

XcalableMPにおける GPU/CPUワークシェアリング

小田嶋 哲哉^{†1}, 朴 泰祐^{†1,2}, 塙 敏博^{†3}, 児玉 祐悦^{†1,2},
Raymond Namyst^{†4}, Samuel Thibault^{†4},
Olivier Aumage^{†4}, 佐藤 三久^{†1,2}

†1 筑波大学 大学院 システム情報工学研究科

†2 筑波大学 計算科学研究センター

†3 東京大学 情報基盤センター

†4 Bordeaux Sud-Ouest INRIA research center

日仏共同研究 FP3C

研究背景

- GPUクラスタにおけるプログラミング
 - CUDAやOpenCLによるGPUの管理
 - プログラム記述量の増加 (MPI+OpenMP+CUDA)
 - >生産性の低下
- 実行モデル
 - オフローディング (機能分散)
 - >GPU+CPUで性能向上が得られるはず (負荷分散)
 - CPUの性能を余らせるのはもったいない

研究の目的

- GPUクラスタ上でのGPU/CPUワークシェアリングを高い言語レベルで容易に行えるプログラミング環境の提供
 - 高性能
 - 高生産性

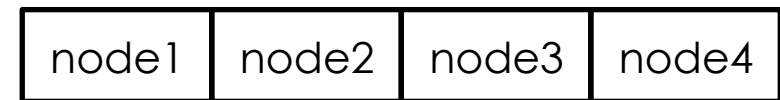
並列言語XcalableMP (XMP)

- PCクラスタコンソーシアムの並列プログラミング言語XcalableMP規格部会で策定
- 分散メモリ型システムを対象としたPGAS並列プログラミングモデル
- 既存のプログラム+指示文
 - プログラミングモデルが分かりやすい
 - プログラムの生産性が高い

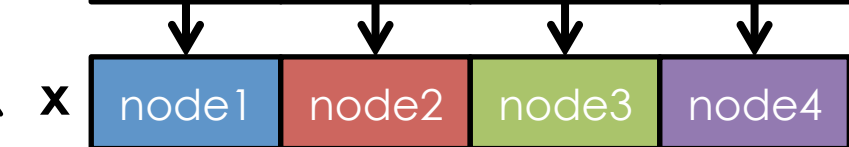
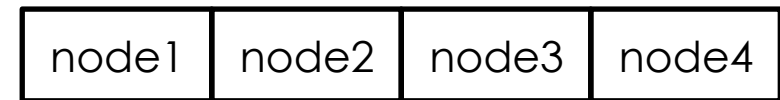
```
#pragma xmp nodes p(4)  
#pragma xmp template t(0 : N-1)
```



```
#pragma xmp distribute t(BLOCK) onto p
```



```
#pragma xmp align x[i] with t(i)
```



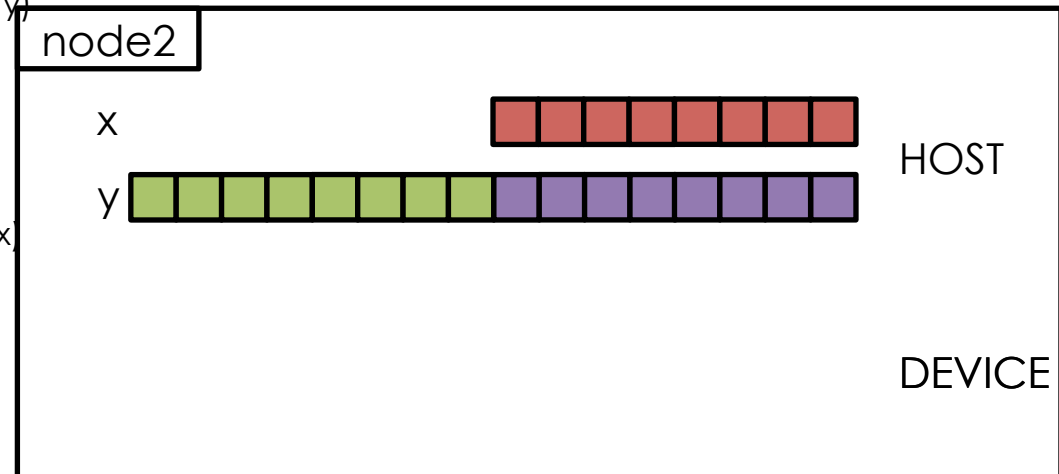
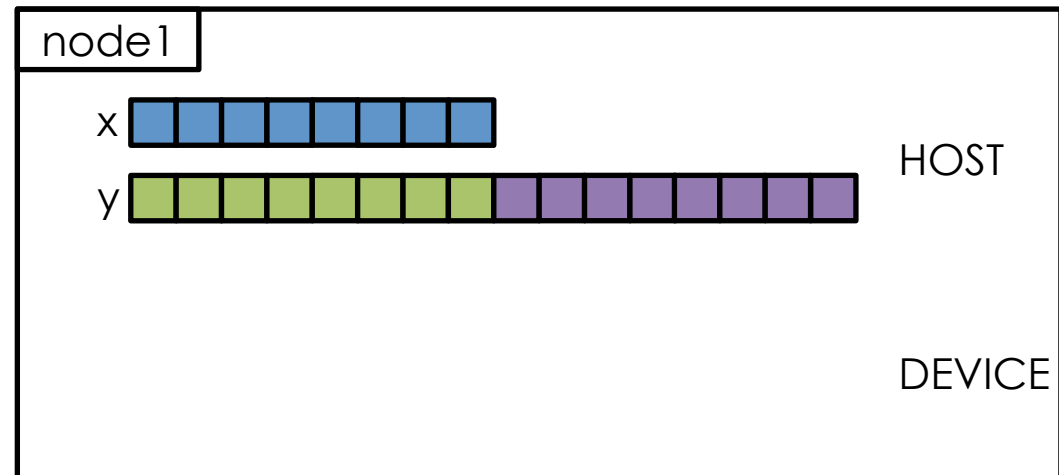
```
x  
#pragma xmp loop on t(i)  
for (i = 0; i < N; i++) { ... }
```

XcalableMP acceleration device extension (XMP-dev)

- 筑波大学HPCS研究室, 理研AICSで開発中
- XMPのGPUクラスタ向け拡張仕様
 - 李らのCUDA, 野水らのOpenCLの実装
 - XMP-dev/CUDAをベースに拡張
 - pragmaからnative cuda/OpenCLコードを生成
- CPU-GPU間のデータ転送, GPUへのオフロードを指示文で簡潔に記述

