Accelerating Octo-Tiger: Stellar Mergers on Intel Knights Landing with HPX

David Pfander*, Gregor Daiß*, Dominic Marcello**, Hartmut Kaiser**, Dirk Pflüger*

* University of Stuttgart ** Louisiana State University

May 14, 2018

SimTech

STE||AR GROUP
@ Louisiana State University

Agenda

HPX and Vc

Stellar Mergers with Octo-Tiger

A Futurized Fast Multipole Method

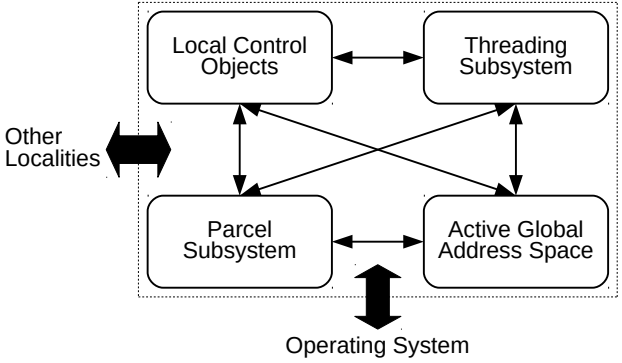
Interactions Stencil Kernels

Results on Intel Knights Landing (and Skylake)

Conclusion and Outlook

HPX for Parallelization

- HPX is a modern C++-based parallelization framework:



- Same syntax and semantics locally and distributed
- Supports futurization-continuation model
- Work stealing scheduler

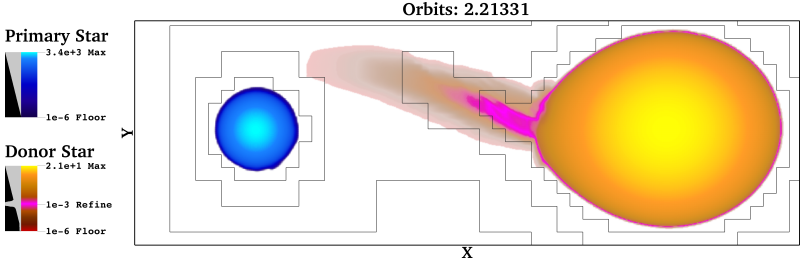
Vc for Vectorization

```
using Vc::double_v;

double_v res(0.0);
for (size_t i = 0; i < N; i++) {
  double_v a(&A[i], flags::element_aligned);
  double_v b(&B[i], flags::element_aligned);
  res += a * b; // FMA (expr. templates)
} // now reduce res
```

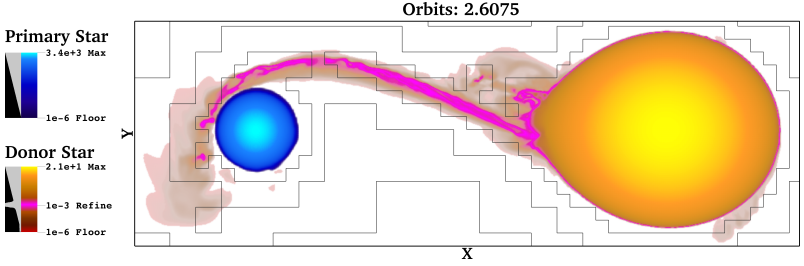
- Scalar-like vector expressions
- Portable (SSE, AVX1/2/512, NEON)
- Prescriptive vectorization (cf. OpenMP SIMD clause)
- (Developed by Matthias Kretz, proposed for standardization (P0214R3))

Stellar Mergers with Octo-Tiger



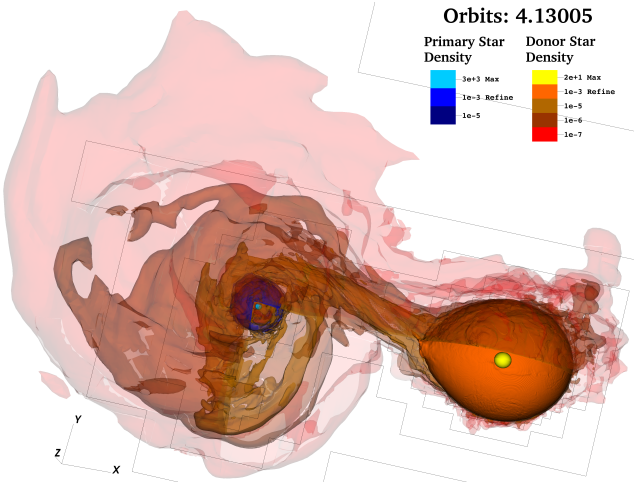
- Simulates star systems with two or more stars
- Important scenario: merger in double white dwarf systems
- Stellar merger lead to astrophysical phenomena such as type Ia supernovae
- (Written in C++11/14)

