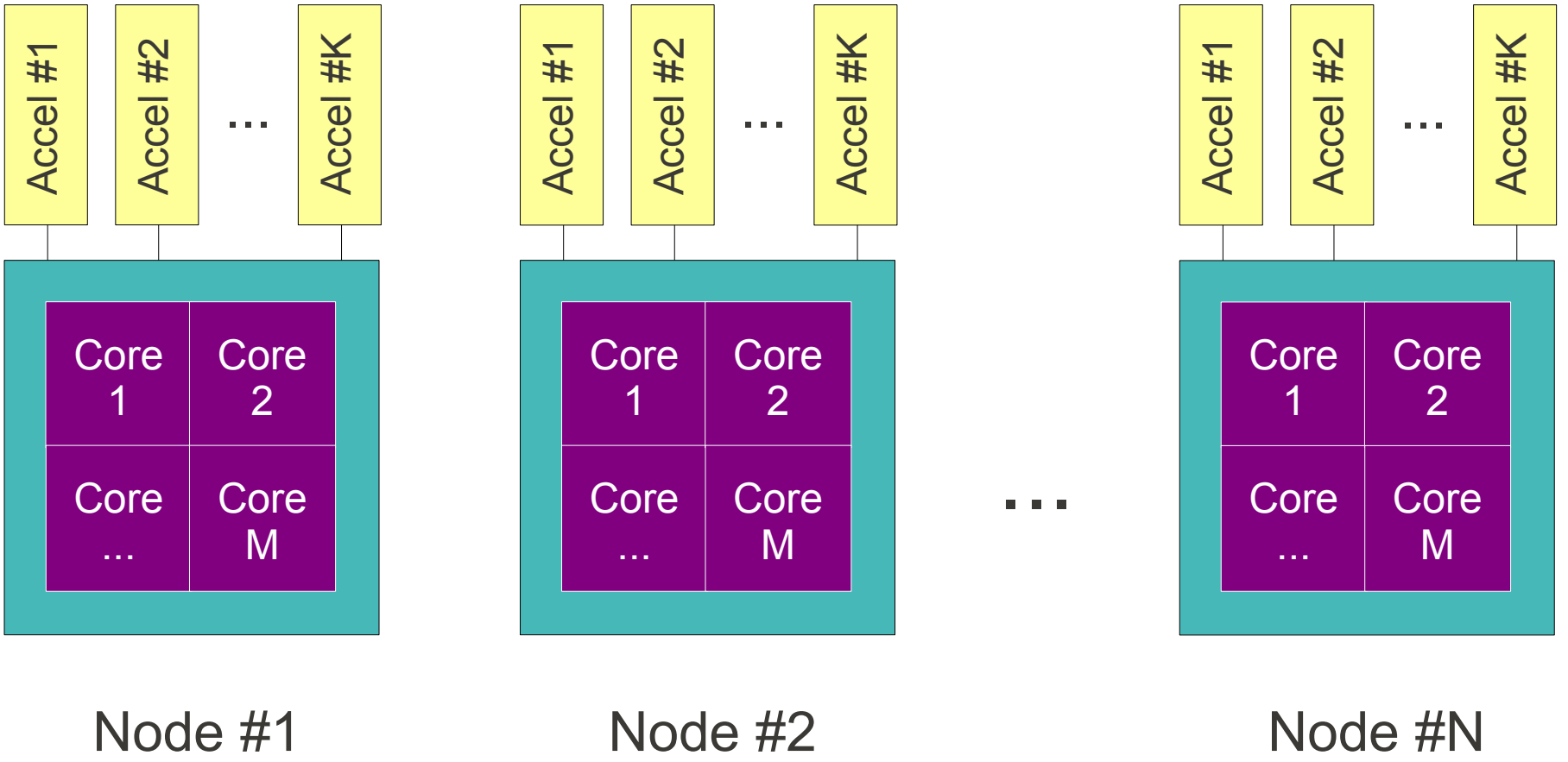

# Использование CUDA совместно с OpenMP и MPI для программирования гетерогенных систем

Микушин Д.Н.

