

(19) 日本国特許庁(JP)

(12) 特許公報(B2)

(11) 特許番号

特許第5224498号  
(P5224498)

(45) 発行日 平成25年7月3日(2013.7.3)

(24) 登録日 平成25年3月22日(2013.3.22)

(51) Int. Cl. F 1  
G 0 6 F 9/50 (2006.01) G 0 6 F 9/46 4 6 2 A

請求項の数 21 (全 39 頁)

(21) 出願番号	特願2007-50269 (P2007-50269)	(73) 特許権者	899000068
(22) 出願日	平成19年2月28日 (2007.2.28)		学校法人早稲田大学
(65) 公開番号	特開2008-217134 (P2008-217134A)		東京都新宿区戸塚町1丁目104番地
(43) 公開日	平成20年9月18日 (2008.9.18)	(74) 代理人	100114236
審査請求日	平成21年9月25日 (2009.9.25)		弁理士 藤井 正弘
		(74) 代理人	100075513
			弁理士 後藤 政喜
		(74) 代理人	100120260
			弁理士 飯田 雅昭
		(72) 発明者	笠原 博徳
			東京都新宿区大久保3-4-1 早稲田大 学理工学術院基幹理工学部情報理工学科内
		(72) 発明者	木村 啓二
			東京都新宿区大久保3-4-1 早稲田大 学理工学術院基幹理工学部情報理工学科内
			最終頁に続く

(54) 【発明の名称】 メモリ管理方法、情報処理装置、プログラムの作成方法及びプログラム

(57) 【特許請求の範囲】

【請求項1】

プロセッサによって使用されるメモリの記憶領域を管理する方法であって、  
前記プロセッサは、タスクの実行時にアクセスされるデータを格納するメモリに接続されてお

り、  
前記タスクを含むプログラムの解析によって得られた情報に基づいて、前記プログラム  
で使用されるデータのサイズに適合するように、前記メモリの記憶領域を分割するための  
ブロックの複数の異なるサイズを決定し、

前記決定された複数のサイズのブロックに前記記憶領域を分割し、

前記タスクの実行時にアクセスされるデータに適合するサイズのブロックを選択し、  
前記選択されたブロックに、前記タスクの実行時にアクセスされるデータを格納するこ  
とを特徴とするメモリ管理方法。

【請求項2】

請求項1に記載のメモリ管理方法であって、

前記複数のサイズのブロックのサイズは互いに整数倍の関係にあることを特徴とするメ  
モリ管理方法。

【請求項3】

請求項1又は2に記載のメモリ管理方法であって、

前記データを前記選択されたブロックへ割り当てることを決定した後、データ転送手段  
によって、前記データを前記選択されたブロックに格納し、

10

20

前記ブロックの解放タイミングまでに、前記データ転送手段によって、前記選択されたブロックに格納されたデータを読み出し、他のメモリに格納することを特徴とするメモリ管理方法。

【請求項 4】

請求項 1 から 3 のいずれか一つに記載のメモリ管理方法であって、

前記プログラムの解析によって得られた情報に基づいて定められた複数の形状及びサイズのテンプレートを、適合する大きさの前記ブロックに割り当て、

前記割り当てられたテンプレートに適合する形状及び大きさのデータを、前記テンプレートに格納することを特徴とするメモリ管理方法。

【請求項 5】

請求項 4 に記載のメモリ管理方法であって、

前記タスクでアクセスされるデータに  $n$  次元の配列データが含まれる場合に、各ブロックに割り当て可能な複数種類のテンプレートから、前記タスクの実行時にアクセスされる配列データの次元  $n$  に 1 を加えた次元  $n + 1$  を有し、加えられた次元以外の各次元の最大値が前記タスクの実行時にアクセスされる配列データの各次元の最大値以上のテンプレートを割り当て、

前記加えられた次元の値によって、アクセスされるブロックが異なるように、次元が異なるか又は同じ次元で各次元のサイズが異なる複数種類の前記テンプレートの一つを前記各ブロックに割り当ててることを特徴とするメモリ管理方法。

【請求項 6】

プロセッサ及び前記プロセッサによってアクセスされるデータを格納するメモリを備える情報処理装置であって、

前記プロセッサは、

前記プロセッサで実行されるプログラムの解析によって得られた情報に基づいて、前記プログラムで使用されるデータのサイズに適合するように、前記メモリの記憶領域を分割するためのブロックの複数の異なるサイズを決定し、

前記決定された複数のサイズのブロックに前記記憶領域を分割し、

前記タスクの実行時にアクセスされるデータに適合するサイズのブロックを選択し、

前記選択されたブロックに、前記タスクの実行時にアクセスされるデータを格納することを特徴とする情報処理装置。

【請求項 7】

請求項 6 に記載の情報処理装置であって、

前記ブロックは、複数のサイズのブロックを含み、前記ブロックの複数のサイズは整数倍の関係にあることを特徴とする情報処理装置。

【請求項 8】

請求項 6 又は 7 に記載の情報処理装置であって、

前記プロセッサは、

前記データを前記選択されたブロックへ割り当ててることを決定した後、データ転送手段によって、前記データを前記選択されたブロックに格納し、

前記ブロックの解放タイミングまでに、前記データ転送手段によって、前記選択されたブロックに格納されたデータを読み出し、他のメモリに格納することを特徴とする情報処理装置。

【請求項 9】

請求項 6 から 8 のいずれか一つに記載の情報処理装置であって、

前記プロセッサは、

前記プログラムの解析によって得られた情報に基づいて定められた複数の形状及びサイズのテンプレートを、適合する大きさの前記ブロックに割り当て、

前記割り当てられたテンプレートに適合する形状及び大きさのデータを、前記テンプレートに格納することを特徴とする情報処理装置。

【請求項 10】

請求項 9 に記載の情報処理装置であって、

前記タスクでアクセスされるデータに  $n$  次元の配列データが含まれる場合に、各ブロックに割り当て可能な複数種類のテンプレートから、前記プログラムでアクセスされる配列データの次元  $n$  に 1 を加えた  $n + 1$  次元を有し、加えられた次元以外の各次元の最大値が前記プログラムでアクセスされる配列データの各次元の最大値以上のテンプレートを割り当て、

前記加えられた次元の値によって、アクセスされるブロックが異なるように、次元が異なるか又は同じ次元で各次元のサイズが異なる複数種類の前記テンプレートの一つを前記各ブロックに割り当ててことを特徴とする情報処理装置。

【請求項 1 1】

プロセッサによって実行可能なプログラムを作成する方法であって、

前記方法は、計算機において実行され、

前記計算機が、

元プログラムの情報をコンパイラによって解析し、

前記元プログラムに含まれる各タスクの実行に必要なデータを特定し、

前記元プログラムの解析によって得られた情報に基づいて、前記作成されるプログラムで使用されるデータのサイズに適合するように、前記メモリの記憶領域を分割するためのブロックの複数の異なるサイズを決定し、前記作成されるプログラムが実行されるプロセッサのメモリの記憶領域を前記決定された複数のサイズのブロックに分割する命令と、前記タスクの実行時にアクセスされるデータに適合するサイズのブロックを選択する命令と、前記選択されたブロックに、前記タスクの実行時にアクセスされるデータを格納する命令とを追加することによって、前記プロセッサ上で実行可能なプログラムを生成することを特徴とするプログラムの作成方法。

【請求項 1 2】

請求項 1 1 に記載のプログラムの作成方法であって、

前記複数の異なるサイズのブロックのサイズは互いに整数倍の関係にあることを特徴とするプログラムの作成方法。

【請求項 1 3】

請求項 1 1 又は 1 2 に記載のプログラムの作成方法であって、

前記計算機は、

前記コンパイラが前記元プログラムを解析して得られた情報に基づいて、

前記データを前記選択されたブロックへ割り当ててことを決定した後、データ転送手段によって、前記データを前記選択されたブロックに格納する命令と、

前記ブロックの解放タイミングまでに、前記データ転送手段によって、前記選択されたブロックに格納されたデータを読み出し、他のメモリに格納する命令を追加することによって、前記プロセッサ上で実行可能なプログラムを生成することを特徴とするプログラムの作成方法。

【請求項 1 4】

請求項 1 1 から 1 3 のいずれか一つに記載のプログラムの作成方法であって、

前記計算機は、

前記元プログラムを前記コンパイラによって解析して得られた情報に基づいて、定められた複数の形状及びサイズのテンプレートを、適合する大きさの前記ブロックに割り当てる命令と、

前記割り当てられたテンプレートに適合する形状及び大きさのデータを、前記テンプレートに格納する命令と、を追加することによって、前記プロセッサ上で実行可能なプログラムを生成することを特徴とするプログラムの作成方法。

【請求項 1 5】

請求項 1 4 に記載のプログラムの作成方法であって、

前記計算機は、

前記元プログラムを前記コンパイラによって解析して得られた情報に基づいて、

10

20

30

40

50

前記タスクでアクセスされるデータに  $n$  次元の配列データが含まれる場合に、各ブロックに割り当て可能な複数種類のテンプレートから、前記タスクでアクセスされる配列データの次元  $n$  に 1 を加えた次元  $n + 1$  を有し、加えられた次元以外の各次元の最大値が前記タスクでアクセスされる配列データの各次元の最大値以上のテンプレートを割り当てる命令と、

前記加えられた次元の値によって、アクセスされるブロックが異なるように、次元が異なるか又は同じ次元で各次元のサイズが異なる複数種類の前記テンプレートの一つを前記各ブロックに割り当てる命令と、を追加することによって、前記プロセッサ上で実行可能なプログラムを生成することを特徴とするプログラムの作成方法。

【請求項 16】

請求項 11 に記載のプログラムの作成方法であって、  
前記計算機は、

前記元プログラムを前記コンパイラによって解析して得られた情報に基づいて、解放する前記領域及び前記領域を解放するタイミングを決定し、

前記割り当てられた領域を解放するために、前記決定されたタイミングまでに前記メモリに書き込まれたデータを読み出す命令を追加することによって、前記プロセッサ上で実行可能なプログラムを生成することを特徴とするプログラムの作成方法。

【請求項 17】

請求項 16 に記載のプログラムの作成方法であって、  
前記計算機は、

前記元プログラムを前記コンパイラによって解析して得られた情報に基づいて、

前記メモリの領域を割り当てた後に、データ転送手段によって、前記データを前記メモリに格納する命令、及び

前記メモリの領域の解放タイミングまでに、前記データ転送手段によって、前記メモリに格納されたデータを読み出し、他のメモリに格納する命令を、前記プロセッサ上で実行可能なプログラムに追加して前記プロセッサ上で実行可能なプログラムを生成することを特徴とするプログラムの作成方法。

【請求項 18】

請求項 11 に記載のプログラムの作成方法であって、

前記元プログラムを前記コンパイラによって解析して得られた情報は、前記元プログラムでアクセスされるデータの情報、前記データが次にアクセスされるタイミングの情報、前記データをアクセスするプロセッサの情報の少なくとも一つを含むことを特徴とするプログラムの作成方法。

【請求項 19】

請求項 11 に記載のプログラムの作成方法であって、

前記作成されるプログラムが実行される前記プロセッサは複数のプロセッサコアを備えるマルチプロセッサであって、

前記方法は、

前記計算機が、

前記タスクをいつどのプロセッサに実行させるかを決定し、

前記決定されたプロセッサに前記タスクを割り当てる命令を追加することによって、前記プロセッサ上で実行可能なプログラムを生成することを特徴とするプログラムの作成方法。

【請求項 20】

請求項 11 に記載のプログラムの作成方法であって、

前記計算機は、

前記元プログラムを前記コンパイラによって解析して得られた情報に基づいて、前記前記プロセッサ上で実行可能なプログラムの実行時にアクセスされるデータを前記一つのブロックに収まるように前記プロセッサ上で実行可能なプログラムに含まれるタスクを分割して、前記プロセッサ上で実行可能なプログラムを生成することを特徴とするプログラム

10

20

30

40

50

の作成方法。

【請求項 2 1】

請求項 1 9 に記載のプログラムの作成方法であって、

前記元プログラムは多重ループを含み、

前記方法は、

前記計算機が、

前記元プログラムを前記コンパイラによって解析して得られた情報に基づいて、外側のループの分割によって生成されたタスクの実行時にアクセスされるデータが前記ブロックに収まるか否かを判定し、

前記外側のループの分割によって生成されたタスクの実行時にアクセスされるデータが前記ブロックに収まらなければ、更に内側のループを分割することによって、前記データのサイズを変更して前記プロセッサ上で実行可能なプログラムを生成することを特徴とするプログラムの作成方法。

10

【発明の詳細な説明】

【技術分野】

【0001】

本発明は、複数のプロセッサコアで構成されるマルチプロセッサシステムにおけるメモリの管理方法に関し、特に、コンパイラが取得した情報に基づいて、プロセッサによってアクセスされるデータをメモリの分割された領域に割り当てる方法に関する。

【背景技術】

20

【0002】

複数のプロセッサコアを一つのチップ上に集積したマルチコアプロセッサ（チップマルチプロセッサ）が、各マイクロプロセッサメーカーによって次々に発表されている。スーパーコンピュータ、サーバ、デスクトップコンピュータ及びPCサーバ分野の他、情報家電及び装置組み込みの分野（例えば、携帯電話機、ゲーム機、カーナビゲーションシステム、デジタルテレビ受像機、HDD/DVDレコーダ・プレーヤ等）においても、マイクロプロセッサのマルチコア化の動きが見られる。

【0003】

このように、現在情報家電からスーパーコンピュータに至るほとんどの情報機器においてマルチコアプロセッサが使われるようになっており、今後、さらに多くの情報機器にマルチコアプロセッサが組み込まれていくと考えられる。

30

【0004】

マルチコアプロセッサは、細粒度命令レベルの並列性だけでなく、より並列性の大きいループレベルの並列性、さらに粒度の粗いループ間の並列性、関数間の粗粒度タスク並列性も利用することができる。このように、マルチコアプロセッサは、より大きな並列性の利用によって、プロセッサの処理性能を向上させることができる点で有利である。また、マルチコアプロセッサは、n台のプロセッサコアを用い同一性能を達成することができるので、クロック周波数をn分の1にし、印加する電圧も下げることによって、消費電力（電圧の2乗で増大する）を低く抑えることができる点でも有利である。

【0005】

40

また、ソフトウェア面では、マルチプロセッサ用の並列プログラミングは、通常、チューニングに多大な時間を要することから、アプリケーションソフトウェアの開発が大変である。しかし、比較的少数のプロセッサが集積されている現時点では、逐次プログラムを自動的に並列化する自動並列化コンパイラによって高性能を得ることができる。情報家電分野ではアプリケーションの質と数が市場での競争力を決めることから、コンパイラによって、4コア、8コア、16コアのマルチプロセッサ用のプログラムの自動並列化が可能となれば、マルチコアの優位性が高まる。

【0006】

また、マルチグレイン並列化では、文レベル、ループレベル、より粗いレベル（例えば、ループ間、サブルーチン間、ベーシックブロック間）の全ての並列性を組み合わせて最

50

早実行可能条件解析によって並列性を抽出する（例えば、特許文献1参照）。

【特許文献1】特開2001-175619号公報

【発明の開示】

【発明が解決しようとする課題】

【0007】

このような、ローカルメモリの最適化は、従来から行われていた。しかし、従来のローカルメモリの最適化は、並列ループが連続する場合に、並列可能なループを連続的に実行することによってメモリ上のデータを使い回すだけのものであった。

【0008】

しかし、一つのループで使用されるデータサイズはローカルメモリのサイズよりも大きい場合が多く、ループの中でデータのロード及びストアが発生していた。このとき、プロセッサは、次に使用するデータが準備されるまで処理を待つ必要があり、プロセッサによる処理のオーバーヘッドが発生していた。

【0009】

また、ローカルメモリ上のデータを使いながらプログラムを実行していくデータローカライゼーションによると、ローカルメモリだけを使って処理をするために、逐次形のループ及びベーシックブロックで使用される大きなデータもローカルメモリに格納しなければならない。ローカルメモリに載せられないデータは共有メモリに格納していた。よって、プログラム全域に渡ってデータをローカライゼーションを適用することが望まれている。

