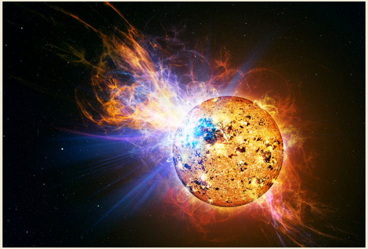
Particle-in-cell Code with LRnLA Algorithms Performance Tests on KNL

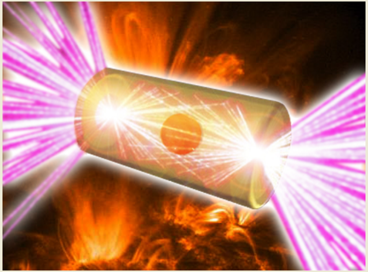
Anastasia Perepelkina,
Vadim Levchenko

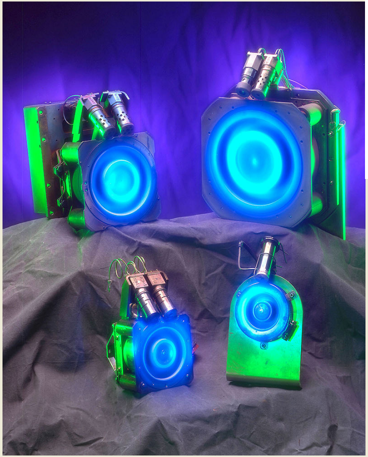
Keldysh Institute of Applied Mathematics, Moscow, Russia

May 11, 2017 mc2series.com
Colfax Research MC^2 Series: Modern Code Contributed Talks

Computer simulation of plasma







- Design of plasma devices
- Fundamental understanding of plasma phenomena

Vlasov Equation

$$\frac{\partial f_\alpha}{\partial t} + \vec{v}\frac{\partial f_\alpha}{\partial \vec{r}_\alpha} + e_\alpha\left(\vec{v} \times \vec{B} + \vec{E}\right)\frac{\partial f_\alpha}{\partial \vec{p}} = 0$$

Maxwell equations

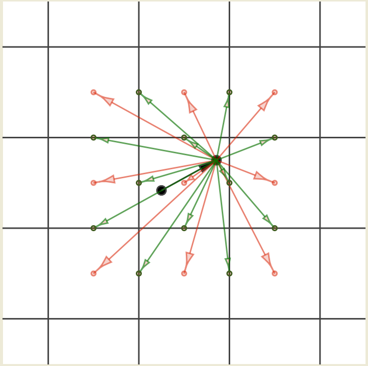
$$\frac{\partial \vec{B}}{\partial t} = -\nabla \times \vec{E}, \quad \frac{\partial \vec{E}}{\partial t} = \nabla \times \vec{B} - \vec{j}, \qquad \nabla \vec{B} = 0, \quad \nabla \vec{E} = \rho.$$

Charge and current densities

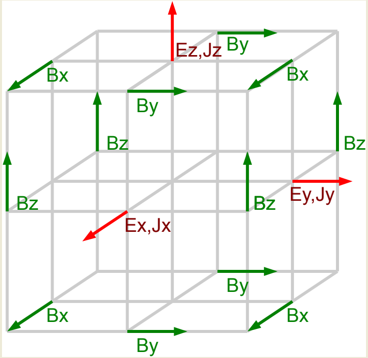
$$\rho = \sum_\alpha \int f_\alpha e_\alpha d\vec{p}, \qquad \vec{j} = \sum_\alpha \int \vec{v}_\alpha f_\alpha e_\alpha d\vec{p}.$$

Numerical methods

Particle-in-cell

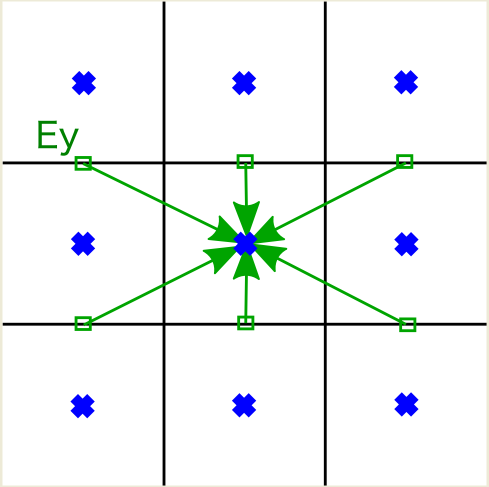


Finite difference on a staggered grid (Yee cell)



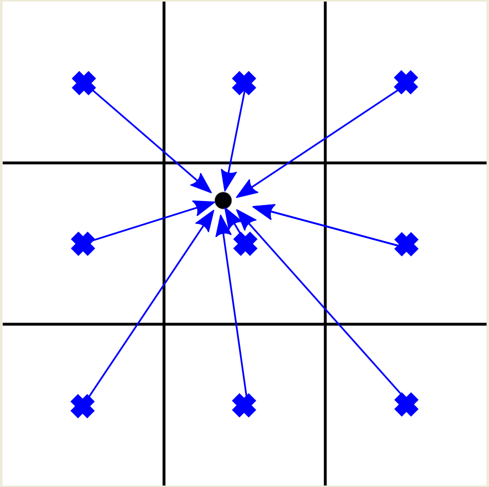
Calculation progress

- Fields on the staggered grid are interpolated to cell centers
- Electromagnetic force field is calculated in the particle positions
- Particle is accelerated and moves
- Current density is updated from particle movement
- Fields are updated with the use of current density



