システムソフトウェアと
オペレーティング・システム 70－12
（１９９５．８．２３）

ユーザ・カーネル間の協調スケジューリング

相場 雄一　青木 久幸　中崎 良成
aiba@csl.cl.nec.co.jp
日本電気(株)　C&C 研究所

並列機の普及に伴い、多数の並列プログラムが一度に動作できる並列 OS が一般的となった。ユーザはプログラムを並列処理させるため、並列実行できる単位(タスク)を切り出す。並列 OS では、各 CPU へ割り当てる処理単位(スレッド)をカーネルが用意し、ライブラリ層のスケジューラによりタスクがスレッドに割り当てられて並列処理される。しかし、カーネルとライブラリのスケジューリングが矛盾する場面が見られ、思わぬ性能低下を引き起こすことがある。そこで、性能低下の起こる場合を調べ、協調したスケジューリングを実現する方式を考察した。また、協調スケジューリングの機能を SX-3 の OS に実装し、実機によって効果が測定できたのでこれを報告する。

Cooperative Scheduling between User and Kernel

Yuichi Aiba　Hisayuki Aoki　Ryousei Nakazaki
C&C Laboratory, NEC corporation

As multiprocessor computers are getting popular, parallel OS's are generally used on which many parallel programs can run at a time. To parallelize a program, a user devides it into tasks which can be processed in parallel. Being a parallel OS, the kernel provides threads as CPU allocation units and tasks allocated onto the threads by a library scheduler. But, scheduling conflicts sometimes occurred between kernel and library, therefore, unexpected low performance was derived. Then, we studied such a case and designed cooperative scheduling. Besides, we implemented it in SX-3 OS, and report performance data derived by evaluation on a real machine.

1 はじめに

並列計算機の普及に伴い、マルチスレッド機構を採用する並列オペレーティングシステムが発展してきた。このような環境でユーザが並列プログラムを動作させることが一般的となりつつあり、今後さらに様々な条件の下で並列プログラムが動作する状況が考えられる。そのような状況で並列処理の様々な利用形態を考えた場合、従来のスケジューリング方式ではユーザの指示する並列プログラムが想定した通りに動作せず、処理効率が低下する場合がある [1]-[4]。

現在の並列 OS のスケジューリングは、プログラム中の指示に従って行なわれるライブラリ層でのスケジューリングとカーネルによるスレッドスケジューリングの 2 階層で実現されている。ところが、実際にはカーネルのスレッドスケジューリングは、ライブラリ層でのスケジューリングをほとんど無視して行なわれ、さらにカーネル内の動きはユーザ空間中には見えない。つまり、2 階層でのスケジューリングに協調性がなく、このため想定した通りの並列処理が行なわれないことがある。

本論文では、上記の問題が実際にどのような場合に発生するかを NEC 製スーパーコンピュータ SX-3 について調べ、解決方法の 1 つとして考察した協調スケジューリングについて報告する [6]。また、本協調スケジューリングを、SX-3 の OS である SUPER-UX への実装を含めて行なった。更に、この機能を追加したことによる効果を SX-3 の実機を使って測定評価を行なったので報告する。

本協調スケジューリング方式が実装可能なマシンの形態としては、SX-3 のような共有メモリ型マシンを想定している。本協調スケジューリング方式では、スレッドを低コストで他 CPU に移動して実行出来ることを前提としており、スレッドのマイグレーションが低コストで実現できる共有メモリ型マシンが当面の対象となる。

2 従来のスケジューリング方式

ここでは従来の並列処理におけるスケジューリング方式を説明し、その問題点を明らかにする。

2.1 スケジューリングの形態と発展の背景

あるプログラムを並列処理したい場合、そのプログラムを、並列に処理できる単位に分割して複数の CPU を割り当てて実行させる必要がある。まず、プログラムの分割はユーザが行ない、タスクと呼ばれる一連の処理単位に分割する。この際、それぞれのタスクがどのように動くべきかを指示してやる必要がある。これをユーザが指示するスケジューリングと呼ぶ。これに対し、マルチスレッド機構を採用する並列 OS のカーネルでは、処理を CPU へ割り当てる単位としてスレッドと呼ばれるカーネルの管理単位が設定されている。

