Vol.2011-HPC-130 No.6
2011/7/27

情報処理学会研究報告
IPSJ SIG Technical Report

ハードウェア同期機構を用いた超軽量スレッドライブラリ

堀 敦 史[†1,†2] 山 本 啓 二[†1,†2] 大 野 善 之[†1,†2]
今 田 俊 寛[†1,†2] 亀 山 豊 久[†1] 石 川 裕[†1,†3]

エクサスケールを視野に置いた、メモリや通信の遅延を隠蔽するための新しいマルチスレッドライブラリを提案する。そのためにはサブマイクロ秒でのスレッド制御を可能にする必要がある。本稿では、スレッドスケジューリングとして、プロセッサの Simultaneous Multi-Threading 機能を用い、ハードウェアによる高速なスレッドスケジューリングを用い、また、スレッド間の同期機構として Intel 製のプロセッサが提供する monitor/mwait 命令を用いた新しいスレッドライブラリ、*Shadow Thread* を提案する。高速な同期と低消費電力を両立させるため、同期フラグを spin-wait と monitor/mwait 命令を組み合わせた 2-phase の同期機構が有効であることを示す。この方式を用いて開発された Shadow Thread は、メモリ領域のコピーにおいて、最大約 20 % の高速化に成功した。

A Ultra-light Thread Library Using Hardware Synchronization

Atsushi Hori,[†1] Keiji Yamamoto,[†1]
Yoshiyuki Ohno,[†1] Toshihiro Konda,[†1]
Toyohisa Kameyama[†1] and Yutaka Ishikawa[†1,†3]

Towards the Exa-scale computing, a new thread library is proposed to hide the latencies of memory and communication. For this purpose, thread management must be fast enough in the order of sub-micro seconds. In this paper, the thread library, named *Shadow Thread*, is developed so that it utilizes Simultaneous Multi-Threading mechanism which schedules threads by hardware in a very fast way and utilizes the *monitor* and *mwait* instructions supported by some Intel processors. It is shown that the two-phase synchronization technique combining the conventional spin-wait method and the pair of the monitor/mwait instructions can satisfy the requirement of speed and low-power consumption simultaneously. Evaluation shows that a memory copy function using the Shadow Thread library can exhibit better performance up to 20 % compared with the normal `memcpy` function.

1. はじめに

エクサスケールに到る性能向上における技術的なトレンドでは、メモリや通信の遅延は、絶対値的には低下するものの、プロセッサの性能向上に見合った程の性能向上は難しく、プロセッサの性能を基準に見た場合の相対的遅延は大きくなると予想されている[1)]。さらに、演算コア当たりのメモリ量は減少するとも予測されている[1)]。このため、これまでの weak scaling（問題サイズを大きくして並列効率を維持する並列プログラミングモデル）が成立せず、strong scaling（問題サイズを大きくしなくても、演算コア数の増大に伴い、トータルの計算性能が向上する並列プログラミングモデル）にならざるを得ない。特に strong scaling においては通信の遅延を小さくすることが重要である。このようにメモリや通信の遅延の影響は、今後の HPC の技術動向を考慮すると非常に大きい。そこでソフトウェアにより遅延を隠蔽し、見掛けの遅延を減少させる技術が今後より一層重要になるものと考えられる。

表 1 Null スレッド起動時間の比較

スレッドの実装	時間 [usec]	Time Stamp Counter
pthread	23.57	65844
MPC++ (ucontext)	0.9481	2648
Shadow (monitor/mwait)	-	2879
Shadow (spin-wait)	-	58

Intel Xeon (X5560、2.8 GHz、6 cores)

遅延を隠蔽する技術として知られているマルチスレッドは現在広く使われている。Linux で最も一般的なマルチスレッドの実装として pthread がある。現時点での通信の遅延がマイクロ秒のオーダーであり、将来はサブマイクロ秒台になる可能性が高い。また、メモリの遅延時間はこの値よりもさらに小さい。Pthread ではシステムコールを伴うため、その制御にはかなりの時間を要する。表 1 に Linux 上でのいくつかのスレッドの実装における、null スレッドの起動から終了までの時間を示す。計測は Intel Xeon (X5560、2.8 GHz) 上でお

