



Improved parallel performance of SIESTA for the HPCx Phase2 system

Joachim Hein
HPCx terascaling team
EPCC
The University of Edinburgh
Mayfield Rd
Edinburgh EH9 3JZ
Scotland, UK

September 1, 2004

Abstract

We investigate the parallel performance of the SIESTA code on the HPCx system. The diagonaliser is the key bottle neck inside the code, which prevents it from scaling to a large number of processors. Changing the processor grid used for the diagonaliser from one to two dimensions, leads to a dramatic improvement in parallel scalability. Given a large enough problem size, the efficient use of SIESTA on the HPCx Phase2 system is now possible. Practical recommendations for running SIESTA on HPCx are included.

This is a Technical Report from the HPCx Consortium

© HPCx UoE Ltd 2004

Neither HPCx UoE Ltd nor its members separately accept any responsibility for loss or damage from the use of information contained in any of their reports or in any communication about their tests or investigations.

1 Introduction

SIESTA is a materials code which uses self-consistent density functional theory. The code is designed for simulations with several thousand atoms. For this reason, the development team choose algorithms which scale linearly with the number of atoms [1]. To obtain acceptable run-times for large systems, good parallel scalability to a large number of processors on a parallel machine is crucial. In this report we investigate the parallel performance of SIESTA 1.3 and demonstrate that changing the processor grid used for the diagonaliser from 1-dimension to 2-dimensions leads to a dramatic improvement of the parallel scalability.

This improvement is very timely for the HPCx service. Following the recent hardware upgrades, the machine is operated as a cluster of 32-way shared memory processors (SMP). For an application to make efficient use of the resource it is therefore required to scale reasonably well to at least 32 processors. Our benchmarking results indicate that given a large enough problem, the proposed improvements make it possible to obtain a good parallel speed-up when using several HPCx SMP-nodes for a single calculation.

This paper begins with a short review of the present HPCx hardware in section 2 and a short review of the SIESTA code in section 3. Section 4 describes how the performance of the code is measured in this report. The input files used for the benchmarking are described in section 5. The performance of the ScaLAPACK routines, used by SIESTA's diagonaliser depends on the block size used for the data distribution. This is discussed in section 6. The parallel performance of SIESTA is discussed in section 7. Section 8 gives the results from a more detailed profiling of individual code segments, aimed towards future improvements. In the later stages of this project, the micro-code used to operate the interconnect between the SMP-nodes was upgraded. In section 9 we demonstrate that SIESTA benefits greatly from this upgrade. Practical hints on how to run SIESTA on HPCx are given in section 10. A summary of the work is in section 11.

2 The HPCx Phase2 system

The HPCx service is funded by the British Government, through the Engineering and Physical Sciences Research Council (EPSRC). The project is run by the HPCx Consortium, a consortium led by the University of Edinburgh (through Edinburgh Parallel Computing Centre (EPCC)), with the Central Laboratory for the Research Councils in Daresbury (CCLRC) and IBM as partners. The HPCx Phase2 system entered user service in late May 2004. The HPCx Phase2 consists of fifty IBM p690+ compute frames plus two IBM p690 service frames. The entire system offers 1600 CPUs for the compute, delivering 6.188 Tflop/s on Linpack.

Each IBM p690+ offers 32 IBM Power4+ processors with a clock of 1.7 GHz and 32 GB of main memory. The frames are connected with a network of IBM's High Performance Switch, also known as "Federation". There are two network adapters per frame, each adapter providing two links, resulting in a total of 4 links per frame.

The system is operated as a cluster of 32-way SMP nodes. This is one of the most significant changes between HPCx Phase1 and HPCx Phase2. The previous Phase1 system was partitioned into logical partitions (LPARs) of 8-way SMP nodes. Users are granted exclusive access to the nodes – each node is used by a single application. To make effective use of the HPCx Phase2 system it is therefore crucial that applications perform well on at least 32 CPUs. In this context it should be noted that the HPCx service is devoted to "*capability computing*" — applications which scale to several hundreds or even one thousand processors.

For this investigation we used version 5.2 of IBM's AIX operating system, version 4.1 of IBM's parallel environment and version 8.1.1 of IBM's xlf Fortran compiler. Unless stated, we used the com-

piler flags “-q64 -qfixed -O3 -qarch=pwr4 -qtune=pwr4 -qstrict” when building the executable with the `mpx1f_r` script. The system is currently set up to use small memory pages of 4 kB each.

During the later stages of this project, the microcode on the HPS switch got upgraded¹. This upgrade reduced the zero message-size MPI-latency from about 10 μ sec to about 6 μ sec and the bandwidth achievable between two frames increased from about 4.5 GB/s to over 8 GB/s. Most of the timings reported in this study were measured before the upgrade. However we noted that SIESTA’s performance benefited from this upgrade. A selection of revised timings are given in section 9.

3 The SIESTA code

SIESTA (Spanish Initiative for Electronic Simulations with Thousands of Atoms) is a materials code, using self-consistent density functional theory (DFT) for the calculation of the electronic structure [1, 2]. In addition to this, it offers options to modify the nuclear variables, such as molecular dynamics simulations, optimisation and phonon calculations. It uses a linear combination of atomic orbitals (LCAO) as basis set. If the direct diagonaliser is not used, the algorithms are designed to scale linearly with the number of atoms. The diagonaliser scales with the third power of the number of atoms. For this investigation we used SIESTA version 1.3.

To enable the code to be used on parallel machines such as HPCx, the code has been parallelised using MPI. The code offers several diagonalisers. The benchmark system used for this investigation invokes the subroutine `rdiag`. `rdiag` invokes the ScaLAPACK expert driver `pdsygvx` [3] to determine the eigenvalues and eigenvectors of the Hamiltonian. ScaLAPACK is built on top of the Basic Linear Algebra Communication Subprograms (BLACS). The BLACS used on HPCx is built on top of MPI.