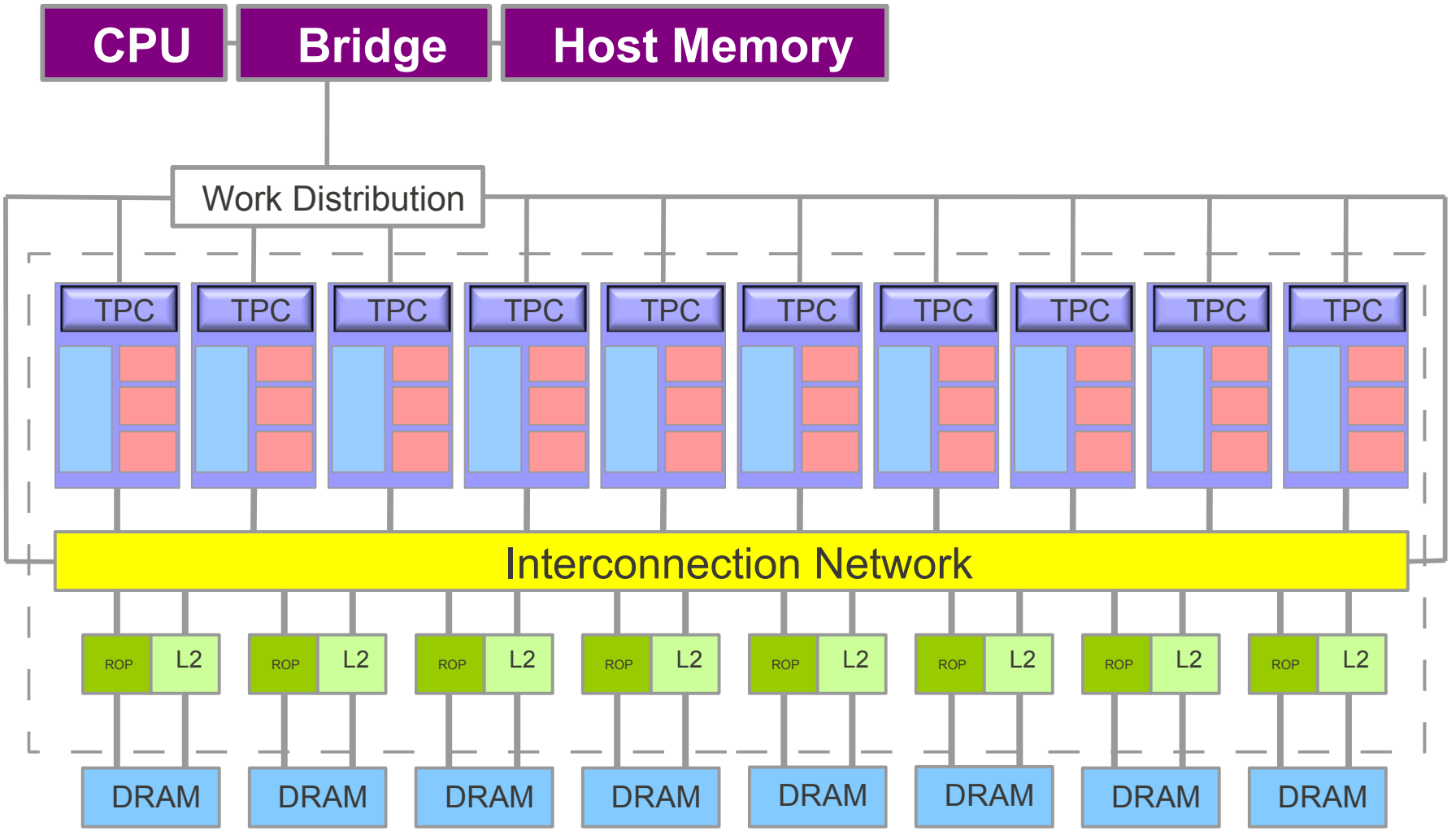
# В этой лекции

- Установка и настройка MPI
- Пример задачи для гибридной системы
  - Алгоритм
  - Реализация на CUDA
  - CUDA + OpenMP
  - CUDA + MPI

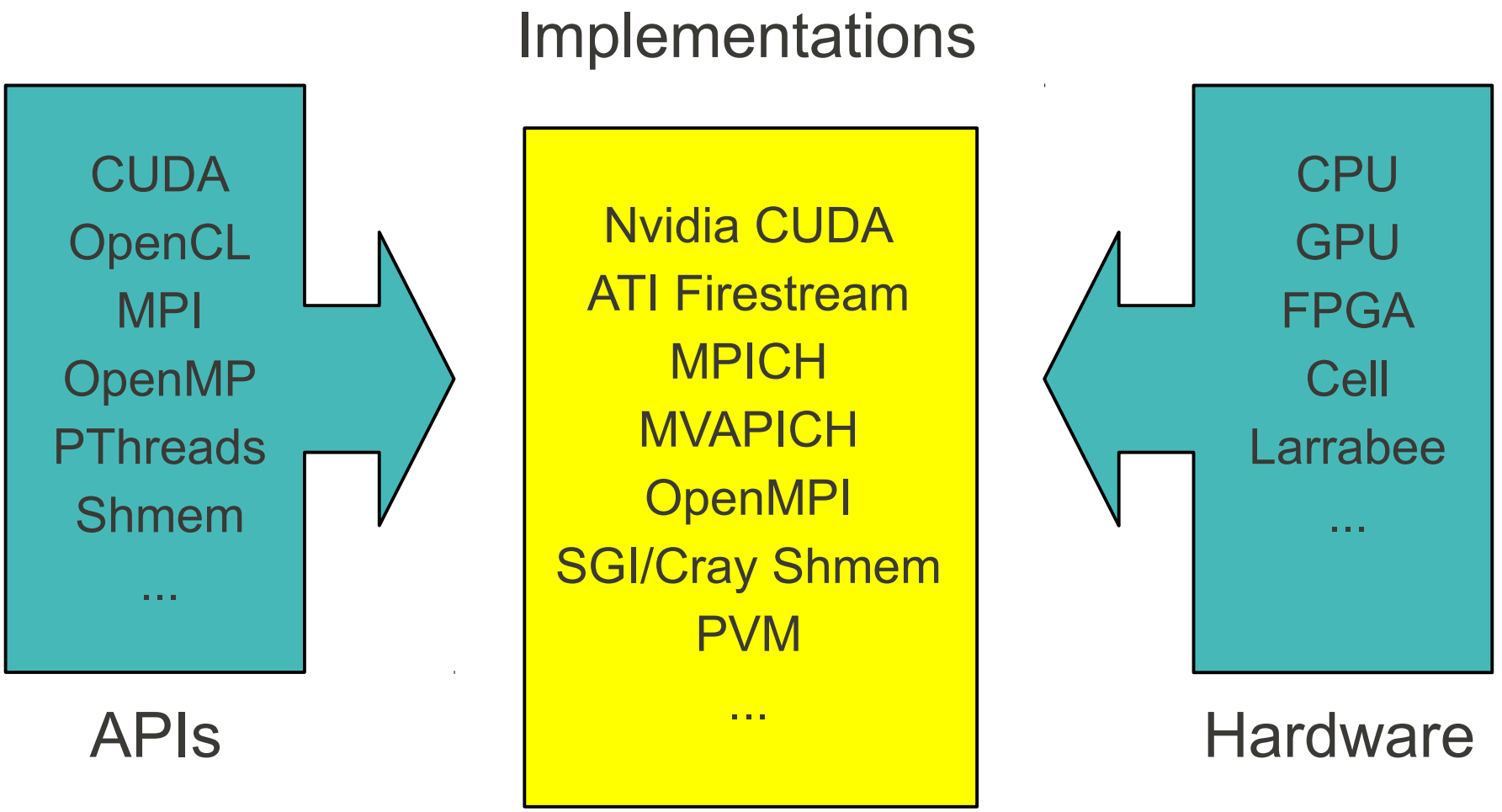
# Гибридная вычислительная система



# Гибридная вычислительная система



# Устройства, стандарты, реализации




# MPI — установка и настройка (Linux)

**Install MPI package**

yum install mpich2 mpich2-devel
yum install openmpi openmpi-devel

**Create .mpd.conf (/etc/mpd.conf)**

MPD_SECRETWORD=<password>
chmod 600 .mpd.conf

**Create mpd.hosts files:**

MPICH:

hostname:<num_cores>
…
hostname:<num_cores>

OpenMPI:

hostname slots=<num_cores>
...
hostname slots=<num_cores>

# MPICH2 — управление (Linux)

**Launch mpd daemon on each node**

```
$ mpdboot -v
LAUNCHED mpd on msiwind via
mpdboot_msiwind (handle_mpd_output 420): from mpd on
msiwind, invalid port info: no_port

$ mpd &
$ mpdallexit
$ mpdboot
LAUNCHED mpd on msiwind via
RUNNING: mpd on msiwind
```

**Execute MPI program**

```
$ mpirun -np <num_processes> <executable>
```

**Shutdown mpd daemons**

```
$ mpdallexit
```

# MPICH2 — управление (Linux)

**Launch mpd daemon on each node**

```
$ mpdboot -v
LAUNCHED mpd on msiwind via
mpdboot_msiwind (handle_mpd_output 420): from mpd on
msiwind, invalid port info: no_port

$ mpd &
$ mpdallexit
$ mpdboot
LAUNCHED mpd on msiwind via
RUNNING: mpd on msiwind
```

- Multiple mpd instances per node
- Multicore CPUs:

```
$ mpdboot --totalnum=-1 --ncpus=2
```

