Performance evaluation for petascale quantum simulation tools

Stanimire Tomov[1], *Wenchang Lu*[2,3], *Jerzy Bernholc*[2,3], *Shirley Moore*[1], and *Jack Dongarra*[1,3]

[1] University of Tennessee, Knoxville, TN
[2] North Carolina State University, Raleigh, NC
[3] Oak Ridge National Laboratory, Oak Ridge, TN

ABSTRACT: This paper describes the performance evaluation and analysis of a set of open source petascale quantum simulation tools for nanotechnology applications. The tools of interest are based on the existing real-space multigrid (RMG) method. In this work we take a reference set of these tools and evaluate their performance with the help of performance evaluation libraries and tools such as TAU and PAPI. The goal is to develop an in-depth understanding of their performance on Teraflop leadership platforms, and moreover identify possible bottlenecks and give suggestions for their removal. The measurements are being done on ORNL's Cray XT4 system (Jaguar) based on quad-core 2.1 GHz AMD Opteron processors. Profiling is being used to identify possible performance bottlenecks and tracing is being used to try to determine the exact locations and causes of those bottlenecks. The results so far indicate that the methodology followed can be used to easily produce and analyze performance data, and that this ability has the potential to aid our overall efforts on developing efficient quantum simulation tools for petascale systems.

KEYWORDS: performance evaluation, petascale tools, quantum simulations, TAU, PAPI.

1 Introduction

Computational science is firmly established as a pillar of scientific discovery promising unprecedented capability. In particular, computational advances in the area of nanoscale and molecular sciences are expected to enable the development of materials and systems with radically new properties, relevant to virtually every sector of the economy, including energy, telecommunications and computers, medicine, and areas of national interest such as homeland security. The advances that would enable this progress critically depend on the development of **petascale-level quantum simulation tools** and their adaptation to the currently available petascale computing systems. However, this tool development and adaptation is a highly non-trivial task that requires a truly interdisciplinary team, and cannot be accomplished by scientists working in a single discipline. This is due to the following challenges, which require expertise in several areas:

– Quantum simulation methods, while by now well-established, provide various "levels" of accuracy at very different cost. The most accurate methods are prohibitively expensive for large systems and scale poorly with system size (up to $O(N^7)$). Disciplinary expertise is required to develop multiscale methods that will provide sufficient accuracy at acceptable cost, while performing well for grand-challenge-size problems at the petascale level.

– Existing codes and algorithms perform well on current terascale systems, but need major performance tuning and adaptation for petascale systems (based for example on emerging multi/many-core and hybrid architectures). This requires substantial computer science expertise, use of advanced profiling and optimization tools, and additional development of these tools to adapt them to different petascale architectures, with different memory hierarchies, latencies and bandwidths. The profiling may also identify algorithmic bottlenecks that inhibit petaflop performance.

– New or improved algorithms can greatly decrease time to solution and thus enhance the impact of petascale hardware. Very large problems often exhibit "slow down" of convergence, requiring "coarse-level" accelerators adapted to the particular algorithm. Algorithmic changes to decrease bandwidth or latency requirements may also be necessary. In time-dependent simulations, sophisticated variable time-stepping and implicit methods can greatly increase the "physical time" of the simulation, enabling the discovery of new phenomena.

This work is a step towards solving some of the computer science problems related to the adaptation of existing codes and algorithms to petascale systems based on multicore processors. In particular, we take a reference set of quantum simulation tools that we are currently developing [4, 5], and show the main steps in evaluating their performance, analyzing it to identify possible performance bottlenecks, and determining the exact locations and causes of those bottlenecks. Based on this, we also give recommendations for possible performance optimizations. We use state-of-the-art performance evaluation libraries and tools including TAU (Tuning and Analysis Utilities [6]) and PAPI (Performance Application Programming Interface [2, 1]). The measurements are being done on Jaguar, a Cray XT4 system at ORNL based on quad-core 2.1 GHz AMD Opteron processors. The results so far indicate that the main steps that we have followed (and described) can be viewed/used as a methodology to not only easily produce and analyze performance data, but also to aid the development of algorithms, and in particular petascale quantum simulation tools, that effectively use the underlying hardware.

2 Performance evaluation

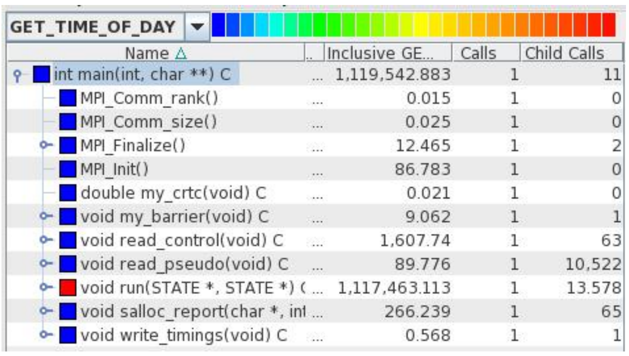
Here we describe the performance evaluation techniques that we found most useful for this study. We also give a performance evaluation for two methodologies that we have implemented so far in our codes. One is global grid method, in which the wave functions are represented in the real space uniform grids [5]. The results show that the most time-consuming part in this method is the **orthogonalization** and **subspace diagonalization**. The code is massively paralleled and it can reach a very good flops performance. Unfortunately, it scales in $O(N^3)$ with system size, which will be prohibitive for large system applications. The other one is the optimally localized orbital method [4], in which the basis set is optimized variationally and it scales in nearly $O(N)$ with system size. This $O(N)$ method has some computational challenges in parallel and strong scaling with the number of processors. In this contribution, we profile (below) and analyze (Section 3) the code using different tools, find the bottlenecks and optimize the performance (Section 4).

