JÜLICH
FORSCHUNGSZENTRUM

Molecular Simulations using Monte Carlo and Molecular Dynamics

November 16, 2012

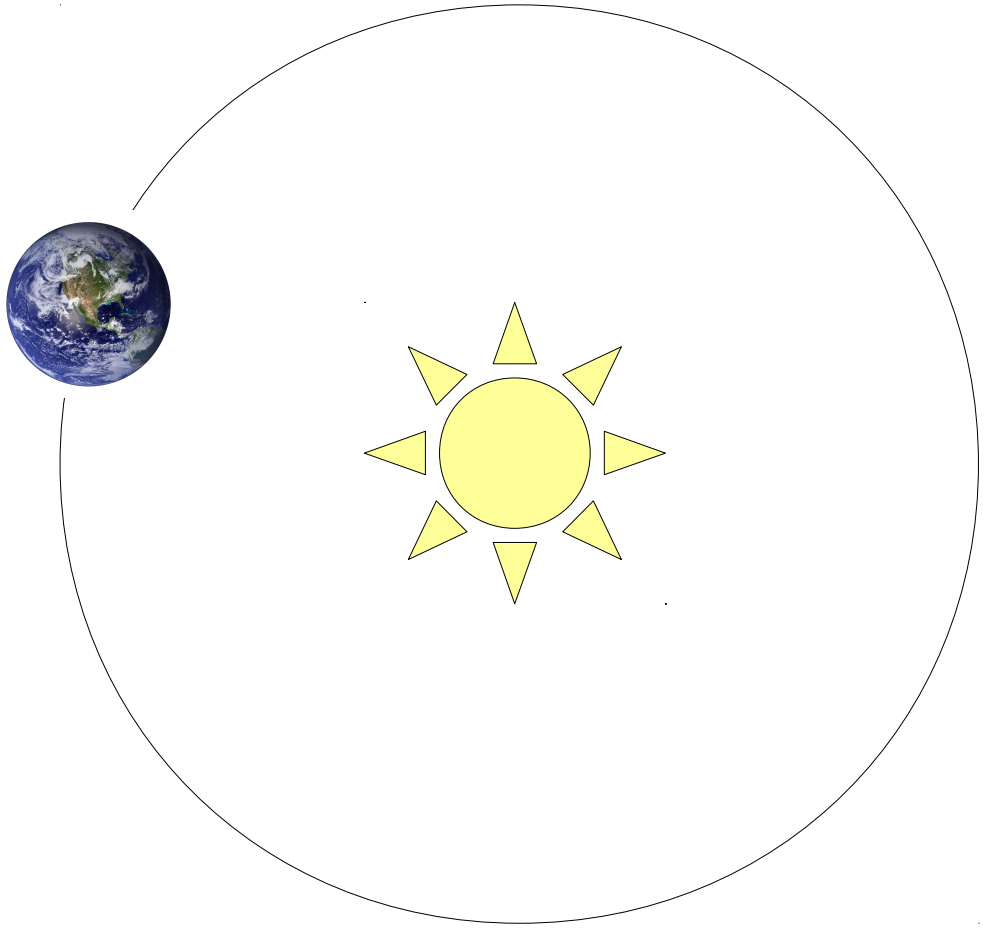
Jan H. Meinke

Research Centre Jülich

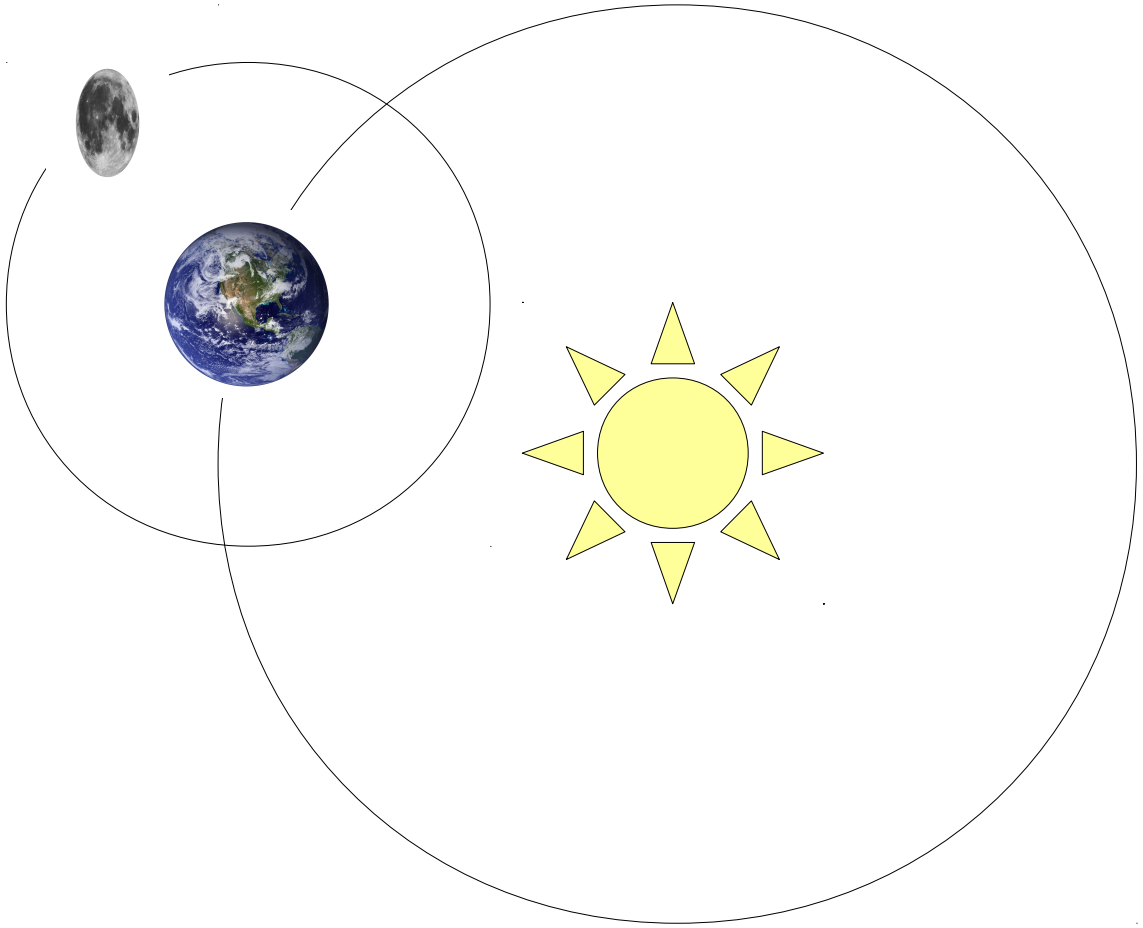
The N Body Problem

November 16, 2012

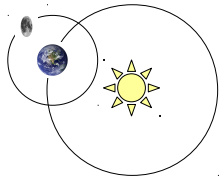
The Two-Body Problem



The Three-Body Problem



The N-Body Problem

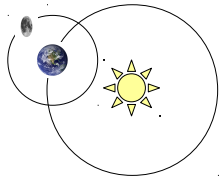


$$\boldsymbol{F_i} = m_i \boldsymbol{a_i}$$

$$\boldsymbol{F_i} = G m_i \sum_{j \neq i} m_j \frac{\boldsymbol{r_{ij}}}{r_{ij}^3}$$

$$\boldsymbol{r_i}(t+dt) = \boldsymbol{r_i}(t) + \boldsymbol{v_i}\, dt + 1/2\, \boldsymbol{a_i}\, dt^2$$

A serial implementation

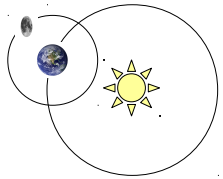


```
void time_step(...){

  const double eps2 = 1e-14;

  for (int i = 0; i < m.size(); ++i){
    double ax = 0, ay = 0, az = 0;
    for (int j = 0; j < m.size(); ++j){
      double dx = (c.x[i] - c.x[j]);
      double dy = (c.y[i] - c.y[j]);
      double dz = (c.z[i] - c.z[j]);
      double d2 = dx * dx  + dy * dy + dz * dz +
eps2;
      double invD3 = 1.0 / (sqrt(d2) * d2);
      ax += m[j] * dx * invD3;
      ay += m[j] * dy * invD3;
      az += m[j] * dz * invD3;
  }
  ax *=  G;
  cnew.x[i] = c.x[i] + v.x[i] * dt + 0.5 * ax * dt *
dt;
…
```

$$\boldsymbol{a}_i = G \sum_{j \neq i} m_j \frac{\boldsymbol{r}_{ij}}{r_{ij}^3}$$

$$\boldsymbol{r}_i(t+dt) = \boldsymbol{r}_i(t) + \boldsymbol{v}_i\, dt + 1/2\, \boldsymbol{a}_i\, dt^2$$

A faster serial implementation



We can take advantage of Newton's third law

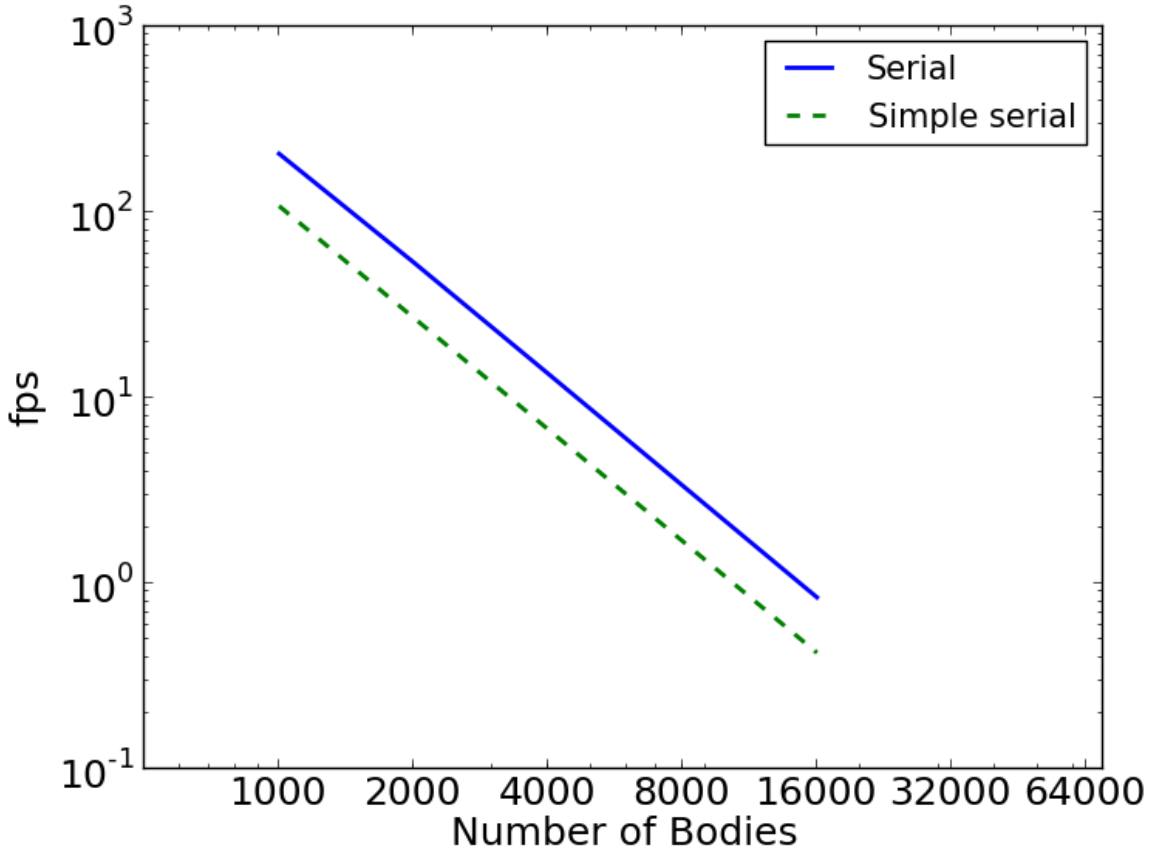
$$\boldsymbol{F}_{ij} = -\boldsymbol{F}_{ji}$$

and cut the work in half.

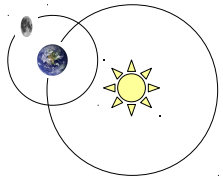
```
for (int i = 0; i < m.size(); ++i){
  for (int j = i + 1; j < m.size(); ++j){
    double dx = (c.x[i] - c.x[j]);
    double dy = (c.y[i] - c.y[j]);
    double dz = (c.z[i] - c.z[j]);
    double d2 = dx * dx  + dy * dy + dz * dz + eps2;
    double invD3 = 1.0 / (sqrt(d2) * d2);
    double fx = m[i] * m[j] * dx * invD3;
    double fy = m[i] * m[j] * dy * invD3;
    double fz = m[i] * m[j] * dz * invD3;
    a.x[i] += fx; a.x[j] -= fx;
    a.y[i] += fy; a.y[j] -= fy;
    a.z[i] += fz; a.z[j] -= fz;
    }
}
```

$$\boldsymbol{a}_i = G \sum_{j \neq i} m_j \frac{\boldsymbol{r}_{ij}}{r_{ij}^3}$$

Scaling with System Size



A faster serial implementation



We can take advantage of Newton's third law

$$\boldsymbol{F}_{ij} = -\boldsymbol{F}_{ji}$$

and cut the work in half.