Stellar Mergers with Octo-Tiger



- Simulates star systems with two or more stars
- Important scenario: merger in double white dwarf systems
- Stellar merger lead to astrophysical phenomena such as type Ia supernovae
- (Written in C++11/14)

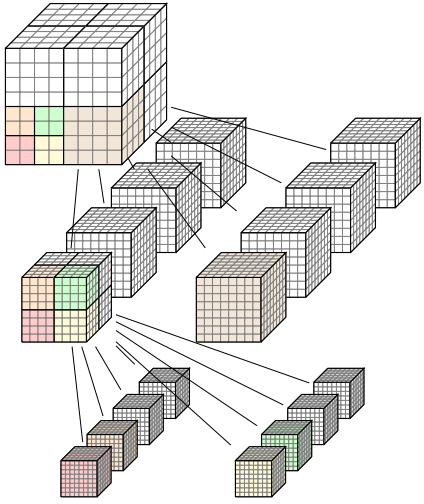
Stellar Mergers in 3d



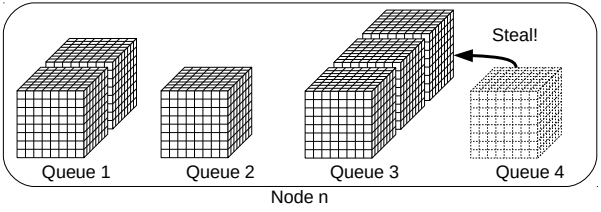
- Unique: Octo-Tiger conserves angular (and linear) momentum

Calculating Gravity

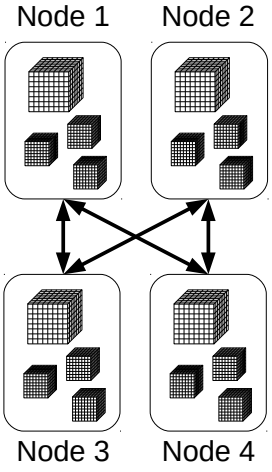
- Fast multipole method (FMM) to compute gravity
- Adaptive octrees as main data structure (AMR)
- Nodes have $8 \times 8 \times 8$ subgrid
- History of fast tree-based codes (Dehnen 2000)



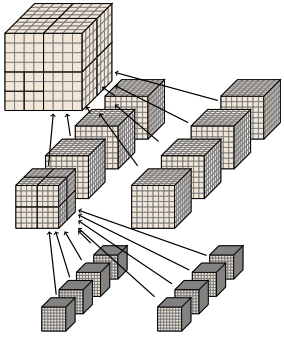
Parallelizing Octo-Tiger with HPX



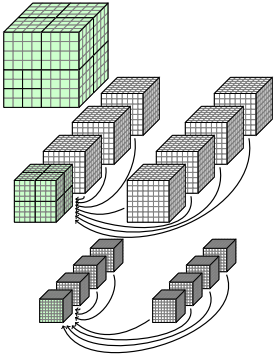
- Octo-Tiger uses $8 \times 8 \times 8$ subgrids as parallelization primitives
- Work stealing by moving octree nodes
- Octree nodes are AGAS objects



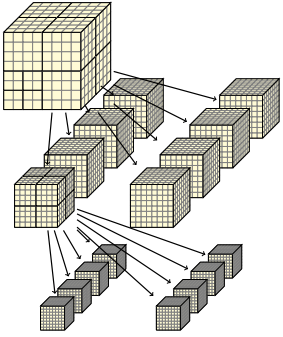
Futurized Tree Traversals



(1) Calc. multipole moments



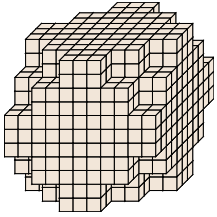
(2) Same-level interactions



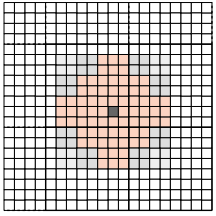
(3) Apply multipole expansions

- (2) most expensive!
- Steps partially overlap, thanks to futurization and HPX!
- Scalability proven with full-system runs (9640 KNLs) on Cori at NERSC

