Schematic/DSM

sumio

平成 10 年 10 月 15 日

要旨

このミーティングでは、 Schematic を DSM マシン (具体的には Origin 2000) で、より高速化するための試みと、現時点での進捗を述べる。

1 SWoPP までの話

大山さんが実装している Schematic[2] は、どの PE からでもメモリのアクセス速度が均等な、 SMP マシンを前提としている。私は、その Schematic に手を加えて、 DSM マシンである Origin 2000 で動作させるようにした。

Origin 2000 のような DSM マシンでは、普通の共有メモリのように、 (物理的に) どの PE にあるメモリでもアクセスできるのだが、自分の PE にあるメモリ (ローカルメモリ) と他人の PE にあるメモリ (リモートメモリ) のアクセス速度が異なるという特徴がある。 Origin 2000 では、ローカルとリモートで、約 2 倍のアクセス速度の差がある。

Schematic は、スレッド (POSIX スレッドや Solaris スレッド) を用いて実装されているため、 pthread ライブラリがある機械では、殆ど移植の手間無しに動かすことができる。 Origin 2000 も pthread ライブラリがあったため、移植そのものはすぐに完了した[1]。しかし、このように移植した処理系は、未だにメモリのアクセス速度が均等であると仮定されているので、運悪く、頻繁にアクセスされるメモリがリモートにあったりすると、効率が悪化する恐れがある。

しかし、「どのメモリが頻繁にアクセスされるか」というのを runtime やコンパイラが知るのは大変なので、プログラマに指定してもらうことにした。具体的には、

```
(...
  (let ((cha (future (f x) :on l)))
    (touch cha)
    ...))
```

[1] こまかいバグ取りは大変だったが...

というように、`:on`という修飾子を使って、その関数 (ここでは`(f x)`) をどこで実行して欲しいか指定できるようにした。

また、あるオブジェクトがどの PE のメモリ上にあるか調べるためのプリミティブ(`whereis` l)を導入した。これは、単にオブジェクトを表わす構造体 (Scheme の vector) に、そのオブジェクト作成時の PE 番号を入れておき、それを参照しているだけである。

ところで、`:on`があったら、必ずそこでやろうとすると、`:on`が細粒度に付いていたときに効率が悪化してしまうし、タスクスケジューリングに必要なメモリ量が莫大になる、という問題点があるので、それを解決するために

- `:on`があったら、それぞれのタスクスタックを交換する
- `:on`があっても、一定時間は実際に交換はしない

という方法を採ることにした。その結果、

- キャッシュに入りきらないような大きなオブジェクトを扱うプログラムでは 15–30% の性能向上がみられた。
- キャッシュに入るようなものは、殆ど性能に変化はない。
- 実用的なアプリケーションとして、 n-body を計測したが、性能は悪化してしまった。

ということがわかった。ここまでが SWoPP の話である。

2 :on と GC

いきなり話は変わるが、 Schematic では、 GC を起こすときには、アプリケーションの実行を一旦中断し、 1 台の PE で逐次の copy GC を行っている。さらに、ある PE に live object が集中する状態を避けるために、 PE を跨いで copy を行なっている。すなわち、 GC によって、いままで PE 1 番にあったものが PE 10 番に行ってしまうかも知れないのである。

「だからどうしたの？」と思われるかも知れないが、このことは色々と問題になる。

2.1 whereis が正しくなくなる

さきほど、`whereis`は、オブジェクトに書いてある「PE 番号」を見ているだけである、と書いた。「PE 番号」は、オブジェクト作成時に格納され、以降 GC が起きても変わらないので、一度 GC が起きてしまうと、それから先は`whereis`は間違った情報を返してしまう。そのため、 SWoPP で行った計測では、 PE 間を跨いだコピーをしないように、 GC を改造して行っていた。

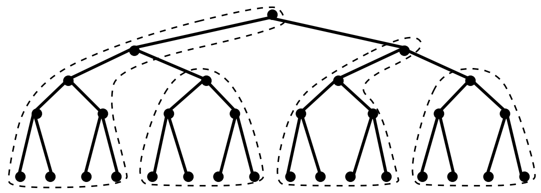

図 1: 二分木を作ったときの、各ノードの PE への割り当て

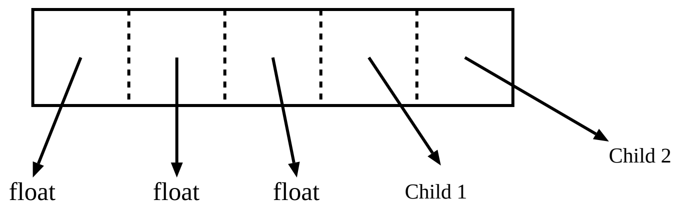

図 2: Schematic におけるオブジェクトの表現
(三次元のベクトルをキーに持つ二分木オブジェクトの場合)

2.2　オブジェクトのローカリティが失われる

ここで言うローカリティとは、「ある場所のメモリを触ったら、その直後に触るメモリも近く (同じ PE) にある」という意味である。

`future`を使って二分木を作るコードを実行すると、 Lazy Task Creation (LTC)[1] のもとでは、 (運が良ければ) 図 1のようになる。この図において、破線で囲まれている部分は同じ PE で実行される (ノードも同じ PE に確保される) ことを表している。

