Document downloaded from:

http://hdl.handle.net/10251/81255

This paper must be cited as:

Lamas Daviña, A.; Ramos Peinado, E.; Román Moltó, JE. (2016). Optimized analysis of isotropic high-nuclearity spin clusters with GPU acceleration. Computer Physics Communications. 209:70-78. doi:10.1016/j.cpc.2016.08.014.

The final publication is available at

http://dx.doi.org/10.1016/j.cpc.2016.08.014

Copyright Elsevier

Additional Information

This is the author's version of a work that was accepted for publication in Computer Physics Communications. Changes resulting from the publishing process, such as peer review, editing, corrections, structural formatting, and other quality control mechanisms may not be reflected in this document. Changes may have been made to this work since it was submitted for publication. A definitive version was subsequently published in Computer Physics Communications, vol. 209, (2016). DOI 10.1016/j.cpc.2016.08.014.

Optimized analysis of isotropic high-nuclearity spin clusters with GPU acceleration[☆]

A. Lamas Daviña, E. Ramos, J. E. Roman[*]

D. Sistemes Informàtics i Computació, Universitat Politècnica de València, Camí de Vera s/n, 46022 València, Spain

Abstract

The numerical simulation of molecular clusters formed by a finite number of exchange-coupled paramagnetic centers is very relevant for many applications, modeling systems between molecules and extended solids. In the context of realistic scenarios, many centers need to be considered, and thus the required computational effort grows very fast. In a previous work [E. Ramos *et al.*, Comput. Phys. Commun. 181 (2010)], a set of parallel programs were presented with standard message-passing parallelization (MPI) for both anisotropic and isotropic systems. In this work, we have further developed the code for isotropic models. On one hand, the computational cost has been significantly reduced by avoiding some of the matrix diagonalizations, corresponding to blocks with negligible contribution for the particular configuration. On the other hand, we have extended the parallelization in order to exploit available graphics processing units (GPUs). The new MPI-GPU paradigm reduces the computational time by at least one additional order of magnitude and enables the resolution of larger problems.

Keywords: Molecular magnetism, High-nuclearity spin clusters, Large-scale eigenvalue problem, Graphics processing units (GPUs)

[☆]This work was partially supported by the Spanish Ministry of Economy and Competitiveness under grant TIN2013-41049-P. Alejandro Lamas Daviña was supported by the Spanish Ministry of Education, Culture and Sport through a grant with reference FPU13-06655.

[*]Corresponding author.

Email addresses: alejandro.lamas@dsic.upv.es (A. Lamas Daviña), ramos@dsic.upv.es (E. Ramos), jroman@dsic.upv.es (J. E. Roman)

1. Introduction

This work is concerned with numerical simulation in the context of molecular magnetism, where the goal is to analyze molecule-based magnetic materials with interesting properties that are becoming important in applications such as high-density information storage. In particular, we focus on magnetic clusters, that is, molecular assemblies of a finite number of exchanged-coupled paramagnetic centers. These assemblies are midway between small molecular systems and the bulk state, being possible to model them as the former, with quantum mechanical principles, rather that with the simplifications required for the latter. This allows for deeper understanding of the magnetic exchange interaction. However, when analyzing clusters with a growing number of exchanged-coupled centers, the complexity soon becomes prohibitive due to the lack of translational symmetry within the cluster.

A flexible methodology for studying high nuclearity spin clusters is based on the use of the technique of irreducible tensor operators [1, 2]. This approach enables the evaluation of eigenvalues and eigenvectors of the system, then deriving from them the magnetic susceptibility, the magnetization, as well as the inelastic neutron scattering spectra. This functionality is provided by the Magpack package [3], covering both anisotropic exchange interactions as well as the simpler isotropic case. Magpack is a set of serial Fortran codes, whose scope of applicability is limited to very small number of centers, due to the high computational cost associated with the creation of the Hamiltonian matrices followed by their diagonalization. The dimension of these matrices grows rapidly with the number of spin cluster basis functions.

In a previous work [4], we reworked the Magpack codes in order to be able to cope with large-scale problems, with many spins. The newly developed codes are parallel, hence enabling the use of large supercomputers, and are based on carrying out a *partial* diagonalization of the system matrices by means of SLEPc, the Scalable Library for Eigenvalue Problem Computations [5]. In this way, the computational load is shared across the various processors participating in the parallel computation, and on the other hand, only a modest percentage of the eigenvalues and eigenvectors is obtained, thus avoiding many superfluous calculations. Still, the computational requirements can be huge and in this work we present further developments to improve the efficiency as much as possible.

This work focuses on ParIso, the parallel code for coping with isotropic spin clusters (although the developments of section 4 could be easily ex-

tended to the anisotropic case). We have made two major improvements in this code. On one hand, we have added a preprocessing step aiming at reducing the number of partial diagonalizations required during the computation. This step computes bounds for the spectrum of the submatrices in which the main Hamiltonian matrix decomposes, and estimates the number of eigenvalues to compute in each submatrix. With this strategy, the computation provides equally satisfactory results with a considerable decrease of the time of calculation, although we will discuss that the way in which the number of eigenvalues is estimated may not work in all cases. On the other hand, we have extended the parallelization paradigm in order to exploit available graphics processing units (GPUs) that are available in many supercomputers nowadays. This represents a finer level of parallelism, to be added to the already mentioned coarse-grain parallelism, that provides an additional speedup factor that can significantly reduce the turnaround time of the computation.

The rest of the paper is organized as follows. Section 2 provides a brief overview of the ParIso code. In section 3 we describe how to preprocess the Hamiltonian blocks and avoid some of the diagonalizations, based on the population value indicated by the user. In section 4 we discuss the main issues related to parallelization of the codes, with GPU as well as with a hybrid GPU-MPI parallelization, and show results concerning the performance of the developed programs. We wrap up with some concluding remarks.

2. The ParIso code

Consider a spin cluster composed of an arbitrary number of magnetic sites, N, with local spins. In order to obtain the set of spin cluster basis functions, the local spins are successively coupled,

$$\left| S_1 S_2 \left(\tilde{S}_2\right) S_3 \left(\tilde{S}_3\right) ....S_{N-1} \left(\tilde{S}_{N-1}\right) S_t \right\rangle = \left| \left(\tilde{S}\right) S_t \right\rangle \tag{1}$$

where $\tilde{S}_i$ refers to the intermediate spin values $S_1 + S_2 = \tilde{S}_2$, $\tilde{S}_2 + S_3 = \tilde{S}_3$, etc., $(\tilde{S})$ is the full set of $\tilde{S}_i$ ($N-1$ intermediate spin states) and S_t is the total spin [1]. The system matrix can be evaluated by applying the Hamiltonian to the created basis set, by means of the irreducible tensor operators (ITO) technique. The advantage of this methodology is that it allows us to completely take into consideration all kinds of magnetic exchange interactions between the metal ions comprised in clusters of arbitrary size. This

is done by expressing the contributions to the spin Hamiltonian (expressed in terms of the conventional spin operators) as a function of the generalized Hamiltonian (written in terms of ITO's).

