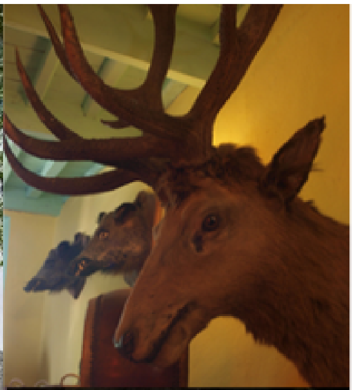
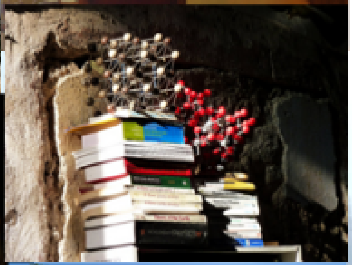
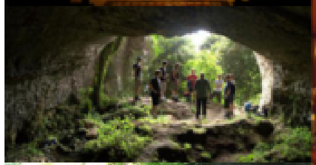
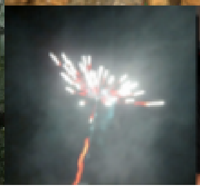
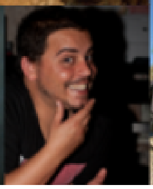





The Towler Institute

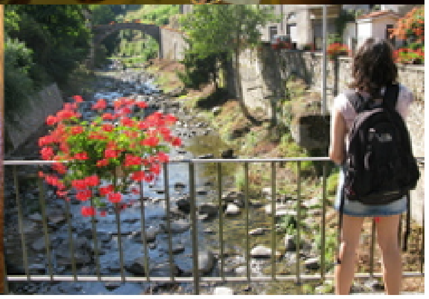
2012 Conference Programme

Quantum Monte Carlo in the Apuan Alps VII

Vallico Sotto, Tuscany, Italy : 28th July - 4th August 2012

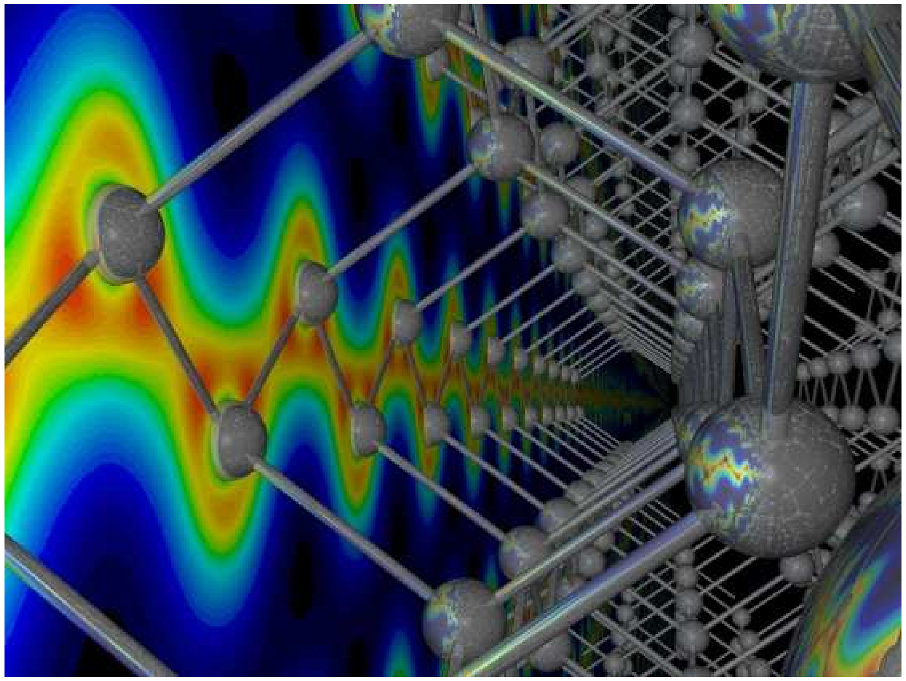
www.vallico.net/tti/tti.html

email : mdt26 at cam.ac.uk



Massively-parallel QMC calculations: CPUs, GPUs and DMC molecular dynamics

Quantum Monte Carlo in the Apuan Alps VII, July/August 2012



Mike Towler

TCM Group, Cavendish Laboratory, University of Cambridge and UCL

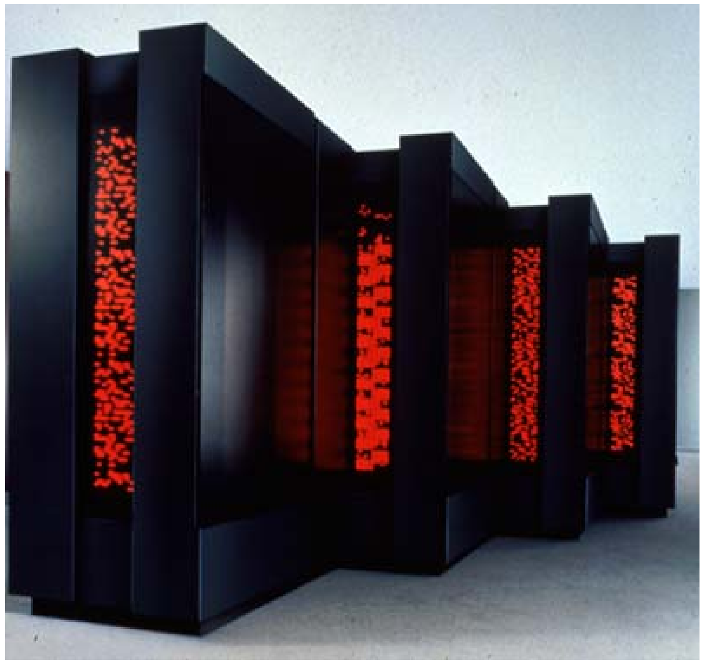
QMC web page: `www.tcm.phy.cam.ac.uk/∼mdt26/casino.html`

Email: `mdt26@cam.ac.uk`

One partial success and two failures

- Can we exploit modern petascale hardware with hundreds of thousands or millions of processors to do DMC calculations?
- The latest generation of machines are mostly being built with GPU accelerators - is this going to help?
- I was asked to implement Grossman-Mitas molecular dynamics into CASINO and so I did. Was it worth it?

Parallel computing



From the mid-1980s until 2004 computers got faster because of *frequency scaling* (more GHz). However, faster chips consume more power, and ever since power consumption (and consequently heat generation) became a significant concern, parallel computing has become the dominant paradigm in computer architecture, particularly with the advent of *multicore processors* - now present even in most of your laptops.

- We do not pretend that QMC is the cheapest technique in the world. Thus the study of anything other than simple systems inevitably requires the use of parallel computers.
- The biggest machines in the world are now approaching a million processors. Some techniques (such as DFT) have difficulty exploiting more than a thousand processors because of the large amount of interprocessor communication required. This leads to the important question:

How does QMC scale with the number of processors?

And consequently, *how many processors can we successfully exploit?*

Parallel computing

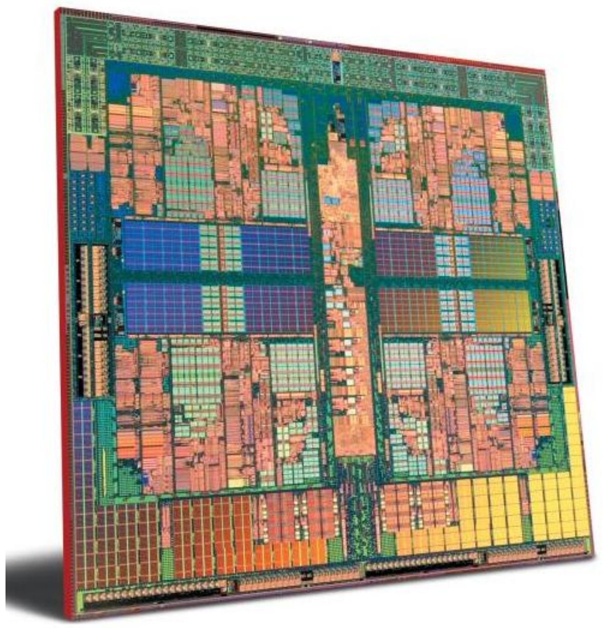
From the mid-1980s until 2004 computers got faster because of *frequency scaling* (more GHz). However, faster chips consume more power, and ever since power consumption (and consequently heat generation) became a significant concern, parallel computing has become the dominant paradigm in computer architecture, particularly with the advent of *multicore processors* - now present even in most of your laptops.

- We do not pretend that QMC is the cheapest technique in the world. Thus the study of anything other than simple systems inevitably requires the use of parallel computers.
- The biggest machines in the world are now approaching a million processors. Some techniques (such as DFT) have difficulty exploiting more than a thousand processors because of the large amount of interprocessor communication required. This leads to the important question:

How does QMC scale with the number of processors?

And consequently, *how many processors can we successfully exploit?*

Increasing complexity and new terminology: CPUs


AMD quad-core processor

In the old days (when we originally wrote CASINO) parallel machines were quite ‘simple’ things. That is, each computing unit (usually referred to as a ‘node’ or a ‘processor’) ran a separate copy of the program, and each had its own local memory.

Nowadays, things are more complex. A computer may have multiple nodes. And those nodes contain multiple sockets. And the processors in those sockets contain multiple (CPU) cores. The memory architecture is also more complex.

Node: a printed circuit board of some type, manufactured with multiple empty *sockets* into which one may plug one of a family of processors.

Processor: this is the object manufactured e.g. by Intel or AMD. Generally there are ‘families’ of processors whose members have differing core counts, a wide range of frequencies and different memory cache structures. One cannot buy anything smaller than a processor.

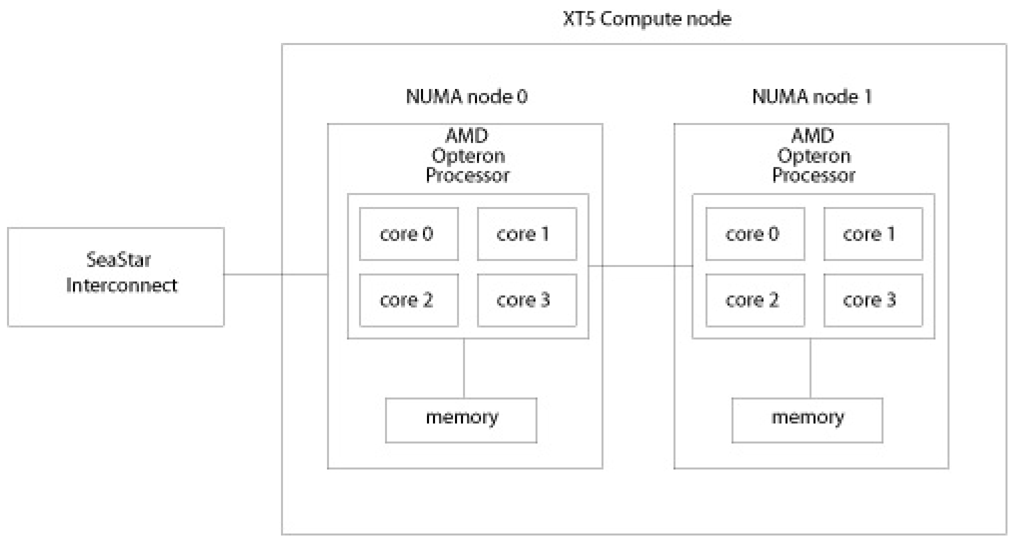
Core: the cores within the processor perform the actual mathematical computations. A core can do a certain number (typically 4) of FLOPs or FLoating-point OPerations every time its internal clock ticks. These clock ticks are called cycles and measured in Hertz (Hz). Thus a 2.5-GHz processor ticking 2.5 billion times per second and capable of performing 4 FLOPs each tick is rated with a theoretical performance of 10 billion FLOPs per second or 10 GFLOPS.

Increasing complexity and new terminology: memory

In complex modern systems we also need to understand how the memory is accessed.

Distributed memory : each processor has its own local private memory.

Shared memory : memory that may be simultaneously accessed by multiple cores with an intent to provide communication among them or avoid redundant copies.



Modern machines containing 'compute nodes' such as this Cray XT5 often have a *non-uniform memory architecture* ('NUMA'). That is a processor can access its own local memory faster than non-local memory, that is, memory local to another processor or memory shared between processors.

Typically we might use shared memory on a 'compute node' which is simultaneously and quickly accessible to all processor cores that are plugged into it. Data is sent between nodes using explicit MPI commands and - in this case - the slower SeaStar Interconnect.

