Performance Evaluation for Petascale Quantum Simulation Tools

Stan Tomov
Innovative Computing Laboratory (ICL)
The University of Tennessee

joint work with *Wenchang Lu*[1,2], *Jerzy Bernholc*[1,2], *Shirley Moore*[3], and *Jack Dongarra*[2,3]
(NCSU[1], ORNL[2], UTK[3])

CUG 09: **Compute the Future**, Atlanta GA
May 4th – May 7th, 2009

Outline

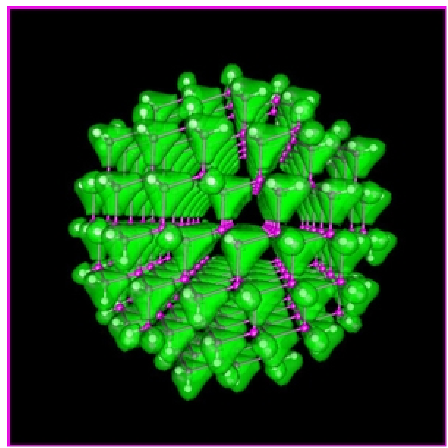
- Background
  - Simulation of nano materials and devices
  - Challenges of future architectures
- Electronic structure calculations
- Performance evaluation
- Performance analysis
- Bottlenecks and ideas for their removal
- Conclusions

Electronic properties of nano-structures

- Semiconductor Quantum dots (QDs)
  - **Tiny** crystals ranging from a few hundred to few thousand atoms in size; made by humans

  At these small sizes electronic properties critically depend on **shape** and **size**
  $\Rightarrow$ electronic properties can be tuned
  $\Rightarrow$ enables remarkable applications

  The dependence is quantum mechanical in nature and can be modelled
  - can not be done on macroscopic scales
  - has to be at **atomic and subatomic level (nanoscale)**
- Quantum wires (QWs) and devices
  - their conducting properties are affected by build-in nano-materials


Total electron charge density of a quantum dot of gallium arsenide, containing just 465 atoms.

Quantum dots of the same material but different sizes have different band gaps and emit different colors

Nano Materials Simulations

predictive power

- Many-body quantum mechanical (QM) first-principles approaches (e.g. Quantum Monte Carlo) 30-200 atoms
- Single particle first-principles (Density Functional Theory) 10^3
- Empirical and Semiempirical methods 10^6
- Continuum methods 10^7

atoms

- **Method classification based on:** Use of empirically or experimentally derived results
  YES ⇒ empirical or semi-empirical methods
  NO ⇒ **ab initio** (very accurate; most predictive power; but scales as $O(N^{3..7})$)
- Major petascale computing challenges:
  - Algorithms with reduced scaling; architecture aware
  - Highly parallelizable (100s of 1,000s of cores)
    - typical basis functions here (plane-wave basis) have global support

Challenges of Future Architectures

- Increase in parallelism
  - Multicores, GPUs, hybrid architectures, etc
- Increase in communication cost (*vs* computation)
  - Gap between processor and memory speed continue to grow (exponentially) [e.g. processor speed improves 59%, memory bandwidth 23%, latency 5.5%]

Approach

- **Basis selection: Plane-waves, grid functions, or Gaussian orbitals, etc.**
- Plane-waves: $\psi_{nk}(r) = \sum_{g,|g|<E_{cut}} C_g^n(k) e^{i(g+k).r}$
  - Good approximation properties
  - Can be preconditioned easily (and efficiently) as the kinetic energy (the laplacian) is diagonal in Fourier space, the potential is diagonal in real space
  - Usually codes are in Fourier space and go back and forth to real with FFTs
  - Concern may be scalability of FFT on 100s of 1,000s of processors as it requires global communication
- Grid functions: e.g. finite elements, grids, or wavelets
  - Domain decomposition techniques can guarantee scalability for large enough problems
  - Interesting as they enable algebraically based preconditioners as well
  - Including multigrid/multiscale
    - **e.g. real-space multigrid methods (RMG) by J. Bernholc et al (NCSU)**

