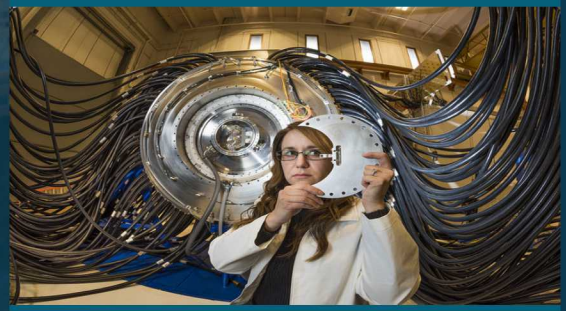
SAND2019-7114C

Sandia National Laboratories

Enabling new capabilities for QMC using HPC without sacrificing developer productivity



PRESENTED BY

Luke Shulenburger

U.S. DEPARTMENT OF ENERGY | NNSA

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Acknowledgements

- Exascale Computing Project
- QMCPACK contributors
- Jeongnim Kim
- Paul Kent
- Ye Luo
- Raymond Clay
- Attila Cangi
- Christian Trott

What are the current limitations of diffusion Monte Carlo in practice?

Nodal surface of unknown quality
- Hard problem, subject of other talks

Lack of routinely calculated observables
- Needs significant focus, development of many different capabilities

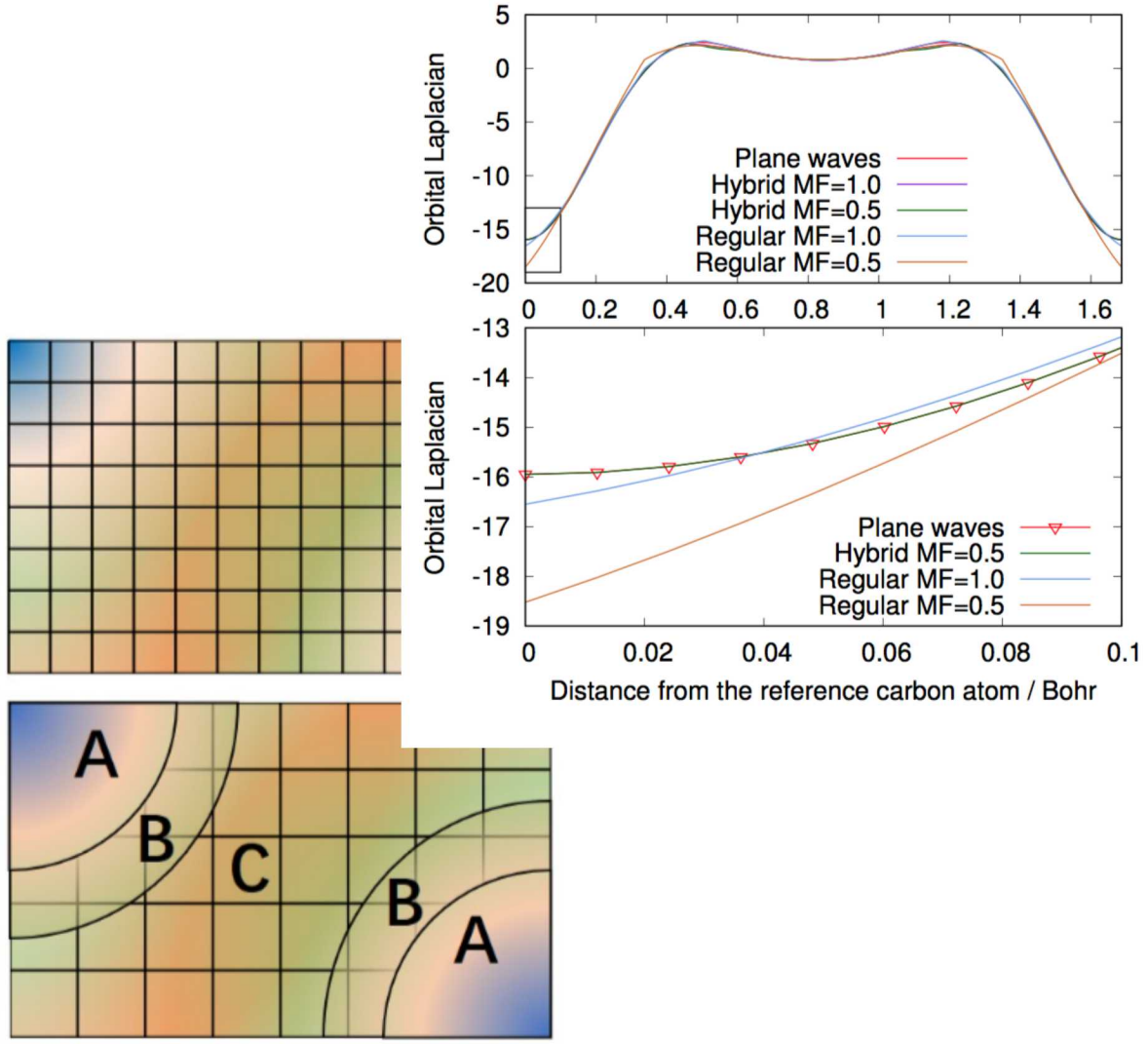
Productivity difficulties for researchers
- Complicated workflow
- Limited statistical accuracy / computer time

