第3回 XcalableMP プログラミングコンテスト 成果報告

原 健太朗（Google）

話の流れ

- 行列積の最適化

- XMPへの期待

- 高生産並列言語への期待

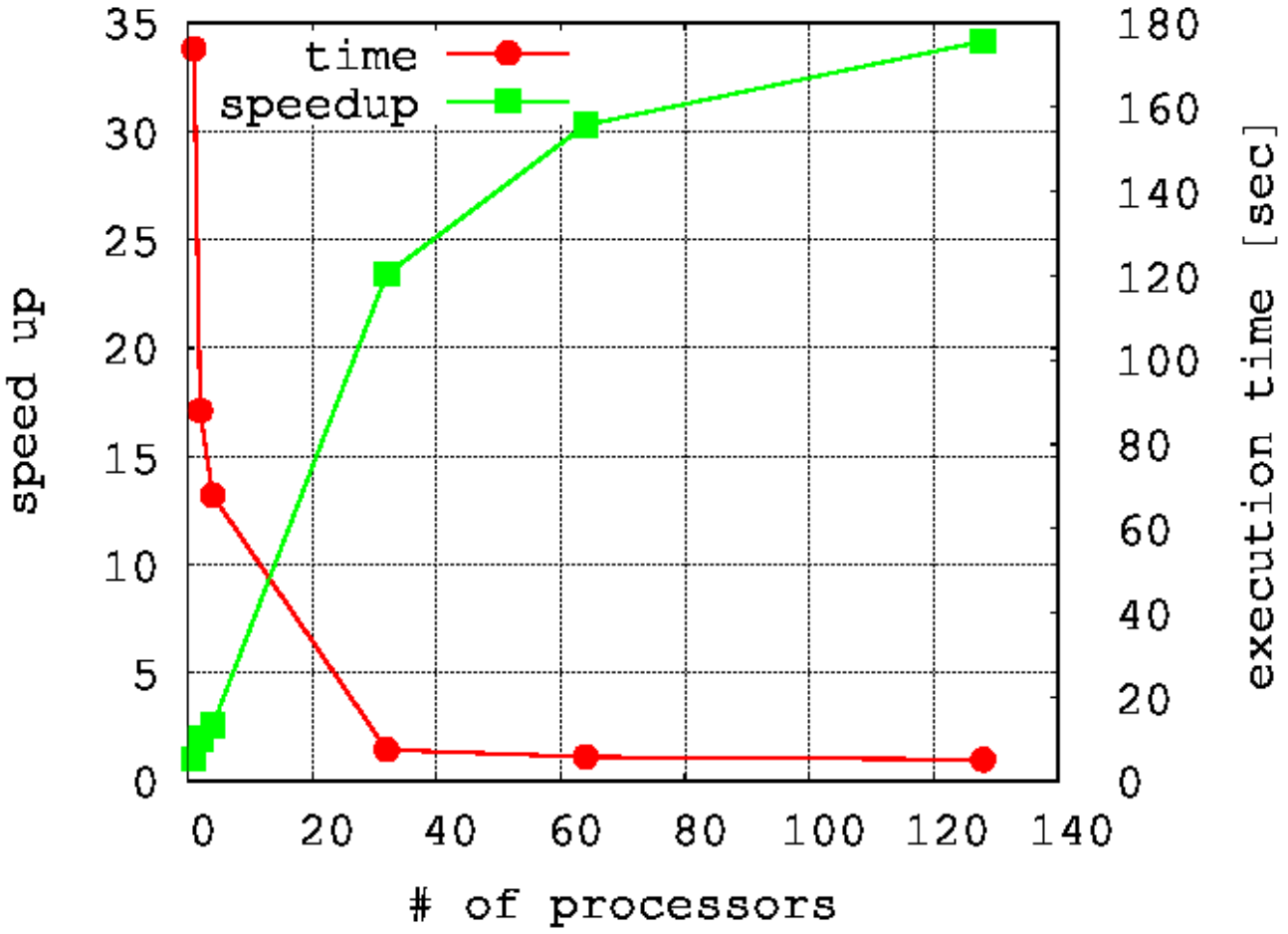
行列積の最適化

規定課題

- 密行列積をXMPで並列化する
  - C = A × B
  - 行列要素はdouble型

- 実行環境
  - Opteron 2214 × 4コア × 32ノード
  - メモリ4GB
  - 1Gbit Ethernet

結果（行列サイズ 4096×4096）



逐次最適化前　：10009 sec

逐次最適化後　：173 sec

128プロセッサで並列化後：5.09 sec

最適化の流れ

Step 1: 逐次プログラムの最適化

Step 2: ノード内並列化

Step 3: ノード間並列化

Step 1: 逐次プログラムの最適化

- とても大事

- 安易な並列化は禁物

最初のプログラム

```
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    for(k=0; k<N; k++){
      C[i][j]+=A[i][k]*B[k][j];
}}}
```