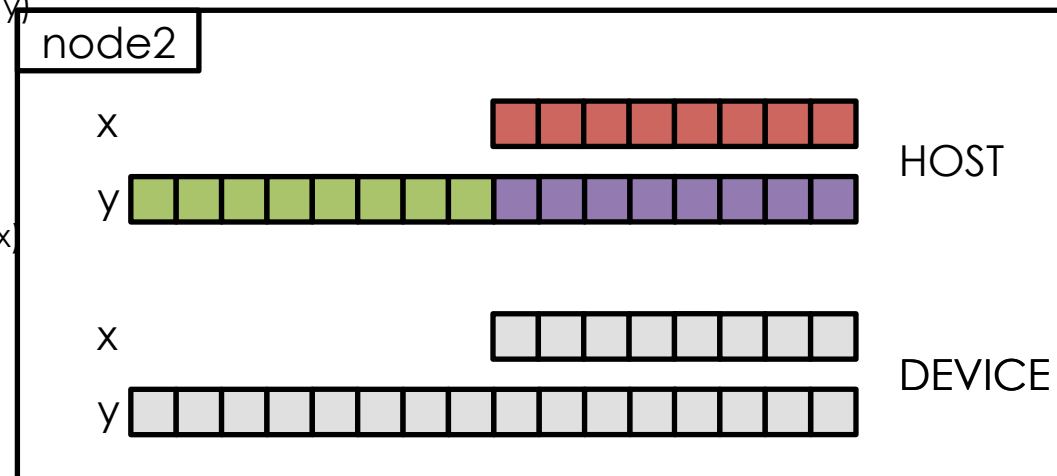
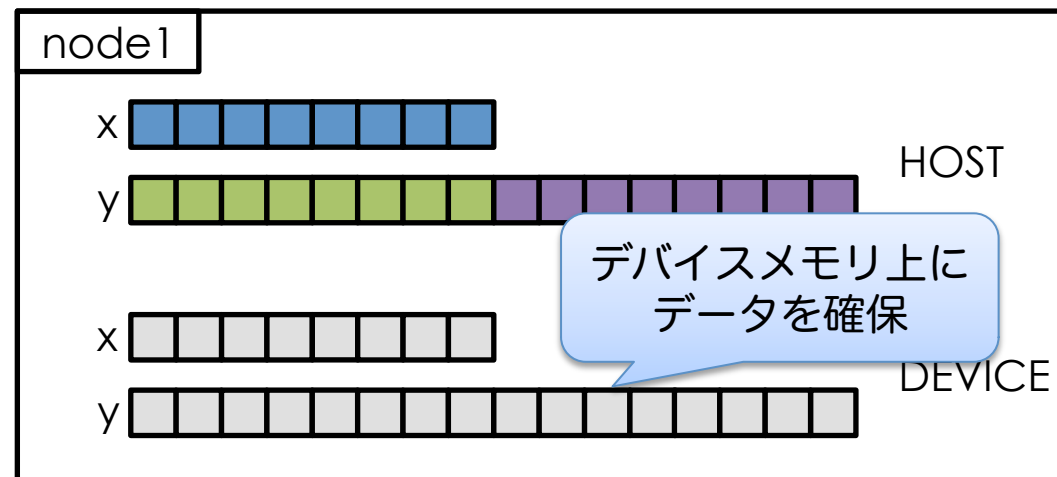
XMP-devの例

```
int x[16], y[16];
#pragma xmp nodes p(2)
#pragma xmp template t(0:16-1)
#pragma xmp distribute t(BLOCK) onto p
#pragma xmp align [i] with t(i) :: x, y
#pragma xmp shadow [*] :: y
int main() {
...
#pragma xmp reflect (y)
#pragma xmp device replicate (x, y)
{
#pragma xmp device replicate_sync in (x, y)
#pragma xmp device loop on t(i)
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            x[i] += y[j];
#pragma xmp device replicate_sync out (x)
}
}
```



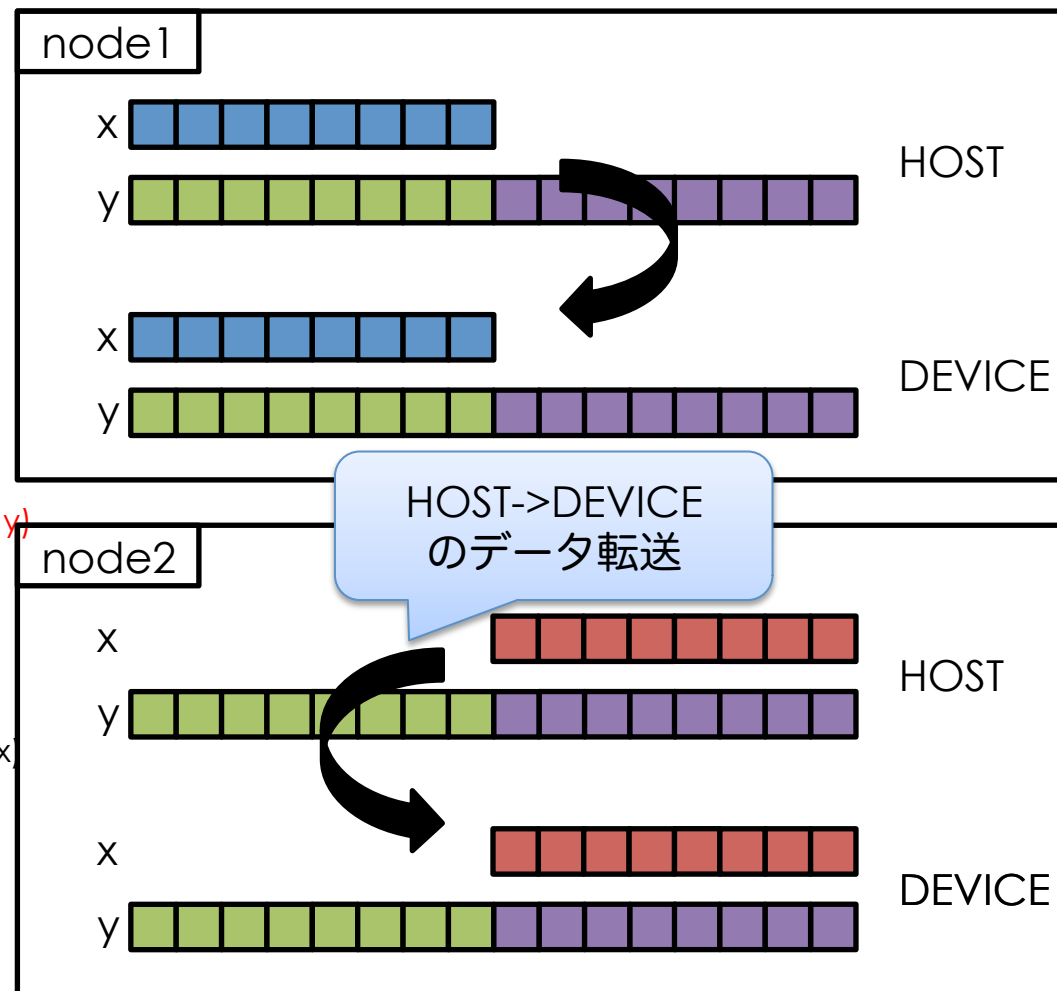
XMP-devの例

```
int x[16], y[16];
#pragma xmp nodes p(2)
#pragma xmp template t(0:16-1)
#pragma xmp distribute t(BLOCK) onto p
#pragma xmp align [i] with t(i) :: x, y
#pragma xmp shadow [*] :: y
int main() {
...
#pragma xmp reflect (y)
#pragma xmp device replicate (x, y)
{
#pragma xmp device replicate_sync in (x, y)
#pragma xmp device loop on t(i)
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            x[i] += y[j];
#pragma xmp device replicate_sync out (x)
}
}
```



