MPIを埋め込み可能な GPUプログラミングフレームワークの検討

三好健文　近藤正章　入江英嗣　吉永努　本多弘樹
電気通信大学大学院情報システム学研究科

発表の概要

MPIを埋め込み可能なGPUプログラミングフレームワークの検討

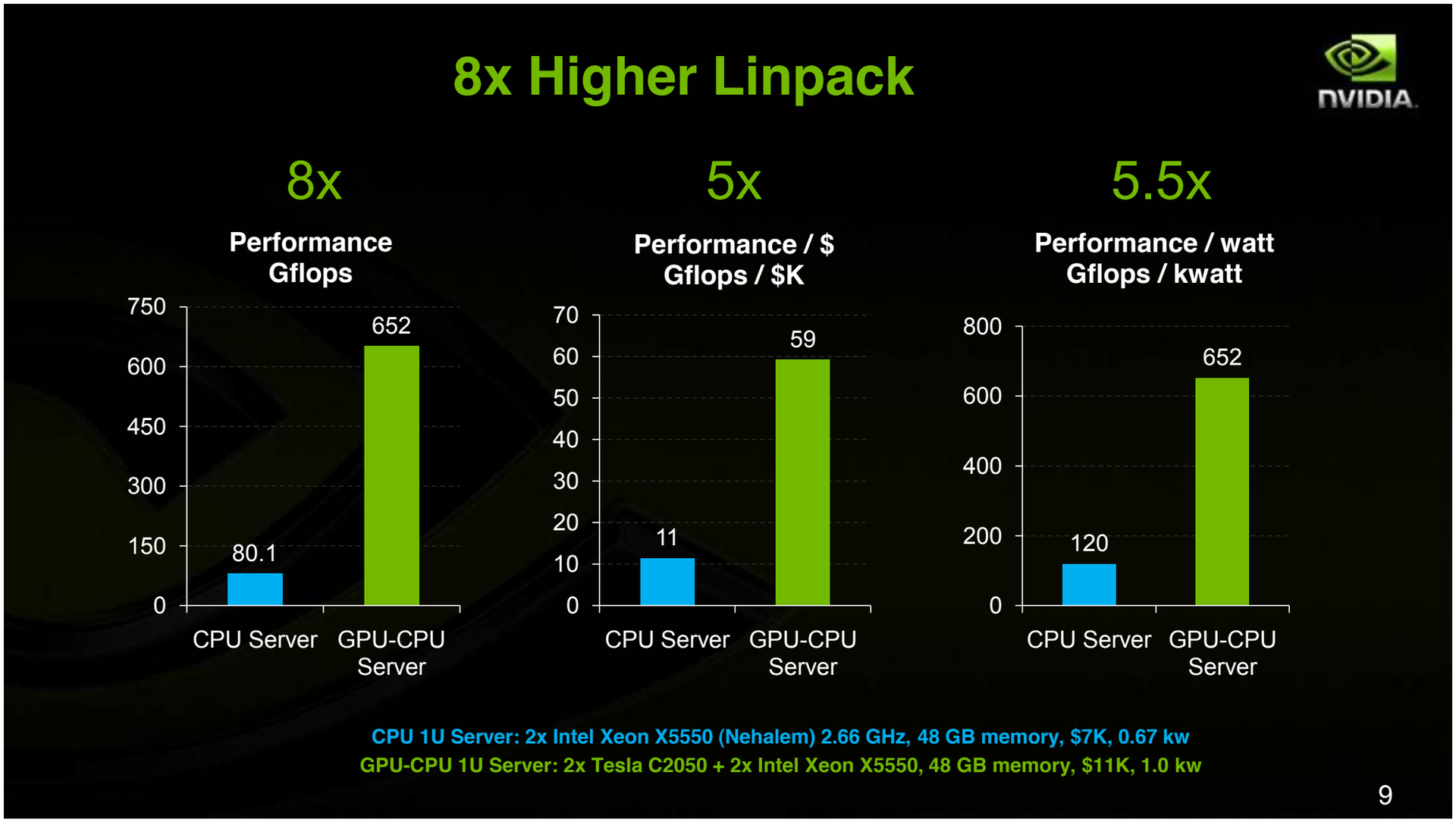
- 手軽になってきたGPU搭載ノードのクラスタ
- GPU(CUDA)プログラムを簡単に並列化したい
  ← CUDA中にMPIが書けると嬉しい
- ...けどどうしたらいい？処理性能は？

GPUを有する高性能計算機

Rank	Site	Computer
1	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C NUDT
2	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz Cray Inc.
3	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU Dawning
4	GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows NEC/HP
5	DOE/SC/LBNL/NERSC United States	Hopper - Cray XE6 12-core 2.1 GHz Cray Inc.
6	Commissariat a l'Energie Atomique (CEA) France	Tera-100 - Bull bullx super-node S6010/S6030 Bull SA
7	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband IBM
8	National Institute for Computational Sciences/University of Tennessee United States	Kraken XT5 - Cray XT5-HE Opteron 6-core 2.6 GHz Cray Inc.
9	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution IBM
10	DOE/NNSA/LANL/SNL United States	Cielo - Cray XE6 8-core 2.4 GHz Cray Inc.

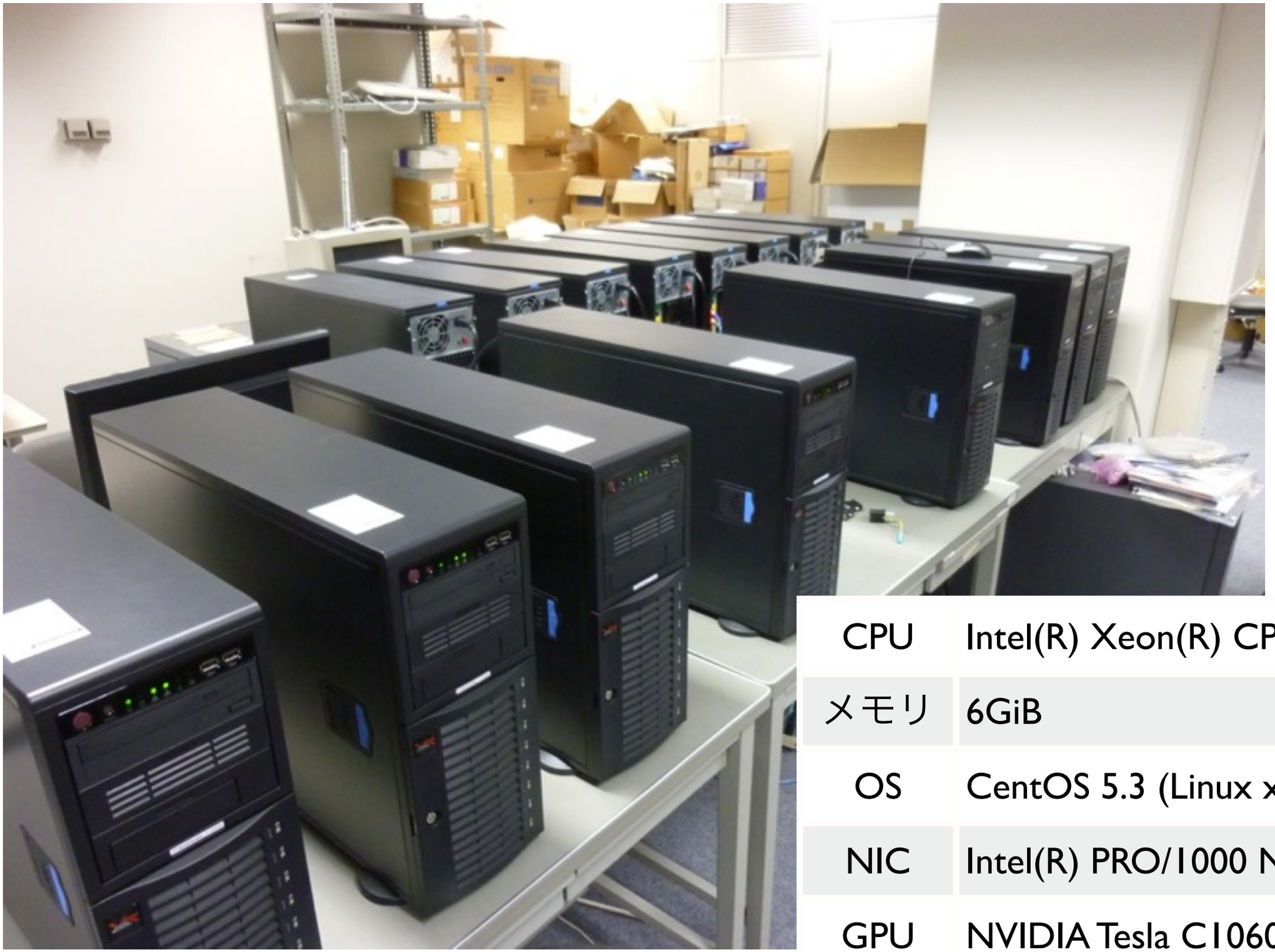
出典:TOP 500 November 2010 http://www.top500.org/lists/2010/11

GPU搭載計算機の価格



- TESLA/FireStream搭載機 ≒ 50万円〜
- C20x0 ≒ 30万円
- Quadro搭載計算機 ≒ 10万円〜

研究室でGPUクラスタ



CPU	Intel(R) Xeon(R) CPU W3520 2.7GHz
メモリ	6GiB
OS	CentOS 5.3 (Linux x86_64 2.6.18-128)
NIC	Intel(R) PRO/1000 NIC
GPU	NVIDIA Tesla C1060