最適化1:ループ交換

```
for(i=0; i<N; i++){
  for(k=0; k<N; k++){
    for(j=0; j<N; j++){
      C[i][j]+=A[i][k]*B[k][j];
}}}
```

最適化2:キャッシュブロッキング

```
for(ii=0; ii<N; ii+=block){
  for(kk=0; kk<N; kk+=block){
    for(i=ii; i<ii+block; i++){
      for(k=kk; k<kk+block; k++){
        for(j=0; j<N; j++){
          C[i][j]+=A[i][k]*B[k][j];
}}}}}
```

- jループをブロッキングしないのは、最内ループを長くしてプリフェッチを効かせたほうが有利だったため

最適化3:ループ展開

```
for{for{for{for{
  double atmp=A[i][k];
  for(j=0; j<N; j+=8){
    C[i][j]  +=atmp*B[k][j];
    C[i][j+1]+=atmp*B[k][j+1];
    ...;
}}}}}
```

最適化4:配列 => ポインタ

```
for{for{for{for{
  for(j=0; j<N; j+=8){
    *(cp)  +=atmp*(*(bp));
    *(cp+1)+=atmp*(*(bp+1));
    ...;
    bp+=8, cp+=8;
}}}}}
```

最適化5:SIMD化

```
for{for{for{for{
  asm("movsd (%0),%%xmm9"::"r"(ap));
  asm("movddup %xmm9,%xmm0");
  for(j=0; j<N; j+=8){
    asm("movapd (%0),%%xmm1"::"r"(bp));   asm("movapd (%0),%%xmm2"::"r"(bp+2));
    asm("movapd (%0),%%xmm3"::"r"(bp+4)); asm("movapd (%0),%%xmm4"::"r"(bp+6));
    asm("mulpd %xmm0,%xmm1");             asm("mulpd %xmm0,%xmm2");
    asm("mulpd %xmm0,%xmm3");             asm("mulpd %xmm0,%xmm4");
    asm("movapd (%0),%%xmm5"::"r"(cp));   asm("movapd (%0),%%xmm6"::"r"(cp+2));
    asm("movapd (%0),%%xmm7"::"r"(cp+4)); asm("movapd (%0),%%xmm8"::"r"(cp+6));
    asm("mulpd %xmm1,%xmm5");             asm("mulpd %xmm2,%xmm6");
    asm("mulpd %xmm3,%xmm7");             asm("mulpd %xmm4,%xmm8");
    asm("movapd %%xmm5,(%0)"::"r"(cp));   asm("movapd %%xmm6,(%0)"::"r"(cp+2));
    asm("movapd %%xmm7,(%0)"::"r"(cp+4)); asm("movapd %%xmm8,(%0)"::"r"(cp+6));
    bp+=8, cp+=8;
} ap++; }}}}
```

最適化6：ソフトウェアプリフェッチ

```
for{for{for{for{
  asm("movsd (%0),%%xmm9"::"r"(ap));
  asm("movddup %xmm9,%xmm0");
  for(j=0; j<N; j+=8){
    asm("prefetcht0 (%0)"::"r"(bp+prefetch));
    asm("prefetcht0 (%0)"::"r"(cp+prefetch));
    asm("movapd (%0),%%xmm1"::"r"(bp));   asm("movapd (%0),%%xmm2"::"r"(bp+2));
    asm("movapd (%0),%%xmm3"::"r"(bp+4)); asm("movapd (%0),%%xmm4"::"r"(bp+6));
    asm("mulpd %xmm0,%xmm1");             asm("mulpd %xmm0,%xmm2");
    asm("mulpd %xmm0,%xmm3");             asm("mulpd %xmm0,%xmm4");
    asm("movapd (%0),%%xmm5"::"r"(cp));   asm("movapd (%0),%%xmm6"::"r"(cp+2));
    asm("movapd (%0),%%xmm7"::"r"(cp+4)); asm("movapd (%0),%%xmm8"::"r"(cp+6));
    asm("mulpd %xmm1,%xmm5");             asm("mulpd %xmm2,%xmm6");
    asm("mulpd %xmm3,%xmm7");             asm("mulpd %xmm4,%xmm8");
    asm("movapd %%xmm5,(%0)"::"r"(cp));   asm("movapd %%xmm6,(%0)"::"r"(cp+2));
    asm("movapd %%xmm7,(%0)"::"r"(cp+4)); asm("movapd %%xmm8,(%0)"::"r"(cp+6));
    bp+=8, cp+=8;
} ap++; }}}}
```

最適化7：パラメータの手動チューニング

- ブロック幅＝32 が最適
- プリフェッチ幅＝64 が最適

- 行列A, B, Cのアドレスを以下に配慮して配置
  - バンクコンフリクト
  - 4KBエイリアシング
  - SIMD命令のためのアラインメント

逐次コードの最適化結果

- 行列サイズ 4096×4096
  - 10009 sec => 173 sec (58倍高速化)