In this paper we focus on the ParIso code [4], that computes the quantities mentioned above for isotropic systems, that is, it generates the spin functions of the system, calculates the energy matrix and obtains its eigenvalues and eigenvectors, all this in parallel. Isotropic systems are a special case where only the isotropic and biquadratic exchange terms are present in the spin Hamiltonian. These terms have the property of not mixing functions with different quantum number S and not breaking the degeneracy of levels with the same S and different M. This decouples the energy matrix into several submatrices, one per each different S quantum number. Taking into account the ITO technique it is possible to eliminate this M quantum number and reduce the size of each S submatrix by a factor of $2S+1$. No Zeeman terms are present in this case, during the diagonalization, but would be included after it.

Thus, in the case of isotropic systems, the energy matrix can be written as a block diagonal matrix,

$$A = \begin{bmatrix} A_1 & & & \\ & A_2 & & \\ & & \ddots & \\ & & & A_b \end{bmatrix}, \qquad (2)$$

where each of the b blocks is a symmetric sparse matrix of different dimension. Finding the leftmost eigenvalues of A amounts to computing the leftmost eigenvalues of each of the blocks, A_i. Thus, the structure of the program ParIso is geared to this block structure, where one partial diagonalization is carried out per block.

The main steps of the computation are the following:

1. Setup of data containing the information of the cluster.
2. For each diagonal block, $i = 1, \ldots, b$, do:
   - (2.1) Generation of starting spin functions.
   - (2.2) Evaluation of energy submatrix, A_i. All nonzero elements are computed and assembled into the matrix.
   - (2.3) Partial diagonalization of A_i. Given the eigenvalue relation $A_i x = \lambda x$, a subset of the spectrum is computed, corresponding to the leftmost eigenvalues.

3. Generation of final results.

The partial diagonalization of each matrix block A_i is carried out by the thick-restart Lanczos method, as detailed in [4]. In particular we use the implementation provided by SLEPc, the Scalable Library for Eigenvalue Problem Computations [5], a software package for the solution of large-scale eigenvalue problems on parallel computers. SLEPc provides a collection of eigensolvers for different kinds of eigenvalue problems. For computing extreme eigenvalues of symmetric matrices, one of the most effective methods is thick-restarted Lanczos [6]. In SLEPc, this solver includes the possibility of specifying the `mpd` parameter (maximum dimension of the projected problem), described in [4], that enables the computation of a large number of eigenpairs in chunks, by locking eigenpairs converged at each restart. This feature is missing in other software such as ARPACK [7].

SLEPc is built on top of PETSc, the Portable, Extensible Toolkit for Scientific Computation [8]. From the software engineering perspective, both PETSc and SLEPc have a modern design, that allows the application programmer to work with abstract data objects and solvers, without knowing the details of internal data structures. This confers the software interesting properties such as scalability to a large number of processors and portability to a wide range of parallel platforms, including those having GPU devices. This latter feature will be exploited in section 4.

3. Optimization of spin eigenanalysis

In the ParIso code, the individual blocks of matrix A in (2) are treated separately. The dimensions of these blocks vary widely, ranging from 1 to a hundred thousand or even more, and the percentage of nonzero elements of each (large) block is about 1–2%. The generation of the matrix is made submatrix by submatrix (each submatrix is a spin energy). Not all submatrices need to be in memory simultaneously, since after generating one submatrix it is possible to compute its partial diagonalization and then the matrix is no longer needed and can be destroyed. For all these reasons, we are able to calculate much larger systems than in the anisotropic case.

The optimization in ParIso that we introduce in this section tries to avoid the computation associated with those blocks that are not going to contribute significantly to the aggregated result. This allows a drastic reduction of the time necessary for the overall execution. The rationale is that

Lanczos methods can provide robust bounds for the spectrum of each of the submatrices with a relatively small cost. Based on the location of the first and last eigenvalues of each block, we can discard some of the blocks if they lie outside the range of interest specified by the user.

The resulting algorithm performs the following steps:

1. In a first pass, the program traverses all the submatrices and, for each of them, it calculates the first and last eigenvalue, filling a table with the information obtained for all blocks.
2. From the table of minimum and maximum eigenvalues, the program determines a point of energetic cut depending on the value of population to be considered (provided as a user input parameter).
3. With the cut point and the dimension of each submatrix, the code determines how many eigenvalues are required in each of them. A number of zero implies that the corresponding submatrix can be discarded, since its energetic levels are outside the requested range.
4. Finally, a second pass computes the wanted eigenvalues, traversing only the submatrices that are going to provide useful information.

This process is illustrated in Figure 1 with an example. The horizontal lines represent the span of the spectrum of each of the matrix blocks (18 in this case). The dimensions of the blocks range from 1 (the last one) to 3150. The vertical line represents the energetic cut to be considered. In the plot we can easily see that this energetic cut leaves out 8 of the submatrices, which are the largest ones in this particular example. The vertical red marks on the horizontal lines represent those eigenvalues that must actually be computed. Note that the plot does not show real eigenvalues, but an estimation based on the size of the submatrix and assuming a roughly uniform distribution of eigenvalues. This approximation is only valid for those systems with one spin state overstabilized from the rest, e.g., completely ferromagnetic systems with very high magnetic coupling which stabilize the high spin ground state or completely antiferromagnetic clusters with a very stabilized intermediate spin state. In this paper, for validation we use the simple systems with this property shown in Table 1, the first one with ferromagnetic and antiferromagnetic interactions and the rest with only ferromagnetic interactions. It remains as a future work to try to extend this procedure to cover more general systems.

We remark that in our code it is also possible to specify an upper bound (`maxev`) for the number of eigenvalues to compute in any submatrix. This is

Table 1: Systems used.