In SIESTA 1.3 the matrices are distributed over a 1-dimensional processor array before calling the eigensolver `pdsygvx`. This is done using a block-cyclic distribution. The parameter `BlockSize` specified in the input files controls the block size used in the matrix distribution.

Initial tests showed that the performance of `pdsygvx` on a larger number of processors is poor, especially when compared to the results of reference [4]. To investigate this, we have modified the code to use a 2-dimensional processor grid for the eigensolver. We redistributed the data between the two processor grids with the ScaLAPACK routine `pdgemr2d`. In the following sections we will compare the performance of these two code versions.

4 Timers and timings

SIESTA 1.3 contains timers to monitor its performance. In the job-output SIESTA reports user time. For an IBM-SP, such as HPCx, the developers recommend the use of `clock` to measure user time. With the present release of the switch software time spent waiting in `MPI_Barrier` are accounted for within system time and hence not reported by `clock`. We therefore changed the code to use the IBM specific timer `irtc`, which measures elapsed time, instead of `clock`. All times in this report are measured with `irtc`. To be fair, SIESTA reports on elapsed time in a separate file, but for practical reasons, we preferred to have the times in the job output file.

The code consists out of two main sections `DHSCF` and `diagon`, which usually consume about 90% of the time of the run. In `DHSCF` the part of the Hamiltonian matrix depending on the real space mesh is constructed and in `diagon` the eigenvalues and eigenvectors of this matrix are determined. Obviously these are the key sections which any code optimisation has to target. For majority of this report we will report the times spent in `DHSCF` and `diagon` in addition to the time taken for the entire application.

¹Service Pack 7

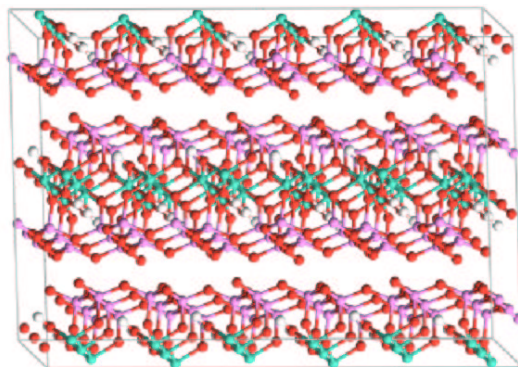


Figure 1: Pyrophyllite [5]

5 Input file

The input configuration used for this study describes pyrophyllite [5], see Figure 1, a layered clay mineral. The benchmark has 720 atoms of 4 different species. We used diagonalisation to solve the eigenvalue problem by specifying `SolutionMethod diagon` in the input file.

To keep the costs with respect to CPU-time of this study reasonable, we restricted the runs to three molecular dynamics steps and allowed a maximum of four SCF-iterations per MD-step. Our input files contain the statements `MD.FinalTimeStep 3` and `MaxSCFIterations 4`. This restriction on the SCF-iterations gives a larger weight to certain parts of the code at the expense of others when compared to a real production run. In particular the force calculation gets invoked more often. However, we do not expect these effects to be overly large and our results should characterise a realistic production run reasonably well.

We used single ζ -functions and double ζ -functions as a basis for the pseudo atomic orbitals by setting `PAO.BasisSize SZ` or `DZ` in the input file. This results in 2664 and 5328 atomic orbitals respectively.

6 Tuning the block size

The parameter `BlockSize` in the input file controls how the data are mapped onto the processors. If `BlockSize` is not specified in your input file the code will use a default of eight. The performance of parallel routines can vary dramatically with the block size, so we start the discussion by looking into the performance as a function of the block size. It should be noted that SIESTA will not work when the number of CPUs multiplied with the block size is larger than the number of orbitals. If this condition is violated, the present version of the code crashes with a segmentation fault.

Figure 2 shows the performance of the original SIESTA 1.3 for 32 and 64 CPUs as a function of the block size. The performance of the improved code using the 2-dimensional processor grid is shown in Figure 3. Here we used 64, 96 and 128 processors. We observed that the performance of the DHSCF part is essentially independent of the block size. We have omitted these times from the figures for reasons of clarity.

The results show that when using 64 or more processors, the default block size of eight does not yield optimal performance. For block size values of twenty or larger the performance becomes essentially independent of the block size.

The results in the above figures have been obtained by using double ζ -functions for the PAO basis. We repeated the investigation for single ζ -functions to check the dependence on problem size. We obtained the same result. A block size larger than twenty yields best performance. For the following

