Hybrid MPI-OpenMP Molecular Dynamics Simulations

Chazirakis Antonios, Tsourtis Anastasios
University of Crete, Greece
Applied Mathematics department

University of Crete, Greece

May 29, 2015

Introduction

Who we are:

Group of the Applied Mathematics Department of the University of Crete in Greece.

Under the supervision of professor Vagelis Harmandaris.

What we do:

Mathematical and Computational Modelling of Complex Molecular Systems.

Soft matter and Mesoscale computer simulations,

Cooperation with experimentalists.

Mostly focus on Molecular Dynamics simulations and Monte Carlo simulations.

Introduction

- Proven software solutions exist for MD, like GROMACS, LAMMPS, NAMD, DL_POLY etc.
- Extending them for specialized research problems can be hard and problematic.
- We are developing an open source parallel MD package for our needs.
- We shall present what is MD and some specific problems and solutions.

Motivating example

$$\begin{cases} \text{system of} N = \sim 10^5 \text{ interacting particles} \\ \text{simulation time} \sim 1\mu s = 10^{-6}\text{sec} \\ \text{integration timestep}\, \delta t = 1fs \\ \text{solve}\, 3N\, \text{eq's per timestep} \end{cases}$$

Motivating example

$$\begin{cases} \text{system of} N =\sim 10^5 \text{ interacting particles} \\ \text{simulation time} \sim 1\mu s = 10^{-6}\text{sec} \\ \text{integration timestep}\, \delta t = 1fs \\ \text{solve}\, 3N \text{ eq's per timestep} \end{cases}$$

serial computation requires weeks!!

Motivating example

$$\begin{cases} \text{system of} N =\sim 10^5 \text{ interacting particles} \\ \text{simulation time} \sim 1\mu s = 10^{-6}\text{sec} \\ \text{integration timestep}\, \delta t = 1fs \\ \text{solve}\, 3N \,\text{eq's per timestep} \end{cases}$$

serial computation requires weeks!!
solution $\rightarrow$ Parallel processing

Motivating example - PMF

Not all problems have to deal with large systems or long-time intergration.

One of the sumulation issues we are trying to tackle at the moment is the *Potential of Mean Force* in order to construct a *Coarse-Grained three-body potential*.

Motivating example - PMF

Not all problems have to deal with large systems or long-time intergration.

One of the sumulation issues we are trying to tackle at the moment is the *Potential of Mean Force* in order to construct a *Coarse-Grained three-body potential*. $\Rightarrow$ **sampling of the available phase-space**

Motivating example - PMF

Not all problems have to deal with large systems or long-time intergration.

One of the sumulation issues we are trying to tackle at the moment is the *Potential of Mean Force*
in order to construct a *Coarse-Grained three-body potential*. $\Rightarrow$
sampling of the available phase-space
massive "embarassingly parallel" serial computations.

Motivating example - PMF

Not all problems have to deal with large systems or long-time intergration.

One of the sumulation issues we are trying to tackle at the moment is the *Potential of Mean Force*
in order to construct a *Coarse-Grained three-body potential.* $\Rightarrow$
sampling of the available phase-space
massive "embarassingly parallel" serial computations.