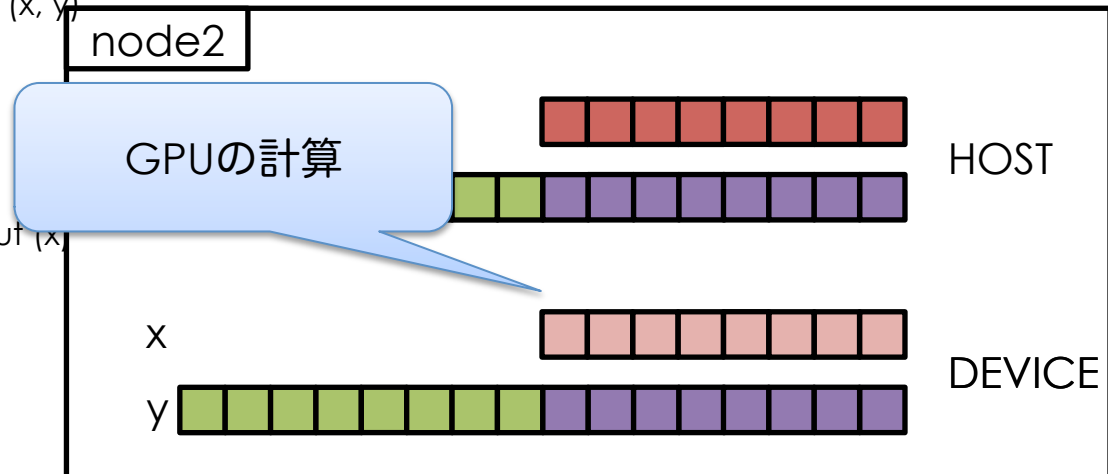
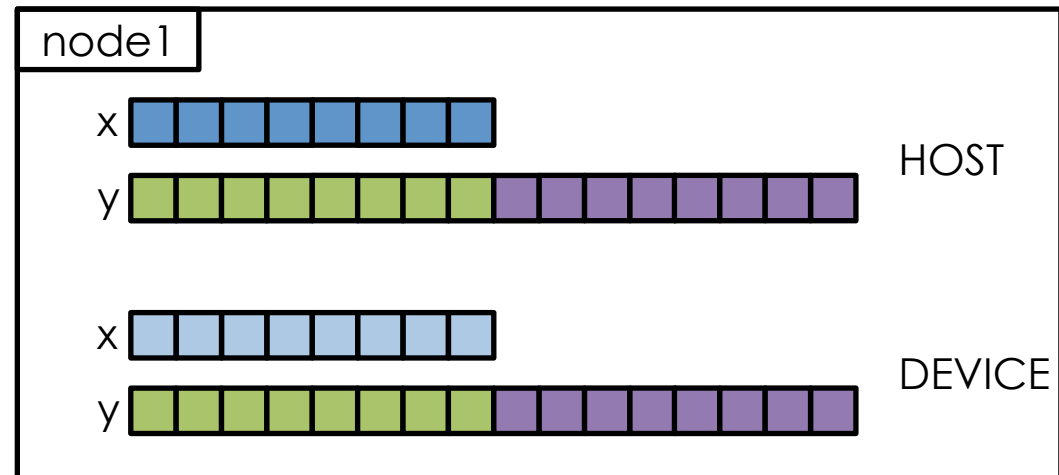
XMP-devの例

```
int x[16], y[16];
#pragma xmp nodes p(2)
#pragma xmp template t(0:16-1)
#pragma xmp distribute t(BLOCK) onto p
#pragma xmp align [i] with t(i) :: x, y
#pragma xmp shadow [*] :: y
int main() {
...
#pragma xmp reflect (y)
#pragma xmp device replicate (x, y)
{
  #pragma xmp device replicate_sync in (x, y)
  #pragma xmp device loop on t(i)
    for (i = 0; i < 16; i++)
      for (j = 0; j < 16; j++)
        x[i] += y[j];
  #pragma xmp device replicate_sync out (x)
}
}
```



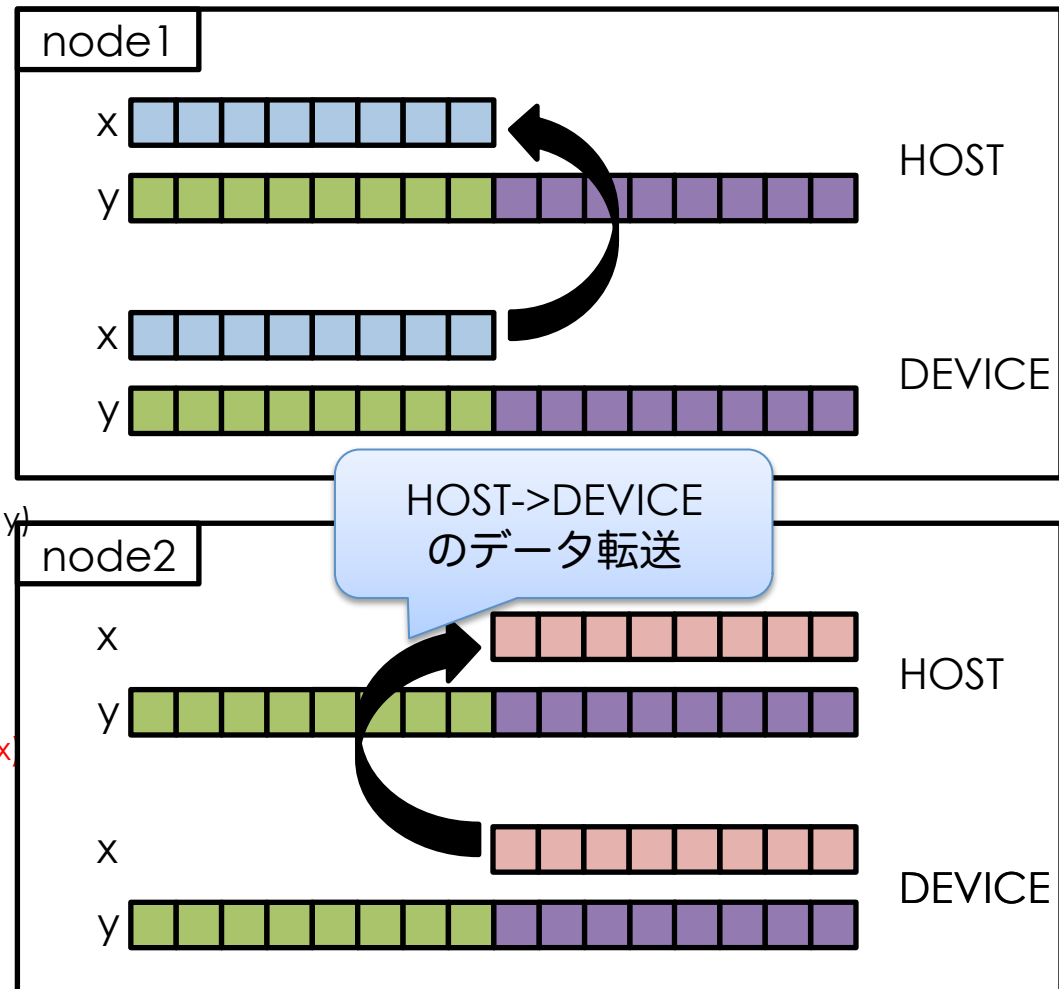
XMP-devの例

```
int x[16], y[16];
#pragma xmp nodes p(2)
#pragma xmp template t(0:16-1)
#pragma xmp distribute t(BLOCK) onto p
#pragma xmp align [i] with t(i) :: x, y
#pragma xmp shadow [*] :: y
int main() {
...
#pragma xmp reflect (y)
#pragma xmp device replicate (x, y)
{
#pragma xmp device replicate_sync in (x, y)
#pragma xmp device loop on t(i)
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            x[i] += y[j];
#pragma xmp device replicate_sync out (x)
}
}
```



