Vol.2022-HPC-186 No.4
2022/9/26

自動タスク並列化トランスパイラの実装と評価

津金 佳祐[1,a)]　新井 正樹[1]　前田 宗則[1]　笠置 明彦[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．HPC183 での報告では，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 Xeon プロセッサ上で性能評価を行った．そこで本稿では，凸多面体モデルによるループ最適化を行うトランスパイラである Pluto を基に，逐次や MPI 並列プログラムから OpenMP のタスク並列プログラムに自動変換する機能の実装とその性能を示す．実装では，Pluto のデータ依存解析の結果を用いたタスクへのデータ依存の追加，MPI や BLAS/LAPACK 関数の対応，OpenMP の `iterator` を用いて通信と演算のデータ依存範囲を合わせるなどを行い，自動タスク並列化機能を実現した．A64FX プロセッサ上の性能評価では，既存のデータ並列実装と比較して Laplace Solver で 13%，PolyBench の fdtd-2d で 7%，ブロックコレスキー分解で 48%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．また，タスク並列実装と自動変換されたタスク並列プログラムでほぼ同等の性能となり，提案機能で簡易にタスク並列化を実施できることを示した．

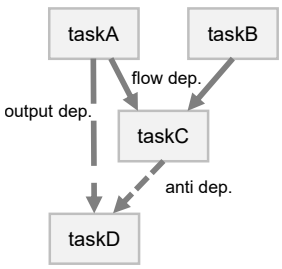
1. 序論

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ることで並列実行する，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，OpenMP や Taskflow[2] などの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．

HPC183 での報告 [3] では，タスク並列プログラミングモデルの OpenMP と OmpSs-2[4] 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 Xeon プロセッサ上で性能評価を行った．データ並列と比較してタスク並列実装では，対象としたベンチマークの Laplace Solver，N-body，ブロックコレスキー分解の全てにおいて性能向上を確認でき，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．そこで本稿では，凸多面体モデル [5][6] によるループ最適化を行うトランスパイラである Pluto[7] を基に，タスク並列プログラムに変換する自動タスク並列化機能の実装とその性能を示す．変換後のプログラムは OpenMP の `task` 指示文と `depend` 節による

[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

実装を想定し，逐次や MPI 並列プログラムを対象にタスク化とデータ依存の追加を自動で行う．提案機能を Laplace Solver，PolyBench[8] の fdtd-2d，ブロックコレスキー分解に適用し，A64FX や Xeon プロセッサ上で性能評価を行う．比較対象はデータ並列やタスク並列実装とする．データ並列実装と比較することでタスク並列実装の優位性を示し，タスク並列実装と比較することで自動変換されたタスク並列プログラムにおいても同等の性能となることを確認し，提案機能で簡易にタスク並列化を実施できることを示す．

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 Pluto を紹介し，第 4 章で Pluto に実装した自動タスク並列化機能の詳細を示す．第 5 章でその性能評価を行い，第 6 章にて結論と今後の課題を述べる．

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，Taskflow，oneAPI Threading Building Blocks（oneTBB）[9]，UPC++[10]，DASH[11]，StarPU[12] 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 がサポートするデータ依存記述のみを説明する．

2.1 OpenMP タスク並列

OpenMP では仕様 4.0 から登場した `depend` 節により，指定された変数からフロー，出力，逆依存からなるデータ依存を判断し，タスクの実行順序を制御できる．`task` 指示文でタスクとして実行する範囲をブロックとし，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，タスク内の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 ランタイムが依存関係を実行時に動的に判定し，同期や並列実行を自動で行う．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が，本稿では基本的にタスク内処理のデータ依存として記述する．`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り，`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．

```
#pragma omp task depend(iterator(iter = 0:size:tile), \
                        in:A[0:tile][iter:tile]) \
                depend(out:req)
{
  MPI_Isend(&A[0][0], size, MPI_DOUBLE, dst, tag,
            MPI_COMM_WORLD, &req);
}
```

図 2 `depend` 節と `iterator` を用いた通信タスクの実装

```
#pragma omp task depend(out:sync_task[count], req) ...
{
  MPI_Wait(&req, MPI_STATUS_IGNORE);
}
if (++count >= num_threads - 1) count = 0;
```