【0010】

また、スタティックスケジューリング時及びダイナミックスケジューリングコードの生成時には、各プロセッサ上のローカルメモリ又は分散共有メモリを有効に使用し、プロセッサ間のデータ転送量を最小化するためのデータローカライゼーションも用いられる。

【0011】

さらに、プロセッサの集積度が高まり、1チップに含まれるプロセッサコア数が増えてくると、プログラムを並列化してもそれほど処理性能が向上しない。なぜなら、プロセッサの動作が早くなっても、プロセッサによる処理速度とメモリアクセス速度との差が拡大することによって、プロセッサが使用するデータを適切なタイミングでメモリへ供給できないという、メモリウォールの問題が生じるからである。

【課題を解決するための手段】

【0012】

そこで、プロセッサに近接して設けられているメモリを上手に使う必要がある。すなわち、プロセッサに近接するメモリの記憶容量は小さいので、大きなデータが必要な場合は、データを分割してメモリに載せることが必要となる。さらに、メモリ間のデータの転送は時間がかかるので、メモリに載っているデータを使い回せるように、プログラムの処理の順序を工夫したスケジューリングをすることが求められる。さらに、メモリ間でデータを転送するときにDMAコントローラを使って、オーバーヘッドを隠す必要がある。

【0013】

コンパイラは、プロセッサで実行されるプログラムの並列性を解析するときに、プログラム内のタスクの実行順序の情報を取得し、タスク間でのデータの依存関係（定義、参照の関係）も解析する。また、分岐が決まると、同じデータを使うプログラムが分かる。このように、コンパイラが取得した情報によって、メモリに格納されたデータが使用されるタイミングが分かり、複数のタスク間でデータを使い回すことができる。

【0014】

すなわち、本発明では、プロセッサ近傍に配置された高速なメモリに格納されたデータを、可能な限り連続して長期間置いたまま処理を続けられるようにするために、データを効率よく配置する。具体的には、必要なデータをプロセッサ近傍の高速なメモリに格納し、不必要となったデータを低速だが大容量のメモリへ順次転送する。さらに、本発明では、データを分割してローカルメモリに割り当てる。また、ローカルメモリに格納されたデータを長期間使えるようにするために、コンパイラが持っている情報（データが何時どこ

10

20

30

40

50

で使われるかの情報)に基づいて、どのデータを追い出すかを定める。また、必要となるデータを先にロードするようにDMAをスケジューリングする。

【0015】

なお、本発明は、同一種類のプロセッサにおけるメモリ管理のみでなく、異なる種類のヘテロジニアス・マルチプロセッサにおけるメモリ管理にも適用することができる。

【発明の効果】

【0016】

本発明によれば、メモリの記憶領域の管理が容易になることから、ローカルメモリ及び分散共有メモリへのデータの配置を最適化することができる。これにより、メモリに載っているデータを使い回すことができ、メモリ間でデータの転送を減らすことができる。

10

【発明を実施するための最良の形態】

【0017】

まず、本発明の概要を説明する。

【0018】

本発明は、コンパイラがプログラムの解析によって取得した情報に基づいて、メモリの記憶領域を管理する。コンパイラはプログラムをコンパイルする際に、プログラムの情報を取得する。具体的には、コンパイラは、前記プログラムで使用されるデータの情報、前記データが次に使用されるタイミングの情報、前記データが使用されるプロセッサの情報を、プログラムの解析によって取得できる。すなわち、プログラムによってデータが使用されるタイミングを取得できる。本発明の第1の特徴は、プログラムを解析して得られた情報に基づいて、メモリの記憶領域の割り当てを管理することである。

20

【0019】

具体的には、コンパイラは、プログラム(例えば、ネストされた各階層の処理)の実行スケジュールの情報を持っているので、データがいつアクセスされるかの情報を取得できる。よって、プログラムの実行スケジュールに基づいてメモリの記憶領域をデータに最適に割り当てることができるので、データの転送を最小化することができる。

【0020】

さらに、メモリに割り当てられたデータを、どのプロセッサが、いつ必要とするかの情報を取得できる。よって、プロセッサによる処理に影響することなくDMAコントローラによって、データをメモリに連続的に供給(ロード)できる。よって、データがメモリに供給されるのを待つために、プロセッサが止まらない。

30

【0021】

さらに、コンパイラは、データがプログラムによっていつアクセスされるかの情報を取得できるので、既に不要となったデータ又は直ぐにはアクセスされないデータを特定することができ、DMAコントローラによって不要となったデータ又は直ぐにはアクセスされないデータをメモリの記憶領域から掃き出す(ストア)ことができる。この将来の情報に基づいたデータの掃き出しは、従来用いられていたLRU(Least Recently Used)と異なり、最近使われていないが直ぐに使われるかもしれないデータが掃き出されることがなく、メモリの利用を最適化して、メモリ間のデータ転送を減らすことができる。

【0022】

すなわち、本発明の第2の特徴は、プログラムを解析して得られた情報に基づいて、メモリへ/からデータを転送するタイミングを決めることである。

40

【0023】

このようにメモリの記憶領域を管理するために、メモリの記憶領域を固定サイズのブロックに分割して、ブロック毎にデータを割り当てる。記憶領域が分割されるブロックのサイズは、コンパイル時に取得したプログラムの特性(プログラムで使用されている配列サイズ、配列の形状、プログラムの実行パターン等)に応じて適切なサイズを決定する。また、ブロックのサイズは整数倍(例えば、2の倍数)の関係にするとよい。

【0024】

すなわち、本発明の第3の特徴は、プログラムを解析して得られた情報に基づいて、メ

50

メモリの記憶領域を適切なサイズのブロックに分割して、データを割り当てることである。プログラムの特性に応じて記憶領域を固定サイズのブロックに分割することは、ハードウェアやOS（オペレーティングシステム）では成し得ない。これは、ハードウェアやOSによって記憶領域を分割すると、いつも決まったサイズで分割されてしまうからである。

【0025】

さらに、アクセスするデータの種類や範囲によって、プログラムに使用されるデータのサイズが異なる場合がある、よって、プログラム中で使用されるデータのサイズに適合するように、複数のサイズのブロックを用意する。そして、データのサイズに適合するように割り当てられるブロックのサイズを変える。

【0026】

さらに、プログラムの実行時に必要となるデータ（ある瞬間にメモリに載っていないデータ）を「ワーキングセット」という）の分割にあわせて、プログラムも分割する。例えば、ループを2分の1に分割すると、使用されるデータも2分の1になる。本発明の第4の特徴は、プログラムを解析して得られた情報に基づいて、メモリの記憶領域にワーキングセットが載るように、プログラムの分割数を決めることである。例えば、プログラムの分割は、一つの大きなループを分割して、細かい単位のループを繰り返し実行する。

【0027】

さらに、多重ループでは、通常、多次元配列変数が使用される。1次元アドレス空間を有する固定サイズのブロックに多次元配列変数を割り当てる際に、1次元のブロックに多次元配列テンプレートを割り当てる。この多次元配列テンプレートの形状及びサイズは、プログラムで使用される配列変数の形状及びサイズに基づいて決定される。本発明の第5の特徴は、プログラムを解析して得られた情報に基づいて、ブロックに割り当てられるテンプレートを決定することである。これによって、配列変数の添字を変換する手間やプログラムの複雑化を避けながら、任意の配列をメモリ上の任意のアドレスに割り当てることができる。

【0028】

なお、以下の本発明の実施の形態では、データの使用について具体的に説明するが、プロセッサによるデータの使用（参照）の他に、データの定義（データを計算してメモリへ格納すること）におけるメモリ管理も本発明の範疇である。すなわち、本発明は、データの使用及び定義の両方を含むデータのアクセスについて適用可能されるものである。

【0029】

次に、本発明の実施の形態について、図面を参照して説明する。

【0030】

図1は、本発明の実施の形態のシングルチップマルチコアプロセッサの構成図である。

【0031】

本発明の実施形態のシングルチップマルチコアプロセッサ10は、複数のプロセッサコア（PC0、PC1、…、PCn）16を含む複数のマルチコアチップ（CMP0、…、CMPm）10、複数の集中共有メモリ（CSM0、…、CSMj）14、入出力制御を行う複数の入出力用チップ（I/O CSP0、…、I/O CSPk）18、及び、チップ間結合網（InterCCN）12を備える。

【0032】

チップ間結合網12は、既存の接続技術（クロスバースイッチ、バス、マルチステージネットワーク等）によって実現され、複数のプロセッサコア16、複数の集中共有メモリ14及び入出力用チップ18を接続する。集中共有メモリ14は、システム中の全プロセッサコア16によって共有され、各プロセッサコア16からアクセス可能なメモリである。集中共有メモリ14は、マルチコアチップ10内に備わる集中共有メモリ28を補完する。

【0033】

各マルチコアチップ10は、複数のプロセッサコア（PC）16、集中共有メモリ（C

10

20

30

40

50



S M / L 2 C a c h e ) 2 8 及びチップ内結合網 ( I n t e r C C N ) 3 4 を備える。各プロセッサコア 1 6 は、C P U 2 0、分散共有メモリ ( D S M : distributed shared memory ) 2 2、ローカルプログラムメモリ ( L P M / I - C a c h e ) 2 4、ローカルデータメモリ ( L D M / D - c a c h e ) 2 6、データ転送コントローラ ( D T C ) 3 0、ネットワークインターフェイス ( N I ) 3 2 及び電力制御レジスタ ( F V R ) 3 6 を備える。

【 0 0 3 4 】

C P U 2 0 は、整数演算及び浮動小数点演算が可能なものであればよく、特に限定されない。例えば、データのロード及びストアのアーキテクチャが単純なシングルイシュー R I S C アーキテクチャの C P U を用いることができる。また、スーパースカラプロセッサ、V L I W プロセッサ等も用いることができる。

10

【 0 0 3 5 】

分散共有メモリ ( D S M ) 2 2 は、デュアルポートメモリで構成されており、データ転送コントローラ 3 0 を介して、他のプロセッサコア 1 6 からデータを直接読み書きすることができ、タスク間のデータ転送に使用される。

【 0 0 3 6 】

ローカルプログラムメモリ ( L P M ) 2 4 は、スケジューラによって定められたタスクの実行順序に従って、実行すべき命令を他のメモリから先読みしてキャッシュする。なお、プログラムの特徴に応じ、通常のデータキャッシュメモリとしても使用でき、ヒットミスを少なくするためのキャッシュとしても使用される。

20

【 0 0 3 7 】

ローカルデータメモリ ( L D M ) 2 6 は、各プロセッサコア 1 6 内だけでアクセスできるメモリであり、各プロセッサコア 1 6 に割り当てられたタスクで使用されるデータ ( 例えば、配列変数 ) を格納する。また、ローカルデータメモリ 2 6 は、L 1 データキャッシュに切り替えることができる。

【 0 0 3 8 】

データ転送コントローラ ( D T C ) 3 0 は、公知の D M A コントローラによって構成され、スケジューラによって定められたタイミングに従って、実行すべき命令や使用されるデータをメモリ間で転送する。具体的には、自又は他のプロセッサコア 1 6 上のローカルメモリ 2 6、自及び他のプロセッサコア 1 6 上の分散共有メモリ 2 2、自及び他のマルチコアチップ 1 0 上の集中共有メモリ 2 8、及び、他のチップに設けられた集中共有メモリ 1 4 間でデータを転送する。

30

【 0 0 3 9 】

なお、ローカルデータメモリ 2 6 とデータ転送コントローラ 3 0 との間の破線は、シングルチップマルチプロセッサの用途に応じて、データ転送コントローラ 3 0 がローカルデータメモリ 2 6 にアクセスできるように構成してもよいことを示している。このような場合、C P U 2 0 が、転送指示を、ローカルデータメモリ 2 6 を介して、データ転送コントローラ 3 0 に与えることができる。また、C P U 2 0 が、転送終了後に転送されたデータをチェックすることができる。

【 0 0 4 0 】

C P U 2 0 は、ローカルデータメモリ 2 6、分散共有メモリ 2 2 又は専用のバッファ ( 図示省略 ) を介して、データ転送コントローラ 3 0 へデータ転送を指示する。また、データ転送コントローラ 3 0 は、ローカルデータメモリ 2 6、分散共有メモリ 2 2 又は専用のバッファ ( 図示省略 ) を介して、C P U 2 0 へデータ転送の終了を報告する。このとき、どのメモリ又はバッファを使うかはプロセッサの用途に応じて、プロセッサの設計時に決められる。又は、複数のハードウェア的な方法を用意し、プログラムの特性に応じて、コンパイラ又はユーザがソフトウェア的に使い分けられるようにしてもよい。

40

【 0 0 4 1 】

データ転送コントローラ 3 0 へのデータ転送指示 ( 例えば、何番地から何バイトのデータを、どこにストアし又はロードするか、及び、データ転送のモード ( 連続データ転送、

50

ストライド転送等)等)は、コンパイラが、データ転送命令をメモリ又は専用バッファに格納して、プログラムの実行時にはどのデータ転送命令を実行するかを示すのみを出すようにして、データ転送コントローラ30を駆動するためのオーバーヘッドを削減することが望ましい。

【0042】

ネットワークインタフェース(NI)32は、各マルチコアチップ10内のプロセッサコア16の間を通信可能にするために、チップ内結合網34に接続される。チップ内結合網34はチップ間結合網12に接続されている。プロセッサコア16は、チップ間結合網12によって、他のマルチコアチップ10内のプロセッサコア16と通信することができる。

10

【0043】

プロセッサコア16は、チップ内結合網34を介して、集中共有メモリ14に接続される。集中共有メモリ14は、チップ間結合網12に接続される。

【0044】

なお、ネットワークインタフェース32は、チップ内接続ネットワーク34を介さずに、チップ間結合網12とを直接接続することもできる。このような構成は、システム中の全プロセッサコア16が、各チップ上に分散して配置された集中共有メモリ28及び分散共有メモリ22に、平等なアクセスを可能にする。また、直結されたパスを設けることによって、チップ間のデータの転送量が多い場合でも、システム全体のデータ転送能力を高めることができる。

20

【0045】

電力制御レジスタ(FVR)36は、プロセッサコア16の動作周波数及び動作電圧を制御するために、プロセッサコア16に供給される電源電圧やクロック周波数が設定される。なお、図示したように、電力制御レジスタは、プロセッサコア16だけでなく、マルチコアチップ10、チップ間結合網12、集中共有メモリ14、入出力用チップ18、集中共有メモリ28及びチップ内結合網34にも設けられ、これらの各構成の動作周波数及び動作電圧を制御するためのデータが格納される。

【0046】

図2は、本発明の実施の形態のマルチグレイン並列処理を説明する図である。

【0047】

マルチグレイン並列処理とは、粗粒度並列性、中粒度並列性及び近細粒度並列性を階層的に利用する並列処理方式である。粗粒度並列性とは、サブルーチン間、ループ間及び基本ブロック間の並列性であり、中粒度並列性とは、ループのイタレーション間の並列性であり、近細粒度並列性とは、ステートメント間及び命令間の並列性である。このマルチグレイン並列処理によって、従来行われてきた局所的かつ単一粒度の並列化(ループの並列化及び命令レベルの並列化等)とは異なり、プログラム全域にわたるグローバルかつ複数粒度にわたるフレキシブルな並列処理が可能となる。

30

【0048】

マルチグレイン並列処理においては、以下の手順で並列化処理が行われる。

- 1) ソースプログラムからマクロタスクを生成。
- 2) マクロタスク間の制御フロー及びデータ依存を解析しマクロフローグラフを生成。
- 3) 最早実行可能条件解析によってマクロタスクグラフを生成。

40

【0049】

以下、この手順を具体的に説明する。

【0050】

単一プログラム中のサブルーチン、ループ、基本ブロック間の並列性を利用するマルチグレイン並列処理では、ソースとなる、例えばフォートランプログラムを、粗粒度タスク(マクロタスク)として、繰り返しブロック(RB: repetition block)、サブルーチンブロック(SB: subroutine block)、及び、疑似代入文ブロック(BPA: block of pseudo assignment statements)の3種類のマクロタスク(MT)に分解する。繰り返し