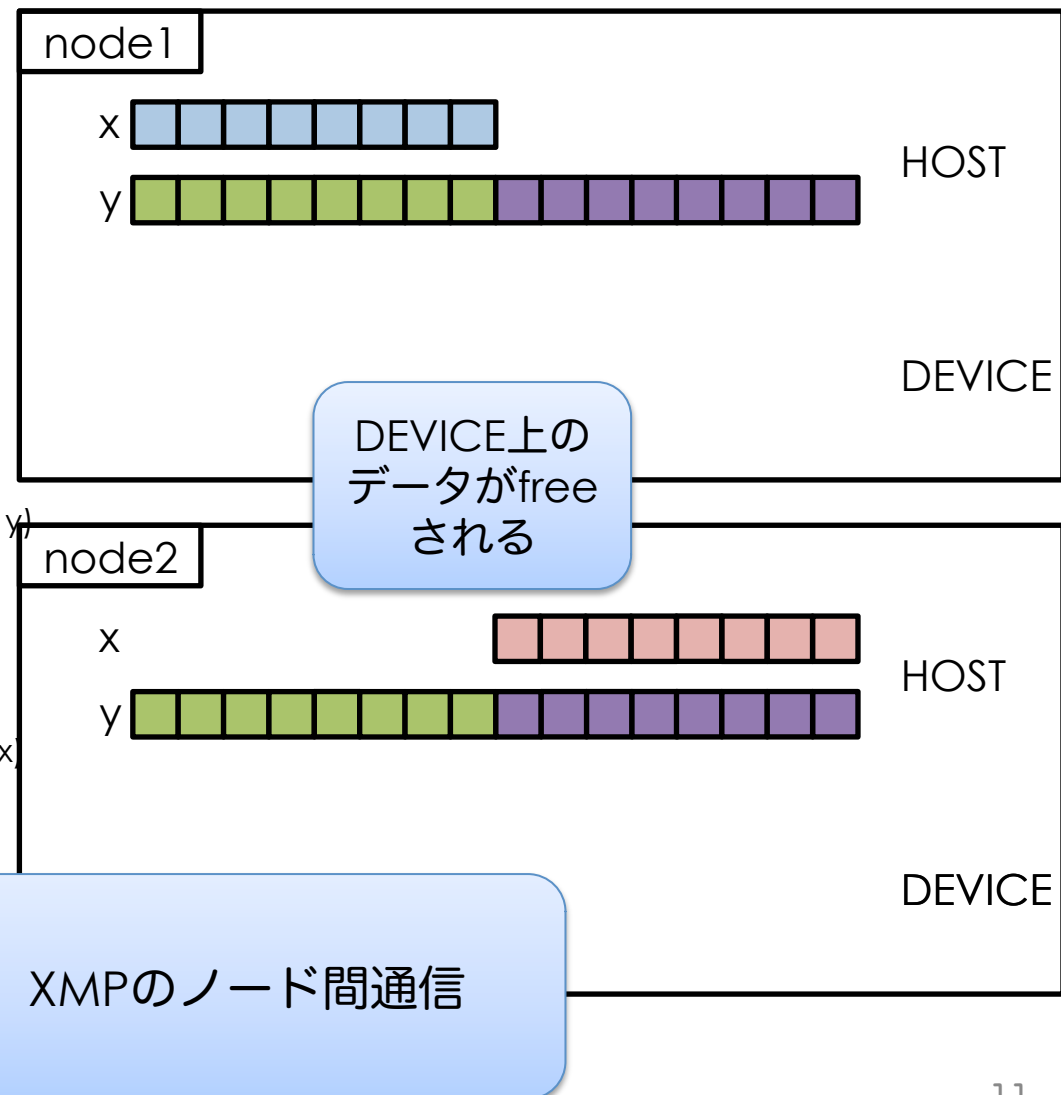
XMP-devの例

```
int x[16], y[16];
#pragma xmp nodes p(2)
#pragma xmp template t(0:16-1)
#pragma xmp distribute t(BLOCK) onto p
#pragma xmp align [i] with t(i) :: x, y
#pragma xmp shadow [*] :: y
int main() {
...
#pragma xmp reflect (y)
#pragma xmp device replicate (x, y)
{
#pragma xmp device replicate_sync in (x, y)
#pragma xmp device loop on t(i)
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            x[i] += y[j];
#pragma xmp device replicate_sync out (x)
}
}
```



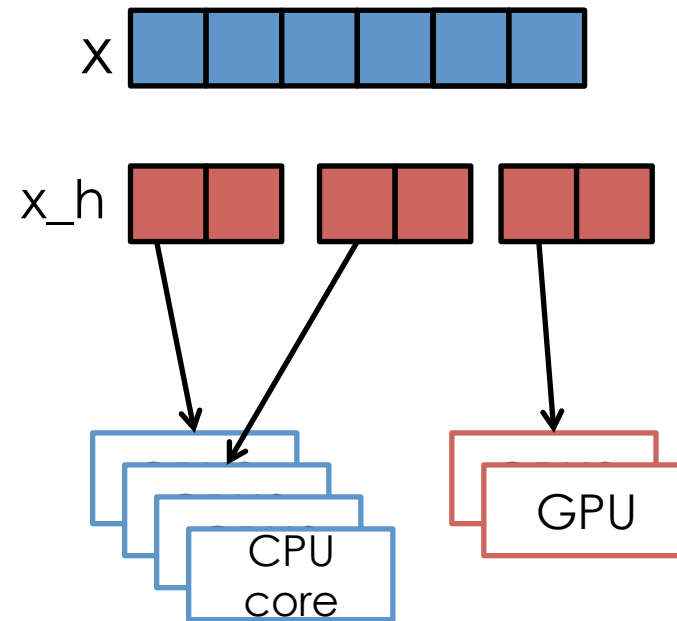
XMP-devの例

```
int x[16], y[16];
#pragma xmp nodes p(2)
#pragma xmp template t(0:16-1)
#pragma xmp distribute t(BLOCK) onto p
#pragma xmp align [i] with t(i) :: x, y
#pragma xmp shadow [*] :: y
int main() {
...
#pragma xmp reflect (y)
#pragma xmp device replicate (x, y)
{
#pragma xmp device replicate_sync in (x, y)
#pragma xmp device loop on t(i)
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            x[i] += y[j];
#pragma xmp device replicate_sync out (x)
}
```



StarPU

- フランスのINRIA Bordeauxが開発
- ノード内のデータの分散, タスクのダイナミックスケジューリングによるGPU/CPU負荷分散
- メモリの一貫性を保証
- 逐次のソースコードをStarPUの形式に書き換える必要がある
 - コストが大きい