GPU搭載計算機でのプログラミング

- 汎用プロセッサからGPUを制御

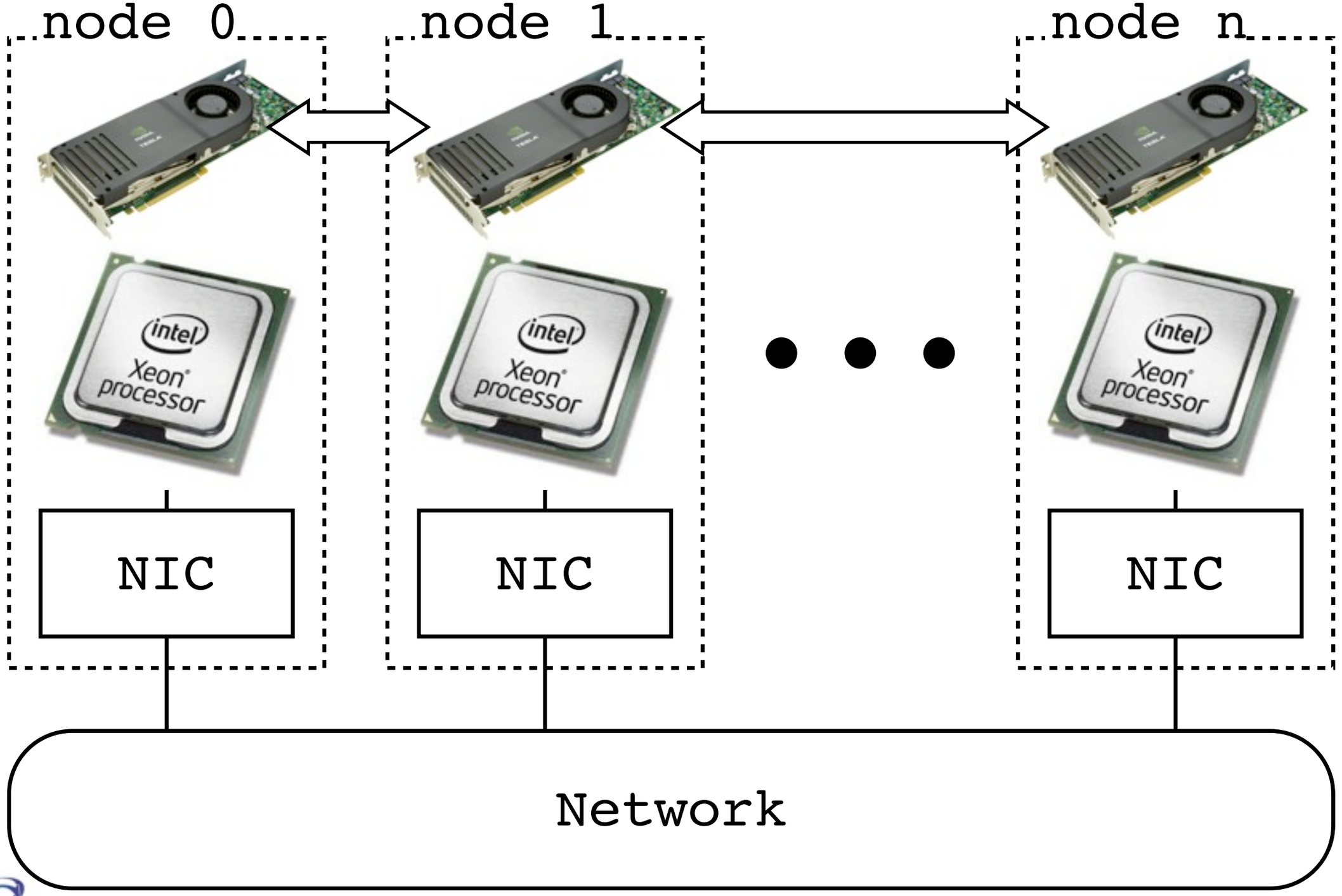
(1)データ転送

(2)コードの実行指示

(3)データ転送

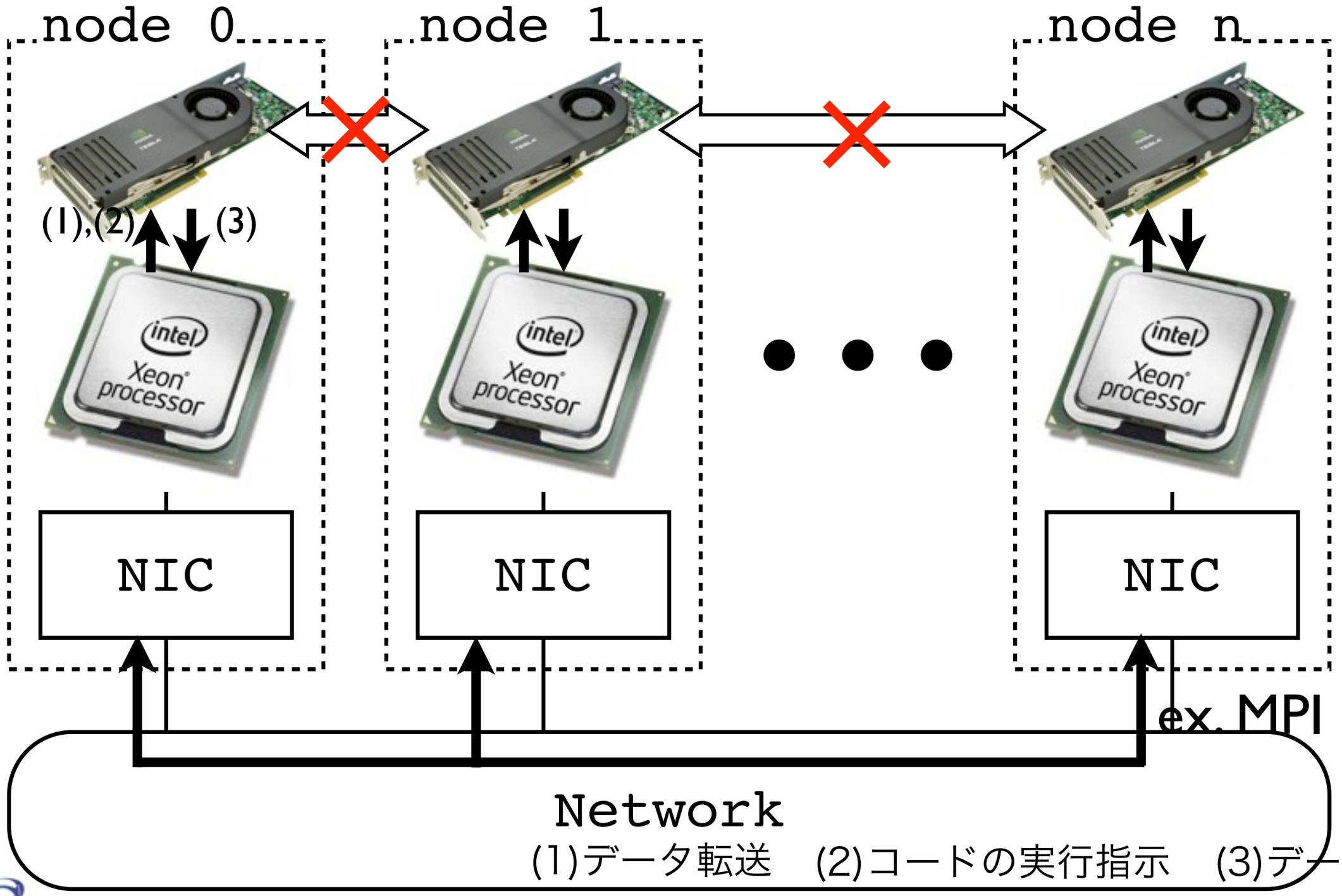
GPUクラスタでのプログラミング

- 複数のGPUで問題を分割して実行可能

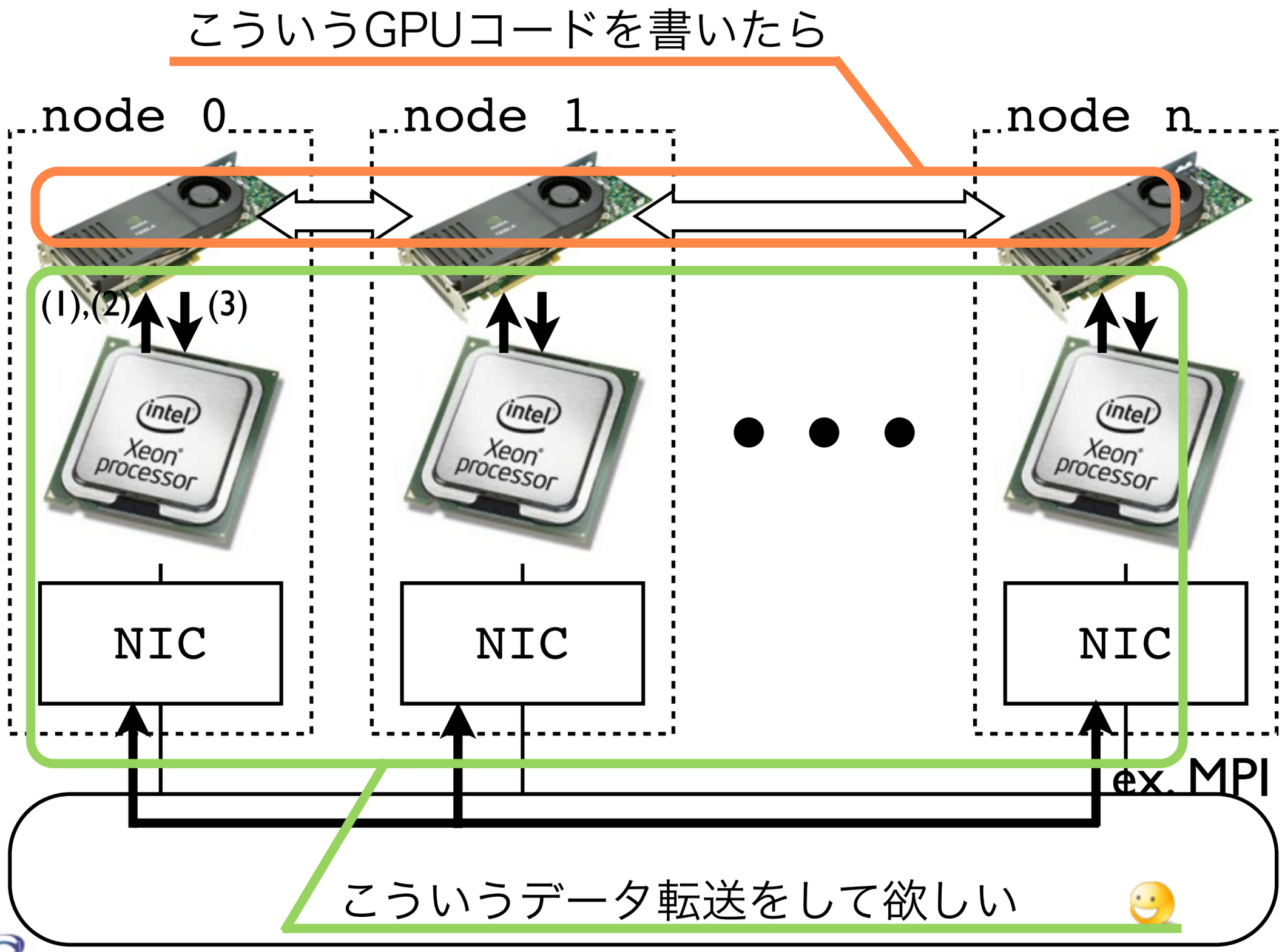


GPUクラスタでのプログラミング

- GPU同士はデータを直接やりとりできない

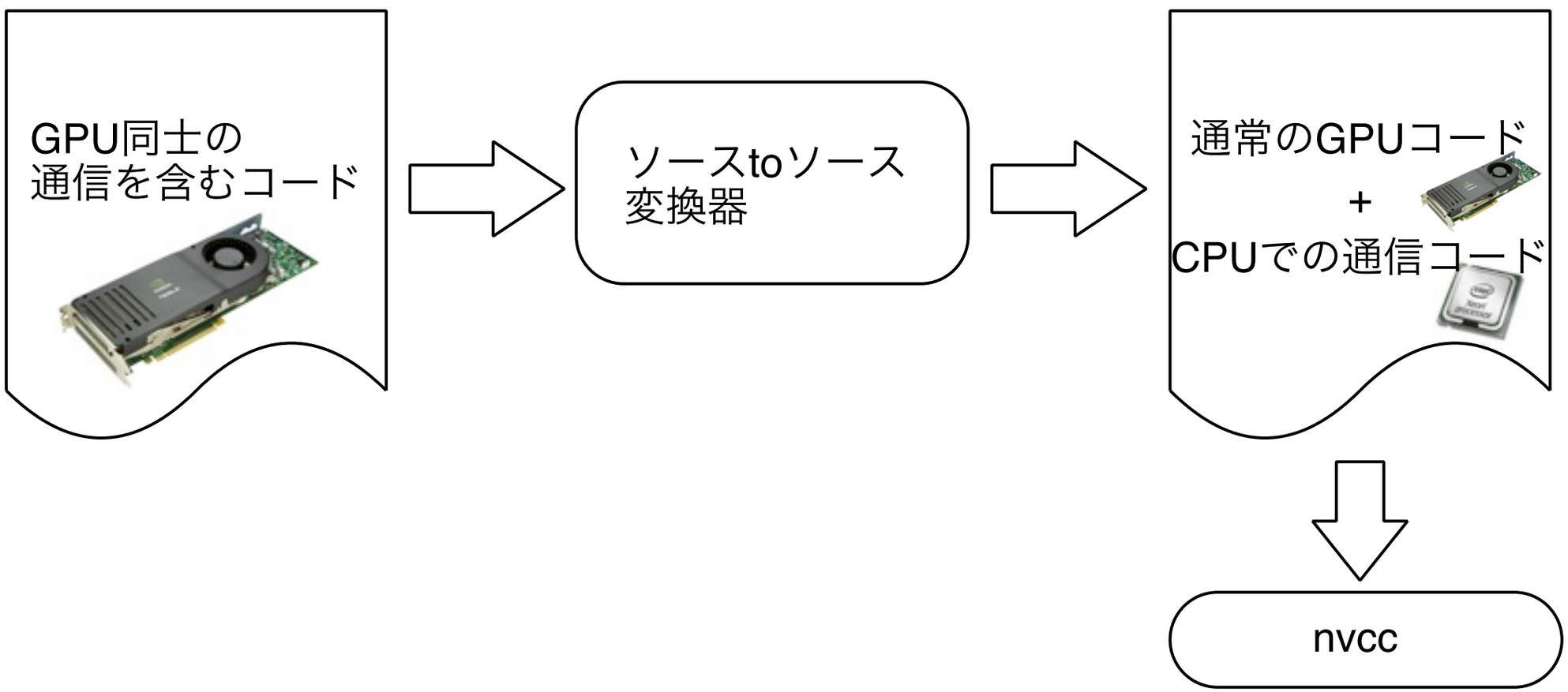


GPUクラスタでのプログラミング



GPU同士の通信を実現する方法

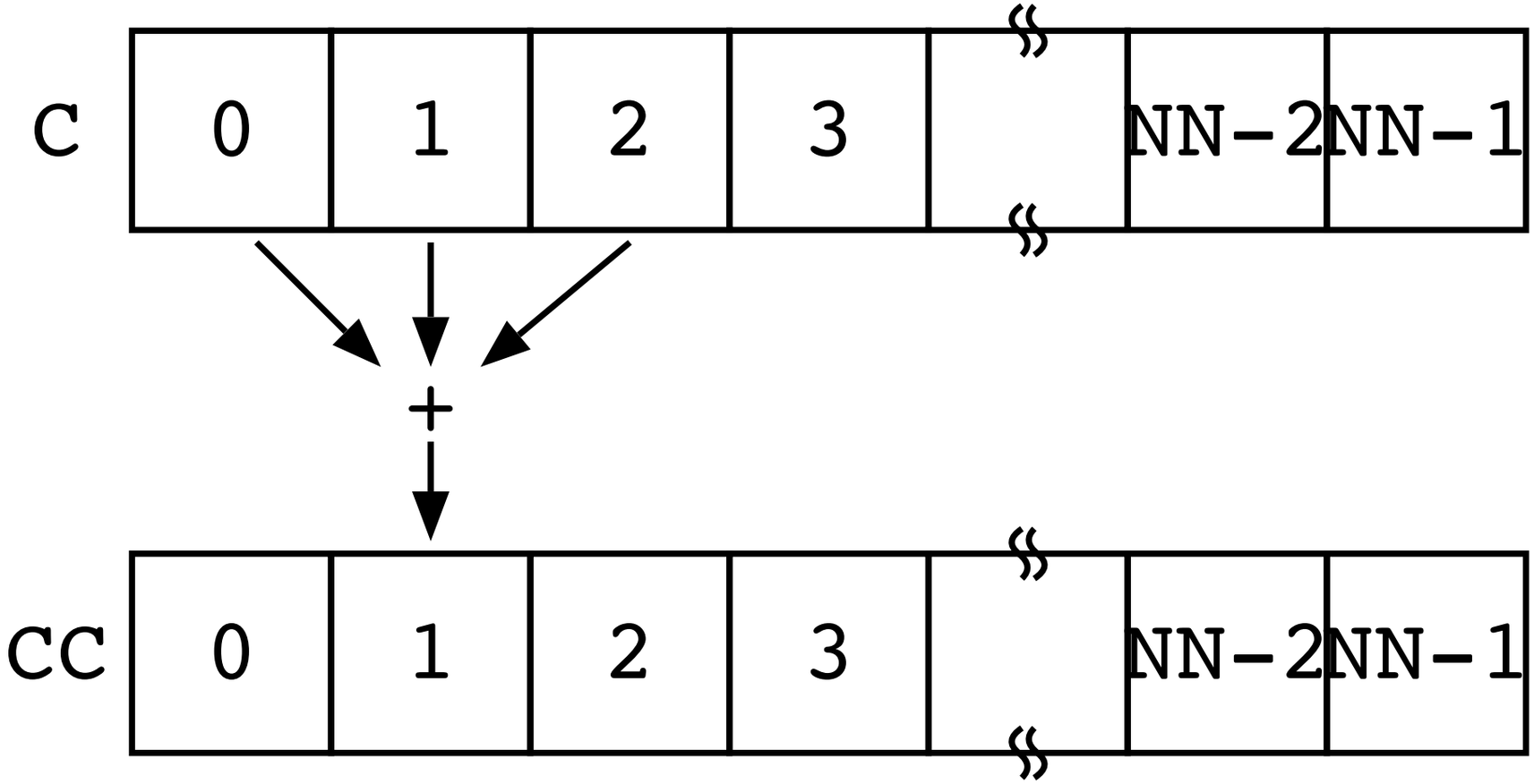
- ソース to ソース変換による実現



- 既存の環境がそのまま利用可能
- 機械的な変換で自動化可能(コンパイラ)
- 性能の低下はどの程度になるだろうか？

ケーススタディ

```
1:  for(i = 0; i < NN; i++){
2:    CC[i] = C[i];
3:    if(i != 0   ){ CC[i] += C[i-1]; }
