資源管理サーバと協力して各計算機上に実行形式を用意するまでが仕事。具体的には

- データ依存解析
- 資源管理サーバに中間表現を渡し、実行形式の生成を依頼

また、次を検討している。

- プログラムのタスク集合への分割、スケジュール

4.2 分散処理の問題

ネットワーク接続された計算機群で並列処理を行う際に問題となるのは、機種 (プロセッサ、OS)、処理能力、通信の遅延、帯域などの非均質性と、相対的に高遅延、低帯域のネットワークでの通信コストの高さである。

MC では、非均質性に対して、コンパイル環境、つまりコンパイラと資源管理サーバ群というアーキテクチャで対応する。

機種の非均質性 資源管理サーバが各計算機の実行形式を生成することで対応

処理能力の非均質性 コンパイラから資源管理サーバへのサブプログラム実行時間見積もり要求、その返答を利用したプログラム分割およびタスクのスケジュールで対応

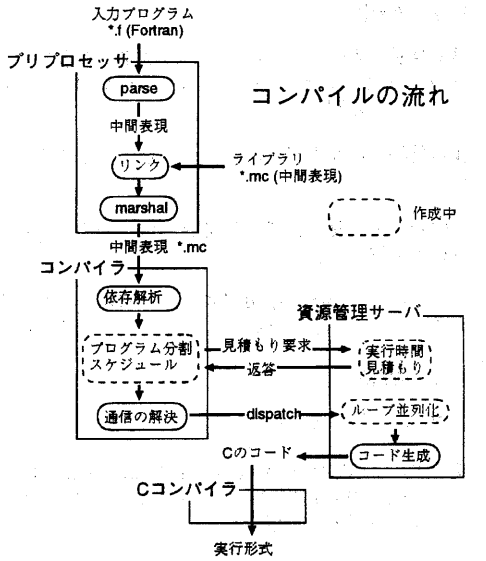

図 3: コンパイル処理の流れ

5 MC 中間表現

中間表現とは、入力プログラムのコンパイラ内部形式、またはその形式で表現されたプログラムを指す。具体的には、構造体とそれに対する操作関数群や、オブジェクト指向言語のクラスライブラリといった、構造とそれに対する操作から成る抽象データ型である。

MC は、一旦パーズしたプログラムは一貫して MC 中間表現の形で扱う。

5.1 構造

制御フローエッジに着目すると階層フローグラフ、先行制約関係に着目すると階層タスクグラフとなる。ループノードや条件分岐ノードが下位の層を持つ。

下位層を持つノードが、変数、定数、関数などシンボルのテーブルを持つ。シンボルの所属するノードが、そのシンボルのスコープとなる。式中の変数の出現は、テーブル中のシンボルへの参照を持つ。

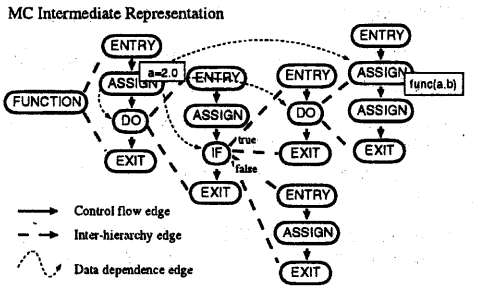

図 4: MC 中間表現

5.2 特徴

基本構造 Abstract Syntax Tree(AST) が基本。高レベルな表現のみ持つ。依存関係、通信などはノード間のエッジで表現する。

実装言語 中間表現、コンパイラ共に Java で実装している。

可読表現を排除 計算機内部表現に対応する可読表現は敢えて排した。

ソフトウェア部品の蓄積形式として利用 中間表現をソフトウェア部品の蓄積フォーマットとしている。

5.3 構造

選択肢は次の通り。

高レベル 伝統的な AST の他に Hierarchical Task Graph(HTG)[3] など。グラフの単位は、いわゆる高級言語の文程度である。

低レベル 四つ組や、SUIF の一部である low-SUIF の RISC assembly code[1] など。

bi-level 高レベル、低レベルな表現双方を持つアプローチ。SUIF、SCORE[5]、PROMIS などで採られている。あらゆる粒度の並列性の利用、または機械語レベルの最適化にまで対応することが目的。