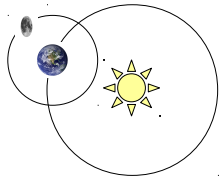
```
for (int i = 0; i < m.size(); ++i){
  for (int j = i + 1; j < m.size(); ++j){
    double dx = (c.x[i] - c.x[j]);
    double dy = (c.y[i] - c.y[j]);
    double dz = (c.z[i] - c.z[j]);
    double d2 = dx * dx + dy * dy + dz * dz + eps2;
    double invD3 = 1.0 / (sqrt(d2) * d2);
    double fx = m[i] * m[j] * dx * invD3;
    double fy = m[i] * m[j] * dy * invD3;
    double fz = m[i] * m[j] * dz * invD3;
    a.x[i] += fx; a.x[j] -= fx;
    a.y[i] += fy; a.y[j] -= fy;
    a.z[i] += fz; a.z[j] -= fz;
    }
}
```

Need more memory

Harder to parallelize

$$\boldsymbol{a}_i = G \sum_{j \neq i} m_j \frac{\boldsymbol{r}_{ij}}{r_{ij}^3}$$

A Serial Implementation

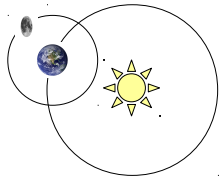


```
void time_step(...){

  const double eps2 = 1e-14;

  for (int i = 0; i < m.size(); ++i){
    double ax = 0, ay = 0, az = 0;
    for (int j = 0; j < m.size(); ++j){
      double dx = (c.x[i] - c.x[j]);
      double dy = (c.y[i] - c.y[j]);
      double dz = (c.z[i] - c.z[j]);
      double d2 = dx * dx  + dy * dy + dz * dz +
eps2;
      double invD3 = 1.0 / (sqrt(d2) * d2);
      ax += m[j] * dx * invD3;
      ay += m[j] * dy * invD3;
      az += m[j] * dz * invD3;
  }
  ax *=  G;
  cnew.x[i] = c.x[i] + v.x[i] * dt + 0.5 * ax * dt *
dt;
…
```

$$\boldsymbol{a}_i = G \sum_{j \neq i} m_j \frac{\boldsymbol{r}_{ij}}{r_{ij}^3}$$

$$\boldsymbol{r}_i(t+dt) = \boldsymbol{r}_i(t) + \boldsymbol{v}_i\, dt + 1/2\, \boldsymbol{a}_i\, dt^2$$

A Parallel Implementation with OpenMP

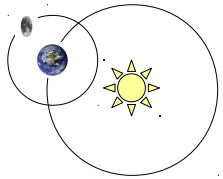


```
void time_step(...){

  const double eps2 = 1e-14;

#pragma omp for
  for (int i = 0; i < m.size(); ++i){
    double ax = 0, ay = 0, az = 0;
    for (int j = 0; j < m.size(); ++j){
      double dx = (c.x[i] - c.x[j]);
      double dy = (c.y[i] - c.y[j]);
      double dz = (c.z[i] - c.z[j]);
      double d2 = dx * dx  + dy * dy + dz * dz +
eps2;
      double invD3 = 1.0 / (sqrt(d2) * d2);
      ax += m[j] * dx * invD3;
      ay += m[j] * dy * invD3;
      az += m[j] * dz * invD3;
  }
  ax *=  G;
  cnew.x[i] = c.x[i] + v.x[i] * dt + 0.5 * ax * dt *
dt;
…
```

$$\boldsymbol{a}_i = G \sum_{j \neq i} m_j \frac{\boldsymbol{r}_{ij}}{r_{ij}^3}$$

$$\boldsymbol{r}_i(t+dt) = \boldsymbol{r}_i(t) + \boldsymbol{v}_i\, dt + 1/2\, \boldsymbol{a}_i\, dt^2$$

A Parallel Implementation with Cilk Plus

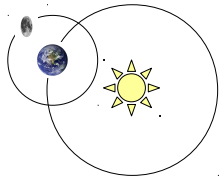


```
void time_step(...){

  const double eps2 = 1e-14;

  _Cilk_for (int i = 0; i < m.size(); ++i){
    double ax = 0, ay = 0, az = 0;
    for (int j = 0; j < m.size(); ++j){
      double dx = (c.x[i] - c.x[j]);
      double dy = (c.y[i] - c.y[j]);
      double dz = (c.z[i] - c.z[j]);
      double d2 = dx * dx  + dy * dy + dz * dz +
eps2;
      double invD3 = 1.0 / (sqrt(d2) * d2);
      ax += m[j] * dx * invD3;
      ay += m[j] * dy * invD3;
      az += m[j] * dz * invD3;
  }
  ax *=  G;
  cnew.x[i] = c.x[i] + v.x[i] * dt + 0.5 * ax * dt *
dt;
…
```

$$\boldsymbol{a}_i = G \sum_{j \neq i} m_j \frac{\boldsymbol{r}_{ij}}{r_{ij}^3}$$

$$\boldsymbol{r}_i(t+dt) = \boldsymbol{r}_i(t) + \boldsymbol{v}_i\, dt + 1/2\, \boldsymbol{a}_i\, dt^2$$

Using the GPU

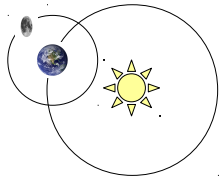


```
void time_step(...){

  for (int i = 0; i < n; ++i){
    data_t ax = 0;
    data_t ay = 0;
    data_t az = 0;
    for (int j = 0; j < n; ++j){
      if (i == j) continue;
      data_t dx = (x[i] - x[j]);
      data_t dy = (y[i] - y[j]);
      data_t dz = (z[i] - z[j]);
      data_t d2 = dx * dx  + dy * dy + dz * dz;
      data_t invD3 = 1.0 / (sqrt(d2) * d2);
      ax += m[j] * dx * invD3;
      ay += m[j] * dy * invD3;
      az += m[j] * dz * invD3;
  }
…
```

$$\boldsymbol{a}_i = G \sum_{j \neq i} m_j \frac{\boldsymbol{r}_{ij}}{r_{ij}^3}$$

This makes a good kernel!

Using the GPU with OpenACC



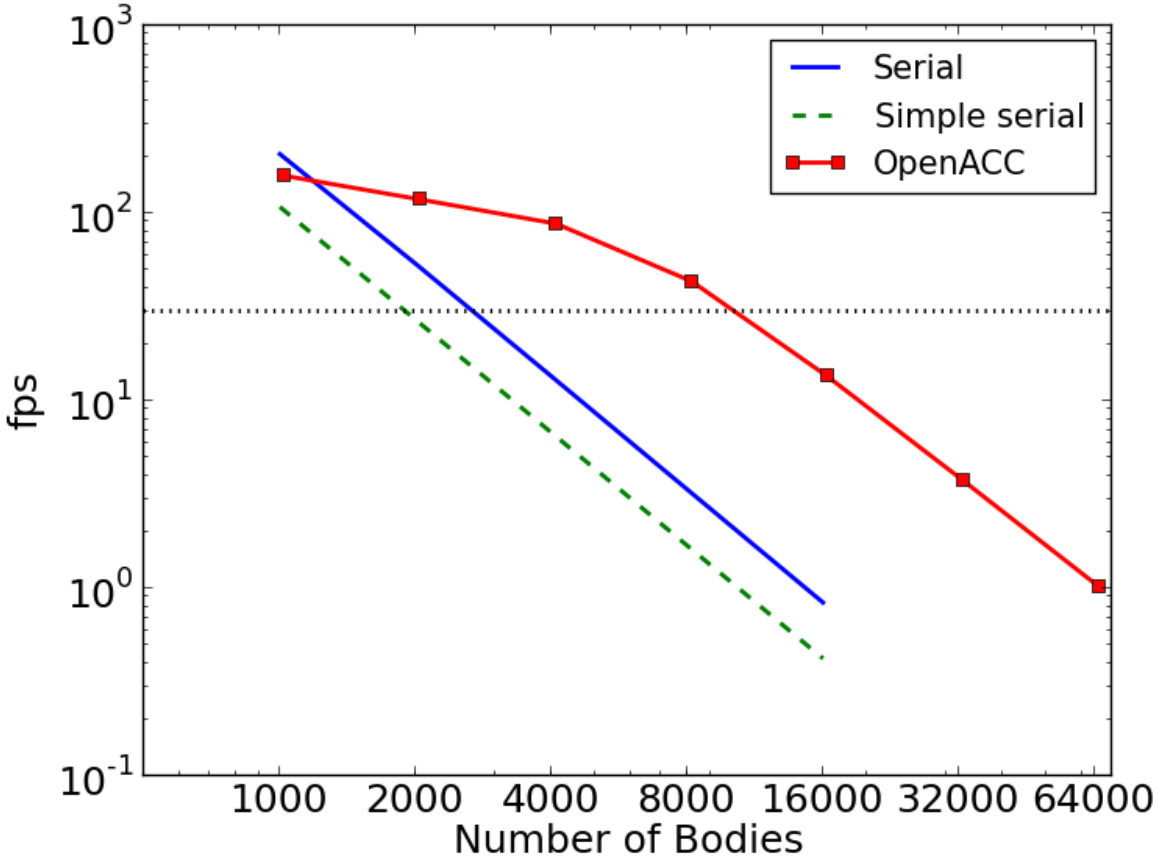
```
void time_step(...){

#pragma acc kernels
  for (int i = 0; i < n; ++i){
    data_t ax = 0;
    data_t ay = 0;
    data_t az = 0;
    for (int j = 0; j < n; ++j){
      if (i == j) continue;
      data_t dx = (x[i] - x[j]);
      data_t dy = (y[i] - y[j]);
      data_t dz = (z[i] - z[j]);
      data_t d2 = dx * dx  + dy * dy + dz * dz;
      data_t invD3 = 1.0 / (sqrt(d2) * d2);
      ax += m[j] * dx * invD3;
      ay += m[j] * dy * invD3;
      az += m[j] * dz * invD3;
  }
…
```

$$\boldsymbol{a}_i = G \sum_{j \neq i} m_j \frac{\boldsymbol{r}_{ij}}{r_{ij}^3}$$

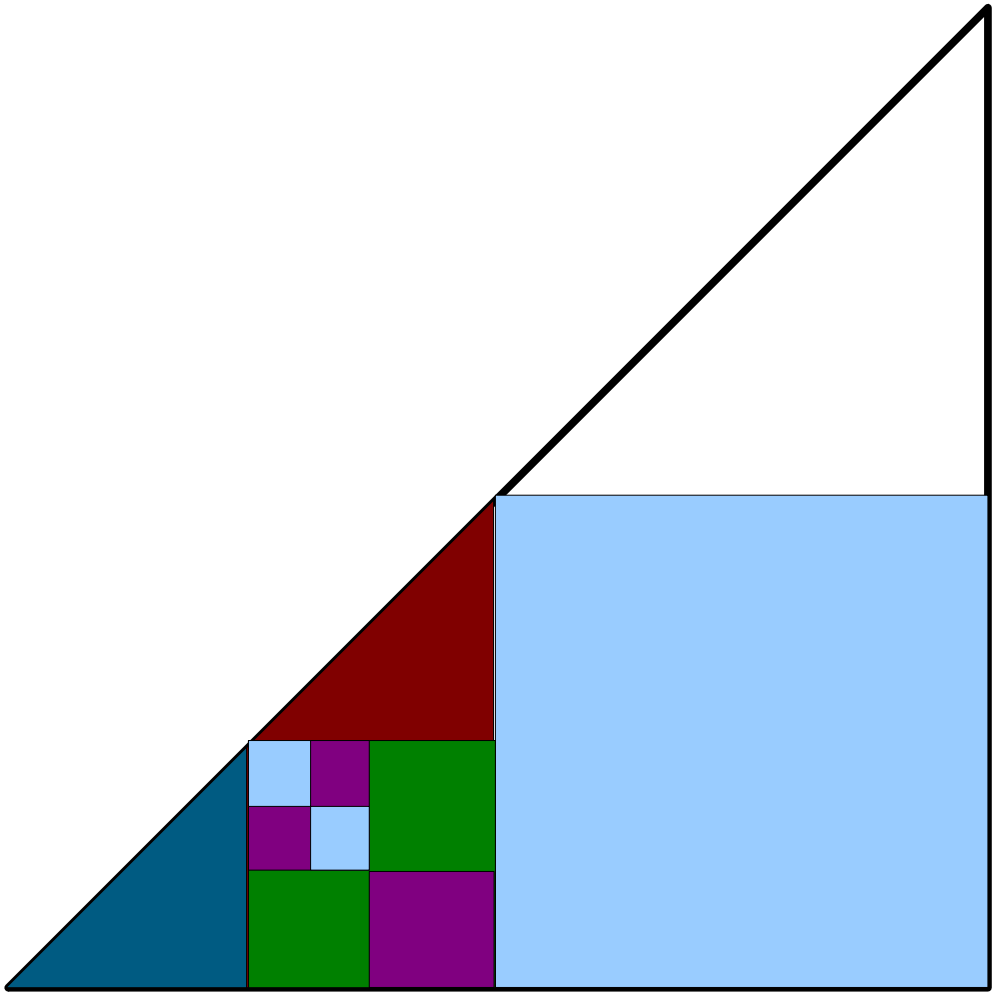
PGI Compiler Feedback for acc kernels

```
32, Generating copyin(m[0:n])
    Generating copyin(z[0:n])
    Generating copyin(y[0:n])
    Generating copyin(x[0:n])
    Generating copy(vx[0:n])
    Generating copyout(xnew[0:n])
    Generating copy(vy[0:n])
    Generating copyout(ynew[0:n])
    Generating copy(vz[0:n])
    Generating copyout(znew[0:n])
    Generating compute capability 1.3 binary
    Generating compute capability 2.0 binary
33, Loop is parallelizable
    Accelerator kernel generated
    33, #pragma acc loop gang, vector(128) // blockIdx.x threadIdx.x
        CC 1.3 : 38 registers; 112 shared, 4 constant, 0 local memory bytes
        CC 2.0 : 38 registers; 0 shared, 168 constant, 0 local memory bytes
38, Loop is parallelizable
```

