XcalabeMP2.0のタスク並列機能(2)
― グローバルビューに基づく分散タスク生成

村井 均 (理研R-CCS)

はじめに ― XMPにおけるタスク並列処理

- OpenMPにおけるタスク記述の利点:
  - 個々のタスクにおけるデータの定義・参照の情報のみから、タスク間の依存関係(実行順序)を自動的に構築できる。
  - 「正しい実行順序」≡「逐次実行における順序」＝「一つのスレッドがタスクを生成する順序」
- SPMDに基づく分散並列実行で、複数のノードが(非同期的に)タスクを生成するなら、「正しい実行順序」を知ることができない。
  - OpenMPの手法を、XMPにそのまま適用することはできない。

➡ 高生産性と高性能を両立する分散タスク生成方式とは？

OpenMP3.0のタスク機能 (task指示文)

```
int fib(int n){
 if (n < 2) return n;
 else {
  #pragma omp task ...
  i = fib(n-1);
  #pragma omp task ...
  j = fib(n-2);
  #pragma omp taskwait
  return i + j;
 }
}
```

```
int main(){
 #pragma omp parallel ...
 #pragma omp single
 fib(n);
}
```

- task指示文では、タスクが生成されるが、実行はされない (cf. スケルトン実行)
- 各タスクを生成するのは、チーム内の一つのスレッド。
- 生成されたタスクは、チーム内のスレッドによって、特定のタイミング(e.g. parallel リージョンの出口や taskwait)で実行される。

OpenMP4.0のタスク機能 (depend節)

```
#pragma omp task depend(in:...) depend(out:...)
  ...
```

- depend節によって、当該タスクにおけるデータの定義・参照を宣言する。
- タスクの生成時に、depend節の情報を元に、逐次実行における定義・参照の順序が保存されるように、タスク間の依存関係が設定される。
- タスクの実行時に、依存関係が満たされたタスクから順に実行される。

従来手法 (ローカルビューに基づくタスク記述)

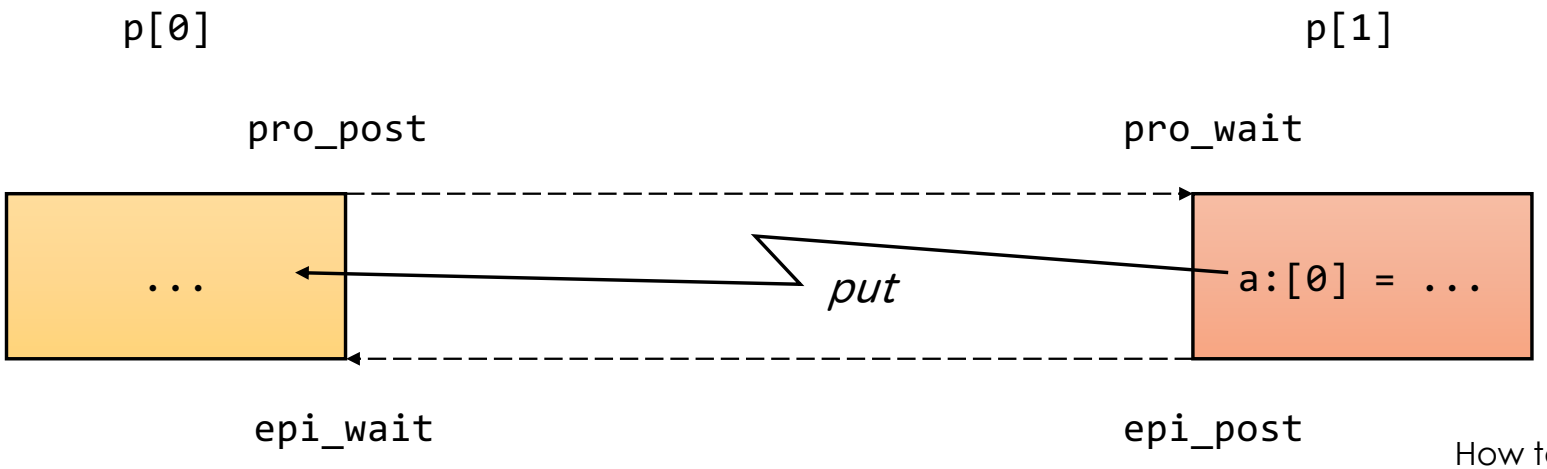
- How to specify dependency?
  - Within a node: `in/out/inout` clauses (cf. OpenMP's `depend` clause)
  - Between nodes:
    - Clauses for one-sided sync. (cf. `post/wait` or `notify/query`) for the notification of ready-for-comm. and ack.
    - The interactions can be specified with any one-sided comms. (e.g. coarray, `MPI_Put/Get`, XMP's `gmove in/out`)

