計算機アーキテクチャ 91－2
（1991. 11. 21）

OSCAR/Fortranコンパイラのインプリメンテーション

本多弘樹
honda@yamanashi.ac.jp
山梨大学電子情報工学科
岡本雅己 合田憲人 笠原博徳
早稲田大学情報学科

本論文では、実行開始条件で並列性が表現されているマクロタスク集合を並列処理するための並列実行管理方式を提案する。本方式では、マルチプロセッサスケジューリングアルゴリズムを応用したダイナミックスケジューリング手法を用いて、プログラム実行時にマクロタスク集合のプロセッサへの割当て（スケジューリング）を行う。また本方式ではダイナミックスケジューリングのオーバーヘッドを軽減するために、コンパイル対象のソースプログラム専用のスケジューリングコードをソースプログラムコンパイル時にオブジェクトコードの一部としてコンパイラが生成する。本方式をマルチプロセッサシステムOSCARにインプリメントし評価を行った結果、本実行方式がダイナミックスケジューリングによるオーバーヘッドの少ない並列処理を実現するものであることが確認された。

Implementation of OSCAR/Fortran Compiler

Hiroki Honda
honda@yamanashi.ac.jp
Yamanashi Univ. JAPAN
Masami Okamoto, Kento Aida, Hironori Kasahara
Waseda Univ. JAPAN

This paper proposes the scheme of parallel execution of a set of macrotasks and shows the performance evaluation of it on OSCAR. In this scheme, the scheduling function is done by a dynamic scheduling algorithm at the run time, because a set of macrotasks to be executed is not known at the compile time. To reduce the run time overheads caused by scheduling, the compiler generates scheduling codes specifically designed for a target program as parts of the object codes . The result of the performance evaluation shows that the scheme allows us efficient parallel processing of macrotasks with low overheads.

1 まえがき

マクロタスク集合 [2] の並列処理に際してはどのマクロタスクが実行されるかが実行時に決定される。よって並列実行管理に当たって、マクロタスクのプロセッサへの割当ては実行時に行う必要がある。

従来のマクロタスク（粗粒度タスク）並列処理における並列実行管理方式は、cobegin/coend のシンタックスでプログラムが書かれている（またはその様にトランスレートされている）ことを前提とし、OS でのマルチプロセスの概念を応用し fork-jion 等の OS のプロセス生成機能を用いて実現されていた。このため当然のこととして並列性は cobegin/coend のシンタックスで表現されている必要がある。また OS（またはランタイムルーチン）によって実行管理がなされるためオーバーヘッドが大きく、並列処理効果を得るためにはマクロタスクを大きくしなければならない。これは並列性を犠牲にすることとなる。

実行開始条件 [2] として並列性が表現されているマクロタスク集合の場合、プログラム全体での並列性を表現しており cobegin/coend タイプのシンタックスになっていないため、直接 fork/join を用いて実現することは出来ない。

そこで実行開始条件として並列性が表現されているマクロタスク集合を並列実行するための管理方式を開発した。

本方式は実行開始条件として並列性が表現されているマクロタスク集合を並列実行するという本来の機能の他に、次の特徴を持つ。

1. 並列実行管理におけるマクロタスクのプロセッサへの割当ては、マルチプロセッサスケジューリングアルゴリズムを応用したダイナミックスケジューリング手法を用いてプログラム実行時に行う。

2. 並列実行管理のオーバーヘッドを軽減するために、ソースプログラム専用の並列実行管理コードをソースプログラムコンパイル時にオブジェクトコードの一部として生成しそれによって並列実行管理をおこなう。

本論文ではまず提案する方式の前提事項と本並列実行管理方式の実行論理について述べる。次に実行管理方式と本方式を実現するコードの生成法について述べる。さらに細粒度タスク・粗粒度タスク・DO ループの並列処理を組み合わせた並列処理手法の実現について述べる。最後に本方式の OSCAR[1] 上での性能評価について述べる。

2 前提

本論文では、次の各事項に関する前提と定義は [2] で述べたものと同じであるとして、以降の議論を進める。

- 対象とするプログラム
- マクロタスクの定義
- マクロタスク集合とマクロフローグラフ
- 実行開始条件とマクロタスクグラフ

なお実行開始条件を求める際には便宜上、出口疑似マクロタスクは全てのマクロタスクからデータ依存が存在するとして実行開始条件を算出する。この出口疑似マクロタスクの実行開始条件は後述するマクロタスクのレベル算出、及び実行終了判定に用いる。

