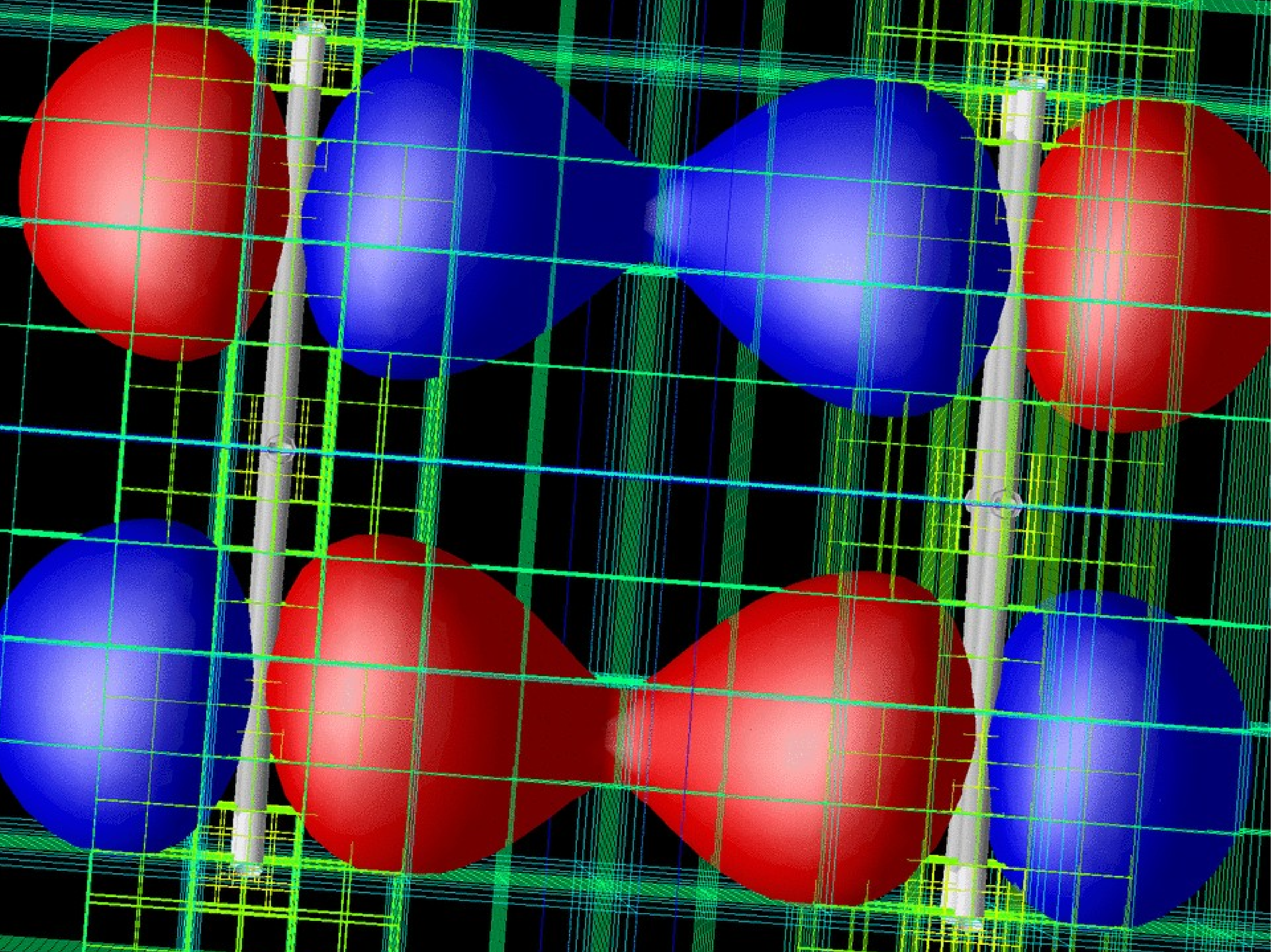
Task-oriented computing within MADNESS

Scott Thornton

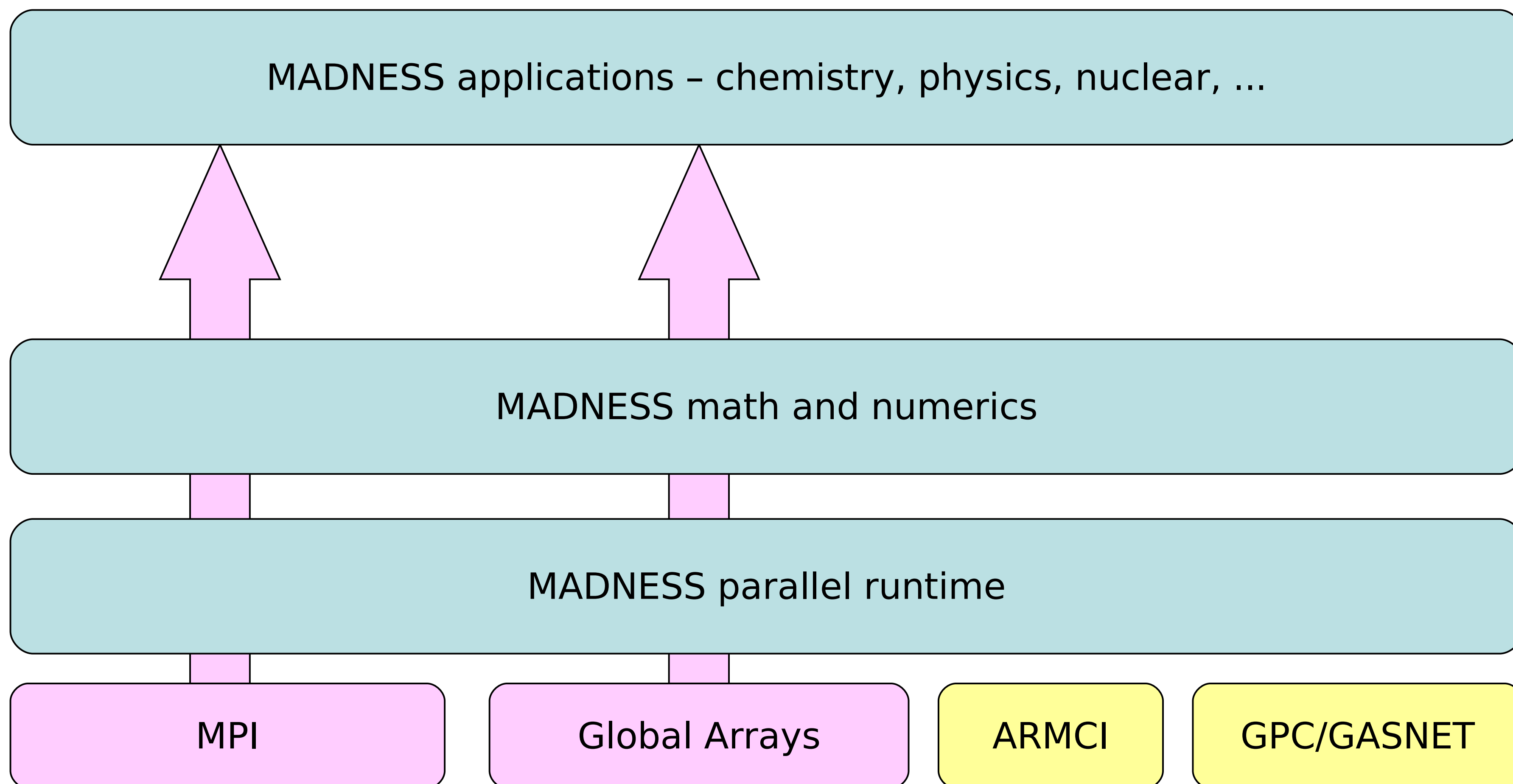
University of Tennessee

MADNESS

Multiresolution
Adaptive
Numerical
Scientific
Simulation



MADNESS architecture



Intel Thread Building Blocks being considered for multicore

Ways to expand a function

There are two different ways to expand a function
Within the MADNESS framework:

1. Scaling function basis

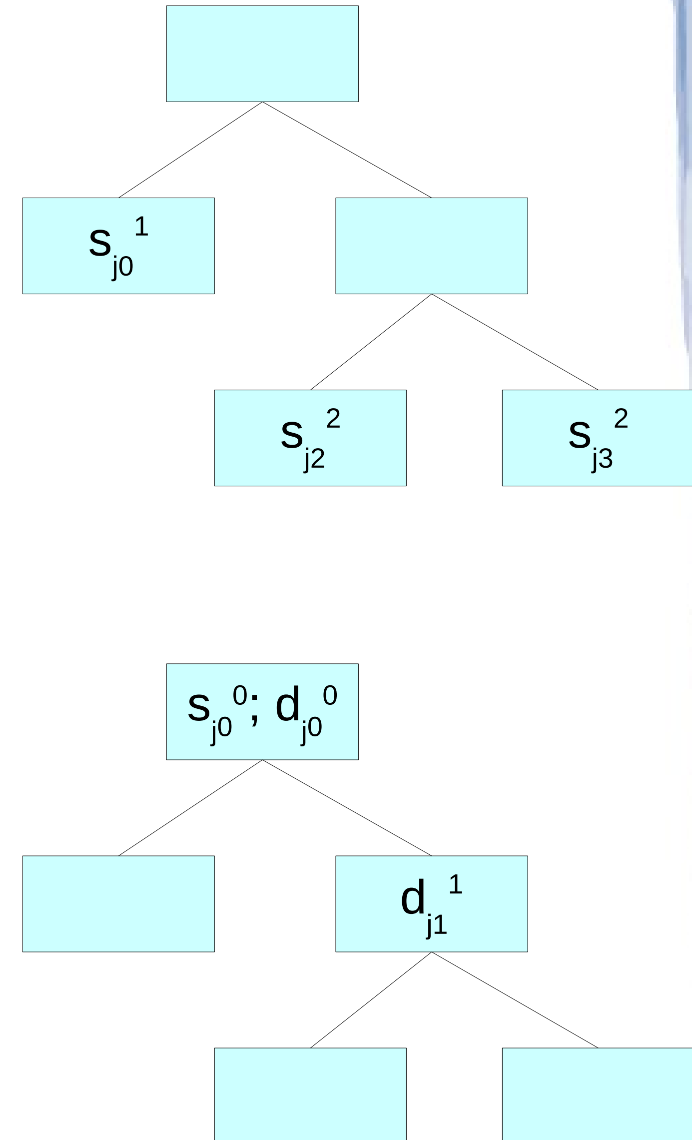