System	Size	Largest submatrix size
$7Mn^{2+}$	24017	3150
$8Mn^{2+}$	135954	16576
$9Mn^{2+}$	767394	88900
$9Mn^{2+} + 1Cu^{2+}$	1534788	177100

useful for large problems where a given threshold would imply computing too many eigenvalues. As an example, in the problem of Figure 1 the maximum number of requested eigenvalues is 33, and setting `maxev`=25, for instance, would imply in practice shifting the vertical line a certain amount to the left.

The above procedure avoids lots of unnecessary computations, with the corresponding gain in the overall simulation time. The only drawback is that the submatrices that participate in the second pass must be regenerated since it is not possible to keep all blocks in memory simultaneously. Still, the new methodology is much faster than the former one, as will be shown below.

Apart from modifying the algorithm, we have also optimized the memory usage by (1) adjusting the dimension of the various arrays to fit the maximum submatrix size, and (2) converting real variables to integer ones whenever possible. These two actions have allowed a significant reduction of the memory footprint per MPI process, up to one third of the previous values. All in all, these improvements enable to cope with larger, more challenging problems.

3.1. Code validation

We have carried out a number of numerical experiments in order to validate the correctness of the parallel code, and to evaluate its performance. The computer system used for the computational experiments in this section is Tirant, an IBM cluster consisting of 4096 JS20 blade computing nodes, each of them with two 64-bit PowerPC 970FX processors running at 2.2 GHz, interconnected with a low latency Myrinet network.

For the validation of the code, we have used a small system ($7Mn^{2+}$) as a test case whose submatrices have the following dimensions: 1050, 1974, 2666, 3060, 3150, 2975, 2604, 2121, 1610, 1140, 750, 455, 252, 126, 56, 21, 6 and 1. This is the system used in Figure 1 to illustrate the algorithm, and

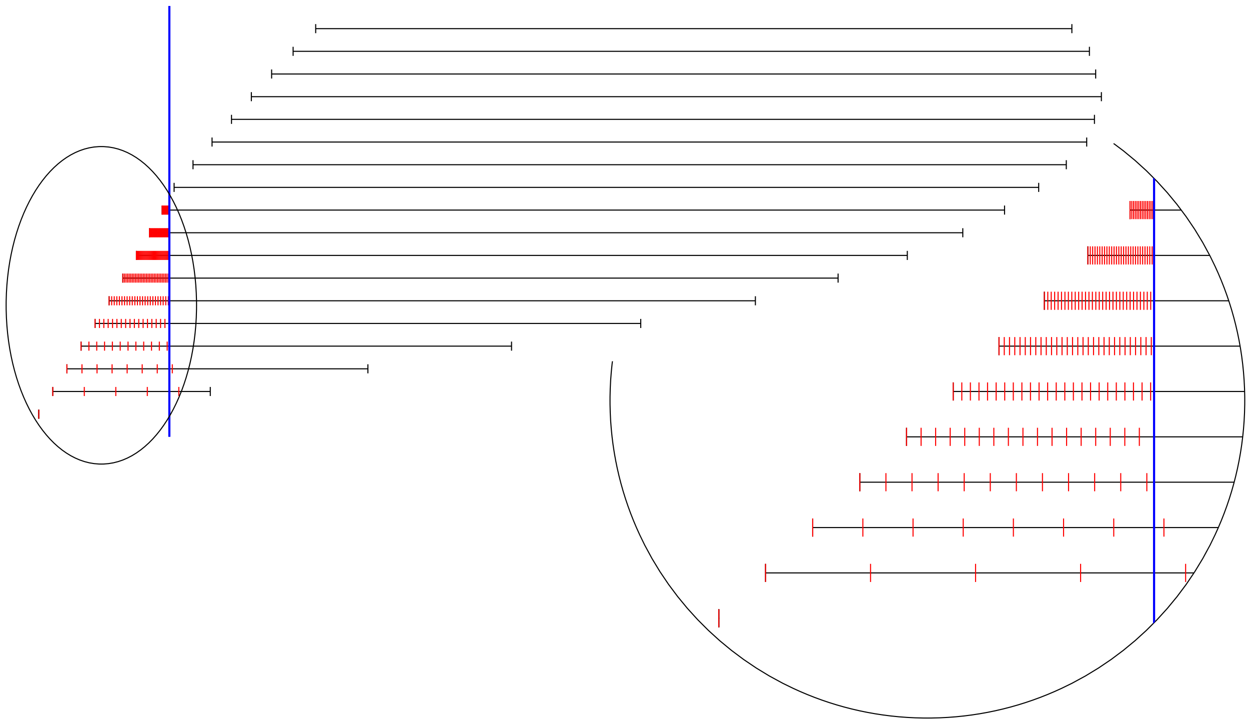

Figure 1: Example of energetic cut for a test problem with 18 submatrices. Only eigenvalues located to the left of the vertical line need to be computed. The number of eigenvalues to compute depends on the dimension of the submatrix. A zoom of the region of interest is shown on the right.