With CASINO, shared memory allows one to treat much bigger systems. A particular problem occurs when using a 'blip' (B-spline) basis set to represent the orbitals; the blip coefficients for a large systems can take up many Gb of memory (and this may exceed the amount locally available to each core). Thus we may have e.g. a node containing two 6-core processors i.e. 12 cores with a single copy of the blip coefficients in the shared memory available to all cores on that node.

State of the art: petascale computers

- A ‘petascale’ system is able to make arithmetic calculations at a sustained rate in excess of a sizzling *1,000-trillion operations per second* (a ‘petaflop’ per second).
- The first computer ever to reach the petascale milestone (in 2008) was the *Roadrunner* at Los Alamos shown above. It contained 122400 cores achieving a peak performance of 1.026 petaflops/s.
- One may consult the ‘Top 500 Supercomputers’ list at `www.top500.org` to see who and what is currently winning. Current fastest (July 2012) is *Sequoia* at Lawrence Livermore (IBM Blue Gene/Q, 1572864 processors, 20.13 petaflops/s).

A usable example: Jaguar

Jaguar is a Cray XK6 machine at Oak Ridge National Laboratory in Tennessee, USA. It has a peak performance of around 2.6 petaflops/s, and has 299008 AMD Opteron processor cores, making it the sixth-fastest computer in the world (July 2012). Like all the best computers, it runs Linux.

It is made of 18,688 XK6 compute nodes. Each such node contains one 16-core AMD Opteron processors and 32 GB of memory. Jaguar was the fastest computer in the world until October 2010, and is currently being upgraded/transformed into ‘Titan’ which will have a NVIDIA Kepler GPU accelerator added to most of the nodes. The upgraded machine will have a performance ‘in excess of 20 petaflops/s’ and will once again be competing for the top spot.

```
ssh -X mdt26@jaguarpf.ccs.ornl.gov
cd CASINO ; make Shm
runqmc -p 299008 --shmem=16 --walltime=6h30m
```

How is CASINO parallelized?

CASINO's parallel capabilities are implemented largely with MPI which allows communication between all cores on the system. A second level of parallelization useful under certain circumstances (usually when the number of cores is greater than the number of configs/walkers) is implemented using *OpenMP* constructs, which functions over small groups of e.g. 2-4 cores.

MPI (Message Passing Interface) is a language-independent API (application programming interface) specification that allows processes to communicate with one another by sending and receiving messages. It is a *de facto* standard for parallel programs running on computer clusters and supercomputers, where the cost of accessing non-local memory is high.

Example: `call MPI_Reduce([input_data], [output_result], [input_count], [input_datatype],[reduce_function], ROOT, [User_communication_set], [error_code])`

By setting the 'reduce function' to 'sum', such a command may be used - for example - to sum a vector over all cores, which is required when computing averages.

OpenMP is an API that supports shared-memory multiprocessing. It implements *multithreading*, where the master 'thread' (a series of instructions executed consecutively) forks a specified number of slave threads and a task is divided among them. The threads run concurrently, with the runtime environment allocating threads to different cores. The section of code meant to run in parallel is marked with a preprocessor directive that causes the threads to form before the section is executed:

```
!$OMP parallel
...
!$OMP end_parallel
```

Why DMC does not scale linearly with the number of cores

- In DMC, config population initially divided evenly between cores. Algorithm not perfectly parallel since population fluctuates on each core; iteration time determined by the core with the largest population. Necessary to even up config population between cores *occasionally* ('load balancing').
- The best definition of '*occasionally*' turns out to be 'after every move', since this minimizes the time taken by the core with the largest number of configurations to finish propagating its excess population.
- From the CASINO perspective, what is a 'config' and how big is it? It is a list of electron positions, together with some associated wave function- and energy-related quantites. For the relatively big systems of interest, a config might be from 1-10kb in size, and up to around five of them might need to be sent from one processor to another. Thus messages can be up to 50kb in size (though usually they are much smaller).
- Transferring configs between cores is thus likely to be time-consuming, particular for large numbers of cores. Thus there is a trade-off between balancing the load on each processor and reducing the number of config transfers.

Formal parallel efficiency

- Cost of propagating all configs in one iteration : $T_{\rm CPU} \approx A\frac{N^\alpha N_C}{P}$

  Here P is number of CPU cores, N_C is number of configs, N is number of particles, and $\alpha = 1$ (localized orbs and basis) or 2 (delocalized orbs, local basis). Add 1 to α for trivial orbs/large systems where determinant update dominates.

- Cost of load balancing : $T_{\rm comm} \approx B\sqrt{N_C P N^3}$

  Require $T_{\rm CPU} \gg T_{\rm comm}$ as DMC algorithm perfectly parallel in this limit.

- Ratio of load balancing to config propagation time :

$$\frac{T_{\rm comm}}{T_{\rm CPU}} = \frac{A}{B}\frac{P^{\frac{3}{2}}N^{\frac{3}{2}-\alpha}}{\sqrt{N_C}}$$

- For $\alpha > 3/2$ (which is true unless time to evaluate localized orbitals dominates), the fraction of time spent on comms falls off with system size.

- By increasing N_C fraction of time spent on comms can be made arbitrarily small, but, in practice number of configs per core limited by available MEMORY.

- Memory issue is the main problem for very large systems or very large number of cores, particularly when using a blip basis set.

Obvious ways to improve load balancing in CASINO

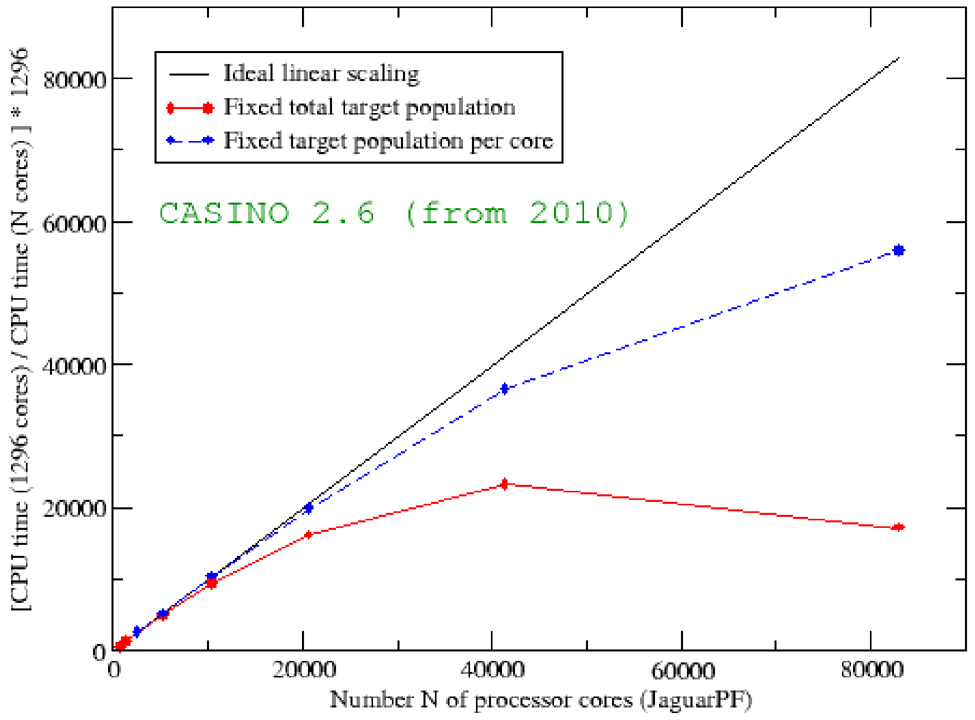
- Increase number of configs per core (without blowing the memory).
- Use weighted DMC (**lwdmc** keyword) to reduce branching (with the default weight limits of 0.5 and 2.0) and disable transfer or large arrays (such as inverse Slater matrices) between cores by using the **small_transfer** keyword.

Obvious ways to avoid blowing the memory in CASINO

- On architectures made up of shared memory nodes with multiple cores: allocate blips on these nodes instead of on each core (`make Shm` to enable this, then `runqmc --shmem`).
- Use OpenMP - extra level of parallelization for loops scaling with number of electrons. Define 'pools' of small numbers of cores (typically 2-4). Parallelisation over configs maintained over pools, but inside each pool work for each config is parallelized by splitting the orbitals over pools (this reduces necessary memory per core). Then, each core in the pool only evaluates the value of a subset of orbitals. That done, all cores within the pool communicate to construct the Slater determinants, which are evaluated again in parallel using the cores in the pool. Gives $\sim$1.5$\times$ speedup on 2 cores, $\sim$2$\times$ speedup on 4 cores.

  To use with CASINO, compile with '`make OpenMP`', then run with e.g. on a 4-core machine '`runqmc --nproc=2 --tpp=2`' where `tpp` means 'threads per process'. Can also run with both Shm and OpenMP (`make OpenmpShm` etc.).
- Use **single_precision_blips** keyword, the blip coefficients using single precision real/complex numbers, which will halve the memory required.

How did CASINO scale with the old standard DMC algorithm?



Scaled ratio of CPU times in DMC statistics accumulation for various numbers of cores on Jaguar using the September 2010 version of CASINO 2.6. System: one H_2O molecule adsorbed on a 2D-periodic graphene sheet containing fifty C atoms per cell. For comparative purposes 'ideal linear scaling' (halving of CPU time for double the number of cores) is shown by the solid black line. Both blue and red lines show results for fixed sample size i.e. number of configs $\times$ number of moves [fixed problem size = 'strong scaling']. However, blue line has fixed target population of 100 configs per core (with an appropriately varying number of moves). Red line has fixed target population of 486000 (and constant number of moves) i.e. the number of configs per core falls with increasing number of cores (from 750 to around 5).

New tricks to effectively reduce T_{comm} to zero

Despite the earlier formal analysis, I discovered last year that with a few tricks one can effectively eliminate all overhead due to config transfers, and hence hugely improve the scaling (this is described in *Petascale computing opens new vistas for quantum Monte Carlo'*, by Mike Gillan, me and Dario Alfè, Psi-k Newsletter 'Scientific Highlight of the Month' Feb 2011).

The new algorithm involved:

(1) Analysis and modification of the procedure for deciding which configs to send between which pairs of cores when doing load balancing (the original CASINO algorithm for this originally scaled linearly with the number of cores – when you need it to be constant – a problem not included in our formal analyses!).

(2) The use of *asynchronous, non-blocking* MPI communications.

- To send a message from one processor to another, one normally calls blocking `MPI_SEND` and `MPI_RECV` routines on a pair of communicating cores. 'Blocking' means that all other work will halt until the transfer completes.
- However, one may also use *non-blocking* MPI calls, which allow cores to continue doing computations while communication with another core is still pending. On calling the non-blocking `MPI_ISEND` routine, for example, the function will return immediately, usually before the data has finished being sent.

Decisions about config transfers: the redistribution problem

- At the end of every move we have a vector (of length equal to the number of cores) containing the current population of configs on each core.
- Relative to a 'target' population, some cores will have an excess of configs, some will have the right amount, and some will have a deficit.
- The problem is to arrange for a series of transfers between pairs of cores in the most efficient way such that each core has as close to the target population as possible. Here 'efficient' means the total number of necessary transfers and the size of those transfers is to be minimized.

OLD ALGORITHM: Requires repeated operations on the entire population vector, asking things like 'what is the location of the current largest element?' [Fortran: `maxloc(popvector)`]. This scales linearly with the number of cores, and if you're asking to find the largest element of a vector of length 1 million and you do it a million times it starts to take some serious time. Any benefit from obtaining the optimum list of transfers is swamped by the process of finding that optimum list.

MORAL: *the algorithm is perfectly reasonable for a routine written in the years when no-one could run on more than 512 cores; however, such things can come back and bite you in the petascale era.*

SOLUTION

- Partition the cores into 'redist groups' of default size 500 and contemplate transfers only within these groups. If e.g. one core has a deficit of 1,2,3,4. . . configs, then in a group of that size it is highly likely that some other core will have a surfeit of 1,2,3,4.. configs, etc. Thus efficiency in config transfers will hardly be affected by not considering the full population vector.
- To avoid imbalances developing in the group populations, the list of cores that belong to each group is changed at every iteration ('`redist_shuffle`').