3 並列実行論理

マクロタスク集合の並列実行論理を次の様に規定する。

1. マクロタスク集合の並列処理とは、マクロタスクをプログラム実行時にプロセッサへ順次、動的に割り当て並列実行することである。

2. ある時点でプロセッサへの割当ての対象となるマクロタスクは、実行開始条件が成立したマクロタスク（レディマクロタスク）である。レディマクロタスクのみが割当て対象となることは、次の意味を持つ。

   (a) 実行されるかどうかが決定していないマクロタスクのプロセッサへの割当て・実行は行わない。すなわち実行されるかどうかが決定していないマクロタスクを割当て実行し、実行・不実行が決定した段階でデータをセーブするか否かを判断する方法は取らない。

   (b) データ依存関係にある二つのマクロタスクを並列に実行することは行わない。すなわちデータ依存関係にあるマクロタスクを同時に実行しマクロタスク内部で個別の同期を取ることはしない。

(c) 他のマクロタスクへの条件分岐を持つマクロタスク内での条件評価と分岐方向の実行管理プログラムへの通達は、そのマクロタスク開始から終了までの間であればいつ行われてもよい。

3. レディマクロタスクの内どのマクロタスクから割当てを行うかの判断は、割当て戦略によって決定される。

4. 実行開始条件の検査と割当てはコントロールプロセッサ (CP) 上の実行管理プログラムが、マクロタスクの終了情報・分岐情報・プロセッサの負荷状況を監視しながらおこなう。

4 ダイナミックスケジューリングアルゴリズム

本ダイナミックスケジューリングアルゴリズムでは、前処理としてマクロタスクのコスト算出、マクロタスクの優先度の決定をコンパイル時に行い、それらの情報を基に実行時にプロセッサへの割当てを行う。

4.1 マクロタスクのコスト算出

マクロタスクのコストは各マクロタスクの中間コード量から算出する。分岐・ループを含む場合は次の様に算出する。

1. 分岐によりいくつかのコストが算出されうる場合には、コスト最大のものをマクロタスクのコストとする。

2. ループを含む場合、ループボディのコストにループの反復数を乗じたものをそのループのコストとする。反復数がコンパイル時に知り得ない場合には、特定の基準によって選択された定数を反復数とする。

4.2 マクロタスクのレベルと優先度

マクロタスクグラフ上での各タスクのレベルを算出し、レベルの大きい順に優先度を与える。レベルはヒューリスティックなスタティックスケジューリング手法であるCP法（クリティカルパススケジューリング）に用いられるレベル [5, 4] の考え方を応用したもので、マクロタスクグラフ上で各マクロタスクから出口疑似マクロタスクに至る最長パスの長さを各マクロタスクのレベルとする。但し、スタティックスケジューリングを適用するタスクグラフと異なり、マクロタスクグラフは条件分岐によるOR関係のエッジを含んでおり、実際に実行されるマクロタスク集合内での最長パスをコンパイル時に知ることはできない。そこで本手法では条件分岐を含むマクロタスクのレベルを次の規則に従い算出する。

1. AND関係にあるエッジ群に対してはその中で最長のパス長のものをANDエッジ群の最長パスとする（CP法と同じ）。

2. OR関係にあるエッジ群に対してはその分岐確率（推定不可能な場合には確率的に平等とする）によって重み付けした平均パス長をORエッジ群の最長パスとする。

4.3 マクロタスク割当ての戦略

4.3.1 先行割当てを行わない場合

マクロタスクのプロセッサへの割当ては、各時点で実行開始条件がtrueのマクロタスク（レディマクロタスク）を優先度の高いものから順に空きプロセッサ（マクロタスクの実行を行っていないプロセッサ）に割り当てる、という方法で行う。

4.3.2 先行割当てを行う場合

先行割当てとは実行が終了していないプロセッサに対してもあらかじめ設定される"先行割当て数"のマクロタスクを割り当てることである。マクロタスクのプロセッサへの割当ては次のように行われる。

