(19)日本国特許庁(JP)　(12)公開特許公報(A)　(11)特許出願公開番号
特開2008-217134
(P2008-217134A)
(43)公開日　平成20年9月18日(2008.9.18)

(51)Int.Cl.		F I			テーマコード（参考）
G06F 9/50	(2006.01)	G06F	9/06	640H	5B060
G06F 12/02	(2006.01)	G06F	12/02	510M	5B176
G06F 12/06	(2006.01)	G06F	12/06	530B	

審査請求 未請求　請求項の数 21　ＯＬ　（全 39 頁）

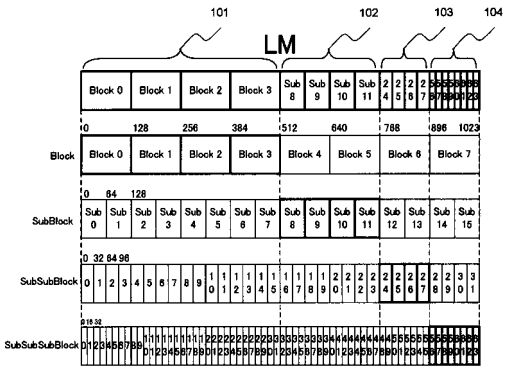
(21)出願番号　特願2007-50269(P2007-50269)
(22)出願日　平成19年2月28日(2007.2.28)

(71)出願人　899000068
学校法人早稲田大学
東京都新宿区戸塚町１丁目１０４番地
(74)代理人　100075513
弁理士　後藤　政喜
(74)代理人　100114236
弁理士　藤井　正弘
(74)代理人　100120260
弁理士　飯田　雅昭
(72)発明者　笠原　博徳
東京都新宿区新大久保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)【要約】
【課題】メモリへデータを効率よく配置する。
【解決手段】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、タスクの実行時に使用されるデータを格納するメモリに接続されており、前記メモリの記憶領域を複数の異なるサイズのブロックに分割し、前記タスクの実行時に使用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使用されるデータを格納する。
【選択図】図５



【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、
前記メモリの記憶領域を複数の異なるサイズのブロックに分割し、
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】
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決定されるサイズに、前記ブロックを分割することを特徴とする請求項１に記載のメモリ管理方法。
【請求項３】
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請求項１又は２に記載のメモリ管理方法。
【請求項４】
前記データに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、
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。
【請求項５】
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、前記タスクでアクセスされる配列データに整合するように選択されたｎ＋１次元のテンプレートを、前記ブロックに割り当て、
データを格納するブロックを指定する場合に、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、次元の値の異なる前記テンプレートを前記各ブロックに割り当てることを特徴とする請求項１から４のいずれか一つに記載のメモリ管理方法。
【請求項６】
プロセッサが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、プログラム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、
前記方法は、
前記メモリの記憶領域をブロックに分割し、
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７】
前記テンプレートを割り当てるステップでは、
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に１を加えた次元を有し、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より大きい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、複数の前記テンプレートを複数の前記ブロックに割り当てることを特徴とする請求項６に記載のプログラムの作成方法。
【請求項８】
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、

前記メモリの記憶領域は、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決定される複数のサイズのブロックに分割され、
前記ブロックに割り当てられるテンプレートの形状及びサイズは、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ており、
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決定される複数のサイズのブロックに、前記メモリの記憶領域を分割し、
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形状及びサイズのテンプレートを、適合する大きさの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９】
前記プログラムでアクセスされる配列データの次元に１を加えた次元を有し、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が割り当てられ、
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、複数の前記テンプレートが複数の前記ブロックに割り当てられることを特徴とする請求項８に記載の情報処理装置。
【請求項１０】
プロセッサによって実行可能なプログラムの作成方法であって、
プログラムの情報をコンパイラによって解析し、
前記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、
前記タスクの実行タイミングに従って、必要なデータをメモリに読み書きするタイミングを決定し、
前記決定されたデータの書き込みタイミングまでに前記メモリの領域を割り当てる命令を、コンパイルされるプログラムに追加することを特徴とするプログラムの作成方法。
【請求項１１】
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、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、前記コンパイルされるプログラムに追加することを特徴とする請求項１０に記載のプログラムの作成方法。
【請求項１２】
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、前記コンパイルされるプログラムに追加することを特徴とする請求項１１に記載のプログラムの作成方法。
【請求項１３】
前記プログラムの解析によって得られた情報は、前記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請求項１０から１２のいずれか一つに記載のプログラムの作成方法。
【請求項１４】
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、前記コンパイルされるプログラムに追加することを特徴とする請求項１０から１３のいずれか一つに記載のプログラムの作成方法。
【請求項１５】

前記メモリの領域は、前記メモリの記憶領域が固定サイズのブロックに分割された領域であることを特徴とする請求項１０から１４のいずれか一つに記載のプログラムの作成方法。

【請求項１６】

前記メモリの領域は複数の異なるサイズのブロックに分割された領域であって、

前記複数のサイズは、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請求項１５に記載のプログラムの作成方法。

【請求項１７】

前記ブロックのサイズは、前記コンパイラがプログラムを解析して得られた情報に基づいて決定されることを特徴とする請求項１５に記載のプログラムの作成方法。

【請求項１８】

前記タスクで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するために、前記プログラムを分割することを特徴とする請求項１５に記載のプログラムの作成方法。

【請求項１９】

前記プログラムは多重ループを含み、

外側のループの分割によって生成されたタスクで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、

前記外側のループが分割によって生成されたタスクで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することを特徴とする請求項１８に記載のプログラムの作成方法。

【請求項２０】

前記プログラム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、前記プログラムでアクセスされる配列データに整合するように選択されたｎ＋１次元のテンプレートを割り当て、

データを格納する領域を指定する場合に、前記加えられた次元の値によって、アクセスされる領域が指定されるように、複数の前記テンプレートを複数の領域に割り当てることを特徴とする請求項１０から１９のいずれか一つに記載のプログラムの作成方法。

【請求項２１】

プロセッサによって実行可能なプログラムであって、

前記プログラムは、コンパイラによって、

プログラムの情報が解析され、

前記プログラムに含まれる各タスクの実行に必要なデータが特定され、

前記タスクの実行タイミングに従って、必要なデータをメモリに読み書きするタイミングが決定され、

前記決定されたデータの書き込みタイミングまでに前記メモリの領域を割り当てる命令が追加されることによって生成されるプログラム。

【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】

【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。
【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】
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決める。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。

【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。

【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。

【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。

【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ければならないデータを「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（ＣＳＭ／Ｌ２　Ｃａｃｈｅ）２８及びチップ内結合網（ＩｎｔｒａＣＣＮ）３４を備える。各プロセッサコア１６は、ＣＰＵ２０、分散共有メモリ（ＤＳＭ：distributed shared memory）２２、ローカルプログラムメモリ（ＬＰＭ／Ｉ－Ｃａｃｈｅ）２４、ローカルデータメモリ（ＬＤＭ／Ｄ－ｃａｃｈｅ）２６、データ転送コントローラ（ＤＴＣ）３０、ネットワークインターフェイス（ＮＩ）３２及び電力制御レジスタ（ＦＶＲ）３６を備える。

【００３４】

ＣＰＵ２０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ッシューＲＩＳＣアーキテクチャのＣＰＵを用いることができる。また、スーパースカラプロセッサ、ＶＬＩＷプロセッサ等も用いることができる。

【００３５】

分散共有メモリ（ＤＳＭ）２２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３０を介して、他のプロセッサコア１６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。

【００３６】

ローカルプログラムメモリ（ＬＰＭ）２４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。

【００３７】

ローカルデータメモリ（ＬＤＭ）２６は、各プロセッサコア１６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１６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２６は、Ｌ１データキャッシュに切り替えることができる。