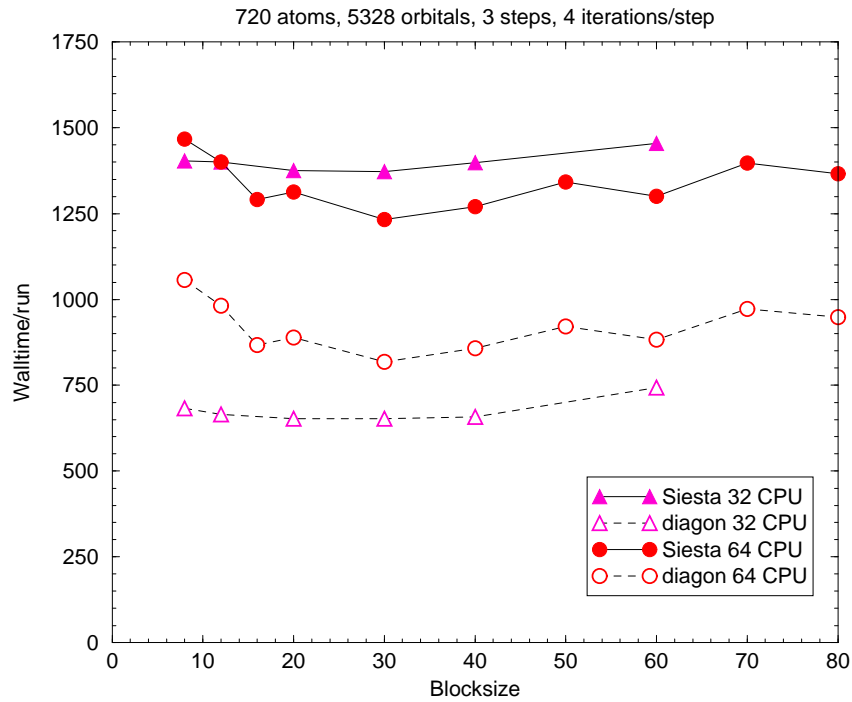


Figure 2: Dependence of the code performance on the block size when using double ζ -function for the basis, resulting in 5328 orbitals. The results are for a 1-dimensional processor grid in the diagonaliser. Full symbols give the total run time, open symbols the time spent in the diagonaliser.

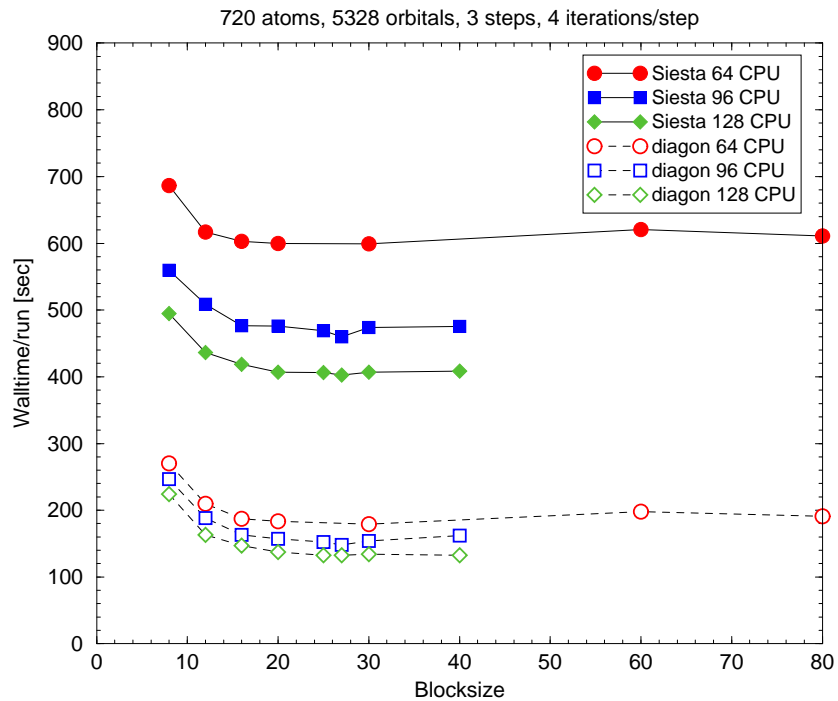


Figure 3: Dependence of the code performance on the block size when using double ζ -function for the basis, resulting in 5328 orbitals. These results are for using a 2-dimensional processor grid in the diagonaliser. Again full symbols give the total run time, open symbols the time spent in the diagonaliser.

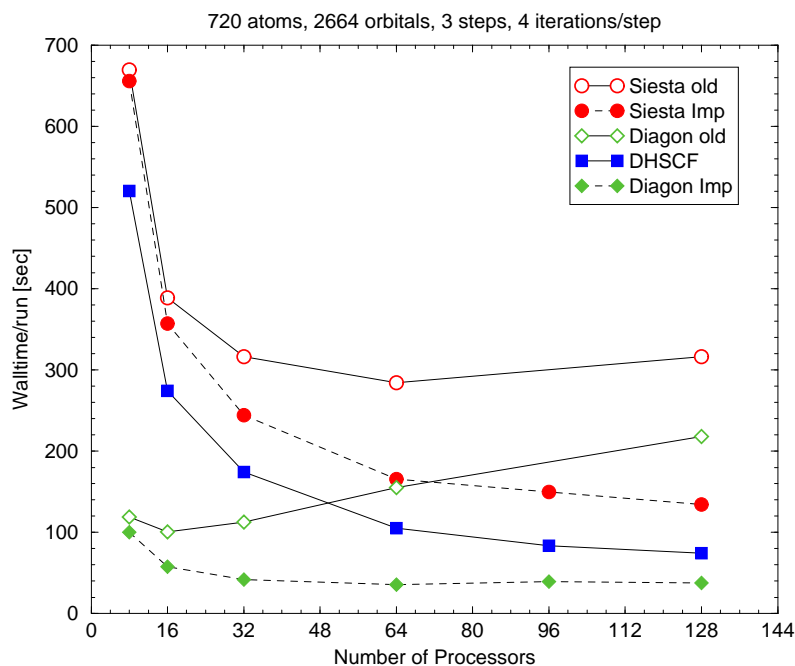


Figure 4: Parallel scaling of SIESTA’s key routines. This graph is for using single ζ -functions, resulting in 2664 orbitals. The runs consisted of 3 MD-steps with 4 SCF-iterations each. If possible a block size of 30 was used. For 96 and 128 processors, the block size had to be reduced to 27 and 20 respectively.

we use a `BlockSize`-value of 30 if possible.

7 Scaling of the code and its key routines

The next question we would like to address is the scaling of the key routines and the entire application when varying the number of processors. This is shown in Figures 4 and 5. For the diagonaliser and the entire application we label the times using the 1-dimensional processor grid as “old” and the ones using a 2-dimensional processor grid as “Imp”. Obviously the time spent in the `DHSCF` part does not depend on this change and only one set of points is given for them.