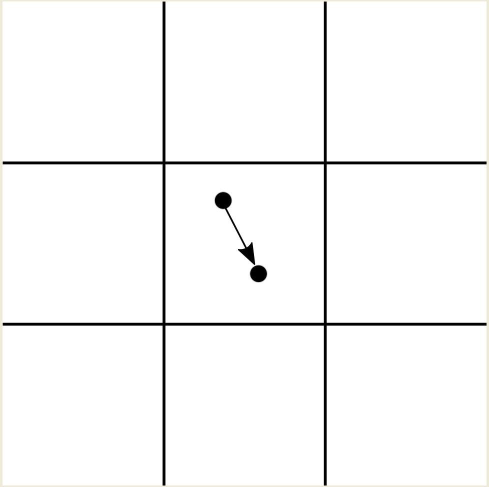
Calculation progress

- Fields on the staggered grid are interpolated to cell centers
- Electromagnetic force field is calculated in the particle positions
- Particle is accelerated and moves
- Current density is updated from particle movement
- Fields are updated with the use of current density



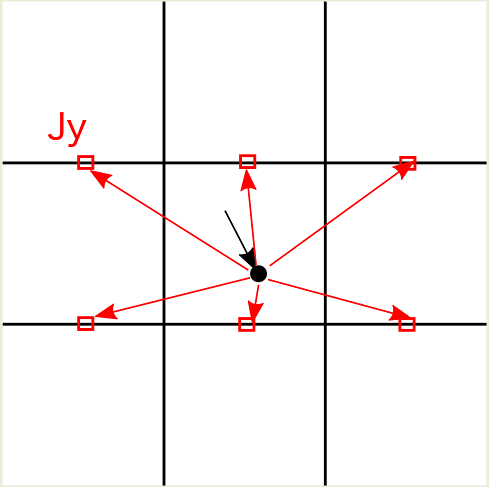
Calculation progress

- Fields on the staggered grid are interpolated to cell centers
- Electromagnetic force field is calculated in the particle positions
- Particle is accelerated and moves
- Current density is updated from particle movement
- Fields are updated with the use of current density



Calculation progress

- Fields on the staggered grid are interpolated to cell centers
- Electromagnetic force field is calculated in the particle positions
- Particle is accelerated and moves
- Current density is updated from particle movement
- Fields are updated with the use of current density



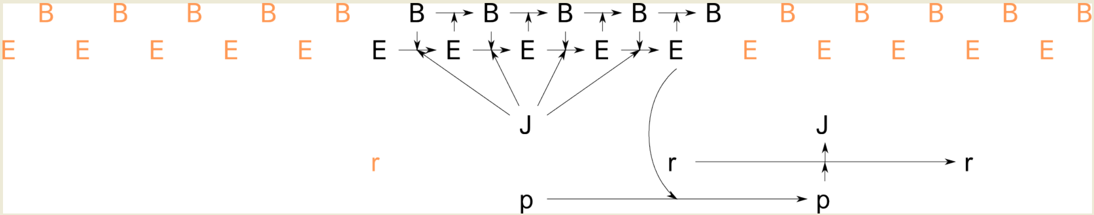
Multiscale model

Different time steps for field updates and for particle movement

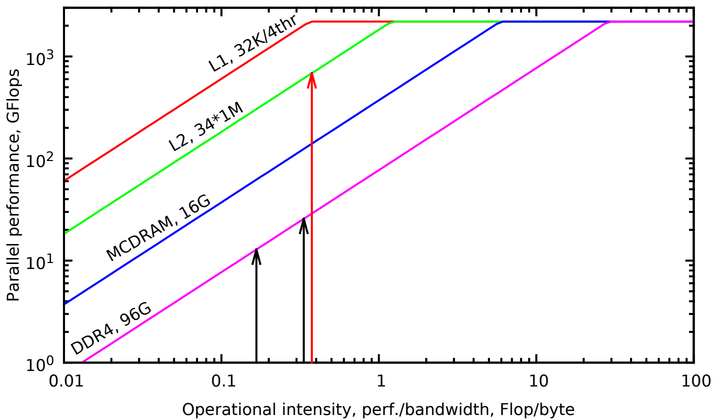
$$dt_{FLD} \leq \frac{dx}{\sqrt{3}c} \sim \frac{\lambda_D}{\sqrt{3}c}, \quad dt_{PIC} < \frac{1}{\omega_e}$$