並列処理における最も原始的なスケジューリング方式は、図 1のように 1 つのタスクを固定的に 1 つのスレッドに対応させて処理するというものである。この場合、スレッドを CPU 上で切替えることがすなわちタスク処理の切替となる。逆に言えば、ユーザの指示によってタスクを切替えるためにはカーネルを呼び出してスレッドを切替えなければならず、ユーザが必要とするスケジューリングの機能を全てシステムコールとして用意してやる必要がある。また、マルチユーザ環境では、カーネル内での処理を他プロセスから排他する必要があるため、カーネルによるタスク切替えの処理が重くなるという問題がある。更に、タスクと同数のスレッドを用意しなければならず、マルチユーザ環境を考えると、1 つのプログラムに与えることのできるスレッド数は限られ、タスクの分割数の上限が低く抑えられてしまう。

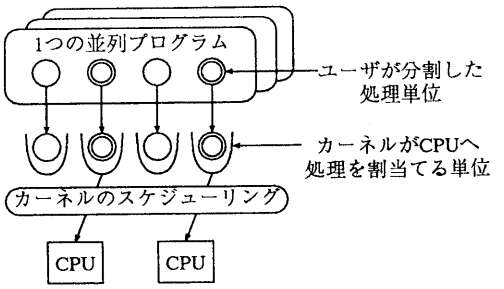

Figure 1: 原始的な並列処理におけるスケジューリング

そこで、スケジューリングの機能を、カーネルとライブラリ層で役割分担する方式が考えられた。まず、スケジューリングの機能全体を、1 つのプログラムに閉じたスケジューリング機能とプログラム相互のスケジューリング機能に分ける。後者の場合は、各プログラム内の資源保護のため相互排他する必要があるので、カーネルの機能として残すが、前者はライブラリの機能として用意すれば、ユーザ空間内で高速な処理の切替を行なうことができる。このように、スケジューリングを 2 階層で役割分担した方式では、タスクを切替えるコストを軽くできるだけでなく、1 つのスレッド上で多数のタスクを切替えるので、スレッドの数とは関係なくタスクを分割でき、タスク分割数の上限を引き上げることができる (図 3)。しかし、この 2 段階のスケジューリング方式にも以降で述べる問題点が存在する。

2.2 2 階層スケジューリングの問題点

ここでは現在主流となっている 2 階層のスケジューリング方式の問題点について大まかに分類して説明する。

(1) **スレッド休眠による実行並列度低下の問題**
スレッドはカーネル内でのスケジューリングにより、様々に状態を変化させる。例えば、(a)CPU 上で処理されている状態、(b)CPU 上で処理されるのを待っている状態、(c)IO や排他制御によるイベントの発生を待ち合わせている状態などがある。我々が以降で述べる実行可能なスレッドとは (a), (b) の状態のスレッドを示す。(c) の状態はイベントが発生するまでは CPU 上で処理されない休眠状態と呼ぶ。

実行並列度とは、プログラムの実行時に割り当てられているスレッドの内、実行可能なスレッドの数と定義する。すなわち、ある時点で実際に処理が進行しているタスクの数を示し、休眠状態のスレッドに割り当てられているタスクは処理が停止しているものと見なす。実行並列度は、スレッドの状態によって刻々と変化する要素であるが、2階層スケジューリングでは実行並列度の変化はユーザには分からない。

例えば、処理中のスレッドの1つがIOを発行すると、IO完了待ちでスレッドが休眠状態となり、実行可能なスレッドの数が減る。しかし、このスレッド休眠はユーザからは分からず、一時的に並列度が下がったような状態のまま処理が継続される。これが実行並列度の低下である。このことは、IOだけでなくカーネル内部での排他制御によりスレッドが休眠状態に陥るようなことがあれば同様の事態を招く。

(2) **ユーザ同期区間への外乱発生によるCPU浪費問題**
ここで問題とされるユーザ同期とは、ユーザ空間中でビジーウェイトによって同期を取る場合である。この場合、同期区間に割込などの外乱が起きると、カーネルはユーザ空間中の処理状態とは関係なく割込処理を優先することがある。この場合、割込まれたタスクの同期区間中の処理が後回しになり、1つのタスクだけ同期点までの到着が遅れることになる。このことで、他のタスクの同期待ちが延長され、無駄なCPU時間を浪費することがある。