図 3 デッドロック回避のための同期タスクの実装

仕様 5.0 からは `iterator` が登場した．`iterator` は，`depend` 節に `iterator(`*var* = *begin:end[:step]*`)` と記述でき，*begin* から *end* の範囲で *step*（*step* が省略された場合は 1）毎を表す変数 *var* を定義し，その変数を配列インデックスに指定することで範囲内の複数の依存関係を記述できる．一方で `depend` 節における配列指定では，インデックスに *lower-bound:length* で依存範囲を記述できるが，OpenMP の仕様上，完全一致か全く異なる依存範囲のみを対象としており，部分一致の場合は期待した動作とならない．そのような依存関係を記述する場合に `iterator` が用いられる．

2.2 通信，同期のタスク化

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，プロセスを跨ぐ依存関係が別

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する．他のタスクは通信が完了するまで，送信バッファには書き込みができず，受信バッファには読み込みも書き込みもできないため，それを満たすプロセス内の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 `depend` 節に指定することで，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記述とも一致する．

通信のタスク化では，通信と演算でタスク粒度が異なる場合を考慮する必要がある．例えば，タイリングを適用したループの 1 タイルをタスク化し，そのループで用いた配列を通信する場合を考える．一般的に通信オーバヘッドは大きいため，一度の通信で複数タイルに跨った要素を対象とする場合が多く，必ずしも 1 タイルに含まれる要素毎に通信を実行しない．その場合に `iterator` によるデータ依存記述が必要となる．**図 2** に通信タスクの実装例を示す．例では，配列 *A[0][0]* から変数 *size* 分の通信を実行する（5，6 行目）．他のタスクで *A* が変数 *tile* でタイリングされた場合，データ依存範囲はタイル毎に *A[0:tile][0:tile], A[0:tile][1*tile:tile], A[0:tile][2*tile:tile], ...* となるため，*MPI_Isend* には 0 から *size* の範囲で *tile* 毎のデータ依存記述が必要となる．`iterator` を用いて変数 *iter* に *0:size:tile* と指定し（1 行目），*iter* を `depend` 節にある配列のインデックスとする（2 行目）．この指定により，*A* が 0 から *size* の範囲で *tile* 毎のデータ依存となり，演算が通信の部分一致の依存関係となる，複数タイルに跨った要素を対象とした通信のデータ依存記述となる．

タスク内で通信を実行する場合，デッドロックを回避するために同期タスクで *MPI_Test* や OpenMP の `taskyield` 指示文により，非同期的な通信完了の確認や，通信が完了していない場合に別のタスクを優先実行させる実装が必要となる．しかし，HPC183 で報告した通り，GNU や Clang/LLVM コンパイラの `taskyield` 指示文の実装は，仕様通りではあるが，実用上は課題があるため，上記の実装だけではデッドロックが起きる可能性がある．その一つの解決策として，**図 3** に示した疑似的なデータ依存の追加により，同期タスクの実行順序を制御する手法を提案した．疑似的なデータ依存で同時実行可能な同期タスク数をスレッド数-1 とすることで（1，5 行目），逐次実行でデッドロックが発生しなければ，スレッド数に関わらずデッドロックが発生しないことを示した．本稿においても同様の手法を用いて MPI+OpenMP におけるタスク並列実行を実現する．

3. Pluto

Pluto は U. Bondhugula らによって開発されたソースコードレベルでループ最適化を適用するトランスパイラである．C 言語で記述されたソースコードを対象に，最適化を適用したい箇所を`#pragma scop` と`#pragma endscop` でブロック化することで，範囲内のみを対象にソースコードを変換する．Pluto では凸多面体モデルを利用し，線形代数学的にプログラム（特にループ）を解析，モデル化し，ステートメント間のデータ依存や境界条件を求めることで，並列性の抽出やループ最適化の適用を可能とする．プログラムの解析では Static Control Part（SCoP）[6][13] 単位でループを検出する．SCoP として検出できるループは，拡張もあるが基本的には，配列インデックスやループ条件文の全てがアフィン式（ループ変数の線形結合と定数で表せる式）の *for* ループであり，SCoP 以外は最適化しない．ループ最適化は，分割，融合，交換，タイリング，スキューイング，アンローリング，ベクトル化に加えて OpenMP のデータ並列化にも対応しており，ユーザがそれらの最適化手法から選択して適用することもできる．

