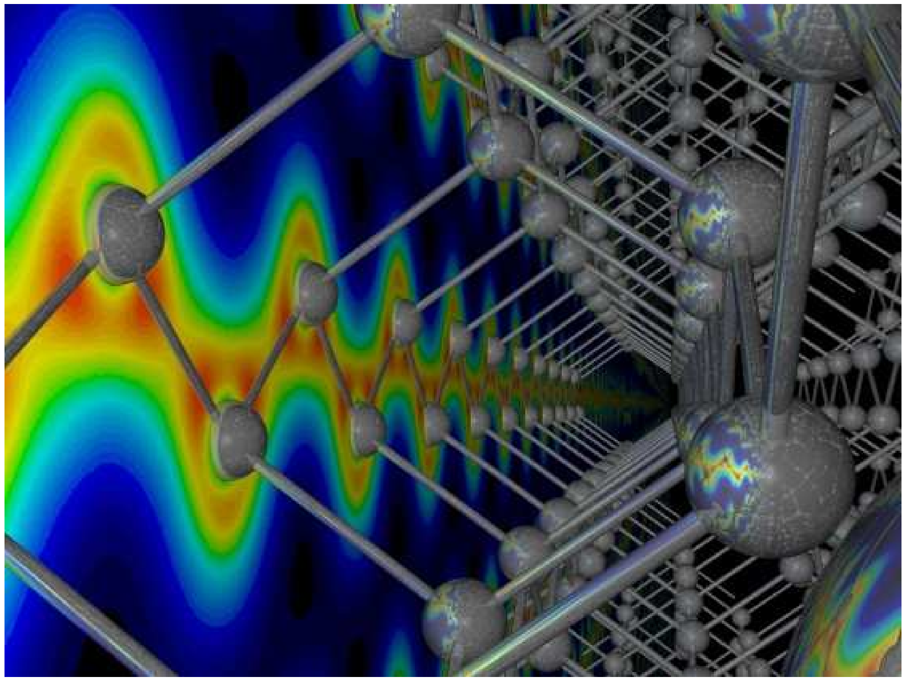
Perfect parallel scaling for quantum Monte Carlo on hundreds of thousands of cores

Electronic Structure Discussion Group, January 2012



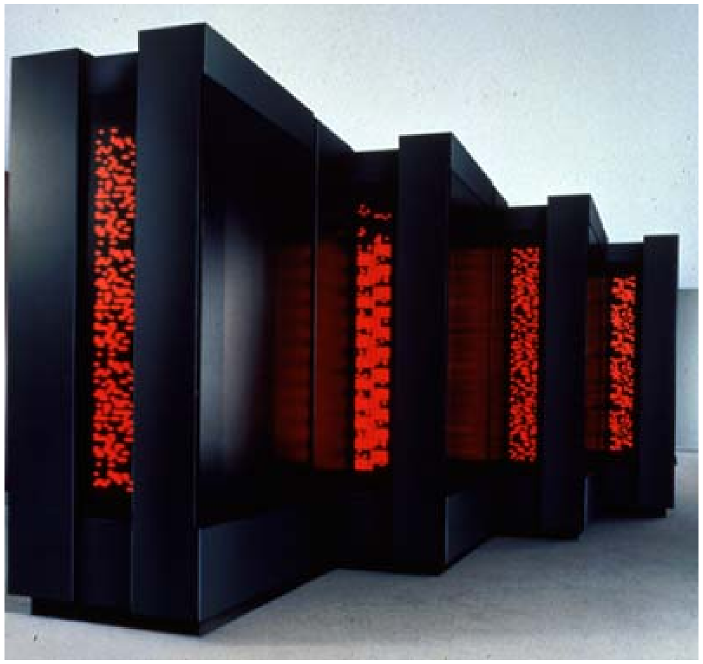
Mike Towler

TCM Group, Cavendish Laboratory, University of Cambridge

QMC web page: `www.tcm.phy.cam.ac.uk/∼mdt26/casino2.html`

Email: `mdt26@cam.ac.uk`

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.

- Here in TCM we develop the *quantum Monte Carlo* (QMC) method – in the form of the CASINO code – to perform ultra-high accuracy electronic structure calculations. We cannot pretend this is the cheapest technique in the world, and 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 an important question:

How does QMC scale with the number of processors?

And consequently, how many processors can we successfully exploit?