The curves show that using a 2-dimensional processor grid is vastly superior to using a 1-dimensional processor grid. This performance gap widens for an increasing number of processors. This behaviour is caused by the diagonaliser’s poor performance on a 1-dimensional processor grid. For more than 16 processors, this part of the code slows down when using more processors. When using a 2-dimensional processor grid, for the smaller problem, the diagonaliser improves up to 64 processors, after which the performance stays essentially constant. For the larger problem, the code’s performance continues improving up to 128 processors. In particular for large numbers of processors this leads to a dramatic improvement of the overall performance of SIESTA. For the large problem on 128 processors the benchmark is more than 3 times faster.

We now move to study the section `DHSCF` of the code in more detail. This code section contains 4 major subsections: initialising the mesh, setting the orbitals up, adding the contribution to the Hamiltonian and the calculation of the forces. If the mesh remains unchanged, the mesh initialisation is only called once per run. The sections to set up the orbitals and calculating the forces get called once per MD-step and the part to update the Hamiltonian gets executed every iteration, that is several times per MD-step. Hence for real production runs, the performance of this part carries more importance than any

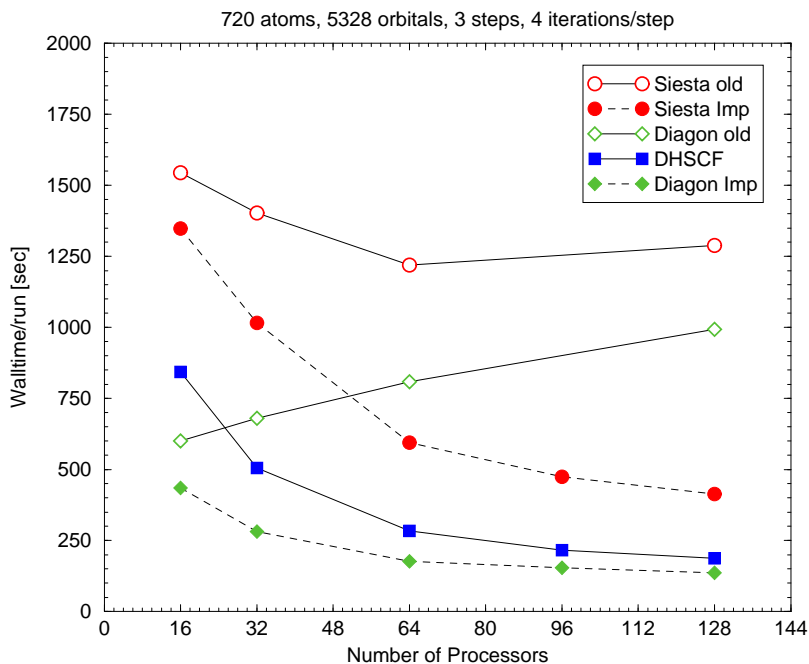


Figure 5: Parallel scaling of SIESTA’s key routines. This graph is for using double ζ -functions, resulting in 5328 orbitals. The runs consisted of 3 MD-steps with 4 SCF-iterations each. We used a block size of 30.

of the other.

Figures 6 and 7 describe the scaling of the individual routines inside DHSCF. The initialisation of the mesh does not improve at all as the number of processors increases. Since this is called very rarely and only takes a few seconds, at present this does not appear to be an issue.

The remaining routines improve when using more processors, however comparison with the dashed lines shows that these routines do not scale particular well, in particular the routine to set up the orbitals. We will discuss some of the reasons behind this behaviour in the next section.

8 Profiling the application

Performance analysis tools are very useful when it comes to understanding an application’s performance. As part of this investigation we used two performance tools: `hpmcount` and `VAMPIR` [6, 7]. The purpose of this chapter is to give an initial report on our findings. Further investigations are necessary in future.

`hpmcount` is an easy to use tool, which allows to read out the hardware counters of the CPU [6]. We have instrumented the code to get separate results for each of the key sections. Table 1 details the floating point rates, which were obtained. The first number gives the lowest observed rate, the second number the highest observed rate.

The comparison of the rows “Diagon (1D)” and “Diagon (2D)” confirms the superiority of the 2-dimensional grid. For the 2-dimensional grid, the rates are larger and do not drop off as quickly when increasing the number of processors. The minimum and maximum rate remain closer together, indicating improved load-balance for the 2-dimensional processor grid.

We now consider the subroutines inside the DHSCF part of the code. We notice that routines to set up the orbitals and to calculate the forces do not achieve a high floating point performance. In par-

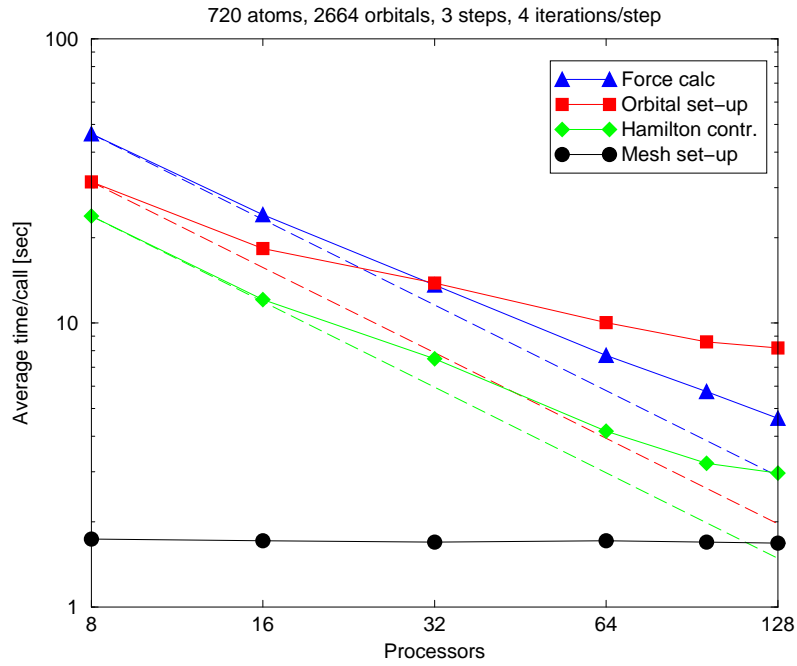


Figure 6: Parallel scaling of individual routines used inside the DHSCF-part of SIESTA. This graph is when using single ζ -functions, resulting in 2664 orbitals. We used a block size of 30 apart from 96 and 128 CPUs. For the latter we used a block size of 27 and 20 respectively. The dashed lines indicate perfect scaling.