Finite system size
- Memory and $O(N^3)$ complexity limit calculations to ~1000 electrons
- For condensed matter, high symmetry!
- Difficult to treat complicated defect structures or low symmetry phases

To reach large systems, memory demands must be reduced while maintaining high efficiency

Have explored two options for the static wavefunction

- Periodic localized orbitals (gaussians, atom centered splines)
  - Drastic memory reduction but slows code (lost verctorization for example)
- Also a combined hybrid B-spline and atom centered wavefunction
  - Use a coarse spline table everywhere and augment with 1D splines times Ylm near core
  - Saves about a factor of 5-10 in memory while retaining speed
  - Most effective for cases where deeper core states are represented

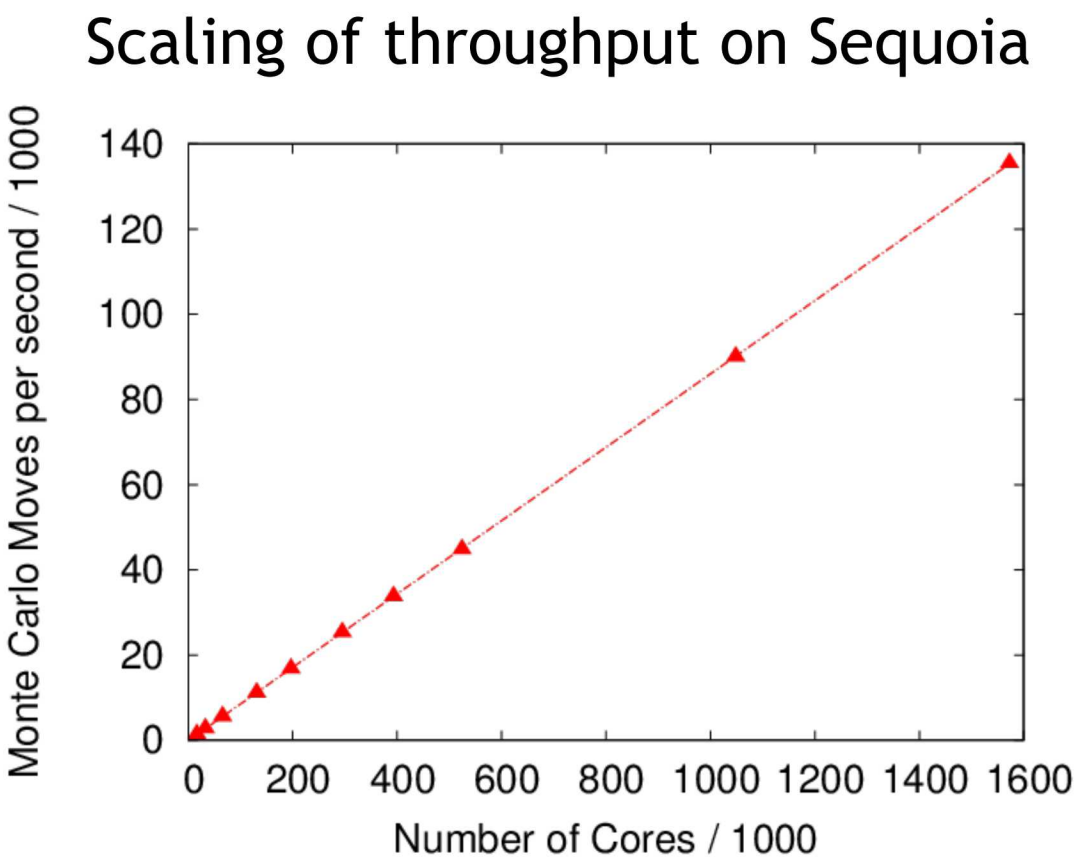


How do we currently address scaling?

Current parallelization strategy relies on distributing walkers among processing elements

Highly scalable algorithm, as demonstrated on Sequoia at LLNL

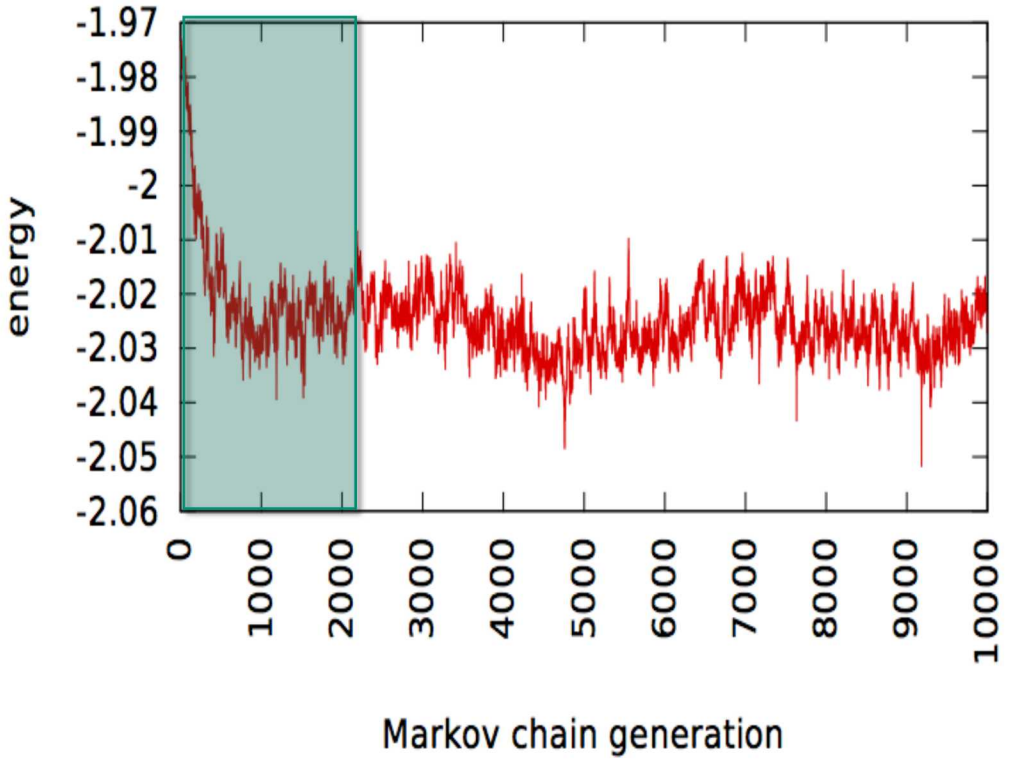
- Nearly perfect parallel efficiency in moving walkers to over 1.5M cores