**Execute MPI program**

```
$ mpirun -np <num_processes> <executable>
```

**Shutdown mpd daemons**

```
$ mpdallexit
```

# Метод Монте-Карло для обращения матриц (MKMI)

Марковские цепочки для вектора $x=(x_1 \dots x_m)$ :

$$k_0 \to k_1 \to \dots \to k_i$$

где m – число всевозможных состояний

Начальные вероятности и вероятности переходов:

$$P(k_0=\alpha)=p_\alpha$$
$$P(k_j=\beta, k_{j-1}=\alpha)=p_{\alpha\beta}$$

# Метод Монте-Карло для обращения матриц (MKMI)

Марковские цепочки для матрицы $A=(a_{ij}), n\,x\,n$ :

$$r \to k_0 \to k_1 \to \ldots \to k_i$$

Весовые коэффициенты марковской цепи для матрицы A :

$$W_j = \frac{a_{k_0 k_1} a_{k_1 k_2} \ldots a_{k_{j-1} k_j}}{p_{k_0 k_1} p_{k_1 k_2} \cdots p_{k_{j-1} k_j}}$$

$$W_j = W_{j-1} \frac{a_{k_{j-1} k_j}}{p_{k_{j-1} k_j}}, W_0 = 0$$

где $p_{k_{j-1} k_j}$ - вероятности перехода

# Метод Монте-Карло для обращения матриц (MKMI)

Задача : дана матрица B, n x n, найти C: BC = I

Метод Монте-Карло :

$$c_{rr'} \approx \frac{1}{N} \sum_{s=1}^{N} [ \sum_{j:k_j=r'} W_j ]$$

$$W_j = W_{j-1} \frac{a_{k_{j-1}k_j}}{p_{k_{j-1}k_j}}, W_0 = 0, A = I - B$$

$(j:k_j=r')$ - учитываются лишь те $W_j$, для которых $k_j=r'$

r, r' = 1, 2, … n, N – число используемых марковских цепочек

# Метод Монте-Карло для обращения матриц (MKMI)

“Почти оптимальные” вероятности перехода (Megson, Alexandrov, Dimov, 1994) :

$$p_{\alpha\beta} = \frac{|a_{\alpha\beta}|}{\sum_{\beta} |a_{\alpha\beta}|}$$

# Метод Монте-Карло для обращения матриц (MKMI)

Оценка сложности : $O(n^2 NT)$

N – число марковских цепочек,
T – мера длины цепочек
**не зависящие от размерности матрицы**

Оценка на длину цепочек
(Megson, Aleksandrov, Dimov, 1994) :

$$N = [\frac{0.6745}{\epsilon(1-\|A\|)}]^2$$

# MKMI — алгоритм

- Вычисление нормы `||A|| = ||I – B||` и строчных сумм элементов `SA[i]`
- Расчёт числа траекторий `ns`, необходимых для достижения требуемой точности
- Для каждого `c = C[i0, j0]` – элемента обратной матрицы:
  - Инициализация `c := 0` для недиагональных и `c := ns` – для диагональных элементов
  - Для каждой траектории `s`:
    - Начальное значение весового коэффициента: `ws := 1`
    - Начальное значение индекса: `i := 1`
    - Пока `ws >= eps`:
      - Генерация очередного целого индекса `j : 1 … N` с плотностью распределения вероятности пропорциональной значениям элементов i-ой строки матрицы `A`
      - Вычисление нового значения `ws := ws * SA[i]  * sgn(A[i,j])`
      - Если `(j == j0) → c := c + ws`
      - `i := j`
  - `c := c / ns → C[i0,j0] := c`

# MKMI — CUDA (А. Грайвер, ИГТУ)

- Минимизация корреляций между генераторами случайных чисел отдельных GPU-нитей
- Минимизация числа условных переходов (?)
- Часть данных в константной памяти