$$dt_{PIC} \sim 10 dt_{FLD}$$

Each $Nt = dt_{PIC}/dt_{FLD}$ field steps one particle push is peformed



Memory bound problem

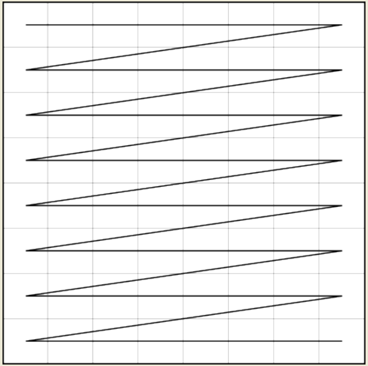


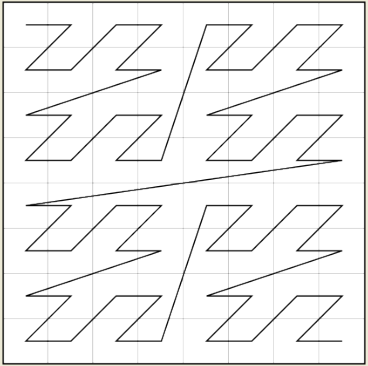
via Doerfler, Douglas, et al. "Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor 2016.

Locally Recursive non-Locally Asynchronous algorithms provide an optimal order of computations in terms of

- locality of data access
- number of communication events

Locally recursive data storage (Morton Z-curve)





2D z-curve array
3rd axis is left for vectorization
Domain size is $(Nblock \cdot 2^{MaxRank}) \times 2^{MaxRank} \times Nz$

Locally recursive data storage (Morton Z-curve)

```
template <int dim, class T, int rank> struct cubeLR {
  cubeLR<dim, T, rank-1> data[1<<dim];
};
template <int dim, class T> struct cubeLR<dim, T, 1> {
  T data[1<<dim];
};
```

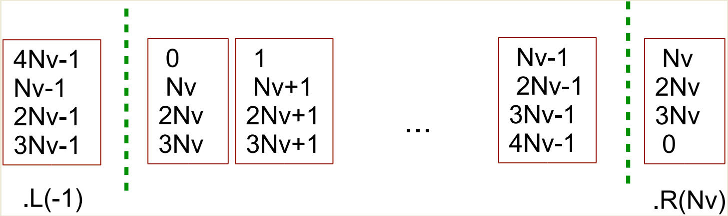
SIMD data type

```
struct fields{
  vecC<doubleV, Nz> Ex, Ey, Ez;
  vecC<doubleV, Nz> Jx, Jy, Jz;
  vecC<doubleV, Nz> Bx, By, Bz;
  ...
  pts_list ptslist;
};
template <class typeV, int Nz> struct vecC {
  const static int Nv=Nz/vec_length;
  typeV v[Nv];
  inline typeV& operator [](const int i) { ...}
  inline typeV  operator ()(const int i) { ...}
  inline typeV L(const int i) {...}
  inline typeV R(const int i) {...}
  };
#if defined(BASIC_VECTOR_SSE_AVX512)
typedef double __attribute__ ((vector_size(64)))
               __attribute__ ((aligned(64))) doubleV;
```

SSE/AVX Vectorization

```
Nv=Nz/4;
for (int iz=1; iz<Nv; iz++){
  F[ix,iy].Ey[iz] += (F[ix,iy].Bz(iz) - F[ix+1,iy].Bz(iz   ))*Cx
                   + (F[ix,iy].Bx(iz) - F[ix   ,iy].Bx(iz-1))*Cz
                   - Cdt*F[ix,iy].Jy(iz);
  };
F[ix,iy].Ey[0] += (F[ix,iy].Bz(0) - F[ix+1,iy].Bz(0)  )*Cx
                + (F[ix,iy].Bx(0) - F[ix,iy].Bx.L(-1))*Cz
                - Cdt*F[ix,iy].Jy(0);
```

.



SSE/AVX Vectorization

```
#include <immintrin.h>
 inline doubleV8 s2v(const double v)
   {return _mm512_set1_pd(v); }
 inline doubleV8 fabs(doubleV8 v)
   {return _mm512_and_pd(v,_mm512_set1_pd((0x7fffffffffffffff)));}
 inline doubleV8 Max(doubleV8 v1,doubleV8 v2)
   {return _mm512_max_pd(v1,v2); }
 inline doubleV8 Min(doubleV8 v1,doubleV8 v2)
   {return _mm512_min_pd(v1,v2); }
 inline double v2s(doubleV8& a, int i) { return ((double*)&a)[i]; }
```

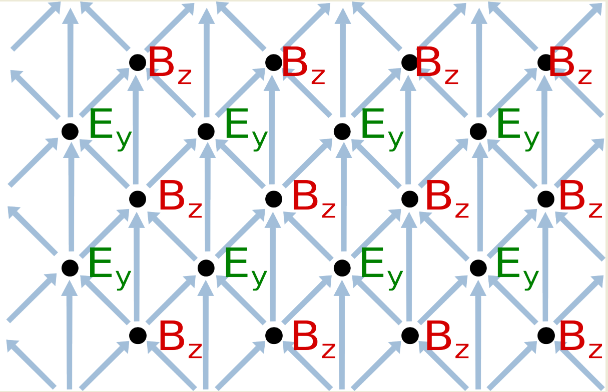
SIMD : AVX2 to AVX512

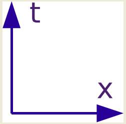
Time for 1536 field time steps on 0.2 billion cells:
$dt_{PIC} = 12 dt_{FLD}$

