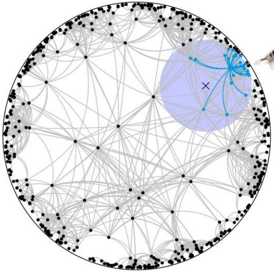
Karlsruher Institut für Technologie

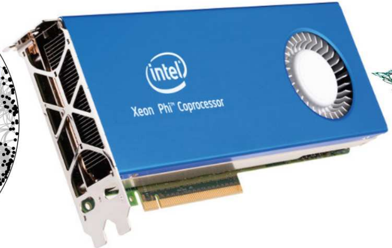
Parallel Graph Algorithms on the Xeon Phi Coprocessor

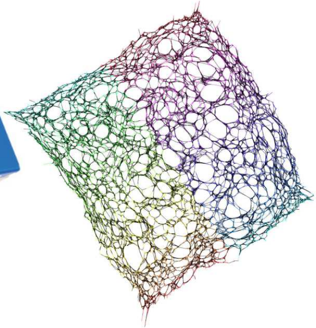
Master Thesis presentation

Dennis Felsing | 2015-09-07

INSTITUTE OF THEORETICAL INFORMATICS, RESEARCH GROUP PARALLEL COMPUTING

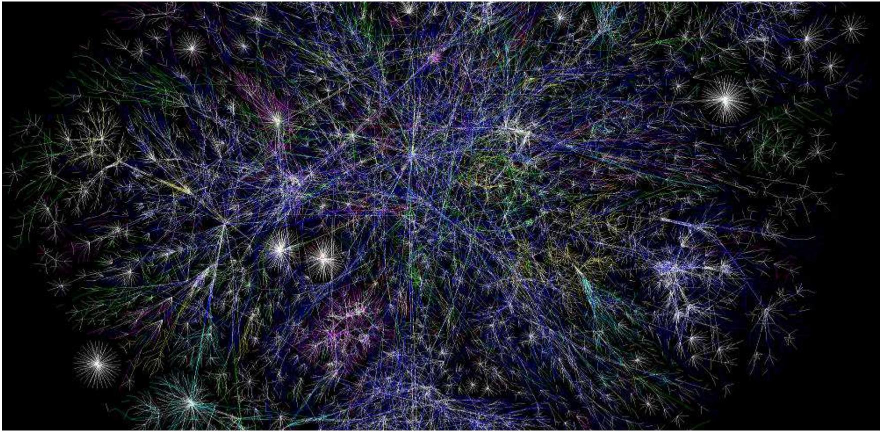






Motivation: Graphs

- Complex Network: graph with non-trivial topology
- Occur in social networks, cell biology, the internet...



Map of the Internet, `http://www.opte.org/maps/`

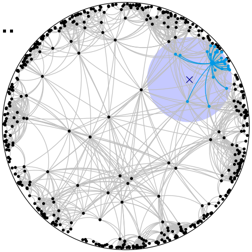
Motivation: Graphs

- Complex Network: graph with non-trivial topology
- Occur in social networks, cell biology, the internet...
- We consider two existing algorithms:

Motivation: Graphs

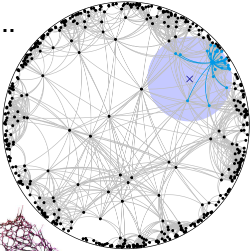
Karlsruher Institut für Technologie

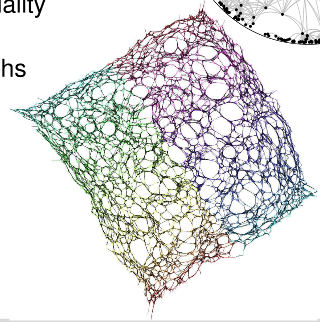
- Complex Network: graph with non-trivial topology
- Occur in social networks, cell biology, the internet...
- We consider two existing algorithms:
- Graph Generation
  - Create realistic complex networks with generator and parameters
  - Preserves privacy and confidentiality
  - No need to transfer big data
  - Scale to smaller and bigger graphs



Motivation: Graphs

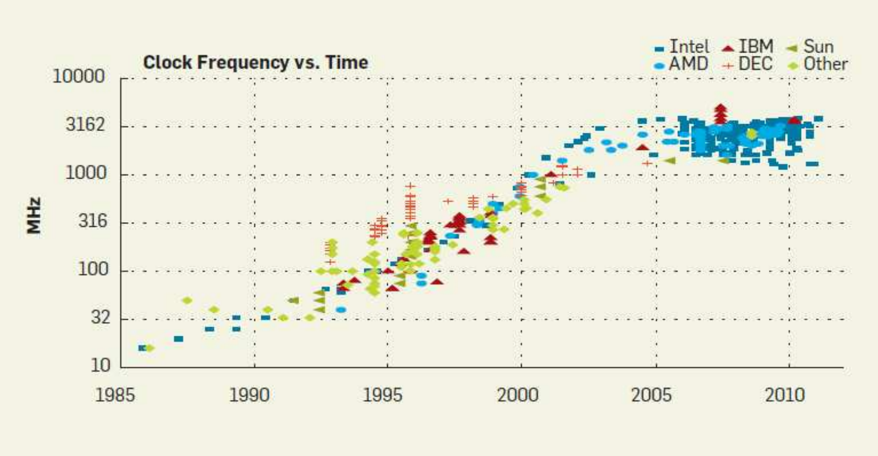
- Complex Network: graph with non-trivial topology
- Occur in social networks, cell biology, the internet...
- We consider two existing algorithms:
- Graph Generation
  - Create realistic complex networks with generator and parameters
  - Preserves privacy and confidentiality
  - No need to transfer big data
  - Scale to smaller and bigger graphs
- Graph Drawing
  - Lay out graph visually, mainly for human perception