Goal of this work

- **Performance evaluation** of petascale quantum simulation tools for nanotechnology applications
  - Based on existing real-space multigrid method (RMG)
  - In-depth understanding of their performance on Teraflop leadership platforms
  - With the help of tools such as TAU, PAPI, Jumpshot, KOJAK, etc
- **Identify performance bottlenecks** and ways/ideas for their removal
- **Aid the development of algorithms**, and in particular petascale quantum simulations tools, that effectively use the underlying hardware

Software/Hardware Environment

- We consider 2 methodologies [implemented so far in our codes]
  - Global grid method
    - Wave functions are represented in the real space uniform grids
    - Most time consuming is orthogonalization and subspace diagonalization
    - Massively parallel, good flops performance, but scales in $O(N^3)$ with system size
  - Optimally localized orbital method
    - Scales in nearly O(N) but has computational challenges

- Hardware: we consider **Jaguar**, a Cray XT4 system at ORNL
  - Based on quad-core 2.1 GHz AMD Opteron processors

Performance evaluation

- Techniques that we found most useful
- Profiling [using TAU with PAPI]
  - To get familiar with code structure
  - To get performance profiles
  - To identify possible performance bottlenecks
- Tracing [using TAU]
  - To determine exact locations and cause of bottlenecks

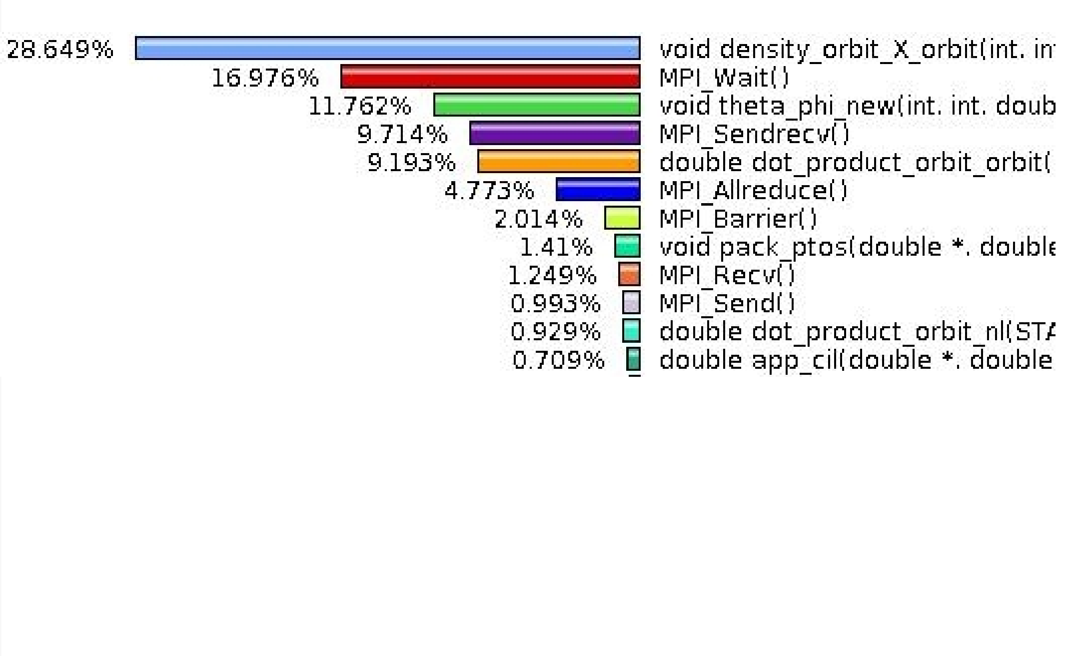
Profiling

- Getting familiar with code structure [by generating callpath data]

GET_TIME_OF_DAY				
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

Profiling

- Performance profiles
  [load balance, what to optimize, etc]



