The extension of DVM-system to solve the problems with intensive irregular memory access

V. Bakhtin, A. Kolganov, V. Krukov, N. Podderyugina, M. Pritula, O. Savitskaya

Keldysh Institute of Applied Mathematics Russian Academy of Sciences

http://dvm-system.org

The class of problems with irregular memory access

- Graph problems;
- Sparce matrices;
- Scientific and technical calculation on irregular grids.

The class of problems with irregular memory access

- Graph problems;
- Sparce matrices;
- Scientific and technical calculation on irregular grids.

They can use the same data format, for example, CSR

Programs with regular access to the memory

Problems:

- A single grid step in the computational domain – no flexibility, impossibly high demands on memory and processing power during grinding;
- Implementation of numerical methods are often tied to the form of a grid - two-dimensional, three-dimensional, cartesian, cylindrical, etc. So we can not replace geometry.

Positive sides:

- Neighborhood relations and spatial coordinates are not stored explicitly – memory saving;
- There is a simple accesses to arrays with constant shifts – freedom for a compiler optimizations, clarity for parallelization (including automatic parallelization).

Programs with irregular access to the memory

Positive sides:

- We can choose any mesh grinding – maintaining degree of grinding in parts of the area;
- Good opportunities for reuse of computing code, the freedom to choose the form of computational areas.

Problems:

- Neighborhood relations and spatial coordinates to be stored explicitly;
- Indirect indexing on arrays accesses – a barrier for a compiler optimizations, the complexity of parallelization (particularly automatic).

```
double A[L][L];

double B[L][L];

int main(int argc, char *argv[]) {
    for(int it = 0; it < ITMAX;  it++) {

        {

            for (int i = 1; i < L - 1; i++)
                for (int j = 1; j < L-1; j++)
                    A[i][j] = B[i][j];


            for (int i = 1; i < L - 1; i++)
                for (int j = 1; j < L - 1; j++)
                    B[i][j] = (A[i - 1][j] + A[i + 1][j] + A[i][j - 1] + A[i][j + 1]) / 4.;
        }
    }
   FILE *f = fopen("jacobi.dat", "wb");

   fwrite(B, sizeof(double), L * L, f);
   fclose(f);
   return 0;
}
```

Jacobi algorithm

```
#pragma dvm array distribute[block][block], shadow[1:1][1:1]
double A[L][L];
#pragma dvm array align([i][j] with A[i][j])
double B[L][L];

int main(int argc, char *argv[]) {
    for(int it = 0; it < ITMAX;  it++) {

        {

            for (int i = 1; i < L - 1; i++)
                for (int j = 1; j < L-1; j++)
                    A[i][j] = B[i][j];


            for (int i = 1; i < L - 1; i++)
                for (int j = 1; j < L - 1; j++)
                    B[i][j] = (A[i - 1][j] + A[i + 1][j] + A[i][j - 1] + A[i][j + 1]) / 4.;
        }
    }
   FILE *f = fopen("jacobi.dat", "wb");

   fwrite(B, sizeof(double), L * L, f);
   fclose(f);
   return 0;
}
```

Jacobi algorithm in the DVMH model

```
#pragma dvm array distribute[block][block], shadow[1:1][1:1]
double A[L][L];
#pragma dvm array align([i][j] with A[i][j])
double B[L][L];

int main(int argc, char *argv[]) {
    for(int it = 0; it < ITMAX; it++) {

        {
          #pragma dvm parallel([i][j] on A[i][j])
          for (int i = 1; i < L - 1; i++)
             for (int j = 1; j < L-1; j++)
                A[i][j] = B[i][j];

          #pragma dvm parallel([i][j] on B[i][j]), shadow_renew(A)
          for (int i = 1; i < L - 1; i++)
             for (int j = 1; j < L - 1; j++)
                B[i][j] = (A[i - 1][j] + A[i + 1][j] + A[i][j - 1] + A[i][j + 1]) / 4.;
        }
    }
   FILE *f = fopen("jacobi.dat", "wb");

   fwrite(B, sizeof(double), L * L, f);
   fclose(f);
   return 0;
}
```