2. Wavelet basis

Scaling function basis:

$$f(x) = \sum_{l=0}^{2^n-1} \sum_{j=0}^{k-1} s_{jl}^n \phi_{jl}^n(x)$$

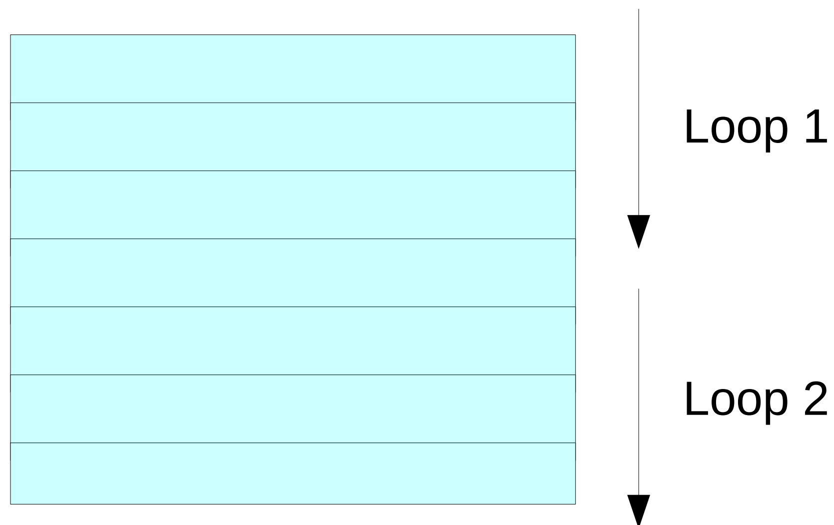
Wavelet basis:

$$f(x) = \sum_{j=0} s_{j0}^0 \phi_{j0}^0(x) + \sum_{n'=0}^{n-1} \sum_{l=0}^{2^n-1} \sum_{j=0}^{k-1} d_{jl}^{n}{}' \psi_{jl}^{n}{}'(x)$$



Data structures

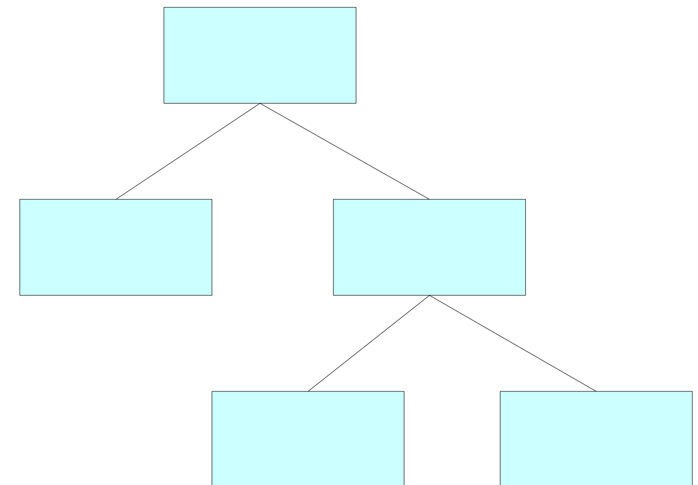
Regular



Parallelism can be implemented over regular data structures by having parallel loops over the structure.

