Vol.2016-HPC-155 No.25
2016/8/9

タスクの依存性を用いたOpenMPプログラムのNUMA最適化

李 珍泌[1,a)]　津金 佳祐[2,b)]　村井 均[1,c)]　佐藤 三久[1,2,d)]

概要：現在の多くの高性能計算機システムにおいて、プロセッサのコア数の増加に伴い、NUMA アーキテクチャの普及および複雑化が進んでいる。最新の Intel プロセッサなどは一つのチップの中でも複数の NUMA ノードを持つアーキテクチャが登場し、それを使いこなす並列プログラミング手法が求められている。OpenMP の仕様の発展によりループ文のワークシェアリングなどによるデータ並列化においては NUMA アーキテクチャを使いこなす技法が整備されつつあるが、task 指示文などによるタスク並列化においてはまだメモリ局所化を明示的に意識したプログラミングができない。本研究では OpenMP 4.0 から導入された task 指示文の depend 節を用いて NUMA ノードの判定を行うことでプログラマが NUMA ノードを意識したタスク並列化を記述できるように GCC OpenMP ランタイムの拡張を行った。データ初期化時に NUMA API を用いた明示的なメモリ割り当てを行い、ランタイムによってタスクの実行ノードをメモリ割り当てに合わせることで無用なノード間のメモリ転送を減らし、並列性能を向上させる。KASTORS ベンチマークを用いた性能評価の結果、memory-intensive な Jacobi カーネルの場合、元の GCC コンパイラに比べて 2 倍の性能向上が得られた。

1. はじめに

高い電力性能比を達成するマルチコア、またはメニーコアアーキテクチャは現在の多くの高性能計算機システムで用いられている。コア数の増加に伴い、メモリバンド幅の制約から一定数のコアを一つのグループとしてメモリ管理を行う NUMA アーキテクチャが普及してきている。NUMA アーキテクチャの構成も次第に複雑化しており、従来ソケット単位で実装されていたものが、現在はチップ内でも現れるようになった (最新の Intel アーキテクチャでは Cluster-On-Die (COD) モードを ON にすることによってチップ内で複数の NUMA ノードを持つ)。このトレンドは今後も続くと予想され、NUMA アーキテクチャの性能を引き出すプログラミングモデルの開発が求められている。

OpenMP は HPC 分野のスレッドレベル並列プログラミングにおける *de facto standard* の言語規格である。従来の仕様はループ文のワークシェアリングによるデータ並列化を中心としたものであった。OpenMP 3.0 では task 指示文によるタスク並列化が導入され、動的なタスク生成による並列化が記述できるようになった。プロセッサのコア数が増加するにつれて従来のデータ並列化や OpenMP 3.0 のタスク並列化でみられるスレッドチームのグローバルな同期 (チーム内のスレッドがすべて参加する同期) のコストがアプリケーションの性能を低下させるようになってきた。このような問題を改善するために OpenMP 4.0 からは depend 節によるタスク間の依存性の記述が導入された。スレッドチーム内のグローバルな同期を依存性のあるタスク同士のみの細粒度な同期に置き換えることで並列性能を改善する。

OpenMP 4.0 では環境変数 OMP_PROC_BIND などを用いることでコアとスレッドの affinity を指定することができ、それを用いることで NUMA アーキテクチャ向けの最適化を行う。しかし、task 指示文においてはこのような環境や技法が整備されておらず、性能はランタイムによる最適化に依存する。本研究は OpenMP タスク並列化において NUMA アーキテクチャ向けの最適化を記述できるプログラミング技法やそれをサポートする環境を提供することを目的とする。タスクが実行される NUMA ノードをユーザが意識してプログラムできるようにすることで明示的な最適化ができるプログラミング環境を目指す。

本稿の構成は次のようである。第 2 章では関連研究を挙

[1] 理化学研究所 計算科学研究機構
RIKEN Advanced Institute for Computational Science
[2] 筑波大学
University of Tsukuba
[a)] jinpil.lee@riken.jp
[b)] tsugane@hpcs.cs.tsukuba.ac.jp
[c)] h-murai@riken.jp
[d)] msato@riken.jp