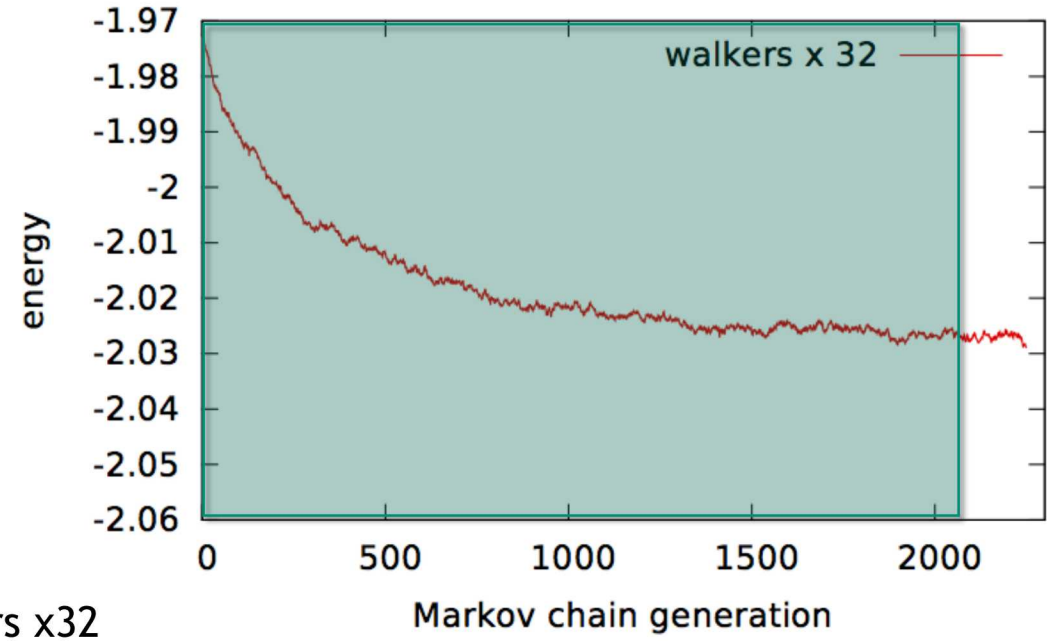


As calculations get larger, use more cores and more walkers

Increasing the number of cores in the case of large core counts

What actually happens as the calculation is scaled?





- Number of walkers x32
- Processors x 32
- Throughput x 32
- Wall time / 5
- Parallel Efficiency ~ 15%

Equilibration dominates

Best strategy is to parallelize walkers over more than a single processing element

In QMCPACK, the code is parallelized so that the walker is the minimum unit of parallel work

For large calculations, target is currently one walker per thread

In order to reduce population by a factor of 64, would have to be able to parallelize the walker over 64 processing elements

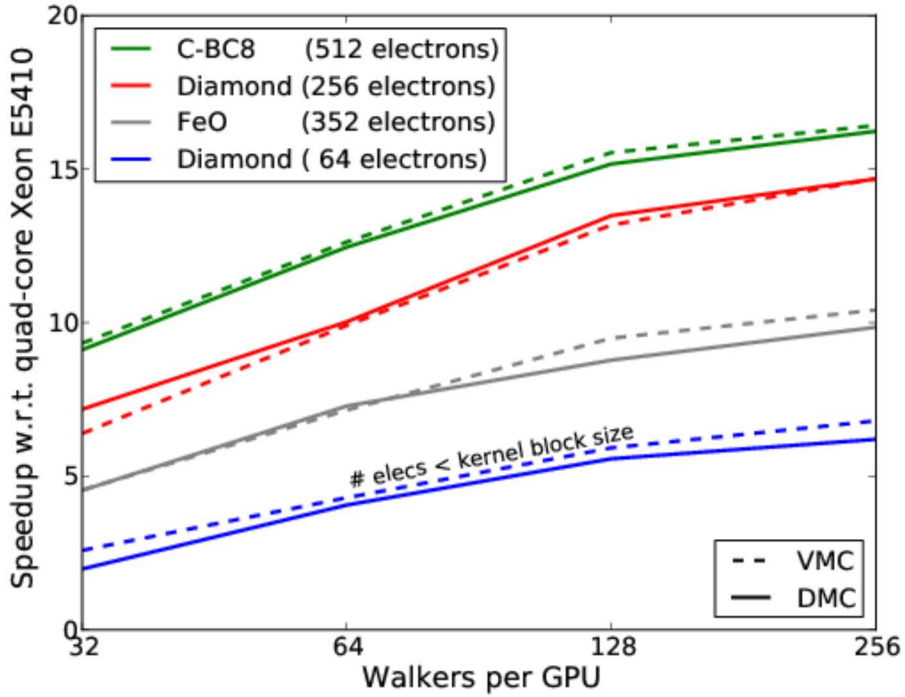
- Can’t currently do this, but it is plausible for large enough calculations