Jacobi algorithm in the DVMH model

```
#pragma dvm array distribute[block][block], shadow[1:1][1:1]
double A[L][L];
#pragma dvm array align([i][j] with A[i][j])
double B[L][L];

int main(int argc, char *argv[]) {
    for(int it = 0; it < ITMAX;  it++) {
        #pragma dvm region inout(A, B)
        {
          #pragma dvm parallel([i][j] on A[i][j])
          for (int i = 1; i < L - 1; i++)
             for (int j = 1; j < L-1; j++)
                A[i][j] = B[i][j];

          #pragma dvm parallel([i][j] on B[i][j]), shadow_renew(A)
          for (int i = 1; i < L - 1; i++)
             for (int j = 1; j < L - 1; j++)
                B[i][j] = (A[i - 1][j] + A[i + 1][j] + A[i][j - 1] + A[i][j + 1]) / 4.;
        }
    }
   FILE *f = fopen("jacobi.dat", "wb");
   #pragma dvm get_actual(B)
   fwrite(B, sizeof(double), L * L, f);
   fclose(f);
   return 0;
}
```

Jacobi algorithm in the DVMH model

Programming tools

C-DVMH = C language + pragmas

Fortran-DVMH = Fortran 95 + pragmas

- Pragmas are high-level specification of parallelism in terms of a sequential program;
- There are no low-level data transfer and synchronization in the program code;
- Sequential programming style;
- Pragmas are "invisible" for standard compilers;
- There is only one instance of the program for sequential and parallel calculations.

Specifications of the parallel execution

- The distribution of arrays between the processors (distribute / align directives);
- Distribution of loop iterations between computing devices (parallel directive);
- Specification of parallel tasks and their mapping to the processors (task directive);
- The effective remote access to data located on other computing devices (shadow / across / remote specifications).

Specifications of the parallel execution

- The effective execution of reduction operations (reduction specification: max/min/sum/maxloc/minloc/…);
- Determination of the program fragments (regions) for execution on accelerators and multi-core CPU (region directive);
- Motion data control between the CPU memory and GPU memory (actual / get_actual directives).

DVM-system components

- Fortran-DVMH compiler;
- C-DVMH compiler;
- DVMH Run Time System;
- DVMH-программ debugger;
- Performance analyzer.

Use of DVMH in MPI-program: reasons

- There are a great foundation and experience of writing parallel programs for clusters;
- DVMH model suggests parallelizing sequential programs;
- The user does not want to give up their parallel program;
- DVMH model does not apply to parallelize some programs (eg, with random access memory).

Use of DVMH in MPI-program: results

- A new mode of DVM-system was addewd locally in each process;
- Undistributed parallel loop construction was added;
- Incremental parallelism and fast evaluation of DVMH-model of the CPU and GPU threads become available;
- Ability to use DVMH-parallelization become available inside the cluster node in the MPI-programs.

Use of DVMH in MPI-program: experience

- Solver with explicit scheme is the part of large developed set of computation programs:
  - C++, 39 000 LOC, templates, polymorphism, etc;

- Local modifications of the one module (~3000 lines) have been made, which are reduced to the addition about 10 DVMH directives;
- We were obtained the accelerations:
  - 2 CPU Intel Xeon X5670 (6 cores on each CPU – **9.8x**;
  - GPU NVidia GTX Titan (Kepler) – **18x**.