【００３８】

データ転送コントローラ（ＤＴＣ）３０は、公知のＤＭＡ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１６上のローカルメモリ２６、自及び他のプロセッサコア１６上の分散共有メモリ２２、自及び他のマルチコアチップ１０上の集中共有メモリ２８、及び、他のチップに設けられた集中共有メモリ１４間でデータを転送する。

【００３９】

なお、ローカルデータメモリ２６とデータ転送コントローラ３０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３０がローカルデータメモリ２６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、ＣＰＵ２０が、転送指示を、ローカルデータメモリ２６を介して、データ転送コントローラ３０に与えることができる。また、ＣＰＵ２０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。

【００４０】

ＣＰＵ２０は、ローカルデータメモリ２６、分散共有メモリ２２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３０へデータ転送を指示する。また、データ転送コントローラ３０は、ローカルデータメモリ２６、分散共有メモリ２２又は専用のバッファ（図示省略）を介して、ＣＰＵ２０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。

【００４１】

データ転送コントローラ３０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、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の指示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。

【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。

【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。
【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。
【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。繰り返しブロックは、各階層での最も外側のループである。
【００５１】
また、疑似代入文ブロックは、スケジューリングオーバ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。
【００５２】
最外側ループである繰り返しブロックがＤｏａｌｌループである場合は、ループインデクスを分割することによって、複数の部分Ｄｏａｌｌループに分割し、分割されたＤｏａｌｌ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ｎｏｎ－Ｄｏａｌｌ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。
【００５３】
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。
【００５４】
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（ＭＴＧ）
を作成する。マクロフローグラフでは、マクロタスク（ＭＴ）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。

【００５５】
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ＣＰ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１４に割り当てるとよい。

【００５６】
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Ｄｏａｌｌループ又はＤｏａｃｒｏｓｓ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１６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。

【００５７】
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　エクスパンション、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、アレイプライベタイゼーション、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、スキューイング等）等の従来の技術がそのまま利用できる。

【００５８】
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。

【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／ＤＴ／ＭＩＳＦ法、ＣＰ／ＥＴＦ／ＭＩＳＦ法、ＥＴＦ／ＣＰ法、又は、ＤＴ／ＣＰ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。

【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。

【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で、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で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。

【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を呼び出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。

【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。

【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。
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。

【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】
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。

【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。

【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。

【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は変更されない。

【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決める。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。

【００９０】

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２６から読み出して、集中共有メモリ２８又は１４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。

【００９１】

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。

【００９２】

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、先の時間で使用されるデータ。

４）自プロセッサですぐに使用されるデータ。

【００９３】

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１６の分散共有メモリ２２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２２へ転送すればよいが、他のプロセッサコア１６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２２へ転送するか、集中共有メモリ２８又は１４へ転送する。このようにローカルメモリ２６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。

【００９４】

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決める。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。

【００９５】

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（ＰＣ０）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（１００１）。

【００９６】

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（ＰＣ０）で、配列変数Ａを定義し（Ｄｅｆ　Ａ）、配列変数Ａを使用している（１００２）。

【００９７】

次に、プロセッサコア１（ＰＣ１）で、別の配列変数Ａを定義した（１００３）。すると、ＰＣ０のローカルメモリにロードされている配列変数Ａは既に変わっているので、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。

【００９８】

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（ＰＣ１）で使用される（１００４）。配列変数Ｃ及びＤは、自プロセッサ（ＰＣ０）で使用される（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】
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。
【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。
【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。

【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。

【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。

【０１１６】

次に、図９に示すように、ＴＬＧ集合を生成する。ＴＬＧが以下の生成条件の全てを満たす場合に、ＴＬＧ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。

【０１１７】

次に、図１０に示すように、分割候補ＴＬＧ集合を生成する。これは、入れ子になったＴＬＧがある場合、コストが最大のＴＬＧ集合を選択し、選択されたＴＬＧ集合を分割候補とする。その後、分割候補ＴＬＧ集合毎にＧＩＲ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、ＴＬＧ集合１が分割候補ＴＬＧ集合１となり、ＧＩＲは［１：１０］となる。

【０１１８】

次に、分割基準領域を決定する。分割基準領域は、各ＴＬＧ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ＴＬＧ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ＴＬＧ集合があり、集合１で使用されるデータサイズが３００ｋ、集合２で使用されるデータサイズが２００ｋ、集合３で使用されるデータサイズが１００ｋであれば、データサイズの比は３：２：１になる。

【０１１９】

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ＴＬＧ集合に割り当てる。具体的には、分割候補ＴＬＧ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の領域に収まるように、分割候補ＴＬＧ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ＴＬＧ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。

【０１２０】

これによって、分割後の各分割候補ＴＬＧ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。

【０１２１】

次に、ブロックサイズを決定する。

【０１２２】

まず、コストが最も大きい分割候補ＴＬＧ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ＴＬＧ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（ＰＧ）内のプロセッサコア（ＰＣ）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。ＣＰＵ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ＴＬＧ集合は、分割候補ＴＬＧ集合に選ばない。そして、再び、分割候補ＴＬＧ集合を生成する。

【０１２３】

具体的には、図１１に示すように、ＴＬＧ集合１は最大分割数で分割されるので、次のＴＬＧ集合２が分割候補ＴＬＧに選択される。分割候補となったＴＬＧ集合２は、ＧＩＲは［１：２０］である。

【０１２４】

次に、ブロックサイズの決定手順について説明する。

【０１２５】

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ＴＬＧ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。求められた分割数でＴＬＧ集合内の各ループの分割を試行する。

【０１２６】

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］＝［１：３０，１：２０，１：３］である。総データサイズは、３０×２０×３×２＝３６００になる。

【０１２７】

次に、ブロックサイズを決定する。

【０１２８】

分割後の配列アクセス範囲に基づいて、ＴＬＧ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。

【０１２９】

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。

【０１３０】

ＴＬＧ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。

【０１３１】

一方、ＴＬＧ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ＴＬＧ集合は、分割候補ＴＬＧ集合に選ばない。そして、再び、分割候補ＴＬＧ集合を生成する。

【０１３２】
次に、分割数を決定する。
【０１３３】
分割候補ＴＬＧ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ＴＬＧ集合の割り当てに失敗した場合、再び、分割候補ＴＬＧ集合を生成する。
【０１３４】
その際、割り当てできなかった分割候補ＴＬＧ集合は最大分割数を採用し、以後、割り当てできなかったＴＬＧ集合は、分割候補ＴＬＧ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ＴＬＧ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ＴＬＧ集合のデータサイズの比に基づいて、再度、ＴＬＧ集合を割り当てる。そして、分割数が決定した分割候補ＴＬＧ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。
【０１３５】
具体的には、ＴＬＧ集合１で使用されるデータサイズが３００ｋ、ＴＬＧ集合２で使用されるデータサイズが２００ｋ、ＴＬＧ集合３で使用されるデータサイズが１００ｋである例を考える。ＴＬＧ集合１を基準にブロックサイズが決定できたとする。ＴＬＧ集合２を最大分割数でも割り当てに失敗した場合、ＴＬＧ集合２の中にＴＬＧ集合４、ＴＬＧ集合５及びＴＬＧ集合６があった場合、これらが次の分割候補ＴＬＧ集合に加わる。
【０１３６】
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。
【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。
【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。これらを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となってしまう。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。

【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。

【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＝１～３、４～６、７～８、９～１０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、３２×３２×４＝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。

【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。

【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。

【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２ｋＢ、各配列要素のサイズは４Ｂ／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。

【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１６×１６×１６×４＝１６ｋＢである。ローカルメモリのサイズは２ｋＢ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】

