

再帰的に生成したタスクに対するデータ局所性を考慮するスケジューラの実装と評価

中谷 翔*, 中島 潤†, 田浦 健次朗†, 近山 隆‡

* 東京大学工学部 † 東京大学情報理工学系研究科 ‡ 東京大学工学系研究科

1 はじめに

これらの計算機は、逐次性能が頭打ちとなり、並列化を今まで以上に指向することになる。この流れの中、マルチコアプロセッサのみならず、メニーコアプロセッサや GPGPU を含んだヘテロ構成のマシンも増えてきた上、ノード間通信を伴うクラスタも身近なものとなってきた。これはソフトウェアに並列化の必要性を強く与える一方、それを難しいものとしてきた。この困難に立ち向かうためのプログラミングモデルとして、タスク並列がある。タスク並列では、プログラマが処理のまとまりをタスクとして定義・生成することで、複数のタスクが並列に処理される。タスクは再帰的に生成して並列度を生み出すことが多い。タスク並列の枠組みには、Cilk, Intel Threading Building Blocks, OpenMP Task, Java Fork/Join Framework など様々存在する。これらはそれぞれいくつかのタスクスケジューリング方式をサポートしているが、その中でも、Lazy Task Creation[4] が多くのアプリケーションで比較的良好な性能を示すことが種々のベンチマークで示されている。

Lazy Task Creation では、タスクを処理する単位としてワーカが存在する。ワーカはタスク生成の際に、現在実行しているタスクを自分のタスクキューに戻し、生成したタスクの実行を始める。実行すべきタスクを持っていないワーカは、別のワーカが最も前にタスクキューに戻したタスクを奪うことができる。この作用をワークスティーリングと呼び、タスクを奪うワーカを thief、奪われるワーカを victim と呼ぶ。Victim はランダムに選ぶ手法が一般的で、これをランダムなワークスティーリングと呼ぶこととする。ランダムなワークスティーリングは多くの問題に対して有効であることが示されている [1] が、一方でアプリケーションによってはより効果的なワークスティーリング戦略を考えることもできる。ワークスティーリング戦略を含め、様々なタスクスケジューリング方式を模索する研究が [3, 2] などに近年見受けられる。

これらを背景とし、本研究では、タスク並列モデルで記述された密行列積アプリケーションの実行性能を向上させることを目的としたタスクスケジューリング方式を実装し、評価した。密行列積はその並列化が十分に研究されており、ピーク性能に極めて近い計算ができることは既に明らかになっているものであるが、このような計算主体の実アプリケーションをタスク並列処理系で実行し、詳細な評価を行う研究は、今後のタスク並列処理系の発展の上で大きな意義があると考え、密行列積アプリケーションをタスク並列処理系で実行し、その性能について議論した研究には [6] があるが、本研究ではより多くのコア数で、よりピーク性能に近い領域で評価を行った。

この研究により、対象とする行列のサイズが大きくなると、注意深く実装されたタスク並列処理系が Lazy Task Creation に基づくタスクスケジューリングによって理想的な台数効果を実現できること、そうでないときは、提案するタスクスケジューリング方式によって台数効果を向上できる余地があったこと、密行列積と同様の特性を持つアプリケーションでは、提案するタスクスケジューリング方式が性能向上をもたらす可能性が十分にあることを示した。

2 タスク並列による密行列積

評価に用いた、密行列積をタスク並列に行なうアルゴリズムを記載する。これは、行列を縦横に再帰的にブロック分割するものである。

Algorithm 1 - タスク並列による密行列積.

```

/* Calculate C += AB */
/* A: (M,K) matrix, B: (K,N) matrix, C: (M,N) matrix */
void mm_recursive(const float *A, const float *B, float *C,
                  int M, int N, int K) {
    if (M, N, K <= LEAF_SIZE)
        /* Actually calculate C += AB for small A, B, C */
        mm_leaf(A, B, C, M, N, K);
    else if (max(M, N, K) == M) {
        /* Split A horizontally */

```

Implementation and Evaluation of a Locality-Aware Scheduler for Recursively Spawning Tasks