Straight-forward using MPI or OpenMP

Irregular



However, with irregular data Structures such as a binary tree, deciding how to parallelize is not as straight-forward?

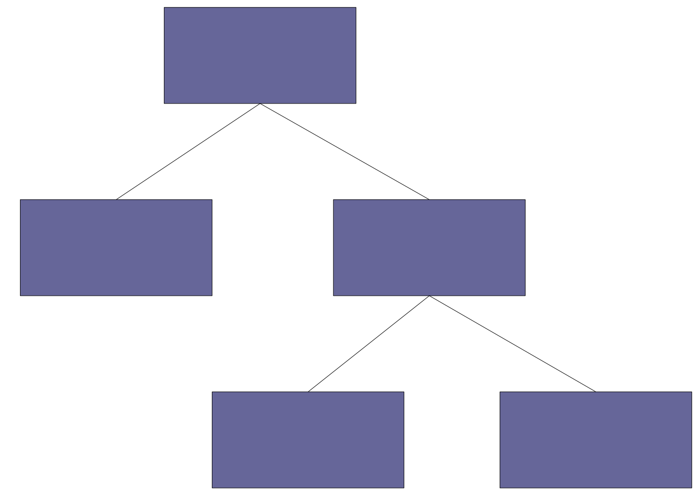
Data structures

Functions are “adaptively refined” meaning that the data structure are constantly changing

Irregular computing has a high degree of complexity in the management and flow of data in a parallel environment

Needs the ability to be able to “walk” up and down the tree in a scalable fashion

MADNESS function objects have 1000's of nodes



Don't want to burden the application developer or user with this complexity.

MADNESS design

Each node of a MADNESS function object
Lives on a different SMP node

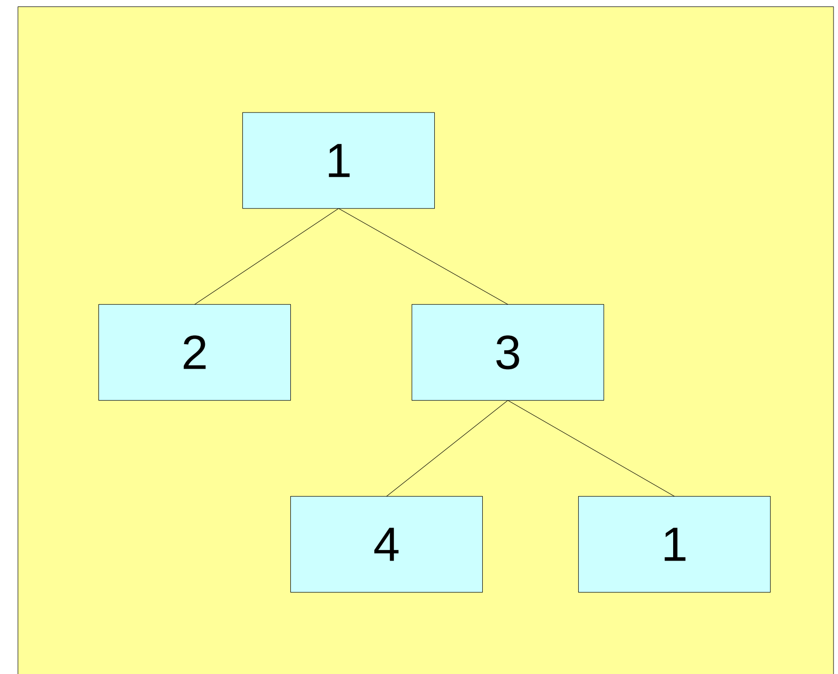
This decomposition allows for a mechanism of load balancing through the reshuffling of the processor map.

Each SMP node has a task queue

On each SMP, there is pool of worker threads
That execute the tasks in the task queue

One worker thread per core

Processor map



Tasks can be executed both locally and remotely (load balancing, work stealing, etc.)

One main thread on each SMP node that drives the whole process (executes main())

One thread on each SMP node that handles all communication between nodes

Dependencies

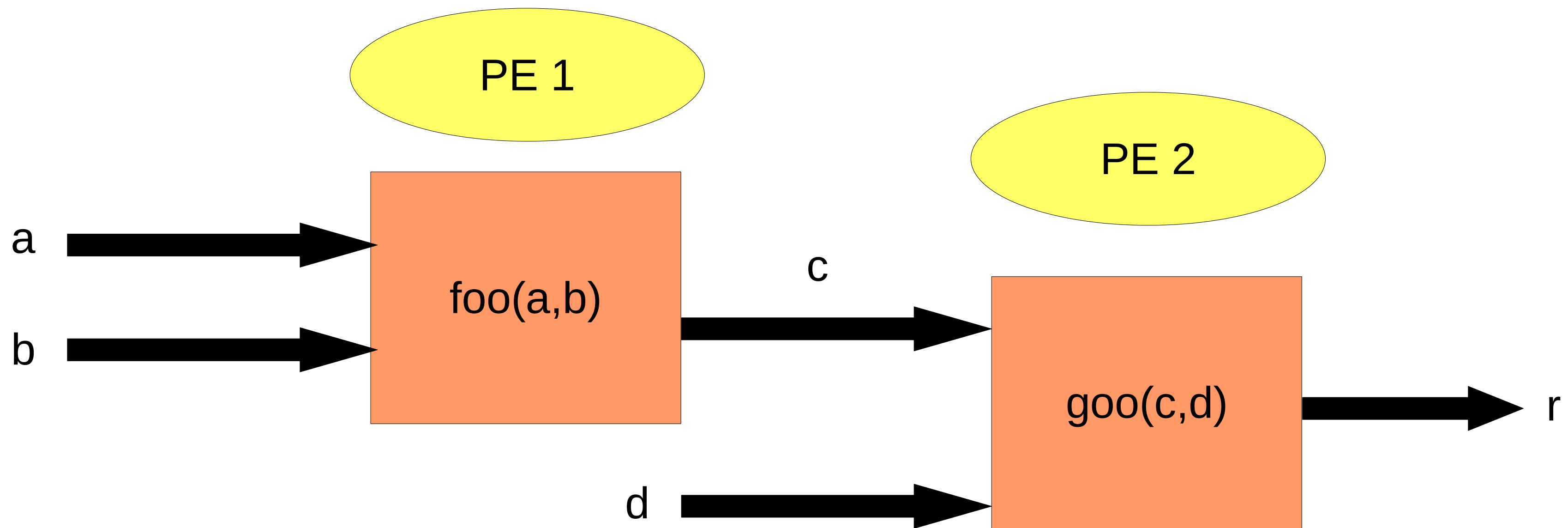
Due to the irregular nature of the MADNESS execution flow, dependencies between tasks need to be expressed.

