Parallelization of an Advection-Diffusion Problem Arising in Edge Plasma Physics Using Hybrid MPI/OpenMP Programming

Matthieu Kuhn[1(✉)], Guillaume Latu[2], Nicolas Crouseilles[3], and Stéphane Genaud[1]

[1] ICube, University of Strasbourg, Strasbourg, France
matthieu.kuhn@inria.fr
[2] CEA, IRFM, 13108 Saint-Paul-lez-Durance, France
[3] INRIA Rennes, IPSO Project and IRMAR, University of Rennes 1, Rennes, France

Abstract. This work presents a hybrid MPI/OpenMP parallelization strategy for an advection-diffusion problem, arising in a scientific application simulating tokamak's edge plasma physics. This problem is the hotspot of the system of equations numerically solved by the application. As this part of the code is memory-bandwidth limited, we show the benefit of a parallel approach that increases the aggregated memory bandwidth in using multiple computing nodes. In addition, we designed some algorithms to limit the additional cost, induced by the needed extra inter nodal communications. The proposed solution allows to achieve good scalings on several nodes and to observe 70 % of relative efficiency on 512 cores. Also, the hybrid parallelization allows to consider larger domain sizes, unreachable on a single computing node.

Keywords: Hybrid MPI/OpenMP · Advection-Diffusion · Plasma physics

1 Introduction

In this work, we present a hybrid MPI/OpenMP parallelization strategy for an advection-diffusion problem, arising in a scientific application simulating tokamak's edge plasma physics called Emedge3D. In a previous work (see [6]), we presented parallelization using OpenMP, but also several optimizations for a shared memory architecture. Enhancing this previous version is needed because of the memory-bound aspect of the application. Some optimizations were described that improved data access patterns, leading to better data locality. Even if one part of the code was successfully optimized with techniques such as loop tiling, the most consuming part of the code still suffered from a lack of performance on a 64-cores shared memory node. However, results were satisfying for a smaller node of 12 cores (bi-socket Intel X5675 @ 3.06 GHz).

M. Kuhn—Currently at INRIA Bordeaux Sud Ouest, HiePACS Project.

J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 545–557, 2015.
DOI: 10.1007/978-3-662-48096-0_42

Hence, we propose here to add a level of parallelism. To do so, we combine the OpenMP paradigm to the MPI standard to target distributed memory architectures. But, to achieve good efficiency, we show that several invasive modifications of the code have to be implemented. For example, considering $1D \times 1D$ FFT versus 2D FFT routines divides the communication volume by a factor 2. Therefore, we consider in this paper an advection-diffusion equation. Even if it is a reduced problem compared to the model used in Emedge3D, the Poisson bracket (advection part) and the diffusion part (which is anisotropic) are the most challenging and time consuming ones in Emedge3D.

In this paper, we consider the unknown temperature $T = T(t, x, y, z)$, the operator $\nabla \cdot (A\nabla .)$ (where $\nabla = (\partial_x, \partial_y, \partial_z)$), and A a 3×3 matrix to be explicited. This operator is coupled with an advection operator $\{\phi, .\} = \partial_x \phi \partial_y . - \partial_y \phi \partial_x .$, also called Poisson bracket. We consider the equation

$$\partial_t T + \{\phi, T\} = \nabla \cdot (A \nabla T), \quad x, y, z \in [-1, 1], t \geq 0, \tag{1}$$

with periodic boundary conditions along y, z, Dirichlet boundary condition along x (classical in tokamak geometry) and ϕ the electric potential, assumed to be given here.

In the following, we first describe the advection-diffusion problem addressed in this work while providing the related work. Then, we present the numerical methods and our validation test case. After that, we detail the proposed parallel solutions to solve the advection-diffusion equation. Lastly, we give a performance analysis of the best known solution.

2 The Advection-Diffusion Problem

Advection-diffusion problems are widely used in physics models (see [2,7]). Their numerical approximation often requires recent techniques (see [11] for example). However, most of the time, the diffusion operator is restricted to a 3D Laplacian, whereas several relevant applications requires an anisotropic diffusion.

The problem considered here is 3D in space and time-dependant. The advection part is 2D, in the plane (x, y), and consists in a Poisson bracket. The same operator can be found in Emedge3D's model (see [2]). For the diffusion part, the diffusion matrix A depends only on the spatial dimension x, corresponding to the radial direction in the SLAB geometry of Emedge3D.