2.1 Profiling

An efficient way to quickly get familiar with large software packages and identify possible performance bottlenecks is to use TAU's profiling capabilities. For example, we use it to first get familiar with the general code structure of the quantum simulation tools of interest (Section 2.2), second to get performance profiles (Section 2.3) revealing the main function candidates for performance optimization, and finally to profile in various hardware configuration scenarios to analyze the code and identify possible performance bottlenecks.

2.2 Code structure

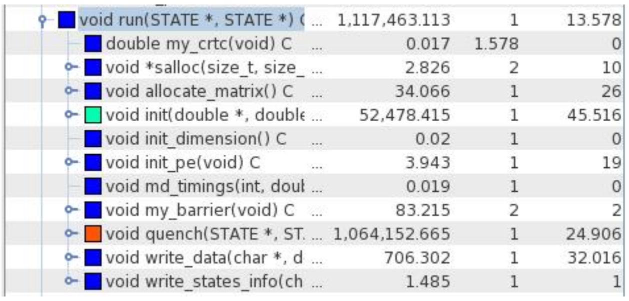
Code structure can be easily studied by generating call-path data profiles. For example TAU is compiled with the -PROFILECALLPATH option and the run time variable TAU_CALLPATH_DEPTH can be set to limit the depth of profiled functions. Figure 1 shows



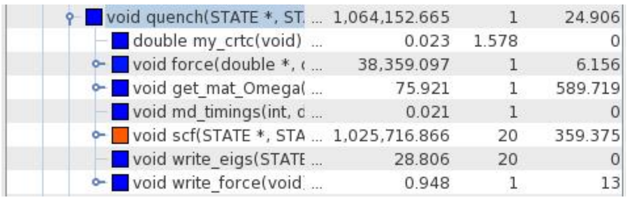
Name △	Inclusive GE...	Calls	Child Calls
🟦 int main(int, char **) C ...	1,119,542.883	1	11
🟦 MPI_Comm_rank() ...	0.015	1	0
🟦 MPI_Comm_size() ...	0.025	1	0
🟦 MPI_Finalize() ...	12.465	1	2
🟦 MPI_Init() ...	86.783	1	0
🟦 double my_crtc(void) C ...	0.021	1	0
🟦 void my_barrier(void) C ...	9.062	1	1
🟦 void read_control(void) C ...	1,607.74	1	63
🟦 void read_pseudo(void) C ...	89.776	1	10,522
🟥 void run(STATE *, STATE *) (...	1,117,463.113	1	13.578
🟦 void salloc_report(char *, int ...	266.239	1	65
🟦 void write_timings(void) C ...	0.568	1	1

Fig. 1. Callpath data.

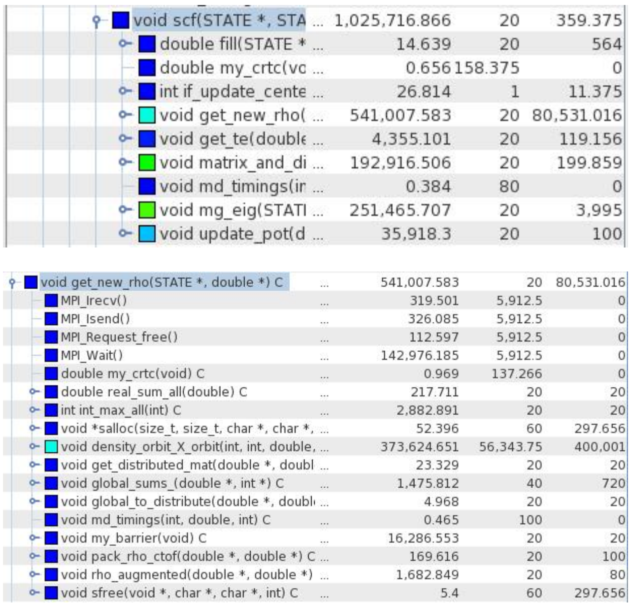
a view of the call-path data (of the optimally localized orbital method code) using the `paraprof` tool, by selecting from the pool-down menu consecutively `Windows`, `Thread`, `Statistics Table` (if finally `Call Graph` is selected one can see a graphical representation/graph of the call path). For example, shown are what are the functions in `main`, their inclusive execution times (the exclusive execution times are minimized for this snapshot), how many times are the functions called, and how many child functions are



🟦 void run(STATE *, STATE *) (...	1,117,463.113	1	13.578
🟦 double my_crtc(void) C ...	0.017	1.578	0
🟦 void *salloc(size_t, size_ ...	2.826	2	10
🟦 void allocate_matrix() C ...	34.066	1	26
🟩 void init(double *, double ...	52,478.415	1	45.516
🟦 void init_dimension() C ...	0.02	1	0
🟦 void init_pe(void) C ...	3.943	1	19
🟦 void md_timings(int, doul ...	0.019	1	0
🟦 void my_barrier(void) C ...	83.215	2	2
🟧 void quench(STATE *, ST. ...	1,064,152.665	1	24.906
🟦 void write_data(char *, d ...	706.302	1	32.016
🟦 void write_states_info(ch ...	1.485	1	1



