マルチメディア通信と分散処理 58−24
(1992. 11. 20)

異機種間分散共有メモリのためのコンパイラシステム

川合史朗 相田仁 齊藤忠夫

東京大学 工学部

共有メモリを分散環境でも実現することにより，プログラミングがしやすい，密結合システムとソフトウェアの互換性がとれる，既存のソフトウェア資産が利用できるといった多くの利点が得られる。

しかし，機種の異なるプロセッサを接続した異機種環境においては，既にそのような環境は一般的になっているにもかかわらず，機種間のデータ表現形式やメモリ管理方式の違いがあるため，分散共有メモリシステムを実現した例は少ない。

本稿では，C言語の仕様を拡張し，機種間のデータ表現形式の違いをデータ型の違いと捉えることで，ポインタ，構造体などの派生型を共有することを容易にしたコンパイラシステムを紹介する。

A Compiler System for Heterogeneous Distributed Shared Memory

Shiro Kawai, Hitoshi Aida and Tadao Saito

Faculty of Engineering, The University of Tokyo

To allow memory sharing among distributed processors brings lots of benefits: such as easiness to program, or reusability of existing software modules.

However, implementation expamples of heterogeneous distributed shared memory systems are scarce, although such environment has already been popular. The fact that different processor may take different internal data representation makes problem difficult.

In this paper, we describe a compiler system which extends C language and supports different data representation as variation of data types. It allows processes to share pointers, structures and other various constructive data in a way identical to the homogeneous shared memory system.

1 はじめに

高速，高性能な CPU が様々なベンダから比較的安価に手にはいるようになったため，マルチベンダ環境で分散処理を行ないたいという要求が高まってきている。既にネットワーク環境では，TCP/IP などの標準プロトコルの上に構築された RPC，XDR などのライブラリを用いて，マルチベンダの分散処理を容易に構築することができるようになった。

だが現段階では，異機種間分散処理アプリケーションを開発しようとするプログラマは何らかの形でネットワークインタフェースを意識しなければならない。初めからメッセージパッシングパラダイムに基づいた設計を行なっているのならばともかく，既存のソフトウェアモジュールを利用したり，動作中のシステムにプロセッサを追加することで性能向上をはかるといったことは容易ではない。

これに対し，分散環境でも共有メモリを実現し，共有メモリセマンティクスに基づいたプログラミングを可能にしようという試みがなされるようになってきた。既に均質な分散環境では多くのシステムが製作されている[3]。しかし，異機種環境での分散共有メモリの実現例は，機種間でのデータ表現の違いなどを吸収する必要があるため，実際には少ない。

本研究では，既存のハードウェア /OS を用い，ユーザレベルで異機種間分散共有メモリシステムを実現するというケースを想定し，プログラマからは機種間の違いを隠蔽して密結合共有メモリシステムと同じ様なコーディングを可能にし，かつできるだけ実行効率を落とさないということを目標にしたコンパイラシステムの開発を試みている。本稿では，2節で異機種間共有メモリシステムの実現にあたっての問題点および関連研究に触れた後，3節で言語仕様およびコンパイラシステムに加えた拡張について説明し，4節でシステムの実装について述べる。

2 実現にあたっての問題点と方策

2.1 データ表現形式の違い

現在，キャラクタ型が 8bit でないといったような計算機は特殊な部類に入るだろうが，少なくとも次のような点では，表現形式の異なるプロセッサ / システムが混在して使用されている。

a. endian (byte-endian, bit-endian)

b. データの alignment

c. 浮動小数点形式

d. 文字コード体系

これらのデータ表現形式の異なるプロセッサを混在して用いるには，同機種間の共有メモリシステムのようにメモリのビットイメージをそのまま共有することはできず，何らかの方法で表現形式を変換する必要がある。

変換の方法としては，ライブラリコールによるもの，実行コードに変換を行なうコードを直接埋め込むもの，ハードウェアによるものなどが考えられる。いずれにせよ，実用的なシステムにするためには，プログラマからはそれらのメカニズムが見えないようにしなければならない。

2.2 表現形式の違いの隠蔽

プログラマから表現形式変換の詳細を隠蔽する方法としては，

- 共有メモリを抽象化し，プログラマにはオブジェクトの共有という形に見せる。共有オブジェクトへのアクセスは，専用のプリミティブを通して行なうものとし，アクセスの際に変換を行なうようにする。
- コンパイラやプリプロセッサによって共有メモリへのアクセスを拾いだし，変換コードを挿入したり，変換ルーチンへのコール命令を挿入したりする。
- ハードウェアがデータ型を検出して変換を行なう。