As in Emedge3D, two kinds of discretization are considered to approximate spatial operators. First, a *semi-spectral* representation of 3D unknown is used to compute the diffusion part, in which y and z directions are expressed in the Fourier basis, and x in the real basis. Second, a representation in the full real space for the 3 directions is employed to compute the Poisson bracket. This kind of discretization is often encountered in nuclear fusion codes. As an illustration, we can cite GKV and GENE (see [8] and [4]), and also XTOR and JOREK (see [7] and [5]). These codes also try to take benefits of parallelization on both shared and distributed memory systems.

In Emedge3D, the execution time of a simulation is mostly driven by the pressure equation (see [2,3,9]), which is similar to Eq. (1). The next section aims to describe the numerical methods employed to solve the advection-diffusion problem given by the following equation:

$$\partial_t T + \{\phi, T\} = \nabla \cdot (A_x \nabla T) \text{ with } A_x = \begin{pmatrix} a(x) & 0 & 0 \\ 0 & b(x) & d(x) \\ 0 & d(x) & c(x) \end{pmatrix}. \tag{2}$$

3 Numerical Methods and Test Case

3.1 Spatial Discretization

This part deals with the spatial discretization used in the code. It presents first the numerical method to compute the advection and then the spatial scheme to solve the diffusion. These methods are extracted from Emedge3D.

The *advection term* is computed in the physical space (and not in semi-spectral representation) with a finite difference method. Indeed, when the Poisson bracket operator is explicited in semi-spectral representation, it leads to a convolution which has a quadratic computational complexity $\Theta(n^2)$ (assuming $n = N_y\, N_z$). This is why the discretization has to change by using FFT. Hence, it results in a more desirable linearithmic computational complexity $\Theta(n\, \log(n))$. This method is commonly used on nonlinear terms in case of a semi-spectral discretization (see [10]). An Arakawa scheme of order 2 (see [1]) is employed to compute this Poisson Bracket. This numerical method is often considered in the plasma physics community because of its robustness and conservation properties. This discretization induces the computation of a 2D stencil in the plane (x, y), the dimension z acts as a parameter. This spatial scheme has already been studied in one of our former work (see [6]).

The *diffusion operator* is characterized by the diffusion matrix A_x given by (2). As it only depends on the radial dimension x, it can be easily written in the semi-spectral form. Hence, the unknown T is expressed in the Fourier basis in the y and z directions. This implies manipulation of quantities of the form:

$$\hat{T}_{i,m,n} := \hat{T}(x_i, m, n) = \int_{\mathbb{R}} \int_{\mathbb{R}} T(x_i, y, z) exp(-i(my + nz)) dy\, dz,$$

where x_i stands for the grid points in the radial direction and (m, n) the Fourier mode. The diffusion operator is solved with a classical finite volume method of order 2 in the x direction, and spectral method in y and z directions:

$$\begin{aligned} \nabla \cdot (A_x \nabla \hat{T})_{|i,m,n} = a(x_{i+1/2}) \frac{\hat{T}_{i+1,m,n} - \hat{T}_{i,m,n}}{\Delta x^2} - a(x_{i-1/2}) \frac{\hat{T}_{i,m,n} - \hat{T}_{i-1,m,n}}{\Delta x^2} \\ - (b(x_i)m^2 + c(x_i)n^2 + 2d(x_i)mn)\hat{T}_{i,m,n}, \end{aligned}$$

where Δx denotes the spatial step in the direction x and $x_{i\pm1/2} = x_i \pm \Delta x/2$.

3.2 Temporal Discretization

The implemented time integration scheme uses an operator splitting between advection and diffusion terms. It is due to the different spatial discretizations employed to solve these operators. We denote by $T^k = T(t^k, x, y, z)$ the solution at time $t^k = k\Delta t$ in the direct representation, with Δt the time step; and $\hat{T}^k = \hat{T}(t^k, x, m, n)$ in the semi-spectral representation, where m (respectively n) stands for the mode number in the poloidal (respectively toroidal) direction. Hence, we first consider the advection $\partial_t T + \{\phi, T\} = 0$, that we decide to solve with a classical (explicit) Euler method $T^* = T^k + \Delta t \left\{\phi, T^k\right\}$.

The second step consists in solving the diffusion part $\partial_t \hat{T} = \nabla \cdot (A_x \nabla \hat{T})$. Recall the diffusion is solved in the semi-spectral space. Hence, the temporal scheme associated with the diffusion part is also applied in this representation. The Euler method to solve this part writes: $\hat{T}^{k+1} = \hat{T}^* + \Delta t \nabla \cdot (A_x \nabla \hat{T}^*)$.