注：double型ではなくfloat型だったらもっと速くできる

Step 2:ノード内並列化

- XMPはハイブリッド並列に未対応
  - 内部的にMPIがノード内通信を共有メモリ経由で実現してくれるとはいえ、ノード内に複数のプロセスを立てるオーバヘッドは無視できない

方針:ノード内は自分でpthreadで並列化する

最適化1：最外ループを分割

```
begin=N/thread_num*thread_id;
end=N/thread_num*(thread_id+1);
for(ii=begin; ii<end; ii+=block){
  for{for{for{for{
          ...;
}}}}}
```

最適化2：CPUの明示的な割当て

```
cpu_set_t mask;
CPU_SET(thread_id, &mask);
sched_setaffinity(..., &mask);
for{for{for{for{for{
            ...;
}}}}}
```

- MPIが通信用スレッドを何本も立ててしまうため、計算スレッドを明示的に CPU に割り当てておくことは性能上重要

Step 3:ノード間並列化

- まず、ノード間並列化の方針を見きわめるため、XMPの基本性能をいろいろ測定してみた

- XMPで十分な性能（＝MPIと同等の性能）を達成できるのはどの機能か？

わかったこと

- gmoveは遅い
  - 定型的な集合通信であっても、MPI_Isend()とMPI_Recv()を組み合わせた汎用的なall-to-all通信に落とされてしまっている

- 全体同期・局所同期ともに同期が遅い

わかったこと

- coarrayによる片側通信は十分な性能が出る
  - ただしmyrinet環境ではないのでRDMAの恩恵は受けられない

- 1Gbit Ethernet環境であるため、通信をうまくスケジューリングしないと輻輳の影響が深刻

ノード間並列化の方針

方針1：coarrayだけで書く

方針2：極力同期を使わない

方針3：通信をスケジューリングする

そこで最初に

- Foxのアルゴリズムを試した
    - 非同期通信と計算をオーバーラップ

- が、意図した性能が出なかった

なぜか？

- 同一ノード上に計算スレッドが走っていると、通信速度が異常に落ちることが判明
  - 別CPUで1本でも計算スレッドが走っていると通信速度が有意に落ちる現象が観測された
  - おそらく実行環境のハードウェアかGASNETの問題

- 結果的に、通信と計算をオーバーラップさせると逆に遅くなる

新たに加わった方針

方針4：計算している最中に通信しない

以上をふまえたアルゴリズム

- 結局、とてもシンプルなものに帰着した

Step 1：プロセッサ0が行列AとBを全員にscatter
Step 2：全体で同期
Step 3：各プロセッサが部分行列積を計算
Step 4：全体で同期
Step 5：結果をプロセッサ0に集める