†1 理化学研究所計算科学研究機構 / RIKEN AICS
†2 科学技術振興機構 CREST / JST CREST
†3 東京大学 / The Univ. of Tokyo

© 2011 Information Processing Society of Japan

Vol.2011-HPC-130 No.6
2011/7/27

こなった。Pthread の場合、約 24 マイクロ秒を要しているため、マイクロ秒オーダーの通信の遅延を隠蔽するには不適切である。表中、MPC++[4),5)] とあるのは、pthread がカーネルレベルで実装されたカーネルスレッドであるのに対し、ユーザレベルで実装したユーザレベルスレッドの実装である。スレッドコンテキストの切替には Linux の `swapcontext()` を使っているため pthread よりも高速であるが、これでも約 1 マイクロ秒かかっている。表 1 で Shadow とあるのは、本稿で新たに提案する *Shadow Thread* の計測結果である。最も高速な Shadow （表中、"Shadow(spin-wait)"）のスレッド起動時間は、pthread の約 1,000 倍、MPC++ の約 50 倍も高速である。

本稿では、エクサスケールに向けた基礎技術の研究開発を目指し、メモリと通信の遅延を隠蔽することを目的とした高速なマルチスレッドライブラリ "Shadow Thread" の実装方式について、その実装の詳細と評価実験の結果について報告する。

2. Shadow Thread

図 1 は、一般的なマルチスレッドをモデル化したものである。マルチスレッドを実現するためには、1) スレッドの生成（fork）, 2) fork されたスレッド間のスケジューリング、3) スレッド処理の終了の同期（join）の制御が最低限必要である。そして、細粒度なスレッドを効率よく実行するためには、これら全ての制御が高速でなければならない。

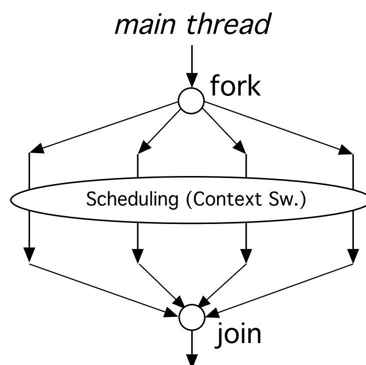

図 1 Multi-Thread

これまで HPC の分野ではあまり細粒度のスレッドは用いられてこなかった。例えば OpenMP による計算ノード内の並列化においては、スレッド数はノード内の演算コア数を上限としていることが多い。演算コア数よりもスレッド数を多くするとスレッドのコンテキストを切り替える必要が生じ、オーバヘッドとなってしまうからである。一方で、演算コア数よりも多くのスレッドが走行しないと、あるスレッドに待ちが生じた場合、その待ちで生じた空隙を埋めるスレッドがないため、効率が低下する。本稿が目的とするメモリや通信の遅延を隠蔽するためには、演算コア数よりも多くのスレッドが必須である。

最近のプロセッサには Simultaneous Multi-Threading（以下、SMT）機構[12)] が提供されている。これはコア内部に複数のハードウェアコンテキストを持ち、ひとつの演算パイプラインに複数のスレッドの命令列を流し込み、複数スレッドを同時に実行するというものである（図 2 には 2-way SMT の例を示す）。ここであるスレッドがキャッシュミス等により演算がストールしたとすると、それにより生じた演算パイプラインの空きは別のスレッドの実行で埋められる。このように SMT は演算コア内に高速なスレッドスケジューラを持ち、メモリアクセスなどの遅延を隠蔽することが可能になっている。

