Lecture Notes in Computer Science, 6782, 630-642, (2011)

A Concurrent Object-Oriented Approach to the Eigenproblem Treatment in Shared Memory Multicore Environments

Alfonso Niño[1] , Camelia Muñoz-Caro[1] and Sebastián Reyes[1],

[1] QCyCAR research group, Escuela Superior de Informática, Universidad de Castilla-La Mancha, Paseo de la Universidad 4, 13071 Ciudad Real, Spain
{Alfonso.Nino, Camelia.Munoz, Sebastian.Reyes}@uclm.es

Abstract. This work presents an object-oriented approach to the concurrent computation of eigenvalues and eigenvectors in real symmetric and Hermitian matrices on present memory shared multicore systems. This can be considered the lower level step in a general framework for dealing with large size eigenproblems, where the matrices are factorized to a small enough size. The results show that the proposed parallelization achieves a good speedup in actual systems with up to four cores. Also, it is observed that the limiting performance factor is the number of threads rather than the size of the matrix. We also find that a reasonable upper limit for a "small" dense matrix to be treated in actual processors is in the interval 10000-30000.

Keywords: Eigenproblem, Parallel Programming, Object-Orientation, Multicore processors

1 Introduction

The eigenproblem plays an important role in both science and engineering. Thus, it appears in problems such as the quantum mechanical treatment of time independent systems [1], in the principal components analysis (PCA) [2], in the specific application of PCA to face recognition (eigenfaces) or in the computation of the eigenvalues of a graph in spectral graph theory applied to complex networks [3]. The eigenproblem implies, in practical terms, the computation of the eigenvalues and eigenvectors of a matrix [4]. Very often the matrix is a symmetric, real, one. However, in the general case, we deal with Hermitian, complex matrices; see for instance [5]. An interesting fact is the quadratic dependence of the matrix with the problem size. This leads easily to large (or very large) matrices, for instance, in the variational treatment of quantum systems with several degrees of freedom. The capability of dealing with large eigenproblems (matrices of size about 10^6) is of great interest for tackling realistic problems in different fields of science and engineering.

The eigenvalue problem is a central topic in numerical linear algebra. The standard approach for the numerical solution of the eigenproblem is to reduce the matrix to some simpler form that yields the eigenvalues and eigenvectors directly [6]. The first method of this kind dates back to 1846 when Jacobi proposed to reduce a real symmetric matrix to diagonal form by a series of plane rotations [7]. From then, the field has experienced a huge development, especially since the availability of the modern computer in the early1950s [6]. A milestone of the field was due to Givens in 1954 [8]. In this work, Givens proposed to use a finite number of orthogonal transformations to reduce a matrix to a form easier to handle, such as a tridiagonal form. In addition, in 1958 Householder showed how to zeroing the elements outside the tridiagonal in a matrix row and column without spoiling any previous similar transformation [9]. The Householder method became the standard reduction method of matrices to tridiagonal form on serial computers [6]. The method is described in detail in any text dealing with the eigenvalue problem, see for instance [10-13]. From the 1960s the way to compute selected eigenvalues and eigenvectors of a tridiagonal matrix involves locating the eigenvalues using a Sturm sequence and obtaining the eigenvectors by inverse iteration [14]. This approach is well presented in the classical Wilkinson's book [10]. On the other hand, for computing the whole set of eigenvalues and eigenvectors the QR technique is more efficient [13]. A different standpoint is represented by the divide and conquer approach initially proposed by Cuppen in 1981 [15]. In this approach, the matrix is reduced to tridiagonal form, splitting this last in two blocks plus a rank-one update. The procedure can be recursively applied until the matrices are small enough. Then, we can treat the resulting blocks by other method such as QR. The "modern", stable implementation of the method was proposed in 1995 by Gu and Eisenstat [16]. The divide and conquer approach is the fastest way to obtain all the eigenvalues and eigenvectors of a symmetric matrix [6]. Besides, the method is well suited for parallel implementation.