	PIC+FLD	FLD	% FLD
AVX2, xeon	510s	380s	75%
AVX2, knl	710s	230s	32%
AVX512, knl	600s	200s	33%

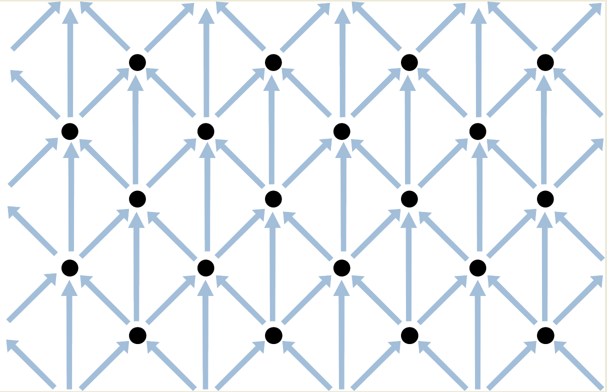
- Scalar performance in the standard Xeon (broadwell) is better
- Vectorization of particles is more important on KNL than it was before

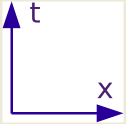
Algorithm as a dependency graph decomposition



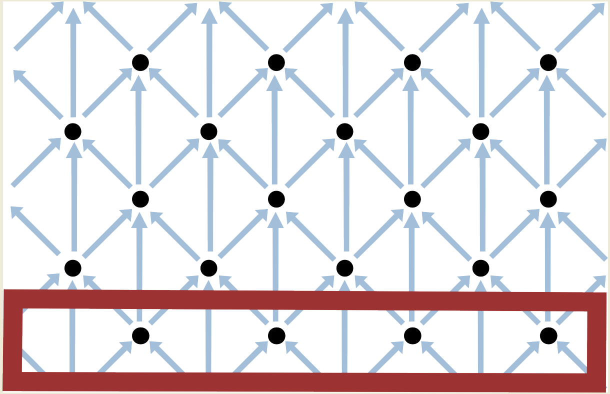


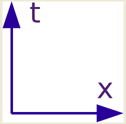
Algorithm as a dependency graph decomposition



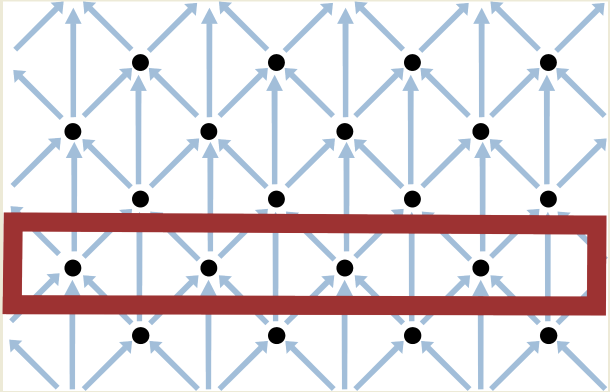


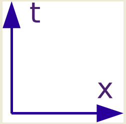
Algorithm as a dependency graph decomposition



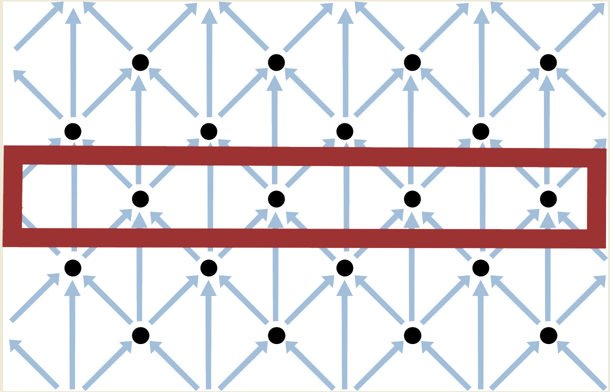


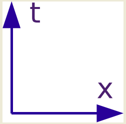
Algorithm as a dependency graph decomposition



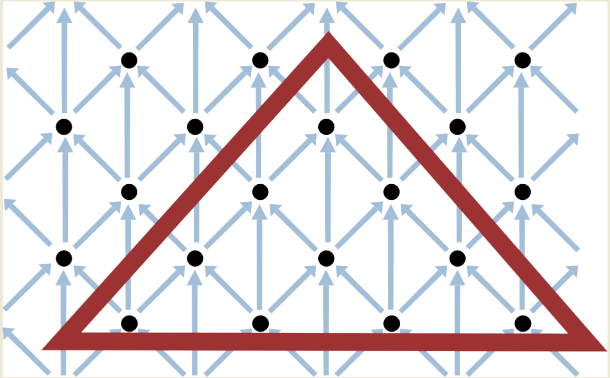


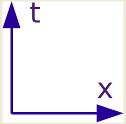
Algorithm as a dependency graph decomposition



