Vol.2022-HPC-183 No.20
2022/3/18

A64FXにおけるタスク並列ベンチマークの性能評価

津金 佳祐[1,a)]　前田 宗則[1]　新井 正樹[1]　吉川 隆英[1]

概要：近年の主流であるメニーコアプロセッサにおいて，多数のコアを効率よく利用するためにタスク並列プログラミングモデルが注目されている．タスクに対してデータ依存を記述することで，従来のスレッド間の全体同期からタスク単位の同期とし，同期オーバヘッドを減らすことでプログラムの高速化が期待される．しかし，タスクに対してデータ依存を全て記述することや適切なタスク粒度を設定することは非常に困難であり，プログラム開発の生産性を低下させることから，我々はタスク並列で記述されたプログラムへの自動変換に関する研究開発を行っている．そこで本稿では，富士通が開発したメニーコアプロセッサである A64FX においてタスク並列ベンチマークの実装や性能評価を行い，タスク並列プログラミングの現状や優位性を報告する．タスク並列プログラミングモデルを OpenMP と OmpSs-2 とし，ベンチマークを Laplace Solver，N-body，ブロックコレスキー分解とした．実装では，OpenMP `taskyield` 指示文の挙動がコンパイラにより異なるため，動作しないことを想定したデータ依存付きタスク並列実装を示した．性能評価では，既存のデータ並列実装と比較して Laplace Solver で 16%，N-body で 15%，ブロックコレスキー分解で 42%の性能向上を確認し，タスク並列プログラミングモデルによる実装の性能の高さを示した．

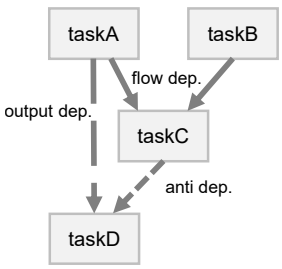
1. 序論

近年，高性能計算分野において消費電力性能比が良いことから，チップ内に大量のコアを搭載したメニーコアプロセッサが広く普及している．Intel Xeon Phi を始めとして，Marvell ThunderX，AWS Graviton，AMD EPYC など様々であり，富士通においてもスーパーコンピュータ「富岳」に搭載されている A64FX[1] を開発した．メニーコアプロセッサ向けの並列プログラミングモデルは OpenMP がデファクトスタンダードであり，ループに対して指示文を指定することでループを分割し，各スレッドに割り当てて並列実行する，データ並列が一般的である．しかし，コア数の増加によりロードインバランスが発生しやすく，データ並列が内包するスレッド間の全体同期のコストが増加し，性能低下を引き起こすという問題がある．この問題を解決するために，タスク並列プログラミングモデルが注目されている．

タスク並列は再帰処理や *while* ループなど動的な演算の並列化を容易に記述でき，処理系による負荷分散を行う点を特徴とする．さらにデータ依存やタスクフローを記述できるプログラミングモデルも登場しており，タスクの挙動を明示することで従来のスレッド間の全体同期からタスク単位の同期とし，同期オーバヘッドを減らすことでプログラムの高速化が期待される．しかし，タスクに対してデータ依存やタスクフローを全て記述することや適切なタスク粒度を設定することは非常に困難であり，プログラム開発の生産性を低下させることから，我々はデータ依存付きのタスク並列で記述されたプログラムへの自動変換に関する研究開発を行っている．そこで本稿では，タスク並列プログラミングモデルを用いてベンチマークを実装し，A64FX で性能評価を行うことで，タスク並列プログラミングモデルの現状やデータ並列と比較しての優位性を報告する．タスク並列プログラミングモデルを OpenMP と Barcelona Supercomputing Center（BSC）が開発している OmpSs-2[2][3] とする．また，プロセス並列時のタスク並列実装の性能も評価するため，タスク並列プログラミングモデルと Message Passing Interface（MPI）を組合わせたハイブリッド並列化を行う．メモリや演算律速などの様々な要因からタスク並列実装の性能調査をするため，Laplace Solver，N-body，ブロックコレスキー分解の 3 種類のベンチマークを対象とした．

本稿の構成を以下に示す．第 2 章ではタスク並列プログラミングモデルを説明する．第 3 章はタスク並列プログラミングモデルと MPI を組合わせたハイブリッド並列化によるベンチマーク実装を示し，第 4 章でその性能評価を行う．第 5 章で関連研究を紹介し，第 6 章にて結論と今後の課題を述べる．

[1] 富士通株式会社
FUJITSU LIMITED
[a)] tsugane.keisuke@fujitsu.com




```
#pragma omp parallel
#pragma omp single
{
  int A, B, C;
  #pragma omp task depend(out:A)
  A = 1; /* taskA */
  #pragma omp task depend(out:B)
  B = 2; /* taskB */
  #pragma omp task depend(in:A, B) ¥
                   depend(out:C)
  C = A + B; /* taskC */
  #pragma omp task depend(out:A)
  A = 2; /* taskD */
}
```

図 1 OpenMP の依存付きタスク並列記述の例

2. タスク並列プログラミングモデル

タスク並列をサポートするプログラミングモデルやライブラリは，OpenMP，OmpSs-2，oneAPI Threading Building Blocks（oneTBB）[4]，Taskflow[5]，UPC++[6]，DASH[7]，StarPU[8] など様々であり，タスク制御のための様々な記述方法や実装に関する研究開発が進められている．タスク制御の一つであるタスク間の同期は，データ依存とタスクフローの 2 種類の代表的な記述方式がある．本稿では，対象とした OpenMP や OmpSs-2 がサポートするデータ依存記述のみを説明する．また，OmpSs-2 は OpenMP とほぼ同じ指示文形式でタスク並列を記述するため，OmpSs-2 特有の機能を除き，以降は全て OpenMP の説明とする．