Sync. between Tasklets

NOTE: `pro_wait` and `epi_wait` may influence tasklet scheduling.

```
#pragma xmp tasklet pro_post(p(2), tag0) ¥
                    epi_wait(p(2), tag1) ¥
                    on p[0]
{
  ...
}
```

```
#pragma xmp tasklet pro_wait(p(1), tag0) ¥
                    epi_post(p(1), tag1) ¥
                    on p[1]
{
  a:[0] = ...;
}
```



How to implement TAGs is an open issue.

```
for (int k = 0; k < nt; k++) {

  potrf(A[k][k]);



  B[:] = A[k][k][:];

  for (int i = k + 1; i < nt; i++) {
    trsm(B, A[k][i]);
  }

  for (int i = k + 1; i < nt; i++) {



    C[i][:] = A[k][i][:];

    for (int j = k + 1; j < i; j++) {
      gemm(A[k][i], C[j], A[j][i]);
    }

    syrk(A[k][i], A[i][i]);

  }
}
```

```
#pragma xmp tasklets
for (int k = 0; k < nt; k++) {

#pragma xmp tasklet out(A[k][k]) on T(k)
  potrf(A[k][k]);

#pragma xmp tasklet accept_rin(A[k][k], T(k+1:), k) on T(k)

#pragma xmp tasklet out(B) rin(A[k][k], T(k), k) on T(k+1:)
#pragma xmp gmove in
  B[:] = A[k][k][:];

  for (int i = k + 1; i < nt; i++) {
#pragma xmp tasklet in(B) out(A[k][i]) on T(i)
    trsm(B, A[k][i]);
  }

  for (int i = k + 1; i < nt; i++) {

#pragma xmp tasklet accept_rin(A[k][i], T(k+1:i-1), ??) on T(i)

#pragma xmp tasklet out(C[i]) rin(A[k][i], T(i), ??) on T(k+1:i-1)
#pragma xmp gmove in
    C[i][:] = A[k][i][:];

    for (int j = k + 1; j < i; j++) {
#pragma xmp tasklet in(A[k][i], C[j]) out(A[j][i]) on T(i)
      gemm(A[k][i], C[j], A[j][i]);
    }

#pragma xmp tasklet in(A[k][i]) out(A[i][i]) on T(i)
    syrk(A[k][i], A[i][i]);

  }
}
```

従来手法の課題

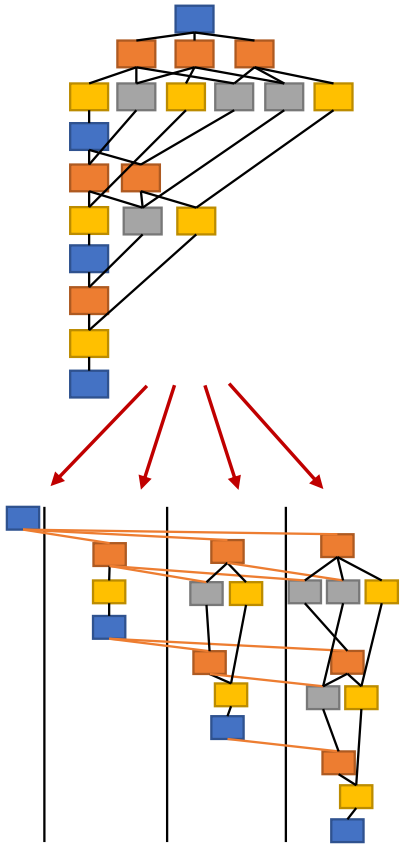
- タスク間の片側同期のためのプリミティブ
  - 必要十分な能力を持つが、プログラミングは極めて煩雑。
  - タスク間の依存関係を把握しなければならない。
- 通信(タスク間のインタラクション)を明示しなければならない。