4:    if(i != NN-1){ CC[i] += C[i+1]; }
5:    CC[i] = CC[i] / 3;
6:  }
```



ケーススタディ:GPUで実行

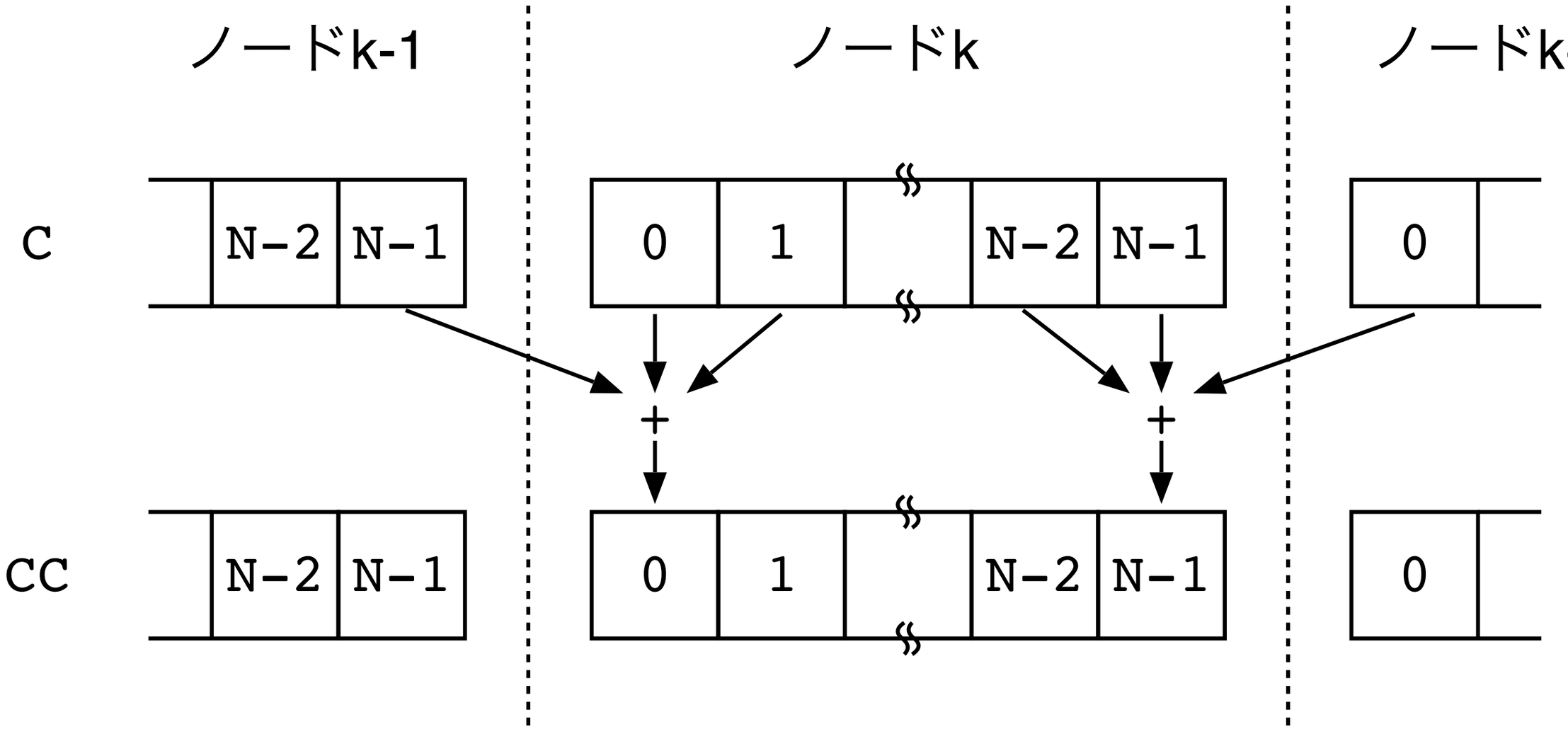
```
cudaMalloc((void **)&Cd0, sizeof(float) * (N+2));
cudaMalloc((void **)&Cd1, sizeof(float) * (N+2));
cudaMemcpy(&Cd[1], Ch, sizeof(float) * N, cudaMemcpyHostToDevice);
matmul<<<1, K>>>(Cd0, Cd1, N, N/K);
```

```
for(int i = 0; i < k; i++){
  int idx = threadIdx.x + i * K + 1;
  CC[idx] = (C[idx] + C[idx+1] + C[idx-1])/3;
}
```

＊K: GPUのスレッド数，k:N/K

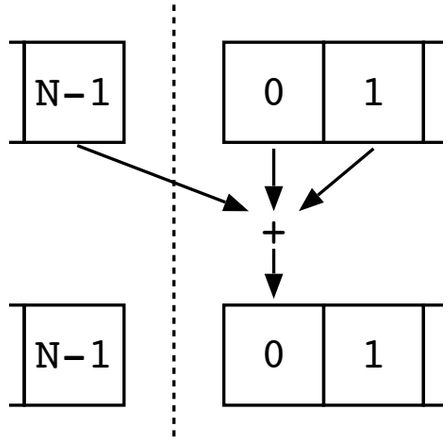
```
cudaMemcpy(Ch, &Cd1[1], sizeof(float) * N, cudaMemcpyDeviceToHost);
```