While working on this, consider changing hardware landscape

- Rapidly increasing number of computing elements
- Stagnating or decreasing memory size per processing e
  - Increasingly complex and varied memory hierarchy
- Changing programming models and diverse capabilities

Previous experience with non CPU architectures

- Ken Esler led an effort to parallelize QMCPACK for GPUs
- Limited memory bandwidth between host and accelerator meant rewriting large sections of the code for the GPU
- Small kernels were problematic, so exposed additional parallelism by reordering the calculation
  - Move all walkers at once
- Significant speedup at the time, nightmare to maintain!



Esler, Kim, Ceperely and Shulenburger, CiSE 14, 40, 2012

Requirements: Performance portability and ease of development

Seek a programming model where minimal code needs to be re-written when moving to new architectures

- Don’t want cuda, hip, rocm, etc ports that have to be maintained

Simultaneously work to parallelize walkers over as many processing elements as possible

Two strong candidates:

- OpenMP 4.0 (offload)
- Kokkos

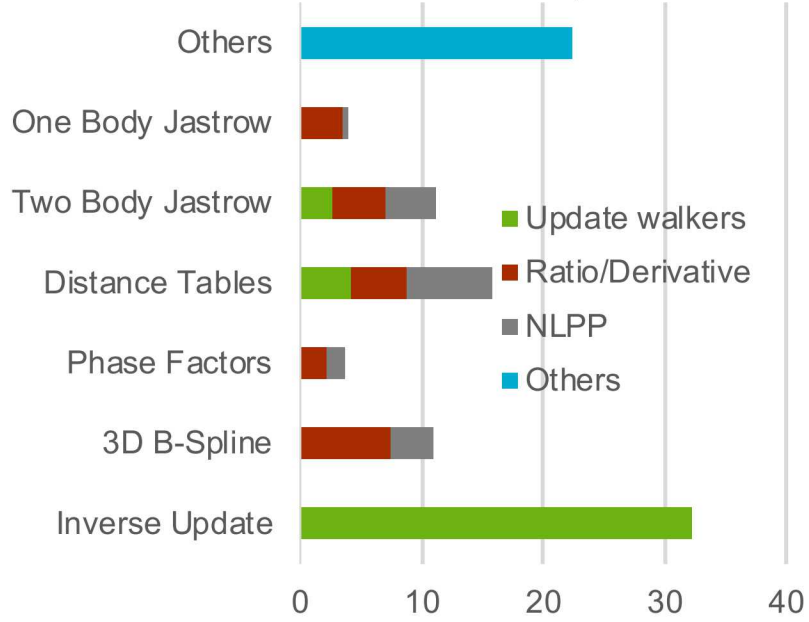
Must be able to do strong scaling for large problems and simultaneously maintain high throughput for smaller calculations

Also cut down the application: qmcpack miniapp

Determined computationally important kernels and separated into a miniapp

Important to keep the same code structure to stress performance portability models

Result is at github.com/QMCPACK/miniqmc



percentage	KNL	BGQ
Dist. Tables	26.9	15.7
3D B-Spline	12.4	10.9
Inv. Update	22.0	32.3