Non-blocking asynchronous communication

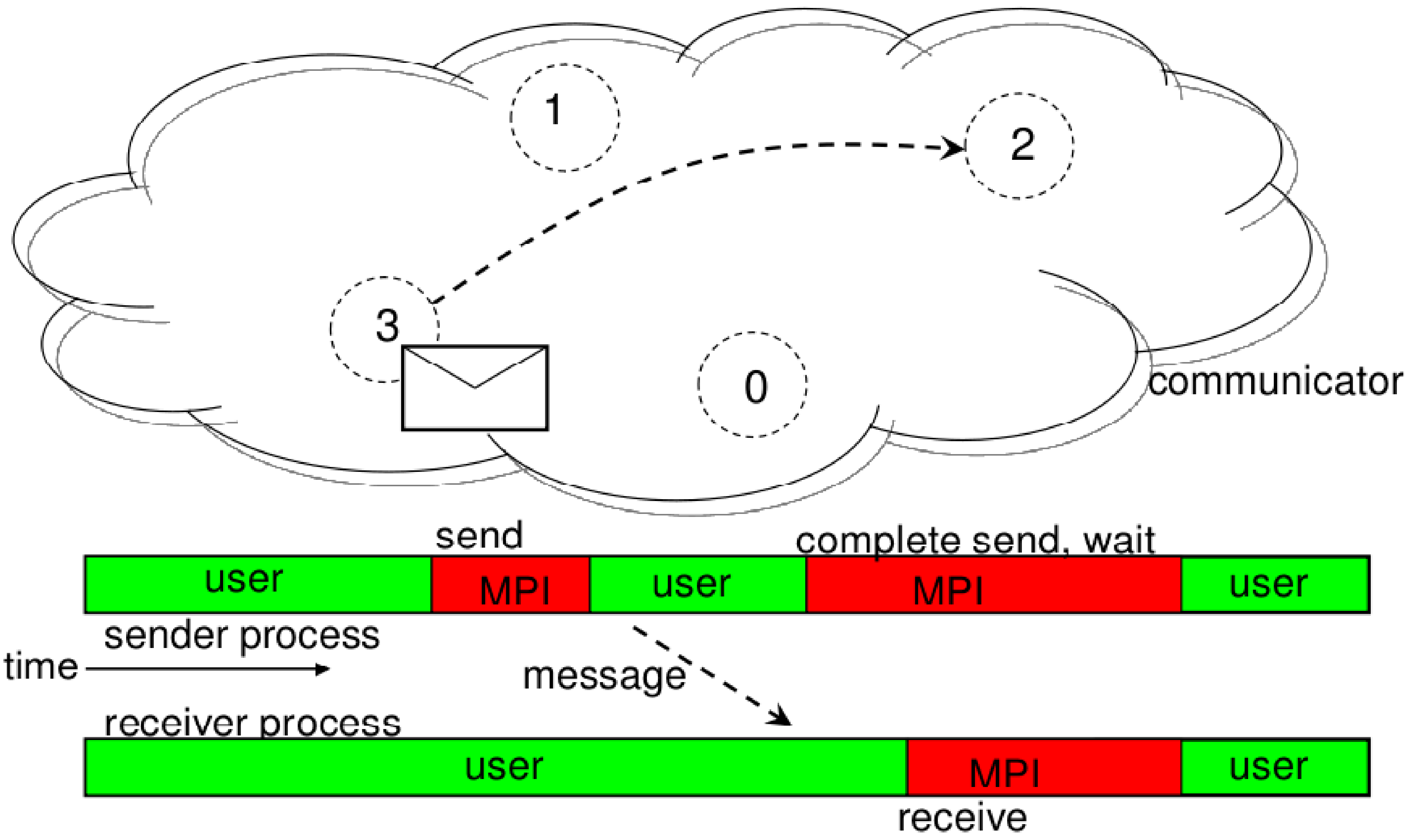
A communication call is said to be non-blocking if it may return before the operation completes (a *local* concept on either sender or receiver). A communication is said to be asynchronous if its execution proceeds at the same time as the execution of the program (a *non-local* concept).

Mode	Command	Notes	synchronous?
synchronous send	MPI_SSEND	Message goes directly to receiver. Only completes when receive begins.	synchronous
buffered send	MPI_BSEND	Message copied to a 'buffer'. Always completes regardless of receiver.	asynchronous
standard send	MPI_SEND	Either synchronous or buffered	both/hybrid
ready send	MPI_RSEND	Assumes the receiver is ready. Always completes regardless.	neither
receive	MPI_RECV	Completes when a message has arrived	

MPI also provides *non-blocking* send (MPI_ISEND) and receive (MPI_IRECV) routines. They return immediately, at the cost of you not being allowed to modify the sent vector/receiving vector until you execute a later MPI_TEST or MPI_WAIT call (or MPI_TESTALL/MPI_WAITALL for multiple communications) to check completion. In the meantime, the code can do some other work.

- Non-blocking routines allow separation of initiation and completion, and allow for the *possibility* of comms and computation overlap. Normally only one comm allowed at a time; non-blocking functions allow initiation of multiple comms, enabling MPI to progress them simultaneously.
- Non-blocking comms, when used properly, can provide a tremendous performance boost to parallel applications.

Non-blocking send operation



New DMC algorithm

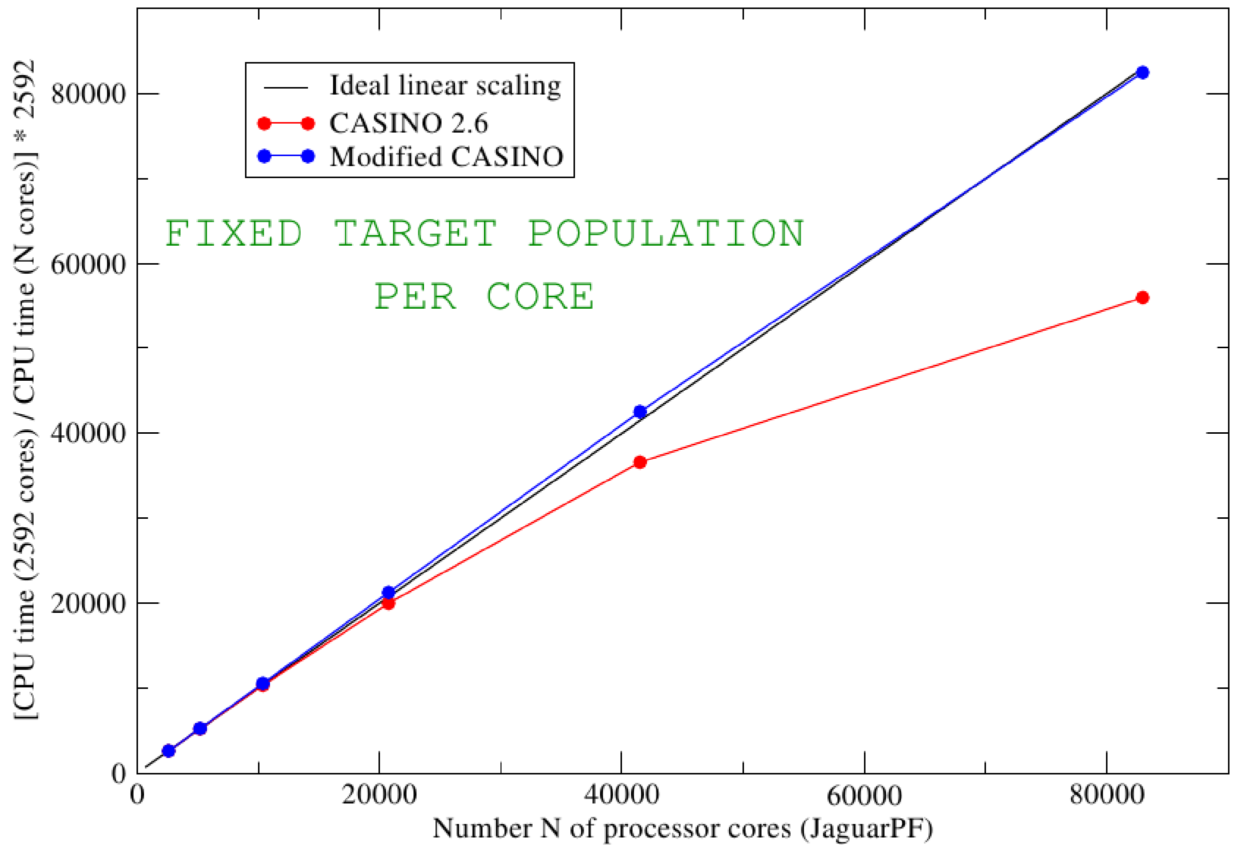
```
MOVE 1
- Move all currently existing configs forward by one time step
- Compute the multiplicities for each config (the number of copies of each
 config to continue in the next move).
- Looking at the current populations of config on each processor, and at the
 current multiplicities, decide which configs to send between which pairs of
 cores, and how many copies of each are to be created when they reach
 their destination.
- Sending cores initiate the sends using non-blocking MPI_ISENDs; receiving
 cores initiate the receives using non-blocking MPI_IRECVs. All continue
 without waiting for the operations to complete.
- Perform on-site branching (kill or duplicate configs which require it on any
 given processor).

MOVE 2 AND SUBSEQUENT MOVES
- Move all currently existing configs on a given processor by one time step (not
 including configs which may have been sent to this processor at the end of the
 previous move).
- Check that the non-blocking sends and receives have completed (they will
 almost certainly have done so) using MPI_WAITALL. When they have, duplicate
 newly-arrived configs according to their multiplicities and move by one time
 step.
- Compute the multiplicities for each moved config.
- Continue as before
```

Any improvement in the load-balancing time?

Number of cores	Time, CASINO 2.6 (s.)	Time, Modified CASINO (s.)
648	1.00	1.05
1296	3.61	1.27
2592	7.02	1.52
5184	18.80	3.06
10368	37.19	3.79
20736	75.32	1.32
41472	138.96	3.62
82944	283.77	1.04

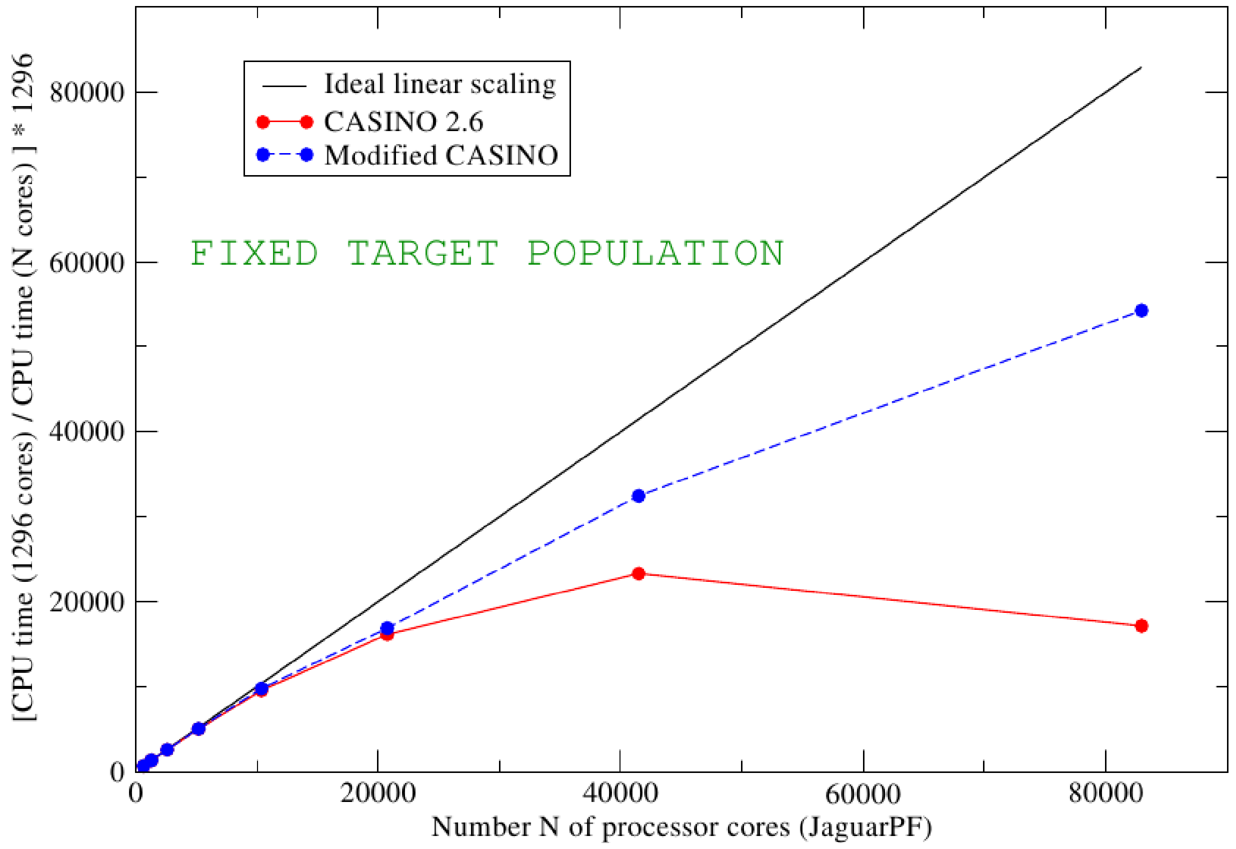
Table 1: *CPU time taken to carry out operations associated with redistribution of configs between cores in CASINO 2.6 (2010) and in my modified version, during one twenty-move DMC block for a water molecule adsorbed on a 2d graphene sheet.*

Perfect parallel efficiency..