複数ノードで分割する場合



GPU上でのコード

```
 1: cudaMemcpy(&Cd0[1], Ch, sizeof(float) * N,
                                        cudaMemcpyHostToDevice);
 2: if(rank != nCPU-1)
 3:  MPI_Send(&Ch[N-1],1,MPI_FLOAT,rank+1,0,MPI_COMM_WORLD);
 4: if(rank != 0)
 5:  MPI_Recv(&v,1,MPI_FLOAT,rank-1,0,MPI_COMM_WORLD,&status);
 6: cudaMemcpy(&Cd0[0],&v,sizeof(float),
                                        cudaMemcpyHostToDevice);
 7: if(rank != 0)
 8:  MPI_Send(&Ch[0],1,MPI_FLOAT,rank-1,0,MPI_COMM_WORLD);
 9: if(rank != nCPU-1)
10:  MPI_Recv(&w,1,MPI_FLOAT,rank+1,0,MPI_COMM_WORLD,&status);
11: cudaMemcpy(&Cd0[N+1], &w, sizeof(float),
                                        cudaMemcpyHostToDevice);
12: kernel<<<1, K>>>(Cd0, Cd1, N, N/K);
```



```
 1: __global__ void
 2: kernel(float *C,float *CC,int N,int k) {
 3:   for(int i = 0; i < k; i++){
 4:     int idx = threadIdx.x + i * K + 1;
 5:     CC[idx] = (C[idx] + C[idx+1] + C[idx-1])/3;
 6: }}
```

```
13: cudaMemcpy(Ch, &Cd1[1], sizeof(float) * N,
                                        cudaMemcpyDeviceToHost);
```

CUDAにMPIが埋め込み可能であれば...

```
1:   cudaMemcpy(&Cd0[1], Ch, sizeof(float) * N,
                                  cudaMemcpyHostToDevice);
2:   kernel<<<1, K>>>(Cd0, Cd1, rank, nCPU, N, N/K);
```

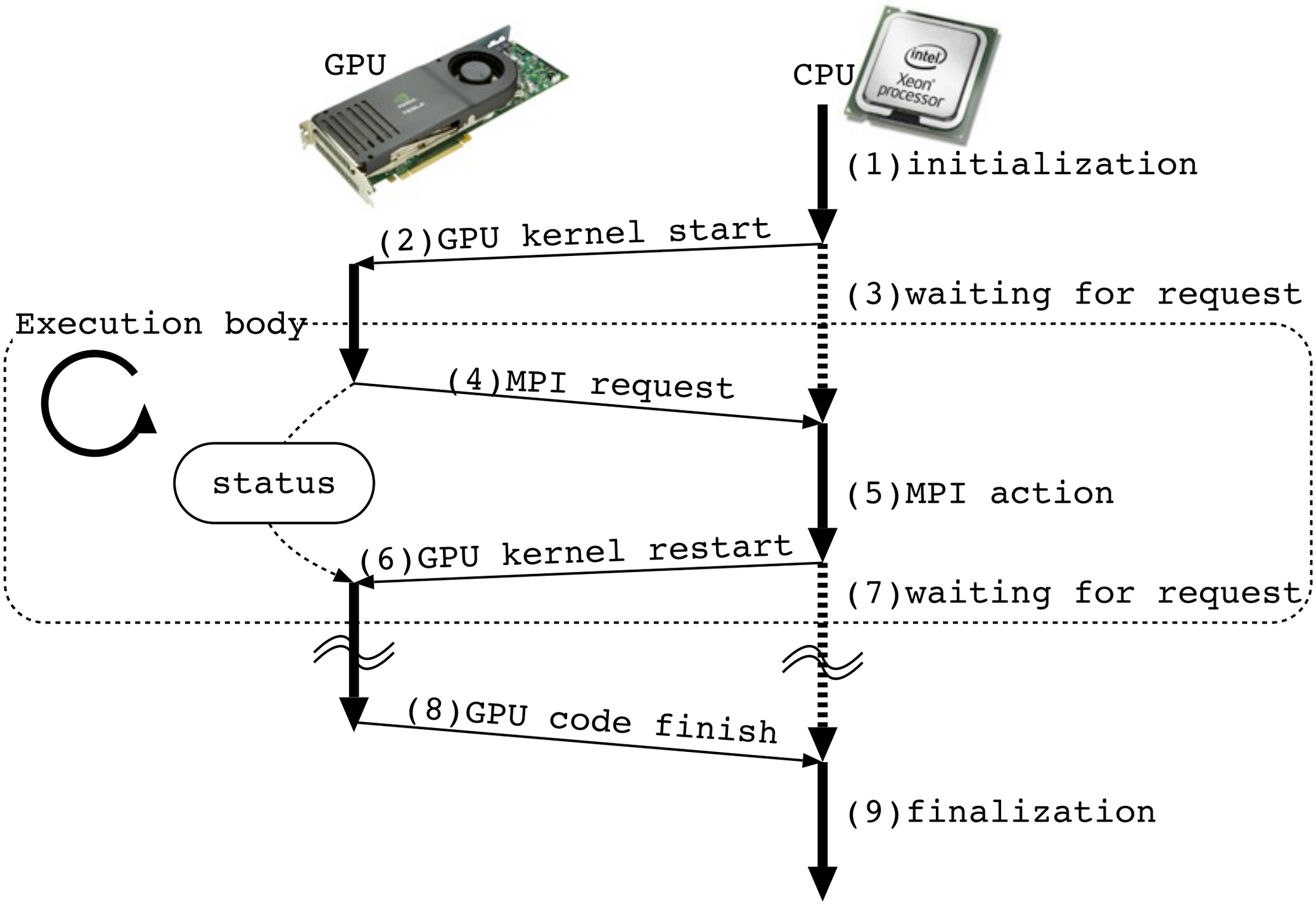
```
 1: __global__ void
 2: kernel(float *C,float *CC,int rank,int nCPU,int N,int k){
 3:   if(rank != nCPU-1)
 4:     cuda_mpi_send(&C[N], sizeof(float)*1, rank+1);
 5:   if(rank != 0)
 6:     cuda_mpi_send(&C[1], sizeof(float)*1, rank-1);
 7:   if(rank != 0)
 8:     cuda_mpi_recv(&C[0], sizeof(float)*1, rank-1);
 9:   if(rank != nCPU-1)
10:     cuda_mpi_recv(&C[N+1], sizeof(float)*1, rank+1);
11:   for(int i = 0; i < k; i++){
12:     int idx = threadIdx.x + i * K + 1;
13:     CC[idx] = (C[idx] + C[idx+1] + C[idx-1])/3;
14: }}
```

直観的

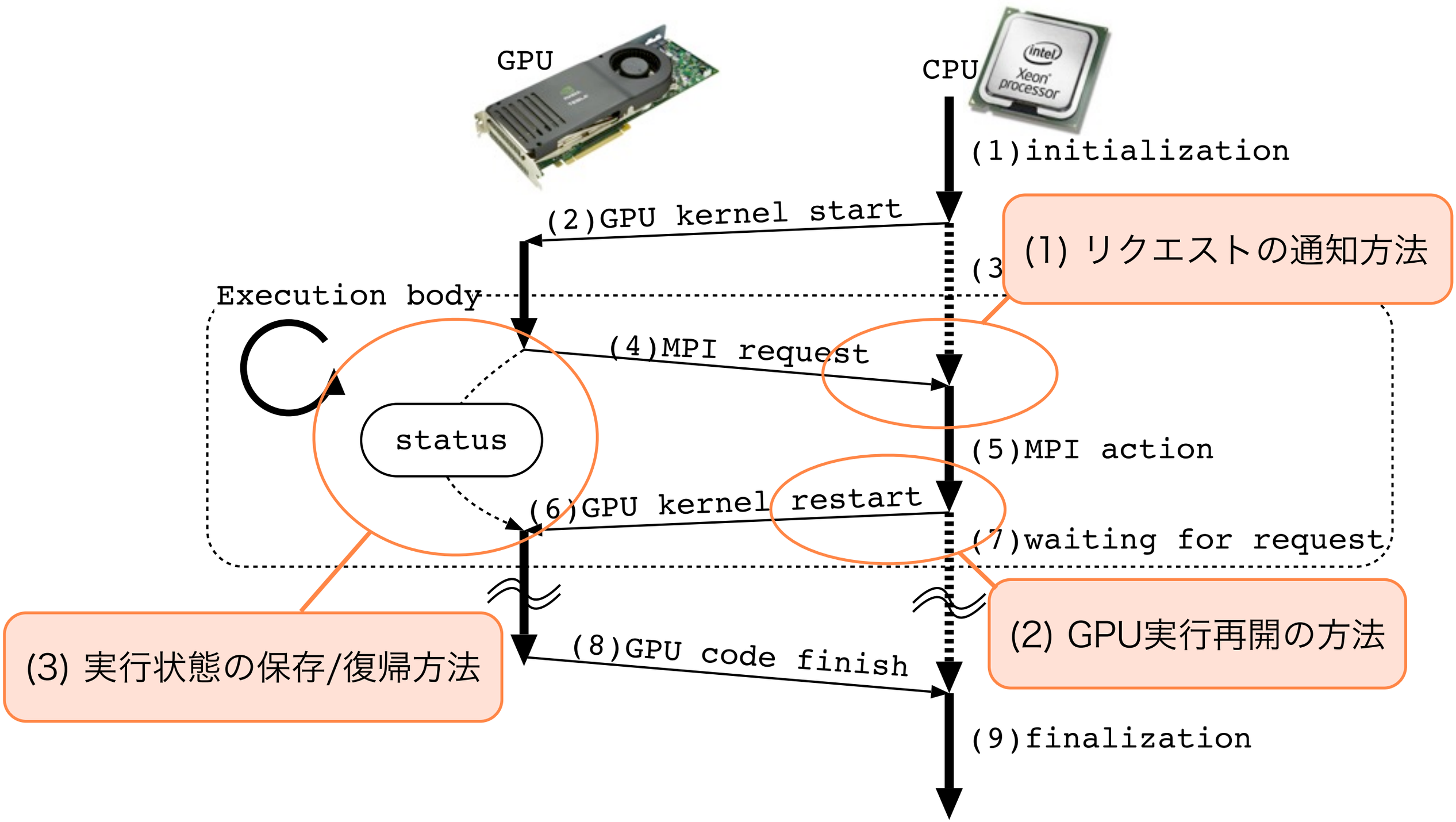
```
3:   cudaMemcpy(Ch, &Cd1[1], sizeof(float) * N,
                                  cudaMemcpyDeviceToHost);
```

どうやって実現するか？

実行モデル


GPU
CPU
(1)initialization
(2)GPU kernel start
(3)waiting for request
Execution body
(4)MPI request
status
(5)MPI action
(6)GPU kernel restart
(7)waiting for request
(8)GPU code finish
(9)finalization

実行モデル実現のための課題



実行モデルの実現手法

- **リクエストの通知方法**
  - GPUコードを終了させてしまう
  - ステータス変数にリクエストをセット
- **GPU実行再開の方法**
  - GPUカーネルコードを再度呼び出す
- **実行状態の保存/復帰**
  - 実行箇所はswitch-caseのラベルで保存/復帰
  - 実行箇所/途中結果はデバイスメモリで待避/復帰