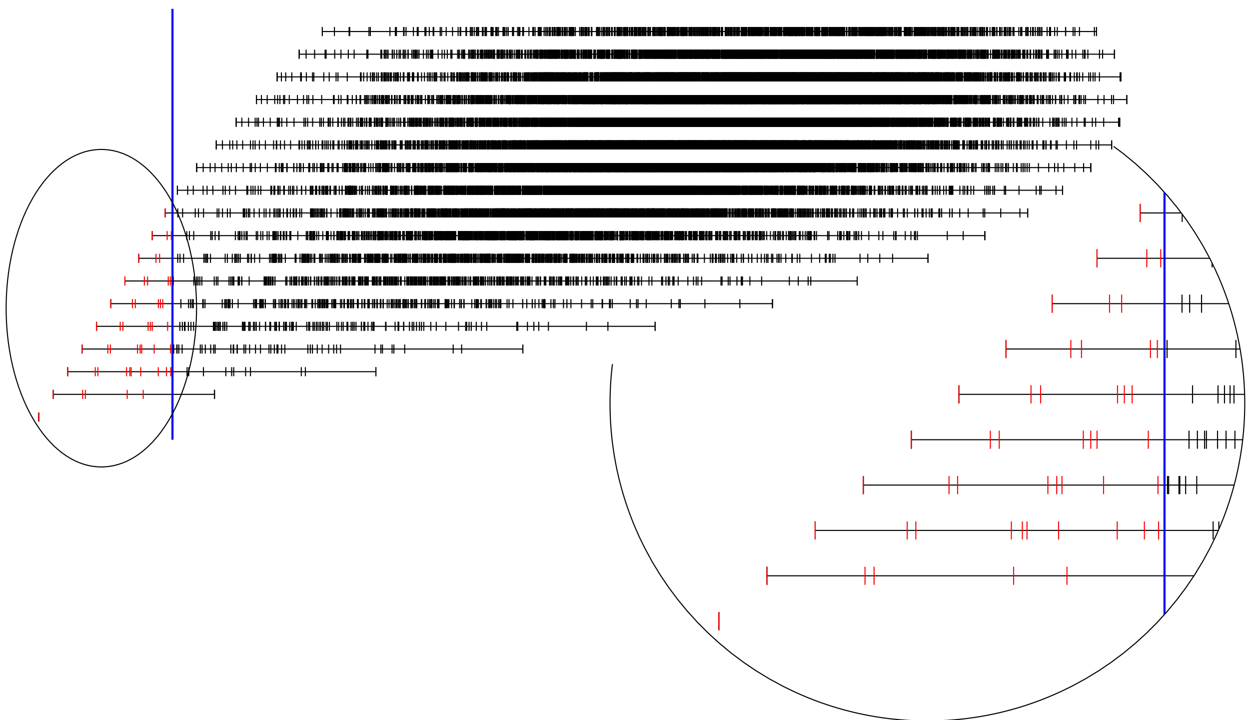

Figure 2: Full representation of the $7Mn^{2+}$ system, showing the energetic cut and the computed eigenvalues.

Table 2: Value of energetic cut for different values of the population parameter in the $7Mn^{2+}$ system, and the corresponding computation time (in seconds) with one processor.

`pobla`	Energetic cut (cm^{-1})	Computation time (s)
10^{-1}	480.11	120
10^{-2}	960.23	133
10^{-3}	1440.35	217
10^{-4}	1920.47	313

its real data can be seen in Figure 2. The largest block is of size 3150 and the susceptibility was computed for a range of temperatures up to 300K. In this case, the cut value varies as a function of the population (`pobla`, a user input parameter), as shown in Table 2, and we have not considered setting the `maxev` parameter because the number of required eigenvalues is small. The vertical line depicted in Figure 2 corresponds to the energetic cut of `pobla`$=10^{-2}$. The total number of eigenvalues per spin of this system and the amount of them selected to compute after the energetic cut has been determined are shown on the left side of Table 3.

The computation time with one processor is also shown in Table 2. As a reference, the total time required in the case of computing all eigenvalues of all submatrices is 1917 seconds (this will be referred to as the *full computation*).

Figure 3 shows the susceptibility results (product χT against the temperature) for the different values of the population, compared to the case of the full computation. For low temperatures, all lines match, while for higher temperatures the graphs diverge from the full computation, being more accurate for smaller values of `pobla`, as expected.

3.2. Performance analysis

In order to assess the performance of the code, both serially and in parallel, we have used a larger system ($8Mn^{2+}$), whose submatrices have dimensions: 2666, 7700, 11900, 14875, 16429, 16576, 15520, 13600, 11200, 8680, 6328, 4333, 2779, 1660, 916, 462, 210, 84, 28, 7 and 1. The largest block is of order 16576 and the temperature range of the simulation is 300K. In this case, the number of eigenvalues that are needed for a given energetic cut may be quite large, so we are interested in studying the influence of the

Table 3: Total number of eigenvalues (n) for each of the spins and the selected values to be computed (k), for the systems $7Mn^{2+}$ (left) and $8Mn^{2+}$ (right) using `maxev`=1000 and `pobla`=10^{-2}.

	Eigenvalues	
Spin	n	k
0	1050	0
1	1974	0
2	2666	0
3	3060	0
4	3150	0
5	2975	0
6	2604	0
7	2121	0
8	1610	14
9	1140	28
10	750	33
11	455	30
12	252	24
13	126	18
14	56	12
15	21	8
16	6	5
17	1	1

	Eigenvalues	
Spin	n	k
0	2666	125
1	7700	372
2	11900	608
3	14875	807
4	16429	1000
5	16576	1000
6	15520	981
7	13600	896
8	11200	766
9	8680	614
10	6328	462
11	4333	86
12	2779	0
13	1660	0
14	916	0
15	462	0
16	210	0
17	84	0
18	28	0
19	7	0
20	1	0

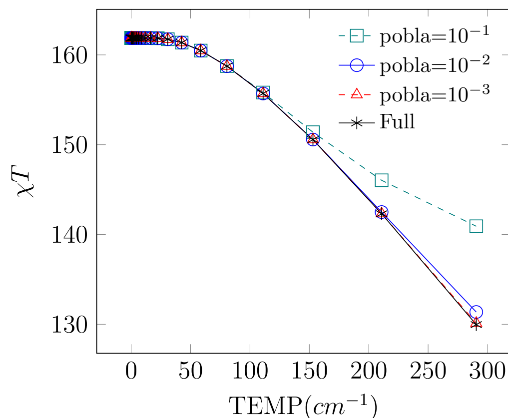

Figure 3: Susceptibility for different values of the population in the $7Mn^{2+}$ system.

Table 4: Value of energetic cut for different values of the `maxev` parameter in the $8Mn^{2+}$ system.

`maxev`	Energetic cut (K)
1000	1312.39
3000	3936.93
5000	6561.56

`maxev` parameter. We have set this value to 1000, 3000 and 5000 eigenvalues, and it is the parameter that determines the energetic cut, see Table 4. The total and computed eigenvalues for a value of `maxev`=1000 can be seen on the right side of Table 3. If we set different values of `pobla` (10^{-1}, 10^{-2}, and 10^{-3} as in the previous example), we will hardly appreciate any noticeable differences in the results.

The sequential execution time corresponding to the full computation is 403000 seconds. Table 5 shows execution times in parallel for the three values of `maxev`. Even computing as many as 5000 eigenvalues, the sequential time has been reduced to one fourth of the original one. And with parallel computing we reduce the times even further, with a speedup factor of around 10 with 16 processors.

In order to assess the accuracy of the computed susceptibility, in Figure 4 we compare the plot obtained with the full computation against the results

Table 5: Parallel execution time (in seconds) for the $8Mn^{2+}$ system with different values of `maxev` and increasing number of MPI processes.