➡ もっと簡便な記述方法が欲しい。

提案手法 (グローバルビューに基づくタスク記述)

一つのノード(マスタノード)がタスクグラフを生成し、他ノードにタスクレットを分配する。

- タスクレットの実行順序(依存関係)は、逐次実行における順序とPGAS上データの定義・参照情報に基づいて、マスタノードが決定。
- マスタノードが、指定されたノード上にタスクレットを「生成」 (cf. spawn or RPC)
- ノードをまたぐデータ依存は、データ転送として実現。



基本設計

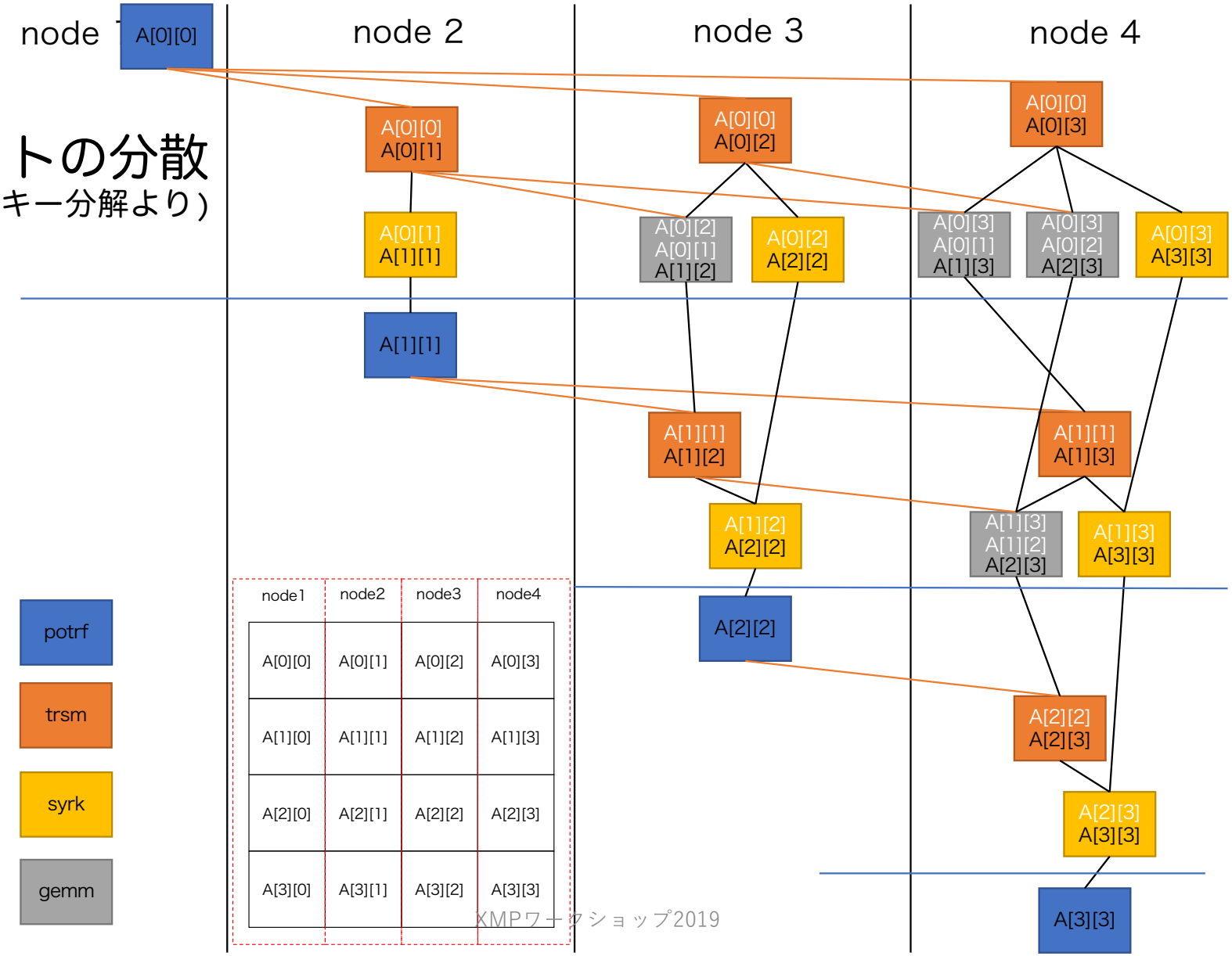
- マスタノードがタスクグラフを構築し、タスクレット間のフロー依存と出力依存を検出
  - フロー依存はデータ転送として実現され、受信側は常にバッファに受け取るので、逆依存は起きない。
  - グローバルデータだけ扱う。
- マスタノードは、各ノードにタスクレットとデータ転送(のスケジュール)を分配する。
- 各ノード(の各スレッド)は、受け取ったスケジュールに基づき、タスクレットを生成、実行する。