2.1 データ依存タスク

OpenMP では仕様 4.0 から登場した `depend` 節により，フロー，出力，逆依存からなるデータ依存をタスクに記述できる．`task` 指示文でタスクとして実行する範囲をブロックとし，そのブロック内で使用するスカラ，配列，ポインタ変数を依存タイプ（`in`，`out`，`inout`）と共に `depend` 節に指定する．例えば，読み込み変数ならば `in`，書き込み変数ならば `out`，そのどちらも含むならば `inout` と共に指定することで，OpenMP ランタイムがデータ依存を実行時に動的に判定し，並列実行することができる．ただし，`depend` 節に与える変数は必ずしもタスク内で使用されている必要はなく，どの依存タイプと指定するかもユーザの自由である．本稿で対象とする OpenMP タスク並列は，`parallel`+`single/master` 指示文ブロック内で `task` 指示文を実行するモデルである．`single/master` 指示文で決定された 1 スレッドがタスクを生成し，実行待機中のスレッドが生成されたタスクを並列実行する．**図 1** に OpenMP の依存付きタスク並列記述の例を示す．`task` 指示文により 4 種類のタスクが生成され，タスク毎に `depend` 節で指定された依存関係を持つ．*taskA* と *taskB* の場合，変数 *A* と *B* に対して書き込みがあるため `depend` 節に `out` と変数をそれぞれ指定し，*taskC* は変数 *A, B* の読み込みがあるため `depend` 節に `in` と変数を指定する．この場合，*taskA* と *taskB* には依存関係がないため並列に実行されるが，*taskA* と *taskC*，*taskB* と *taskC* にはフロー依存が存在するため，*taskA* と *taskB* の実行が完了するまで *taskC* は実行されない．同様に *taskA* と *taskD* は出力依存，*taskC* と *taskD* は逆依存がそれぞれ存在し，各依存関係が解消されるまで後続タスクは実行されない．

```
1 void sync(MPI_Request *req) {
2  int comp = 0;
3  MPI_Test(req, &comp, MPI_STATUS_IGNORE);
4  while(!comp) {
5   #pragma omp taskyield
6   MPI_Test(req, &comp, MPI_STATUS_IGNORE);
7  }
8 }
```

図 2 MPI+OpenMP による同期タスク内処理の例

2.2 通信，同期タスク

MPI+OpenMP のハイブリッド並列化でタスク並列実装する場合，OpenMP は MPI プロセス内のみを対象としたデータ依存記述であるため，プロセスを跨ぐデータ依存記述が別途必要となる．そこで本稿では，タスク内で MPI 通信を実行し，その通信の完了をプロセスを跨ぐ依存関係とした．通信が完了するまで他のタスクより，送信バッファは書き込みができず，受信バッファは読み込みも書き込みもできないため，それを実現する依存関係が必要となる．そこで，送信バッファを `in`，受信バッファを `out` と指定することで，送信バッファに書き込む場合には逆依存，受信バッファを読み込む場合にはフロー依存，書き込む場合には出力依存がそれぞれ発生し，プロセス内の他のデータ依存とも一致する．通信をタスク内で実行するため，MPI のマルチスレッド通信レベルは，スレッドが同時に MPI 通信を実行可能な MPI_THREAD_MULTIPLE が必要となる．

タスク内で通信を実行する場合にも，一般の MPI プログラムと同様にデッドロックを回避するプログラミングが求められる．例えば，ブロッキング通信（*MPI_Send* や *MPI_Recv* など）をタスク実行する場合，通信完了までタスクがスレッドを占有するため，スレッド数やデータ依存記述によってはデッドロックが発生する．そこで本稿では，ノンブロッキング通信（*MPI_Isend* や *MPI_Irecv* など）と *MPI_Test/Testall*，OpenMP の `taskyield` 指示文を用いた実装により，デッドロックを回避する．**図 2** に実装例を示す．*MPI_Test/Testall* は非同期的に通信完了を確認する API である．`taskyield` 指示文は `task` 指示文ブロック内で記述する指示文であり，`taskyield` 指示文を実行したタスクを一時停止させ，他に実行可能なタスクを優先実行させる．従って，*MPI_Test* を用いて通信完了を確認し（3, 6 行目），完了していなければ `taskyield` 指示文によって別タスクの実行を優先する（5 行目）．以上の処理を *while* ループ内で行うことでデッドロックを回避する．しかし，

コンパイラ毎に taskyield 指示文の挙動は異なることがわかっている [9]．GNU コンパイラの場合は何も実行されず，Clang/LLVM コンパイラの場合も退避したタスクをスタックで保持する実装のため，多くのタスクが taskyield 指示文を実行した場合にスタック領域不足で動作しなくなる恐れがある．2022 年 2 月現在，最新の GNU（11.2.0），Clang/LLVM（13.0.0）コンパイラを確認したが，GNU コンパイラは変わらず，Clang/LLVM コンパイラは実装が変わっていたが，[9] で示された taskyield 指示文の検証プログラムで動作確認ができなかった．従って，現時点で MPI+OpenMP のハイブリッド並列化でのタスク並列実行は，taskyield 指示文が動作しないことを考慮した実装が必要であると言える．

一方で，OmpSs-2 の仕様に taskyield 指示文は無く，TAMPI[10] と呼ばれる通信ライブラリを用いることを推奨している．TAMPI は MPI 通信のラッパー関数として実装されたライブラリであり，内部的に taskyield 指示文相当の処理を行いデッドロックを回避する．本稿ではタスク並列ベンチマークの実装を MPI+OpenMP，TAMPI+OmpSs-2 の組合せで実装する．

3. 実装

評価対象のベンチマークである Laplace Solver，N-body，ブロックコレスキー分解のタスク並列実装を示す．また，各ベンチマークの概要や既存のデータ並列実装に変更を加えた点も合わせて示す．タスク並列実装の基本方針は以下の通りである．