といったもの，あるいはこれらを組合わせた方法が考えられ，また実現されている。

抽象化されたデータオブジェクトの共有という形をとり，アクセスの際，あるいはブロードキャストの際に表現形式の変換を行なっているシステムとしては，Linda[1] の異機種対応版 (POSYBL[5] 等) や Agora[2] 等がある。これらのシステムでは，コンパイラシステム (プリプロセッサ) の拡張も併用している。

ハードウェアによるデータ型の変換を行なうシステムについては，Strevell らが検討している[6]。その場合でも，データをどのように変換すべきかという情報を得るために，コンパイラやリンカのサポートが必要となる。

3 コンパイラシステムの拡張

3.1 目標

ここでは，特に既存のハードウェア /OS を用い，ユーザレベルで異機種間分散共有メモリシステムを実現するというケースを想定する。この場合，次のような点が望まれる。

- プログラマが，密結合共有メモリシステムと同じような感覚でコーディングできるようにする，あるいは，既存のソフトウェアモジュールを少ない労力で利用しやすくすること。
- カーネルの加工は避け，ユーザレベルで実現可能とすること

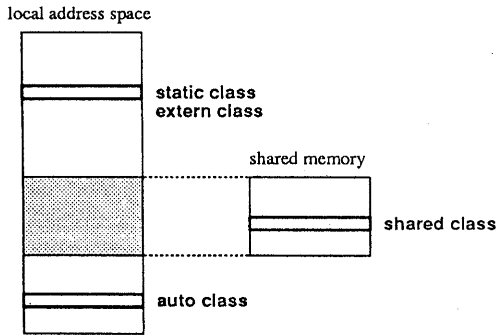

図 1: 記憶クラス shared

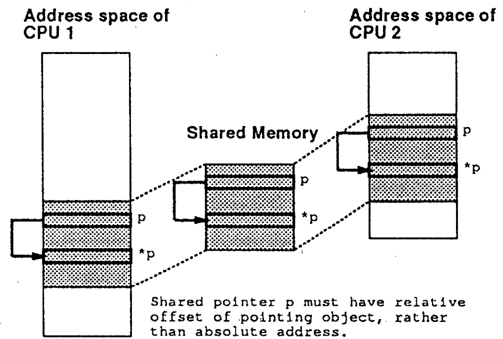

図 2: 共有メモリに置かれたポインタ

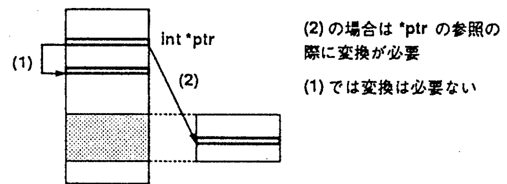

図 3: 2 種類のローカルポインタ

- 実行オブジェクトの効率が良いこと

そこで我々は，広く用いられているC言語をとりあげ，そのデータ型の概念を拡張することにより，専用のアクセス関数を用いずに，プログラマからの見かけ上，データを同機種間の共有メモリシステムとほぼ同じように共有することを可能にするコンパイラシステムを設計した。ポインタや構造体，共用体等の多様な派生型もかなり自由に共有できる。

さらに，コンパイラレベルでデータ表現形式の差異をサポートすることにより，プロセッサが内蔵の表現形式変換インストラクションを持っている場合はそれを直接使用することができ，ライブラリコールによる変換に比べて高いパフォーマンスが期待できる。

なお，ベースとなる分散共有メモリシステムの実装方式については，ハードウェアによる複製型共有メモリシステム[4]を念頭に置いているが，必ずしもそれに限定されるものではない。ビットイメージでメモリを共有できるものならば，ソフトウェアによる分散共有メモリシステムをベースに用いることも可能である。

3.2 記憶クラスの拡張

共有する変数は，密結合プロセッサシステムでしばしば行なわれているように，新たな記憶クラス shared を用いることによって指定する。共有変数は共有メモリ上ではシステム間での共通表現形式で格納され，アクセスの際にプロセッサのローカル表現形式に変換される (図 1)。

ポインタを共有メモリに置く場合，プロセッサ内での絶対アドレスはプロセッサ間では意味を持たないので，共有メモリ内での相対アドレスに変換した上で endian 変換等を行なう必要がある (図 2)。なお，共有メモリに置かれたポインタを以後共有ポインタと呼ぶ。