Different available software packages allow treating the eigenvalue problem. Most of them implement descendants of the algorithms presented in the classical Wilkinson and Reinsch book [17]. In particular, many of these algorithms were codified in Fortran, in the 1970s, in the LINPACK (for numerical linear algebra) and EISPACK (for eigenproblems) packages [18, 19]. LINPACK and EISPACK give rise to LAPACK (also in Fortran) in the 1990s [20]. The last descendant in this family is ScaLAPACK, which provides a parallel implementation of a subset of LAPACK using a distributed memory parallel programming approach [21]. In this context, it is interesting to mention ARPACK (ARnoldi PACKage), which is a FORTRAN 77 numerical software library for solving large scale eigenvalue problems [22]. ARPACK is designed to compute a few eigenvalues, and its corresponding eigenvectors, of a general n by n matrix A. It is especially well suited for large sparse or structured matrices. The package is based on an algorithmic variant of the Arnoldi process called the Implicitly Restarted Arnoldi Method [23]. A parallel ARPACK (PARPACK) is available for distributed memory architectures [24]. Another package worth mentioning is SLEPc [25]. SLEPc (Scalable Library for Eigenvalue Problem Computations) is a software library for the treatment of large sparse eigenproblems on parallel computers with a distributed memory architecture.

These packages have been essentially implemented, or are intended to be used, under a traditional imperative programming model. However, to ease the modeling of complex application problems, it would be interesting to use an object-oriented approximation. In this context, we have proposed recently an object-oriented approach for the uniform treatment of eigenproblems in real symmetric and Hermitian matrices [26]. This approach has shown to yield speedups up to three over the standard LAPACK routines in small matrices (i.e. for maximum sizes of 10^4) [26]. On the other hand, it is clear that to deal with larger problems (say for matrices of size 10^5-10^6), we must resort to parallel programming.

When considering the parallel computing landscape over the last few years, we find an interesting evolution. Microprocessors based on a single processing unit (CPU) drove performance increases and cost reductions in computer applications for over two decades. However, this process reached a limit point around 2003 due to heat dissipation and energy consumption issues [27]. These problems limit the increase of CPU clock frequency and the number of tasks that can be performed within each clock period. The solution adopted by processor developers was to switch to a model where the microprocessor had multiple processing units known as cores [28]. Nowadays, we can speak of two approaches [28]. The first, multicore approach, integrates a few cores (at present between two and eight) into a single microprocessor, seeking to keep the execution speed of sequential programs. Actual laptops and desktops incorporate this kind of processors. The second, many-core approach, uses a large amount of cores (at present as many as several hundred) and are specially oriented to the execution throughput of parallel programs. This approach is exemplified by the Graphical Processing Units (GPUs) available today. Thus, parallel capabilities are available in the commodity machines we find everywhere. Clearly, this change of paradigm has had (and will have) a huge impact on the software developing community [29].

Traditionally, parallel systems, or architectures, fall into two broad categories: shared memory and distributed memory [30]. In shared memory architectures we have a single memory address space accessible to all the processors. Shared memory machines have existed for a long time in the servers and high-end workstations segment. On the other hand, in distributed memory architectures there is not global address space, but each processor owns its own memory space. This is a popular architectural model encountered in networked or distributed environments such as clusters or Grids of computers. As a consequence of the popularity of computer clusters in the past years, today’s most used parallel approach in scientific computing is message passing. However, it is interesting to realize that, at present, we can exploit the shared memory nature of multicore microprocessors in the individual computer nodes of any cluster.

The conventional parallel programming practice involves a pure shared memory model [30], usually using the OpenMP API [31], in shared memory architectures, or a pure message passing model [30], using the MPI API [32], on distributed memory systems. Accordingly to this hybrid architecture, different parallel programming models can be mixed in what is called hybrid parallel programming. A wise implementation of hybrid parallel programs can generate massive speedups in the otherwise pure MPI or pure OpenMP implementations [31]. The same can be applied to hybrid programming involving GPUs and distributed architectures [32, 33].

As a needed step in the treatment of large eigenproblems, we present in this work an extension of our previous object-oriented approach to the parallel treatment of eigenproblems on the memory shared architecture of present multicore systems. Our aim is to obtain a solution with acceptable scaling for a few (say less than 10) concurrent execution threads. As an initial case, we apply the proposed solution to the computation of eigenvalues and eigenvectors for real symmetric matrices.

2 Proposed Object-Oriented Approach to the Eigenproblem

As commented above, it is of interest treating the eigenproblem in large matrices. For that, we can resort to some variant of the divide and conquer approach [15, 16]. In this form, we can factorize recursively the large matrix in smaller ones until the resulting matrices are small enough to be solved individually. This process can be implemented concurrently in a distributed memory system such as a computer cluster. However, the treatment of the small matrices is done on individual cluster nodes. Here, we can use a shared memory parallel model to take advantage of the multicore architecture of the nodes. In addition, this use of parallelism allows for increasing the size these "small" matrices can have. Therefore, the number of times the divide and conquer process must be applied can be reduced. The result would be a reduction of the overall computational effort.

