3P−2

共有メモリ型並列機のためのアクティビティ方式並列実行機構 (2)
— 環境切替により後処理実行を行なう方式の提案 —

小林健一, 中山泰一, 永松礼夫, 森下巖 (東京大学工学部)

1　はじめに

共有メモリ型並列計算機上で多数の細粒度プロセスを効率良く処理するために、プロセス生成要求と実行を分け、あらかじめプロセッサと同数だけ用意されていた軽量プロセスを実行時に割り振って利用するというアクティビティ方式が提唱されている [1]。

しかし、fork-join 形式の並列プログラムにおいては、子タスク完了の待ち合わせにより、多数のサスペンドが発生する。タスクがサスペンドする場合にはプロセス実体が新たに生成され効率が低下する。

この場合の効率の低下を防ぐための二つの改良案がある。一つは親タスクの処理から後処理を明示的に「遺言」として分離する方式で、これならば子タスク待ちにおいて新たに軽量プロセスを生成する必要がない [2]。

もう一つの案が本稿で報告する方法である。新たな軽量プロセス生成のコストを軽くすれば効率の低下は少なくなる。そこで、既存のスタック領域上に新しいスタック環境を積み、しかも既存の環境と新しい環境を別個のプロセッサで実行可能にする。

2　環境切替方式の必要性

fork-join 型の並列処理の場合には並列実行要求を出した後、すべての処理が終了するまでサスペンドする必要がある。この時、通常は新たに軽量プロセスを生成して、プロセッサ資源　の有効利用をはかる。しかしこの生成コストが高いものであれば、アクティビティ方式の利点は失われ、通常の軽量プロセスを生成する方法と効率は変わらなくなってしまう。

そこで新たに生成するタスクのスタック領域を、サスペンドしたタスクのスタック領域の上に積む形でそのまま割り当ててやれば、軽量プロセスの生成コストは非常に低くなる。

しかし、ここで重大な問題が発生する。スタック領域の下部にある既存の環境は、上部に積まれた新しいタスクの環境の実行が完了するまでは、実行が再開できなくなる。さらに、積まれた環境の中に (親 → 子 → 親) という構造が入れ子になっていた場合など、別のタスクが上に積まれると、子タスクが終了しても親タスクが再開できなくなり、並列度の低下をまねく (図 1)。

この問題は異なるタスクにスタックを共用させたために、停止した親タスクが元のプロセッサに固定されてしまうことに起因している。親タスクの再開が他のプロセッサで行なえるのならば上記のような問題は全くない。

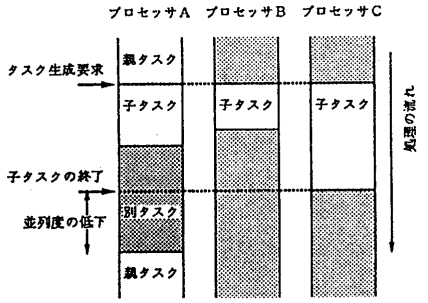

図 1: 直列性を内在する処理

そこで、同期待ち条件が解消した時点、つまり最後に処理を残していた子タスクが処理を終えた時点で、その子タスクを実行していたプロセッサに元の親タスクを移動して、実行再開することにすれば、上述の並列度の低下も発生することはない。

親タスクのプロセッサ間移動をどう処理するかが本稿の主題である。本稿ではローカル変数などのプロセス環境をプロセッサ間で移動する方法を示す。この方法はスタックポインタの切替だけで実現できるので、オーバーヘッドが非常に少なくできる。

3　環境切替方式の実現

ここではタスクのプロセッサ間移動を実現する機構の詳細について述べる。共有メモリ型並列機であるので、コード及びグローバルデータ、プロセス情報は以前の資源をそのまま利用できる。タスク毎に確保されるローカルデータの処理に気をつけなければならない。ローカルデータは対象としたプロセッサにおいてはスタック上に保存され、スタックフレーム毎に順序 (上下) 関係を持つので、単純にポインタを切替えて共有するわけにもいかない。またブロック転送などはコストもかかる上に、アドレス参照を狂わせてしまう弊害を持つ。

今回提案する方法では、スタックにリンク構造を導入する。図 2(a) に示す通り、切替先プロセッサのスタックポインタが切替元の親タスクのフレームを指すようにする。この時に親のフレームの上には当然別のタスクのフレームが位置しているのでここからサブルーチンコールなどで新たにフレームを確保することはできない。そこで、新たなフレームはプロセッサが個々に持つスタック確保ポインタから取るようにする。このポインタはフレームの確保の時にはスタックポインタの

