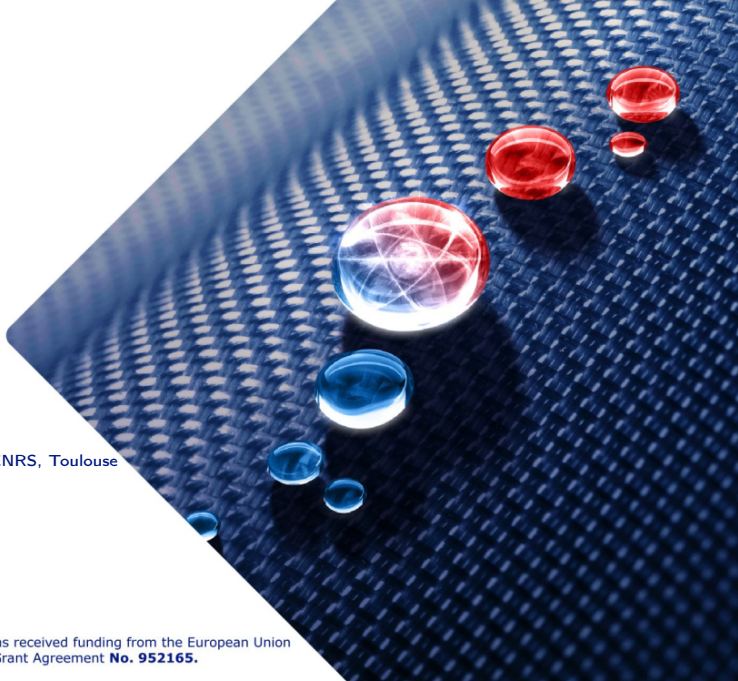
Targeting Real chemical accuracy at the EXascale

Exascale algorithms

Anthony Scemama

03/03/2022

Lab. Chimie et Physique Quantiques, IRSAMC, UPS/CNRS, Toulouse (France)



Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizoon 2020 research and innovation programme under Grant Agreement **No. 952165.**

Difficulties at the Exascale

HPC *vs* Grid/Cloud

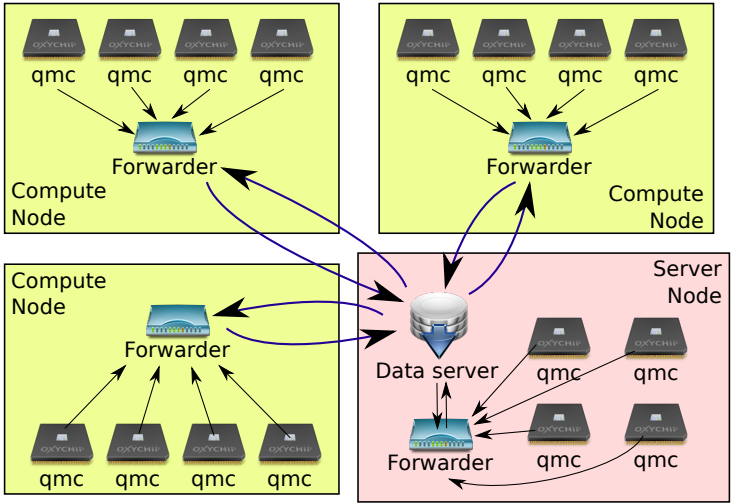
HPC	Grid/Cloud
Reliable network	Internet network
Very low latency	Very high latency
Homogeneous hardware	Heterogeneous hardware
Network topology is known	Network topology unknown
Tightly-coupled parallelism	Embarrasingly parallel
The hardware is assumed reliable	Hardware is unreliable
Efficient for synchronous applications	Asynchronous