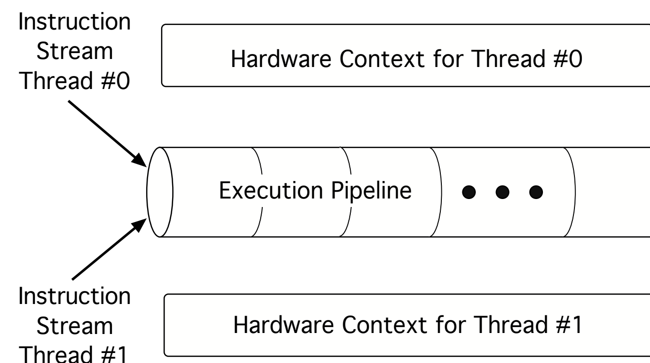

図 2 Simultaneous Multi-Threading

本稿で提案する *Shadow Thread* は、SMT を活用することで高速なスレッドの実現が目的である。ユーザレベルスレッド（例えば、MPC++[4)]）は表 1 に示したようにそれなりに高速ではあるが、演算コアから見たスレッド数はひとつ（図 2 における命令ストリームがひとつ）なので SMT によるメモリ遅延を隠蔽することができない。このため、なんらかの方法でカーネルレベルのスレッドを生成する必要がある。また Shadow Thread におい

© 2011 Information Processing Society of Japan

ては、演算コア内の SMT スケジューラを用いることから、Shadow Thread としてのスケジューラは存在しない。

以下、先に示したスレッド制御における３つの局面のうち、スレッドスケジューリングを除く fork と join の局面で、我々が提案する　Shadow Thread の実装方式について説明する。

2.1 Fork と Join の高速化

一般にスレッドの fork は新しいスレッドのコンテキストを生成することである。Shadow Thread では SMT を活用するため、実際にカーネルレベルのスレッドを生成する必要があり、スタックの生成やカーネル内でのスケジューリングのためのエントリの追加などの処理があり、サブマイクロ秒オーダーでの実現は難しい。

そこで、Shadow Thread ライブラリ初期化時に pthread を生成しておき、fork 時にそのスレッドに実際に処理する関数アドレスを指定することで、スレッド生成のオーバヘッドを fork 処理から取り除くことが可能になる。処理が終わった段階で、Shadow Thread のスレッドは新たな処理が指定されるまで待ち状態になる。

Shadow Thread で fork されるスレッドの数は、演算コアがサポートする SMT が N-way であった場合、高々 $N-1$ である。これより多くのスレッドが走るとカーネルのスケジューラの関与を生じてしまう。

Join では fork したスレッドの終了を待つ。Shadow Thread では、fork の時に処理したい関数が指定されるまでと、join 時のスレッドの終了のふたつの待ち（同期）が生じ、これらの待ちをいかに高速かするかが重要なポイントとなる。

2.2 同期の高速化

```
void wait_until( int *addr, int val ) {
    while( *addr != val ) {}
}
```

図 3　Spin-Wait 関数の例

同期を実現する手法として最も単純で高速な方法として知られるのが spin-wait である（図 3 に例を示す）。Intel の x86 プロセッサでは SSE 拡張命令として monitor と mwait 命令がサポートされているものがある[3]。Monitor 命令で指定されたアドレスに書込みがあるまでそれに続く mwait 命令が待つという動作をする。この monitor-mwait 命令の主な特徴を以下に列挙する。

- mwait 命令で省エネルギーモードに移行する
- monitor 命令で指定されたアドレスに書込みがあった場合以外でも mwait 命令が終了することがある（例えば割込）
- 特権命令である

この monitor/mwait 命令は特権命令なので、ユーザプログラム内で用いることはできないが、以下、本稿では Kernel Mode Linux[11] を用いて実験を行った。図 4 に monitor/mwait 命令を用いて同期を実現したプログラム例である。

```
void wait_until( int *addr, int val ) {
    while( *addr != val ) {
        MONITOR( addr );
        MWAIT();
    }
}
```

図 4　Monitor-Mwait 関数の例

図 5 は、spin-wait を用いて、ふたつのスレッドを異なるコア（X 軸と Y 軸）にバインドした時に、1 秒間に何回 null スレッドを起動出来るか（Z 軸）を示した図で、図 6 は monitor/mwait 命令を用いた場合の図である。Z 軸（上下方向）の値が大きい程性能が高いことを示す。同じコアの指定はは対角に並ぶ。これらの図では前後方向が対角になるような視点になっている。これらの図から、特に spin-wait では、monitor/mwait 命令を使った場合に比べ、同じコアの SMT を用いた場合に高性能であることが顕著である。この結果から Shadow Thread　では、メインスレッドを同じ演算コアの SMT となるように fork するのが効率的である。

