Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 541 – 550

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

# Hybrid-parallel algorithms for 2D Green's functions

Alejandro Álvarez-Melcón[a], Domingo Giménez[b,1], Fernando D. Quesada[a], Tomás Ramírez[b]

[a]Departamento de Tecnologías de la Información y las Comunicaciones, Polytechnic University of Cartagena, Spain
[b]Departamento de Informática y Sistemas, University of Murcia, Spain

**Abstract**

Green's functions are used to solve non homogeneous integral equations with boundary conditions. In electromagnetism they are used for a large class of problems, including the analysis of microwave circuits, antennas, and the investigation of breakdown phenomena in microwave transmission lines. In many cases it is necessary to compute a large number of Green's functions between pairs of source and observation points. Today, basic computational systems are composed of multiple CPUs (multicores) and GPUs, and it is necessary to adapt the software used for the solution of scientific problems to these systems. This paper studies the optimization of the computation of 2D rectangular waveguide Green's functions in clusters of multicores with GPUs. Different basic algorithms are considered, and the most efficient implementations are obtained by adequately combining basic algorithms in hybrid parallelism programs.

*Keywords:* Green's functions, Hybrid Parallel Computing, Performance Estimation, Clusters, Multicore, Multi-GPU

## 1. Introduction

Integral and differential equations are mathematical tools with applications in a variety of scientific problems. There are several methods to solve them: variables separation, Fourier series, method of moments, etc., [1]. Some methods are based on Green's functions [2], which are used to formulate non homogeneous integral equations subject to boundary conditions. They are used in different fields, and this paper focuses on Green's functions inside rectangular waveguides [3], which are used in the analysis and design of microwave circuits, especially for high power applications. These functions can be expressed in the form of infinite series in spatial or spectral domains. They present some computational problems due to slow convergence of these series.

When solving practical problems, the need to calculate hundreds or thousands of Green's functions generates a high computational cost, and methods to accelerate the convergence of the series and to reduce the execution time are necessary. With the Ewald's method only a reduced number of terms in the series is calculated, and the convergence speed increases [4]. This method exploits the extraction of electrostatic interactions, and involves the calculation of both short and long spectral range interactions. Following this technique, the calculation can be represented in the real space as the sum of two terms:

$$\phi(r) \overset{def}{\Rightarrow} \phi_{sr}(r) + \phi_{lr}(r) \quad (1)$$

*Email addresses:* alejandro.alvarez@upct.es (Alejandro Álvarez-Melcón), domingo@um.es (Domingo Giménez), Fernando.Quesada@upct.es (Fernando D. Quesada), tomas.ramirez@um.es (Tomás Ramírez)

[1]Corresponding author

doi:10.1016/j.procs.2013.05.218

where $\phi_{sr}(r)$ represents the short-range term, which can be quickly calculated in the real space, and $\phi_{lr}(r)$ stands for the long-range term, which is calculated in the Fourier space (or spectral domain) under the assumption that the system is periodic. The two terms are separated by the so-called splitting parameter between spatial and spectral contributions.

An alternative technique for the acceleration of Green's functions series inside waveguides is the Kummer's method. Unlike the previous technique, Kummer's method accelerates the convergence of the spectral domain series directly, without splitting the whole contribution in two parts. The basic idea is to extract the asymptotic term from the original series, so that the convergence of the remainder is improved. The asymptotic series have simpler analytical expressions, and closed solutions can be obtained.

As seen, there are some numerical methods to accelerate the calculation of Green's functions. However, the calculation of Green's functions is very critical for the implementation of efficient analysis tools of waveguide devices. This is because a large number of Green's functions evaluations are usually required during the analysis of practical waveguide devices. Therefore, it is necessary to increase the efficiency by developing efficient algorithms and satisfactory implementations for current computer systems.

In some electromagnetic applications large systems of linear equations need also to be solved, and it may be preferable to parallelize the whole application and not the computation of each Green's function, or even to develop algorithms combining different types of parallelism: coarse grained parallelism for the whole application and fine grained parallelism for the computation of individual Green's functions. In a previous work we developed basic non-hybrid algorithms for multicore, GPUs and clusters [5]. Because today basic computational systems are multicore+GPUs and they are combined, forming clusters of multicores, in this paper the previous basic algorithms are adapted and combined to obtain efficient implementations combining different sources of parallelism for those today's hybrid systems. This paper focuses on the solution of 2D rectangular waveguide Green's functions in computational electromagnetism, but the methodology followed to obtain efficient versions for clusters of multicores+GPUs can be applied to other types of Green's functions and in different fields.

The paper is organized as follows. In section 2 the basic ideas of Green's functions in rectangular uniform waveguides are reviewed, together with the basic sequential algorithm from which the parallel versions are developed. Section 3 summarises the characteristics of the computational systems where the parallel algorithms are experimentally analysed. In section 4 some algorithms are developed and analysed. Finally, in section 5, the conclusions are summarised and some research lines are outlined.

