Technological Aspects of the Hybrid Parallelization with OpenMP and MPI

Oleg Bessonov(✉)

Institute for Problems in Mechanics of the Russian Academy of Sciences,
101, Vernadsky ave., 119526 Moscow, Russia
bess@ipmnet.ru

Abstract. In this paper we present practical parallelization techniques for different explicit and implicit numerical algorithms. These algorithms are considered on the base of the analysis of characteristics of modern computer systems and the nature of modeled physical processes. Limits of applicability of methods and parallelization techniques are determined in terms of practical implementation. Finally, the unified parallelization approach for OpenMP and MPI for solving a CFD problem in a regular domain is presented and discussed.

1 Introduction

Previously, only distributed memory computer systems (clusters) were available for parallel computations, and the only practical way of parallelization was the MPI distributed-memory approach (provided a user was privileged enough to have an access to such a system). Currently multicore processors have become widely available, and parallelization is no more an option as before. Users have to parallelize their codes because there is no other way to fully utilize the computational potential of a processor. Besides, cluster nodes are now built on multicore processors too. Thus, the shared-memory OpenMP approach as well as the hybrid OpenMP/MPI model become important and popular.

The progress in the computer development can be illustrated by comparing solution times of the CFD problem [1] with 10^6 grid points and 10^6 time-steps:

- 2005, 1-core processor – 280 h;
- 2009, 4-core processor – 30 h;
- 2011, cluster node with two 6-core processors – 11 h;
- 2013, cluster node with two 10-core processors – 4.5 h;
- 2015, 4 cluster nodes with two 10-core processors – 1.5 h.

We can see the acceleration by two orders of magnitude. This has become possible both due to the development of computers and implementation of new parallel methods and approaches.

In previous papers [1–3] we have analyzed parallelization methods from the mathematical, convergence and efficiency points of view. In this work we will consider practical parallelization techniques taking into account characteristics and limitations of modern computer systems as well as essential properties of modeled physical processes.

© Springer International Publishing AG 2017
V. Malyshkin (Ed.): PaCT 2017, LNCS 10421, pp. 101–113, 2017.
DOI: 10.1007/978-3-319-62932-2_9

2 Parallel Performance of Modern Multicore Processors

Modern multicore microprocessors belong to the class of throughput-oriented processors. Their performance is achieved in cooperative work of processor cores and depends both on the computational speed of cores and on the throughput of the memory subsystem. The latter is determined by the configuration of integrated memory controllers, memory access speed, characteristics of the cache memory hierarchy and capacity of intercore or interprocessor communications.

For example, a typical high-performance processor used for scientific or technological computations has the following characteristics:

- 6 to 12 computational cores, with frequencies between 2.5 and 3.5 GHz;
- peak floating point arithmetic performance 300–500 GFLOPS (64-bit);
- 4 channels of DDR4-2133/2400 memory with peak access rate 68–77 GB/s;
- hierarchy of cache memories (common L3-cache, separate L2 and L1);
- ability to execute two threads in each core (hyperthreading).

Many scientific or technological application programs belong to the memory-bound class, i.e. their computational speed is limited by the performance of the memory subsystem. Thus, with increasing the number of cores, it is necessary to make the memory faster (frequency) and/or wider (number of chsannels).

Importance of the memory subsystem can be illustrated by running several applications on computers with different configurations (Fig. 1): 8-core (3.0 GHz) and 6-core (3.5 GHz) processors with 4 memory channels (68 GB/s), and 6-core processor with 2 channels (34 GB/s). All processors belong to the same family Intel Core i7-5900 (Haswell-E). The first computer system has faster memory than two others. Application programs used in the comparison are the following:

- Cylflow: Navier-Stokes CFD code [1] (regular grid, 1.5 M grid points);
- CG AMG: Conjugate gradient solver with the multigrid preconditioner [2] (Cartesian grid in the arbitrary domain, sparse matrix, 2 M grid points);
- CG Jacobi: Conjugate gradient solver with the explicit Jacobi preconditioner [2] (the same grid).

It is seen from Fig. 1 that there is the saturation of the memory subsystem in all cases. For the CFD code, it is less visible: 4-channel computers look similarly, while the 2-channel system is 1.3 times slower. For the multigrid solver, some difference between first two systems appears, and the third one is about two times slower. For the explicit solver, effect is more significant: the 8-core computer additionally gains owing to its faster memory and larger cache, while the 2-channel system additionally loses. The maximum of performance is achieved in this case if only part of processor cores are active (5, 4 and 3 cores, respectively).