🟦 void quench(STATE *, ST. ...	1,064,152.665	1	24.906
🟦 double my_crtc(void) ...	0.023	1.578	0
🟦 void force(double *, (...	38,359.097	1	6.156
🟦 void get_mat_Omega(...	75.921	1	589.719
🟦 void md_timings(int, c ...	0.021	1	0
🟧 void scf(STATE *, STA ...	1,025,716.866	20	359.375
🟦 void write_eigs(STATE ...	28.806	20	0
🟦 void write_force(void) ...	0.948	1	13



void scf(STATE *, STA ...	1,025,716.866	20	359.375
double fill(STATE * ...	14.639	20	564
double my_crtc(vo ...	0.656	158.375	0
int if_update_cente ...	26.814	1	11.375
void get_new_rho(...	541,007.583	20	80,531.016
void get_te(double ...	4,355.101	20	119.156
void matrix_and_di ...	192,916.506	20	199.859
void md_timings(ir ...	0.384	80	0
void mg_eig(STATI ...	251,465.707	20	3,995
void update_pot(d ...	35,918.3	20	100

void get_new_rho(STATE *, double *) C ...	541,007.583	20	80,531.016
MPI_Irecv() ...	319.501	5,912.5	0
MPI_Isend() ...	326.085	5,912.5	0
MPI_Request_free() ...	112.597	5,912.5	0
MPI_Wait() ...	142,976.185	5,912.5	0
double my_crtc(void) C ...	0.969	137.266	0
double real_sum_all(double) C ...	217.711	20	20
int int_max_all(int) C ...	2,882.891	20	20
void *salloc(size_t, size_t, char *, char *, ...	52.396	60	297.656
void density_orbit_X_orbit(int, int, double, ...	373,624.651	56,343.75	400,001
void get_distributed_mat(double *, doubl ...	23.329	20	20
void global_sums_(double *, int *) C ...	1,475.812	40	720
void global_to_distribute(double *, doubl ...	4.968	20	20
void md_timings(int, double, int) C ...	0.465	100	0
void my_barrier(void) C ...	16,286.553	20	20
void pack_rho_ctof(double *, double *) C ...	169.616	20	100
void rho_augmented(double *, double *) ...	1,682.849	20	80
void sfree(void *, char *, char *, int) C ...	5.4	60	297.656

Fig. 2. Callpath data.

called from them. In this case, function **`run`** is the most time consuming and left-mouse clicking on it reveals similar information for its children. This is shown on Figure 2, where from **`run`** we get to **`quench`**, and next to **`scf`** (obviously called in an outer loop 20 times), **`get_new_rho`** (again called 20 times), and **`density_orbit_X_orbit`** (called $56,343$ times for this run) as the most time consuming functions. Starting from here, and revealed furthermore by consequent profiling and tracing, we can say that as most of the work is done in **`density_orbit_X_orbit`** (further denoted by `DOxO`), the most probable improvements in performance will come from optimizing it. The optimization though, as expected, is coupled with the function's relation to load balancing and patterns of computation and MPI communication, which is the subject of the finer performance studies below. The considerable time spent in `MPI_Wait` hints to further look for possible load dis-balance or different ways of organizing/mixing computation and communication.

2.3 Performance Profiles

Figure 3 shows a `TAU` profile using `paraprof`. On the top we have a display of the most time consuming functions (from left to right) along with indication for their execution time for the different threads of execution used for this run. At the bottom we see a legend giving the correspondence of color-to-function name, and its execution time as a percentage from the total execution time (in this case taken for the mean). This indicates at a glance