ⓒ 2016 Information Processing Society of Japan

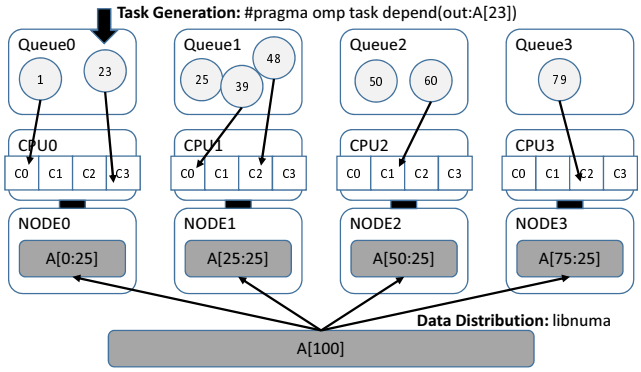

図 1　NUMA ノード割り当てと depend 節による判定

げ、本研究との違いを述べる。第 3 章では本研究の概要や GCC コンパイラを用いた実装について述べる。第 4 章では実装したランタイムや NUMA API を用いた KASTORS ベンチマークの最適化について述べる。第 5 章では元の GCC ランタイムとの性能比較を行う。第 6 章では結論と今後の課題について述べる。

2. 関連研究

Barcelona OpenMP Task Suite (BOTS)[1][2] は OpenMP 3.0 の task 指示文を用いて SparseLU、FFT、NQueens などのカーネルのタスク並列化を行っている。Inria で開発されている KASTORS ベンチマーク (以後、KASTORS)[3] は BOTS の一部のカーネルをベースに OpenMP 4.0 の depend 節を用いてタスク間の依存性を用いた並列化を行ったものである。Virouleau ら [4] はタスク並列化におけるスレッドチーム内のグローバルな同期を依存するタスクのみで行われる細粒度な同期に置き換えることで並列性能が向上することを示している。

NUMA アーキテクチャ向けタスクスケジューリングについて様々な研究が行われており [5][6][7][8]、多くが再帰アルゴリズムを対象にしたワークスティーリングに関するものである。本研究は再帰アルゴリズムを主なターゲットとせず、OpenMP depend 節で記述されるデータフロー形式のタスク並列化を対象とする。Muddukrishna ら [9] は NUMA API を用いた明示的なメモリ割り当てと独自のスケジューリングアルゴリズムを用いることで並列性能が向上することを示している。本研究のアプローチもこれに類似したものであるが、実行 NUMA ノードをランタイムのみで決定するのではなく依存性の情報を用いることでユーザが明示的に行う。

3. タスク並列化における NUMA 最適化手法の検討

NUMA アーキテクチャは名前の通り、コアによって異なるメモリ性能を持つメモリアーキテクチャである。データの局所化によるリモートメモリアクセスの削減が有効であり、OpenMP のタスク並列化でも同じことが言える。タスクがアクセスするデータが割り当てられた NUMA ノードと近いコアで実行されるときに本来のメモリ性能を引き出せる。本章ではタスクが実行される NUMA ノードを意識したプログラミングを実現するために task 指示文の depend 節を用いた OpenMP ランタイムの拡張を行う。

3.1 OpenMP depend 節を用いた実行 NUMA ノードの判定

OpenMP 4.0 から導入された depend 節によってタスク間の依存性を記述できるようになった。タスクがデータを扱う場合、入力依存 (input dependency) にタスクが読み込むデータを、出力依存 (output dependency) にタスクが書き出すデータを記述することで依存性を表現する。図 1 に depend 節を用いたタスク並列化とその実行を NUMA アーキテクチャ向けに最適化する手法を示す。あるアプリケーションが配列 A を処理するとき、これを OpenMP の task 指示文で並列化すると仮定する。配列の各要素を一つのタスクで処理する。実行環境は 4 つの NUMA ノードを持ち、各ノードに一つのプロセッサが直結されているとする。

