Final Report: Large-scale Quantum Monte Carlo simulations for Chemistry **PRACE Preparatory Access Call PA0356**

Michel Caffarel, Anthony Scemama

—

Laboratoire de Chimie et Physique Quantiques

CNRS-IRSAMC, Université de Toulouse, France

—

April 20, 2011

Abstract

In this document various tests realized on the Curie machine thanks to the preparatory access call are presented. The runs performed have allowed us to test the scalability of our quantum Monte Carlo code and also to improve it to get the optimal efficiency. Before running on Curie our program had already been tested up to 512 cores and presented a very good scalability. Thanks to this preparatory access it has been possible to run in parallel a much larger number of cores (up to 10000) In this new regime a number of bottlenecks have appeared and were cured. Using the results of these benchmarks, we estimate that in real (production) situations our code is expected to have a wall-time parallel efficiency close to 99% with 10 000 CPU cores, and this efficiency will get even better in the near future. These results confirm the fact that quantum Monte Carlo simulations are indeed ideally suited to massive parallelism with maximal scalability.

Quantum Monte Carlo (QMC) methods are known to be powerful stochastic approaches for solving the Schrödinger equation. Although they have been widely used in computational physics during the last twenty years, they are still of marginal use in computational chemistry. Two major reasons can be invoked for that:

1. the N-body problem encountered in chemistry is particularly challenging (a set of strongly interacting electrons in the field of highly-attractive nuclei)

2. the level of numerical accuracy required is very high (the so-called "chemical accuracy").

In practice, DFT methods are the most popular approaches, essentially because they combine both a reasonable accuracy and a favorable scaling of the computational effort as a function of the number of electrons. On the other hand, post-HF methods are also employed since they lead to a greater and much controlled accuracy than DFT. Unfortunately, the price to pay for such an accuracy is too high to be of practical use for large molecular systems.

QMC appears as a third promising alternative method essentially because it combines the advantages of both approaches: a favorable scaling together with a very good accuracy. In addition to this, the QMC approaches are ideally suited to High-Performance-Computing (HPC) and, more specifically, to massively parallel computations. As most "classical" or "quantum" Monte Carlo approaches, the algorithm is of the number crunching type, the memory requirements remain small and bounded and the I/O flows are marginal. Due to these extremely favorable computational aspects plus the rapid evolution of computational infrastructures towards more and more numerous and efficient processors, it is likely that QMC will play in the next years a growing role in computational chemistry.

1 Overview of a QMC simulation

A walker is a vector $\mathbf{X}$ of the $3N$-dimensional space containing the entire set of the three-dimensional coordinates of the N electrons of the molecular system. During the simulation, a walker (or a population of walkers) samples via a Monte Carlo Markov Chain process the $3N$-dimensional space according to some target probability density (the precise density may vary from one QMC method to another). From a practical point of view, the averages of the quantities of interest (energy, densities, etc.) are calculated over a set as large as possible of independent random walks. Random walks differ from each other only in the initial electron positions $\mathbf{X}_0$, and in the initial random seed S_0 determining the entire series of random numbers used.

In QMC=Chem, the quantum Monte Carlo program developed by our group, the main computational object is a *block*. In a block, N_{walk} independent walkers realize random walks of length N_{step}, and the quantities of interest are averaged over all the steps of each random walk. If N_{step} is significantly larger than the auto-correlation time (which is usually rather small), the positions of the walkers at the end of the block can be considered as independent of their initial positions and a new block can be sampled using these configurations as $\mathbf{X}_0$ and using the current random seed as S_0.

The final Monte Carlo result is obtained by averaging all the results obtained for each block. If the data associated with each block are saved on disk, the averages can be calculated by post-processing the data and the calculation can be easily restarted using the last positions of the walkers and the last random seeds.

Note that the computation of the averages does not require any time ordering. If the user provides a set of N_{proc} different initial conditions (walker positions and random seed), the blocks can be computed in parallel. In figure 1, we give a pictorial representation of four independent processors computing blocks sequentially, each block having different initial conditions.