Motivation: Computation

- Data sets grow fast: Internet size doubles every 5 years
- Clock frequency of processors stagnated in last decade



Motivation: Computation

Karlsruher Institut für Technologie

- Data sets grow fast: Internet size doubles every 5 years
- Clock frequency of processors stagnated in last decade
- Instead more parallelism in processors

Motivation: Computation

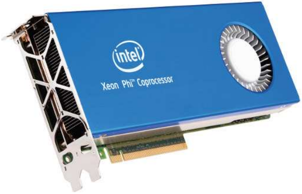
- Data sets grow fast: Internet size doubles every 5 years
- Clock frequency of processors stagnated in last decade
- Instead more parallelism in processors
- Modern GPUs as even more parallel alternative:
  - Massively parallel, thousands of cores
  - Large performance increases with more parallelization

Motivation: Computation

- Data sets grow fast: Internet size doubles every 5 years
- Clock frequency of processors stagnated in last decade
- Instead more parallelism in processors
- Modern GPUs as even more parallel alternative:
  - Massively parallel, thousands of cores
  - Large performance increases with more parallelization
  - General purpose programming more difficult
  - Graph algorithms with irregular data access challenging

Motivation: Computation

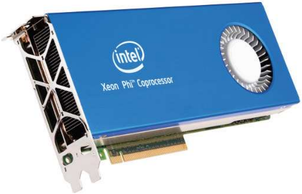
- Data sets grow fast: Internet size doubles every 5 years
- Clock frequency of processors stagnated in last decade
- Instead more parallelism in processors
- Modern GPUs as even more parallel alternative:
  - Massively parallel, thousands of cores
  - Large performance increases with more parallelization
  - General purpose programming more difficult
  - Graph algorithms with irregular data access challenging
- Intel Xeon Phi as alternative to the alternative:
  - 60 cores, more than a CPU, fewer than a GPU
  - Similar to program as CPU
  - Viable choice for graph algorithms?



Motivation: Computation

- Data sets grow fast: Internet size doubles every 5 years
- Clock frequency of processors stagnated in last decade
- Instead more parallelism in processors
- Modern GPUs as even more parallel alternative:
  - Massively parallel, thousands of cores
  - Large performance increases with more parallelization
  - General purpose programming more difficult
  - Graph algorithms with irregular data access challenging
- Intel Xeon Phi as alternative to the alternative:
  - 60 cores, more than a CPU, fewer than a GPU
  - Similar to program as CPU
  - Viable choice for graph algorithms?

$\Rightarrow$ Port two graph algorithms to Xeon Phi
and evaluate the porting and their performance

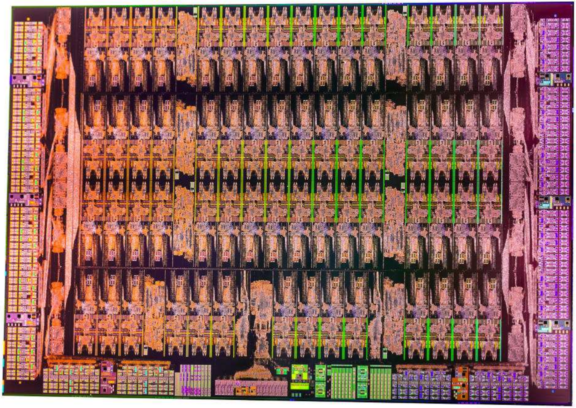


Overview

1. Xeon Phi Coprocessor
   - Hardware Architecture
   - Programming
2. Generation of Massive Complex Networks
   - Algorithm
   - Results
3. Graph Drawing using Graph Clustering
   - Algorithm
   - Results

Xeon Phi: Hardware Architecture

- Xeon Phi 5110P used, 60 in-order cores at 1 GHz
- Simple cores based on original Pentium design from 1994
- Augmented with 64-bit support (not x86-64)



Xeon Phi: Hardware Architecture

Karlsruher Institut für Technologie

- Xeon Phi 5110P used, 60 in-order cores at 1 GHz
- Simple cores based on original Pentium design from 1994
- Augmented with 64-bit support (not x86-64)

	Host System	Accelerator Card
Name	2× Xeon E5-2680	Xeon Phi 5110P
Release Date	2012	2012
Clock Frequency	2.7 GHz	1.05 GHz
Cores	2 × 8 (32 threads)	60 (240 threads)
RAM Capacity	256 GB	8 GB
RAM Bandwidth	51 GB/s	320 GB/s
SIMD Instructions	MMX, SSE, AVX (256 bit)	IMCI (512 bit)