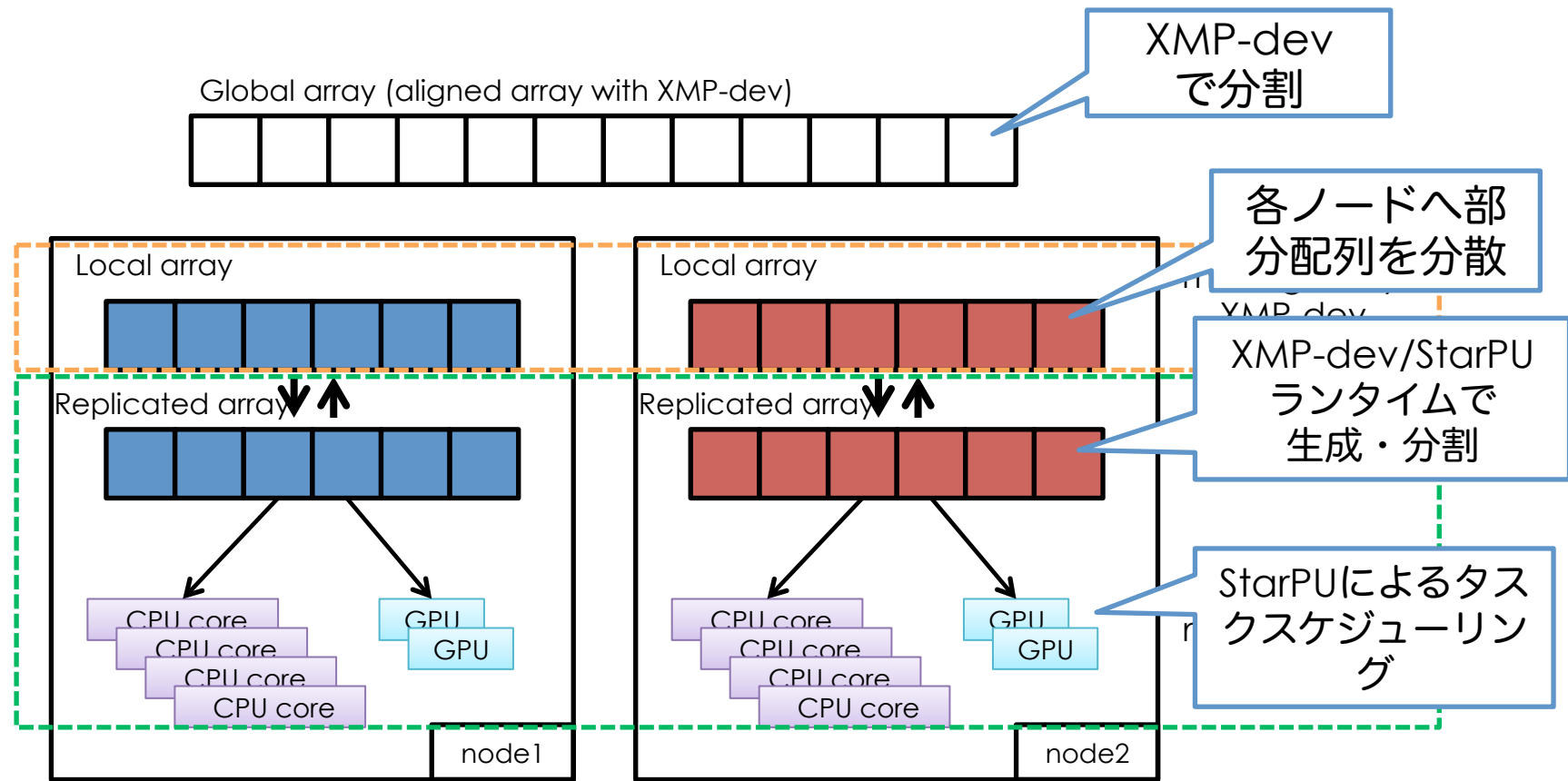
XMP-dev/StarPUの概要

- XMP-devとStarPUを組み合わせる
 - GPUクラスタ上でのGPU/CPUのワークシェアリングが可能



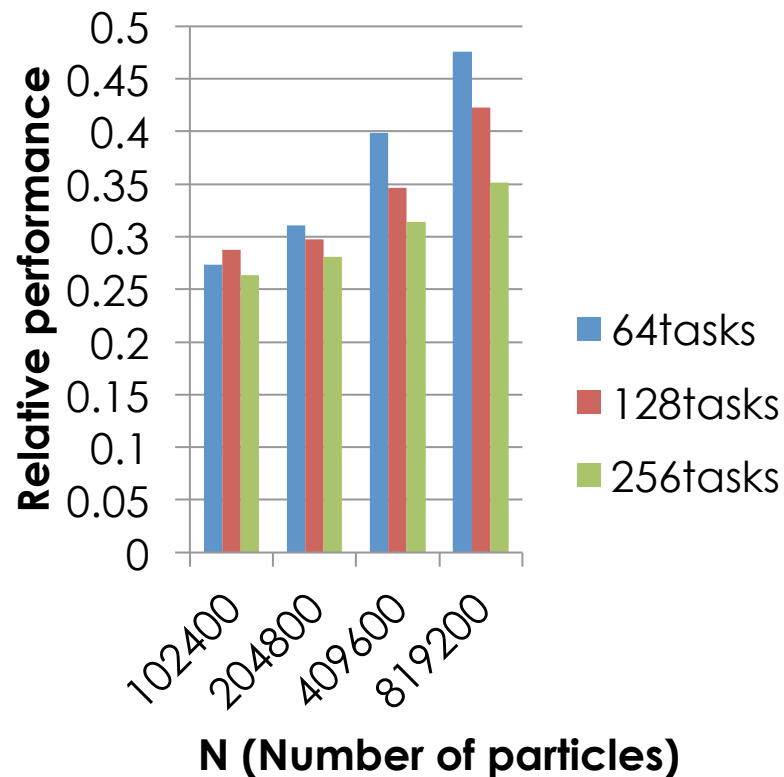
- XMP-devに対するメリット
 - GPUへのオフロードだけでなくCPUも計算に利用できる
- StarPUに対するメリット
 - StarPU形式で記述することなく、高いレベルでのプログラミングが出来る

XMP-dev/StarPUの実装

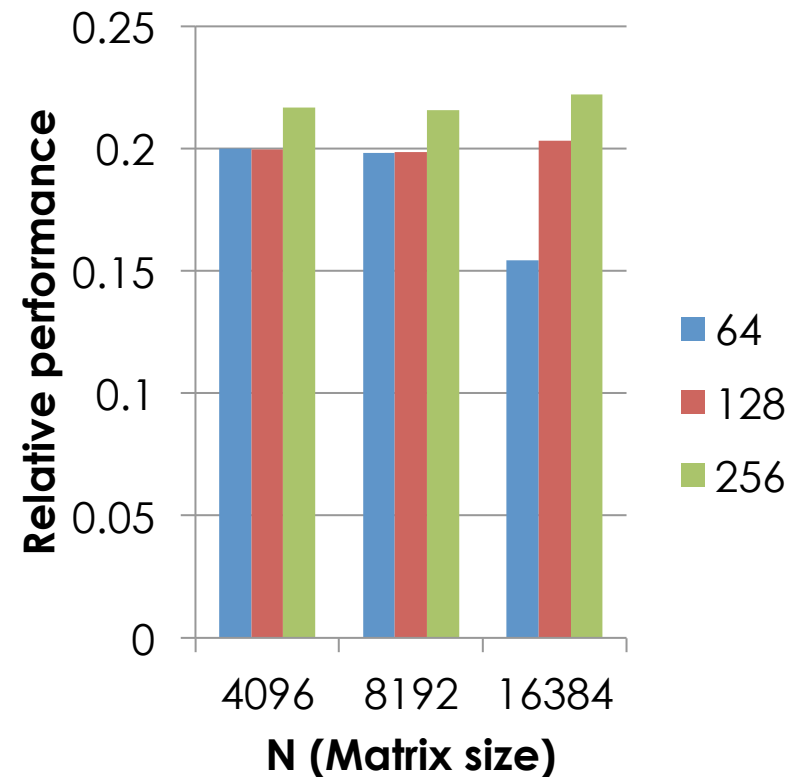


XMP-dev/CUDA (GPUのみを使用) に対する相対性能

N体問題



行列積



HA-PACS 2ノード 1GPU, 15CPU core使用

一般的なHybrid Work Sharingの問題

- 十分なタスク数,十分なタスクサイズがあるとき



CPU

GPU

CPUが演算中に多数の
タスクをGPUが処理



ロードバランスは良好

- 十分なタスク数がないとき



CPU

GPU

- 十分なタスクサイズがないとき



CPU

GPU

トレードオフの関係



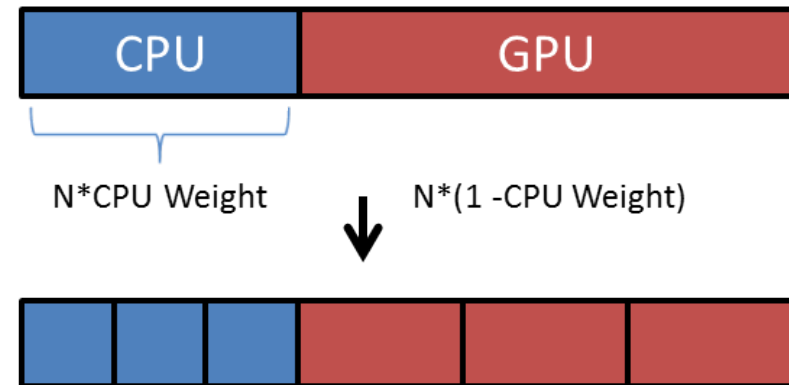
最適解が見つからない
可能性もある



計算リソースごとに
タスクサイズを調節
する必要がある

負荷バランス調整

- 十分なタスクサイズを割り当てるためには、リソースごとに異なるサイズを設定する必要がある
- 計算リソースに応じた演算量を割り当てる
 - “CPU Weight”パラメータの定義
 - $0 \leq \text{CPU Weight} \leq 1.0$
 - Replicated arrayのCPUが計算を担当する領域を設定
 - それぞれの領域を分割し、StarPUで各デバイスにスケジューリングをする
 - 負荷バランスの調整が可能
 - CPU Weightは
 - 問題
 - 問題サイズ
 - アプリケーション特性
 - etc...