また，共有変数の alignment の制約は最も厳しいプロセッサに合わせられる。このため，同じ定義の構造体でも，メンバのオフセットは共有メモリに置かれた場合とローカルメモリに置かれた場合とで異なることが有り得る。

3.3 表現形式指定子

さらに，同機種間でのメモリ共有の場合と大きく異なる問題として，ポインタを用いた間接参照の場合がある。

共有ポインタは共有変数しか指すことはできない (ローカル変数を指しても意味がない) ので問題はないが，ローカルメモリにおかれたポインタはローカルメモリ上のオブジェクトを指す場合と共有メモリ上のオブジェクトを指す場合があり，ソースコードの表記からは区別ができない。しかし，そのポインタを通して参照されるオブジェクトを評価する際には，そのオブジェクトがローカルな表現形式なのか共通表現形式なのかがわからなければならない (図 3)

対応策としては，

- 実行時にポインタが指すアドレスの範囲を検査して表現形式の変換が必要かどうかを決めるコードを挿入する
- ポインタの指す先のオブジェクトの表現形式を示すキーワードを言語に導入する

といった方法が考えられる。前者の方法は同機種間共有メモリシステムで使われていたソースコードをそのまま

Notation	Memory representation
`standard int i = 0x12345678;`	i 78563412
`standard int *p = &i;`	p → *local address of i* → i 78563412
`standard int *standard q = &i`	q → *relative address of i in shared memory* → i 78563412

(It is assumed that standard integer type is little endian.)

図 4: 表現形式指定子の使用例

利用可能であるという利点があるが，実行時のオーバヘッドが非常に大きくなってしまう。

後者の方法では，コンパイル時にポインタの指す先のオブジェクトの表現形式を知ることができるので，表現形式変換のインストラクションを効率良く挿入することができる。本システムでは実行時のパフォーマンスを重視し，この方式をとった。

表現形式指定子 (representation specifier)`standard` を新たに導入する。すべての変数は従来の型の他に表現形式の属性を持ち，standard 型とローカル型に分類される。ローカル型の変数はプロセッサ内部での形式で表され，standard と指定された変数はプロセッサ間での標準形式で表される。文法上は type qualifier とほとんど同じように用いることができる (図 4)。

この表記を厳密に適用すると，共有ポインタは図 4の 3番目の例に見られるように非常に複雑な表記になってしまうため，表現形式があきらかな所 (shared 変数の宣言等) では `standard` は省略可能とする。そうすると結局，キーワード `standard` が必要になるのは，ローカルポインタが共有変数を指す場合 (図 4の 2番目の例) だけになり，プログラミング上の負担はさほど大きくないと考えられる。

3.4 変換

表現形式の差異をデータ型の違いの一種と捉えることにより，データ表現形式の変換は通常の型変換の拡張と考えることができる。

算術変換

通常の算術変換の前に，被演算数が standard 型であった場合，同型のローカル表現への変換が行なわれる (図 5)。

キャストによって，この暗黙の型変換を抑制することができる。次の例では，`i` は変換を受けずに printf に渡される。

```
shared int i;
printf("%d\n", (standard)i);
```

これは，standard 型の表現を調べる場合などに便利であるが，standard 型とローカル型とでサイズ等が異なる可能性もあるので注意が必要である。

構造体 / 共用体

構造体は，同じ構成を持つ別の表現形式の構造体へ implicit に変換可能である。つまり，alignment の制約の違いによりローカル形式と standard 形式とで構造体のメンバへのオフセットが異なっていても，プログラマは気にせずに同じタグをもつ構造体へ代入することができる。コンパイラは表現形式の違いを検出し，メンバごとに変換・代入を行なうコードに展開する。

共用体はメンバが指定されればそれに従って表現形式を変換するが，共用体同士の代入でメンバが指定されない場合は，最初のメンバを仮定する。

ポインタ

表現形式が異なるオブジェクトを指すポインタは，原則として変換不可能である。つまり，“pointer to standard int” 型をもつポインタを “pointer to int” 型のポ