Xeon Phi: Hardware Architecture

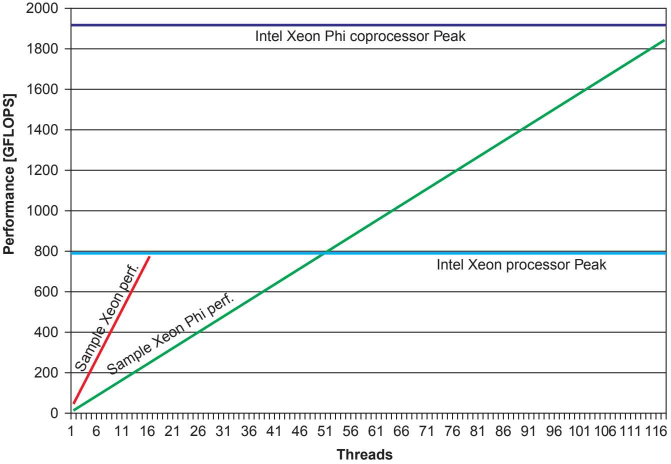
- Xeon Phi 5110P used, 60 in-order cores at 1 GHz
- Simple cores based on original Pentium design from 1994
- Augmented with 64-bit support (not x86-64)

	Host System	Accelerator Card
Name	2× Xeon E5-2680	Xeon Phi 5110P
Release Date	2012	2012
Clock Frequency	2.7 GHz	1.05 GHz
Cores	2 × 8 (32 threads)	60 (240 threads)
RAM Capacity	256 GB	8 GB
RAM Bandwidth	51 GB/s	320 GB/s
SIMD Instructions	MMX, SSE, AVX (256 bit)	IMCI (512 bit)

⇒ Parallelization and vectorization necessary to reach high performance

Xeon Phi: Hardware Architecture

- Xeon Phi 5110P used, 60 in-order cores at 1 GHz
- Simple cores based on original Pentium design from 1994
- Augmented with 64-bit support (not x86-64)



Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods:
  - OpenMP
  - OpenMP Offloading
  - Cilk Plus
  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods

Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods:
  - OpenMP

```
float a[MAX], b[MAX], c[MAX];

for (i = 0; i < MAX; i++)
  c[i] = a[i] + b[i];
```

  - OpenMP Offloading
  - Cilk Plus
  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods

Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods:
  - OpenMP

    ```
    float a[MAX], b[MAX], c[MAX];
    #pragma omp parallel for
    for (i = 0; i < MAX; i++)
      c[i] = a[i] + b[i];
    ```

  - OpenMP Offloading
  - Cilk Plus
  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods

Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods:
  - OpenMP
  - OpenMP Offloading

```c
float a[MAX], b[MAX], c[MAX];
#pragma offload target(mic) in(a, b) out(c)
{
  #pragma omp parallel for
  for (i = 0; i < MAX; i++)
    c[i] = a[i] + b[i];
}
```

  - Cilk Plus
  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods

Xeon Phi: Programming

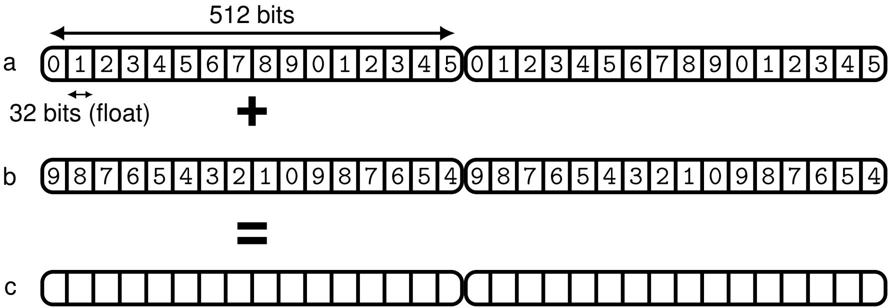
- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods:
  - OpenMP
  - OpenMP Offloading

```
float a[MAX], b[MAX], c[MAX];
#pragma offload target(mic:0) in(a, b) out(c) signal(c)
{
  #pragma omp parallel for
  for (i = 0; i < MAX; i++)
    c[i] = a[i] + b[i];
}
#pragma offload_wait target(mic:0) wait(c)
```

  - Cilk Plus
  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods

Xeon Phi: Programming

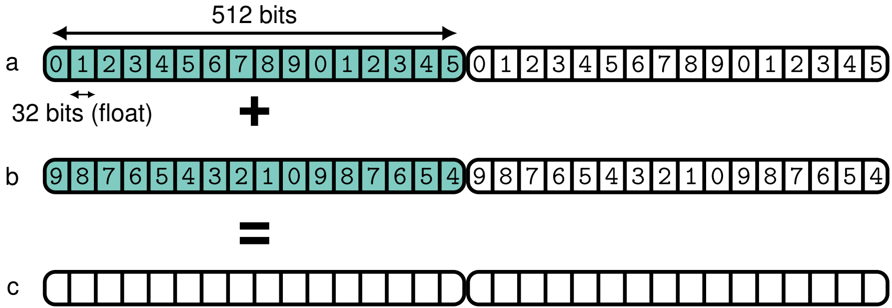
- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization: `c = _mm512_add_ps(a, b)`