50

ブロックは、各階層での最も外側のループである。

【 0 0 5 1 】

また、疑似代入文ブロックは、スケジューリングオーバーヘッド及び並列性を考慮して、結合及び / 又は分割された基本ブロックである。ここで、疑似代入文ブロックは、基本的には通常の基本ブロックであるが、並列性抽出のために単一の基本ブロックを複数に分割してもよい。また、一つの疑似代入文ブロックの処理時間が短く、ダイナミックスケジューリング時のオーバーヘッドが無視できない場合には、複数の疑似代入文ブロックを結合して一つの疑似代入文ブロックを生成する。

【 0 0 5 2 】

最外側ループである繰り返しブロックが D o a l l ループである場合は、ループインデックスを分割することによって、複数の部分 D o a l l ループに分割し、分割された D o a l l ループを新たに繰り返しブロックと定義する。繰り返しブロックが n o n - D o a l l ループである場合は、繰り返しブロック内の並列性に、階層的マクロデータフロー処理を適用するとよい。

【 0 0 5 3 】

サブルーチンブロックは、可能な限りインライン展開するとよい。しかし、コード長を考慮した結果、効果的にインライン展開ができないサブルーチンは、そのままサブルーチンブロックとする。この場合、サブルーチンブロック内の並列性に、階層的マクロデータフロー処理を適用するとよい。

【 0 0 5 4 】

次に、マクロタスク間の制御フローとデータ依存を解析し、図 3 に示すようなマクロタスクグラフ ( M T G ) を作成する。マクロフローグラフでは、マクロタスク ( M T ) 間の制御フローを表している。マクロタスクグラフを作成する際は、マクロタスク間の制御依存及びデータ依存を同時に解析し、各マクロタスクが最も早く実行できる条件 ( 最早実行可能条件 ) の形でマクロタスク間の並列性を検出する。また、この最早実行開始条件をグラフで表現したものがマクロタスクグラフである。

【 0 0 5 5 】

そして、コンパイラは、マクロタスクグラフ上のマクロタスクを、プロセッサクラスタ ( コンパイラ又はユーザによって定義されるプロセッサのグループ ) へ割り当てる。このタスクの割り当てには、コンパイル時に割り当てるスタティックスケジューリングと、実行時に割り当てるダイナミックスケジューリングがある。ダイナミックスケジューリングの場合、ダイナミック C P アルゴリズムを用いてダイナミックスケジューリングコードを生成し、生成されたダイナミックスケジューリングコードをプログラム中に埋め込む。なお、ダイナミックスケジューリング時には、実行時までどのマクロタスクがどのプロセッサで実行されるか分からないので、マクロタスク間で共有されるデータは全プロセッサから等距離に見える集中共有メモリ 1 4 に割り当てるとよい。

【 0 0 5 6 】

マルチグレイン並列化では、マクロデータフロー処理によってプロセッサクラスタに割り当てられるループブロックは、そのループブロックが D o a l l ループ又は D o a c r o s s ループである場合、プロセッサクラスタ内の複数のプロセッサコア 1 6 によって処理がされるように、イタレーションレベルでループが分割され、ループが並列化される。

【 0 0 5 7 】

ループの再構築には、ステートメントの実行順序の変更、ループディストリビューション、ノードスプリッティングスカラー エクспанション、ループインターチェンジ、ループアンローリング、ストリップマイニング、アレイプライベートイゼーション、及び、ユニモジュラー変換 ( ループリバーサル、パーミュテーション、スキューニング等 ) 等の従来の技術がそのまま利用できる。

【 0 0 5 8 】

また、ループ並列処理が適用できないループには、近細粒度並列処理、又は、ループの

10

20

30

40

50

ボディ部を階層的にマクロタスクに分割する粗粒度タスク並列処理を適用する。

【0059】

プロセッサクラスタに割り当てられるマクロタスクが疑似代入文ブロックであるか、又は、ループ並列化も階層的なマクロデータフロー処理も適用できないループブロックの場合は、疑似代入文ブロック内のステートメント又は命令を近細粒度タスクとして、プロセッサクラスタ内のプロセッサで並列処理する。

【0060】

マルチプロセッサシステムでの近細粒度並列処理では、プロセッサ間の負荷バランスだけでなくプロセッサ間のデータ転送を最少にするように、近細粒度タスクをプロセッサにスケジューリングすることによって、効率よい並列処理を実現する。さらに、この近細粒度並列処理で要求されるスケジューリングでは、近細粒度タスク間にはデータ依存による実行順序の制約があるため、タスクの実行順序が問題となる。

10

【0061】

このようにして生成された近細粒度タスクグラフを各プロセッサにスタティックにスケジューリングする。この際、スケジューリングアルゴリズムとして、データ転送オーバーヘッドを考慮し実行時間を最小化するために、公知のヒューリスティックアルゴリズム（CP/D T/M I S F法、CP/E T F/M I S F法、E T F/C P法、又は、D T/C P法）を適用し最適なスケジュールを決定する。

【0062】

スケジューリングの終了後、コンパイラはプロセッサコアに割り当てられたタスクの命令列を順番に並べ、データ転送命令や同期命令を必要な箇所に挿入することによって、各プロセッサ用のマシンコードを生成する。このとき、挿入されるデータ転送命令は、マクロタスク間の制御依存及びデータ依存によって、ローカルメモリ26にデータを格納する及びローカルメモリ26からデータを掃き出すタイミングを決められる。

20

【0063】

近細粒度タスク間の同期にはバージョンナンバー法を用い、同期フラグの受信は受信側プロセッサコアのビジーウェイトによって行うとよい。ここで、データ転送指示及び同期フラグの設定は、送信側のプロセッサが受信側のプロセッサコア16上の分散共有メモリ22に直接書き込むことによって、低オーバーヘッドで行うことができる。

【0064】

<ローカルメモリ管理>

図4は、本発明の実施の形態のローカルメモリ管理の概要を説明する図である。

30

【0065】

プログラムは、一般に、サブルーチン及び多重ループによって複数の階層に分かれている。よって、プログラムの実行に必要なデータをどのタイミングで転送するかを考えることが重要である。例えば、ループにおいては、ループの前後でデータを転送するとよい。具体的には、ループの実行前にデータをローカルメモリに転送し、ループの実行後にデータをローカルメモリから転送する。そして、ループ内ではデータをローカルメモリに載せたまま、ループが実行できるようにデータを配置するとよい。このように、データをローカルメモリに載せたまま実行できるプログラムの単位を「ローカルメモリ管理マクロタスク」という。

40

【0066】

すなわち、ローカルメモリ管理マクロタスクで扱うデータは必ずローカルメモリ上に載るサイズのデータである。また、ローカルメモリ管理マクロタスクの実行に必要なデータは、ローカルメモリ管理マクロタスクの実行前又は実行後の適切なタイミングで転送（ロード、ストア）される。さらに、タスク内でデータの転送が発生しないように、ローカルメモリ管理マクロタスクを決定する。

【0067】

そして、必要な全てのデータをローカルメモリに載せることができないループは、そのループの中で、使用される全てのデータがローカルメモリに載る部分をローカルメモリ管

50

理マクロタスクと定義する。すなわち、必要な全てのデータがローカルメモリに載るようにプログラムを分割し、ローカルメモリ管理マクロタスクを決める。このようにローカルメモリ管理マクロタスクを決めて、ローカルメモリ管理マクロタスクの実行に必要なデータは、マクロタスクの実行前後で転送（ローカルメモリへのロード、ローカルメモリから集中共有メモリへのストア）する。このため、マクロタスク内において、データの転送が発生しない。

【0068】

以上、ループについて説明したが、プログラム中のベーシックブロック及びサブルーチンでも同じである。なお、サブルーチンについては、後述する例外がある。

【0069】

また、本明細書では、ローカルメモリの管理について説明するが、容量に制限のある（記憶容量が使用されるデータより少ない）メモリであれば、本発明を適用することができる。例えば、ローカルメモリ26の他、プロセッサコア16内の分散共有メモリ22、オンチップの集中共有メモリ28及びオフチップの集中共有メモリ14にも、本発明を適用することができる。

【0070】

以上説明したプログラムを分割して、ローカルメモリ管理マクロタスクを生成する方法について、図4を参照して説明する。

【0071】

ローカルメモリに配列変数の要素が1000個載ると仮定する。また、図4に示すように、このプログラムは変数*i*、*j*による2重ループが含まれている。ループ中で配列変数A[1:30, 1:20]は600要素が使用され、配列変数B[1:30]は30要素が使用され、配列変数C[1:30, 1:20]は600要素が使用される。合計すると、このループでは1230個の配列要素が使用される。よって、全てのデータをローカルメモリに載せて、このループを実行することができない。

【0072】

そこで、本発明の実施の形態のコンパイラは、変数*i*のループを、*i* = 1 ~ 10及び、*i* = 11 ~ 20の二つのループに分割する。すると、各ループでアクセスされるデータは630要素になるので、全てのデータをローカルメモリに載せたまま、ループを最初から最後まで実行することができる。この分割されたループが、ローカルメモリ管理マクロタスクである。そして、このマクロタスクの実行前後に必要なデータがロード及びストアされる。

【0073】

データのローカルメモリへの転送（ロード）はマクロタスクの実行直前でなくても、他のデータのブロックへの割り当てを考慮して、もっと前の時点で実行してもよい。このように、配列変数（データ）が使用されるマクロタスクの実行開始前までの任意のタイミングで、そのマクロタスクで使われる配列変数をメモリにロードすることを「プレロード」という。このプレロードは、他のマクロタスクの実行中であっても、そのデータがロードされるべきブロックが空いていれば、DMAによってデータの転送が可能である。このように、ブロックの空き状態によって、マクロタスクの実行前でもデータをロードすることができ、プログラム実行までに必要なデータを揃えることができる。このため、メモリに必要なデータがロードされていないことによるプロセッサの待ち時間を削減することができる。

【0074】

また、データのローカルメモリからの転送（ストア）はマクロタスクの実行終了直後でも、他のデータのブロックへの割り当てを考慮して、もっと後の時点で実行してもよい。このように、配列変数（データ）が使用されるマクロタスクの終了後の任意のタイミングで、そのマクロタスクで使われた配列変数をメモリにロードすることを「ポストストア」という。このポストストアは、他のマクロタスクの実行中であっても、DMAによってローカルメモリから集中共有メモリへのデータの転送が可能である。このように、任

10

20

30

40

50

意のタイミングでデータをストアすることによって、DMAの負荷がマクロタスクの実行前後に集中することを避けることができる。

【0075】

次に、前述したサブルーチンにおける例外について説明する。

【0076】

前述したように、一般的には、ローカルメモリ管理マクロタスク内でデータの転送が発生することはない。しかし、サブルーチンがローカルメモリ管理マクロタスクとなった場合、及び、内部でサブルーチン呼び出ししているループがローカルメモリ管理マクロタスクとなった場合の二つの場合には、サブルーチン内においてデータを転送（ロード及び/又はストア）する必要がある。

10

【0077】

具体的には、サブルーチンの開始時に、サブルーチン呼び出し元で使用していた配列変数をロードする。例えば、フォートランにおけるセーブ、コモン及びデータ変数、C言語におけるスタティック変数及びグローバル変数を使用すると、これらの変数は呼び出し元では管理することができない。よって、マクロタスク内のサブルーチンの処理が終わったら使用された変数を共有メモリに転送する必要がある。サブルーチン終了時に変数をローカルメモリから読み出して共有メモリに書き込まないと、データの整合性がとれないからである。なお、マクロタスクは一つの物理プロセッサによって実行されるので、マクロタスク内でデータをロード及びストアしても正しい値が保証される。

【0078】

< 記憶領域の分割 >

図5は、本発明の実施の形態のメモリの記憶領域の分割の状態を説明する図である。

20

【0079】

本発明で、記憶領域の管理の対象となるメモリは、ローカルメモリ及び分散共有メモリである。これらの管理対象のメモリの記憶領域は、複数のブロック（サブブロック等も含む）101～104に分割されている。

【0080】

図5に、ローカルメモリの記憶領域を分割したブロックを示す。ブロック101は、ローカルメモリの記憶領域が2のべき乗分の1（図示する状態では8分の1）に分割された固定長の領域である。更に、ブロック4～7は、ブロックサイズの半分のサブブロック8～15（102）に分割されている。さらに、サブブロック12～15は、サブブロックサイズの半分のサブサブブロック24～31（103）に分割されている。さらに、サブサブブロック28～31は、サブサブブロックサイズの半分のサブサブサブブロック56～63（104）に分割されている。

30

ブロック101、サブブロック102、サブサブブロック103、サブサブサブブロック104は独立して管理される。コンパイラは、メモリの各アドレス空間に任意のサイズのブロックを設定することができる。コンパイラは、コンパイルされるプログラムに応じて適切なサイズのブロック等を設定する。すなわち、大きいデータを扱うプログラムでは大きなサイズのブロックを、小さいデータを扱うプログラムでは小さなサイズのブロックを用意することによって、ローカルメモリを無駄なく、かつ効率よく使用することができる。

40

【0081】

なお、ブロックの管理を容易にするため、サブブロック等も、記憶領域の先頭アドレスから通し番号を付与する。このため、一つのアドレスによって示される領域が、複数のブロック、サブブロックに含まれる。例えば、ブロック0、サブブロック0～1、サブサブブロック0～3、及びサブサブサブブロック0～7は、同じメモリ空間（アドレス0～127）を示す。このように、複数種類のブロックをメモリ空間の同じアドレスに設定することによって、随時、適切な種類のブロック等を使用するようにメモリを管理することができる。

【0082】

50

このように設定された、サブブロックのサイズはブロックのサイズの1/2であり、サブサブブロックのサイズはブロックのサイズの1/4であり、サブサブサブブロックのサイズはブロックのサイズの1/8となるように分割されている。つまり、メモリの記憶領域は、サイズが2のべき乗の関係（隣接するサイズのブロックと2倍の関係）にある複数のサイズのブロックに分割され、分割された複数のサイズのブロックが記憶領域として提供される。

【0083】

なお、ブロックサイズは、コンパイラがプログラムをコンパイルする際に取得したプログラムの情報によって決定されるので、プログラム実行開始から終了までの間変更されない。しかし、コンパイラが、別のプログラムをコンパイルすると、コンパイル対象のプログラムの特性に適合するように、ブロックの数及びサイズは異なってくる。すなわち、本発明のローカルメモリの記憶領域の管理単位となるブロックは、完全に固定されたサイズではなく、コンパイラがプログラムを解析した情報に基づいて、プログラムで使用されるデータサイズに最適なブロックサイズを決めることができる。ブロックサイズはプログラム内では固定だが、プログラム毎に最適なブロックサイズを選択する。

10

【0084】

なお、プログラム実行中にブロックサイズを変えてもよい。具体的には、プログラムのあるステップまでは大きな配列データを使ったが、あるステップ以後は小さなスカラーデータ（1次元変数）を使うときは、配列データの使用終了時にブロックを分割してサブブロックにしてもよい。また、テンプレートの割り当てを変えることによって、ブロックに載せられる配列変数のサイズを変更することができる。テンプレートのブロックへの割り当ては後述する。

20

【0085】

本発明では、一つのローカルメモリ管理マクロタスクで使用される全てのデータが一つのブロックに格納できるように、ブロックサイズが決定される。換言すると、ブロックのサイズによって、ローカルメモリ管理マクロタスクのサイズが定まる。なお、定められたブロックサイズにデータを出し入れするタイミングは、その後、マクロタスクに実行タイミングに基づいて、スケジューラによって決められる。

【0086】

このようにして、ローカルメモリの記憶領域を複数のサイズのブロックに分割することによって、ローカルメモリ管理マクロタスクにおいて使用されるデータに最適なサイズのブロックに、必要なデータがロードされる。また、ローカルメモリの記憶領域を固定サイズの領域で管理することによって、可変サイズの領域で管理する場合に生じる断片化（フラグメンテーション）の問題を回避することができる。