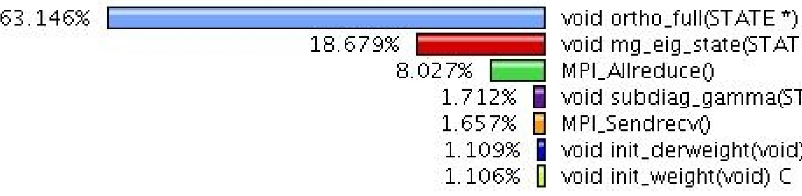
Metric: Time
Value: Exclusive

std. dev.
mean
n.c.t 0.0.0
n.c.t 1.0.0
n.c.t 2.0.0
n.c.t 3.0.0
n.c.t 4.0.0
n.c.t 5.0.0
n.c.t 6.0.0
n.c.t 7.0.0
n.c.t 8.0.0
n.c.t 9.0.0
n.c.t 10.0.0
n.c.t 11.0.0
n.c.t 12.0.0
n.c.t 13.0.0
n.c.t 14.0.0
n.c.t 15.0.0
n.c.t 16.0.0
n.c.t 17.0.0
n.c.t 18.0.0
n.c.t 19.0.0
n.c.t 20.0.0
n.c.t 21.0.0
n.c.t 22.0.0
n.c.t 23.0.0
n.c.t 24.0.0
n.c.t 25.0.0
n.c.t 26.0.0
n.c.t 27.0.0
n.c.t 28.0.0
n.c.t 29.0.0
n.c.t 30.0.0
n.c.t 31.0.0
n.c.t 32.0.0
n.c.t 33.0.0
n.c.t 34.0.0
n.c.t 35.0.0
n.c.t 36.0.0
n.c.t 37.0.0
n.c.t 38.0.0
n.c.t 39.0.0
n.c.t 40.0.0
n.c.t 41.0.0
n.c.t 42.0.0
n.c.t 43.0.0
n.c.t 44.0.0
n.c.t 45.0.0
n.c.t 46.0.0
n.c.t 47.0.0
n.c.t 48.0.0
n.c.t 49.0.0
n.c.t 50.0.0
n.c.t 51.0.0
n.c.t 52.0.0
n.c.t 53.0.0
n.c.t 54.0.0
n.c.t 55.0.0
n.c.t 56.0.0
n.c.t 57.0.0
n.c.t 58.0.0
n.c.t 59.0.0
n.c.t 60.0.0
n.c.t 61.0.0
n.c.t 62.0.0
n.c.t 63.0.0

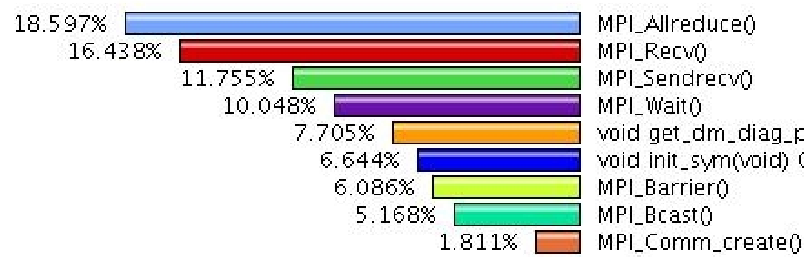
Profiling

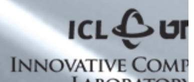
- Performance evaluation results
  [PAPI counters; example with PAPI_FP_INS; right]
  [Profiles for the 2 codes on large problems;
  1024 cores; below]
  [1st code about 6 x faster; 2nd more sparse op.]