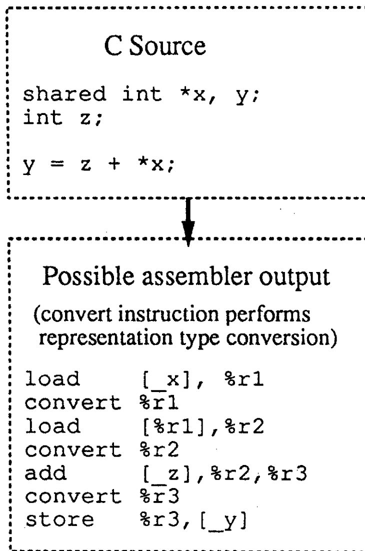

図 5: 暗黙の算術変換

インタに代入したり，相互に演算したりすることはできない。(これらのポインタが指す先のオブジェクト同士は変換可能であることに注意が必要である。 さらに，“pointer to standard int” 型と “standard pointer to standard int” 型は変換可能である。図 6参照。)

ただし，特例としてポインタが指し示すオブジェクトの表現形式が standard とローカルで一致する場合は，standard からローカルへの変換に限って可能である。アラインメントの制約は，ローカル型の方が緩いので (ある型 T に対して，alignof(T) $\leq$ alignof(standard T) が常に成り立つ)，この方向の変換は制約に違反することはない。

このことにより，ソースコードが完全にシステム独立にならない可能性がある。しかし，例えばローカルと standard で文字列の表現が同じ場合に，文字列へのポインタを統一して扱うことができるというメリットがある。(そしておそらく多くのプロセッサでこの仮定は成り立ち，プログラマもこの仮定が成り立っていることを期待しているであろう)

```
shared int *sp;          /* sp is a standard pointer to
                            standard int */
int *lp;                 /* lp is a local pointer to
                            local int */
standard int *lsp;       /* lsp is a local pointer to
                            standard int */

lp = sp;                 /* illegal: type mismatch */

lsp = sp;                /* legal: implicit conversion */

*lp = *sp;               /* legal: standard int is converted
                            and stored as local int */
*lsp = *sp;              /* legal: type matches */
```

図 6: ポインタの型変換

この逆に，ローカル型のオブジェクトを指すポインタの内容を standard 型のオブジェクトを指すポインタへ代入することは，共有ポインタがローカルメモリを指すようになることから，禁止される。

```
char localstr[] =
       "This is a local string.";
shared char sharedstr[] =
       "This is a shared string.";

char *lptr;
standard char *lsptr;
shared char *sptr;

lptr = localstr;           /* 通常の代入    */
lptr = sharedstr;          /* 暗黙の型変換 */

sptr = localstr;           /* 禁止 */
sptr = sharedstr;          /* 通常の代入 */

lsptr = localstr;          /* 禁止 */
lsptr = sharedstr;         /* 通常の代入 */
```

関数にポインタを渡す場合，その関数のプロトタイプ宣言がなければ，関数はローカル表現形式へのポインタをとるものと仮定される。したがって，次のようなコード int の表現が standard とローカルで同じでない限り無効である。

```
int func();

...
{
   standard int *i;

   func(i);
}
```

func が次のように宣言されていればよい。

```
int func(standard int *);
```

3.5 同期機構と最適化

共有変数参照の最適化

CPU が表現形式の変換用のインストラクションを備えていたとしても，パフォーマンスの点からは少しでも不要な変換を減らす方がよい。また，共有メモリからのフェッチは通常のメモリアクセスより遅いということも十分ありうる。したがって，共有変数の参照・代入に関して最適化を行なうことは重要であると考えられる。

ところが，一般に共有変数は volatile であり，optimizer がメモリ参照を勝手に削ってしまったり，コード

```
shared int X;

{
  while ()
    switch (X) {
      case A:  actionA()
      case B:  actionB();
      ...
    }
}
```

(a) Monitoring shared variable and do action. Reference of X must not be moved.

```
shared int X;

{
  int array[], *p;

  <lock shared memory>
  for (p in array)
    *p *= X;
  <unlock shared memory>
}
```

(b) Use shared variable X. Reference and representation type conversion of X may be moved to outside of for.

