Parallel Strategies for FCIQMC

Fionn Malone, James Spencer, Mathew Foulkes and Derek Lee
Imperial College London

28th July 2014

FCIQMC

FCIQMC

- Full Configuration Interaction Quantum Monte Carlo (FCIQMC) (Booth, Thom, Alavi 2009) is a relatively recent projector QMC method working in the space of Slater Determinants.

FCIQMC

- Full Configuration Interaction Quantum Monte Carlo (FCIQMC) (Booth, Thom, Alavi 2009) is a relatively recent projector QMC method working in the space of Slater Determinants.
- Can treat Hilbert spaces orders of magnitude larger than exact diagonalisation.

FCIQMC

- Full Configuration Interaction Quantum Monte Carlo (FCIQMC) (Booth, Thom, Alavi 2009) is a relatively recent projector QMC method working in the space of Slater Determinants.
- Can treat Hilbert spaces orders of magnitude larger than exact diagonalisation.
- Relatively straightforward to parallelise.

FCIQMC

- Full Configuration Interaction Quantum Monte Carlo (FCIQMC) (Booth, Thom, Alavi 2009) is a relatively recent projector QMC method working in the space of Slater Determinants.
- Can treat Hilbert spaces orders of magnitude larger than exact diagonalisation.
- Relatively straightforward to parallelise.
- It scales to 1000s of cores so we can use big computers.

Big Computers

Archer: 2014, cray XC30, 76000 cores, 1.5 petaflops, £43, 000, 000

www.archer.ac.uk

Big Computers

Fionn: 2014, 7600, 147 teraflops, € 3.7,000,000

www.ichec.ie

Can we use them?

Can we use them?

- There is an upper limit to how many parallel processes you can use (Amdahl's law) before the efficiency drops significantly.

Can we use them?

- There is an upper limit to how many parallel processes you can use (Amdahl's law) before the efficiency drops significantly.
- QMC can often make use of much larger machines compared with DFT or exact diagonalisation.

Can we use them?

- There is an upper limit to how many parallel processes you can use (Amdahl's law) before the efficiency drops significantly.
- QMC can often make use of much larger machines compared with DFT or exact diagonalisation.
- Two of the biggest barriers to improved scaling are load imbalances and communication overhead.

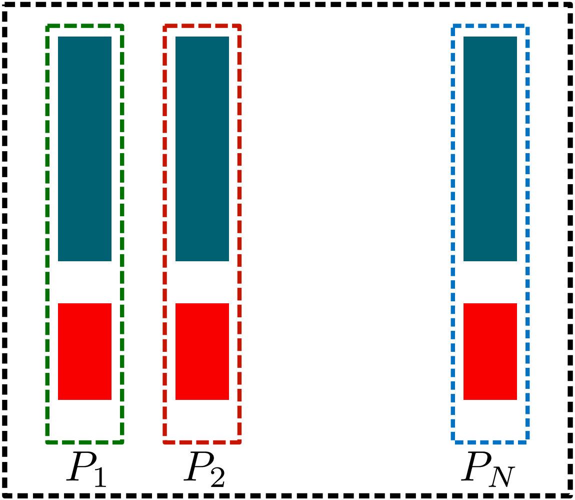
Can we use them?

- There is an upper limit to how many parallel processes you can use (Amdahl's law) before the efficiency drops significantly.
- QMC can often make use of much larger machines compared with DFT or exact diagonalisation.
- Two of the biggest barriers to improved scaling are load imbalances and communication overhead.
- The use of non-blocking communications and improved load balancing was successful for CASINO (Gillan, Towler, Alfe) so can we use similar ideas here?

Parallel Strategies

Parallel Implementation (Booth, Smart, Alavi 2014)

- Distribute list of occupied determinants across all processors.

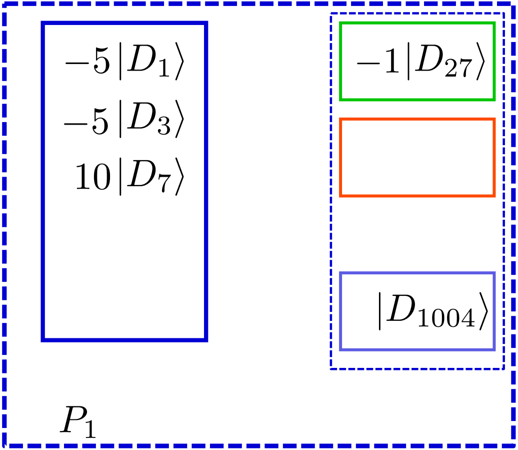


Parallel Implementation

- Each processor evolves main list and spawn into spawned walker list.