Sho NAKATANI*, Jun NAKASHIMA†, Kenjiro TAURA†, and Takashi CHIKAYAMA‡

*Faculty of Engineering, the University of Tokyo

†Graduate School of Information Science and Technology, the University of Tokyo

‡Graduate School of Engineering, the University of Tokyo

```

/* Create tasks by 'spawn' */
spawn mm_recursive(A_top, B, C_top, M/2, N, K);
spawn mm_recursive(A_bottom, B, C_bottom, M/2, N, K);
sync; /* Wait for the both tasks to finish */
} else if (max(M, N, K) == N) {
    /* Split B vertically */
    spawn mm_recursive(A, B_left, C_left, M, N/2, K);
    spawn mm_recursive(A, B_right, C_right, M, N/2, K);
    sync;
} else {
    /* Split A vertically and B horizontally */
    mm_recursive(A_left, B_top, C, M, N, K/2);
    mm_recursive(A_right, B_bottom, C, M, N, K/2);
}
}

void main() {
    mm_zero(C, M, N); /* Set each element of C zero */
    mm_recursive(A, B, C, M, N, K);
}

```

このアプリケーションを逐次実行した場合、キャッシュを効率よく利用できる。例えば、行列 A, B, C が L2 キャッシュに収まるサイズであったとしよう。 $C += AB$ の計算を上記の方法によって行なう際、 $C_{top} += A_{top}B$, $C_{bottom} += A_{bottom}B$ と分割し、 A_{top}, B, C_{top} は L1 キャッシュに収まるとする。 $C_{top} += A_{top}B$ の計算では、 A_{top}, B, C_{top} を L1 キャッシュに載せるだけのキャッシュミスが生じるが、 $C_{top} += A_{top}B$ の計算が分割を繰り返して進む間は、対象となるデータは全て L1 キャッシュに収まった状態となる。このように、再帰的な記述により自然にメモリの階層性を活用することができている。

この特性のため、再帰的な密行列積の実行効率性は、計算/通信比で議論できる。行列 A, B, C が l レベルキャッシュに収まるサイズであったとき、これら行列を l レベルキャッシュに載せるコストは、行列のデータサイズ $MK + KN + MN$ に比例する。一方で、 $C += AB$ の計算に要する浮動小数点演算の回数は $2MNK$ である。前者を「通信」、後者を「計算」のコストと呼んだとき、計算/通信比は $MNK/(MK + KN + MN)$ に比例する。計算/通信比が高いほど計算で通信のコストを隠蔽しやすくなるため、その比が高くなる大きな密行列積ほど高い性能になる。

表 1 - 実験に使用したマシンの構成と、密行列積アプリケーションのリーフサイズ.

Processor	Intel E7540 @ 2.0GHz (Nehalem-EX) 6 Cores × 4 Sockets
Cache	L1:32KB, L2:256KB L3:18MB (Per Socket)
Peak Performance	8 flops/clock × 2.0 GHz × 24 = 384 GFLOPS
MM Leaf	$M = N = K = 32$

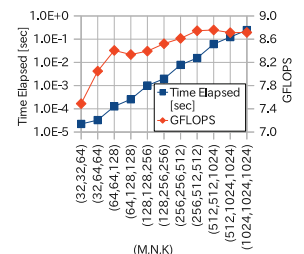


図 1 - 行列サイズと逐次性能の関係.

実際に表 1 の構成の共有メモリマシンで、種々のサイズの密行列積を実行した場合の性能を示す (図 1)。行列のデータはキャッシュに載らずメインメモリにのみ存在する初期状態で計測を行った。なお、以下すべての密行列積の評価において表 1 のマシンを使っており、実際に浮動小数点演算を行なうリーフには $M = N = K = 32$ のサイズを与えている。

必ずしも行列サイズの大小と実行性能の高低が一致しないことと、行列サイズが大きくなると 1 コアでの理論性能 384GFLOPS/24 = 14GFLOPS が出ていないことは、リーフ計算の最適化が不十分であることを反映している。とはいえ、この結果から行列のサイズが大きくなるほど実行性能が向上する傾向が見て取れる。

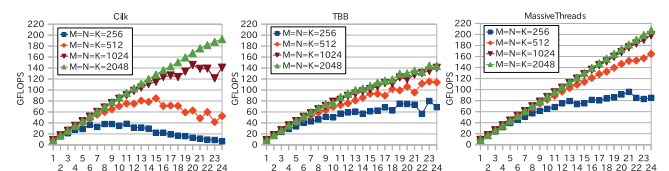


図 2 - 各種タスク並列処理系の密行列積アプリケーション実行性能.