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１６×１６×１×４＝１ｋＢである。ローカルメモリのサイズは２ｋＢなので、用意できるブロック数は、分割基準領域サイズ（２ｋＢ）／ブロックサイズ（１ｋＢ）＝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。

【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１６×８×１×４＝５１２Ｂである。ローカルメモリのサイズは２ｋＢなので、分割基準領域サイズ（２ｋＢ）／ブロックサイズ（５１２Ｂ）によって用意できるブロック数を求めると、用意できるブロックは４個となる。

【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。
【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２ｋＢ、各配列要素のサイズは４Ｂ／要素と仮定し、分散共有メモリの存在は考えない。
【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［１：３０，１：１０，１：１］）を使用し、ループ１０４３とループ１０５３とは同じデータ（配列変数Ａ［１：３０，１１：２０，１：１］及び配列変数Ｂ［１：３０，１１：２０，１：１］）を使用する。
【０１６９】
しかし、このプログラムでは、ループ１０４２、ループ１０４３、ループ１０５２、ループ１０５３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１５に示すように、最外ループ１０４１を、内側ループ１０４２（ｊ＝１：１０）と内側ループ１０４３（ｊ＝１１：２０）とに分割する。同様に、最外ループ１０５１を、内側ループ１０５２（ｊ＝１：１０）と内側ループ１０５３（ｊ＝１１：２０）とに分割する。
【０１７０】
この分割によって、分割された最外ループ１０４１Ａと最外ループ１０５１Ａとを順に実行することができる（すなわち、内側ループ１０４２と内側ループ１０５２とが順に実行される）。このため、ループ１０４２で使用した配列データを、そのまま、ループ１０５２で使用することができる。つまり、ループ１０４１Ａの実行終了時とループ１０５１Ａ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。
【０１７１】
同様に、分割された最外ループ１０４１Ｂと最外ループ１０５１Ｂとを順に実行することができる（すなわち、内側ループ１０４３と内側ループ１０５３とが順に実行される）。このため、ループ１０４３で使用した配列データを、そのまま、ループ１０５３で使用することができる。つまり、ループ１０４１Ｂの実行終了時とループ１０５１Ｂ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。
【０１７２】
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、ループのディストリビューションを実行して、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。

【０１７３】
＜テンプレートの作成手順の例＞
図１６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。
【０１７４】
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。
【０１７５】
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。
【０１７６】
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。
【０１７７】
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。
【０１７８】
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。
【０１７９】
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。
【０１８０】
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。
【０１８１】
例えば、ブロック数が８個であり、その各々のサイズが［１：２，１：１６，１：４］である場合、ｔｅｍｐＡ［１：２，１：１６，１：４，０：７］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［１：２，１：１６，１：４，０：４］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。
【０１８２】
図１７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。

【０１８３】
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。
【０１８４】
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［１：２，１：１６，１：４，０：７］である。よって、ブロック０（アドレス０～１２７）にはｔｅｍｐＡ［１，１，１，０］が割り当てられ、ブロック１（アドレス１２８～２５５）にはｔｅｍｐＡ［１，１，１，１］が割り当てられる。
【０１８５】
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。
【０１８６】
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
```

を実行することによって、配列変数ＧＡのデータが、ローカルメモリのブロック２に格納される。
【０１８７】
図１８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。
【０１８８】
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［１：２，１：８，１：４，０：１５］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。
【０１８９】
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。
【０１９０】
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［１：２，１：８，１：４，０：１５］である。よって、サブブロック０（アドレス０～６３）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６４～１２７）にはｔｅｍｐ＿ｓｕｂＡ［１，１，１，１］が割り当てられる。
【０１９１】
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
```

を実行することによって、配列変数ＧＡ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。

【０１９２】
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。
【０１９３】
＜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１９から図３４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１９から図３４の説明は、ローカルメモリサイズを１０２４、ブロックサイズは１２８とし、ローカルメモリの領域を４個のブロック１０１、４個のサブブロック１０２、４個のサブサブブロック１０３、８個のサブサブサブブロック１０４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。
【０１９４】
図１９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。
【０１９５】
図２０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。ＬＭ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。
【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】
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で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】

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。

【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、サブルーチンで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。

【０２０４】

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。

【０２０５】

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。

【０２０６】

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。

【０２０７】

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。

【０２０８】

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。

【０２０９】

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。

【０２１０】

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。

【０２１１】

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。

【０２１２】

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。

【０２１３】

図３４は、サブルーチンの実行終了時の状態（状態１６）を示す。図３４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】

【０２１４】

【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。

【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１０】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１１】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１２】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１３】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１４】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。

【図１５】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。

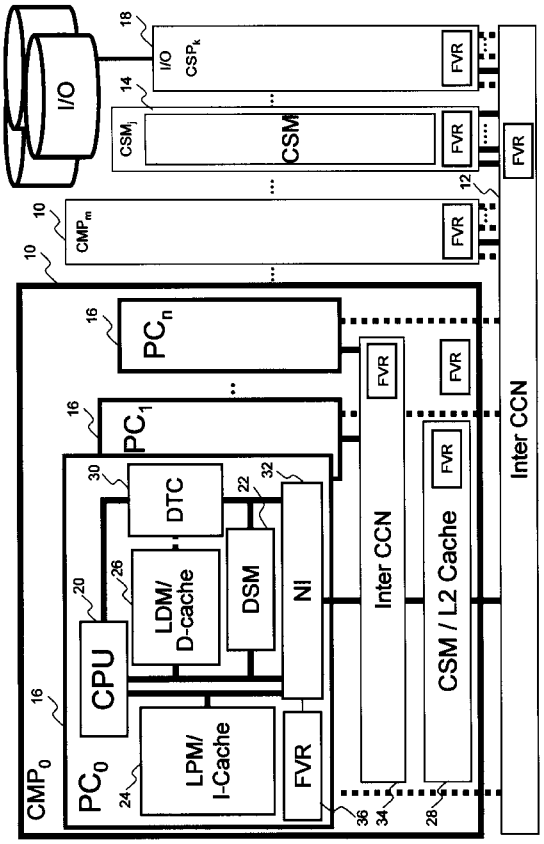
【図１６】本発明の実施の形態のテンプレートの作成手順の例を示す。

【図１７】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。

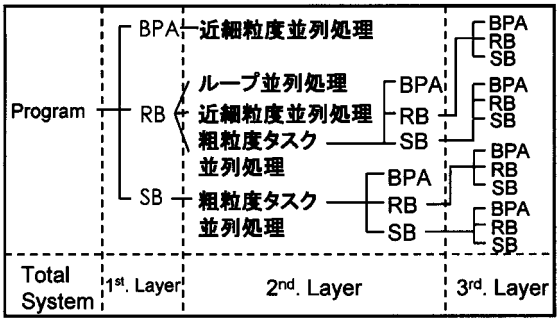
【図１８】本発明の実施の形態のテンプレートがマッピングされたローカルメモリの状態

の説明図である
【図１９】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２０】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。
【図２１】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２２】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２３】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２４】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２５】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２６】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２７】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２８】本発明の実施の形態のコードイメージの作成例（状態１０）の説明図である。
【図２９】本発明の実施の形態のコードイメージの作成例（状態１１）の説明図である。
【図３０】本発明の実施の形態のコードイメージの作成例（状態１２）の説明図である。
【図３１】本発明の実施の形態のコードイメージの作成例（状態１３）の説明図である。
【図３２】本発明の実施の形態のコードイメージの作成例（状態１４）の説明図である。
【図３３】本発明の実施の形態のコードイメージの作成例（状態１５）の説明図である。
【図３４】本発明の実施の形態のコードイメージの作成例（状態１６）の説明図である。
【符号の説明】
【０２１５】
１０　シングルチップマルチプロセッサ
１６　プロセッサコア（ＰＣ）
１４　集中共有メモリ（ＣＳＭ）
１８　入出力用チップ（Ｉ／Ｏ　ＣＳＰ）
１２　チップ間結合網（ＩｎｔｅｒＣＣＮ）
２８　集中共有メモリ（ＣＳＭ／Ｌ２　Ｃａｃｈｅ）
３４　チップ内結合網（ＩｎｔｒａＣＣＮ）
２０　ＣＰＵ
２２　分散共有メモリ（ＤＳＭ）
２４　ローカルプログラムメモリ（ＬＰＭ／Ｉ－Ｃａｃｈｅ）
２６　ローカルデータメモリ（ＬＤＭ／Ｄ－ｃａｃｈｅ）
３０　データ転送コントローラ（ＤＴＣ）
３２　ネットワークインターフェイス（ＮＩ）
３６　電力制御レジスタ（ＦＶＲ）