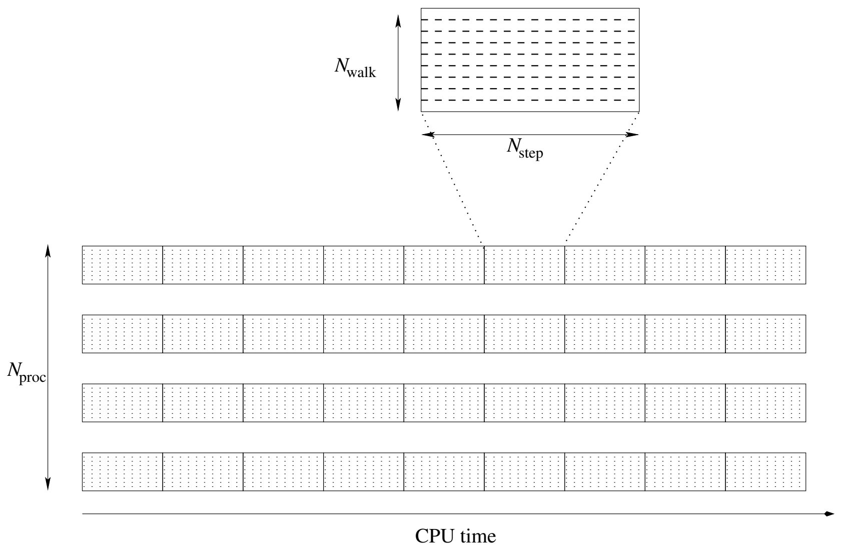


Figure 1: Graphical representation of a QMC simulation. Each process generates blocks, each block being composed of N_{walk} walkers realizing N_{step} Monte Carlo steps.

2 Design of QMC=Chem before we start the preparatory access

QMC=Chem was designed specifically to run efficiently both on heterogeneous clusters via MPI and in grid environments using Python scripts. The memory requirements, disk input/outputs and network communications were minimized as much as possible, and the code was written in order to allow asynchronous processes (figure 2) using the manager/worker model.

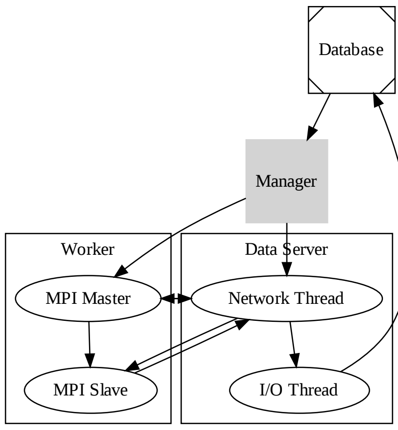


Figure 2: Overview of QMC=Chem before the preparatory access.

When the program starts its execution, the manager runs on the master node and forks two other processes: a worker process and a data server. The worker is an efficient Fortran/MPI executable with minimal memory and disk space requirements (typically a few tens of megabytes for each), where the only MPI communication is the broadcast of the input data (wave function parameters, initial positions in the $3N$-dimensional space and random seed). The outline of the task of a worker is the following:

```
while ( Running )
{
    compute_a_block_of_data();
    Running = send_the_results_to_the_data_server();
}
```

The data server is an XML-RPC server implemented in Python. When it receives the computed data of a worker, it replies to the worker the order given by the manager to compute another block or to stop. The received data is then stored in a database using an asynchronous I/O mechanism. The manager is always aware of the results computed by all the workers and controls the running/stopping state of the workers and the interaction of the user during the simulation.

3 Design of QMC=Chem after the preparatory access

This preparatory access allowed us to discover several bottlenecks that appeared when using a very large number of processors.

3.1 Network communication improvements

First, we observed with ∼2 000 cores that the native Python XML-RPC server is not efficient enough. A lower level RPC server has been re-written from scratch and gives satisfactory results up to 10 000 cores.

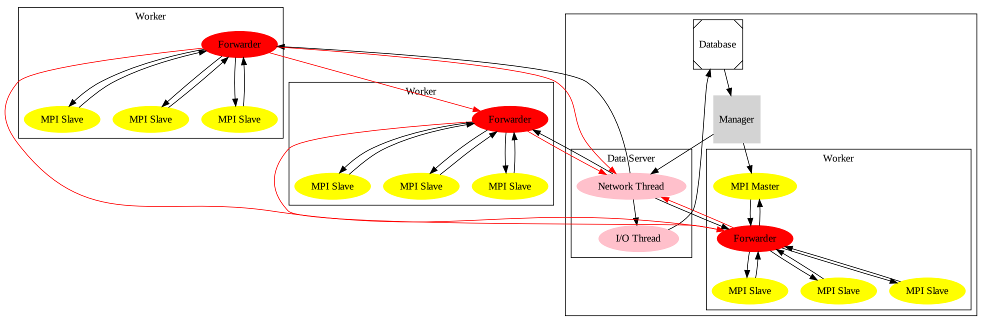


Figure 3: Overview of QMC=Chem after the preparatory access.