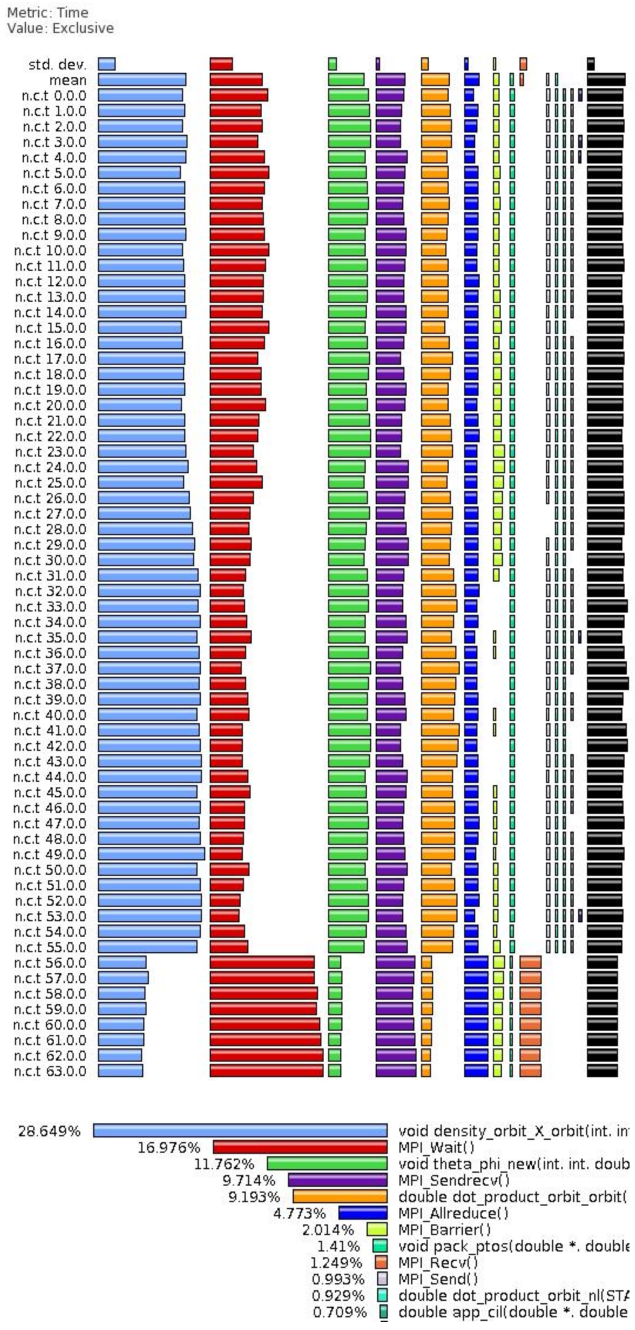


Fig. 3. Code profile.

the main function candidates for performance optimization. Namely, as already determined, this is function `DOxO`. Moreover, it is obvious that there is some load dis-balance between two groups of execution threads, namely the ones from 0-to-55 and from 56-to-63, or that the two groups have different algorithm-specific functionalities. In either case, the time spent on `MPI_Wait` seems to be excessive.

Similar profiles can be easily generated for various performance counters. For example, Figure 4 shows

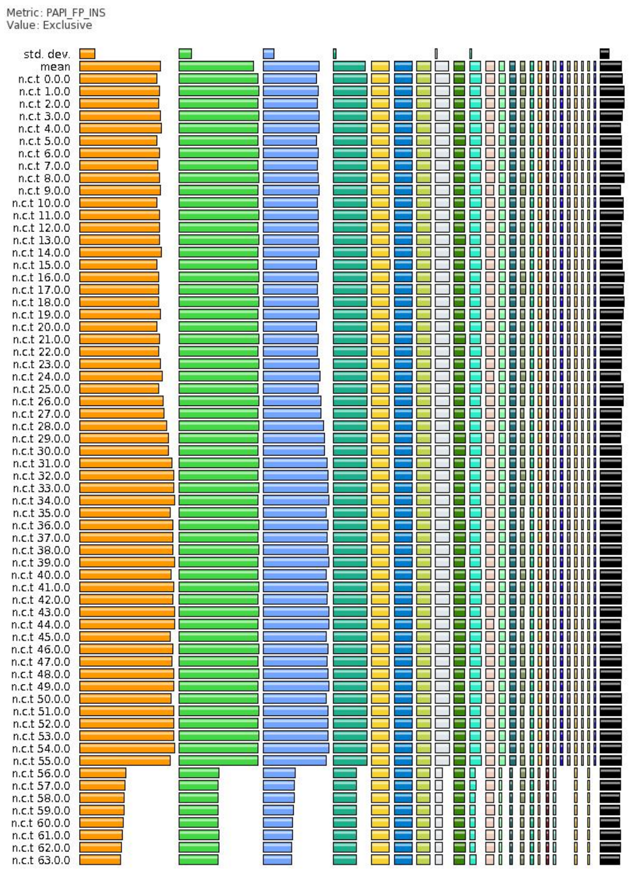


Fig. 4. Floating point instructions.

profiles for the floating point instructions counter `PAPI_FP_INS` from PAPI. This shows for example that the most time consuming function is not the one performing most of the Flops (actually, it is third in terms of Flops performed) further underscoring the need to look for its possible optimization. Moreover, the Flops in the second thread group are approximately twice less, requiring a check for load dis-balance, or in the case of different functionality, further analysis on how this 2-way splitting influences the scalability.

2.4 Performance evaluation results

The results shown so far are for the purpose of illustration. Both codes were profiled for larger problems and using larger number of processing elements (up to 1024 was enough for the purpose of this study). As mentioned at the beginning of this section, the most time-consuming part in the real space uniform grids method is the **orthogonalization** and **subspace diagonalization**. Figure 5, Top shows the profile for the most time consuming functions on a run using 1024 cores.

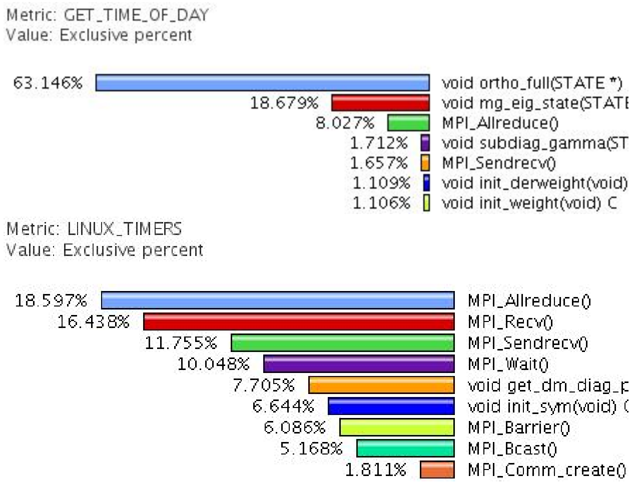


Fig. 5. Code profiles for large problems on 1024 cores.