Dependencies are pieces of information that are needed before a given task is executed.

Task = asynchronous function calls

Dependencies provide a way to manage latencies:

- Algorithmic latency
- Communication latency



The bottom line is: “foo” MUST execute before “goo”

Dependencies

Now, suppose we put our functions in a queue to be executed.

How can we ensure that “foo” will not execute until inputs, “a” and “b” are ready?

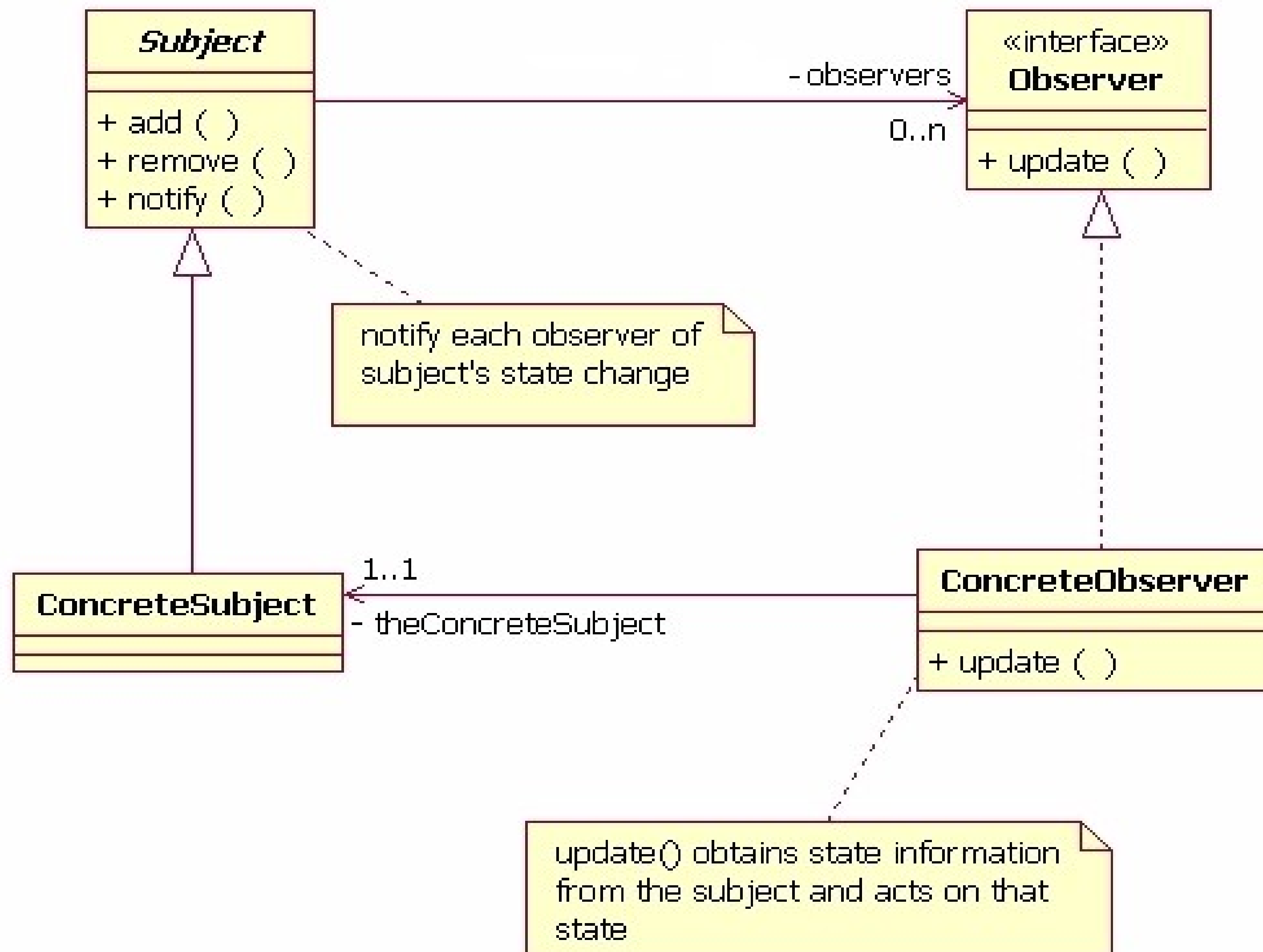
What about “goo”?

What we would like is an object that can “wrap” a piece of data and manage the dependencies.

task queue
foo(a, b) goo(c, d)

Observer Pattern OOD

Observer pattern



Futures

- Result of an asynchronous computation
  - Cilk, Java, HPCLs
- Hide latency due to communication or computation
- Management of dependencies
  - Via callbacks

```
int f(int arg);
ProcessId me, p;

Future<int> r0=task(p, f, 0);
Future<int> r1=task(me, f, r0);

// Work until need result

cout << r0 << r1 << endl;
```

Process “me” spawns a new task in process “p” to execute `f(0)` with the result eventually returned as the value of future `r0`. This is used as the argument of a second task whose execution is deferred until its argument is assigned. Tasks and futures can register multiple local or remote callbacks to express complex and dynamic dependencies.

Task-oriented programming

Main thread converts the algorithm / calls to the API into a series of tasks that get scheduled into a node's task queue.