On two-processor configurations, the number of memory channels and their integral capacity is doubled. Due to this, performance of most memory-bound programs can be almost doubled (see example of 1.95-times increase in [3]).

On the other hand, hyperthreading usually doesn't help to such programs. In fact, for the above applications, running twice the number of threads with the active hyperthreading reduces performance by about 10%.

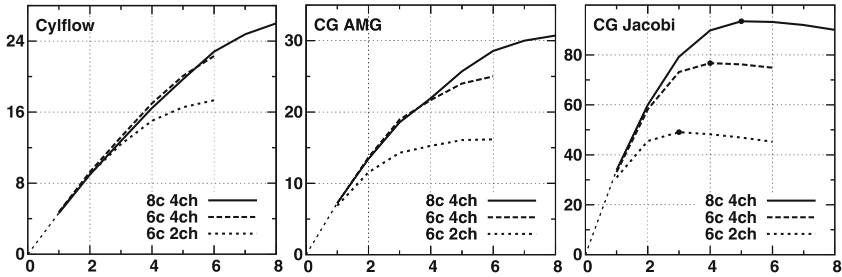

Fig. 1. Parallel performance of application programs (iterations per second) as a function of the number of threads on computers with different memory configurations

3 Properties of Explicit and Implicit Algorithms

There are two main classes of computational algorithms: explicit, with computations like $\boldsymbol{x} = A\boldsymbol{y}$ (multiplication of a matrix by a vector), and implicit, that look as $A\boldsymbol{x} = \boldsymbol{y}$ or $A\boldsymbol{x} \approx \boldsymbol{y}$ (exact or approximate solution of a linear system). Here vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ represent some physical quantities in the discretized domain, and matrix A corresponds to the discretization stencils applied to them.

In the analysis of explicit and implicit algorithms we will consider the lowest level of the computational method. For example, the explicit time integration scheme is an explicit algorithm per se, while the implicit integration scheme may be either resolved by an immediate implicit method, or solved by means of some iterative method employing any sort of the iterative scheme at the lower level.

Explicit algorithms act locally by stencils of the limited size and propagate information with low speed (one grid distance per iteration). Thus, they require $O(n)$ iterations for full convergence where n is the diameter of the domain (in terms of grid distances). On the other hand, implicit algorithms operate globally and propagate information much faster (approximate methods) or even instantly (direct solvers). For example, the Conjugate gradient method with the Modified Incomplete LU decomposition as a preconditioner needs $O(\sqrt{n})$ iterations [4].

Applicability of the methods depends on the nature of underlying physical processes. For example, incompressible viscous fluid flows are driven by three principal mechanisms with different information propagation speeds:

- convection: slow propagation, Courant condition can be applied CFL $= O(1)$ (one or few grid distances per time-step); using an explicit time integration scheme or an iterative solver with few iterations;
- diffusion: faster propagation (tens grid distances per time-step), well-conditioned linear system; using an iterative solver with explicit iterations or an Alternating direction implicit (ADI) solver;
- pressure: instant propagation, ill-conditioned linear system; using an iterative solver with implicit iterations or multigrid or a direct solver.

Parallel properties of computational methods strongly depend on how information is propagated. Iterations of explicit algorithms can be computed

independently, in any order, thus giving the freedom in parallelization. In contrast, implicit iterations have the recursive nature and can't be easily parallelized.

Below we will consider parallelization approaches for several variants of computational algorithms of the explicit, implicit and mixed type.

3.1 Natural Parallelization of Explicit Algorithms

The simplest way of parallelizing an explicit method is to divide a computational domain (geometric splitting) or a matrix A (algebraic splitting) into several parts for execution in different threads. It can be easily implemented in Fortran with the OpenMP extension. For example, one-dimensional geometric splitting by the last spatial dimension can be programmed with the use of `!$OMP DO` statement.

This sort of parallelization is very convenient and is almost automatic. However, it has a natural limitation: with larger number of threads, subdomains become narrow. This can increase parallelization overheads due to load disbalances and increased costs of accesses to remote caches across subdomain boundaries.