30

【0087】

図4に示した分割したループの例では、配列変数Aはローカルメモリの一つのブロックに収まる。また、配列変数Bは一つのサブサブブロックに収まる。換言すれば、ブロックのサイズは、コンパイラによって、プログラムの性質に応じて決められる。より具体的には、配列変数Aがローカルメモリの一つのブロックに収まるようにブロックサイズが決められ、ループが分割される。そして、通常は、ブロックサイズはコンパイル対象のプログラムの開始時から終了時までには変更されない。

40

【0088】

<データのロード及びストア>

次に、ブロックへのデータの書き込み（ロード）、読み出し（ストア）、及び、ブロックの割り当てについて説明する。

【0089】

図5に示すように記憶領域が分割されたメモリには、ローカルメモリ管理マクロタスクにおいて使用されるデータがロードされる。まず、スケジューラは、データをロードするメモリが、ローカルメモリか分散共有メモリかを定める。このとき、複数のプロセッサコアによって共有されるデータは分散共有メモリにロードするとよい。

50

## 【 0 0 9 0 】

次に、スケジューラは、必要なデータが既にロードされているブロックがある場合、そのブロックにロードされているデータをそのまま使用する。一方、必要なデータがいずれのブロックにもロードされていない場合、空いているブロックを当該ロードされるデータに割り当て、割り当てられたブロックに必要なデータをロードする。さらに、空いているブロックがなければ、掃き出し優先度の最も高いデータをローカルメモリ 2 6 から読み出して、集中共有メモリ 2 8 又は 1 4 に書き込み、空きブロックとなった記憶領域に必要なデータをロードする。

## 【 0 0 9 1 】

図 6 は、本発明の実施の形態のメモリの掃き出し優先度を説明する図であり、横軸に時間の経過を示す。 10

## 【 0 0 9 2 】

本発明の実施の形態では、掃き出し優先度は以下の順で決められる。

- 1 ) 以後アクセスされないデータ。
- 2 ) 他プロセッサでアクセスされるが、自プロセッサではアクセスされないデータ。
- 3 ) 再度自プロセッサで使用されるが、先の時間で使われるデータ。
- 4 ) 自プロセッサですぐに使用されるデータ。

## 【 0 0 9 3 】

以後アクセスされないデータは、例えば、新たに再計算されてしまっている変数である。このような既に死んでいる変数は残しておく必要がないため、掃き出し優先度は一番高くなる。他のプロセッサでアクセスされるが、今後自プロセッサでアクセスされないデータは、そのデータを必要とするプロセッサコア 1 6 の分散共有メモリ 2 2 に転送するとよいので、次に掃き出し優先度が高くなる。他プロセッサでアクセスされるデータは直ぐに分散共有メモリ 2 2 へ転送すればよいが、他のプロセッサコア 1 6 のメモリの状況によって、すぐに転送できない場合には、少し時間をずらしてから分散共有メモリ 2 2 へ転送するか、集中共有メモリ 2 8 又は 1 4 へ転送する。このようにローカルメモリ 2 6 からのデータの転送タイミングに自由度を持たせるために、アクセスされないデータより掃き出し優先度を低く設定している。 20

## 【 0 0 9 4 】

最後に、再び自プロセッサでアクセスされるデータについては、次に使用されるまでの時間によって優先度を定める。そのデータが使用される時間が先である程、掃き出し優先度は高く、すぐ近くで使用されるデータは掃き出し優先度を低くし、なるべくメモリに載ったまま残るようにする。 30

## 【 0 0 9 5 】

図 6 を参照して、時間の経過と共にメモリの掃き出し優先度について説明する。図 6 では、現在、プロセッサコア 0 ( P C 0 ) で実行されているローカルメモリ管理マクロタスクにおいて、配列変数 A、B、C 及び D がローカルメモリにロードされている状態を考える ( 1 0 0 1 )。

## 【 0 0 9 6 】

このローカルメモリ管理マクロタスクでは、始め、プロセッサコア 0 ( P C 0 ) で、配列変数 A を定義し ( D e f A )、配列変数 A を使用している ( 1 0 0 2 )。 40

## 【 0 0 9 7 】

次に、プロセッサコア 1 ( P C 1 ) で、別の配列変数 A を定義した ( 1 0 0 3 )。すると、P C 0 のローカルメモリにロードされている配列変数 A は既に変更されているので、今後アクセスされることはない。よって、掃き出し優先度が最高位になる。キャッシュのコヒーレンス制御と同様に、整合性がとれないデータは、不要なデータと判断すればよいからである。

## 【 0 0 9 8 】

次に、後に実行されるべきマクロタスクを検討する。配列変数 B は、他のプロセッサ ( P C 1 ) で使用される ( 1 0 0 4 )。配列変数 C 及び D は、自プロセッサ ( P C 0 ) で使 50



用される(1005、1006)。よって、配列変数Bの掃き出し優先度は、配列変数C及びDの掃き出し優先度より高くなる。

【0099】

配列変数Cと配列変数Dを比較すると、配列変数Dの方が配列変数Cより先に使用される。よって、配列変数Cの掃き出し優先度は、配列変数Dの掃き出し優先度より高くなる。なお、配列変数C及びDは、後に利用されるので、集中共有メモリCSMに一旦書き戻して、次に必要になるときにロードすればよい。

【0100】

このため、掃き出し優先度はA、B、C、Dの順となる。

【0101】

図7、本発明の実施の形態のメモリにロードされている変数の推移を説明する図である。

【0102】

図7は、一つのプロセッサコア上で、二つのローカルメモリ管理マクロタスク(MT1、MT2)が含まれるプログラムが一つの階層で実行される場合に、プログラムの実行開始前のメモリの状態を、マクロタスク1(MT1)の実行終了時のメモリの状態、マクロタスク2(MT2)の実行開始時のメモリの状態、及び、マクロタスク2(MT2)の実行終了時のメモリの状態を示す。なお、ローカルメモリの記憶領域は、図5で示したように分割されている。

【0103】

マクロタスク1(MT1)の実行開始前には、全てのメモリの記憶領域(ブロック)は、空き状態である(1011)。そして、マクロタスク1の開始時には、マクロタスク1で必要とされるデータ(配列変数A、B、C、D、E、F)にブロックが割り当てられ、各配列変数がメモリにロードされる。その後、マクロタスク1による処理が開始する。具体的には、宣言文def Aによって、配列変数Aがブロック0に割り当てられる。同様に、配列変数Bがブロック1に割り当てられ、配列変数Cがブロック2に割り当てられ、配列変数Dがブロック3に割り当てられ、配列変数Eサブブロック8に割り当てられ、配列変数Fがサブブロック9に割り当てられる。

【0104】

マクロタスク1の実行終了時には、各配列変数がブロックにロードされている(1012)。

【0105】

マクロタスク2(MT2)の実行開始時には、マクロタスク2で使用される全てのデータがメモリにロードされている必要がある。マクロタスク2では、配列変数A、B、C、G、E、H、I及びJが使用されるので、4個のブロック及び4個のサブブロックが必要である。必要な配列変数のうち、配列変数A、B、C及びEは、既にローカルメモリにロードされているので、マクロタスク2の実行時に新たにロードすべき配列変数は、配列変数G、H、I及びJである。このうち、配列変数Hは、マクロタスク2の実行開始前にサブブロック10にプレロードされている。また、サブブロック11は空いている。よって、マクロタスク2の実行開始の段階で1個のブロック及び1個のサブブロックを空ける必要がある。

【0106】

そこで、必要なデータをロードするブロックを確保するために、掃き出し優先度に従って配列変数Dをブロック3から掃き出し、配列変数Fをサブブロック9から掃き出す。これによって、1個の空きブロック及び1個の空きサブブロックを確保する(1014)。

【0107】

このため、マクロタスク1の実行終了後、マクロタスク2の実行前には、配列変数D及びFの集中共有メモリへの転送、配列変数Hのローカルメモリへの転送が必要となる(1013)。

【0108】

10

20

30

40

50

マクロタスク 2 の実行開始前に、配列変数 G がブロック 3 に割り当てられ、配列変数 I がサブブロック 9 に割り当てられ、配列変数 J がサブブロック 11 に割り当てられる。その後、マクロタスク 2 が実行され、配列変数 G、I 及び J がマクロタスク 2 で使用される (1014)。

【0109】

このように、前述した優先度によって、メモリからデータを分散共有メモリ又は集中共有メモリにストアするので、従来の LRU とは異なり、メモリの利用を最適化することができ、メモリ間のデータ転送を減らすことができる。すなわち、従来の LRU によると、最近使われていないが直ぐに使われるかもしれないデータもメモリから転送されてしまう。しかし、本発明のように、コンパイラが取得した情報によると、そのデータが次に使用されるタイミングが分かり、メモリを最適に使用することができる。

10

【0110】

<ループの分割>

次に、図 8 から図 11 を参照して、ループの分割手順の具体例を説明する。

【0111】

ループの分割は、複数のループを見て整合分割を行う。多重化されたループで一番広くアクセスする範囲を解析して、グローバルインデックスレンジとする。すなわち、ローカルメモリをアクセスすればよい範囲と、隣のプロセッサと通信をしなければいけない範囲とがあるので、データのアクセス範囲を解析し、これを切り分ける。このため、プログラムの構造を解析し、ターゲットループグループ (TLG) を選択する。本実施形態では、従来のコンパイラによる並列的なループを連続的に実行するための解析と異なり、複数のループにわたってどのようにメモリがアクセスされるかを解析する。

20

【0112】

ここで、二つのループが整合するとは、以下の全ての条件を満たすことである。

- 1) 各ループが、Doallループ、Reductionループ、ループキャリッドデータ依存(リカレンス)によるSequentialループのいずれかである。
- 2) ループ間に配列変数のデータ依存が存在する。
- 3) 各ループのループ制御変数が同一配列の同じ次元の添字式で使用されており、次元の配列添字がループ制御変数の一次式で表されている。
- 4) ループ間にデータ依存を導く各配列に対して、配列添字中のループ制御変数係数のループ間での比が一定である。

30

【0113】

このとき、選択されていない単一のループも全てターゲットループグループとし、ターゲットループグループの入れ子を許容し、間接参照を含むループも選択する。すなわち、ターゲットループグループに選ばれたループの内側にもループが存在していた場合、内側のループに対してもターゲットループグループを生成する。また、他のループと整合可能でないループは、そのループのみでターゲットループグループを成す。

【0114】

ターゲットループグループとは、マクロタスク上でループ整合分割が適用可能な繰り返しブロック (RB) の集合であり、マクロタスクグラフ上で直接データ依存先行、後続関係を持つ繰り返しブロックの集合である。これは、実行時のコスト (メモリ及びプロセッサ等のリソースの消費) が大きい繰り返しブロックとその繰り返しブロックに直接データ依存先行、後続関係を持つ繰り返しブロックは大きなデータを扱うので、分割の効果が高いためである。これによって、ループ間で同じデータを使う場合に、同じ領域を使い回して、キャッシュミスを防止することができる。

40

【0115】

具体的に、図 8 に示すプログラムでは、変数 i による二つのループが TLG1 となり、各 TLG1 内の変数 j によるループが TLG1-1 及び TLG1-2 となる。さらに、TLG1-1 内の変数 k によるループが TLG1-1-1 なり、TLG1-2 内の変数 k によるループが TLG1-2-1 なる。

50

## 【 0 1 1 6 】

次に、図 9 に示すように、T L G 集合を生成する。T L G が以下の生成条件の全てを満たす場合に、T L G 集合が生成される。

- 1) 少なくとも一つ以上共有配列を持つ (依存関係も考慮される)。
- 2) 共有配列の整合次元が全て一致する。
- 3) サブルーチンを跨った場合は、共有配列の形状が一致する。

## 【 0 1 1 7 】

次に、図 1 0 に示すように、分割候補 T L G 集合を生成する。これは、入れ子になった T L G がある場合、コストが最大の T L G 集合を選択し、選択された T L G 集合を分割候補とする。その後、分割候補 T L G 集合毎に G I R を計算する。このようにすると、プログラム中の全てのコードをカバーすることができる。具体的には、T L G 集合 1 が分割候補 T L G 集合 1 となり、G I R は [ 1 : 1 0 ] となる。

10

## 【 0 1 1 8 】

次に、分割基準領域を決定する。分割基準領域は、各 T L G 集合で使用されるデータを収めなければならない記憶領域である。具体的には、分割候補 T L G 集合で使用されるデータサイズの比を計算する。ここで、3 個の分割候補 T L G 集合があり、集合 1 で使用されるデータサイズが 3 0 0 k、集合 2 で使用されるデータサイズが 2 0 0 k、集合 3 で使用されるデータサイズが 1 0 0 k であれば、データサイズの比は 3 : 2 : 1 になる。

## 【 0 1 1 9 】

この比に基づいて、最小メモリ領域 (ローカルメモリと分散共有メモリとのうち、容量が小さいもの) を各分割候補 T L G 集合に割り当てる。具体的には、分割候補 T L G 集合で使用されるデータが最小メモリ領域よりも 小さい領域 に収まるように、分割候補 T L G 集合を分割する。なお、実際には、この時に割り当てられた領域以外の領域にも分割候補 T L G 集合で使用されるデータを載せることができるが、分割用の目安としてこのような処理をする。

20

## 【 0 1 2 0 】

これによって、分割後の各分割候補 T L G 集合で使用されるデータを同時にローカルメモリに載せることが可能となる。なお、実際にデータをメモリに載せるかは、スケジューリング及びメモリ管理ルーチンによって決まる。

## 【 0 1 2 1 】

次に、ブロックサイズを決定する。

30

## 【 0 1 2 2 】

まず、コストが最も大きい分割候補 T L G 集合を、ブロックサイズを決定する基準にする。但し、多重分割が必要となった場合、分割候補 T L G 集合には最大分割数が採用される。ここで、最大分割数とは、割り当てられるプロセッサグループ ( P G ) 内のプロセッサコア ( P C ) の構成と、並列処理によるオーバーヘッドを考慮した場合の最大の分割数である。C P U の数が一つである場合、最大分割数は、ループのイタレーション数である。以後、多重分割が必要となった T L G 集合は、分割候補 T L G 集合に選ばない。そして、再び、分割候補 T L G 集合を生成する。

## 【 0 1 2 3 】

具体的には、図 1 1 に示すように、T L G 集合 1 は最大分割数で分割されるので、次の T L G 集合 2 が分割候補 T L G に選択される。分割候補となった T L G 集合 2 は、G I R は [ 1 : 2 0 ] である。

40

## 【 0 1 2 4 】

次に、ブロックサイズの決定手順について説明する。

## 【 0 1 2 5 】

まず、ブロックサイズ決定前処理を実行する。基準となる分割候補 T L G 集合でアクセスするデータが、分割基準領域よりも小さいサイズになる分割数を計算する。ここではアクセスされるデータのサイズのみを考え、間接的に参照されるデータのアクセスについては考慮しない。分割数は、プロセッサグループの数の整数倍となるように選択する。求め

50

られた分割数で T L G 集合内の各ループの分割を試行する。

【 0 1 2 6 】

具体的には、最外ループ ( i のループ ) で 4 分割を試行する。アクセスされる配列変数 A 及び B のサイズは、 $[ k , j , i ] = [ 1 : 3 0 , 1 : 2 0 , 1 : 3 ]$  である。総データサイズは、 $3 0 \times 2 0 \times 3 \times 2 = 3 6 0 0$  になる。

【 0 1 2 7 】

次に、ブロックサイズを決定する。

【 0 1 2 8 】

分割後の配列アクセス範囲に基づいて、T L G 集合中の全てのローカル配列のテンプレートを作成し、作成されたテンプレートの大きさを仮ブロックサイズとする。テンプレートの作成の詳細は後述する。ここでも、間接参照される配列データのアクセスについて考慮しない。テンプレートの作成に失敗した場合、分割数を大きくして、ブロックサイズ決定前処理からやり直す。

10

【 0 1 2 9 】

決定された仮ブロックサイズを用いて、分割基準領域に割り当てできるかを判定する。このステップでは、データを間接的に参照している場合は、その次元については配列の宣言サイズを用いて判定する。間接的に参照されている配列がブロックに収まらなかった場合、共有メモリに置くことを決定してもよい。

【 0 1 3 0 】

T L G 集合で使用されるデータが仮ブロックサイズに割り当てできる場合、テンプレートを作成したときの分割数を採用する ( すなわち、さらに分割はしない )。そして、仮ブロックサイズをブロックサイズと決定する。