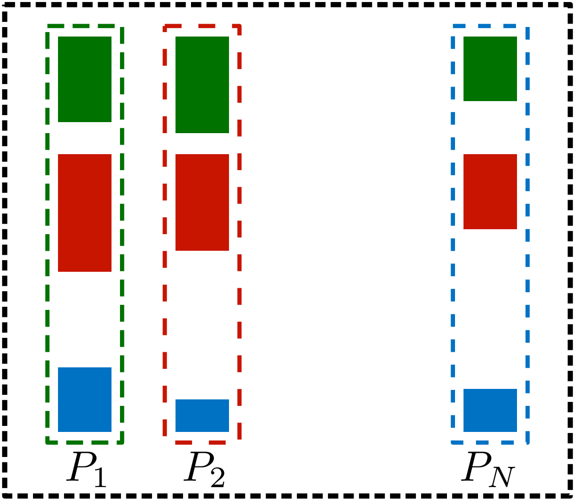
$$p_{\text{spawn}}(i|j) \propto \Delta\tau |H_{ij}|$$

$$p_{\text{die/clone}}(i|i) \propto \Delta\tau |H_{ii} - S|$$



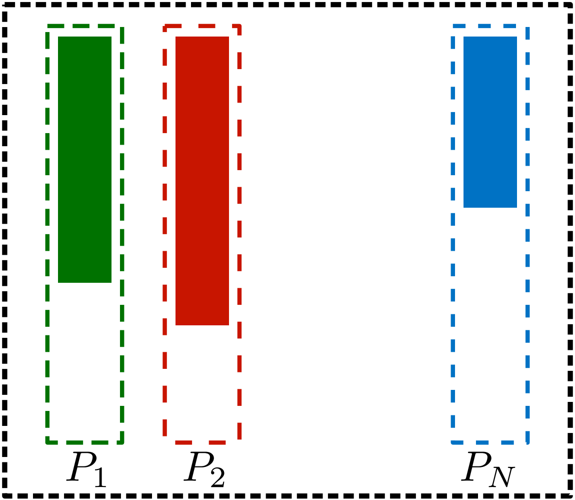
Parallel Implementation

- Communicate spawned array using MPIAlltoAllv.



Parallel Implementation

- Communicate spawned array using MPIAlltoAllv.



Hashing

Hashing

- At any point in time we need to know on which processor a given determinant should reside for annihilation to take place.

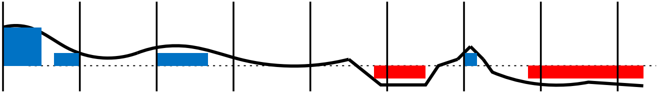
Hashing

- At any point in time we need to know on which processor a given determinant should reside for annihilation to take place.
- Storing a list is not practical.

Hashing

- At any point in time we need to know on which processor a given determinant should reside for annihilation to take place.
- Storing a list is not practical.
- Distributing according to integer label not likely to succeed.

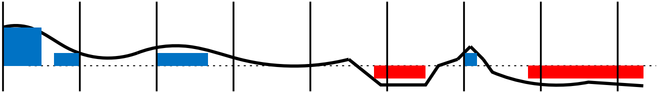
$$|\Psi\rangle =$$



Hashing

- At any point in time we need to know on which processor a given determinant should reside for annihilation to take place.
- Storing a list is not practical.
- Distributing according to integer label not likely to succeed.

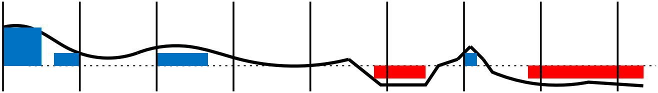
$$|\Psi\rangle =$$



- Use a hash function to randomise procedure somewhat.

Hashing

- At any point in time we need to know on which processor a given determinant should reside for annihilation to take place.
- Storing a list is not practical.
- Distributing according to integer label not likely to succeed.

$|\Psi\rangle =$ 

- Use a hash function to randomise procedure somewhat.
- Assign determinant to processor as $p = \text{hash}(|D_i\rangle)\text{mod}N_p$, $\text{hash}(x) = a, a \in [0, N_{\max})$.

Hashing (contd.)

- Hashing should result in an even distribution of walkers.

Hashing (contd.)