For this reason, two-dimensional splitting may become attractive. OpenMP has no natural mechanism for such splitting. Nevertheless, the nested loops can be easily reorganized by the appropriate remap of control variables of two outer loops (Fig. 2). Here, all changes of the original code are shown by capital letters.

The difference between one- and two-dimensional splittings is the placement of data belonging to subdomains. In the first case, each part of data is a single continuous 3-dimensional array. In the second case, data look as a set of 2D arrays, decoupled from each other. The size of each array is $N \times M/P$, where N and M are dimensions, and P is the splitting factor in the second dimension. For $N = M = 100$ and $P = 4$ this corresponds to 2500 data elements, or 20 KBytes.

```
!$OMP DO PRIVATE(IR,IZ)
    DO IP=0,15
      IR=IP/4+1
      IZ=IP-IR*4+1
      do k=NR(IR),MR(IR)
        do j=NZ(IZ),MZ(IZ)
          do i=1,nx
      w3(i,j,k)= . . .
          enddo i
        enddo j
      enddo k
    ENDDO IP
!$OMP END DO
```

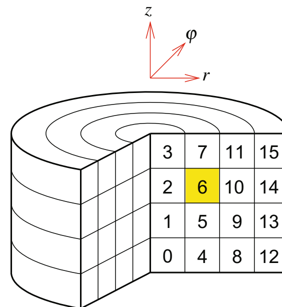

Fig. 2. Parallelization with the two-dimensional splitting

The main problem of split data is associated with the streamlined prefetch: this mechanism is efficient if arrays are long and continuous, and in the case

of piecewise-continuous arrays it stops after the end of each piece of data thus taking additional time for restart and reducing the overall efficiency of prefetch.

For the same reasons, it is not possible to efficiently implement 3-dimensional splitting by the similar way: the size of each piece of data would be N/P, or only 200 Bytes (for the above parameters).

Additional problem appears in the case of a Non-uniform memory computer (NUMA), consisting of two or more processor interconnected by the special links. Each processor controls its own part of memory. Logically, each thread can transparently access any memory location in a system, but remote accesses are much slower than local ones. For this reason, all data should be divided between processors as accurately as possible.

However, data are allocated in a particular processor's memory by pages of the typical size 4 KBytes. Therefore, some data on a boundary between subdomains always fall into the remote memory area (on average, half the page size). For the 1D splitting, this is much less than the size of a boundary array ($N \times M$, or 80 KBytes for the above parameters). In case of the 2D splitting, most boundary arrays are of the size N (only 800 Bytes). Thus, a significant part of each array would fall into the remote memory.

Therefore, for NUMA, splitting should be arranges such that interprocessor communications occur only across boundaries in the last spatial dimension. For the same reason it is not reasonable to use large pages (2 MBytes).

3.2 Parallelization Properties of Implicit Algorithms

Most often implicit algorithms are used as preconditioners for the Conjugate gradient (CG) method [3]. Typically, such preconditioners are built upon variants of the Incomplete LU decomposition (ILU) applied to sparse matrices. This decomposition looks as a simplified form of the Gauss elimination when fill-in of zero elements is restricted or prohibited.

Within the CG algorithm, this preconditioner is applied at each iteration in the form of the solution of a linear system $LU\boldsymbol{x} = \boldsymbol{y}$, that falls into two steps: $L\boldsymbol{z} = \boldsymbol{y}$ and $U\boldsymbol{x} = \boldsymbol{z}$.

These steps are recursive by their nature. There is no universal and efficient method for parallelizing ILU. One well-known approach is the class of domain decomposition methods [5], where the solution of the original global linear system is replaced with the independent solutions of smaller systems within subdomains, with further iterative coupling of partial results. However, this approach is not enough efficient because it makes the convergence slower or impossible at all.

For ill-conditioned linear systems associated with the action of pressure in the incompressible fluid it is important to retain convergence properties of the preconditioning procedure. This can be achieved by finding some sort of parallelization potential, either geometric or algebraic.

For domains of regular shape it is natural to discover a sort of the geometric parallelization. For example, Cartesian discretization in a parallelepiped produces a 7-diagonal matrix, that looks as a specific composition of three 3-diagonal matrices corresponding to three directions. It can be seen that the procedure of

twisted factorization of a 3-diagonal system can be naturally generalized to two or three dimensions [4]. Figure 3(a, b) shows factors of the LU-decomposition of a 5-diagonal matrix that corresponds to a 2D domain. Illustration of the computational scheme for three dimensions is shown on Fig. 3(c): the domain is split into 8 octants, and in each octant elimination of non-zero elements for the first step $\boldsymbol{Lz} = \boldsymbol{y}$ is performed from the corner in the direction inwards. For the second step $\boldsymbol{Ux} = \boldsymbol{z}$ data within octants are processed in the reverse order.