## 2. Green's functions in rectangular uniform waveguides

Green's functions can be applied in rectangular uniform waveguides of different dimensions. The two-dimensional problem is considered here. Figure 1 shows a parallel plate waveguide (one-dimensional to simplify the figure). Within the waveguide we consider a set of source and observer points. One source point is fixed, and we compute the Green's functions for an observer point moving along the axes $\hat{y}$ and $\hat{z}$. To do so, the series resulting from the application of the Kummer technique are computed. Thus, for each of the calculated Green's functions, a number of terms in the spectral series needs to be computed. The number of terms to be summed can be fixed for all source-observer pairs or it can be dynamically calculated as a function of the distance between the two points along the longitudinal axis $|z_o - z_s|$.

In two-dimensional problems, there is a reference system within a rectangular waveguide. We again consider a fixed source point, while the observer points are moving in a transversal $(x, y)$ plane. Green's functions are calculated using the Ewald method: the Green's function is decomposed as a sum of two terms, one calculated in the spectral domain and the other in the spatial domain. For the spectral part the number of terms or modes needed in the series is calculated. For the calculation of the spatial part the images method is used, and the functions are evaluated along two separate axes, which means a higher computational cost.

In the sequential algorithm the Green's functions for each pair of points in the observation plane are calculated. A scheme is shown in algorithm 1. Initially, the excitation modes in the rectangular waveguide (*nmod*) are calculated. They correspond to the first modes propagating in the waveguide, ordering the wave numbers in increasing order. The splitting parameter [6] to be used in the calculation of the Green's functions is also obtained. The whole series is decomposed in spectral and spatial parts. The work done during the initialization phase (ordering of the

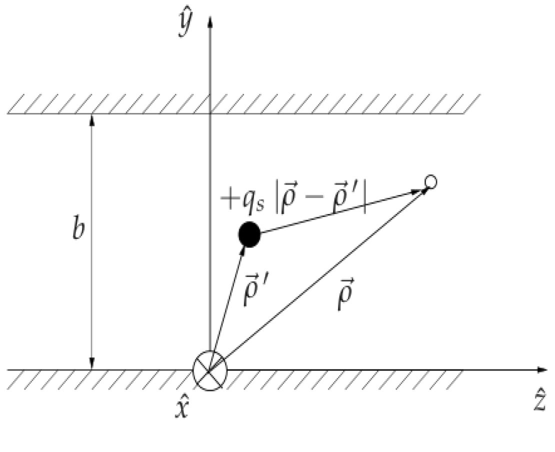

Fig. 1: Parallel plate waveguide in a reference system.

modes with increased wave number values) makes it possible to compute the spectral part with a single loop in the number of modes. The spatial part is more costly, and it comprises two loops in the number of images along the $x$ and $y$ axes (*mimag* and *nimag*). Additionally, there is another inner loop depending on the number of precision digits. The number of terms used in this last loop is always within the interval [5, 15], and it is considered to be constant in the following theoretical and experimental analyses. So, the execution time can be represented as

$$t(m, n, nmod, mimag, nimag) = m \cdot n \cdot (k_1 \cdot nmod + k_2 \cdot mimag \cdot nimag) \in \theta\,(m \cdot n \cdot mimag \cdot nimag) \qquad (2)$$

**Algorithm 1** Calculation of Green's functions in two-dimensional problems: GF-2D(*m*,*n*,*nmod*,*mimag*,*nimag*)

```
Initialization: obtain and sort modes
for $i = 1$ to $m$ do
  for $j = 1$ to $n$ do
    {Spectral part}
    for $k = 1$ to $nmod$ do
      $GF[i, j] += spectral(k)$
    end for
    {Spatial part}
    for $r = -mimag$ to $mimag$ do
      for $s = -nimag$ to $nimag$ do
        $GF[i, j] += spatial(r, s)$
      end for
    end for
  end for
end for
```

The influence of the number of modes (spectral series) in the execution time is small if this number is not very big in comparison with the number of images (spatial series). In experiments with only one point and 100000 images the percentage of the execution time in the spatial part is approximately 99.5%. However, for the calculation of 10 × 10 points and 1000 images the percentage of the spectral part increases to 7%. In the following, we will concentrate on the parallelization of the spatial part, although for some combinations of the numerical parameters, the parallelization of the spectral part can contribute to reduce the execution time even further.