配列がマスタースレッド (スレッド 0) によって初期化された場合、CPU0 以外にスケジュールされたタスクはメモリデータを NODE0 からコピーしなければならない。よってリモートメモリアクセスによる並列性能の低下を招く。配列を *libnuma* のような NUMA API を用いることで NUMA ノードに分散させ、データが割り当てられたノードでタスクを実行するようにするとこのような問題は発生しない。図 1 では各 NUMA ノードに 25 個の要素を block 分割で割り当てている。

parallel for 指示文などによってループ文の並列化を行う場合はスレッドが特定の NUMA ノードにアクセスすることをある程度明示的に指定することができる (環境変数 OMP_PROC_BIND を用いるなど)。しかし task 指示文による並列化ではそのような情報を与える手段がない。本研究では depend 節を用いることで実行 NUMA ノードの判定を行うことを検討する。タスクで配列に対する演算が行われる場合、出力依存によって結果が書き込まれるはずである。depend 節に記述された配列の情報を解析することで実行 NUMA ノードの判定を行う。指定された配列の要素のアドレスを解析することで割り当てられた NUMA ノードを特定し、そのノードに直結したノードにタスクを割り当てる。

提案手法ではタスクが処理するデータをプログラマが明示的にノード間で分散させる。コンパイラは depend 節の出力依存の解析によってデータが置かれた NUMA ノードを特定し、それに近いコアにタスクをスケジュールする。このような手法によってタスクとメモリ割り当てのノード

```
for (it = itold + 1; it <= itnew; it++) { ...
  for (block_x = 0; block_x < max_blocks_x; block_x++)
    for (block_y = 0; block_y < max_blocks_y; block_y++)
#pragma omp task firstprivate(block_x, block_y, xdm1, xdp1, ydp1, ydm1) ¥
  depend(out: unew[block_x * block_size: block_size][block_y * block_size: block_size]) ¥
  depend(in: f[...], u[...], ... )
      compute_estimate(block_x, block_y, u_, unew_, f_, dx, dy, nx, ny, block_size);
```

図 2　Jacobi カーネルのソースコード

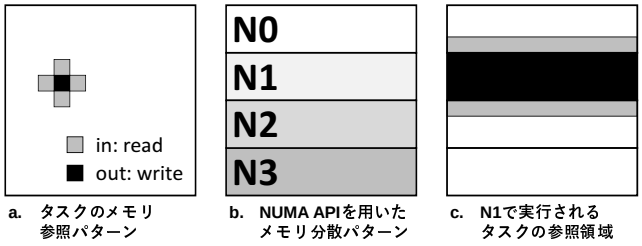

図 3　Jacobi カーネルのメモリ参照と NUMA 割り当て

を一致させ、無用なリモートメモリアクセスを削減することができる。

3.2　ランタイムの実装

本研究では GNU Compiler Collection (GCC) 5.3.0 をベースにランタイムの拡張を行った。GCC 5.3.0 は OpenMP 4.0 を実装しているため、depend 節をサポートする。コンパイラが depend 節で指定された変数のアドレスをタスク生成関数 GOMP_task() の引数リストに入れる。変数のアドレスは依存性グラフを表すハッシュテーブルに用いられるが、本研究ではこれを用いて NUMA ノードの判定を行う。Linux システムコールである get_mempolicy() 関数を呼び出して、与えられたアドレスがどの NUMA ノードに割り当てられているか判定を行う。ノード番号が得られたらそれに対応するタスクキューに生成されたタスクを入れる。

元の GCC 実装は一つのグローバルタスクキューのみを持ち、すべてのコアがキューからタスクを実行する。変更されたランタイムでは NUMA ノード毎に一つの専用タスクキューを持つように変更した。depend 節によって NUMA ノードが特定された場合はこのキューにタスクがスケジュールされる。ノードが特定されなかった場合や depend 節が記述されなかった場合はグローバルタスクキューにタスクがスケジュールされる。キューが空になっていない限り、コアは専用タスクキューからタスクを実行する。キューが空になった場合はグローバルタスクキューからタスクを実行する。グローバルタスクキューにもタスクが存在しない場合は他の NUMA ノードの専用タスクキューからタスクを実行するように実装を行った。他の NUMA ノードやグローバルキューからタスクを実行する場合はリモートメモリアクセスが発生する可能性が高くなるが、その代わりにコア間のスレッド実行のロードバランスが改善される。