先に示した表 1 において、”Shadow (monitor/mwait)” とあるのは、fork と join の同期を、monitor/mwait 命令を用いて実現した場合、”Shadow (spin-wait)” とあるのが spin-wait を用いた場合のそれぞれで、null スレッドを fork してから join するまでの時間である。この結果および図 5 と図 6 から、monitor/mwait 命令による同期は spin-wait に比べかなり遅いことが分かる。

Monitor/mwait 命令を用い同期の実現では、spin-wait より遅いが、より省エネルギーであることが期待される。そこで、これらを両立させる方法を考える。最初しばらくは spin-wait で待ち、その後は monitor/mwait 命令で待つという 2-phase での実現である（図 7）.

ⓒ 2011 Information Processing Society of Japan

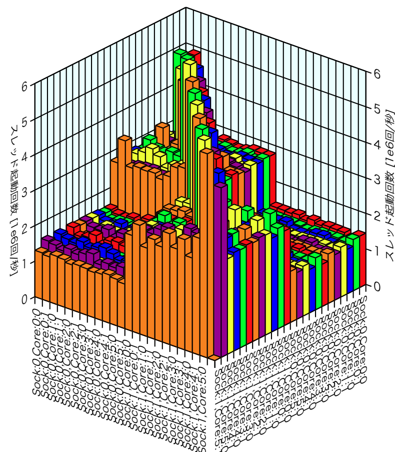

図 5 コア割当とスレッド起動速度 (spin-wait)

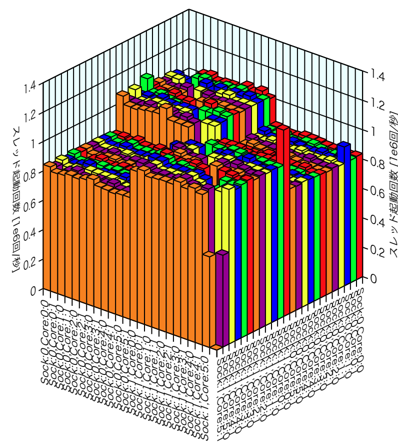

図 6 コア割当とスレッド起動速度 (monitor/mwait)

```
void wait_until( int *addr, int val ) {
    int k = threshold;
    while( --k >= 0 ) {
        if( *addr == val ) return;
    }
    while( *addr != val ) {
        MONITOR( addr );
        MWAIT();
    }
}
```

図 7 2 Phase の同期関数の例

図 8 は、図 7 における threshold の値を変化させて、shadow thread を生成し、2 秒毎に 10 回、演算コアの温度を計測した場合の結果を示したものである。計測の間、メインのスレッドは join 待ちとなる。この表の凡例にある例えば "THR=10^5" というのは、threshold の値が 10^5 であることを示す。演算コアの温度は lm_sensor を用いて計測した。また、温度変化が顕著になるよう、2 ソケット、計 12 個の演算コア全てで同じ処理を行い、全ての演算コアの温度を平均化した値を用いた。

この図から、monitor/mwait は spin-wait に比べ、かなり省エネルギー効果が大きいことが分かる。エクサスケールにおいては省エネルギーも重要な性能指標のひとつであり、その意味から monitor/mwait 命令を用いる価値があると考える。"THR=10^9" において、最初の 4 秒間は spin-wit 同様の温度変化を、その後は monitor/mwait の場合と同様の温度変化を示している。このことから、spin-wait から monitor/mwait への相変化が、計測開始の 4 秒から 6 秒の間で発生したものと考えられる。