Scaled ratio of CPU times in DMC statistics accumulation for various numbers of cores on Jaguar using both the September 2010 version of CASINO 2.6 (red line) and the current public release CASINO 2.8 (blue line). System: one H_2O molecule adsorbed on a 2D-periodic graphene sheet containing fifty C atoms per cell. For comparative purposes 'ideal linear scaling' (halving of CPU time for double the number of cores) is shown by the solid black line. In both cases there is a fixed target population of 100 configs per core (with an appropriately varying number of moves to maintain constant number of configuration space samples).

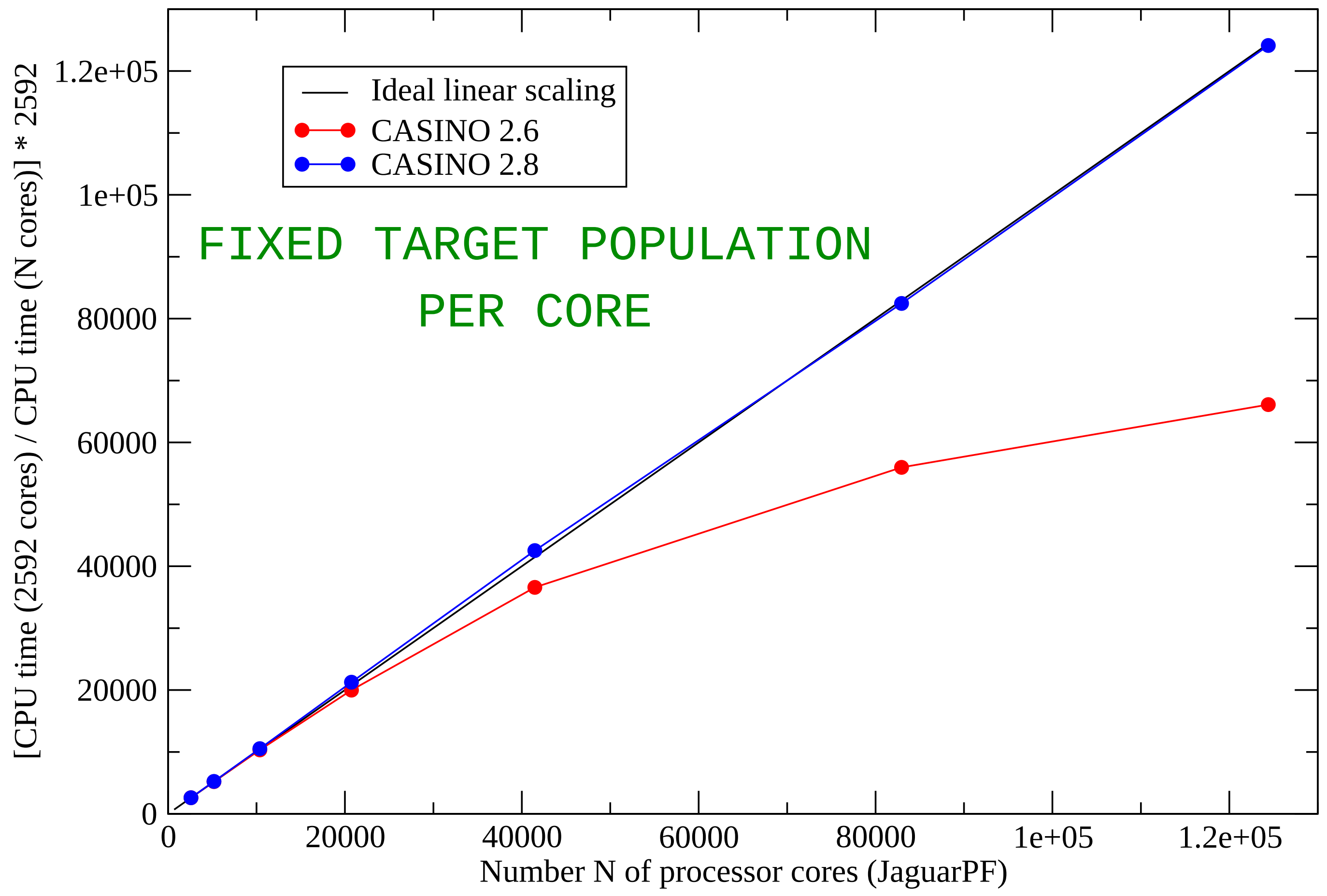
..if you give the processors enough to do



Similar graph for the same number of configuration space samples, but using a fixed target of 486000 for total config population and a fixed number of moves, rather than a fixed target per core.

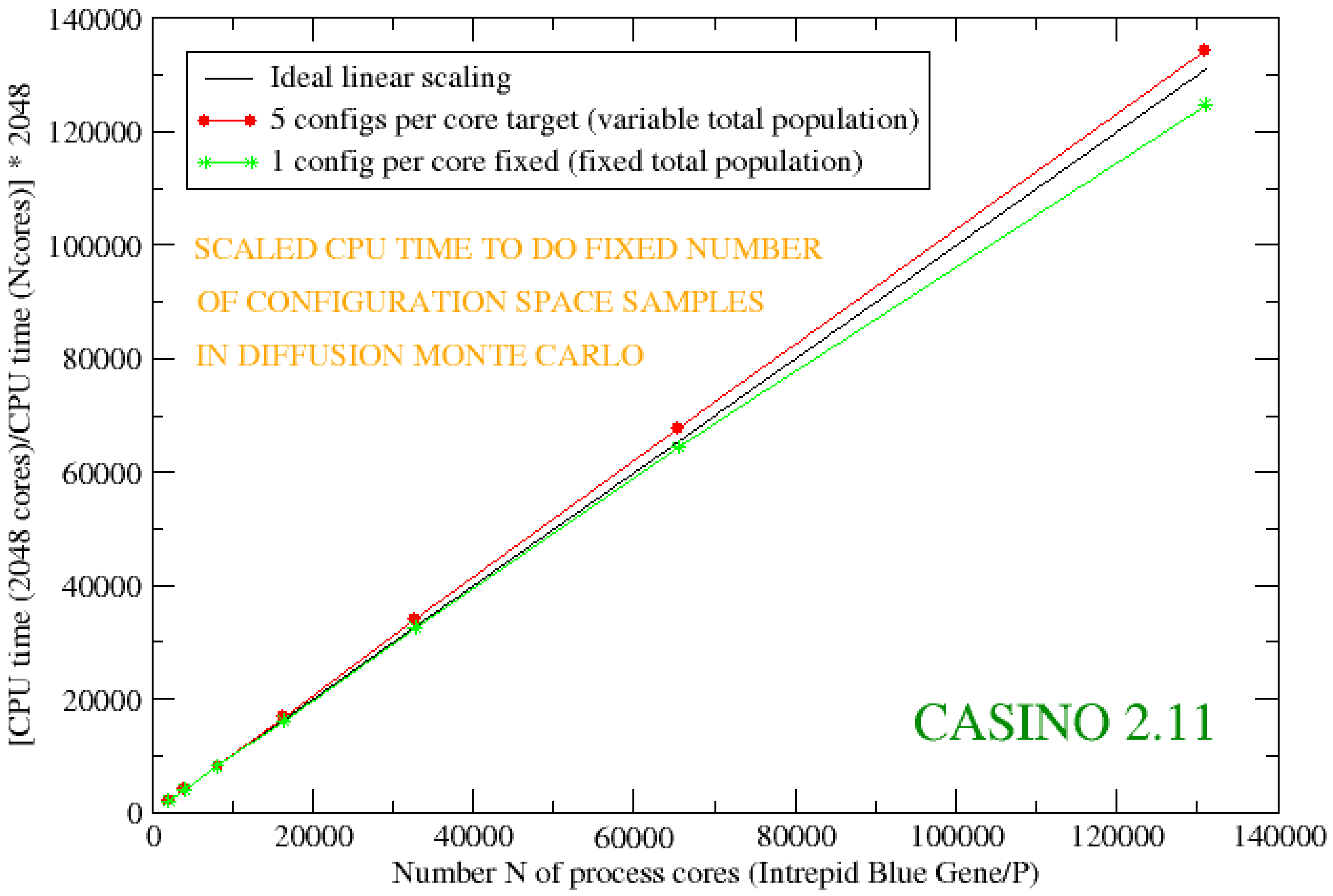
Note that fixing the total target population can introduce considerable inefficiency at higher core counts (since cores end up without enough work to do as the number of configs per node decreases). This graph should not be looked on as representing CASINO's general scaling behaviour. The inefficiency can generally be decreased by increasing the number of configs per core.

Can we push it to more than 100000 cores?



Yes! Not even the hint of a slowdown on 124416 cores.. Reasonable to assume we could use all 299008 cores of the Jaguar machine, if we could be bothered to sit through the queueing time.

..and up to 131072 cores on an IBM Blue Gene/P



How many cores can we exploit?

- Because QMC is a sampling technique then, for any given system, there is a maximum number of cores you can exploit if you insist that your answer has no less than some required error bar and that it has a minimum number of moves (in order to reblock the data, for example).
- E.g. we require 1000000 random samples of the wave function configuration space to get the required error bar ϵ. Let's say we need at least 1000 sampling moves to accurately reblock the results. And let's say we have a 1000 processor computer. In that case only one config per node is required to get the error bar ϵ (even though the available memory may be able to accommodate many more than this).
- We now buy a 2000 processor machine. How do we exploit it to speedup the calculation? We can't decrease the number of moves, since then we can't reblock. It is wasteful to just run the calculation anyway, since then the error bar will become smaller than we require. We can split each config over two nodes, and use OpenMP to halve the time taken to propagate the configs, but let's say we find that OpenMP doesn't really work very well over more than two cores.
- How then do we exploit a 4000 processor machine? Answer - we can't. The computer is simply too big for the problem if you don't need the error bar to be any smaller.

A problem: including the effects of equilibration time

Important point that we have ignored so far: *the DMC equilibration time cannot be reduced towards zero by using more cores*. And when the equilibration time becomes comparable to the statistics accumulation time (which *is* reduced by using more cores) our scaling analysis will be affected. Thus:

- In equilibration, the RMS distance diffused by each electron needs to be greater than some characteristic length. This translates into a requirement for a *minimum number of moves* (which obviously depends on the DMC time step τ). In fact, with valence electron density parameter r_s:

$$N_{\mathrm{equil}}^{\mathrm{min}} > \frac{N_{\mathrm{elec}}^{\frac{2}{3}} r_s^2}{3\tau}$$

- If you use more processors, with a fixed number of configs per core, the equilibration time will be independent of processor count (and will be smaller the fewer configs per core you use).
- The time taken to accumulate the data with the required error bar (through M samples of the configuration space) will go down with increasing core count, with 'perfect linear scaling'.

Computers like Jaguar have maximum job times (typically 12 or 24 hours) and a time that you have to sit in the queue before a job starts. So you can define a (somehwat arbitrary) maximum time that you are prepared to wait for DMC equilibration to complete. Neil has a nice internal paper, where he defines this as 4 days, and thus concludes (depressingly) that

(1) It will be difficult to do DMC calculations with more than around 800 electrons (since one has to equilibrate for days regardless of the size of the computer).

(2) For an 800 electron system, the maximum number of cores worth using is 14400 (when calculating the total energy of the simulation cell) or er.. 36 (when calculating the energy per atom).

Cheaper equilibration: the preliminary DMC scheme

With a small config population, equilibration is a small fraction of the total DMC run time. When running on a large number of cores however, the configuration population is necessarily large, since each core must have at least one configuration on average; the equilibration time will likely be large.

An alternative procedure which can significantly reduce the expense of equilibration is Neil's preliminary DMC scheme (see section 13.1 of the CASINO manual):

- Instead of using VMC to generate all the $N_{\rm conf}$ required configs for the DMC calculation, use VMC to generate a much smaller number of configs $N_{\rm conf}^{\rm small}$ (at least 1000 to be large enough to avoid population control bias).
- Equilibrate the $N_{\rm conf}^{\rm small}$ configs on some small system without a delay-inducing queueing system.
- Run enough statistics accumulation on the equilibrated $N_{\rm conf}^{\rm small}$ in order to generate $N_{\rm conf}$ independent configs for use in the full DMC calculation. (Can also use the energies of these configs in the accumulation data..).