## 3. Computational systems

Nowadays, most computational systems are composed of multicore components together with one or more GPU cards, and they are combined in homogeneous or heterogeneous clusters. So, scientific codes should be adapted to the characteristics of these systems. The experiments in this paper have been carried out in a variety of systems to cover the different available parallelism combinations. The basic systems are:

- *Saturno* is a NUMA system with 24 cores, Intel Xeon X7542 (hexa-core) processors, 1.87 GHz, 32 GB of shared-memory. The memory is organized hierarchically, and there are several possible memory locations for the data we are working with when solving a problem. There is an NVIDIA Tesla C2050, CUDA Compute Capability 2.0, with 14 SMP of 32 cores, with a total of 448 CUDA cores, 2.8 Gb and 1.15 GHz.
- *Marte* and *Mercurio* are AMD Phenom II X6 1075T (hexa-core), 3 GHz, 15 GB *Marte* and 8 GB *Mercurio*, private L1 and L2 caches of 64 KB and 512 KB, and L3 of 6 MB shared by all the cores. Each machine has an NVIDIA GeForce GTX 590 with two devices, CUDA Compute Capability 2.0, each with 16 SMP of 32 cores, with a total of 512 CUDA cores per device, 1.2 GHz. The total amount of memory in each GPU is 1.5 GB. The two machines are connected in a homogeneous cluster.
- *Luna* is an Intel Core 2 Quad Q6600, 2.4 GHz, 4 GB, with an NVIDIA GeForce 9800 GT, CUDA Compute Capability 1.1, with 14 SMP of 8 cores, with a total of 112 CUDA cores, 0.5 Gb and 1.5 GHz. It is connected with *Marte* and *Mercurio* in a heterogeneous cluster.

The use of these systems allows us to experiment with different configurations so that the versatility and validity of the programs developed are tested. In shared-memory there are from small multicores (*Luna*) to a medium-size NUMA system (*Saturno*). Different types of GPUs are available, including two devices in a card (*Marte* and *Mercurio*), and the three types of parallelism (shared-memory, message-passing and SIMD in GPU) can be studied in homogeneous (*Marte*+*Mercurio*) and heterogeneous (*Marte*+*Mercurio*+*Luna*) clusters. With such a variety of systems, different parallelization strategies must be studied to obtain programs that run efficiently in the different systems and in homogeneous and heterogeneous clusters. For some problem sizes or particular inputs it may be preferable to use only CPU or GPU computation, but in other cases an appropriate combination of the different components may produce a higher reduction in the execution time, so CPU and GPU parallelism have been combined to use all the potential the computational system on hands offers. Thus, in the paper fine and coarse grained approaches are considered, and mixed parallelism combining the two strategies and the three programming paradigms (shared-memory, message passing and SIMD in GPU) is also studied.

### *3.1. Parallelization strategies*

In the scheme of the sequential two-dimensional Green's function (algorithm 1) we see that the parallelism can be introduced at different levels. Two possibilities are considered:

- Fine grained parallelism is obtained by parallelizing the computation of the function between pairs of points, in the loop in $r$ in the code. The advantage of this version is that it facilitates load balancing and it can be adapted to GPUs, where fine grained parallelism is well exploited. On the other hand, the cost of management of the threads can produce a reduction in the performance. The theoretical cost of this version is:

$$t(m, n, nmod, mimag, nimag, c) = mn\left(\left\lceil\frac{nmod}{c}\right\rceil S_1 + \left\lceil\frac{2mimag+1}{c}\right\rceil(2nimag+1)S_2 + R(c) + M(c)\right) \quad (3)$$

  where $R(c)$ is the function corresponding to the time of a reduction, and we consider that the cost of threads or processes management depends linearly on the number of threads or processes, $c$. The values of the constants $S_1$, $S_2$, $R$ and $M$ should be estimated for the system we are working in and the problem we are working with, so the equation can be a tool for execution time prediction and for the selection of the best method to solve the problem.
- It is also possible to have a coarse grained parallelism version just by distributing among the computational components the computation of the Green's functions between pairs of points, which is done by parallelizing

the outermost loop in the algorithm. In this case load balancing is more difficult, but there are fewer synchronization points, and the overhead might be lower than with the fine grain version. Now the theoretical cost is:

$$t(m, n, nmod, mimag, nimag, c) = \left\lceil \frac{m}{c} \right\rceil n\,(nmod\,S_1 + (2mimag + 1)\,(2nimag + 1)\,S_2 + R(c) + M(c)) \quad (4)$$

The term of highest cost in equations 3 and 4 coincides, and so the two versions will produce similar results for big problems (large values of $m$, $n$, $nmod$, $mimag$ and $nimag$) in small systems (low values of $c$). But in a medium-sized system, the mentioned pros and cons determine the best version as a function of the five parameters which represent the problem size. Some selection engine to determine the best method should be developed, or the different types of parallelism could be combined to obtain a mixed version which combines the best of each method, making it necessary to determine the optimum number of computational components of each type to work on the solution of the problem.