(3) **並列プログラム実行過程でのスレッド配分の問題**
現在、プログラムの実行全体を通しては、スレッドとタスクの対応関係が最適化されていない。例として図2のようなプログラムを考える。プログラムは最初1つのスレッドで動作しており、ある時点で2つのタスクに分割される。各タスクは、DOループを並列処理するために更にタスク分割するが、できるだけ多くのスレッドを生成して並列度を上げようとする。このプログラムがスレッドを3つまで使用できるならば、最初に到達したDOループ1では2つのタスクで並列動作する。しかし、後から到達したDOループ2のタスク分割では、スレッドを1つしか取れないので並列処理できない。この状態はDOループ1が終了しても継続する。スレッドとタスクの対応関係を最適化すれば、プログラム全体の処理時間を短縮することは可能である。

これらの問題点の根本的原因を考えてみると、結論的に以下の2つ原因が考えられる。

- カーネル内のスレッドディスパッチがユーザプログラムの想定するスケジューリングをほとんど無視している。
- ユーザからはスレッドの状態に関する情報が全く得られない。

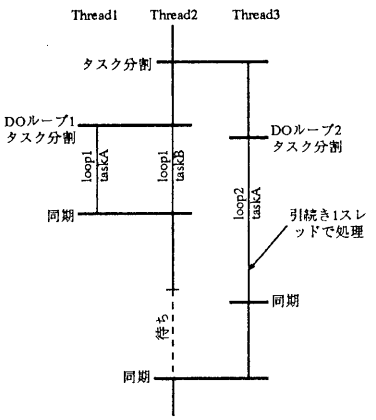

Figure 2: プログラム実行時の不適切なスレッド配分

ユーザのスケジューリングに従ってライブラリがタスクをスレッドに割り付けても、カーネルはそのスケジューリングを無視してスレッドをCPUに割り付け、しかも、どのようにCPUに割り当てられて動いているかはライブラリには見えない。つまり、カーネルとライブラリ層の2階層のスケジューリングの間で協調性がなく、このために、実際にはユーザの意図した通りに処理されないことがあり得る。以下、(1)のスレッド休眠の問題について例を用いて問題点を見ていく。

2.3 スレッド休眠による実行並列度低下問題

図3の例によって具体的に説明する。1つのプログラムで4つのタスクを使って並列処理している様子を示す。この例では、2つのスレッドを生成し、2つ以上のタスクがライブラリ層のスケジューラによって、ユーザ空間中で2つのスレッド上にスケジューリングされる。ここで、タスクの1つがIOを要求すると、そのタスクのスレッドがカーネル内でIO待ちを行ない、休眠状態となって停止する。この時、カーネルは別のスレッドにCPUを割り付けるが、このプログラムにとっては実行可能なスレッドの数が減少しているので、他に実行させたいタスクが存在していても、実行させることができない。実質的にこのプログラムの実行並列度が下がることになる。

IOについては、ユーザが明示的に書いた意図的なIOならば非同期IOなどの仕組みを作ればこの問題を回避できるが、IOの中にはユーザが意識していないものもある。例えば、ページフォールトなどであり、これはメモリサイズによってどんなタイミングでも起こり得るし、今のままではユーザには全く認識できない。また、スレッドがカーネルのディスパッチ対象から外されるのはIO待ちの時だけでなく、カーネル内共有変数のアクセス競合で待つ場合も同様の事態を招く。

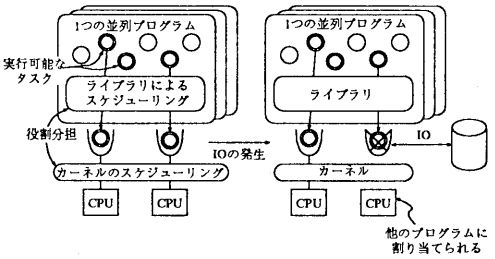

Figure 3: IO 発生による実行時の並列度低下

3 効率的なスケジューリング方式

ここでは、前節で述べたいくつかの問題点の内、スレッド休眠による問題について、効率的な並列処理方式を考察し、協調スケジューリング方式の基本的な方針を示す。

3.1 実行並列度安定化の必要性

前節で述べた通り、スレッドの1つがIO などで休眠状態となると実行並列度が下がる処理時間が長引く問題がある。これに対する簡単な解決法として、IO 発行時に新たにスレッドを生成しこれにタスクを割り当てて実行させるという方法が考えられる。しかし、単純にこの方法のままでは、IO が完了した際に休眠していたスレッドがカーネルのディスパッチ対象として復帰し、実行可能なスレッドの数が増え過ぎてしまう。この状態は、そのプログラムから見ると並列度が高くなっていて効率が良く思われるが、以下のような問題がある。