4.　KASTORS カーネルの最適化

本章では拡張 GCC ランタイムと NUMA API を用いた KASTORS の最適化について説明する。KASTORS は Jacobi、SparseLU、Strassen の三つのベンチマークカーネルで構成される。各カーネルに対して OpenMP 3.0 相当の task 指示文を用いた実装 (以後、TASK バージョン) と OpenMP 4.0 以後の task depend 節を用いた実装 (以後、TASK DEP バージョン) が存在する。TASK バージョンは BOTS と同等のものである。TASK DEP バージョンは TASK バージョンでみられるスレッドチーム内のグローバルな同期がタスク依存性によるスレッド同士の同期に置き換えられている。

4.1　Jacobi カーネルの最適化

Jacobi カーネルは 2 次元ポワソン方程式のソルバーを実装したものである。KASTORS の Jacobi カーネルは TASK、TASK DEP バージョン以外に parallel for 指示文を用いてループのワークシェアリングを行った実装 (FOR バージョン) が存在する。2 次元ステンシル計算に分類され、memory-intensive なカーネルである。図 2 に Jacobi カーネルの TASK DEP バージョンのソースコードを示す。各 time step 毎にデータ配列 f と u の値が参照され、unew の更新が行われる (compute_estimate())。要素の計算毎にタスクを生成することはオーバーヘッドが大きいため、ユーザから与えられた大きさで 2 次元ブロッキングを行い、タスクに割り当てる。図 3.a に Jacobi カーネルのタスクのメモリ参照パターンを示す。黒い領域がタスクによって値が更新される部分である (out:unew)。灰色の領域は参照のみが行われる領域を示す (in:f, u)。Jacobi カーネルはステンシル計算であるため、データの参照はタスクが書き出すデータ (黒い領域) の周辺に限られる (灰色の領域)。メモリ参照の局所性が高いため、実行 NUMA ノードとデータの割り当てノードが一致するともっとも高いメモリ性能を達成する。

元のソースコードではデータ配列の初期化を task 指示文によって行っている。タスクを実行するコアの NUMA

```
for (ii=kk+1; ii<matrix_size; ii++)
  for (jj=kk+1; jj<matrix_size; jj++) {
    if (BENCH[ii*matrix_size+jj]==NULL) BENCH[ii*matrix_size+jj] = allocate_clean_block_numa(submatrix_size, jj);
#pragma omp task firstprivate(kk, jj, ii) shared(BENCH) ¥
  depend(in: BENCH[ii*matrix_size+kk:submatrix_size*submatrix_size],
             BENCH[kk*matrix_size+jj:submatrix_size*submatrix_size]) ¥
  depend(inout: BENCH[ii*matrix_size+jj:submatrix_size*submatrix_size])
    bmod(BENCH[ii*matrix_size+kk], BENCH[kk*matrix_size+jj], BENCH[ii*matrix_size+jj], submatrix_size);
```

図 4　SparseLU カーネルのソースコード

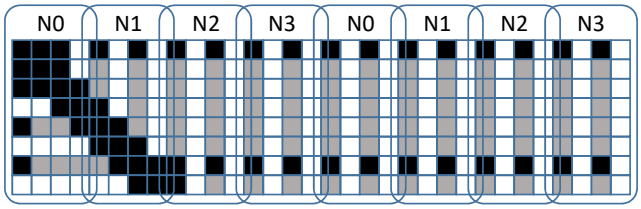

図 5　SparseLU カーネルの NUMA ノード割り当て

ノードはデータ配列の割り当てと無関係に選ばれるため、ノードが一致しない場合はリモートメモリアクセスが発生する。変更したコードでは NUMA API を用いて初期化時に NUMA ノード割り当てを明示的に指定する。図 3.b に Jacobi カーネルのノード割り当てを示す (4 ノードを仮定)。2 次元配列を 1 次元分割し、異なるノードのコアで初期化するように実装を行った。実装では NUMA API を用いたが、Linux OS の first-touch scheme を仮定して、parallel for 指示文を用いることで同等のことが実現できる。図 3.c にノード 1 (N1) に割り当てられた領域に出力依存を持つタスクがアクセスするメモリ領域を示す。タスク実行とデータ割り当てのノードが一致するとリモートメモリアクセスは周辺の一部の領域に限られる。