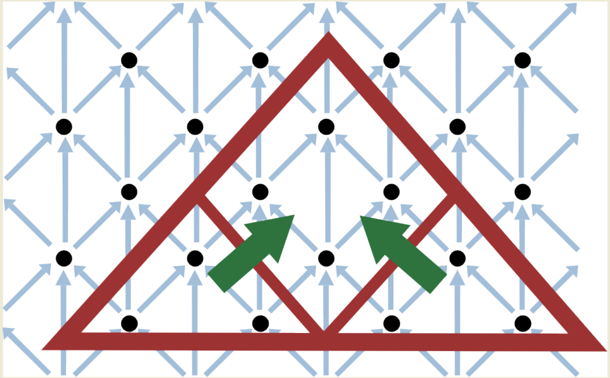


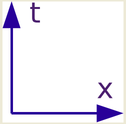
Algorithm as a dependency graph decomposition



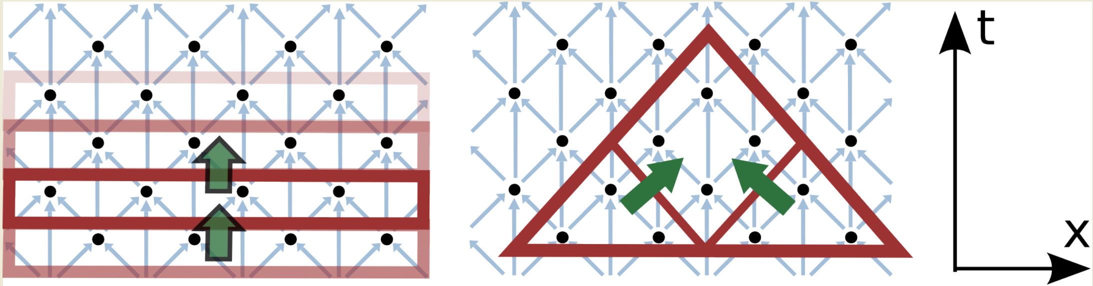


Algorithm as a dependency graph decomposition

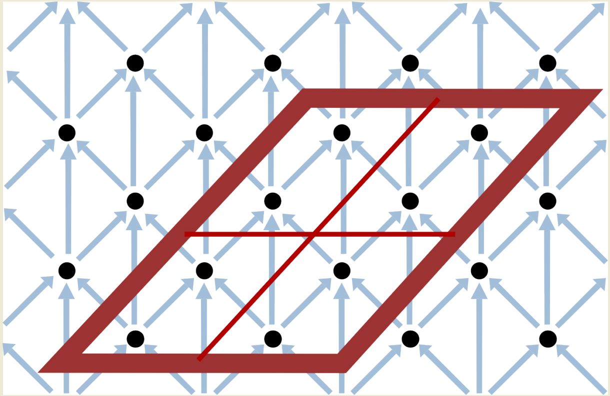


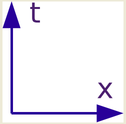


Algorithm as a dependency graph decomposition

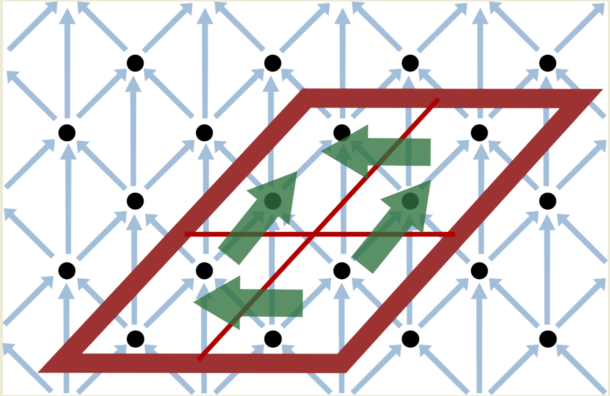


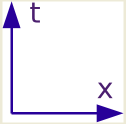
Algorithm as a dependency graph decomposition



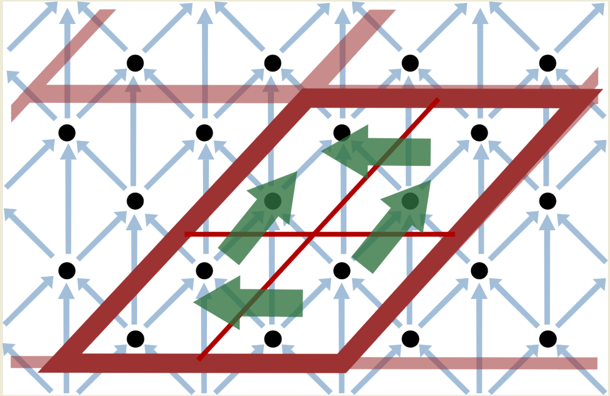


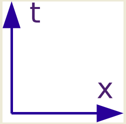
Algorithm as a dependency graph decomposition