1: **for** MC generation $= 1 \cdots M$ **do**
2: **for** walker $= 1 \cdots N_w$ **do**
3: let $\mathbf{R} = \{\mathbf{r}_1 \ldots \mathbf{r}_N\}$
4: **for** particle $i = 1 \cdots N$ **do**
5: set $\mathbf{r}_i^{'} = \mathbf{r}_i + \delta$
6: let $\mathbf{R}^{'} = \{\mathbf{r}_1 \ldots \mathbf{r}_i^{'} \ldots \mathbf{r}_N\}$
7: **ratio** $\rho = \Psi_T(\mathbf{R}^{'})/\Psi_T(\mathbf{R})$
(One Body, Two Body, 3D B-Spline)
8: **derivatives** $\nabla_i \Psi_T, \nabla_i^2 \Psi_T$
(One Body, Two Body, 3D B-Spline)
9: **if** $\mathbf{r} \rightarrow \mathbf{r}'$ is accepted **then**
10: update state of a walker
(Inverse Update)
11: **end if**
12: **end for**{particle}
13: **local energy** $E_L =$ $\hat{H}\Psi_T(\mathbf{R})/\Psi_T(\mathbf{R})$
(One Body, Two Body, 3D B-Spline)
14: reweight and branch walkers
15: **end for**{walker}
16: update E_T and load balance
17: **end for**{MC generation}

Current architectures favor different parallelization schemes

GPUs require extreme levels of parallelization

- Where possible, parallelize over electrons and walkers simultaneously
- Memory is very limited, also transfers are slow

CPUs require care with vectorization

- Often factor of 2-4 when vectorizable
- Memory layout is crucial

Kokkos allows all of these to be handled in the same framework

Flexible data container

- Multidimensional data container: view view<datatype***, layout, memorySpace, ...>
- Encapsulates where the data lives and its layout
- Avoid expensive implicit copy operations, only pointers are copied when view is assigned
- Makes it easy to move things around
- Can automatically change data layout depending on type of target device

Abstractions for parallel operations

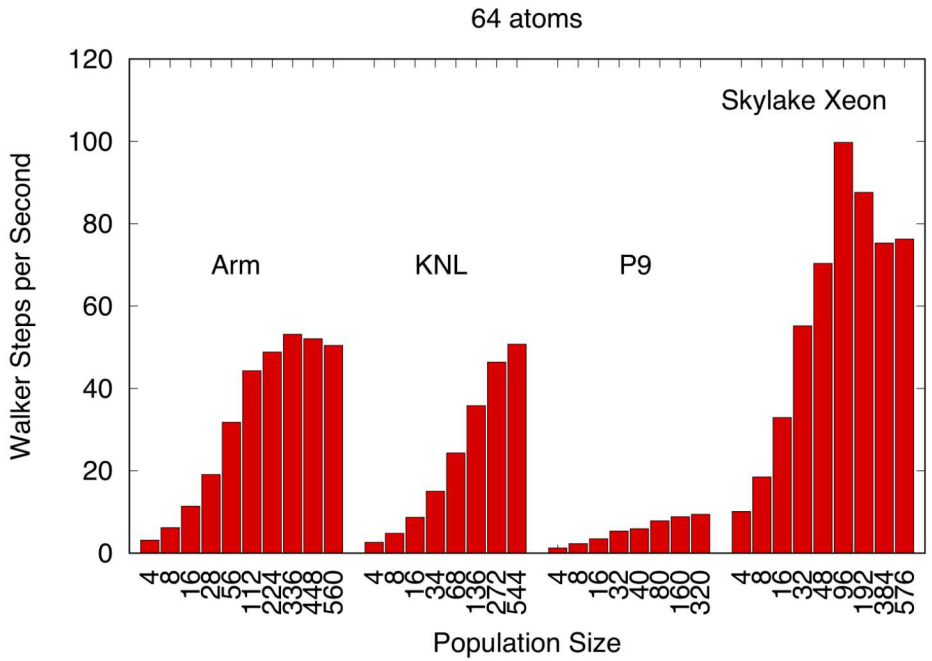
- When operating in parallel, take advantage of generic constructs: parallel_for, parallel_reduce, parallel_scan
- Easy to specify hierarchical parallelism through execution policies (who is doing the parallel_for)
- Works with the data abstraction to make memory accesses favorable for the architecture