Example

Want to do 50000 core calculation, with 2 configs per core - require 100000 equilibrated configs.

Normally use 100000 samples of VMC wave function to provide initial configs to be equilibrated, then in order to equilibrate DMC we need to run - say - 750 steps on each of the 100000 configs (2 configs per core).

Instead use 1000 samples of VMC wave function on (say) 500 cores (2 configs per core), then run each of those for 750 steps to achieve DMC equil - 100 times faster than before because I have far fewer configs.Turn on stats accumulation. Run each of the 1000 samples for $100 \times T_{\rm corr}$ moves to get 100000 independent samples in total. Takes roughly the same time as equilibration. Thus overall around 50 times faster than equilibration done on 100000 configs.

Drawback: more costly in human time as one has to run equilibration on small computer then transfer to Jaguar or whatever..

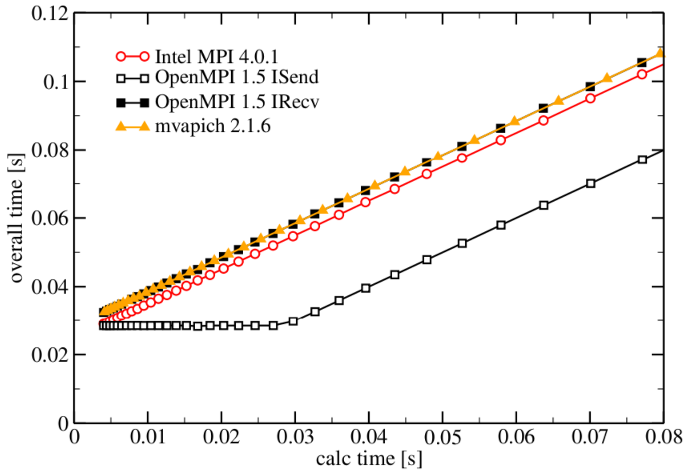
Is non-blocking communication really asynchronous?

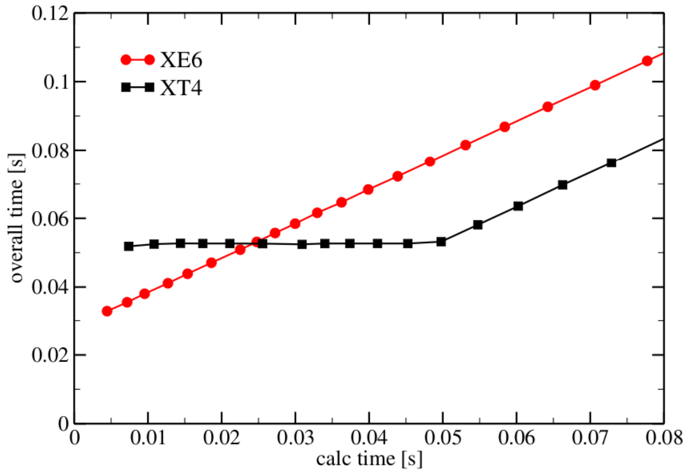
Not necessarily! MPI standard doesn't *require* non-blocking calls to be asynchronous. Two problems:

(1) Hardware may not support asynchronous communication. Some networks provide communication co-processors that progress message passing regardless of what application program does (e.g. Infiniband, Quadrics, Myrinet, Seastar and some forms of TCP that have offload engines on the NIC). Then communication can be started by the computation processor which in turn gives task of sending data over the network to the communication processor.

(2) Unfortunately, even if the hardware supports it, people implementing MPI libraries may not bother to code up truly asynchronous transfers (since the standard allows them not to!). MPI progress is actually performed within the MPI_TEST or MPI_WAIT functions. This is cheating!

Test: initiate MPI_IRECV with large 80Mb message, then do some computation for a variable amount of time. If comms really do overlap with computation then total runtime will be constant so long as computation time is smaller than comms time.
Ref: Hager *et al*. `http://blogs.fau.de/hager/files/2011/05/Hager-Paper-CUG11.pdf`



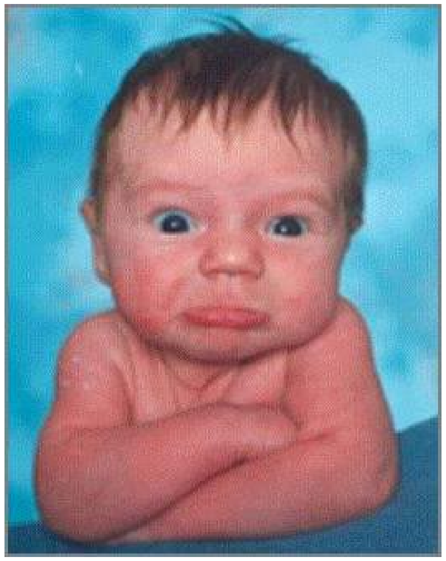


If your MPI doesn't provide true asynchronous progress, then some form of periodic poll through a MPI_TESTALL operation may be required to achive optimal performance. Can also overlap computation and comms via mixed-mode OpenMP/MPI - use dedicated communication thread.

Conclusions

- In general it seems to be the case that, following my modifications, the cost of DMC statistics accumulation in CASINO is now linear scaling with the number of cores providing the problem is large enough to give each core enough work to do.
- This should normally be easy enough to arrange, and if you find yourself unable to do this, then you don't need a computer that big.
- On typical machines like Jaguar, very large priority is given to jobs using large numbers of cores (where 'large' means greater than around 40000). Being allowed to use the machine in the first place increasingly means being able to demonstrate appropriate scaling of the code beforehand. CASINO can do this; many, even most, other techniques cannot.
- That said, it seems clear that we have to address issues related to equilibration time if we are to use CASINO for big systems on very large numbers of cores.. (hence the phrase 'partial success' on my first slide..).
- Massively parallel machines are now increasingly capable of performing highly accurate QMC simulations of the properties of materials that are of the greatest interest scientifically and technologically.

I don't have access to a petascale computer (sulk..)



So you have three options:

(1) Don't do QMC calculations on very big systems.

(2) Wait for 10 years until everyone has a petascale computer under their desk.

(3) Unless you happen to be North Korean or Iranian or otherwise associated with the Axis of Evil, apply for some time on one. We did. You might consider:

The INCITE program

`www.doeleadershipcomputing.org/guide-to-hpc/`

The European DEISA program

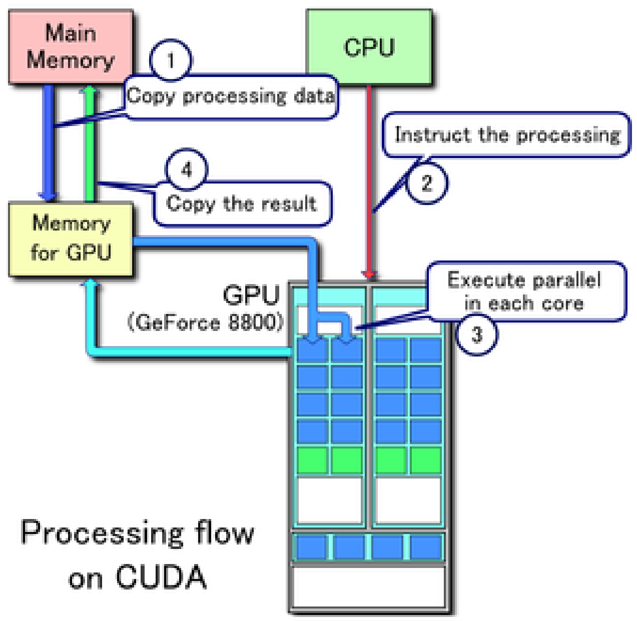
`www.deisa.eu`

The future? GPU accelerators

- A GPU (graphics processing unit) is a specialized processor designed to answer the demands of real-time high-resolution 3D-graphics compute-intensive tasks (whose development was driven by rich nerds demanding better games). They are produced by big companies such as Nvidia and ATI.
- Decent modern GPUs in machines with only a few CPU-cores are engineered to perform *hundreds* of computations in parallel. In recent years there has been a trend to use this additional processing power to perform computations in applications traditionally handled by the CPU.
- Modern GPUs have evolved into highly parallel multicore systems allowing very efficient manipulation of large blocks of data. This design is more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel.
- To do general purpose computing on GPUs, people originally had to 'pretend' to be doing graphics operations and learn things like OpenGL or DirectX (and so very few people bothered). Nowadays, new architectures such as CUDA allow people to operate GPUs using more familiar programming languages, and their use is booming.
- In fact, it might be said, that computing is evolving from 'central processing' on the CPU to 'co-processing' on the CPU and GPU. Of of the current top 500 supercomputers, 57 of them use GPU accelerators in their design (up from 39 in 2011). As we have seen, the Jaguar system is being upgraded to use them.

The highly parallel nature of Monte Carlo algorithms suggest that CASINO might benefit considerably from GPU co-processing, and so one of my jobs this year is to explore the possibility of doing just that.

Programming for Nvidia GPUs: CUDA



CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by Nvidia. It is the computing engine in Nvidia GPUs that is accessible to developers through variants of industry-standard languages. Programmers typically use 'C for CUDA' (C with Nvidia extensions and certain restrictions) to code algorithms for execution on the GPU.

CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs so that they become accessible for computation like CPUs. Unlike CPUs however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly.

CASINO is written in Fortran95, so we would like to code in Fortran directly, rather than the officially-supported C. Fortunately (see e.g. `www.pgroup.com/resources/cudafortran.htm`) there are available third-party solutions such as PGI CUDA Fortran, so one can do things like this:

```
REAL :: a(m,n)              ! a instantiated in host memory
REAL,DEVICE :: adev(m,n)    ! adev instantiated in GPU memory
adev = a                    ! Copy data from a (host) to adev (GPU)
a = adev                    ! Copy data from adev (GPU) to a (host)
```

Understand Nvidia *threads*, *warps*, *blocks*, *grids*. See e.g.:
`www.pgroup.com/lit/articles/insider/v2n1a5.htm`
`www.tomshardware.com/reviews/nvidia-cuda-gpu,1954-7.html`

```
call matmul_kernel<<<dimGrid,dimBlock>>>( adev,bdev,cdev,n,m,l )

ATTRIBUTES(global) SUBROUTINE matmul_kernel( a,b,c,n,m,l)
REAL,DEVICE ::  a(n,m),b(m,l),c(n,l)
```

Programming for Nvidia GPUs: Accelerator directives

An alternative to CUDA programming is the use of the *Accelerator Directive Model* which allow you to program GPUs/accelerators simply by adding compiler directives (which look like comments) to existing code and recompiling with special flags (`www.pgroup.com/resources/accel.htm`). This is very similar to how OpenMP works. As the web site blurb says:

- *The Accelerator Model describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.*
- *The directives and programming model allow programmers to create high-level host+accelerator programs without the need to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown.*
- *All of these details are implicit in the programming model and are managed by the accelerator-enabled compilers and runtimes. The programming model allows the programmer to augment information available to the compilers, including specification of data local to an accelerator, guidance on mapping of loops onto an accelerator, and similar performance-related details.*

Up to now, I have been using the the PGI Accelerator model, embodied in the pgfortran compiler. This has now been formalized in the OpenACC standard (in a collaboration between Cray, Nvidia, CAPS Enterprise, and PGI - with the support of many HPC labs and other organizations). OpenACC has a slightly different syntax to the original PGI Accelerator directives. As of version 12.6 of pgfortran (released last week), full versions of both sets of directives are implemented. The former will presumably eventually be deprecated in favour of the OpenACC standard (which is now supported by three compilers, and has been demonstrated to work with Nvidia, ATI, and Intel GPUs).

Example: Fortran matmul loop

PGI Accelerator

```
!$acc region
      do k = 1,n1
       do i = 1,n3
        c(i,k) = 0.0
        do j = 1,n2
         c(i,k) = c(i,k) + a(i,j) * b(j,k)
        enddo
       enddo
      enddo
!$acc end region
```

OpenACC

```
!$acc kernels
      do k = 1,n1
       do i = 1,n3
        c(i,k) = 0.0
        do j = 1,n2
         c(i,k) = c(i,k) + a(i,j) * b(j,k)
        enddo
       enddo
      enddo
!$acc end kernels
```

What GPU approach shall I adopt for CASINO?