Then, we noticed that with our original design, 10 000 workers would yield 10 000 connections to the data server with small messages. We introduced forwarders, as shown in figure 3. Each compute node has now one forwarder that accepts the data from all the workers running on the same node, and forwards the data to the data server. As on the Curie machine there are 32 cores per node, this first step reduced by 32 the number of connections to the data server, and each connection contains a larger message. In addition, it also reduced the time spent in communications on the compute node: as soon as the forwarder has received the data, the workers are ready to start the calculation of the next block. They don't have to wait until the data has arrived to the data server.

Finally, the number of connections to the data server has been reduced even more by adding a feature to the forwarders. When a forwarder is instantiated, it receives from the data server the list of all the other existing forwarders. When the forwarder has data to send, it will send it randomly to any other forwarder of this list or to the data server: The first forwarder will always send the data to the data server, the second forwarder will send data to the data server or to the first forwarder, etc. This mechanism makes fewer and larger messages to be received by the data server. At that point, the communications are no longer a bottleneck.

3.2 I/O improvements

During a run, the I/O thread of the data server writes the computed data to disk in a BSD database file. The keys contain the rank of the processor that computed the block and the index of the block on this processor. The values contain the computed data. The manager simultaneously reads this file to be aware of the current status of the calculation. In order to perform an update of the running averages, the manager has to find which blocks were added since the last update. Before the preparatory access, this was done with a $\mathcal{O}(N \times n)$ algorithm, where N is the number of keys in the file (the number of computed blocks) and n is the number of blocks to consider for the update, proportional to the number of workers. This algorithm was changed using hash tables and the manager updates are now realized in $\mathcal{O}(N)$ time.