## 4. Parallel implementations

This section presents the parallel versions developed, and analyses their performance in the different systems considered. We begin by analysing the basic OpenMP, MPI and GPU versions in the simplest systems, and continue with hybrid combinations of the single parallelism versions.

### *4.1. OpenMP implementations*

The fine and coarse grained implementations described in subsection 3.1 are easily obtained just by parallelizing the corresponding loops with OpenMP. We call the fine grained version `OMPFG`, and the coarse version `OMPCG`. Figure 2 shows the speed-up achieved in *Saturno* with the two versions. The figure shows the optimum achievable speed-up (the number of cores used), the speed-up with `OMPFG` and `OMPCG`, and the theoretical speed-up of `OMPCG` (equation 4). Experiments have been carried out with $mimag \cdot nimag = 1000$ and 10000, and with the same number of source and observer points ($25 \times 25$ and $50 \times 50$). The same sizes and smaller and larger problems are considered for the experiments with the other implementations in the paper. Thus we have experiments from small (100 and 10) to large (100000 and 75) problem sizes. The figure shows the speed-up for small and large sizes. Similar results were obtained in the other systems, but *Saturno* is the largest system, and the variations are better observed in it. Some conclusions are drawn from the figures:

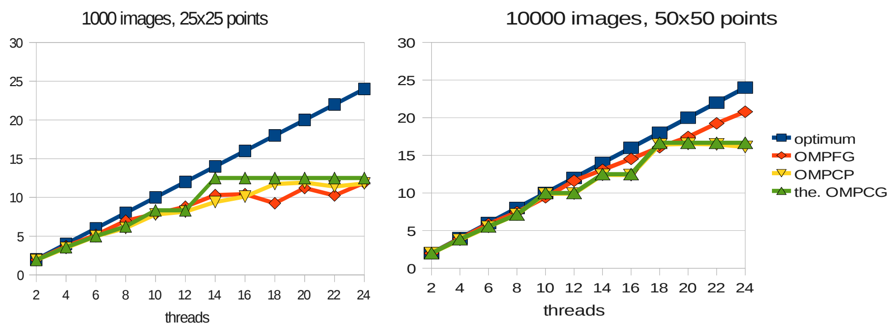

Fig. 2: Speed-up achieved in *Saturno* with two OpenMP implementations.

- The theoretical execution time of `OMPCG` is not well predicted for small problems, but for large problems there is practically no influence of the parameters $R(c)$ and $M(c)$ in equation 4 and the prediction is satisfactory.
- In `OMPFG` the cost of these parameters is higher, which is more apparent for small problems. For the largest problem the results are satisfactory, with a speed-up close to the optimum when not many cores are used.

- The step-wise behaviour of the speed-up of `OMPCG` makes this version non competitive with `OMPFG` for large problems in large systems.

## 4.2. MPI and MPI+OpenMP implementations

An MPI implementation (`MPI`) is obtained from the coarse grained version. The same number of steps of the outermost loop is assigned to each process. When processes are assigned to different cores, the theoretical cost is that of equation 4, but now $R(c)$ and $M(c)$ have higher values. The difference with the use of OpenMP varies depending on the MPI implementation we are using and the system we are working in. For example, in *Saturno*, where MPICH is used, the difference in the execution time of `OMPCG` and `MPI` is lower than 1% in all the cases, which is not a significant difference. But in *Marte*, with AMD processors and where we use OpenMPI, the differences in the execution time are slightly higher, between 1% and 9%.

A hybrid MPI+OpenMP version (`MPI+OMP`) is easily obtained generating $p$ processes and assigning to each process $\frac{m}{p}$ steps of the outermost loop (we consider $m$ is multiple of $p$ for simplicity), and each process generates $h$ threads to work with in the steps assigned to it. Figure 3 compares the speed-up achieved in *Marte* with `OMPCG` and `MPI` when using six threads or processes, and in the homogeneous cluster formed by *Marte+Mercurio* with `MPI` using 12 processes and `MPI+OMP` with 2 processes and 6 threads per process. We can see that in the comparison with 6 threads or processes the speed-up obtained with the use of MPI is higher than that with OpenMP, which is not what we expected, due to the higher cost of managing and communicating processes than threads. The 6 processes in the MPI version are assigned to the two nodes in the cluster, and the unexpected behaviour may be due to a better use of the memory when the data are calculated in two nodes. However, it is slightly better to use OpenMP parallelism than MPI inside a node, and so the hybrid MPI+OpenMP version gives a slightly higher speed-up than MPI with 12 processes. In all the cases, the experimental speed-up is not far from the maximum theoretical and has a shape similar to the theoretical one.