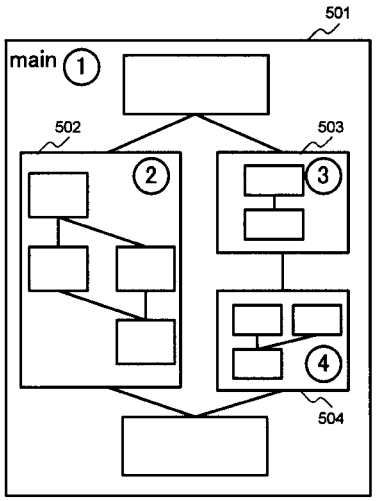
【図１】


I/O
CSP
CSM
FVR
Inter CCN
CMP
PC
DTC
LDM/
D-cache
DSM
NI
CPU
LPM/
I-Cache
CSM / L2 Cache

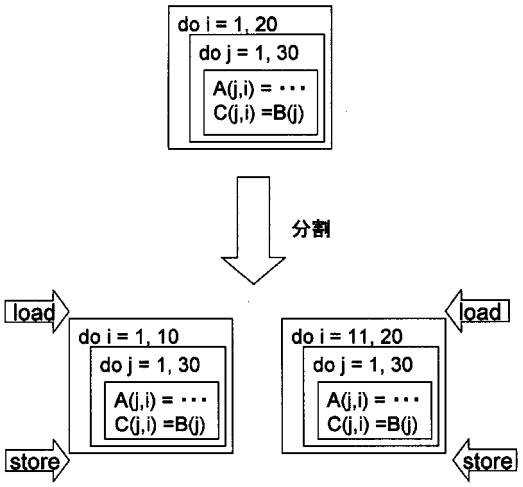
【図２】


Program
Total System
BPA
RB
SB
近細粒度並列処理
ループ並列処理
近細粒度並列処理
粗粒度タスク
並列処理
粗粒度タスク
並列処理
1st. Layer
2nd. Layer
3rd. Layer

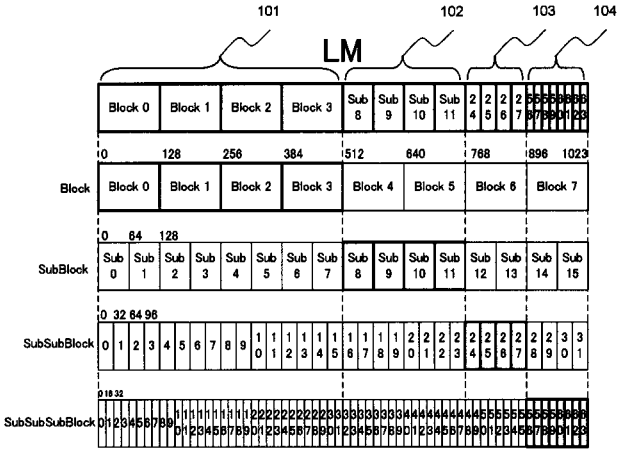
【図３】


501
main
1
502
503
2
3
4
504

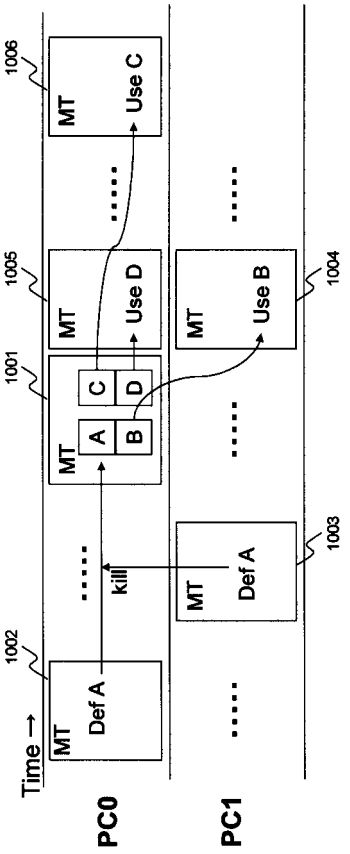
【図４】


do i = 1, 20
do j = 1, 30
A(j,i) = ・・・
C(j,i) =B(j)
分割
load
do i = 1, 10
do j = 1, 30
A(j,i) = ・・・
C(j,i) =B(j)
store
load
do i = 11, 20
do j = 1, 30
A(j,i) = ・・・
C(j,i) =B(j)
store

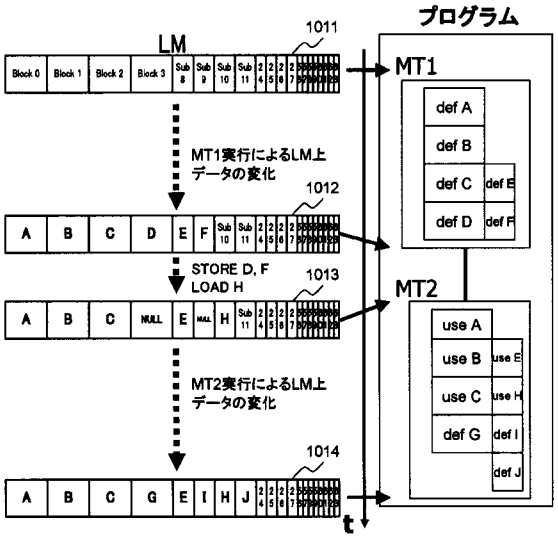
【図５】


101
102
103
104
LM
Block 0
Block 1
Block 2
Block 3
Sub 8
Sub 9
Sub 10
Sub 11
Block
Block 4
Block 5
Block 6
Block 7
SubBlock
SubSubBlock
SubSubSubBlock

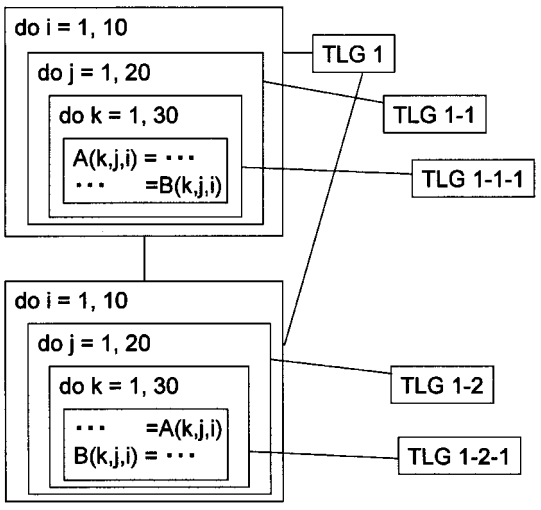
【図６】


1006
1005
1001
1002
1004
1003
MT
Use C
Use D
Use B
Def A
kill
C
D
A
B
Time →
PC0
PC1