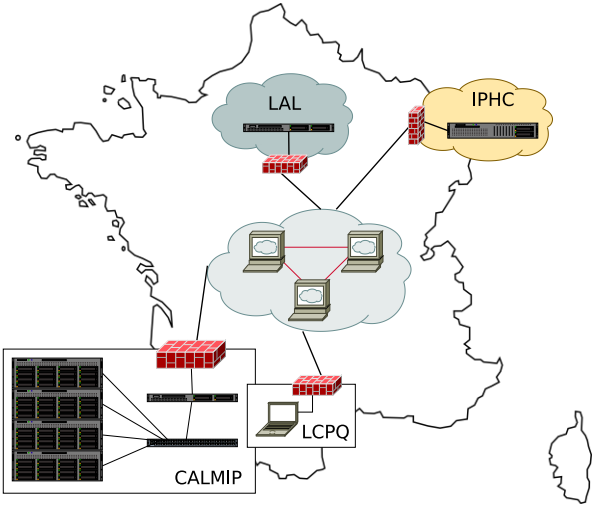
- When a single MPI process crashes, the whole simulation is killed
- Failure is inexistent in MPI design

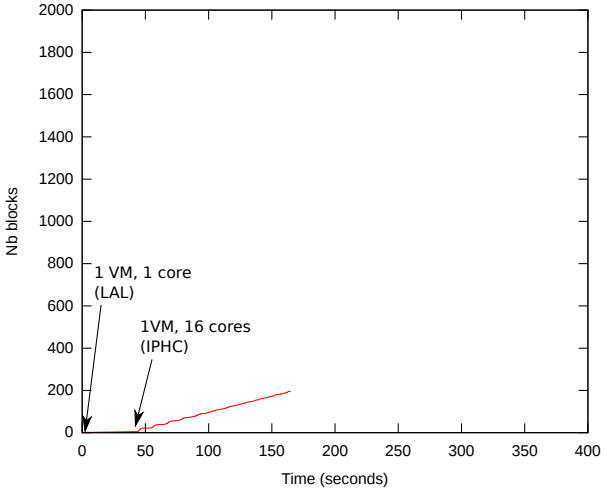
- Only the compute peak performance makes a 1000×
- Heterogeneous hardware (GPUs)
- Latencies are problematic (network, GPU)
- If we normalize to compute speed, everything becomes slow

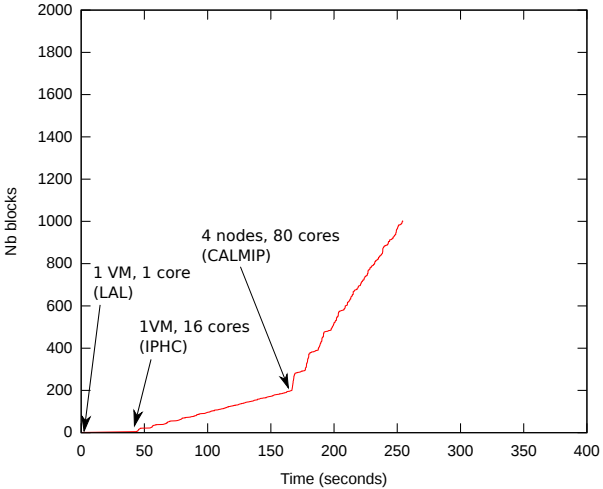
Possible solution

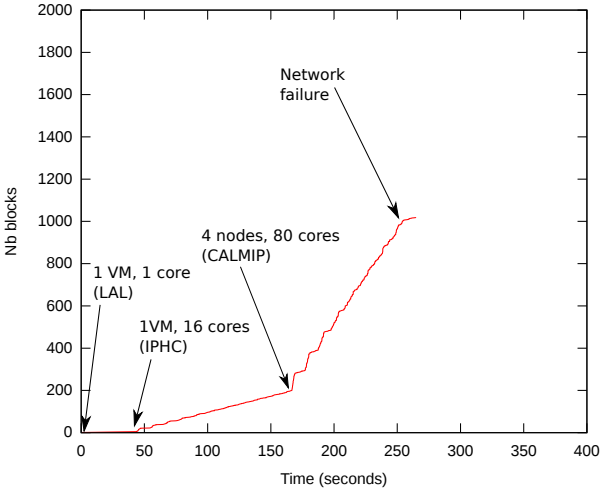
Grid/Cloud algorithms can be good candidates for exascale