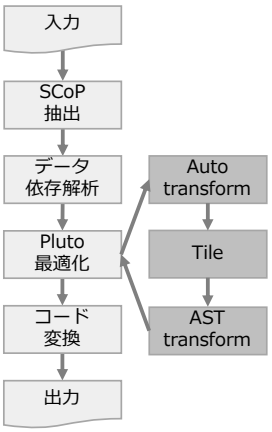


図 4 Pluto の実行フロー

Pluto は様々な OSS で構成される．**図 4** に Pluto の実行フローを示す．Clan[14]/pet[15] によってソースコードから SCoP を抽出し，isl[16]/Candl[17] でデータ依存解析，Pluto によるループ最適化を適用後に，CLooG[18] が SCoP をソースコードに変換する．本稿では Pluto がデフォルトに設定している Clan と isl を使用する．また，Pluto による最適化は Auto transform，Tile，AST transform の 3 種類で構成される．Auto transform はループ分割，融合，交換，スキューイング，Tile はループタイリング，AST transform はループベクトル化，アンローリング，OpenMP のデータ並列化をそれぞれ適用する．

4. 実装

Pluto に実装した自動タスク並列化機能の詳細を示す．対象はオリジナルの Pluto と同様に C 言語のソースコードであり，さらに MPI 記述にも対応する．Pluto による自動 MPI 並列化の研究 [19] は既に行われているが，MPI は広く普及しており，既に MPI 実装されたアプリケーション

も多いことから，本稿ではユーザが MPI 実装したソースコードを対象とする．また，既に OpenMP でデータ並列化されている場合は対象としない．しかし，OpenMP は一部を除いて指示文を無視すれば逐次実装と同等となるため，指示文を無視したソースコードとして入力時に解釈すれば良く，今後対応予定である．自動タスク並列化機能は Pluto のオプションで制御可能とし，`polycc --task a.c` でタスク並列プログラムに変換する．実装方針は以下の通りである．

- タイリングを適用できるループは，1 タイルを 1 タスクとする．
- タイリングを適用できないループは，そのループ内にある 1 ステートメントを 1 タスクとする．
- ループ以外は，1 ステートメントを 1 タスクとする．
- MPI や BLAS/LAPACK に対応し，通信や同期，線形代数演算ライブラリの関数に適切なデータ依存を設定する．

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．また，タイリングが適用できない，もしくは Pluto が適用する必要がないと判断したループの場合は，そのループ内にあるステートメントをそれぞれ 1 タスクとする．最後にループ以外のステートメントはそれぞれ 1 タスクとする．タイリング結果を基にタスク化するループやステートメントを決定するため，図 4 に示す実行フローの Pluto 最適化内にある，Tile 後の AST transform 内に実装する．

本稿で用いる OpneMP のタスク並列モデルは，タスク並列実行区間の全体を `parallel`+`single/master` 指示文でブロック化し，その中で `task` 指示文を記述する．Pluto はループ毎に `parallel for` 指示文を追加する実装しかないため，ユーザ指定区間を基に `parallel`+`master` 指示文で全体をブロック化し，ループやステートメントに `task` 指示文を追加する実装を行う．

Pluto の最適化区間内に関数呼び出しがある場合，全ての実引数を read としてデータ依存解析を行うため，write を行う関数で正しい結果が得られず，Pluto の最適化や自動タスク並列化を実施できない．本稿では，MPI と BLAS/LAPACK に対応するため，それらの関数の依存情報を予め Pluto に実装しておくことでデータ依存解析を可能とする．第 4.2 章で詳細を述べるが，MPI と BLAS/LAPACK のどちらも同様の処理で対応するため，MPI のみ説明する．

4.1 演算タスク

演算部分の自動タスク並列化に関して，変換例を用いて詳細な説明を行う．**図 5** が自動タスク並列化適用前のプログラムである．2 種類の *for* ループで構成され，1 ループ目はループ変数 i のループ内で配列 *B[i]* に i を代入し（2〜4 行目），2 ループ目で *B[i]* とその前後の *B[i+1]* と *B[i-1]* を用いて，配列 *A[i]* を更新する 1 次元ステンシル演算の一部を用いた例である（5〜7 行目）．これらのループをタイリングした場合の依存関係は，*A[i]* や *B[i]* が演算するタイルに加えて，*B[i+1], B[i-1]* によって前後のタイルも含まれる．従って，隣接領域へのアクセスがある場合は，隣接するタイルも自動的にデータ依存として `depend` 節に追加する必要がある．

```
1 #pragma scop
2 for (i = 0; i < N; i++) {
3   B[i] = i;
4 }
5 for (i = 1; i < N - 1; i++) {
6   A[i] = (B[i + 1] + B[i] + B[i - 1]) / 3.0;
7 }
8 #pragma endscop
```