Parallel computing

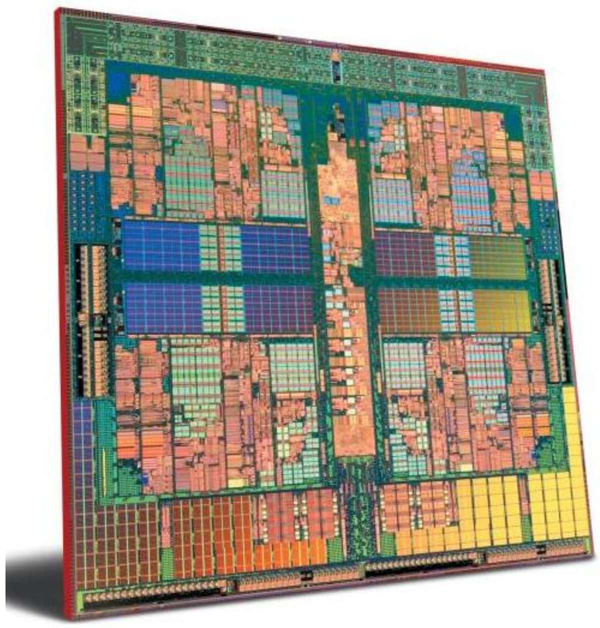
From the mid-1980s until 2004 computers got faster because of *frequency scaling* (more GHz). However, faster chips consume more power, and ever since power consumption (and consequently heat generation) became a significant concern, parallel computing has become the dominant paradigm in computer architecture,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

- Here in TCM we develop the *quantum Monte Carlo* (QMC) method – in the form of the CASINO code – to perform ultra-high accuracy electronic structure calculations. We cannot pretend this is the cheapest technique in the world, and 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 an 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.

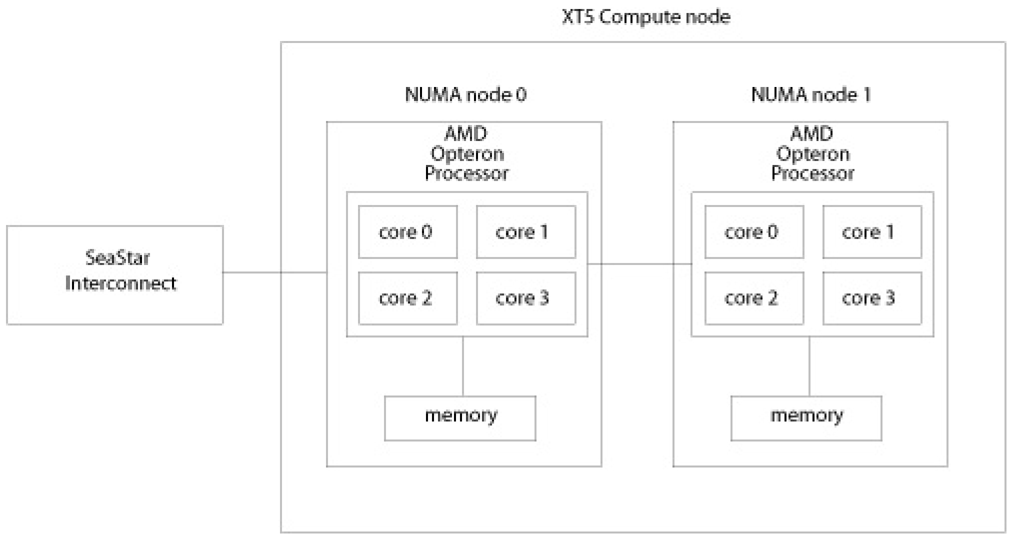
Core: the cores within the processor perform the actual mathematical computations. A core can do a certain number (typically 4) of FLOPs or FLoating-point OPerations every time its internal clock ticks. These clock ticks are called cycles and measured in Hertz (Hz). Thus a 2.5-GHz processor ticking 2.5 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 2011) is the *K computer* at the Riken Institute in Japan (548352 cores, 8.162 petaflops/s).

A usable example: Jaguar

Jaguar is/was** a Cray XT5 machine at Oak Ridge National Laboratory in Tennessee, USA. It has a peak performance of around 1.75 petaflops, and has 224256 AMD Opteron processor cores, making it the third-fastest computer in the world (January 2012).