以上をふまえたアルゴリズム

Step 1：プロセッサ0が行列AとBを全員にscatter

↑↑↑工夫の余地あり

Step 2：全体で同期

Step 3：各プロセッサが部分行列積を計算

Step 4：全体で同期

Step 5：結果をプロセッサ0に集める

↑↑↑ふつうに集めるしかない。どう集めようが、プロセッサ0のネットワーク I/O (1本)で律速されるため

行列AとBをどうばらまくか？

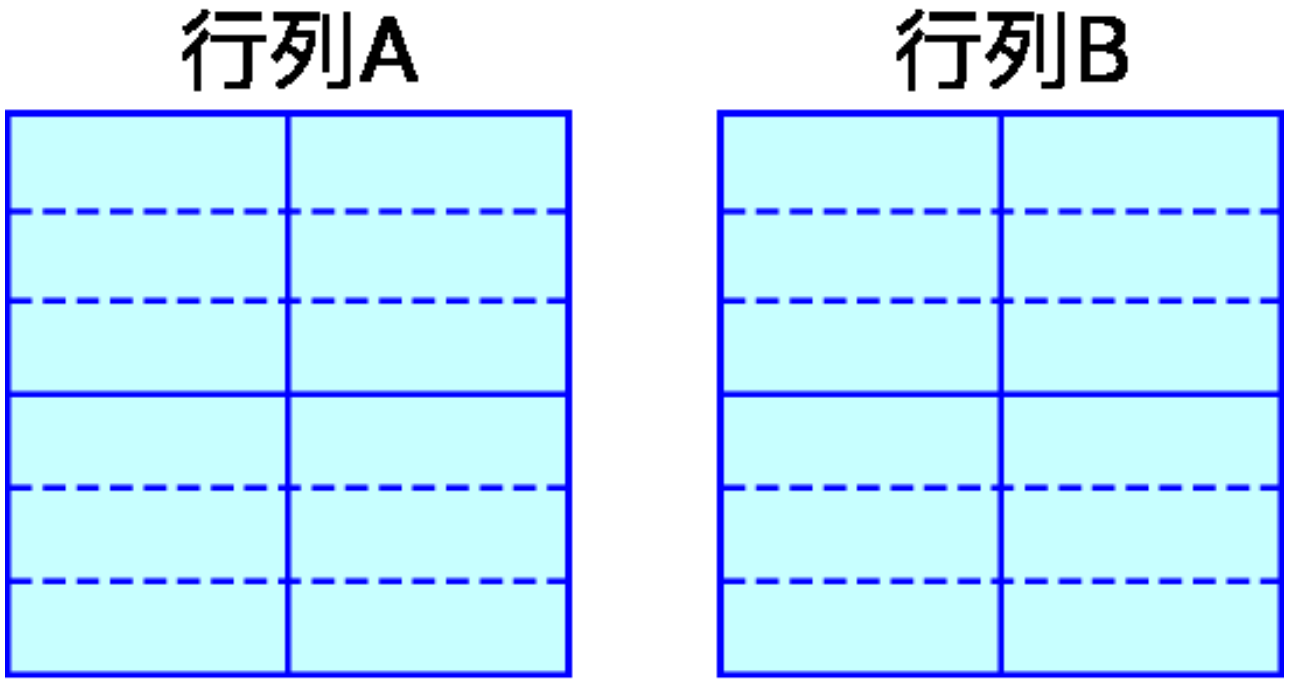