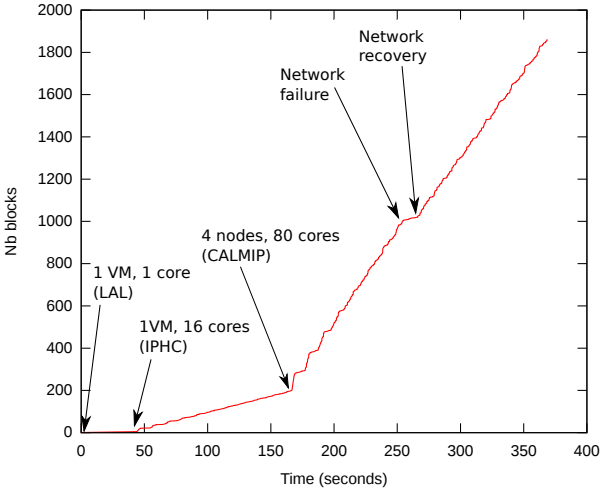

qmc
qmc
qmc
qmc
Forwarder
Compute
Node
qmc
qmc
qmc
qmc
Forwarder
Compute
Node
Compute
Node
Forwarder
qmc
qmc
qmc
qmc
Server
Node
Data server
qmc
qmc
Forwarder
qmc
qmc


LAL
IPHC
LCPQ
CALMIP

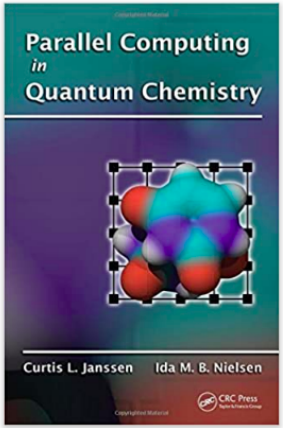

2000
1800
1600
1400
1200
1000
800
600
400
200
0
Nb blocks
1 VM, 1 core
(LAL)
1VM, 16 cores
(IPHC)
0
50
100
150
200
250
300
350
400
Time (seconds)


Nb blocks
Time (seconds)
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
50
100
150
200
250
300
350
400
1 VM, 1 core
(LAL)
1VM, 16 cores
(IPHC)
4 nodes, 80 cores
(CALMIP)


Nb blocks
Time (seconds)
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
50
100
150
200
250
300
350
400
Network
failure
4 nodes, 80 cores
(CALMIP)
1 VM, 1 core
(LAL)
1VM, 16 cores
(IPHC)


Nb blocks
Time (seconds)
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
50
100
150
200
250
300
350
400
Network recovery
Network failure
4 nodes, 80 cores (CALMIP)
1 VM, 1 core (LAL)
1VM, 16 cores (IPHC)

Can we do traditional quantum chemistry with a Grid?



2.2.4.7 Ad Hoc Grid

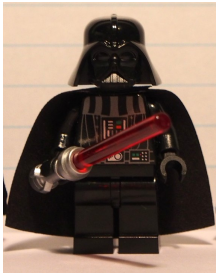
An *ad hoc grid* topology typically arises in computing environments that were not purpose-built to function as a single computer. The nodes in the grid are loosely coupled, as illustrated in Figure 2.14; a node may be administratively independent of the other nodes and may be distant as well, perhaps even connected over the Internet. The field of grid computing concerns itself with authentication, resource scheduling, data movement, and loosely-coupled computation in such environments. A grid network's performance, however, is too low (the bisection width is too small and the latency too large) to be of direct interest for the quantum chemistry applications discussed in this book.