4 Benchmark results

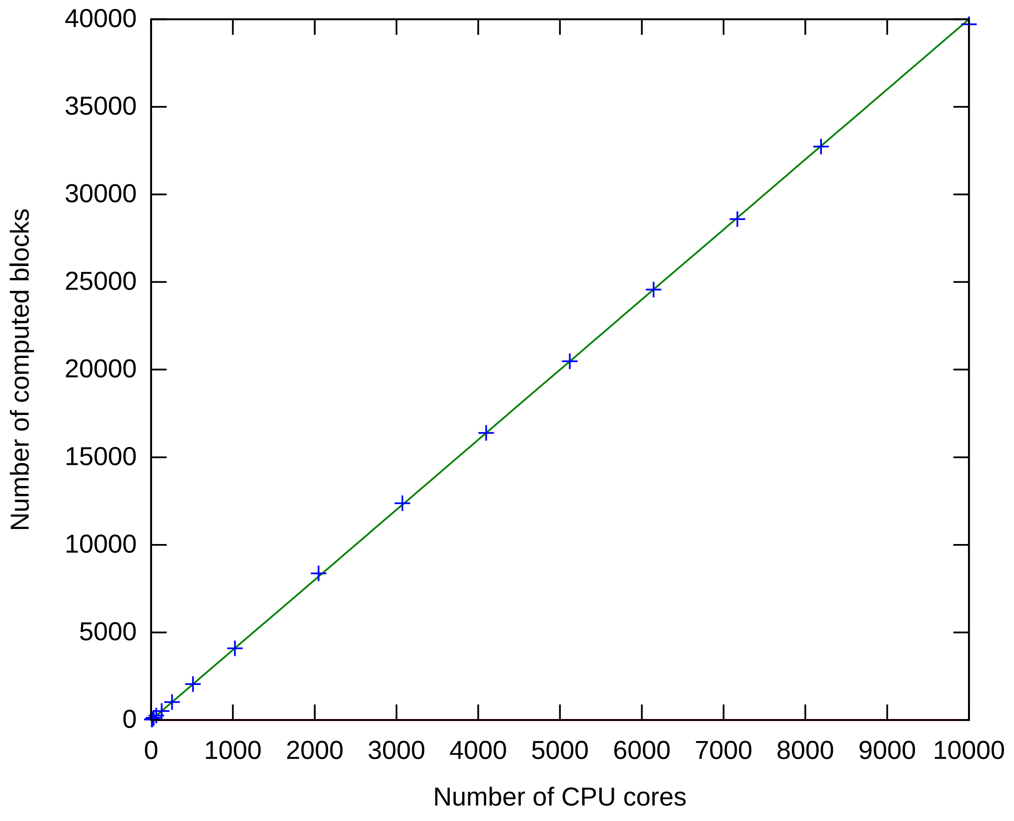
4.1 Simulation details

An input file was prepared for a wave function of the $CuCl_2$ molecule. The disk occupation of the input files is 1.5 MB for this molecular system. The disk occupation corresponding to the results of one computed block is independent of the block parameters (length and number of random walks), and is equal to 1.2 KB. The memory required by one core to compute one block does not depend either on the block parameters, and was measured equal to 32 MB. The CPU time used by one core to compute a block is proportional to the number of random walks and to the number of Monte Carlo steps.

The calculation was set up such that a block is made of 10 walkers realizing 2 000 Monte Carlo steps. The CPU time needed to compute one block is 84 seconds, and the stopping condition of the calculation is “wall time > 5 minutes”. After 5 minutes, a soft termination signal is sent to all the workers. The workers then finish the computation of the current block, send to the manager the result corresponding to the block as well as the final positions of the walkers and the last random seed.

Note that in real applications, the typical number of random walks per block is around 100, and the number of Monte Carlo steps is in the $[2\,000, 50\,000]$ range. Hence, the CPU time per block is multiplied by a factor of 10–250 keeping all the rest is identical (memory, I/Os, network traffic, etc.) These benchmark results are representative of a very bad conditioning of the simulation where the I/O and network loads are too large by more than an order of magnitude.

4.2 Results



N_{cores}	N_{blocks}	CPU time	Wall time
8	32	2638.24	352
16	64	5243.97	352
32	128	10487.02	349
64	256	21070.63	357
128	512	42717.23	361
256	1024	84581.99	361
512	2048	167951.37	366
1024	4096	338153.60	368
2048	8192	673271.54	382
3072	12288	1016684.54	402
4096	16383	1370949.05	410
5120	20480	1697630.23	418
6144	24570	2033701.44	434
7168	28668	2387299.03	465
8192	32736	2721079.24	474
9999	39713	3543935.63	483

As expected, the number of computed blocks scales linearly with the number of CPU cores. However, the wall time increases and this will be discussed below.

A 20% speed-up was observed when only one worker was running on one (8-core) processor. This may be twofold:

- The "turbo" feature of the CPUs may have been activated
- As the L3-cache per processor is 24 MB large and the requested memory per core is 32 MB, most of the data fits in the cache.

Therefore, the data for one to four workers have not been reported

4.3 Wall time analysis

Following Amdahl's law, a perfect speed-up in terms of wall time is not achievable. The need to broadcast the input data, retrieve the final data and write it to disk is essentially a serial procedure that can't be avoided.

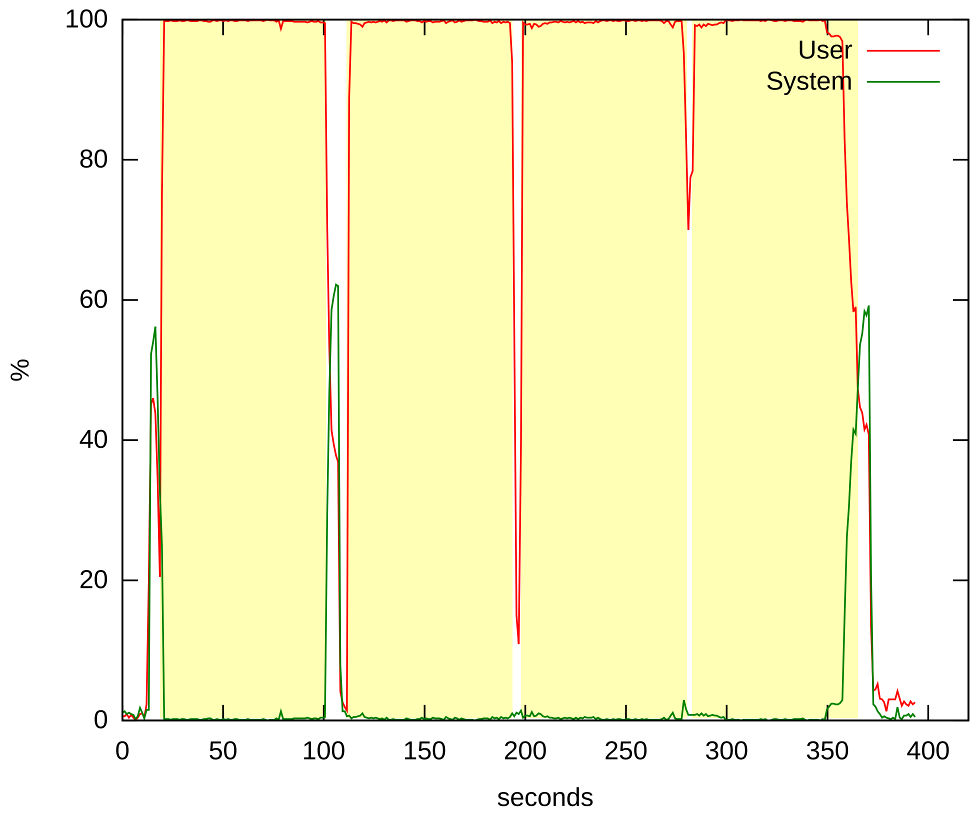
Initialization. In our program, the following serial procedures are performed before the calculation of the blocks calculation starts. A preliminary check of the consistency of the input file is performed, and a CRC32 key is computed from the input data. In this way it will be further possible to check that when data arrives to the data server it corresponds to the same input. Then the MPI process is started. The input data is read from disk by the master MPI process and sent to the MPI slaves. Then the input data is processed before the block calculation starts (matrix sparsifications, cusp fitting, etc...).