	Processes				
`maxev`	1	2	4	8	16
1000	27351	17020	9543	5167	2810
3000	80172	51217	26932	14476	7926
5000	106384	62590	32909	17655	9690

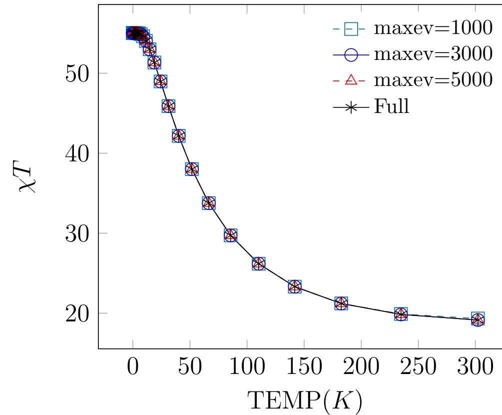

Figure 4: Susceptibility for different values of the `maxev` parameter in the $8Mn^{2+}$ system.

for the three values of `maxev` used. As expected, the larger the value of `maxev`, the closer to the full computation, but any of the values used provide a quite accurate susceptibility curve. Even for `maxev`=1000 the susceptibility curve only diverges at the end of the range used, and it would be necessary to double the temperature to appreciate the divergence. The plots correspond to a value of `pobla`=10^{-1}, but as mentioned before the graphs are essentially the same for 10^{-2} and 10^{-3}.

4. Acceleration with graphics processors

The inherent parallelism of computer graphics has motivated the development of the GPUs in recent years, enabling the simultaneous use of multiple computational cores. With the CUDA API released to the public, it is possi-

ble to exploit the many-core architecture with the single instruction multiple thread (SIMT) execution model used by NVIDIA GPUs in an affordable way [9]. Parallel computations executed on the GPU can be launched by sequential programs running on the CPU and, depending on the problem, run-time improvements can go up to two orders of magnitude. Although such exceptional performance can only be achieved with specific problems, its relatively easy programming makes it worth giving it a try, as good speedups can be achieved even with non-optimized kernels by simply migrating the functions to run on the GPU. In recent times, we are seeing many authors that are using GPUs in many contexts, including sparse computations similar to the one we are concerned with, see e.g. [10].

Furthermore, more and more computational libraries are beginning to provide functionality to exploit the power of GPUs without requiring major modifications of application code. In its development version, PETSc incorporates support for NVIDIA GPUs by means of Thrust and CUSP[1], performing vector operations and matrix-vector products through `cusp` and `aijcusp`, special vector and matrix classes whose arrays are mirrored in the GPU, and whose data is transparently synchronized between the host and the device as needed to guarantee coherence of the mirrored data-structures. The GPU model considered in PETSc uses MPI for communication between different processes, each of them having access to a single GPU [11]. Since only vector and matrix operations are performed on the GPU, the implementation of solvers in PETSc and SLEPc follows a hybrid CPU-GPU approach, as the algorithms use different calls to CUDA kernels to speed up some parts of the computation, but their main logic remains on the CPU.

In this section, we present a GPU-enabled implementation of ParIso that can operate also with multiple GPUs (one per MPI process). We remark that these new developments are independent of the optimization discussed in section 3, and could also be integrated in the original ParIso code. ParIso can benefit from GPU computation mainly in two ways: when computing matrix coefficients and when solving the eigenproblems with SLEPc. The former is very appropriate for GPU computing, since matrix coefficients can be evaluated independently from each other. Regarding the latter, expected

[1]Thrust is a C++ template library included in the CUDA software development toolkit that makes common CUDA operations concise and readable. CUSP is an open source library based on Thrust that targets sparse linear algebra.

Table 6: Different versions of PARISO with GPU support.

Version	Matrix created on	Eigencomputation on
CPU-`sbaij`	CPU	CPU
CPU-`aijcusp`	CPU	GPU
GPU-`sbaij`	GPU	CPU
GPU-`aijcusp`	GPU	GPU

gains are modest since it corresponds to a sparse linear algebra computation.

Implementing sparse linear algebra operations on GPUs efficiently is difficult, and still a topic of active research. Usually, the most relevant operation is the sparse matrix-vector multiplication, that can be approached in different ways [12, 13]. In our case, we tried all different sparse storage formats available in CUSP and CUSPARSE, and the difference among them is not noticeable in our application, since the percentage of time devoted to this operation is just about 3–4% of the total computation.

4.1. Details of GPU implementation

Even though it is possible to use CUDA from Fortran with a non-free compiler, or by means of calling CUDA-C wrapper functions, in order to make use of the GPU computing power, we have ported the original Fortran code to C, as it enables a simpler development. The code uses CUDA in two ways, one by migrating parts of the application to run on the GPU as CUDA kernels, and another one by means of SLEPc, as it allows us to do the computation exclusively on the CPU or with the help of the GPU. We call 'GPU version' the one that uses CUDA kernels for the generation of the matrix (independently of the storage type used), even when the original CPU version can be instructed to use the GPU by means of the `aijcusp` matrix storage type.

With both versions and the different storage types it is possible to run the software in four main ways: CPU and GPU, each one with a matrix storage type that replicates the data into the GPU (`aijcusp`) or not (`sbaij`). Table 6 summarizes the four combinations. Note that the user can select one of them at run time by means of command-line arguments.

One of the benefits of working with symmetric matrices is that the storage can make use of that symmetry to halve the memory usage, as it is done in

the code that runs exclusively on the CPU by means of the `sbaij` matrix storage type. In the case of the GPU, the type of matrix used by PETSc to store the data (`aijcusp`) does not take into account the symmetry and needs to allocate and fill both triangular parts (upper and lower) of the matrix.

The resulting code can be compiled to work with single and double precision arithmetic, but due to several values exceeding by far the single precision limits, some of the variables have been explicitly declared as double even when working with single precision. This mixed-precision approach allows us to reduce the memory requirements by sacrificing some accuracy in the results. There is also a reduction of computation time, but as will be shown later, the benefits of the single precision arithmetic in terms of performance are quite limited.

The migration has been done by selecting the most computationally demanding functions and moving them to run on the GPU. The selected functions are those related with the generation of the submatrices, and one of the functions that compute the final thermodynamic results. The cost of the generation of the magnetic susceptibility is negligible, but the time needed to compute the magnetization depends on the number of samples of the magnetic field intensity used. Even though this step is not very computationally demanding, as it is done exclusively by a single process, its relative time within the whole computation increases when increasing the number of processes and the total computation time is reduced. That is why porting it to the GPU was highly recommended.

In order to avoid heavy performance issues during the sparse matrices assembly it is necessary that every process preallocates the memory corresponding to the part of the matrix that has been assigned to it [8]. For the preallocation to be done, it is necessary to specify how many diagonal and off-diagonal nonzero elements per row the matrix has (here, diagonal elements refer to matrix entries located in column indices matching the range of rows assigned to the current process, see red blocks on Figure 5). Once the preallocation has been done, the process continues by setting the values of the elements and finally assembling the matrix. The matrix is ready to use once the assembly is finished.