- プログラムの持っているスレッド数がCPU 数よりも多くなると、プログラム内でカーネルによるコストの重いスレッド切替えが生じ、処理効率が悪化する。
- 1つのプログラムに対して実行可能なスレッドの数が多過ぎるため、他のプログラムの実行に影響を及ぼす。

つまり、プログラムの実行に際して最適なスレッド数というものが存在する。そこで、これに対して、想定並列度という新たな概念を定義する。ユーザが並列プログラムを書く場合、ある並列度を想定してプログラミングを行なうが、このユーザの想定する並列度を想定並列度と呼ぶことにする。通常、想定並列度はCPU 数となっており、並列プログラムの実行中は、実行可能なスレッド数の変動を抑え、実行並列度を想定並列度に安定させる必要がある。こうすることで、他のプログラムに対して影響を与えずに、安定した処理効率を得ることができる。

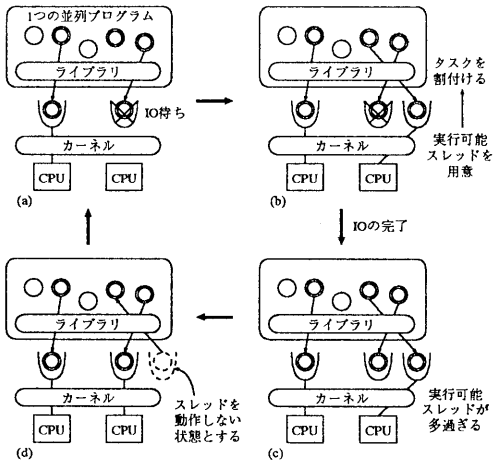

Figure 4: 実行並列度を一定に保つスケジューリング

3.2 協調スケジューリング方式

1つのプログラムの実行可能スレッド数を、一定に保つスケジューリング方式を考える。そのためには、ライブラリとカーネルでのスケジューリングで相互に情報を交換することによって、実行可能なスレッドの数の増減を行なう必要がある。これを実現する協調スケジューリング方式の基本的な方針を以下の通りとする。

(1) スレッドの1つが休眠状態になろうとする際に、新たに実行可能スレッドを用意する。

(2) これに対し、ライブラリは実行可能なタスクを割り当て処理させる。

(3) 休眠状態となったスレッドが実行可能状態に復帰した際，実行可能なスレッドの数が多過ぎれば実行可能なスレッドの1つを動作しない状態とする。

大まかな流れを図4に示す。並列プログラムが2つのスレッドを使って並列処理しているが、スレッドの1つがIO 待ちによりカーネル内で休眠した際 (a) の処理の様子を示す。この時、実行並列度が低下しないように、カーネルが実行可能なスレッドを用意し、他のタスクを割付け処理させる (b)。その後、IO が完了し休眠状態のスレッドが再び実行可能スレッドとして復帰すると、実行可能なスレッド数が増えてしまう (c)。このままでは、他のプロセスに影響を及ぼすなど前節に述べた問題があるので、実行可能スレッドの1つを動作しない状態とし、実行並列度を2に戻す (d)。以上により、実行並列度を安定させた動作が行なわれる。

4 協調スケジューリングの実現方法

前節で述べた協調スケジューリング方式の方針に基づいてSX-3のOSであるSUPER-UX上への実装方式を検討した。

4.1 待機状態スレッドの新設

協調スケジューリングの基本的方針では、実行可能スレッドの1つを動作しない状態にする場面がある。カーネルでは、スレッドの動作しない状態として休眠という状態が用意されているが、本協調スケジューリングでは、これとは別に待機状態という新たな状態を設定した。待機状態スレッドの操作は主に以下の場面を想定して行なわれる。

- 待機状態のスレッドを実行可能状態に移行
  実行可能状態のスレッドがIOなどにより休眠状態になる場合、実行可能なスレッドの数が想定並列度よりも少なくなるが、この時、待機状態のスレッドを実行可能状態に移行する。
- 実行可能状態のスレッドを待機状態に移行
  並列プログラム実行中には、実行可能状態のスレッド数が想定並列度よりも大きくなることがある。このような場合、実行可能状態のスレッドを待機状態にして、想定並列度を維持する。