選択 高レベル表現である AST を選択した。理由は次の通り。

- source-to-source translation を想定
- 機械語レベルの最適化を行う計画はない

機械語レベルの最適化は C コンパイラに任せる。

5.4 実装言語

以前は C を選択するプロジェクトが多かったが、最近は C++がよく選ばれている [4]。

選択 MC ではオブジェクト指向言語である Java を選択した。唯一の問題は、Java のプログラムがソフトウェアの Virtural Machine 上で実行されることからくる処理の遅さである。選択理由は次の通り。

- 実行時コンパイル、C への変換など、処理速度の改善手段がいくつかあること。
- Garbage Collector があり、ユーザによるメモリ管理が不要であること。
  中間表現をデーモンプロセスが扱うのでメモリリークが許されない。
- marshaling(Sun の言葉で serialization) で、作成、維持の人的コストが高い中間言語を排除できること。

5.5 可読表現の排除

本稿では、抽象的な構造または計算機内部表現を中間表現、可読なテキスト表現を中間言語と呼ぶ。

中間表現と中間言語の相互変換のために、中間言語ジェネレータ、パーザが必要である。例えば PROMIS プロジェクトでは、PROMIS 中間表現を核に、HTG の可読表現、SUIF の中間言語などとの相互変換のためのジェネレータ、パーザを用意することが計画されている。

選択 MC では中間言語を排した。

中間言語ジェネレータ、パーザの作成は非常に手間、時間がかかる上、作成後も、中間表現の仕様変更や拡張に追従していく必要がある。この作成、維持の人的コストが非常に高いため、可読表現を排した。

可読表現を用意することの理由は以下である。

- 中間表現の保存、ネットワーク経由の受渡し、周辺ツールからの利用が可能になる。
- 中間表現を可視化フォーマットとして利用。
- 手で操作しやすい。テキストの編集によって中間表現を操作できる。

中間表現の保存、受渡しは、marshaling で対応した。オブジェクト指向言語で実装された中間表現ライブラリでは、内部表現はオブジェクトのグラフである。Java では、オブジェクトを整列 (marshal) して、ファイルに保存したりストリームに流し込むことが可能である。C++では難しい。

可視化と中間表現の操作は

- MC プリプロセッサが疑似コードを生成
- 中間表現エディタを開発

することで対処している。

5.6 ソフトウェア部品の蓄積形式

整列、ファイルに保存した中間表現を、ソフトウェア部品として利用している。

中間表現を保存しておいて後でリンクして再利用することが可能である。この仕組みでライブラリを構成できる。実際に、組み込み関数、手続きは、中間表現の形で保存、ライブラリ化してある。

関数、手続きが一旦機械語になってしまうとアーキテクチャ中立ではなくなるし、インライン展開も並列化もできない。再利用の方法が非常に限られてしまう。MC では、関数、手続きは中間表現の形式で保存してあるので、過去に記述、保存された組み込み関数をインライン展開したり並列化したりすることが可能である。

レポジトリ 中間表現レポジトリの開発を計画している。現在は、プリプロセッサで中間表現のリンクを行う際に、中間表現の場所を URL で指定できる。リンク作業の例を示す。

```
% mcc -o aout.mc main.mc \\
        http://foo.ac.jp/irlib/dgemm.mc
```

6 実行時環境

実行時環境はCで記述されている。中心はRPCライブラリである。既存のものは利用せず、並列処理向けに通信タイミングの制約が緩いものを実装した。通常のRPCと比較して、次の並列処理向けの特徴を持つ。

- callerは引数を任意の順で送信できる。
- callerは任意のタイミングで返り値を要求できる。

実行時環境は次から成る。

- MC RPCライブラリ

機能	使用者
リモートプロセス起動	caller
引数の送信	caller
返り値の送信登録	callee
返り値の要求、受信	caller
引数受信フラグの制御	caller