実行モデルの実現手法

- **リクエストの通知方法**
  - GPUコードを終了させてしまう
  - ステータス変数にリクエストをセット
- **GPU実行再開の方法**
  - GPUカーネルコードを再度呼び出す
- **実行状態の保存/復帰**
  - 実行箇所はswitch-caseのラベルで保存/復帰
  - 実行箇所/途中結果はデバイスメモリで待避/復帰

実行箇所の保存/復帰の実現手法

- MPIコードが埋め込まれたCUDAコード
  （＝対象となるユーザが記述するコード）

```
1: __global__ void
2: kernel(float *C,float *CC,int rank,int nCPU,int N,int k){
3:   if(rank != nCPU-1)
4:     cuda_mpi_send(&C[N], sizeof(float)*1, rank+1);
5:   if(rank != 0)
6:     cuda_mpi_send(&C[1], sizeof(float)*1, rank-1);
7:   if(rank != 0)
8:     cuda_mpi_recv(&C[0], sizeof(float)*1, rank-1);
9:   if(rank != nCPU-1)
10:    cuda_mpi_recv(&C[N+1], sizeof(float)*1, rank+1);
11:  for(int i = 0; i < k; i++){
12:    int idx = threadIdx.x + i * K + 1;
13:    CC[idx] = (C[idx] + C[idx+1] + C[idx-1])/3;
14: }}
```

実行箇所の保存/復帰の実現手法

- MPI処理のとこでGPU処理を中断したい!!

```
 1: __global__ void
 2: kernel(float *C,float *CC,int rank,int nCPU,int N,int k){
 3:   if(rank != nCPU-1)
 4:     cuda_mpi_send(&C[N], sizeof(float)*1, rank+1);
 5:   if(rank != 0)
 6:     cuda_mpi_send(&C[1], sizeof(float)*1, rank-1);
 7:   if(rank != 0)
 8:     cuda_mpi_recv(&C[0], sizeof(float)*1, rank-1);
 9:   if(rank != nCPU-1)
10:     cuda_mpi_recv(&C[N+1], sizeof(float)*1, rank+1);
11:   for(int i = 0; i < k; i++){
12:     int idx = threadIdx.x + i * K + 1;
13:     CC[idx] = (C[idx] + C[idx+1] + C[idx-1])/3;
14: }}
```

実際にはCPUが処理を行う箇所

＝一度GPUコードを終了する場所

実行箇所の保存/復帰のための追加コード片

```
  1: __global__ void
  2: kernel(float *C,float *CC,int rank,int nCPU,int N,int k){
* 3:  restore();
  4:  switch(*_status) {
* 5:   case 0:
  6:   if(rank != nCPU-1){
  7:     cuda_mpi_send(&C[N], sizeof(float)*1, rank+1);
+ 8:     *_status = 1; store(); goto GPU_CODE_END; }
* 9:   case 1:
 10:   if(rank != 0){
 11:     cuda_mpi_send(&C[1], sizeof(float)*1, rank-1);
+12:     *_status = 2; store(); goto GPU_CODE_END; }
*13:   case 2:
 14:   if(rank != 0){
 15:     cuda_mpi_recv(&C[0], sizeof(float)*1, rank-1);
+16:     *_status = 3; store(); goto GPU_CODE_END; }
*17:   case 3:
 18:   if(rank != nCPU-1){
 19:     cuda_mpi_recv(&C[N+1], sizeof(float)*1, rank+1);
+20:     *_status = 4; sotre(); goto GPU_CODE_END; }
*21:   case 4:
 22:   for(int i = 0; i < k; i++){
 23:     int idx = threadIdx.x + i * K + 1;
 24:     CC[idx] = (C[idx] + C[idx+1] + C[idx-1])/3;
 25:   }}
 26:  cuda_mpi_finalize();
+27:  GPU_CODE_END:
 28: }
```

実行箇所の保存/復帰のための追加コード片

```
  1: __global__ void
  2: kernel(float *C,float *CC,int rank,int nCPU,int N,int k){
* 3:  restore();
  4:  switch(*_status) {
* 5:   case 0:
  6:   if(rank != nCPU-1){
  7:     cuda_mpi_send(&C[N], sizeof(float)*1, rank+1);
+ 8:     *_status = 1; store(); goto GPU_CODE_END; }
* 9:   case 1:
 10:   if(rank != 0){
 11:     cuda_mpi_send(&C[1], sizeof(float)*1, rank-1);
+12:     *_status = 2; store(); goto GPU_CODE_END; }
*13:   case 2:
 14:   if(rank != 0){
 15:     cuda_mpi_recv(&C[0], sizeof(float)*1, rank-1);
+16:     *_status = 3; store(); goto GPU_CODE_END; }
*17:   case 3:
 18:   if(rank != nCPU-1){
 19:     cuda_mpi_recv(&C[N+1], sizeof(float)*1, rank+1);
+20:     *_status = 4; sotre(); goto GPU_CODE_END; }
*21:   case 4:
 22:   for(int i = 0; i < k; i++){
 23:     int idx = threadIdx.x + i * K + 1;
 24:     CC[idx] = (C[idx] + C[idx+1] + C[idx-1])/3;
 25:   }}
 26:  cuda_mpi_finalize();
+27:  GPU_CODE_END:
 28: }
```

実行箇所の保存/復帰のための追加コード片

```
  1: __global__ void
  2: kernel(float *C,float *CC,int rank,int nCPU,int N,int k){
* 3:  restore();
  4:  switch(*_status) {
* 5:   case 0:
  6:   if(rank != nCPU-1){
  7:     cuda_mpi_send(&C[N], sizeof(float)*1, rank+1);
+ 8:     *_status = 1; store(); goto GPU_CODE_END; }
* 9:   case 1:
 10:   if(rank != 0){
 11:     cuda_mpi_send(&C[1], sizeof(float)*1, rank-1);
+12:     *_status = 2; store(); goto GPU_CODE_END; }
*13:   case 2:
 14:   if(rank != 0){
 15:     cuda_mpi_recv(&C[0], sizeof(float)*1, rank-1);
+16:     *_status = 3; store(); goto GPU_CODE_END; }
*17:   case 3:
 18:   if(rank != nCPU-1){
 19:     cuda_mpi_recv(&C[N+1], sizeof(float)*1, rank+1);
+20:     *_status = 4; sotre(); goto GPU_CODE_END; }
*21:   case 4:
 22:   for(int i = 0; i < k; i++){
 23:     int idx = threadIdx.x + i * K + 1;
 24:     CC[idx] = (C[idx] + C[idx+1] + C[idx-1])/3;
 25:   }}
 26:  cuda_mpi_finalize();
+27:  GPU_CODE_END:
 28: }
```

ホストCPUでのリクエスト処理ルーチン

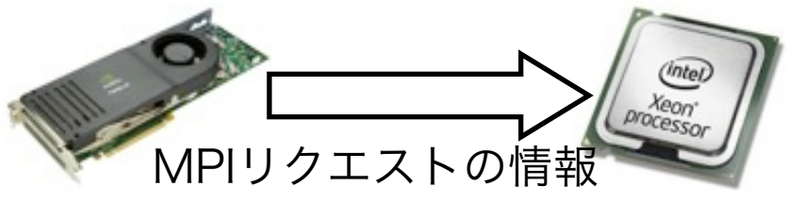
- CPU側でのリクエストハンドラ


```
1:  do{
2:    kernel_stub<<<1, N>>>
        (Cd, rank, nCPU, info->info_dev, info->status_dev);
3:    cudaMemcpy((void*)info->info_host,
                 (void*)info->info_dev,
                 sizeof(gpu_info),
                 cudaMemcpyDeviceToHost);
4:    switch(info->info_host->request){
5:    case MPI_SEND:
6:      cuda_mpi_send((gpu_info*)info->info_host); break;
7:    case MPI_RECV:
8:      cuda_mpi_recv((gpu_info*)info->info_host); break;
9:    case MPI_BARRIER:
10:     MPI_Barrier(MPI_COMM_WORLD); break;
11:   }
12: }while(info->info_host->request != DONE);
```

ホストCPUでのリクエスト処理ルーチン

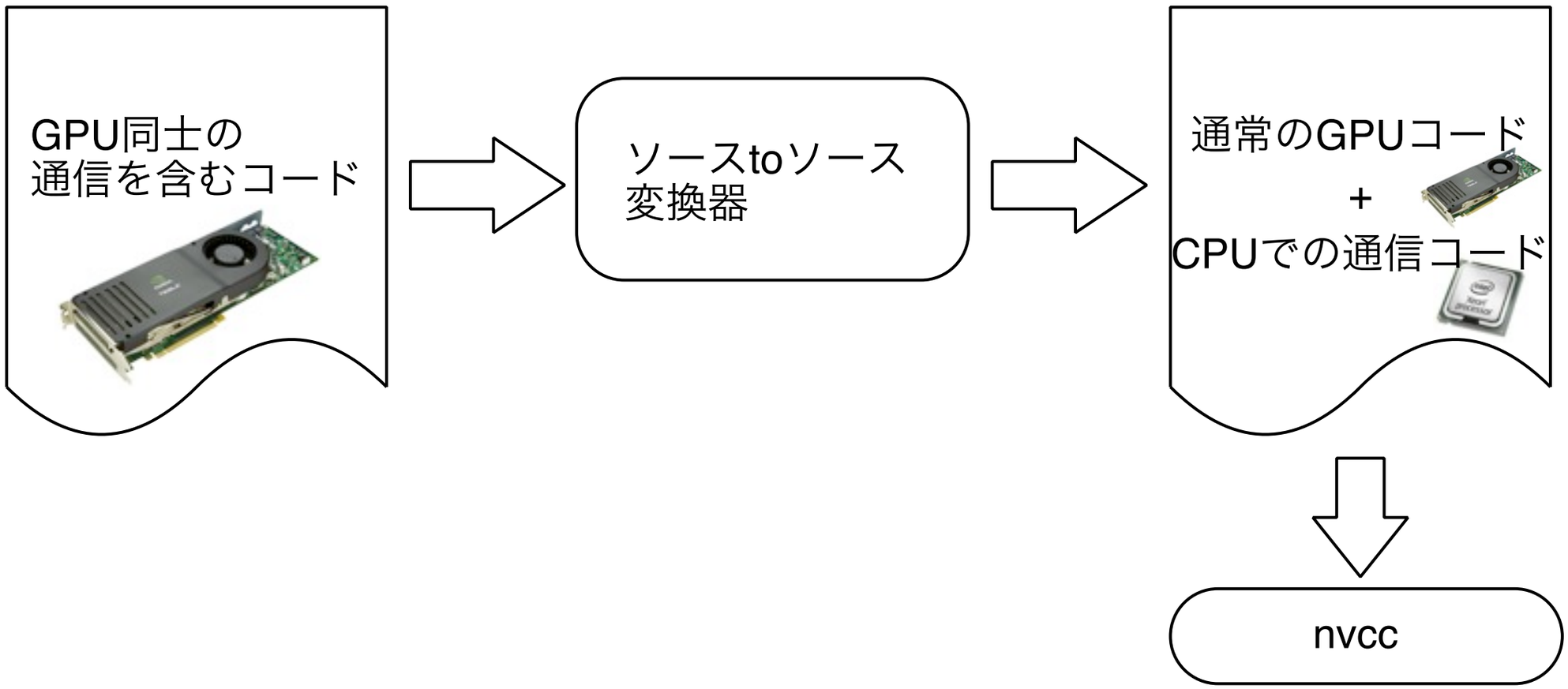
- CPU側でのリクエストハンドラ


```
1:  do{
2:    kernel_stub<<<1, N>>>
        (Cd, rank, nCPU, info->info_dev, info->status_dev);
3:    cudaMemcpy((void*)info->info_host,
                 (void*)info->info_dev,
                 sizeof(gpu_info),
                 cudaMemcpyDeviceToHost);
4:    switch(info->info_host->request){
5:    case MPI_SEND:
6:      cuda_mpi_send((gpu_info*)info->info_host); break;
7:    case MPI_RECV:
8:      cuda_mpi_recv((gpu_info*)info->info_host); break;
9:    case MPI_BARRIER:
10:     MPI_Barrier(MPI_COMM_WORLD); break;
11:   }
12: }while(info->info_host->request != DONE);
```



gpu_info構造体

プログラマの負担軽減のために...