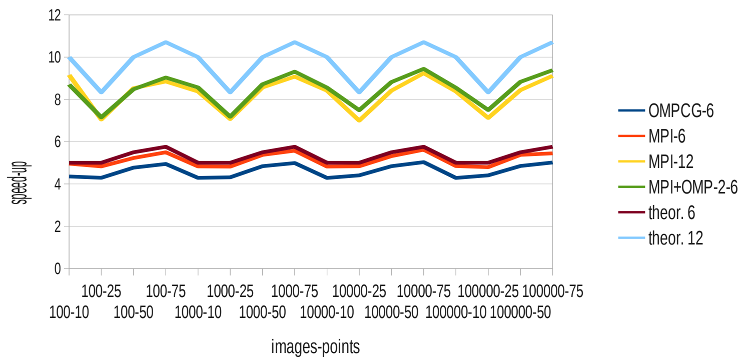

Fig. 3: Comparison of the speed-up achieved in *Marte* with `OMPCG` or `MPI` with 6 threads or processes (lines at the bottom), and in *Marte+Mercurio* with `MPI` with 12 processes and `MPI+OMP` with 2 processes and 6 threads each process (lines at the top), and maximum theoretical speed-up with 6 and 12 computing components.

## 4.3. GPU implementations

Algorithm 1 follows a scheme appropriate for SIMD computing which leads to the satisfactory results obtained with the fine grained OpenMP version, and makes it adequate for GPU computing. A GPU version (`GPU`) is obtained just by kernel calling for the computation of the Green's functions of each pair of points, with GPU cores computing partial sums, which are accumulated to obtain the total sum. So, satisfactory results are obtained when the number of images and points is large enough. This is seen in figure 4, where the execution times of `OMPCG` with a number of threads equal to that of available cores and of `GPU` are compared in the three systems (*Luna*, *Marte* and *Saturno*) for the different numbers of images and points experimented with. The figure shows the quotient of the execution time of OMPCG with respect to GPU. A reduction in the execution time for large problems is obtained with GPU, and the reduction changes with the system due to the speed differences of CPUs and GPUs. The small value of the quotient for small problems is due to the high cost of the GPU initialization and the reduced computational cost of the problem assigned to the GPU, which means that for small problem sizes the OpenMP implementation is preferable in some systems where the CPU speed is relatively high in comparison with the GPU.

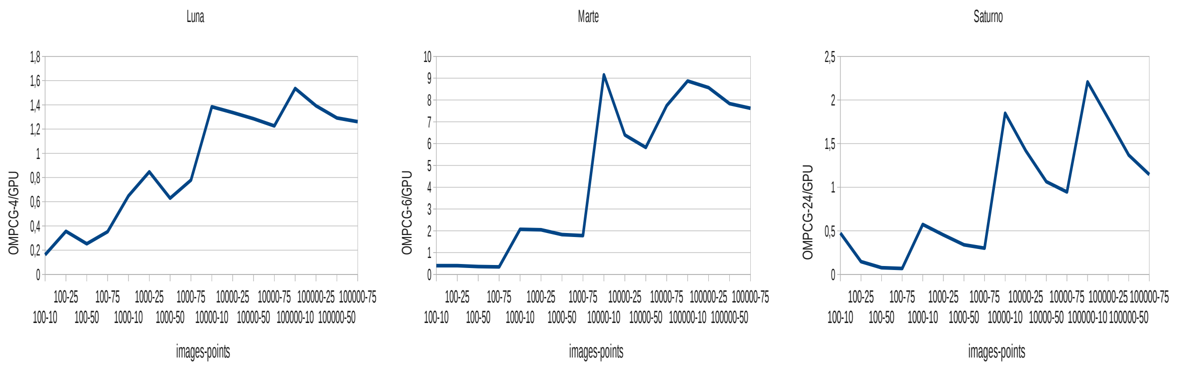

Fig. 4: Quotient of the execution time of OMPCG with respect to GPU in different systems. In each system the number of threads coincides with the number of cores in the system.

The GPU can be further exploited. For that it would be necessary to optimize the GPU version for the particular card we are working with. As an alternative, we are using a basic version of the algorithm, parameterized to adapt it to the card in use. GPU is parameterized so that $g$ kernels are started in parallel inside an OpenMP parallel for-loop with $g$ threads, one kernel for each thread. The execution time varies with $g$, and the preferred value for $g$ depends on the system and the problem size. Figure 5 shows the quotient of the execution time of GPU when no simultaneous kernels are started with respect to that when several kernels are started at the same time. The behaviour varies with the system and the problem size. In *Luna* and *Saturno* the highest values are obtained with the largest problems, but the opposite occurs in *Marte* (with a more modern graphic card) where for large problems it is preferable not to start several kernels.