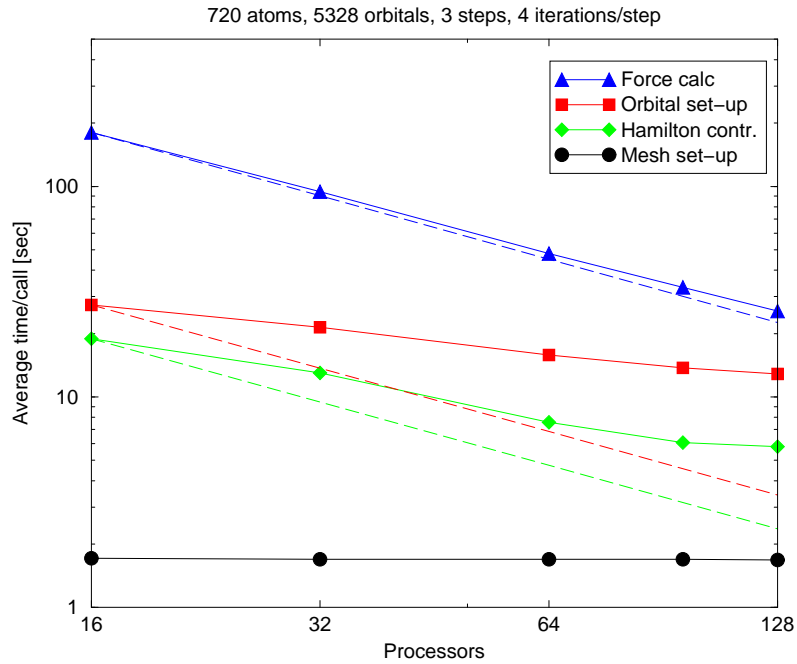


Figure 7: Parallel scaling of individual routines inside the DHSCF-part of the application. This graph is when using double ζ -functions, resulting in 5328 orbitals. We used a block size of 30.

| Basis: Processors: | single ζ -function | | | double ζ -function | | |
|-----------------------|--------------------------|---------|---------|--------------------------|----------|---------|
| | 16 | 32 | 64 | 16 | 32 | 64 |
| Diagon (1D) | 737-910 | 224-367 | 71-186 | 853-1011 | 348-444 | 120-201 |
| Diagon (2D) | 1124-1315 | 745-913 | 398-570 | 1180-1448 | 882-1018 | 678-825 |
| Orbit-Init | 226-231 | 121-180 | 86-128 | 243-246 | 127-189 | 85-127 |
| Hamilton contr. | 474-480 | 317-451 | 293-422 | 449-458 | 240-426 | 200-358 |
| Force-Calc | 182-191 | 152-178 | 125-161 | 37-42 | 30-68 | 28-64 |

Table 1: Floating point performance in Mflop/s of key routines. Diagon (1D) denotes the performance of the diagonaliser with a 1-dimensional processor grid and Diagon (2D) with a 2-dimensional grid. All results were obtained using a block size of 30.

ticular the floating point rate for the force calculation for the larger problem using double ζ -functions is low. Also the difference between the minimum and maximum increases with increasing number of processors, indicating problems with the load-balance for a larger number of processors.

To improve our understanding about the behaviour of the code, we investigated the routines with VAMPIR [7]. VAMPIR is particularly suitable for investigating load-balance issues. In Figure 8 we show the graphical output of VAMPIR for part of the code, adding the mesh contribution to the Hamiltonian when using 16 and 96 processors. To keep the output from VAMPIR small, we used the input files with single ζ -functions. In the figure, dark green colour visualises time spend inside the calculation part of the code and the red colour the time spend in MPI calls. The comparison of the upper and lower part of Figure 8 shows the substantial increase of the communication overheads on the larger number of processors. It also shows the load-balance problems of the code for the larger number of processors. Processors 2, 3, 8, 9, ... spend a substantial amount of time in `MPI_Barrier` and `MPI_Allreduce`, the latter is marked with the number 253 in Figure 8, while waiting for processors 0, 5, 6, 11, ... to finish their calculation. We further noticed by enlarging sections of the graphs, that the code is performing several communicator splits during its execution instead of storing the split communicators. The change is simple and removing the calls to `MPI_Comm_split` in favour of storing reduced the average time in determination of the Hamiltonian contribution from 3.21s to 3.01s per call when using 96 processors for the small problem with single ζ -functions. For the present code, this is not dramatic, but a single call to `MPI_Comm_split` costs between 40 and 50 μ sec. When attempting to run the code on even large numbers of processors in the future this overhead will become costly and avoiding it is very simple.