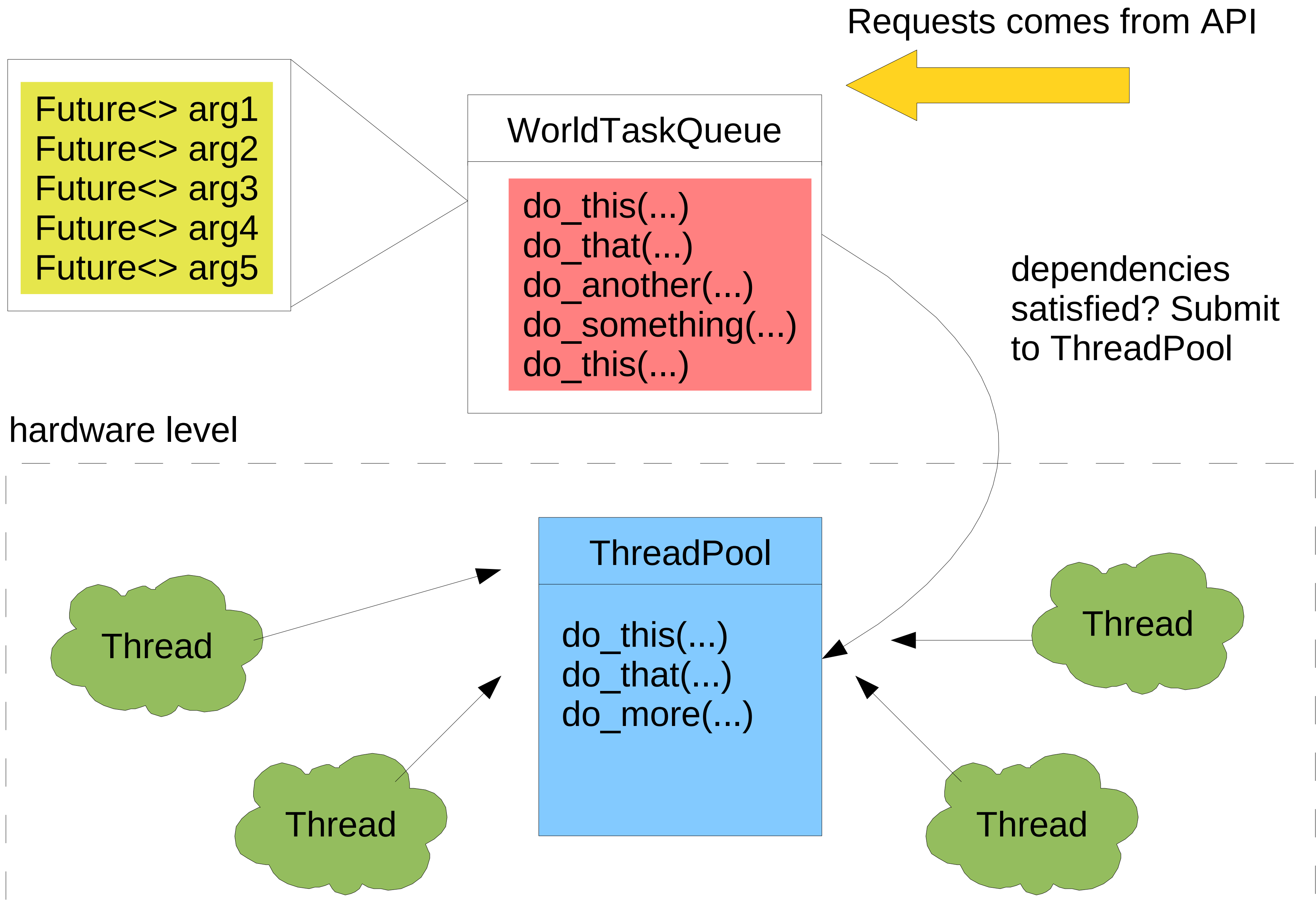
```
for (things to be done):
        world.task(where, &::do_this, arg1, arg2, arg3);
        world.task(here, &::do_that, arg1, arg2);
        .
        .
end for
```

Tasks can run anywhere on the machine.

Many algorithms used in MADNESS require iterating through the tree structure. The task-oriented nature of the runtime is a natural fit for these types of algorithms.

```
for (child in (my children)):
        tree(child, &::go_forth, arg);
end for
```

parallel runtime overview



Periodic DFT solver

Kohn-Sham equation:

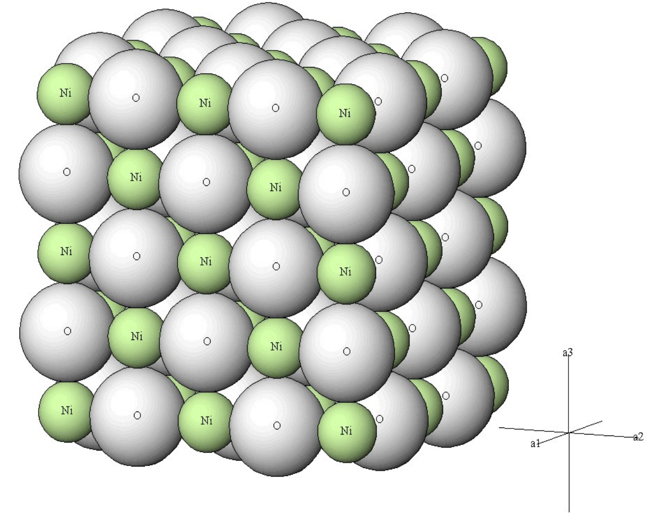
$$\left[-\frac{1}{2}\nabla^2+v_{eff}[n(r)](r)\right]\phi_j(r)=\epsilon_j\phi_j(r)$$

electronic density:

$$n(r)=\sum_j|\phi_j(r)|^2$$

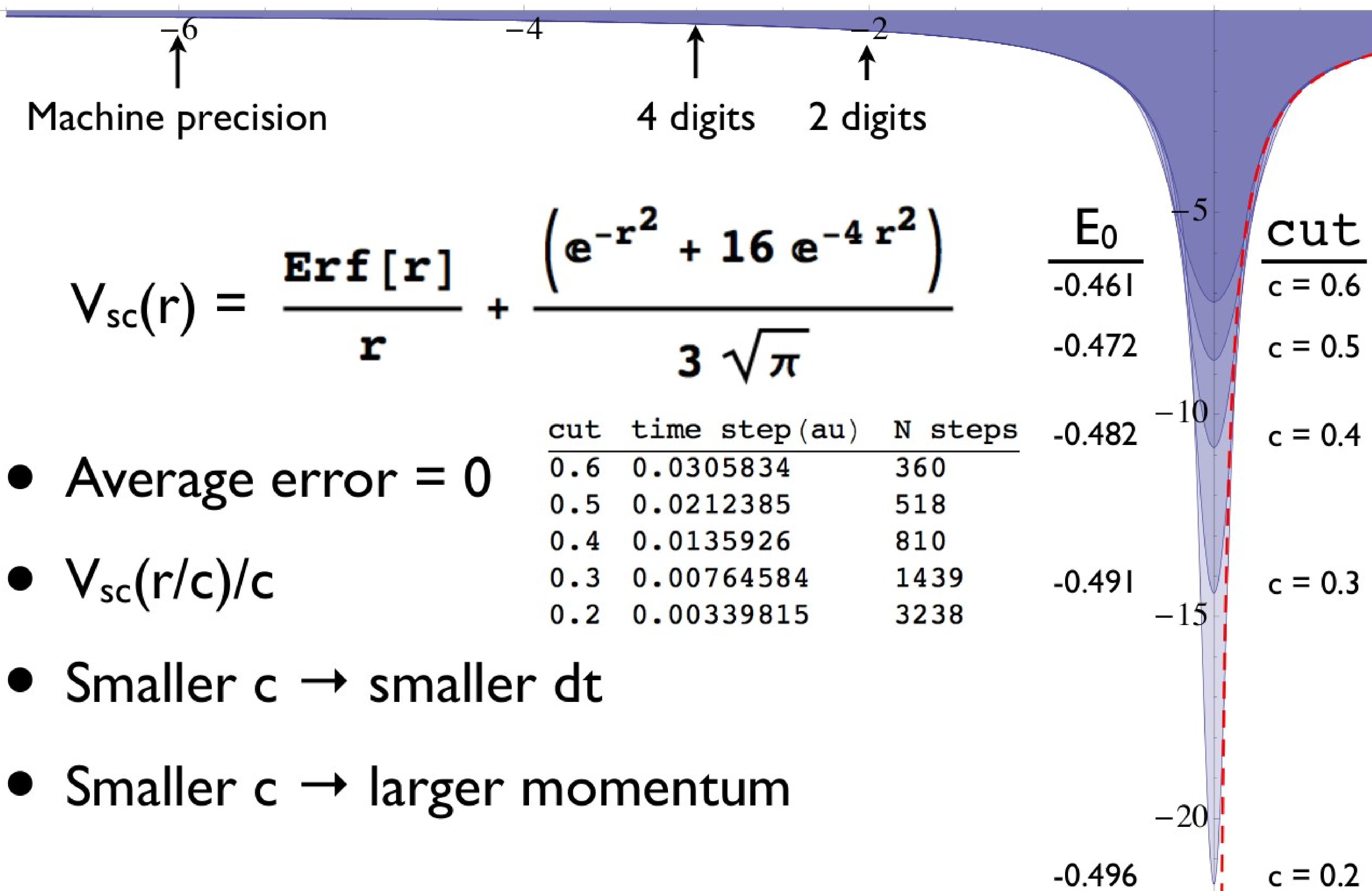
effective potential:

$$v_{eff}(r)=v_{ext}(r)+\int\frac{n(r')}{|r-r'|}d^3r'+v_{xc}(r)$$

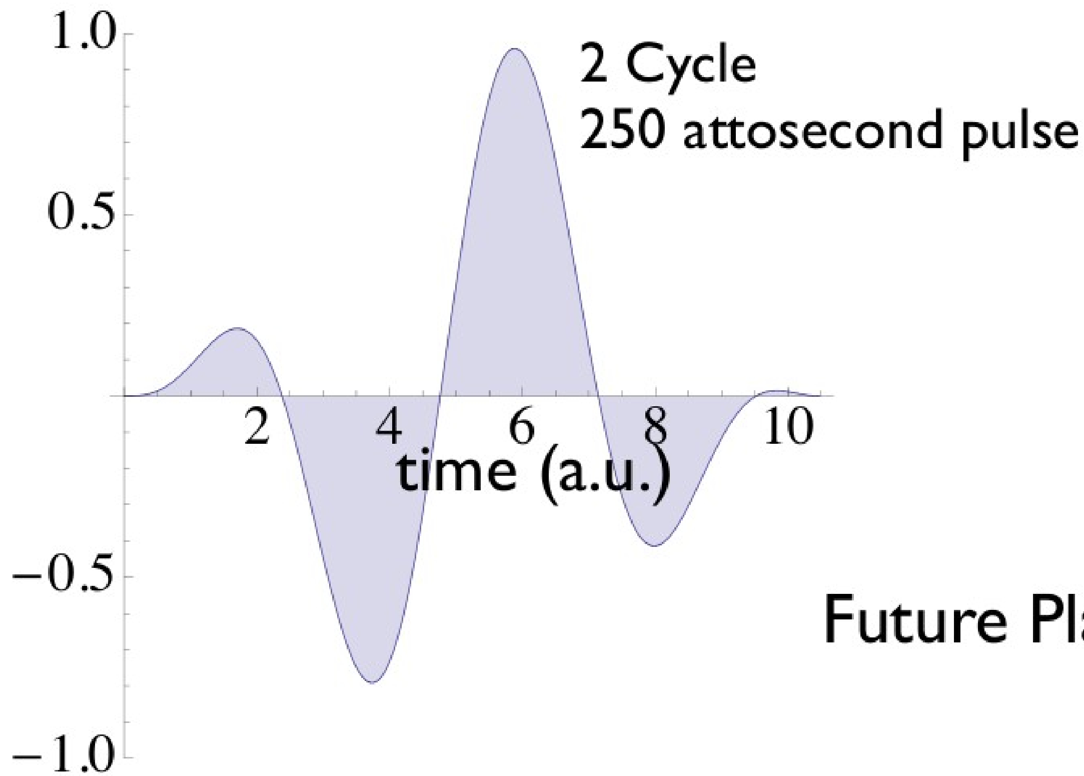


The Kohn-Sham equation is nonlinear and therefore needs to be solved in a self-consistent manner.

The Hydrogen Model Potential



Laser Pulse



$$\mathbf{F}(t) = \hat{\mathbf{z}}\, f(t)\, \sin(\omega t)$$

$$f(t) = F_0 \sin^2(\omega t/n)$$

$$n = 2.2$$

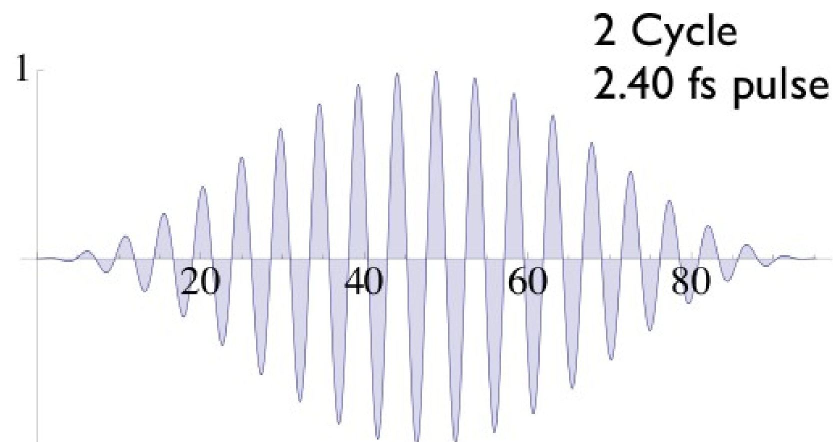
$$\omega = 1.32,\ \lambda = 35.1\text{nm}$$