XMPにおける実現 — tasklet構文

```
#pragma xmp tasklet in(...) out(...) inout(...)
  ...
```

- タスクレットを宣言する。
  - in/out/inout節 (cf. OpenMPのdepend節) には、タスクレット内で定義または参照されるグローバルデータを指定する。
  - [実行時] out節で指定されたデータを保持するノードに割り当てられる (cf. owner-computes rule)。
  - [実行時] in/out/inout節の情報に基づき、タスクレット間の依存関係(データ転送)が設定される。
    - in節で指定されたデータを受信し、out節で指定されたデータを送信する。
    - 受信したデータは自動的にバッファにコピーされるため、ユーザによるコード書き換えは不要 (cf. copyin)

タスクレットの分散
(ブロックコレスキー分解より)



tasklets / taskletの例 (ブロックコレスキー分解より)

```
// Aを分散



#pragma xmp tasklets
{
for (int k = 0; k < nt; k++){
 #pragma xmp tasklet inout(A[k][k][:])
 potrf(...);
 for (int i = k + 1; i < nt; i++){
  #pragma xmp tasklet in(A[k][k][:]) inout(A[k][i][:])
  trsm(...);
  for (int i = k + 1; i < nt; i++){
   for (int j = k + 1; j < i; j++){
    #pragma xmp tasklet in(A[k][i][:], A[k][j][:]) inout(A[j][i][:])
    gemm(...);
   }
  }
  #pragma xmp tasklet in(A[k][i][:]) inout(A[i][i][:])
  syrk(...);
 }
}
```

基本的に、OpenMPと同程度の記述

実装方式 (1) ― コンパイル時の処理

- taskletsが現れる手続きの冒頭に、xmp_regist_taskletの呼び出しを挿入。
- taskletの位置に、xmp_create_taskletの呼び出しを挿入。
- taskletsの末尾に，xmp_distribute_taskletとxmp_tasklet_wait_allの呼び出しを挿入。
- taskletで指定されたブロックを、新たな関数として切り出す。

実装方式 (2) — 実行時の処理

- 主に次の4つのランタイムによる。
  - `xmp_regist_tasklet` – タスクレット本体のコードを登録する。
  - `xmp_create_tasklet` – 依存関係を解析し、タスクグラフを構築する。
  - `xmp_distribute_tasklet` – タスクレットを各ノードに配布する。
  - `xmp_tasklet_wait_all` – 配布されたタスクレットを実行する。

tasklets / taskletの変形 (ブロックコレスキー分解より)

```
// Aを分散



#pragma xmp tasklets on t[0]
{
for (int k = 0; k < nt; k++){
 #pragma xmp tasklet inout(A[k][k][:])
 potrf(...);
 for (int i = k + 1; i < nt; i++){
  #pragma xmp tasklet in(A[k][k][:]) inout(A[k][i][:])
  trsm(...);
  for (int i = k + 1; i < nt; i++){
   for (int j = k + 1; j < i; j++){
    #pragma xmp tasklet in(A[k][i][:], A[k][j][:]) inout(A[j][i][:])
    gemm(...);
   }
  }
  #pragma xmp tasklet in(A[k][i][:]) inout(A[i][i][:])
  syrk(...);
 }
}
```

```
int tid0 = xmp_regist_tasklet(
                tasklet_potrf, "potrf");
...

if (me == 0){

 for (int k = 0; k < nt; k++){
  xmp_create_tasklet(tid0, ...);

  for (int i = k + 1; i < nt; i++){
   xmp_create_tasklet(tid1, ...);

   for (int i = k + 1; i < nt; i++){
    for (int j = k + 1; j < i; j++){
     xmp_create_tasklet(tid2, ...);

    }
   }
   xmp_create_tasklet(tid3, ...);

  }
 }

}

xmp_distribute_tasklet();
xmp_tasklet_wait_all();
xmp_barrier();
```

```
void tasklet_potrf(...){
 potrf(...);
}
```

```
void tasklet_trsm(...){
 trsm(...);
}
```

```
void tasklet_gemm(...){
 gemm(...);
}
```

```
void tasklet_syrk(...){
 syrk(...);
}
```