Scaling with System Size

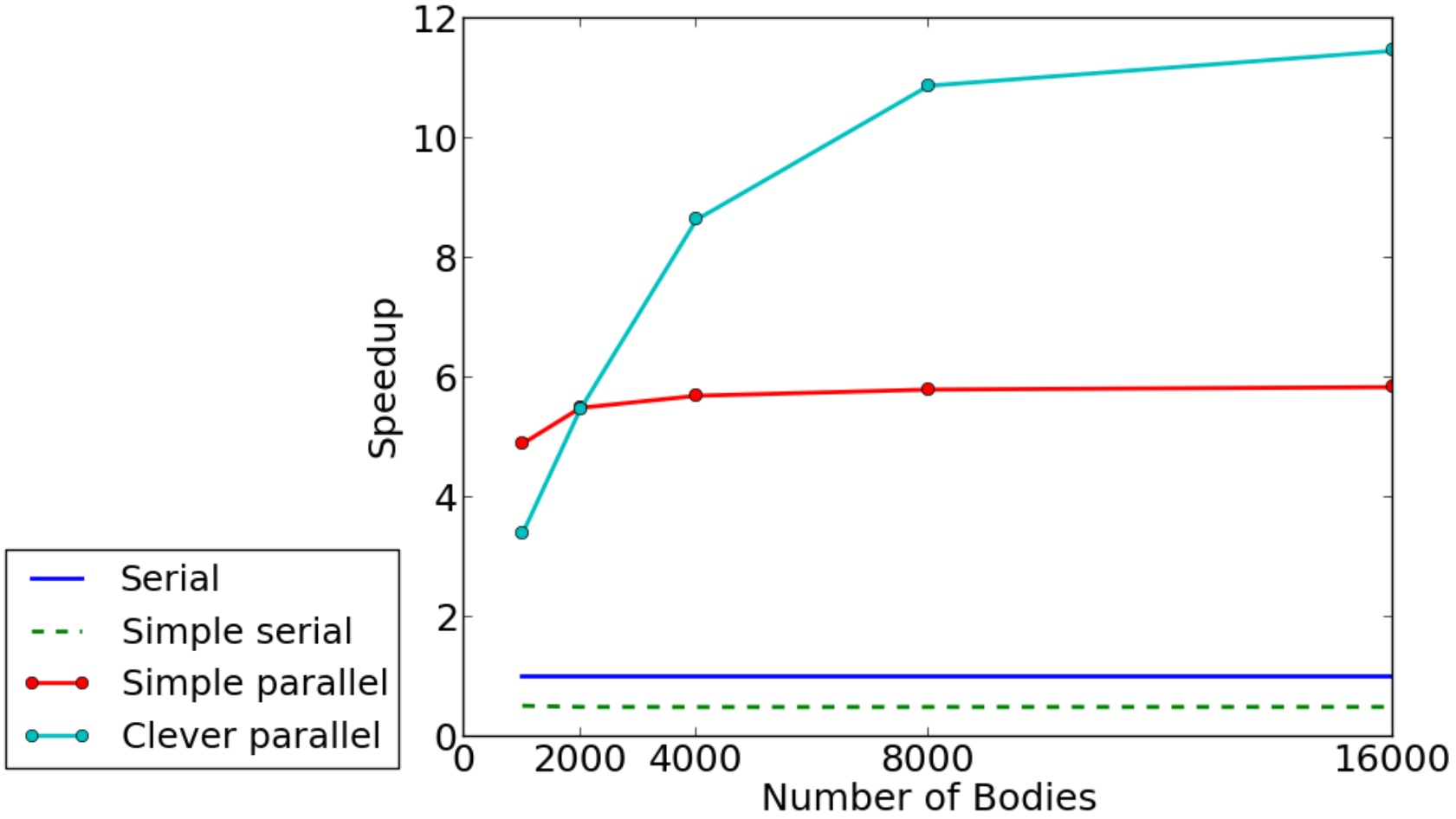


A Clever Parallel Implementation

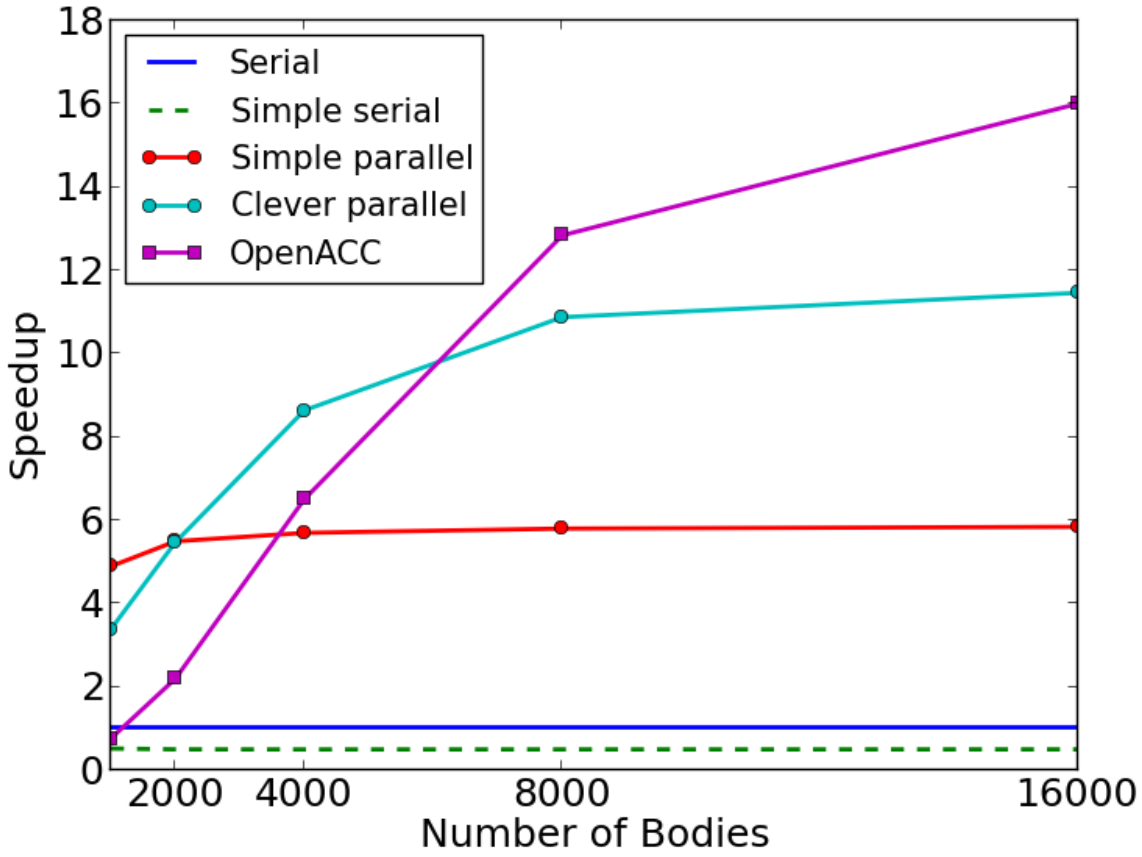


Steven Lewin-Berlin,
http://software.intel.com
/en-us/articles/A-cute-
technique-for-avoiding-
certain-race-conditions

A Clever Parallel Implementation with Cilk Plus



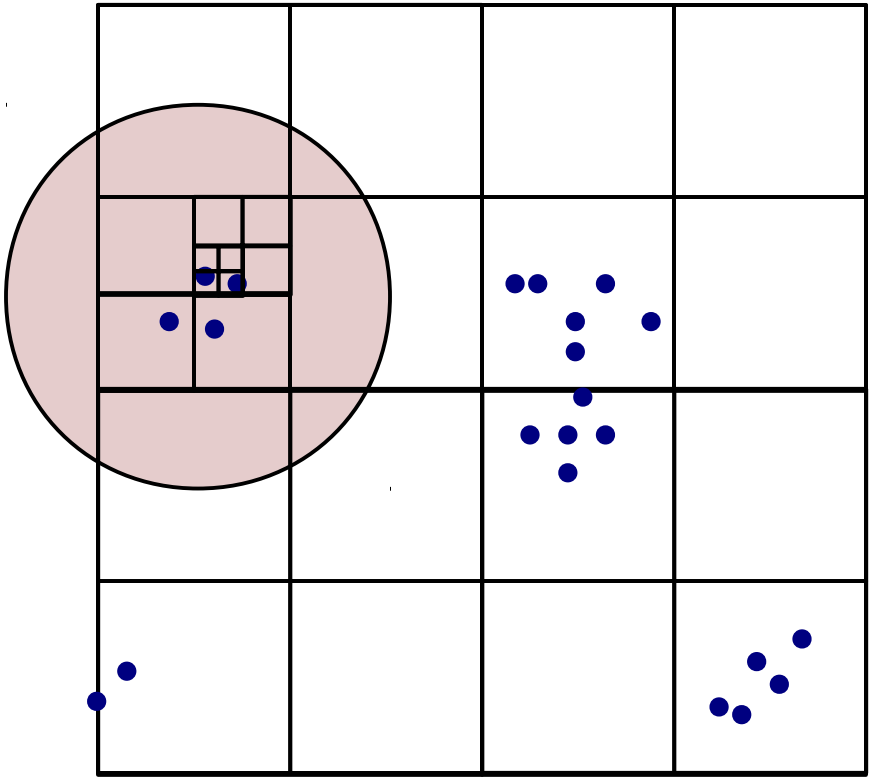
Comparing all implementations



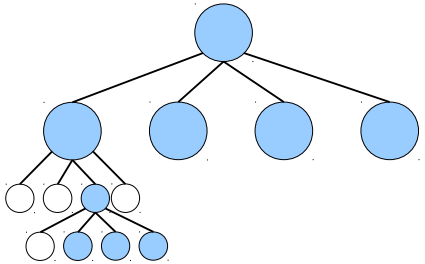
Choosing a Better Algorithm

$$O(n^2) \rightarrow O(n \log n)$$

Staying in the 'Hood

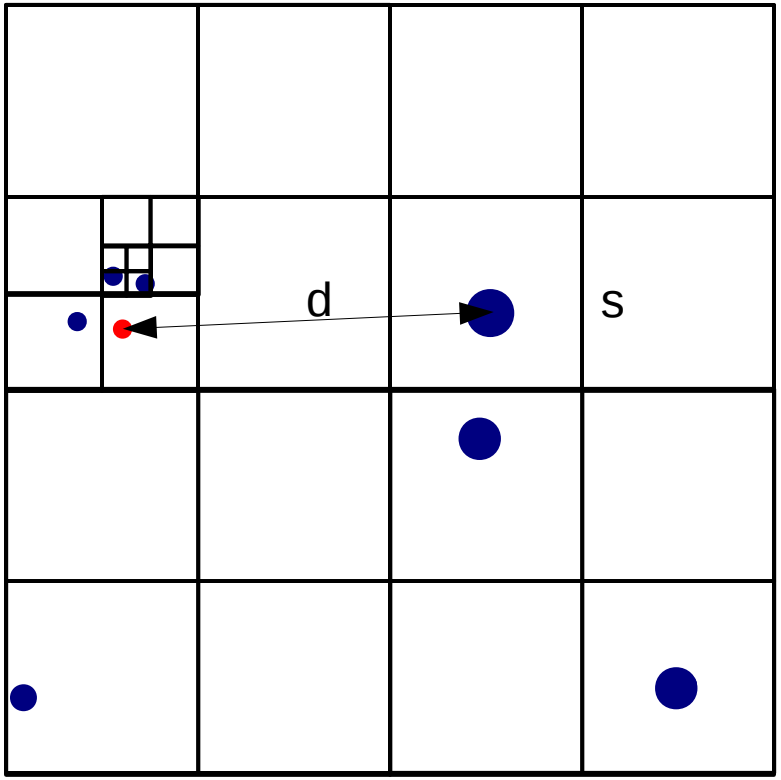


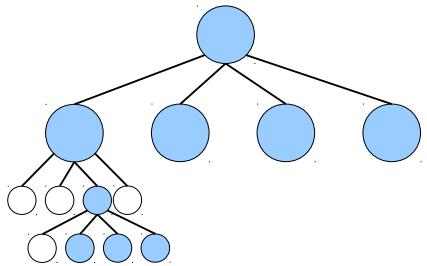
- 1, 2, 3, 4
- ...



Long Range Interactions

$s/d < \theta$



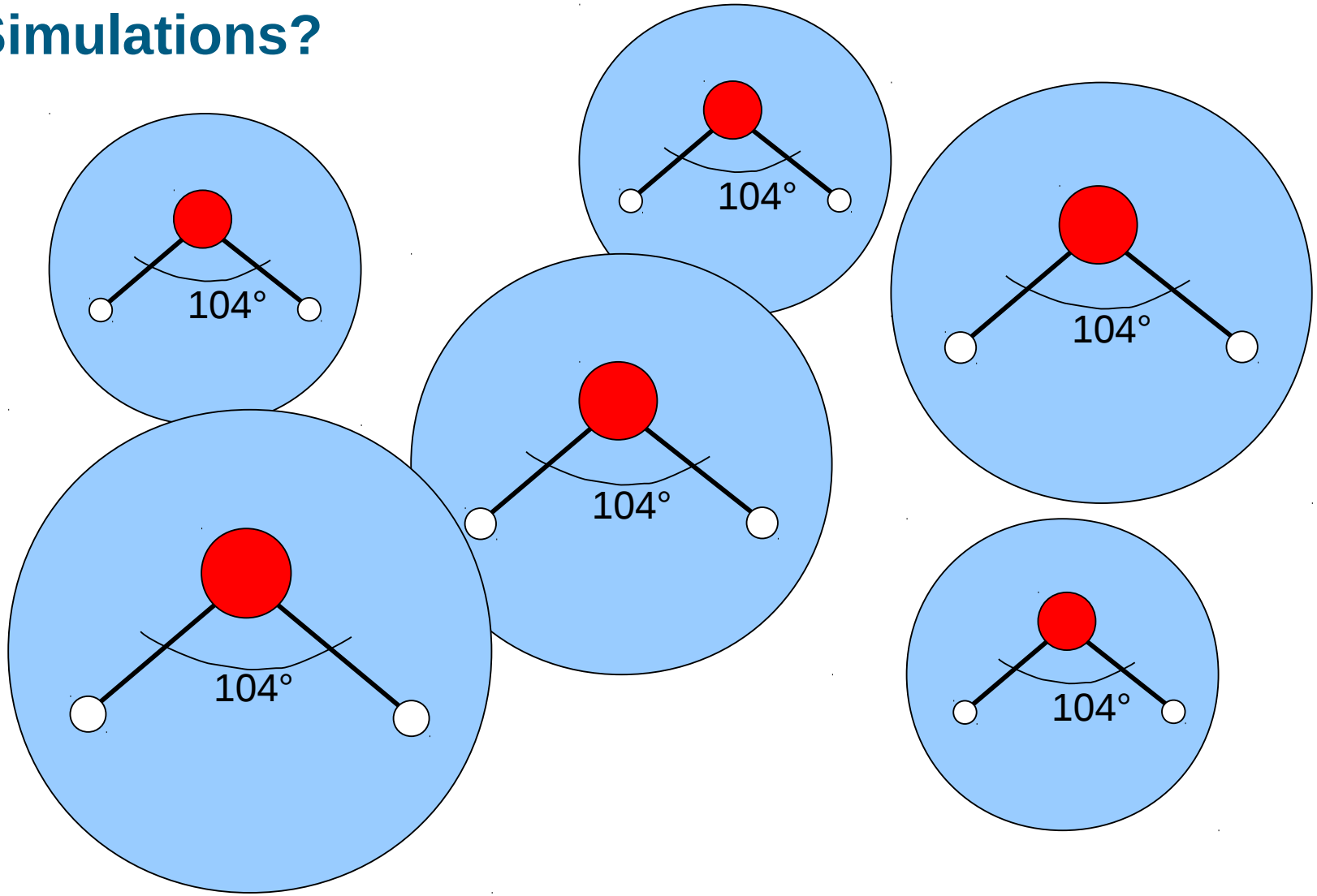


J. Barnes and P. Hut, Nature **324**, 446 (1986).
S. Pfalzner and P. Gibbon, Many-Body Tree Methods in Physics (Cambridge University Press, 1996).

Better Integration Schemes

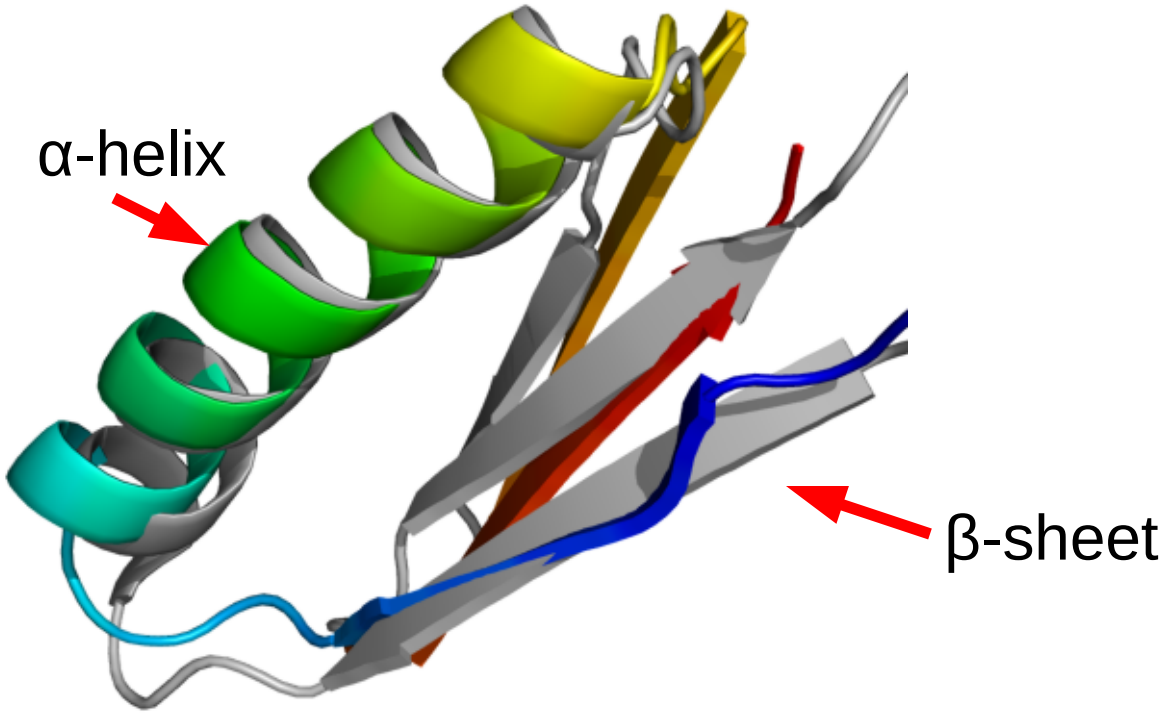
- Leap frog
- Velocity Verlet

What does this have to do with Molecular Simulations?