スレッドの生成コストが重いため、一度生成したスレッドは、使用しなくなっても待機状態で確保しておき、必要な時に再利用する。また、1つのプログラムに対し、待機状態のスレッドの数が増え過ぎることは望ましくないので、1つのプログラムで生成できるスレッドの数に上限を設ける。現在は、スレッド生成数の上限はユーザが変えることができるようなインターフェースを試験的に用意しておく。

4.2 タスクの割り付け

カーネルはスレッドが休眠する際に、実行可能状態のスレッドを用意する。このスレッドには実行可能なタスクを割り付ける必要があるので、ライブラリ層のスケジューラによってタスクを選び出す必要がある。そこで、実行可能状態として用意するスレッドは、ライブラリ層のスケジューラを呼び出すように初期化しておく。

4.3 実行可能スレッドの削減

休眠状態からスレッドが実行可能となり、実行可能スレッドの数が多くなり過ぎると、削減してやる必要がある。

理想的には休眠状態からスレッドが実行可能となった時点でユーザのスケジューラが動作し、スレッドからタスクを切り離し、そのスレッドを待機状態とすることが望ましい。そのためには、スレッドの動作を途中で止め、タスクを切り離すためその時のスレッドのコンテキストを全てライブラリ層のスケジューラで記録する必要がある。しかし、SX-3のようなベクトルマシンの場合、このコンテキストの量が膨大であるため、記録のためのオーバヘッドが無視できない。

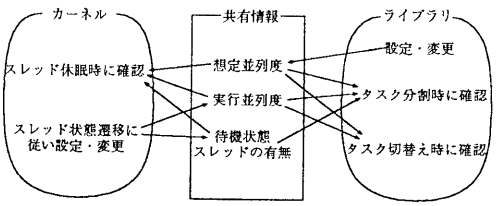

Figure 5: 実行並列度安定のための情報共有

そこで、タスクが切り替わる時点でスレッドを待機状態にする。この時点ならばタスクをスレッドから切り離すことが低コストで行なえる。タスクが切り替わる際には、次のタスクを選択するためにライブラリ層のスケジューラが動作するが、この時に実行可能なスレッド数が想定並列度を越えていないかどうか調べる。もし実行並列度が大きい場合には、次のタスクを割り付けず、スレッドを待機状態とする。ライブラリからスレッドを待機状態にするには、システムコールを発行することになるので、新たなシステムコールを用意する。

4.4 共有情報

前節までの処理の説明中で述べたように、いくつかの情報をライブラリとカーネルの両方で参照している。協調スケジューリング方式を実現するためには、カーネルとライブラリ層で以下の情報を共有する必要があり、図5に参照の仕方をまとめる。

- ユーザの想定する並列度
- その時点の実行並列度
- 待機状態のスレッドの有無

4.5 情報共有の方法

カーネルとライブラリで情報を共有する方法としては以下の方法が考えられる。

(a) カーネル、ユーザ空間の双方に変数を置き、変数操作の時点で同時に値を変化させる。

(b) カーネル空間中に変数を置き、ユーザ空間からはシステムコールで読み出す。

(c) カーネル空間中のユーザから見える領域に変数を格納し、ユーザ空間からはポインタを辿って値を読む。

上記の方法の中で、(b)の方法では値の参照の都度システムコールのコストがかかるので望ましくない。(a)については、ユーザ空間中に変数を置くことによってプログラム実行の安全性を保証できない場合があるという

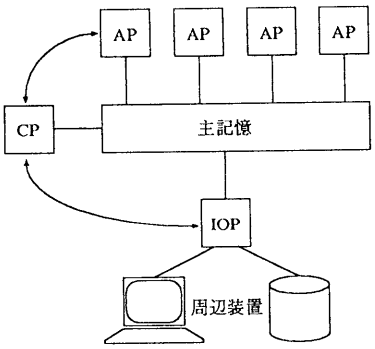

Figure 6: SX-3 のハードウェア構成

危険性をはらんでいる。(c) については、変数をカーネル空間中に置くので安全性は確保できるが、システムによってそういった領域を用意していない場合があり、その場合は不可能となる。SX-3 の OS での実装方法を、前節で示した各情報について以下に示す。