  - Auto-Vectorization
  - Cilk Plus

Xeon Phi: Programming

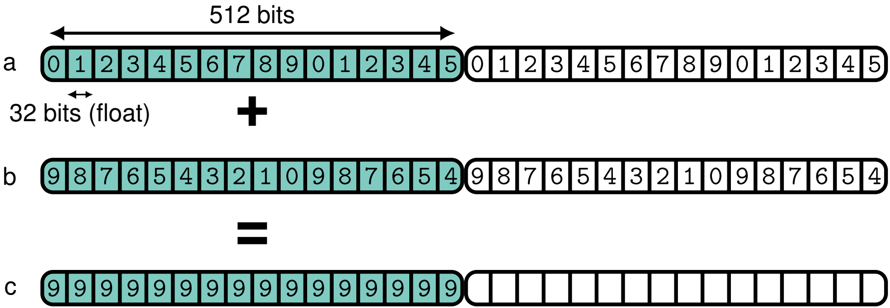
- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization: `c = _mm512_add_ps(a, b)`



  - Auto-Vectorization
  - Cilk Plus

Xeon Phi: Programming

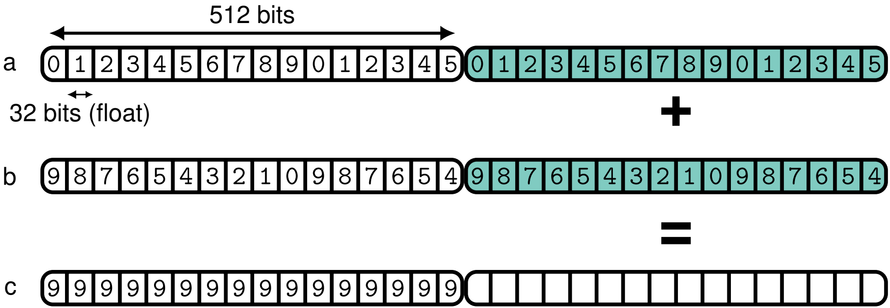
- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization: `c = _mm512_add_ps(a, b)`



  - Auto-Vectorization
  - Cilk Plus

Xeon Phi: Programming

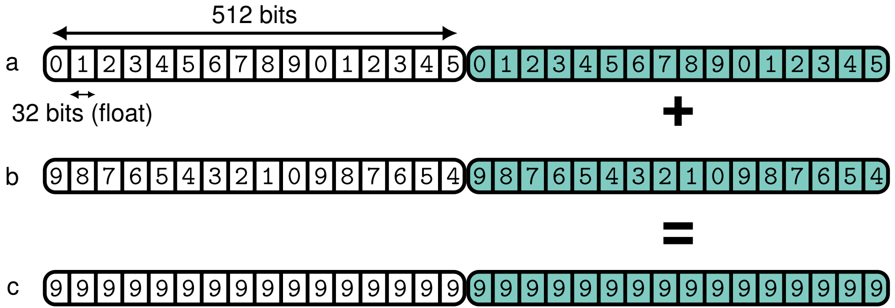
- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization: `c = _mm512_add_ps(a, b)`



  - Auto-Vectorization
  - Cilk Plus

Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization: `c = _mm512_add_ps(a, b)`



  - Auto-Vectorization
  - Cilk Plus

Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization
  - Auto-Vectorization

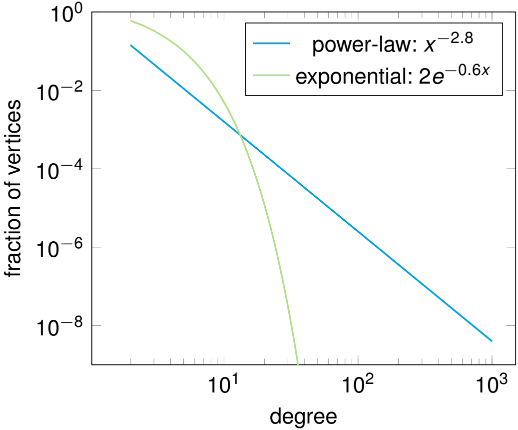
```c
float a[MAX] __attribute__((aligned(64)));
float b[MAX] __attribute__((aligned(64)));
float c[MAX] __attribute__((aligned(64)));
#pragma simd
for (i = 0; i < MAX; i++)
  c[i] = a[i] + b[i];
```

  - Cilk Plus

Network Generation: Algorithm

Desired properties of a complex network:

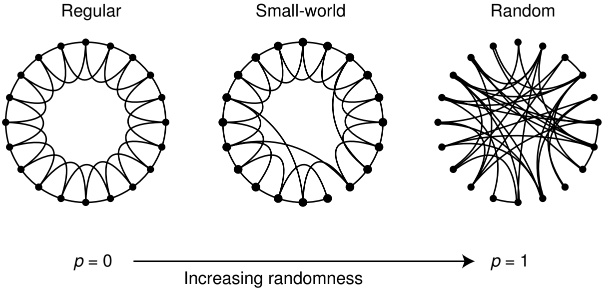
- Scale-Free: No typical vertex degree
  $\Rightarrow$ Degree distribution follows power law



Network Generation: Algorithm

Desired properties of a complex network:

- Scale-Free: No typical vertex degree
  $\Rightarrow$ Degree distribution follows power law
- Small-World: All nodes connected by short paths

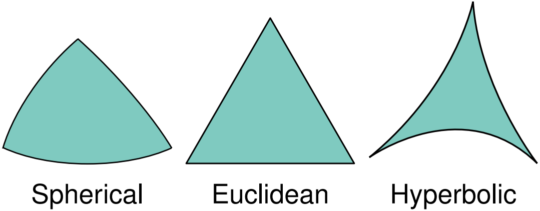


Network Generation: Algorithm

Desired properties of a complex network:

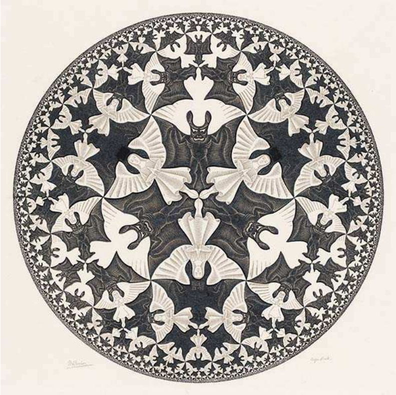
- Scale-Free: No typical vertex degree
  $\Rightarrow$ Degree distribution follows power law
- Small-World: All nodes connected by short paths

$\Rightarrow$ Generator using hyperbolic geometry performs well in both properties



Network Generation: Algorithm

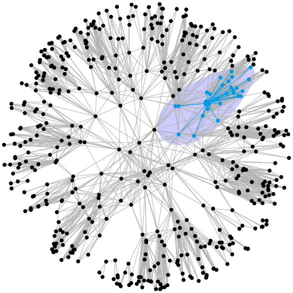
- Exponential expansion of space in hyperbolic geometry:
  Area of circle grows exponentially with distance from center
  ⇒ Natural embedding of graphs with tree-like structure
  ⇒ May also be good for generating graphs



M.C. Escher: Circle Limit IV

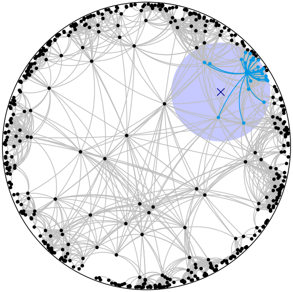
Network Generation: Algorithm

- Exponential expansion of space in hyperbolic geometry:
  Area of circle grows exponentially with distance from center
  ⇒ Natural embedding of graphs with tree-like structure
  ⇒ May also be good for generating graphs
- Hyperbolic generator: Distribute vertices in hyperbolic plane
  Edge when two vertices are close to each other (hyperbolic circle)



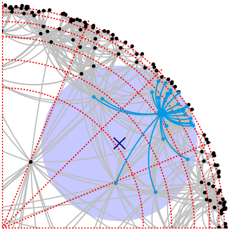
Network Generation: Algorithm

- Exponential expansion of space in hyperbolic geometry:
  Area of circle grows exponentially with distance from center
  ⇒ Natural embedding of graphs with tree-like structure
  ⇒ May also be good for generating graphs
- Hyperbolic generator: Distribute vertices in hyperbolic plane
  Edge when two vertices are close to each other (hyperbolic circle)
- Poincaré disk model: Mapping to Euclidean unit disk
  ⇒ Neighborhood transformed to Euclidean circle



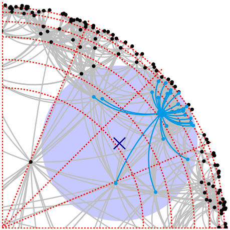
Network Generation: Algorithm

- Exponential expansion of space in hyperbolic geometry:
  Area of circle grows exponentially with distance from center
  ⇒ Natural embedding of graphs with tree-like structure
  ⇒ May also be good for generating graphs
- Hyperbolic generator: Distribute vertices in hyperbolic plane
  Edge when two vertices are close to each other (hyperbolic circle)
- Poincaré disk model: Mapping to Euclidean unit disk
  ⇒ Neighborhood transformed to Euclidean circle
- Polar quadtree: Efficiently determine neighborhood



Network Generation: Algorithm

- Exponential expansion of space in hyperbolic geometry:
  Area of circle grows exponentially with distance from center
  $\Rightarrow$ Natural embedding of graphs with tree-like structure
  $\Rightarrow$ May also be good for generating graphs
- Hyperbolic generator: Distribute vertices in hyperbolic plane
  Edge when two vertices are close to each other (hyperbolic circle)
- Poincaré disk model: Mapping to Euclidean unit disk
  $\Rightarrow$ Neighborhood transformed to Euclidean circle
- Polar quadtree: Efficiently determine neighborhood
  $\Rightarrow$ Subquadratic running time $O((n^{3/2} + m) \log n)$



Network Generation: Implementation

- Implementation part of NetworKit, high level C++11 code
- NetworKit ported to Intel C++ compiler 15.0 and Xeon Phi
- Working around compiler restrictions and bugs with *constexprs*, implicit conversions, null pointers, the standard library, and function traits on lambdas
- Three execution modes implemented and tested:
  No offloading: Entire code runs on Xeon Phi, not enough memory
  Full offloading: Offload parts of the calculation, keep results in memory of host system
  Partial offloading: Offload part of the calculation, other part on the host system

Network Generation: Results

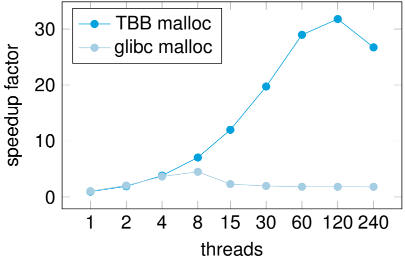
- Many memory allocations during algorithm to create dynamically sized lists of neighbors
- Allocations (`malloc`) are locking in default C library `glibc`
- On Xeon Phi more threads run in parallel than on CPU, so more allocations block each other