- Hashing should result in an even distribution of walkers.
- In practice as the number of processors increases load imbalances become more of an issue.

Hashing (contd.)

- Hashing should result in an even distribution of walkers.
- In practice as the number of processors increases load imbalances become more of an issue.
- Can isolate troublesome determinants (Booth, Smart, Alavi 2014).

Hashing (contd.)

- Hashing should result in an even distribution of walkers.
- In practice as the number of processors increases load imbalances become more of an issue.
- Can isolate troublesome determinants (Booth, Smart, Alavi 2014).
- Can we do better?

Load Balancing

- Difficult problem, especially when N_{dets} very large.

Load Balancing

- Difficult problem, especially when N_{dets} very large.
- Perfect load balancing would require storage of mapping on every processor.

Load Balancing

- Difficult problem, especially when N_{dets} very large.
- Perfect load balancing would require storage of mapping on every processor.
- Instead look for simple approach which should be good enough.

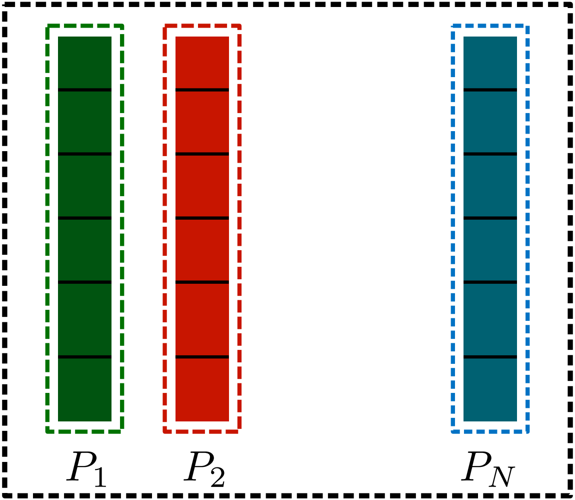
Load Balancing

- Difficult problem, especially when N_{dets} very large.
- Perfect load balancing would require storage of mapping on every processor.
- Instead look for simple approach which should be good enough.
- Idea: split hash range into M bins and redistribute these bins.

Load Balancing

- Difficult problem, especially when N_{dets} very large.
- Perfect load balancing would require storage of mapping on every processor.
- Instead look for simple approach which should be good enough.
- Idea: split hash range into M bins and redistribute these bins.

Picture



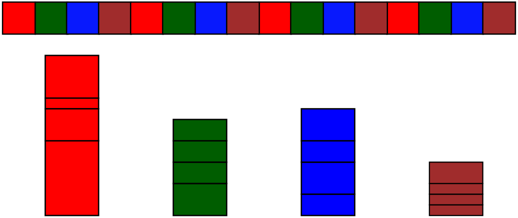
Load Balancing

Procedure:

1. Find processors with populations either above (P_i^d) or below (P_i^r) the ideal average walker population ($N_{\rm av} \pm \delta$).
2. Sort list of donor bins in increasing order of bin size.
3. Redistribute donor bins to receiver processors while $N_w(P_i^{\rm d}) \geq N_{\rm av} - \delta$ and $N_w(P_i^r) \leq N_{\rm av} + \delta$.

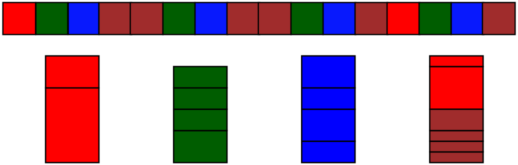
Picture

- Define array $p_{\text{map}}[i] = (0, 1, \dots, N_p, 0, \dots, N_p, \dots)$. So, $P(|D_i\rangle) = p_{\text{map}}[\text{hash}(|D_i\rangle)\text{mod}(N_p \times M)]$



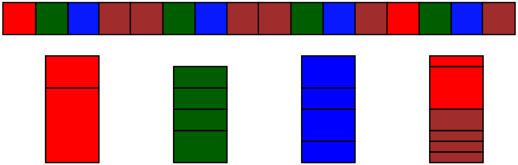
Picture

- Just modify entries in p_{map} so processors are mapped to new processor in the future.



Picture

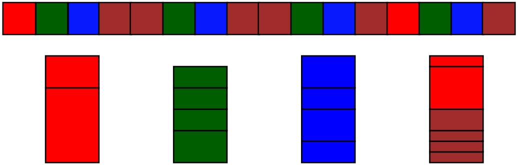
- Just modify entries in p_{map} so processors are mapped to new processor in the future.



