Optimization and parallelization of Emedge3D on shared memory architecture

Matthieu Kuhn, Guillaume Latu, Stéphane Genaud, Nicolas Crouseilles

Optimization and parallelization of Emedge3D on shared memory architecture.

Matthieu Kuhn , Guillaume Latu , Stéphane Genaud , Nicolas Crouseilles

RESEARCH REPORT
N° 8336
July 2013
Project-Team IPSO

ISSN 0249-6399   ISRN INRIA/RR--8336--FR+ENG

Optimization and parallelization of Emedge3D on shared memory architecture.

Matthieu Kuhn [*], Guillaume Latu [†], Stéphane Genaud [*], Nicolas Crouseilles [‡]

Project-Team IPSO

Research Report n° 8336 — July 2013 — 15 pages

Abstract: This report presents a study of techniques used to speedup a scientific simulation code. The techniques include sequential optimizations as well as the parallelization with OpenMP. This work is carried out on two different multicore shared memory architectures, namely a cutting edge 8x8 core CPU and a more common 2x6 core board. Our target application is representative of many memory bound codes, and the techniques we present show how to overcome the burden of the memory bandwidth limit, which is quickly reached on multi-core or many-core with shared memory architectures. To achieve efficient speedups, strategies are applied to lower the computation costs, and to maximize the use of processors caches. Optimizations are: minimizing memory accesses, simplifying and reordering computations, and tiling loops. On 12 cores processor Intel X5675, aggregation of these optimizations results in an execution time 21.6 faster, compared to the original version on one core.

Key-words: shared memory, optimization, parallelization, scientific computing, memory bandwidth bound.

[*] Icube, CNRS - Université de Strasbourg, France.
[†] CEA, IRFM, F-13108, Saint-Paul-lez-Durance, France.
[‡] Inria Rennes-Bretagne Atlantique, IPSO Project, IRMAR (UMR 6625), Université de Rennes 1, France.

Optimisation et parallélisation OpenMP du code Emedge3D.

Résumé : Ce rapport présente des stratégies pour la réduction du temps d'exécution d'un code de simulation numérique. Ces stratégies interviennent autant au niveau optimisation séquentielle qu'au niveau de la parallélisation OpenMP. Cette étude a été menée sur deux architectures à mémoire partagée : une carte à la pointe de la technologie comprenant 8x8 coeurs et une autre plus commune intégrant un processeur de 2x6 coeurs. Comme pour beaucoup d'applications du même type, les performances de la parallélisation du code numérique cible sont limitées par la bande passante mémoire. Les techniques que nous présentons dans ce document montrent comment contourner cette limitation. Afin d'obtenir des accélérations efficaces, différentes stratégies sont mises en oeuvre au niveau des calculs, mais aussi au niveau de l'accès aux données. Les optimisations en question sont la minimisation du nombre d'accès à la mémoire, la simplification et le ré-ordonnancement des calculs et le tiling pour maximiser l'utilisation des caches mémoire. Sur le processeur Intel X5675 (12 coeurs), l'accumulation de ces optimisations et la parallélisation permet d'obtenir un code 21.6 fois plus rapide par rapport à la version initiale sur un seul coeur.

Mots-clés : mémoire partagée, optimisation, parallélisation, calcul scientifique, limitation par la bande passante mémoire.

1 Introduction

Nuclear fusion is an actual challenge for energy production. A main goal is to produce energy at large scale with the desirable property of having a low environment impact. One way to reach nuclear fusion is to confine plasmas within a strong magnetic field, using a toric device called tokamak, such as the tokamak JET in Culhlam, Great Britain, or the ITER project in France. As real experiments are very expensive, and complex to set up and analyze, physicians, mathematicians and computer scientists collaborate to develop simulation codes, in order to understand and predict the plasma's behavior inside tokamaks.

Emedge3D [5] is a C scientific code which has been developed to simulate the behavior of plasmas at the tokamak's edge. This code is driven by a physical fluid model whose equations can be written in the compact form:

$$\partial_t U = L(U) + NL(U),$$

where $U = U(t, X)$ is the vector of physical variables (3D scalar fields), $X = (x, y, z)$ is the vector of 3D variables, L is a spatial differential linear operator and NL a nonlinear differential spatial operator. This equation is solved using state of the art of numerical schemes.

However, the simulations we target in terms of scale are currently unreachable because of long execution times. The execution complexity lies in the time and multi-scale phenomena being simulated. Indeed, *barrier relaxations* physical phenomenon appears at large time values, typically 1000 time units, and as the timestep is very small (about 10^{-4}: limited by numerical schemes). Therefore, the code needs a huge amount of time steps to produce a consistent solution. In addition, phenomena such as *magnetic islands* also appear at a very small space scale, implying to compute spatial operators on a large number of cells, resulting in numerous computations and data accesses per time iteration.

Ours efforts to reduce the execution time take place at two levels. At the first level, we design the numerical schemes with the objective of significantly increasing the time step value, thereby lowering the number of time iterations needed. At the second level, we focus on code optimization and parallelization.

The aim of the present study is to optimize and parallelize Emedge3D on shared memory machines with the OpenMP paradigm [3]. We are here especially interested in running the code efficiently on the following many-core boards: a 64 (8×8) cores Intel E7-8837 @ 2.67 GHz SMP. Also, a 12 (2×6) cores Intel X5675 @ 3.06 GHz Intel has been used for comparison throughout the study, as it is a more common processor found in clusters.

As in other scientific applications [11, 7], this code is memory bound, and this limits the speedup that can be achieved through parallelization. Hence, a first phase of optimization is required to improve the code in several aspects: improve data locality, simplify computations, minimize data loads from memory. In order to improve the latter, we use classical techniques of loop reordering and loop tiling [12] to take benefit of caches.