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
1  __global__ void inversionKernel(float *C, float *B, float *A,
2      int* seeds, int nchains, float invnchains)
3  {
4      int idx = blockIdx.x * blockDim.x + threadIdx.x;
5
6      // Set unique RNG state
7      mt19937si(seeds[blockIdx.x]);
8      float cIdx = C[idx];
9
10     // Diagonal elements start with W0 = 1.0
11     // (multiplied by number of chains averaged)
12     if (idx % (MATRIX_SIZE + 1) == 0)
13         cIdx = (float)nchains;
14
15     __syncthreads();
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
1  __global__ void inversionKernel(float *C, float *B, float *A,
2      int* seeds, int nchains, float invnchains)
3  {
4      int idx = blockIdx.x * blockDim.x + threadIdx.x;
5
6      // Set unique RNG state
7      mt19937si(seeds[blockIdx.x]);
8      float cIdx = C[idx];
9
10     // Diagonal elements start with W0 = 1.0
11     // (multiplied by number of chains averaged)
12     if (idx % (MATRIX_SIZE + 1) == 0)
13         cIdx = (float)nchains;
14
15     __syncthreads();
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
1   __global__ void inversionKernel(float *C, float *B, float *A,
2       int* seeds, int nchains, float invnchains)
3   {
4       int idx = blockIdx.x * blockDim.x + threadIdx.x;
5
6       // Set unique RNG state
7       mt19937si(seeds[blockIdx.x]);
8       float cIdx = C[idx];
9
10      // Diagonal elements start with W0 = 1.0
11      // (multiplied by number of chains averaged)
12      if (idx % (MATRIX_SIZE + 1) == 0)
13          cIdx = (float)nchains;
14
15      __syncthreads();
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
1   __global__ void inversionKernel(float *C, float *B, float *A,
2       int* seeds, int nchains, float invnchains)
3   {
4       int idx = blockIdx.x * blockDim.x + threadIdx.x;
5
6       // Set unique RNG state
7       mt19937si(seeds[blockIdx.x]);
8       float cIdx = C[idx];
9
10      // Diagonal elements start with W0 = 1.0
11      // (multiplied by number of chains averaged)
12      if (idx % (MATRIX_SIZE + 1) == 0)
13          cIdx = (float)nchains;
14
15      __syncthreads();
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
1   __global__ void inversionKernel(float *C, float *B, float *A,
2       int* seeds, int nchains, float invnchains)
3   {
4       int idx = blockIdx.x * blockDim.x + threadIdx.x;
5
6       // Set unique RNG state
7       mt19937si(seeds[blockIdx.x]);
8       float cIdx = C[idx];
9
10      // Diagonal elements start with W0 = 1.0
11      // (multiplied by number of chains averaged)
12      if (idx % (MATRIX_SIZE + 1) == 0)
13          cIdx = (float)nchains;
14
15      __syncthreads();
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
1   __global__ void inversionKernel(float *C, float *B, float *A,
2       int* seeds, int nchains, float invnchains)
3   {
4       int idx = blockIdx.x * blockDim.x + threadIdx.x;
5
6       // Set unique RNG state
7       mt19937si(seeds[blockIdx.x]);
8       float cIdx = C[idx];
9
10      // Diagonal elements start with W0 = 1.0
11      // (multiplied by number of chains averaged)
12      if (idx % (MATRIX_SIZE + 1) == 0)
13          cIdx = (float)nchains;
14
15      __syncthreads();
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
16  // For each Markov chain
17  for (int s = 0; s < nchains; s++)
18  {
19      // Each chain starts with the row index (i)
20      int i = idx / MATRIX_SIZE;
21
22      // Each W starts with initial value 1.0
23      float w0 = 1.0f;
24
25      // Build random chain.
26      for (int cn = 0; cn < CHAIN_LENGHT; ++cn)
27      {
28          // Generate next chain state using A's chain0 row magnitudes
29          int j = 0;
30          float Aidx = A[j * MATRIX_SIZE + i];
31          float SAi = c_RowSum[i];
32
33          float val = ((float)mt19937s() + 1.0f) / 4294967296.0f;
34          for (int m = 0; m < MATRIX_SIZE; ++m)
35          {
36              val -= fabs(Aidx) / SAi;
37              if (val < 0.0f) break;
38              j++;
39          }
40
41          __syncthreads();
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
16  // For each Markov chain
17  for (int s = 0; s < nchains; s++)
18  {
19      // Each chain starts with the row index (i)
20      int i = idx / MATRIX_SIZE;
21
22      // Each W starts with initial value 1.0
23      float w0 = 1.0f;
24
25      // Build random chain.
26      for (int cn = 0; cn < CHAIN_LENGHT; ++cn)
27      {
28          // Generate next chain state using A's chain0 row magnitudes
29          int j = 0;
30          float Aidx = A[j * MATRIX_SIZE + i];
31          float SAi = c_RowSum[i];
32
33          float val = ((float)mt19937s() + 1.0f) / 4294967296.0f;
34          for (int m = 0; m < MATRIX_SIZE; ++m)
35          {
36              val -= fabs(Aidx) / SAi;
37              if (val < 0.0f) break;
38              j++;
39          }
40
41          __syncthreads();
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
16  // For each Markov chain
17  for (int s = 0; s < nchains; s++)
18  {
19      // Each chain starts with the row index (i)
20      int i = idx / MATRIX_SIZE;
21
22      // Each W starts with initial value 1.0
23      float w0 = 1.0f;
24
25      // Build random chain.
26      for (int cn = 0; cn < CHAIN_LENGHT; ++cn)
27      {
28          // Generate next chain state using A's chain0 row magnitudes
29          int j = 0;
30          float Aidx = A[j * MATRIX_SIZE + i];
31          float SAi = c_RowSum[i];
32
33          float val = ((float)mt19937s() + 1.0f) / 4294967296.0f;
34          for (int m = 0; m < MATRIX_SIZE; ++m)
35          {
36              val -= fabs(Aidx) / SAi;
37              if (val < 0.0f) break;
38              j++;
39          }
40
41          __syncthreads();
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
16  // For each Markov chain
17  for (int s = 0; s < nchains; s++)
18  {
19      // Each chain starts with the row index (i)
20      int i = idx / MATRIX_SIZE;
21
22      // Each W starts with initial value 1.0
23      float w0 = 1.0f;
24
25      // Build random chain.
26      for (int cn = 0; cn < CHAIN_LENGHT; ++cn)
27      {
28          // Generate next chain state using A's chain0 row magnitudes
29          int j = 0;
30          float Aidx = A[j * MATRIX_SIZE + i];
31          float SAi = c_RowSum[i];
32
33          float val = ((float)mt19937s() + 1.0f) / 4294967296.0f;
34          for (int m = 0; m < MATRIX_SIZE; ++m)
35          {
36              val -= fabs(Aidx) / SAi;
37              if (val < 0.0f) break;
38              j++;
39          }
40
41          __syncthreads();
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
16  // For each Markov chain
17  for (int s = 0; s < nchains; s++)
18  {
19      // Each chain starts with the row index (i)
20      int i = idx / MATRIX_SIZE;
21
22      // Each W starts with initial value 1.0
23      float w0 = 1.0f;
24
25      // Build random chain.
26      for (int cn = 0; cn < CHAIN_LENGHT; ++cn)
27      {
28          // Generate next chain state using A's chain0 row magnitudes
29          int j = 0;
30          float Aidx = A[j * MATRIX_SIZE + i];
31          float SAi = c_RowSum[i];
32
33          float val = ((float)mt19937s() + 1.0f) / 4294967296.0f;
34          for (int m = 0; m < MATRIX_SIZE; ++m)
35          {
36              val -= fabs(Aidx) / SAi;
37              if (val < 0.0f) break;
38              j++;
39          }
40
41          __syncthreads();
```

# МКМІ — CUDA (А. Грайвер, ИГТУ)

```
42              // Update W
43              float w1 = w0 * SAi;
44              if (Aidx < 0)
45                  w1 = -w1;
46
47              // Update matrix element only if state index matches
48              // target column index
49              if (j == (idx % MATRIX_SIZE))
50                  cIdx += w1;
51
52              // Switch to next chain state
53              i = j;
54              w0 = w1;
55          }
56      }
57
58      __syncthreads();
59
60      // Write the result
61      cIdx *= invnchains;
62      C[idx] = cIdx;
63  }
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
42              // Update W
43              float w1 = w0 * SAi;
44              if (Aidx < 0)
45                  w1 = -w1;
46
47              // Update matrix element only if state index matches
48              // target column index
49              if (j == (idx % MATRIX_SIZE))
50                  cIdx += w1;
51
52              // Switch to next chain state
53              i = j;
54              w0 = w1;
55          }
56      }
57
58      __syncthreads();
59
60      // Write the result
61      cIdx *= invnchains;
62      C[idx] = cIdx;
63  }
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
42              // Update W
43              float w1 = w0 * SAi;
44              if (Aidx < 0)
45                  w1 = -w1;
46
47              // Update matrix element only if state index matches
48              // target column index
49              if (j == (idx % MATRIX_SIZE))
50                  cIdx += w1;
51
52              // Switch to next chain state
53              i = j;
54              w0 = w1;
55          }
56      }
57
58      __syncthreads();
59
60      // Write the result
61      cIdx *= invnchains;
62      C[idx] = cIdx;
63  }
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
42              // Update W
43              float w1 = w0 * SAi;
44              if (Aidx < 0)
45                  w1 = -w1;
46
47              // Update matrix element only if state index matches
48              // target column index
49              if (j == (idx % MATRIX_SIZE))
50                  cIdx += w1;
51
52              // Switch to next chain state
53              i = j;
54              w0 = w1;
55          }
56      }
57
58      __syncthreads();
59
60      // Write the result
61      cIdx *= invnchains;
62      C[idx] = cIdx;
63  }
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

```
42              // Update W
43              float w1 = w0 * SAi;
44              if (Aidx < 0)
45                  w1 = -w1;
46
47              // Update matrix element only if state index matches
48              // target column index
49              if (j == (idx % MATRIX_SIZE))
50                  cIdx += w1;
51
52              // Switch to next chain state
53              i = j;
54              w0 = w1;
55          }
56      }
57
58      __syncthreads();
59
60      // Write the result
61      cIdx *= invnchains;
62      C[idx] = cIdx;
63  }
```