solution $\rightarrow$ Parallel processing

Motivating example - PMF

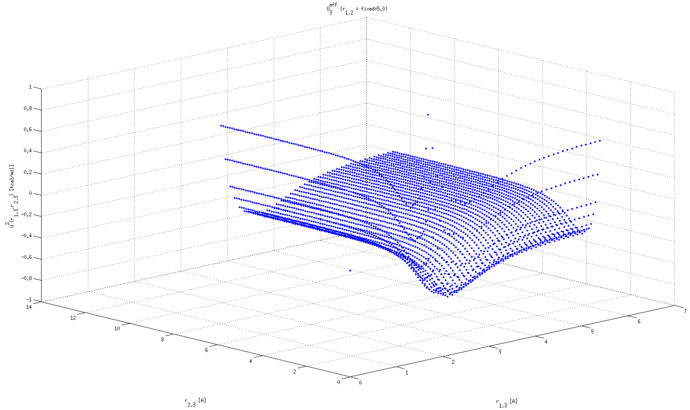


Figure: 3-body effective potential of CH_4

Molecular Dynamics - Simulated System

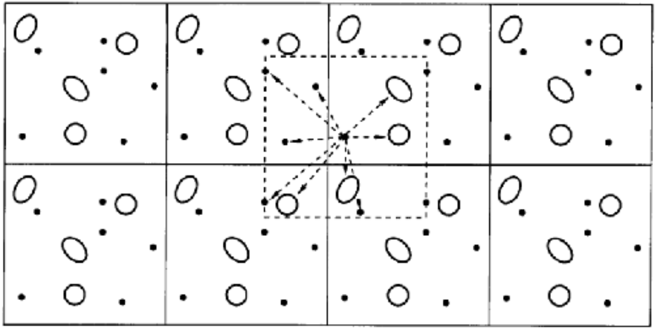
- Few particles are simulated ($\propto 10^5$).
- $\mathbf{X}(t) = \{\vec{r_1}(t), \vec{r_2}(t), ..., \vec{r_N}(t)\}$
- $M\ddot{\mathbf{X}}(t) = \mathbf{F}(\mathbf{X}(t))$
- Initial Value Problem, Velocity Verlet numerical integration.

$$\mathbf{X}(t+dt) = \mathbf{X}(t) + \mathbf{V}(t)dt + \frac{1}{2}M^{-1}\mathbf{F}(t)dt^2$$
$$\mathbf{V}(t+dt) = \mathbf{V}(t) + M^{-1}\frac{\mathbf{F}(t) + \mathbf{F}(t+dt)}{2}dt$$

- Statistics are done by sampling the integrated trajectories.

Molecular Dynamics - Bulk System

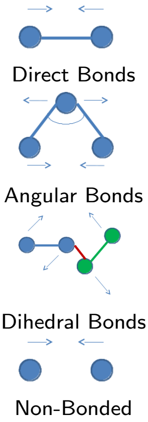
- The simulation volume is infinitely replicated to emulate the bulk system ($\propto 10^{23 particles}$).



- Pair distance = Shortest of replicated distances. *(Periodic boundary conditions)*
  - $x = x - roundInf(\frac{x}{L}) \times L$
  - $y = y - roundInf(\frac{y}{L}) \times L$
  - $z = z - roundInf(\frac{z}{L}) \times L$

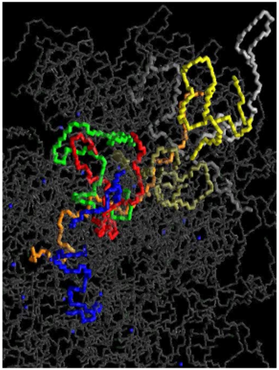
Molecular Dynamics - Forces

- **More than 95% of CPU time is spent calculating forces.**
- Bonded interactions model chemical bonds.
  - Direct: $U = \frac{k}{2}(r - r_0)^2$
  - Angular: $U = \frac{k}{2}(\theta - \theta_0)^2$
  - Dihedral: $U = \sum_n c_n cos^n(\phi)$
- Non bonded interactions model van der Waals induced dipole interactions.
  $U = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^6 \right]$
- Electrostatic interactions model the electronegativity of atoms.
  $U = \frac{q_a q_b}{r}$



Bonded Interactions (in a Small Polyethylene System)

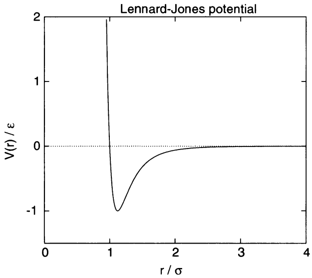
- United Atom Model, only CH_2 and CH_3 united atoms, **no electrostatic charges.**
- 200 chains, 16 atoms each, 3200 atoms in total.
- We need 3000 direct bonds, 2800 angular bonds and 2600 dihedral bonds to fully model the bond forces.
- Each atom carries a list of bonds of each type and interacts with the same atoms throughout the simulation (there is no chemistry).
- Relatively easy, static data dependencies.
- Relatively fast, $N_{bonds} \propto N_{atoms}$, roughly 25% of the simulation time.



Polyethylene Example System

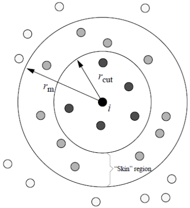
Non-Bonded Interactions

- Pairwise.
- Short range.
- There is a natural cut off distance after which the forces become negligible.
- Roughly 75% of CPU time.
- *(Electrostatic forces also inhibit a short range part through Ewald Summation)*

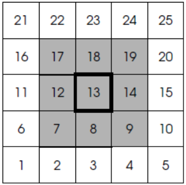


Lennard Jones Potential

- To reduce complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(Nlog(N))$, Verlet Lists are used.
- Potential short range interactions are first identified in per atom lists ($\approx \frac{100}{atom} \approx 300K\ total$).
- Some margin is left for imminent integration displacements.
- The lists remain valid for roughly 15 integration steps, until any atom displacement exceeds the margin set. Afterwards we must rebuild them.
- Linked Cell Lists are used as a spatial index to speed up Verlet List calculations. Nearby atoms reside in nearby boxes.