- 代表的な集合通信アルゴリズムを実装してみたが、意図する性能が出なかった

- なぜか？
  - XMPの局所同期が遅い
  - coarrayのメモリバリアも遅い

- 同期回数の少ない集合通信アルゴリズムを発明する必要がある

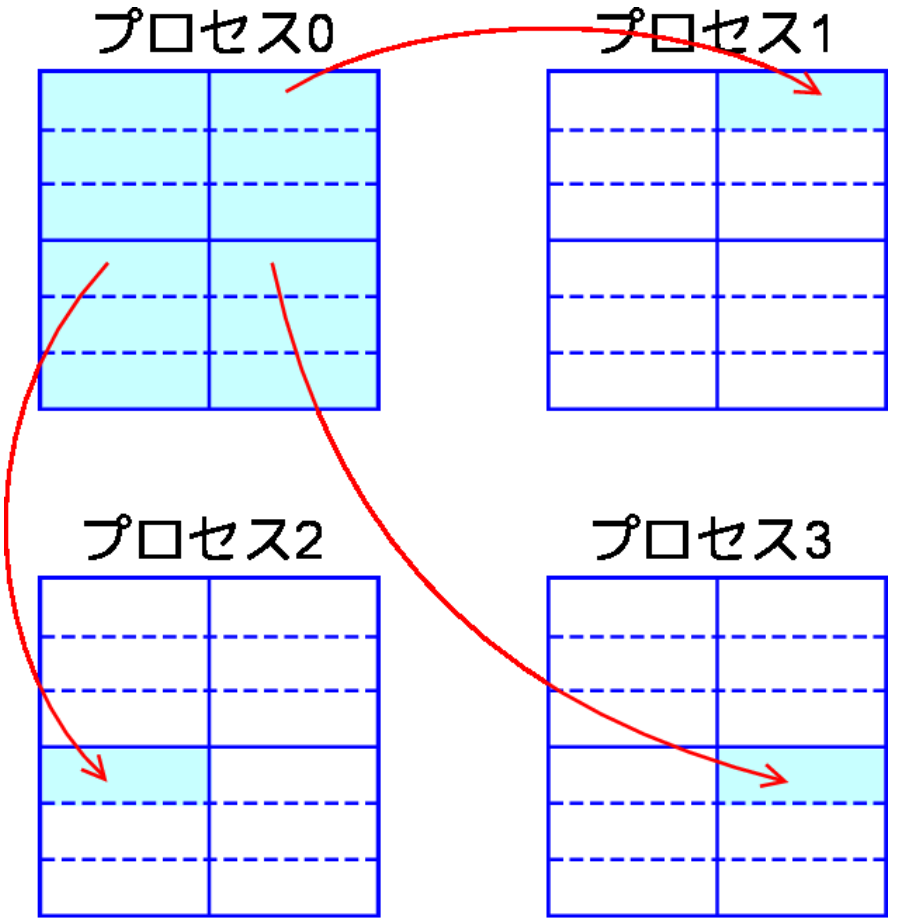
行列をストライピングしてばらまく

- 行列AとBをグリッド分割し、各グリッドを横方向にストライピング