4.2　SparseLU カーネルの最適化

SparseLU カーネルは疎行列に対する LU 分解を行う。図 4 に SparseLU カーネルの TASK DEP バージョンのソースコードを示す。2 次元配列 BENCH は各要素に部分行列へのポインターを持つ。該当する領域が非ゼロ要素を持つときのみ、部分行列が割り当てられる。非ゼロ要素の場所は問題に依存する。図 5 に SparseLU カーネルのデフォルト入力のメモリ参照パターンを示す。黒い領域が初期化時に与えられる非ゼロ要素である。これらの領域に対して LU 分解が行われ、計算中に灰色の領域に対するメモリ割り当てが行われる。白い領域はゼロ要素なので参照されない。

図 5 からデフォルト入力の疎行列が偶数 index の列で必ず非ゼロ要素を持つことがわかる。変更したコードでは block-cyclic 分割を用いて NUMA ノード間でデータを分散させる。ブロックサイズを 4 とし、初期化時に各列の部分行列を round-robin で割り当てる (4 つのノードを仮定)。灰色の領域は計算中に割り当てられるため、カーネルコードの変更も行った。部分行列の割り当て関数 allocate_clean_block() を変更し、割り当て対象の NUMA ノード番号を配列の列番号で指定できるようにした (allocate_clean_block_numa())。

4.3　Strassen カーネルの最適化

Strassen カーネルは Strassen のアルゴリズムを用いて密行列に対する行列積を行う。このアルゴリズムでは行列を 2 次元で均等に分割し、部分行列に対する行列積を行うことで計算量を減らす。部分行列に対しても再帰的に同じ処理が行われる。部分行列に対する演算がタスクとして記述され、並列化されている。図 6 に Strassen カーネルのソースコードを示す。疎行列 A、B、C が 4 分割され、部分行列に対する再帰呼び出しが task 指示文によってタスク化されている。S1、S2 などは計算結果を一時的に格納する配列である。

図 7 に出力行列 C が再帰呼び出しによって分割される様子を示す (4 つの NUMA ノードを仮定)。配列 C は最初の再帰呼び出しで C0-3 の 4 つの行列に分割される (Depth=1)。各々の部分行列はさらに細かく分割されるが (Depth=2)、それらの配列は最初の部分行列 C0-3 の一部である。再帰呼び出し毎にタスクが生成されることを考えると、Depth=2 以後の子タスクが親タスクの部分行列 C0-3 にのみアクセスすることがわかる (A、B の場合はその通りではない)。変更した Strassen カーネルでは配列の初期化時に NUMA API を用いて図 7 のような 2 次元分割を行った。また、ランタイムによる NUMA ノードの判定でも、再帰的に子タスクによる再判定が行われることを防ぐために親タスクで判定が終了している場合は子タスクがその結果を引き継ぐようにした。

5.　性能評価

本章では元の GCC と本研究で実装した GCC のランタイムを用いて KASTORS ベンチマークの性能を測定し、性能比較を行う。表 1 に評価環境と STREAM ベンチマークによるメモリバンド幅の評価結果を示す。各 CPU は 18 個のコアを持ち、COD モードが ON になっているときに 2 つの NUMA ノードに分かれる。STREAM ベンチマークは OpenMP バージョンを用いて並列実行したものを使う。OpenMP の環境変数は OMP_PROC_BIND に