Example of Quantum Chemistry on a grid: Quantum Package

MPMD : Multiple Program / multiple data

- One executable : the task scheduler
- One executable : the master compute process (OpenMP)
- One/Many executable(s) : slave compute processes (MPI/OpenMP, 1 process/node)
- One process to tunnel data through different networks
- Inter-process communication with ZeroMQ

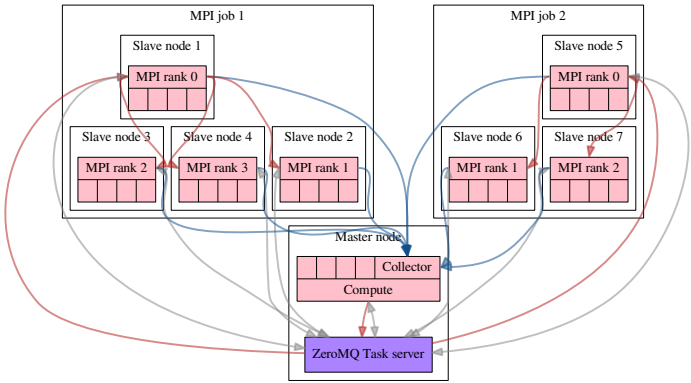
Design



Master

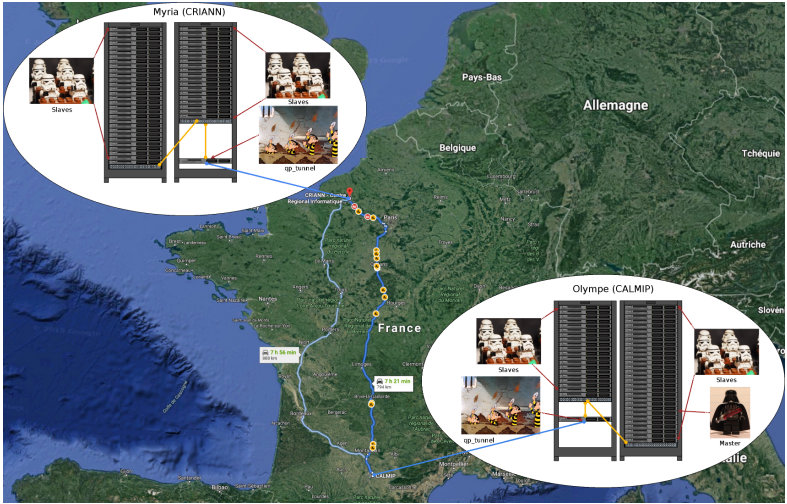
Slave

Tunnel



Each task is computed with all possible OpenMP threads.

Experimental setup



Bandwidth

CALMIP login	CALMIP compute	IB EDR 100Gb/s
CRIANN login	CALMIP login	Renater : 74.1 MB/s
CRIANN login	CRIANN compute	Omnipath 100GiB/s

Latency (ping)

CALMIP login	CALMIP compute	0.09 ms
CRIANN login	CALMIP login	16.72 ms
CRIANN login	CRIANN compute	0.23 ms

- Size of the vectors : $N = 21\,691\,814$, 109 tasks
- 412 MiB sent to each MPI group at the beginning
- 165 MiB sent to each MPI group per Davidson iteration
- 1.5 MiB as a result of a task
- Starting from a bad guess : $[1\ 0\ \ldots\ 0\ 0] \longrightarrow$ 17 iterations

Configuration	N_{core}	Wall time
40 nodes Olympe	1440	36:51
40 nodes Myria	1120	44:10
20 nodes Myria, 20 nodes Olympe	1280	43:48

- Size of the vectors : $N = 21\,691\,814,\ 21\,854\,665$ tasks
- Stop when relative error is 0.1% $\longrightarrow \sim 3\%$ of the tasks
- 412 MiB sent to each MPI group at the beginning
- Each task returns 40 bytes
- Each ZeroMQ client fetches m tasks, where m is dynamically adjusted such that the computation of the m tasks takes $\sim N_{\text{core}}$ seconds.
- The next m tasks are prefetched during the current computation