動的負荷分散にむけて

- 静的にCPU Weightを設定することは困難
 - 問題やサイズによって適切な値は異なる
 - パラメータを変えつつ値を求める必要がある
 - >コストが非常に大きい



- プログラム中で動的に変化させる機能を導入
 - 時間発展するプログラム中でCPU Weightが最適値に均衡することが期待できる

XMP-dev/StarPUによる動的負荷分散機能

- reset_weight指示文の導入

```
double cpu_weight;  
#pragma xmp device reset_weight (cpu_weight) :: list
```

- cpu_weight何らかの値で初期化する
 - Default : 0.2 (CPU 20%, GPU 80%)
- 時間発展ループ中に複数のfor文があるとき, それぞれにcpu_weightを設定することもできる

reset_weight指示文の例

```
double new_cpu_weight = 0.2;  
for (int t = 0; t < STEP; t++) {  
    // 演算など
```

```
double cpu_time = xmp_cpu_wtime();  
double gpu_time = xmp_gpu_wtime();
```

各デバイスの
実行時間

```
double cpu_ratio =  
    cpu_time / (cpu_time + gpu_time) * 100;
```

CPUの実行時間
の割合計算

```
if (cpu_ratio > 51) {new_cpu_weight -= 0.01;  
#pragma xmp device reset_weight (new_cpu_weight) }  
else if (cpu_ratio < 49) {new_cpu_weight += 0.01;  
#pragma  
}
```

調整アルゴリズムは一例であり、ユーザがより最適なアルゴリズムを用いることでより速い収束を得ることが可能になる

評価環境

- ノード構成 (HA-PACS)

CPU	Intel Xeon E5-2670 * 2 (16 cores)
Memory	DDR3 128GB
GPU	NVIDIA Tesla M2090 * 4
CUDA toolkit	4.2
MPI	MVAPICH2 1.8.1
Interconnection	Infiniband QDR 4x 2 rails
# of node	2~16

– StarPUは1GPUの管理に1 CPU core消費

- ex) 4 GPUs -> 12 (=16 – 4) CPU cores

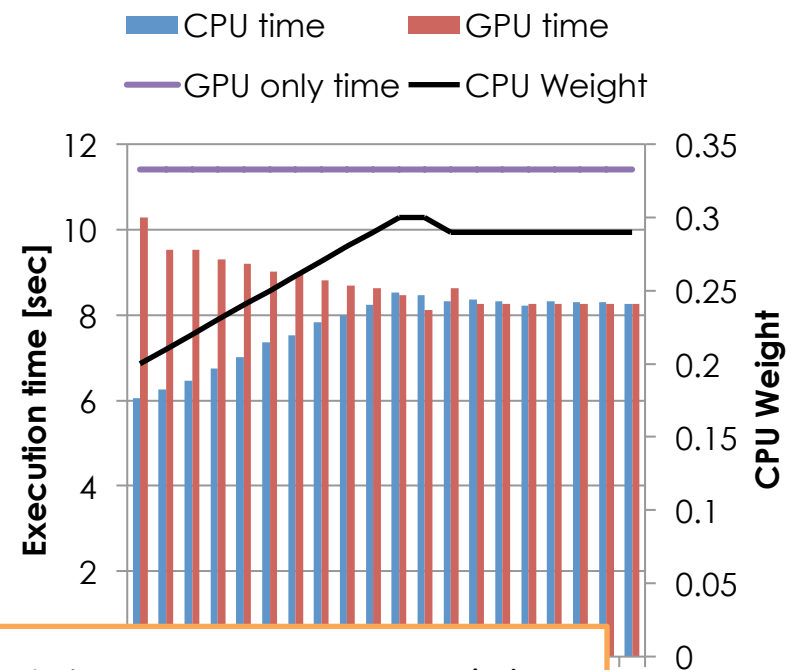
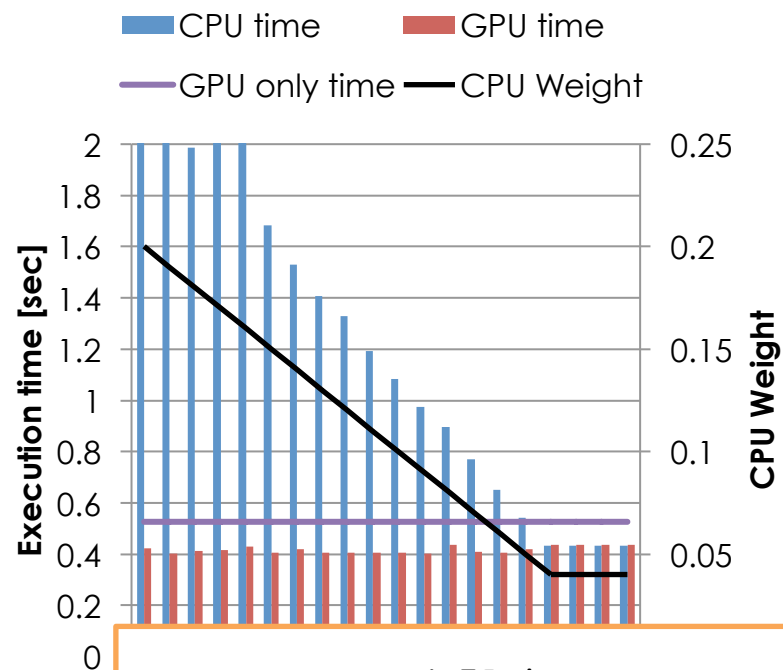
評価手法

- ベンチマーク
 - N体問題, 行列積
 - 最外ループを追加し, その中で何度も計算を行う
 - 計算にはBLAS (MAGMA blas, Gotoblas) を用いる
- 評価について
 - 計測はMPIの通信を含まない (∴割合が小)
 - CPU Weightを0.2から始めてTIME SETPの1イテレーションごとに計算し, 新たな値を設定する
 - 問題サイズ, ノード内のGPU数を変化させ測定
 - CPU Weightの動的調整を利用
 - 先ほどのアニーリングアルゴリズムを利用