The CASINO 'philosophy' up to now is that there should be *single* version of the code which is:

- Portable across all supported hardware (i.e. any reasonably modern serial or parallel computer).
- Can be compiled on any machine by typing '`make`' (with the necessary compiler flags etc. stored permanently in the distribution in 'architecture files' - see PLR talk).
- Can be run on any machine by typing '`runqmc`' (with possible some command line arguments to specify the number of cores, wall time limits etc..).

Advantages of the OpenACC model:

- Potentially allows single version of the code (no multiple 'GPU versions'). Easier to support.
- Very much easier to code. CUDA requires you to program at a detailed level including a need to understand and specify data usage information and manually construct sequences of calls to manage all movement of data between the CPU and GPU.
- Much more portable across devices. No need to reprogram if a new GPU architecture is released.
- With their participation in OpenACC, Nvidia seem to be planning for a future away from CUDA?

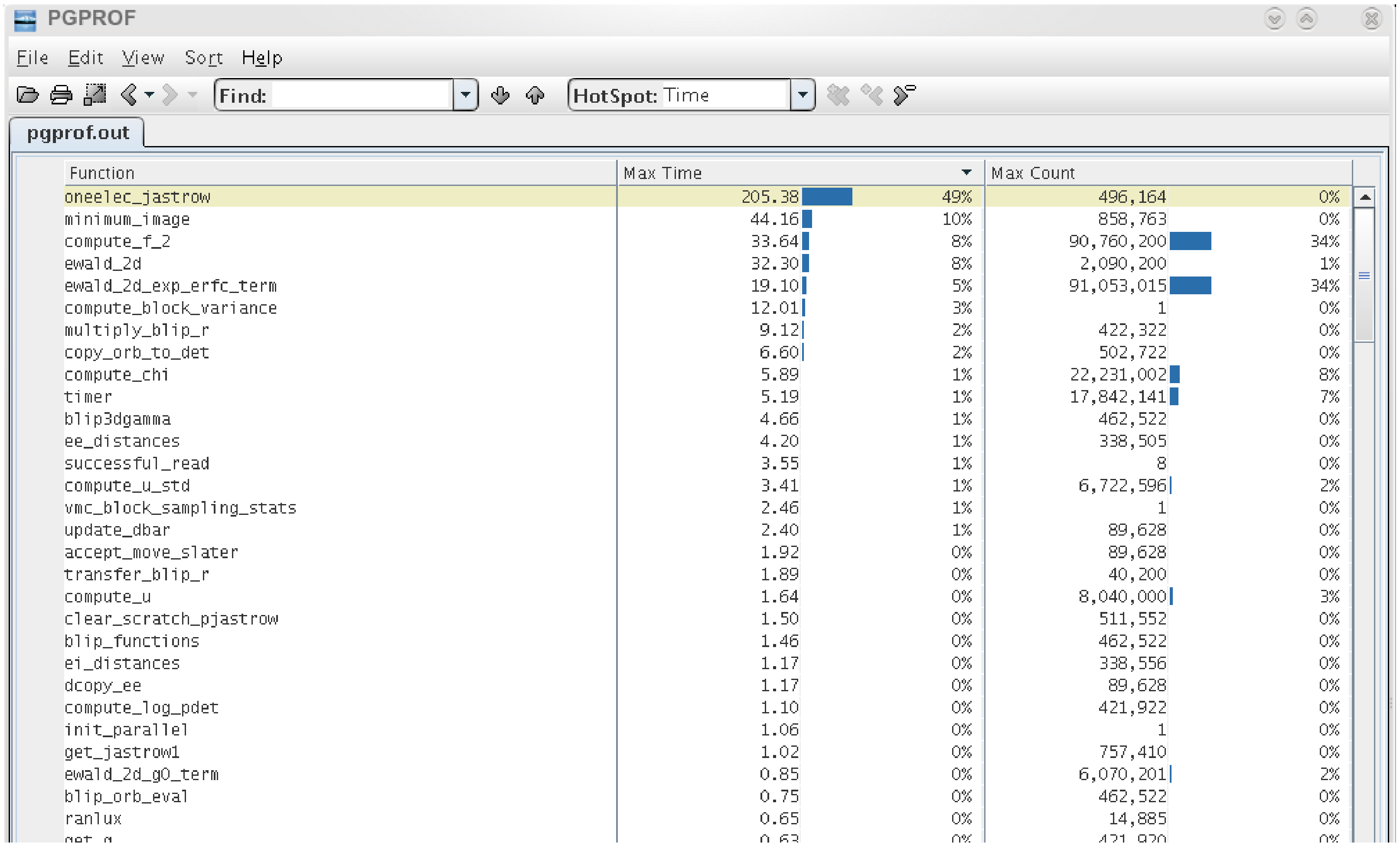
Disadvantages of the OpenACC model:

- Compiler support maybe not very mature. Possibility of bugs or other infelicities?
- Less fine-grained control, so possibility of small performance hit compared to CUDA.

I will therefore go with OpenACC for now. If necessary I will work with compiler developers to make sure CASINO's requirements are met. Ryo has chosen the CUDA route - he will speak on this next.

What are the rate-limiting steps in CASINO?

Depends heavily on application, basis set, etc. Let's pick one recent heavily-used example and focus on that i.e. Dario's 1 H atom on a 50 C-atom per cell 2D-periodic graphene slab, blip basis set. Use PGI Fortran compiler and accompanying 'pgprof' utility to profile the code:



Function	Max Time		Max Count	
oneelec_jastrow	205.38	49%	496,164	0%
minimum_image	44.16	10%	858,763	0%
compute_f_2	33.64	8%	90,760,200	34%
ewald_2d	32.30	8%	2,090,200	1%
ewald_2d_exp_erfc_term	19.10	5%	91,053,015	34%
compute_block_variance	12.01	3%	1	0%
multiply_blip_r	9.12	2%	422,322	0%
copy_orb_to_det	6.60	2%	502,722	0%
compute_chi	5.89	1%	22,231,002	8%
timer	5.19	1%	17,842,141	7%
blip3dgamma	4.66	1%	462,522	0%
ee_distances	4.20	1%	338,505	0%
successful_read	3.55	1%	8	0%
compute_u_std	3.41	1%	6,722,596	2%
vmc_block_sampling_stats	2.46	1%	1	0%
update_dbar	2.40	1%	89,628	0%
accept_move_slater	1.92	0%	89,628	0%
transfer_blip_r	1.89	0%	40,200	0%
compute_u	1.64	0%	8,040,000	3%
clear_scratch_pjastrow	1.50	0%	511,552	0%
blip_functions	1.46	0%	462,522	0%
ei_distances	1.17	0%	338,556	0%
dcopy_ee	1.17	0%	89,628	0%
compute_log_pdet	1.10	0%	421,922	0%
init_parallel	1.06	0%	1	0%
get_jastrow1	1.02	0%	757,410	0%
ewald_2d_g0_term	0.85	0%	6,070,201	2%
blip_orb_eval	0.75	0%	462,522	0%
ranlux	0.65	0%	14,885	0%
get_q	0.63	0%	421,920	0%

It's spending half its time calculating the Jastrow factor (specifically the f function)! Other fairly heavy steps are the minimum image routine and the routine for computing 2D Ewald interactions. Blip orbital evaluation only 3% at the most.

Focus on Jastrow f function

The expression for f is the most general expansion of a function of r_{ij}, r_{iI} and r_{jI} that does not interfere with the Kato cusp conditions and goes smoothly to zero when either r_{iI} or r_{jI} reach cutoff lengths (L_f = 5.82 au for C and 2.87 au for H in DA's calculations):

$$f_I(r_{iI}, r_{jI}, r_{ij}) = (r_{iI} - L_{fI})^C (r_{jI} - L_{fI})^C \Theta(L_{fI} - r_{iI}) \Theta(L_{fI} - r_{jI}) \sum_{l=0}^{N_{fI}^{\mathrm{eN}}} \sum_{m=0}^{N_{fI}^{\mathrm{eN}}} \sum_{n=0}^{N_{fI}^{\mathrm{ee}}} \gamma_{lmnI} r_{iI}^l r_{jI}^m r_{ij}^n$$

Best way to speed up these calculations might be to not use Jastrow f functions! But the cases where one routine dominates such as this is actually an ideal case for GPUs (one would think). We have a triple loop over electrons, electrons, and nuclei that one can try to 'accelerate'.

Even before I looked at GPUs however, I noted that the CASINO Jastrow f function evaluation was written in a compact, beautiful and elegant way - which is always a danger sign. By rearranging it to look repetitive, ugly and inelegant, and with a few other tricks, one can make a slight difference:

```
Evaluation of f function (subroutine timers on):
42.34 sec (OLD) --> 11.48 sec (NEW) = 3.69 x speedup

Speed of the full calculation (subroutine timers on):
68.59 sec (OLD) --> 36.20 sec (NEW) = 1.91 x speedup

Speed of the full calculation (subroutine timers off):
65.39 sec (OLD) --> 33.60 sec (NEW) = 1.95 x speedup
```

The 49% Jastrow contribution in the profile was done *after* these mods - it used to be 70-80% !

Some simple tests of PGI accelerator directives

```
TEST 1
------
!$acc region
outer loop of length 128
 select case
 case (1)
  inner loop of length 1024
   compute f
  end loop
 case (2)
  inner loop of length 1024
   compute f some other way
  end loop
 case (default)
  inner loop of length 1024
   compute f even some other way
  end loop
end loop
!$acc end region

 "/tmp/mdt26/pgaccI2ncM38dPHwf.gpu(50): error: duplicate parameter name'

BUG 1 : Can't have select case inside loop. Delete it.
```

Some simple tests of PGI accelerator directives II

```
TEST 2
------
!$acc region
outer loop of length 128
  inner loop of length 1024
   compute f
  end loop
end loop
!$acc end region

Compiles, but when you run it:

"call to cuMemcpyDtoH returned error 700: Launch failed"

BUG 2 : OK - do inner loop only
```

Some simple tests of PGI accelerator directives III

```
TEST 3
------

outer loop of length 128
 !$acc region
  inner loop of length 1024
   compute f
  end loop
 !$acc end region
end loop

Compiles, but when you run it:

"call to cuMemcpyDtoH returned error 702: Launch timeout"

BUG 3 - sigh, give up..
```

Some simple tests of PGI accelerator directives IV

```
TEST 4
------

Would like to retain some data on the GPU in between subroutine calls. To do
this you use the 'mirror' directive.. This directive tells the compiler that
allocate and deallocate statements for this array should allocate copies both
on the host and on the GPU. When these arrays appear in host code, the host
copy is used; when they appear in PGI Accelerator compute regions, the GPU copy
is used. Using the mirror directive with module allocatable arrays gives them
global visibility.

I can make this feature crash the compiler with the following perfectly
legitimate 6 line program:

 MODULE test
  IMPLICIT NONE
  REAL(kind=kind(1.d0)) a(1409)
  INTEGER,ALLOCATABLE :: arse(:)
 !$acc mirror (arse)
 END MODULE test

 If I compile that with 'pgfortran -ta=nvidia test.f90', I get :

 pgfortran-Fatal-/home/user/bin/pgi/linux86-64/12.4/bin/pgf902 TERMINATED
 by signal 11
 Arguments to /home/user/bin/pgi/linux86-64/12.4/bin/pgf902
 /home/user/bin/pgi/linux86-64/12.4/bin/pgf902
 /tmp/user/pgfortranBMZr3HsbP_R.ilm -fn test.f90 -opt 2 -terse 1 -inform
 warn -x 51 0x20 -x 119 0xa10000 -x 122 0x40 -x 123 0x1000 -x 127 4 -x 127
 17 -x 19 0x400000 -x 28 0x40000 -x 120 0x10000000 -x 70 0x8000 -x 122 1 -x
 125 0x20000 -quad -x 59 4 -x 59 4 -tp nehalem -x 120 0x1000 -x 124 0x1400
 -y 15 2 -x 57 0x3b0000 -x 58 0x48000000 -x 49 0x100 -x 120 0x200 -astype 0
 -x 124 1 -x 163 0x1 -x 186 1 -accel nvidia -x 176 0x140000 -x 177
 0x0202007f -cmdline '+pgfortran test.f90 -ta=nvidia' -asm
 /tmp/user/pgfortranJMZPGoyiKWs.s
```

PGI accelerator - initial observations

- The above is a sort of cartoon of my actual experience, but it is true to say that the compiler is full of silly bugs, producing very uninformative error messages both at compile time and run time. Thanks to some helpful people at PGI, some (but not all) of these have been fixed.
- Finally, following certain bug fixes, I have been able to compile and run the code to completion for some test cases.