Configuration	N_{core}	Wall time
50 nodes Olympe	1800	11:58
50 nodes Myria	1400	14:07
25 nodes Myria, 25 nodes Olympe	1600	13:19

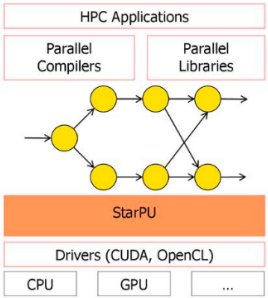
- We should not fight against the latency, and accept it
- We have seen that asynchronous task-based algorithms can accept very high latencies
- Can we use a more HPC-friendly solution?
  - GPI/GAPSI: efficient + fault tolerant one-sided communications
  - StarPU: Task-based parallelism

- StarPU uses this paradigm, and uses MPI (more efficient, no fault tolerance)
- It can distribute tasks on CPUs and GPUs (or both)

The StarPU runtime system

The need for runtime systems

- "do dynamically what can't be done statically anymore"
- Compilers and libraries generate (graphs of) tasks
  - Additional information is welcome!
- StarPU provides
  - Task scheduling
  - Memory management



Software using StarPU

Some software is known for being able to use StarPU to tackle heterogeneous architectures, here is a non-exhaustive list (feel free to ask to be added to the list!):

- **AL4SAN**, dense linear algebra library
- **Chameleon**, dense linear algebra library
- **Exa2pro**, Enhancing Programmability and boosting Performance Portability for Exascale Computing Systems
- **ExaGeoStat**, Machine learning framework for Climate/Weather prediction applications
- **FLUSEPA**, Navier-Stokes Solver for Unsteady Problems with Bodies in Relative Motion
- **HiCMA**, Low-rank general linear algebra library
- hmat, hierarchical matrix C/C++ library
- **K'Star**, OpenMP 4 - compatible interface on top of StarPU.
- **KSVD**, dense SVD on distributed-memory manycore systems
- **MAGMA**, dense linear algebra library, starting from version 1.1
- **MaPHyS**, Massively Parallel Hybrid Solver
- MASA-StarPU, Parallel Sequence Comparison
- **MOAO**, HPC framework for computational astronomy, servicing the European Extremely Large Telescope and the Japanese Subaru Telescope
- **PaStiX**, sparse linear algebra library, starting from version 5.2.1
- PEPPHER, Performance Portability and Programmability for Heterogeneous Many-core Architectures
- **QDWH**, QR-based Dynamically Weighted Halley
- **qr_mumps**, sparse linear algebra library
- **ScalFMM**, N-body interaction simulation using the Fast Multipole Method.
- **SCHNAPS**, Solver for Conservative Hyperbolic Non-linear systems Applied to PlasmaS.
- **SignalPU**, a Dataflow-Graph-specific programming model.
- **SkePU**, a skeleton programming framework.
- **StarNEig**, a dense nonsymmetric (generalized) eigenvalue solving library.
- **STARS-H**, HPC low-rank matrix market
- **XcalableMP**, Directive-based language eXtension for Scalable and performance-aware Parallel Programming

Reasonable solutions for exascale:

- Use StarPU within a small group of nodes: MPI/CPU/GPU task distribution
- Interconnect multiple MPI simulations with GPI/GASPI or ZeroMQ to enable fault tolerance

Asynchronous Algorithms for DMC and FCIQMC

```
E = 0.
for kStep in range(nSteps):
    newCoordinates = []

    for iWalker in range(nWalkers):
       x_old = coordinates[iWalker]
       x = DiffusionDrift(x_old)

       eWalk[iWalk] = Energy(x)
       E += eWalk[iWalk]
       w = exp(-timeStep * (Energy(x) - E_ref))
```

```
       if w > 1.: # Random death of the walker
          if random.uniform(0.,1.) < w:
             newCoordinates.append(x)

       else: # Random duplication
          if random.uniform(0.,1.) < w-1.:
             newCoordinates.append(x)

    coordinates = newCoordinates
    E_ref = f(eWalk)
return E / nSteps
```