Verlet Lists



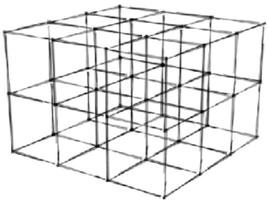
Linked Cell Lists

Simulation Cycle

1. Calculate bond forces.
2. Calculate non bonded forces.
   - Rebuild pairwise interaction lists if invalidated.
3. Integrate using Velocity Verlet *(almost negligible CPU time)*.
4. Every some thousands of time steps.
   - Calculate statistical quantities, energies, temperature, pressure.
   - Write trajectory to file.
5. Repeat for a few millions of time steps.

Hybrid MPI-OpenMP-(GPU?) Parallelism - Domain Decomposition

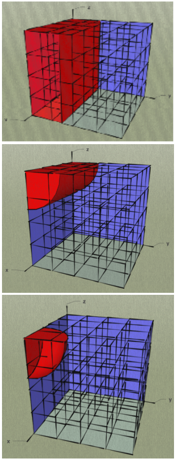
- Spatial locality naturally translates to computational locality.
- The simulations volume is split into subvolumes.
- Each MPI Rank computes interactions for atoms in it's subvolume.
- Each MPI Rank might use many OpenMP threads in multicores.
- (todo): each MPI Rank might use an accelerator (GPU)



Domain Subvolumes

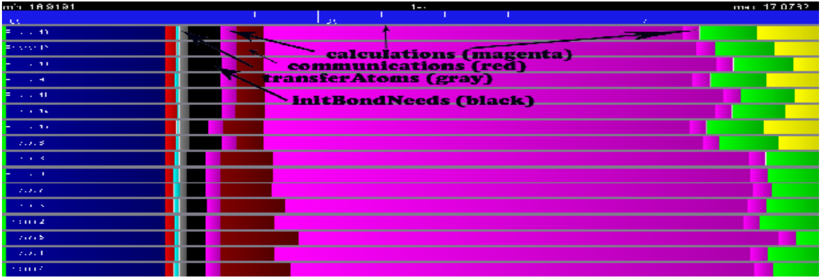
Hybrid MPI-OpenMP Parallelism - Domain Decomposition cont.

- Each rank needs to get some positions from nearby ranks to calculate forces.
- It also needs to send back some reaction forces (law of action and reaction).
- Statistics and output are global communication operations.



Subvolume Interaction Areas

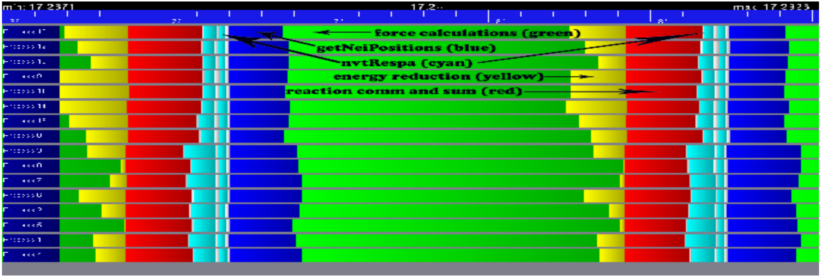
Tracing - VampirTrace - Verlet Lists



- 1st Magenta block: Locating near boundary atoms.
- Red block: Sending and receiving near boundary atoms.
- 2nd Magenta block: Creating Verlet Lists.
- Black block: Transfering atoms to appropriate mpi ranks.

Based on older data. Communication times became more significant as optimizations improved calculation times.

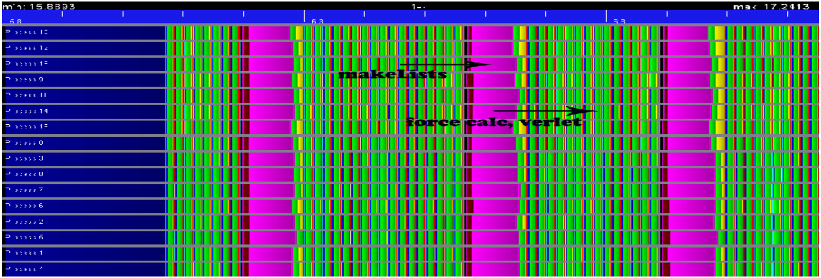
Tracing - VampirTrace - Forces



- Blue block: Receiving positions.
- Green block: Calculating forces.
- Yellow block: Energy reduction.
- Red block: Sending reactions.
- White and light blue blocks: Integration

Based on older data. Communication times became more significant as optimizations improved calculation times.