図 7: 共有変数参照・変換に関する最適化

を移動したりすることはできない。たとえば図 7(a) では，共有変数 X はループのたびに毎回評価と変換を行なわなければならない。

しかし，排他制御が行なわれていると，ある程度の最適化が可能になる場合がある。図 7(b) のような場合は，ロックされている範囲内で共有変数の参照や代入のコードを動かすことができる。

本システムではこの点に着目し，通常の並列計算機用 C 言語ではライブラリコールとして実装されている同期プリミティブを構文要素に取り込むことにより最適化を試みている。

同期構文要素

キーワード **acquire** により，排他制御を変数単位で行なう (図 8)。実際の排他制御は，メモリ共有機構の実現方式により，その変数を含むページ単位，あるいはいくつかの変数をまとめたグループに対して行なわれる。

acquire 構文は，つぎのようなインストラクションに落とされる。

1. acquire 宣言の箇所の前に，共有メモリに対するアクセス権獲得のリクエスト (通常，特定のアドレスへの書き込み) のインストラクションが挿入される。
2. acquire 宣言自体は，アクセス権を獲得できるまで待つインストラクションに 展開される。
3. 構文ブロックの終りで，アクセス権を放棄するインストラクションが 挿入される。

```
shared X;

{
  acquire X;

  <code with X>
  ...

}
```

A page of shared memory which holds value of X is locked between acquire declaration and the end of that block.

図 8: 同期構文要素

これらのインストラクションは，さらに次のようにして最適化の対象になる。

- アクセス権獲得リクエストは，制御フローが狂わない限り，早めに出しておくことができる。(リクエストと獲得待ちの間に通常のメモリアクセスや演算のインストラクションを挿入することができる)。アクセス権の要求／獲得はローカルメモリの参照などに比べて時間のかかる処理であるため，このような最適化は有効であると考えられる。

  ただし，アクセス権獲得要求の頻度等によって最適なスケジューリングは大きく変化することが予想される。これはコンパイル時には予測不可能なため，どの程度リクエストを前に出すかについては実際のアプリケーションによる性能評価を行なって経験的に決めるしかない。

- アクセス権が獲得された範囲では，ローカルなデータの依存関係を保っている限りは，共有変数のアクセスインストラクションを自由に移動することができる。
- アクセス権が獲得された範囲内で複数回の共有変数参照／代入がある場合は，それぞれ 1 回の参照／代入にまとめることができる。当然，データの表現形式の変換も 1 回だけ行なえば良い。
- アクセス権の放棄は，ブロックの終りまで待たないでもブロック内での最後の共有変数へのアクセスの直後に行なうことができる。

4 実装

以上述べてきた仕様に基づき，現在プロトタイプを試作中である。

実装に当たっては，次の点に留意した。

- 最適なシステム間の標準形式はアプリケーションを走らせる環境によって異なる。ソースに手を加えないでも，CPU の構成によって適当な標準形式を採用できるようになっていることが望ましい。

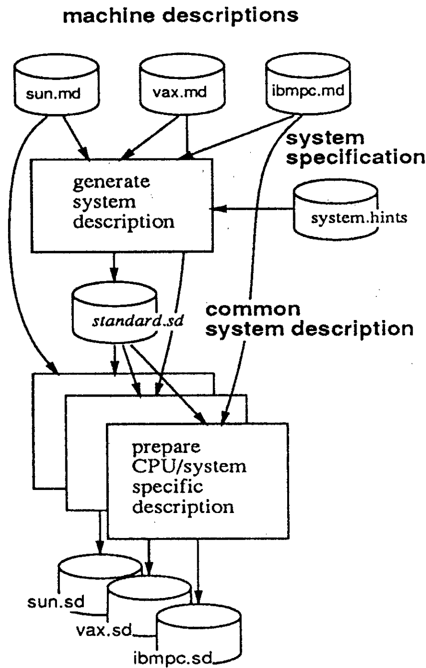

図 9: システム記述ファイルの生成

- 密結合マルチプロセッサシステムの場合と異なり，コンパイラ／ローダは各々の CPU で独立して走る。共有変数の配置を矛盾なく決定する方法を用意してやらなければならない。
- ローダはターゲット CPU のメモリ管理方式に適した戦略をとる必要がある。

システム記述ファイル

システム間の標準形式など，特定のシステム構成に依存する情報はソースに埋め込むのではなく，システム記述ファイルという別ファイルに用意する。コンパイラ，ローダはこのファイルを参照し，適当なコードを生成するようにする。システム構成が変わった場合でも，システム記述ファイルを差し替えるだけでよい。

実際には，システム記述ファイルは，各 CPU ごとに特性を記述したマシン記述ファイルとシステム設計者が与えるシステム構成定義ファイルをもとに自動生成される。さらに，処理速度の向上のため，システム記述ファイルとマシン記述ファイルをもとに，各 CPU ごとのシステム記述ファイルが生成される (図 9)。この操作は複雑だが，システム構成が決まったら一回行なうだけで良い。