図 5 自動タスク並列化適用前のプログラム（演算タスク）

```
1  int t2, t3;
2  #pragma omp parallel default(shared) private(t2, t3)
3  #pragma omp master
4  {
5  if (N >= 1) {
6    for (t2 = 0; t2 <= floord(N-1, 32); t2++) {
7  #pragma omp task depend(out:B[32*t2:32]) \
8                firstprivate(t2) private(t3)
9      for (t3 = 32*t2; t3 <= min(N-1, 32*t2+31); t3++) {
10       B[t3] = t3;
11     }
12   }
13   if (N >= 3) {
14     for (t2 = 0; t2 <= floord(N-2, 32); t2++) {
15       int t2_prev = (t2 == 0) ? 0 : 1;
16       int t2_next = (t2 == floord(N-1, 32)) ? 0 : 1;
17 #pragma omp task depend(in:B[32*t2:32], \
18                              B[32*(t2+t2_next):32], \
19                              B[32*(t2-t2_prev):32]) \
20                  depend(out:A[32*t2:32]) \
21              firstprivate(t2) private(t3)
22       for (t3 = max(1, 32*t2); t3 <= min(N-2, 32*t2+31);
              t3++) {
23         A[t3] = (B[t3 + 1] + B[t3] + B[t3 - 1]) / 3.0;
24       }
25     }
26   }
27 }
28 }
```

図 6 自動タスク並列化適用後のプログラム（演算タスク）

図 6 が自動タスク並列化適用後のプログラムである．2 種類のループ共にタイリングが適用可能なループのため（6，9，14，22 行目），タイルを演算するループに `task` 指示文を追加する（7，8，17〜21 行目）．Pluto のデータ依存解析により，1 ループ目で B が write，2 ループ目で A が write，B が read という情報は保持しているため，`depend` 節にそれぞれ `out`，`in` でデータ依存を指定できる（7，17〜

```
1 #pragma scop
2 /* ... */
3 MPI_Isend(&A[0], size, MPI_INT, dst, 0, MPI_COMM_WORLD,
4           &req);
5 MPI_Wait(&req, MPI_STATUS_IGNORE);
6 /* ... */
7 #pragma endscop
```

図 7 自動タスク並列化適用前のプログラム（通信，同期タスク）

```
1  int t2, t3;
2  int _num_threads, _count = 0, *_sync_task;
3  #pragma omp parallel default(shared) private(t2, t3)
4  #pragma omp master
5  {
6  /* ... */
7  _num_threads = omp_get_num_threads();
8  _sync_task = (int *)malloc(sizeof(int) * _num_threads);
9  #pragma omp task depend(iterator(iter0 = 0:size:32), \
10                          in:A[iter0:32]) \
11                  depend(out:req)
12 MPI_Isend(&A[0], size, MPI_INT, dst, 0, MPI_COMM_WORLD,
13          &req);
14 #pragma omp task depend(iterator(iter0 = 0:size:32), \
15                          in:A[iter0:32]) \
16                  depend(out:_sync_task[_count], req)
17 MPI_Wait(&req, MPI_STATUS_IGNORE);
18 if (++_count >= _num_threads - 1) _count = 0;
19 /* ... */
20 }
21 free(_sync_task);
```

図 8 自動タスク並列化適用後のプログラム（通信，同期タスク）

20 行目）．ただし，2 ループ目の B は隣接領域へのアクセスも含むため，配列インデックスのループ変数と加減算する定数を基に隣接するタイルもデータ依存として追加する（18，19 行目）．タイリング適用時の配列の端のタイルは隣接するタイルが存在しないため，タスク外で境界条件を求めて（15，16 行目），その結果をデータ依存として与える変数に反映する（18，19 行目）．