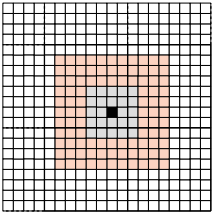
Multipole Interaction Stencil



3d stencil



Top view



Center slice

- Opening criterion:

$$\frac{1}{|\boldsymbol{Z}_1^f - \boldsymbol{Z}_2^f|} \leq \Theta < \frac{1}{|\boldsymbol{Z}_1^c - \boldsymbol{Z}_2^c|}$$

- $\boldsymbol{Z}_1^f$ and $\boldsymbol{Z}_2^f$ coordinate centers of multipoles, $\boldsymbol{Z}_1^c$ and $\boldsymbol{Z}_2^c$ coordinate center of parents
- $\Theta := 0.35 \implies$ stencil with 1074 elements
- Need access to neighbor's subgrids (27 element neighborhood)

Multipole Expansion Calculation

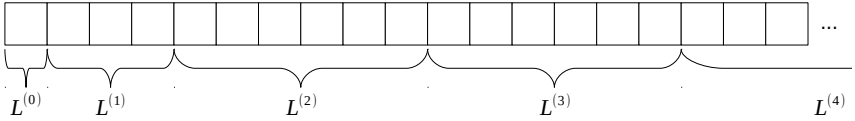
- Calculation of multipole expansion coefficients $L_q^{(i)}$ for cell q
- (1) Calculate $D^{(0)}, D_m^{(1)}, D_{mn}^{(2)}, D_{mnp}^{(3)}$ (114 FLOPS each), (2) Calculate $L_q^{(i)}$:

$$
\begin{aligned}
L_q^{(0)} &:= \sum_l \left[M_l^{(0)} D^{(0)}(\boldsymbol{R}_{l,q}) + M_{l,m}^{(1)} D_m^{(1)}(\boldsymbol{R}_{l,q}) + M_{l,mn}^{(2)} D_{mn}^{(2)}(\boldsymbol{R}_{l,q}) + M_{l,mnp}^{(3)} D_{mnp}^{(3)}(\boldsymbol{R}_{l,q}) \right] \\
L_{q,m}^{(1)} &:= \sum_l \left[M_l^{(0)} D_m^{(1)}(\boldsymbol{R}_{l,q}) + M_{l,n}^{(1)} D_{mn}^{(2)}(\boldsymbol{R}_{l,q}) + M_{l,np}^{(2)} D_{mnp}^{(3)}(\boldsymbol{R}_{l,q}) \right] \\
L_{q,mn}^{(2)} &:= \sum_l \left[M_l^{(0)} D_{mn}^{(2)}(\boldsymbol{R}_{l,q}) + M_{l,p}^{(1)} D_{mnp}^{(3)}(\boldsymbol{R}_{l,q}) \right] \\
L_{q,mnp}^{(3)} &:= \sum_l M_l^{(0)} D_{mnp}^{(3)}(\boldsymbol{R}_{l,q})
\end{aligned}
$$

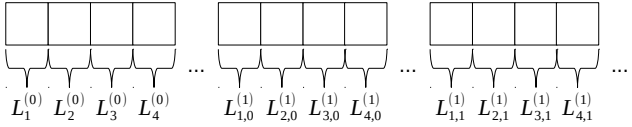
  (Multipole moments $M^{(i)}$ given)
- l iterates neighboring cells through stencil (1074 elements)
- m, n, p iterated in $\{0, 1, 2\} \implies$ 40 coefficients (20 unique)

Vector-Friendly Storage

- Small arrays per cell in subgrid, for $L^{(i)}$:



- Implemented as Struct-of-Array for vectorization:



- For $L^{(i)}, M^{(i)}, D^{(i)}$: 20 + 20 + 35 = 75 coefficients:
  - Per cell in subgrid
  - Loaded for each stencil element
- Scalar-like branch-free formulation of FMM kernels possible

Obtained FMM Compute Kernels

kernel	FLOPs per interaction	stencil size	floating point op.	Mem.	arith. intensity
M2M/M2P corr.	455	1,074	173,089,280	920 kB	183 F/B
M2M/M2P	295	1,074	112,222,720	896 kB	122 F/B
P2P/P2M corr.	360	1,074	136,949,760	872 kB	153 F/B
P2P/P2M	215	1,074	81,789,440	752 kB	106 F/B