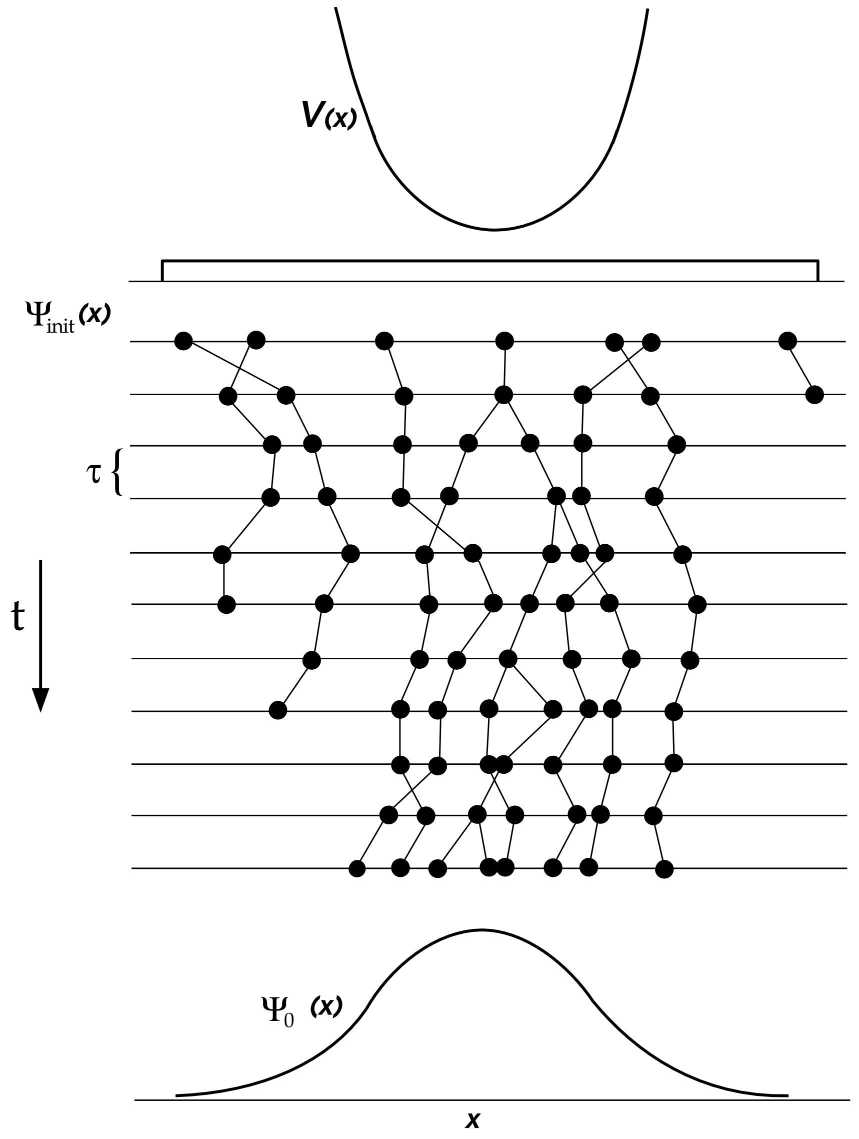
It is made of 18,688 XT5 compute nodes. Each such node contains two 6-core AMD Opteron processors and 16 Gb of memory. From CASINO's point of view it is thus a shared-memory machine with 12 cores per node. It was the fastest computer in the world until October 2010.

```
ssh -X mdt26@jaguarpf.ccs.ornl.gov ; cd CASINO ; make Shm
```

```
runqmc -p 224256 --shmem=12 --walltime=6h30m
```

** [Jaguar currently being transformed into 'Titan': 18688 16-core Opteron nodes (299008 cores), 32 Gb memory per node, and lots of GPUs (of which more later)].

What you need to know about QMC to understand this talk



- *Diffusion Monte Carlo* (DMC): trial many-body wave function can be made to evolve towards correct ground state wave function by 'evolving it in imaginary time'.
- Wave function represented by distribution in configuration space of an ensemble of copies of the system (each member of the ensemble is called a 'config' or a 'walker').
- The 'shape' of the wave function is changed by deleting configs that move into high-energy regions, and by duplicating ones that move into low-energy regions, according to some magic algorithm.

CASINO is parallelized by dividing the number of walkers over the cores

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) 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