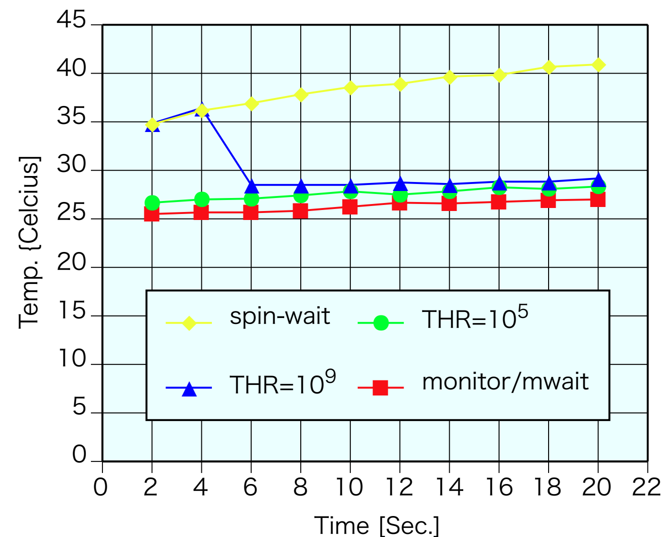

図 8 同期の実装方式の違いとコアの温度変化

表 2 は 2-phase での同期の実装における threshold の値と null スレッドの fork から join までに要した時間（time stamp counter）を表にしたものである。この表と図 8 の結果から、2-phase における threshold の値を適切に選ぶことで、高速なスレッドの起動と省エネルギーが同時に実現されていることが分かる。

3. Shadow Thread API

表 3 に Shaodw Thread の API を列挙する。`shadow_create()` 関数は、この関数を呼

© 2011 Information Processing Society of Japan

表 2 しきい値を変化させた時の起動時間の変化

しきい値	Time Stamp Counter
0 (monitor/mwait)	2879
10^2	74
10^4	71
10^6	70
10^8	71
無限大 (spin-wait)	58

び出したスレッドを引数で指定された演算コアにバインドし、Shadow Thread のためのスレッド（pthread）を生成する。このスレッドは、引数で指定された演算コアの SMT となるべく対応する演算コアにバインドされ、`shadow_fork()` 関数で実行すべき処理が指定されるまで待ち状態にある。

SMT ではキャッシュを共有するため、Shadow Thread で生成されたスレッドを同じ演算コアにバインドすることで、同期のためのフラグが共有されている同じ 1 次キャッシュにアクセスすることになり、同期が高速になるという利点がある。

`shadow_fork()` 関数は Shadow Thread が成すべき処理を記述した関数を指定する。この関数アドレスは、先に `shadow_create()` 関数で作られたスレッドに渡され、待ちから抜けて処理を開始する。`shadow_join()` 関数は `shadow_fork()` で指定された関数が終了するのを待つ。`shadow_destroy()` 関数は、`shadow_create()` で生成されたスレッドを終了させる。

表 3 Shadow Thread API

`shadow_create()`	初期化
`shadow_fork()`	fork
`shadow_join()`	join
`shadow_destroy()`	終了

4. 評価実験

Intel Xeon (X5560、2.8 GHz、6 cores）を 2 ソケット持つサーバ（計 12 物理演算コア）上で Shadow Thread の有効性を研修するための評価実験をおこなった。Shadow Thread の目的のひとつがメモリの遅延の隠蔽であるため、memcpy を Shadow Thread により並列化し、どれだけ性能向上が得られるかを調べた。Shadow Thread による memcpy の並列化は 1) preftech、2) block 分割、3) split 分割、の 3 種類おこなった。

Prefetch

Prefetch 方式では、Shadow Thread において、memcpy の対象となるメモリ領域を全て preftech するという処理を走らせる（図 9）。

```
while( from < from_max ) {
    PREFETCH_READ( from );
    PREFETCH_WRITE( to );
    from += CACHE_BLKSZ;
    to   += CACHE_BLKSZ;
}
```

図 9 Prefetch

Block 分割

Memcpy の対象となるメモリ領域を指定された大きさのブロックに分割し、分割されたブロックをメインのスレッドと shadow スレッドのそれぞれが飛び飛びにコピーする方式である（図 10、コードの概略は図 11）。