Figure 5, Bottom shows the profile for the most time consuming functions on the optimally localized orbital method, again for 1024 cores. The first code runs at an overall 670 MFlop/s per core *vs* 114 MFlop/s per core for the second code (in both cases using all 4 core in the nodes). Both codes are based on domain decomposition and have good weak scalability. The optimally localized orbital method has some computational challenges in parallel and strong scaling with the number of processors. The basis set in this method is optimized variationally which makes it "richer" on sparse operations and MPI communications (compared to the first code).

3 Performance analysis

There are 3 main techniques that we found useful in analyzing the performance. These are tracing (and in particular comparing traces of various runs), scalability studies, and experimenting for multicore use, which are all described briefly as follows.

3.1 Tracing

Tracing in general is being used to try to determine the exact locations and causes of bottlenecks. We used TAU to generate trace files and analyzed them using **Jumpshot** [8] and tools like **KOJAK** [7]. The codes that we analyze are well written - in the sense that communications are already blocked, asynchronous, and intermixed (and hence overlapped) with computation to a certain degree. We found that in our case, when domain decomposition guarantees to a degree weak scalability, to improve performance we have to concentrate mainly on the efficient use of the multicores within a node. Related to using the tracing tools and MPI, we discovered that it may help if we increase the degree of posting early MPI_Irecvs. This is again related to the efficient use of multicores and MPI and is further discussed in our recommendations for performance improvements. We also found it useful to compare traces of various runs - and thus study the effect of various code changes on the performance. Using traces, one can also easily generate profile-type statistics for various parts of the code, which we found very helpful in understanding the performance.

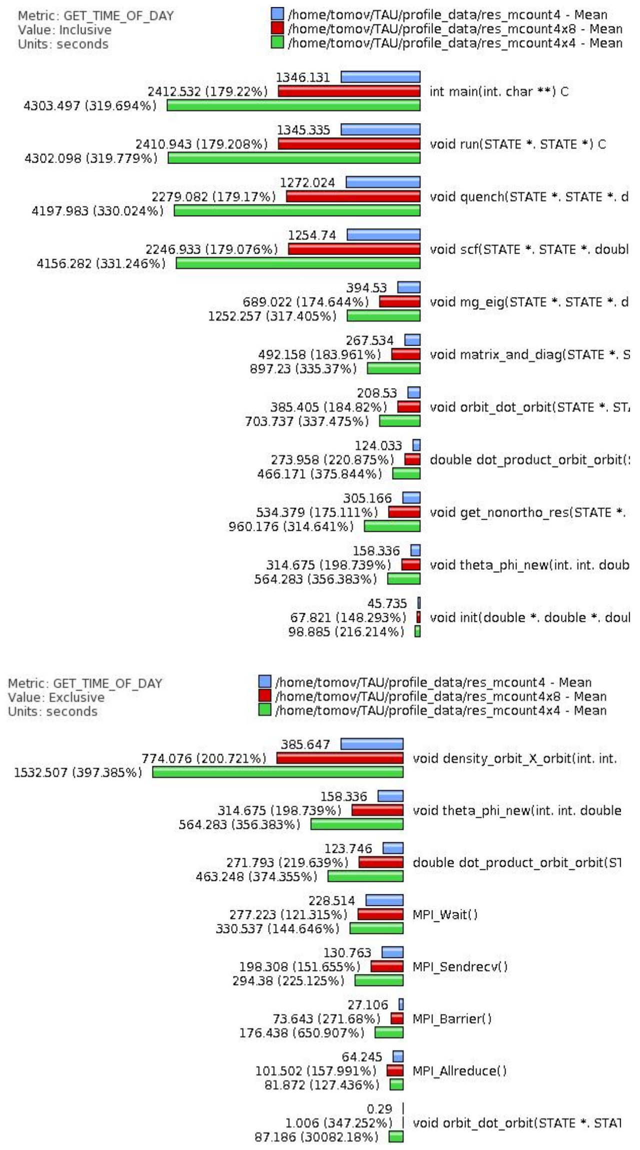


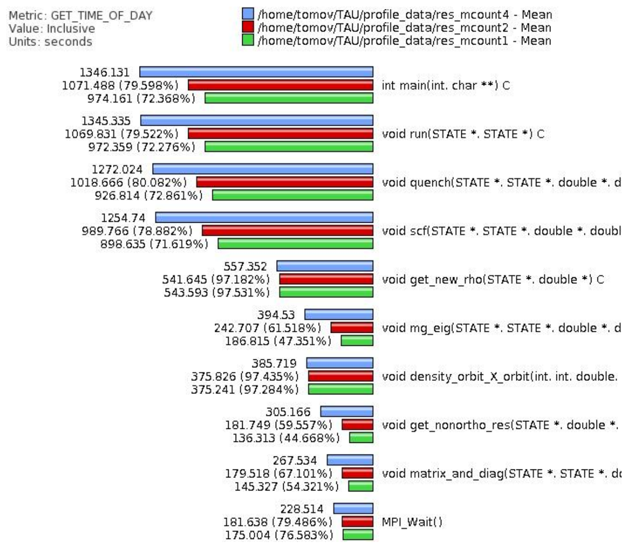
Fig. 6. Scalability using 4, 8, and 16 quad-core nodes.