- 演算ループにタイリング（1 次元ループの場合はストリップマイニング）を適用し，1 タイルを 1 タスクとする
- 通信はタイルが演算する範囲に合わせて細分化する
- 通信と同期は別タスクとし，同期タスクにはデッドロックを回避する実装を行う

ループ単位でタスク化した場合は，演算粒度が大きく十分な並列性が得られない可能性があるため，タイリング適用後のタイル単位でタスク化する．通信を演算の粒度に合わせて行うことで通信回数は増加するが，データ依存による通信と演算のオーバラップを最大限に行う実装とする．前章で述べた通り，同期タスクによってデッドロックを引き起こす可能性があるため，それを回避する実装も示す．

3.1 タスク並列ベンチマーク実装

3.1.1 Laplace Solver

Laplace Solver は，2 次元ラプラス方程式をヤコビ法で解くベンチマークである．2 次元格子状における近傍 4 点の平均による値の更新が主なステンシル演算であり，メモリ律速なベンチマークである．Omni Compiler[11] が提供する逐次実装をベースに 2 次元ブロック分割でタスク並列実

```
1  iiend = CEIL(imax - 2, ITILE);
2  jjend = CEIL(jmax - 2, JTILE);
3  for (ii = 0; ii < iiend; ii++)
4   for (jj = 0; jj < jjend; jj++) {
5    iprev = (ii == 0) ? 0 : 1;
6    inext = (ii == (iiend - 1) ? 0 : 1;
7    jprev = (jj == 0) ? 0 : 1;
8    jnext = (jj == (jjend - 1) ? 0 : 1;
9    #pragma omp task private(i, j) firstprivate(ii, jj) \
10    depend(in:uu[ii * ITILE][jj * JTILE], \
11               uu[(ii - iprev) * ITILE][jj * JTILE], \
12               uu[(ii + inext) * ITILE][jj * JTILE], \
13               uu[ii * ITILE][(jj - jprev) * JTILE], \
14               uu[ii * ITILE][(jj + jnext) * JTILE]) \
15    depend(out:u[ii * ITILE][jj * JTILE])
16   {
17    for (i = MAX(1, ITILE * ii);
18         i < MIN(imax - 1, ITILE * ii + ITILE); i++)
19     for (j = MAX(1, JTILE * jj);
20          j < MIN(jmax - 1, JTILE * jj + JTILE); j++)
21      u[i][j] = (uu[i - 1][j] + uu[i + 1][j]
22               + uu[i][j - 1] + uu[i][j + 1]) / 4.0;
23   }
24  }
```

図 3　Laplace Solver の主要演算ループのタスク並列実装

```
1  for (jj = 0; jj < jjend; jj++) {
2   #pragma omp task firstprivate(jj) \
3    depend(in:uu[0][jj * JTILE]) \
4    depend(out:upper_req[jj])
5   {
6    MPI_Isend(&uu[1][jj * JTILE], JTILE, MPI_DOUBLE,
7              upper, jj, comm, &upper_req[jj]);
8   }
9   #pragma omp task firstprivate(jj) \
10   depend(out:uu[(iiend - 1) * ITILE][jj * JTILE], \
11              lower_req[jj])
12  {
13   MPI_Irecv(&uu[imax - 1][jj * JTILE], JTILE, MPI_DOUBLE,
14             lower, jj, comm, &lower_req[jj]);
15  }
16 }
17 for (jj = 0; jj < jjend; jj++) {
18  #pragma omp task firstprivate(jj) \
19   depend(in:uu[0][jj * JTILE]) \
20   depend(out:upper_req[jj])
21  {
22   /* Same as the function in Fig. 2 */
23   sync(&upper_req[jj]);
24  }
25  #pragma omp task firstprivate(jj) \
26   depend(out:uu[(iiend - 1) * ITILE][jj * JTILE], \
27              lower_req[jj])
28  {
29   /* Same as the function in Fig. 2 */
30   sync(&lower_req[jj]);
31  }
32 }
```

図 4　Laplace Solver の通信と同期のタスク並列実装

装を行った．**図 3** に主要演算ループのタスク並列実装を示す．2 次元ループに対してタイリングを適用し（1 ～ 4, 17 ～ 20 行目），1 タイルを 1 タスクとした（9 ～ 23 行目）．また，ステンシル演算で近傍 4 点に対するメモリアクセスがあるため，depend 節には隣接する 4 タイルを in とし（10 ～ 14 行目），更新する要素を持つタイルを out とした（15 行目）．行列の端のタイルは隣接するタイルが存在しないため，端のタイルかどうかの判定を行い（5 ～ 8 行目），その結果を依存関係として与える変数に反映する（11 ～ 14 行目）．**図 4** に通信と同期のタスク並列実装の一部を示す．処理内容は *lower* から *upper* プロセスへの通信と同期である．Laplace Solver では，イテレーション毎に別プロセスが持つ袖領域の値を更新する通信が実行される．各袖領域に対して相互に通信が実行されるため，2 次元ブロック分割の場合は，計 4 回の *MPI_Isend/Irecv* となる．データ並列実装では，for 指示文で主要演算を並列実行した後に同期をとり，1 スレッドが通信を実行する．タスク並列実装では，袖領域を持つタイルが演算され次第，その領域のみを通信することで通信と演算をオーバラップする実装とした．具体的には，まず演算と同様のタイルサイズで通信を分割，タスク化する（1 ～ 16 行目）．通信タスクが全て生成された後，対応する同期がタスク化される（17 ～ 32 行目）．同期タスクで実行される関数 *sync* は，図 2 に示した関数である．通信タスクの依存関係は，*MPI_Isend* のバッファを in，*MPI_Irecv* のバッファを out とし，各通信で用いた *MPI_Request* を out とする．同期タスクの場合は，対応する通信タスクと同様の依存関係を記述する．通信と同期タスクを同じ依存記述とすることで，各バッファが通信完了後で使用可能となるタイミングを保証する．また，*MPI_Request* を out とすることで，対になる通信と同期タスク間で出力依存が生じ，順序が入れ替わることがない．