ⓒ 2016 Information Processing Society of Japan

```
static void OptimizedStrassenMultiply_par(double *C, double *A, double *B, ...) { ...
#pragma omp task untied depend(in: S3, S7) depend(out: C22) // C3
   OptimizedStrassenMultiply_par(C22, S3, S7, QuadrantSize, ..., Depth+1, cutoff_depth, cutoff_size);
#pragma omp task untied depend(in: A12, B21) depend(out: C) // C0
   OptimizedStrassenMultiply_par(C, A12, B21, QuadrantSize, ..., Depth+1, cutoff_depth, cutoff_size);
#pragma omp task untied depend(in: S4, B22) depend(out: C12) // C1
   OptimizedStrassenMultiply_par(C12, S4, B22, QuadrantSize, ..., Depth+1, cutoff_depth, cutoff_size);
#pragma omp task untied depend(in: A22, S8) depend(out: C21) // C2
   OptimizedStrassenMultiply_par(C21, A22, S8, QuadrantSize, ..., Depth+1, cutoff_depth, cutoff_size);
```

図 6　Strassen カーネルのソースコード

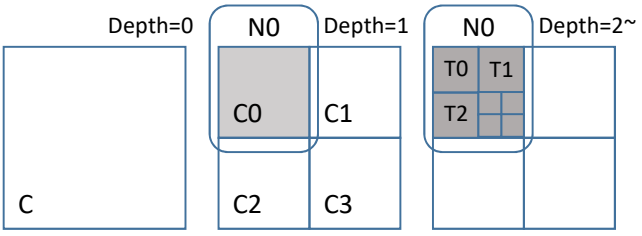

図 7　Strassen カーネルの NUMA ノード割り当て

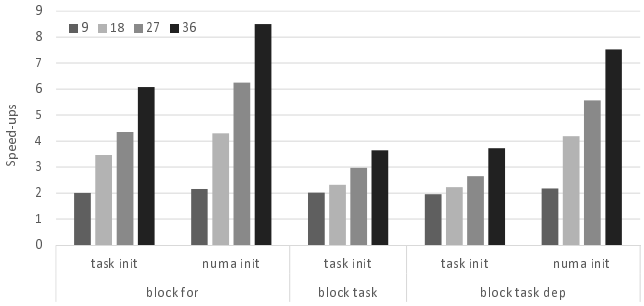

図 8　Jacobi カーネルの評価結果

CLOSE を設定することでスレッドチームがなるべく少ない数の NUMA ノードに集まるようにした。同様の設定を KASTORS ベンチマークカーネルの評価にも用いている。

5.1　Jacobi カーネルの評価結果

図 8 に逐次コードと比較した Jacobi カーネルの性能向上率を示す。行列サイズは 16384x16384 で、ブロックサイズは 1024 である。*task init* は元のコードで実装されている通りに task 指示文によって配列の初期化を行った結果である。*numa init* は NUMA API を用いることで図 3 のようなデータの分散を行った結果を示す。parallel for による並列化は本研究の範囲外であるため、FOR バージョンの *numa init* は元の GCC でコンパイルされた結果である。

評価結果の中でもっとも高い性能を見せたのは FOR バージョンの *numa init* である。FOR バージョンの *task init* はブロックの割り当てとコアの実行ノードが一致しない。その結果、リモートメモリアクセスが発生し、本来並列性が高いステンシル計算であるにも関わらず性能向上率がスレッド数に比例して伸びない結果になっている。TASK や TASK DEP バージョンの場合も同じことが言える。FOR や TASK DEP バージョンの *numa init* はデータの初期化時に配列を各ノードに分散させている。分散された NUMA ノードで計算を行うために FOR バージョンは parallel for 指示文によるループワークシェアリングを行っており、TASK DEP バージョンは本研究で実装した depend 節からの NUMA ノード判定を用いている。FOR バージョンの並列処理は静的にスケジュールされており、TASK DEP バージョンのようなランタイムのオーバーヘッドが少ない。そのため、同等のメモリアクセスを実現している TASK DEP バージョンに比べて高い性能を達成している。しかし、この結果から memory-intensive なアプリケーションのタスク並列化に本研究の提案手法が有効であることが示せた。