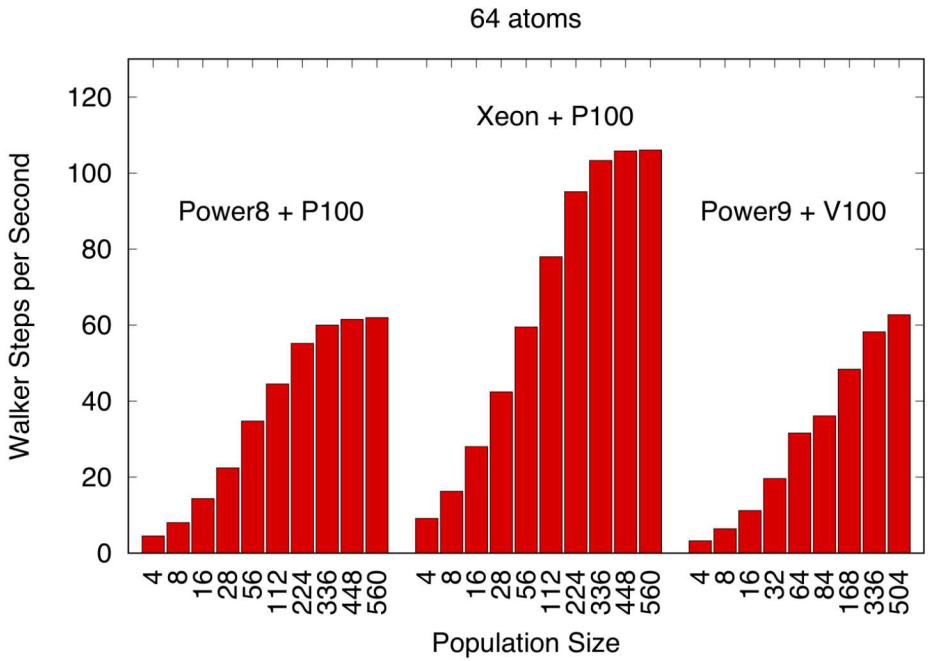
Process

1. Change data structures to Views
2. Rewrite OpenMP parallelism in terms of parallel_for and parallel_reduce
3. Express all remaining algorithms in parallel_for regions to avoid data transfer
4. For CPUs, most algorithms use a hierarchical scheme: Policy<nwalkers, 1, vector_size>
5. For GPUs, most algorithms use a flattened scheme: Policy<nwalkers*nelectrons>
6. For GPUs, needed to free up as much memory as possible
   - Was storing distance table between all electrons and ions, now computing on fly