Sample timings with CASINO 2.11.xx:

CPU only: 278 seconds

CPU+GPU : 7478 seconds

i.e. it runs so lowly that I assumed it had crashed and went to bed. I was very surprised to see it had run to completion the following morning.

This turns out to be partly due to unexpected compiler behaviour (see later).

These results are for 1 CPU core and 1 expensive high-end GeForce GTX 480 GPU. If you try to use more than 1 CPU core talking to the same GPU, then the timings get considerably worse! (likely to be a problem with the new supercomputers being constructed which have e.g. 1 GPU per 16-core node,or whatever). How to combine MPI with GPU? Hmmm...

A PGI developer speaks

He notes that much of the timing issues are caused by supposedly ‘private’ variables mistakenly being allocated and deallocated on each loop iteration by the compiler and this can be partially alleviated by using variables that are only active in that particular block of code.

Quotes from Dave Norton at PGI:

“I’m not sure why the compiler is doing this, but will be with Michael in Germany at ISC all next week, so will ask him about it.”

“The compiler doesn’t seem to have a broad enough view of the code to understand that the variables declared as ‘private’ are not used before set in the code after this block. I’ll talk with Michael about this too, but I don’t know if it’s reasonable for the compiler to have that large a view of the code.”

“I do see a couple other things that don’t make sense in the NVDEBUG output. I see that valc is allocated and deallocated on each loop iteration - which I would expect. It appears that something unnamed is being allocated and that ‘b1’ is being uploaded. This doesn’t make any sense. It may be that ‘b1’ is an internal compiler variable (as well as one of your variables), so I’m still tracking that mystery down.”

“I think the big take home message from this is - that there needs to be a much larger section of code that runs on GPU to overcome the cost of moving the data there and back. Perhaps we can get the entire subroutine to run on the GPU, and then maybe even a bigger section of code. Of course - we have to be able to compile a larger section of code - so hopefully we’ll have these issues fixed in the next few weeks and can see about broadening the scope of what runs on the GPU (and perhaps minimizing the number of data transfers.).”

A CASTEP developer speaks

```
Date: Thu, 14 Jun 2012 14:34:42 +0100
From: Phil Hasnip <phil.hasnip@york.ac.uk>
To: Mike Towler <mdt26@cam.ac.uk>
Subject: Re: GPUs

Dear Mike,

We played around with GPUs in a noddy fixed-potential code I use for
teaching. I think we did a report on it somewhere, I'll see if I can
find it. Basically it took a good couple of months to get anything
much from the GPUs. The main lessons were:

1. Minimise memory transfers from the CPU to GPU and back - in other
words do as much as possible on the GPU

2. Don't bother with GPUs for any non-square matrix work (the
performance tails off dramatically as nrows diverges from ncols)

When you've done the above and everything else as well as you can, you
basically end up with a speed-up of about 5. This speed-up comes from
the improved memory bandwidth of the GDDR5 on the video card, you get
nothing from the parallelism. At this point I decided I would never
port CASTEP to GPUs, I'd just wait for CPUs to move to DDR5.

People who have extremely parallelisable, low memory operations (which
I guess might include CASINO) may be able to get another factor or 2
to 4 from the card's GPUs for a grand total of 20, but the speed-ups
quoted on NVidia's webspage are largely fictitious (not just my view,
this is NVidia's official position).

The other serious problem from a CASTEP point of view is that NVidia
told us they plan to completely change the card architectures every 3
years, in such a way that any existing optimisation will have to be
totally redone.

Phil
```

Another CASTEP developer speaks

```
Date: Thu, 14 Jun 2012 14:54:59 +0100
From: Dr Matt Probert <matt.probert@york.ac.uk>
To: Mike Towler <mdt26@cam.ac.uk>
Subject: Re: GPUs

Hi Mike,
   Interesting! We in the CDG have looked into GPUs in detail (and I've done
some elementary stuff here in York) and decided that it is not for CASTEP at
the moment. The problems are:

a) Slow interface to main memory - takes MANY clock cycles to get into GPU -
so the only algorithms that do well are those with many operations on a
small amount of data. A nice example that the micromagnetic MC guys do is to
put the random number generator on the GPU - makes it very fast. In DFT
world, there was a paper recently about putting the non-local XC functional
onto a GPU - again, VERY intensive chunk of code, with modest amount of data
involved, that is a real bottleneck so good Amdahl target!

b) Proprietary interface & hardware - lot of vendor lock-in problems - which
is made worse by the fact that when we spoke to nVidia, they explicitly said
that they had an 18 month development cycle, and were at liberty to totally
change the API every 18 months => need lots of regular code updates.

c) Other problems include how to combine GPU with MPI - can be done in
principle but not many have done it yet in practice.

Hence we've chosen to "wait and see" for now.

xx
Matt
```

The final situation with CASINO and PGI Accelerator Fortran

- It just doesn't work. But I continue to live in hope that working with the PGI developers will help.
- PGI claimed that version 12.6 of the compiler will be much better than all the earlier versions, and that in fact CASINO was holding up the release!
- Version 12.6 was released just a few days ago, and I haven't had the opportunity to test it. However, Dave Norton says:

```
From: Dave Norton <norton@hpfa.com>
To: Mike Towler <mdt26@cam.ac.uk>

Hi Mike -

Yes - 12.6 finally came out a couple of days ago.  I'm just running through
my list of issues and checking on them.

As I suspected a few weeks ago, when push came it shove in getting the
release out, engineering passed over a number of issues in favor of getting
some of the remaining OpenACC functionality implemented. I'm not sure the
status of all of your issues - I'll check over the next day or two, but
don't believe some of the optimization issues were addressed.  Sorry.

-dave
```

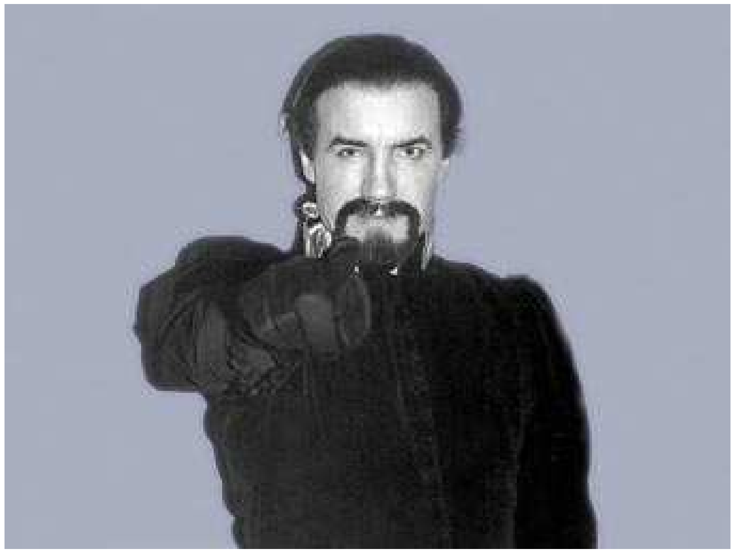
So it's all very depressing isn't it! Let's hope Ryo can do better..

The third problem: Grossman-Mitas molecular dynamics

Dario Alfè asked me to code this up in CASINO (to finish a project started by Norbert Nemec).

Jeff Grossman

Lubos Mitas



‘*Efficient quantum Monte Carlo energies for molecular dynamics*’, J.C. Grossman and L. Mitas, *Phys. Rev. Lett* **94**, 056403 (2005)

This has now been done but, when I eventually got around to thinking more deeply about what the method was useful for, I found I had some concerns. That is what this part of the talk is about.

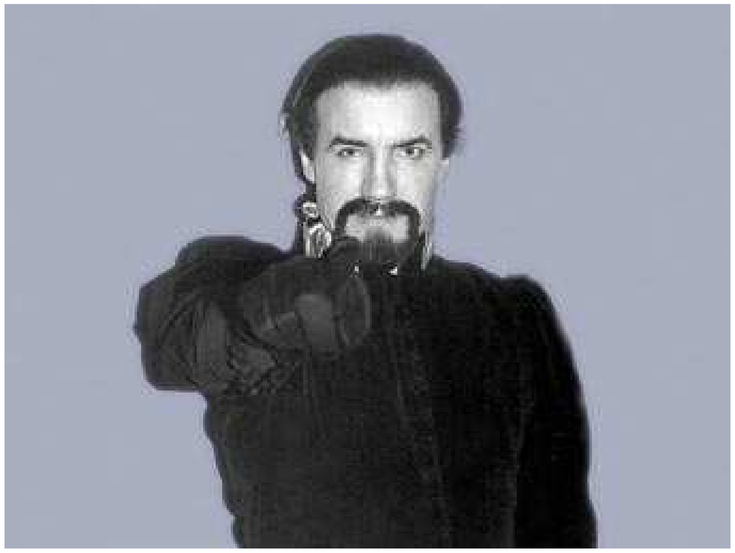
The third problem: Grossman-Mitas molecular dynamics

Dario Alfè asked me to code this up in CASINO (to finish a project started by Norbert Nemec).

Jeff Grossman

Lucas Wagner

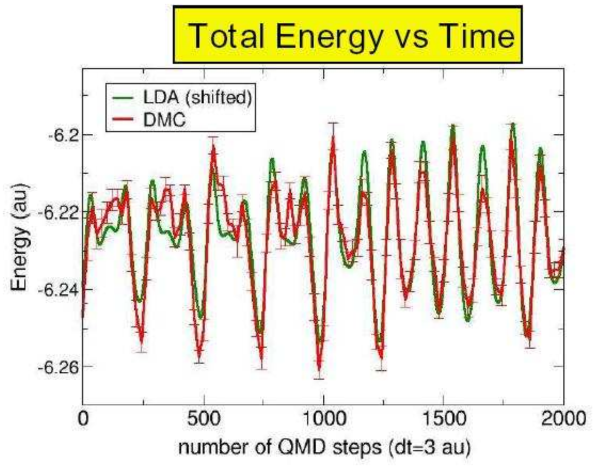
Lubos Mitas

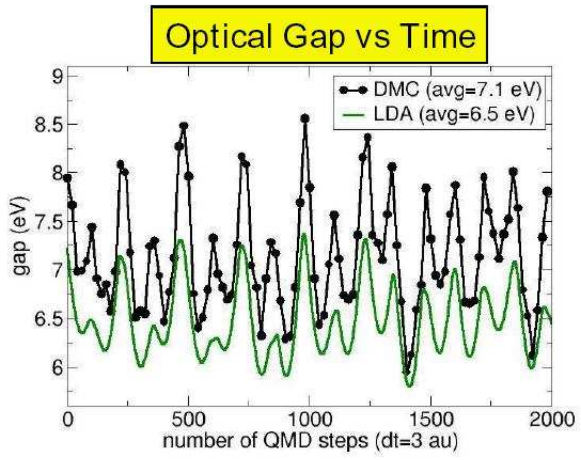


'*Efficient quantum Monte Carlo energies for molecular dynamics*', J.C. Grossman and L. Mitas, *Phys. Rev. Lett* **94**, 056403 (2005)

This has now been done but, when I eventually got around to thinking more deeply about what the method was useful for, I found I had some concerns. That is what this part of the talk is about.

Grossman-Mitas molecular dynamics





- Grossman-Mitas molecular dynamics (GM-MD) is a method to treat electrons within QMC 'on-the-fly' throughout a molecular dynamics simulation.
- The technique involves using a DFT code (the CASINO implementation uses PWSCF) to generate trial wave functions for each nuclear configuration along an MD trajectory, then doing separate DMC calculations at each point.
- Rather than doing a full DMC calculation for each trial function, each DMC run after the first is 'restarted' from the walker set stored in the file ('`config.out`') from the previous nuclear configuration.
- On being read in each walker is 'reweighted' by the ratio of the square of the new and old wavefunctions and the best estimate of the DMC energy recomputed.
- As these walkers are 'almost' equilibrated for the slightly changed nuclear positions, it is suggested that very little, if any, DMC equilibration is required. Furthermore, only two or three DMC stats accumulation moves are performed for each MD step, apparently resulting in a huge speedup over doing repeated full DMC calculations.

How does the DMC-MD implementation work in CASINO?

Shell script `runqmcmd` used to automate DMC-MD using CASINO and the PWSCF DFT code (part of Quantum Espresso available for free at `www.quantum-espresso.org` - must use version 4.3+).

The script works by repeatedly calling CASINO run scripts `runpwscf` and `runqmc` which know how to run the two codes on any known machine. See instructions in `utils/runqmcmd/README`.