Thus, LU-decomposition in a 3D parallelepipedic domain can by parallelized by 8 threads with small amount of inter-thread communications and without sacrifying convergence properties of the iterative procedure.

Additional parallelization within octants can be achieved by applying the staircase (pipelined) method [6,7]. Here, each octant is split into two halves in the direction j (Fig. 3, d), and data in each half are processed simultaneously for different values of the index in the direction k (this looks like a step on stairs).

This method needs more synchronizations between threads, and its application is limited by the factor of 2 or (at most) 4. Thus the resulting parallelization potential for a parallelepipedic domain is limited by 16 or 32 threads.

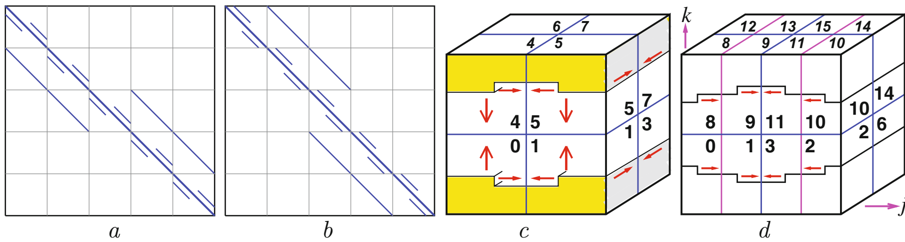

Fig. 3. Nested twisted factorization: factors of a 2D LU-decomposition (a, b); example of the parallel elimination of non-zero elements in a 3D domain (c); illustration of the staircase method (d)

For general domains and non-structured grids there is no more geometric symmetry. Thus the only way of parallelization is to find an algebraic potential. The idea is again to use twisted factorization. Figure 4 (left, center) shows factors of the LU-decomposition of a banded sparse matrix. Each factor consists of two parts, and most calculations in each part can be performed in parallel.

Unfortunately, this way allows to parallelize the solution for only two threads. Additional parallelization can be achieved by applying a variant of the pipelined approach, namely the block-pipelined method [8]. The idea of the approach is to split each part of a factor into pairs of adjacent trapezoidal blocks that have no mutual data dependences and can be processed in parallel (Fig. 4, right). As a result, parallelization of the Gauss elimination will be extended to 4 threads.

Performance of the block-pipelined method depends on the sparsity pattern of a matrix. If the matrix contains too few non-zero elements, the overall effect of the splitting may happen to be low because of synchronization overheads.

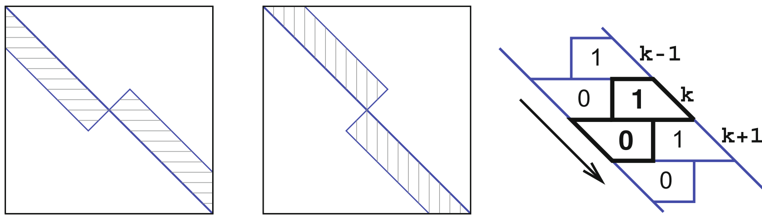

Fig. 4. Twisted factorization of a sparse matrix (left, center); splitting of a subdiagonal part of a factored matrix into pairs of blocks in the block-pipelined method (right)

The above examples demonstrate complexity of parallelization of implicit algorithms that strongly depend on the property of fast propagation of information in underlying physical processes.

3.3 Peculiarities of the Multigrid

There exists a separate class of implicit methods, multigrid, which possess very good convergence and parallelization properties. Multigrid solves differential equations using a hierarchy of discretizations. At each level, it uses a simple smoothing procedure to reduce corresponding error components.

In a single multigrid cycle (V-cycle, Fig. 5), both short-range and long-range components are smoothed, thus information is instantly transmitted throughout the domain. As a result, this method becomes very efficient for elliptic problems that propagate physical information infinitely fast.