# MKMI — CUDA (А. Грайвер, ИГТУ)

- Оптимизация функции выбора траектории

(parallel prefix sum + binary search + select)

```
33          float val = ((float)mt19937s() + 1.0f) / 4294967296.0f;
34          for (int m = 0; m < MATRIX_SIZE; ++m)
35          {
36              val -= fabs(Aidx) / SAi;
37              if (val < 0.0f) break;
38              j++;
39          }
```

- Больше часто используемых данных в

разделяемой памяти

# MPI / Threads specifics

- Модель обмена данных
- Синхронизация потоков
- Декомпозиция области

# MKMI — OpenMP frontend

```
 1  int npart = n2 / nthreads;
 2  int nthreads1 = n2 % nthreads;
 3  int imin[nthreads], imax[nthreads];
 4
 5  for (int ithread = 0; ithread < nthreads1; ithread++)
 6  {
 7      imin[ithread] = (npart + 1) * ithread + 1;
 8      imax[ithread] = imin[ithread] + npart;
 9      printf("B Thread %d : [%d .. %d]\n",
10          ithread, imin[ithread], imax[ithread]);
11  }
12  for (int ithread = nthreads1; ithread < nthreads; ithread++)
13  {
14      imin[ithread] = (npart + 1) * nthreads1 +
15          (ithread - nthreads1) * npart + 1;
16      imax[ithread] = imin[ithread] + npart - 1;
17      printf("N Thread %d : [%d .. %d]\n",
18          ithread, imin[ithread], imax[ithread]);
19  }
20
21  #pragma omp parallel for
22  for (int ithread = 0; ithread < nthreads; ithread++)
23  {
24      matinversemc(B, C, &n, &eps, &delta,
25          &imin[ithread], &imax[ithread]);
26  }
```

# MKMI — OpenMP frontend

```
 1  int npart = n2 / nthreads;
 2  int nthreads1 = n2 % nthreads;
 3  int imin[nthreads], imax[nthreads];
 4
 5  for (int ithread = 0; ithread < nthreads1; ithread++)
 6  {
 7      imin[ithread] = (npart + 1) * ithread + 1;
 8      imax[ithread] = imin[ithread] + npart;
 9      printf("B Thread %d : [%d .. %d]\n",
10          ithread, imin[ithread], imax[ithread]);
11  }
12  for (int ithread = nthreads1; ithread < nthreads; ithread++)
13  {
14      imin[ithread] = (npart + 1) * nthreads1 +
15          (ithread - nthreads1) * npart + 1;
16      imax[ithread] = imin[ithread] + npart - 1;
17      printf("N Thread %d : [%d .. %d]\n",
18          ithread, imin[ithread], imax[ithread]);
19  }
20
21  #pragma omp parallel for
22  for (int ithread = 0; ithread < nthreads; ithread++)
23  {
24      matinversemc(B, C, &n, &eps, &delta,
25          &imin[ithread], &imax[ithread]);
26  }
```

# MKMI — OpenMP frontend

```
 1  int npart = n2 / nthreads;
 2  int nthreads1 = n2 % nthreads;
 3  int imin[nthreads], imax[nthreads];
 4
 5  for (int ithread = 0; ithread < nthreads1; ithread++)
 6  {
 7      imin[ithread] = (npart + 1) * ithread + 1;
 8      imax[ithread] = imin[ithread] + npart;
 9      printf("B Thread %d : [%d .. %d]\n",
10          ithread, imin[ithread], imax[ithread]);
11  }
12  for (int ithread = nthreads1; ithread < nthreads; ithread++)
13  {
14      imin[ithread] = (npart + 1) * nthreads1 +
15          (ithread - nthreads1) * npart + 1;
16      imax[ithread] = imin[ithread] + npart - 1;
17      printf("N Thread %d : [%d .. %d]\n",
18          ithread, imin[ithread], imax[ithread]);
19  }
20
21  #pragma omp parallel for
22  for (int ithread = 0; ithread < nthreads; ithread++)
23  {
24      matinversemc(B, C, &n, &eps, &delta,
25          &imin[ithread], &imax[ithread]);
26  }
```

# MKMI — OpenMP frontend

```
 1  int npart = n2 / nthreads;
 2  int nthreads1 = n2 % nthreads;
 3  int imin[nthreads], imax[nthreads];
 4
 5  for (int ithread = 0; ithread < nthreads1; ithread++)
 6  {
 7      imin[ithread] = (npart + 1) * ithread + 1;
 8      imax[ithread] = imin[ithread] + npart;
 9      printf("B Thread %d : [%d .. %d]\n",
10          ithread, imin[ithread], imax[ithread]);
11  }
12  for (int ithread = nthreads1; ithread < nthreads; ithread++)
13  {
14      imin[ithread] = (npart + 1) * nthreads1 +
15          (ithread - nthreads1) * npart + 1;
16      imax[ithread] = imin[ithread] + npart - 1;
17      printf("N Thread %d : [%d .. %d]\n",
18          ithread, imin[ithread], imax[ithread]);
19  }
20
21  #pragma omp parallel for
22  for (int ithread = 0; ithread < nthreads; ithread++)
23  {
24      matinversemc(B, C, &n, &eps, &delta,
25          &imin[ithread], &imax[ithread]);
26  }
```

# MKMI — OpenMP frontend

```
 1  int npart = n2 / nthreads;
 2  int nthreads1 = n2 % nthreads;
 3  int imin[nthreads], imax[nthreads];
 4
 5  for (int ithread = 0; ithread < nthreads1; ithread++)
 6  {
 7      imin[ithread] = (npart + 1) * ithread + 1;
 8      imax[ithread] = imin[ithread] + npart;
 9      printf("B Thread %d : [%d .. %d]\n",
10          ithread, imin[ithread], imax[ithread]);
11  }
12  for (int ithread = nthreads1; ithread < nthreads; ithread++)
13  {
14      imin[ithread] = (npart + 1) * nthreads1 +
15          (ithread - nthreads1) * npart + 1;
16      imax[ithread] = imin[ithread] + npart - 1;
17      printf("N Thread %d : [%d .. %d]\n",
18          ithread, imin[ithread], imax[ithread]);
19  }
20
21  #pragma omp parallel for
22  for (int ithread = 0; ithread < nthreads; ithread++)
23  {
24      matinversemc(B, C, &n, &eps, &delta,
25          &imin[ithread], &imax[ithread]);
26  }
```