For the linear part of the model, these sequential optimizations allow to obtain a 3.2 speedup factor and parallelizing with tiling provides an additional factor 10 on the 12 cores Intel X5675, leading to a 32 total acceleration factor on 12 cores.

In the following, Emedge3D will be first described. Then, the linear L and nonlinear NL parts will be introduced and their progressive optimizations will be detailed. Finally, we will conclude with perspectives on this study.

2 Emedge3D

2.1 Physical and Numerical Aspects

Emedge3D is a fluid numerical simulation code. It shows the evolution over time of three physical parameters (scalar fields) on a given 3D spatial domain. The three physical values are the pressure p, the electrostatic potential ϕ, and the electromagnetic potential ψ. These three unknown (p, ϕ, ψ) depend on the spatial variable $X = (x, y, z) \in \mathbb{R}^3$ describing a simplified toric geometry. The different directions are called radial (x), toroidal (y) and poloidal (z) directions. Neumann boundary condition are imposed to the radial direction, and periodic boundary conditions are applied to toroidal and poloidal directions.

This code aims at simulating physical phenomena such as transport barriers and their relaxations [2], and how to control these relaxations by for example inducing resonant magnetic perturbations [8], and to characterize magnetic islands [10].

The model, and hence the code, can be split into two parts: a linear part (in which spatial operators to solve are linear), and a nonlinear part.

To reduce algorithmic costs, two discretizations are used to solve spatial operators (see 2.2.2): a semi-spectral representation of 3D fields to compute the linear part, in which toroidal and poloidal directions are expressed in the Fourier basis, and the radial one in the real basis, and a representation in real space for the 3 directions to compute the nonlinear part.

To solve the equations over time, a Runge-Kutta (RK) scheme of the desired order is performed. Over space, the operators are solved with finite differencing methods or spectral methods, leading to a linear computational cost in most of the code.

If needed, the code allows to downgrade the model by desactivating operators.

In the following, we only consider the most expensive part of the code. We call M_x, M_y, M_z the number of mesh points in the radial, toroidal and poloidal directions (respectively) for the semi-spectral representation, and N_x, N_y, N_z (with $N_x = M_x$) the number of mesh points in the radial, toroidal and poloidal directions (respectively) for the full real representation. In terms of memory storage, this means we will have to manipulate 4D arrays in the semi-spectral representation of the form $(Re|Im, M_z, M_y, M_x)$, where $Re|Im$ denotes the real or imaginary part.

2.2 Operator Description

2.2.1 Linear Part

During each timestep, several operators have to be solved:

- three perpendicular (in plan (x, y)) diffusion operators:
$$\nabla_\perp \cdot (\nu(x)\nabla_\perp W),\ \nabla_\perp \cdot (\chi(x,y)\nabla_\perp p),\ \nabla_\perp^2 \psi, \tag{1}$$
with $\nabla_\perp = (\partial_x, \partial_y, 0)$, $W = \nabla_\perp^2 \phi$, and $J = \nabla_\perp^2 \psi$.
- two curvature operators:
$$Gp,\ G(\delta_c \phi - \Gamma p), \tag{2}$$
with $G = \cos(y)\partial_y + \sin(y)\partial_x$,

as many times as required by the desired RK scheme. The time complexity to solve these two operators numerically is linear: $\Theta(M_x \times M_y \times M_z)$.

2.2.2 Nonlinear Part

Again, at each RK timestep, the code has to compute parallel gradients, also called Poisson brackets:
$$\nabla_\| \cdot (\chi_\|(x,y)\nabla_\| p), \nabla_\| J, \nabla_\| \phi,$$
with $\nabla_\| = \{\psi, .\}$ and $\{f, g\} = \partial_x f \partial_y g - \partial_y f \partial_x g$.

Considering two 3D complex arrays $\hat{f}(x,y,z)$ and $\hat{g}(x,y,z)$ in the semi-spectral form, the formula used to solve the Poisson bracket for a given radial position and for a given $y = m, z = n$ Fourier mode is :
$$\{\hat{f}, \hat{g}\}_{m,n} = \sum_{i=0}^{M_z} \sum_{j=0}^{M_y} \Big((m-j)\partial_x \hat{f}_{j,i} \partial_y \hat{g}_{m-j,n-i} \\ - j \partial_y \hat{f}_{ji} \partial_x \hat{g}_{m-j,n-i} \Big).$$

Even if this formula is numerically exact, the computational complexity for the whole array in semi-spectral representation is in $\Theta(M_x \times M_y^2 \times M_z^2)$: it is a convolution. To reduce this cost, the solution is to perform several 2D Discrete Fourier Transform (DFT) and to compute the Poisson bracket in the real space, and finally go back to the semi-spectral representation. This results in a computational complexity $\Theta(M_x \times M_y \times M_z \times \log(M_y \times M_z))$ (due to 2D FFTs). Poisson brackets are computed with a second order Arakawa method [1].

3 Linear Part

This part is dedicated to the optimization and parallelization of perpendicular diffusion and curvature operators, solved in the semi-spectral representation of scalar fields. Optimizations address mainly data organization and accesses while parallelization concerns the distribution of computations regarding the data.

3.1 Sequential Optimizations

For a sake of clarity, the initial code (although in C) had been written with an object oriented approach. Practically, each time an operator was applied on one or several fields, the code accessed the whole 4D arrays described in Section 2.1. Algorithm 1 sketches the original form chosen to solve the operators in Equations (1) and (2). In this algorithm, for each function call, one traversal of the input ($*\_in$) and output ($*\_out$) data structures is involved.