20

【 0 1 3 1 】

一方、T L G 集合で使用されるデータが仮ブロックサイズに割り当てできない場合、分割数を大きくして、ブロックサイズ決定前処理からやり直す。さらに、最大分割数でも割り当てできなかった場合、最大分割数を採用し、以後、割り当てできなかった T L G 集合は、分割候補 T L G 集合に選ばない。そして、再び、分割候補 T L G 集合を生成する。

【 0 1 3 2 】

次に、分割数を決定する。

【 0 1 3 3 】

分割候補 T L G 集合毎に、先に決められたブロックサイズに基づいて、分割基準領域決定で決めた領域中にいくつのブロックが確保できるか計算し、割り当てできる分割数を求める。その結果、いずれかの分割候補 T L G 集合の割り当てに失敗した場合、再び、分割候補 T L G 集合を生成する。

30

【 0 1 3 4 】

その際、割り当てできなかった分割候補 T L G 集合は最大分割数を採用し、以後、割り当てできなかった T L G 集合は、分割候補 T L G 集合に選ばず、ブロックサイズは再計算しない。既に、分割数が決められた分割候補 T L G 集合のデータサイズをローカルメモリのサイズから減じて、まだ残っている分割候補 T L G 集合のデータサイズの比に基づいて、再度、T L G 集合を割り当てる。そして、分割数が決定した分割候補 T L G 集合内にある各ループを。ローカルメモリ管理マクロタスク候補とする。

40

【 0 1 3 5 】

具体的には、T L G 集合 1 で使用されるデータサイズが 3 0 0 k、T L G 集合 2 で使用されるデータサイズが 2 0 0 k、T L G 集合 3 で使用されるデータサイズが 1 0 0 k である例を考える。T L G 集合 1 を基準にブロックサイズが決定できたとする。T L G 集合 2 を最大分割数でも割り当てに失敗した場合、T L G 集合 2 の中に T L G 集合 4、T L G 集合 5 及び T L G 集合 6 があつた場合、これらが次の分割候補 T L G 集合に加わる。

【 0 1 3 6 】

全てのデータがローカルメモリに配置できるループの分割数が決まった場合、どのデータがどのようなパターンでアクセスされるか、及び、どのデータがどの大きさのブロック

50

等を使用すると仮定して分割したかの情報を、データをローカルメモリに割り当てるときのために、記憶しておく。

【0137】

次に、マクロタスクの分割、ループの再構築を行う。ここで行われる処理は、マクロタスクの分割、ループディストリビューション、ループフュージョン及びループの再構築である。

【0138】

ここで、ループディストリビューション (Loop distribution) とは、具体的には図 14 及び図 15 で後述するが、多重分割を行った際にデータを使い回すための処理である。ループフュージョン (Loop fusion) とは、レジスタを使い回すための処理である。

10

【0139】

そして、ループの再構築 (Loop restructuring) 後、外側階層のループから順に分割数が設定されているマクロタスクを探し、見つかったマクロタスクをローカルメモリ管理マクロタスクとする。ローカルメモリ管理マクロタスクに設定されたマクロタスクの内側では、マクロタスクの探索を行わない。

【0140】

<テンプレートの作成>

次に、テンプレートの作成手順について説明する。

【0141】

本実施の形態において、テンプレートとは、配列変数をローカルメモリに割り当てる単位である。コンパイラは、プログラムがデータにアクセスするパターンに応じてテンプレートを準備する。提供されるテンプレートのサイズは、ブロック又はサブブロックのサイズと同じである。また、テンプレートは、次元毎 (1次元配列、2次元配列、3次元配列、・・・) に用意され、プログラムによってアクセスされるサイズ以上の大きさである。

20

【0142】

ブロックを用いてローカルメモリを管理する場合に、同一アドレス領域のブロックに様々なデータ (形状、次元が異なる配列等) を載せる必要がある。すなわち、データのサイズがブロックに収まるものであっても、1次元で宣言されているスカラー変数であったり、2次元、3次元の配列変数である場合がある。また、同じ次元のデータであっても各次元のサイズが異なるデータである場合もある。これらと同じアドレス空間のブロックに載せるために、すべてのデータを1次元のデータに変換してメモリアドレスと一致させることもできる。しかし、配列変数の場合、添字変換が必要となり、ユーザーが書いたプログラムと異なるものになってしまう。このような添字変換をすると、プログラムが分かりにくくなり、デバッグも困難になり、並列性の解析が分かりにくくなる。

30

【0143】

そこで、プログラムの可読性を保ったまま、ローカルメモリを管理するためにブロック等のサイズと同じサイズのテンプレートを利用する。テンプレートとは、配列変数が格納されるテンプレートである。テンプレートに必要なデータを格納することによって、ローカルメモリ上の任意のブロックにデータを載せることを実現する。

【0144】

テンプレートは以下の手順によって作成される。

40

【0145】

まず、TLG集合内の全ての配列について、各次元のアクセスサイズよりも大きく、かつ最も小さい2のべき乗の数を求める。そして、各次元が求められた大きさの仮テンプレートを作成する。

【0146】

前述した例では、最外の変数  $i$  のループで4分割を試行する。例えば、 $i = 1 \sim 3$ 、 $4 \sim 6$ 、 $7 \sim 8$ 、 $9 \sim 10$  の4個のループに分割すれば、3回転の2個のループ、2回転の2個のループができる。分割されたループ内での配列変数  $A$  及び  $B$  のアクセスサイズは、共に、 $[k, j, i] = [1:30, 1:20, 1:3]$  である。なお、3次元目はルー

50

プの回転数のうち大きい方を選択し、3回転とする。

【0147】

次に、仮テンプレートの大きさを計算する。テンプレートの各次元は配列変数の各次元のサイズより大きな2のべき乗の数としているので、テンプレートサイズは、 $32 \times 32 \times 4 = 4k$ 要素となる。そして、最も大きい仮テンプレートのサイズをブロックサイズとする。前述した例では、ブロックサイズは $4k$ 要素となる。

【0148】

その後、分割基準領域サイズをブロックサイズで除算し商を求める(分割基準領域サイズ/ブロックサイズ)。この求められた商が1以上である場合は、除算に依って求められた商の小数点以下を切り捨てることによって、用意できるブロックの数(Block\_num)を求める。一方、求められた商が1未満である場合は、このテンプレートサイズでは一つもブロックが作成できないので、テンプレートの作成が失敗したと判断する。

10

【0149】

また、ブロックサイズを各テンプレートのサイズで除算し(ブロックサイズ/各テンプレートのサイズ)、その商をサブブロックの数とする。

【0150】

前述した例では、分割されるローカルメモリの領域が $12k$ 要素分なので、用意できるブロック数は、 $12k / 4k = 3$ 個と求まる。最終的に、 $4k$ 要素のサイズの3個のテンプレート[1:32, 1:32, 1:4, 0:2]が用意される。テンプレートサイズ及びテンプレート数が決まったので、ループの分割数は4で確定する。

20

【0151】

すなわち、この処理では、プログラム全体を見て、最適なブロックサイズを決定する。このため、マクロタスクで使用されるデータ(ワーキングセット)をローカルメモリに載せるためのプログラムの分割数を定める。そして、分割されたデータサイズより大きなブロックサイズとなるように、分割数を選択する。

【0152】

<ローカルメモリ管理マクロタスクの決定の例1>

次に、図12~図13を参照して、ブロックサイズの決定手順の別な具体例について説明する。

【0153】

ブロックサイズを決定するためには、まず、ループ内でアクセスされるデータを解析して、2のべき乗の大きさのテンプレートを作る。

30

【0154】

この例では、ローカルメモリのサイズ(フラグ領域除く)が $2k$ B、各配列要素のサイズは $4$ B/要素と仮定し、分散共有メモリの存在は考えない。

【0155】

まず、最外ループを最大分割数で分割した場合を考える。

【0156】

図12に示すように、変数*i*による最外ループ1020、ループ1020の中に変数*j*による内側ループ1021、及び、ループ1021の中に変数*k*による最内ループ1022がある、3重ループ構造となっている。

40

【0157】

具体的には、ループの分割を考えない場合、最内ループ1022で使用される配列変数A、B及びCのサイズは、 $[k, j, i] = [1:10, 1:10, 1:10]$ である。前述した手順によって仮テンプレートを作成する。作成される仮テンプレートのサイズは、 $[k, j, i] = [1:16, 1:16, 1:16]$ となる。この仮テンプレートに必要なブロックサイズは $16 \times 16 \times 16 \times 4 = 16k$ Bである。ローカルメモリのサイズは $2k$ Bなので、一つもブロックを用意できない。そこで、最外ループ1020を分割することを考える。

【0158】

50

図13に示すように、最外ループ(変数*i*)1020を最大分割数で分割(10分割)する。最大分割数は、ループのイタレーション数である。なお、外側のループは限界まで分割しても必要なブロック数が確保できない場合に、内側のループを分割することが望ましい。

【0159】

この場合、最内ループ1022で使用される配列変数A、B及びCのサイズは、 $[k, j, i] = [1:10, 1:10, 1:1]$ である。前述した手順によって作成される仮テンプレートのサイズは、 $[k, j, i] = [1:16, 1:16, 1:1]$ となる。この仮テンプレートに必要なブロックサイズは $16 \times 16 \times 1 \times 4 = 1k B$ である。ローカルメモリのサイズは $2k B$ なので、用意できるブロック数は、分割基準領域サイズ( $2k B$ ) / ブロックサイズ( $1k B$ ) = 2個となる。このループでは3個の配列変数を使用するので、この状態では必要なブロックが確保できない。そこで、次に、内側ループ1021を分割することを考える。

10

【0160】

内側ループ1021を2分割(2等分)した場合、最内ループ1022で使用される配列変数A、B及びCのサイズは、 $[k, j, i] = [1:10, 1:5, 1:1]$ である。前述した手順によって作成される仮テンプレートのサイズは、 $[k, j, i] = [1:16, 1:8, 1:1]$ となる。この仮テンプレートに必要なブロックサイズは $16 \times 8 \times 1 \times 4 = 512 B$ である。ローカルメモリのサイズは $2k B$ なので、分割基準領域サイズ( $2k B$ ) / ブロックサイズ( $512 B$ )によって用意できるブロック数を求めると、

20

【0161】

よって、このループで使用される3個の配列変数が割り当てられるブロックが確保できるので、ブロックのサイズ及び個数が決定する。作成されるテンプレートは、 $[1:16, 1:8, 1:1, 0:3]$ となる。

【0162】

そして、内側ループ1021を*j* = 1:5と、*j* = 6:10とに分けたループが、ローカルメモリ管理マクロタスクとなる。

【0163】

ループ1030、1031も、同様に分割する。

30

【0164】

このように、外側ループの分割に失敗した場合、内側(他の次元)のループでも分割(多次元分割)することによって、ローカルメモリのサイズに適したローカルメモリ管理マクロタスクを決定することができる。

【0165】

<ローカルメモリ管理マクロタスクの決定の例2>

次に、図14から図15を参照して、ローカルメモリ管理マクロタスクの生成の別な具体例を説明する。

【0166】

この例でも、前述の例と同様に、ローカルメモリのサイズ(フラグ領域除く)が $2k B$ 、各配列要素のサイズは $4 B$  / 要素と仮定し、分散共有メモリの存在は考えない。

40

【0167】

このプログラムは、図14に示すように、変数*i*、*j*、*k*による3重ループを有する。変数*i*による最外ループ1041内に、内側ループ1042(*j* = 1:10)及び内側ループ1043(*j* = 11:20)が含まれている。同様に、変数*i*による最外ループ1051内に、内側ループ1052(*j* = 1:10)及び内側ループ1053(*j* = 11:20)が含まれている。

【0168】

ループ1051は、ループ1041の後に実行される、また、ループ1042とループ1052とは同じデータ(配列変数A  $[1:30, 1:10, 1:1]$  及び配列変数B  $[$

50

1 : 3 0 , 1 : 1 0 , 1 : 1 ] ) を使用し、ループ 1 0 4 3 とループ 1 0 5 3 とは同じデータ ( 配列変数 A [ 1 : 3 0 , 1 1 : 2 0 , 1 : 1 ] 及び配列変数 B [ 1 : 3 0 , 1 1 : 2 0 , 1 : 1 ] ) を使用する。

【 0 1 6 9 】

しかし、このプログラムでは、ループ 1 0 4 2、ループ 1 0 4 3、ループ 1 0 5 2、ループ 1 0 5 3 の順に実行されるため、同じデータをローカルメモリ上に載せたまま使い回すことができない。そこで、図 1 5 に示すように、最外ループ 1 0 4 1 を、内側ループ 1 0 4 2 ( j = 1 : 1 0 ) と内側ループ 1 0 4 3 ( j = 1 1 : 2 0 ) とに分割する。同様に、最外ループ 1 0 5 1 を、内側ループ 1 0 5 2 ( j = 1 : 1 0 ) と内側ループ 1 0 5 3 ( j = 1 1 : 2 0 ) とに分割する。

10

【 0 1 7 0 】

この分割によって、分割された最外ループ 1 0 4 1 A と最外ループ 1 0 5 1 A とを順に実行することができる ( すなわち、内側ループ 1 0 4 2 と内側ループ 1 0 5 2 とが順に実行される ) 。このため、ループ 1 0 4 2 で使用した配列データを、そのまま、ループ 1 0 5 2 で使用することができる。つまり、ループ 1 0 4 1 A の実行終了時とループ 1 0 5 1 A の実行開始時との間でデータ ( 配列変数 ) の転送が発生しない。

【 0 1 7 1 】

同様に、分割された最外ループ 1 0 4 1 B と最外ループ 1 0 5 1 B とを順に実行することができる ( すなわち、内側ループ 1 0 4 3 と内側ループ 1 0 5 3 とが順に実行される ) 。このため、ループ 1 0 4 3 で使用した配列データを、そのまま、ループ 1 0 5 3 で使用することができる。つまり、ループ 1 0 4 1 B の実行終了時とループ 1 0 5 1 B の実行開始時との間でデータ ( 配列変数 ) の転送が発生しない。

20

【 0 1 7 2 】

このように、プログラムの実行順序と、そのプログラムで使用されるデータとが整合しない場合、ループのディスクリプションを実行して、同じデータを扱うループを連続的に実行するようにする。これによって、ループの実行時にデータの転送が発生しないようにすることができる。

【 0 1 7 3 】

< テンプレートの作成手順の例 >

図 1 6 は、本発明の実施の形態のテンプレートの作成手順の例を示す。

30

【 0 1 7 4 】

テンプレートは、ローカルメモリをマクロタスク上で扱うために、ローカルメモリに配列変数を割り当てる単位である。

【 0 1 7 5 】

テンプレートは、1 次元配列、2 次元配列、3 次元配列・・・等が用意され、その形はマクロタスクで使用される配列変数によって様々である。例えば、2 次元配列を考えると、配列変数の各添字の最大値が等しい正方形や、添字の最大値が異なる長方形 ( 縦長、横長 ) が、マクロタスクで使用される配列変数の大きさに合うように用意される。

【 0 1 7 6 】

テンプレートのサイズは、ローカルメモリ管理マクロタスクで使用されるデータのサイズより大きくなる。さらに、テンプレートの各次元の添字の最大値は、ローカルメモリ管理マクロタスクで使用される配列変数の各次元の添字の最大値より大きく、かつ最も小さい 2 のべき乗の数が選択される。このため、テンプレートは、その形が変わっても、その大きさはブロック及びサブブロック等のいずれかのサイズと等しい。

40

【 0 1 7 7 】

よって、テンプレートのサイズは、ブロックサイズと等しい又はブロックサイズの 2 のべき乗分の 1 となる。これによって、データが収まる最小の大きさのテンプレートを作り、この作られたテンプレートにデータが収まるようにプログラム ( ループ ) を分割する。そして、配列変数をローカルメモリに割り当てる際に、同じサイズのブロック等に割り当てることができ、ローカルメモリの記憶容量を無駄なく使用することができる。