1	Pre-smooth	$x_1 = S_1\,(x_0, b)$
2	Residual	$b_1 = b - Ax_1$
3	Restriction	$\tilde{b}_1 = R\,b_1$
4	Next level	$\tilde{A}\tilde{x}_2 \approx \tilde{b}_1$
5	Prolongation	$x_2 = P\tilde{x}_2$
6	Correction	$x_3 = x_1 + x_2$
7	Post-smooth	$x_0 = S_2\,(x_3, b)$

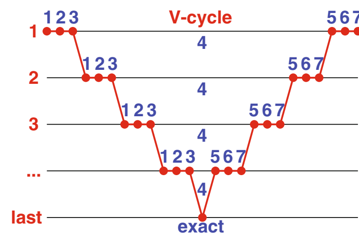

Fig. 5. Multigrid algorithm (left) and illustration of V-cycle (right) (Color figure online)

At the same time, multigrid can be efficiently and massively parallelized because processing at each grid level is performed in the explicit manner, and data exchanges between subdomains are needed only at the end of a cycle.

Here we consider the Algebraic multigrid (AMG) approach [2,9] which is based on matrix coefficients rather than on geometric parameters of a domain. This approach is applied in frame of Cartesian discretization in the arbitrary

domain. The resulting sparse matrices are stored in the Compressed Diagonal Storage (CDS) format [2] that is more efficient for processing on modern throughput processors than the traditional Compressed Row Storage (CRS).

The main computational operations in an AMG cycle are smoothing (iteration of the Gauss-Seidel or SOR method), restriction (fine-to-coarse grid conversion by averaging) and prolongation (coarse-to-fine conversion by interpolation).

Formally, iteration of the Gauss-Seidel method looks as an implicit procedure: $(D + L)\boldsymbol{x}_{k+1} = b - U\boldsymbol{x}_k$ (here D, L and U are diagonal, subdiagonal and superdiagonal parts of the matrix A in the equation $A\boldsymbol{x} = \boldsymbol{b}$). In order to avoid recursive dependences, the multicolor grid partitioning can be applied. For discretizations with 7-point stencils, two-color scheme is sufficient (red-black partitioning). With this scheme, the original procedure falls into two explicit steps: $D^{(1)}\boldsymbol{x}^{(1)}_{k+1} = \boldsymbol{b}^{(1)} - U\boldsymbol{x}^{(2)}_k$ and $D^{(2)}\boldsymbol{x}^{(2)}_{k+1} = \boldsymbol{b}^{(2)} - L\boldsymbol{x}^{(1)}_{k+1}$ (superscripts $^{(1)}$ and $^{(2)}$ refer to red-colored and black-colored grid points, respectively).

To ensure consecutive access to data elements, it is necessary to reorganize all arrays, i.e. to split them into "red" and "black" parts. After that, any appropriate parallelization can be applied, either geometric or algebraic. In particular, the algebraic splitting by more than 200 threads was implemented for Intel Xeon Phi manycore processor [2].

Similarly, the restriction (averaging) procedure can be parallelized. Implementation of the prolongation procedure is more difficult because different interpolation operators should be applied to different points of the fine grid depending on their location relative to the coarse grid points.

The above considerations are applied to the first (finest) level of the multigrid algorithm. Starting from the second levels, all discretization stencils have 27 points, and 8-color scheme becomes necessary. As a result, computations become less straightforward, with a proportion of indirect accesses.

On coarser levels of the algorithm, the number of grid points becomes not sufficient for efficient parallelization on large number of threads. This effect is most expressed at the last level. Usually, the LU-decomposition or the Conjugate gradient is applied at this level. However, these methods either can't be parallelized or involve very large synchronization overheads.

To avoid this problem, a solver based on the matrix inversion can be used. Here, the original last-level sparse matrix is explicitly inverted by the Gauss-Jordan method at the initialization phase. The resulting full matrix is used at the execution phase in the simple algorithm of matrix-vector multiplication. This algorithm is perfectly parallelized and doesn't require synchronizations.

Efficiency and robustness of the multigrid algorithm is higher if it is used as a preconditioner in the Conjugate gradient method. In this case, it becomes possible to use the single-precision arithmetic for the multigrid part of the algorithm without loosing the overall accuracy. Due to this, the computational cost of the algorithm can be additionally decreased because of reduced sizes of arrays with floating point data and corresponding reduction of the memory traffic.