To parallelize the computation of eigenvalues and eigenvectors on a shared memory system, we propose an extension of the object-oriented approach previously developed [26]. The corresponding UML class diagram is shown in Fig. 1.

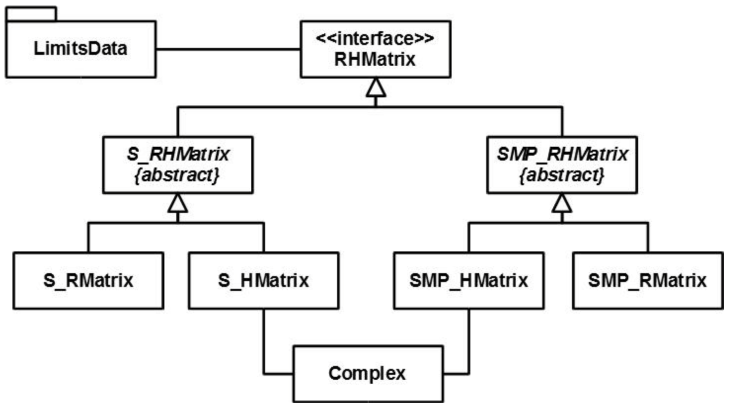

Fig. 1. Class diagram for the sequential and concurrent proposed treatment of real and Hermitian eigenproblems in multicore systems.

Fig. 1 shows at the top of the diagram an interface defining (but not implementing) the functional behavior of every operative class. The system is organized in two branches corresponding to the Sequential (S_) and Shared Memory Parallel (SMP_) cases. At the lower end of the diagram we have the specialized classes that deal with the real (R) and Hermitian (H) cases. The use of the inheritance relationship along the diagram allows for the use of polymorphic references of the base class (RHMatrix) for referring to objects of any concrete class at the bottom. In this way, the same code can be used to invoke the processing of real or Hermitian problems either sequentially or concurrently. The diagram also shows that the RHMatrix interface class exhibits an association relationship with a package (LimitsData). This is used to define the different numerical limits needed in the algorithms. Finally, the classes dealing with

Hermitian matrices have an association relationship with a “Complex” class, which is used to represent the behavior of complex numbers.

3 Sequential Algorithm

To compute eigenvalues and eigenvectors of real symmetric and Hermitian matrices, we use the classical procedure described in the introduction [14]. First, we reduce the matrix to tridiagonal form using a series of Householder reflections. Here, we introduce the method of Shukuzawa et al. [36] to transform the Hermitian matrices to real tridiagonal matrices by using modified Houselholder reflections [26]. In this form, real symmetric as well as Hermitian matrices yield a real tridiagonal matrix. Second, we compute the desired eigenvalues of the real tridiagonal matrix using a Sturm sequence and the bisection method [10-13]. Third, we compute the eigenvectors for each previous eigenvalue using inverse iteration [10-13]. The pseudocode for the algorithm used is shown in Algorithm 1. Here, too low level details are not shown for the sake of clarity. For specific details consult [26].

```
Begin_algorithm
  // Tridiagonalization of the n x n A matrix
  for i←0 to n-2 (real symmetric matrix) or n-1 (Hermitian matrix)
    Compute $s = \left( \sum_{j=i+1}^{n-1} |a_{j,i}|^2 \right)^{1/2}$
    Compute vector x with $x_j=0, \forall j \le i$; $x_i=a_{i,\,i+1} \pm s$; $x_j=a_{i,j}, \forall j > i$
    Compute vector $\mathbf{u}=\mathbf{x} / |\mathbf{x}|$
    Compute $\beta=2$ (real case) or
          $\beta=1+(s+a^*_{i,\,i+1})/(s+a^*_{i,\,i+1})$ (Hermitian case)
    Compute vector $\mathbf{p}=\beta^+ \cdot \mathbf{A} \cdot \mathbf{u}$
    Compute $K=2 \cdot \mathbf{u}^T \cdot \mathbf{A} \cdot \mathbf{u}$ (real case) or
          $K=[1-real(\kappa)] \cdot \mathbf{u}^+ \cdot \mathbf{A} \cdot \mathbf{u}$ (Hermitian case)
    Compute vector $\mathbf{q}=\mathbf{p}-K \cdot \mathbf{u}$
    Update matrix $\mathbf{A}=\mathbf{A} - \mathbf{q} \cdot \mathbf{u}^+ - \mathbf{u} \cdot \mathbf{q}^+$
  end_for

  // Computing m of the n possible eigenvalues ($m \le n$)
  Obtain whole eigenvalues interval using Gershgorin theorem
  for i←m-1 to 0
    while error in eigenvalue $e_i$ larger than a given limit
      Bracket eigenvalue $e_i$ using bisection and a Sturm sequence
    end_while
    Make last non-degenerate $e_i$ the new upper limit of the eigenvalues interval
  end_for

  //Computing the m eigenvectors associated to the eigenvalues (optional)
  for i←0 to m-1
    Generate guess vector $\mathbf{v}_i$ randomly
    Perform LU decomposition of tridiagonal matrix
```

```
      while error in eigenvector $\mathbf{v}_i$ larger than a given limit
        Get $\mathbf{v}_i$ using inverse iteration
      end_while
      if i>0 and $\mathbf{v}_i$ is degenerate then
        $j \leftarrow i-1$
         while $e_j = e_i$
           Orthonormalize $e_i$ respect to $e_j$
          $j \leftarrow j-1$
        end_while
      end_if
      Rotate vector back to the original A matrix  $\mathbf{v}_i = \mathbf{v}_i - \beta_j \cdot \mathbf{u}_j^+ \cdot [\mathbf{u}_j \cdot \mathbf{v}_i]$
   end_for
End_algorithm
```