- OSプロセスの起動要求を受けるデーモン
- データ表現整合ライブラリ

異機種間でデータ表現の整合を取るためのデータ表現形式の標準として、Sun RPCのExternal Data Representation(XDR)や、OSIのBasic Encoding Rules(BER) for Abstract Syntax Notation One(ASN.1)などがある。MCではデータ表現の変換にかかる時間を抑えるために、次の方針でライブラリを用意した。

- 数値はネットワークバイトオーダで表現
- 浮動小数点数はIEEE754形式を仮定

7 コンパイル処理

コンパイラは、入力プログラム中のリモート関数呼び出しに対して

- calleeを独立したプログラムに
- 関数呼び出しを
  - リモートのプロセス起動、終了
  - 引数および返り値の送受信

に展開し、通信のタイミングを最適化する。

7.1 calleeに対する処理

calleeには、引数の受信待ちなどを含むwrapperが用意され、独立したプログラムとなる。実行時には独立したOSプロセスとなる。

コード生成後のwrapperは次のような関数となる。

```
int main(int argc, char **argv) {
  返り値の送信登録
  引数の受信登録
  while (True) {
    登録された引数の受信待ち(ブロックされる)
    callee 本体の呼び出し
  }
}
```

引数を受信すべき順に制約はない。受信した順に、受信フラグが立つ。引数がそろうと、callee本体が呼び出される。

引数受信待ちの中でcallerからの次の要求も受け付ける。

- 返り値の要求
- 引数受信フラグのマスク、アンマスク要求

7.2 callerに対する処理

通常のRPCとは異なり、コンパイラが通信タイミングを最適化して、並列性が引き出される。通信タイミングは、データ依存解析で得られた依存関係を考慮して決定される。基本方針は、実引数の値が決定したらなるべく早いタイミングで送信することである。

引数の送信 caller側から引数を送信するタイミングを決定するアルゴリズムを示す。

準備として中間表現に関する用語を定義する。

下方	ツリーに見立てた中間表現の葉の方向
上方	根の方向
上方ノード	階層間エッジを上方に辿って到達し得るノードすべて
共通上方ノード	ノード群すべての上方ノードでかつ最も深いノード

前処理(単純化)

- ▷ 実引数が式である場合、変数に変換。
  (`func(a+b);` ⇒ `tmp=a+b; func(tmp);`)
- ▷ 呼び出しが式の一部であった場合、式の右辺が呼び出しのみになるように変換。
  (`a=func(b)+c;` ⇒ `tmp=func(b); a=tmp+c;`)

送信タイミング群(ノード集合S_{send})取得

- ▷ 関数呼び出しを含むノードをN_{sink}とする。
- ▷ 以下の処理を引数中の変数すべてに対して行う。(変数V)
  - ▷ Vを依存先変数とするデータ依存エッジ(E)を取得。
  - ▷ Eの依存元ノード集合を得て、以下の処理を含まれるノードすべてに対して行う。(ノードN_{src})
    - ▷ 送信タイミング決定procedureを呼ぶ

送信タイミング決定procedure

- ▷ N_{src}とN_{sink}の共通上方ノード(N_{common})を得る。
- ▷ N_{common}の一階層下方でかつN_{src}の上方ノードをS_{send}に加える。

得られたノード集合S_{send}に含まれるノードの直後に、引数送信ノード(文)を挿入する。

仮引数受信フラグのマスク 引数がそろい次第calleeのfunction本体が呼び出されるというcalleeプロセスの実行制御方式のため、上のように引数を解決しただけではループ中のリモート関数呼び出しがkickされない可能性がある。そこで、必要に応じてcaller側からcalleeプロセス中の引数受信フラグにマスクをかける。マスクをかけられた仮引数は、実際には受信されずとも受信されたものと扱われる。

マスク操作文をcallerのどこに挿入すべきかを決定するアルゴリズムを示す。

マスク操作文挿入タイミング (ノード N_{mask}) の決定

- ▷ S_{send}に含まれるノードすべてに対して、N_{sink}との共通上方ノードを求め、最も下方にあるものを選び $N_{deepestcommon}$とする。
- ▷ N_{sink}の上方ノードでかつ $N_{deepestcommon}$より下方にあるループノードの中で、最も下方にあるものを N_{mask}とする。