4.2 通信，同期タスク

MPI で引数に対して write を行う関数を説明し，その関数呼び出しが Pluto の最適化区間内にある場合の対応を示す．*MPI_Isend* は *MPI_Request* に対する write があり，*MPI_Irecv* は通信バッファと *MPI_Request* に対する write がある．また，*MPI_Wait* は *MPI_Request* と *MPI_Status* のみを引数に持つが，通信完了を保証する処理のため，*MPI_Isend/Irecv* で扱うバッファに対する read/write が行われたことを保証する．そのため *MPI_Wait* には，完了を保証する *MPI_Isend/Irecv* のバッファに対する read/write も必要となる．本稿では，MPI 関数毎にどの引数が read-/write であるかの情報を Pluto に実装することで，データ依存解析を可能とした．例えば *MPI_Irecv* の場合は，通信バッファと *MPI_Reqeust* を write とすることでデータ依存解析が可能となり，その情報を基に `depend` 節にデータ依存を追加できる．

通信，同期部分の自動タスク並列化に関して，変換例を用いて詳細な説明を行う．**図 7** が自動タスク並列化適用前のプログラムである．配列 A と変数 *size*, *dst* を用いて，*MPI_Isend* で *A[0]* から *size* 分の要素を *dst* プロセスへとノンブロッキングで送信し（3，4 行目），*MPI_Wait* による同期で通信完了を保証する MPI 実装の例である（5 行目）．また，プログラムの省略箇所（2, 6 行目）に，例えば，図 5 のようなタイリング適用可能なループがあり，そのループ内で通信バッファ A が演算されることを想定する．

図 8 が自動タスク並列化適用後のプログラムである．省略箇所で通信バッファ A に対して read や write を行うループが，サイズ 32 でタイリングされた場合，通信もサイズ 32 毎の依存関係とする必要がある．従って，`iterator` を用いて，*begin* に通信開始の要素 0，*end* に通信サイズ *size*，*step* をタイルサイズの 32 とすることで，演算と通信の多対 1 のデータ依存を指定できる（9，10 行目）．*MPI_Request* は write であるため，データ依存解析の結果を基に `depend` 節に追加する（11 行目）．*MPI_Wait* は，対になる *MPI_Isend/Irecv* と同じ通信バッファをデータ依存として追加する必要がある．*MPI_Request* を基に対象の通信を探索し，通信バッファと *MPI_Request* をデータ依存として追加する（14〜16 行目）．制約として，通信と同期で用いる *MPI_Request* がソースコードレベルで一致している必要がある．最後に，HPC183 で報告したデッドロックを回避するための疑似的なデータ依存を追加する（2，7，8，16，18，21 行目）．今回は省略箇所で通信バッファが用いられたタイリング可能なループがある例だが，通信バッファが無い場合は，通信開始の要素から通信サイズ分の依存関係があるとするのみで `iterator` は使わない．現状では，*MPI_Isend/Irecv*，*Wait* に対応済みであり，今後は他の MPI 関数の対応も進める予定である．

表 1 実験環境（A64FX）

CPU	A64FX 2.0GHz
Number of cores	48
Memory	HBM2 32GB
Compiler	Clang/LLVM 14.0.6
MPI	OpenMPI 4.1.2

5. 評価

Pluto に実装した自動タスク並列化機能の性能評価を行う．評価用のベンチマークを Laplace Solver，PolyBench の fdtd-2d，ブロックコレスキー分解とする．実験環境として A64FX プ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

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	Clang/LLVM 14.0.6
MPI	OpenMPI 4.1.2

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 はオフとした．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 Clang/LLVM 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* を共通とし，A64FX は*-march=armv8.2-a+sve -mcpu=a64fx*，SKL は*-mtune=skylake* とした．タスク並列化自体の有用性を示すため，Pluto のタイリング以外の最適化は全てオフとしたが，今後は様々なループ最適化とタスク並列化を組み合わせた場合の対応や性能評価も行う予定である．

性能評価では各実験環境を 1 ノードのみ用いる．ブロッククコレスキー分解は OpenMP 実装だが，Laplace Solver と fdtd-2d は MPI+OpenMP 実装であるため，プロセス数は NUMA ノードを考慮し，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．スレッド数は A64FX で 4 ～ 48，SKL は 4 ～ 40 と変化させる．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を示す．グラフの凡例は，OMP For がデータ並列，OMP Task がタスク並列，Pluto + Task が提案機能により自動変換されたタスク並列プログラムである．