3.1.2 N-body

N-body は，粒子間の相互作用を演算するベンチマークである．本稿で用いた実装は，OmpSs-2 が提供する全粒子間の相互作用を演算するナイーブな $O(N^2)$ のアルゴリズムであり，演算律速なベンチマークである．OmpSs-2 がタスク並列実装を提供しているため，OmpSs-2 から OpenMP への書き換えとデータ並列実装のみを行った．粒子データに対してストリップマイニングを適用し，分割したサイズ毎に task 指示文と depend 節でタスク並列実装を行った．また，Laplace Solver と比較して 1 タスクあたりの演算量が多いため，taskloop 指示文を用いてタスク内でさらに細かい粒度のタスクを生成する実装とした．

3.1.3 ブロックコレスキー分解

ブロックコレスキー分解は，正定値対象行列を下三角行列とその転置の積に分解するコレスキー分解の各処理をブロック化したベンチマークである．BLAS と LAPACK により，POTRF:コレスキー分解，TRSM:三角行列を係数行列とする行列方程式を解く，SYRK:対象行列のランクを更新，GEMM:行列積の 4 種類の演算で構成される．処理の大部分を行列積が占めており，本稿で用いる OmpSs-2 の実装では密行列を対象としているため，演算律速なベンチマークとなる．また，処理毎の依存関係が多く，データ並列実装で並列性が得られにくい特徴を持つ．OmpSs-2 が提供するタスク並列実装をベースに，OpenMP への書き換えと MPI を組合わせたハイブリッド並列化を行う．三角行列をブロック分割した場合，プロセス毎に要素数の偏りが生じるため，2 次元のブロックサイクリック分割とした．

```
1 #pragma omp task depend(out:sync_task[count], req) ...
2 {
3   sync(&req); /* Same as the function in Fig. 2 */
4 }
5 if (++count >= num_threads - 1) count = 0;
```

図 5 デッドロック回避のための同期タスクの実装

表 1 実験環境（A64FX）

CPU	A64FX 2.0GHz
Number of cores	48
Memory	HBM2 32GB
Compiler	GNU 11.2.0, OmpSs-2 2021.06
MPI	OpenMPI 4.1.2

3.2 デッドロックを回避する同期タスク

コンパイラによって taskyield 指示文の動作が保証できないため，動作しないことを考慮した同期タスクの実装を行う．**図 5** に実装を示す．各ベンチマークの同期タスクには，図 2 の関数 *sync* を用いている．その実装にさらに依存関係として一つの配列を out で depend 節に指定する（1 行目）．配列インデックスは single/master 指示文で指定された 1 スレッドが実行するカウンタの値を用い，同期タスク毎にインクリメントした値を渡す（1 行目）．その値は実行スレッド数-1 になった場合に 0 へと初期化する（5 行目）．この処理により，同期タスクはスレッド数-1 しか並列に実行されず，1 スレッドは同期以外のタスクが実行される．また，新たに out で指定した配列の出力依存によって同期順序が入れ替わることがない．同期タスクにプログラム中の出現順序を考慮した依存関係を与えるため，本実装は逐次実行した場合にデッドロックが起きないことを前提としている．以上の実装を各ベンチマークの同期タスクとして用いる．

4. 評価

実装したタスク並列ベンチマークの性能評価を行う．また，予備評価として EPCC OpenMP micro-benchmark suite[12][13] を用いてデータ並列やタスク並列で用いる指示文のオーバヘッドも調査する．実験環境として A64FX プ

表 2　実験環境（SKL）

CPU	Intel Xeon Gold 6148×2 2.4GHz
Number of cores	20×2
Memory	DDR4 2666MHz 192GB
Compiler	GNU 11.2.0, OmpSs-2 2021.06
MPI	OpenMPI 4.1.2

ロセッサが搭載された FX700 と Intel Xeon（Skylake-SP）プロセッサが搭載されたマシンを用いる．以降は各環境を「A64FX」，「SKL」と呼ぶ．ハードウェアやソフトウェアの詳細は**表 1**，**表 2** に示す通りである．

本稿で用いた FX700 は，48 コア，動作周波数 2.0GHz の A64FX プロセッサを持つ．FX700 は，「富岳」や FX1000 と比較してアシスタントコアの有無やインターコネクトの違い（Tofu インターコネクト D か InfiniBand）などがある．A64FX は Armv8.2-A をベースに Scalable Vector Extension（SVE）拡張された命令セットアーキテクチャを持ち，SIMD 長は 512bit である．メモリは High Bandwidth Memory 2（HBM2）を 32GB 搭載し，メモリバンド幅は 1024GB/s である．1 ノードは 4 つの Core Memory Group（CMG）に分けられ，1CMG あたりは 12 コアである．ソフトウェアは CMG を NUMA ノードとして扱える．CMG 間は双方向のリングバスで接続され，メモリバンド幅は 115×2GB/s である．SKL は Intel Xeon Gold 6148 プロセッサを 2 ソケット持つ構成であり，1 ソケット 20 コアで計 40 コアである．Hyper-Threading は off とした．メモリは DDR4 2666MHz を 192GB 搭載し，メモリバンド幅は 128×2GB/s である．コンパイラはどちらの環境においても GNU，OmpSs-2 とし，MPI ライブラリには OpenMPI を用いる．コンパイラオプションは*-Ofast -fopenmp* は共通で，A64FX は*-march=armv8.2-a+sve -mtune=a64fx -msve-vector-bits=512*，SKL は*-march=skylake* とした．また，ブロックコレスキー分解に用いる BLAS や LAPACK は，A64FX で富士通 BLAS/LAPACK，SKL で Intel MKL とした．