Algorithm 1 Linear part algorithm.

```
$\nabla_\perp^2$(p_out, p_in, ...);
$\nabla_\perp^2$(W_out, W_in, ...);
$\nabla_\perp^2$(psi_out, psi_in, ...);
G(p_out, phi_in, ...);
G(p_out, p_in, ...);
G(W_out, p_in, ...);
```

The first optimization consists in improving data locality: complex arrays dimensions order are transposed from $(Re|Im, M_z, M_y, M_x)$ to $(M_z, M_y, M_x, Re|Im)$. This results in a code running 2.3 times faster. As a second optimization, local variables have been used to store data loads into registers (similarly to the optimization described in Algorithm 6). For the diffusion operator, this yields a reduction from 35 to 15 data accesses per iteration, and a 1.4 acceleration factor.

3.2 Parallelization

The parallelization consists in distributing the outermost dimension M_z over the threads using an OpenMP static scheduling. Figure 1 shows the performance in terms of speedup of this parallel version. We observe a limit in the speedup increase when using more than 4 and 16 cores on the 12 and 64-core boards (respectively). The reason is the memory bandwidth requirement of the application, which scales linearly with the number of threads. This is verified by running the Stream benchmark [6], whose performance is also plotted on Figure 1. It shows that the memory bandwidth performance scaling factors matches the observed speedups.

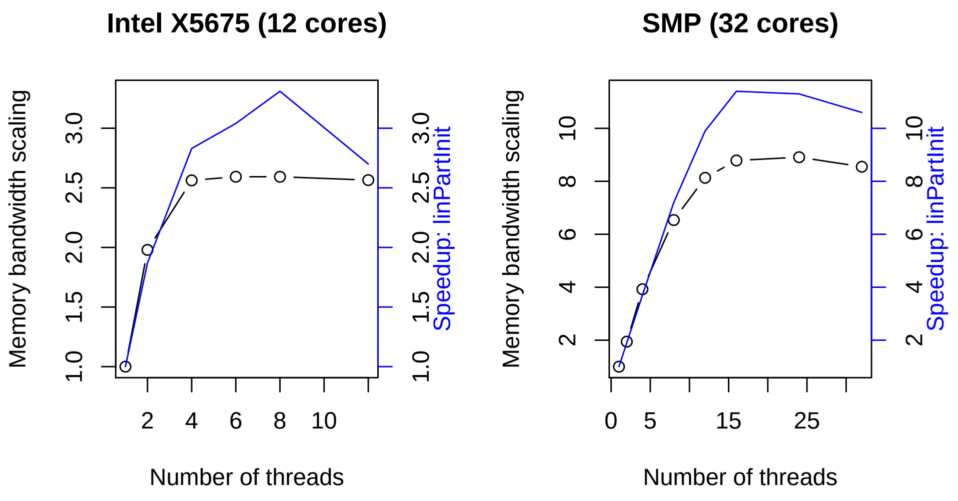

Figure 1: Bandwidth scaling factors and speedups on two multicore architectures.

3.3 Parallel Optimizations

The parallel optimizations tend to bypass the memory bandwidth limitation. The main objective here is to keep data as much as possible in processor caches, minimizing accesses to the global memory.

The first parallel optimization consisted in extracting the outermost loop over M_z, solving all operators on a same (x, y) slice. However, considering the target sizes for Emedge3D application, the data volume accessed is still too large to take fully benefit of cache effects.

The second parallel optimization applied is the reordering of function calls. Indeed, juxtaposing calls that involve the same scalar fields might improve data locality. This intuition comes from Equations (1) and (2): they show that the pressure scalar field appears in one perpendicular diffusion and in two curvature operations, indicating these operators should be solved in sequence. But this optimization, detailed in Algorithm 2, is still not sufficient for significant improvements.

Algorithm 2 Linear parallel algorithm: sliced and calls-reordered version.

```
#pragma omp parallel for private(idz)
for ( idz=0;idz<M_z;idz++ ){
  G_Z(p_out, phi_in, ..., idz);
  $\nabla_\perp^2$_Z(p_out, p_in, ..., idz);
  G_Z(p_out, p_in, ..., idz);
  G_Z(W_out, p_in, ..., idz);
  $\nabla_\perp^2$_Z(W_out, W_in, ..., idz);
  $\nabla_\perp^2$_Z(psi_out, psi_in, ..., idz);
}
```

The solution found to solve our scalability problem is to apply tiling accross the operators involved in the algorithm. Indeed, tiling consists in dividing a large iteration space into smaller chunks of desired size and depth [12]. It changes the execution order of a loop's instructions, which in turn modifies the order in which data are accessed. Algorithm 3 is the result of this optimization.

Algorithm 3 Linear parallel algorithm: tiled version.

```
#pragma omp parallel for private(idz,iblockX,iblockY)
  for ( idz=0;idz<M_z;idz++ ){
    for(iblockY=0; iblockY<nblockY; iblockY++)
    {
      for(iblockX=0; iblockX<nblockX; iblockX++)
      {
        G_Z_tiled(p_out, phi_in , ..., idz, iblockX, blocksizeX,
            iblockY, blocksizeY);
        $\nabla_\perp^2$_Z_tiled(p_out, p_in , ..., idz, iblockX, blocksizeX,
            iblockY, blocksizeY);
        G_Z_tiled(p_out, p_in , ..., idz, iblockX, blocksizeX,
            iblockY, blocksizeY);
        G_Z_tiled(W_out, p_in , ..., idz, iblockX, blocksizeX,
            iblockY, blocksizeY);
        $\nabla_\perp^2$_Z_tiled(W_out, W_in , ..., idz, iblockX, blocksizeX,
            iblockY, blocksizeY);
        $\nabla_\perp^2$_Z_tiled(psi_out, psi_in, ..., idz, iblockX, blocksizeX
            , iblockY, blocksizeY);
      }
    }
  }
```

3.4 Evaluation of the Linear Part

In order to find the best tile size, a profiling of the code has been done on the two multicore architecture introduced in Section 1. Multiple domain sizes have been tested, but, as results are very similar, only one domain size was chosen to illustrate our tests. In the following, we consider the 3D domain size $(M_x, M_y, M_z) = (256, 256, 192)$. Table 1 and 2 give the best times (and hence the best tile sizes) results for a given number of threads and Figure 2 shows speedups for these best tile sizes on two multicore architectures. These results clearly highlight that the tile size to be used depends on the number of threads and on the machine architecture. Figure 2 shows that tiled codes with appropriate tile sizes are able to reach much higher speedups and reduce execution times on both multicore architectures.