Calculation. At that point, all the MPI processes work independently. When a process has finished to compute a block, it calls a Python script to send the computed data to the forwarder, and starts the next block. Meanwhile, the forwarders route the computed results to the data server. On the data server, each network connection initiates a new thread and the received data is appended to a queue. In parallel, the I/O thread of the data server pops elements from this queue to write the data to disk.

Termination. When the forwarders run, they regularly send a heartbeat to the data server. The reply from the data server is the order to perform a new block or to stop. Upon a stopping request (because the user asks for it, or because the stopping condition has been reached), all the forwarders are informed. The workers continue to run, finish their block, and launch the Python script to send the data to the forwarder. At that point, they obtain the information to stop. Then, they send back the current random seed and the last positions of the walkers. The time required by the data server to receive all the walker positions and random seeds is a serial procedure, affecting the speed-up.

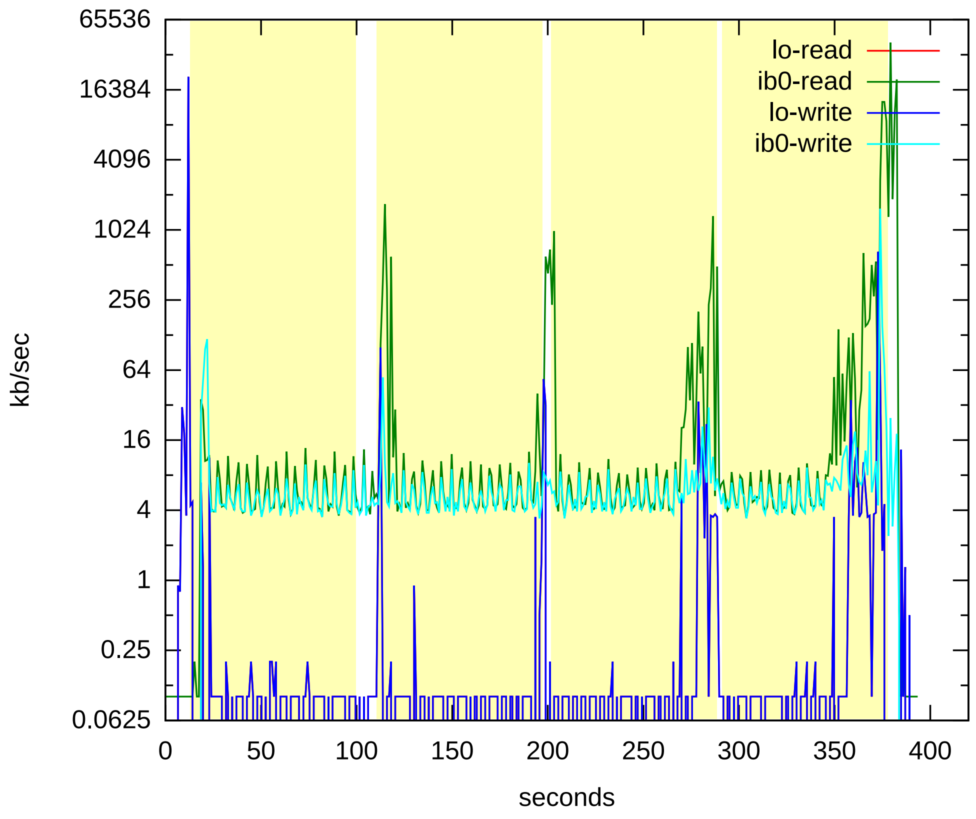
To better understand why the wall time increases, several measures have been realized on the master node of a 4096-core run.