実装方式 (3) ― タスクレット・スケジューラ

- ノード内のタスクレットの実行とデータの送受信を制御。
- 依存関係が満たされた = 必要なデータの受信が完了したタスクレットの実行を開始する。
- ANLが開発しているユーザレベルスレッドライブラリであるArgobotsを用いて実装。

予備評価 (1) — 評価環境

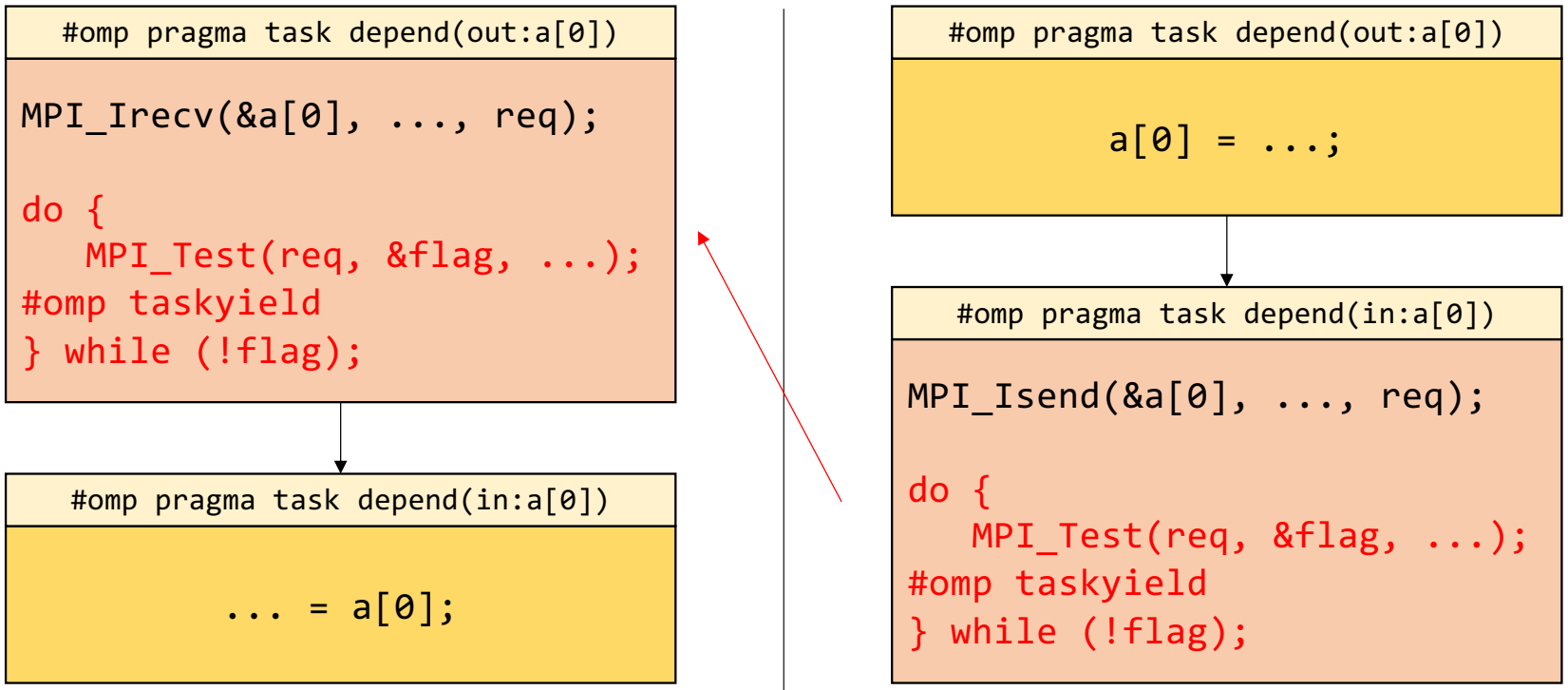
- ブロックコレスキー分解
  - 配列サイズ: 16384x16384
  - ブロックサイズ: 512x512
- ハイブリッド並列
  - 1XMPノード/計算ノード
  - 1-24スレッド/XMPノード