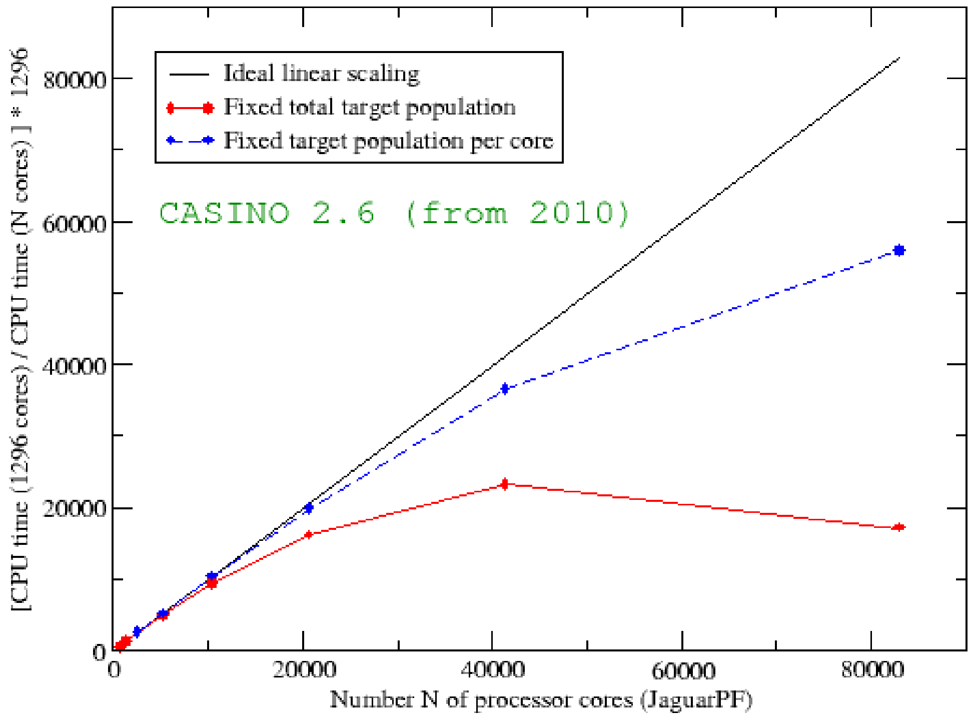
- Increase number of configs per core (without blowing the memory).
- Use weighted DMC (**lwdmc** keyword) to reduce branching (with the default weight limits of 0.5 and 2.0) and disable transfer or large arrays (such as inverse Slater matrices) between cores by using the **small_transfer** keyword.

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, 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 does CASINO scale?



*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

Rendering the earlier formal analysis somewhat redundant, 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 me, Mike Gillan 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 – yet this was never mentioned in 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

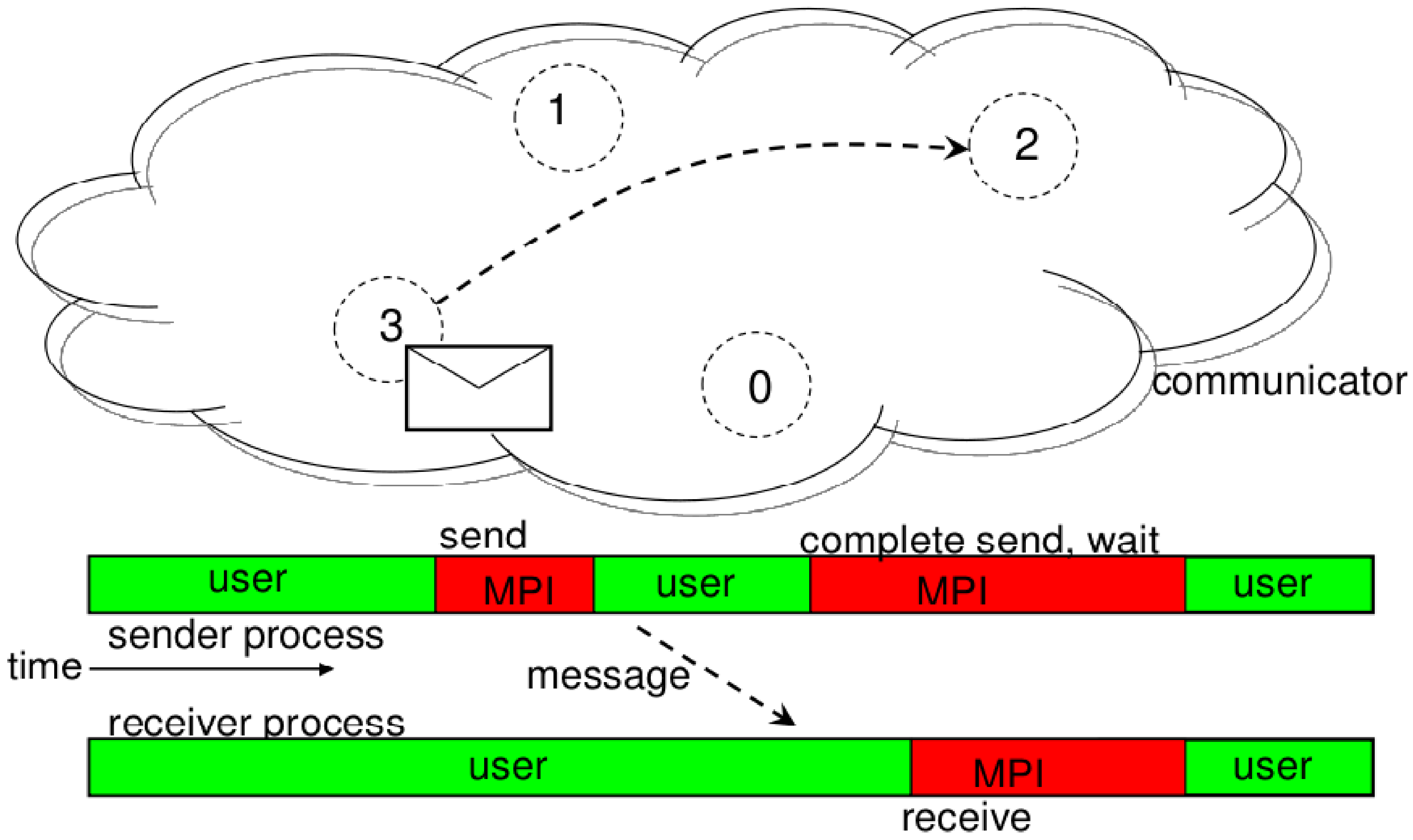
A communication call is said to be non-blocking if it may return before the operation completes (a *local* concept on either sender or receiver). A communication is said to be asynchronous if its execution proceeds at the same time as the execution of the program (a *non-local* concept).