Algorithm as a dependency graph decomposition

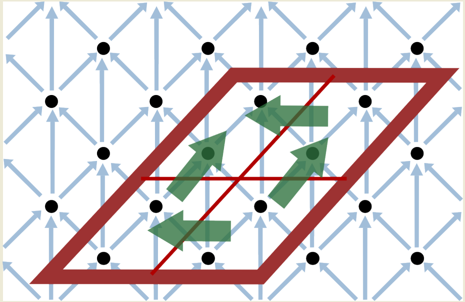


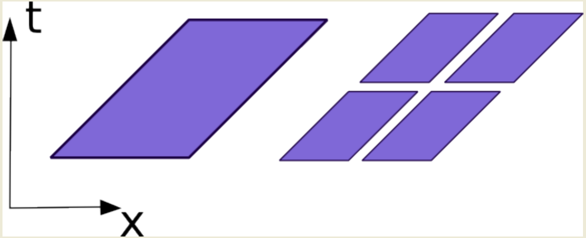


Algorithm as a dependency graph decomposition

Algorithm = shape + decomposition rule in space-time domain

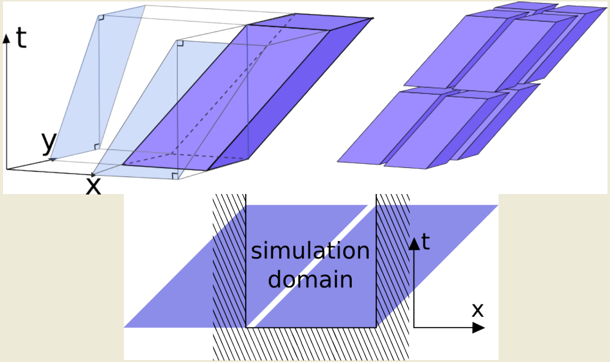
- Shape: perform calculation for the dependency graph points that fall inside a shape
- Decompostion rule: divide task into subtasks. Data dependencies should be unilateral.

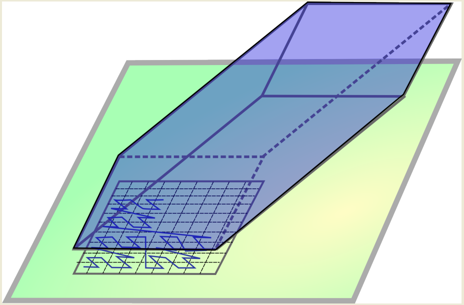




LRnLA algorithm ConeFold

- Cover all dependency graph with a ConeFold
- Decompose into 8 similar shapes
- Repeat recursively until 1 shape covers 1 computation



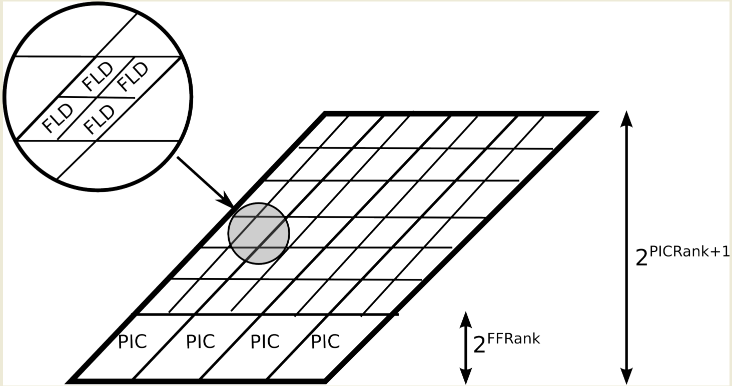


ConeFold: recursive template

```
template <int rank, class T> struct ConeFold {
  T* dat0;
  ConeFold(T* d): dat0(d) {}
  inline void update (int ix, int iy) {
    ConeFold <rank-1,T> c(dat0);
    c.update(2*ix+1, 2*iy+1); c.update(2*ix+2, 2*iy+2);
    c.update(2*ix+0, 2*iy+1); c.update(2*ix+1, 2*iy+2);
    c.update(2*ix+1, 2*iy+0); c.update(2*ix+2, 2*iy+1);
    c.update(2*ix+0, 2*iy+0); c.update(2*ix+1, 2*iy+1);
  }};
template <class T> struct ConeFold<0,T> {
  T* dat0;
  ConeFold(T* d): dat0(d) {}
  inline void update (int ix, int iy) {
    for (int iz=0; iz++; iz<Nz){
      dat0[LRind(ix,iy)].Ex[iz] += ...
      ...
    }};
```

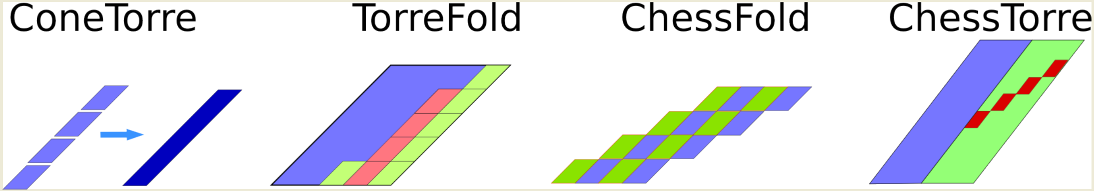
ConeFold for multiscale particle-in-cell