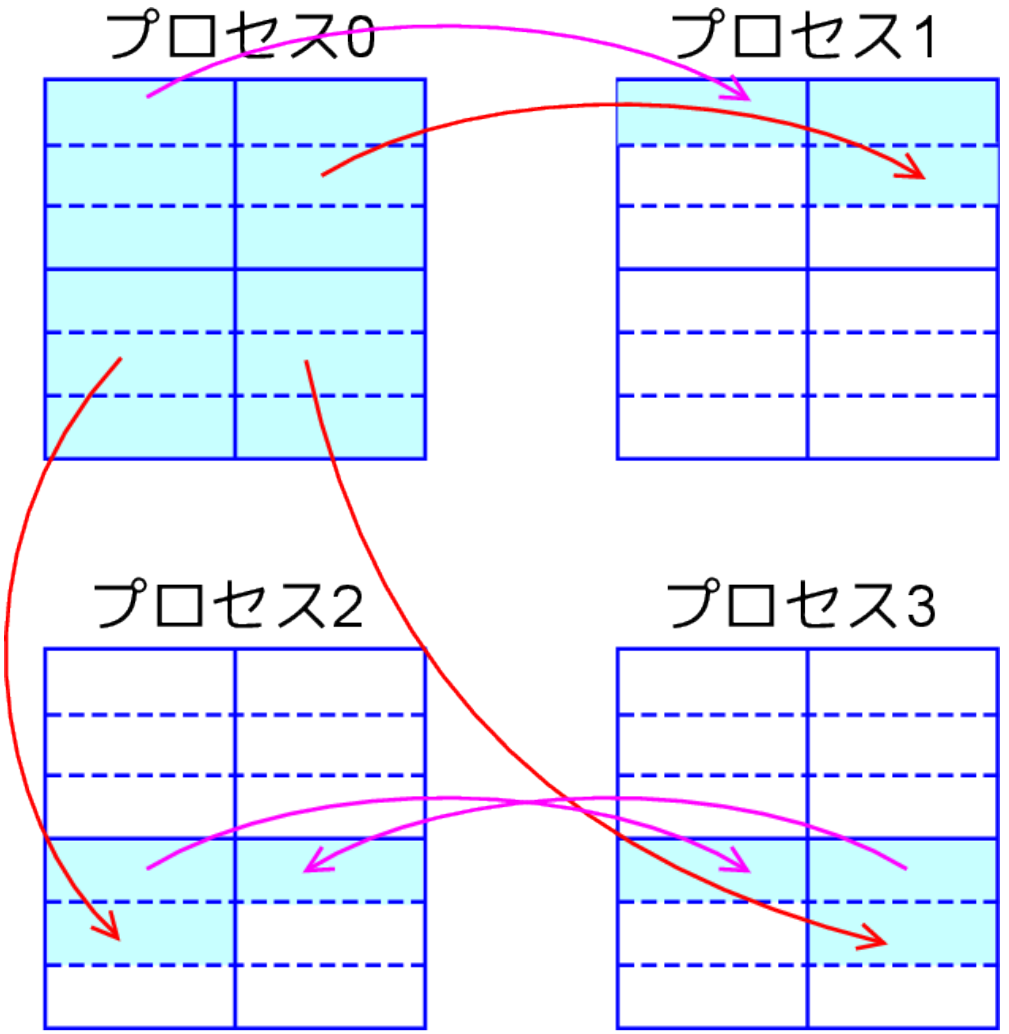


行列をストライピングしてばらまく

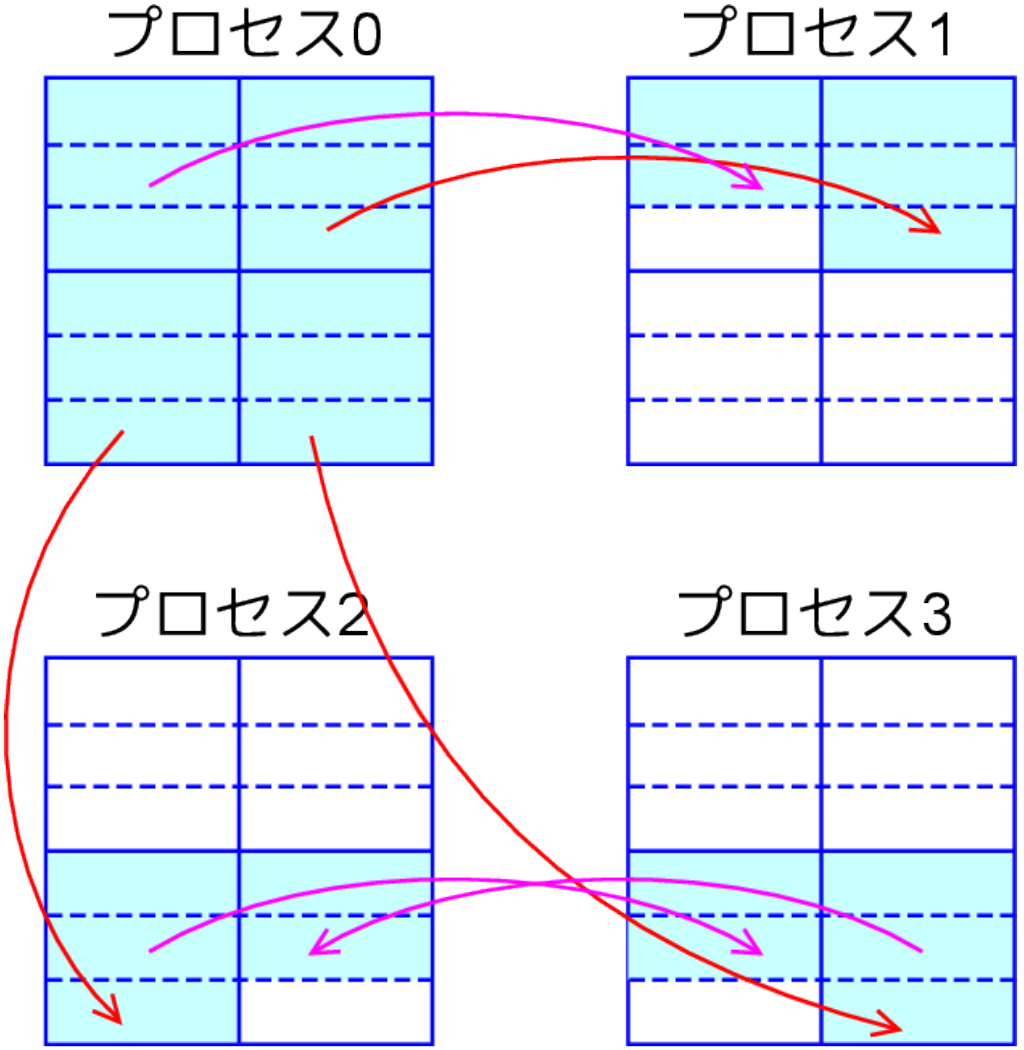
- 最初のストライプをプロセッサ0が各プロセッサに順番に配る
- 全体で同期



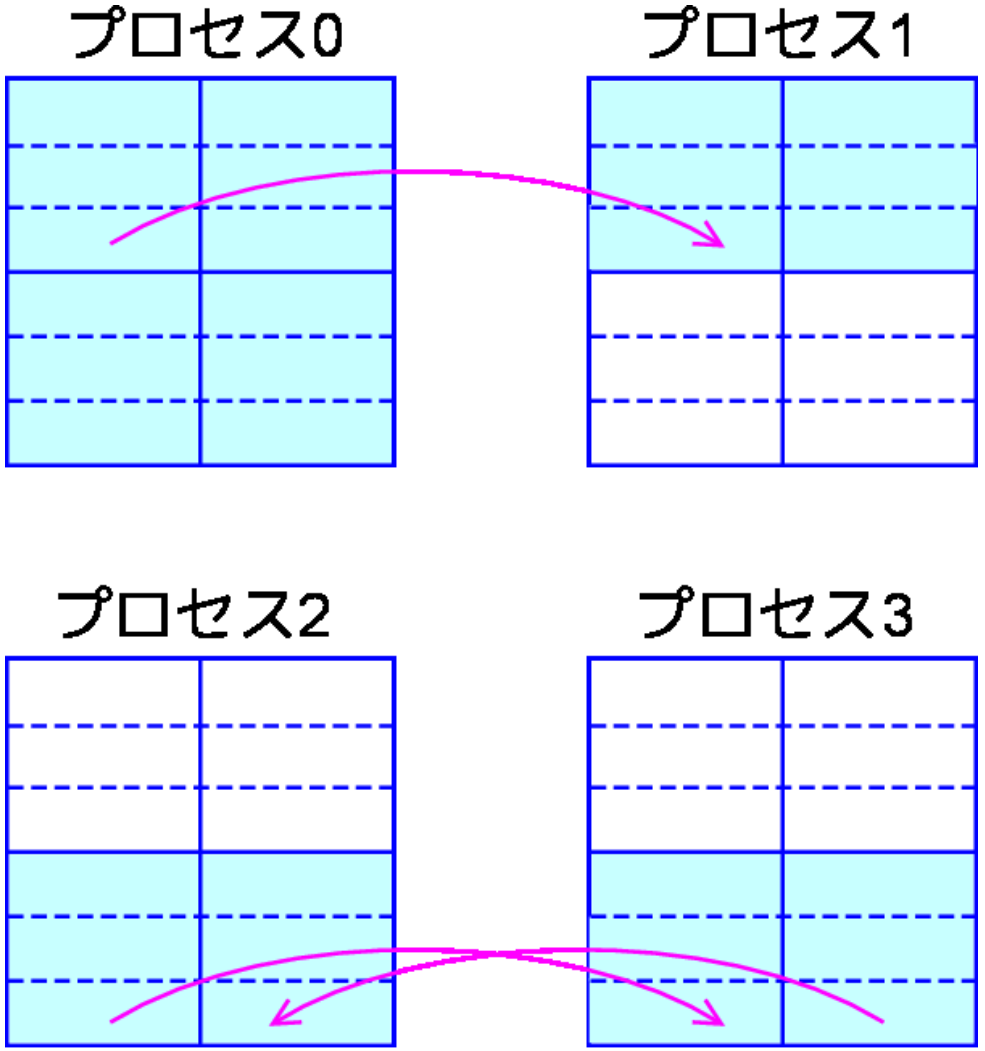
- 各プロセッサは「横方向の」プロセッサたちからストライプを受信
- その間にプロセッサ0は次のストライプを各プロセッサに配る
- 全体で同期