1. レディマクロタスクを優先度の高いものから順に空きプロセッサに割り当てる。

2. レディマクロタスクの中で最も優先度の高いマクロタスクを最も負荷（実行中のマクロタスク及び先行割当てされているマクロタスクの推定処理時間の総和）の小さいプロセッサへ割り当てる。これを全プロセッサに先行割当て数のマクロタスクが割り当てられるまで繰り返す。

なお実際のインプリメントでは先行割当てを行わない場合は先行割当て数=0として実現している。

5 並列実行管理コード

並列実行管理コードはコントロールプロセッサコードとプロセッサコードとからなる。

コントロールプロセッサコードは次の機能からなる。

1. プロセッサへのマクロタスクの割当て（割当て）
2. プロセッサから通達されるマクロタスク実行情報の監視（主プログラム）
3. 実行開始条件の検査とレディマクロタスクキューへのマクロタスクの投入（STATE処理・条件検査）
4. レディマクロタスクキューの管理（ソート）

プロセッサコードは次の機能からなる。

1. マクロタスクの割当ての待ち受け（主プログラム）
2. マクロタスクの実行と実行情報のコントロールプロセッサへの通達（MT処理）

各コードはコンパイル時にソースプログラム専用のものがオブジェクトコードと共に生成される。図1図2図3に各コードの基本的な構成を示す。

5.1 コントロールプロセッサとプロセッサ間の共有変数

コントロールプロセッサとプロセッサ間のデータ通信は共有変数Pを介して行われる。図1図2図3のプログラムではPを共有メモリ中の変数としているが、OSCAR上での実際のインプリメントでは各PE/CPのデュアルポートメモリを使用することにより、システムバスの使用回数を減らしている。変数Pは3つのエントリを持つ構造体で次のように使用される。

Assign: コントロールプロセッサがプロセッサに割り当てるマクロタスクのMT番号を書き込む。

Branch: プロセッサが分岐に対応したState番号を書き込む。State番号とは全ての原子条件につけた通し番号である。

Finish: プロセッサが終了に対応したState番号を書き込む。

Pは各プロセッサに対応して先行割当て数分（NumOfPe*NumOfPre）確保される。

```
NumOfPe  : プロセッサ台数
NumOfPre : 先行割当て数
NumOfMT  : MT数
共有メモリ中の変数
  P: array[0:NumOfPe-1][0:NumOfPre-1] of
    record
      Assign: MT番号;
      Branch: State番号;
      Finish: State番号
    end
program CP
CPローカルメモリ中の変数
  Load:array[0:NumOfPe] of integer;initialized 0
  ReadyQ : レディマクロタスクキュー
  Ptr : array[0:NumOfPre-1] of
    record
      Head:integer initialized 0;
      Tail:integer initialized 0;
    end
  MTcondtion: array[0:NumOfMT] of
    record
      done: boolean initialized false
      conditions[]: boolean initialized false
    end
  Time:array[0:NumOfMT-1] of 各MTの処理時間
begin
実行可能条件=trueのMTを優先度順にReadyQに投入;
割当て;
repeat
  i := 1
  while(i<NumOfPe) /* Unroled */
    if P[i][Ptr[i].Tail].Branch <> 0
      then begin
        STATE処理(P[i][Ptr[i].Tail].Branch);
        P[i][Ptr[i].Tail].Branch := 0
        割当て;
        ソート
      end
    if P[i][Ptr[i].Tail].Finish <> 0
      then begin
        割当て;
        STATE処理(P[i][Ptr[i].Tail].Finish);
        P[i][Ptr[i].Tail].Finish := 0;
        Ptr[i].Tail:=mod(Ptr[i].Tail+1,NumOfPre)
        割当て;
        ソート
      end
    i := i + 1
  endwhile
until all MTcondition[endMT].conditions = true
end.
```

図1: コントロールプロセッサコード

```
procedure STATE処理(StateNo)
begin
StateNoに対応した"個別STATE処理"にジャンプ
個別STATE処理:
if StateNoがMTkの終了に対するものである
  then Load[i] := Load[i] - Time[MTk]
    StateNoに関係ある全てのMTのMTconditionの
    対応するエントリをtrueとする;
StateNoに関係ある全てのMTlに対して条件検査(MTl);
return
procedure 条件検査(Mt)
begin
if MTcondition[Mt].done = true then return
if all MTcondition[Mt].conditions = true
then begin
  MTcondition[Mt].done := true;
  Mt をReadyQに投入
end
return
```