- Cygnus @ 筑波大CCS
  - CPU: Intel Xeon Gold 6126 2.6GHz (12 core) x 2
  - メモリ: 192 GiB, 255.9 GB/s
  - ネットワーク: InfiniBand HDR100 x 4
- ソフトウェア環境
  - Omni XMP 1.3.2
  - GCC 4.8.5
  - OpenMPI 4.0.0
  - netlib-lapack 3.8.0
  - cblas 2015-06-06
  - Argobots 1.0b1
    - 簡単なFIFOスケジューラを使用
    - スタックサイズを32KBに拡張

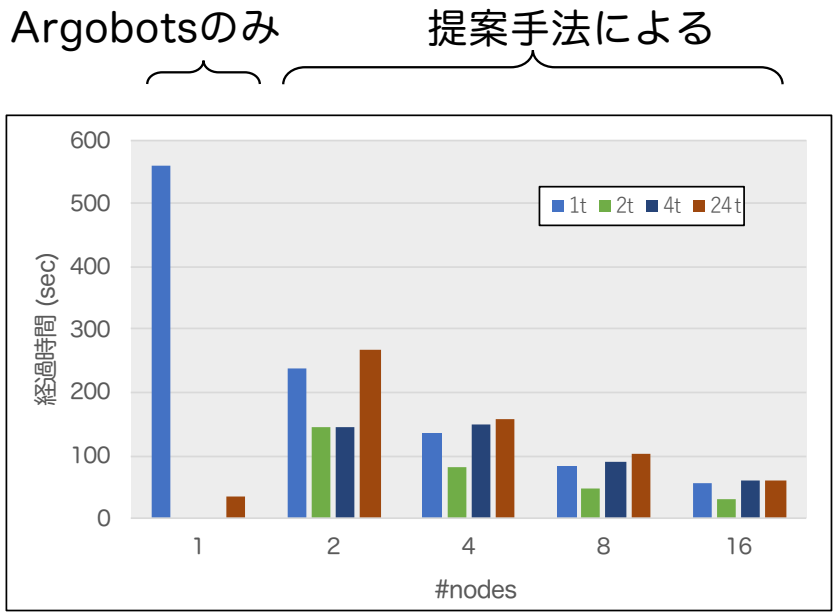
予備評価 (2) ― プロトタイプ実装

- 処理系による変換を想定して、コードを手変換。
- 各ランタイムのプロトタイプを実装。
- タスクレット・スケジューラは実装せず。
  - タスクレットの入口: `Irecv` → `Wait` & `yield`
  - タスクレットの出口: `Isend` → `Wait` & `yield`
- ノード内のデータ依存関係も、データ転送として実現。

プロトタイプ実装におけるノード間依存



予備評価 (3) — 評価結果



- 1スレッドで、16ノードまで速度向上。
- ノード内では、4スレッド以上で速度向上せず。
  - 各タスクレットが通信完了までyieldを繰り返すオーバヘッド
  - MPI_THREAD_MULTIPLEによる性能低下
    → タスクレット・スケジューラが必要
  - send/recvのオーバヘッド
    → より高速(低レベル)な通信方式が必要

おわりに

- グローバルビューに基づく分散タスク生成方式を提案
  - ある一つのノードがタスク間の依存関係を解析し、他ノードへ分配する。
  - タスク間のデータ依存関係は、データ転送として実現する。
- 提案方式のプロトタイプを実装し、ブロックコレスキー分解を用いた予備評価によって、有効性を確認した (マルチスレッド性能には課題あり)。
- 今後の課題
  - タスクレット・スケジューラの実装
  - タスク間通信の高速化