Two functions are in charge of the generation and, in the GPU version, both compute all the elements of the matrix (not only the upper half), so we double the work to be done with respect to the previous CPU implementation. The first one counts the number of nonzero elements on the diagonal blocks and outside them, to do the preallocation, and the second one stores

the values of these nonzero elements in memory. Both functions do almost the same work, but the amount of memory used by them differ significantly, as the counting of the elements needs only a small bi-dimensional array of integers to store the sum (with one row for each one of the matrix rows, and two columns, one for the nonzero diagonal elements and other for the nonzero off-diagonal), and the function that fills the matrix in with its values needs two bi-dimensional arrays of the full size of the matrix, one of floating point numbers and another one of integers where it records the column indices of the nonzero elements. The accounting of the elements is only done on the first pass, and stored to avoid repeating the computation on the second pass.

Each one of these two generation functions have been split in two, the main part runs as a CUDA kernel making use of a series of auxiliary functions that run as `__device__` functions, and the remaining work is done as a host wrapper function that allocates the memory on both CPU and GPU, calls the kernel with the appropriate arguments and copies the results to host memory. The function that computes the final results follows the same scheme of wrapper and kernel split, but it makes use of two kernels that need to be called serially.

The host memory used by the wrapper function is allocated with the `cudaHostAlloc` instruction. The difference between this and a traditional `malloc` is that the `malloc` instruction allocates standard pageable memory, and `cudaHostAlloc` allocates page-locked memory. The CPU implements a virtual memory system that allows programs to use more memory than the available in the system by swapping out unused pages and swapping them in again when needed. A transfer from this virtual memory to the memory of the GPU would imply two copies: a copy to a page-locked intermediate buffer and another copy from this buffer to the GPU memory by means of DMA (Direct Memory Access). The page-locked memory is guaranteed to live in the main physical memory without ever being swapped out, so its use eliminates the intermediate buffer and the copies to/from it, saving time. The copy of the values of the matrix and their accounting from the GPU is done with asynchronous instructions, but as the data is used immediately after it, the transfer can be considered synchronous with no concurrency of these data transfers and arithmetic operations.

Inside the GPU there are several memory types available, with different sizes, latency accesses, scopes and lifetimes. Within the generation functions, application lifetime data is copied once at the beginning of the program to `__constant__` memory, and block dependent data is copied to `__global__`

memory in advance to the submatrix computation (of which a small array is accessed through texture cache).

The amount of memory in the device limits the registers a thread can make use of, and the functions that need a large amount of registers limit the number of concurrent threads running on the device. The matrix generation functions result in computation bound kernels due to the high register use. Their launch can not fully populate the symmetric multiprocessors of the GPU, so they are infra utilized and the performance obtained is far from optimal.

Having said that, the computation of each different matrix element has no dependencies with the others, and this allows us to populate the GPU with any possible distribution of the grid and block dimensions and size. The launch of the kernels takes into account two different things, on one hand, the size of the CUDA grid and blocks (number of blocks and number of threads per block respectively) per dimension, and on the other, the tile size (amount of work, measured in number of matrix elements, that a single thread has to compute) per dimension.

The work distribution scheme within the GPU device is the same for all the kernels. For the generation functions, for each of the dimensions of the matrix, two constants are defined, `BLOCK_SIZE` and `TILE_SIZE`, that are used to obtain the kernel call arguments. The calculus of these arguments begins by setting the number of blocks to one, and the number of threads per block to `BLOCK_SIZE`. Next, it is checked if the number of rows (or columns) is greater than the `BLOCK_SIZE` multiplied by the `TILE_SIZE`. If it is greater, the grid dimension (number of blocks) is increased to

```
dimGrid->x = (rows + ((BLOCK_SIZE_X * TILE_SIZE_X) - 1)) /
                              (BLOCK_SIZE_X * TILE_SIZE_X),
```

if not, the block size is decreased to

```
dimBlock->x = (rows + (TILE_SIZE_X - 1)) / TILE_SIZE_X.
```

In CUDA, all the threads in a block are grouped in warps, being 32 the warp size since the beginning of CUDA. All threads in a warp fetch and execute a single instruction per clock cycle. In the latter case, when the block size is reduced, it ends up not necessarily being a multiple of the warp size, and that means that we are working with a very small matrix so the performance obtained from the use of the GPU is not going to be good.

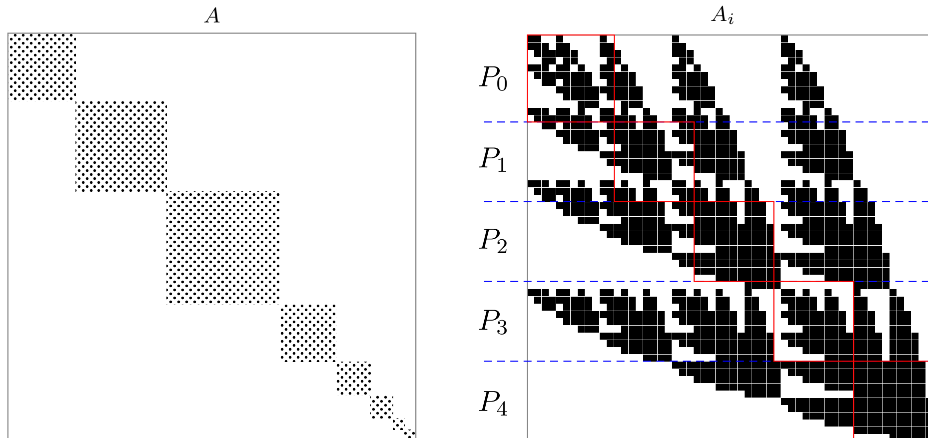

Figure 5: Example of an isotropic system expressed as a block diagonal matrix (left) and the partitioning of one of their symmetric sparse blocks between five MPI processes P_i (right).

Once the grid dimension is set, it is necessary to check that it does not exceed the limits of the device, and in that case, reduce its size to the maximum allowed value, and establish a counter in order to do several calls to the kernel. This is necessary in the case of very large matrices.

The current scheme creates blocks of one thread for the rows axis, and 64 threads for the columns axis, with tile sizes of one. Other work distributions have been tested with no better performance obtained. An analogous procedure is used for the kernels that compute the magnetization.