図 2: コントロールプロセッサコード (続き)

```
program PE
begin
Ptr := 0;
lbl:
if P[MyP][Ptr].Assign <> 0
  then P[MyP][Ptr].Assignに対応した"MT処理"へ
goto lbl
/*MT処理のテンプレート*/
MT処理:
MT本体
P[MyP][Ptr].Branch := state番号;/*必要に応じて*/
MT本体
P[MyP][Ptr].Finish := state番号;
P[MyP][Ptr].Assign := 0;
goto lbl
```

図 3: プロセッサコード

5.2 実行情報の監視・実行開始条件の検査

コントロールプロセッサは全プロセッサの Branch/Finish を常時ポーリングし、Branch または Finish に State 番号の書き込みが行われた場合には、実行開始条件中にその State 番号の原子条件を含んでいるマクロタスクの実行開始条件を検査する。検査コードは各マクロタスクの実行開始条件式の評価計算に対応した専用のものである。

5.3 プロセッサコード内の MT 処理コード

各プロセッサは全てのマクロタスクに対する MT 処理コードを持っている。プロセッサ内でのマクロタスクの起動はそのマクロタスクの MT 処理コードへの分岐という軽い形で行われ、MT 起動に際してレジスタの退避等は行わない。

6 OSCAR 上での階層的並列処理の実現

OSCAR 上でのプログラムの階層的並列処理は

1. マクロタスクを複数のプロセッサエレメント (PE) から構成されるプロセッサクラスタ (PC) に割り当て、マクロタスク間での並列性を活かした並列処理を行い、
2. 各 PC では、基本ブロックの並列処理手法 [3] や従来の DOALL/DOACROSS 手法によって各々のマクロタスクをさらに並列処理する

処理方法である。

6.1 メモリマップ

階層的並列処理を行う際の OSCAR 上のメモリマップを図 5に示す。図中 A から E がステートメントレベルの細粒度並列処理を行うために必要な領域、a) から h) がマクロタスクレベルでの粗粒度並列処理を行うために必要な領域、1) から 8) が両方に共通している領域である。