# MKMI — OpenMP frontend

```
 1  int npart = n2 / nthreads;
 2  int nthreads1 = n2 % nthreads;
 3  int imin[nthreads], imax[nthreads];
 4
 5  for (int ithread = 0; ithread < nthreads1; ithread++)
 6  {
 7      imin[ithread] = (npart + 1) * ithread + 1;
 8      imax[ithread] = imin[ithread] + npart;
 9      printf("B Thread %d : [%d .. %d]\n",
10          ithread, imin[ithread], imax[ithread]);
11  }
12  for (int ithread = nthreads1; ithread < nthreads; ithread++)
13  {
14      imin[ithread] = (npart + 1) * nthreads1 +
15          (ithread - nthreads1) * npart + 1;
16      imax[ithread] = imin[ithread] + npart - 1;
17      printf("N Thread %d : [%d .. %d]\n",
18          ithread, imin[ithread], imax[ithread]);
19  }
20
21  #pragma omp parallel for
22  for (int ithread = 0; ithread < nthreads; ithread++)
23  {
24      matinversemc(B, C, &n, &eps, &delta,
25          &imin[ithread], &imax[ithread]);
26  }
```

# MKMI — OpenMP frontend

```
 1  int npart = n2 / nthreads;
 2  int nthreads1 = n2 % nthreads;
 3  int imin[nthreads], imax[nthreads];
 4
 5  for (int ithread = 0; ithread < nthreads1; ithread++)
 6  {
 7      imin[ithread] = (npart + 1) * ithread + 1;
 8      imax[ithread] = imin[ithread] + npart;
 9      printf("B Thread %d : [%d .. %d]\n",
10          ithread, imin[ithread], imax[ithread]);
11  }
12  for (int ithread = nthreads1; ithread < nthreads; ithread++)
13  {
14      imin[ithread] = (npart + 1) * nthreads1 +
15          (ithread - nthreads1) * npart + 1;
16      imax[ithread] = imin[ithread] + npart - 1;
17      printf("N Thread %d : [%d .. %d]\n",
18          ithread, imin[ithread], imax[ithread]);
19  }
20
21  #pragma omp parallel for
22  for (int ithread = 0; ithread < nthreads; ithread++)
23  {
24      matinversemc(B, C, &n, &eps, &delta,
25          &imin[ithread], &imax[ithread]);
26  }
```

# MKMI — OpenMP frontend

```
 1  int npart = n2 / nthreads;
 2  int nthreads1 = n2 % nthreads;
 3  int imin[nthreads], imax[nthreads];
 4
 5  for (int ithread = 0; ithread < nthreads1; ithread++)
 6  {
 7      imin[ithread] = (npart + 1) * ithread + 1;
 8      imax[ithread] = imin[ithread] + npart;
 9      printf("B Thread %d : [%d .. %d]\n",
10          ithread, imin[ithread], imax[ithread]);
11  }
12  for (int ithread = nthreads1; ithread < nthreads; ithread++)
13  {
14      imin[ithread] = (npart + 1) * nthreads1 +
15          (ithread - nthreads1) * npart + 1;
16      imax[ithread] = imin[ithread] + npart - 1;
17      printf("N Thread %d : [%d .. %d]\n",
18          ithread, imin[ithread], imax[ithread]);
19  }
20
21  #pragma omp parallel for
22  for (int ithread = 0; ithread < nthreads; ithread++)
23  {
24      matinversemc(B, C, &n, &eps, &delta,
25          &imin[ithread], &imax[ithread]);
26  }
```

# MKMI — OpenMP frontend

```
 1  int npart = n2 / nthreads;
 2  int nthreads1 = n2 % nthreads;
 3  int imin[nthreads], imax[nthreads];
 4
 5  for (int ithread = 0; ithread < nthreads1; ithread++)
 6  {
 7      imin[ithread] = (npart + 1) * ithread + 1;
 8      imax[ithread] = imin[ithread] + npart;
 9      printf("B Thread %d : [%d .. %d]\n",
10          ithread, imin[ithread], imax[ithread]);
11  }
12  for (int ithread = nthreads1; ithread < nthreads; ithread++)
13  {
14      imin[ithread] = (npart + 1) * nthreads1 +
15          (ithread - nthreads1) * npart + 1;
16      imax[ithread] = imin[ithread] + npart - 1;
17      printf("N Thread %d : [%d .. %d]\n",
18          ithread, imin[ithread], imax[ithread]);
19  }
20
21  #pragma omp parallel for
22  for (int ithread = 0; ithread < nthreads; ithread++)
23  {
24      matinversemc(B, C, &n, &eps, &delta,
25          &imin[ithread], &imax[ithread]);
26  }
```

# MKMI — OpenMP frontend

```
 1  int npart = n2 / nthreads;
 2  int nthreads1 = n2 % nthreads;
 3  int imin[nthreads], imax[nthreads];
 4
 5  for (int ithread = 0; ithread < nthreads1; ithread++)
 6  {
 7      imin[ithread] = (npart + 1) * ithread + 1;
 8      imax[ithread] = imin[ithread] + npart;
 9      printf("B Thread %d : [%d .. %d]\n",
10          ithread, imin[ithread], imax[ithread]);
11  }
12  for (int ithread = nthreads1; ithread < nthreads; ithread++)
13  {
14      imin[ithread] = (npart + 1) * nthreads1 +
15          (ithread - nthreads1) * npart + 1;
16      imax[ithread] = imin[ithread] + npart - 1;
17      printf("N Thread %d : [%d .. %d]\n",
18          ithread, imin[ithread], imax[ithread]);
19  }
20
21  #pragma omp parallel for
22  for (int ithread = 0; ithread < nthreads; ithread++)
23  {
24      matinversemc(B, C, &n, &eps, &delta,
25          &imin[ithread], &imax[ithread]);
26  }
```

# MKMI — MPI frontend

```
21  call MPI_Init(ierr)
22
23  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
24  call MPI_Comm_size(MPI_COMM_WORLD, szcomm, ierr)
25
26  ! Declare root node
27  root = szcomm - 1
28
29  npart = n / szcomm
30  imin = 1
31  imax = n
32  jmin = rank * npart + 1
33  jmax = jmin + npart - 1
34
35  ! Generate B matrix
36  if (rank .eq. root) then
37    npart = npart + mod(n, szcomm)
38    jmax = n
39    jmin = n - npart + 1
40
41    call MatGenAB(A, B, n)
42    call MatOut('B', B, n)
43    call MatOut('A', A, n)
44
45    call cpu_time(start)
46  endif
```

# MKMI — MPI frontend

```
21  call MPI_Init(ierr)
22
23  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
24  call MPI_Comm_size(MPI_COMM_WORLD, szcomm, ierr)
25
26  ! Declare root node
27  root = szcomm - 1
28
29  npart = n / szcomm
30  imin = 1
31  imax = n
32  jmin = rank * npart + 1
33  jmax = jmin + npart - 1
34
35  ! Generate B matrix
36  if (rank .eq. root) then
37    npart = npart + mod(n, szcomm)
38    jmax = n
39    jmin = n - npart + 1
40
41    call MatGenAB(A, B, n)
42    call MatOut('B', B, n)
43    call MatOut('A', A, n)
44
45    call cpu_time(start)
46  endif
```