Algorithm 1. Sequential algorithm for the computation of eigenvalues and eigenvectors of real symmetric and Hermitian matrices.

Taking into account the different vector-vector and matrix-vector products, the pseudocode above shows that the tridiagonalization has $O(n^3)$ complexity. In addition, computation of the m eigenvalues exhibits $O(m \cdot n)$ complexity. Finally, the eigenvectors computation is $O(m \cdot n^2)$. The question now is how to use concurrency to lower the complexity of each of these processes.

4 Concurrent Algorithm

Different data oriented approaches have been proposed for parallelizing the Householder tridiagonalization. However, these approaches rely either in a distributed memory model [37, 38] or in the availability of a perfect square number of processors [39]. On shared memory systems it is interesting to mention the work of Honecker and Schüle [40]. These authors present an OpenMP version of the Householder algorithm for Hermitian matrices. However, they do not consider the subsequent problem of parallelizing the computation of eigenvalues and eigenvectors.

In the present work, we use a memory shared approach considering a limited number of cores. Here, Algorithm 1 is used as a basis for the concurrent computation of eigenvalues and eigenvectors. The algorithm shows that a task based parallelizing approach offers little room for improvement, since very few different tasks can be performed concurrently. Therefore we resort to a data (domain) decomposition to achieve a scalable solution. In our case the whole process is based in matrix and vector manipulations. Thus, a data decomposition approach is obtained by considering independent rows or columns. That depends on the most appropriate strategy for profiting from cache data locality in the used language. In practice, this is achieved by performing concurrently the iterations of the different loops presented in Algorithm 1.

Analyzing Algorithm 1 several facts are clear. First, in the tridiagonalization step, we have an update of the **A** matrix within the outer loop. Therefore, the different iterations of the outer loop are coupled together. So, they cannot be performed concurrently without incurring in a data race condition. Second, the computation of eigenvalues does not involve mixing eigenvalues. Thus, an independent eigenvalues computation is possible. For P concurrent execution threads, the complexity of this step would be reduce to *O(m·n/P)*. Finally, calculation of each eigenvector is independent of the others, except if degeneracy is present. However, placing the degenerate eigenvectors orthonormalization outside the main loop solves the problem. Taking into account that in the general case the amount of degeneracy is small, if any, the complexity would be reduced to $O(m\cdot n^2/P)$.

Algorithm 2 shows that the main source of imbalance in the concurrent algorithm is due to the tridiagonalization part. Here, the overload due to the opening and closing of the concurrent threads is within the main loop. It is also the sequential, single thread, computation of the β factor (and some steps of the K factor), which introduces a sequential limit on the grounds of Amdahl´s law. Therefore, we can expect a decrease of performance with the size of the matrix and the number of concurrent threads.