Activity Based Parallel Execution Mechanism on Shared Memory Machines (2)
— A Shared-Stack Context Generating Method —
Kenichi Kobayashi, Yasuichi Nakayama, Leo Nagamatsu and Iwao Morishita (University of Tokyo)

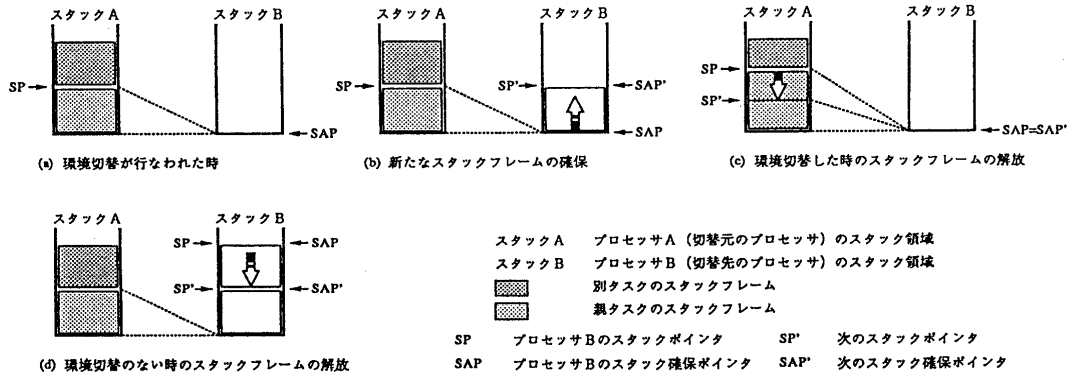

図 2: 環境切替方式におけるスタックの操作

更新後の値と等しくなる (図 2(b))。しかし、解放の時にはそのフレームが環境切替が行なわれたものである場合は変化しない (図 2(c))。こうして常に新たなフレームを得ることができる。なお、環境切替が行なわれていない場合は、スタック確保ポインタはスタックポインタと一緒に後退し、メモリ資源の浪費が抑えられる (図 2(d))。

このスタック構造の変更にともなう時間的オーバーヘッドはわずかである。実際、この変更は多くの プロセッサに おいて、サブルーチンコール一回あたりに高々数命令の追加ですむ (図 3)。またこの変更はすべてのサブルーチンコールにおいて行なう必要はなく、内部に並列処理の起動を含まない木構造の末端にあるルーチンでは必要ない。このことは、従来のライブラリも変更もなく使用できることを意味している。また、この変更自体も、コンパイラにわずかな変更を加えるだけで実現できる。

従来方式	環境切替方式		
call		call	
ret restore		sub cmp be nop ret restore	%fp, FrameSize, %l7 %sp, %l7 LLL %sp, %g0, %sp
	LLL:	ret restore	

図 3: SparcCPU における call-return の変更

この方式には、スタックの回収がおこなわれない領域があるので、従来の方式に比べ、メモリ資源を消費しがちであるという問題点がある。実際に環境切替が起こらなければ、切替にかかるコストもメモリ消費も抑えられるので、切替の発生する数を減らすスケジューリングが重要である。子タスクの負荷が予測できない場合でも、最低限一つの子タスクを親タスクを処理していたのと同じプロセッサに優先的に割り当てることで、最後に実行を終える子タスクと親タスクを処理するプロセッサが同じである可能性が増し、環境切替の発生を少なくできる。

4 実験

シミュレータを用いて、7都市巡回セールスマン問題の処理時間を比較した。8プロセッサでの実験の結果を表1に示す。A方式というのはスタックを共有をするが、下に積まれた環境は実行できない方式であり、B方式というのが今回提案した環境切替方式である。

システム時間がいくぶん増加したもののプロセッサの停止するアイドル時間が減り、総実行時間は短くなっている。

表 1: 巡回セールスマン問題の処理時間

	A方式	B方式
初期化	113443	64120
システム	236988	280225
ユーザ	902802	891107
アイドル	26321	7068
総計	1279554	1242530

5 おわりに

本稿で提案した環境切替方式により、プロセッサ間におけるプロセス環境の高速な切替が実現できた。これによって共有メモリ型並列機におけるアクティビティ方式の性能を向上することができた。

参考文献

[1] 田胡和哉, 檜垣博章, 森下巖, “共有メモリ型並列計算機のためのアクティビティ方式を用いる並列実行環境,” 情報処理学会論文誌, Vol.32, No.2, pp.229–236, 1991.

[2] 中山泰一, 白木光彦, 永松礼夫, 森下巖, “共有メモリ型並列機のためのアクティビティ方式並列実行機構 (1) — 子プロセス待ちにおいて後処理を分割し性能改善を図る方式 —,” 第 45 回情報処理学会全国大会, 3P-01, 1992.