Table 1: Best times by thread number on Intel X5675.

Tile size (Tx, Ty)	Threads	Time (μs)	Speedup
(256,256)	1	1401605.00	1.00
(256,128)	1	1400773.00	1.00
(256,64)	2	717327.00	1.95
(128,32)	4	373240.00	3.76
(32,4)	6	259935.00	5.39
(32,4)	8	199425.00	7.03
(16,2)	12	137913.00	10.16

Table 2: Best times by thread number on SMP.

Tile size (Tx, Ty)	Threads	Time (μs)	Speedup
(256,256)	1	1776686.00	1.00
(256,32)	1	1733988.00	1.02
(8,8)	2	952863.00	1.86
(4,16)	4	515770.00	3.44
(4,2)	6	357161.00	4.97
(2,8)	8	276104.00	6.43
(4,2)	12	182989.00	9.71
(4,2)	16	138960.00	12.79
(4,2)	24	95690.00	18.57
(4,2)	32	71879.00	24.72
(2,2)	64	44876.00	39.59

To explain those results, a profiling of cache performance was done with the PAPI library [9]. Figure 3 shows L3 cache performance for Intel X5675 and SMP. These curves clearly show that L3 cache performance per thread number is improved by using the appropriate tile size, allowing the code to be less memory bound. For example, with tile size $(16, 2)$ cache misses drop linearly indicating tiles stay in the cache whatever the number of threads used. But with tile size $(128, 32)$, cache misses begin to increase when running with 8 threads, showing that some tiles are reloaded from global memory to L3 cache. Moreover, performances on X5675 almost indicate the best tile size as a function of the number of threads used by the application (see also Table 1).

4 Nonlinear Part

This part deals with the Poisson brackets computations, that is core to solve the nonlinear part of Emedge3D. More precisely, it concerns optimization and parallelization of the Arakawa method and DFTs. Here, optimizations address data operations, computation simplifications while parallelization relates to distributing instructions.

For comprehension, data are stored in the following order :

- comp$[M_x * M_z * M_y * 2]$ for complex fields, meaning $Re(comp(idx, idy, idz))$ is stored at position $idx * (M_z * M_y * 2) + idz * (M_y * 2) + idy * 2$ and $Im(comp(idx, idy, idz))$ at position $idx * (M_z * M_y * 2) + idz * (M_y * 2) + idy * 2 + 1$,
- real$[N_x * N_z * N_y]$ for real fields, with the same storage format.

4.1 Initial State

As explained in Section 2.2.2, Poisson brackets are solved in the real space. We consider here the data fields inputs and outputs to be in the semi-spectral decomposition, which implies that this nonlinear part must include the DFTs computations (see Algorithm 4). In this algorithm, $f1comp$ and $f2comp$ are the inputs of Arakawa's method in the semi-spectral space. These data fields are first transformed by FFTs giving $f1\_in$ and $f2\_in$ respectively. Then, Arakawa's