With the previous considerations a concurrent algorithm can be devised as the one shown in Algorithm 2.

```
Begin_algorithm
  // Tridiagonalization of the n x n A matrix
  for i←0 to n-2 (real symmetric matrix) or n-1 (Hermitian matrix)
    open P concurrent threads
      do among the P threads
         Compute $s = \left(\sum_{j=i+1}^{n-1} |a_{j,i}|^2\right)^{1/2}$
         Compute vector x with $x_j=0, \forall j \le i;\ x_i=a_{i,i+1}\pm s;\ x_j=a_{i,j}, \forall j > i$
         Compute  vector $\mathbf{u}=\mathbf{x} / |\mathbf{x}|$
      end_do_threads
      do single thread
         Compute      $\beta=2$ (real case) or
                      $\beta=1+(s+a^*_{i,i+1})/(s+a^*_{i,i+1})$ (Hermitian case)
      end_single_thread
      do among the P threads
         Compute vector $\mathbf{p}=\beta^+\cdot\mathbf{A}\cdot\mathbf{u}$
      end_do_threads
      do among the P threads
         Compute      $K=2\cdot\mathbf{u}^T\cdot\mathbf{A}\cdot\mathbf{u}$ (real case) or
                      $K=[1-\mathrm{real}(\kappa)]\cdot\mathbf{u}^+\cdot\mathbf{A}\cdot\mathbf{u}$ (Hermitian case)
      end_do_threads
      do among the P threads
         Compute vector $\mathbf{q}=\mathbf{p}-K\cdot\mathbf{u}$
      end_do_threads
      do among the P threads
         Update matrix $\mathbf{A}=\mathbf{A}-\mathbf{q}\cdot\mathbf{u}^+-\mathbf{u}\cdot\mathbf{q}^+$
      end_do_threads
    close concurrent threads
  end_for
  // Computing m of the n possible eigenvalues (m ≤ n)
  Obtain whole eigenvalues interval using Gershgorin theorem
  open P concurrent threads
    do among the P threads
      for i←m-1 to 0
        while error in eigenvalue $e_i$ larger than a given limit
          Bracket eigenvalue $e_i$ using bisection and a Sturm sequence
        end_while
        Make last non-degenerate $e_i$ the new upper limit of the eigenvalues interval
      end_for
    end_do_threads
 close concurrent threads


 //Computing the m eigenvectors associated to the eigenvalues (optional)
  open P concurrent threads
    do among the P threads
      for i←0 to m-1
        Generate guess vector $\mathbf{v}_i$ randomly
        Perform LU decomposition of tridiagonal matrix
        while error in eigenvector $\mathbf{v}_i$ larger than a given limit
          Get $\mathbf{v}_i$ using inverse iteration
        end_while
        Rotate vector back to the original A matrix  $\mathbf{v}_i = \mathbf{v}_i - \beta_j \cdot \mathbf{u}_j^+ \cdot [\mathbf{u}_j \cdot \mathbf{v}_i]$
      end_for
   end_do_threads
 close concurrent threads

  if  i>0 and $\mathbf{v}_i$ is degenerate then
    j←i-1
   while $e_j = e_i$
      open P concurrent threads
        do among the P threads
          Orthonormalize $e_i$ respect to $e_j$
           j←j-1
         end_do_threads
      close concurrent threads
    end_while
  end_if
End_algorithm
```

Algorithm 2. Concurrent algorithm for the computation of eigenvalues and eigenvectors of real symmetric and Hermitian matrices.

There are two implicit considerations not shown in Algorithm 2. The first is how to balance the workload among the different concurrent threads. The second is how to profit from the cache. These questions are implementation related and are considered in the next section.

5 Results and Discussion

As test case, the classes in Figure 1 and the Algorithm 2 are implemented for real symmetric matrices in C++ using OpenMP for shared memory parallelization. To account for a good workload balancing, the different loop indexes are associated to the concurrent threads using the guided self-scheduling algorithm [41] available in OpenMP. For the class Complex, see Figure 1, we use the C++ standard complex class. In addition, we profit from the symmetry in the matrix to store it in packed upper triangular form. To take advantage of the cache, we use row-major order. Compilation is carried out under the Linux operating system using the Solaris CC compiler with the –O5 compiling option, *i.e.*, the higher level of code optimization. The compiler version 5.11 is used. Performance of the implementation is tested considering matrix sizes of 5000 to 10000 in increments of 1000. The matrices are dense matrices, filled with real numbers generated randomly in the [0, 99] interval using the C language `rand()` function. The system used is a Quad-Core AMD Opteron™ 2376 HE with 4x512 KB and 6MB of L2 and L3 cache, respectively, and 8 GB of main memory. Assuming we use 64 bits for the real data type, the 512KB of L2 cache are enough to store rows up to 40000 elements. As shown later, this cache size is larger than the practical limit for a matrix to be considered “small”. Therefore, for the matrices the present approach is intended for, there is no need to block the rows in smaller pieces in order to fit the cache. Finally, to use the worst case, we consider computation of 100% of the eigenvalues and eigenvectors in the series of matrices.