50



## 【 0 1 7 8 】

割り当てられたテンプレートを使用することで、ブロック 0 に割り当てられたテンプレートは、ブロック 0 のメモリ空間を使用し、ブロック 1 に割り当てられたテンプレートは、ブロック 1 のメモリ空間を使用する。

## 【 0 1 7 9 】

テンプレートは同じ形状のものをブロック数分用意する。そしてブロック番号によって使用するテンプレート配列を変える。そのために、用意されるテンプレートは同じテンプレートを複数個並べた形（データを載せて使用する配列変数の次元 + 1 次元）が実際に作成されるテンプレートとなる。新たに作られた次元はブロック指定用の次元となり、要素数はブロックの数となる。

10

## 【 0 1 8 0 】

すなわち、テンプレートの次元は、配列変数の次元より 1 次元大きくなっている。これは、テンプレートの追加された次元の添字の値によって、複数の配列変数を切り替え、アクセスするブロックを変えるためである。なお、各テンプレートは、ローカルメモリの異なるブロック（異なるアドレス）に割り当てられる。配列変数の形状及び大きさが同じである場合、同じテンプレートを使用することができる。例えば、均等に分割されたループは、同じ形状及び大きさの配列変数を使用することから、このようなテンプレートを用意することが有効である。

## 【 0 1 8 1 】

例えば、ブロック数が 8 個であり、その各々のサイズが [ 1 : 2 , 1 : 1 6 , 1 : 4 ] である場合、tempA [ 1 : 2 , 1 : 1 6 , 1 : 4 , 0 : 7 ] のテンプレートをローカルメモリに割り当てる。なお、このとき、プログラム中に表れる配列変数が 5 個であった場合は、tempA [ 1 : 2 , 1 : 1 6 , 1 : 4 , 0 : 4 ] として、ブロック 0 から 4 のみにテンプレートを割り当てる。他のブロックは、更に分割してサブブロックとして利用してもよい。

20

## 【 0 1 8 2 】

図 1 7 に、テンプレートがマッピングされたローカルメモリの状態を示す。

## 【 0 1 8 3 】

テンプレートのマッピングには、例えば FORTRAN では、EQUIVALENCE 文が用いられる。具体的には、EQUIVALENCE(LM(1), tempA(1, 1, 1, 0)) と宣言することによって、ローカルメモリのブロック 0 に、テンプレート A を割り当てることができる。

30

## 【 0 1 8 4 】

テンプレート A は、3 次元配列用のテンプレートで、各次元は tempA [ 1 : 2 , 1 : 1 6 , 1 : 4 , 0 : 7 ] である。よって、ブロック 0（アドレス 0 ~ 1 2 7）には tempA [ 1 , 1 , 1 , 0 ] が割り当てられ、ブロック 1（アドレス 1 2 8 ~ 2 5 5）には tempA [ 1 , 1 , 1 , 1 ] が割り当てられる。

## 【 0 1 8 5 】

すなわち、前述したように、テンプレートの最外側の 4 次元目はテンプレート自体の次元ではなく、テンプレートが割り当てられるブロック番号を示す。

## 【 0 1 8 6 】

さらに具体的に例示すると、

```
do dim3 = 1, 4
  do dim2 = 1, 16
    do dim1 = 1, 2
      tempA(dim1, dim2, dim3, 2) = GA(dim1, dim2, dim3)
    enddo
  enddo
enddo
```

40

を実行することによって、配列変数 G A のデータが、ローカルメモリのブロック 2 に格納される。

50

## 【 0 1 8 7 】

図 1 8 に、別なテンプレートがマッピングされたローカルメモリの状態を示す。

## 【 0 1 8 8 】

前述した例と異なり、プログラム中に表れる配列サイズが [ 1 : 2 , 1 : 8 , 1 : 4 ] である場合、サブブロックサイズに適合するテンプレート `temp_subA [ 1 : 2 , 1 : 8 , 1 : 4 , 0 : 1 5 ]` をローカルメモリに割り当てられる。このように最外側の要素の値によってアクセスするサブブロックが可変となる。

## 【 0 1 8 9 】

前述と同様にEQUIVALENCE文を用いて、`EQUIVALENCE(LM(1), temp_subA(1,1,1,0))`と宣言することによって、ローカルメモリのサブブロック 0 に、テンプレート (サブ) A を割り当てることができる。

10

## 【 0 1 9 0 】

テンプレート A は、3次元配列用のテンプレートで、その大きさは `temp_subA [ 1 : 2 , 1 : 8 , 1 : 4 , 0 : 1 5 ]` である。よって、サブブロック 0 (アドレス 0 ~ 6 3) には `temp_subA [ 1 , 1 , 1 , 0 ]` が割り当てられ、サブブロック 1 (アドレス 6 4 ~ 1 2 7) には `temp_subA [ 1 , 1 , 1 , 1 ]` が割り当てられる。

## 【 0 1 9 1 】

さらに具体的に例示すると、

```
do dim3 = 1, 4
  do dim2 = 1, 8
    do dim1 = 1, 2
      temp_subA(dim1, dim2, dim3, 4) = GA(dim1, dim2, dim3)
    enddo
  enddo
enddo
```

20

を実行することによって、配列変数 G A のデータが、ローカルメモリのサブブロック 4 に格納される。

## 【 0 1 9 2 】

このように、コンパイラがプログラムを解析して取得した情報に基づいて、プログラム中で使用される配列変数の形を特定してテンプレートを作成し、その変数をどのテンプレートに割り当てられるかを決める。これによって、メモリの1次元のアドレス空間を多次元に見せることができ、プログラム中で使用された多次元配列をそのままの形でメモリに割り当てることができる。

30

## 【 0 1 9 3 】

<テンプレート配列を用いたコードイメージの作成>

次に、図 1 9 から図 3 4 を参照して、テンプレート配列を用いたコードイメージの作成の具体例について説明する。図 1 9 から図 3 4 の説明は、ローカルメモリサイズを 1 0 2 4、ブロックサイズは 1 2 8 とし、ローカルメモリの領域を 4 個のブロック 1 0 1、4 個のサブブロック 1 0 2、4 個のサブサブブロック 1 0 3、8 個のサブサブサブブロック 1 0 4 に分割した場合に、コンパイラがコードを書き換える様子及びプログラム実行時のローカルメモリの状態を示す。

40

## 【 0 1 9 4 】

図 1 9 は、コンパイル前のオリジナルコード及びローカルメモリの状態 (状態 1) を示す。このオリジナルコード中には、三つのループ及び二つのサブルーチン呼び出しが含まれており、これらのループ及びサブルーチン呼び出しが、ローカルメモリ管理マクロタスクとなる。

## 【 0 1 9 5 】

図 2 0 は、テンプレートがブロック等に設定される状態 (状態 2) を示す。L M 領域及びテンプレート配列を定義し、EQUIVALENCE文によってテンプレートをローカルメモリのアドレス空間に割り当てる。この割り当てによって、ローカルメモリとテンプレートとは

50

同じ領域を指す。そして、テンプレートの添字（0～7）を変えることによって対応する領域が変わる。なお、テンプレート配列によって宣言されるが `templ(1, 4)` 以後はサブブロック等に割り当てられた領域なので、ブロックとしては使用されない。

【0196】

具体的には、以下の文がプログラムに挿入される。

```
Integer a(128), b(128), c(128), d(128), e(128)
```

```
Integer LM(1024)
```

```
Integer templ(128, 0:7)
```

```
EQUIVALENCE (LM, templ)
```

これによって、テンプレートがローカルメモリに割り当てられる。

10

【0197】

図21は、テンプレート配列へ変換される状態（状態3）を示す。ローカルメモリに載せて使用したい配列を、新しく定義したテンプレート配列へ変換する。オリジナルコード中の配列をテンプレート配列にすることによって、ローカルメモリを使用していることになる。ブロックを指定する次元の値（添字）を変えることによって、使用されるブロック（ローカルメモリのアドレス）を変えることができる。

【0198】

具体的には、オリジナルのコード中の配列名  $a(i)$ 、 $b(i)$ 、 $c(i)$  は、 $templ(i, 0)$ 、 $templ(i, 1)$ 、 $templ(i, 2)$  に書き換えられる。

【0199】

20

図22は、ブロックからデータが掃き出される状態（状態4）を示す。二つ目のマクロタスクでは、四つの配列  $b$ 、 $c$ 、 $d$  及び  $e$  を使用するため、4個のブロックが必要である。一つ目のマクロタスクが終了した時点で、次に実行されるマクロタスクに必要なブロック数が空いていない場合、掃き出し優先度に従って必要な数のブロックを空ける。具体的には、三つの配列  $a$ 、 $b$  及び  $c$  が、ローカルメモリ上に載っているが、配列  $b$  及び  $c$  は継続して使用される。よって、配列  $e$  をロードするために、配列  $a$  が掃き出される。掃き出されるブロックに格納されていたデータは集中共有メモリ28又は14に転送される。よって、テンプレート配列に格納されたデータを集中共有メモリ28又は14の配列へ転送する命令がオリジナルコードに挿入される。

【0200】

30

図23は、テンプレート配列へ変換される状態（状態5）を示す。状態3（図21）に示した状態と同様に、ローカルメモリに載せて使用したい配列を、新しく定義したテンプレート配列へ変換する。ブロック0から配列  $a$  が掃き出され、配列  $d$  が格納されている。

【0201】

図24は、サブルーチンが解析される状態（状態6）を示す。サブルーチンの引数と、サブルーチン内の処理に必要なブロック数を解析する。図示する例では、サブルーチン `sub1` では引数配列  $x$  及び自動変数である配列  $y$  を使用している。つまり、引数用に1個のブロック、内部処理用に1個のブロック。合計2個のブロックが必要である。

【0202】

図25は、サブルーチン内でブロック指定変数を用いたテンプレート配列へ変換される状態（状態7）を示す。ここで、前述したように、サブルーチンは複数箇所から呼ばれる可能性があるため、テンプレート配列のブロック指定次元を定数によって指定すると、メモリ管理上の制限が強くなる。そのため、ブロック指定変数 `block_no1` を用いて、テンプレートを任意の場所に置けるように、テンプレート配列を変換をする。

40

【0203】

図26は、サブルーチン処理用のブロックを確保する状態（状態8）を示す。既に、サブルーチン内の解析が終わり、サブルーチンに必要なブロック数が分かっているので、必要な数のブロックをサブルーチン呼び出し時に空けて、サブルーチン処理用のブロックを確保する。必要な数のブロックが空いていない場合は、既にロードされているデータを掃き出す。掃き出されるデータは、掃き出し優先度に従って決められる。

50

## 【 0 2 0 4 】

具体的には、このサブルーチンでは、1個の引数ブロック及び1個の内部処理用ブロックが必要である。メモリ上に配列 b、c、d 及び e が載っているが、配列 e は引数として使用される。よって、1個のブロックを内部処理用に空ける必要がある。掃き出し優先度を考慮すると、配列 c 及び d は直ぐに使用されるため、配列 b が掃き出される。

## 【 0 2 0 5 】

図 27 は、ブロック指定変数が設定される状態（状態 9）を示す。サブルーチン内で使用される内部処理用の配列は、ブロック指定変数 block\_no1 によって任意のブロックを使用できる。このため、内部処理用の配列変数に割り当てられるブロック番号を指定変数に設定する。

10

## 【 0 2 0 6 】

図 28 は、サブルーチンが実行される状態（状態 10）を示す。サブルーチン呼び出し時に設定されたブロック指定変数によって、使用されるブロックが決定される。すなわち、ブロック 1 は内部処理用の配列 y に割り当てられ、ブロック 3 は引数用の配列 x に割り当てられる。サブルーチンでは、指定されたブロックを使用して、サブルーチンの処理が行なわれる。

## 【 0 2 0 7 】

図 29 は、サブルーチンの実行終了時の状態（状態 11）を示す。サブルーチンの処理が終わると、内部処理用のブロックは NULL となる。引数用のブロックは、引数として受け取った元の配列に戻る。

20

## 【 0 2 0 8 】

図 30 は、テンプレート配列へ変換される状態（状態 12）を示す。状態 3（図 21）及び状態 5（図 23）に示した状態と同様に、ローカルメモリに載せて使用したい配列を、新しく定義したテンプレート配列へ変換する。

## 【 0 2 0 9 】

図 31 は、サブルーチン処理用のブロックを確保する状態（状態 13）を示す。既に、サブルーチン内の解析が終わり、サブルーチンで必要なブロック数が分かっているので、サブルーチン呼び出し時に必要な数のブロックを空けて、サブルーチン処理用のブロックを確保する。必要な数のブロックが空いていない場合は、既にロードされているデータを掃き出す。掃き出されるデータは、掃き出し優先度に従って決められる。

30

## 【 0 2 1 0 】

具体的には、次のマクロタスク（サブルーチン呼び出し）で、サブルーチンの内部処理用に1個のブロックを空ける必要がある。ローカルメモリに載っている配列 d、a、c 及び e のうち、配列 a はサブルーチンの引数として使用される。配列 d、c 及び e の掃き出し優先度は同じなので、ブロック番号の最も小さいブロック 0 に格納されている配列 d を掃き出す。また、次のマクロタスクで配列 a が必要なことが分かっているので、データ転送ユニットによって、ローカルメモリの空いているブロック 1 に配列 a を転送する。

## 【 0 2 1 1 】

図 32 は、ブロック指定変数が設定される状態（状態 14）を示す。サブルーチン内で使用される内部処理用の配列は、ブロック指定変数 block\_no1 によって任意のブロックを使用できる。このため、サブルーチン呼出時に、内部処理用の配列変数に割り当てられるブロック番号を指定変数に設定する。前のサブルーチン呼出時（図 27 に示す状態 9）と異なるブロック番号を設定することができる。

40

## 【 0 2 1 2 】

図 33 は、サブルーチンが実行される状態（状態 15）を示す。サブルーチン呼び出し時に設定されたブロック指定変数によって、使用されるブロックが決定される。すなわち、ブロック 0 は内部処理用の配列 y に割り当てられ、ブロック 1 は引数用の配列 x に割り当てられる。サブルーチンでは、指定されたブロックを使用して、サブルーチンの処理が行なわれる。前のサブルーチン呼出時（図 28 に示す状態 10）とは別の領域で処理を行っている。

50

## 【 0 2 1 3 】

図 3 4 は、サブルーチンの実行終了時の状態（状態 1 6）を示す。図 3 4 に示すコードがコンパイル完了時のコードである。サブルーチンの処理が終わると、内部処理用のブロックは NULL となる。引数用のブロックは、引数として受け取った元の配列に戻る。