From Sequence to 3D Shape

ERVRISITARTKKEAEKFAAILIKVFAELGYNDINVTWDGDTVTEGQL



Levinthal Paradox

150 amino acid – 3 conformations each

$$3^{150}$$

$$=$$

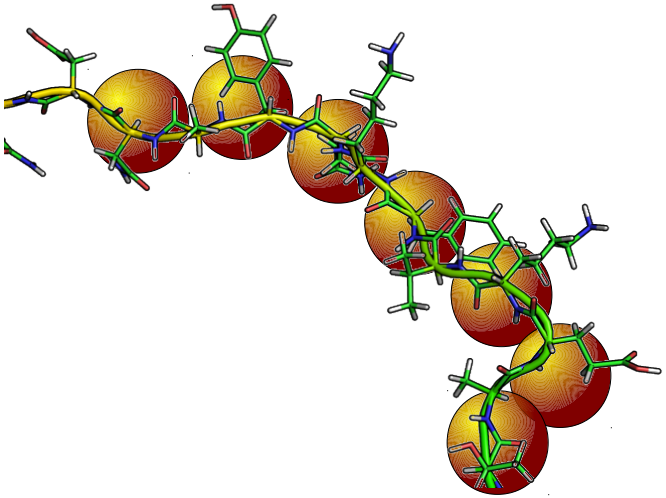
369,988,485,035,126,972,924,700,782,451,696,644,186,473,100,389,722,973,815,184,405,301,748,249

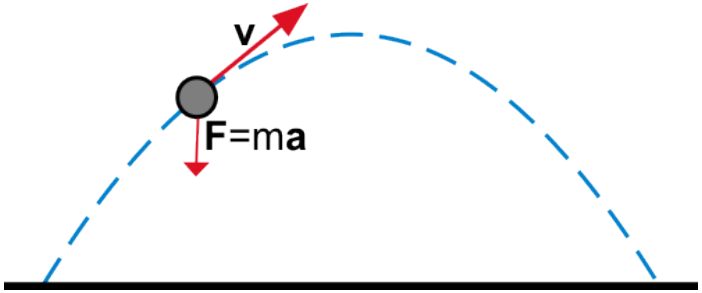
$$\approx$$

$$(370 \times 10^{69})$$

$$\rightarrow 10^{53} \text{ years}$$

Modeling proteins



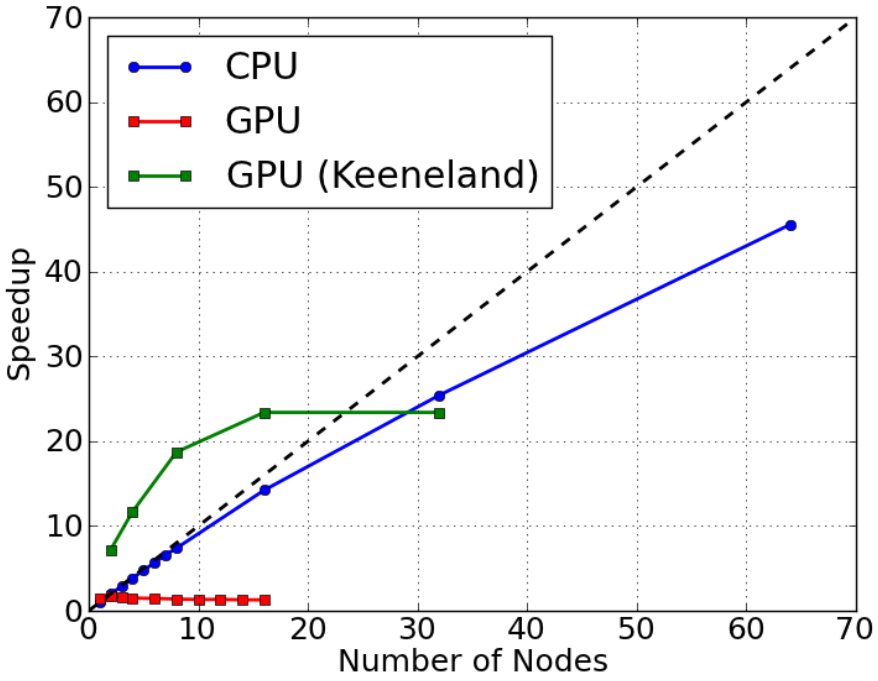


Molecular Dynamics

Molecular Modeling

- Molecular Dynamics
  - Gromacs
  - NAMD

} Perform part of the force calculation on GPU

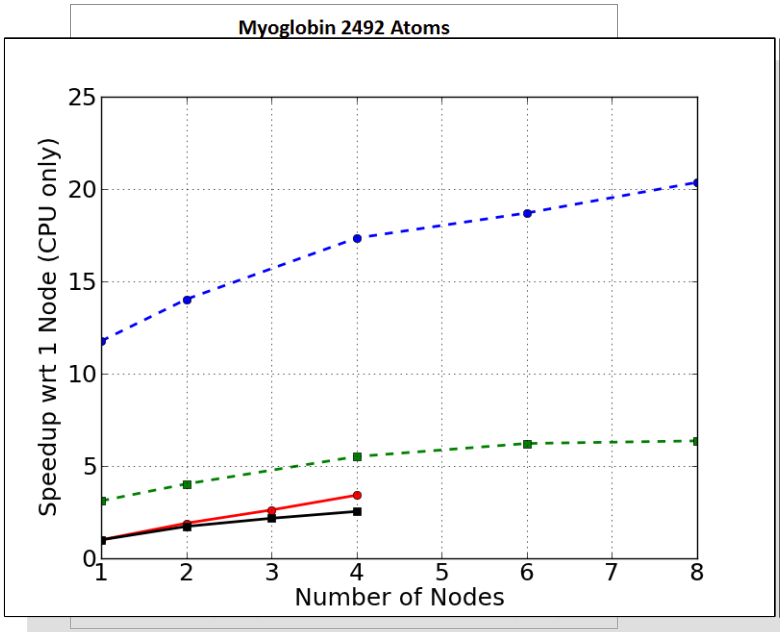


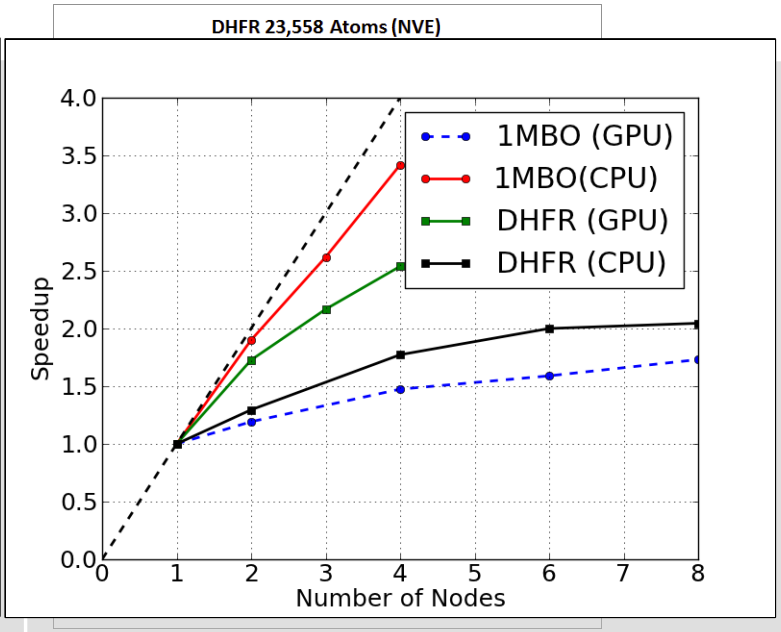
Molecular Modeling

- Molecular Dynamics
  - Gromacs
  - NAMD
  - AMBER

} Perform part of the force calculation on GPU

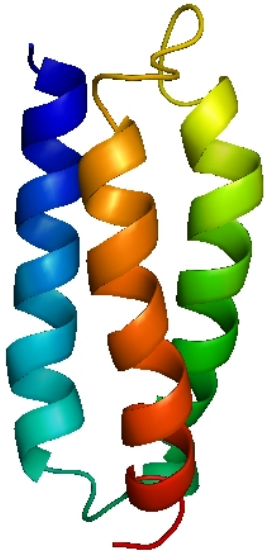
} Entire simulation on GPU





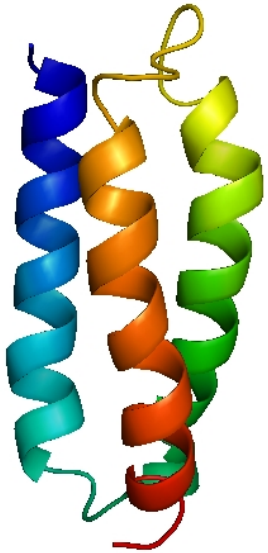
Molecular Modeling

- Molecular Dynamics
  - Gromacs
  - NAMD
  - AMBER
- Monte Carlo
  - SMMP (SLBio)
  - ProFASi (SLBio, planned)



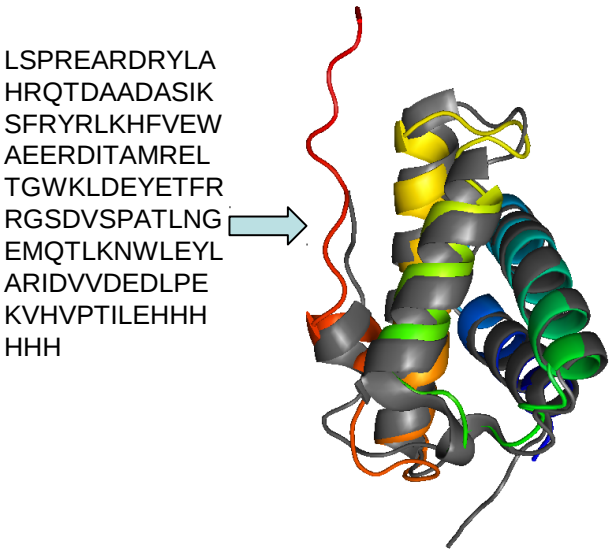
Molecular Modeling

- Molecular Dynamics
  - Gromacs
  - NAMD
  - AMBER
- Monte Carlo
  - SMMP (SLBio)
  - ProFASi (SLBio, planned)
- Brownian Dynamics
  - BDBox



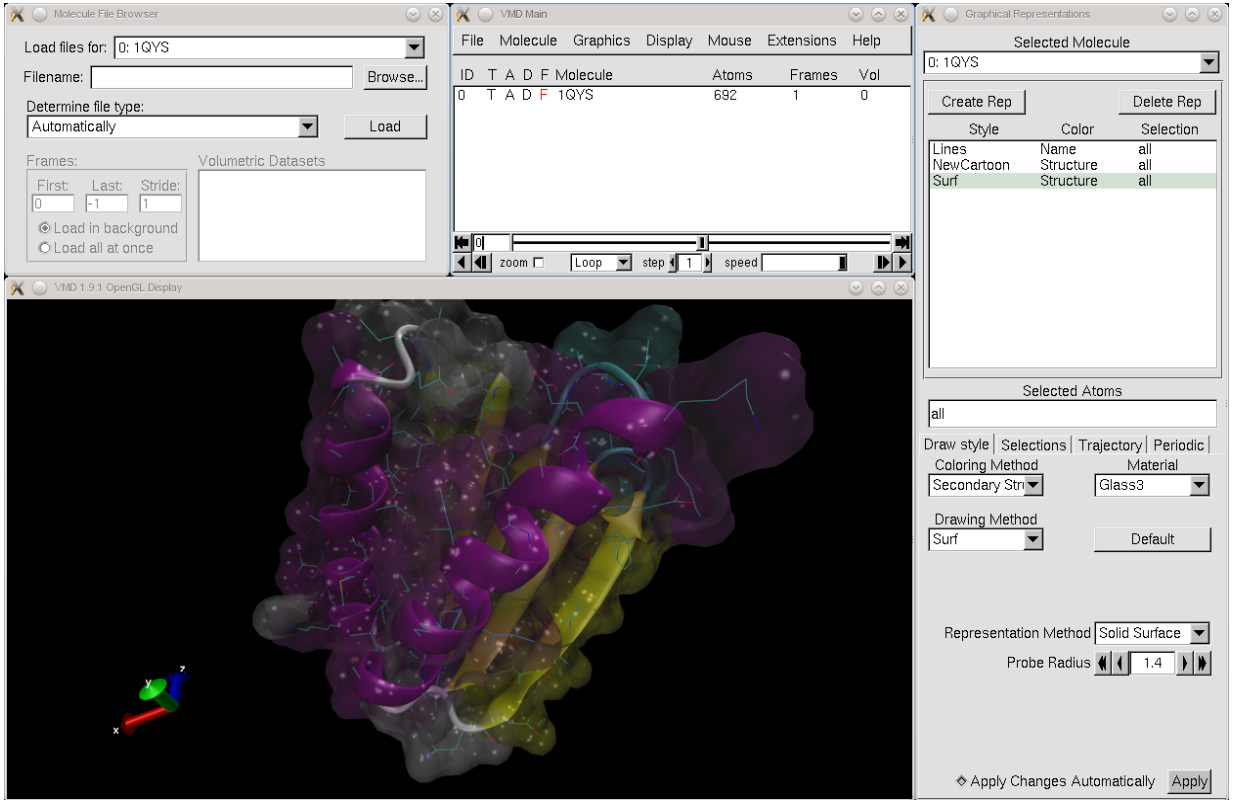
Sequence Alignment & Machine Learning

- MummerGPU (NGS)
- cudasw++
- GPUSvm
- GPUMLib



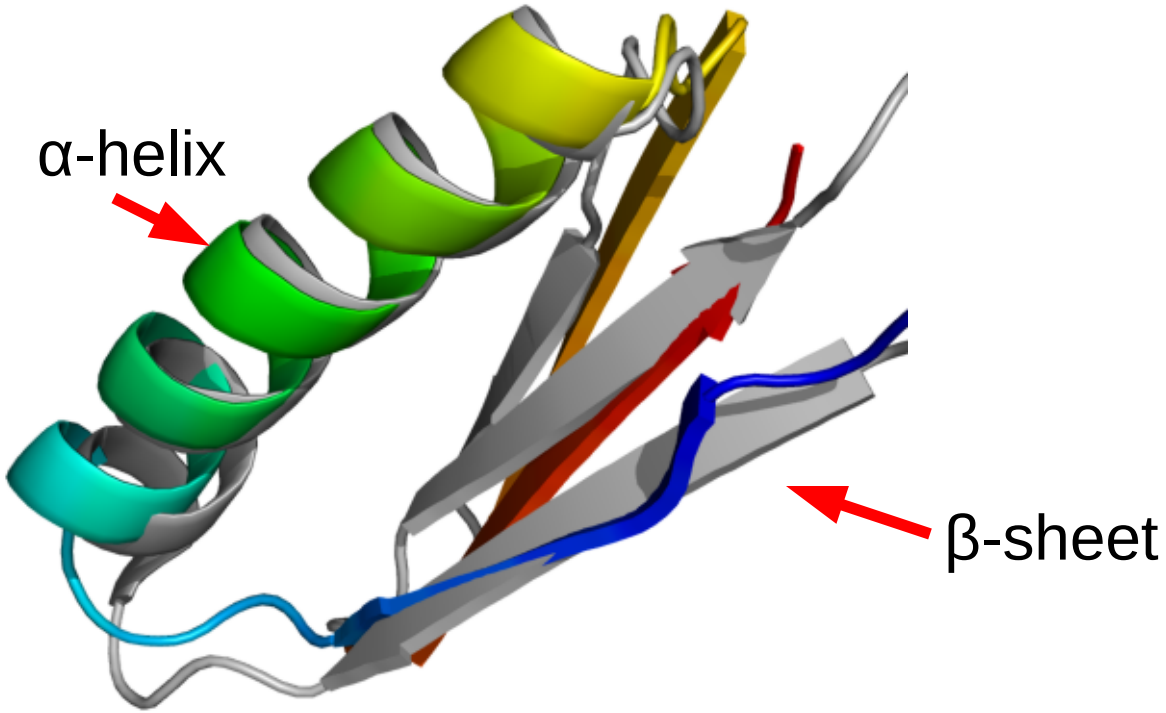
Data Analysis & Visualization

- Dr. L (Dimensional Reduction Library)
- VMD
- PyMAFIA (SLBio, planned)