Notice this scheme is referred as the Lie splitting, which is of first order. It can be upgraded to higher orders by using Strang splitting method. Also, as it is an explicit method, a stability condition is imposed on the value of Δt. The more restrictive stability condition comes from the diffusion operator. To bypass this limitation, it is possible to implement implicit or (well chosen) semi-implicit method (see [11] for example).

3.3 Analytical Test Case

This part gives a three dimensional analytical test case used to validate the numerical methods presented earlier and the parallelization implementations. The technique employed to construct our test case is called the Method of Manufactured Solution (MMS). The equation to solve is:

$$\partial_t T + \{\phi, T\} = \nabla \cdot (A_x \nabla T) + f, \tag{3}$$

where $\phi = \phi(x, y) = \cos(\pi x)\ \cos(\pi y)$ and $f = f(t, x, y, z)$ is a given source function added to perform the MMS. The solution we choose to reach for this test case writes:

$$T(t, x, y) = 1 + \sin(\pi x) \sin(\pi y) \sin(\pi z) e^{-t}, \tag{4}$$

where $x, y, z \in [-1, 1], t \geq 0$. For the matrix A_x, we consider the functions:

$$\begin{aligned} a(x) &= (2 + \sin(\pi x)), b(x) = (2 + \sin(\pi x))^2, \\ c(x) &= (2 + \cos(\pi x))^2, d(x) = (2 + \sin(\pi x))(2 + \cos(\pi x)). \end{aligned}$$

Then, (4) is an analytical solution of (3) with the computed source term:

$$\begin{aligned} f(t, x, y, z) = &-(T - 1) + \partial_x \phi \partial_x T - \partial_y \phi \partial_y T + b(x)\pi^2 (T - 1) + c(x)\pi^2 (T - 1) \\ &- a'(x)\partial_x T - a(x)\partial_x^2 T - 2d(x)\partial_{y,z}^2 T. \end{aligned}$$

4 Parallelization MPI/OpenMP

In this section, we first introduce the sequential algorithm description. Then, we explore the parallelization potential in the case of a distributed memory architecture. To finish, we present the OpenMP and the hybrid MPI/OpenMP parallelization of the code.

4.1 Sequential Algorithm

This part details the organization of the time loop in our simulation code, given by Algorithm 1. The algorithm for the advection-diffusion can be decomposed into 4 parts: the advection, the diffusion, the Fourier transforms and the transpositions of data in memory. In order to perform one temporal iteration, two arrays are used: $T_1[N_z, N_y, N_x]$ to store the value of the time evolutive temperature and $T_2[N_z, N_y, N_x]$ to store temporary results. In both arrays, data are in semi-spectral data representation. Here, the notation $T_1[z, y, x]$ refers to the value stored at position $z * (N_y * N_x) + y * (N_x) + x$, with N_d the number of points in direction d. As the advection source term f_{adv} and the diffusion one f_{diff} are known analytically, they are computed on the fly.

Discretization changes (real to semi-spectral and inverse) are encapsulated into the diffusion step. Hence, the diffusion step divides into three parts, detailed in Algorithms 2 and 3. Algorithm 2 consists in Fourier transforms in forward direction (from $\mathbb{R}$ to $\mathbb{C}$) in dimension y. In this Algorithm, $\text{buffer}_y[*]$ and $\text{buffer2}_y[*]$ are buffers of size N_y, used to store $(z, y = *, x)$ slices contiguously into the memory. This improves the temporal locality. The $*$ notation denotes an operation along all the points of the given dimension. Endly, notation $\hat{T}$ means that data are in semi-spectral representation ($\hat{T} \in \mathbb{C}$). Notice that storage dimensions order changes between input and output, going from $[z, y, x]$ to $[y, z, x]$. We compute this on the fly because the diffusion step and the FFT in z direction are in a same external loop on y.

Algorithm 3 details Fourier transforms in z direction and the diffusion computations. Here, the storage location $\text{buffer}_z[*]$ is an array of size N_z, used to

Algorithm 1. One time loop iteration

Input: $T_1 = T(t^n)$

Algorithm:

$T_2 \leftarrow \nabla \cdot (A_x \nabla T_1)$: Diffusion
$T_1 \leftarrow T_1 + \Delta t T_2$: Euler scheme
$T_1 \leftarrow T_1 + \Delta t f^n_{\text{diff}}$: Diffusion source
$T_2 \leftarrow \{\phi, T_1\}$: Advection operator
$T_1 \leftarrow T_1 - \Delta t T_2$: Euler scheme
$T_1 \leftarrow T_1 + \Delta t f^n_{\text{adv}}$: Advection source