4.3.1 CPU usage



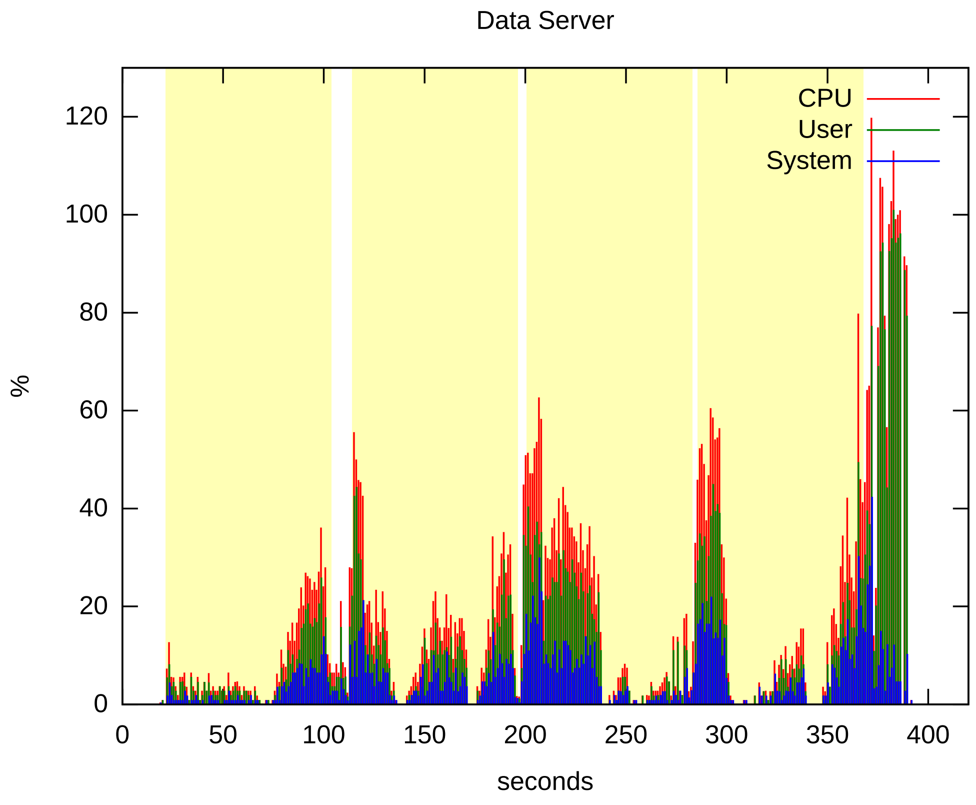
On this graph, 100% corresponds to the full CPU utilization of the 32 cores. The time spent to compute the blocks is colored in yellow. From this data, one can see that 16 seconds are required for the initialization step. The termination of the last worker is shifted from the termination of the first worker by 10 seconds. 22 additional seconds are needed to receive and store the final walker positions and store them to disk. On average, 2 seconds are needed for the Python script to send the data to the forwarder at the end of each block.