4.2. Hybrid MPI-GPU approach for multi-GPU support

The partitioning of the work between the different MPI processes done by PETSc uses a block-row distribution. Figure 5 shows an example of how each of the sparse blocks that form the block diagonal matrix are partitioned. The horizontal dashed blue lines delimit the P_i MPI process partition and the red squares show the diagonal block of each process, formed by the columns whose indices correspond to the rows assigned to it.

The execution of the kernels is independent of MPI. Since the MPI version of the code evenly distributes the problem matrices across the processes, several CUDA devices can be used to individually accelerate the execution at each process, provided that the code is run on a cluster with GPUs available in all nodes. If one node has more than one GPU, the processes select

the CUDA device to be used based on their MPI rank, in the same way as PETSc does. This way, each process will use a different CUDA device (when available). To maximize the performance and to avoid overloading the devices, the MPI launch should take into account the problem size, the number of available nodes, and the number of devices per node.

4.3. Performance evaluation

In this section we analyze the performance obtained with the GPU version of the software and compare it with the CPU-only version. For this purpose, several runs have been done in the cluster Minotauro, where each node has two Intel Xeon E5649 processors at 2.53 GHz, 24 GB of main memory, and two NVIDIA Tesla M2090 GPU with 512 cores at 1.3 MHz and 6 GB of GDDR5. The nodes are interconnected with a low latency Infiniband network, and their operating system is RHEL 6.0 with GCC 4.6.1, MKL 11.1 and CUDA 7.0.

The value of the `pobla` and `maxev` parameters used in these runs has been set to 10^{-2} and 1000, respectively, and SLEPc's parameter `mpd` has been set to 50.

Three different cases have been used to evaluate the performance. We have started using the $8Mn^{2+}$ system of subsection 3.2 to have a connection with the executions in the cluster Tirant. The matrix in this case has a size of 135954 and its largest block is of size 16576. The other two systems that we have used to complete the evaluation are $9Mn^{2+}$ and $9Mn^{2+} + 1Cu^{2+}$ with a size of 767394 and 1534788, and with their largest blocks being of size 88900 and 177100, respectively.

The $8Mn^{2+}$ system has been run with the CPU and with the GPU version in single and double precision arithmetic. At the same time, the GPU version has been run with two different matrix storage types: `sbaij` and `aijcusp`. The two largest systems have been run exclusively with the GPU version, with the `aijcusp` and `sbaij` matrix storage types, and with single and double precision arithmetic.

Figure 6 shows the execution times obtained with the $8Mn^{2+}$ system. In the figure we can see how the normal performance of the CPU-only version is improved by the two GPU runs. We can see that for a single process, both GPU runs reduce the time more than one order of magnitude with respect to CPU. For the multi-process runs, the GPU-`sbaij` run shows a similar speedup to the one obtained by the CPU version, maintaining the curves in parallel (up to 32 processes), while the `aijcusp` run does not reduce the

time with the same rate or even increases it while increasing the number of processes. This kind of behaviour, where the performance decreases when the number of processes is increased, is due to the small size of the submatrices. The GPU-`sbaij` run shows clearly the great benefit provided by the kernels that generate the matrix with respect to its CPU counterpart, as both versions use a symmetric aware storage that reduces the memory and arithmetic operations (on the CPU). In the same curve it is possible to appreciate that the speedup is reduced when increasing the number of processes due to the higher cost of the inter-process communications and the infra utilization of the GPU devices due to the reduction of the workload. The GPU-`aijcusp` run is even more affected by the problem size, as the overall performance decreases drastically compared with the `sbaij` versions due to two main drawbacks, it uses twice the memory needed by `sbaij` and it has to synchronize the data between the GPU and CPU besides between the processes, during the computation. As the SLEPc GPU implementation uses vector operations, the performance is directly dependent of the size of the problem. We can appreciate such dependency if we compare the results of the different test cases. While in Figure 6 we see that GPU-`aijcusp` starts with the smallest run times and quickly drops the performance, as we increase the size of the problem (Figures 7 and 8), we can see how it behaves much better and only reduces the performance with 128 processes. It is also noticeable how also the `sbaij` run improves with the increment of the size as its performance does not decrease so quickly when the number of processes is increased.

The figures show that the more the GPU devices are used, the better is the performance obtained. Both GPU runs clearly improve the CPU-`sbaij` run times, maintaining a similar performance and a good scalability and being able to solve a large system of 1.5 million elements in less than 230 seconds, with 128 GPUs and using double precision arithmetic. For such improvement to be possible it is necessary to maximize the use of the GPUs, as we can see how the performance decreases in all GPU runs when the processes do not have enough workload. The work done by the GPU needs to be enough to fully populate and use the device, as the performance depends directly on the usage of the device. The behaviour is the same for the kernel executions and for the eigenvalue computation.

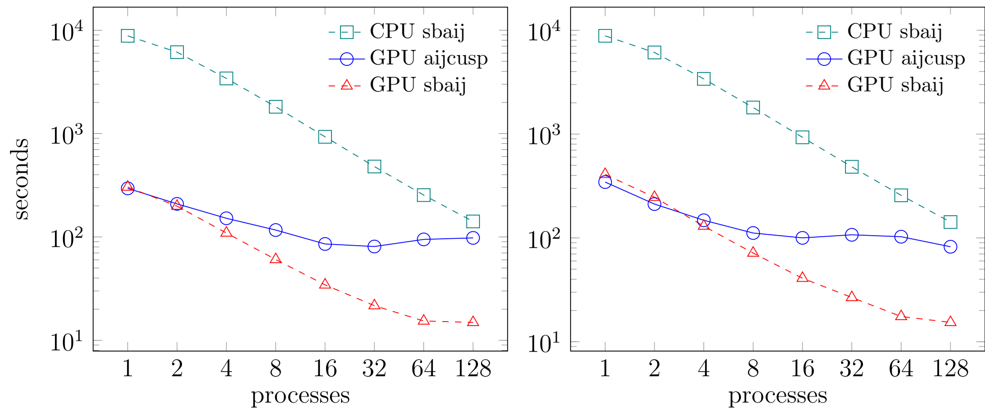

Figure 6: Total problem solve time for the $8Mn^{2+}$ system with single (left) and double precision arithmetic (right).