Tracing - VampirTrace - Overview



- Magenta blocks: Verlet Lists.
- Other blocks: Force caclulation and integration.

Based on older data. Communication times became more significant as optimizations improved calculation times.

Development Issues

- Extensibility
  - New force functional forms.
  - New integration schemes.
  - New statistical quantities.
  - Tracing and performance timers.
- Efficiency
  - Optimizations should be in the core.
  - Extensions should benefit from them.
- Maintainability
  - Trivial extensions should be easy.
  - Trivial extensions should not require tampering with the core code.
  - Similar code paths in optimization should be implemented avoiding replication (copy-paste).

Serial optimization first. *Focus on short range pair forces.* A tiny piece of code that becomes large and dominates, $\approx$ 75% of CPU time.

- Trivial: do not repeat calculations, choose lightweight operations, inline code to avoid function calls, Minimize cache misses (not so trivial).
- Problem: cannot use virtual methods, they will not inline, plus vtable lookup becomes comparable to small method execution time.
- Interesting bonus: minimize branch misses.

Efficiency - Simple Stuff

Inline code to avoid function calls.

```
inline double Vector3D::dotProduct(const Vector3D & other) const {
  return x*other.x + y*other.y + z*other.z;
}
```

Do not repeat calculations, choose lightweight operations.

```
// Slow
void LJForce::calcEnergy(const unsigned &pairType, const double &rsq,
                         double *energySum) {
  *energySum += m_a[pairType]/pow(rsq,3) - m_b[pairType]/pow(rsq,6);
}
// Fast
inline void LJForce::calcEnergy(const unsigned &pairType, const double &rsq,
                         double *energySum) {
  double r6i, r12i;
  r6i = 1.0/(rsq*rsq*rsq);
  r12i = r6i*r6i;
  *energySum += m_a[pairType] * r12i - m_b[pairType] * r6i;
}
```

Speedup $\approx$ 30 *times*

(Not due to inlining, inlining is difficult to measure but still matters for small methods.)

Efficiency - Serious Issue - Virtual Methods

Small virtual methods that dominate cpu time are inefficient.

```
// Bad, virtual method call for each atom pair, 300K calls total
class ShortPairForce {
  virtual void calc(const double &rsq, Vector3D &f, double *energySum) {
    // very few operations
  }
};
// Good, a single virtual method call for all atom pairs
class ShortPairForce {
  virtual void calc(const TypedAtomPairGroups *pairGroups, double *energy) {
   // much to do
  }
};
```

Solution: static polymorphism, compile time, no vtables, allows inlining.

```
// Base
template <class DerivedForce>
class ShortPairForceImpl : public ShortPairForce {
  virtual void calc(...) { for (...) calcForce(...); }
  inline void calcForce(const unsigned &pairType, const double &rsq, double *e);
}

template <class DerivedForce>
inline void ShortPairForceImpl<DerivedForce>::calcForce(const unsigned &pairType,
                                       const double &rsq, double *energySum) {
  // Each derived defines doCalcForce, also as an inline
  static_cast<DerivedForce*>(this)->doCalcForce(pairType, rsq, energySum);
}
```

Efficiency - Serious Issue - Virtual Methods cont.

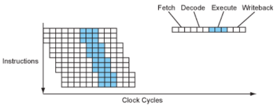
Derived class implementation.

```
// Derived
class LJForceImpl : public ShortPairForceImpl<LJForceImpl> {
  inline void doCalcForce(const unsigned &pairType, const double &rsq,
                                                     double *energySum) {
    double r6i, r12i;
    r6i = 1.0/(rsq*rsq*rsq);
    r12i = r6i*r6i;
    *energySum += m_a[pairType] * r12i - m_b[pairType] * r6i;
  }
};
```

Speedup $\approx 15\%$

Efficiency - Minimizing Branch Misses

- Modern CPUs might handle say $14 \times 8 = 112$ instructions at the same time (an exaggeration).
    - A CPU's pipeline comprises of the different stages of a single instruction **(Fetch, Decode, Execute, Write back)**.
    - The execute stage comprises of different functional units. Multiple independent pipelines for each functional unit. **(Integer, Float, Memory instructions)**.
- CPU predicts branches to keep the pipelines loaded.
- Eventually branches are evaluated and if they have been mispredicted part of the pipelines must be trashed as invalid, ≈ 50 cycles lost.