- For $M = 1$ get usual procedure, for $M \to N_{\text{Dets}}$ get 'perfect' load balancing.

Picture

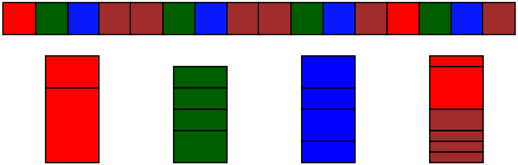
- Just modify entries in p_{map} so processors are mapped to new processor in the future.



- For $M = 1$ get usual procedure, for $M \to N_{\mathrm{Dets}}$ get 'perfect' load balancing.
- Trade off between overhead and improving load balancing. $M \sim 20 - 100$ is usually good enough.

Picture

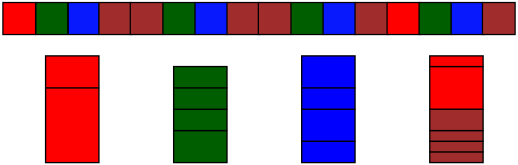
- Just modify entries in p_{map} so processors are mapped to new processor in the future.



- For $M = 1$ get usual procedure, for $M \to N_{\text{Dets}}$ get ‘perfect’ load balancing.
- Trade off between overhead and improving load balancing. $M \sim 20 - 100$ is usually good enough.
- Only need to distribute infrequently after equilibration.

Picture

- Just modify entries in p_{map} so processors are mapped to new processor in the future.



- For $M = 1$ get usual procedure, for $M \to N_{\text{Dets}}$ get 'perfect' load balancing.
- Trade off between overhead and improving load balancing. $M \sim 20 - 100$ is usually good enough.
- Only need to distribute infrequently after equilibration.

Non-Blocking Communications

- Another barrier to optimal scaling is the communication overhead which gets worse as the processor count increases.

Non-Blocking Communications

- Another barrier to optimal scaling is the communication overhead which gets worse as the processor count increases.
- Non-blocking communications can potentially mitigate these effects by overlapping computation with communication.

Non-Blocking Communications

- Another barrier to optimal scaling is the communication overhead which gets worse as the processor count increases.
- Non-blocking communications can potentially mitigate these effects by overlapping computation with communication.
- How can this be achieved in FCIQMC when psips need to annihilate?

Non-Blocking Communications

- Another barrier to optimal scaling is the communication overhead which gets worse as the processor count increases.
- Non-blocking communications can potentially mitigate these effects by overlapping computation with communication.
- How can this be achieved in FCIQMC when psips need to annihilate?
- Solution: They don't need to annihilate every step once they annihilate at the same point in time (continuous time extension: Spencer, Foulkes).

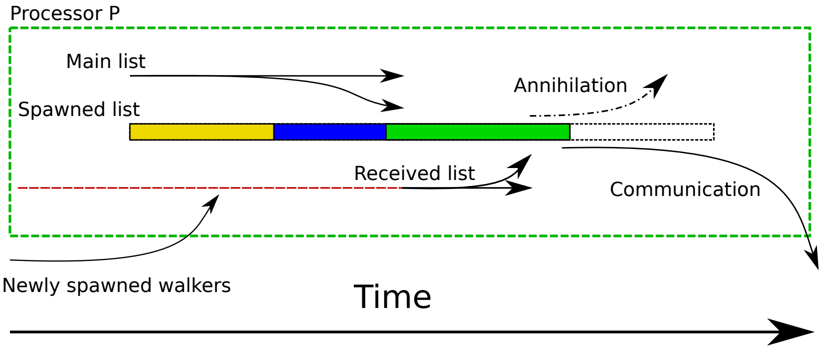
Non-Blocking Communications

- Another barrier to optimal scaling is the communication overhead which gets worse as the processor count increases.
- Non-blocking communications can potentially mitigate these effects by overlapping computation with communication.
- How can this be achieved in FCIQMC when psips need to annihilate?
- Solution: They don't need to annihilate every step once they annihilate at the same point in time (continuous time extension: Spencer, Foulkes).