得られた N_{mask}の直前にマスクをかけるノード (文)、直後にマスクをはずすノードを挿入する。

コンパイル例　caller に対して通信タイミングの最適化を施した例を挙げる。

入力プログラム

```
      flag = 0
      a = 2.0
      d = 5.0

      do i = 1, n
         b = 4.0
         if (flag.eq.0) then
            d = 6.0
            flag = 1
         else
            flag = 0
         endif

         do j = 1, n
!HDS$ func PROCESS_ON cafe
            ret = func(a,b,d)
            b = 5.0
            d = 7.0
         enddo
      enddo
      end
```

コンパイル、コード生成結果

```
(_tmp_1 = mc_popen("cafe", "func"));
      /* リモートでプロセスを起動 */
      /* プロセスハンドル: _tmp_1 */
(flag = 0);
(a = 2.0);
mc_send_param(_tmp_1, "a", 2052, (&(a)), 1);
      /* 変数 a の値を送信 */
(d = 5.0);
mc_send_param(_tmp_1, "c", 2052, (&(d)), 1);

mc_mask_param(_tmp_1, "a", 1);
      /* callee 中の、仮引数 a の受信フラグをマスク */

for (i = 1; i <= 5; i += 1) {
  (b = 4.0);
  mc_send_param(_tmp_1, "b", 2052, (&(b)), 1);
  if ((flag == 0)) {
    (d = 6.0);
    (flag = 1);
  }
  else {
    (flag = 0);
  }
  mc_send_param(_tmp_1, "c", 2052, (&(d)), 1);
  for (j = 1; j <= 5; j += 1) {
    double _tmp_0;

    mc_recv_ret(_tmp_1, "func", 2052, (&(_tmp_0)), 1);
      /* 返り値を要求、受信 */
    (ret = _tmp_0);

    (b = 5.0);
    mc_send_param(_tmp_1, "b", 2052, (&(b)), 1);
    (d = 7.0);
    mc_send_param(_tmp_1, "c", 2052, (&(d)), 1);
  }
}

mc_mask_param(_tmp_1, "a", 0);
      /* 仮引数 a の受信フラグをアンマスク */

mc_pclose(_tmp_1);
      /* リモートプロセスの後始末 */
```

8　まとめ

ネットワーク接続された計算機群をシンプルなプログラミングモデルで利用することを目的とした並列化分散システム MC、目標へのパスである現在の並列プログラミング環境の設計を述べた。また、研究基盤である中間表現の設計における選択、実行時環境の概要、通信タイミングの最適化手法を述べた。

今後は以下を計画している。

- 現在フローグラフをベースに通信タイミングの最適化を行っているところ、タスクグラフベースを用いた並列化を実装していく。
- プログラムの分割、スケジュールアルゴリズムの検討。
  非均質な環境が対象なので、従来通り、分割終了後にスケジュールを行う手法では限界がある。分割とスケジュールの統合アルゴリズムを検討している。
- 並列計算機に割り当てられたタスク内でのループ並列処理

他には、アプリケーションによる評価が目下の課題である。

参考文献

[1] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN '93 Conf. on Prog. Lang. Design and Implementation*, pp. 126–138, June 1993.

[2] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*, January 1997.

[3] M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Parallel and Distributed Syst.*, Vol. 3, No. 2, pp. 166–178, March 1992.

[4] Kathryn S. McKinley, J. Eliot B. Moss, Sharad K. Singhai, Glen E. Weaver, and Charles C. Weems. Compiling for heterogeneous systems: A survey and an approach. Technical Report CMPSCI Technical Report 95-82, University of Massachusetts, November 1995.

[5] Glen E. Weaver, Kathryn S. McKinley, and Charles C. Weems. Score: A compiler representation for heterogeneous systems. In *Proceedings of the 1996 Heterogeneous Computing Workshop*, April 1996.