Metric: GET_TIME_OF_DAY
Value: Exclusive percent



Metric: LINUX_TIMERS
Value: Exclusive percent





Metric: PAPI_FP_INS
Value: Exclusive

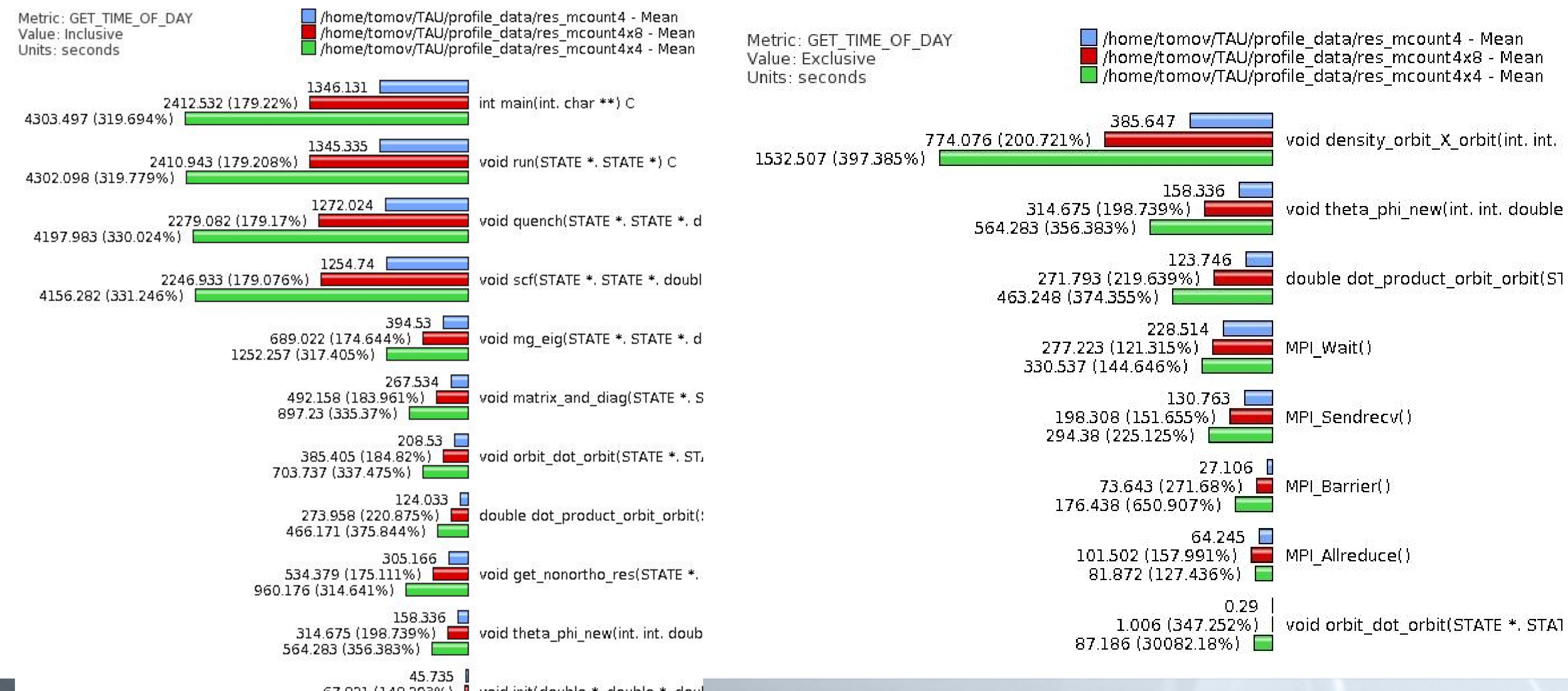
std. dev.
mean
n.c.t 0.0.0
n.c.t 1.0.0
n.c.t 2.0.0
n.c.t 3.0.0
n.c.t 4.0.0
n.c.t 5.0.0
n.c.t 6.0.0
n.c.t 7.0.0
n.c.t 8.0.0
n.c.t 9.0.0
n.c.t 10.0.0
n.c.t 11.0.0
n.c.t 12.0.0
n.c.t 13.0.0
n.c.t 14.0.0
n.c.t 15.0.0
n.c.t 16.0.0
n.c.t 17.0.0
n.c.t 18.0.0
n.c.t 19.0.0
n.c.t 20.0.0
n.c.t 21.0.0
n.c.t 22.0.0
n.c.t 23.0.0
n.c.t 24.0.0
n.c.t 25.0.0
n.c.t 26.0.0
n.c.t 27.0.0
n.c.t 28.0.0
n.c.t 29.0.0
n.c.t 30.0.0
n.c.t 31.0.0
n.c.t 32.0.0
n.c.t 33.0.0
n.c.t 34.0.0
n.c.t 35.0.0
n.c.t 36.0.0
n.c.t 37.0.0
n.c.t 38.0.0
n.c.t 39.0.0
n.c.t 40.0.0
n.c.t 41.0.0
n.c.t 42.0.0
n.c.t 43.0.0
n.c.t 44.0.0
n.c.t 45.0.0
n.c.t 46.0.0
n.c.t 47.0.0
n.c.t 48.0.0
n.c.t 49.0.0
n.c.t 50.0.0
n.c.t 51.0.0
n.c.t 52.0.0
n.c.t 53.0.0
n.c.t 54.0.0
n.c.t 55.0.0
n.c.t 56.0.0
n.c.t 57.0.0
n.c.t 58.0.0
n.c.t 59.0.0
n.c.t 60.0.0
n.c.t 61.0.0
n.c.t 62.0.0
n.c.t 63.0.0