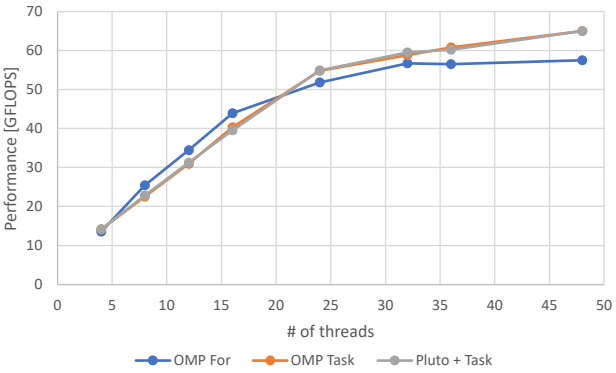


図 9　Laplace Solver の性能評価（A64FX）

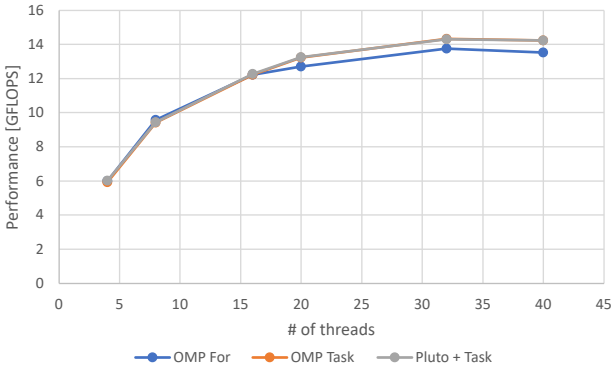


図 10　Laplace Solver の性能評価（SKL）

5.1　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[20] が提供する逐次実装を基に 2 次元ブロック分割で MPI 実装し，さらに OpenMP データ並列とタスク並列実装を行った．**図 9** に A64FX，**図 10** 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 64×64 ～ 8192×8192 の範囲で次元毎に 2 冪で変化させた．OMP For と比較して OMP Task は，A64FX で 24，SKL で 16 スレッド以上で性能が高く，最大コア数使用時に A64FX で 13%，SKL で 5%性能が向上した．Laplace Solver は，典型的なステンシル演算でデータ並列実行時の演算のばらつきが少ないため，タスク並列による性能向上率は低いが，全体同期からタスク間同期による同期オーバヘッドの削減や，データ依存による通信と演算のオーバラップが性能向上に繋がったと考える．また，OMP Task と Pluto + Task を比較すると，全てのスレッド数においてほぼ同等の性能が得られたことから，自動タスク並列化機能により，隣接領域へのアクセスがある場合のデータ依存の追加や MPI 関数のタスク化ができ，タスク並列実装と同等の変換ができたと言える．