CPU Weightの推移

N体問題 (粒子数 : 102400)
2node, 4GPU/node

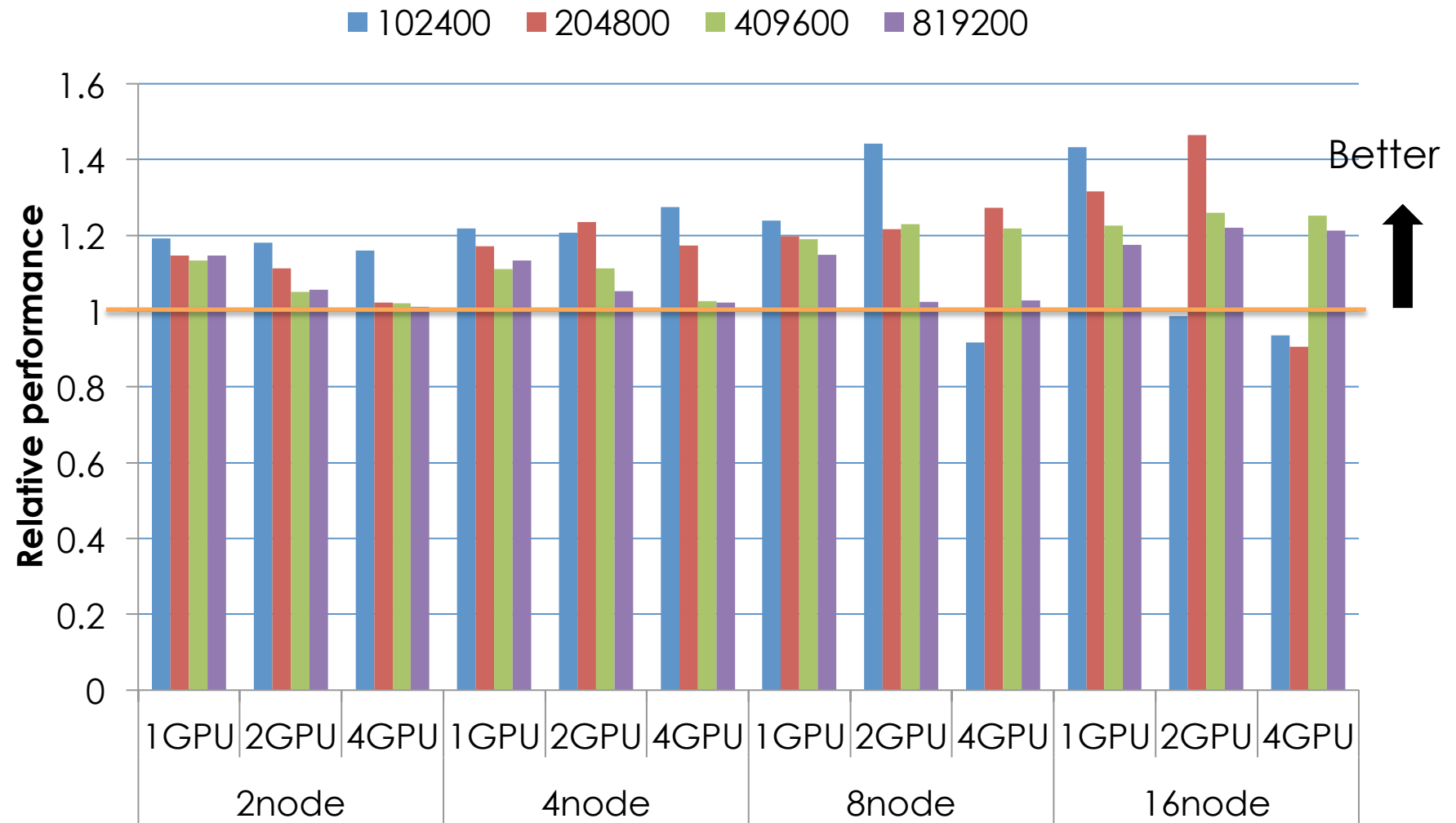
行列積 (行列サイズ : 16384)
2node, 1GPU/node



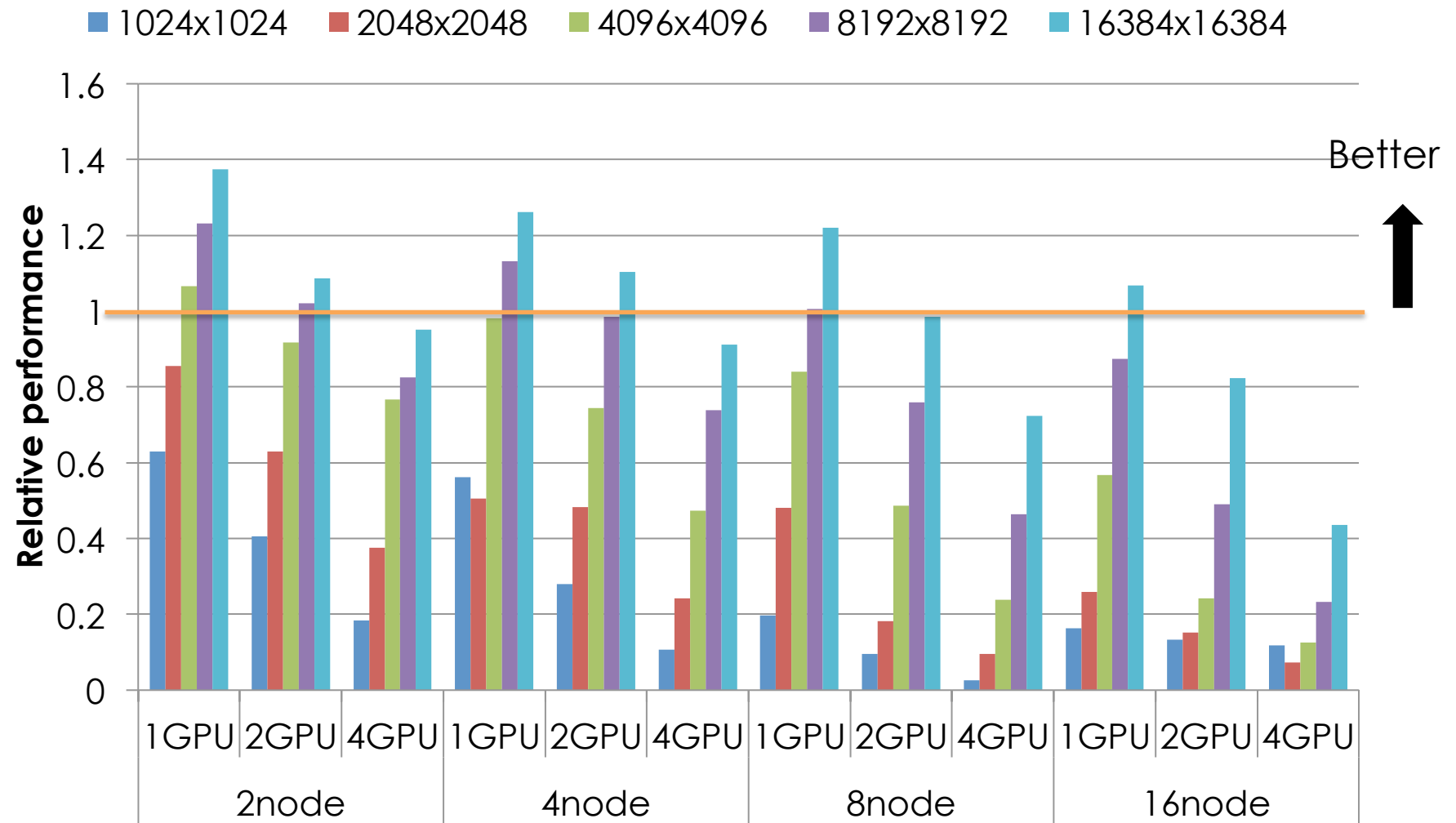
CPU Weightを設定していくことにより, CPUとGPUの実行時間が近づいている
→動的負荷分散がうまくいっている