Output : $T_1 = T(t^{n+1})$

Algorithm 2. FFT forward y

Input: $T_1[z, y, x] = T^n$

Algorithm:

for all z **do**
  for all x **do**
    $\text{buffer}_y[*] \leftarrow T_1[z, *, x]$
    $\text{buffer2}_y[*] \leftarrow \text{FFT}(\text{buffer}_y[*])$
    $\hat{T_1}[*, z, x] \leftarrow \text{buffer2}_y[*]$
  end for
end for

Output : $\hat{T_1}[y, z, x] = \hat{T}^n$

temporarily store Fourier representation along z direction. Notice that these FFT are also applied in-place, in order to maximize temporal locality on the buffer. Buffers $\text{in}_{xz}[x, *]$ and $\text{out}_{xz}[x, *]$ aim to store 2D (x, z) slices, $\text{in}_{xz}[x, *]$ for the input of the diffusion computation and $\text{out}_{xz}[x, *]$ for the output.

The backward Fourier transform on y dimension (from $\mathbb{C}$ to $\mathbb{R}$) is very similar to Algorithm 2 and is not presented here.

4.2 Parallelization Potential

For simplicity (implementation, readability and maintenance of the code done by the physicists), we choose to consider algorithms which do not need ghost cells between MPI processes.

The advection is solved in the direct representation (real space) of the unknown. It consists in a 2D stencil on variables x, y. The third direction z acts as a parameter here, and so it is a good candidate as parallelization axis. The diffusion part is solved in the semi-spectral representation of the unknown. It consists of a stencil in the x direction. Hence, it allows easier parallelization along y or z axes.

Between the two last operators, it seems natural to change the domain decomposition: arrays are parallelized with MPI along z for the advection and along y for the diffusion part. Moreover, another distribution change occurs at the same time to switch from semi-spectral to full real representation. It is computed via Discrete Fourier Transforms (DFT), with the FFTW3 library. These FFT act on the plane (y, z), corresponding to toroidal and poloidal directions of the tokamak geometry.

Table 1. Parallelization potential on distributed memory architectures. Dependencies include read statements (ghost cells not considered).

Step	Axe	Dependencies at (i, j, k)	Parallelization considered
Advection	x	$i-1, i, i+1$	no
	y	$j-1, j, j+1$	no
	z	k	yes
Diffusion	x	$i-1, i, i+1$	no
	y	j	yes
	z	k	yes
FFT 1D y	x	i	yes
	y	$j = *$	no
	z	k	yes
FFT 1D z	x	i	yes
	y	j	yes
	z	$k = *$	no

Instead of computing the FFT with the 2D functions proposed by the FFTW3 library (as in the Emedge3D code), we decide to compute FFT dimension by dimension (using 1D functions). It presents 3 major advantages:

- it operates on smaller data volumes, allowing more benefit from cache effects,
- it permits a larger set of possibilities for the parallelization on distributed memory architecture, as we will see afterwards,
- it does not imply a loss of performance in the sequential case (even if an additionnal transposition is needed). In particular, it allows to reuse data loads between FFT 1D and other parts of the algorithm.

Finally, as parallelization axes change between the different parts of the code, it remains to perform transpositions and redistributions of data, in order to have needed data locally on the computation node. These transpositions will depend on the chosen algorithm, and in particular on the way FFT are computed. Table 1 gives a summary of the last exposed parallelization possibilities.

4.3 OpenMP Parallel Version

Hereafter, an OpenMP parallel solution is introduced. Typically, the code consists in applying spatial operators compounded of loop nests of depth 3 (one

Algorithm 3. FFT$^{\pm 1}$ z direction and diffusion

Input: $\hat{T}[y,z,x] = \hat{T}^n$

Algorithm:

for all y **do**
  for all x **do**
    $\text{buffer}_z[*] \leftarrow \hat{T}[y,*,x]$
    $\text{buffer}_z[*] \leftarrow \text{FFT}(\text{buffer}_z[*])$
    $\text{in}_{xz}[x,*] \leftarrow \text{buffer}_z[*]$
  end for
  for all x **do**
    $\text{out}_{xz}[x,*] \leftarrow \text{in}_{xz}[x,*] + \Delta t \nabla \cdot (A_x \nabla \text{in}_{xz}[x,*])$
  end for
  for all x **do**
    $\text{buffer}_z[*] \leftarrow \text{out}_{xz}[x,*]$
    $\text{buffer}_z[*] \leftarrow \text{FFT}^{-1}(\text{buffer}_z[*])$
    $\hat{T}[y,*,x] \leftarrow \text{buffer}_z[*]$
  end for
end for