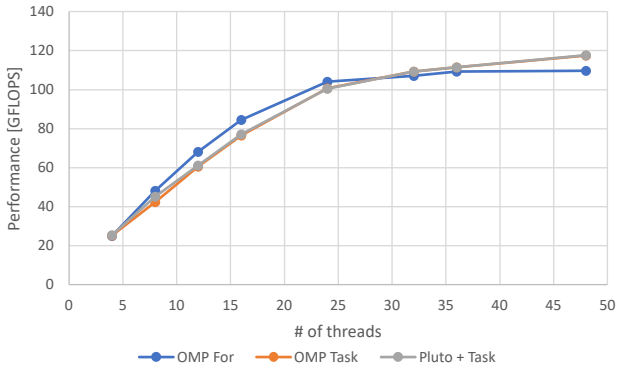


図 11　fdtd-2d の性能評価（A64FX）

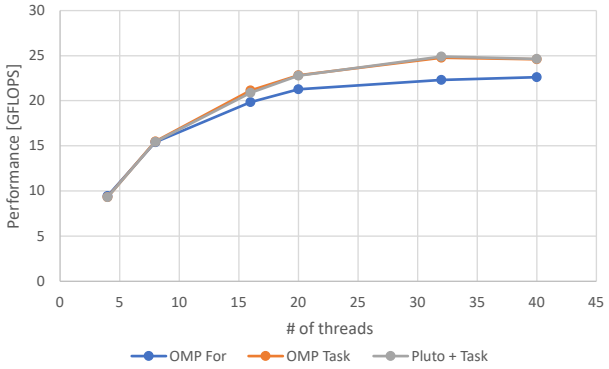


図 12　fdtd-2d の性能評価（SKL）

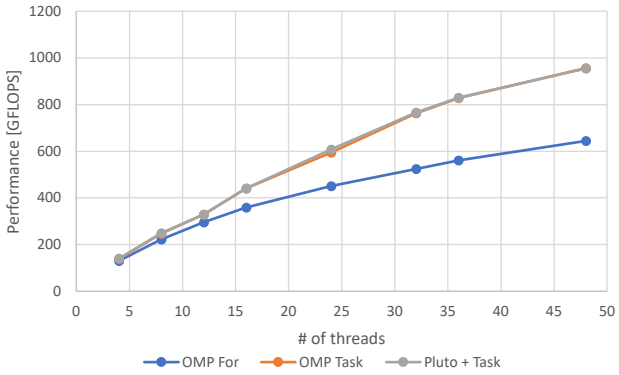


図 13　ブロックコレスキー分解の性能評価（A64FX）

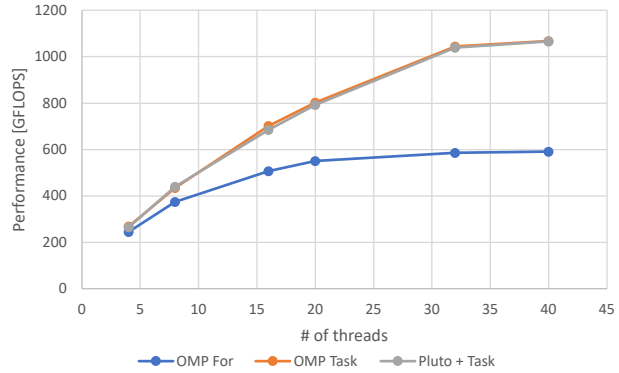


図 14　ブロックコレスキー分解の性能評価（SKL）

5.2　fdtd-2d

fdtd-2d は，PolyBench に含まれるベンチマークの一つで，電磁界シミュレーションの主要な数値計算手法である FDTD 法を行う．Laplace Solver と同じくステンシル演算であり，メモリ律速なベンチマークとなる．PolyBench は全て逐次実装であるため，1 次元ブロック分割で MPI 実装し，さらに OpenMP データ並列とタスク並列実装を行った．**図 11** に A64FX，**図 12** に SKL で実行した fdtd-2d 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 16×16 ～ 8192×8192 の範囲で次元毎に 2 冪で変化させた．OMP For と比較して OMP Task は，A64FX で 32，SKL で 8 スレッド以上で性能が高く，最大コア数使用時に A64FX で 7%，SKL で 9%性能が向上した．fdtd-2d は，ループ数や演算格子点数が Laplace Solver と異なるが同じく典型的なステンシル演算のため，Laplace Solver と同様の理由で性能が向上したと考える．また，OMP Task と Pluto + Task を比較すると，全てのスレッド数においてほぼ同等の性能が得られたことから，自動タスク並列化機能により，タスク並列実装と同等の変換ができたと言える．

5.3　ブロックコレスキー分解

ブロックコレスキー分解は，正定値対称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り，OmpSs-2 が提供する実装を用いる．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:対称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．OmpSs-2 が提供するタスク並列実装を基に，OpenMP データ並列とタスク並列実装を行った．また，既にタイル（ブロック）化されたベンチマークのため，Pluto によるループタイリングはオフ（`--notile`）として自動タスク並列化を適用した．**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．問題サイズを 8192×8192 とし，タイルサイズを 64×64 ～ 8192×8192 の範囲で 2 冪で変化させた．OMP For と比較して OMP Task は，A64FX，SKL 共に 4 スレッド以上で性能が高く，最大コア数使用時に A64FX で 48%，SKL で 80%性能が向上した．Laplace Solver や fdtd-2d と異なり，ブロックコレスキー分解は処理毎の依存関係が多いため，データ並列実行では全体同期で多くのスレッドが同期待ち状態となる．一方で，タスク並列実行ではデータ依存によるタスク間同期としたことでスレッドを有効に利用でき，性能に大きな差がでたと考える．また，OMP Task と Pluto + Task を比較すると，全てのスレッド数においてほぼ同等の性能が得られたことから，自動タスク並列化機能により，BLAS/LAPACK 関数