Instruction Flow

Depth	Processors
12	Pentium Pro/II/III
14/19	Core i*2/i*3 Sandy/Ivy
14/19	Core i*4/i*5 Haswell
14/19	Core i*4/i*5 Broadwell
20	Pentium 4
31	Pentium 4E Prescott

Pipeline Depths

Width	Processors
3	Pentium Pro/II/III/M
3	Pentium 4
6	Athlon,Athlon 64/Phenom
6	Core 2, Core i*1 Nehalem
6	Core i*2/i*3 Sandy/Ivy
8	Core i*4/i*5 Haswell
8	Core i*4/i*5 Broadwell

Issue Widths

Efficiency - Minimizing Branch Misses cont.

- We try to minimize branch misses.
- Periodic Boundary Conditions:

```
// Two branches * 3 dimensions = 6 branches
if (position.x > m_halfSize.x) position.x -= m_size.x;
if (position.x < m_negHalfSize.x) position.x += m_size.x;
```

- Verlet List Skin:

```
for (rootIndex=0; rootIndex < rootCount; ++rootIndex) {
  while (neiIndex < neiMax) {
    r.setToDiff(*rootPos, positions[nei]);
    // 6 branches hidden below from PBC's
    mca.applyMinImgConv(r);
    rsq = r.lengthSq();

    // Another branch here from Verlet List Skin
    if (rsq < this->m_rcsq[pairType]) {
      m_force->calcForce(pairType, r, rsq, m_f, m_energySum);
      rootForce->add(m_f);
      neiForce.subtract(m_f);
    }
  }
}
```

- 7 branches total in the 300K atom pairs, per integration step, frequent misses cost.

Efficiency - Minimizing Branch Misses cont. 2

- Atom pair neighbour lists are recreated every 15 or so steps.
- Initial pair distances can only vary so much.
- We can predict beforehand the PBC correction of each pair.
  - At each dimension: X, Y, Z
  - No correction needed (most pairs): $\|D_{pair} \pm D_{skin}\| < \frac{L_{box}}{2}$
  - Plus correction: $D_{pair} + D_{skin} < -\frac{L_{box}}{2}$
  - Minus correction: $D_{pair} - D_{skin} > \frac{L_{box}}{2}$
  - Not predictable, check: $\|D_{pair}\| \approx \frac{L_{box}}{2}$
- The Liked Cell Lists spatial index is used to speed up predictions.
- We also have a hint on proximity:
  - Near: $D_{pair} < r_{cutoff}$
  - Not predictable, check: otherwise
- There are $4^3 \times 2 = 64 \times 2 = 128$ cases.

Efficiency - Minimizing Branch Misses cont. 3

- Optimizing for branch misses competes optimizing for cache misses.
- A head atom plus neighbour atoms list is broken into many, the head atom coordinates do not stay in cache or registers as much as they should.
- Also: More types of lists = More list maintenance overhead.
- We only keep the dominating cases for corrections and proximity:
  - Corrections: Not Needed,Unpredicted,X_{plus}, X_{minus}, Y_{plus}, Y_{minus}, Z_{plus}, Z_{minus}
  - Proximity: Initially Close ($D_0 <= r_{cutoff}$) , Initially Far
  - Only 16 cases kept (out of 128).
- Roughly 20% speedup on large systems.

- Optimizing of efficiency produces similar code paths with minor differences. Replicated (copied and pasted) code is very hard to debug and maintain. Implementation by using templates preferred.
- The generic force calculation loop depends on:
  - **MinImgConvApplier**: periodic boundary corrections.
  - **SinglePairCalculator, Force**: derived force analytical form.

```
template <class Force, class MinImgConvApplier, class SinglePairCalculator>
class ShortPairForceLooper {
public:
  void loop(ShortPairForceImpl<Force> *force, const TypedAtomPairs &pairs) {
    MinImgConvApplier mca(force->domain());
    SinglePairCalculator spc(force, force->squaredCutoffs(), energySum,
                                                              virialSum);
    for (rootIndex=0; rootIndex < rootCount; ++rootIndex) {
      while (neiIndex < neiMax) {
        r.setToDiff(*rootPos, positions[nei]);
        mca.applyMinImgConv(r);
        rsq = r.lengthSq();
        spc.calcPair(type, r, rsq, rootForce, forces[nei]);
      }
    }
  }
};
```

Maintainability & Efficiency - Using Templates cont.