Non-Blocking algorithm

- Evolve main list to $\tau + \Delta\tau$ (receiving spawned walkers in background)
- Complete receive
- Evolve walkers spawned onto current processor to $\tau + \Delta\tau$
- Non-blocking send of walkers to their new processors.
- Annihilate walkers on current processor

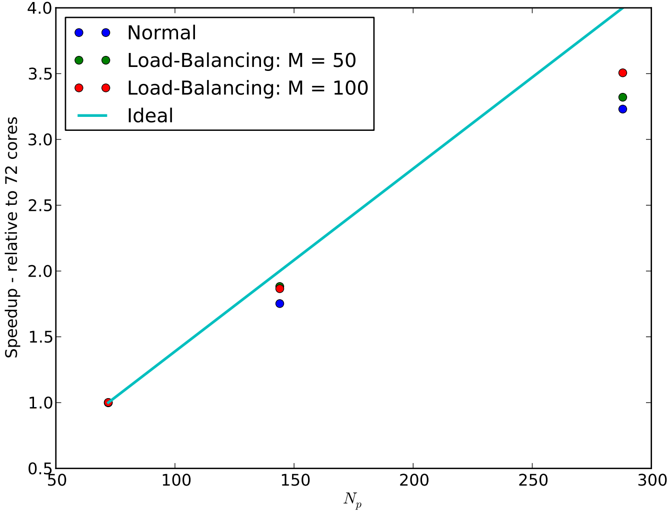
Picture


Processor P
Main list
Annihilation
Spawned list
Received list
Communication
Newly spawned walkers
Time

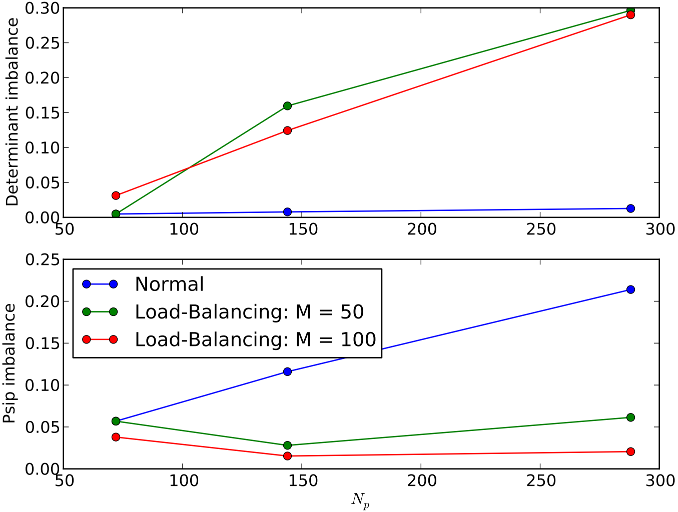
Test Case

- 18-site Hubbard model in momentum space basis.
- 86 million psips occupying 61.2 million determinants
- Run for 20000 iterations

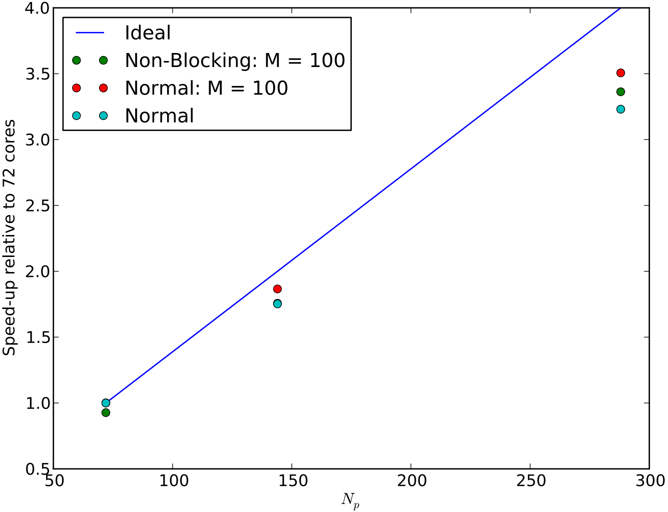
Preliminary results: Load Balancing



Preliminary results: Load Balancing



Preliminary results: Load Balancing + Non-Blocking



Conclusions

Conclusions

- Simple load balancing can improve the parallel performance of FCIQMC-like codes.

Conclusions

- Simple load balancing can improve the parallel performance of FCIQMC-like codes.
- Non-blocking communications should improve scaling when running on more processors.

Conclusions

- Simple load balancing can improve the parallel performance of FCIQMC-like codes.
- Non-blocking communications should improve scaling when running on more processors.
- Next step run on larger computers - 1000s of cores.

Acknowledgemets

James Spencer & HANDE development team
Mike Towler
Supervisors: Matthew Foulkes & Derek Lee
Computer Time: Imperial College HPC service
Funding: Imperial College PhD scholarship