Performance analysis

- Tracing
  - To determine exact locations and causes of bottlenecks
  - TAU to generate trace files and analyze them with Jumpshot and tools like KOJAK
  - Codes well written
    - Blocked communications, asynchronous, intermixed with computation
  - Domain decomposition guarantees weak scalability
    - We have to concentrate on efficient use of multicores within a node
  - We found early posting of MPI_Irecv will benefit our codes
  - We found useful to compare traces of different runs – to study effects of code changes
  - Generate profile-type statistics for various parts of the codes

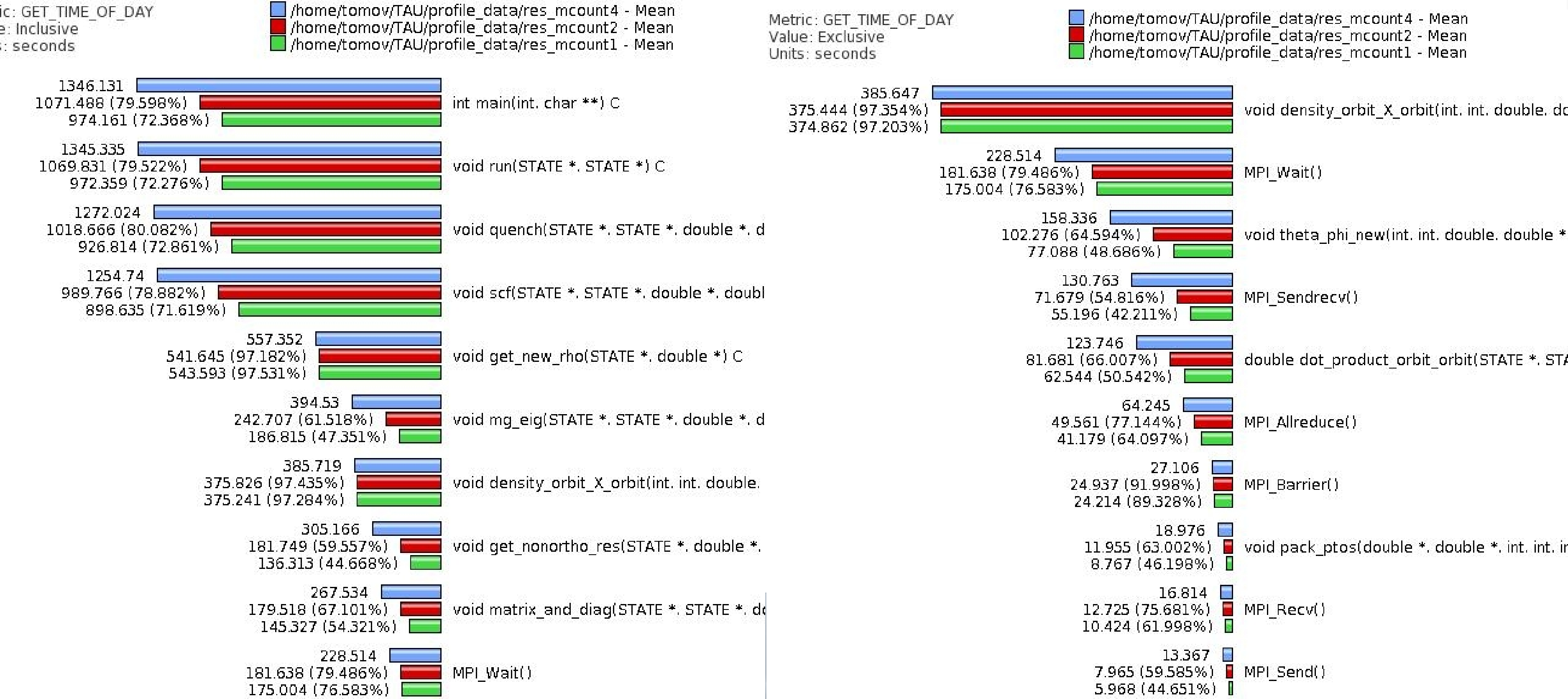
Performance analysis

- Scalability
  - Studied both strong and weak [example on strong scalability]