Thereby, the multigrid method is very efficient and convenient for parallelization. However, it is very complicated and difficult for implementation, especially

for non-structured grids. Its convergence is not satisfactory in case of regular anisotropic grids (though it can be overcome by so-called semi-coarsening [9] when the grid becomes non-structured). In some cases, the behaviour of the method becomes uncertain. Finally, there is no reliable theory and procedure for systems of equations. Thus it is not a universal solution, and applicability of traditional methods remains wide.

3.4 Methods of Separation of Variables and ADI

For solving well-conditioned linear systems, the Alternating direction implicit (ADI) method can be used. If the original matrix is presented as $A = I+L$, where I is the unit matrix and $||L|| \ll 1$, then it can be approximately decomposed as $(I + L) \approx (I + L_x)(I + L_y)(I + L_z)$. The final procedure looks as the solution of several 3-diagonal systems. Also, 3-diagonal systems appear in the direct method of separation of variables for solving the pressure Poisson equation [1].

Parallelization of the solution of a 3-diagonal linear system can be done by applying the twisted factorization for 2 threads, or two-way parallel partition [1] for 4 threads (Fig. 6, left). In the latter method, twisted factorization is applied separately to the first and second halves of a matrix. After two passes (forward and backward) the matrix has only the main diagonal and the column formed due to fill-in. To resolve this system, additional substitution pass is needed.

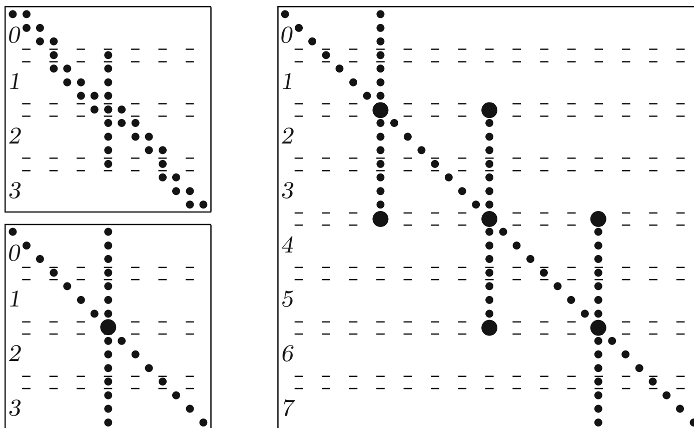

Fig. 6. Illustration of the two-way parallel partition method: matrix view for 4 threads (after the first and the second passes) and for 8 threads (after the second pass)

The two-way parallel partition method can be naturally extended for 8 threads (Fig. 6, right). After two passes of the twisted factorization, the matrix has in this case more complicated structure with 3 partially filled columns. Three equations of the matrix form the reduced linear system (its elements are shown by bold points) that should be resolved before the substitution pass.

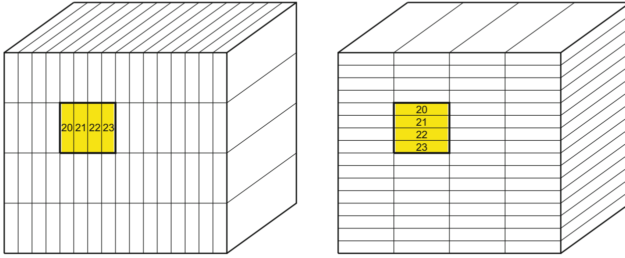

Fig. 7. Two variants of the splitting of a computational domain: 4×16 and 16×4

Computational expenses of this method increase rapidly with the number of threads. Also, there is the sharp increase of the number of exchanges in case of the distributed memory (MPI) parallelization. Therefore, the reasonable level of parallelization for 3-diagonal matrices is limited by 4 or (at most) by 8 threads.

Such limitations can restrict the total level of parallelization. If it is necessary to solve 3-diagonal linear systems in two outer directions with the limit of 4 threads (in each direction), then the total limit will be $4 \times 4 = 16$. To override this, we can use different splittings for different parts of the algorithm. For example, if the splitting of a computational domain is 4×16, all parts of the algorithm can be parallelized except the 3-diagonal systems in the last direction (Fig. 7, left). For solving them, we will use another splitting 16×4 (Fig. 7, right).