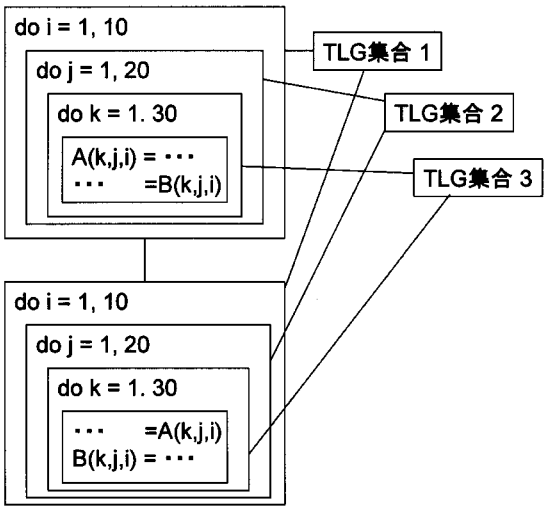
【図７】


プログラム
LM
1011
MT1
def A
def B
def C
def D
def E
def F
MT1実行によるLM上
データの変化
1012
STORE D, F
LOAD H
1013
MT2
use A
use B
use C
def G
use E
use H
def I
def J
MT2実行によるLM上
データの変化
1014
t

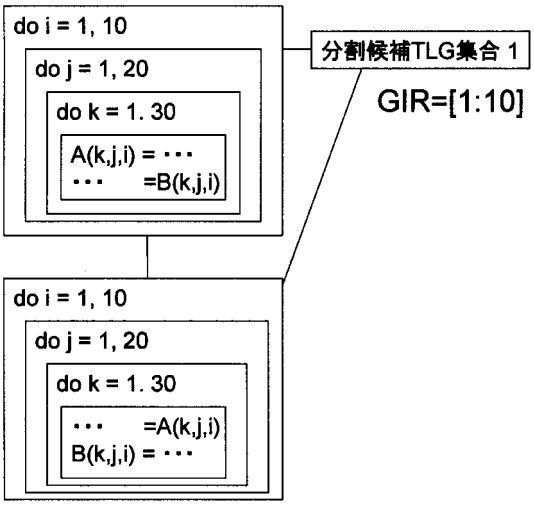
【図８】


do i = 1, 10
do j = 1, 20
do k = 1, 30
A(k,j,i) = ・・・
・・・ =B(k,j,i)
TLG 1
TLG 1-1
TLG 1-1-1
do i = 1, 10
do j = 1, 20
do k = 1, 30
・・・ =A(k,j,i)
B(k,j,i) = ・・・
TLG 1-2
TLG 1-2-1