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

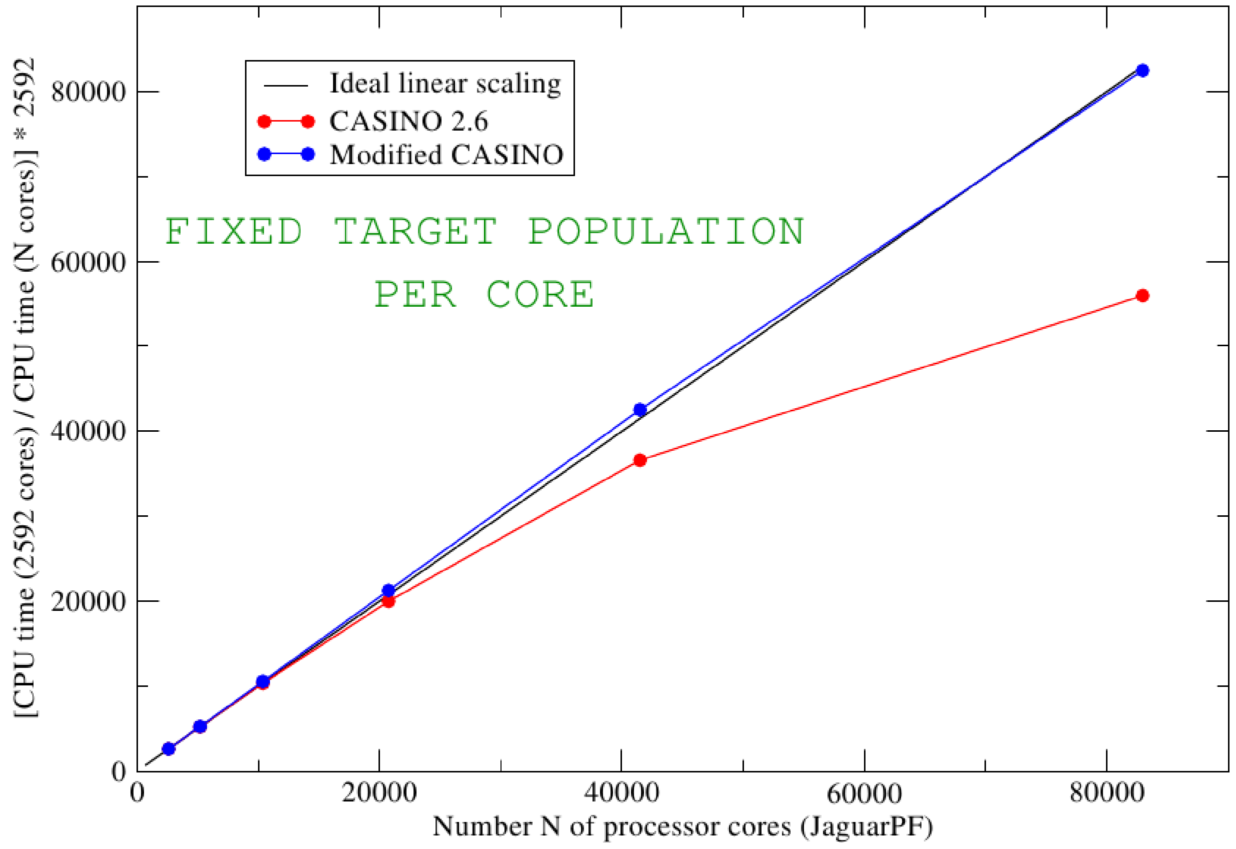
```
MOVE 1
- Move all currently existing configs forward by one time step
- Compute the multiplicities for each config (the number of copies of each
 config to continue in the next move).
- Looking at the current populations of config on each processor, and at the
 current multiplicities, decide which 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

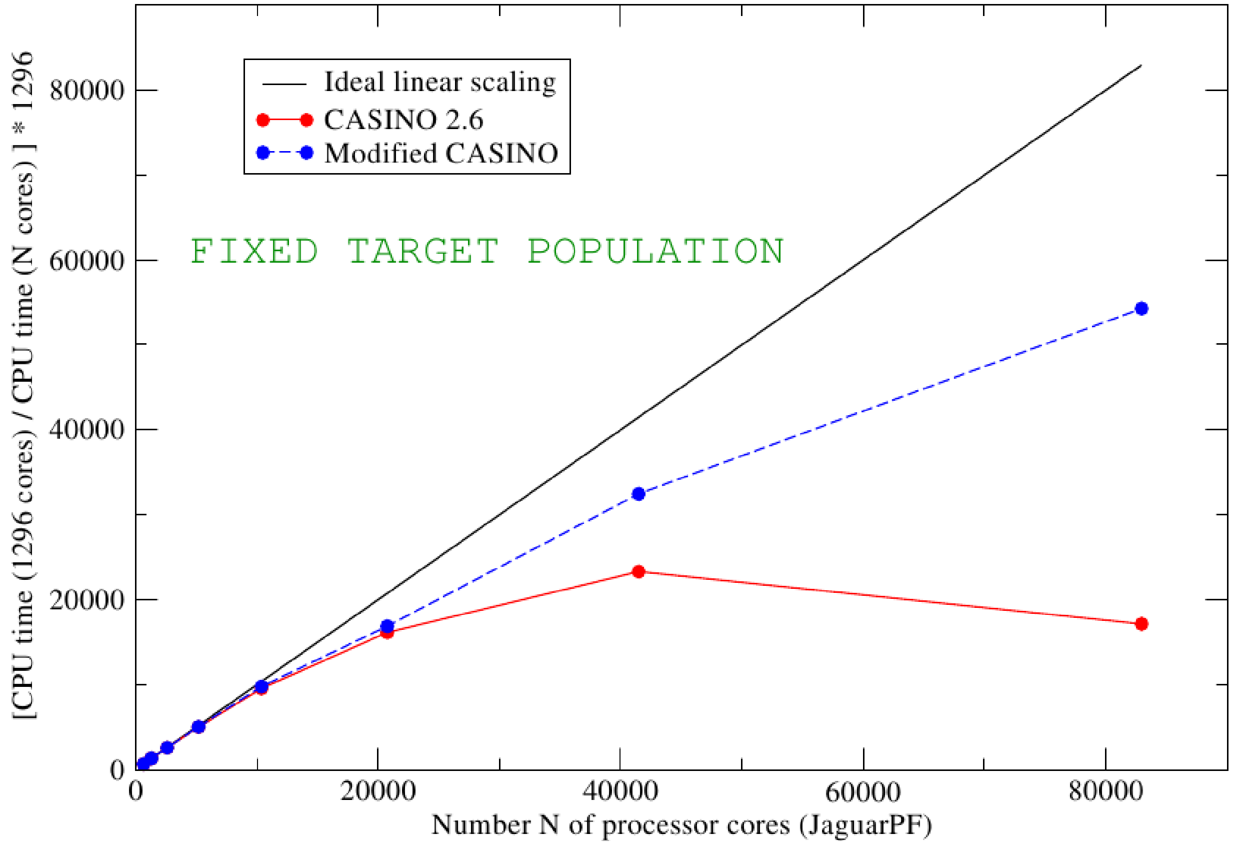
Table 1: *CPU time taken to carry out operations associated with redistribution of configs between cores in CASINO 2.6 (2010) and in my modified version, during one twenty-move DMC block for a water molecule adsorbed on a 2d graphene sheet.*

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).