コンパイルの流れ

コンパイラはシステム記述ファイルを参照し，表現形式の変換コードなどを挿入したリロケータブルオブジェクトを生成する。その際，共有変数の定義に関する情報を共通のファイル (共有変数参照テーブル) に書き出しておく。ローダはそのテーブルを取り込んで，共有変数の配置を決定する。その様子を図 10に示す。

なお，設計開始段階では，拡張された文法の処理をプリプロセッサに行なわせ，コンパイラはベンダ提供のものを用いるという方針であった。プリプロセッサを用いれば，新たな機種への移植が容易になり，また特殊なコンパイルテクニックを要求するプロセッサにも対応できるという利点があるが，プロセッサが表現形式変換インストラクションを持っていても利用できない等，実行効率に問題がある。さらに，前節で述べた仕様を満足するにはプリプロセッサ内でコンパイラ本体と同程度の構文解析が要求され，コンパイル時にプリプロセッサを通すオーバヘッドがかなり大きくなってしまうという問題点もあった。

しかし，新たな機種への対応を考えた場合，コンパイラ本体を移植するのは大きな負担である。関数のインライン展開とインラインアセンブラの表記法が標準でベンダ提供のコンパイラにサポートされれば，プリプロセッサによるものでも実行時の効率を落さないで済むことと思われる。

5 まとめ

データの内部表現形式が異なる機種間で，ビットイメージによるメモリの共有を可能にするコンパイラシステムの仕様について述べた。

本システムの有効性を確認するためには，実用的な規模のアプリケーションによる性能評価が不可欠である。現在，ハードウェアプラットフォームを含め評価システムを準備中であり，性能評価結果については追って報告できるものと思う。

今後は，実際に分散アプリケーションを開発しての性能評価や，他言語 (C++ 等) への拡張を予定している。

参考文献

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *IEEE Computer*, 19(8):26–34, Aug. 1986.

[2] R. Bisiani and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Trans. Comput.*, 37(8):930–945, Aug. 1988.

[3] B. Nitzverg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8):52–60, Aug. 1991.

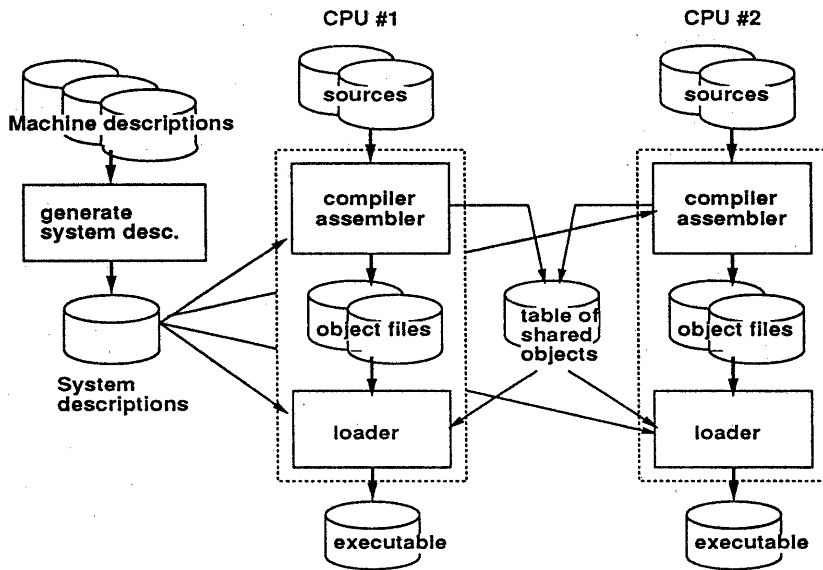

図 10: コンパイルの流れ

[4] T. Saito, H. Aida, M. Oguchi, and M. Kuroiwa. Implementation and Evaluation of a Shared Memory Using Broadcast Memory with FIFO, 1993. (in Japanese),*to be appeared in Transactions of IEICE.*

[5] G. Schoinas. (DRAFT)Issues on the implementation of PrOgramming SYstem for distriButed appLications. unpublished, 1991.

[6] M. W. Strevell and H. G. Cragon. Data Type Coherency in Heterogeneous Shared Memory Multiprocessors. In *1990 International Conference on Parallel Processing*, volume 1, pages 555–556, 1990.