## 【 図面の簡単な説明 】

## 【 0 2 1 4 】

【 図 1 】 本発明の実施の形態のシングルチップマルチコアプロセッサの構成図である。

【 図 2 】 本発明の実施の形態のマルチグレイン並列処理の説明図である。

【 図 3 】 本発明の実施の形態のマクロフローグラフの説明図である。

【 図 4 】 本発明の実施の形態のローカルメモリ管理の概要の説明図である。 10

【 図 5 】 本発明の実施の形態のメモリの記憶領域の分割の状態の説明図である。

【 図 6 】 本発明の実施の形態のメモリの掃き出し優先度の説明図である。

【 図 7 】 本発明の実施の形態のメモリにロードされている変数の推移の説明図である。

【 図 8 】 本発明の実施の形態のループの分割手順の具体例の説明図である。

【 図 9 】 本発明の実施の形態のループの分割手順の具体例の説明図である。

【 図 1 0 】 本発明の実施の形態のループの分割手順の具体例の説明図である。

【 図 1 1 】 本発明の実施の形態のループの分割手順の具体例の説明図である。

【 図 1 2 】 本発明の実施の形態のブロックサイズの決定手順の具体例の説明図である。

【 図 1 3 】 本発明の実施の形態のブロックサイズの決定手順の具体例の説明図である。

【 図 1 4 】 本発明の実施の形態のローカルメモリ管理マクロタスクの生成の具体例（ディストリビューション前）の説明図である。 20

【 図 1 5 】 本発明の実施の形態のローカルメモリ管理マクロタスクの生成の具体例（ディストリビューション後）の説明図である。

【 図 1 6 】 本発明の実施の形態のテンプレートの作成手順の例を示す。

【 図 1 7 】 本発明の実施の形態のテンプレートがマッピングされたローカルメモリの状態の説明図である。

【 図 1 8 】 本発明の実施の形態のテンプレートがマッピングされたローカルメモリの状態の説明図である

【 図 1 9 】 本発明の実施の形態のコードイメージの作成例（状態 1）の説明図である。

【 図 2 0 】 本発明の実施の形態のコードイメージの作成例（状態 2）の説明図である。 30

【 図 2 1 】 本発明の実施の形態のコードイメージの作成例（状態 3）の説明図である。

【 図 2 2 】 本発明の実施の形態のコードイメージの作成例（状態 4）の説明図である。

【 図 2 3 】 本発明の実施の形態のコードイメージの作成例（状態 5）の説明図である。

【 図 2 4 】 本発明の実施の形態のコードイメージの作成例（状態 6）の説明図である。

【 図 2 5 】 本発明の実施の形態のコードイメージの作成例（状態 7）の説明図である。

【 図 2 6 】 本発明の実施の形態のコードイメージの作成例（状態 8）の説明図である。

【 図 2 7 】 本発明の実施の形態のコードイメージの作成例（状態 9）の説明図である。

【 図 2 8 】 本発明の実施の形態のコードイメージの作成例（状態 1 0）の説明図である。

【 図 2 9 】 本発明の実施の形態のコードイメージの作成例（状態 1 1）の説明図である。

【 図 3 0 】 本発明の実施の形態のコードイメージの作成例（状態 1 2）の説明図である。 40

【 図 3 1 】 本発明の実施の形態のコードイメージの作成例（状態 1 3）の説明図である。

【 図 3 2 】 本発明の実施の形態のコードイメージの作成例（状態 1 4）の説明図である。

【 図 3 3 】 本発明の実施の形態のコードイメージの作成例（状態 1 5）の説明図である。

【 図 3 4 】 本発明の実施の形態のコードイメージの作成例（状態 1 6）の説明図である。

## 【 符号の説明 】

## 【 0 2 1 5 】

1 0 シングルチップマルチプロセッサ

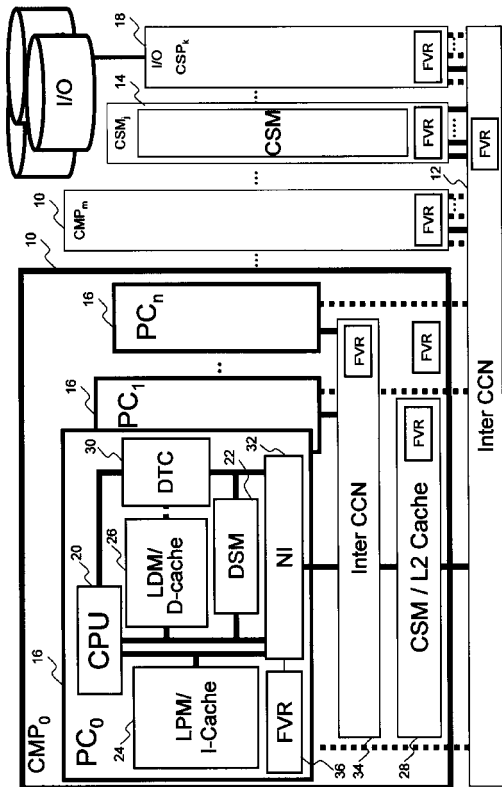
1 6 プロセッサコア（PC）

1 4 集中共有メモリ（CSM）

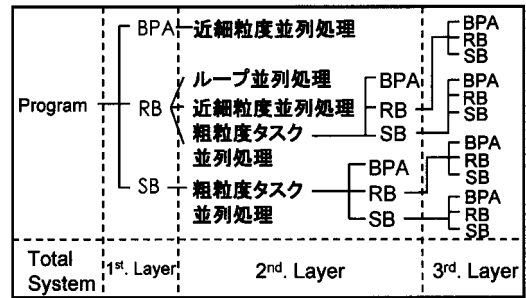
1 8 入出力用チップ（I/O CSP） 50

- 1 2 チップ間結合網 ( InterCCN )
- 2 8 集中共有メモリ ( CSM / L2 Cache )
- 3 4 チップ内結合網 ( IntraCCN )
- 2 0 CPU
- 2 2 分散共有メモリ ( DSM )
- 2 4 ローカルプログラムメモリ ( LPM / I - Cache )
- 2 6 ローカルデータメモリ ( LDM / D - cache )
- 3 0 データ転送コントローラ ( DTC )
- 3 2 ネットワークインターフェイス ( NI )
- 3 6 電力制御レジスタ ( FVR )