- Four kernels:
  - Less work needed for leaf nodes
  - Angular correction
- M2P to be read as Multipole-to-Monopole (“Particle”)
- Generally compute bound, large caches needed

Experiments

Intel Xeon Phi 7250 (KNL):

- 68C, 1MB L2 cache per dual-core tile
- 1.4 GHz base, 1.32 GHz under heavy AVX512 load
- 2872 GFLOPS double precision peak

2xIntel Xeon Silver 4116 (SKL):

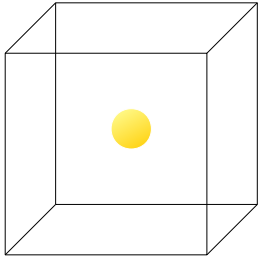
- 2x12C, 1 MB L2 cache per core
- 2.1 GHz base, 1.5 GHz under heavy AVX512 load
- 576 GFLOPS double precision peak

Evaluation scenario:

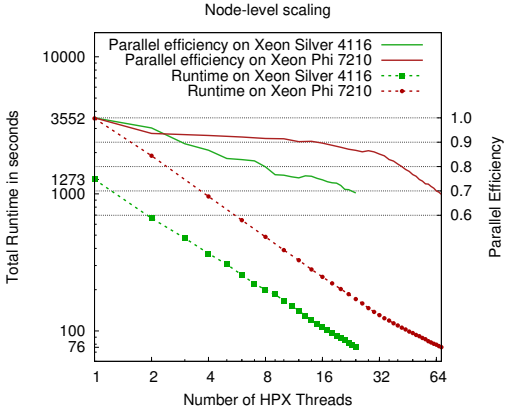
- Rotating star in equilibrium
- Domain 30x larger than star, adaptively-refined grid
- Demonstrates angular momentum conservation

©Intel

©Intel

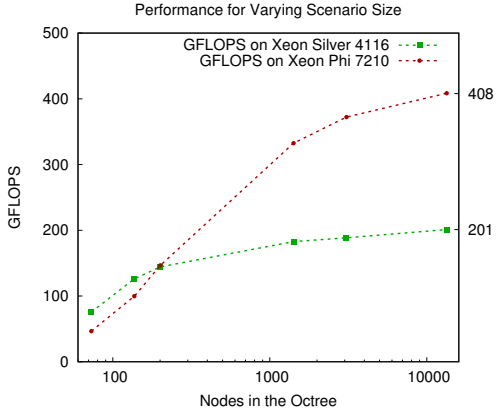


Node-Level Scaling



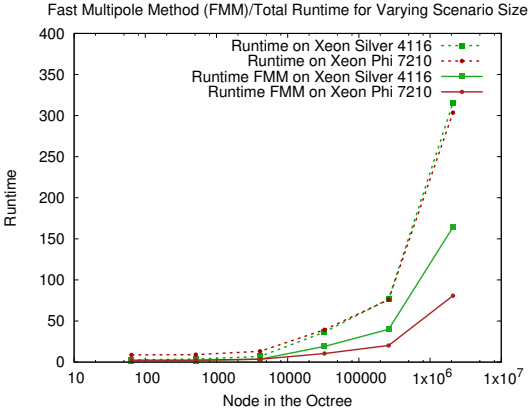
- Calculated with level 6 grid (3081 nodes), whole application!
- Cache too small on KNL

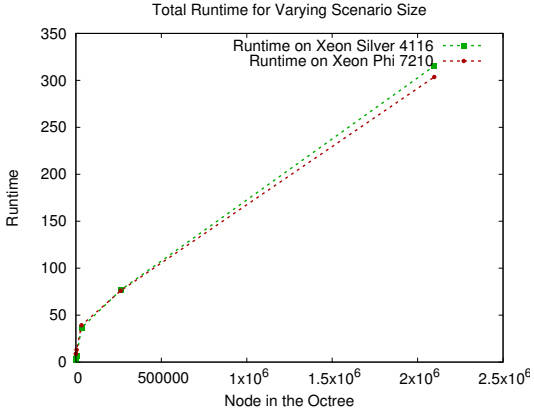
FMM Kernels Performance



- GFLOPS average over all four kernel variants
- KNL needs a larger grid
- 14% peak on KNL, 34% peak on SKL, KNL faster

Application Runtime





- Strong improvements for FMM
- Other application parts need to be further improved (regridding)