New rules for distribution

- Indirect distribution:

```
distribute A[indirect(B)]
```

- Derived distribution:

```
distribute A[derived([cells[i][0]:
          cells[i][2]] with cells[@i])]
```

New shadow edges

- Shadow edges are the set of elements that are not owned by the current process;

- New directive for inderect distribution:
  shadow_add`(nodes[neigh[i][0]:neigh[i][numneigh[i]-1] with nodes[@i]] = neighbours)`

The transition to a local indexing

- The procedure for the convert of the global (initial) index to the local (for direct memory access) <u>is too long</u>;

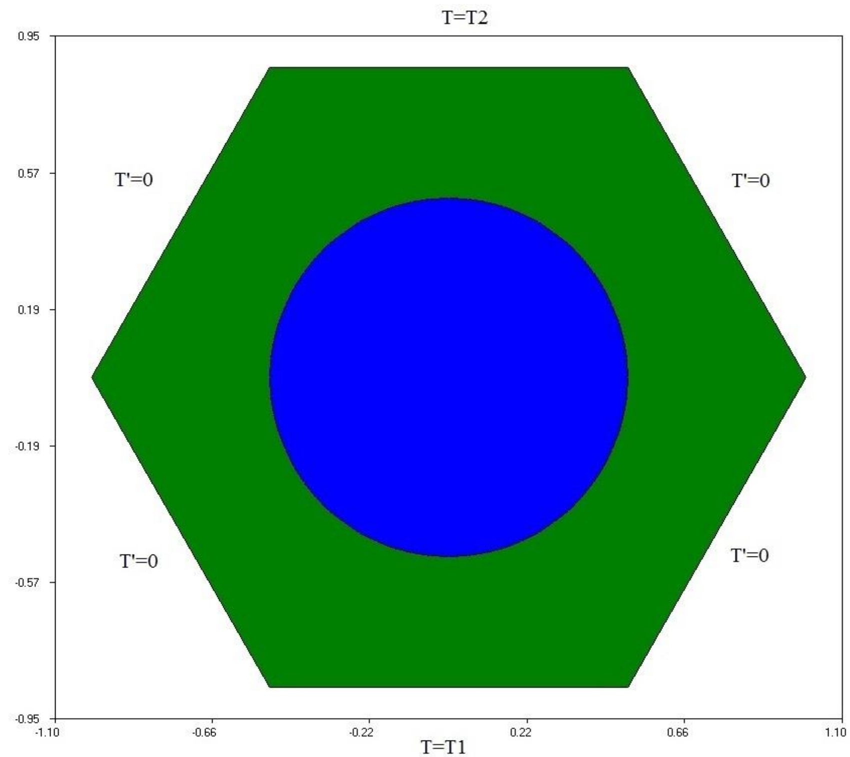
- For regular distributions the global and local indexes are the same;

- The executable directive was introduced for localization arrays indexes for indirect distributions:
  ```
  localize(neigh => nodes[:])
  ```

The test problem

- Two-dimensional heat conduction problem with a constant but discontinuous coefficient in the hexagon.

- The area consists of two materials with different coefficients of thermal.