4.3.2 Network usage



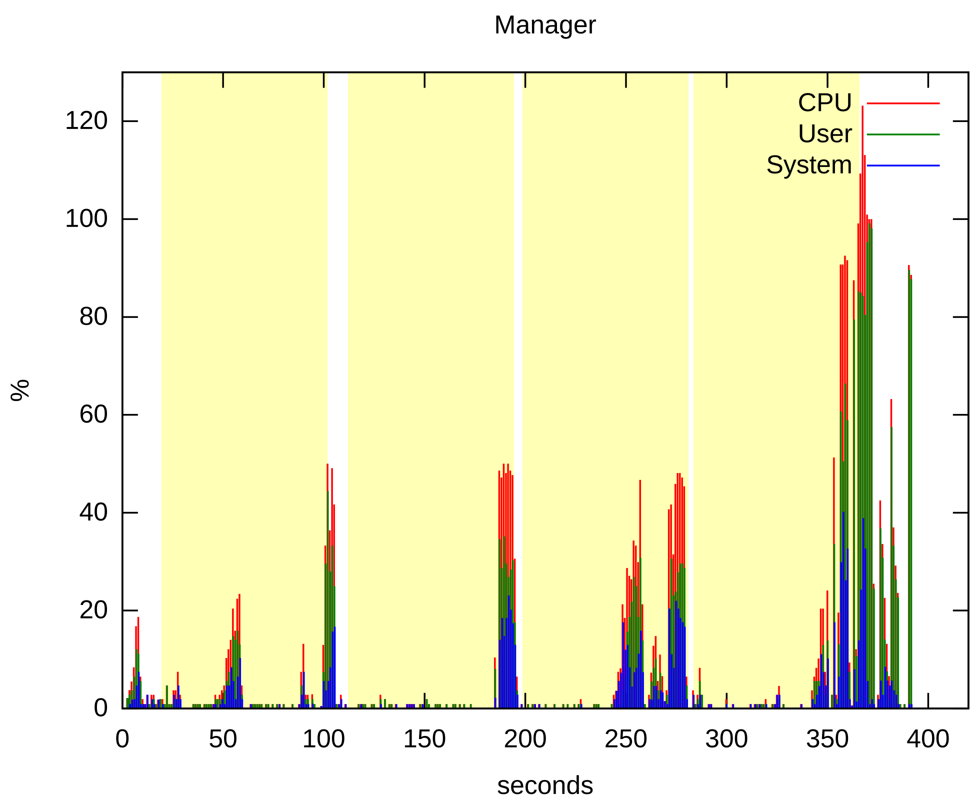
The lo network interface is used for intra-node communications and the ib0 interface is used for inter-node communications. The base-line of ib0-read and ib0-write is around 8 KB/sec. This comes from the heartbeat of the forwarders and the reply of the data server. The network communications after the end of the blocks can be clearly identified, and one can remark that the maximum of the peak happens when the next block has already started, giving evidence that the routing of the data to the data server is done in parallel with the computation of the blocks. One can also note an important traffic starting at 350 seconds, corresponding to the retrieval of the final data. As the runs on 8–32 cores last 349–352 seconds, we can conclude that the wall time increase with the number of cores is mainly due to the finalization step.

4.3.3 CPU usage of the data server



On this graph, 100% corresponds to the CPU utilization of one core. This graphs shows that the data is stored in the database in parallel with the calculation (system time). A large amount of CPU time is needed during the retrieval of the final data. This point will be investigated.

4.3.4 CPU usage of the manager



5 seconds are necessary to check the input files, compute the CRC32 key and start the data server. Then, the peaks correspond to a updates of the running averages from the database.

4.3.5 Summary and conclusions

The parallel part of the calculation is spent to compute blocks. The block parameters can be adjusted by the user and don't impact the serial part of the program. The serial part depends only on the number of computed blocks, and does not depend on the block parameters. This serial part contains input checking (constant, 5 seconds), MPI initialization (3–6 seconds on 4096–8192 cores), input pre-processing (constant, 7 seconds), data transfer from the workers to the forwarders (2 seconds/block), finalization (the largest part, depending on the number of walkers and the number of cores).

When the number of computed blocks becomes large, the CPU loads of the manager and the data server are non-negligible. As these two processes run on the first compute node they share their CPU resources with the running workers, and the additional wall time can be computed as:

$$\text{Additional wall time} = (\text{CPU time per block} \times N_{\text{blocks/core}}) \left(\frac{N_{\text{processes/node}}}{N_{\text{cores/node}}} - 1 \right) \tag{1}$$

On these benchmarks, the workers running on the first compute node can in the worst case finish 21 seconds after all the other workers. This explains the 10 second shift observed in the termination of the workers on the CPU-load data.

The additional wall time needed per computed block can be estimated by taking the asymptote of:

$$\text{Wall time per block} = \left(\text{Total Wall Time} - \frac{N_{\text{blocks/core}}}{N_{\text{blocks}}} \times \text{CPU time} \right) \times \frac{1}{N_{\text{blocks}}} \tag{2}$$

when the number of cores becomes large (figure 4). The additional wall-time for one block is estimated equal to 4 milliseconds.