ここで、密行列積アプリケーションを Cilk, Intel TBB と、我々の開発するタスク並列処理系 MassiveThreads [7] *1 で実行した際の性能を示す (図 2)。ここでは、Lazy Task Creation に基づくタスクスケジューラを実装した MassiveThreads と Cilk が良好な性能を与えている。本研究では、更なる性能向上を目指すタスクスケジューラを MassiveThreads に実装することとする。

MassiveThreads での実行性能を見ると、行列サイズが大きくなるとは優れた台数効果を示している。これは、ワークスティーリングの結果得られるタスクが、図 1 で高い逐次実行性能を示すような大きなサイズの密行列積を担当しているためと考えられる (この点については、評価の節でも触れる)。逆にサイズの小さい密行列積を実行させると、ワークスティーリングによって奪うタスクの計算/通信比が不十分のため、実行効率が低下し、台数効果が発揮されていない。

そこで、ランダムなワークスティーリングよりも大きなサイズの密行列積を各ワーカが実行できるようなタスクスケジューラを提案する。

3 提案手法: Depth-Aware Work Stealing

図 1 に示された結果や計算/通信比の観点から、ワークスティーリングの際に、なるべくサイズの大きな密行列積を担当しているタスクを持ったワーカを victim とする戦略が考えられる。これを、元のサイズの密行列積を担当するタスクを根とし、タスクの生成関係を親子関係と捉えた木構造から考えると、なるべく根からの距離が小さい (深さの浅い) タスクを持つワーカを victim として選ぶことに対応する。この戦略を Depth-Aware Work Stealing: DAWS とし提案する。

Lazy Task Creation では、victim を選出した後は victim が最も前に生成したタスクを盗むので、victim の持つタスクのうち最も深さの浅いタスクを奪うことになる。DAWS は、victim の候補を複数ランダムに選び出し、その候補同士で奪うことのできるタスクの深さを比較し、最も浅いタスクを持ったワーカを victim として確定する戦略である。Victim の候補数は 1 以上ワーカ数未満から自由に選ぶことができるが、候補数を 1 にしたときは、ランダムなワークスティーリングと同様の動作になる。

このような手法自体は新しいものではない。[5] では、DAWS と同様の手法を並列木探索に適用し、線形やそれ以上の台数効果を得ている。

本研究は、タスクの持つ計算/通信比に着目し、タスク木構造の深さが浅いタスクほど計算効率が上がるような実アプリケーションに対して、DAWS の戦略を適用し評価したという点で新規性を持つ。

DAWS は MassiveThreads に実装し、Lazy Task Creation と同様のタスク生成・実行と組み合わせたタスクスケジューリング機構を作成した。

4 評価 1: DAWS での密行列積の性能

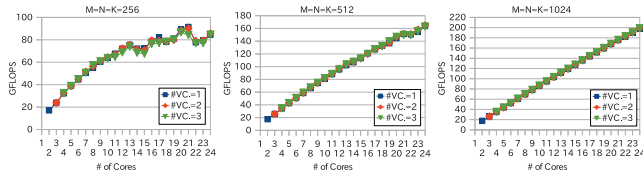


図 3 - 各種行列サイズで victim を depth-aware に選んだ場合の密行列積アプリケーションの実行性能の変化。"VC" は victim 候補数を表す。

DAWS を密行列積に適用した際の実行性能を示す (図 3)。図から、DAWS の効果が顕著には現れていないことが分かるので、次節からこの原因を考察する。

5 評価 2: DAWS により奪うタスクの行列サイズ

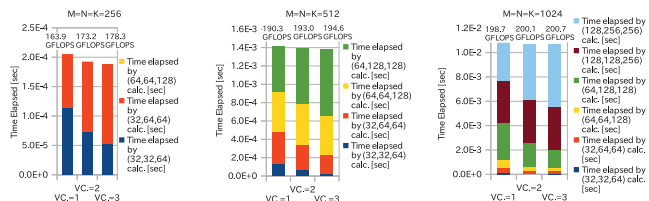


図 4 - あるワーカがワークスティーリング後に計算した密行列積のサイズと、図 1 から近似的に求まる所要時間。"VC" は victim 候補数。棒グラフ上部の GFLOPS は、このグラフの示す実行時間だけで密行列積が完了したと見た場合の実行性能。