Output : $\hat{T}[y,z,x] = \hat{T}^n + \Delta t \nabla \cdot (A_x \nabla \hat{T}^n)$

Algorithm 4. Comms, FFT$^{\pm 1}$ z direction and diffusion

Input: Z_dist$[y,z,x] = \hat{T}(t^n)$

Algorithm:

pid $\leftarrow$ current process rank
for all y local to process pid **do**
  comm_{zy}: Y_dist[y,*,*] $\leftarrow$ Z_dist[y,*,*]
  for all x **do**
    $\text{buffer}_z[*] \leftarrow \text{Y\_dist}[y,*,x]$
    $\text{buffer}_z[*] \leftarrow \text{FFT}(\text{buffer}_z[*])$
    $\text{in}_{xz}[x,*] \leftarrow \text{buffer}_z[*]$
  end for
  for all x **do**
    $\text{out}_{xz}[x,*] \leftarrow \text{in}_{xz}[x,*] + \Delta t \nabla \cdot (A_x \nabla \text{in}_{xz}[x,*])$
  end for
  for all x **do**
    $\text{buffer}_z[*] \leftarrow \text{out}_{xz}[x,*]$
    $\text{buffer}_z[*] \leftarrow \text{FFT}^{-1}(\text{buffer}_z[*])$
    $\text{Y\_dist}[y,*,x] \leftarrow \text{buffer}_z[*]$
  end for
  comm_{yz}: Z_dist[y,*,*] $\leftarrow$ Y_dist[y,*,*]
end for

Output : Z_dist$[y,z,x] = \hat{T}(t^n) + \Delta t \nabla \cdot (A_x \nabla \hat{T}(t^n))$

for each spatial dimension). The parallelization strategy resides in distributing the outermost loops. The clause `collapse(2)` is used in order to combine iterations of two successive loops. All steps are parallelized with OpenMP. Regarding the advection and the computations of the source terms, arrays are stored in order z, y and x (C-like notation). Parallelization occurs on z and y dimensions. Concerning the diffusion part, the FFT in y direction are parallelized along z axis too (see Algorithm 2). For the FFT part in z and the diffusion operator (see Algorithm 3), the parallelization directive is on the intermediate loops on dimension x. Notice the loops on dimension x can not be trivially merged, because of a Write/Read dependency between the FFT and the diffusion parts. However, as computations are coupled in a same y loop, the 2D slice computed for each y index is small enough to fit in cache (L3 or L2).

4.4 Hybrid MPI/OpenMP Parallel Version

This part proposes a hybrid MPI/OpenMP parallel version of the code that lowers the volume of communications.

The algorithm remains close to the OpenMP version, but with data and outermost spatial loops distributed on the MPI processes. Hence, there are two dimensions along which data are distributed. The first one is z direction. Data are stored in $[z, y, x]$ order, and 2D $[y, x]$ slices are uniformly distributed on the different processes. This distribution addresses the Arakawa method, the source terms computations and FFT on y dimension. The second one is y direction. This is the case for Algorithm 3, (FFT on z and diffusion). Indeed, to compute 1D FFT for direction z, each process must have all the points in that direction.

The two domain decompositions lead to two transposition steps implying communications. These communications are added to Algorithm 3, using non-blocking subroutines, in order to minimize communication overhead. Several versions have been tested, but only the fastest one is presented. The communications are performed within the y loop of Algorithm 3 as we will see afterwards.

Algorithm 4 gives the communication steps, coupled with the FFT in direction z and the diffusion operator. In this Algorithm, notation comm_{zy} corresponds to z to y transpose step (and inverse for comm_{yz}). Finally, Algorithm 5 describes how to transpose from a distribution in z to a distribution in y for one given index iy. The inverse transformation is not detailed as it is completely symmetric.

5 Performance Analysis

This section presents the numerical results and performances obtained for the algorithms detailed in Sect. 4. For each run, 10 temporal iterations were performed, with a $(N_x, N_y, N_z) = (256, 256, 128)$ grid for mono-node tests and both $(256, 256, 128)$ and $(1024, 1024, 512)$ for the multi-node case.

Tests were performed on two parallel computers: the Rheticus cluster based at Aix-Marseille University, France, compounded of 1152 cores organized in 96 nodes of 2 bi-socket X5675; and the Helios cluster based in Rokkasho, Japan at the International Fusion Energy Research Center, compounded of 2 bi-socket Xeon E5-2600 nodes. In terms of configuration, we used Intel compiler together with Open MPI version 1.6.3 and the FFTW3 library in version 3.3. Source codes were compiled with `-O2 -axSSE4.2` flags.