性能評価では各実験環境を 1 ノードのみ用いる．1 プロセスと NUMA ノード毎に 1 プロセス（A64FX は 4，SKL は 2 プロセス）を割り当てる 2 種類の評価を行う．スレッド数は A64FX で 4 ～ 48，SKL は 4 ～ 40 と変化させる．例えば SKL 上の評価の場合，1 プロセス 4 スレッド実行ならば，2 プロセスでは各プロセス 2 スレッド実行とし，合計スレッド数が同数の結果を比較する．タイルサイズはスレッド数毎に変更し，最も性能の良いタイルサイズの結果のみを示す．データ並列実行時の OpenMP `parallel` 指示文のスケジュールは `static` とした．以上の設定で性能のスケーリングと最大コア数使用時の実行時間の内訳を示す．タスク並列実行の場合，実行毎に各スレッドで実行される処理が変わるため，`master` スレッドのみの実行時間の内訳では比較とならない．そのため，スレッド毎に演算，通信，同期オーバヘッドなどの時間を取得し，それぞれの項目毎に全スレッドでの平均を出して実行時間の内訳とした．

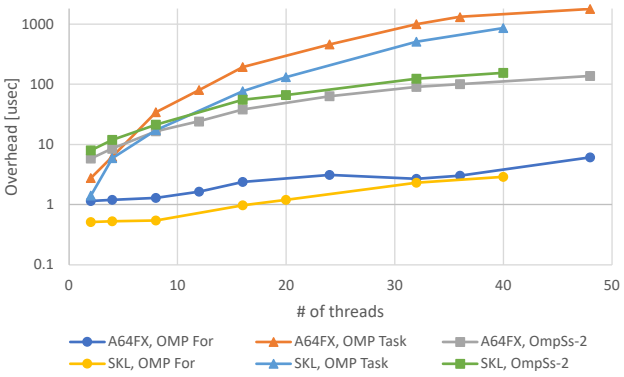


図 6　OpeMP `for`, OpenMP/OmpSs-2 `task` 指示文のオーバヘッド

4.1　予備評価

EPCC OpenMP micro-benchmark suite を用いて，各実験環境でのデータ並列やタスク並列で用いる指示文のオーバヘッドを調査する．データ並列は OpenMP `for` 指示文，タスク並列は OpenMP/OmpSs-2 `task` 指示文を対象とし，ベンチマーク集の syncbench.c の関数 *testfor*，taskbench.c の関数 *testMasterTaskGeneration* を用いた．**図 6** に性能評価を示す．`for` 指示文と `task` 指示文のオーバヘッドを比較すると，`for` 指示文はスレッド数が増加しても，A64FX，SKL で 6μsec，3μsec 程度に対して，`task` 指示文は OmpSs-2 で 130μsec，150μsec，OpenMP で 1800μsec，850μsec とどちらの環境においても非常にオーバヘッドが大きいことがわかる．以上のことから，タスク粒度は少なくとも 100μsec 以上が必要で，ステートメント単位ではなく，ループや関数呼び出しなどのまとまった処理を対象とする必要がある．また，タスク間同期による同期オーバヘッドの削減や通信と演算のオーバーラップの実現などによって，データ並列以上の並列性が得られる見込みがなければ，単にデータ並列からタスク並列実装に書き換えるだけでは性能が低下すると言える．

4.2　タスク並列ベンチマーク評価

4.2.1　Laplace Solver

図 7 に A64FX，**図 8** に SKL で実行した Laplace Solver の性能評価を示す．問題サイズを 8192×8192 とし，タイルサイズを 128×128 ～ 8192×8192 の範囲で次元毎に 2 冪で変化させた．どちらの環境においても 1 プロセス実行は複数プロセス実行と比較して性能が低く，特にタスク並列実装の性能が低い．これはデータ並列の場合，OpenMP `for` 指示文により静的にループが分割，並列実行されるため，イテレーション間で同一スレッドで実行されやすく，同じ