- Now similar code paths do not have to be replicated.

```
switch (pairs.periodx()) {
  case pkPlusPeriod: {
    ShortPairForceLooper<DerivedForce, XPlusMinImgConvApplier,
        GenPairCalculator<ShortPairForceImpl<DerivedForce> > > looper;
    looper.loop(this, pairs); return;
  }
  case pkMinusPeriod: {
    ShortPairForceLooper<DerivedForce, XMinusMinImgConvApplier,
        GenPairCalculator<ShortPairForceImpl<DerivedForce> > > looper;
    looper.loop(this, pairs); return;
  }
  case pkCheckPeriod: {
    ShortPairForceLooper<DerivedForce, XCheckMinImgConvApplier,
        GenPairCalculator<ShortPairForceImpl<DerivedForce> > > looper;
    looper.loop(this, pairs); return;
  }
}
```

- Handle both static polymorphism and similar code paths by using templates:
  - Efficiency: compile time inline code expansion, no function calls, no virtuals, avoidance of those at a few levels of depth.
  - Maintainability: you can debug and update heavily optimized core code at single points, no need to worry about 16 (times force kinds of course) occurrences of the same logic.

Extensibility & Maintainability - Class Factories

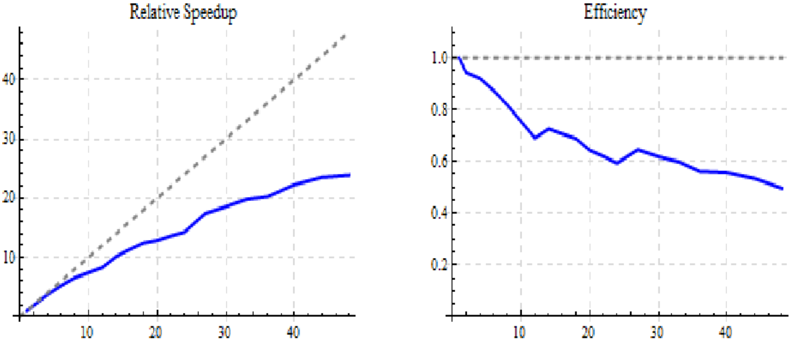
- Integrity: extending the simulation should leave the core code unaltered.
- The core should somehow become aware of new force classes.
- Class factories: A map, kept at the core, from class names (strings) to class instantiation functions (function pointers).
- Generic implementation in **VirtualMpiTrasferable** from which all force parameters are derived.
- Student and researchers than implement new forces register them in their own cpp files without altering the core.
- Each force parameter can be transferred from rank to rank and can instantiate the corresponding virtual class that implements the force.

Efficiency - Parallel Optimizations

- The basics:
  - MPI datatypes to send and receive hundreds of coordinates in a single message.
  - Asynchronous communications to simultaneously send to and receive from multiple ranks.
  - Care taken to avoid OpenMP false memory sharing.
- Potential improvement, measured, todo:
  - Roughly 60% of atom pairs are fully contained within a single rank.
  - While calculating 60% of the forces the coordinates needed for the remaining 40% should have largely arrived.
  - Overlap communication with computation to drastically reduce communication costs.

Efficiency - Results

- Acceptable speedup so far, more work to do.
- 27 cpus - 0.65 efficiency - 948 atoms/cpu (ok)
- 48 cpus - 0.50 efficiency - 533 atoms/cpu (few atoms, expected)



Future - GPUs

- Can we accelerate the force calculations only?
  - Redesign data access to favor coalescence.
  - Transferring coordinates in an efficient manner. (use of streams, zero copy memory, cudaMemcpyAsync)
- Can we find some way embedding CUDA code in our C++ class hierarchy?
  - Embed CUDA backbone in the core MD code
  - Require the definition of analytical force definition in a small __device__ function

Future - SIMD - Manycores

- SIMD
  - Also support single precision
  - Alter core data structures
  - Potential 4x speedup
- What about manycores like the Intel Phi?
  - Can we get away with OpenMP and SIMD ?

Conclusions

- Challenging problem.
- Manageable with acceptable performance, scalability and maintainability.
- First version will be out on September.
- Intended for academic use, expected to mature in a few years.

Acknowledgements

This research has been co-financed by the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program “Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF) – Research Funding Programs: **THALES**, Aristia

EUROPEAN UNION
EUROPEAN
SOCIAL FUND