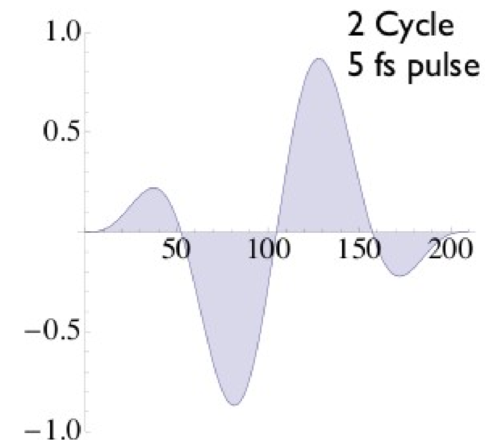
$$F_0 = 0.172$$

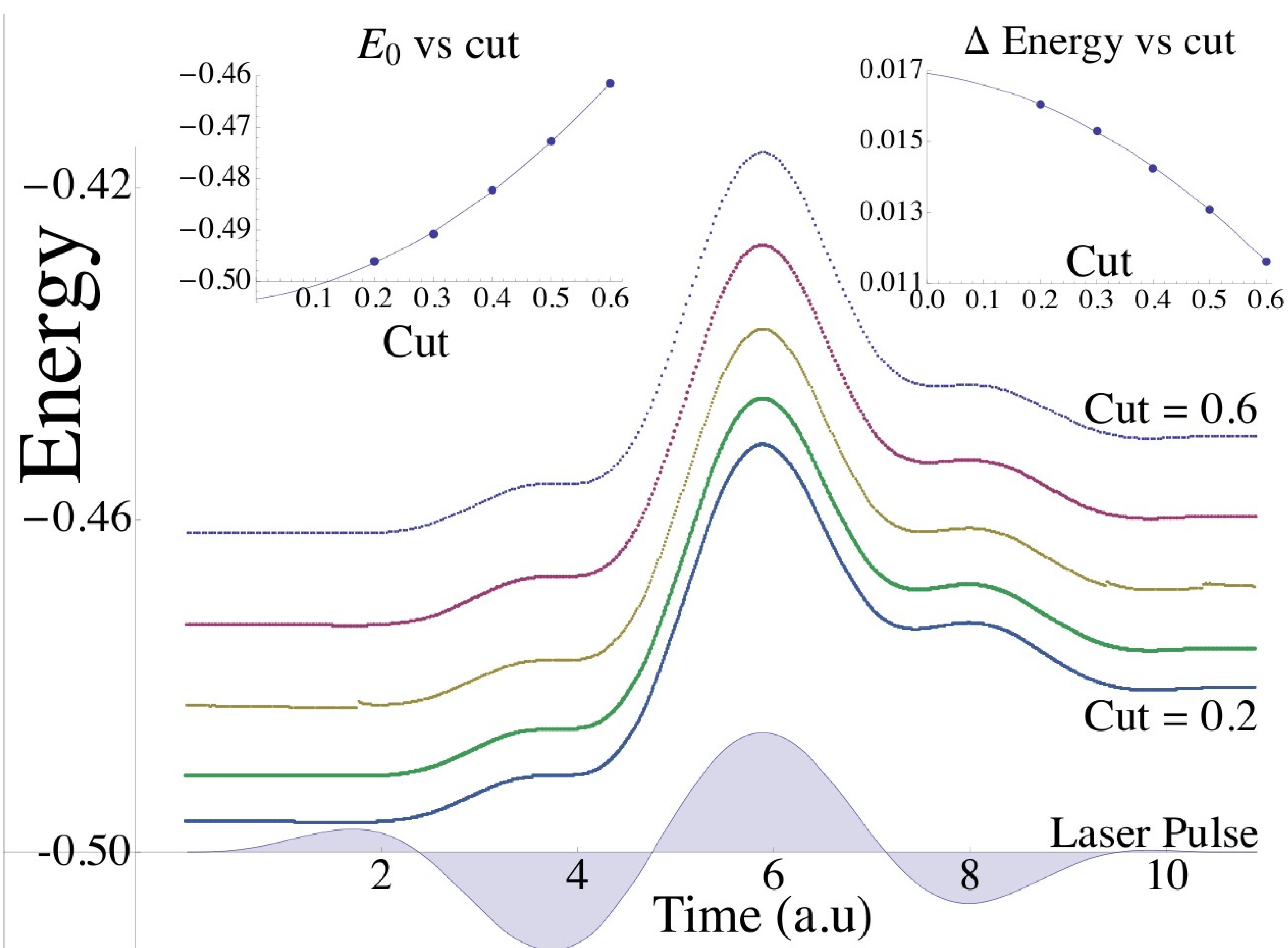
Future Plans:

Titanium Sapphire

$\omega = 0.06$ au, $\lambda = 760$ nm

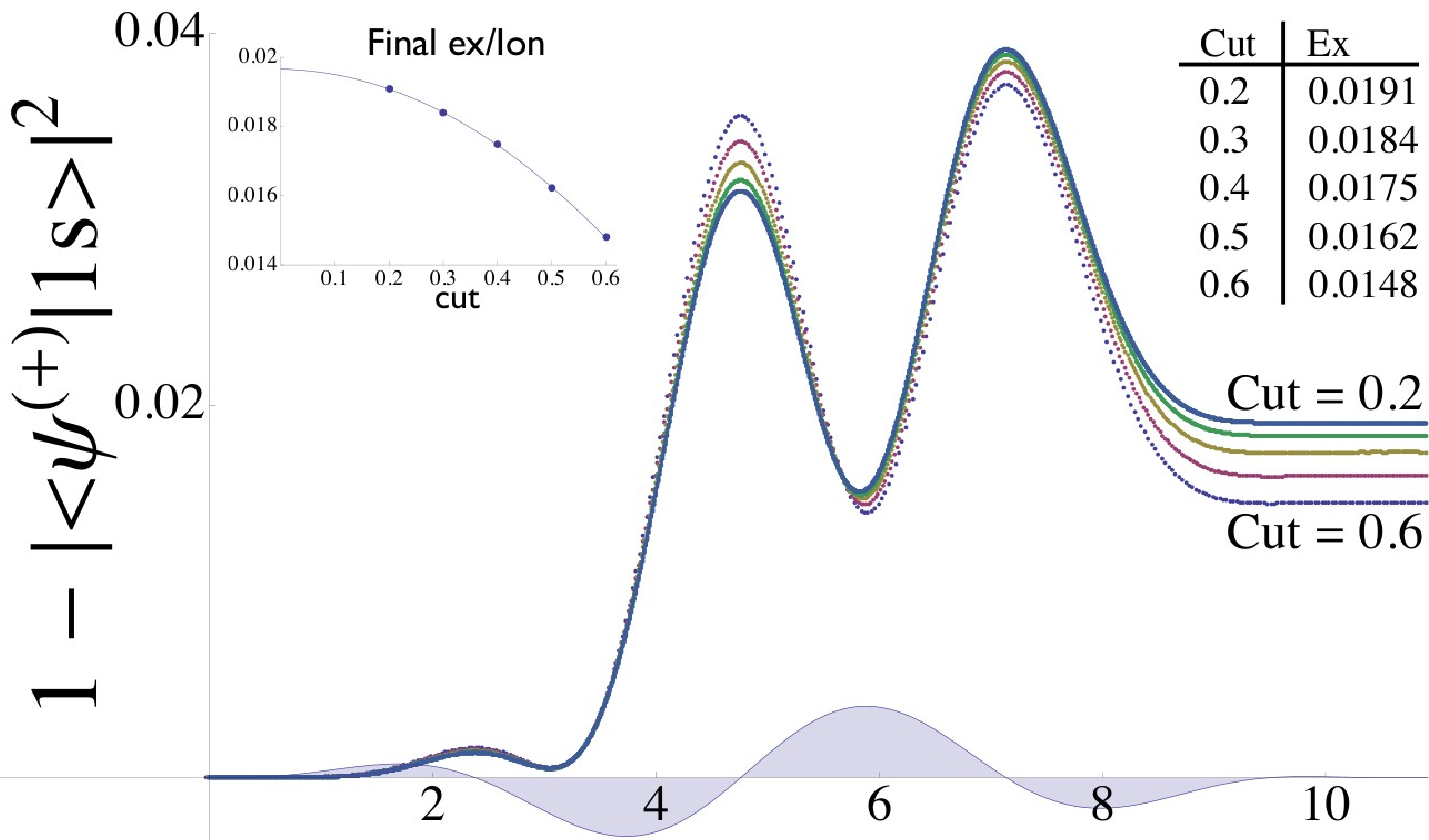


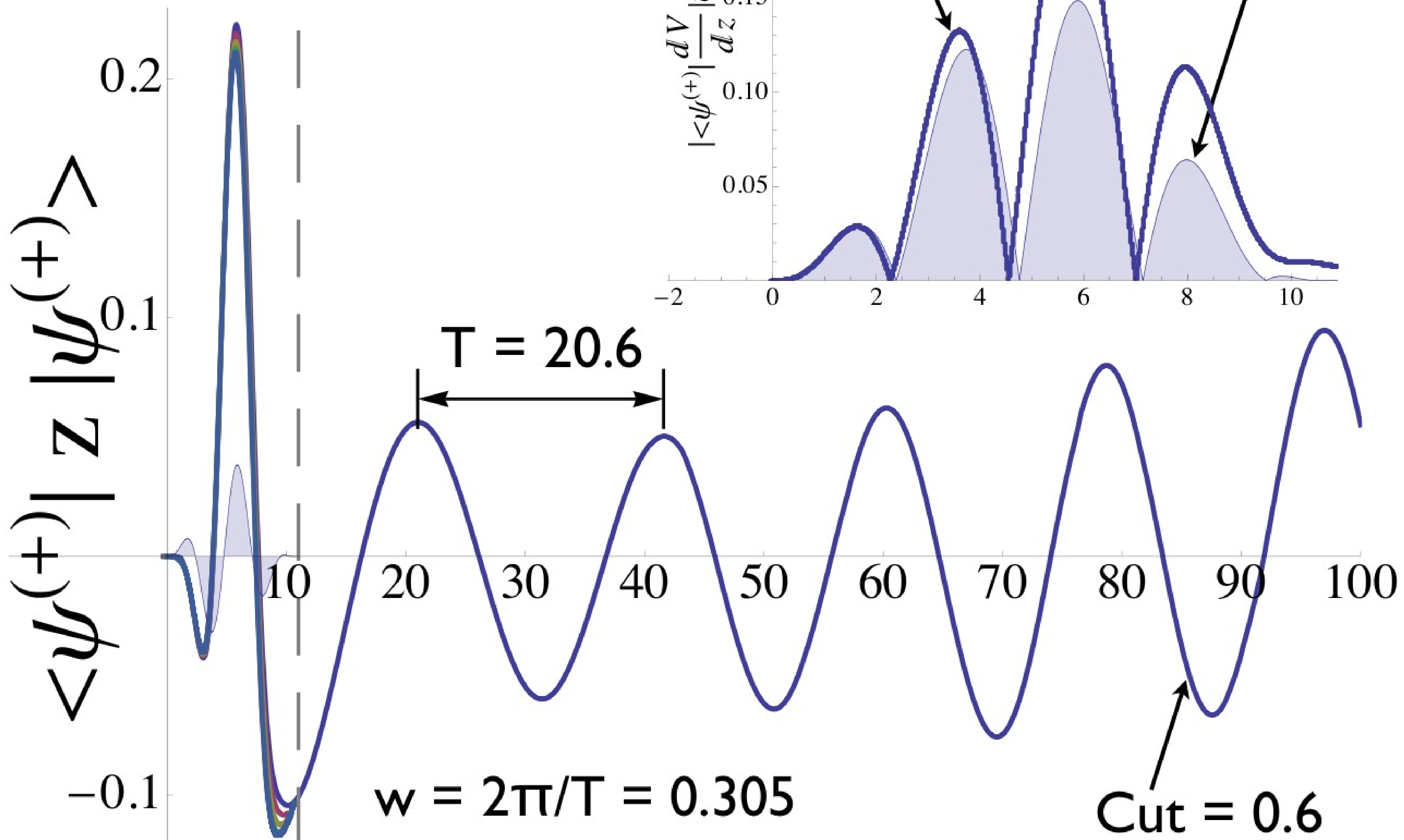




E_0 vs cut
−0.46
−0.47
−0.48
−0.49
−0.50
0.1
0.2
0.3
0.4
0.5
0.6
Cut
Δ Energy vs cut
0.017
0.015
0.013
0.011
0.0
0.1
0.2
0.3
0.4
0.5
0.6
Cut
Energy
−0.42
−0.46
-0.50
Cut = 0.6
Cut = 0.2
Laser Pulse
2
4
6
8
10
Time (a.u)

Total Excitation Probability




z-dipole
|accel|
|laser|
T = 20.6
w = 2π/T = 0.305
Cut = 0.6
$<\psi^{(+)}| z |\psi^{(+)}>$
$|<\psi^{(+)}|\frac{dV}{dz}|\psi^{(+)}>|$
0.2
0.1
−0.1
10
20
30
40
50
60
70
80
90
100
0.20
0.15
0.10
0.05
−2
0
2
4
6
8
10

Summary

- Multiresolution at the petascale offers interesting challenges for the design of software
- Task-oriented computing is just the right fit for the MADNESS architecture
- Multithreading used for fine grain parallelism
- Futures help maintain causality between tasks