For both variants of splitting in this example, there are groups of 4 threads that share the same data (as indicated by shaded rectangles on Fig. 7). Switching from the first splitting to the second one looks like a transposition of data. For the shared memory environment, no real transposition occurs and data are simply accessed in another order. However, for the distributed memory, costly data transfers would take place. For this reason, such groups of subdomains should never be split between cluster nodes. As a consequence, the level of the distributed memory parallelization is limited by the level of parallelization of a 3-diagonal linear system in the last direction, i.e. by 4 or 8 cluster nodes.

The maximal reasonable level of parallelization of the above approach for two-dimensional splitting is between $32 \times 4 = 128$ and $32 \times 8 = 256$, depending on the parallelization scheme for 3-diagonal systems and taking into account the reasonable limitation of 32 threads for each direction.

4 Unified Parallelization Approach for OpenMP and MPI

In the comparison of parallelization environments, it is important to pay attention on the basic characteristics of a distributed memory computer system:

- internode communication speed: $O(1)$ GB/s;
- memory access rate: $O(10)$ GB/s;
- computational speed: $O(10^2)$ GFLOPS, or $O(10^3)$ GB/s.

In fact, not all computations require memory accesses, thus the computational speed expressed in memory units is close to $O(10^2)$ GB/s. Nevertheless, it is clear that the memory subsystem is one order of magnitude slower than the processor, and communications are two orders of magnitude slower.

Therefore exchanges in the MPI model should be kept to a minimum and allowed only on boundaries between subdomains. This applies also to the use of MPI in a shared memory (or multicore) computer though to the less extent. It means that transmission of full (3D) data arrays by MPI should be avoided.

For clusters, it is optimal to use the hybrid parallelization with OpenMP and MPI. Programming with MPI requires serious reorganization of the code: it is necessary to change the natural allocation of data, replacing each monolithic data array with several subarrays in accordance with the splitting and re-adjusting the addressing scheme. These subarrays should be logically overlapped, i.e. contain additional layers (e.g. ghost elements for calculating derivatives). Such complications, together with the need to organize explicit data exchanges between cluster nodes, make development and debug of a code much more difficult. At last, it may become necessary to develop and support two (or more) versions of a code.

These complications can be partly avoided if the splitting between cluster nodes is done only by the last spatial direction. Then, only the numeration in this direction has to be changed. For example, if a 3D domain of the size $L \times M \times N$ is split into 4 subdomains by the last dimension such as $K = N/4$, each MPI process will have to allocate data arrays of the dimensions `(L,M,0:K+1)`. Here, bounds of the last index are expanded to support overlap in such a way, that the slice K of the array in a process corresponds to the slice 0 in the next process, and the same applies to the slices K+1 and 1 in these processes (Fig. 8, left).

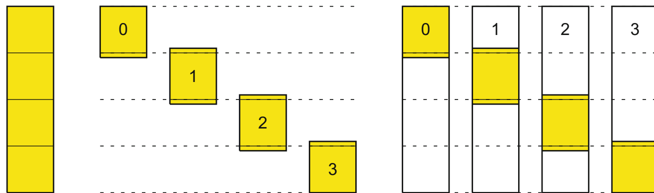

Fig. 8. Illustration of the allocation of a data array in MPI processes: independent subarrays with overlapped areas (left); subsets in the unified address space (right)

As a next step, we can organize the unified address space for the MPI-parallelized program. The simple way is to logically allocate full data array in each MPI process rather than its selected part as in the above example (Fig. 8). This will not lead to the unnecessary occupation of memory because in modern processors only those memory pages are physically allocated which are accessed in a program. In this scheme, addressing of array elements remains unchanged.

In case of dynamic memory allocation, we can avoid logical allocation of full arrays: in Fortran it is possible to allocate an array in a particular process

indicating exact bounds of the last index, e.g. `ALLOCATE (X(L,M,2*K:3*K+1))`. This example corresponds to the process 2 on Fig. 8 (right).

With unified address space, character of the two-dimensional splitting of the computational domain and structure of DO-loops in the hybrid OpenMP/MPI parallelization become very similar to those of the pure OpenMP approach: it is enough to remap bounds of control variables of two outer loops (as shown on Fig. 2) and, after completion of loops, perform exchanges with neighbour processes for sending (receiving) boundary elements of data arrays. Thus, the only difference is that some subdomain boundaries by the last spacial direction become the internode boundaries that require MPI communications.