Figure 9 shows the output from VAMPIR for the force calculation on 96 processors. Again we use the small input file. In this figure the dark green shows the time spent in the force calculation and the red colour time spent in the MPI-library. Time inside the application code but outside the actual force calculation is shown in a lighter shade of green. The figure shows that when using such a large number of processors for the problem size under investigation, problems with the load balance start to show up. Most of this effects the application code following the force calculation. Individual processors spend more than two seconds waiting in `MPI_Bcast`

In Figure 10 we show VAMPIR traces for the initialisation of the orbitals. Again we observe the same pattern of load-imbalance as for the other routines inside the DHSCF-part of the code. However the total time spent inside the MPI-library for this routine can not explain the scaling of this routine as observed in Figure 6. We expect algorithmic reasons to be the cause. Further investigation is needed.

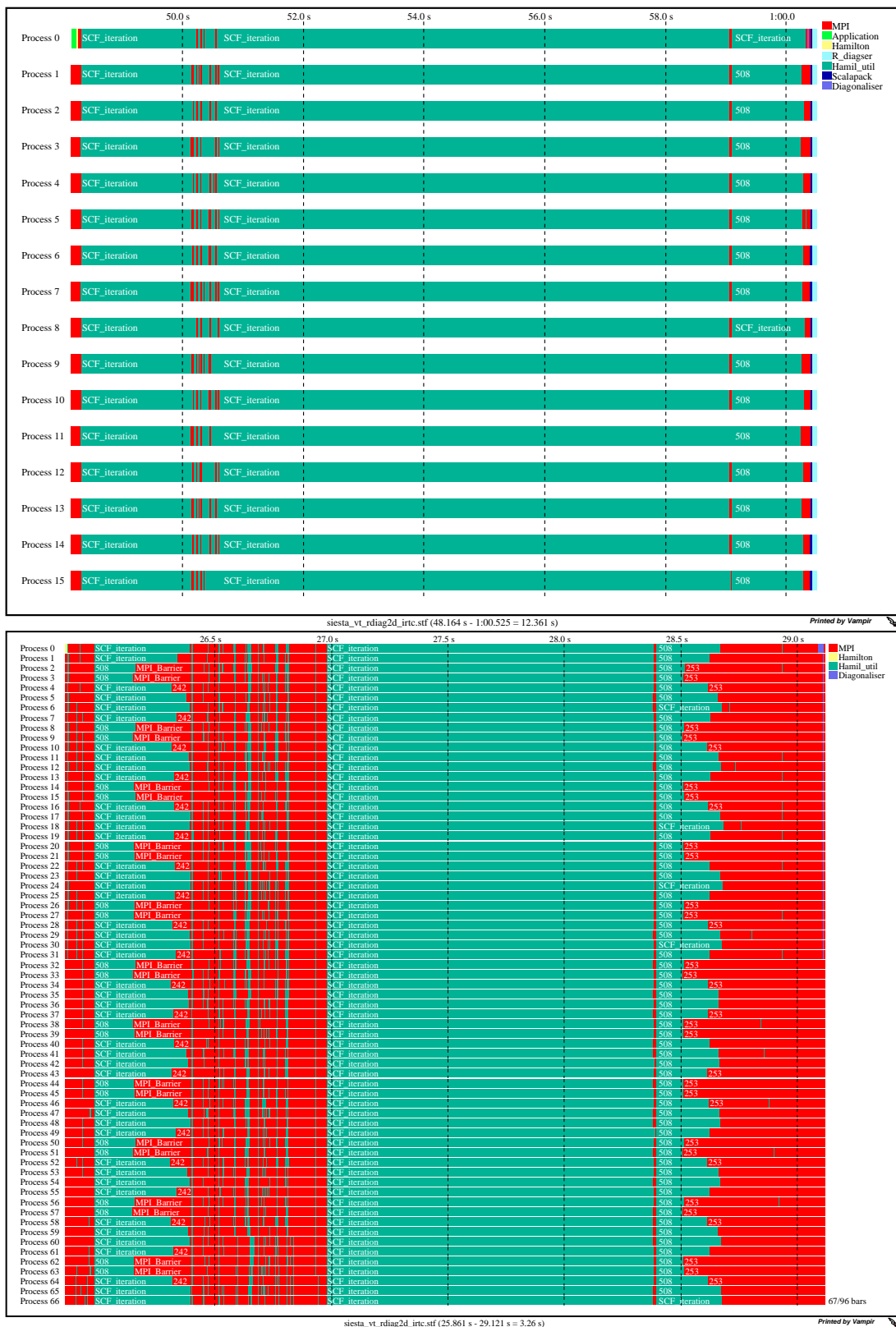


Figure 8: VAMPIR trace of the code to add the mesh contribution to the Hamiltonian. The top portion show the trace for 16 processors, the bottom portion for 67 out of 96 processors. We used the small problem with single ζ -functions.