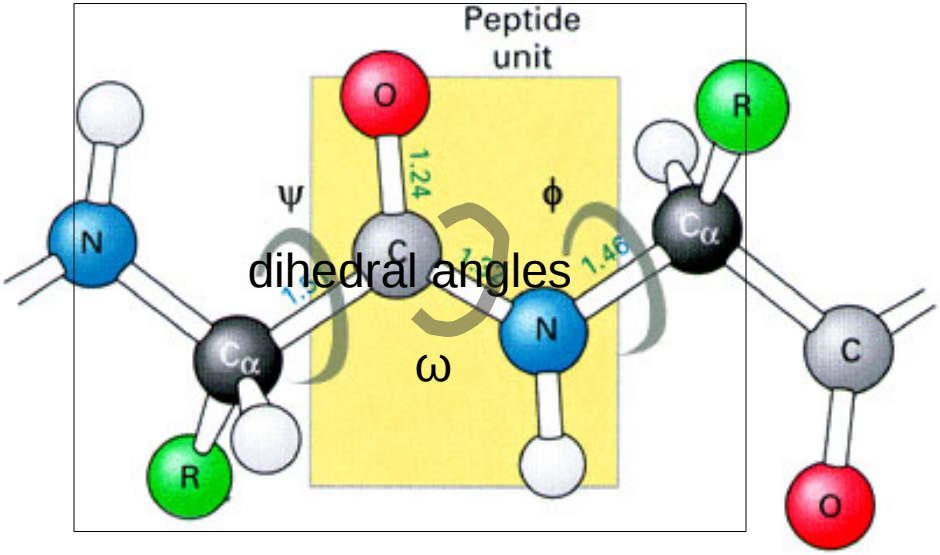
Proteins

ERVRISITARTKKEAEKFAAILIKVFAELGYNDINVTWDGDTVTEGQL



Simple Molecular Mechanics for Proteins (SMMP)

- Protein simulations with Monte Carlo
- Standard geometry (bond length and angle fixed)
- Dihedrals are degrees of freedom
- Force field: ECEPP/3



http://apple.sysbio.info/~mjhsieh/sstour/

Python

and GPUs

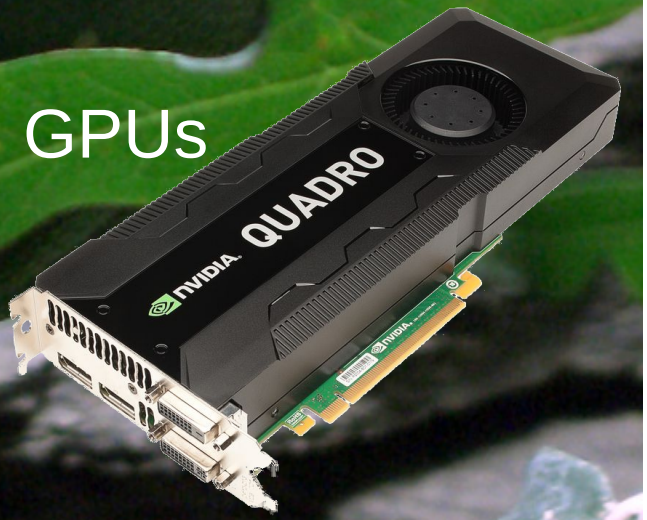


Photo by Mike Wesemann

Python and GPUs

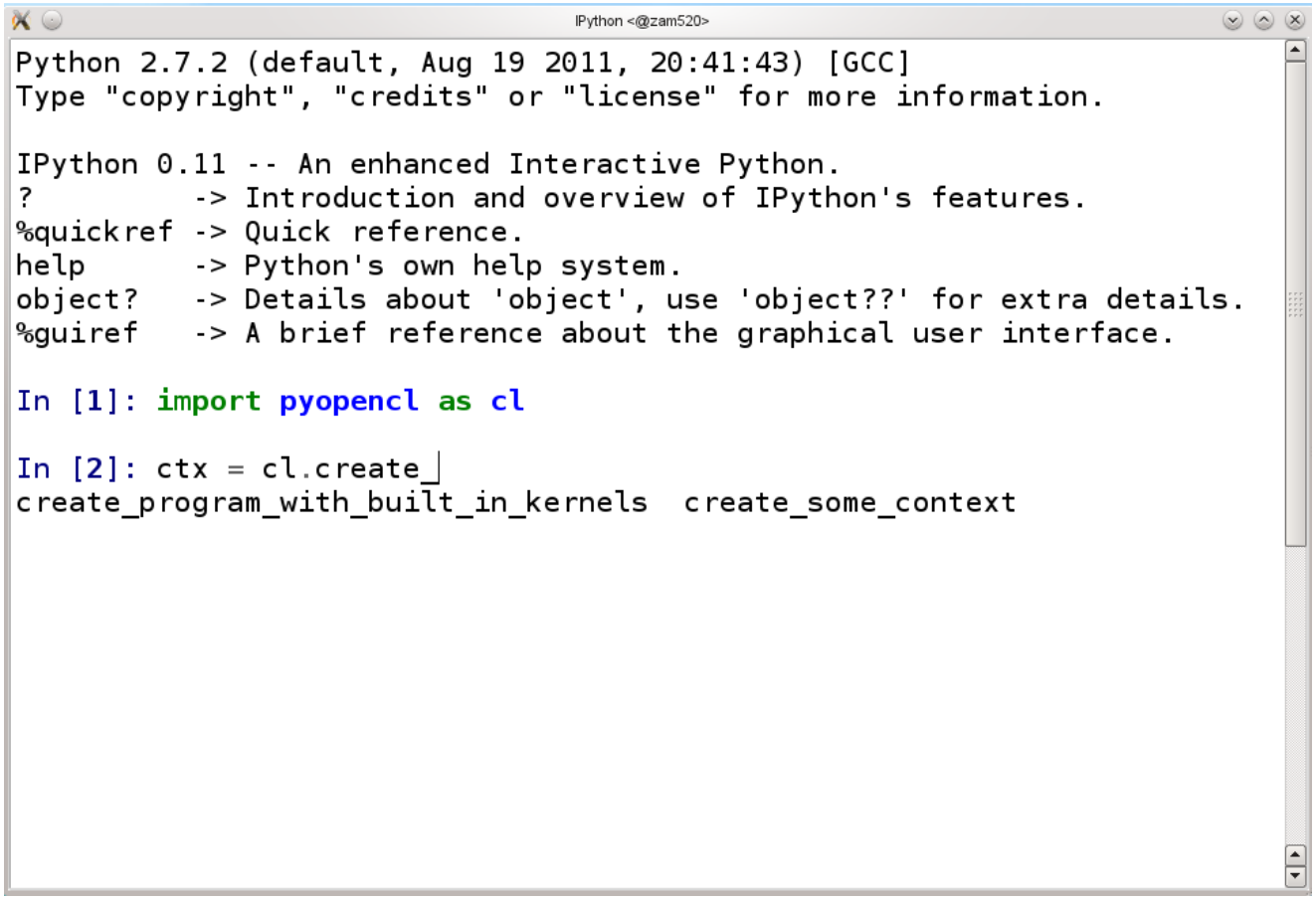
- Excellent bindings for CUDA (PyCUDA) and OpenCL (PyOpenCL) by Andreas Klöckner (http://mathema.tician.de/software/pycuda)
- Copperhead executes annotated Python code on GPU (http://copperhead.github.com/)
- Atomic Hedgehog translates annotated Python code to OpenCL (http://ahh.bitbucket.org)

For some nice talks see GTC on Demand and search for python

Copperhead

```
from copperhead import *
import numpy as np
@cu
def axpy(a, x, y):
  return [a * xi + yi for xi, yi in zip(x, y)]
x = np.arange(100, dtype=np.float64)
y = np.arange(100, dtype=np.float64)
with places.gpu0:
  gpu = axpy(2.0, x, y)
with places.openmp:
  cpu = axpy(2.0, x, y)
```