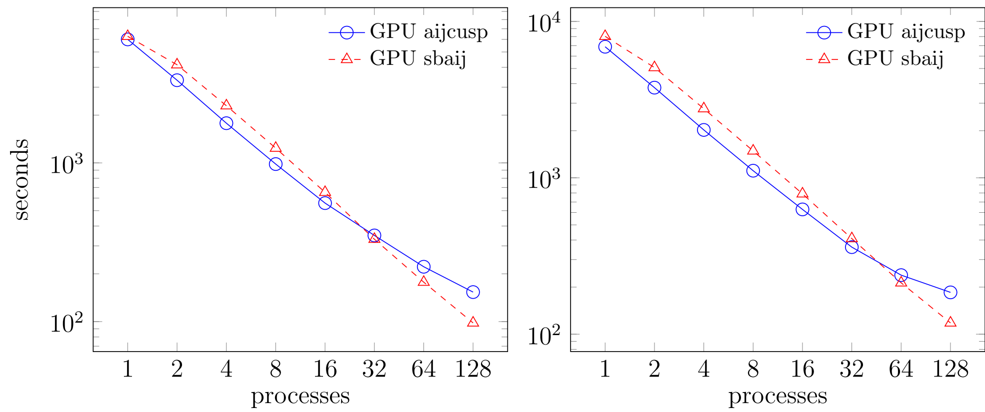

Figure 7: Total problem solve time for the $9Mn^{2+}$ system with single (left) and double precision arithmetic (right).

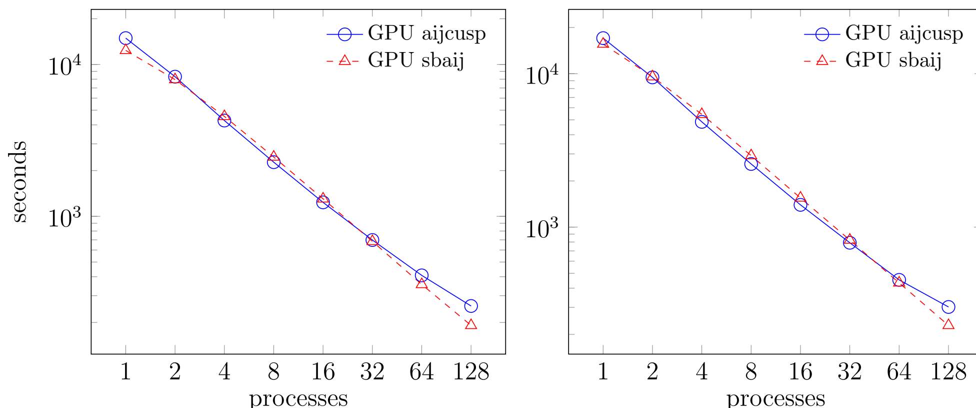

Figure 8: Total problem solve time for the $9Mn^{2+} + 1Cu^{2+}$ system with single (left) and double precision arithmetic (right).

5. Conclusions and future work

In this paper we have presented two main optimizations to ParIso, a program for simulation of isotropic molecular clusters with the ITO computational technique.

The first optimization consists in avoiding the computation that does not contribute significantly to the aggregate results. In our tests, this has allowed a drastic reduction of the execution time without losing validity in the results. However, our heuristics for determining the energetic cut (as well as to estimate the number of eigenvalues required in each block of the Hamiltonian matrix) assumes a uniform distribution of eigenvalues. This assumption is valid only for systems with specific properties, as discussed in section 3, so the method may not be appropriate for general systems. As a future work, we plan to extend the code in such a way that not only the first and last eigenvalue of each block is computed in the first phase, but also a rough approximation of how all eigenvalues are distributed within that range. This information, usually known as density of states (DOS), is difficult to obtain, but recent efforts are trying to do this cost effectively [14].

The second major optimization is the implementation of a GPU-enabled version that can perform either the computation of matrix coefficients or the computation of the partial diagonalizations, or both, on a high-performance graphics processor. The performance gain is very significant, especially associated to the computation of the matrices. Regarding the efficiency of diagonalization on the GPU, this step relies on the efficiency achieved by

the SLEPc library. There is still room for improving this in SLEPc, and our code will automatically benefit from these improvements as they are made available in future versions of the library.

With the multi-GPU version we are able to reduce the computation one order of magnitude with respect to the parallel MPI version running on CPUs. This will make it possible to solve much larger problems, those with real scientific interest, that would otherwise be impossible to address due to memory limitations or lack of computational power.

Acknowledgements. We are indebted to J. M. Clemente-Juan and S. Cardona-Serra, with whom we collaborated to develop the MPI version of ParIso, the starting point of the current work. We also thank them for valuable advise related to the developments of section 3. The simulations corresponding to section 3 were carried out on the supercomputer Tirant at Universitat de València. The simulations corresponding to section 4 were carried out on the supercomputer Minotauro, belonging to the Spanish Supercomputing Network (RES).

References

[1] J. J. Borras-Almenar, J. M. Clemente-Juan, E. Coronado, B. S. Tsukerblat, Inorg. Chem. 38 (1999) 6081–6088.

[2] B. L. Silver, Irreducible Tensor Methods. An Introduction for Chemists, Academic Press, London, 1988.

[3] J. J. Borras-Almenar, J. M. Clemente-Juan, E. Coronado, B. S. Tsukerblat, J. Comput. Chem. 22 (2001) 985–991.

[4] E. Ramos, J. E. Roman, S. Cardona-Serra, J. M. Clemente-Juan, Comput. Phys. Commun. 181 (2010) 1929–1940.

[5] V. Hernandez, J. E. Roman, V. Vidal, ACM Trans. Math. Software 31 (2005) 351–362.

[6] K. Wu, H. Simon, SIAM J. Matrix Anal. Appl. 22 (2000) 602–616.

[7] R. B. Lehoucq, D. C. Sorensen, C. Yang, ARPACK Users' Guide, Solution of Large-Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998.

[8] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, K. Rupp, B. Smith, S. Zampini, H. Zhang, PETSc Users Manual, Technical Report ANL-95/11 - Revision 3.6, Argonne National Laboratory, 2015.

[9] J. Cohen, M. Garland, Comput. Sci. Eng. 11 (2009) 58–63.

[10] W. Rodrigues, A. Pecchia, M. Lopez, M. A. der Maur, A. D. Carlo, Comput. Phys. Commun. 185 (2014) 2510–2518.

[11] V. Minden, B. Smith, M. G. Knepley, in: GPU Solutions to Multi-scale Problems in Science and Engineering, Springer, 2013, pp. 1–9.

[12] R. Li, Y. Saad, J. Supercomput. 63 (2013) 443–466.

[13] I. Reguly, M. Giles, in: Innovative Parallel Computing (InPar), pp. 1–12.

[14] L. Lin, arXiv:1504.07690 : retrieved 18 Nov 2015 (2015).