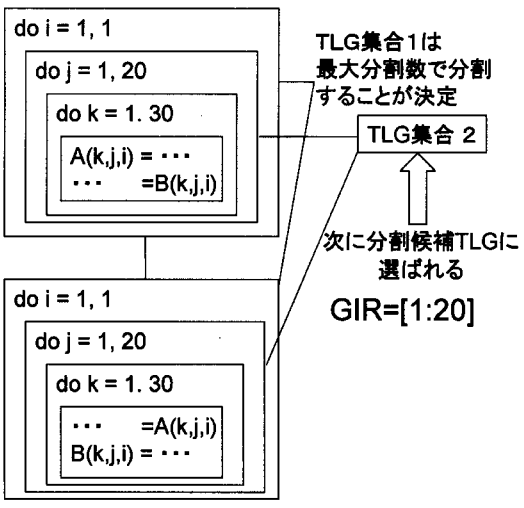
【図９】



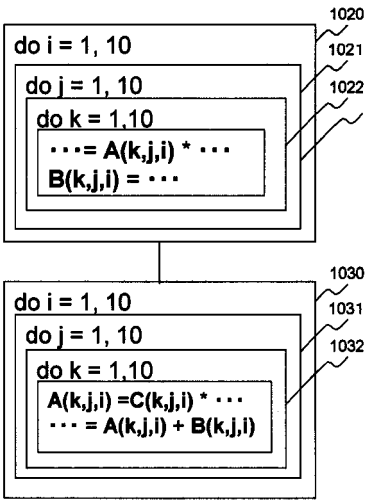
【図１０】



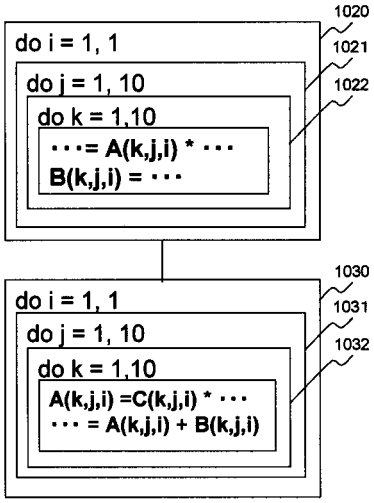
【図１１】



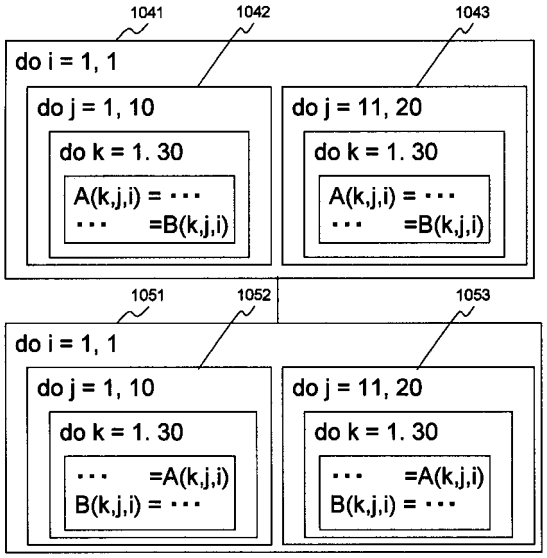
【図１２】



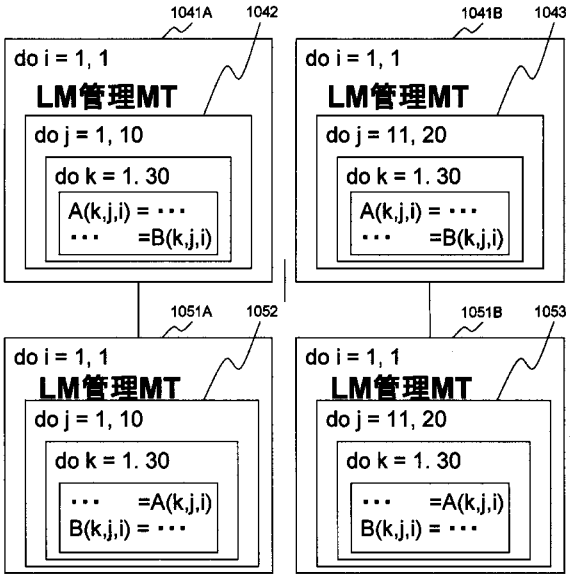
【図１３】



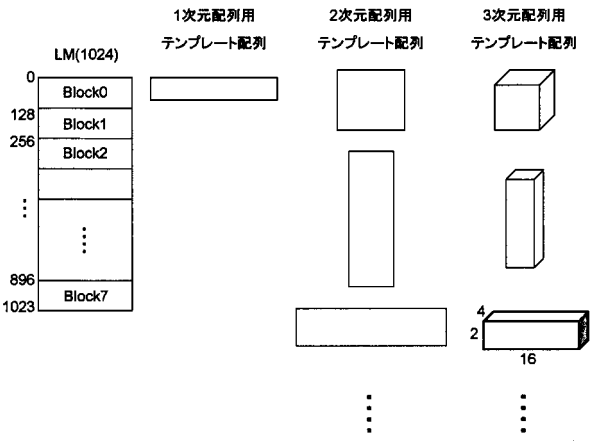
【図１４】



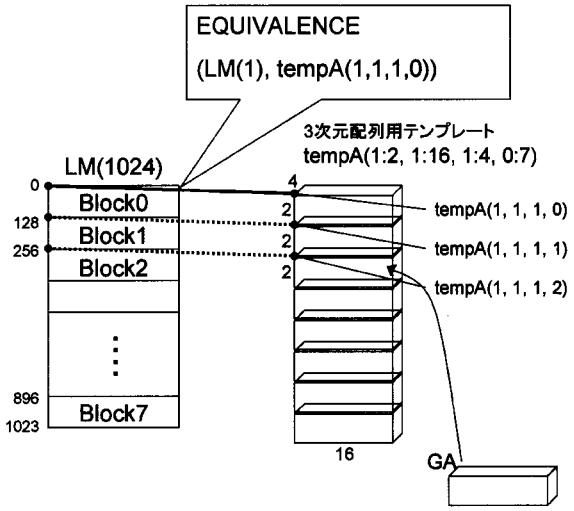
【図１５】



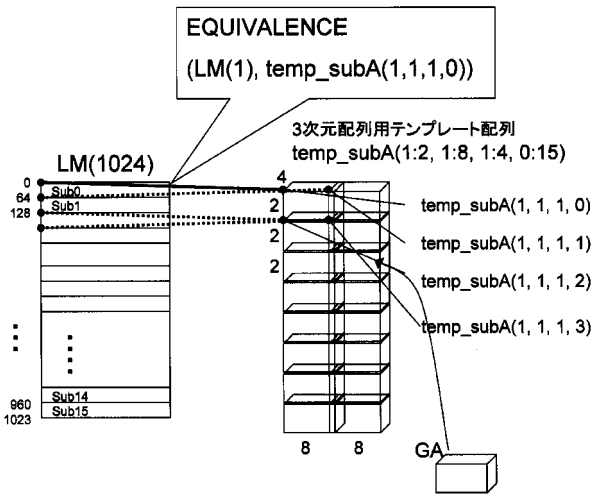
【図１６】



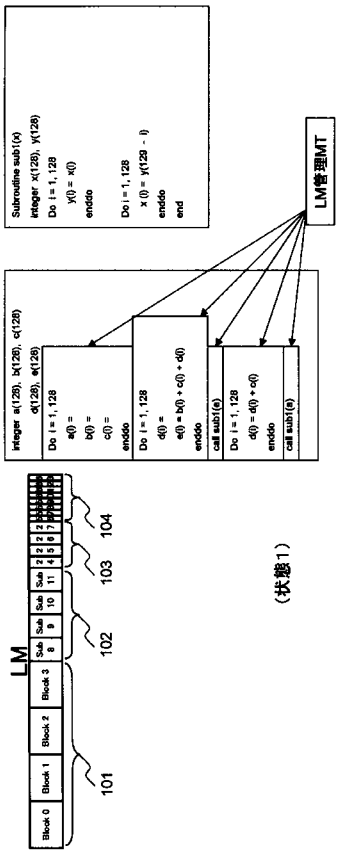
【図１７】



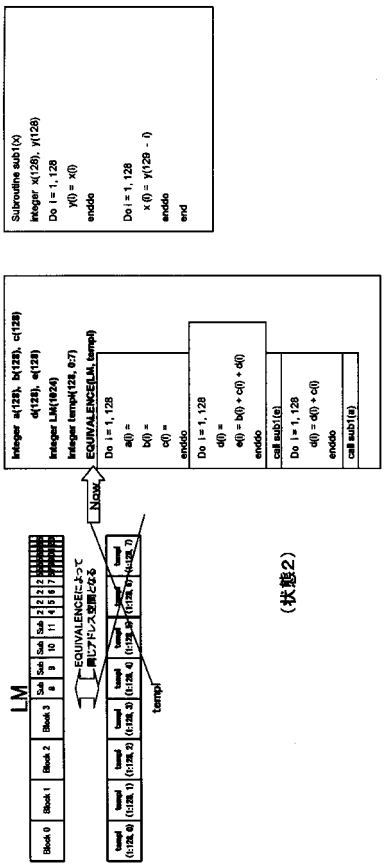
【図１８】



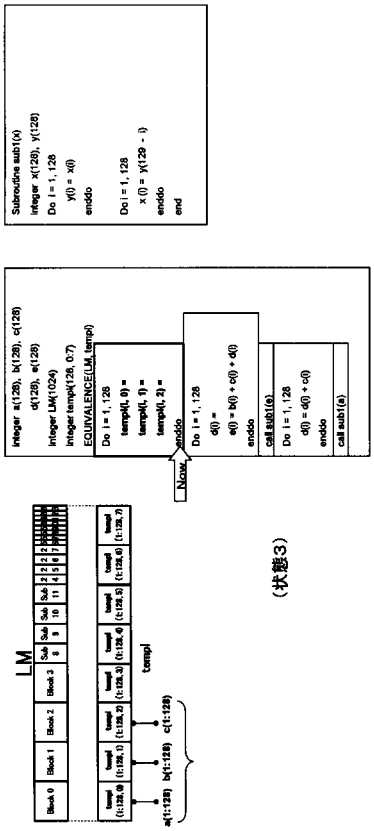
【図１９】



【図２０】

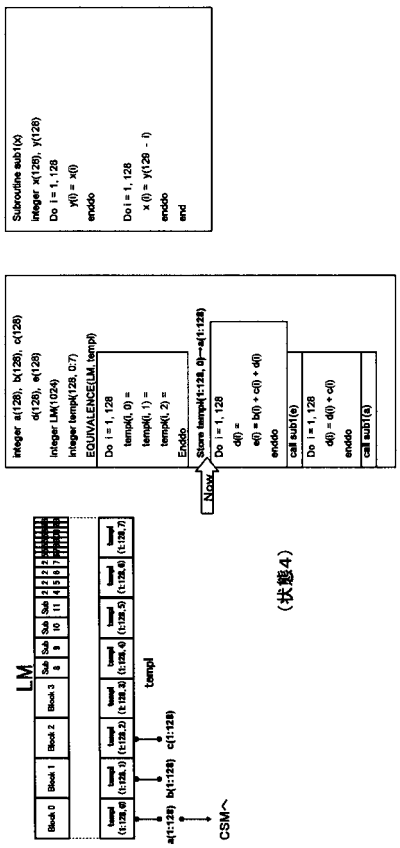