Using this data, one can estimate the wall time required for a real calculation on 10 000 cores and evaluate the parallel efficiency. Blocks will be made of 100 walkers realizing 10 000 steps. The CPU time per block is expected to be 70 minutes. If the stopping condition is "wall time $>$ 8 hours", 8 hours and 10 minutes will be spent in the parallel part to compute 7 blocks per CPU core. The serial part will contain the $\sim$20 second initialization, and 14 seconds of data transfer from the workers to the forwarders. In total, 70 000 blocks will be computed and the additional wall time will be estimated equal to 280 seconds for the finalization step. The total wall time will be equal to 495.23 minutes, and the total CPU time will be equal to 4 900 000 minutes. Hence, the parallel efficiency will be close to 99%. With this estimation, our code is now ready to run efficiently on Curie.

5 Future improvements

Some improvements have already been implemented but have not be tested on Curie since the allocation was exceeded. The Python script used to send the data to the forwarders which is far too slow on these benchmarks (2 seconds) has now been accelerated by a factor of 10. Some of the work of the data server at the finalization step (the sorting of the final walker positions, for instance) has also been delegated to the forwarders and is now done in parallel. Too many read-accesses to the database have been observed due to unnecessary updates of the manager. These have been identified and were

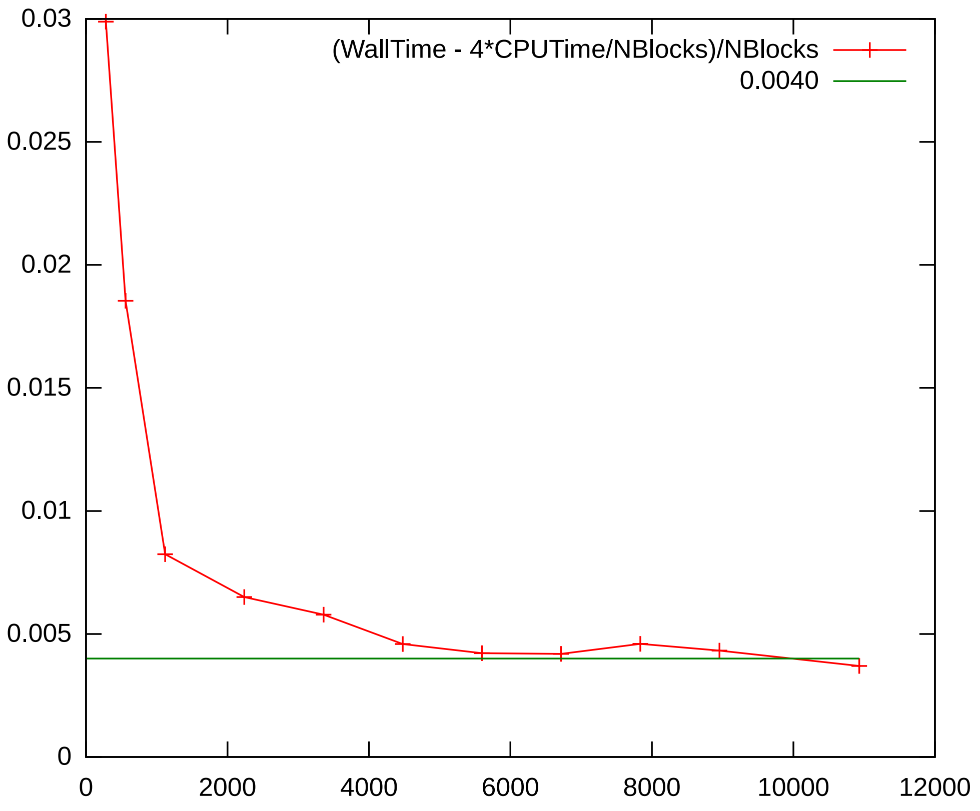


Figure 4: The asymptote of this curve is an estimation of the additional wall-time needed per computed block.

removed just before the end of the preparatory access. An improvement of 6 wall-time seconds was measured with 4096 cores.

In the near future, the following improvements will be made. The forwarders will be organized in a binary tree structure to gain in efficiency for the routing of the results to the data server. Some redundancies have also been found in the stored data and the disk occupation of the database will readily be decreased, reducing also the I/O bandwidth and CPU load of both the manager and the data server. This should reduce the additional wall-time per computed block, which is the main source of speed-up degradation.