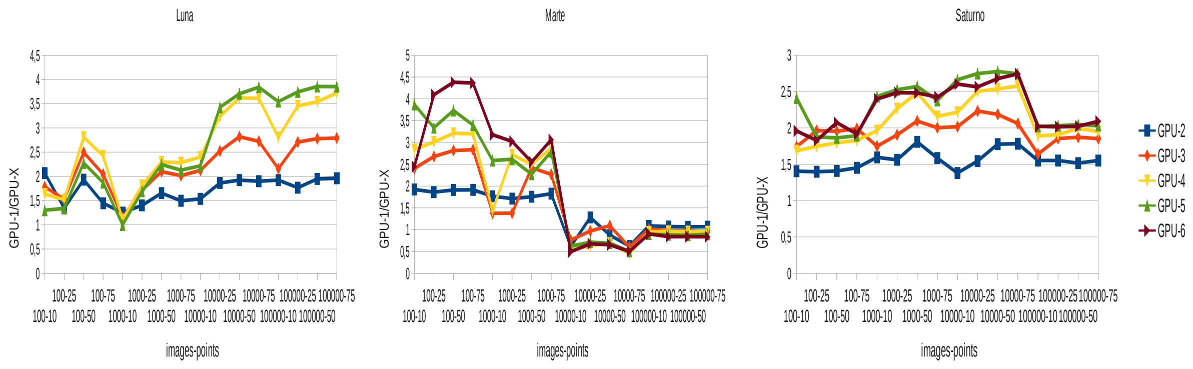

Fig. 5: Quotient of the execution time of GPU without simultaneous kernels with respect to the execution time when starting several kernels

### 4.4. MPI+OpenMP+GPU implementations

We have seen the preferred implementation is in general and for large problems GPU, but in some cases lower execution times can be obtained with a different version or combination. For instance: it is possible to use a more optimized C compiler for OpenMP or MPI versions; in large systems for small problems the initialization cost of GPU can produce large times which are surpassed with the use of a large number of cores; in clusters with several CPUs and GPUs message-passing and GPUs are combined, etc. So, we have developed a hybrid algorithm combining the three programming paradigms. The scheme is shown in algorithm 2. The steps of the outermost loop are distributed among $p$ MPI processes (in the scheme $m$ is considered divisible by $p$ for simplicity). Each process generates $h+g$ OpenMP threads, $h$ to work in the CPUs and $g$ for kernels to work in the GPU. The steps in the loop are dynamically assigned, so that the load distribution dynamically adapts to the relative speed of the cores and the GPU. The OpenMP threads perform coarse grained computation, and the CUDA kernels use fine parallelism. The combinations of $p$, $h$ and $g$ give the different versions shown in table 1.

**Algorithm 2** Hybrid algorithm for the computation of two-dimensional Green's functions: GF-2D-HY($m$,$n$,$nmod$,$mimag$,$nimag$,$p$,$h$,$g$)

```
For each MPI process $P_k$, $0 \leq k < p$:
omp_set_num_threads($h+g$)
#pragma omp parallel for schedule(dynamic,1)
for $i = k\frac{m}{p}$ to $(k+1)\frac{m}{p} - 1$ do
  $node$=omp_get_thread_num()
  if $node < h$ then
    Compute with OpenMP thread
  else
    Call to CUDA kernel
  end if
end for
```

| $p \times (h+g)$ | $1+0$ | $h+0$ | $0+g$ | $h+g$ |
|---|---|---|---|---|
| 1 | SEQ | OMP | GPU | OMP+GPU |
| $p$ | MPI | MPI+OMP | MPI+GPU | MPI+OMP+GPU |

Table 1: Hybrid versions for the different number of MPI processes, $p$, OpenMP threads, $h$, and CUDA kernels, $g$

Until now, we have analysed the behaviour of the versions not combining MPI and GPU. The versions combining these two paradigms can be evaluated in multicore+GPU systems, but their natural environment are clusters of multicore+GPU. Because the best results for large problems are obtained with the GPU version, the use of the two GPU cards in *Marte+Mercurio* could give lower execution times. The two cards can be combined in different ways. Figure 6 compares different ways of using the two cards. The figure on the left shows the quotient of the execution time when using only one GPU and one kernel with respect to the time when using the two GPUs with a variable number of kernels. The use of the GPUs in the two nodes gives a practically optimum speed-up, and when more kernels are started the behaviour is similar to that in *Marte* (Figure 5). The graph on the right in figure 6 compares the execution time when starting several kernels in the two GPUs, with one process in each node and several kernels in each process, and with several processes in each node and one kernel per process. In both cases only the GPUs in the two nodes are used, but the management of simultaneous kernels working inside different processes seems to be a better option (at least for the system we are working in).