共通領域: プログラム中で定義される全ての変数は共有メモリ上にその格納場所を持つ (1)2))。定数等は

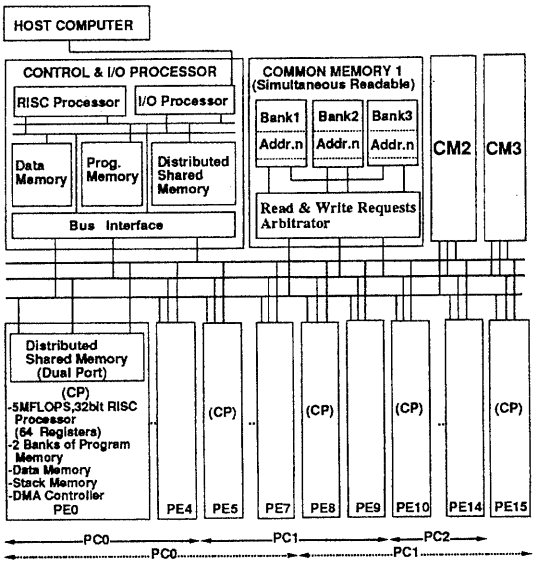

図 4: OSCAR のアーキテクチャ

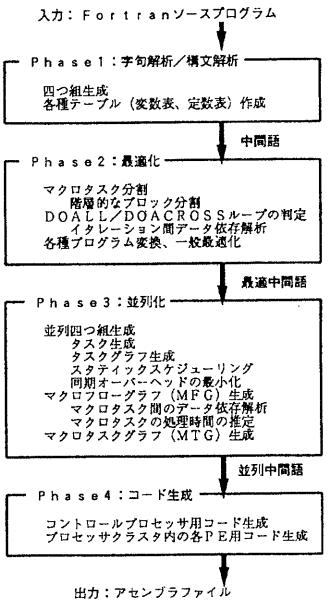

図 6: OSCAR/Fortran コンパイラの構成

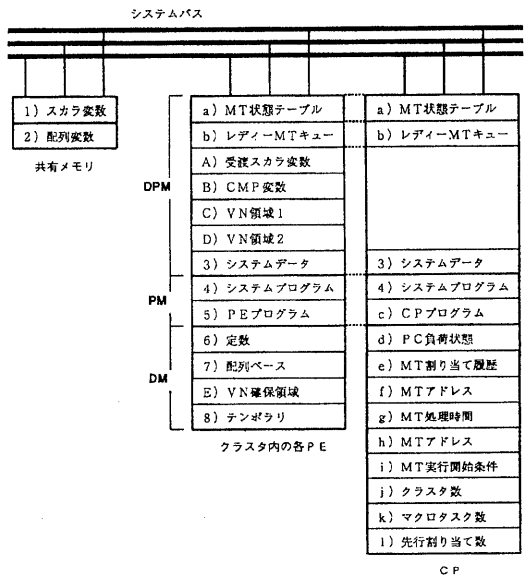

図 5: OSCAR 上のメモリマップ

全 PE の DM 上に配置される（6)7)8)）。プログラムの命令コードは各 PE の PM に配置される。

細粒度並列処理のための領域: 異なる PE 間でデータの授受が必要な場合はその変数がスカラ変数の場合は DPM 上にも領域を取る（A）。その他条件分岐の際の分岐方向を示す変数（B）、タスク（細粒度タスク）間での同期のためのバージョンナンバー（C,E）等、疑似クラスタ内の PE 間で共有されるデータが DPM 上にマップされる。

粗粒度並列処理のための領域: CP から PE への MT の割り当てならびに PE から CP への MT に関する情報を授受する領域を DPM 上に配置する（a)）。また MT のダイナミックスケジューリングに関する各種情報は CP の DM に配置される（d)~l)）。

6.2 コンパイラの構成

OSCAR/Fortran コンパイラは図 6に示す構成となっている。

Phase1: 字句・構文解析を行い逐次中間コードを生成する。

Phase2: プログラムの構造解析を行いマクロタスクを生成する。

Phase3: 各PE用の中間コード（並列中間コード）を生成する。またマクロタスク間のデータ依存解析をおこないマクロフローグラフを生成し、マクロフローグラフからマクロタスクグラフを生成する。

Phase4: 中間語からアセンブラコードを生成する。この際マクロタスクグラフをもとにマクロタスク制御用CP・PEコードも生成する。

7 評価

ダイナミックスケジューリングのオーバーヘッド評価のために作ったプログラム（計算自体には意味の無い）に対する性能評価を示す。このプログラムは図7のマクロフローグラフで示されるマクロタスク構成を持ち、マクロタスク MT1、MT8、MT15 以外の 18 個のマクロタスクは 6 行（22 演算）の代入文を持つ DOALL 処理可能な DO ループであり、MT1、MT8、MT15の条件分岐によって実際に実行されるものは9個（MT2,MT3,MT4,MT9,MT10,MT11,MT16,MT17,MT18）である。

このプログラムを複数台クラスタ（各 PC は 2 台の PE で構成）上で並列処理した際の処理時間及び各 PC に割り当てられた MT の実行時間を表したのが図8である。図中横軸が時間を表し、横棒は使用する PC 数を 1 台、2 台、3 台とした時のそれぞれの PC での処理実行時間を表している。また白枠が割り当てられた MT の処理を、黒塗の部分は PC がマクロタスク割り当て待ち（アイドル状態）になっていることを表す。処理時間の測定は DO ループのイタレーション回数を 100 回、50 回、10 回、2 回としてそれぞれ行った。イタレーション 100 回の場合、3PC では 1PC の時の約 1/2.84 の処理時間となり、50 回の場合約 1/2.76、10 回の場合約 1/2.39、2 回の場合約 1/1.97 となっている。この例のイタレーションが 2 回の場合、DO ループのマクロタスクの計算量はループ制御コードを除くと 22 演算×2 となるがクラスタ内では 2 つの PE で DOALL 処理をしているので各 PE の演算量は 22 演算となる。この場合マクロタスク