Results for small test problem

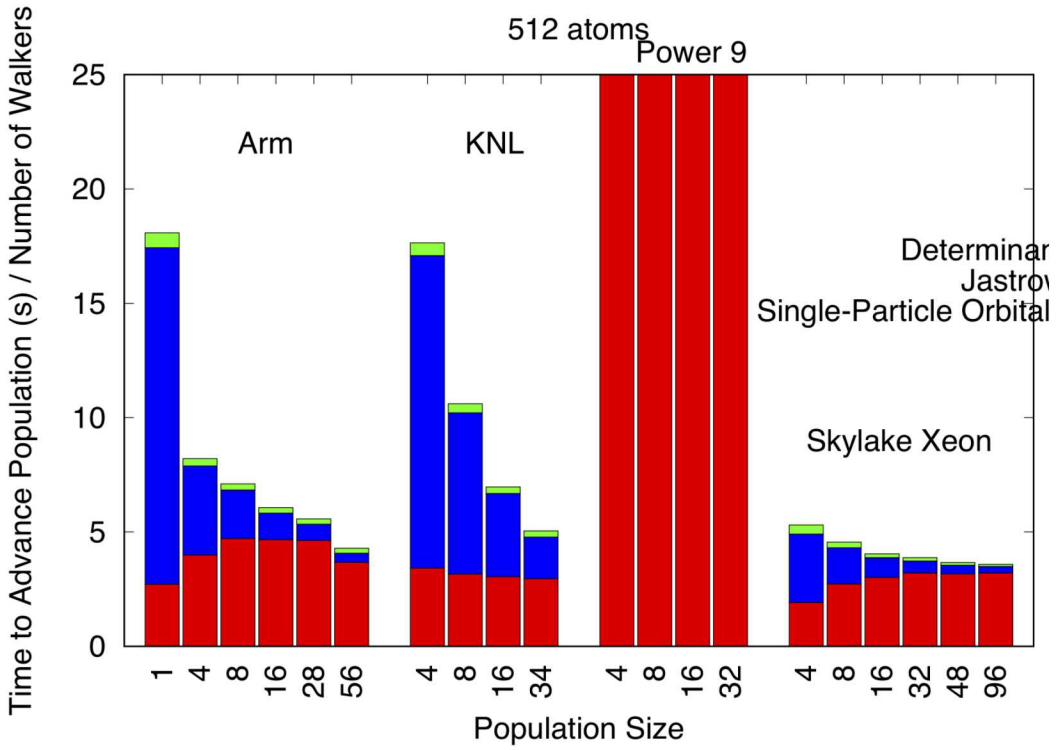
768 electrons

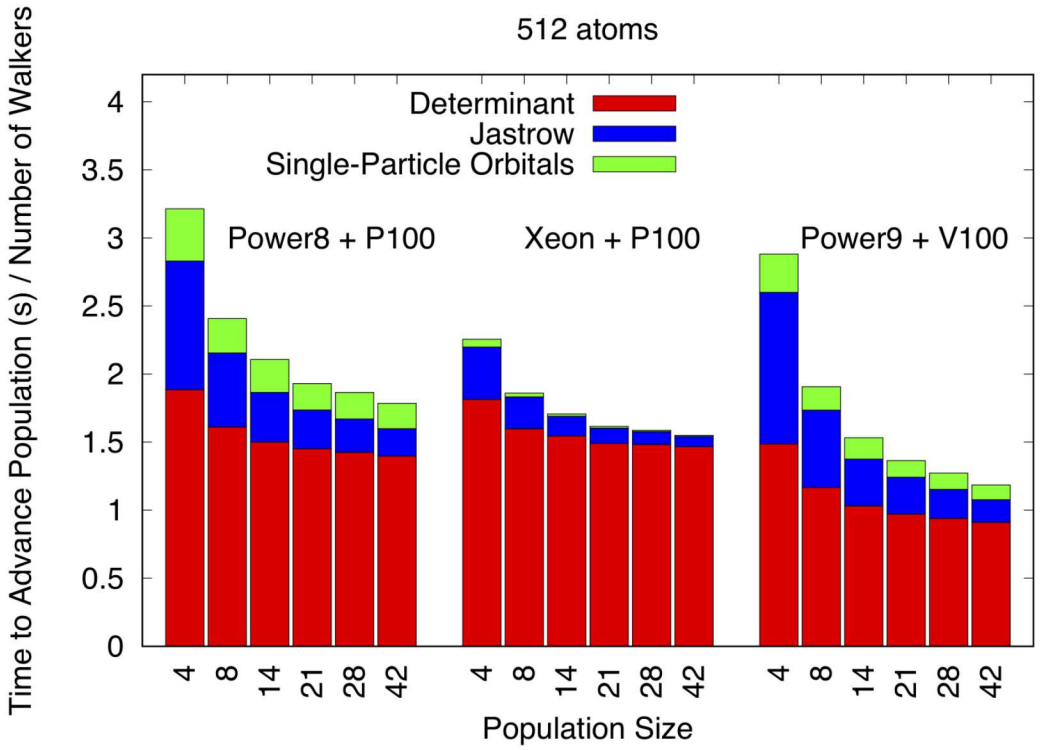




Results for large problems

6144 electrons





It appears possible to achieve performance portability within a single code base

Can use the same framework to achieve parallel algorithms that are suitable for both GPUs and CPUs

Work of this nature can allow codes to take advantage of new classes of hardware without requiring developers to learn new frameworks each time