3.2 Scalability

We studied both strong and weak scalability of our codes. The results were briefly summarized in subsection 2.4. Here we give a brief illustration on some strong scalability results using, as before, the small size problems with the optimally localized orbital method. Figure 6 shows the inclusive (top) and exclusive (bottom) execution time comparison when using 16 quad-core nodes (in blue), 8 (red), and 4 nodes (green). We note that the top 3 compute intensive functions from Figure 4 scale well (see Figure 6 bottom). This is important, since there was the concern of load dis-balance for a group of threads, as explained earlier. This result shows that the balance is properly maintained and that the 2 groups of threads observed before have different functionality (and their load also gets proportionally split). The less than perfect overall scaling (see inclusive time on top) may be due to the fact that the measurement here is for strong scalability, i.e. the problem size has been kept fixed and the cost of communication has not become yet small enough, compared to computation, due to surface to volume effects that we have due to domain decomposition techniques that we employ. Improvements though are possible, as will be discussed further.

3.3 Multicore use

To understand how efficient the code is in using multicore architectures, we perform measurements in different hardware configurations. Namely, we compare runs using 4, 2, and single core of the quad-core nodes. In all cases we vary the number of nodes used so that the total number of execution threads (one per core) is 64. Figure 7 shows results for comparing the inclusive (top) and exclusive (bottom) execution times.



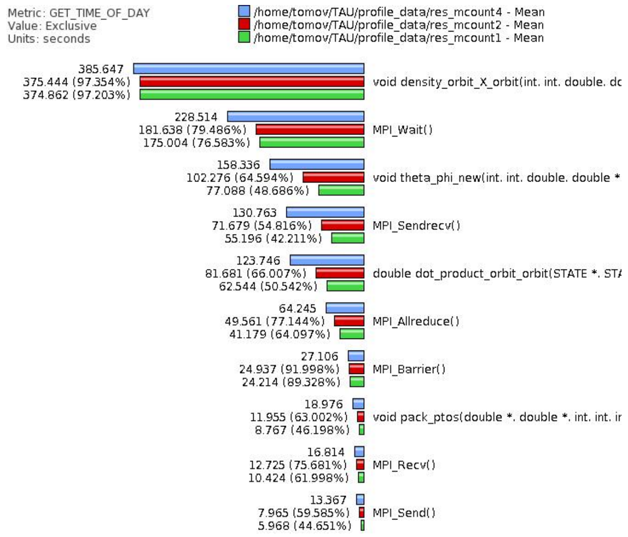


Fig. 7. Comparison using 4, 2, and 1 cores.

Note that `D0x0`, the most time consuming function, efficiently uses multicores as the time for 2 (given in red) and 4 cores (in blue) is the same as that for a single core (green). The same is not true though for the other most time consuming functions. For example, comparing single and quad-core exclusive run times for the next 5 most time consuming functions (from Figure 3), we see that the quad-core based runs are correspondingly 23, 51, 58, 49, and 36% slower, which results in an overall slow down of about 28% (as seen from the inclusive times for `main`).

These results are not surprising as multicore use is almost always of concern. In our case, a general reason that can explain the slowdown is that execution threads are taken as separate MPI processes that do not take advantage of the fact that locally, within a node, we have shared memory and can avoid MPI communications. The fact is that the local MPI communications (within a multicore), even if invoked as non-blocking, would end up getting executed as a copy that is blocking, and therefore there would be no overlap of communication and computation, contributing to the increase of `MPI_Wait` time. But besides slowdown due to local (within a multicore) communication, there is more load on the communications coming to and from a node. These bottlenecks have to be addressed: copies should be avoided within the nodes and the inter-nodes communications related to the cores of a single node should be blocked whenever possible to avoid latencies associated with multiple instances of communication.

As mentioned above, multicores are efficiently used for our most time consuming function, and not that well used for the communication related functions. Looking at the compute intensive functions, namely `dot_product_dot_orbit` and `theta_phi_new` (from Figure 4), we see they slow down with multicore use significantly more than even `MPI_Wait`: correspondingly 51 and 49% when comparing single *vs* quad-core, as already mentioned above. Understanding this, especially in relation to why the third compute intensive function `D0x0` (and also most time consuming) is fine under multicores, requires further measurements and analysis, as done next.