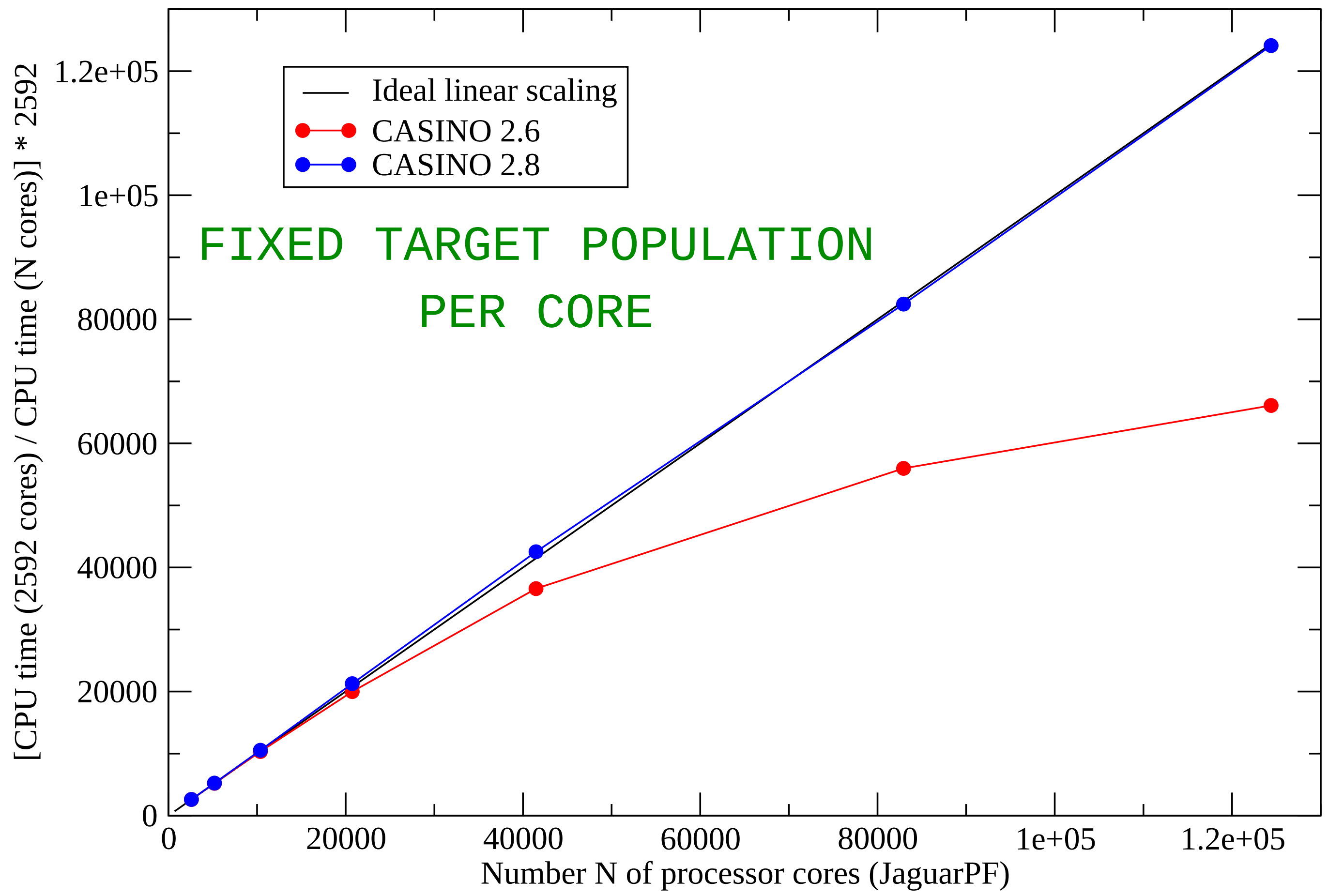
..if you give the processors enough to do



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 224256 cores of the Jaguar machine, if we could be bothered to sit through the queueing time.

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 (so we can reblock the data).
- 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.

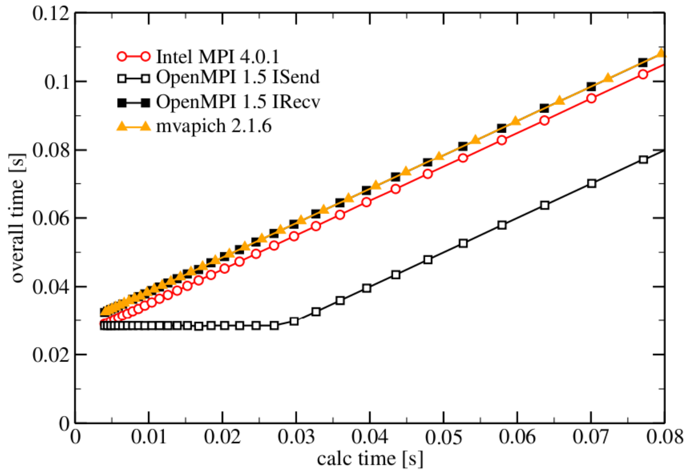
Is non-blocking communication really asynchronous?

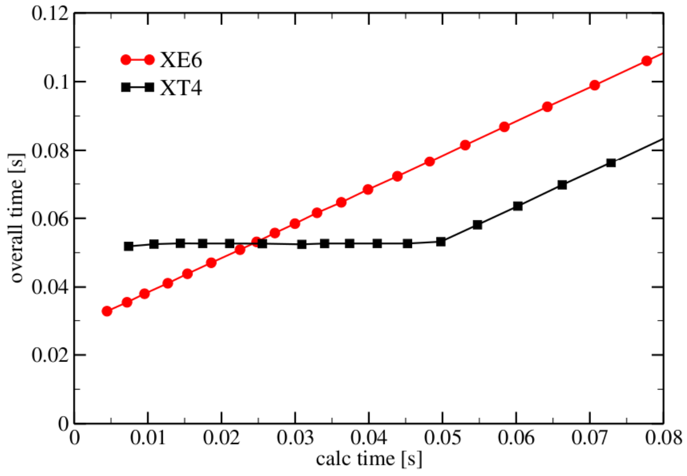
Not necessarily! MPI standard doesn't *require* non-blocking calls to be asynchronous. Two problems:

(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`





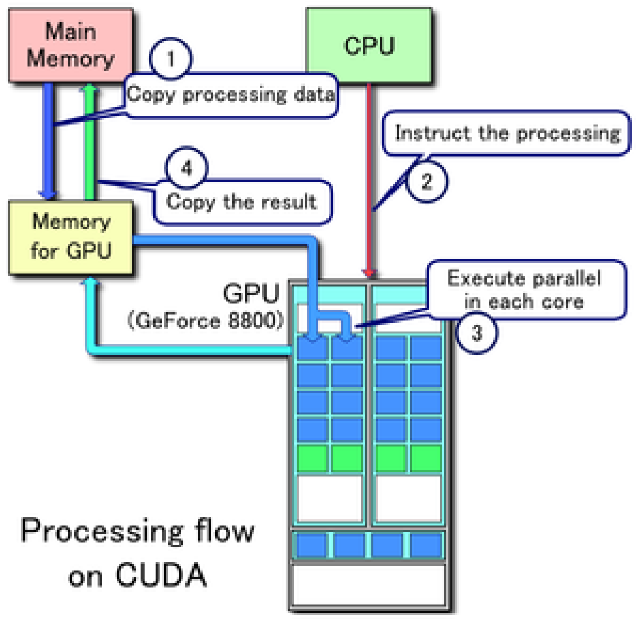