のタスク化ができ，タスク並列実装と同等の変換ができたと言える．

6. 結論

本稿では，凸多面体モデルによるループ最適化を行うトランスパイラである Pluto を基に，タスク並列プログラムへの自動変換を行う自動タスク並列化機能を実装し，A64FX と Xeon（SKL）プロセッサ上で性能評価を行った．タイリングの適用の可否でタスク粒度を決定する方針とし，以下の実装で自動タスク並列化機能を実現した．

- Pluto のデータ依存解析の結果を基にタスクのデータ依存を `depend` 節に追加する．
- タイリング可能なループで隣接領域へのアクセスを解析し，隣接するタイルをデータ依存として追加する．
- MPI や BLAS/LAPACK 関数の引数情報を予め Pluto に持たせることでデータ依存解析を可能とする．
- `iterator` を用いて，タイリングされたループ内の演算と通信のデータ依存範囲を合わせる．
- デッドロック回避のための疑似的なデータ依存を同期タスクに追加する．

性能評価は，データ並列とタスク並列実装，提案機能により自動変換されたタスク並列プログラムの比較とし，対象のベンチマークを Laplace Solver，PolyBench の fdtd-2d，ブロックコレスキー分解とした．A64FX と SKL 共にデータ並列と比較してタスク並列実装の性能が高く，Laplace Solver は A64FX で 13%，SKL で 5%，fdtd-2d は A64FX で 7%，SKL で 9%，ブロックコレスキー分解は A64FX で 48%，SKL で 80%の性能向上を示し，タスク並列プログラミングによる実装の性能の高さを示した．また，タスク並列実装と自動変換されたタスク並列プログラムの性能はほぼ同等であったため，提案機能を用いることで逐次や MPI 並列プログラムから簡易にタスク並列化を実施できることを示した．

今後の課題として，様々なプログラムに対して本稿の提案機能を適用し，タスク並列実装の性能評価や適用できるプログラムの種類を増やすことが挙げられる．また，変換後のプログラムは OpenMP のみを対象としたため，他のデータ依存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対象とす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] 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.
[3] 津金 佳祐, 前田 宗則, 新井 正樹, 吉川 隆英, “A64FX におけるタスク並列ベンチマークの性能評価”, 研究報告ハイパフォーマンスコンピューティング (HPC), 2022-HPC-183(20), pp. 1–8.
[4] 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.
[5] P. Feautrier, “Some efficient solutions to the affine scheduling problem: I. One-dimensional time”, International Journal of Parallel Programming, Vol. 21, Issue 5, pp.313–347, October 1992.
[6] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, O. Temam, “Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies”, International Journal of Parallel Programming, Vol. 34, Issue 3, pp.261–317, June 2006.
[7] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, “Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model”, International Conference on Compiler Construction (ETAPS CC), Apr 2008, Budapest, Hungary.
[8] PolyBench/C the Polyhedral Benchmark suite, http://web.cse.ohio-state.edu/ pouchet.2/software/polybench
[9] oneAPI Specification 1.1-rev-1 documentation/oneTBB, https://spec.oneapi.io/versions/latest/elements/oneTBB/source/nested-index.html, 2021
[10] 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.
[11] 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.
[12] 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.
[13] P. Feautrier, “Dataflow analysis of array and scalar references”, International Journal of Parallel Programming, Vol. 20, Issue 1, pp.23–53, February 1991.
[14] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, O. Temam, “Clan A Polyhedral Representation Extractor for High Level Programs”, 2008.
[15] S. Verdoolaege, T. Grosser, “Polyhedral Extraction Tool”, Second International Workshop on Polyhedral Compilation Techniques (IMPACT’12), 2012.
[16] S. Verdoolaege, “isl: An Integer Set Library for the Polyhedral Model”, International Congress on Mathematical Software (ICMS 2010), pp. 299–302.
[17] Candl, https://github.com/periscop/candl
[18] C. Bastoul. “Code Generation in the Polyhedral Model Is Easier Than You Think”, PACT’13 IEEE International Conference on Parallel Architecture and Compilation Techniques, pp. 7–16, 2004.
[19] U. Bondhugula, “Compiling affine loop nests for distributed-memory parallel architectures”, SC ’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–12.
[20] Omni Compiler, https://github.com/omni-compiler/omni-compiler