Python Interactive

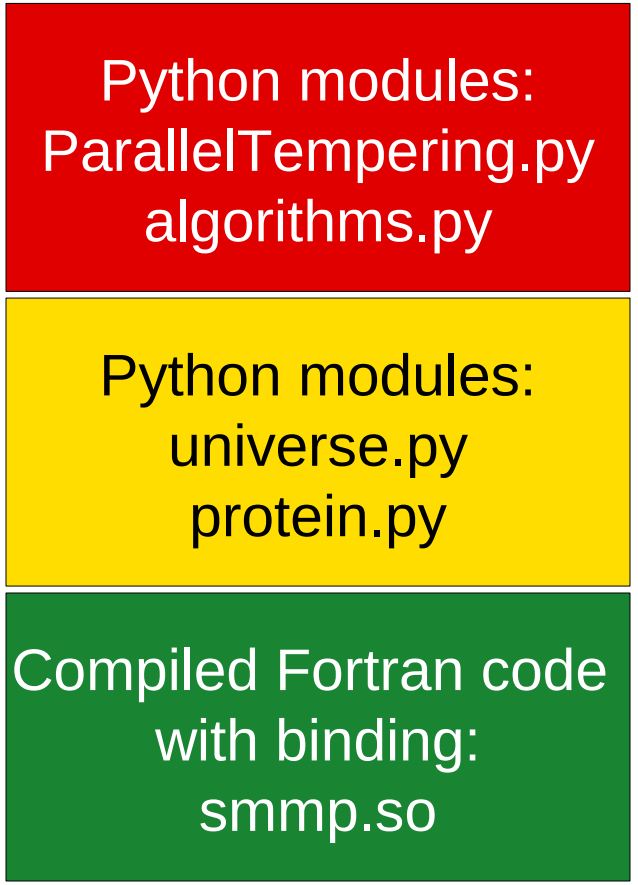


PySMMP

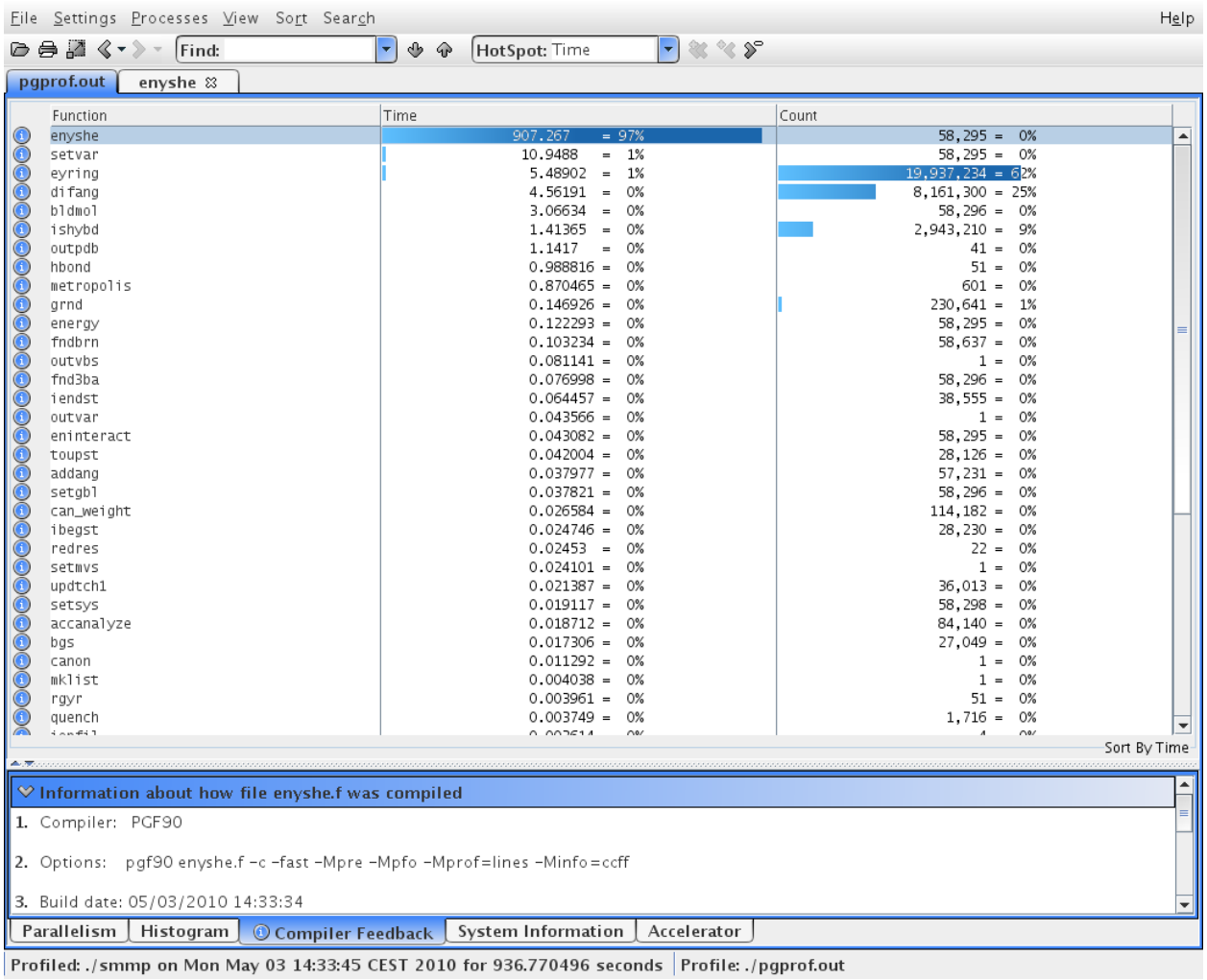
Algorithms implemented on top of PySMMP

Wrapper around SMMP's internal data structure and property functions

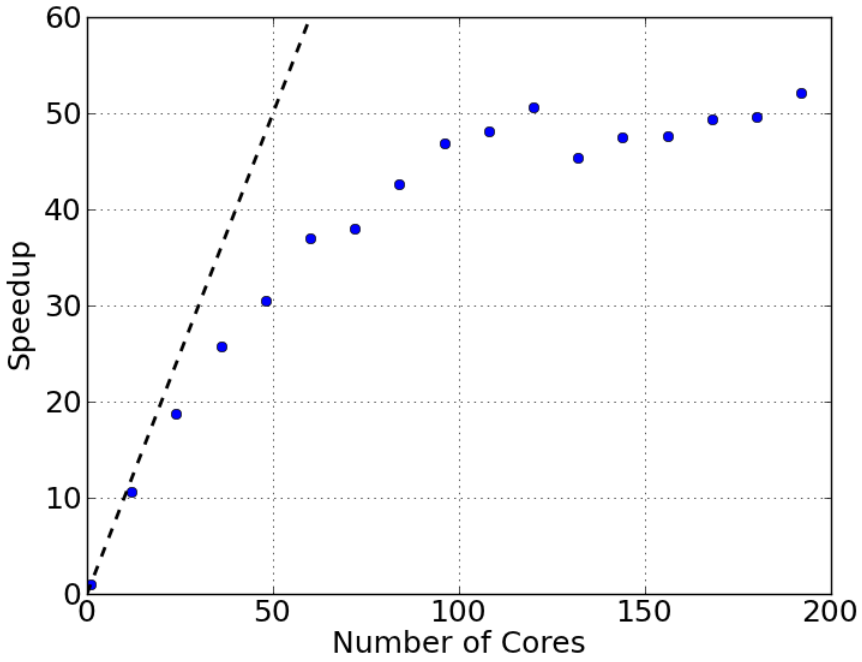
Built with f2py

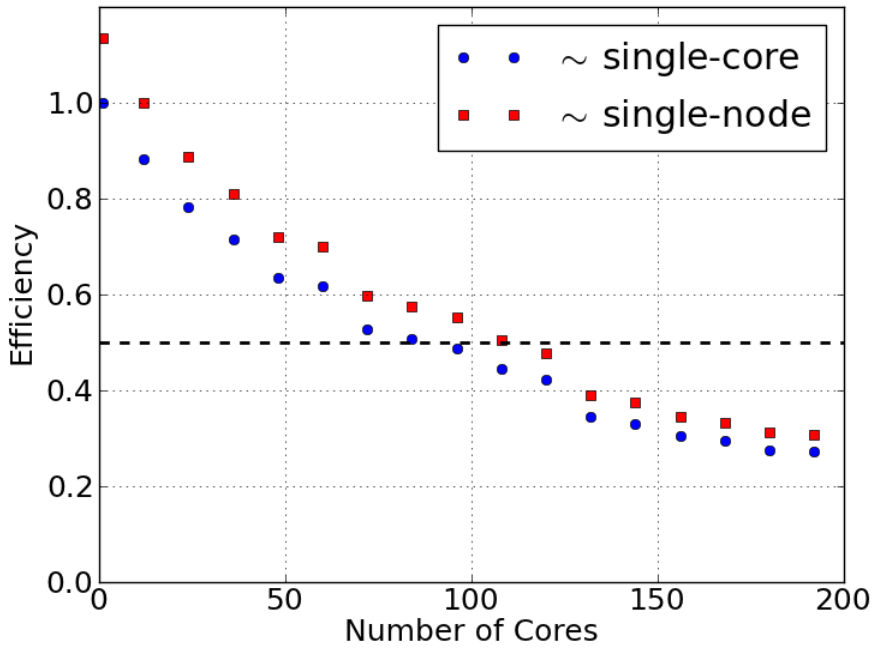


Profiling Results



Scaling of ECEPP/3 w/ MPI





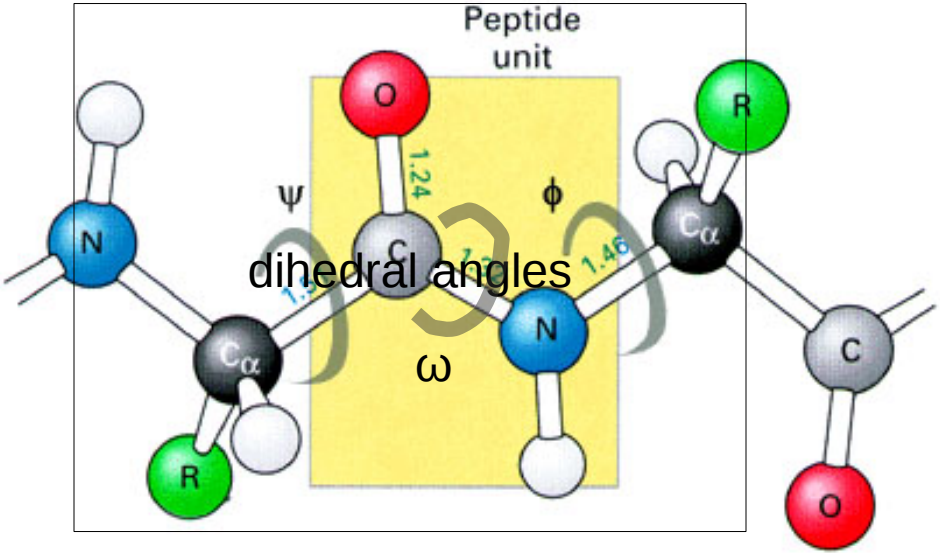
2.65 ms on a single node (Q80).

ECEPP/3

$$E = 320 \sum_{ij} \frac{q_i q_j}{\epsilon r_{ij}} + \sum_{ij} \left(\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}} \right) + \sum_{ij} \left(\frac{C_{ij}}{r_{ij}^{12}} - \frac{D_{ij}}{r_{ij}^{10}} \right) + \sum_{l} U_l (1 + \cos(n_l \xi_l))$$

ECEPP/3

$$E = 320 \sum_{ij} \frac{q_i q_j}{\epsilon r_{ij}} + \sum_{ij} \left(\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}} \right)$$
$$+ \sum_{ij} \left(\frac{C_{ij}}{r_{ij}^{12}} - \frac{D_{ij}}{r_{ij}^{10}} \right)$$
$$+ \sum_{l} U_l \left(1 + \cos\left(n_l \xi_l \right) \right)$$

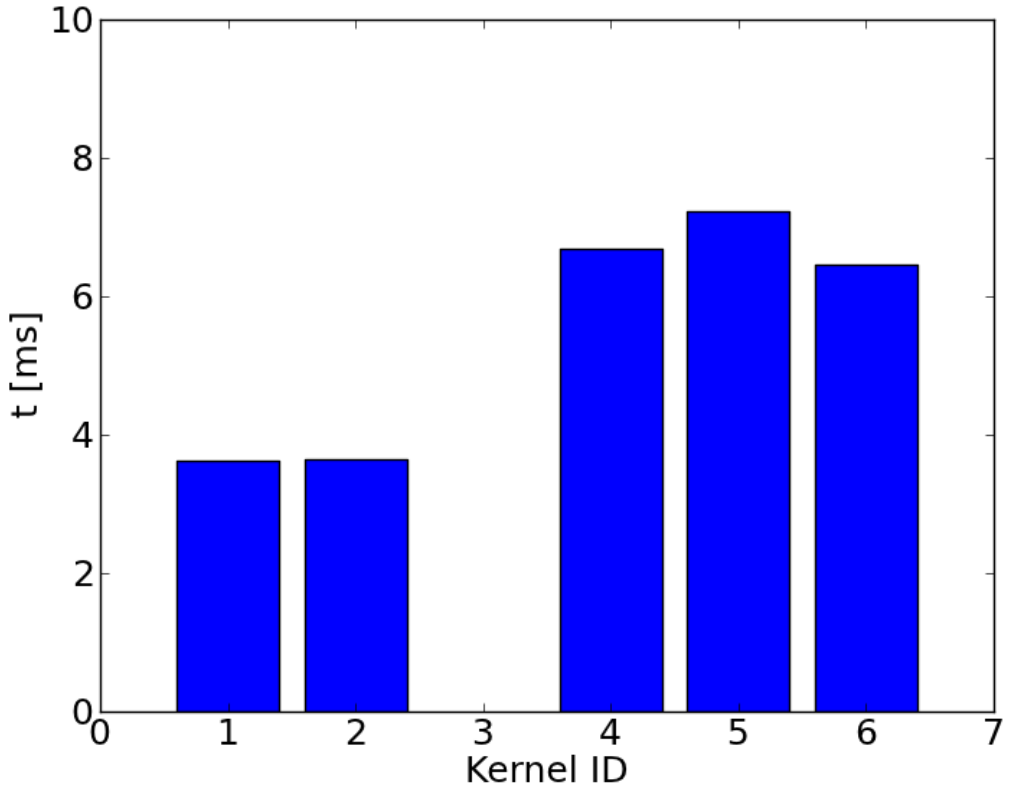


http://apple.sysbio.info/~mjhsieh/sstour/

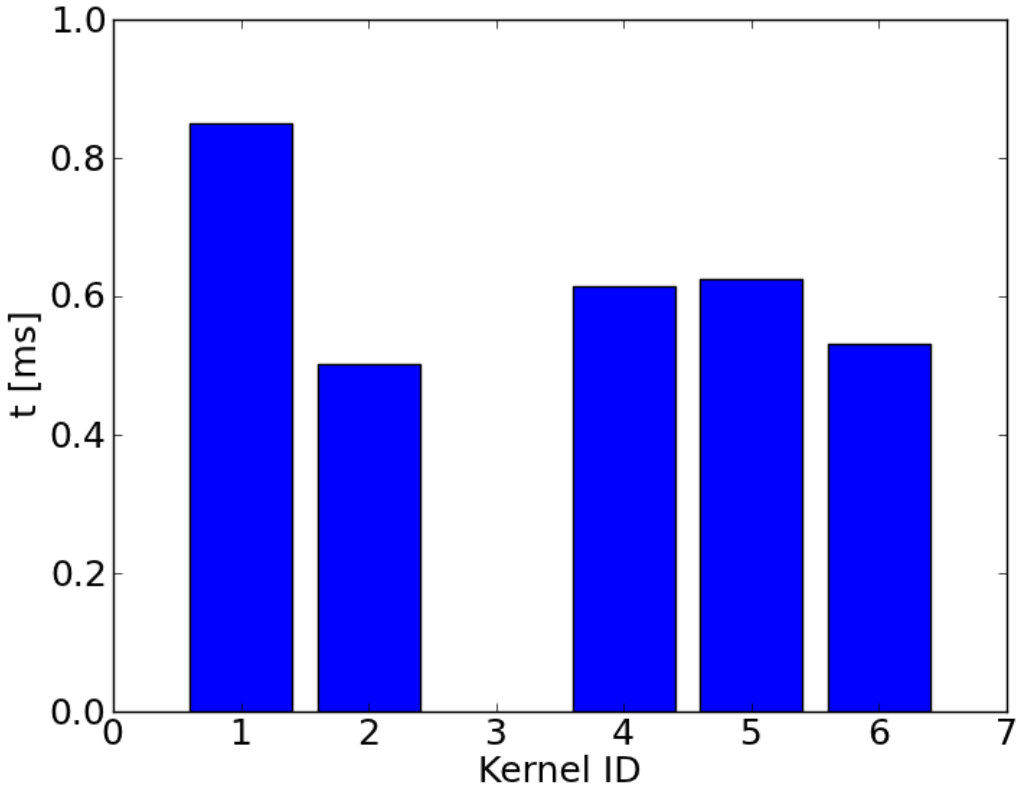
Implementation

- PyOpenCL
- Different kernels
  - Simple kernel
  - Use local memory
  - Use float4 for distance calculation
  - Calculate several interaction per work item
  - Unroll loop and vectorize
- Tested with Intel SDK, AMD SDK, NVIDIA SDK on CPUs and GPUs

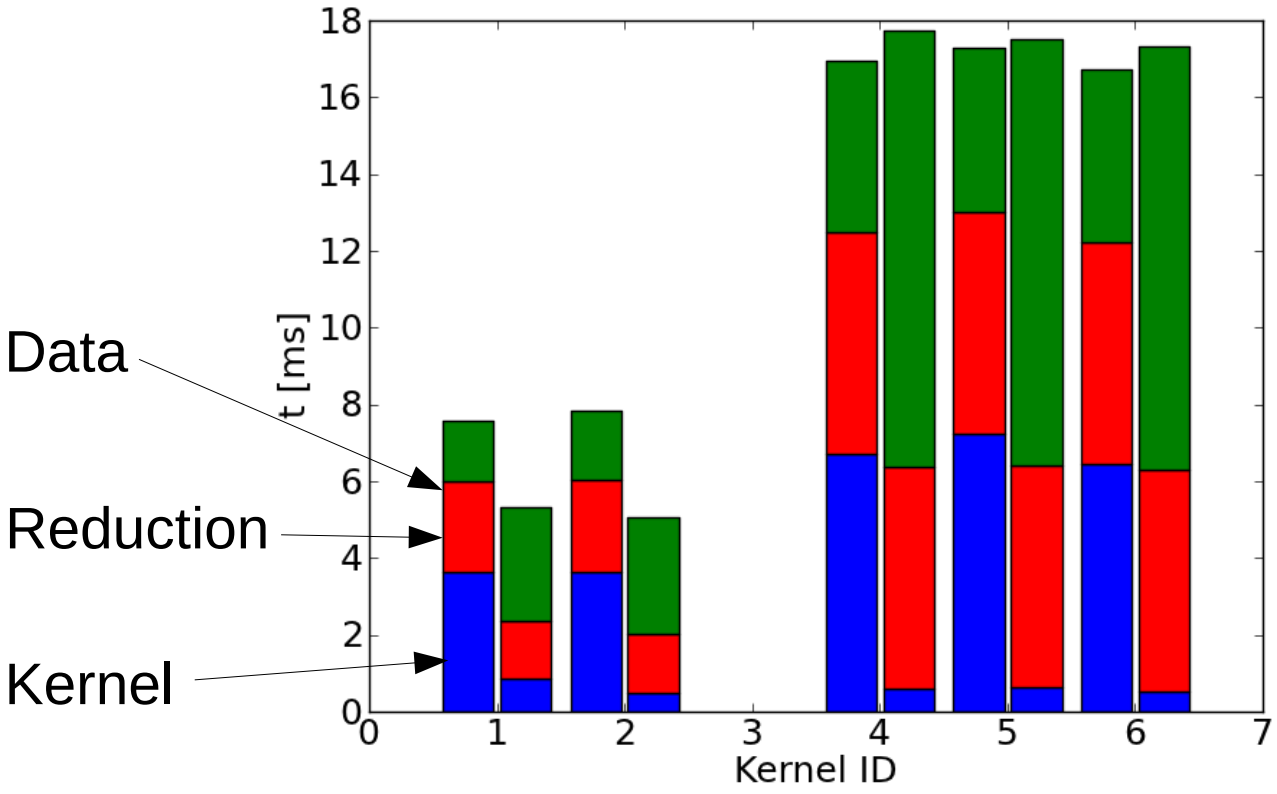
Performance of Quadratic Kernels (CPU)