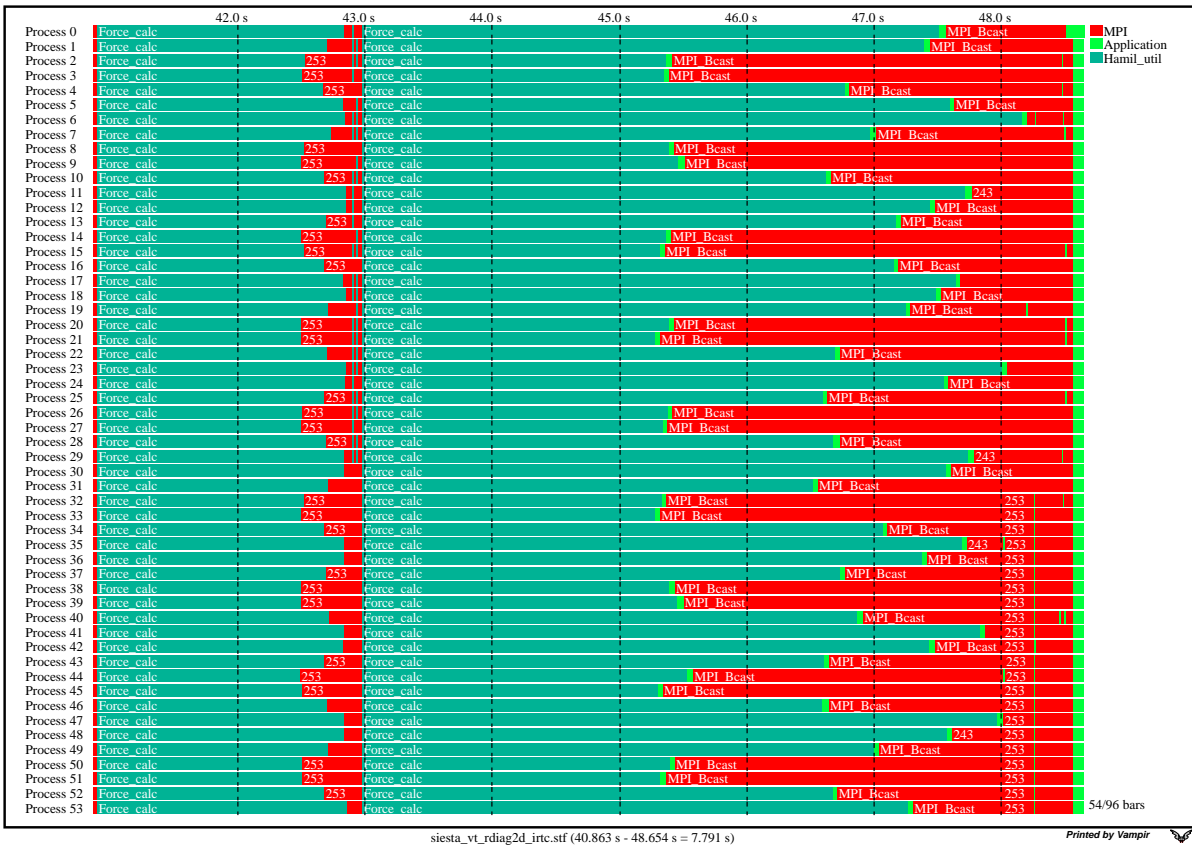


Figure 9: VAMPIR trace file for the force calculation on 67 out of 96 processors.

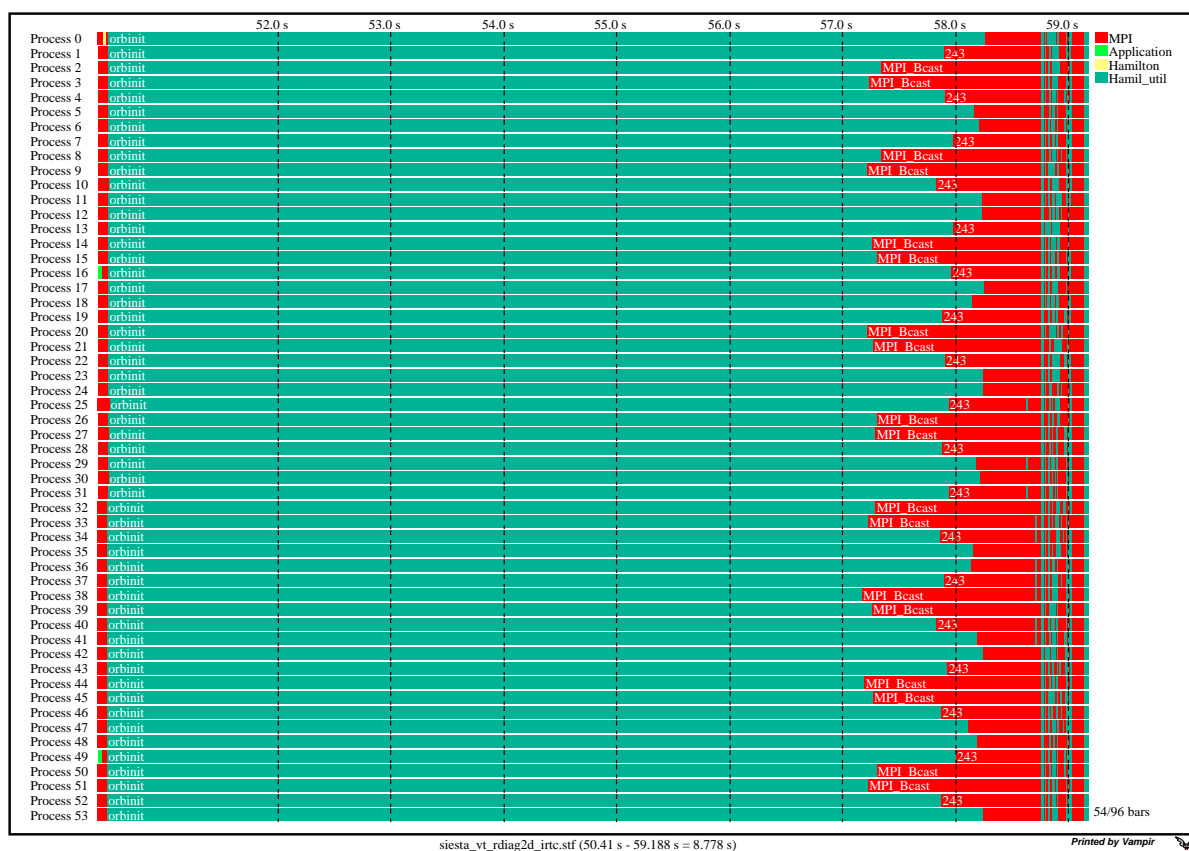


Figure 10: VAMPIR trace file for the initialisation of the orbitals on 67 out of 96 processors.

| Basis: | single ζ -function | | | double ζ -function | | |
|--------|--------------------------|-------|-----------|--------------------------|-------|-----------|
| | before | after | TASK_AFF. | before | after | TASK_AFF. |
| SIESTA | 134 s | 131 s | 118 s | 415 s | 381 s | 364 s |
| DHSCF | 74 s | 73 s | 66 s | 188 s | 180 s | 170 s |
| diagon | 38 s | 32 s | 31 s | 136 s | 114 s | 111 s |

Table 2: Effect of the microcode upgrade on the performance on 128 processors. We compare the performance before and after the upgrade. We also study the effect of the new environment variable `MP_TASK_AFFINITY`, introduced with the upgrade.