【図２１】



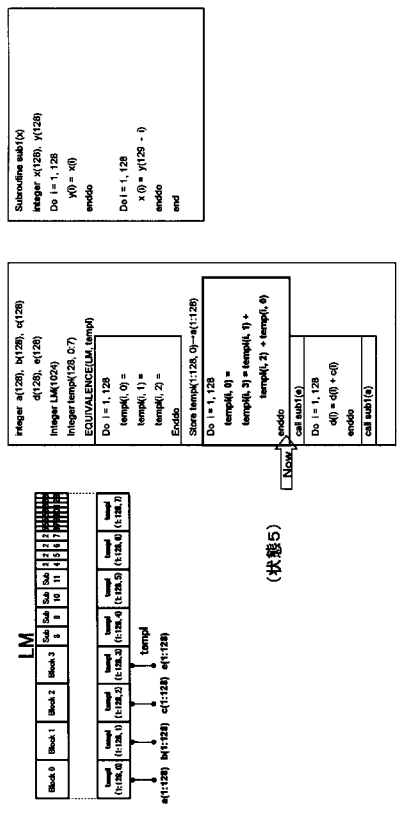
（状態3）

【図２２】



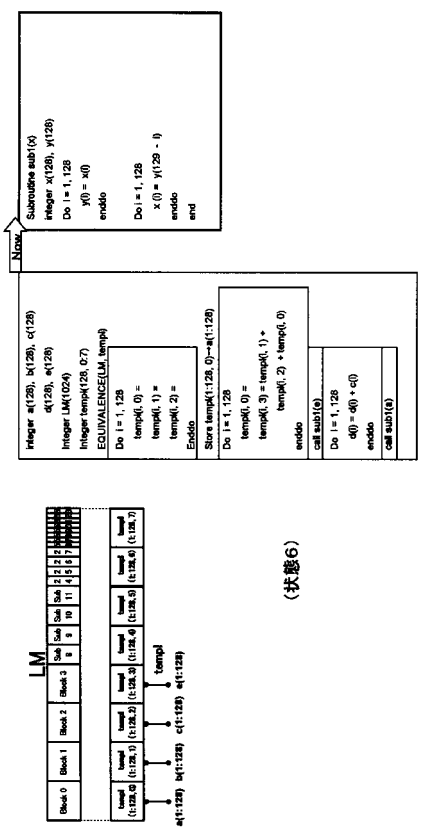
（状態4）

【図２３】



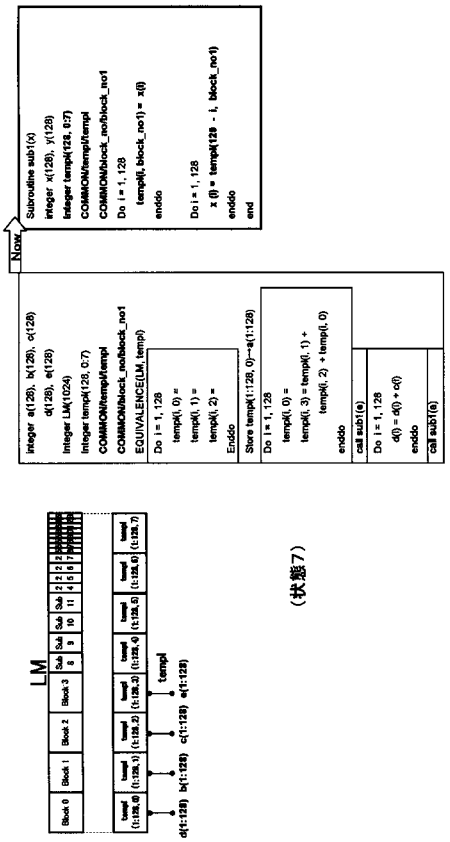
（状態5）

【図２４】



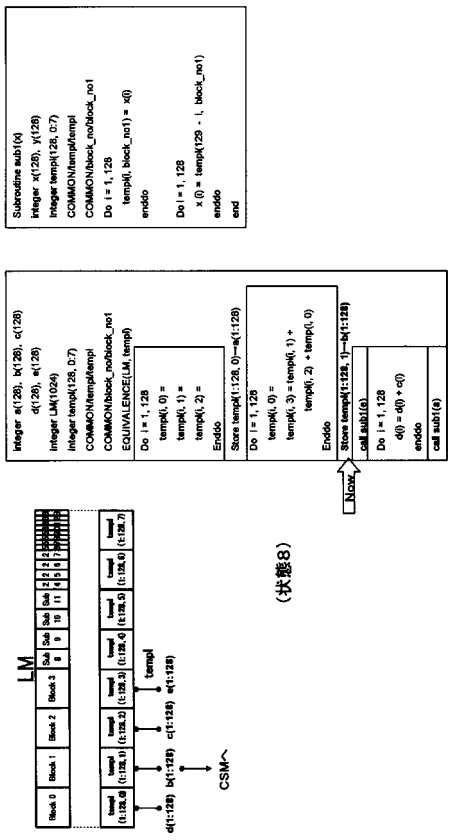
（状態6）

【図２５】



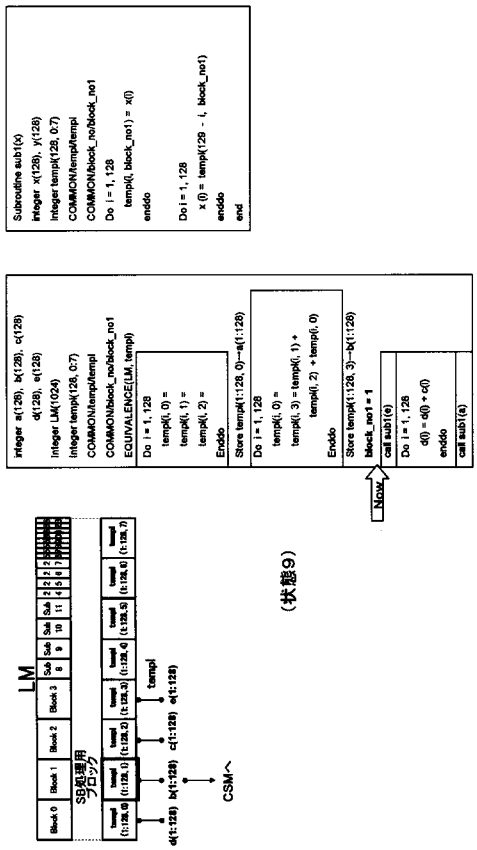
（状態７）

【図２６】



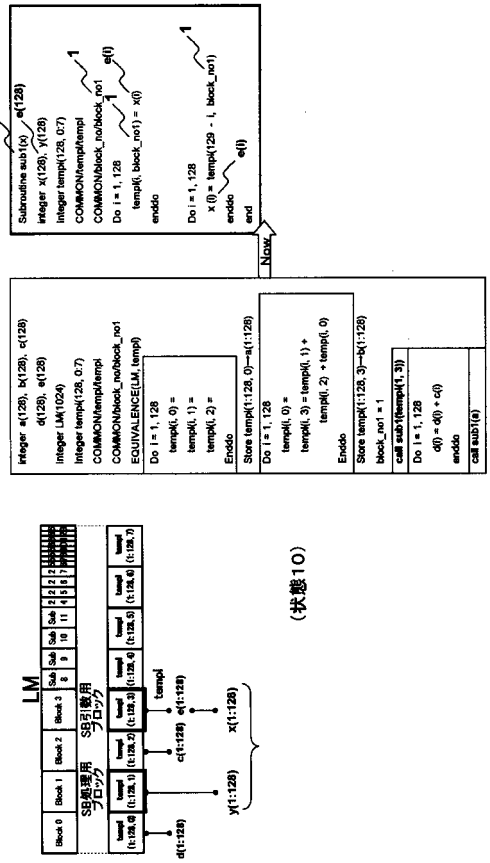
（状態８）

【図２７】



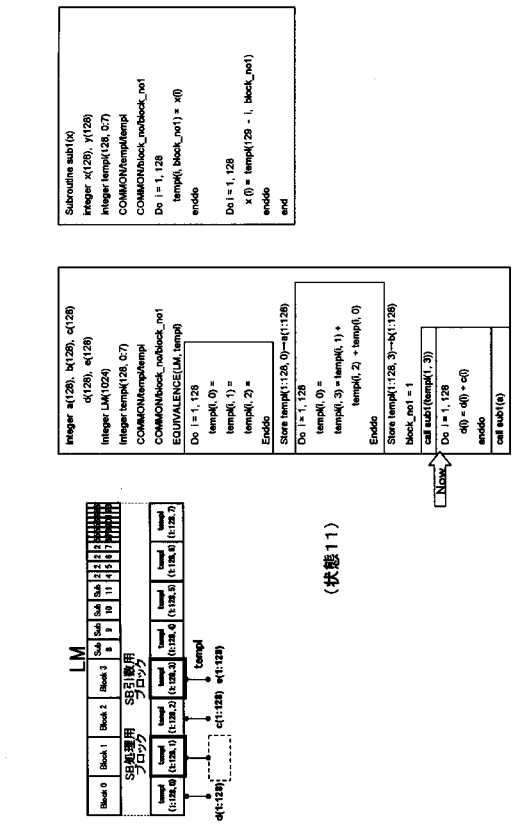
（状態９）

【図２８】



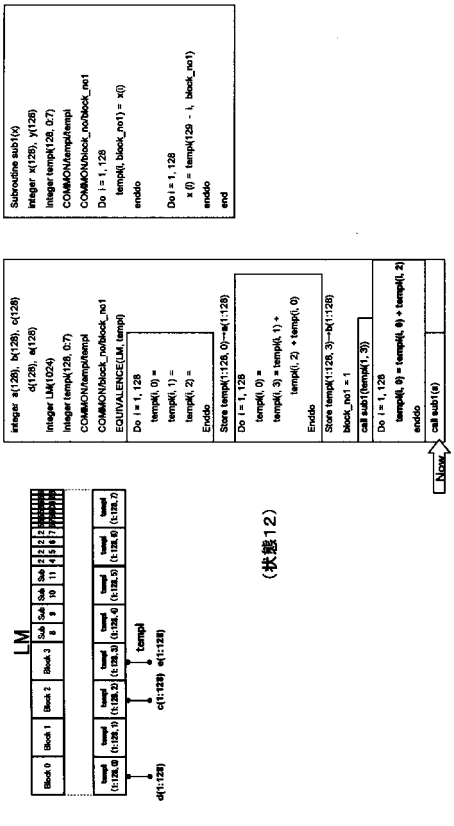
（状態１０）

【図２９】



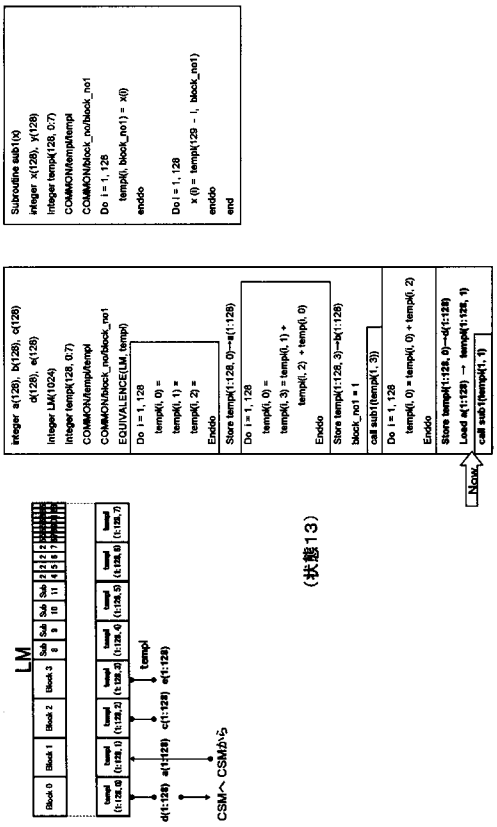
（状態１１）

【図３０】



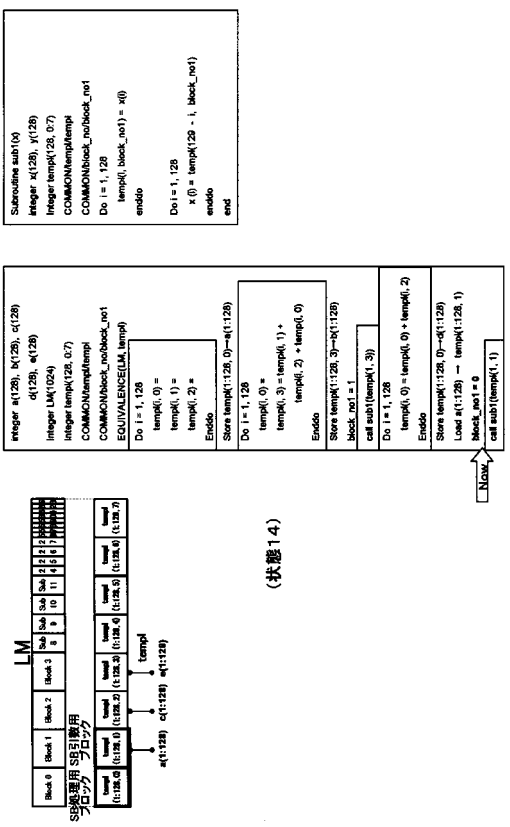
（状態１２）

【図３１】



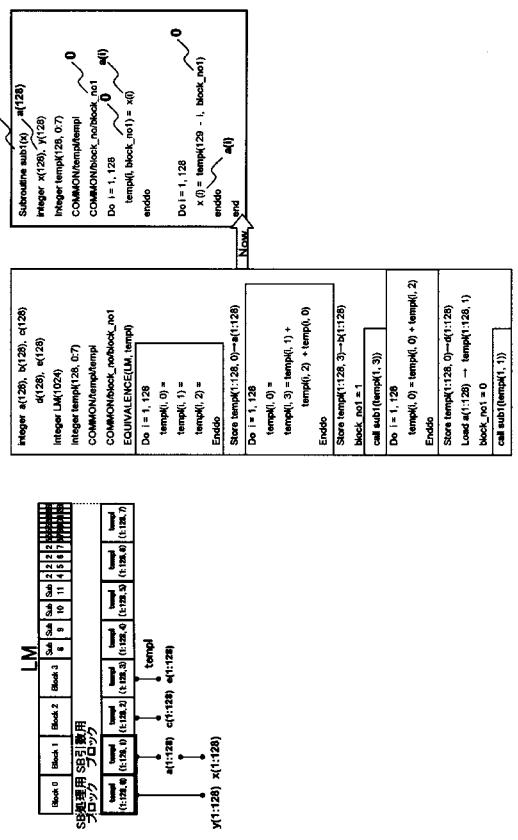