- 各プロセッサは「横方向の」プロセッサたちからストライプを受信
- その間にプロセッサ0は次のストライプを各プロセッサに配る
- 全体で同期



- 各プロセッサは「横方向の」プロセッサたちからストライプを受信
- 全体で同期



プロセッサの割当て方

- このアルゴリズムは平方数個のプロセッサでしか動かない

- XMPプロセス数が平方数になるようにして、スレッド数でつじつまを合わせる
  - 128プロセッサ ＝ 64 XMPプロセス × 2スレッド on 32ノード

最終的な結果

- 行列サイズ 4096×4096

- 逐次プログラムの最適化
  - 10009 sec => 173 sec (58倍高速化)

- 1プロセッサ => 128プロセッサ
  - 173 sec => 5.09 sec (34倍高速化)

最終的な結果

- コード：640行

XMPへの期待

期待1：動的なデータ構造のサポート

- XMPの分散配列とcoarrayは、静的にサイズが決定されたグローバル配列でなければならない

```
#define N 1024
int array[N];
#pragma xmp coarray array : [*]
#pragma xmp template t(0:N)
#pragma xmp nodes p(*)
#pragma xmp distribute t(block) onto p
int matrix[N][N];
```

期待1：動的なデータ構造のサポート

- 行列サイズを ./a.out のパラメータとして渡すこともできない

- グローバル変数なので、行列matrix[N][N]が全プロセスにメモリ確保されてしまう
    - プロセス0でしか使わないのに...

期待1：動的なデータ構造のサポート

- しかし、リアルアプリでは動的なデータ構造が必須

- MPIでは書いていられないようなリアルアプリを自然に記述できるかどうかが、高生産言語の見せ場
  - 非定型な領域分割をともなう有限要素法
  - Adaptive Mesh Refinement
  - 大規模な疎グラフ探索

期待1：動的なデータ構造のサポート

- とはいえ、「ディレクティブベースのトランスレータ」というアプローチを取るかぎり、動的なデータ構造を華麗に扱うのは難しいような気もする

- ディレクティブというのは「静的な」ものだが、それで「動的な」データ分散をうまく記述することはできるのか？？

期待1：動的なデータ構造のサポート

- OpenMPはディレクティブベースで成功を収めているのは事実だが...

- OpenMPのディレクティブとXMPのディレクティブは本質的に意味が異なる
  - OpenMPは「処理の分散」を記述させている
  - XMPは「データの分散」を記述させている

期待1：動的なデータ構造のサポート

- 「処理の分散」
  - forループなどの処理（control flow）はプログラムの字面に静的に記述されている場合が多い

  => 静的なディレクティブでうまくいく

- 「データの分散」
  - たいていのデータ構造は動的に決まる

  => 静的なディレクティブだけでは難しいように思う

期待1：動的なデータ構造のサポート

- ディレクティブベースの記述を動的なデータ構造にまでスケールさせるのはおそらく難しい

- XMPで動的なデータ構造を扱うには、coarray機能の充実が鍵になるだろう

期待2：高速な同期がほしい

- 共有メモリにおいては同期の性能がきわめて重要
  - 局所同期（2ノード間での同期）
  - メモリバリア（coarrayのコンシステンシ維持）

期待2：高速な同期がほしい

- たいていの処理では、なんらか「writeしたものがreadできる」ことを順序保証する必要がある
    - メッセージパッシングでは、必要となる同期がsend/receiveのなかに暗黙的に自然に含有されているのに対して、
    - 共有メモリでは、read/writeと同期が分離されてしまっている
- よって、共有メモリでメッセージパッシング並の性能を達成しようと思えば、高速な同期が非常に重要

期待3：真にグローバルビューな配列がほしい

- XMPでは分散配列のA[i]にアクセスしたとき、
  - そのノードがA[i]を持っていれば正しい値が返る
  - そのノードがA[i]を持っていなければ未定義な値が返る

期待3：真にグローバルビューな配列がほしい

- 結果的に、プログラマがデータ分散を完全に把握していないと正しいプログラムが書けない

- 「グローバルビュー」をうたうからには、どのノードがアクセスしようが `A[i]` はつねに正しい値を返してほしい
    - 少なくとも、違うノードがアクセスした場合にエラーは出てほしい

期待4：透過的なハイブリッド並列

- 今回は手動でpthreadを作ったが...