Setup the PWSCF input ('`in.pwscf`') and the CASINO input ('`input`' etc. but no wave function file) in the same directory. For the moment we assume you have an optimized Jastrow from somewhere (this will be automated later). Have the PWSCF setup as '**calculation = "md"**', and '**nstep = 100**' or whatever. The `runqmcmd` script will then run PWSCF once to generate 100 `xwfn.data` files, then it will run CASINO on each of the `xwfn.data`. The first will be a proper DMC run with full equilibration (using the values of **dmc_equil_nstep**, **dmc_stats_nstep** etc.). The second and subsequent steps (with slightly different nuclear positions) will be restarts from the previous converged `config.in` - each run will use new keywords **dmcmc_equil_nstep** and **dmcmd_stats_nstep** (with the number of blocks assumed to be 1. The latter values are used if new keyword **dmc_md** is set to T, and they should be very small).

It is recommended that you set **dmc_spacewarping** and **dmc_reweight_conf** to T in CASINO `input` when doing such calculations.

The calculation can be run through plane-wave `pwfn.data`, blip `bwfn.data` or binary blip `bwfn.data.b1` formats as specified in the `pw2casino.dat` file (see CASINO/PWSCF documentation).

```
runqmcmd --nproc=124680 --walltime=12hr --shmem
```

Though of course no one has ever used it. Sigh..

What question am I trying to answer?

Dario asked me to "*show that GM-MD was* xxx *times faster than standard sampling*".

This question is based on the claims in the original Grossman-Mitas paper that:

- '*This continuous evolution of the QMC electrons results in highly accurate total energies for the full dynamical trajectory at a fraction of the cost of conventional, discrete sampling.*'
- '[It provides]*an improved, significantly more accurate total energy for the full dynamical trajectory*.'
- '*This approach provides the same energies as conventional, discrete QMC sampling and gives error bars comparable to separate, much longer QMC calculations.*'

A casual reading of this might lead to the conclusion that we get effectively the same results as conventional DMC at each point along the trajectory. As we are only doing e.g. 3 DMC stats accumulation per step, we thus appear to be getting 'something for nothing'. What are we missing?

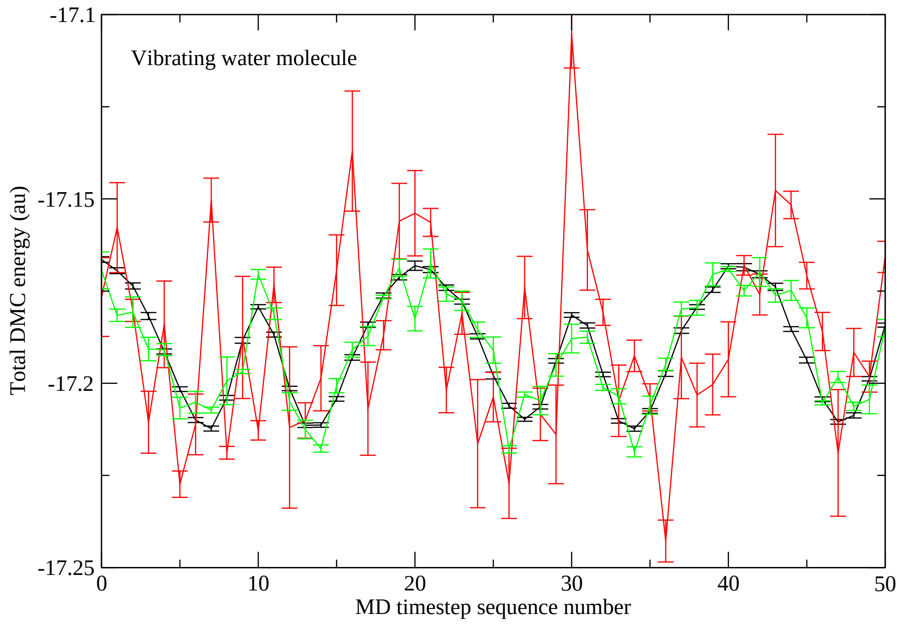
What are we missing?

(1) The point about GM-MD - and Lubos has agreed with this in an email conversation - is that its true purpose is to calculate *thermodynamic averages* - an example being given in the paper of the heat of vaporization of H_2O. This is calculated as an average over the evolution path of the ions (at a given $T > 0$) and QMC and thermodynamic averages are done at the same time. Think of it as a method of Monte Carlo sampling a distribution in 3N-dimensional configuration space that is changing shape in time. Each nuclear configuration is sampled via far fewer (e.g. three moves times the number of walkers) electron configurations than usual. It is not intended that accurate energies for all nuclear configurations are calculated, only an accurate Monte-Carlo-sampled 'thermodynamic average' as the molecule (or whatever) vibrates or otherwise moves. This is not, in my opinion, made clear in the paper.

(2) If you want accurate answers and small error bars for the energies at each point along the MD trajectory then - done under normal conditions (number of DMC walkers etc.) - the method actually does not save you any time at all, apart from that spent doing DMC equilibration (and Neil's preliminary DMC method already helps with that). It gives extremely poor answers with huge error bars for the individual MD points - if you want proper DMC accuracy then you have to do the usual amount of statistical accumulation work.

(3) The only reason they are able to claim '*the same energies* [and error bars] *as conventional, discrete QMC sampling*' is because they use extremely large populations of walkers (a technique which is difficult to scale to large systems). The requirement for a large number of walkers is not stressed in the paper; it is merely stated that '*we chose a number of walkers such that the statistical fluctuations in the GM-MD energies are one-tenth the size of the variation in the total energy as a function of the MD time* with no emphasis that this is considerably more than usual.

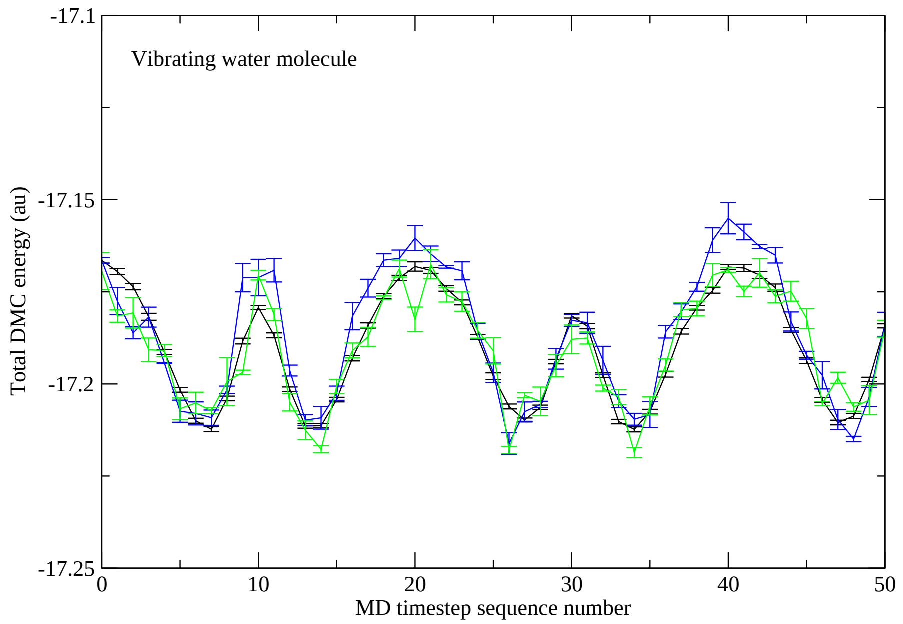
Some illustrative calculations (not GM-MD!)



Discrete DMC energies for vibrating water molecule. Black curve: converged DMC result with small error bar. Red curve: 1000 walkers (a 'typical' number), 1000 equil moves, 3 stats moves. Green curve: 14400 walkers, 1000 equil moves, 3 stats moves

- The red curve essentially does not follow the accurate curve. It can be made to do so by using a sufficiently large population of walkers - the green curve repeats the red calculations, but using 14400 walkers (14 times more) and the same number (3) of stats accumulation moves. This is effectively what GM do in order to claim '*the same energies as conventional discrete QMC sampling*' (though note the error bars in both the red and green curves are not accurate, there being insufficient data - only 3 moves worth - to calculate them properly).
- Because they do not mention the number of walkers used, it is possible to misconclude the nature of the speedup from their data e.g. if we normally do 3000 moves and now we're doing 3, we might think that a GM-MD calculation is 1000 times faster, but it's not because we're using 14 times more configs than usual.

Some GM-MD results



Discrete and GM-MD DMC energies for vibrating water molecule. Black curve: converged discrete DMC result with small error bar. Green curve: 14400 walkers, 1000 equil moves, 3 stats moves. Blue curve: 14400 walkers, no equil moves (each MD point restarted from previous), 3 stats moves.

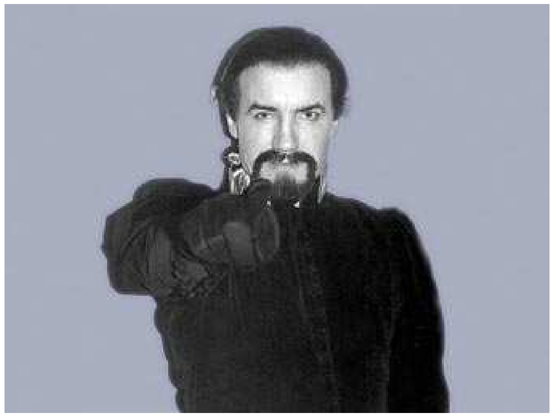
- A GM-MD restarted calculation is an *approximation* to the previous green curve (repeated above) (the approximation is done in order to save the time normally spent doing DMC equilibration).
- The new blue curve above is the GM-MD result with 14400 configs and 3 moves, but skipping the equilibration step for all MD steps after the first, and restarting from the equilbrated walkers from the previous step. Walker reweighting and spacewarping were both turned on.
- The GM-MD curve follows the green curve - where the equilibration is done explicitly - quite well (though its agreement with the full black curve is possibly not as good as that of the green curve, which is expected, as the reweighting is an additional approximation).
- Note that the approximation does not appear to 'get worse' over the course of MD time i.e. the error does not build-up.

Equilibration of walkers

Question: Can configs read at the restart be properly equilibrated for the new Ψ in so few moves?

- Can do 'extreme GM-MD' by just reweighting the configs when you read them in, recomputing the energy with the reweighted configs, and then not doing any DMC accumulation moves at all. The energies will be shifted compared to those of the original DFT calcs, but this is obtained almost in its entirety by the full DMC equilibration carried out at the initial nuclear configuration.
- Can reproduce demonstration calculation in GM paper this way. (where GM-MD total energies are shown tracking the discrete QMC energies for vibrating Si_xH_y molecules). The DMC energy curve simply runs parallel to the DFT energies. DMC does not demonstrate any new physics besides a shift in the total energy. This shift is obtained by the initial equilibration, then it is simply translated on along the trajectory. Nothing gained. Why do DMC-MD at all in this case?
- One might suspect that this issue should be important when calculating a trajectory where DMC shows a feature in the energy landscape that is not present in DFT. The GM approach ought simply to 'iron out' these features because the distribution does not have enough time to equilibrate into the special quantum-correlations that cause the energy differences.
- In the end, if you want to get any meaningful DMC energies along a DFT trajectory, it is essential to do some re-equilibration for each MD step to allow the population to properly respond to the new nuclear configuration. If this re-equilibration is too short, any DMC-specific features will be smeared out in the DMC-energy curve.
- Note that equilibration is an exponential process and the timescale is determined by physics rather than the initial distribution. The initial distribution largely determines the magnitude of the error in the DMC energy, not the rate at which it decays. Hence if you want to do a good job of equilibrating (significantly improve your distribution), *you always need to equilibrate for at least the time taken to diffuse across the longest physically relevant length scale*.

GM-MD conclusions



- Rather than using 1000 times fewer stats accumulation moves than normal, and 10 times more configs than normal (say), then we would get the same amount of statistical sampling by using 100 times fewer stats accumulation moves with the usual number of configs. However, in the latter case, we have more propagation in imaginary time, and therefore we expect the wave function to adjust better to the new nuclear configuration. So why not do this? Certainly it sounds more impressive to say 'you only need to do 2 moves' without mentioning the larger number of configs required.
- Despite the fact that having a large number of walkers relative to the number of moves might facilitate parallelization, it remains the case that the total CPU time does not change when (number of walkers $\times$ number of DMC steps) is fixed.
- One might expect that it would be more efficient to sample widely differing nuclear configurations to get a thermodynamic average, rather than ones extremely close together as done in GM-MD.
- It would be nice to see - stated clearly and succinctly in a few lines, why we *expect* GM-MD to give us any speedup at all, even for thermodynamic averages.

What to do next with this..? Hmmmm..