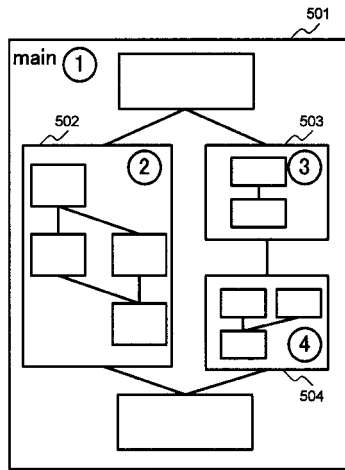
【 図 1 】



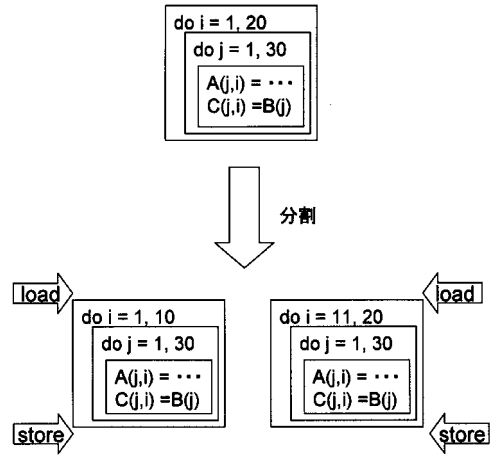
【 図 2 】



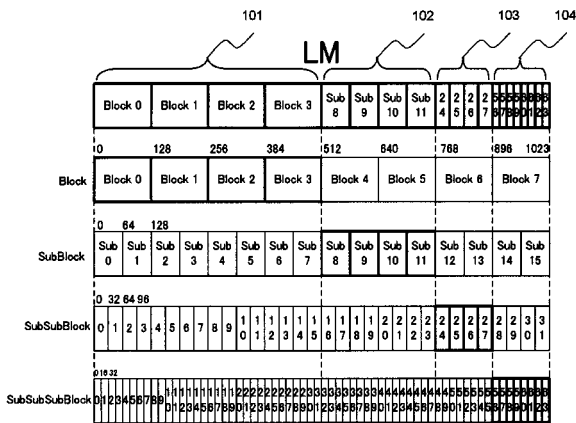
【 図 3 】



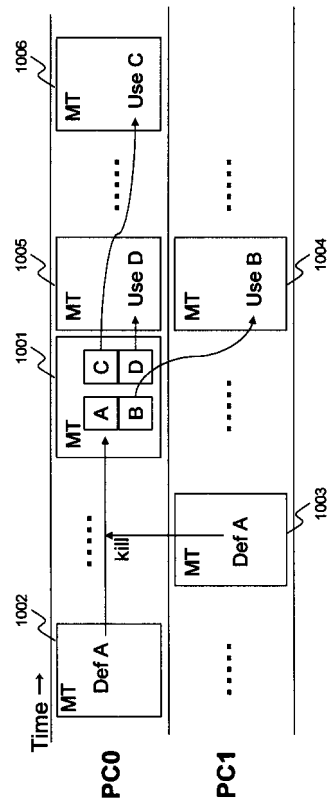
【 図 4 】



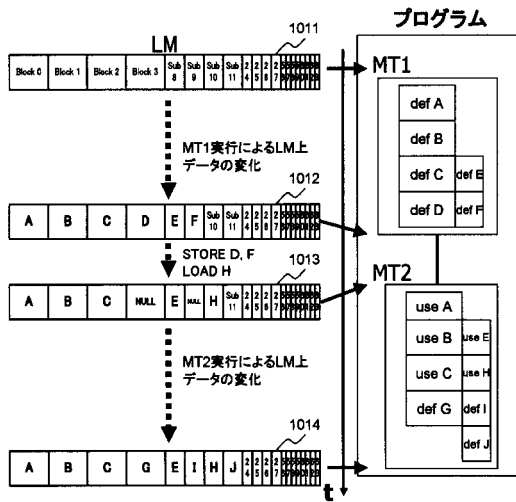
【 図 5 】



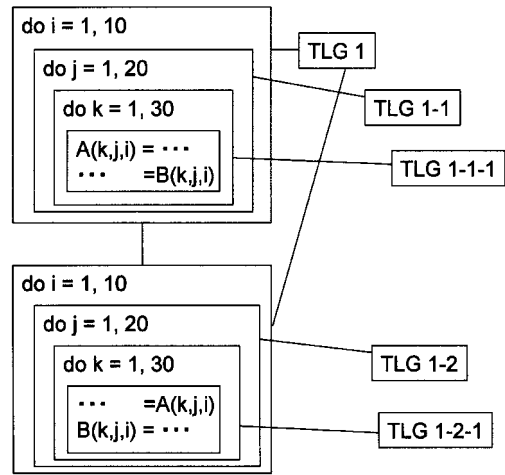
【 図 6 】



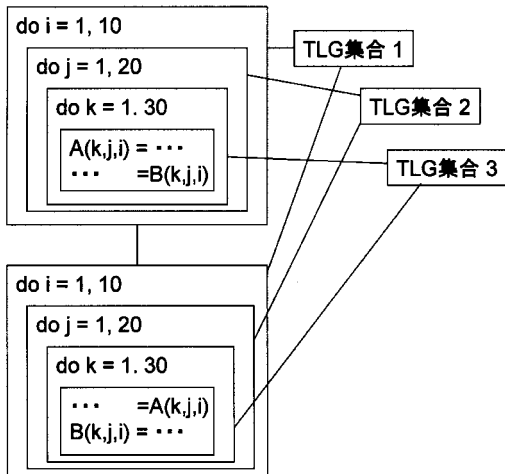
【 図 7 】



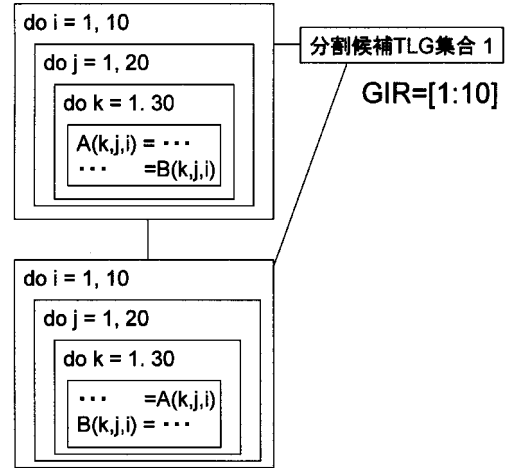
【 図 8 】



【 図 9 】

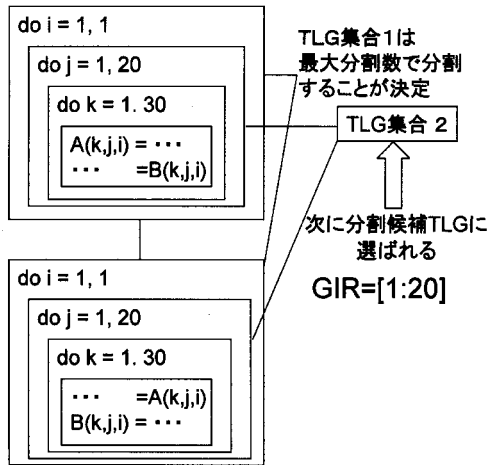


【 図 10 】

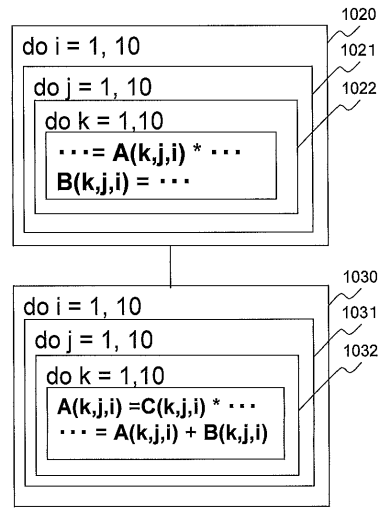




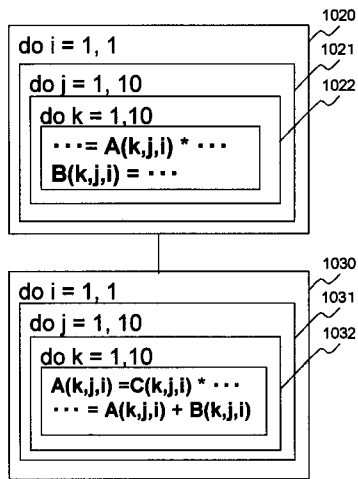
【図11】



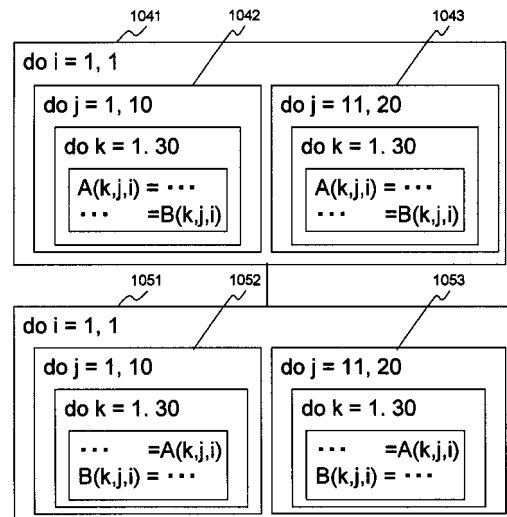
【図12】



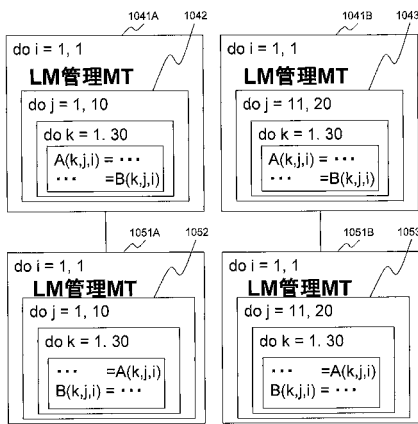
【図13】



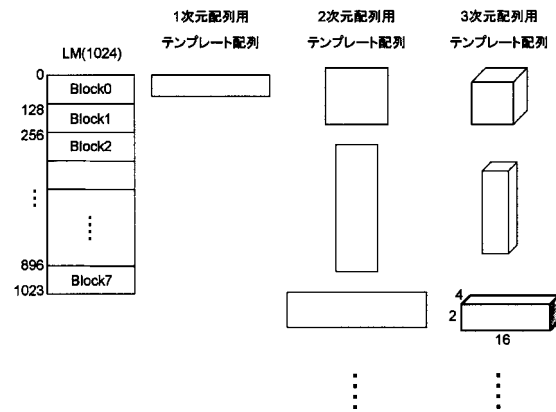
【図14】



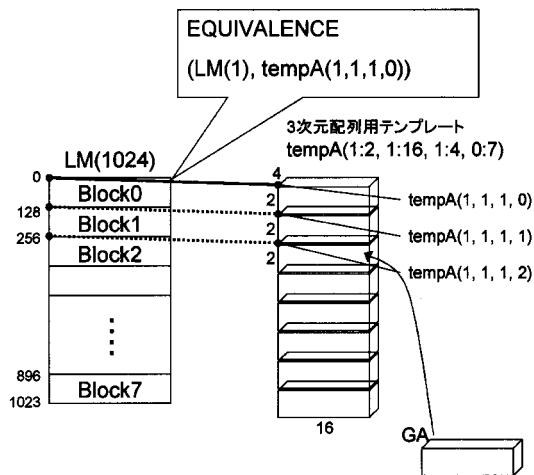
【 図 1 5 】



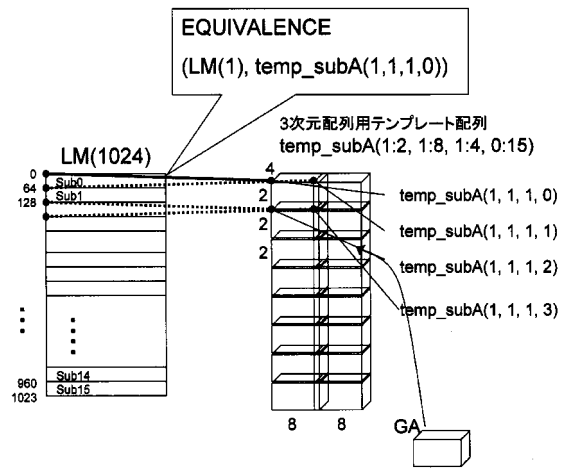
【 図 1 6 】



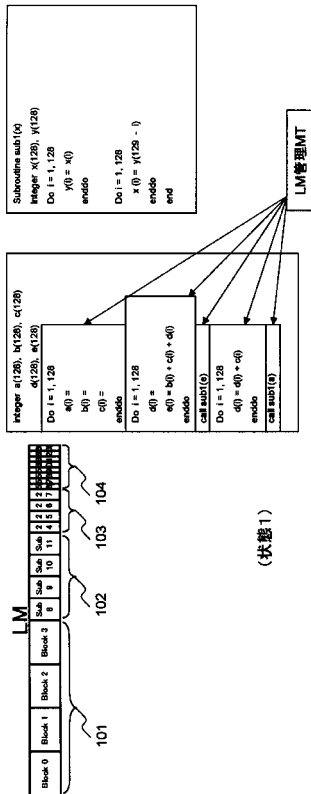
【 図 1 7 】



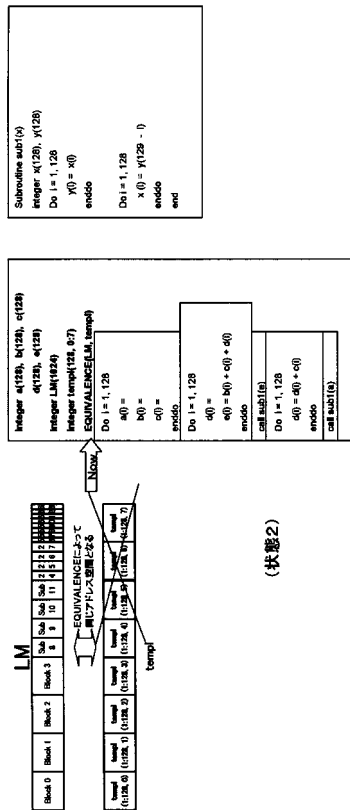
【 図 1 8 】



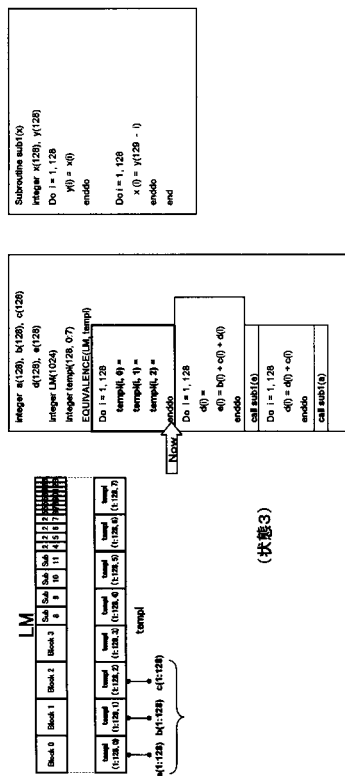
【 図 19 】



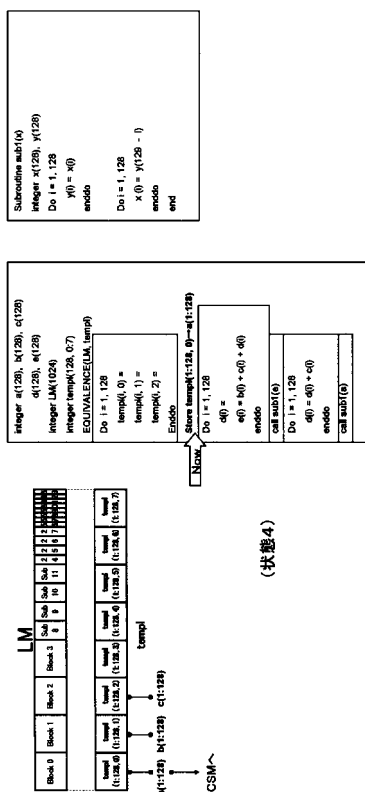
【 図 20 】



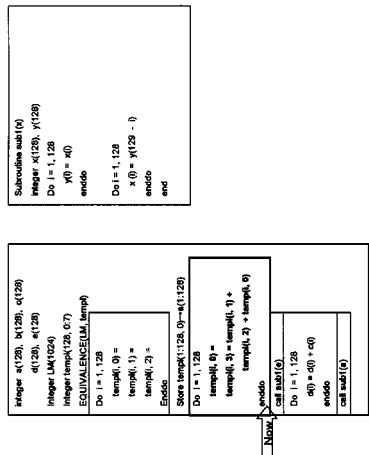
【 図 21 】



【 図 22 】

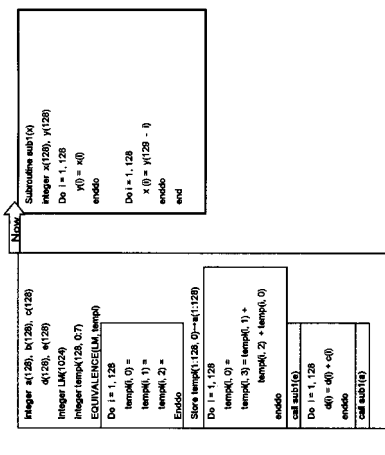


【 2 3 】



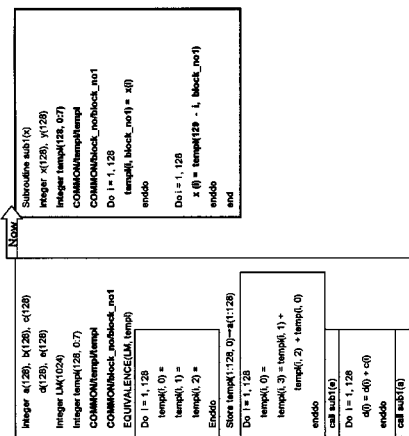
(状態5)

【 2 4 】



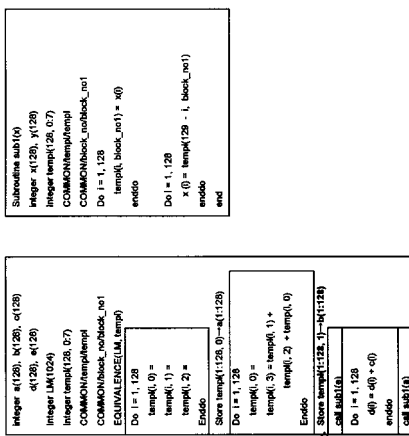
(状態6)

【 2 5 】



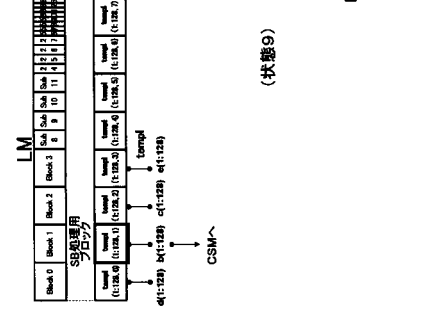
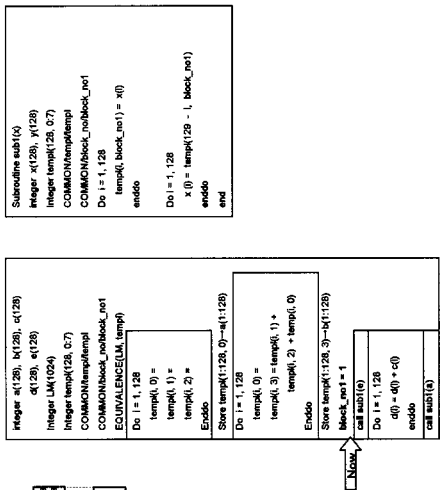
(状態7)

【 2 6 】

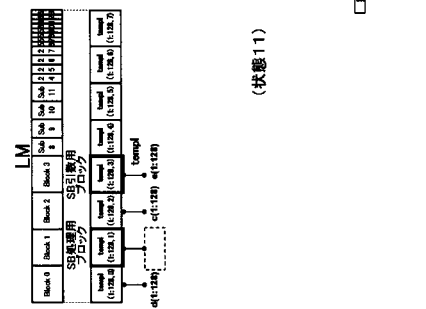
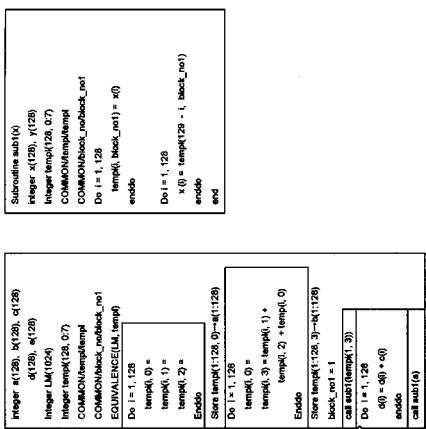


(状態8)

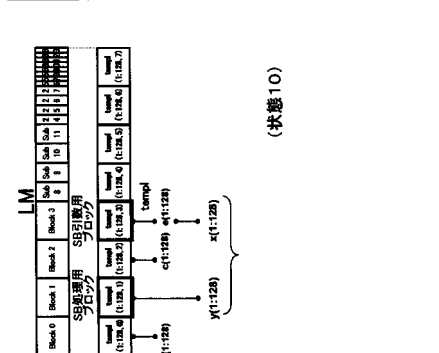
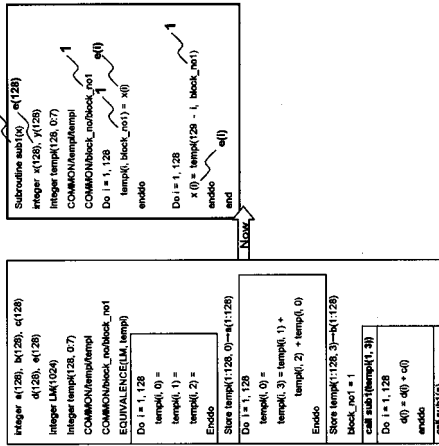
【 27】



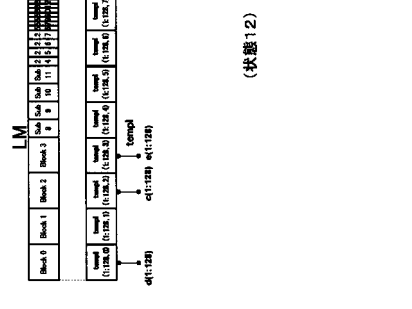
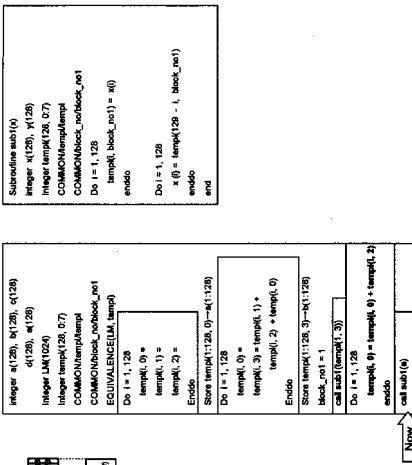
【 29】



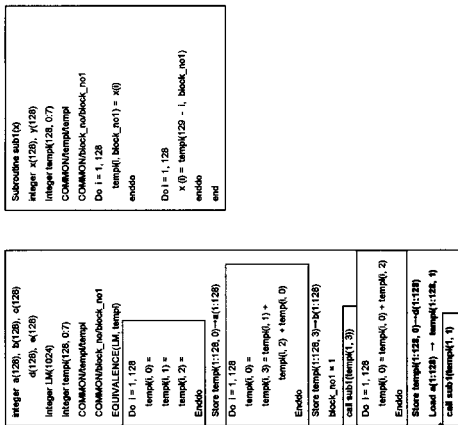
【 28】



【 30】

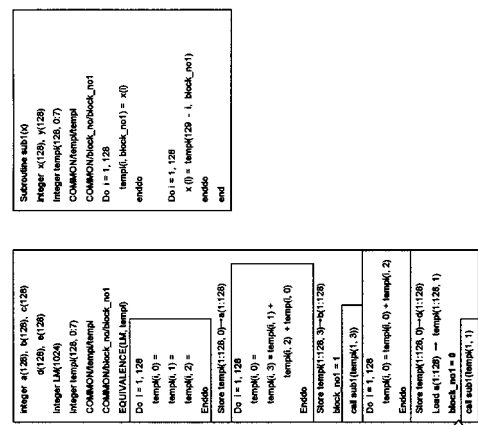


【 3 1 】



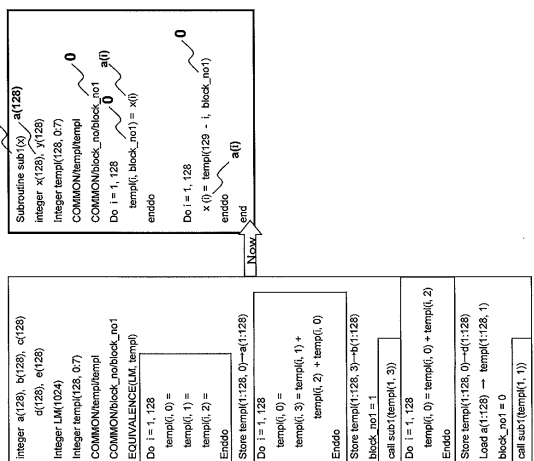
(状態 13)

【 3 2 】



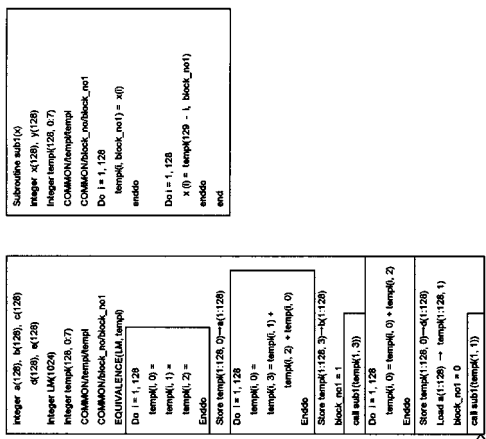
(状態 14)

【 3 3 】



(状態 15)

【 3 4 】



(状態 16)

(以下処理同様ニ続ク)

## フロントページの続き

- (72)発明者 中野 啓史  
東京都新宿区大久保3 - 4 - 1 早稲田大学理工学術院基幹理工学部情報理工学科内
- (72)発明者 仁藤 拓実  
東京都新宿区大久保3 - 4 - 1 早稲田大学理工学術院基幹理工学部情報理工学科内
- (72)発明者 丸山 貴紀  
東京都新宿区大久保3 - 4 - 1 早稲田大学理工学術院基幹理工学部情報理工学科内
- (72)発明者 三浦 剛  
東京都新宿区大久保3 - 4 - 1 早稲田大学理工学術院基幹理工学部情報理工学科内
- (72)発明者 田川 友博  
東京都新宿区大久保3 - 4 - 1 早稲田大学理工学術院基幹理工学部情報理工学科内

審査官 鈴木 修治

- (56)参考文献 国際公開第2005/081113(WO, A1)  
特表2007-523412(JP, A)  
特開2003-015883(JP, A)  
吉田 明正 他, マルチグレインデータローカライゼーションのための近細粒度タスクスケジューリング, 電子情報通信学会技術研究報告, 日本, 社団法人電子情報通信学会, 1996年 8月27日, 第96巻, 頁71-78

(58)調査した分野(Int.Cl., DB名)

G06F 9/46 - 9/54  
G06F 12/02  
G06F 12/06