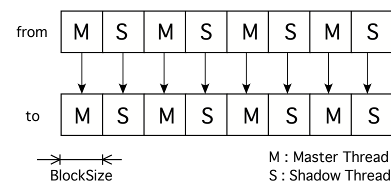

図 10 Block 分割

```
while( from < from_max ) {
    memcpy( to, from, blocksz );
    from += blocksz*2;
    to   += blocksz*2;
}
```

図 11 Block コピー

Split 分割

Memcpy の対象となるメモリ領域を単純に 2 分割し、そのそれぞれをメインのスレッドと shadow スレッドとでコピーする方式である（図 12）。

実 験 結 果

図 13 に、比較のために glibc の memcpy の性能と、上記、prefetch 方式, block 方式はブロックサイズを 256、1024、4096 バイトの 3 パラメータで、そして split 方式、の全部で 6 ケースについて計測した結果を示す。右側の Y 軸は、4096 バイトの block コピー

© 2011 Information Processing Society of Japan

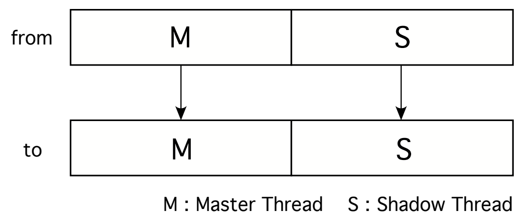

図 12　Split 分割

のバンド幅を glibc の memcpy のバンド幅を割った値を示す。この値が 1 より大きい場合は、glibc の memcpy より高いバンド幅であることを示す。コピーの対象領域は常にフレッシュな領域をコピーするようにし、キャッシュに載っていない状態で計測をおこなった。Shadow Thread を用いた場合の計測では、計測時間に fork/join の時間が含まれている。Threshold の値は全てのケースにおいて 1,000 に設定した。

コピーする領域のサイズが 32 KB より小さい場合では glibc の memcpy が最も性能が高い。例えば、バンド幅が 5 GB/s であったとすると、1 KB のコピー時間は 0.2 マイクロ秒と短時間で終了する。このため Shadow Thread のわずかなオーバヘッドでもコピー性能に与える影響は大きいと考えられる。

Prefetch 方式では、memcpy と全く同期せずに prefetch が先に進むため、データサイズが大きくなると、memcpy している最中の場所と prefetch する場所のズレが大きくなり、無駄な prefetch が悪影響を及ぼしたものと考えられる。コピーするサイズが 1 MB から 32 MB まではブロックサイズ 4,096 バイトの block コピー方式が最も性能が高かった。次に性能が高いのはコピー領域を単純に 2 分する split 方式であった。Block コピー方式は、最大で memcpy の 20 % 程度（右側の Y 軸）の速度向上であった。

5. 関連研究

文献[10] の記事では、SMT を用いることでかえって性能が低下した例が報告されている。これは多くの SMT の実装では、コアに固有のキャッシュが SMT 間で共有されているにも関わらず、普通のスレッドとして SMT を用いた結果、コア当たりのワーキングセットが増大し、キャッシュのスラッシングが生じた結果と推測される。この報告が示すように SMT を意識せずに普通のスレッドとして用いることは必ずしも性能向上とはならない。SMT であることを意識してワーキングセットを大きくしないようにして用いるべきである[8]。第 4