Performance analysis

- Multicore use
  - Measurements in different hardware configurations: runs using 4, 2, and single core of the quad-core nodes [number of cores to be the same]



Performance using single, 2, and 4 cores

Metric: PAPI_FP_INS / GET_TIME_OF_DAY
Value: Inclusive
Units: Derived metric shown in microseconds format

- /home/tomov/TAU/profile_data/res_mcount4 - Mean
- /home/tomov/TAU/profile_data/res_mcount2 - Mean
- /home/tomov/TAU/profile_data/res_mcount1 - Mean

Function	res_mcount4 - Mean	res_mcount2 - Mean	res_mcount1 - Mean
int main(int. char **) C	110.541	138.873 (125.63%)	152.754 (138.188%)
void run(STATE *. STATE *) C	110.575	139.048 (125.749%)	152.993 (138.361%)
void quench(STATE *. STATE *. double	97.778	122.103 (124.878%)	134.204 (137.254%)
void scf(STATE *. STATE *. double *. c	97.807	123.997 (126.778%)	136.572 (139.634%)
void get_new_rho(STATE *. double *)	37.774	38.871 (102.904%)	38.732 (102.536%)
void mg_eig(STATE *. STATE *. double	119.675	194.549 (162.564%)	252.751 (211.198%)
void density_orbit_X_orbit(int. int. dou	53.234	54.568 (102.505%)	54.642 (102.645%)
void get_nonortho_res(STATE *. doub	89.123	149.715 (167.986%)	199.702 (224.075%)
void matrix_and_diag(STATE *. STATE	149.569	222.828 (148.981%)	275.185 (183.986%)

Metric: PAPI_FP_INS / GET_TIME_OF_DAY
Value: Exclusive
Units: Derived metric shown in microseconds format

- /home/tomov/TAU/profile_data/res_mcount4 - Mean
- /home/tomov/TAU/profile_data/res_mcount2 - Mean
- /home/tomov/TAU/profile_data/res_mcount1 - Mean

Function	res_mcount4 - Mean	res_mcount2 - Mean	res_mcount1 - Mean
void density_orbit_X_orbit(int. int. dou	53.244	54.624 (102.594%)	54.699 (102.733%)
MPI_Wait()	3.7E-4	5.9E-4 (159.321%)	6.1E-4 (163.679%)
void theta_phi_new(int. int. double. dc	178.324	268.518 (150.578%)	352.846 (197.868%)
double dot_product_orbit_orbit(STATE	249.025	367.389 (147.531%)	474.653 (190.604%)
MPI_Sendrecv()	0.007	0.015 (210.042%)	0.019 (271.58%)
MPI_Allreduce()	1.712	2.193 (128.079%)	2.649 (154.688%)
MPI_Barrier()	2.7E-5	5.5E-5 (204.461%)	5.5E-5 (204.545%)
MPI_Recv()	0.044	0.047 (107.498%)	0.051 (117.176%)
void pack_ptos(double *. double *. in	2.059	3.311 (160.836%)	4.513 (219.196%)

Bottlenecks

- Maximum performance
  - Jaguar has quad-core Opterons 2.1 GHz
    - Theoretical maximum : 8.4 GFlop/s per core (~ 32 GFlop/s per quad-core)
    - Memory bandwidth : 10.6 GB/s (shared between the 4 cores)
  - Close to peak – only for operations of high enough ratio of
    Flops *vs* data needed
    - e.g. Level 3 BLAS for large enough N (~200)
  - Otherwise, in most cases, memory bandwidth and latencies are limits for the maximum performance
    - e.g. stream (copy) is ~ 10 GB/s (1 core enough to saturate the bus)
    - dot product ~ 1 GFlop/s (16 Bytes for 2 operations; 1 core saturates bus)
    - FFT ~ 0.7 GFlop/s (2 cores), 1.3 GFlop/s (4 cores)
    - Random sparse ~ 0.035 GFlop/s (2 cores), 0.052 GFlop/s (4 cores)