- Stencil of particle influence is wider than FDTD stencil
- Different scales of time steps for fields and particles

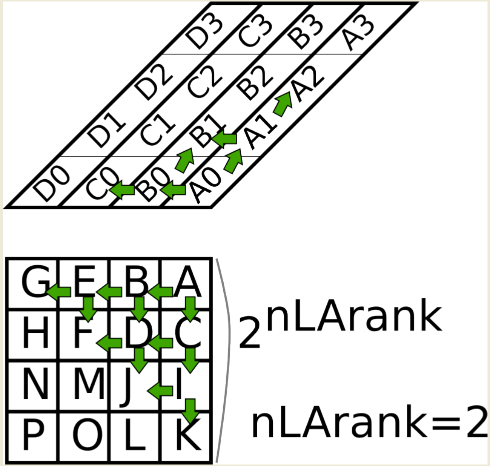


ConeFold extensions for different models of parallelism

- Stack ConeFolds on top of each other for even higher locality
- Trace data dependencies between shapes to find asynchronous computation blocks
- Combine approaches and adjust parameters to adapt to the available hardware (many-core, NUMA, GPGPU, etc.)

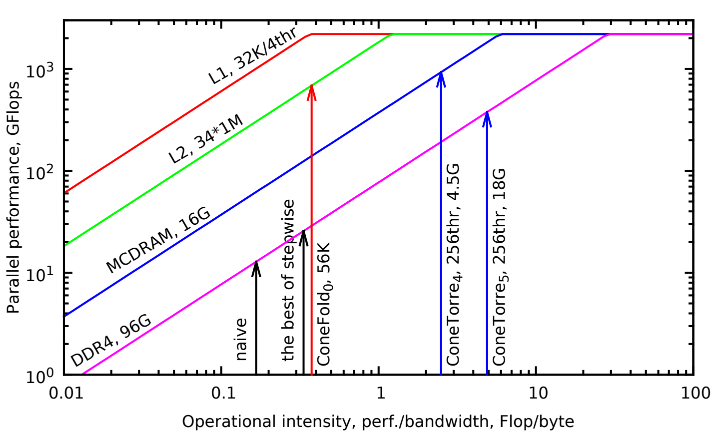


TorreFold: ConeFold shape with different decomposition rule



A0			
A1	B0	C0	
A2	B1	C1	D0
A3	B2	C2	D1
E0	B3	C3	D2
...	...	...	...

LRnLA algorithm advantages



- More operational intensity
- Better localization

For real LRnLA vs roofline results see

`http://www.mdpi.com/2079-3197/4/3/29`

`http://on-demand-gtc.gputechconf.com/gtc-quicklink/bdstAaW`

Performance on KNL

Porting to KNL:

- Enable AVX512
- Select more threads

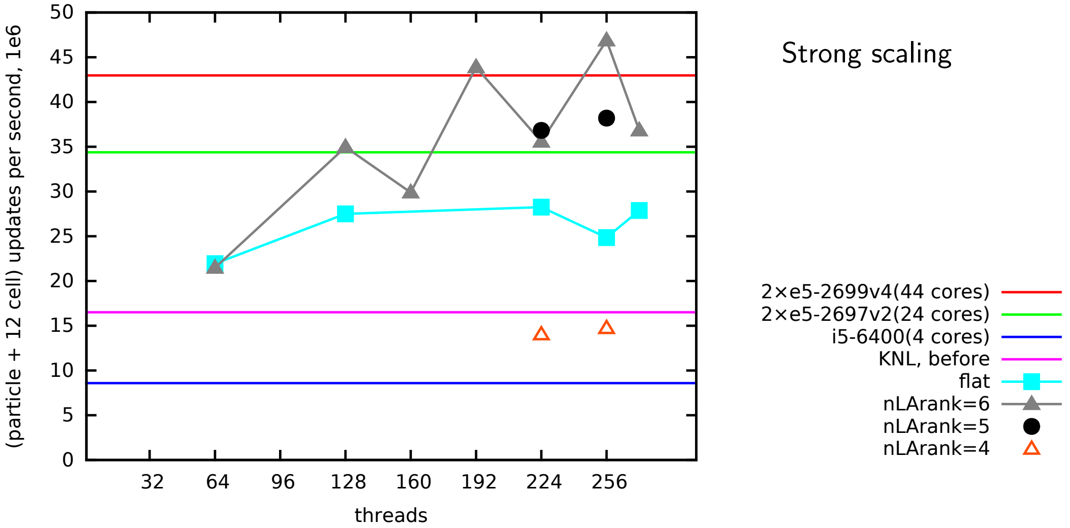
Problem for the performance test:

- $\sim 4.0 \cdot 10^8$ cells, 1 particle per cell
- $dt_{PIC} = 12 dt_{FLD}$
- 60 GB data