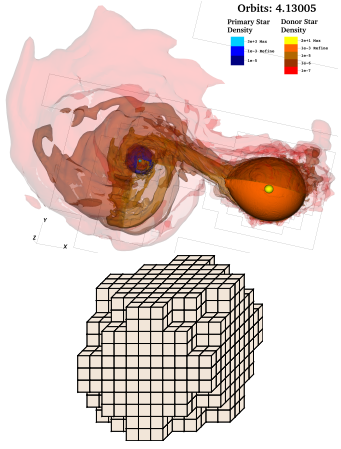
Conclusion and Outlook

Lessons learned:

- Futurization-continuation scaled up to 9640 nodes
- HPX+Vc enables fast and scaling high-level impl.
- KNL is a tough target for node-level performance

Next Steps:

- More cache-friendly FMM algorithms
- Deal with non-FMM (grid-related) bottlenecks
- Porting to Piz Daint at CSCS (Nvidia P100)



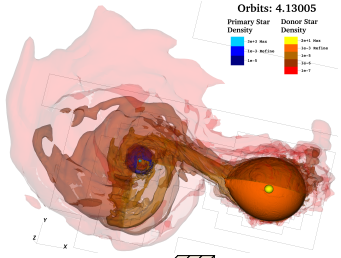
Conclusion and Outlook

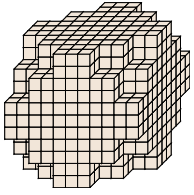
Lessons learned:

- Futurization-continuation scaled up to 9640 nodes
- HPX+Vc enables fast and scaling high-level impl.
- KNL is a tough target for node-level performance

Next Steps:

- More cache-friendly FMM algorithms
- Deal with non-FMM (grid-related) bottlenecks
- Porting to Piz Daint at CSCS (Nvidia P100)





Questions?

Computing the Gradients of the Gravitational Potential

- $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{R}^3$ center of masses, with $\boldsymbol{R} := \boldsymbol{X} - \boldsymbol{Y}$, $d := ||\boldsymbol{R}||_2$
- Gradients of the gravitational potential are

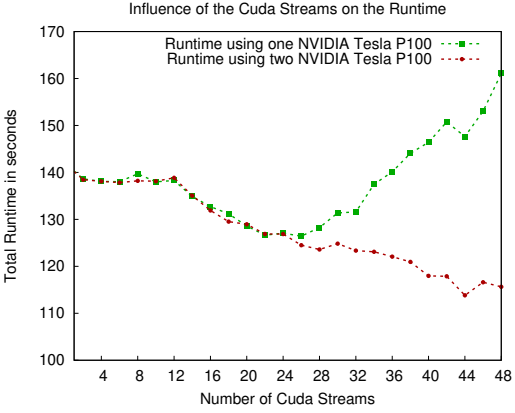
$$\nabla^{(n)} g(\boldsymbol{R}) = -\nabla^{(n)} \frac{1}{d} := D^{(n)}$$

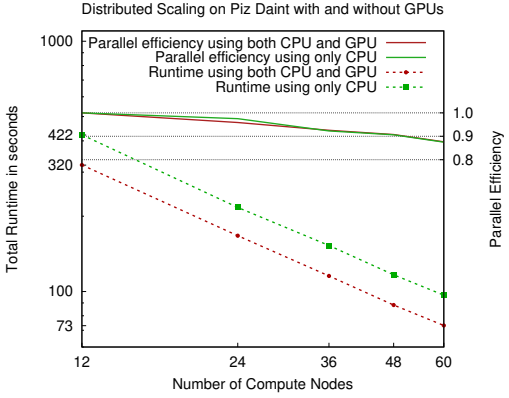
- Thus,

$$D^{(0)}(\boldsymbol{R}) := -\frac{1}{d},$$
$$D_i^{(1)}(\boldsymbol{R}) := \frac{R_i}{d^3},$$
$$D_{ij}^{(2)}(\boldsymbol{R}) := -\frac{3R_iR_j - \delta_{ij}d^2}{d^5},$$
$$D_{ijk}^{(3)}(\boldsymbol{R}) := \frac{15R_iR_jR_k - 3(\delta_{ij}R_k + \delta_{jk}R_i + \delta_{ki}R_j)d^2}{d^7}$$

- Iterating $i, j, k \in \{0, 1, 2\}$, 40 expression to evaluate (20 unique)
- Omitted $D^{(4)}$

Early Result on P100





- Using GPU as pure Co-processor through CUDA stream
- Implementation through HPX compute + CUDA clang
- Runtime improvements despite somewhat starving GPU