想定並列度は、値が頻繁に変化する性質ではないので (a) の方法をとる。プログラムの実行初期と値の変更時にシステムコールでカーネルに知らせる。

実行並列度は、カーネル内部の操作で頻繁に変化する性質があり (c) の方法をとる。プログラムの実行初期にカーネルに領域を用意してもらい、ユーザ空間からポインタを辿って随時値を読む。

待機状態のスレッドの有無については、スレッドの状態が変化する時、確実にカーネルとライブラリ層の間の往来があるので (a) の方法をとる。

5 SX-3 実機による評価

SX-3 の実機を、協調スケジューリング方式を組み込んだ OS と従来の OS で立ち上げ、同じアプリケーションを動作させ比較評価を行なった。

5.1 評価環境

図 6 に SX-3 のハードウェア構成の概略を示す。AP が演算プロセッサであり、内部にスカラユニットとベクトルユニットを持っている。今回評価に用いた SX-3 では、AP が 4 台主記憶に接続されている。CP は周辺装置制御用のプロセッサであり、AP とは全く異なるアーキテクチャのものが使われている。周辺装置は IOP 経由で接続されている。AP から入出力を行なう場合、一旦 CP へ通知し、CP が IOP にチャネルプログラムを渡す。逆に、入出力の完了は IOP から CP に通知され、必要に応じて CP は AP に通知を行なう。

評価に使うプログラムとしては、スレッドがカーネル内で休眠する状況を作り出すため、IO を行なうプログラムを用意する。

今回評価に使ったプログラム d2, d4 の動作の概略を図 7 に示す。プログラムは、最初にファイルからデータを読み込み、続いて計算を行なうというプログラムであり、いくつかのタスクに分割して並列実行する。d2, d4 ともに最初は 4 つのタスクに分割され、他にプロセスがなければ 4 つの AP 上で並列動作する。更に、d2 では 2 つのタスクに分割され、d4 では 4 つのタスクに分割される。

また、各プログラムについて入力データ量を変化させたものをいくつか用意する。ただし、IO の量だけを変化させ、演算量は一定となるようにテストプログラムを加工しておく。

5.2 評価結果

d2, d4 の 2 つのプログラムを用い、SX-3 実機上で評価した結果を図 8 に示す。縦軸にプログラムの処理時間、横軸に入力データ量をとり、2 つのプログラムの処理時間を、従来 OS 上で測定した場合と協調スケジューリング方式を実装した OS の上で測定した場合を比較した。

図中の cnvd2, cnvd4 は従来 OS 上で各々 d2, d4 を動作させた時の結果であり、cpd2, cpd4 は協調スケジューリングを実装した OS 上で各々 d2, d4 を動作させた時の結果である。従来 OS 上では d2, d4 とも処理時間に変わりがないが、協調スケジューリングでは処理時間が短縮されていることが確認できる。

ここでプログラムに手を加え、AP 台数の倍の 8 つの実行可能スレッドを割り当てるようにしたプログラムを作り、従来 OS 上で動作させたところ、図 8 中の cnvd2_tune, cnvd4_tune のような結果となった。プログラムに手を加え、IO に備えて多めの実行可能スレッドを割り当てておけば協調スケジューリングと同等の処理時間となる。

更に、従来 OS 上のプログラムについて、プログラムに割り当てる実行可能スレッド数を変化させて測定した。プログラムは d4 を使用したところ、図 9 の cnv に示すような結果となった。スレッド数が 4 ～ 8 ではスレッド数を増やすことによって処理時間が短縮されているが、それ以上スレッドを割り当てても意味のないことが分かる。同図の cp は協調スケジューリングを実装した OS 上での結果であるが、スレッド数の意味が少し異なる。ここに示すスレッド数とは、1 つのプログラムで生成することのできるスレッド数の上限である。実際にはその値までスレッドが生成されているとは限らない。協調スケジューリングについてはスレッド数の上限を変化させても処理時間に影響はない。

5.3 結果の考察

評価では、IO を行なうテストプログラムを用意し、協調スケジューリングやプログラミングの工夫によって、処理時間が短くなることが示された。ここでは、どのような原理によって処理時間が短縮されるかを考察する。

テストプログラム d4 について、理想的な動作の概略を図 10 に示す。d4 では各 AP 上で動くタスクが更に 4 つのタスクに分割される。それらを task1 ～ 4 と示す。各タスクでは最初にデータ入力を行ない、引き続いて演