## 5. Conclusions and future work

The application of three parallel programming paradigms to the computation of 2D rectangular waveguide Green's functions has been analysed. Shared-memory, message-passing and SIMD GPU computing have been

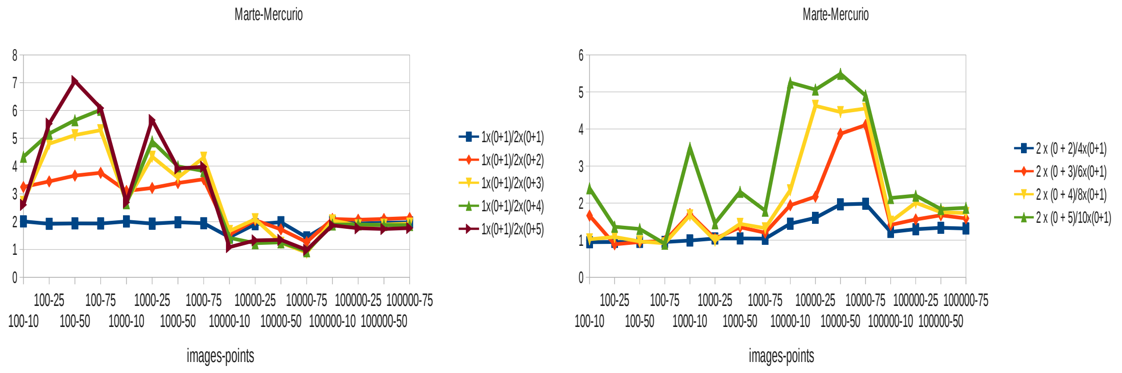

Fig. 6: Analysis of the use of several kernels in each node (*Marte* and *Mercurio*). Left: speed-up when using the two nodes, one process per node and several kernels per process, with respect to one GPU with one kernel. Right: quotient of the execution time when using the two GPUs with one process per node and several kernels per processor, with respect to when starting several processes per node and one kernel per process, with the same total number of kernels.

applied, isolated and combined in different ways. Algorithms have been developed with the inclusion of parameters that can be configured for low execution times in different systems, and the optimum values of the parameters depend on the computational system and the problem size. The parameters to be obtained are the number of MPI processes, of slave OpenMP threads inside each MPI process, and of kernels to be computed in the GPUs. Table 2 compares the lowest execution time obtained with the different configurations experimented with the times of two configurations that seem appropriate for each system. *Luna* is a quadcore with a GPU, so the lowest execution time (Low.) is compared with that of 4 threads ($1 \times (4+0)$) and with the GPU version with 3 kernels ($1 \times (0+3)$). *Saturno* is a multicore with 24 cores with a GPU, and the lowest execution time is compared with the versions $1 \times (24+0)$ and $1 \times (0+3)$. When using *Marte* and *Mercurio* as a homogeneous cluster, the lowest execution time is compared with an MPI+OpenMP version with one process in each node and 6 threads per process ($2 \times (6+0)$), and with an MPI+GPU version with one process per node and 3 kernels in each process ($2 \times (0+3)$). The times are in seconds, and for the lowest execution times the numbers of processes, threads and kernels which give the lowest time are shown.

More optimized versions can be developed to better exploit the memory hierarchy, both in multicores and in GPU cards, and they can be integrated in the hybrid scheme, to produce additional reduction in the execution time. Table 2 shows that the best configuration changes with the system and the problem size, and a large reduction in the execution time with respect to configurations that seem appropriate for the system in question is obtained. It would, therefore, be interesting to have some auto-tuning engine to select satisfactory values for the number of processes, threads and kernels. The auto-tuning methodology should be independent of the basic versions included in the hybrid scheme. We are working on the development of such an engine, with both empirical and modelling techniques. The use of models gives satisfactory results in OpenMP and MPI versions (Figure 2), but it is much more difficult to obtain satisfactory models for GPU versions.

Some experiments have been carried out in a heterogeneous system (*Marte*+*Mercurio*+*Luna*), but the best results were always obtained with the homogeneous system *Marte*+*Mercurio*. For a better management of the heterogeneity it is necessary to include parameters to indicate different loads for the different nodes and different numbers of OpenMP threads and CUDA kernels in each node. This increment in the parameters will make the development of an auto-tuning engine more difficult.