Network Generation: Results

- Many memory allocations during algorithm to create dynamically sized lists of neighbors
- Allocations (`malloc`) are locking in default C library `glibc`
- On Xeon Phi more threads run in parallel than on CPU, so more allocations block each other

⇒ Use non-locking allocations of Intel's Threading Building Blocks

- Scaling of initial implementation:



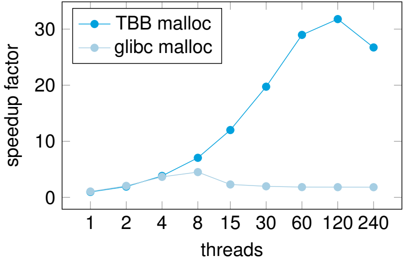
Network Generation: Results

- Many memory allocations during algorithm to create dynamically sized lists of neighbors
- Allocations (`malloc`) are locking in default C library `glibc`
- On Xeon Phi more threads run in parallel than on CPU, so more allocations block each other

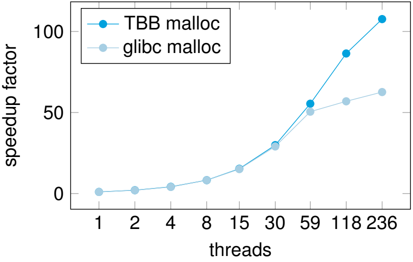
⇒ Use non-locking allocations of Intel's Threading Building Blocks

⇒ Reduce number of (re)allocations by reusing memory and preallocating expected size

- Scaling of initial implementation:

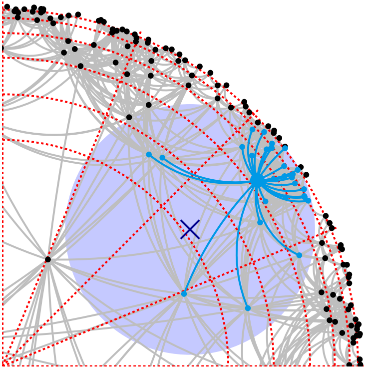


- Scaling of final, optimized code:



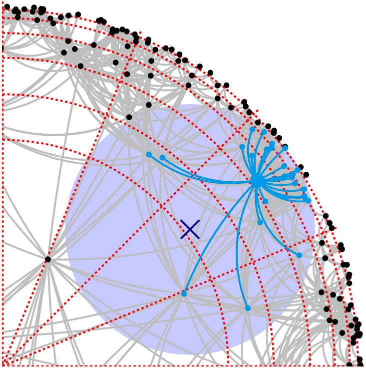
Network Generation: Results

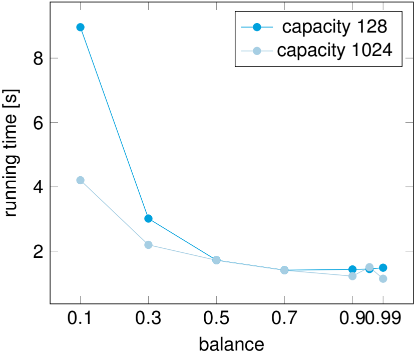
- Tuning parameters of the algorithm:
  Capacity: Maximum number of vertices in a leaf cell before split
  Balance: Share of area in outer children when splitting



Network Generation: Results

- Tuning parameters of the algorithm:
  Capacity: Maximum number of vertices in a leaf cell before split
  Balance: Share of area in outer children when splitting



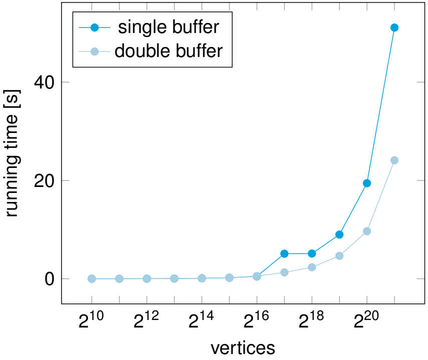


⇒ Imbalanced quadtree with greater space to outer children

Network Generation: Results

- Transferring parts of graph back to host system is slow
  $\Rightarrow$ Double Buffering on Phi and host

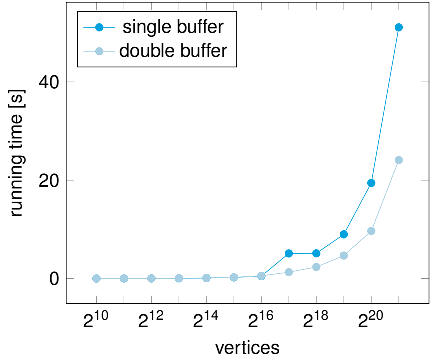
- Buffering for full offloading:



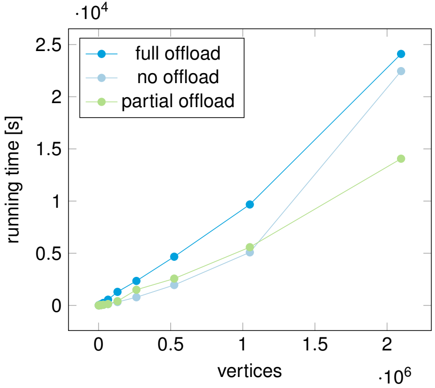
Network Generation: Results

- Transferring parts of graph back to host system is slow
  ⇒ Double Buffering on Phi and host

- Buffering for full offloading:



- Offloading comparison:



⇒ With many optimizations similar speed as dual Xeon, but not faster

Graph Drawing: Algorithm

We assume to have graphs with predefined target edge lengths

- Full Stress Model: Physical springs connecting all pairs of vertices
- Maxent-Stress Model: Minimize stress, maximize entropy:

$$M(x) = \underbrace{\sum_{\{u,v\} \in E} \overbrace{w_{uv}}^{\text{weight factor}} (||x_u - x_v|| - \overbrace{d_{uv}}^{\text{target edge length}})^2}_{\text{stress for target edge lengths}} - \alpha \underbrace{\sum_{\{u,v\} \notin E} \ln || \overbrace{x_u}^{\text{coordinate vector}} - x_v||}_{\text{entropy for rest}}$$

Graph Drawing: Algorithm

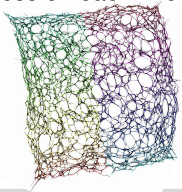
We assume to have graphs with predefined target edge lengths

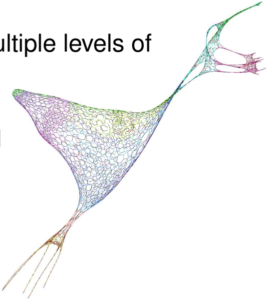
- Full Stress Model: Physical springs connecting all pairs of vertices
- Maxent-Stress Model: Minimize stress, maximize entropy:

$$M(x) = \underbrace{\sum_{\{u,v\}\in E} \overbrace{w_{uv}}^{\text{weight factor}} (||x_u - x_v|| - \overbrace{d_{uv}}^{\text{target edge length}})^2}_{\text{stress for target edge lengths}} - \alpha \underbrace{\sum_{\{u,v\}\notin E} \ln || \overbrace{x_u}^{\text{coordinate vector}} - x_v ||}_{\text{entropy for rest}}$$

- Multilevel Maxent-Stress Algorithm:
  - Minimize maxent-stress by clustering graph in multiple levels of hierarchy
  - Contract clusters into new supervertices
  - Iteratively solve maxent-stress on each finer level

$\Rightarrow$ Parallelizes well



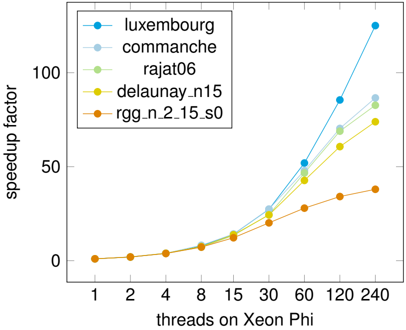


Graph Drawing: Results

- Preliminary implementation of multilevel maxent-stress graph drawing algorithm
- Parallelized with OpenMP
- Graphs of interest ($< 10^7$ edges) easily fit into Xeon Phi memory
  $\Rightarrow$ No expensive offloading necessary
- Source code libraries had to be fixed for ICPC
- No major dynamic allocations in algorithm
  $\Rightarrow$ Intel TBB's *malloc* has small effect
- Inner loop vectorizes well when isolated: speedup factor 7.0
  Smaller effect when embedded in real program
  $\Rightarrow$ Other calculations at same time, hyper-threading, memory connection busy

Graph Drawing: Results

Graph	n	m	Description	Phi	Host
nyc	264 346	365 050	Road Network	960.0	1845.9
luxembourg	114 599	119 666	Road Network	89.5	166.5
commanche	7920	11 880	Helicopter Mesh	2.6	3.5
rajat06	10 922	18 061	Circuit Simulation	3.5	4.5
delaunay_n15	32 768	98 274	Delaunay Triangulation	7.8	9.0
rgg_n_2_15_s0	32 768	160 240	Random Graph	5.3	3.6



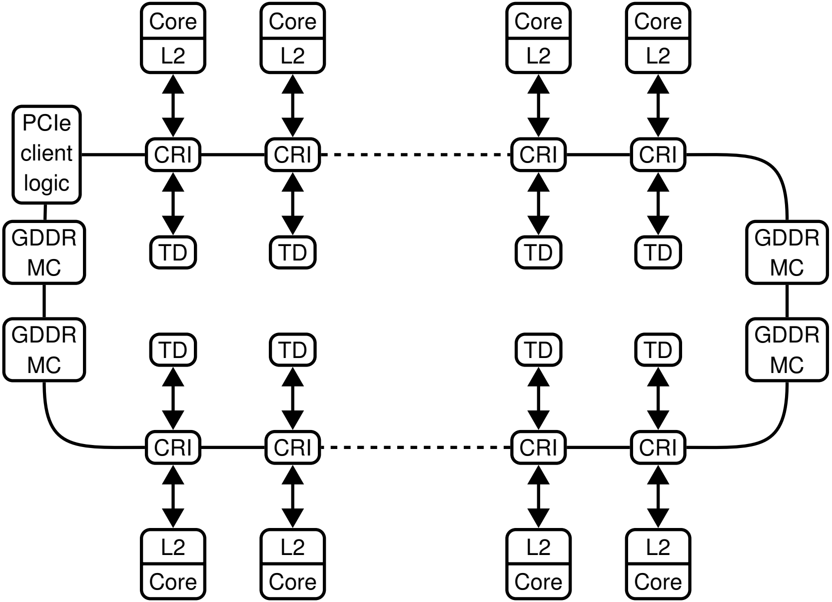
- All graphs small enough to fit into Xeon Phi memory
  $\Rightarrow$ Executed on Xeon Phi directly
- Good speedup for large sparse graphs