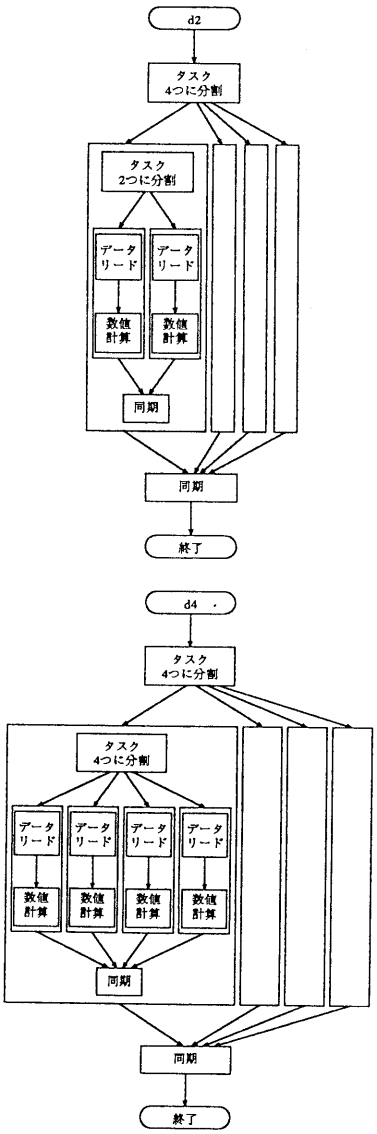

Figure 7: プログラム d2, d4

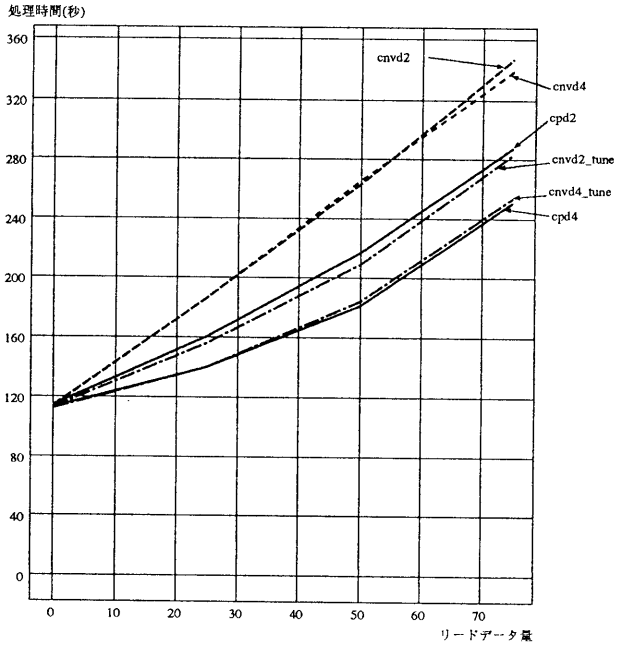

Figure 8: 処理時間と入力データ量の関係

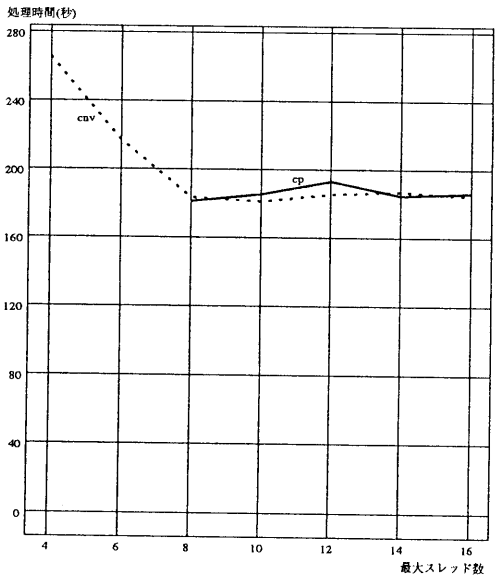

Figure 9: 処理時間とスレッド数の関係

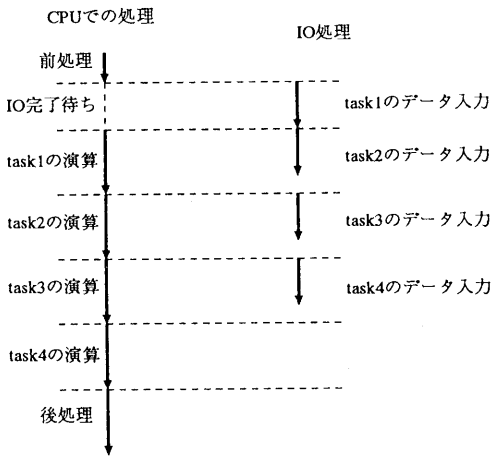

Figure 10: d4 における理想的な動作の様子

算を行なう。データ入力の際には、タスクがIO 完了待ちとなるが、この時、他のタスクが処理可能であれば、図に示すようにデータ入力と演算処理とが同時に処理される部分があり、IO 待ち時間が隠蔽される。協調スケジューリングを組み込んだ場合や実行可能スレッド数を増やした場合にこれが可能となり、IO 待ちの隠蔽の時間だけプログラムの処理時間が短縮される。

この図によると、演算によって隠蔽されないIO 待ちの時間はtask1 のデータ入力の時間となる。つまり、この部分のIO 時間をできるだけ短くすれば処理時間を短縮できる。従って、プログラムをできるだけ細かく分割してtask1 のIO の時間を短くした方が処理時間短縮の効果が大きくなり、これがd2 とd4 の処理時間の差となっていると考えられる。

また、図10では演算時間の方がデータ入力の時間よりも長いものとして示してあるが、逆にデータ入力時間の方が長くなると、IO 時間の隠蔽部分が演算時間と等しくなる。今回の評価では演算時間を一定としているので、データ入力時間が演算時間よりも長くなれば、処理時間の短縮分は一定となる。結果の図8で、各グラフが途中から平行になっているのは、このためである。

以上の考察で、協調スケジューリングあるいは割り当てる実行可能スレッド数を増加することによって、IO を含むプログラムの処理時間を短縮するメカニズムが示された。この評価では、割り当てる実行可能スレッド数を増加することによって協調スケジューリングと同様の効果があることが分かるが、1 つのプログラムで余分に実行可能スレッドを使うことによる影響も合わせて考察する必要がある。

まず、実行可能スレッドを余分に使うことによって、プログラム内で重いスレッド切替が生じ、処理効率が悪化する。しかし、この評価では処理の切替の重さを論じるような時間尺度で測定していないので、評価結果には見えてこない。また、実行可能スレッドを余分に使うと、他のプログラムの動作に影響を及ぼすことが考えられるが、今回の評価では他にプログラムが動いていない静的な環境で実験を行なったので、他のプログラムに対する影響が見える形での評価も必要と思われる。