- GPU処理の保存/復帰のための追加コードを挿入
  （計算の実行箇所と途中結果を保存）
- forループのwhileループへの変換

シングルモード v.s. マルチモード

- シングルモード
  - gpu_infoが一つだけ
    ＝GPUが保存できる状態は一つだけ
  - GPU全スレッドの中断/再開箇所が同じ
- マルチモード
  - 複数のgpu_infoをもつ
  - 各スレッド毎の中断/再会箇所をもてる

シングルモード v.s. マルチモード

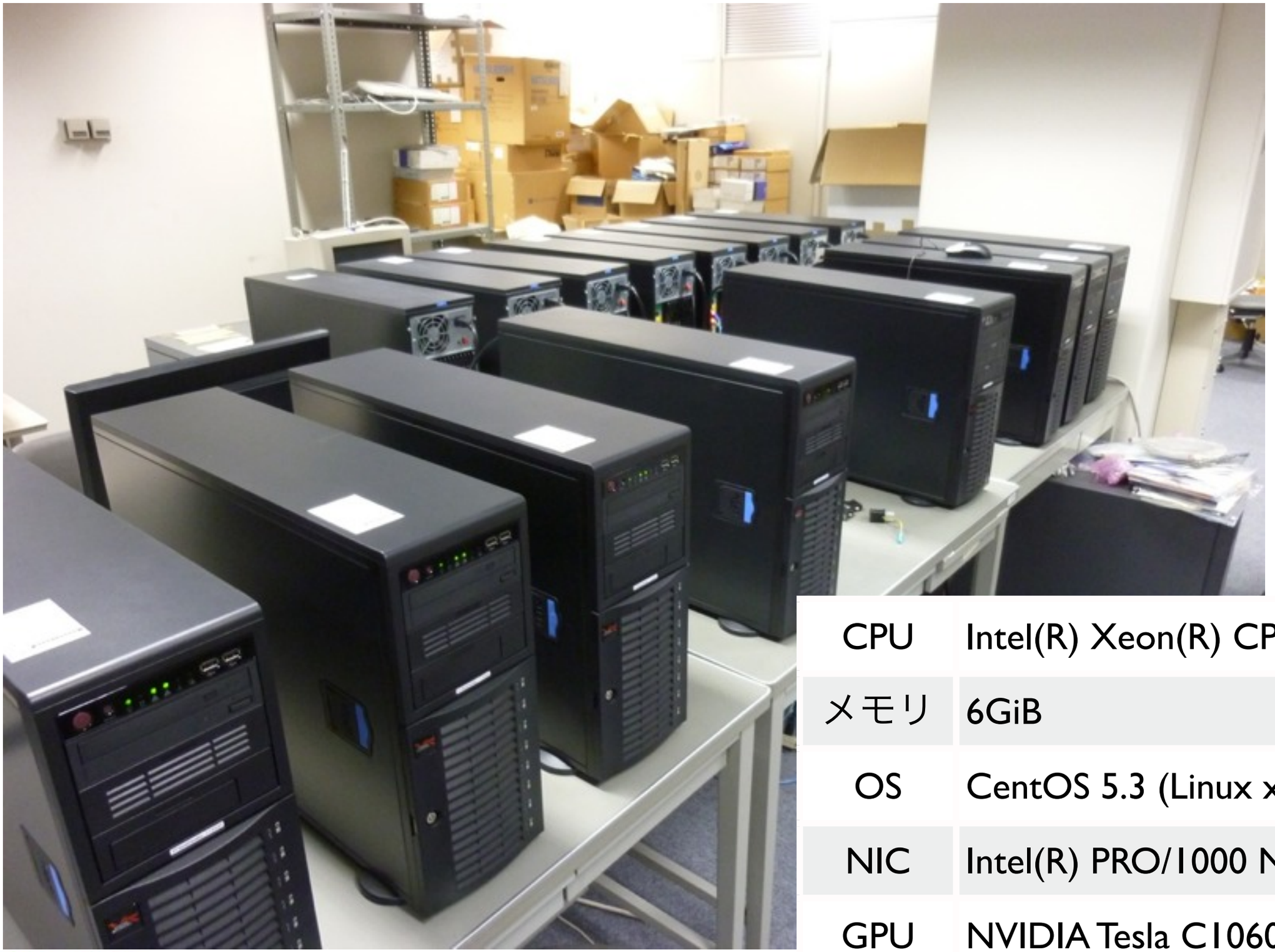
- シングルモード
  - gpu_infoが一つだけ
    ＝GPUが保存できる状態は一つだけ
  - GPU全スレッドの中断/再開箇所が同じ
- マルチモード
  - 複数のgpu_infoをもつ
  - 各スレッド毎の中断/再会箇所をもてる

パフォーマンスは？

性能評価

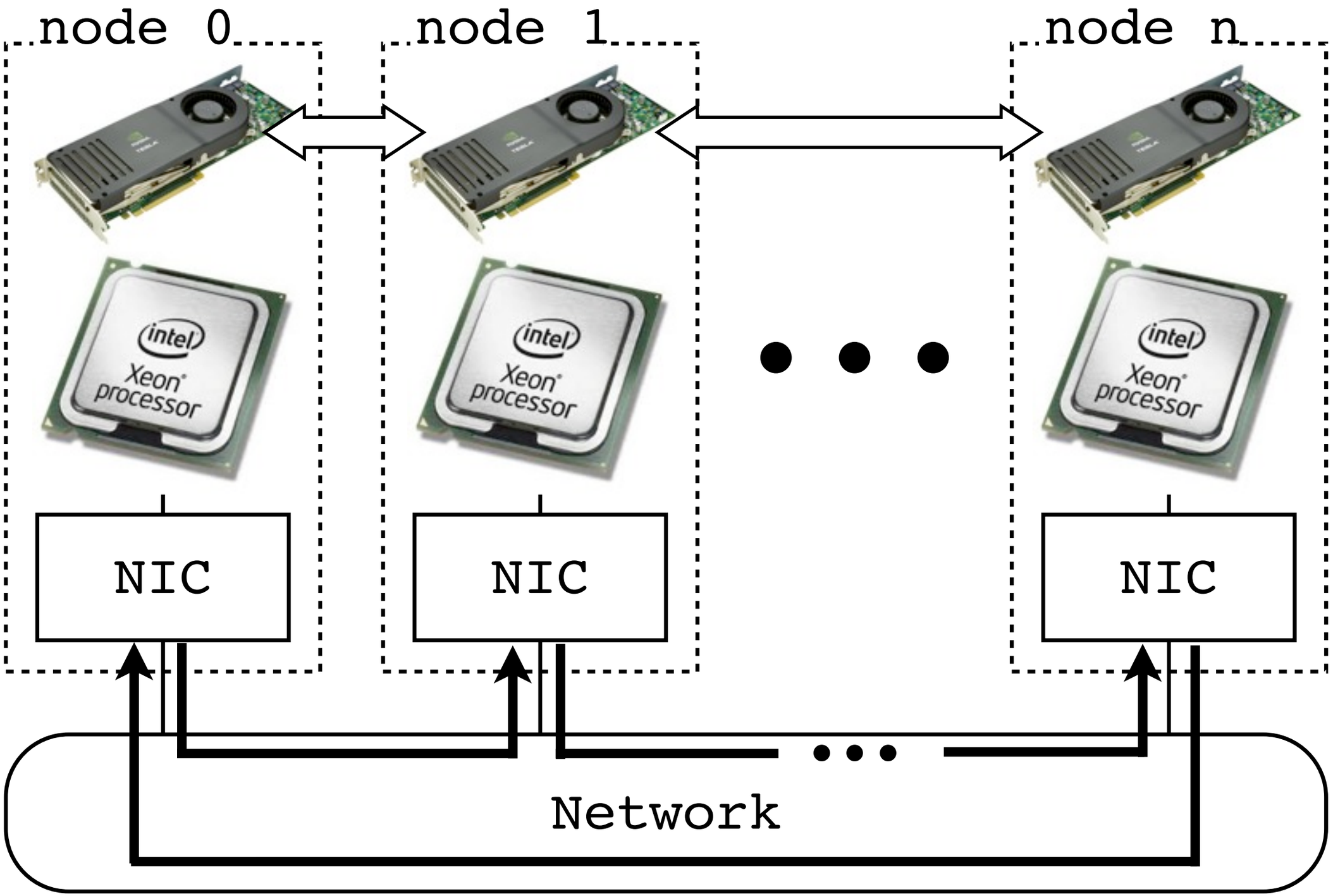
- **MPI処理にかかるオーバヘッド**
  - GPUカーネル実行/終了/再実行のコスト
  - CPUのビジーループなハンドリング性能
- **プログラム実行時間の評価**
  - 隣接要素の平均をとるプログラム
  - 台数効果が得られるか？
  - べたに書いたばあいとの性能差