| | *Luna* | | | *Saturno* | | | *Marte+Mercurio* | | |
|---|---|---|---|---|---|---|---|---|---|
| images-points | Low. | 1×(4+0) | 1×(0+3) | Low. | 1×(24+0) | 1×(0+3) | Low. | 2×(6+0) | 2×(0+3) |
| 100-10×10 | 0.0412 1×(4+0) | 0.0412 | 0.211 | 0.0058 12×(2+1) | 0.0275 | 0.0336 | 0.0075 12×(1+2) | 0.0134 | 0.0245 |
| 100-25×25 | 0.258 1×(4+0) | 0.258 | 0.478 | 0.0193 4×(6+1) | 0.0537 | 0.186 | 0.0443 6×(0+5) | 0.0999 | 0.0863 |
| 100-50×50 | 0.876 1×(4+0) | 0.876 | 1.39 | 0.0762 6×(4+1) | 0.113 | 0.748 | 0.167 12×(0+5) | 0.337 | 0.327 |
| 100-75×75 | 1.90 1×(4+2) | 1.99 | 2.76 | 0.172 3×(8+1) | 0.226 | 1.67 | 0.411 4×(0+5) | 0.710 | 0.708 |
| 1000-10×10 | 0.249 1×(2+2) | 0.254 | 0.341 | 0.0173 24×(1+1) | 0.0430 | 0.0431 | 0.0084 12×(0+1) | 0.0821 | 0.0291 |
| 1000-25×25 | 0.957 1×(2+3) | 1.58 | 1.09 | 0.106 6×(4+1) | 0.211 | 0.246 | 0.0603 10×(0+1) | 0.611 | 0.114 |
| 1000-50×50 | 3.53 1×(1+3) | 5.59 | 4.23 | 0.424 4×(6+1) | 0.637 | 0.901 | 0.201 10×(0+5) | 2.02 | 0.555 |
| 1000-75×75 | 7.08 1×(0+4) | 12.6 | 8.02 | 0.712 6×(4+1) | 1.27 | 2.12 | 0.565 8×(0+5) | 4.25 | 1.04 |
| 10000-10×10 | 0.627 2×(0+3) | 2.37 | 0.807 | 0.063 4×(0+2) | 0.391 | 0.105 | 0.0260 10×(1+5) | 0.740 | 0.097 |
| 10000-25×25 | 2.92 1×(0+5) | 13.4 | 3.96 | 0.386 2×(0+4) | 1.88 | 0.595 | 0.209 10×(0+1) | 5.28 | 0.671 |
| 10000-50×50 | 8.74 4×(0+3) | 51.1 | 14.1 | 1.76 4×(0+2) | 5.64 | 2.42 | 0.755 12×(0+1) | 17.9 | 4.52 |
| 10000-75×75 | 18.9 4×(0+3) | 110 | 32.8 | 2.83 6×(4+1) | 11.3 | 5.79 | 1.83 12×(0+1) | 38.0 | 10.4 |
| 100000-10×10 | 4.21 4×(0+3) | 23.7 | 7.15 | 0.694 3×(0+2) | 3.74 | 1.04 | 0.322 10×(0+5) | 7.32 | 0.805 |
| 100000-25×25 | 17.9 4×(0+3) | 132 | 35.1 | 3.88 2×(0+4) | 18.9 | 5.71 | 2.09 12×(0+4) | 52.1 | 5.54 |
| 100000-50×50 | 98.7 1×(0+5) | 491 | 137 | 18.5 4×(0+2) | 58.0 | 22.6 | 10.3 10×(0+3) | 177 | 21.8 |
| 100000-75×75 | 222 1×(0+5) | 1077 | 306 | 27.9 6×(4+1) | 113 | 51.4 | 21.7 12×(0+4) | 377 | 47.9 |

Table 2: Comparison of the lowest execution time with OpenMP versions with the number of threads equal to the number of cores in the system and with a GPU version with three kernels, in *Luna* and *Saturno*; or with an MPI+OpenMP version with 2 processes and 6 threads each process and with an MPI+GPU version with 2 processes and 3 kernels each process, in *Marte+Mercurio*.

**Acknowledgment**

Partially supported by Fundación Séneca, Consejería de Educación de la Región de Murcia, 08763/PI/08, and the Spanish Ministry of Education and Science through TIN2012-38341-C04-03 and TEC2010-21520-C04-04, and Feder fundings.

## References

[1] J. Heinbockel, Mathematical Methods for Partial Differential Equations, Trafford Publishing, 2004.

[2] D. G. Duffy, Green's Functions with Applications, Studies in Advanced Mathematics, Chapman & Hall/CRC, 2001.

[3] F. J. Pérez, F. D. Quesada, D. Cañete, A. Álvarez, J. R. Mosig, A novel efficient technique for the calculation of the Green's functions in rectangular waveguides based on accelerated series decomposition, IEEE Transactions on Antennas and Propagation 56 (10) (2008) 3260–3270.

[4] F. Capolino, D. R. Wilton, W. A. Johnson, Efficient computation of the 2-D Green's function for 1-D periodic layered structures using the Ewald method, in: IEEE Antennas and Propagation Society International Symposium, 2002, pp. 194–197.

[5] C. Pérez-Alcaraz, D. Giménez, A. Álvarez-Melcón, F. D. Quesada-Pereira, Parallelizing the computation of Green functions for computational electromagnetism problems, in: IPDPS Workshops, 2012, pp. 1370–1377.

[6] A. Kustepely, A. Q. Martin, On the splitting parameter in the Ewald method, Trans. Microw. Guided Wave Lett. 10 (5) (2000) 168–170.