Algorithm 5. Transpose $z \rightarrow y$: comm_{zy}

Input: Z_dist[NY,NZloc,NX],iy
Algorithm:
  pid ← current process rank
  for all process p ≠ pid **do**
    Irecv(Y_dist[iy,NZloc × p,NX]) from p
  end for
  for all process p ≠ pid **do**
    Isend(Z_dist[iy × p,NZloc,NX]) to p
  end for
  Wait for all communication to finish
Output: Y_dist[iy,NZ,NX]

In the following, programs performances are presented using notations: `NCU` the number of computing units (with `NCU` = `NTH` × `NP`), `NTH` the number of OpenMP threads, `NP` the number of MPI processes, `t` the execution time, `SU` the speedup relative to `NCU`, `Eff%` the relative efficiency, and `Tot%` the percentage of time relative to the total execution time.

Notice that the performance analysis does not take into account initialization and diagnostics execution times. When not specified, results are obtained with the Rheticus cluster. In a first part, results are presented for the OpenMP version, then the hybrid MPI/OpenMP version on only one node, and finally the hybrid MPI/OpenMP version on several nodes.

5.1 OpenMP Parallel Version

This part aims to evaluate the OpenMP parallelization of the code. Results are presented for the unique grid size $(N_x, N_y, N_z) = (256, 256, 128)$ as bigger tested sizes do not change speedup and efficiency results.

Table 2 shows results for 10 temporal iterations. With this parallel version, it is possible to reach a speedup of 7.7 using the 12 cores of the computing node, giving an efficiency of 64 %. It can obviously be observed that the efficiency decreases when the number of used threads increases. The reason is that the needs of memory bandwidth resource increases together with the added cores, as we will see afterwards.

NCU	NTH	t(sec)	SU	Eff%	Tot%
1	1	8.76	1.00	100.0	100.0
4	4	2.54	3.44	85.9	100.0
8	8	1.50	5.85	73.1	100.0
12	12	1.14	7.68	64.0	100.0

Table 2. OpenMP: time loop.

Tables 3 and 4 show performances for the main parts of the time loop (the diffusion and the advection). When increasing the number of threads for the advection part, one can see execution times, speedups and hence efficiency scale very well. On the contrary, the diffusion part still suffers from an efficiency degradation, lowering to 55.3 % on the 12 cores of the node.

The diffusion part contains the one dimensional FFT computations in y and z directions. Let us have a look to the detail of computations inside the diffusion step. The times of Table 3 include times of Tables 5 and 6. They also contain the source term performance that is not explicited (although it shows nearly ideal scalings). Notice that Table 6 contains the diffusion operator computations together with the FFT in the z direction. The efficiency loss appears to be in the parts containing the FFT computations: for example, efficiency drops to 41.4 % for FFT on y on 12 threads. The FFT computations involve a high number of memory operations (*e.g.* data reorganizations between FFT), and hence increase the memory bandwidth requirements when adding computational cores.

Table 3. OpenMP: diffusion, source and fft 1D y and z.

NCU	NTH	t(sec)	SU	Eff%	Tot%
1	1	5.77	1.00	100.0	65.9
4	4	1.80	3.21	80.4	70.4
8	8	1.12	5.17	64.6	74.6
12	12	0.87	6.63	55.3	76.3

Table 4. OpenMP: advection.

NCU	NTH	t(sec)	SU	Eff%	Tot%
1	1	1.50	1.00	100.0	17.2
4	4	0.38	3.96	99.1	14.9
8	8	0.19	7.81	97.7	12.9
12	12	0.15	10.36	86.3	12.7

Table 5. OpenMP: fft 1D y.

NCU	NTH	t(sec)	SU	Eff%	Tot%
1	1	2.77	1.00	100.0	31.5
4	4	0.99	2.80	69.9	38.7
8	8	0.67	4.11	51.3	44.8
12	12	0.56	4.97	41.4	48.7

Table 6. OpenMP: diffusion and fft 1D z.

NCU	NTH	t(sec)	SU	Eff%	Tot%
1	1	1.38	1.00	100.0	15.7
4	4	0.40	3.46	86.4	15.6
8	8	0.24	5.77	72.1	16.0
12	12	0.18	7.76	64.7	15.6

5.2 Hybrid MPI/OpenMP Parallel Version

This part evaluates the multi-node version of the code presented in Sect. 4.4. This version aims to increase the number of computational nodes and the memory resource (bandwidth and space). This is particularly critical when attempting to reach targeted grid sizes. First, the deployment problem is addressed on one node (NTH and NP per node) in order to get the best mono-node performance.