# MKMI — MPI frontend

```
21  call MPI_Init(ierr)
22
23  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
24  call MPI_Comm_size(MPI_COMM_WORLD, szcomm, ierr)
25
26  ! Declare root node
27  root = szcomm - 1
28
29  npart = n / szcomm
30  imin = 1
31  imax = n
32  jmin = rank * npart + 1
33  jmax = jmin + npart - 1
34
35  ! Generate B matrix
36  if (rank .eq. root) then
37    npart = npart + mod(n, szcomm)
38    jmax = n
39    jmin = n - npart + 1
40
41    call MatGenAB(A, B, n)
42    call MatOut('B', B, n)
43    call MatOut('A', A, n)
44
45    call cpu_time(start)
46  endif
```

# MKMI — MPI frontend

```
21  call MPI_Init(ierr)
22
23  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
24  call MPI_Comm_size(MPI_COMM_WORLD, szcomm, ierr)
25
26  ! Declare root node
27  root = szcomm - 1
28
29  npart = n / szcomm
30  imin = 1
31  imax = n
32  jmin = rank * npart + 1
33  jmax = jmin + npart - 1
34
35  ! Generate B matrix
36  if (rank .eq. root) then
37    npart = npart + mod(n, szcomm)
38    jmax = n
39    jmin = n - npart + 1
40
41    call MatGenAB(A, B, n)
42    call MatOut('B', B, n)
43    call MatOut('A', A, n)
44
45    call cpu_time(start)
46  endif
```

# MKMI — MPI frontend

```
21  call MPI_Init(ierr)
22
23  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
24  call MPI_Comm_size(MPI_COMM_WORLD, szcomm, ierr)
25
26  ! Declare root node
27  root = szcomm - 1
28
29  npart = n / szcomm
30  imin = 1
31  imax = n
32  jmin = rank * npart + 1
33  jmax = jmin + npart - 1
34
35  ! Generate B matrix
36  if (rank .eq. root) then
37    npart = npart + mod(n, szcomm)
38    jmax = n
39    jmin = n - npart + 1
40
41    call MatGenAB(A, B, n)
42    call MatOut('B', B, n)
43    call MatOut('A', A, n)
44
45    call cpu_time(start)
46  endif
```

# MKMI — MPI frontend

```
21  call MPI_Init(ierr)
22
23  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
24  call MPI_Comm_size(MPI_COMM_WORLD, szcomm, ierr)
25
26  ! Declare root node
27  root = szcomm - 1
28
29  npart = n / szcomm
30  imin = 1
31  imax = n
32  jmin = rank * npart + 1
33  jmax = jmin + npart - 1
34
35  ! Generate B matrix
36  if (rank .eq. root) then
37    npart = npart + mod(n, szcomm)
38    jmax = n
39    jmin = n - npart + 1
40
41    call MatGenAB(A, B, n)
42    call MatOut('B', B, n)
43    call MatOut('A', A, n)
44
45    call cpu_time(start)
46  endif
```

# MKMI — MPI frontend

```
21  call MPI_Init(ierr)
22
23  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
24  call MPI_Comm_size(MPI_COMM_WORLD, szcomm, ierr)
25
26  ! Declare root node
27  root = szcomm - 1
28
29  npart = n / szcomm
30  imin = 1
31  imax = n
32  jmin = rank * npart + 1
33  jmax = jmin + npart - 1
34
35  ! Generate B matrix
36  if (rank .eq. root) then
37    npart = npart + mod(n, szcomm)
38    jmax = n
39    jmin = n - npart + 1
40
41    call MatGenAB(A, B, n)
42    call MatOut('B', B, n)
43    call MatOut('A', A, n)
44
45    call cpu_time(start)
46  endif
```

# MKMI — MPI frontend

```
47  ! Distribute B matrix
48  call MPI_Bcast(B, n * n, MPI_REAL, root, MPI_COMM_WORLD, ierr)
49  call MPI_Barrier(MPI_COMM_WORLD, ierr)
50
51  call MatInverseMC(B, C, n, eps, delta, imin, imax, jmin, jmax)
52
53  if (rank .ne. root) then
54    nnpart = (imax - imin + 1) * (jmax - jmin + 1)
55    call MPI_Send(C(imin, jmin), nnpart,
56      MPI_REAL, root, rank, MPI_COMM_WORLD, ierr)
57  else
58    do i = 0, root - 1
59      imin = 1
60      imax = n
61      jmin = i * npart + 1
62      jmax = jmin + npart - 1
63      nnpart = (imax - imin + 1) * (jmax - jmin + 1)
64      call MPI_Recv(C(imin, jmin), nnpart,
65        MPI_REAL, i, i, MPI_COMM_WORLD, rstatus, ierr)
66    enddo
67  endif
68
69  if (rank .eq. root) then
70   call cpu_time(finish)
71  endif
72
73  call MPI_Finalize(ierr)
```

# MKMI — MPI frontend

```
47  ! Distribute B matrix
48  call MPI_Bcast(B, n * n, MPI_REAL, root, MPI_COMM_WORLD, ierr)
49  call MPI_Barrier(MPI_COMM_WORLD, ierr)
50
51  call MatInverseMC(B, C, n, eps, delta, imin, imax, jmin, jmax)
52
53  if (rank .ne. root) then
54    nnpart = (imax - imin + 1) * (jmax - jmin + 1)
55    call MPI_Send(C(imin, jmin), nnpart,
56      MPI_REAL, root, rank, MPI_COMM_WORLD, ierr)
57  else
58    do i = 0, root - 1
59      imin = 1
60      imax = n
61      jmin = i * npart + 1
62      jmax = jmin + npart - 1
63      nnpart = (imax - imin + 1) * (jmax - jmin + 1)
64      call MPI_Recv(C(imin, jmin), nnpart,
65        MPI_REAL, i, i, MPI_COMM_WORLD, rstatus, ierr)
66    enddo
67  endif
68
69  if (rank .eq. root) then
70   call cpu_time(finish)
71  endif
72
73  call MPI_Finalize(ierr)
```

# MKMI — MPI frontend

```
47  ! Distribute B matrix
48  call MPI_Bcast(B, n * n, MPI_REAL, root, MPI_COMM_WORLD, ierr)
49  call MPI_Barrier(MPI_COMM_WORLD, ierr)
50
51  call MatInverseMC(B, C, n, eps, delta, imin, imax, jmin, jmax)
52
53  if (rank .ne. root) then
54    nnpart = (imax - imin + 1) * (jmax - jmin + 1)
55    call MPI_Send(C(imin, jmin), nnpart,
56      MPI_REAL, root, rank, MPI_COMM_WORLD, ierr)
57  else
58    do i = 0, root - 1
59      imin = 1
60      imax = n
61      jmin = i * npart + 1
62      jmax = jmin + npart - 1
63      nnpart = (imax - imin + 1) * (jmax - jmin + 1)
64      call MPI_Recv(C(imin, jmin), nnpart,
65        MPI_REAL, i, i, MPI_COMM_WORLD, rstatus, ierr)
66    enddo
67  endif
68
69  if (rank .eq. root) then
70   call cpu_time(finish)
71  endif
72
73  call MPI_Finalize(ierr)
```

