Survey: OpenMP on Networks of Workstations

田中 義純

1999 年 10 月 18 日

概要

OpenMP とはどのようなものかという簡単な紹介と、最近の OpenMP の研究のサーベイを行います。

1 Introduction

1.1 What is OpenMP

OpenMP[3] は、共有メモリ並列計算機上において、自動並列化を行うための API である。具体的には、逐次言語で書かれた逐次プログラムの、並列に実行したい部分に並列化指示子を挿入することによって自動並列化を行っている。そのため、異なったアーキテクチャのマシン間において、ソースコードレベルでの移植性が非常に高いという特徴がある。また、逐次プログラムに並列化指示子を挿入するだけで並列プログラミングをすることが出来るので、並列プログラミングに慣れていない人でも容易に並列プログラムが書けるという特徴がある。

1.2 An Example

例えば、

```
int Summation() {
    int i, sum = 0;
    for(i = 0; i < 100; i++)
        sum += i;
    return sum;
}
```

のような逐次のプログラムがあったときに、この `for` の部分を 10 台のスレッドを用いて並列に実行するように書き換えるとしたら、

```
void Loop(int* s, int a, int b) {
    if (b-a == 1) {
        int i, j = a*10;
        for(i = j; i < j+10; i++)
            *s += i;
    } else {
        int s1, s2, m=(a+b)/2;
        THREAD_CREATE(Loop(&s1, a, m));
        Loop(&s2, m, b);
    }
}

int Summation() {
    int sum = 0;
    Loop(&sum, 0, 10);
```

```
        return sum;
    }
```

のようになるであろう。（ただし、スレッド生成の部分にはかなりの省略があり、また正しく実行するためにはきちんと同期をとる必要があるので、正確にはもう少し複雑になる。）

しかしながら、このプログラムを OpenMP を用いて並列化すると、

```
int Summation() {
    int i, sum = 0;
#pragma omp parallel for reduction(+:sum)
    for (i = 0; i < 100; i++)
        sum += i;
    return sum;
}
```

のように、プラグマ指令を挿入するだけで自動的に並列実行してくれるわけである。

このように、プログラマの負担は大幅に軽減されるわけであるが、効率のよい実行をするためにはその分コンパイラが頑張る必要がある。

2 OpenMP の最近の研究の動向

さて、 OpenMP の説明はこのくらいにして、 OpenMP 関連で最近行われている研究の紹介をする。OpenMP は、共有メモリ並列計算機向けの API であるわけだが、最近よく行われている研究は、分散メモリ並列計算機や SMP クラスタ上における並列プログラミングを OpenMP を用いて効率的に行おうというものである。 OpenMP は、並列化が容易であること、ソースコードレベルでの移植性が高いことが売りであるので、プログラムに変更を加えずに、共有メモリ並列計算機、分散メモリ並列計算機上で効率的に動いてくれればうれしいわけである。

3 OpenMP on Networks of Workstations

この論文 [2] では、 Networks of Workstations(NOWs) 上で、 OpenMP を用いて書かれたプログラムを効率的に動かすことを目標として書かれている。しかしながら、 OpenMP は共有メモリマシンをターゲットにした仕様であるので、直接分散メモリマシン用にコンパイルするのは容易ではない。そのため、この論文では、ソフトウェア DSM の一つである TreadMarks[1] を導入することにより、 OpenMP のプログラムを TreadMarks 上で動かすようにコンパイルしている。しかしながら、共有メモリマシンと NOWs ではアーキテクチャが違うので、普通にコンパイルしたのでは性能が出ない結果になってしまう可能性がある。

そこでこの論文では、 OpenMP のプログラムをソフトウェア DSM 上で実行するときの問題点とその改良法について述べ、 MPI を用いて書かれたプログラムとの比較をしながら性能を評価している。

3.1 問題点と改良案

まず、この論文では、 NOWs に OpenMP を実装する際に 2 つの提案をしている。

1. 並列領域における変数を default で shared ではなく private にする
2. *flush* を廃止して、代りに *condition variable* と *semaphore* を導入する

1. はアーキテクチャの違いを表している。共有メモリマシンの場合には、 static なグローバル変数や heap 上の変数は共有変数であるだけでなく、スレッドローカルなスタック上の変数でさえもポインタを渡すことによって参照することが可能である。しかしながら、ソフトウェア DSM の場合は、どのような変数まで共有されるかはシステム依存であり、静的に割り当てられた変数だけが共有であったり、 heap だけが共有であったり、また、専用の関数で確保されたメモリ領域だけが共有であったりする。 TreadMarks の場合には、基本的にはすべての変数は private であり、 *Tmk_malloc* を用いることによって、共有 heap 上に変数を割り当てることが出来るようになっている。

さて、「並列領域における変数を default で shared ではなく private にする」ことの意義であるが、コンパイル時と実行時のコストの削減のためである。共有メモリマシンの場合、共有変数の実装は簡単