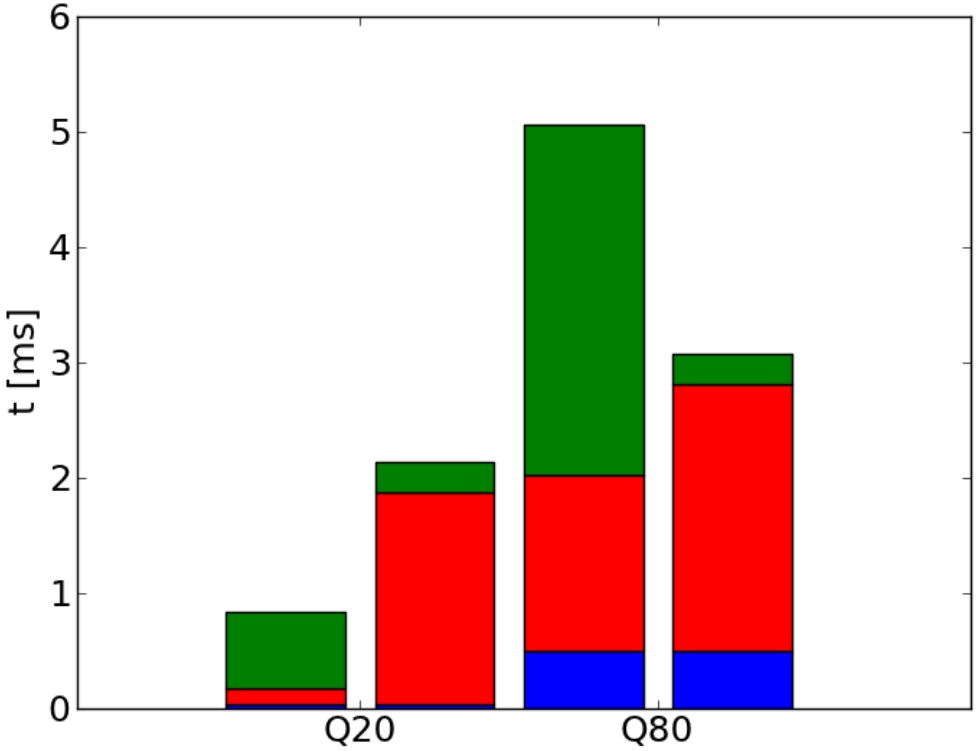
Performance of Quadratic Kernels (GPU)



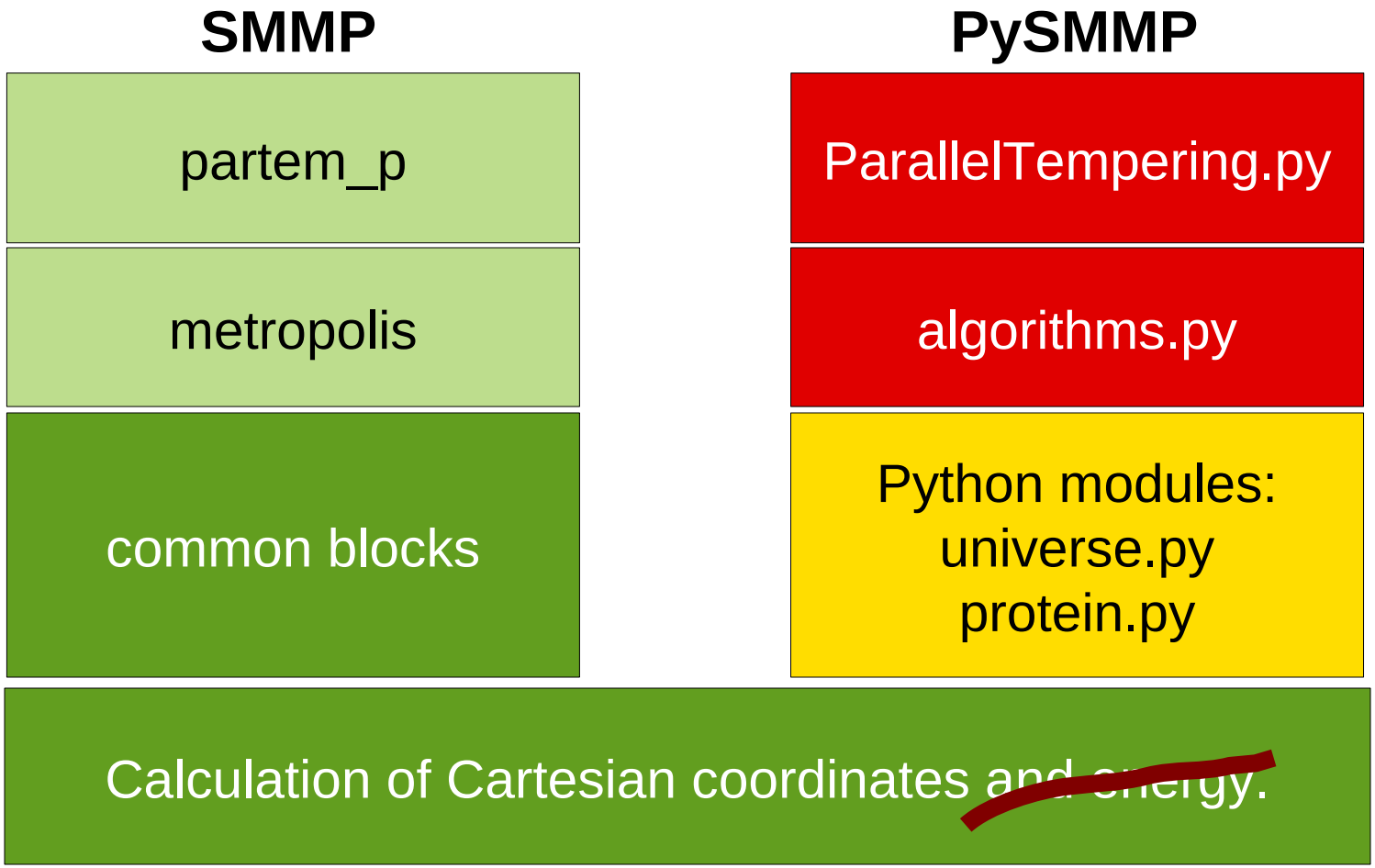
Where does the time go?



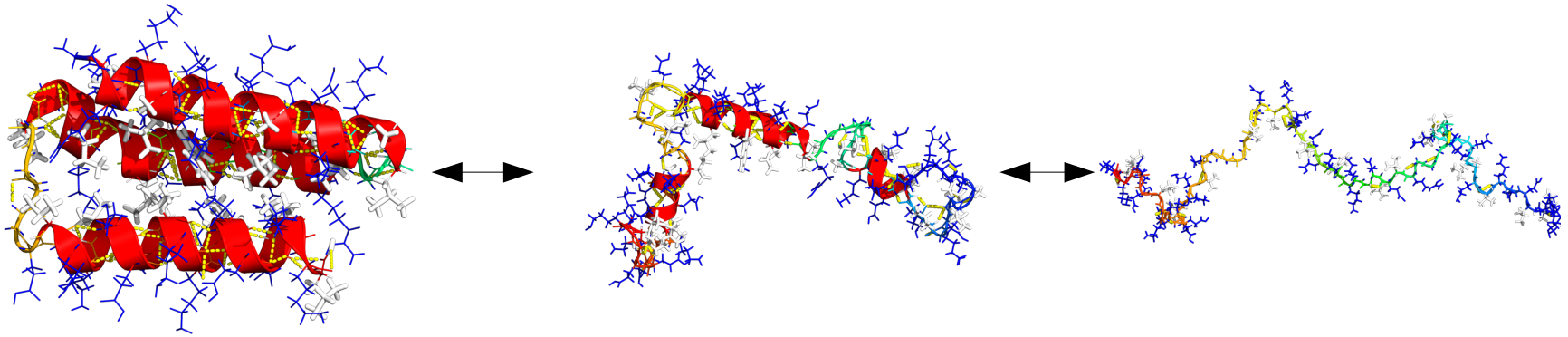
Reduction on CPU or GPU



Parallel Tempering with SMMP and PySMMP



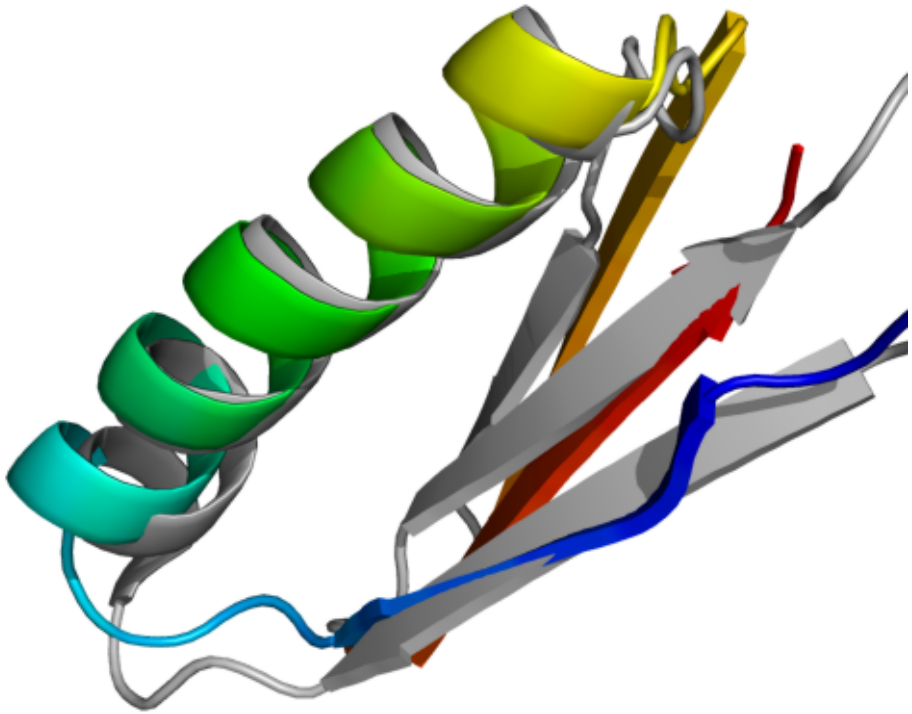
Parallel Tempering Monte Carlo



$$P_{PT} = e^{\Delta\beta\Delta E} \qquad P_{PT} = e^{\Delta\beta\Delta E}$$

$T_0 < T_1 < T_2$

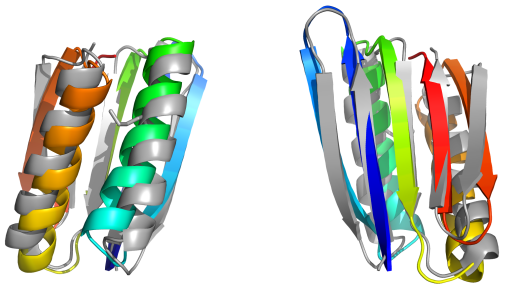
C-terminal Fragment of Top 7



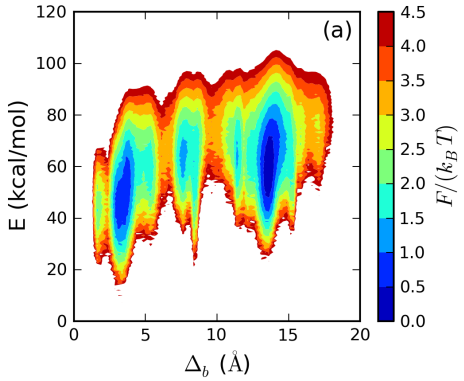
ProFASi: Protein Folding and Aggregation Simmulator