# MKMI — MPI frontend

```
47  ! Distribute B matrix
48  call MPI_Bcast(B, n * n, MPI_REAL, root, MPI_COMM_WORLD, ierr)
49  call MPI_Barrier(MPI_COMM_WORLD, ierr)
50
51  call MatInverseMC(B, C, n, eps, delta, imin, imax, jmin, jmax)
52
53  if (rank .ne. root) then
54    nnpart = (imax - imin + 1) * (jmax - jmin + 1)
55    call MPI_Send(C(imin, jmin), nnpart,
56      MPI_REAL, root, rank, MPI_COMM_WORLD, ierr)
57  else
58    do i = 0, root - 1
59      imin = 1
60      imax = n
61      jmin = i * npart + 1
62      jmax = jmin + npart - 1
63      nnpart = (imax - imin + 1) * (jmax - jmin + 1)
64      call MPI_Recv(C(imin, jmin), nnpart,
65        MPI_REAL, i, i, MPI_COMM_WORLD, rstatus, ierr)
66    enddo
67  endif
68
69  if (rank .eq. root) then
70   call cpu_time(finish)
71  endif
72
73  call MPI_Finalize(ierr)
```

# MKMI — MPI frontend

```
47  ! Distribute B matrix
48  call MPI_Bcast(B, n * n, MPI_REAL, root, MPI_COMM_WORLD, ierr)
49  call MPI_Barrier(MPI_COMM_WORLD, ierr)
50
51  call MatInverseMC(B, C, n, eps, delta, imin, imax, jmin, jmax)
52
53  if (rank .ne. root) then
54    nnpart = (imax - imin + 1) * (jmax - jmin + 1)
55    call MPI_Send(C(imin, jmin), nnpart,
56      MPI_REAL, root, rank, MPI_COMM_WORLD, ierr)
57  else
58    do i = 0, root - 1
59      imin = 1
60      imax = n
61      jmin = i * npart + 1
62      jmax = jmin + npart - 1
63      nnpart = (imax - imin + 1) * (jmax - jmin + 1)
64      call MPI_Recv(C(imin, jmin), nnpart,
65        MPI_REAL, i, i, MPI_COMM_WORLD, rstatus, ierr)
66    enddo
67  endif
68
69  if (rank .eq. root) then
70   call cpu_time(finish)
71  endif
72
73  call MPI_Finalize(ierr)
```

# MKMI — MPI frontend

```
47  ! Distribute B matrix
48  call MPI_Bcast(B, n * n, MPI_REAL, root, MPI_COMM_WORLD, ierr)
49  call MPI_Barrier(MPI_COMM_WORLD, ierr)
50
51  call MatInverseMC(B, C, n, eps, delta, imin, imax, jmin, jmax)
52
53  if (rank .ne. root) then
54    nnpart = (imax - imin + 1) * (jmax - jmin + 1)
55    call MPI_Send(C(imin, jmin), nnpart,
56      MPI_REAL, root, rank, MPI_COMM_WORLD, ierr)
57  else
58    do i = 0, root - 1
59      imin = 1
60      imax = n
61      jmin = i * npart + 1
62      jmax = jmin + npart - 1
63      nnpart = (imax - imin + 1) * (jmax - jmin + 1)
64      call MPI_Recv(C(imin, jmin), nnpart,
65        MPI_REAL, i, i, MPI_COMM_WORLD, rstatus, ierr)
66    enddo
67  endif
68
69  if (rank .eq. root) then
70   call cpu_time(finish)
71  endif
72
73  call MPI_Finalize(ierr)
```

# MKMI — MPI frontend

```
47  ! Distribute B matrix
48  call MPI_Bcast(B, n * n, MPI_REAL, root, MPI_COMM_WORLD, ierr)
49  call MPI_Barrier(MPI_COMM_WORLD, ierr)
50
51  call MatInverseMC(B, C, n, eps, delta, imin, imax, jmin, jmax)
52
53  if (rank .ne. root) then
54    nnpart = (imax - imin + 1) * (jmax - jmin + 1)
55    call MPI_Send(C(imin, jmin), nnpart,
56      MPI_REAL, root, rank, MPI_COMM_WORLD, ierr)
57  else
58    do i = 0, root - 1
59      imin = 1
60      imax = n
61      jmin = i * npart + 1
62      jmax = jmin + npart - 1
63      nnpart = (imax - imin + 1) * (jmax - jmin + 1)
64      call MPI_Recv(C(imin, jmin), nnpart,
65        MPI_REAL, i, i, MPI_COMM_WORLD, rstatus, ierr)
66    enddo
67  endif
68
69  if (rank .eq. root) then
70   call cpu_time(finish)
71  endif
72
73  call MPI_Finalize(ierr)
```

# MKMI — MPI frontend

```
47  ! Distribute B matrix
48  call MPI_Bcast(B, n * n, MPI_REAL, root, MPI_COMM_WORLD, ierr)
49  call MPI_Barrier(MPI_COMM_WORLD, ierr)
50
51  call MatInverseMC(B, C, n, eps, delta, imin, imax, jmin, jmax)
52
53  if (rank .ne. root) then
54    nnpart = (imax - imin + 1) * (jmax - jmin + 1)
55    call MPI_Send(C(imin, jmin), nnpart,
56      MPI_REAL, root, rank, MPI_COMM_WORLD, ierr)
57  else
58    do i = 0, root - 1
59      imin = 1
60      imax = n
61      jmin = i * npart + 1
62      jmax = jmin + npart - 1
63      nnpart = (imax - imin + 1) * (jmax - jmin + 1)
64      call MPI_Recv(C(imin, jmin), nnpart,
65        MPI_REAL, i, i, MPI_COMM_WORLD, rstatus, ierr)
66    enddo
67  endif
68
69  if (rank .eq. root) then
70   call cpu_time(finish)
71  endif
72
73  call MPI_Finalize(ierr)
```

# MKMI — особенности

- Параллельный приближенный метод
- Быстрый рост сложности с увеличением точности (работа в качестве предобуславливателя?)
- Требуется диагональное преобладание
- Вычислительная сложность модификации для использования с матрицами общего вида слишком велика (?)

# MKMI — модификация для матриц общего вида

S. Branford, C. Sahin, A. Thandavan, C. Weihrauch, V.N. Alexandrov, I.T. Dimov

*Monte Carlo methods for matrix computations on the grid*

```
1   for (int r = 0; r < *n; r++)
2   {
3       real s = dfactor / (1.0 - C[r * *n + r] * dfactor);
4
5       for (int i = 0; i < *n; i++)
6       {
7           A[i] = s * C[r * *n + i];
8           SA[i] = C[i * *n + r];
9       }
10      for (int i = 0; i < *n; i++)
11          for (int j = 0; j < *n; j++)
12              C[j * *n + i] += A[i] * SA[j];
13  }
```

**O(n^3) ?!**

# Заключение

• CUDA встраивается в существующие программные модели гибридных вычислений

• Взаимодействие со множеством API, на различных языках → возможно портирование на CUDA частей существующих параллельных приложений