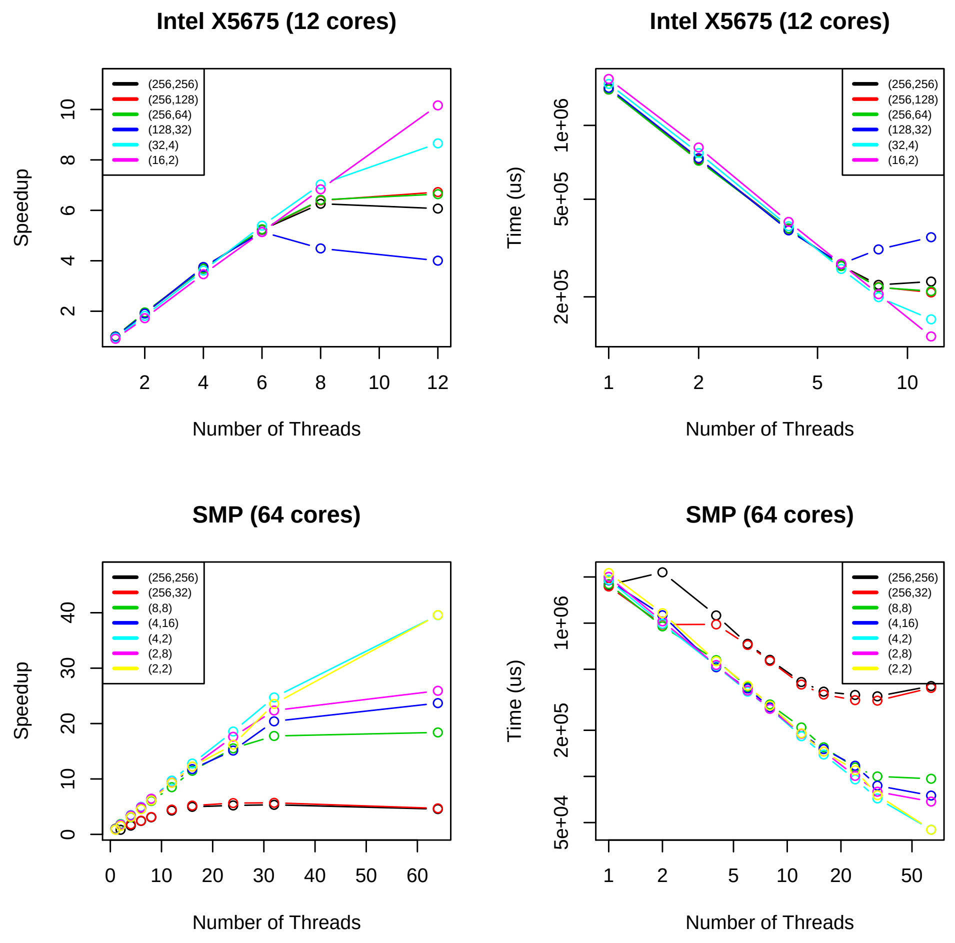

Figure 2: Performance corresponding to Table 1 and 2 tile sizes on two architectures.

method is applied on $f1\_in$ and $f2\_in$, and the result is stored in f_out. Finally, f_out is transformed back from real to semi-spectral representation by FFT.

4.2 Sequential Optimizations

4.2.1 Arakawa method

This classical numerical method is used to solve the Poisson bracket. In codes such as Emedge3D, this scheme is very popular because of its energy conservation property [1]. In the following, data accesses and computation simplifications are presented and evaluated.

For i and j given, the formula to compute Arakawa's scheme is :

$$
\begin{aligned}
& J_{i,j}(\xi,\psi) = -\frac{1}{\Delta x \Delta y}\,[\\
& \quad (\psi_{i,j-1}+\psi_{i+1,j-1}-\psi_{i,j+1}-\psi_{i+1,j+1})\,(\xi_{i+1,j}+\xi_{i,j})\\
- & \quad (\psi_{i-1,j-1}+\psi_{i,j-1}-\psi_{i-1,j+1}-\psi_{i,j+1})\,(\xi_{i,j}+\xi_{i-1,j})\\
+ & \quad (\psi_{i+1,j}+\psi_{i+1,j+1}-\psi_{i-1,j}-\psi_{i-1,j+1})\,(\xi_{i,j+1}+\xi_{i,j})\\
- & \quad (\psi_{i+1,j-1}+\psi_{i+1,j}-\psi_{i-1,j-1}-\psi_{i-1,j})\,(\xi_{i,j}+\xi_{i,j-1})\\
+ & \quad (\psi_{i+1,j}-\psi_{i,j+1})\,(\xi_{i+1,j+1}+\xi_{i,j})\\
- & \quad (\psi_{i,j-1}-\psi_{i-1,j})\,(\xi_{i,j}+\xi_{i-1,j-1})\\
+ & \quad (\psi_{i,j+1}-\psi_{i-1,j})\,(\xi_{i-1,j+1}+\xi_{i,j})\\
- & \quad (\psi_{i+1,j}-\psi_{i,j-1})\,(\xi_{i,j}+\xi_{i+1,j-1})]\,.
\end{aligned}
$$

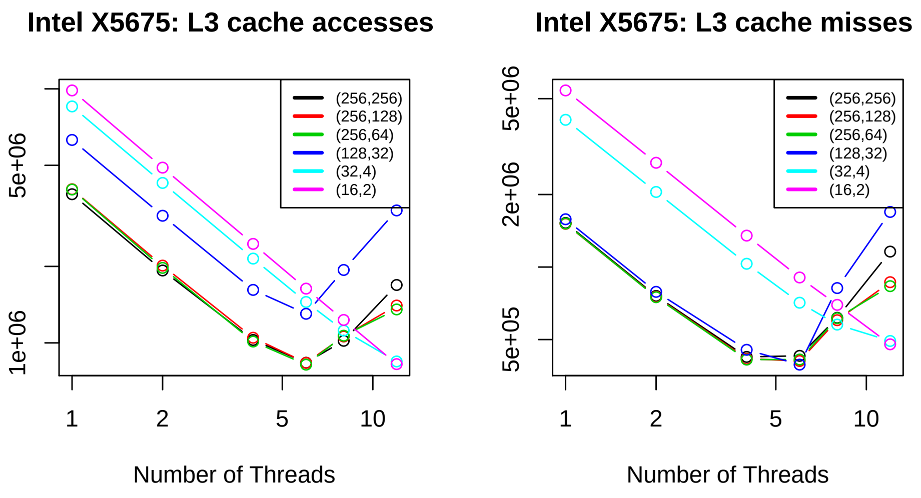

Figure 3: L3 cache performance for Table 1 and 2 tile sizes on two architectures.

After expansion, $\psi_{*,*}\xi_{i,j}$ eliminates each other so 8 additions can be saved up :

$$
\begin{aligned}
J_{i,j}(\xi,\psi) = -\frac{1}{\Delta x \Delta y}\,[\\
& (\psi_{i,j-1} + \psi_{i+1,j-1} - \psi_{i,j+1} - \psi_{i+1,j+1})\,\xi_{i+1,j} \\
- & (\psi_{i-1,j-1} + \psi_{i,j-1} - \psi_{i-1,j+1} - \psi_{i,j+1})\,\xi_{i-1,j} \\
+ & (\psi_{i+1,j} + \psi_{i+1,j+1} - \psi_{i-1,j} - \psi_{i-1,j+1})\,\xi_{i,j+1} \\
- & (\psi_{i+1,j-1} + \psi_{i+1,j} - \psi_{i-1,j-1} - \psi_{i-1,j})\,\xi_{i,j-1} \\
+ & (\psi_{i+1,j} - \psi_{i,j+1})\,\xi_{i+1,j+1} \\
- & (\psi_{i,j-1} - \psi_{i-1,j})\,\xi_{i-1,j-1} \\
+ & (\psi_{i,j+1} - \psi_{i-1,j})\,\xi_{i-1,j+1} \\
- & (\psi_{i+1,j} - \psi_{i,j-1})\,\xi_{i+1,j-1}]\,.
\end{aligned}
$$

We have named this first optimization AddOpti.