Bottlenecks

- A list of suggestions for performance improvements
  - Try some standard optimization techniques on the most compute intensive functions
  - Change the current all-MPI implementation to a multicore-aware implementation where communications are performed only between nodes
  - Try different strategies/patterns of intermixing communication and computation (e.g. early MPI_Irecvs)
  - Consider changing the algorithms if performance is still not satisfactory

Bottlenecks removal

- Example of standard performance optimization techniques
  [e.g. DoxO, runs 29% of time, accelerated **2.6 x**; overall brings 28% acceleration]

Original (pseudo-)code

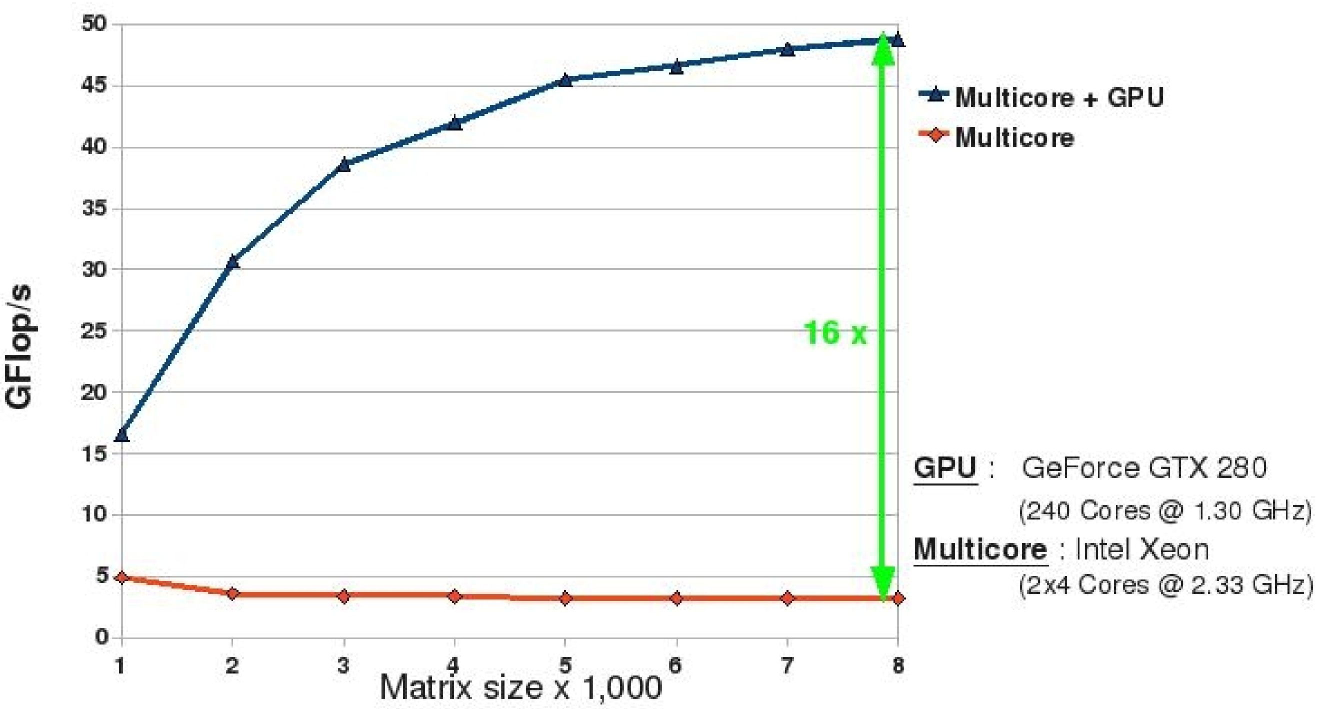
```
for ix = ...
   ...
   for iy = ...
      ...
      for iz = ...
         ...
         rho_global[ f(iz) ] += ...

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

Optimized code

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

Bottlenecks removal

- **New algorithms** – advances from linear algebra for multicore and emerging hybrid architectures
  [e.g. hybrid Hessenberg reduction in double precision; accelerated 16x; related to the subspace diagonalization bottleneck]



ICL UT INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY of TENNESSEE Computer Science Department

Conclusions

- We profiled and analyzed 2 petascale quantum simulation tools for nanotechnology applications
- We used different tools to help in understanding the 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.