- ハイブリッド並列は「処理系が勝手にやってくれるもの」であってほしい
  - ハードウェアが多様化すればますますその要求は強まるだろう
    - メニーコア、GPU

XMPへの期待のまとめ

- 動的なデータ構造のサポート
  - ディレクティブベースでは難しいように思う
  - coarray機能の充実が鍵だろう

- 高速な同期

- 真にグローバルビューな配列

- 透過的なハイブリッド並列

高生産並列言語への期待

コンテストをふりかえって...

- たしかにXMPの記法は、逐次プログラムを少しいじるだけで並列化できるようデザインされている

- しかし、性能最適化の見通しが良くないのも事実
  - 満足のいく性能のプログラムを書くのに、MPIと同程度の苦労が必要だった

コンテストをふりかえって…

- 「高生産」であるとはどういうことか？

- 「高生産＝逐次プログラムからの飛躍が少ない」というのはどこまで本当なのか？

「高生産」の定義

- 逐次言語における生産性＝「プログラムを書き始めてから、プログラムを実行し終えるまでの時間の最短化」
    - 「1時間で書いて10秒で実行が終わるC言語」よりも「10分で書いて3分で実行が終わるPython」のほうが生産性が高い

- これは並列言語でも成り立つか？
    - 成り立たない

「高生産」の定義

- そもそもなぜ、わざわざ「並列」で書こうとしているのか？
    - 実行時性能を求めているからにほかならない

- その要請上、並列言語は性能至上主義から脱却するわけにはいかない

「高生産」の定義

- 並列言語における生産性＝「プログラムを書き始めてから、十分な実行時性能を出せるプログラムを完成させられるまでの時間の最短化」
  - 「逐次プログラムからの飛躍が少ない＝高生産」とはかぎらない
  - 「プログラムが短い＝高生産」ともかぎらない

- 要するに、「実行時性能の引き出しやすさ」が最大の鍵

「実行時性能の引き出しやすさ」とは

- 強力な性能最適化の手段が提供されていること
  - 非同期通信
  - 高速な同期
  - 集合通信

- 性能最適化の見通しが良いこと
  - 処理系が勝手にいろいろやらない
  - プログラマがデータの流れを容易に把握できる

どのプログラミングモデルが適切か？

代表的な並列プログラミングモデル:

- メッセージパッシング（send/receive）
  - MPI
- ローカルビューの共有メモリ（get/put）
  - Coarray Fortran
- グローバルビューの共有メモリ（read/write）
  - UPC、分散共有メモリ

どのプログラミングモデルが適切か？

- グローバルビューの共有メモリは、「データの所在」を管理するのが処理系

- そうである以上、プログラマ側でデータの流れを把握するのが難しく、性能最適化の見通しの悪さは避けられない
  - （先ほど定義した意味での）「高生産」を達成するのは難しいのではないか
  - 例：分散共有メモリの失敗

どのプログラミングモデルが適切か？

- ローカルビューの共有メモリは、「データの所在」を管理するのがプログラマ

- よって、データの流れがプログラマにはっきりと見え、性能最適化の見通しが良い

- 「高生産」を達成するには、ローカルビューの共有メモリがひとまずの良い出発点な気がしている
    - 例：Co-array Fortranは実際に普及した

具体的には

- 強力なcoarray機能

- さらに、ふつうのget/putに加えて強力な最適化手段を提供する
  - 高速な同期
  - 高速なメモリバリア
  - 高速な集合通信
  - 非同期get/put
  - 透過的なハイブリッド並列

そうはいっても

- ローカルビューの共有メモリを以って、「高生産です」と主張するのはどこか心もとない

- 性能最適化の見通しの良さを失うことなく、ローカルビューの生産性を高めることはできないものだろうか？

ローカルビューの生産性を高めるには

- なぜローカルビューの生産性は低いのか？
  - 理由：プログラマがイメージしているアルゴリズムはグローバルなインデクシングであるにもかかわらず、実際に記述しなければならないのはローカルなインデクシングだから

- つまり、グローバルインデクシングとローカルインデクシングの変換が生産性低下の主因
  - 非定型な領域分割をともなうリアルアプリではとくに深刻

ローカルビューの生産性を高めるには

- ならば、うまい記法を入れることでインデクシング変換の煩雑さを処理系が吸収することはできないだろうか？
    - そこさえ克服できれば、性能最適化の見通しの良さを失うことなく、ローカルビューの生産性を高められる

- XMPのグローバル配列 <=> ローカル配列変換機能がそれをやっている！

まとめ

- 並列言語の生産性 ≠「逐次プログラムからの飛躍の小ささ」「プログラムの短さ」

- 並列言語の生産性＝「プログラムを書き始めてから、十分な実行時性能を出せるプログラムを完成させられるまでの時間の最短化」
  - ひとことで言えば「実行時性能の引き出しやすさ」

- 高生産並列言語に期待する方向性:

ローカルビューの共有メモリ（強力なcoarray機能）＋自動インデクシング変換