近況報告

大山恵弘

1999 年 3 月 4 日

1 JSPP に論文を投稿した

大規模共有メモリ計算機上において、ボトルネックになるオブジェクトでのメソッド実行を効率化する方法について論文を書きました。基本アイデアは、オブジェクトのロックの解放を待っているスレッドのコンテキストのキューをオブジェクトから切り離し、1 プロセッサがそれらのスレッドを逐次的に順番に実行する、というものです。書くのに苦労はしましたが、採録されるかどうか、いま一つ自信がありません。でも、採録された場合に備えて、現在、実験データを揃えています。その論文の内容は、近いうちに PL ミーティングで話させていただきたいと思っています。

2 軽いスレッドを作れる Java 処理系を作っている

2.1 概要

Schematic のように、

- Java のスレッドよりも手軽にスレッドが作れて
- 速度面、領域面でのコストを気にせずスレッドが作れて
- 並列計算機上では良いスケーラビリティが得られる

Java 処理系を作り、世界にインパクトを与えたいと考えています。ご存じのように Java のスレッドは、継承やらインタフェースやらが絡んできて使うのがややこしく、Java でアプリケーションを並列化するのはいささか面倒です。加えて、スレッドの生成は、定量的には確認してませんが、だいぶ重いです。そして、すでにランチミーティングで話させていただいたように、世間の並列に注目した Java の拡張はどれも HPF の記憶に縛られており、かなり泥くさい印象があります。

言語設計に関してはひとまず MTCAMP の言語仕様をほとんどそのまま借りてきました。私がより深く Java 言語を知るにつれ、また、処理系の実装が進むにつれ、仕様に関して色々な問題が出てくると思いますが、臨機応変に対処したいと思います。

言語の名前はまだ思案中です。色々な人に相談してそのうち決めます。

2.2 プロトタイプ処理系の実装

プロトタイプを作り終えました。一杉さんの Java プリプロセッサキット EPP を使って Java ソース to Java ソースの変換で実装しました。性能を度外視し、単純に 1 つの細粒度スレッドの生成を 1 つの Java スレッドの生成に対応させます。フィボナッチ数列の変換前と変換後のプログラムを付録に示します。言語仕様や変換アルゴリズムの概要をつかむにはそれらのプログラムを見てください。

プロトタイプを使っていくつかアプリケーションを書き[1]、 celesta 上で下に示す計測結果を得ました。「ちゃんとプロトタイプを実装し終えてます」以上のメッセージはありません。

	fib (15)	qsort (1000)	knapsack (12 items)
Schematic (1 PE)	< 1	8	< 1
逐次版 (JDK1.1)	22	116	42
逐次版 (JDK1.2β)	33	24	69
並列版 (JDK1.1, 1 PE)	60940	7571	294995
並列版 (JDK1.2β, 4 PE)	6730	4524	8276

(単位はすべてミリ秒)

Nqueen や Prime も逐次版は書いたのですが、まだ並列化してません。

実装上の都合で、現在の処理系には下の細かい制限があります。一生懸命対処しても実りがない気がするので放っておき、今後の活動の中心は本番の処理系の実装に移すつもりです。

- fork 文の本体文で private 変数、 this、 super を使ってはならない
- Java のスレッドを使ってはならない

ヨイショ　EPP を使うと容易に Java を言語拡張できます。(私が MTCAMP プロジェクトでフロントエンドの実装で試行錯誤した経験があったからかもしれませんが、) プロトタイプの実装はわずか 1 週間で終わりました。

2.3　本番の処理系の実装

以前ランチミーティングで述べさせていただきましたが、 JVM の拡張は禁断だと感じています。処理系を使ってくれる人が減る他に、なぜプラットフォームの言語に Java を選んだかの動機づけがしにくくなるからです。そこで、初めは javac に手を入れて実装することを考えました。しかし、一杉さんの

> 「Java はソースレベルとバイトコードレベルでできることがほとんど変わらない。バイトコードをいじるとベリファイヤが通らなくなる問題が出てくるので、ソースレベルで変換する手も考えてみては」

という言葉に影響されつつあります[2]。

今は関口さんの JavaGO の手法を使って、ソースレベルでスタック (スレッド) の管理ができないかと検討しているところです[3]。具体的には、ブロックするかもしれないメソッドの呼び出しを try で囲みます。そのメソッドがブロックせず復帰した場合はそのまま実行を続けます。ブロックした場合には、そのメソッドは例外を投げ、制御は上で作った try 文の catch 節に移ります。コンパイラは catch 節に、そのメソッド呼び出し復帰後に使用する変数、仮想的なプログラムカウンタなどのフレーム情報を、データ構造 (いわゆるコンテキスト) に明示的に格納するコードを出力します。 catch 節の末尾には、再び例外を投げるコードが出力されます。この例外処理の連鎖によってスタックを巻き戻ることができます。負荷 (スレッド) のプロセッサ間の移動もブロックと同じ方針で処理できるとふんでいます。

このアルゴリズムで処理系を実装したとして、その後に残る最大の問題は、学術的な研究としての contribution を見つけられるか、だと思います。「JVM (すなわち call/return/throw でしかスタックを操作できない言語) の上にスレッドを実現しました」で研究になるほど研究は甘くないと思ってます。

[1] Java を精力的に使いだしたのはごく最近ですが、その仕様のきれいさにウットリする日々が続いています。処理速度が遅いですけどね。

[2] 一杉さんにアドバイスを頂くためはるばる筑波まで行って参りました。

[3] 筑波の加藤さんの卒研生の阿部くんが、似たようなことを EPP でやっているといると聞き、連絡をとりました。加藤さんは「腕の立つ学生」とほめていましたが、メールをやりとりするうち、私もそう思いました。