Sandipan Mohanty

C++

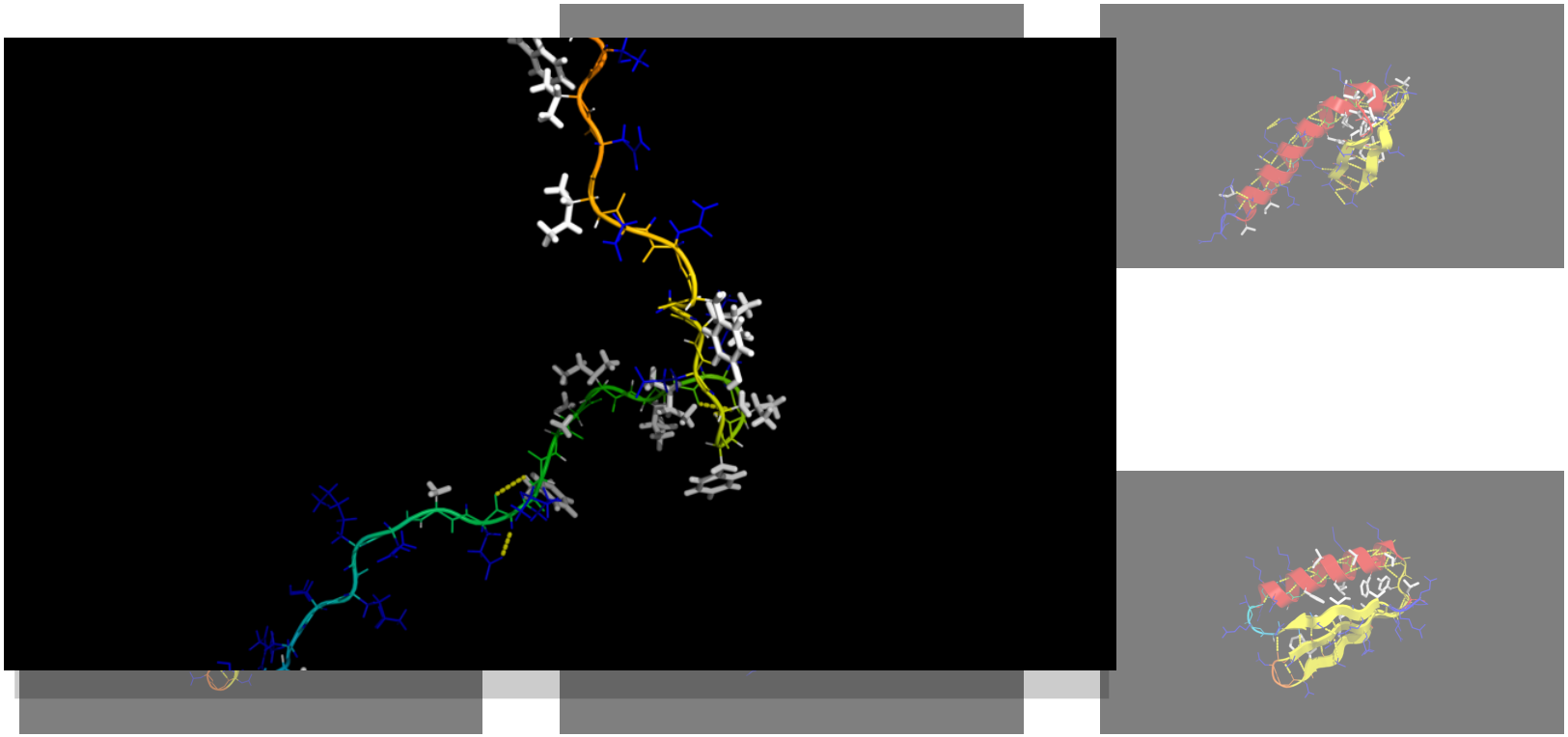


Monte Carlo

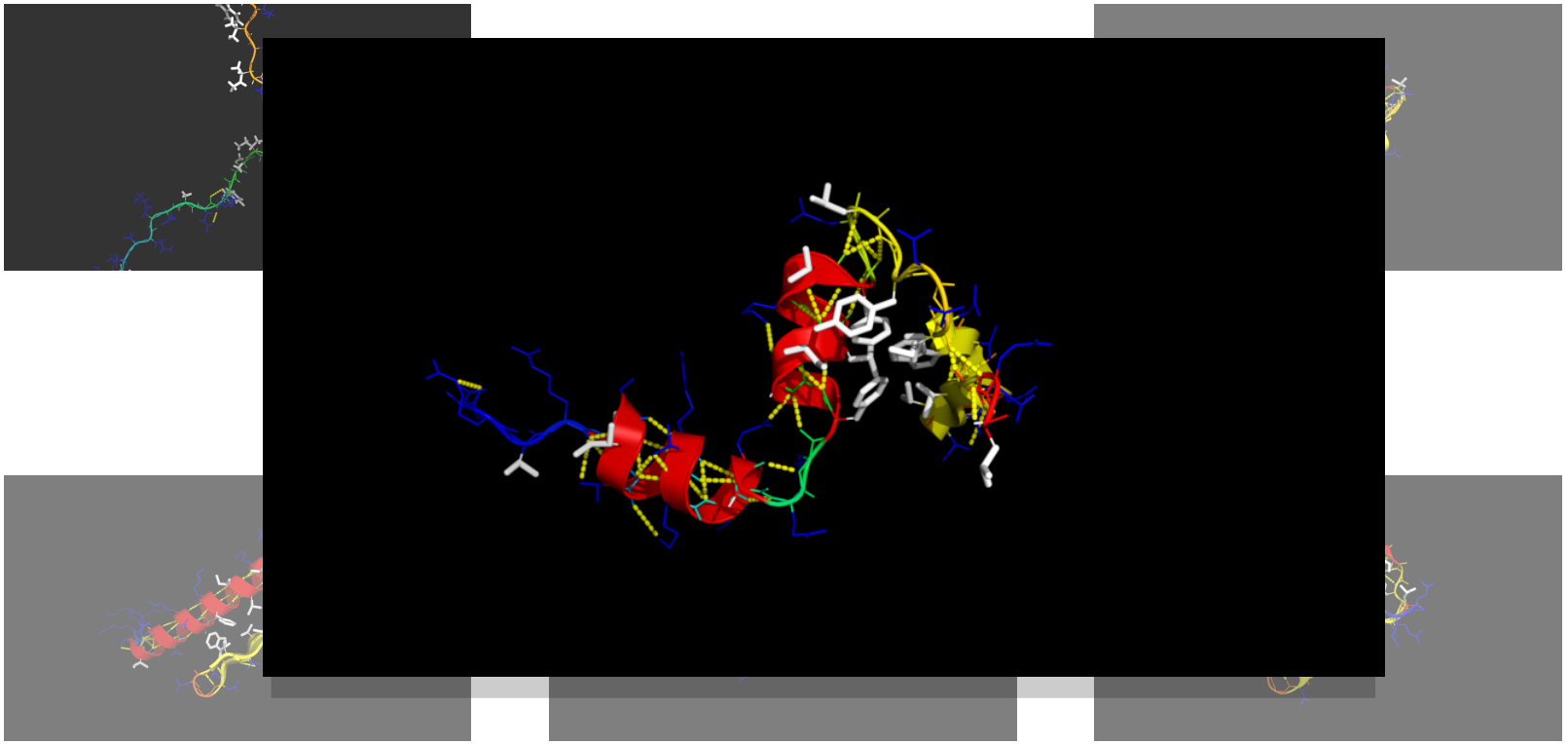


Highly Optimized

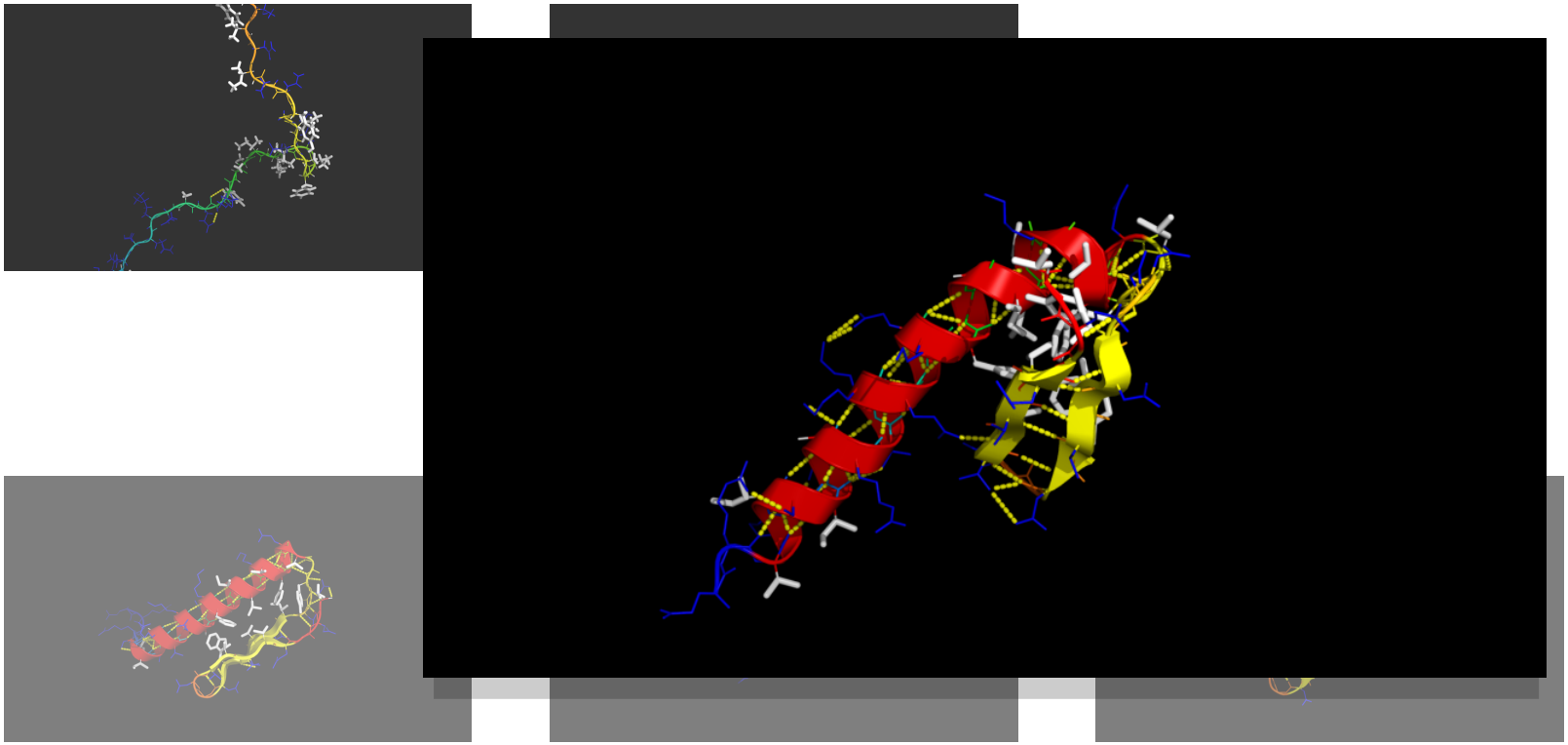
C-terminal Fragment of Top 7



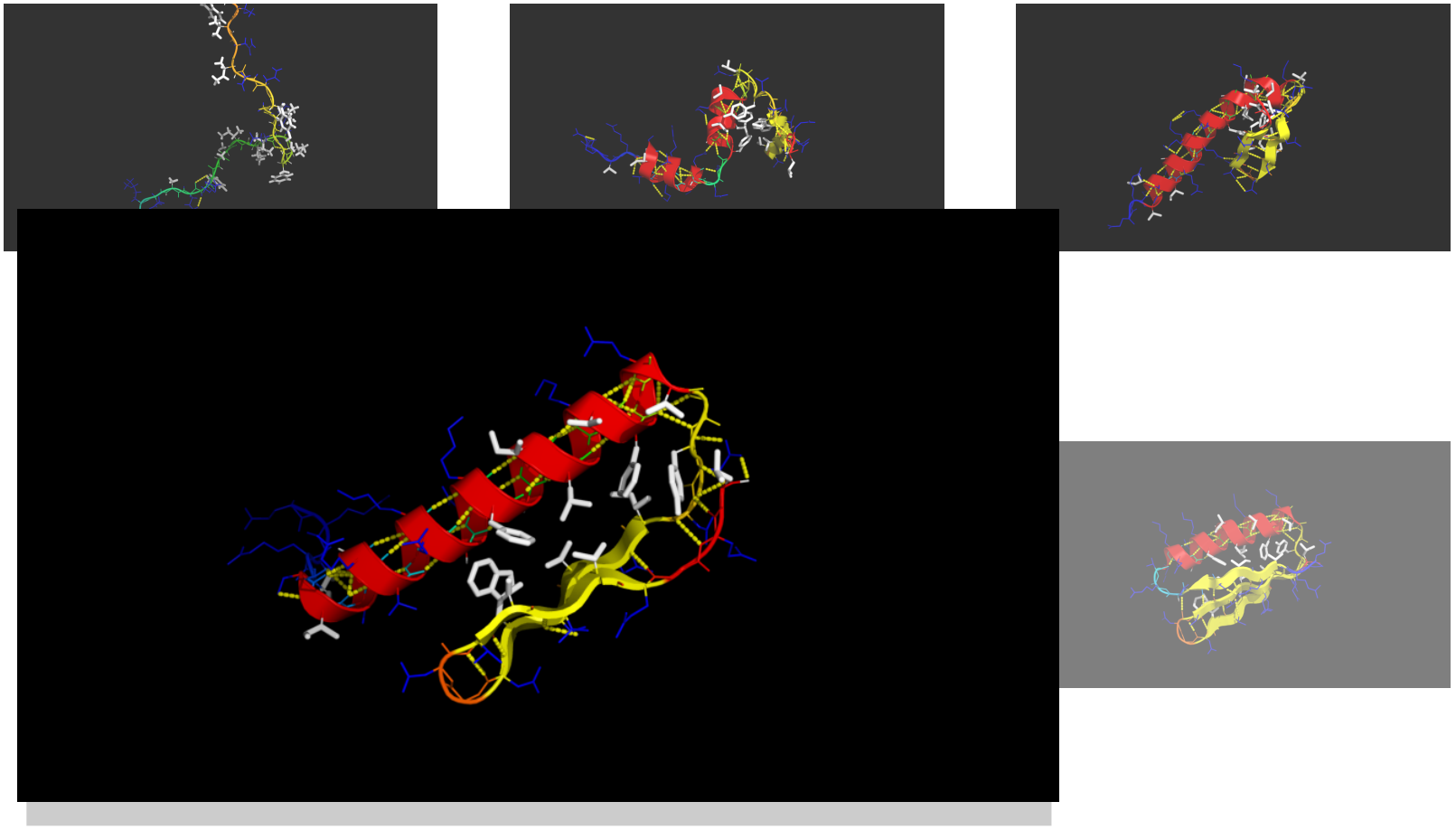
C-terminal Fragment of Top 7



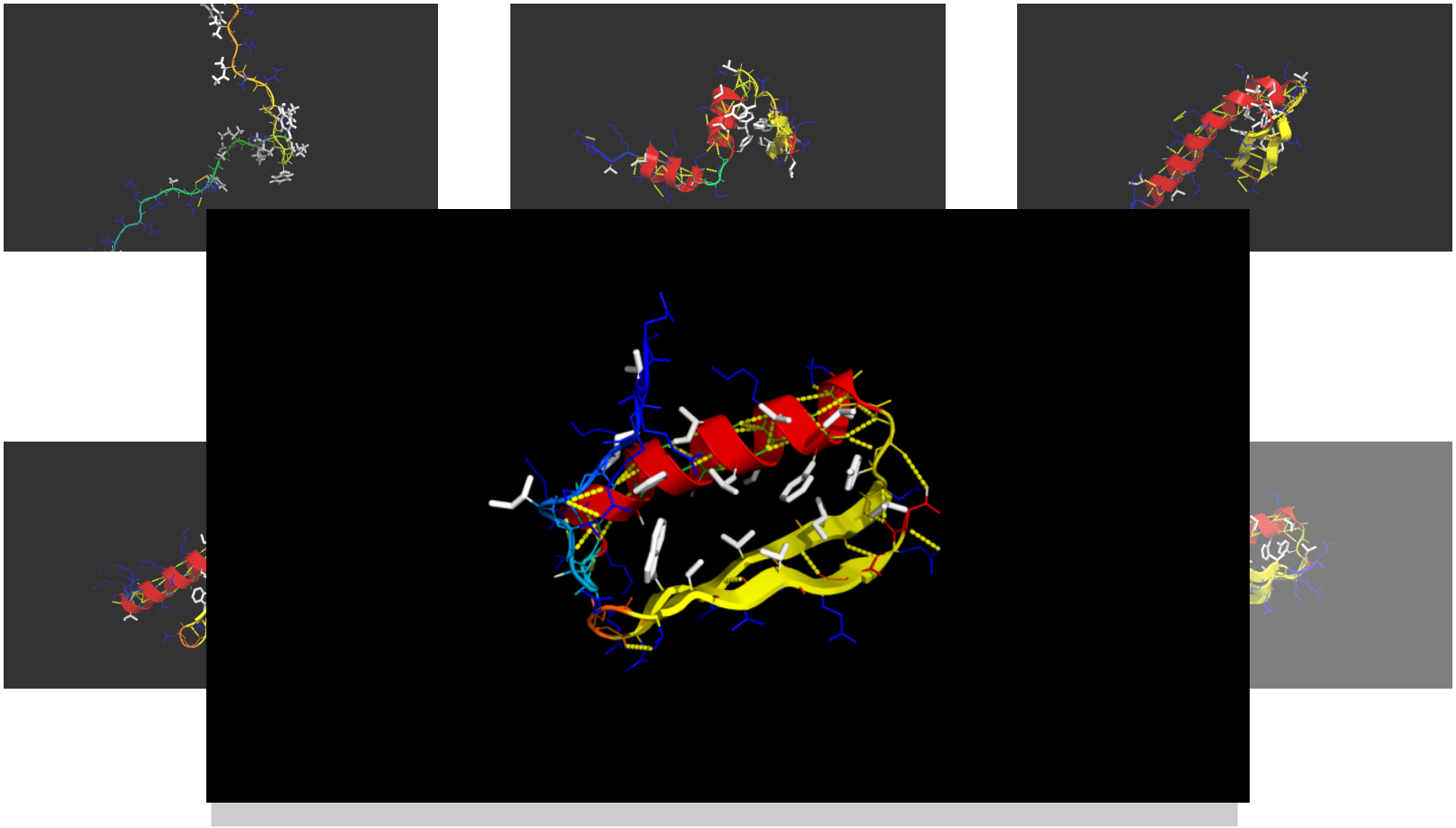
C-terminal Fragment of Top 7



C-terminal Fragment of Top 7



C-terminal Fragment of Top 7



C-terminal Fragment of Top 7