9 Upgraded switch microcode

In the later stages of the project the microcode used on the switch was upgraded. We found this upgrade to be very beneficial to SIESTA’s performance. The upgrade also introduced a new environment variable `MP_TASK_AFFINITY`, which has to be used in connection with the environment variable `MEMORY_AFFINITY`. We observed SIESTA to respond very well to setting both variables to “MCM”. Setting these two environment variables ensure that processes use the memory most local to the CPU they are utilising, which minimises CC-NUMA effects inside the p690+ frames.

In Table 2 we compare the performance of the entire application, the `DHSCF` and the `diagon` parts on 128 processors for both problem sizes. In each case we observe a performance improvement of 14% when comparing the new microcode with `MP_TASK_AFFINITY` to the performance before the upgrade.

10 Running SIESTA on HPCx

Before moving to our conclusion, we give a few practical hints to our users when running SIESTA on HPCx.

To ensure good performance of SIESTA on HPCx, we recommend you to set the following environment variables, but please test the effect of these settings on your own configurations:

```
export MP_SHARED_MEMORY=yes
export MP_EAGER_LIMIT=65536
export MP_USE_BULK_XFER=yes
export MEMORY_AFFINITY=MCM
export MP_SINGLE_THREAD=yes
export MP_TASK_AFFINITY=MCM
```

These settings have to be included into your LoadLeveler script after “#@ queue” and before calling SIESTA. The above syntax is for K-shell and you might have to use a different syntax when using a different UNIX-shell inside your LoadLeveler script.

As discussed in section 6, the default value of eight for the parameter `BlockSize` in the input file is not optimal on HPCx. SIESTA users on HPCx should increase this to at least 20. Where possible we have used a value of 30. Re-testing the dependence on the `BlockSize` for your own input file is highly encouraged. In this context we want to highlight once more the number of processors multiplied by the `BlockSize` needs to be smaller than the number of orbitals. If this condition is violated, your run will fail with a segmentation fault.

The last hint concerns the input files. SIESTA reads its input from the standard input. The SIESTA user guide recommends redirecting the input file to standard input. However we observed that

on the HPCx system the initialisation of the MPI-library in `MPI_Init` times out, when the size of the file exceeds several 10 kB. SIESTA 1.3 has an undocumented feature, by naming your input-file `INPUT_DEBUG` the redirection can be avoided. We start SIESTA with the line “`poe siesta`” inside the LoadLeveler script and SIESTA reads its input from `INPUT_DEBUG`.

11 Conclusion

This study investigates the performance of SIESTA on the HPCx Phase2 system. We have demonstrated that changing the processor grid of the diagonaliser from one to two dimensions leads to a substantially improved parallel scaling. This improvement is very timely with respect to the recent changes to the HPCx hardware. SIESTA is now in good shape to use several 32-way SMP-nodes for a single calculation.

When using about 100 processors for the problem size under investigation, problems with the load-balance start to show. If one wants SIESTA to scale to an even larger number of processors, this needs addressing in a future release.

Our investigation shows further that the density of floating point operations of some of the key routines, in particular the calculation of the forces is quite low. The reason behind this should be studied in future. Our investigation shows that this is not caused by problems with the load balance. Improvements to the flop density would benefit all SIESTA calculation not just those on massively parallel machines.

We have liaised with the SIESTA developers and the improved diagonaliser will be included into a future SIESTA release. If you are interested in this work or in obtaining modified routines of the diagonaliser, please contact the HPCx helpdesk, email: `support@hpcx.ac.uk`.

Acknowledgement

This study would not have been possible without Jon Wakelin. He suggested this study and provided us with the input files required for this study. Together with Andrey Gal, he gave us useful hints on how to run SIESTA. Jon and Andrey are both users of the HPCx service. I would like to thank Mark Bull and Lorna Smith for encouragement and advise throughout this study. Particular thanks go to Elena Breitmoser, Ian Bush and Andrew Sunderland for their explanations of ScaLAPACK and sharing their experiences with different processor grids. In the late stages of this project I had useful exchanges with Julian Gale from the SIESTA development team. I would like to thank Fiona Reid for commenting on the manuscript prior to publication.

References

- [1] J.M. Soler, et al., *J. Phys.: Condens. Matter* **14**, 2745 (2002).
- [2] There is a dedicated SIESTA website with information on the code, user documentation and how to obtain a licence:
<http://www.uam.es/departamentos/ciencias/fismateriac/siesta/>
- [3] ScaLAPACK website http://www.netlib.org/scalapack/scalapack_home.html
- [4] Elena Breitmoser, Andy G. Sunderland, “A performance study of PLAPACK and ScaLAPACK Eigensolvers on HPCx for the standard problem”, HPCx technical report HPCxTR0406
http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0406.pdf
- [5] Jon Wakelin, private communication.

- [6] For information on how to use hpmcount on HPCx check the HPCx Userguide
<http://www.hpcx.ac.uk/support/documentation/UserGuide/HPCxuser/Tools.html>
or HPCx technical report HPCxTR0307
http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0307_choose.html
- [7] For information on how to use VAMPIR on HPCx and links to full documentation, please check the HPCx Userguide
<http://www.hpcx.ac.uk/support/documentation/UserGuide/HPCxuser/Tools.html>