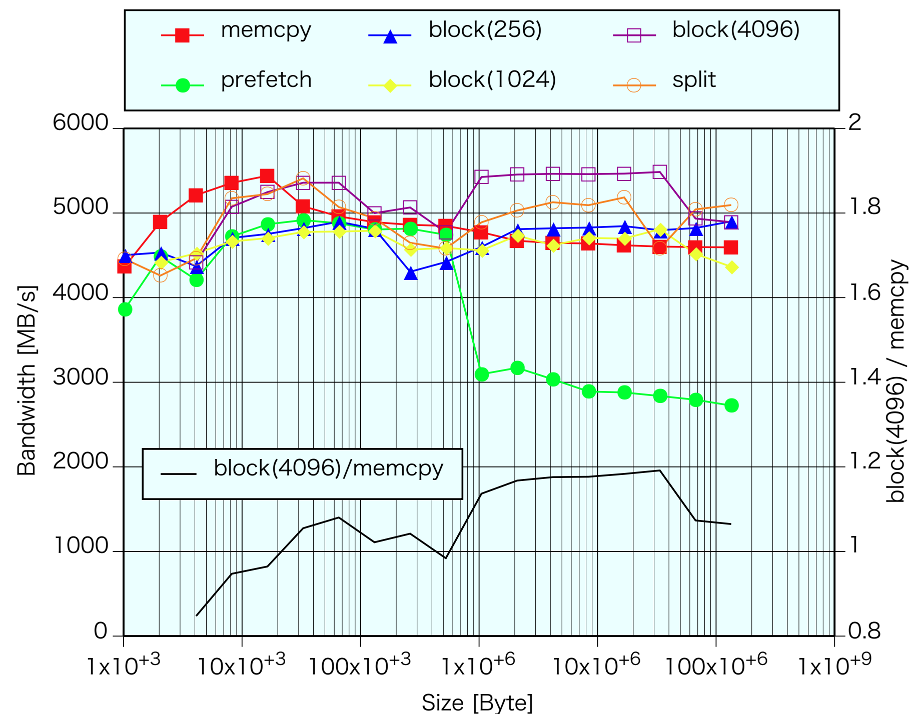

図 13　メモリコピーの性能比較

ⓒ 2011 Information Processing Society of Japan

Vol.2011-HPC-130 No.6
2011/7/27

章の評価実験において、split の性能が block に及ばないのは、split の方が離れた領域をコピーするためにワーキングセットが大きくなり過ぎたものと考えられる。

OpenMP でのスレッドは、例えばあるループの処理を複数のスレッドで分担して処理をする、いわゆる *work share* である。一方、SMT がキャッシュを共有するという性質を活用し、例えば第 4 章の prefetch のようにメインのスレッドがアクセスするデータを、SMT のスレッドがプリフェッチするような非対称な使い方は *Helper Thread* と呼ばれている（例えば文献[6)–8)]）。Kim らは Windows 上で Windows が提供するスレッド API を用いて Helper Thread を実装した。しかしながらスレッド管理のためのオーバヘッドが大きく、アプリケーションを用いた評価では 10 % 未満の効果しか観測されていない。このことからも軽量なスレッドの実装が望まれていることが示唆される。笹田らは、SMT スレッドを高速にソフトウェアで制御可能とするハードウェア命令を提案している[9)]。Kamruzzaman らは異なるコアのデータをプリフェッチするために Helper Thread を用い、アプリケーションを使った評価で 30 % 以上の速度向上を得たと報告している[7)]。Kamruzzaman らの Helper Thread の実装は Linux の ucontext を用いており、高速なスレッド切替を実現している。

Shadow Thread のように SMT のスレッドとメインのスレッドをペアで用いるアイデアとしては Vouk の *Buddy Thread* がある[13)]。Buddy Thread では、Shadow Thread 同様に monitor/mwait 命令を用いた同期による Buddy Thread の実現と、MPICH の isend/irecv 処理に Buddy Thread を応用することで、ベンチマークプログラムによる評価では最大 30 % の速度向上を実現したとしている。

Goumas らは、MPI の実装ではなく、MPI で書かれたプログラムを通信と計算部分に分け、通信部分を Helper Thread で処理するように変更し、計算速度を最大 20 % 向上させる事に成功している[2)]。

本研究の貢献は、monitor/mwait 命令による同期の実現だけでは速度的に十分でないため、spin-wait と組み合わせた 2-phase の待ちの実現手法が、速度と消費電力の観点からベストであるという主張にある。Shadow Thread のスレッドの使い方を work share とするか Helper Thread とするかは特に制限はない。

マルチスレッドを用いたデザインパターン（例えば、[14)]）においても、Shadow Thread 同様、処理に先行してスレッドを生成しておくことがある。しかしながら Shadow Thread ではスレッドの用途を特定してはいない点がマルチスレッドのデザインパターンと異なっている。プログラムの TPO（Time、Place、Occasion）に応じた目的のスレッドを実行することが Shadow Thread の目的である。