（状態１３）

【図３２】

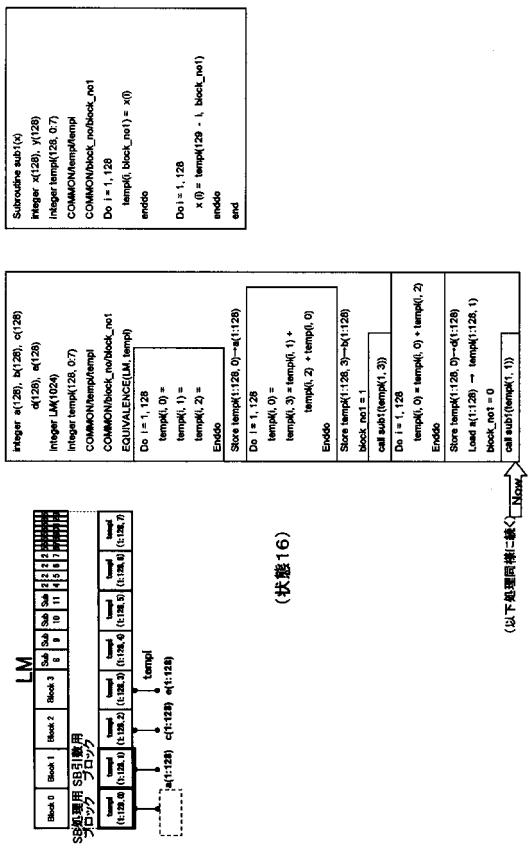


（状態１４）

【図３３】



【図３４】



(状態16)

(以下処理同様に繰り返し)

フロントページの続き

(72)発明者　木村　啓二
東京都新宿区新大久保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　中野　啓史
東京都新宿区新大久保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
東京都新宿区新大久保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
東京都新宿区新大久保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
東京都新宿区新大久保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
東京都新宿区新大久保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

Ｆターム(参考) 5B060 AA12　AA16　AB14　KA02　KA06　KA08
5B176 AB08　AB15