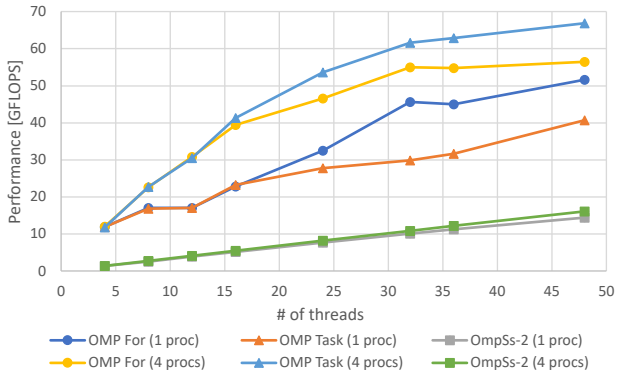


図 7　Laplace Solver の性能評価（A64FX）

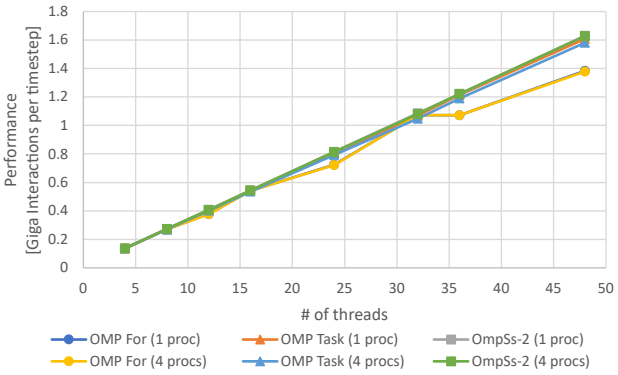


図 10　N-body の性能評価（A64FX）

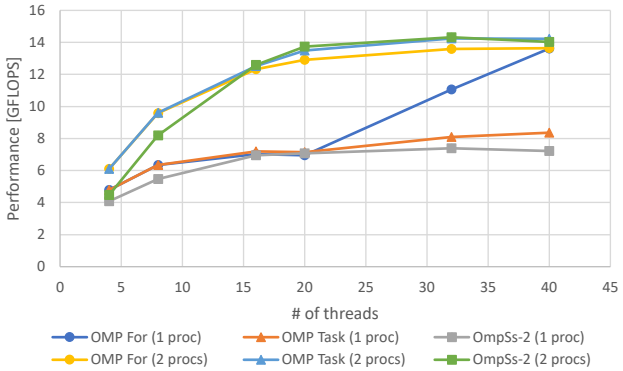


図 8　Laplace Solver の性能評価（SKL）

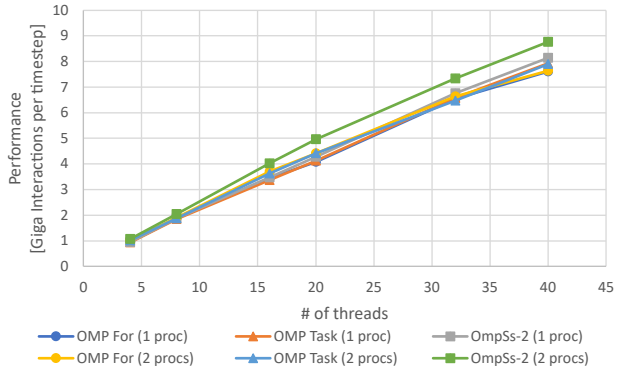


図 11　N-body の性能評価（SKL）

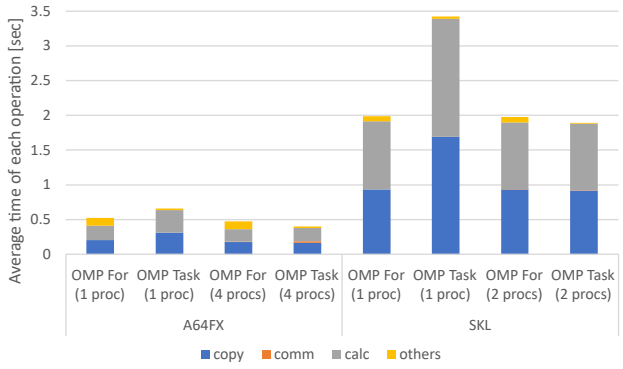


図 9　Laplace Solver の最大コア数使用時の内訳

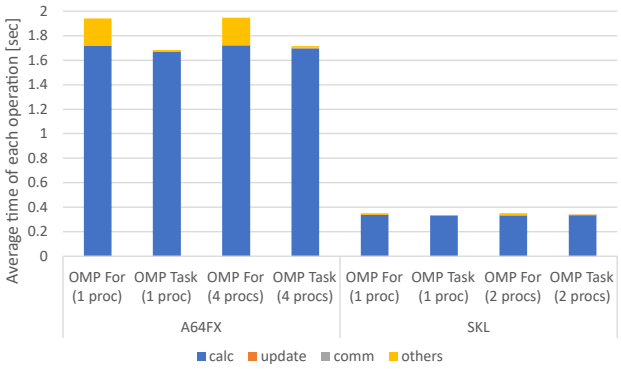


図 12　N-body の最大コア数使用時の内訳

NUMA ノード内のメモリアクセスとなりやすい．一方で，タスク並列の場合は，依存関係が解消されていて，かつスレッドに空きがあればタスクが実行されるため，NUMA ノードは考慮されない．そのため，NUMA ノードを跨いだメモリアクセスの頻発により性能が低下したと考えられる．また，データ並列の場合もステンシル演算で隣接領域のアクセスがあるため，完全に NUMA ノード内のメモリアクセスにはならず，複数プロセス実行と比較して 1 プロセス実行の性能が低い．複数プロセス実行では，データ並列と比較してタスク並列実装の性能が高く，最大コア数使用時に A64FX で 16%，SKL で 5%性能が向上した．**図 9** に最大コア数使用時の実行時間の内訳を示す．*copy* が一時配列へのメモリコピー，*comm* が袖領域通信，*calc* が 4 点ステンシル演算，*others* が実行時間から各処理の時間を引いた値であり，スレッドやタスク生成，同期などのオーバヘッドを表す．1 プロセス実行のタスク並列実装は，データ並列や複数プロセス実行と比較して *copy* と *calc* が表す演算時間が伸びており，複数プロセス実行のタスク並列実装では演算時間が伸びていないことから，NUMA ノードを跨いだメモリアクセスによる性能低下だと考えられる．また，複数プロセス実行のデータ並列とタスク並列実装を比較すると，タスク並列実装の *others* が減少しており，タスク間同期による同期オーバヘッドの削減で性能が向上したと言える．