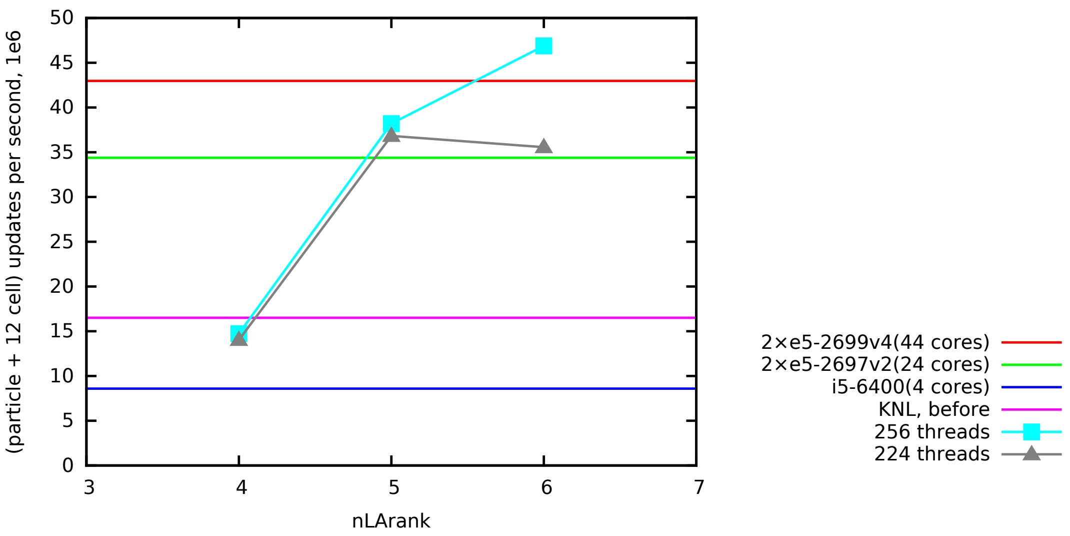
Processors for the comparison

Model	Architecture	Clock speed	Cores	Bandwidth	Power	Price
Core i5-6400	Skylake	2.7 GHz	4	34 GB/s	65 W	\$180
https://ark.intel.com/products/88185/Intel-Core-i5-6400-Processor-6M-Cache-up-to-3_30-GHz						
Xeon E5-2697v2	Ivy Bridge	2.7 GHz	2×12	2×60 GB/s	2×130W	\$2600 $\times 2$
http://ark.intel.com/products/75283/Intel-Xeon-Processor-E5-2697-v2-30M-Cache-2_70-GHz						
Xeon E5-2699v4	Broadwell	2.2 GHz	2×22	2×77 GB/s	2×145W	\$4100 $\times 2$
http://ark.intel.com/products/91317/Intel-Xeon-Processor-E5-2699-v4-55M-Cache-2_20-GHz						
Xeon Phi 7250	Knights Landing	1.4 GHz	68	115/500 GB/s	215 W	\$4900
http://ark.intel.com/products/94035/Intel-Xeon-Phi-Processor-7250-16GB-1_40-GHz-68-core						

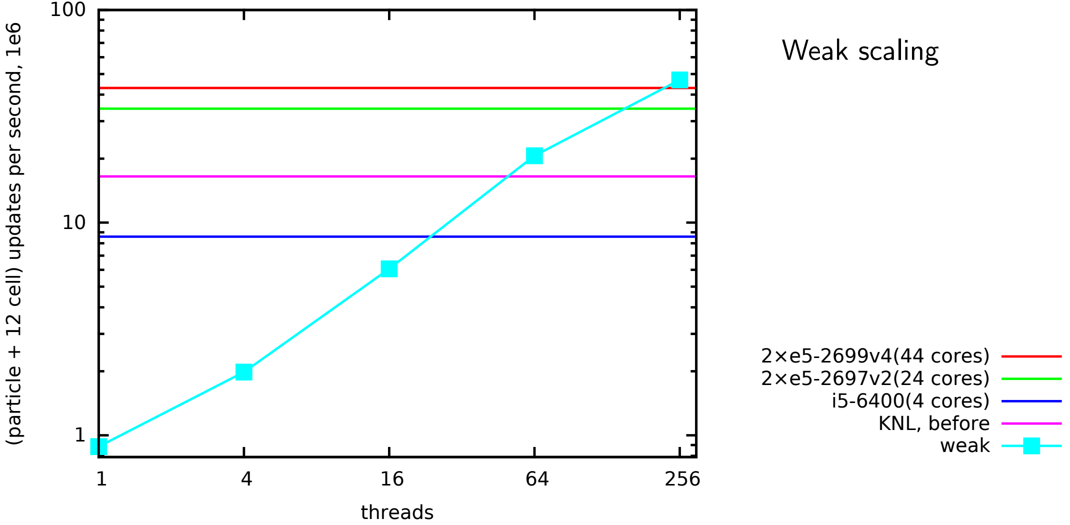
Performance results on KNL



Performance results on KNL



Performance results on KNL



Conclusion

- Since KNL acts as an extension to the usual SIMD/many-core paradigm, porting to KNL was not difficult
- Points of interest
  - MCDRAM mode
  - AVX512 instructions
  - thread affinity
- The use of space-time decomposition algorithm enhances the locality of computations and scaling efficiency

Future work

- Enable SIMD for particle computation
- Control the affinity of POSIX threads