Table 7 presents results for different couples (NTH, NP)[1] on one node. The best couple is $(3, 4)$, giving a 7.8 speedup on the 12 cores of the node. Also, computation times on the 12 cores are very similar to the OpenMP parallel version (see Table 2). This is surprising because this MPI version contains additional memory operations and overheads due to communications between processes.

Table 8 shows how performances scale on several nodes, using the previous (NTH, NP)=(3,4) per node deployment. The code was run on 12, 48, 96 and 192 computing units. On the 192 computing units, the code reaches a speedup of 81, leading to 42.4 % efficiency. To understand the loss of efficiency, each part of the code is also analyzed.

[1] Our code imposes NP as a power of 2.

Table 7. MPI/OpenMP: time loop.

NCU	NTH	NP	t(sec)	SU	Eff%	Tot%
1	1	1	8.87	1.00	100.0	100.0
12	3	4	1.14	7.79	64.9	100.0
12	6	2	1.28	6.92	57.7	100.0
12	12	1	1.43	6.21	51.7	100.0

Table 8. MPI/OpenMP: time loop.

NCU	NTH	NP	t(sec)	SU	Eff%	Tot%
1	1	1	8.87	1.00	100.0	100.0
12	3	4	1.13	7.84	65.3	100.0
48	3	16	0.31	28.32	59.0	100.0
96	3	32	0.19	46.40	48.3	100.0
192	3	64	0.17	81.37	42.4	100.0

The advection part (Arakawa scheme and source term) and the diffusion source term show very good efficiencies when increasing the number of nodes (close to 100 %). Table 9 shows performances for the FFT on the dimension y. Recall it does not include MPI communications. Whereas this part suffered from a limited efficiency in case of the OpenMP only parallelization (see Table 5), it is now able to reach a much better efficiency on the 192 computing units, reaching 81.8 %. Between 12 and 48 computing units (*i.e.* 1 and 4 nodes fully occupied), we observe a surlinear speedup due to positive cache effects: the volume of processed data per node is small enough to hold in the L3 cache.

Table 9. MPI/OpenMP: fft 1D y.

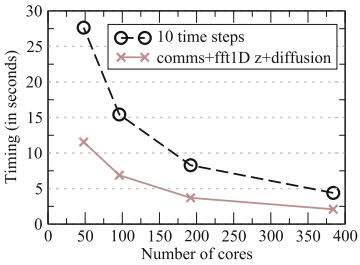
NCU	NTH	NP	t(sec)	SU	Eff%	Tot%
1	1	1	2.74	1.00	100.0	30.9
12	3	4	0.43	6.32	52.6	38.3
48	3	16	0.07	38.65	80.5	22.6
96	3	32	0.04	70.51	73.5	20.3
192	3	64	0.02	156.97	81.8	16.0

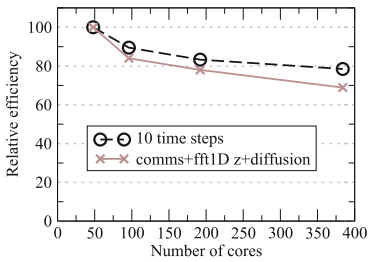
Table 10. MPI/OpenMP: comms, fft 1D z and diffusion.

NCU	NTH	NP	t(sec)	SU	Eff%	Tot%
1	1	1	1.46	1.00	100.0	16.4
12	3	4	0.30	4.77	39.7	27.0
48	3	16	0.14	10.07	21.0	46.2
96	3	32	0.10	14.34	14.9	53.2
192	3	64	0.07	21.86	11.4	61.1

Hence, it is the last part containing the communications which is responsible for the loss of efficiency. Indeed, Table 10 shows a drop in efficiencies due to the additional MPI communications needed to transpose data. Notice that this drop is particularly important between 1 and 4 processes (from 100 % to 39.7 %) because of the apparition of intranode communications and saturation of memory bandwidth, and again between 4 and 16 processes (from 39.7 % to 21 %) because of apparition of internode communications (involving the network). After 4 nodes, the drop of efficiency starts to become less stringent. Moreover, considering several nodes allows the user to handle bigger computational domains. This is the topic of the next Paragraph. The same study has been performed on the former presented machine Helios. Results are globally very close to those obtained on the Rheticus cluster, showing a speedup of 92 on 256 cores.

The bigger grid size $(1024, 1024, 512)$ targeted by Emedge3D implies the manipulation of 4 GB of memory per array. The no-MPI versions are not able to run: the memory requirements are to high to be handled by a unique node.