Then, as there are numerous memory transactions, data loads on ψ and then on both ψ and ξ have been explicitely transferred to registers in order to minimize them. We called these optimizations SemiLoadReuse and LoadReuse respectively. Algorithms 5 and 6 (original and then optimized) illustrate how these improvements are implemented. These optimizations include AddOpti.

Moreover, it is also possible to save computations on ψ. Indeed, let us define the following variables:

$$
\begin{aligned}
\psi^A_{i,j} &= (\psi_{i,j-1} + \psi_{i+1,j-1} - \psi_{i,j+1} - \psi_{i+1,j+1})\,, \\
\psi^C_{i,j} &= (\psi_{i+1,j} + \psi_{i+1,j+1} - \psi_{i-1,j} - \psi_{i-1,j+1})\,, \\
\psi^E_{i,j} &= (\psi_{i+1,j} - \psi_{i,j+1})\,, \\
\psi^G_{i,j} &= (\psi_{i,j+1} - \psi_{i-1,j})\,,
\end{aligned}
$$

the last expression solving Arakawa's scheme on a mesh point becomes:

$$
\begin{aligned}
J_{i,j}(\xi,\psi) &= -\frac{1}{\Delta x \Delta y}\,[\psi^A_{i,j}\xi_{i+1,j} - \psi^A_{i-1,j}\xi_{i-1,j} \\
&\quad +\psi^C_{i,j}\xi_{i,j+1} - \psi^C_{i,j-1}\xi_{i,j-1} \\
&\quad +\psi^E_{i,j}\xi_{i+1,j+1} - \psi^E_{i-1,j-1}\xi_{i-1,j-1} \\
&\quad +\psi^G_{i,j}\xi_{i-1,j+1} - \psi^G_{i+1,j-1}\xi_{i+1,j-1}]\,.
\end{aligned}
$$

The last form shows that it is possible to reuse computations from one iteration to the other. For example, at iteration $i' = i, j' = j+1$, $\psi^C_{i',j'-1}$ has already been computed at iteration i, j. We have also tested two strategies based on the amount of temporary storage used: reuse ψ^C

Algorithm 4 Nonlinear algorithm.

```
/* FFT 2D: Complex ----> Real */
int lenComp = M_y*M_z;
int lenReal = N_y*N_z;
for(idx=0;idx<M_x;idx++)
{
  int sliceComp = idx*lenComp;
  int sliceReal = idx*lenReal;
  fftw_execute_dft_c2r(p2D_C2R,&(f1comp[sliceComp]), &(f1_in[
     sliceReal]));
  fftw_execute_dft_c2r(p2D_C2R,&(f2comp[sliceComp]), &(f2_in[
     sliceReal]));
}
ArakawaMultiplication(f1_in,f2_in,f_out,...);
/* FFT 2D: Real ----> Complex */
for(idx=0;idx<N_x;idx++)
{
  int sliceComp = idx*lenComp;
  int sliceReal = idx*lenReal;
  fftw_execute_dft_r2c(p2D_R2C,&(f_out[sliceReal]), &(f3comp[
     sliceComp]));
}
```

Algorithm 5 Arakawa initial algorithm.

```
void ArakawaMultiplication(psi, xi, result, N_x, N_y, N_z, dx,
   dy)
{
...
  for(i=0;i<N_x;i++){
    for(j=0;j<N_z;j++){
      for(k=0;k<N_y;k++){
        result[resIndex] = (psi[ijm1] + psi[ip1jm1] - psi[ijp1]
            - psi[ip1jp1]) * ...;
      }
    }
  }
}
```

Algorithm 6 Arakawa algorithm: memory accesses optimized.

```
void ArakawaMultiplication(psi, xi, result, N_x, N_y, N_z, dx,
    dy)
{
...
  for(i=1;i<NX-1;i++){
    for(j=0;j<N_z;j++){
      /* load values in registers */
      /* idx=i-1, idz=j, idy=0 */
      psi_im1jkm1 = psi[im1jkm1];
      /* idx=i-1, idz=j, idy=1 */
      psi_im1jk = psi[im1jk];
      /* idx=i-1, idz=j, idy=2 */
      psi_im1jkp1 = psi[im1jkp1];
      /* and so on for i and i+1 */
      ...
      for(k=1;k<N_y-1;k++){
        result[resIndex] = (psi_ijkm1 + psi_ip1jkm1 - psi_ijkp1
            - psi_ip1jkp1) * ...;
        /* swap variables to avoid extra loads */
        psi_im1jkm1 = psi_im1jk;
        psi_im1jk = psi_im1jkp1;
        psi_im1jkp1 = psi[im1jkp2];
        /* and so on for i and i+1 */
        ...
      }
    }
  }
}
```

(psiCReuse) only, or both ψ^C and ψ^A (psiACReuse). These optimizations include LoadReuse or SemiLoadReuse (when specified in the version name).

Table 3 shows how the proposed optimizations impact Arakawa's computation execution times on Intel X5675. It indicates PsiACReuse is the best of the proposed algorithms. The same results are observed on SMP.

Table 3: Arakawa: Times by sequential optimization on Intel X5675. Case size: (512,512,128).