表 1　Evalustion Environment

Item	Name/Value
CPU	Intel (R) Xeon (R) CPU E5-2699 v3, x2 18 cores with HT, 2.30 GHz, COD on
Memory	DDR4 128GB
Stream Performance (Triad, GB/s)	1 thread: 14.49 9 threads (1 NUMA node): 21.82 18 threads (2 NUMA nodes): 43.36 27 threads (3 NUMA nodes): 65.01 36 threads (4 NUMA nodes): 86.49
OS	Red Hat Enterprise Linux Server 7.1 Linux Kernel: 3.10.0-229.7.2.el7.x86_64
Compiler	GNU Compiler gcc version 5.3.0

5.2　SparseLU カーネルの評価結果

図 9 に SparseLU カーネルの評価結果を示す。元の GCC による評価結果を *task*、*task dep* に示す。変更したコードの評価結果は *task dep (modified)* の通りである。行列のサイズは 128 であり、中の部分行列のサイズは 64 にしている。

SparseLU カーネルは計算時間のほとんどが密行列である部分行列の行列積 (図 4 の bmod() 関数など) で占められている。このような compute-intensive なカーネルでは NUMA ノード間のリモートメモリアクセス削減の効果が実行時間に反映されにくい。また、計算中に NUMA API

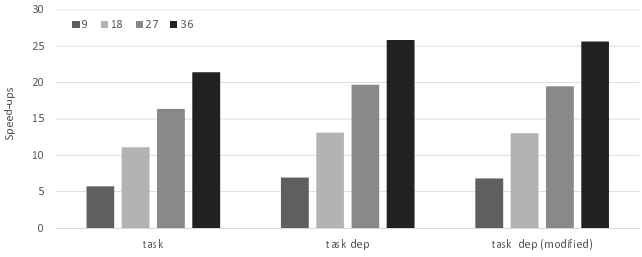

図 9 SparseLU カーネルの評価結果

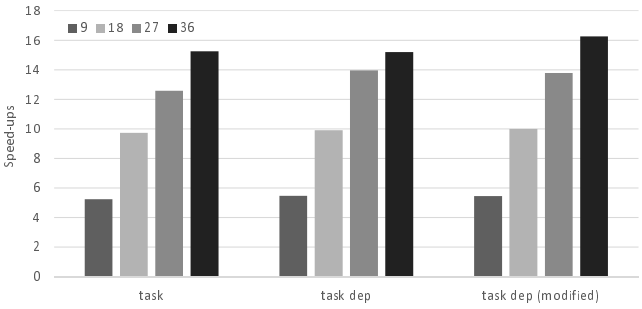

図 10 Strassen カーネルの評価結果

を呼び出すため、通常のメモリ割り当て関数に比べてオーバーヘッドが増加している。その結果、変更後のコードでも性能に変化が見られなかった。メモリ割り当てのオーバーヘッドを削減するためには NUMA API を用いず、TASK バージョンのようにタスクの中で部分行列の割り当てを行うことが考えられる。しかし、それでは depend 節による NUMA ノードの判定がタスク生成時にできない。図 5 の灰色の領域を計算時ではなく初期化時に割り当てるなどの最適化を検討中である。また、現在は NUMA API を直接用いてメモリ割り当てを行っているが、そのためには低レベルな Linux のシステムコールを呼び出す必要がある。OpenMP の言語仕様として使いやすいインターフェイスを提供することも生産性を向上させるために必要である。

5.3 Strassen カーネルの評価結果

図 10 に Strassen カーネルの評価結果を示す。SparseLU カーネルと同様、Strassen カーネルの実行時間も密行列である部分行列の行列積が大半を占める。また、出力行列 C の参照は再帰関数呼び出しで図 7 のように局所化されるが、入力行列 A、B に関してはアルゴリズムの性質上リモートメモリアクセスが発生してしまう。しかし、SparseLU カーネルのような NUMA API 関数のオーバーヘッドは存在しない。評価の結果、変更したコード (*task dep (modified)*) が元のコード (task dep) より 7%程度の性能向上を見せることが確認できた。