First of all, we compare the sequential results with the one-thread results using the concurrent algorithm. We find that the concurrent version is slightly faster by just a 0.7% in the worst case (matrix with size 7000 and sequential computation time of 1512 seconds). Therefore, the concurrent version can be considered as efficient as the sequential one.

The next question to answer is the performance of the concurrent version as a function of the matrix size and the number of cores. Thus, we compute the speedup for each matrix as a function of the number of cores. For the different matrix sizes, the speedup is defined as the quotient of the one core case with respect to each computing time. The result is shown in Figure 2. We observe that, for a given number of cores, the speedup is fairly independent of the matrix size. The computing time, using the most consuming case, n=10000, ranges from 4425 seconds (for P=1) to 1304 seconds (for P=4). The speedup reaches almost 3.5 in all the P=4 cases.

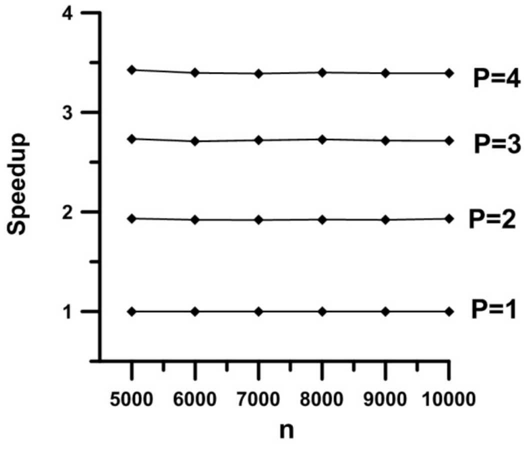

Fig. 2. Speedup for the computation of the whole set of eigenvalues and eigenvectors in real symmetric matrices as a function of the matrix size (n) and the number of cores (P).

The variation rate of the speedup with the number of cores, P, is shown in Figure 3 for the worst case, n=10000. By fitting the speedup to P, we observe a quadratic variation with a good correlation coefficient, R=0.999. As shown in Figure 3, the variation is suboptimal. The maximum, obtained by extrapolating the regression curve, is predicted to be a speedup of 5 for P=9 processors. Anyway, the present results suggest that the limiting performance factor is the number of concurrent threads rather than the size of the matrix.

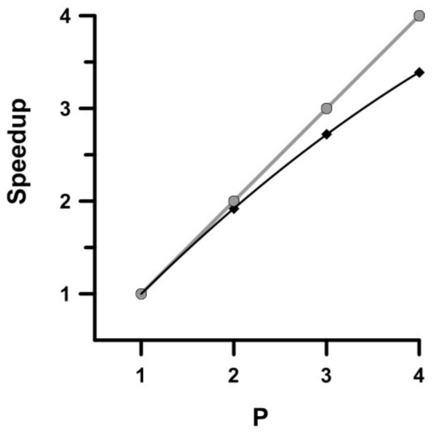

Fig. 3. Variation of the Speedup for the computation of the whole set of eigenvalues and eigenvectors in real symmetric matrices of n=10000 as a function of the number of cores (P). The continuous grey line represents the ideal scaling case. The diamonds represent the experimental data for a matrix size n=10000. The black line corresponds to the quadratic fitting of the data.

Since in the worst case, n=10000, we compute with P=4 the whole set of eigenvalues and eigenvectors in 1304 seconds, the question arises as what is the largest size we can treat in a reasonable time. In other words, what a "small" eigenproblem means in actual architectures. Thus, we have extended the computation from n=10000 to n=40000 in 5000 steps. The results are shown in Figure 4.