CPU Weight
初期値 : 0.2

XMP-dev/CUDA (=GPUのみを使用) に対する相対性能 : N体問題



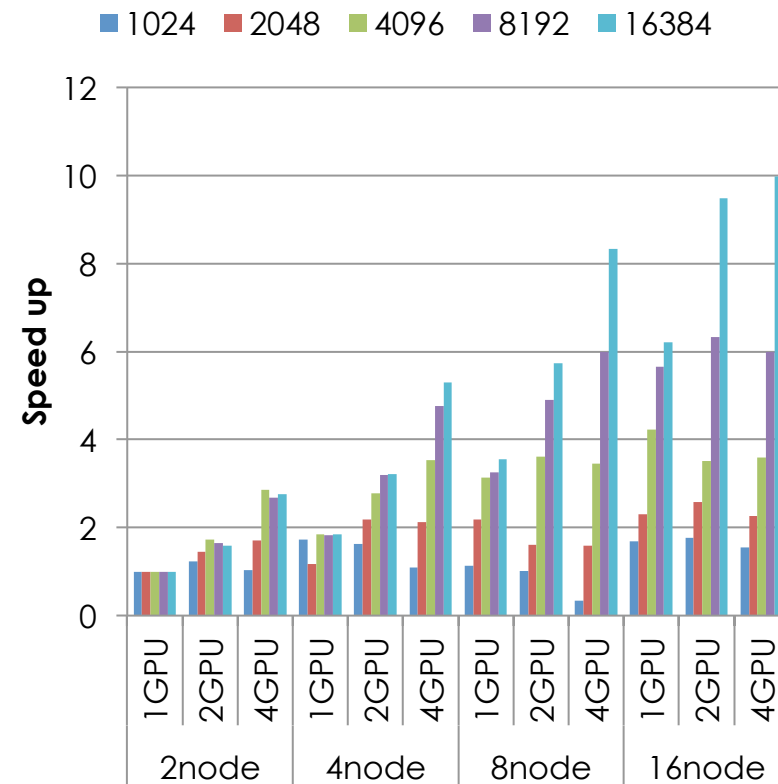
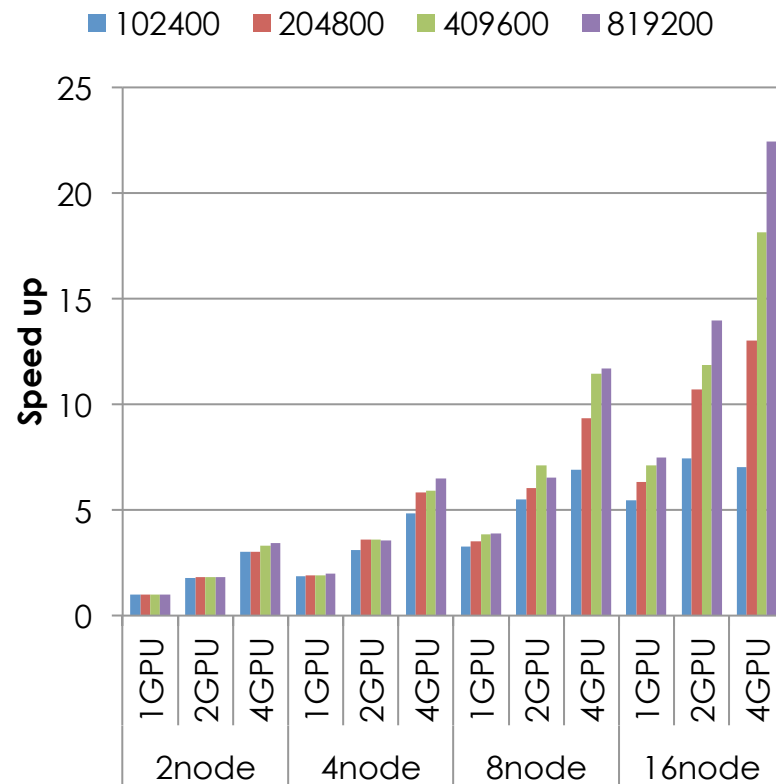
XMP-dev/CUDA (=GPUのみを使用) に対する相対性能：行列積



スピードアップ (2node 1GPUを基準)

N体問題

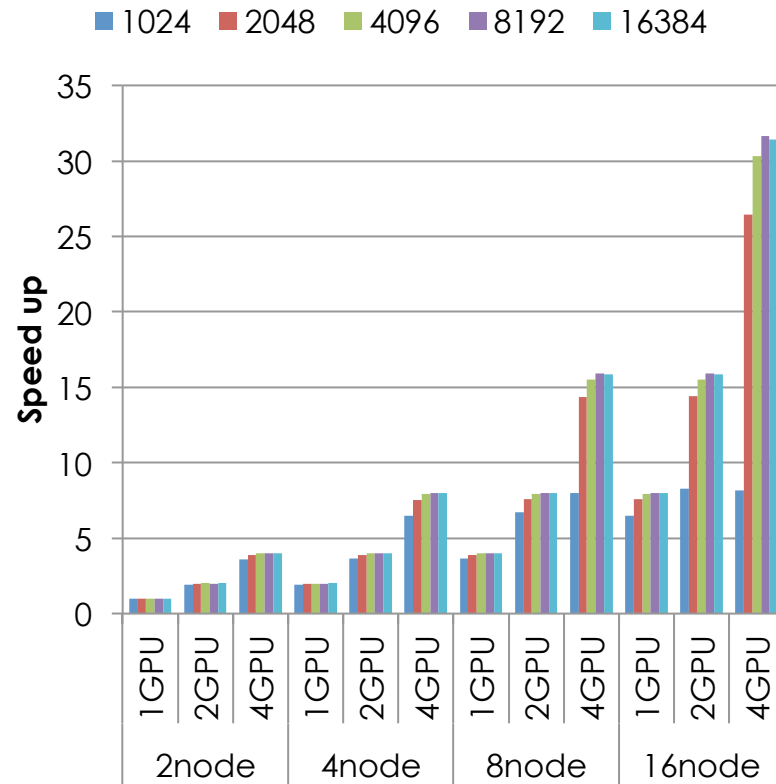
行列積



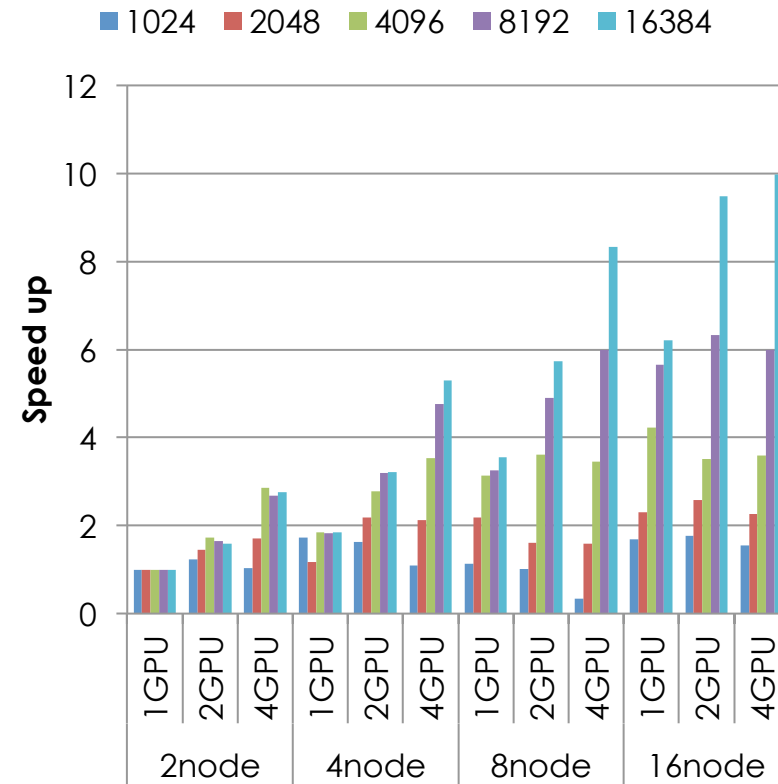
- ・ N体問題はリソースが増えるごとに性能が向上している
- ・ 行列積は性能向上が小さく、さらに性能が劣化している部分が多く見られる

スピードアップ比較：行列積

XMP-dev/CUDA



XMP-dev/StarPU



- ・ XMP-dev/CUDA：問題サイズが十分なときにはスケール
- ・ XMP-dev/StarPU：収束したCPU Weightの時に最適な性能が出るとは限らない -> あえてCPU Weightをずらす必要もある

まとめ・今後の課題

- 動的負荷分散をユーザが意識することなく記述することができた
- 動的にCPU Weightを調整することで、最大で20%～40%程度の速度向上を得ることができた
- 様々なアプリケーションでの評価
- スケールアップ実験
 - 行列積の2次元分割対応

ご清聴ありがとうございました