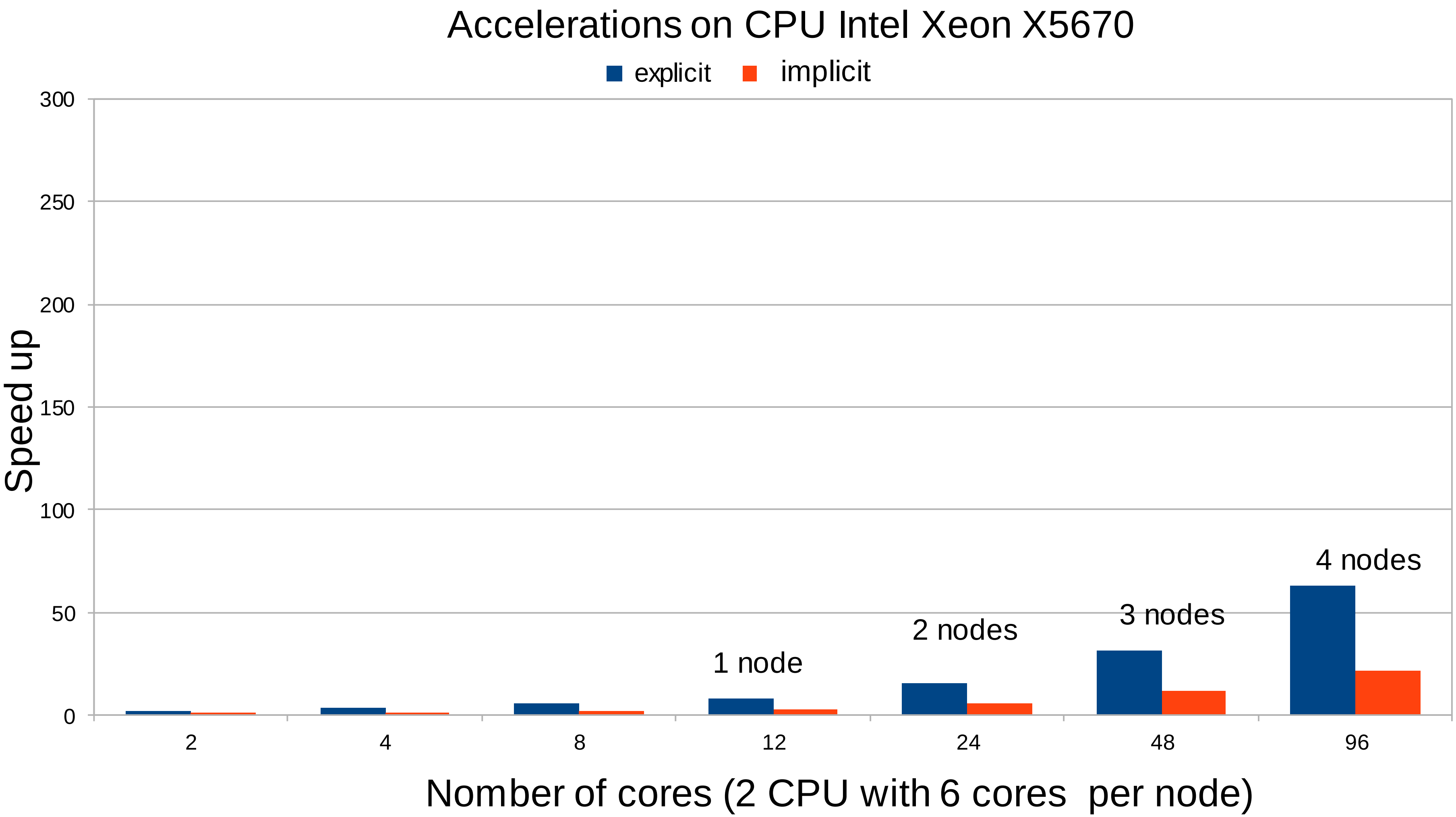


The test problem

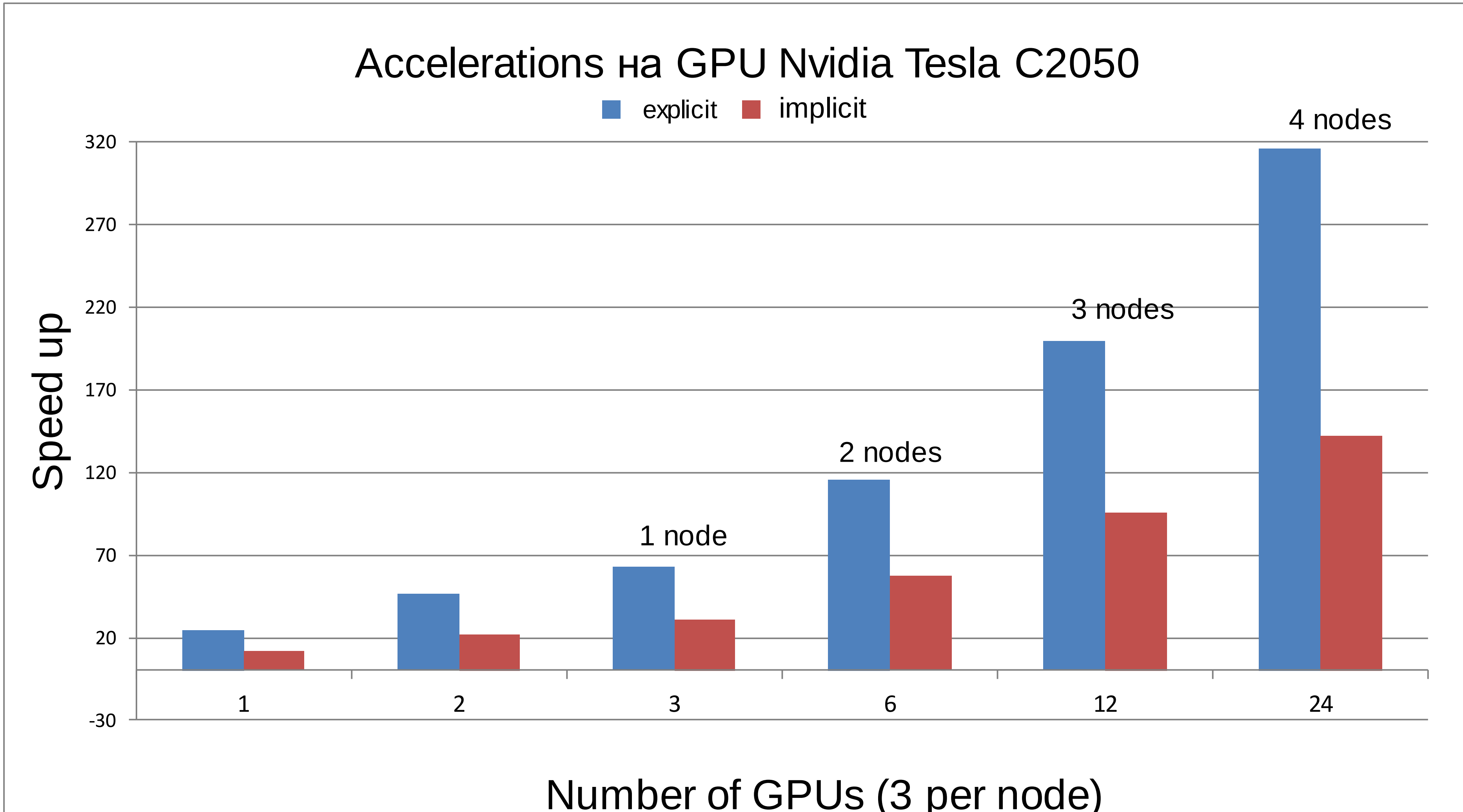
- Arrays are one-dimensional – **tt1,tt2**
- Variable number of "neighbors" – **ii**
- Links are specified by array – **jj**

```
do i = 1, np2
  nn = ii(i)
  nb = npa(i)
  if (nb.ge.0) then
    s1 = FS(xp2(i),yp2(i),tv)
    s2 = 0d0
    do j = 1, nn
      j1 = jj(j,i)
      s2 = s2 + aa(j,i) * tt1(j1)
    enddo
    s0 = s1 + s2
    tt2(i) = tt1(i) + tau * s0
  else if (nb.eq.-1) then
    tt2(i) = vtemp1
  else if (nb.eq.-2) then
    tt2(i) = vtemp2
  endif
  s0 = (tt2(i) - tt1(i)) / tau
  gt = DMAX1(gt,DABS(s0))
enddo
do i = 1, np2
  tt1(i) = tt2(i)
enddo
```

Results, 8 million nodes



Results, 8 million nodes



cite: http://dvm-system.org

mail: dvm@keldysh.ru