If your MPI doesn't provide true asynchronous progress, then some form of periodic poll through a MPI_TESTALL operation may be required to achive optimal performance. Can also overlap computation and comms via mixed-mode OpenMP/MPI - use dedicated communication thread.

The future? GPUs

- 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. Three of the five fastest machines in the top 500 - including the Chinese Tianhe-1 that is currently in second place (and faster than Jaguar) - use GPUs in their design. Jaguar itself is currently 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 software developers through variants of industry-standard programming languages. Programmers typically use 'C for CUDA' (C with Nvidia extensions and certain restrictions) to code algorithms for execution on the GPU.

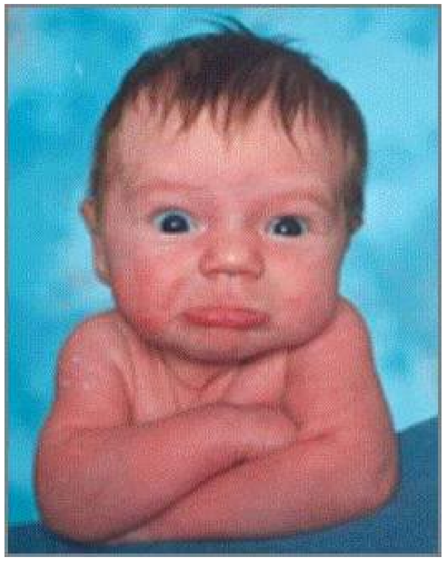
CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs so that they become accessible for computation like CPUs. Unlike CPUs however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly,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

Cray are also doing some interesting things:
`www.hpcwire.com/hpcwire/2011-05-24/cray_unveils_its_first_gpu_supercomputer.html`

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. I did. You might consider, for example:

The INCITE program

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

The European DEISA program

`www.deisa.eu`

Conclusions

- In general it seems to be the case that, following my modifications, 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.
- People need to start rewriting their codes to use GPUs, if they haven't already.
- 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.