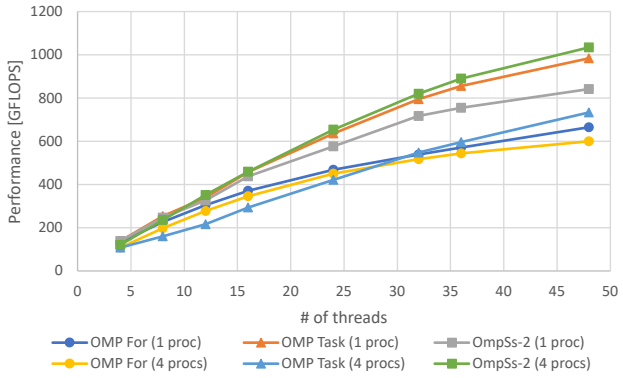


図 13 ブロックコレスキー分解の性能評価（A64FX）

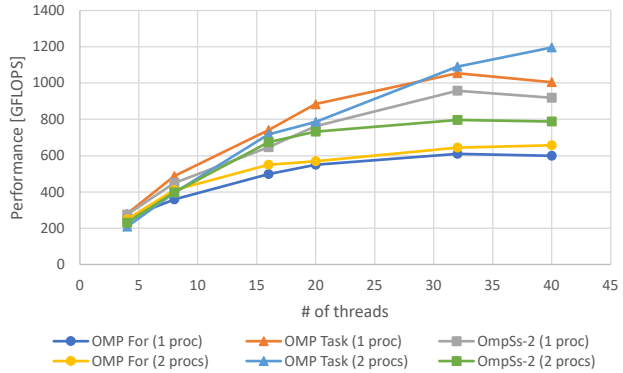


図 14 ブロックコレスキー分解の性能評価（SKL）

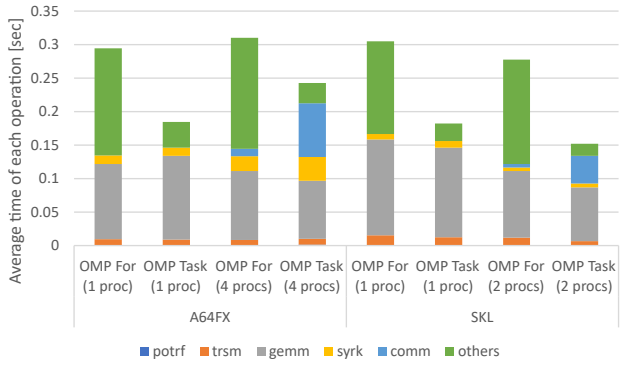


図 15 ブロックコレスキー分解の最大コア数使用時の内訳

4.2.2 N-body

図 10 に A64FX，**図 11** に SKL で実行した N-body の性能評価を示す．粒子数を 16384 とし，ストリップマイニングは 8 ～ 8192 の範囲で 2 冪に変化させたサイズを適用した．Laplace Solver の評価とは異なり，1 プロセスと複数プロセス実行で A64FX，SKL 共に性能に大きな差はない．これはベンチマークの特性で演算量が非常に大きいのと，Laplace Solver の 4 近傍のメモリアクセスと比較して，ストリップマイニングされた粒子に対する連続アクセスのみであるため，メモリアクセス時間が比較的小さく，プロセス数の違いで性能に差が出なかったと考える．最大コア数使用時に A64FX，SKL の順でデータ並列とタスク並列実装の性能を比較すると，1 プロセス実行では OpenMP で 14%，4%，OmpSs-2 で 15%，7%性能が向上した．また，複数プロセス実行も同様に比較すると OpenMP で 13%，3%，OmpSs-2 で 15%，13%性能が向上した．**図 12** に最大コア数使用時の実行時間の内訳を示す．*calc* がブロック単位での粒子相互作用の演算，*update* が粒子情報の更新，*comm* が更新した粒子の交換，*others* がスレッド，タスク関連のオーバヘッドである．A64FX の実行時間の内訳をみると，タスク並列実装の *others* が減少しており，タスク間同期による同期オーバヘッド削減が確認できる．また，SKL も僅かではあるが同様にタスク並列実装の *others* の削減を確認できた．ただし，A64FX と SKL を比較すると A64FX で主な演算部分の *calc* が非常に低速であることがわかる．A64FX と SKL のアセンブリコードを確認した結果，主な演算部分がベクトル化されていないことを確認した．よって，動作周波数や命令レイテンシの違いから，そのような性能差となったと考えるが，詳細は現在調査中である．

4.2.3 ブロックコレスキー分解

図 13 に A64FX，**図 14** に SKL で実行したブロックコレスキー分解の性能評価を示す．行列サイズを 8192×8192 とし，タイルサイズを 64×64 ～ 8192×8192 の範囲で 2 冪に変化させた．最大コア数使用時に A64FX，SKL の順でデータ並列とタスク並列実装の性能を比較すると，1 プロセス実行では OpenMP で 32%，40%，OmpSs-2 で 20%，35%性能が向上した．また，複数プロセス実行も同様に比較すると OpenMP で 18%，45%，OmpSs-2 で 42%，17%性能が向上した．**図 15** に最大コア数使用時の実行時間の内訳を示す．*potrf*，*trsm*，*syrk*，*gemm* が BLAS や LAPACK による演算，*comm* がブロックの送受信，*others* がスレッド，タスク関連のオーバヘッドである．複数プロセス実行において，データ並列と比較してタスク並列実装の *comm* の通信時間が長いことがわかる．1 回あたりの通信サイズを比較すると，Laplace Solver はタイル化した袖領域，N-body はストリップマイニングを適用した粒子数に対して，ブロックコレスキー分解の場合は 1 タイル全体と大きい．また，MPI_THREAD_MULTIPLE での通信性能が悪いことは既知の問題 [14] であり，同時に通信するスレッド数の増加と共に性能が低下する．これにより，A64FX で 4 プロセス実行した場合のメモリアクセス性能改善と同期オーバヘッドの削減以上に通信時間が伸びてしまい，性能が低下したと考える．

5. 関連研究

OpenMP や OmpSs-2 以外にもデータ依存記述によるタスク間同期を実現するプログラミングモデルやライブラリは様々ある．StarPU は INRIA が開発している CPU や演算加速機構を対象にタスク並列や負荷分散を記述可能なラ