プログラムに割り当てる実行可能スレッド数というパラメータは、他のプログラムの実行にも影響を及ぼすという点で、本来、ユーザが自由に選べる性質のパラメータではない。しかし、これを一律にある定数に固定しようとすると、従来のOS ではどのような値を選択するかで一長一短が生じる。この点、協調スケジューリングでは実行可能スレッド数を実行時に最適化するので、上記のようなパラメータをユーザが意識する必要はない。協調スケジューリングでは、1 つのプログラムで生成することのできるスレッド数の上限というパラメータを設けているが、これはシステムの運用管理上のパラメータとして一律にある定数に決めておくことが可能である。

6 むすび

本論文では、ユーザプログラムによって明示的にIO を発生させるという環境を作り、SX-3 の実機上の動作において協調スケジューリングの効果を確認した。

References

[1] Brian D. Marsh, Michael l. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos: "First-Class User-Level Threads", Proceedings of the 13th ACM Symposium on Operating Systems Principles (Oct. 1991).

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy: "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", ACM Transactions on Computer Systems (Feb. 1992).

[3] Paul Barton-Davis, Dylan McNamee, Raj Vaswani, and Edward D. Lazowska: "Adding Scheduler Activations to Mach 3.0", Proceedings of the 3rd USENIX Mach Symposium (Apr. 1993).

[4] 岡坂 史紀, 清水 謙多郎, 芦原 評, 亀田 寿夫: "マイクロプロセス：カーネルに支援された利用者レベルスレッドの設計と実現", コンピュータシステム・シンポジウム論文集, 情処シンポジウム論文集 Vol. 93, No.7 (Oct. 1993).

[5] "SUPER-UX FORTRAN77/SX 並列処理機能利用の手引", NEC, SX システムソフトウェア, GUF25-1 (1992).

[6] 相場 雄一, 青木 久幸: "並列ジョブ効率的実行のための協調スケジューリング", 情報処理学会 第48回全国大会 講演論文集 (4) (Mar. 1994)