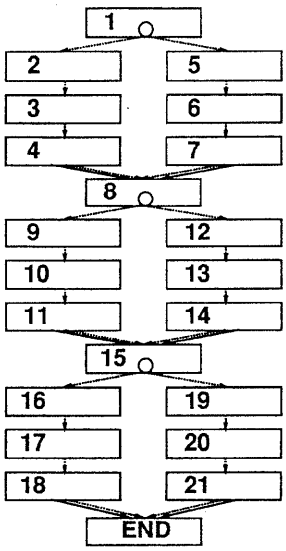

図 7: 例題プログラムのマクロフローグラフ

の計算量が小さくなるなるので相対的に待ち状態が大きくなり並列処理効果が低下し始める。待ち状態には、マクロタスク間の先行制約によってレディーマクロタスクが存在しないために起こるものと、論理的にはレディーマクロタスクが存在しているはずであるがダイナミックスケジューリングによるオーバーヘッドによって実際にPCにマクロタスクが割り当てられないことにより起こるものがある。例えばイタレーション＝2で3クラスタによる処理の場合、MT4 の実行が始まる迄の黒塗部分は、

- MT1 の終了情報を CP が検知し
- レディマクロタスクとなる MT2,MT3,MT4 をレディーマクロタスクキューに投入し
- MT4 を PC に割り当てる

ためのオーバーヘッドである。また MT4 の実行開始から MT2 の実行開始までの時間差は CP が MT4 の割り当てを行ってから MT2 の割り当てを行うためのオーバーヘッドである。現在のインプリメント方式において、この割り当てのためのオーバーヘッド時間は主に

- 各マクロタスクの実行開始条件中の論理積数

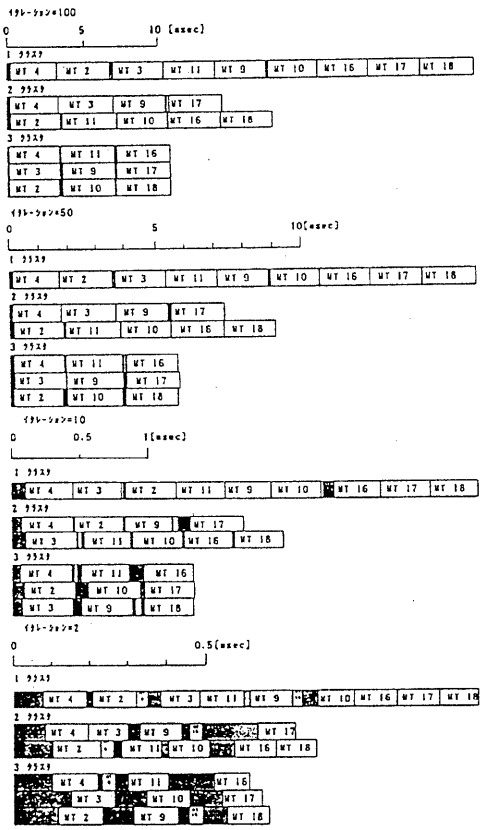

図 8: 階層的並列処理時間

- 一つのマクロタスク実行情報によって影響を受けるマクロタスクの数

に左右される。

8 むすび

本論文では、実行開始条件で並列性が表現されているマクロタスク集合を並列処理するための並列実行管理方式を提案した。また本方式をマルチプロセッサシステム OSCAR にインプリメントした際の評価結果について述べた。さらに階層的並列処理の実現方法についても述べた。

また今回のインプリメントでは最外側ループとそのループ間の基本ブロックをマクロタスクとし、マクロフローグラフにはループが無いとした。しかしこのような分割だけでは各マクロタスク処理時間の違いによって並列処理効果が得られないプログラムも存在する。これに対処するためには最外側ループのループボディをさらにマクロタスクに分割しそのマクロタスク集合を並列処理する実行方式を実現する必要があり、これは今後の課題である。

参考文献

[1] 笠原博徳, 本多弘樹, 成田誠之助, 橋本 親. 汎用マルチプロセッサシステム OSCAR のアーキテクチャ. , コンピュータアーキテクチャシンポジウム論文集, 175–182 ページ, 5 1988.

[2] 本多弘樹, 岩田雅彦, 笠原博徳. Fortran プログラム粗粒度タスク間の並列性検出手法. 信学論, J73-D-1(12):951–960, 12 1990.

[3] 本多弘樹, 水野聡, 笠原博徳, 成田誠之助. Fortran プログラム基本ブロックの並列処理手法. 信学論, J73-D-1(9):756–766, 9 1990.

[4] E. G. Jr. Coffman. *Computer and Job shop Scheduling Theory.* Wiley, New York, 1976.

[5] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms. *IEEE Transactions on Computers*, c-33(11):1023–1, Nov. 1984.