この二分木に対して、深さ優先にトラバースする場合を考える。例えば、 Origin 2000 で、「ローカルメモリの方が速い」ことを利用しようとして「全部のノードにおいて、そこへアクセスする PE は、そのノードの確保されているメモリを持っている PE にしよう」などと決めて実行させたとしよう。あらゆるノードが本当にバラバラな PE に割り当てられてしまっている場合は、ノードを移るたびに実行する PE を変えなけらばならなくなり、ものすごいオーバヘッドが掛かることになるが、この図のようなノード配置であれば、ある程度深い所では、全てのノードが同じ PE に割り当てられているので、ノードを移っても実行する PE を変更しなくても良くなり、結果的に、低オーバヘッドで実行できることになる。

このように、 LTC では、ある程度ローカリティの良い配置をしてくれるのだが、 GC が起こると、これらのノードが、良く分からない場所へコピーされて、結果的にバラバラになってしまう恐れがある。

また、 Schematic のオブジェクトの実体は vector で、それぞれの要素にインスタンス変

数へのポインタ (1 word で表せる場合は実体) が格納されている (図 2)。しかし、 GC からは、これがオブジェクトなのか、「浮動小数点が 3 つと vector へのポインタが 2 つ入っている 5 要素の vector」なのか区別が付かないので、 GC 後には、これらが全くバラバラの PE に配置されてしまう可能性がある。これは、「当然オブジェクトは 1 つの PE にあるだろう」と思っているプログラマの意図に反することになるし、`whereis`も、あまり意味をなさなくなってしまう。

3 解決方法

3.1 オブジェクトを指すアドレスの値から PE 番号を求める

今迄の`whereis`の実装がいけなかった点は、オブジェクトを作った時点の PE 番号を保存してしまったことである、これを解決するには、 GC の度に、その PE 番号を書き換える、という手もあるが、それは面倒なので、オブジェクトを指すアドレスの値から PE 番号を求められるように変更した。

各 PE に確保されているヒープの先頭アドレスと終端アドレスは、共有変数に保存されている。`whereis`を実行するたびに、これらのアドレス全てと比較して、 PE 番号を見つけるのは、あまりに無駄が多いので、各 PE のヒープをまとめて 1 回の malloc で取るようにして、それを各 PE に分割するようにした。これで、`whereis`は引き算と割り算 1 回ずつで値を返せるようになる。

3.2 GC の変更 (未実装)

GC とローカリティの問題については、まだ何もしていないし、サーベイもしていないが、結局、 copy する際の粒度が問題なのではないかと思っている。今では、 vector とか cons cell とかが copy の単位なのだが、これを、もう少し大きな単位でコピーすれば、すくなくとも、その単位の中ではローカリティは保たれることになる。問題は、最適な大きさはどれくらいか、ということなのだが、色々試してみたいと思っている。

4 今後の予定

4.1 GC

とりあえず、オブジェクトの構成要素までがバラバラに配置されるのはいただけないので、 vector に似たデータ構造である「オブジェクト」を新たにタグに追加して、次のような戦略で GC をするように変更しようと考えている。

- オブジェクトからポインタを 1 つだけ辿って到達できるところは、一緒にコピーする。

- 前項において、ポインタを辿った先がオブジェクトだった場合は、そこはコピーしない。

すなわち、図 2の場合だと、浮動小数点の部分は、オブジェクト本体と一緒にコピーされるが、 Child1 と Child2 の部分はコピーされない。

また、最終的はローカリティを保存する GC というのを作って、次いで[2]に Schematic の GC を並列化できたら、と思っている。

4.2 n-body(余談)

SWoPP で実験した n-body の結果が芳しくなかったのだが、その原因として「作った質点がランダムだから、ローカリティが無くなっているのでは」という御指摘を受けたので、物理的に近くの質点はまとめて木に挿入するように、ランダムに作った質点をのログを並び換えてから測ってみたのだが、結果はいままでと全く変わらなかった。

ひょっとしたら、 perl で書いた sort のアルゴリズムにバグがあるかも知れないのだが、本当に正しく並んでいるのか調べる術がなくて困っている。 X で順々に点をプロットしていくプログラムでも作って確かめるのがいいのか？

また、「そのように並べたとして、本当に図 1のように行儀良いローカリティになっているのか」という疑問もある。これも、各ノードの PE 番号をダンプさせて調べようと思っている。

とにかく、どこかに原因あがるはずなので、何とか突き止めたい。

参考文献

[1] Eric Mohr, David A. Kranz, and Robert Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 264–280, July 1991.

[2] Yoshihiro Oyama, Kenjiro Taura, Toshio Endo, and Akinori Yonezawa. An Implementation and Performance Evaluation of Language with Fine-Grain Thread Creation on Shared Memory Parallel Computer. In *Proceedings of PDCS '98*, Las Vegas, Nevada, USA, October 1998. IASTED.

[2] どっちが次いでなのか分からんが...