評価環境



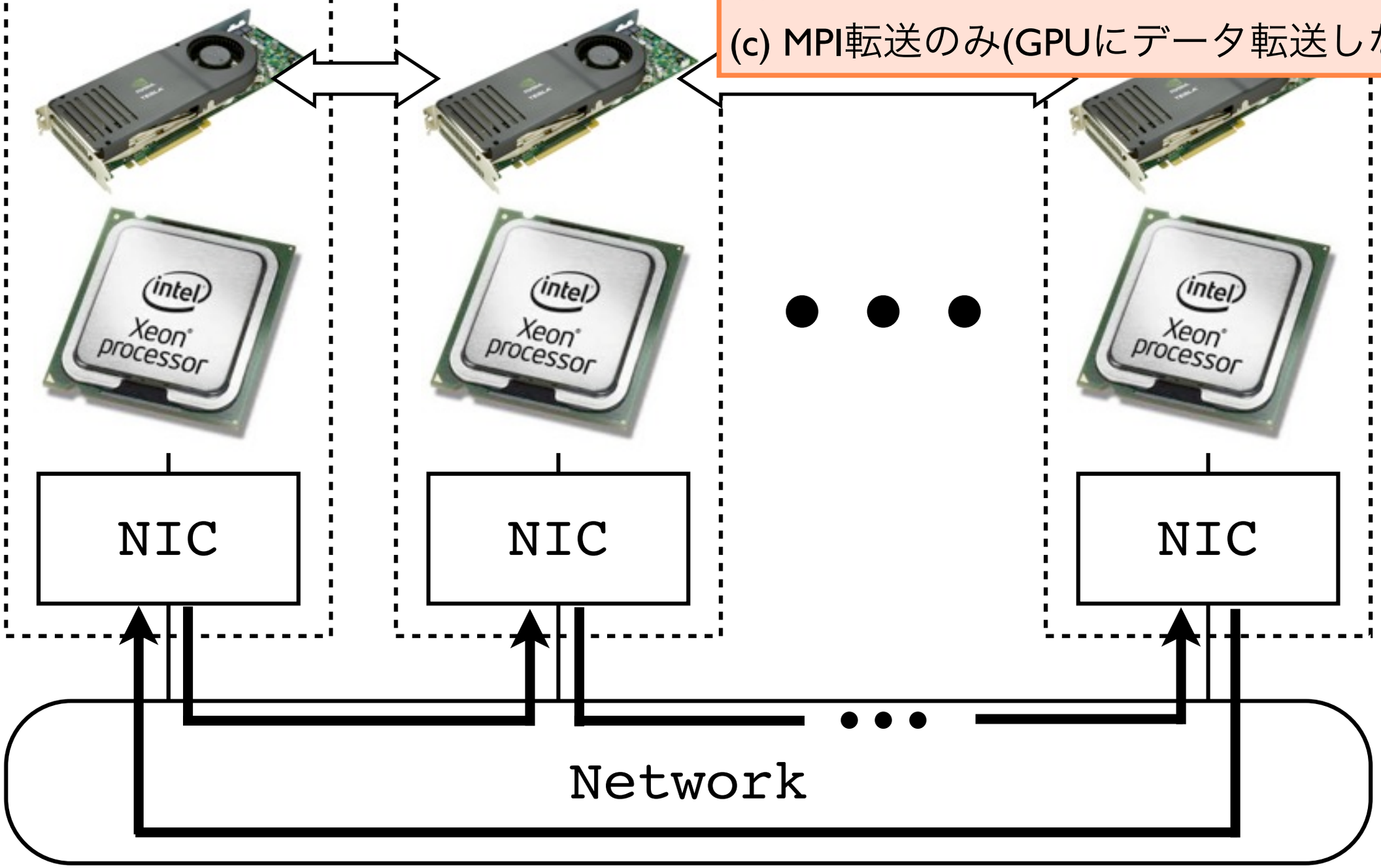
CPU	Intel(R) Xeon(R) CPU W3520 2.7GHz
メモリ	6GiB
OS	CentOS 5.3 (Linux x86_64 2.6.18-128)
NIC	Intel(R) PRO/1000 NIC
GPU	NVIDIA Tesla C1060

MPI転送オーバヘッドの評価

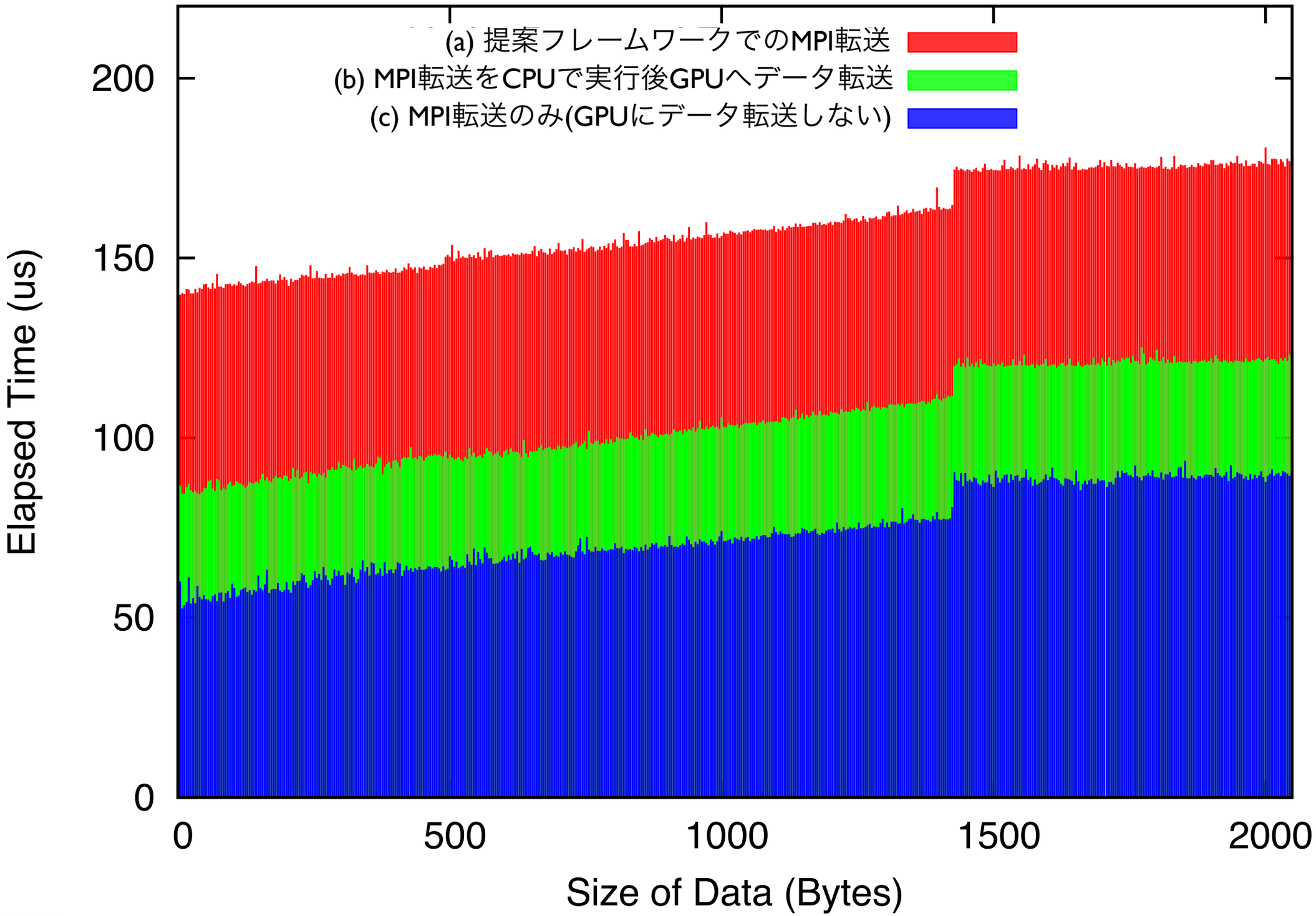


MPI転送オーバヘッドの評価

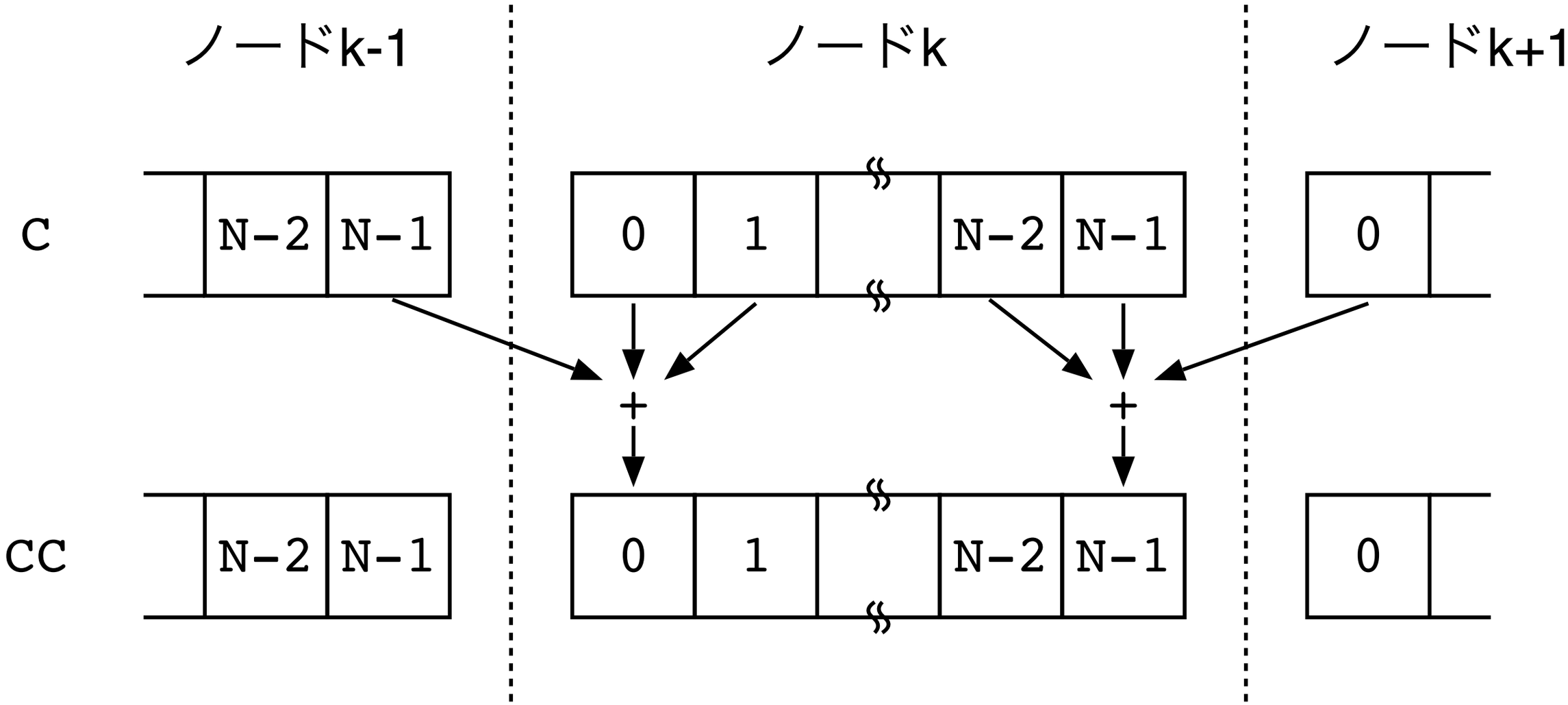
(a) 提案フレームワークでのMPI転送
(b) MPI転送をCPUで実行後GPUへデータ転送
(c) MPI転送のみ(GPUにデータ転送しない)