Version	Time (μs)
Original	1144520.00
AddOpti	987053.00
PsiCSemiLoadReuse	727701.00
PsiACSemiLoadReuse	697066.00
LoadReuse	687248.00
SemiLoadReuse	683341.00
PsiCReuse	676677.00
PsiACReuse	654606.00

4.2.2 DFTs

Each time a Poisson bracket is computed, the code has to perform three main steps (see Algorithm 4):

- two backward transforms (from semi-spectral to full real) for the inputs of the Arakawa method, $f1comp$ and $f2comp$,
- and one forward transform for the output, $f3comp$.

These steps consist in performing M_x times the 2D DFTs per data set: one for each point in the radial direction. DFTs are computed with the sequential FFTW [4] library version 3.3.3 – this library is generally well regarded for the fine-tuned optimizations of the algorithms it implements.

However, incrementing or decrementing toroidal or poloidal dimension size, keeping approximately the same data volume, may change execution time by up to a factor 4. Indeed, FFT computations is much more efficient considering powers of 2 sizes due to the algorithm employed. In Emedge3D, plans for FFTs are defined at initialization once for the whole program execution. This plan creation requires array sizes in the real space. As the data fields are initialized in the semi-spectral representation (complex arrays of size (M_x, M_y, M_z)), real domain sizes must be deduced to create the FFT plans, with the formula $(N_x, N_y, N_z) = (M_x, (M_y - 1) \times 2, M_z)$. For example, with a 2D size $(M_y, M_z) = (\mathbf{512}, 128)$, the FFT execution time is 4 times longer than a $(M_y, M_z) = (\mathbf{513}, 128)$ size. The reason why $M_y = 513$ is 4 times faster is that it gives $N_y = 1024 = 2^{10}$ real size, leading to a 2D FFT of size $(\mathbf{1024}, 128)$, whereas $M_y = 512$ gives a $(\mathbf{1022}, 128)$ FFT size.

4.3 OpenMP Parallel Version

As for Section 3.2, the outermost dimension is parallelized with OpenMP, *i.e.* the loop over the radial dimension (to be compared to the poloidal direction for linear operators).

4.3.1 Arakawa method

Table 4 shows best time and hence best optimization depending on the number of threads on Intel X5675. Speedups reached are not really satisfying because of the low compute intensity for this function. This table also shows that the best OpenMP parallel version, *i.e.* SemiLoadReuse is not the same as the best sequential one: PsiACReuse. Indeed, as scaling is once more limited by the memory bandwidth, and as PsiACReuse implies adding an array, and hence some extra data loads, this optimization is not profitable on the targeted architectures. Also, SemiLoadReuse is better than LoadReuse because it uses less registers data movements, noticeable on the assembly code generated by these two versions. On SMP the best optimization in parallel is also SemiLoadReuse, giving a speedup of 19.99 relative to optimized version on the 64 cores. This result is consistent with the Intel X5675 one.

Table 4: Arakawa: Best time by threads number on Intel X5675.

Version	Threads	Time (μs)	Speedup
PsiACReuse	1	654606.00	1.00
PsiACReuse	2	338784.00	1.93
SemiLoadReuse	4	193072.00	3.39
SemiLoadReuse	6	151341.00	4.33
SemiLoadReuse	8	137241.00	4.77
SemiLoadReuse	12	126286.00	5.18

4.3.2 DFTs

To compute DFTs in parallel, the outermost dimension is parallelized with OpenMP, *i.e.* the loop over the radial dimension. Table 5 compares DFTs for different domain sizes on Intel X5675. Data volumes involved in one DFT operation are keys to understand these results. A domain size (M_x, M_y, M_z) yields $M_x \times M_y \times M_z \times 2$ number of points per data field. As values are stored in double precision, this leads to a $M_x \times M_y \times M_z \times 2 \times 8/(1024^2)$ MB data volume to transform. In addition, DFTs are not computed inplace (because slower), so that the last data size has to be multiplied by two to obtain the total size required for one DFT computation. Table 5 is interpreted as follows:

- the two first sizes $(128, 257, 8)$ and $(128, 257, 16)$ imply 4 and 8 MB of data to access respectively. As Intel X5675 has two L3 caches of size 12 MB, data volumes are cached for the whole DFT operation.

- the two last sizes $(128, 257, 32)$ and $(128, 257, 64)$ results in 32>24 and 64>24 MB of data, exceeding L3 caches sizes and therefore memory bandwidth requirements.

The Table clearly highlights the more DFTs' inputs exceed caches sizes, the less the parallel algorithm is efficient. Even if FFT algorithm is known to be CPU intensive, the last result means the domain size targeted in Emedge3D application (about $(512, 512, 128)$) is too large to be balanced with DFT computations.

On SMP, the best speedup we achieve is 32.3 with a $(1024, 65, 32)$ domain size. However, speedups on this archicteture are lower in a number of configurations. It seems that the complexity of the SMP architecture, including NUMA effects, require further investigations to better understand performance degradation in some cases.

Table 5: DFTs: Comparison small VS large number of points on Intel X5675.