Strassen カーネルは再帰アルゴリズムを用いるため、本研究がターゲットとするデータフロー形式の並列化とは異なる性質を持つ。計算の途中で生成される一時部分配列に関してはランタイムによる NUMA 最適化が有効であると考えられる。再帰アルゴリズムの NUMA 最適化のために様々なタスクスケジュールアルゴリズムが提案されており、それらと組み合わせることで本研究でカバーできなかった再帰的に生成される一時部分配列の最適化が可能になる。

6. 結論と今後の課題

本研究では OpenMP タスク並列化の NUMA アーキテクチャ最適化のためのプログラミング技法として、NUMA API による明示的なメモリ割り当てと depend 節によるランタイム時のノード判定を提案した。出力依存のデータが割り当てられたノードでタスクを実行すると仮定し、明示的にデータを分散させることでプログラマがアーキテクチャとアプリケーションの両方を意識したプログラミングを行う。KASTORS ベンチマークを用いた性能評価では 4 つの NUMA ノードを持つシステムで memory-intensive な Jacobi カーネルで 2 倍の性能向上が得られた。コードの変更はデータの初期化関数やメモリ割り当て関数のみに対して行われ、OpenMP の指示文や計算のコードなどは既存のものを用いている。

今後の課題として以下のようなものが挙げられる。

- NUMA ノード割り当てを行う OpenMP のプログラミングインターフェイスを設計する。
- 実アプリケーションの使用例を見つけ、本研究で提案した手法を用いた最適化を適用する。
- タスク割り当てをより明示的にコントロールするために depend 節を用いず、実行 NUMA ノードを直接指定する OpenMP の拡張構文を検討する。
- GCC ランタイムはスレッドチームのマスタースレッドに負荷が集中する実装になっているため、タスクスケジューラを改善してタスク実行のロードバランスを改善する。

参考文献

[1] Barcelona OpenMP Task Suite (BOTS): https://pm.bsc.es/projects/bots/.
[2] Duran, A., Teruel, X., Ferrer, R., Martorell, X. and Ayguade, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP, *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, Washington, DC, USA, IEEE Computer Society, pp. 124–131 (online), DOI: 10.1109/ICPP.2009.64 (2009).
[3] KASTORS Benchmark: https://gforge.inria.fr/projects/kastors/.
[4] Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O. and Gautier, T.: Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite, *Using and Improving OpenMP for Devices, Tasks, and More: 10th International Workshop on OpenMP, IWOMP 2014, Salvador, Brazil, September 28-30, 2014. Proceedings*, Cham, Springer International

Publishing, pp. 16–29 (online), DOI: 10.1007/978-3-319-11454-5_2 (2014).

[5] Drebes, A., Heydemann, K., Drach, N., Pop, A. and Cohen, A.: Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages, *ACM Trans. Archit. Code Optim.*, Vol. 11, No. 3, pp. 30:1–30:25 (online), DOI: 10.1145/2641764 (2014).

[6] Olivier, S. L., Porterfield, A. K., Wheeler, K. B., Spiegel, M. and Prins, J. F.: OpenMP Task Scheduling Strategies for Multicore NUMA Systems, *Int. J. High Perform. Comput. Appl.*, Vol. 26, No. 2, pp. 110–124 (online), DOI: 10.1177/1094342011434065 (2012).

[7] Olivier, S. L., de Supinski, B. R., Schulz, M. and Prins, J. F.: Characterizing and Mitigating Work Time Inflation in Task Parallel Programs, *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Los Alamitos, CA, USA, IEEE Computer Society Press, pp. 65:1–65:12 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2388996.2389085⟩ (2012).

[8] Tahan, O.: Towards Efficient OpenMP Strategies for Non-Uniform Architectures, *CoRR*, Vol. abs/1411.7131 (online), available from ⟨http://arxiv.org/abs/1411.7131⟩ (2014).

[9] Muddukrishna, A., Jonsson, P. A., Vlassov, V. and Brorsson, M.: Locality-Aware Task Scheduling and Data Distribution on NUMA Systems, *OpenMP in the Era of Low Power Devices and Accelerators: 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16-18, 2013. Proceedings*, Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 156–170 (online), DOI: 10.1007/978-3-642-40698-0_12 (2013).

ⓒ 2016 Information Processing Society of Japan