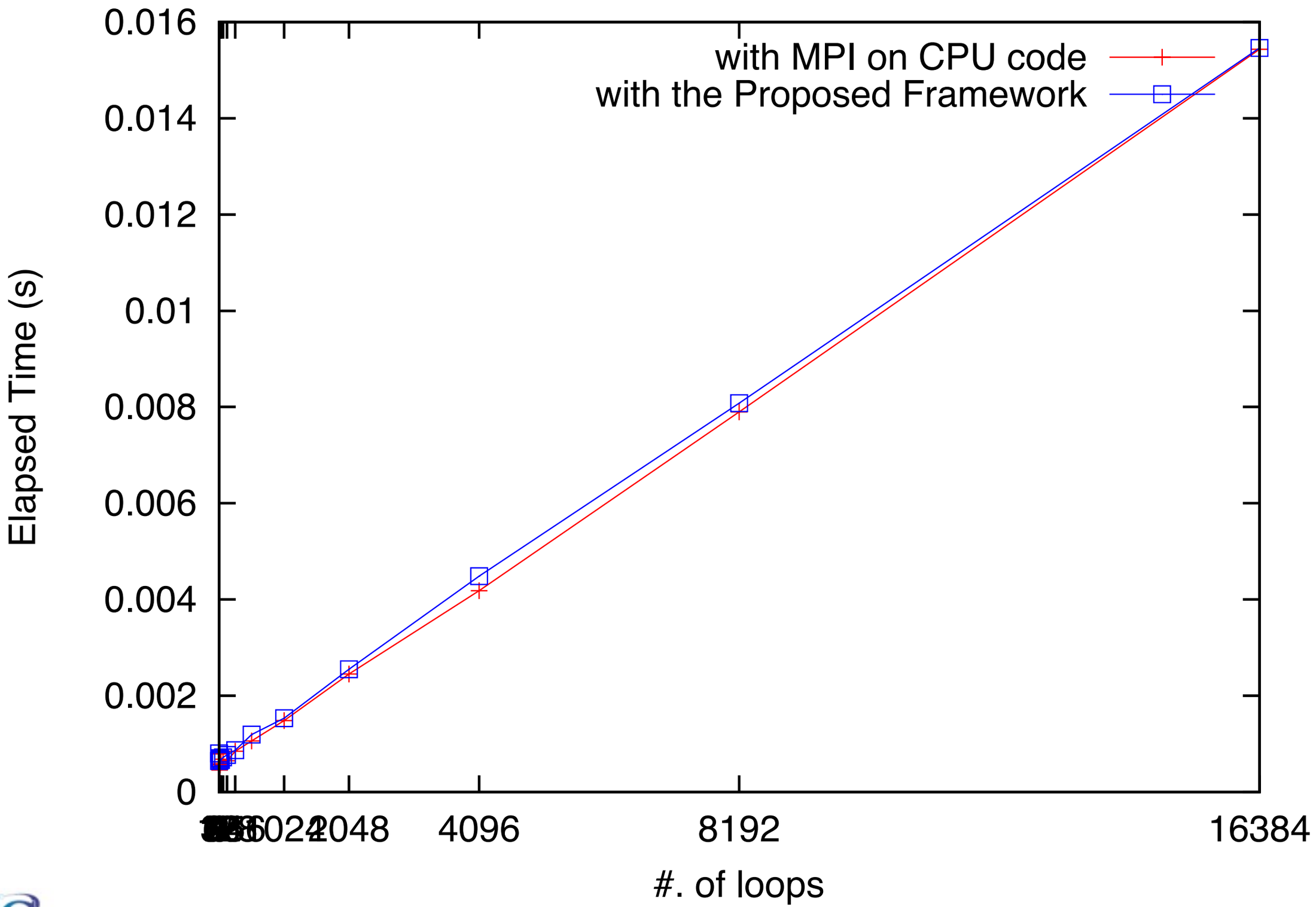
MPI転送オーバヘッドの評価



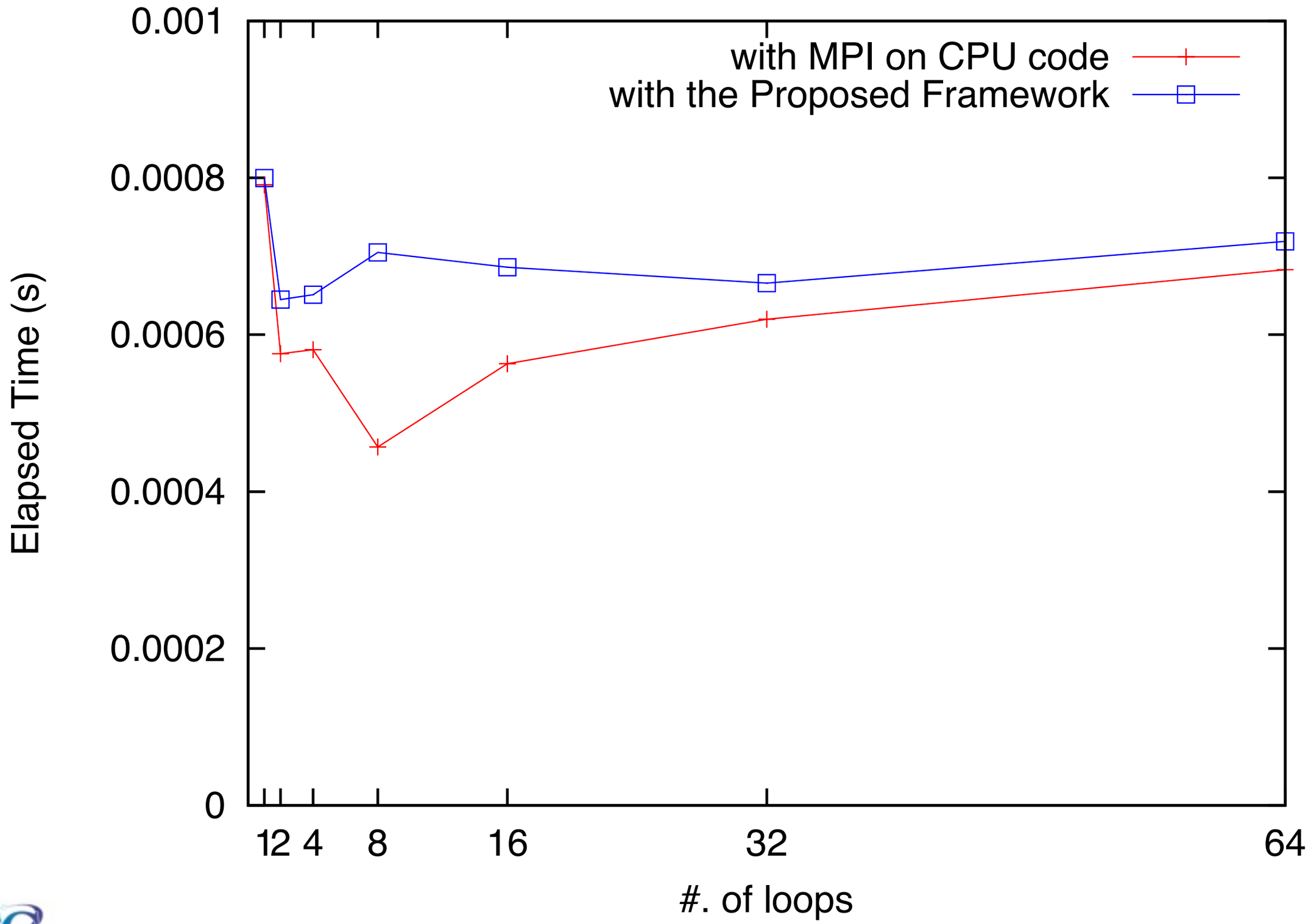
プログラムの実行性能



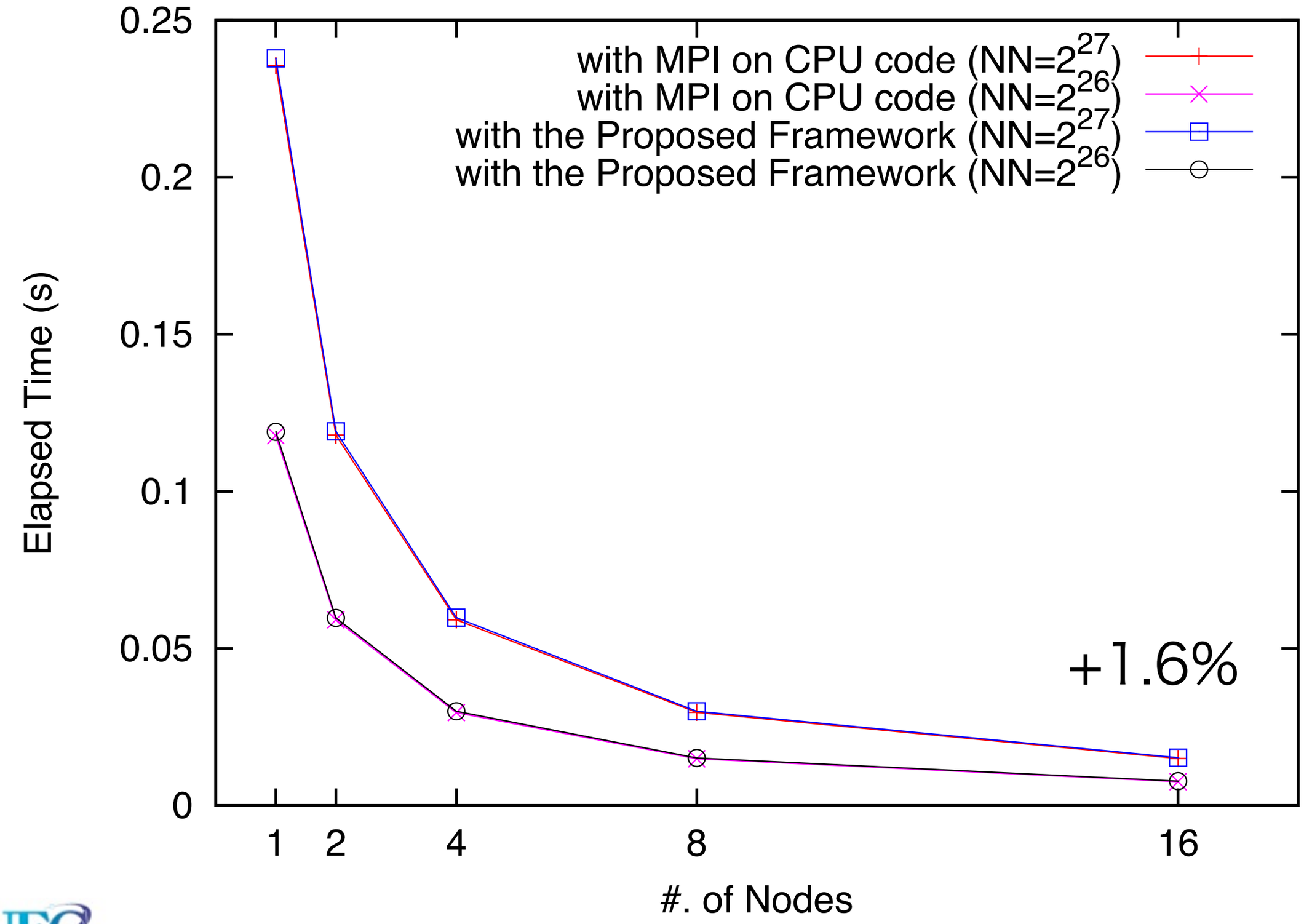
ループ数 v.s. 実行時間


with MPI on CPU code
with the Proposed Framework
Elapsed Time (s)
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
2048
4096
8192
16384
#. of loops

ループ数 v.s. 実行時間


0.001
0.0008
0.0006
0.0004
0.0002
0
Elapsed Time (s)
with MPI on CPU code
with the Proposed Framework
1
2
4
8
16
32
64
#. of loops

実行速度の比較


with MPI on CPU code (NN=2^27)
with MPI on CPU code (NN=2^26)
with the Proposed Framework (NN=2^27)
with the Proposed Framework (NN=2^26)
Elapsed Time (s)
0.25
0.2
0.15
0.1
0.05
0
1
2
4
8
16
#. of Nodes
+1.6%

まとめ

- **GPUでデータ通信可能なフレームワークを検討**
  - コード変換による実現方法を提案
  - 提案フレームワーク使用時の実行時間を測定
- **今後の課題**
  - 各種ベンチマークでの評価
  - 計算と通信のオーバラップの実現
  - 「複数GPU/ノード」上での通信のサポート

関連研究

- **GPUクラスタ向けプログラミングフレームワーク**
  - R-Stream→複数GPUコードな変換器実現[5]
  - cudaMPI[6]
- **GPU-CPUハイブリッドプログラミング環境**
  - HPCPE[7]
  - OpenMP→GPUコード変換手法[8,9]
  - OpenMPとMPIによる複数GPU利用手法
  - DSL[12,13]