あるワーカがサイズ (M, N, K) の密行列積を担当するタスクを奪い、他のワーカに奪われることなくそのタスクの実行を終えたなら、

そのワーカはサイズ (M, N, K) の密行列積を実行したと言える。このように各ワーカの処理した密行列積のサイズを追跡し、そのサイズの実行時間を図 1 のデータで得られたものと想定したとき、24 個の各ワーカがそれぞれのサイズの密行列積で費やした時間を示す (図 4)。 $M = N = K = 256$ の計算では、victim 候補数を増やすことで実行効率が比較的高い $(32, 64, 64)$ の計算に掛ける時間が、実行効率の低い $(32, 32, 64)$ の計算に掛ける時間に対して増加して、そのために全体の実行時間を減らせることが分かる。逆に $M = N = K = 512, 1024$ のときは、victim 候補数によらず大部分の時間が、十分に計算/通信比が大きく実行効率も出せるサイズの密行列積に費やされており、そのため DAWS は効果が薄いことが分かる。

次の節では $M = N = K = 256$ での DAWS の効果を詳細に検討する。

6 評価 3: DAWS が適するパラメタでの密行列積性能

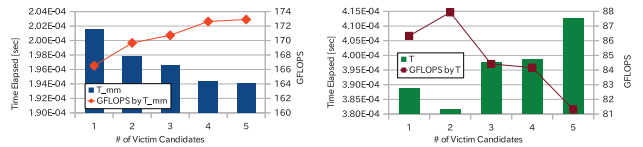


図 5 - $M = N = K = 256$, 24 コアでの DAWS の効果。GFLOPS by $(T | T_{mm})$ は、それぞれの値から計算される実行性能を示す。縦軸が 0 から始まっていないことに注意。

DAWS によりワーカが奪うタスクの計算/通信比が高くなれば、そのタスクの担当する密行列積の実行効率は高くなる。密行列積の全体の実行時間を T 、あるワーカが密行列積を実際に計算するリーフ部分に要す時間を T_{mm} としたとき、victim 候補数の増加に伴い T_{mm} が減少することが予想される。図 4 に基づき、特にその傾向が顕著に現れやすい $M = N = K = 256$ かつ 24 コアという条件での測定結果を示す (図 5)。

この結果からも、行列サイズが $M = N = K = 256$ と小さい場合には、DAWS により奪うタスクの計算/通信比が向上し、行列計算自体のコスト T_{mm} が下がることが分かる。一方、実際の実行時間である T は、victim 候補数が増えるほど増加する傾向にあることも観測できる。この理由として、victim 候補数が多いほど候補間でタスクの深さを比べ victim を決定するまでの時間が増加することが考えられる。

7 結論

本研究では、DAWS を Lazy Task Creation のタスク生成・実行ポリシーと組み合わせて使用することにより、タスクの親子関係を示した木構造の深さが浅いタスクほど計算/通信比が高い処理を担当するようなアプリケーションの実行効率を向上させる余地があることを示した。その一方で、DAWS の victim 候補数はある程度に抑えなければ、victim 決定時のオーバーヘッドにより実行効率を落とす可能性があることを指摘した。更に、アプリケーションやパラメタによってはランダムなワークスティーリングでも十分に計算/通信比の高いタスクを奪うことができ、その結果良い台数効果を得られることを実測により示した。

今後の課題としては、コア数を更に増やした共有メモリ環境において、より大きな密行列積をスケールさせる試みが挙げられる。コア数が多くなれば、各ワーカの担当する密行列積のサイズは小さくなるため、十分な計算/通信比を得るために、DAWS のような戦略が有効になることが考えられる。同様の試みは、分散メモリ環境についても行なうことができるので、これも本研究の今後の課題としていきたい。

参考文献

- [1] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.
- [2] G. Chu, C. Schulte, and P. J. Stuckey. Confidence-based work stealing in parallel constraint programming. In *Proceedings of the 15th international conference on Principles and practice of constraint programming*, CP'09, pp. 226–241, Berlin, Heidelberg, 2009. Springer-Verlag.
- [3] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine. Pfunc: modern task parallelism for modern high performance computing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pp. 43:1–43:11, New York, NY, USA, 2009. ACM.
- [4] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 2:264–280, July 1991.
- [5] C. Schulte. Parallel search made simple. "Technical Report" "TRA9/00", "School of Computing, National University of Singapore", "55 Science Drive 2, Singapore 117599", sep 2000. "To appear."
- [6] G. Tsilikas and M. Fleury. Matrix multiplication performance on commodity shared-memory multiprocessors. In *Proceedings of the international conference on Parallel Computing in Electrical Engineering*, pp. 13–18, Washington, DC, USA, 2004. IEEE Computer Society.
- [7] 中島 高効率な I/O と軽量性を両立するマルチスレッド処理系の設計と実装. Master's thesis, 東京大学, March 2011.

*1 <http://code.google.com/p/massivetthreads/> にて公開。