である。単に、ポインタを渡してやればいいだけのことである。それに対して、 private 変数は、スタック上に新たな変数を定義してやる必要があるので少し手間がかかる。それに対して、 TreadMarks の場合には、基本的にすべての変数は private であるので、ある変数を共有変数として用いたい場合には、先ほどとは逆に新たに定義してやる必要がある。また、共有変数へのアクセスは同期を伴うので、大変コストのかかる操作である。したがって、単に値を渡すだけの操作の場合には *firstprivate* を用いることによって解決している。

2. も 1. に関係していて、通信にコストがかかることが原因の問題である。 *flush* を行うと、すべてのスレッド間で共有変数の一貫性がとられる。この時、共有メモリマシンであれば、レジスタの変数をメモリに書き戻して write barrier をすればいいだけであるが、ソフトウェア DSM の場合には、どのスレッドがどのような状況で待っているのか分からないので、 n 個のスレッドがあったとすると、 $2(n-1)$ 回の通信ををする必要が出てきてしまう。この操作は、共有メモリマシンの操作に比べるとかなり重いものがあるので、問題である。

そこで、この論文では、新たに *condition variable* と *semaphore* を導入することにより、 *flush* よりも軽い操作で *flush* と同等の効果を得ようとしている。

具体的に、どのような場面で *flush* が使われるかということであるが、 pipeline や task queue などの並列実行のための機構を実装するときに有効のようである。まず、 pipeline であるが、 flush を用いて書くと Figure 1 のようにして実装することが出来る。同様に、 task queue は、 Figure 2 のようにして実装することが出来る。ここで、 *condition variable*、 *samaphore* を以下のように実装する。

```
sema_wait(S): while (S <= 0) do no-op;
              S--;
sema_signal(S): S++;
```

まず *semaphore* であるが、これは普通の計数セマフォである。

```
cond_wait(id)      : Block on a condition variable
cond_signal(id)    : Unblock one waiting thread
cond_broadcast(id): Unblock all waiting threads
```

次に *condition variable* であるが、これは critical section 内でのみ用いることができる。どういうことかというと、 critical section 内で `cond_wait(id)` が呼ばれると、そのスレッドは critical seciton から抜けた状態で block する。そして、 `cond_signal(id)` や `cond_broadcast(id)` が呼ばれると、 critical section の中から再開される。

これらを用いて先ほどの pipeline、 task queue を書き直すと、 Figure 3、 Figure 4 のようになる。今度の場合、 *flush* と違うのは、 *conditon variable* や *semaphore* の管理者はどのスレッドが待っているのかを知っているので、 *flush* に比べれば通信量が減り、また、必要のない同期をとらなくてよくなるという利点がある。

3.2 性能評価

性能評価は、 Table 1 にある 5 つのアプリケーションで行われている。このうち、 Sweep3D では、先ほどの pipeline を、 QSORT では task queue を用いて実装されている。実験は、 166MHz の Pentium Pro 8 台が、 switched、 full-duplex 100Mbps のイーサネットでつながれている上で行われた。比較の対称は、 OpenMP によるプログラミング、 TreadMarks で直接プログラミング、 MPI でプログラミングの 3 つで行われた。 Figure 5 が実験結果であり、 OpenMP によるプログラミングの結果は、 TreadMarks によるプログラミングに比べると、 $3-17\%$ 程度のオーバーヘッドであったが、 MPI に比べると 41% ものオーバーヘッドがかかっている。その原因は、 Table 2 における通信量の差から来ており、TreadMarks が間で頑張って通信している分だけオーバーヘッドがかかっているようである。

4 Omni on StackThreads/MP の現状

現時点での実装は、 SWoPP99 に出したときとは変更し、 for や sections では細粒度スレッドを生成しない方向で進めています。 SWoPP99 で、 parallel の他に for や sections でも細粒度スレッドを生成していたのは性能を出すためだったわけですが、性能が出なかった原因が、スレッドの freelist を管理す

るときに lock をとっていたためであることが解明されたので、その部分を改良して現時点での実装になりました。

現状の問題点としては、 barrier が遅いということがあげられます。細粒度スレッドを LWP 数以上に生成するため、 barrier を実装するためには、細粒度スレッドの中断・再開を行わないといけないわけですが、どうしてもコストがかかってしまうため、 Omni に比べて実行時間が 180

現在は、 Omni のバージョン 1.0 のリリースに合わせて Omni on StackThreads/MP の β バージョンをリリースできるように実装を進めているところです。

参考文献

[1] C. Amza, A.L.Cox, S.Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W Zwaenepoe. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[2] Honghui Lu, Y Charlie Hu, and Willy Zaenepoel. Openmp on networks of workstations. In *Supercomputing'98*, November 1998.

[3] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface*, October 1998. http://www.openmp.org/.