イブラリである．OpenMP や OmpSs-2 などの指示文形式と比較すると，タスクを StarPU ランタイムで実行するための関数化や，タスク内で用いるデータを予め専用のハンドルとして定義する必要があるなど，記述は複雑な傾向にある．また，OpenMP や OmpSs-2 と同じように MPI と組み合わせることでプロセス間の依存関係が記述できる一方で，全てのプロセスが全てのタスクに到達する制約を設けることで，プログラム全体のタスクグラフを生成し，ランタイムが自動的にプロセス間のデータ依存を判断し，並列実行できるモードも実装されている．タスクスケジューリング方法が豊富に実装され，専用のオンライン/オフラインパフォーマンスツールなど様々な機能がある．我々は今後，性能評価を行う予定である．

Taskflow は University of Illinois によって開発された C++を対象にタスクフローが記述可能なライブラリである．タスクとして実行したい処理をラムダ式で記述し，*precede*，*succeed*，*gather* などの API を用いてタスクフローを記述する．GPU にも対応しており，CUDA や SYCL をタスク内で記述することができる．他にもタスクフローを記述できるプログラミングモデルやライブラリは，oneTBB，UPC++，DASH などがあり，データ依存と比較してタスクフローで記述することによる性能や記述面のメリットを調査する必要がある．

6. 結論

本稿では，タスク並列プログラミングモデルの OpenMP と OmpSs-2 を用いて既存のデータ並列ベンチマークをタスク並列実装し，A64FX と SKL で性能評価を行った．ベンチマークは，Laplace Solver，N-body，ブロックコレスキー分解の 3 種類を対象とした．タスク内で MPI 通信を実行する場合，*MPI_Test/Testall* や OpenMP `taskyield` 指示文を用いてデッドロックを回避するような実装が必要である．しかし，現状の OpenMP `taskyield` 指示文はコンパイラ毎の挙動が異なるため，動作しないことを考慮した通信，同期タスクの実装を示した．プロセス数を変化させた各ベンチマークの性能評価では，ベンチマークの特性に合わせてノード内のプロセス数を変える必要があるのは，A64FX，SKL 共に同じであり，同様の最適化が必要であると言える．また，A64FX におけるデータ並列とタスク並列実装の比較では，Laplace Solver で 16%，N-body で 15%，ブロックコレスキー分解で 42%の性能向上を示し，タスク並列プログラミングモデルによる実装の性能の高さを示した．

今後の課題として，StarPU や Taskflow などの他のデータ依存記述やタスクフロー記述のプログラミングモデルやライブラリを用いて，様々なベンチマークや実アプリケーションを実装し，タスク並列プログラミングモデルの性能を示すことが挙げられる．また，タスク並列記述はデータ並列記述と比較して複雑であるため，より簡易な記述の提案や既存プログラムからの自動変換に関する研究を進めることが挙げられる．

参考文献

[1] Fujitsu Limited, A64FX(R) Microarchitecture Manual, https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_jp_1.6.pdf, 2021.
[2] D. Alejandro, A. Eduard, B. Rosa M, L. Jesus, M. Luis, M. Xavier, P. Judit, “OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures”, Parallel Processing Letters, 2011, Vol. 21, pp. 173–193.
[3] Programming Models Group BSC, OmpSs-2 Specification, https://pm.bsc.es/ftp/ompss-2/doc/spec/OmpSs-2-Specification.pdf, 2021.
[4] oneAPI Specification 1.1-rev-1 documentation/oneTBB, https://spec.oneapi.io/versions/latest/elements/oneTBB/source/nested-index.html, 2021
[5] T. -W. Huang, C. -X. Lin, G. Guo, M. Wong, “Cpp-Taskflow: Fast Task-Based Parallel Programming Using Modern C++,” 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2019, pp. 974–983.
[6] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, K. Yelick, “UPC++: A PGAS Extension for C++”, 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014, pp. 1105-1114.
[7] K. Fürlinger, J. Gracia, A. Knüpfer, T. Fuchs, D. Hünich, P. Jungblut, R. Kowalewski, J. Schuchart, “DASH: Distributed Data Structures and Parallel Algorithms in a Global Address Space”, Software for Exascale Computing - SPPEXA 2016-2019, 2020, pp. 103–142.
[8] A. Cedric, T. Samuel, N. Raymond, W. Pierre-Andre, “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures”, Concurrency and Computation: Practice and Experience, 2011, Vol. 23, No. 2, pp. 187–198.
[9] J. Schuchart, K. Tsugane, J. Gracia, M. Sato, “The Impact of Taskyield on the Design of Tasks Communicating Through MPI”, International Workshop on OpenMP (IWOMP 2018): Evolving OpenMP for Evolving Architectures, 2018, pp. 3–17.
[10] K. Sala, J. Bellón, P. Farré, X. Teruel, J. M. Perez, A. J. Peña, D. Holmes, V. Beltran, J. Labarta, “Improving the Interoperability between MPI and Task-Based Programming Models”, EuroMPI’18: Proceedings of the 25th European MPI Users’ Group Meeting, 2018, No. 6, pp. 1–11.
[11] Omni Compiler, https://github.com/omni-compiler/omni-compiler
[12] J. M. Bull, D. O’Neill, “A microbenchmark suite for OpenMP 2.0”, ACM SIGARCH Computer Architecture News, Vol. 29, Issue 5, 2001, pp. 41–48.
[13] J. M. Bull, F. Reid, N. McDonnell, “A microbenchmark suite for OpenMP tasks”, IWOMP’12: Proceedings of the 8th international conference on OpenMP in a Heterogeneous World, 2012, pp. 271–274.
[14] K. Tsugane, J. Lee, H. Murai, M. Sato, “Multi-tasking Execution in PGAS Language XcalableMP and Communication Optimization on Many-core Clusters”, HPC Asia 2018: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, 2018, pp. 75–85.