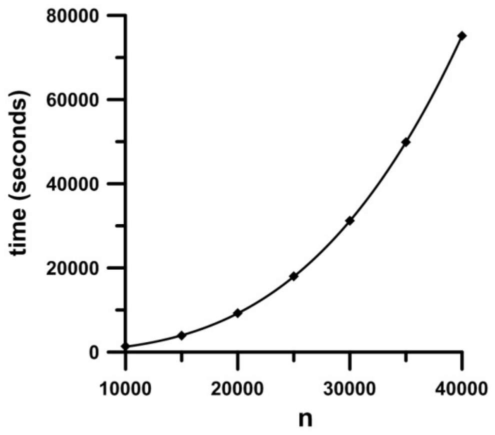

Fig. 4. Computation time, in seconds, as a function of the matrix size, n, for the P=4 case. The diamonds represent the experimental data. The black line corresponds to the cubic fitting of the data.

We observe that the worst case, n=40000, can be handled in less than 21 hours. Fitting the data to a third degree polynomial the correlation coefficient is very good, R=1.000, as expected from the algorithmic complexity. In addition, this uniform behavior shows that no degradation of the cache use does exist. So, what is the limit for a matrix to be considered small? Actually, Figure 4 shows that even the largest size here considered is affordable in a reasonable time. However, limiting ourselves to sizes processed in less than half a day the limit is about n=30000, which is processed in 8.7 hours.

Acknowledgments. This work has been co-financed by FEDER funds and the Consejería de Educación y Ciencia de la Junta de Comunidades de Castilla-La Mancha (grant # PBI08-0008). The Ministerio de Educación y Ciencia (grant # AYA 2008-00446) is also acknowledged.

References

1. Levine, I. N.: Quantum Chemistry, 5th edition. Prentice-Hall (1999)
2. Johnson, R. A., Wichern, D. W.: Applied Multivariate Statistical Analysis. Fifth Edition. Prentice Hall (2002)
3. Mieghem, P. N.: Graph Spectra for Complex Networks. Cambridge University Press (2011)

4. G.H. Golub, C.F. van Loan: Matrix Computations, Third Edition. Johns Hopkins University Press (1996)
5. Castro M. E., Niño A., Muñoz-Caro C.: Evaluation and Optimal Computation of Angular Momentum Matrix Elements: An Information Theory Approach. WSEAS Trans. on Inf. Sci. and Appl. 7:2, 263-272 (2010)
6. Golub G. H., van der Vorst H. A.: Eigenvalue Computation in the 20th Century. J. Comput. and Appl. Math. 123, 35-65 (2000)
7. Jacobi C. G. J.: Ueber ein Leichtes Verfahren, die in der Theorie der Säcularstörungen Vorkommenden GleichungenNnumerisch Auflösen. J. Reine Angew. Math. 30, 51-94 (1846)
8. W. Givens, Numerical Computation of the Characteristic Values of a Real Symmetric Matrix, Oak Ridge Report Number ORNL 1574 (physics) (1954)
9. Householder A.S.: Unitary Triangularization of a Nonsymmetric Matrix. J. ACM. 5, 339-342 (1958)
10. Wilkinson J. H.: The Algebraic Eigenvalue Problem. Clarendon Press, Oxford (1965)
11. Parlett B. N.: The symmetric Eigenvalue Problem. SIAM, Philadelphia (1998). Republication of the original work published by Prentice-Hall (1980)
12. Golub G. H., van Loan C. F.: Matrix Computations, Third Edition. Johns Hopkins University Press (1996)
13. Press W. H., Flannery B. P., Teukolsky S. A., Vetterling W. T.: Numerical Recipes. The Art of Scientific Computing. Cambridge University Press, Cambridge (2007)
14. Ortega J. M.: Mathematics for Digital Computers, Vol. 2, Ed. by Ralston and Wilf. John Wiley & Sons, p. 94 (1967)
15. Cuppen J. J. M.: A Divide and Conquer Method for the Symmetric Tridiagonal Eigenproblem. Numer. Math. 36, 177-195 (1981)
16. Gu M., Eisenstat S. C.: A Divide-and-Conquer Algorithm for the Symmetric Tridiagonal Eigenproblem. SIAM J. Matrix Anal. Appl. 16, 172-191 (1995)
17. Wilkinson J. H., Reinsch C.: Handbook for Automatic Computation, Vol. 2, Linear Algebra. Springer-Verlag (1971)
18. Dongarra J. J., Moler C. B., Bunch J. R., Stewart G.W.: LINPACK Users' Guide (1979); LINPACK, http://www.netlib.org/lapack/ Last access December, 2010
19. EISPACK, http://www.netlib.org/eispack/, Last access December, 2010
20. Anderson E., Bai Z., Bischof C., Blackford S., Demmel J., Dongarra J. J., Du Croz J., Greenbaum A., Hammarling S., McKenney A., Sorensen D.: LAPACK Users' Guide, Third Edition. SIAM (1999); LAPACK: http://www.netlib.org/lapack/, Last access July 22, 2010
21. Blackford L. S., Choi J., Cleary A., D'Azevedo E., Demmel J., Dhillon I., Dongarra J. J., Hammarling S., Henry G., Petitet A., Stanley K., Walker D., Whaley R. C.: ScaLAPACK Users' Guide. SIAM (1997); ScaLAPACK, http://www.netlib.org/scalapack/, Last access December, 2010
22. ARPACK, http://www.caam.rice.edu/software/ARPACK/, Last access December 2010
23. Lehoucq R. B., Sorensen D. C.: Deflation Techniques for an Implicitly Restarted Arnoldi Iteration, SIAM. J. Matrix Anal. & Appl., 17, 789-821 (1996)
24. PARPACK, http://www.caam.rice.edu/~kristyn/parpack_home.html, Last access December 2010
25. SLEPc, http://www.grycap.upv.es/slepc/, Last access December 2010
26. Castro, M. E., Díaz, J, Muñoz-Caro, C., Niño A.: A Uniform Object-Oriented Solution to the Eigenvalue Problem for Real Symmetric and Hermitian Matrices. Comput. Phys. Comun. In press. doi:10.1016/j.cpc.2010.11.022
27. Kirk D., Hwu W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers (2010)
28. Hwu W., Keutzer K., Mattson, T. G.: The Concurrency Challenge. IEEE Design and Test of Computers, 25, 312-320, (2008)