Figure 8 shows a performance comparison when

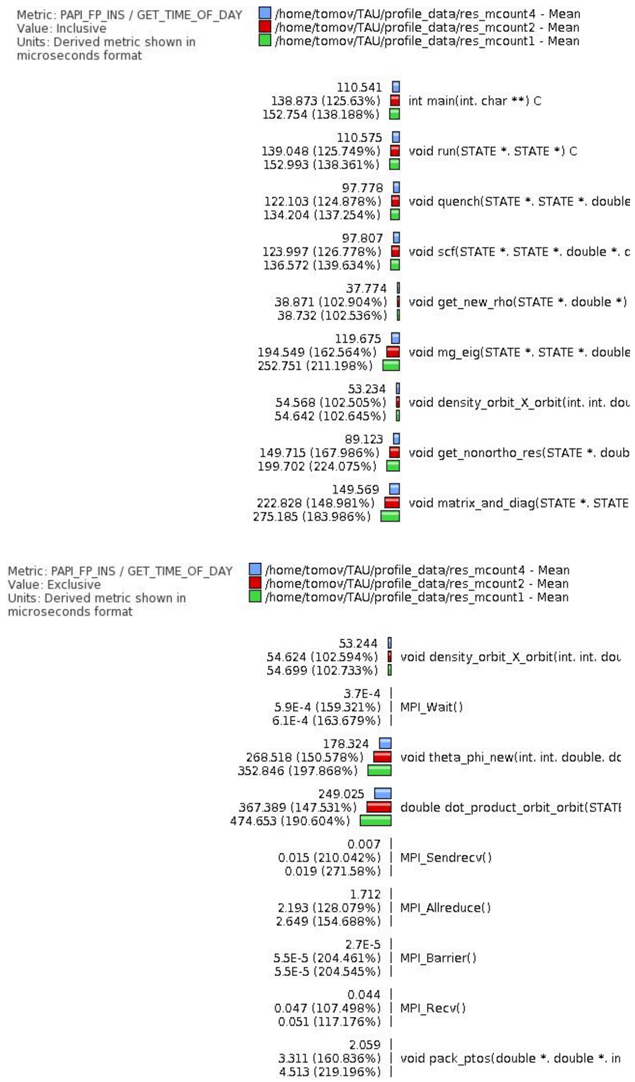


Fig. 8. Performance using 4, 2, and 1 cores.

using the 4, 2, and single core regimes, i.e. this is a combination of the time (as in Figure 3) and Flops measurements (as in Figure 4). The numbers give MFlop/s ratios. This shows for example that the code runs at an overall speed of about 110 MFlop/s per core when using quad-cores and 152 MFlop/s when using a single core per node. We note that the maximum performance is 8.4 GFlop/s per core and the main memory bandwidth is 10.6 GB/s. These are just two numbers to keep in mind in evaluating maximum possible speedups depending on the operations being performed. In our case, it looks like `DOxO` uses very irregular memory accesses, so performance is very poor (about 53 MFlop/s which is about 9 and 7 times less than the two most compute intensive functions, correspondingly `dot_product_dot_orbit` and `theta_phi_new`. Moreover, the accesses are so irregular that the bottleneck is not in bandwidth (otherwise performance would have degraded with multicore use) but in latencies. Consideration should be given here to see if the computation can be reorganized to have more locality of reference. The same should be done for the other two flops reach functions, which although have better performance, their performance degrade drastically with the use of multicores, as shown on Figure 8, bottom. It must be determined if this is due to a bandwidth bottleneck, in which case certain types of blocking may help.

4 Bottlenecks

Based on our performance analysis, the biggest performance increase of the current codes can come from more efficient use of the multicores. We described the bottleneck in subsection 3.3. Namely, the cores of a node are taken as separate MPI processes, which further increases the load (without need) of the shared between the cores memory bus. This causes, for example, extra copies (in an otherwise shared memory environment) and no overlap of communication and computation (locally). Not to have additional load on the multicores memory bus is important because our type of computation involves some sparse linear algebra (especially the 1st code) which are notorious for running much below the machine's peaks, especially for the case of multicore architectures.

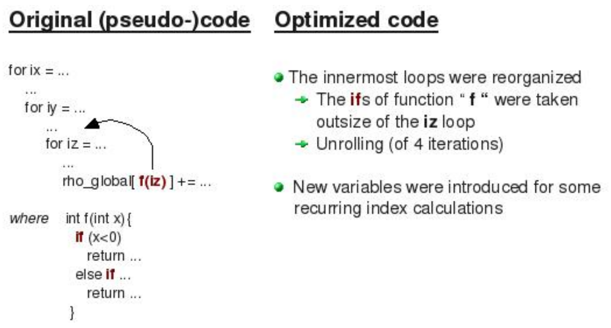
Another example on overloading the multicores' memory bus (which is happening in our current codes) are posting `MPI_Irecv` late, in particular after communication data has already started to arrive. This also results in extra copy as MPI would start putting the data in temporary buffers, and later copy it to the user specified buffers.

When looking for performance optimization opportunities, it is also important to keep in mind what are roughly the limits for improvement based on machine and algorithmic requirements. One can get close to machine peak performance only for operations of high enough ratio of Flops *vs* data needed. For example Level 3 BLAS can achieve it for large enough matrix size (e.g. approximately at least 200 on current architectures). Otherwise, in most cases, memory bandwidth and latencies are limits for the maximum performance. Jaguar, in particular, has quad-core Opterons at 2.1GHz with theoretical maximum performance of 8.4 GFlop/s per core (about 32 GFlop/s per quad-core node) and memory bandwidth of 10.6 GB/s (shared between the 4 cores). With these characteristics, if an application requires for example streaming (copy), one can expect about 10GB/s and 1 core will saturate the buss, dot product is $\approx$ 1GFlop/s (again one core saturates the bus), FFT is $\approx$ 0.7 GFlop/s for 2 cores and 1.3GFlop/s for 4 cores, random sparse are $\approx$ 0.035 GFlop/s for 2 cores and 0.052 GFlop/s for 4 cores, etc. The point here is that if certain performance is not satisfactory, we may have to look for ways to change the algorithm itself (some suggestions given below).