Each deployment uses 4 processes per node. The two last plots shows the timings (left) and efficiencies (right) for 10 time loop iterations (dashed line) and for the substep that includes communications (continuous line) as a function of the number of cores. They show a good scalability, leading to a relative efficiency of 78.5 % for 384 cores. On 768 cores, we do not expect dramatic loss of performance. Indeed, communication times diminish when adding cores and remain a fraction of the global execution time (comprised between 40 % and 50 %). This is because the reference time on 4 nodes already includes internodal communications. On Helios, the same $(1024, 1024, 512)$ grid size led to an efficiency of 70 % on 512 cores.

6 Conclusion

This work proposes a hybrid MPI/OpenMP parallelization strategy for an advection-diffusion problem relevant to the model simulated by Emedge3D, which is a dedicated code to study edge plasma physics in tokamaks.

The obtained parallel version allows to overcome the memory bandwith limitation, which was one of the main bottlenecks of Emedge3D. Indeed, considering additional nodes allows one to add memory resources that are needed when increasing the number of computing units. Plus, algorithm modifications (data organization in memory, FFT 1D in each direction) are particularly critical to reduce the amount of communications needed by the MPI version of the code. For a domain of size $(256, 256, 128)$, the parallel code is able to reach a speedup of 81 on 192 computing units. In addition to that, the code is also able to handle larger domain sizes, because adding nodes also increases available memory space. For example, it is able to handle grids of size $(1024, 1024, 512)$, leading to 4 GB for each 3D array, with an efficiency of 78.5 % on 384 cores.

As an immediate extension, tests could be performed on larger parallel systems, to see the evolution of the communications' scaling. Another axis is to couple this parallelization strategy with semi-implicit numerical method in order to increase the value of the time step Δt. Eventually, the integration of this parallelization strategy in Emedge3D code would enable to reach lower execution times and larger domain sizes.

Acknowledgements. The authors acknowledge the ANR (National Research Agency, project reference: ANR-10-BLAN-0940) and ERC Starting Grant Project GEOPARDI No. 279389 for financial supporting, the use of the Aix-Marseille University Computing Facility, and associated support services. The authors also express their acknowledgements to the IFERC for the access to Helios super-calculator. Finally, the authors thank the members of the research team PIIM of Aix-Marseille University for precious help.

References

1. Arakawa, A.: Computational design for long-term numerical integration of the equations of fluid motion: 2D incompressible flow. J. Comput. Phys. **1**(1), 119–143 (1966)
2. Beyer, N., et al.: Nonlinear dynamics of transport barrier relaxations in tokamak edge plasmas. Phys. Rev. Lett. **94**, 105001 (2005). http://link.aps.org/doi/10.1103/PhysRevLett.94.105001
3. Fuhr, G., et al.: Evidence from numerical simulations of transport-barrier relaxations in tokamak edge plasmas in the presence of electromagnetic fluctuations. Phys. Rev. Lett. **101**, 195001 (2008). http://link.aps.org/doi/10.1103/PhysRevLett.101.195001
4. Gorler, T., et al.: The global version of the gyrokinetic turbulence code GENE. J. Comput. Phys. **230**(18), 7053–7071 (2011). http://www.sciencedirect.com/science/article/pii/S0021999111003457
5. Huysmans, G., et al.: Non-linear MHD simulations of edge localized modes (ELMs). Plasma Phys. Controlled Fusion **51**(12), 124012 (2009)
6. Kuhn, M., et al.: Optimization and parallelization of Emedge3D on shared memory architecture. In: 2013 15th International Symposium on SYNASC, pp. 503–510. IEEE (2013)
7. Lütjens, H., Luciani, J.F.: The XTOR code for nonlinear 3D simulations of MHD instabilities in tokamak plasmas. J. Comput. Phys. **227**(14), 6944–6966 (2008)
8. Maeyama, S., et al.: Computation-Communication techniques for parallel spectral calculations in Gyrokinetic Vlasov simulations. Plasma Fusion Res. **8**, 1403150 (2013)
9. Monnier, A., et al.: Penetration of resonant magnetic perturbations at the tokamak edge. In: 38th EPS Conference on Plasma Physics (2011)
10. Orszag, S.A.: Transform method for the calculation of vector-coupled sums: application to the spectral form of the vorticity equation. J. Atmos. Sci. **27**(6), 890–895 (1970)
11. Zhang, Q., et al.: A fourth-order accurate finite-volume method with structured adaptive mesh refinement for solving the advection-diffusion equation. SIAM J. Sci. Comput. **34**(2), 179–201 (2012). http://dx.doi.org/10.1137/110820105