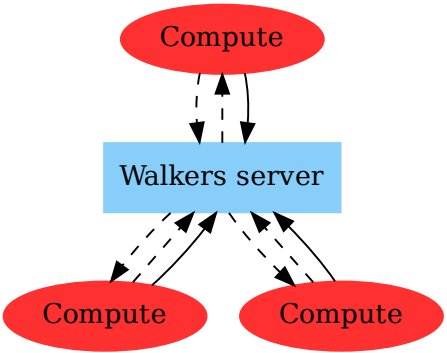
- Walkers have all performed the same number of steps
- Load balancing problems

```
for kStep in range(nSteps):
    w[iWalker] = 1.

    for iWalker in range(nWalkers):
       x_old = coordinates[iWalker]
       x = DiffusionDrift(x_old)
       coordinates[iWalker] = x

       E += w[iWalker] * Energy(x)
       sumWeight += w[iWalker]
       w[iWalker] *= exp(-timeStep * (Energy(x) - E_ref))

    # end for iWalker
# end for kStep
return E / sumWeight
```

- Works with a single walker
- No need to synchronize walkers $\Longrightarrow$ embarrasing paralellism
- But: weight `w` goes to zero or infinity $\Longrightarrow$ unstable

Asynchronous Diffusion Monte Carlo algorithm

```
E = 0.
sumWeight = 0.
while (continueRun):

    newCoordinates = []
    for iWalker in range(nWalkers):
       w = 1.
       x = coordinates[iWalker]
       for kStep in range(nSteps): # <- Max nSteps
          x = DiffusionDrift(x)

          E += w * Energy(x)
          sumWeight += w
          w *= exp(-timeStep * (Energy(x) - E_ref))

          if w < 0.5:    # <- Death threshold
             if random.uniform(0.,1.) < w:
                newCoordinates.append(x)
             break
          if w > 2.0: # <- Birth threshold
             newCoordinates.append(x)
             w -= 1.
       # end for kStep

       if w > 1.:
          newCoordinates.append(x)
          w -= 1.
       if random.uniform(0.,1.) < w:
          newCoordinates.append(x)

    # end for iWalker
    coordinates = newCoordinates
# end while
return E / sumWeight
```


Compute
Walkers server
Compute
Compute

```
E = 0.
sumWeight = 0.
# Non-blocking coordinates request
promise = asyncFetchWalkers(server, nWalkers)
# Wait for initial coordinates to arrive
coordinates = asyncWait(promise)
while (continueRun):
    # Request next coordinates
    promise = asyncFetchSomeWalkers(server, nWalkers)
    for iWalker in range(nWalkers):
       w = 1.
       x = coordinates[iWalker]
       for kStep in range(nSteps):
          x = DiffusionDrift(x)
          sumWeight += w
          E += w * Energy(x)
          w *= exp(-timeStep*(Energy(x)-E_ref))
          if w < 0.5:
             if random.uniform(0.,1.) < w:
                asyncSendCoordinates(server, x)
             break
          if w > 2.0:
             asyncSendCoordinates(server, x)
             w -= 1.
       # end for kStep
       if w > 1.:
          asyncSendCoordinates(server, x)
          w -= 1.
       if random.uniform(0.,1.) < w:
          asyncSendCoordinates(server, x)
    # end for iWalker
    coordinates = asyncWait(promise)
# end while
return E / sumWeight
```

Advantages

- 1 walker per node is possible: full GPU acceleration
- No load balancing problem
- No synchronization required
- Fault tolerance: Any compute node can crash
- Trajectories can be stopped and requeued to improve ergodization
- Multiple walker servers can be added for redundancy, and organized as a network
- One VMC trajectory can be implemented as one StarPU task