Here is a list of suggestions for performance improvement:

– Try some standard optimization techniques on the most compute intensive functions;
– Change the current all-MPI implementation to a multicore-aware implementation where MPI communications are performed only between nodes (and not within them);
– Try different strategies/patterns of intermixing communication and computation. For example, the current pattern in `get_new_rho` is to have a queue of 2 `MPI_Irecvs` (and corresponding `MPI_Isends`) where there is an `MPI_Wait` associated with the first `MPI_Irecv` of the queue, ensuring the data needed has been received, followed by the computation associated with that data. This pattern insures some overlapping of communication and computation but it is worth investigating larger sets of `MPI_Irecvs` combined for example with MPI_Waitsome. The idea is to first have enough `MPI_Irecvs` posted to avoid a case of data arriving (from some `MPI_Isend`) before a corresponding `MPI_Irecv` is posted (in which case there would be a copy overhead). Second, we want to start immediately the computation associated with the communication that has first completed;
– Consider changing the algorithms if performance is still not satisfactory.

Related to item one, we can give an example with function `DOxO`, which was the most time consuming function for most of the small runs used also as illustrations in this paper. We managed to accelerate it approximately 2.6× which brought about 28% overall improvement (as `DOxO` was running 29% of the total time). The techniques used here are given on Figure 9. The candidates for this type of optimization are determined. We note that the opportunity



Original (pseudo-)code

```
for ix = ...
   ...
   for iy = ...
      ...
      for iz = ...
         ...
         rho_global[ f(iz) ] += ...

where   int f(int x){
          if (x<0)
             return ...
          else if ...
             return ...
        }
```

Optimized code

- The innermost loops were reorganized
  - The ifs of function " f " were taken outsize of the iz loop
  - Unrolling (of 4 iterations)
- New variables were introduced for some recurring index calculations

Fig. 9. Code optimization.

for speedup here may be better for the second code, because it has more sparse operations. A difficulty is that there is not a single function to optimize - Figure 5 shows the profiles for the two codes as run on large problems of interest. Note that the first code has a single most time consuming function, but speedup here would most probably come from algorithmic innovations.

The optimizations related to items 2 and 3 are work in progress. Finally, for item 4, we consider for example certain new algorithms, designed for example to avoid/minimize communication [3], and in general new linear algebra developments for multicores and emerging hybrid architectures.

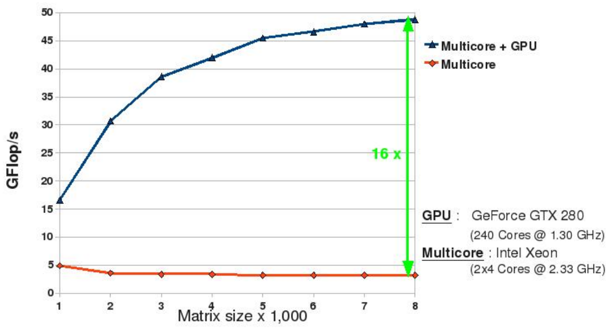


Fig. 10. Hybrid GPU-accelerated Hessenberg reduction in double precision.

For example, related to accelerating the **subspace diagonalization** problem, Figure 10 shows a performance acceleration of the reduction to upper Hessenberg form using hybrid GPU-accelerated computation - the improvement is 16× from the current implementation, obviously a bottleneck.

5 Conclusions

We profiled and analyzed two petascale quantum simulation tools for nanotechnology applications. We used different tools to help in understanding their performance on Teraflop leadership platforms. We identified bottlenecks and gave suggestions for their removal. The results so far indicate that the main steps that we have followed (and described) can be viewed/used as a methodology to not only easily produce and analyze performance data, but also to aid the development of algorithms, and in particular petascale quantum simulation tools, that effectively use the underlying hardware.

Acknowledgments. This work was supported by the U.S. National Science Foundation under contract # 0749293. We used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

1. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, *A portable programming interface for performance evaluation on modern processors*, The International Journal of High Performance Computing Applications **14** (2000), 189–204.
2. Shirley Browne, Christine Deane, George Ho, and Philip Mucci, *Papi: A portable interface to hardware performance counters*, (June 1999).
3. James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou, *Communication-avoiding parallel and sequential qr factorizations*, CoRR **abs/0806.2159** (2008).
4. J.-L. Fattebert and J. Bernholc, *Towards grid-based $o(n)$ density-functional theory methods: Optimized nonorthogonal orbitals and multigrid acceleration*, Phys. Rev. B **62** (2000), no. 3, 1713–1722.
5. Miroslav Hodak, Shuchun Wang, Wenchang Lu, and J. Bernholc, *Implementation of ultrasoft pseudopotentials in large-scale grid-based electronic structure calculations*, Physical Review B (Condensed Matter and Materials Physics) **76** (2007), no. 8, 085108.
6. Sameer S. Shende and Allen D. Malony, *The tau parallel performance system*, Int. J. High Perform. Comput. Appl. **20** (2006), no. 2, 287–311.
7. F. Wolf and B. Mohr, *Kojak - a tool set for automatic performance analysis of parallel applications*, Proc. of the European Conference on Parallel Computing (Euro-Par) (Klagenfurt, Austria), Lecture Notes in Computer Science, vol. 2790, Springer, August 2003, Demonstrations of Parallel and Distributed Computing, pp. 1301–1304.
8. Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider, *Toward scalable performance visualization with Jumpshot*, High Performance Computing Applications **13** (1999), no. 2, 277–288.