In order to simplify and unify the computational code, it is worth to organize special "high level" routines for doing exchanges rather than to calls MPI functions directly. Depending on a process number, these routines can determine addresses of data to be transmitted and directions of transmissions. Depending on the total number of MPI processes, they can decide whether the particular data transmission should take place at all. Thus, the code becomes invariant with respect to the number of processes and to the fact of the use of MPI itself.

By avoiding explicit calls of MPI functions, the principal parts of a program become MPI-independent and it becomes possible to compile these parts by any compiler which is not necessarily integrated into an MPI environment. Only a collection the above "high level" routines (grouped into a separate file) works with MPI and should be compiled in the appropriate environment. For convenience, several variants of a file with these routines can be made (for a single-node run without MPI, for 2 nodes, for 4 nodes etc.). By this way, the unified approach for pure OpenMP and for hybrid OpenMP/MPI model is established.

This approach was used for the hybrid parallelization of the CFD code for modeling incompressible viscous flows in cylindrical domains (see [1] for the OpenMP-only version). Below are parameters of a problem for solving on four cluster nodes with two 10-core processors each, 80 cores (threads) total:

- problem size $192 \times 160 \times 160$ (φ, z, r);
- general splitting 4×20 (z, r), size of a subdomain $192 \times 40 \times 8$;
- specific splitting 20×4 (for solving 3-diagonal systems in the last direction), size of a subdomain $192 \times 8 \times 40$.

These splitting parameters correspond to the requirements set out in Subsects. 3.1 and 3.4.

In order to run the above problem correctly, all necessary MPI and OpenMP parameters should be set, such as OMP_NUM_THREADS environment variable for parallelization within a cluster node, number of nodes and their list in the "mpirun" directive, appropriate binding of threads to processor cores in the "taskset" utility etc.

5 Conclusion

In this paper we have considered different practical and technological questions of the parallelization for shared and distributed memory environments. Most

examples were done for the geometric approach of parallelization but in many cases they can be extended to the algebraic and other sorts of decomposition.

Special attention was paid on methods with limited parallelization potential that is associated with the essential properties of underlying processes, namely fast propagation of physical information relative to the temporal scale of a numerical method. It is clear that explicit methods while possessing good parallelization properties are not enough efficient for solving such problems. Thus implicit methods remain very important despite they are not always convenient for optimization in general and for parallelization in particular. For this reason, approaches for low and medium scale parallelization are still in demand.

Acknowledgements. This work was supported by the Russian Foundation for Basic Research (projects 15-01-06363, 15-01-02012). The work was granted access to the HPC resources of Aix-Marseille Université financed by the project Equip@Meso (ANR-10-EQPX-29-01) of the program Investissements d'Avenir supervised by the Agence Nationale pour la Recherche (France).

References

1. Bessonov, O.: OpenMP parallelization of a CFD code for multicore computers: analysis and comparison. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 13–22. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23178-0_2
2. Bessonov, O.: Highly parallel multigrid solvers for multicore and manycore processors. In: Malyshkin, V. (ed.) PaCT 2015. LNCS, vol. 9251, pp. 10–20. Springer, Cham (2015). doi:10.1007/978-3-319-21909-7_2
3. Bessonov, O.: Parallelization properties of preconditioners for the conjugate gradient methods. In: Malyshkin, V. (ed.) PaCT 2013. LNCS, vol. 7979, pp. 26–36. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39958-9_3
4. Accary, G., Bessonov, O., Fougère, D., Gavrilov, K., Meradji, S., Morvan, D.: Efficient parallelization of the preconditioned conjugate gradient method. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 60–72. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03275-2_7
5. Saad, Y.: Iterative Methods for Sparse Linear Systems. PWS Publishing, Boston (2000)
6. Bastian, P., Horton, G.: Parallelization of robust multi-grid methods: ILU-factorization and frequency decomposition method. SIAM J. Stat. Comput. **12**, 1457–1470 (1991)
7. Elizarova, T., Chetverushkin, B.: Implementation of multiprocessor transputer system for computer simulation of computational physics problems (in Russian). Math. Model. **4**(11), 75–100 (1992)
8. Bessonov, O., Fedoseyev, A.: Parallelization of the preconditioned IDR solver for modern multicore computer systems. In: Application of Mathematics in Technical and Natural Sciences: 4th International Conference. AIP Conference Proceedings, vol. 1487, pp. 314–321 (2012)
9. Stüben, K.: A review of algebraic multigrid. J. Comput. Appl. Math. **128**, 281–309 (2001)