29. Sutter H., Larus J.: Software and the Concurrency Revolution. ACM Queue, 3, 54-62 (2005)
30. Sottile M. J., Mattson T. G., Rasmussen C. E.: Introduction to Concurrency in Programming Languages, CRC Press (2010)
31. OpenMP API Specification for Parallel Programming, http://openmp.org. Last access December 2010.
32. Gropp W., Huss-Lederman S., Lumsdaine A., Lusk E., Nitzberg B., Saphir W., Snir M.: MPI: The Complete Reference, 2nd Edition, Volume 2 - The MPI-2 Extensions. The MIT Press, (1998)
33. Kedia K.: Hybrid Programming with OpenMP and MPI, Technical Report 18.337J, Massachusetts Institute of Technology (2009)
34. Jacobsen D. A., Thibaulty J. C., Senocak I.: An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. In: 48th AIAA Aerospace Sciences Meeting and Exhibit, Florida (2010)
35. Jang H., Park A., Jung K.: Neural Network Implementation using CUDA and OpenMP. In: Proc. of the 2008 Digital Image Computing: Techniques and Applications, pp. 155-161. Canberra (2008)
36. Shukuzawa, O., Suzuki, T., Yokota, I.: Real tridiagonalization of Hermitian matrices by modified Householder transformation. Proc. Japan. Acad. Ser. A, 72, 102-103 (1996)
37. Bischof, C., Marques, M., Sun, X.: Parallel Bandreduction and Tridiagonalization. Proceedings. In: Proc. Sixth SIAM Conference on Parallel Processing for Scientific Computing, pp. 383-390. SIAM, Philadelphia (1993)
38. Smith, C., Hendrickson, B., Jessup, E.: A Parallel Algorithm for Householder Tridiagonalization. In: Proc. 5th SIAM Conf. Appl. Lin. Alg. SIAM, Philadelphia (1994)
39. Chang, H. Y., Utku S., Salama M., Rapp D.: A Parallel Householder Tridiagonalization Stratagem Using Scattered Square Decomposition. Parallel Computing, 6, 297-311 (1988)
40. Honecker, A., Schüle, J.: OpenMP Implementation of the Householder Reduction for Large Complex Hermitian Eigenvalue Problems. C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.) Parallel Computing: Architectures, Algorithms and Applications. John von Neumann Institute for Computing, NIC Series, vol. 38, pp. 271-278 (2007)
41. Polychronopoulos, C. D., Kuck, D.: Guided Self-Scheduling: a Practical Scheduling Scheme for Parallel Supercomputers. IEEE Trans. on Computers, 36, 1425-1439 (1987)