第 2.2 章での述べたように、本稿で用いた monitor/mwait 命令は特権命令であるため、普通のユーザプロセスでは実行できない。文献[13)] には “In future implementations of the instruction, Intel plans to make this instruction (mwait) available at ring-3 execution privilege level. “（括弧内は意味が通じるよう著者が挿入した）という記述があり、将来はユーザプロセスが普通に使えるようになる可能性もある。

おわりに

本稿では、エクサスケールのスーパーコンピュータに向け、メモリと通信の相対的な遅延が増大するとの問題意識から、SMT を用いた高速なスレッドライブラ、Shadow Thread、を開発し、その詳細について報告した。高速な同期と低消費電力を同時に実現するために、spin-wait と monitor/mwait 命令を組み合わせた 2-phase の同期方式が有効であることを示した。開発された Shadow Thread は評価実験としてメモリコピーに応用され、最大で約　20 % の速度向上を実現することができた。

今後、より幅広い応用に Shadow Thread を適用し、評価と改良を進め、Shadow Thread の有効性を検証する予定である。

謝辞　本研究は, 科学技術振興機構 (JST) の戦略的創造研究推進事業「CREST」における研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」 によるものである。

参考文献

1) Dongarra, J., Choudhary, A., Kale, S. et al.: The International Exascale Software Project Roadmap, White paper, Argonne National Laboratory (2010).
2) Goumas, G.I., Anastopoulos, N., Koziris, N. and Ioannou, N.: Overlapping computation and communication in SMT clusters with commodity interconnects, *IEEE Cluster Computing*, pp.1–10 (2009).
3) Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual* (2011).
4) Ishikawa, Y.: The MPC++ Programming Language V1.0 Specification with Commentary Document Version 0.1, Technical report, Tsukuba Research Center, Real World Computing Partnership (1994).
5) Ishikawa, Y.: Multiple Threads Template Library – MPC++ Version 2.0 Level 0 Document – Document Revision 0.13, Technical Report RWC-TR-096-012, Tsukuba Research Center, Real World Computing Partnership (1997).

© 2011 Information Processing Society of Japan

6) Jung, C., Lim, D., Lee, J. and Solihin, Y.: Helper Thread Prefetching for Loosely-Coupled Multiprocessor Systems, *In Proceedings of 20th IEEE International Parallel & Distributed Processing Symposium* (2006).
7) Kamruzzaman, M., Swanson, S. and Tullsen, D. M.: Inter-core prefetching for multicore processors using migrating helper threads, *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, New York, NY, USA, ACM, pp.393–404 (2011).
8) Kim, D., Liao, S. S.-w., Wang, P.H., Cuvillo, J.d., Tian, X., Zou, X., Wang, H., Yeung, D., Girkar, M. and Shen, J.P.: Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors, *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, Washington, DC, USA, IEEE Computer Society, pp.27– (2004).
9) KOICHI, S., MIKIKO, S., KANAME, U., YOSHIYASU, O., HIRONORI, N. and MITARO, N.: A Lightweight Synchronization Mechanism for an SMT Processor Architecture(Processor Architectures), 情報処理学会論文誌、コンピューティングシステム， Vol.46, No.16, pp.14–27 (2005-12-15).
10) Rupert Goodwins: Does hyperthreading hurt server performance? http://news.cnet.com/2100-1006_3-5965435.html.
11) Toshiyuki Maeda: Kernel Mode Linux : Execute user processes in kernel mode. http://web.yl.is.s.u-tokyo.ac.jp/tosh/kml/.
12) Tullsen, D.M., Eggers, S.J. and Levy, H.M.: Simultaneous multithreading: maximizing on-chip parallelism, *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, New York, NY, USA, ACM, pp.533–544 (1998).
13) Vouk, N.: Buddy Threading in Distributed Applications on Simultaneous Multi-Threading Processors (2005).
14) Wikipedia: Thread pool pattern. http://en.wikipedia.org/wiki/Thread_pool_pattern.

© 2011 Information Processing Society of Japan