DFT Size	Threads	Time (μs)	Speedup
128x257x**8**	1	14401.00	1.00
	2	7380.00	1.95
	4	3763.00	3.83
	8	1964.00	7.33
	12	1446.00	9.96
128x257x**16**	1	29094.00	1.00
	2	15431.00	1.89
	4	7806.00	3.73
	8	4020.00	7.24
	12	2970.00	9.80
128x257x**32**	1	61598.00	1.00
	2	32892.00	1.87
	4	17100.00	3.60
	8	9357.00	6.58
	12	7273.00	8.47
128x257x**64**	1	131233.00	1.00
	2	69716.00	1.88
	4	36929.00	3.55
	8	22447.00	5.85
	12	19193.00	6.84

4.3.3 Limitations

In the two last sections, we showed both Arakawa method and DFTs parallel scalings are limited by the memory bandwidth. As in Section 3.3, tiling loops could be used to limit global memory accesses. However, applicable tile sizes are too large to improve data locality. Indeed, Arakawa scheme for a given radial index idx requires 2D DFTs at positions $idx - 1$, idx and $idx + 1$ (stencil in the radial direction). In addition, 2D DFTs computations cannot be decomposed, *i.e.* it cannot be computed on smaller portions of the 2D array. An algorithm using 1D DFTs has been developed and tested but resulting speedups are not better due to the needed additional transpositions (involving more memory operations).

5 Overall Results

This section presents a summary of the optimizations and parallelizations performed in the previous algorithms. Table 6 shows one RK step on Intel X5675 with $(512, 512, 128)$ domain size. The test case used for this benchmark is called RMP_C1 [8] and simulates the transport barrier relaxation. The table shows that the linear part performances are satisfactory, with a good parallel scaling relative to optimized times. Concerning the nonlinear part, speedups for DFTs are acceptable, but Arakawa's ones are still limited because of its weak compute intensity. It explains why the percentage for Arakawa's computation in the parallel version grows compared to the previous versions. We also notice that total speedups appear to be governed by DFTs speedups, in agreement with computational costs of the overall algorithm.

As explained in 4.3.2, DFTs parallel results on SMP are highly domain size sensitive, and so are the performances for one RK step. Again, further investigations have to be performed to improve those results.

Table 6: Optimization and parallelization overall results on Intel X5675. Speedup$_{\text{init}}$ is computed as a function of initial times, and Speedup$_{\text{opt}}$ as a function of optimized times.

Code		DFTs	Arak	Linear	Total
Initial 1 core	Time (s)	13.47	2.29	5.30	21.06
	Percent	63.9	10.9	25.2	100
	Speedup	1	1	1	1
Optimized 1 core	Time (s)	4.10	1.34	1.70	7.14
	Percent	57.5	18.7	23.8	100
	Speedup$_{\text{init}}$	3.3	1.7	3.1	3
Parallel 12 core	Time (s)	0.58	0.25	0.16	0.99
	Percent	58.6	25.2	16.2	100
	Speedup$_{\text{opt}}$	7.0	5.3	10.5	7.2
	Speedup$_{\text{init}}$	23.1	9.0	32.6	21.6

6 Conclusion

In this work, we have focused on the optimization and OpenMP parallelization of the scientific code Emedge3D. Our main contribution is to reach an overall speedup of 21.6 on the 12 cores Intel X5675 by overcoming the memory bandwidth limitation. The principal strategy to achieve this objective consists in maximizing the use of processor caches. Therefore, our work focuses on the optimization of memory operations, using loads into registers, reorderings of instruction calls, and simplifications of computations in both parts of the code. Tiling loops was particularly critical for the linear part, allowing to reuse data field loads into caches from one operator call to another.

Unfortunatly, tiling is not applicable for the nonlinear part, whose parallel scaling is still suffering from memory bandwidth limitation. In addition, results on the SMP architecture are quite disappointing, especially for DFTs computations due to lack of cache reuse.

A way to increase accessible memory bandwidth is to parallelize the application on several computation nodes. In a future work, we will target a MPI parallelization of the code to overcome the memory bandwidth limitation.

7 Aknowledgement

The authors acknowledge the financial support of ANR (Agence Nationale de la Recherche) for project E2T2, and the services of the Mésocentre d'Aix-Marseille Université Computing Facility.

References

[1] A. Arakawa. Computational design for long-term numerical integration of the equations of fluid motion: Two-dimensional incompressible flow. part i. *Journal of Computational Physics*, 1(1):119 – 143, 1966.

[2] P. Beyer, S. Benkadda, G. Fuhr-Chaudier, X. Garbet, Ph. Ghendrih, and Y. Sarazin. Nonlinear dynamics of transport barrier relaxations in tokamak edge plasmas. *Phys. Rev. Lett.*, 94:105001, Mar 2005.

[3] O. A. R. Board. Openmp application program interface, version 3.1. 2011.

[4] M. Frigo and S.G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[5] G. Fuhr, P. Beyer, S. Benkadda, and X. Garbet. Evidence from numerical simulations of transport-barrier relaxations in tokamak edge plasmas in the presence of electromagnetic fluctuations. *Phys. Rev. Lett.*, 101:195001, Nov 2008.

[6] J. MacCalpin. The stream benchmark page. *http://www.cs.virginia.edu/stream/*.

[7] S.A. McKee, W.A. Wulf, and T.C. Landon. Bounds on memory bandwidth in streamed computations. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *EURO-PAR '95 Parallel Processing*, volume 966 of *Lecture Notes in Computer Science*, pages 83–99. Springer Berlin Heidelberg, 1995.

[8] A. Monnier, G. Fuhr, P. Beyer, S. Benkadda, and X. Garbet. Penetration of resonant magnetic perturbations at the tokamak edge. In *38th EPS Conference on Plasma Physics*, 2011.

[9] P.J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proc. Department of Defense HPCMP Users Group Conference*, 1999.

[10] M. Muraglia, O. Agullo, S. Benkadda, X. Garbet, P. Beyer, and A. Sen. Nonlinear dynamics of magnetic islands imbedded in small-scale turbulence. *Phys. Rev. Lett.*, 103:145001, Sep 2009.

[11] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178 – 194, 2009. Revolutionary Technologies for Acceleration of Emerging Petascale Applications.

[12] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *ACM Sigplan Notices*, 26(6):30–44, 1991.

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399