Conclusion and Outlook

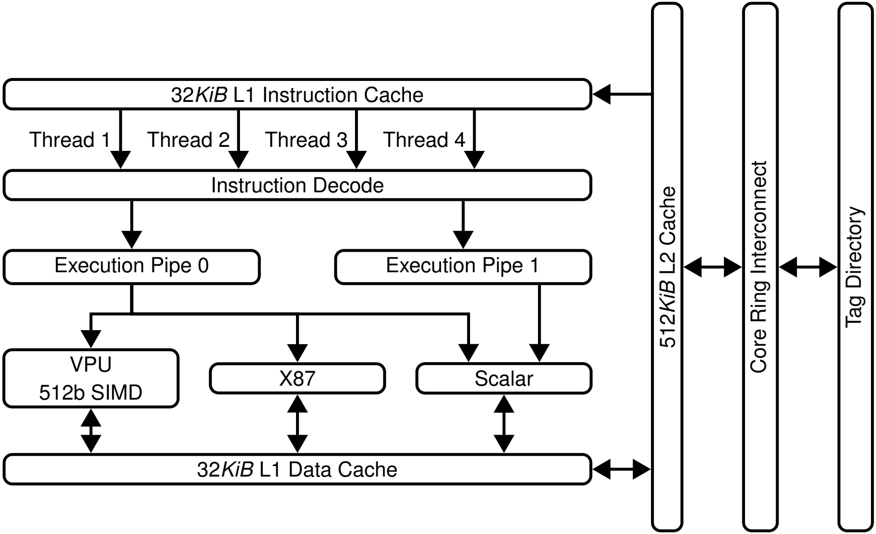
Karlsruher Institut für Technologie

- Complex algorithms and their framework/libraries ported to Xeon Phi
- Good scaling in both algorithms on Xeon Phi
- Offloading large amounts of data too expensive
- Graph drawing algorithm outperforms two-socket Intel Xeon system, especially on sparse graphs
- Future Research: Direct comparison between graph algorithms on GPU and Xeon Phi
- New Xeon Phi “Knights Landing” this year with modern cores and 384 GB of memory

[Appendix] Xeon Phi: Layout Overview



[Appendix] Xeon Phi: Core Pipeline



[Appendix] Xeon Phi: Programming

Karlsruher Institut für Technologie

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods:
  - OpenMP
  - OpenMP Offloading
  - Cilk Plus
  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods

[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods:
  - OpenMP

    ```
    float a[MAX], b[MAX], c[MAX];

    for (i = 0; i < MAX; i++)
      c[i] = a[i] + b[i];
    ```

  - OpenMP Offloading
  - Cilk Plus
  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods

[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods:
  - OpenMP

    ```
    float a[MAX], b[MAX], c[MAX];
    #pragma omp parallel for
    for (i = 0; i < MAX; i++)
      c[i] = a[i] + b[i];
    ```

  - OpenMP Offloading
  - Cilk Plus
  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods

[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods:
  - OpenMP
  - OpenMP Offloading

```
float a[MAX], b[MAX], c[MAX];
#pragma offload target(mic) in(a, b) out(c)
{
  #pragma omp parallel for
  for (i = 0; i < MAX; i++)
    c[i] = a[i] + b[i];
}
```

  - Cilk Plus
  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods

[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods:
  - OpenMP
  - OpenMP Offloading

```
float a[MAX], b[MAX], c[MAX];
#pragma offload target(mic:0) in(a, b) out(c) signal(c)
{
  #pragma omp parallel for
  for (i = 0; i < MAX; i++)
    c[i] = a[i] + b[i];
}
#pragma offload_wait target(mic:0) wait(c)
```

  - Cilk Plus
  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods

[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods:
  - OpenMP
  - OpenMP Offloading
  - Cilk Plus

```
float a[MAX], b[MAX], c[MAX];
cilk_for (i = 0; i < MAX; i++)
  c[i] = a[i] + b[i];
```

  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods

[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods:
  - OpenMP
  - OpenMP Offloading
  - Cilk Plus
  - Threading Building Blocks

    ```
    float a[MAX], b[MAX], c[MAX];
    parallel_for(size_t(0), MAX, size_t(1), [=](size_t i) {
      c[i] = a[i] + b[i];
    });
    ```
  - MPI
- Vectorization (Single Instruction Multiple Data) methods

[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization: `c = _mm512_add_ps(a, b)`

```c
#include <immintrin.h>
float a[MAX] __attribute__((aligned(64)));
float b[MAX] __attribute__((aligned(64)));
float c[MAX] __attribute__((aligned(64)));
__m512 _a, _b, _c;
for (i = 0; i < MAX; i += 16) {
  _a = _mm512_load_ps(&a[i]);
  _b = _mm512_load_ps(&b[i]);
  _c = _mm512_add_ps(_a, _b);
  _mm512_store_ps(&c[i], _c);
}
```

  - Auto-Vectorization
  - Cilk Plus

[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization
  - Auto-Vectorization

```
float a[MAX] __attribute__((aligned(64)));
float b[MAX] __attribute__((aligned(64)));
float c[MAX] __attribute__((aligned(64)));
#pragma simd
for (i = 0; i < MAX; i++)
  c[i] = a[i] + b[i];
```

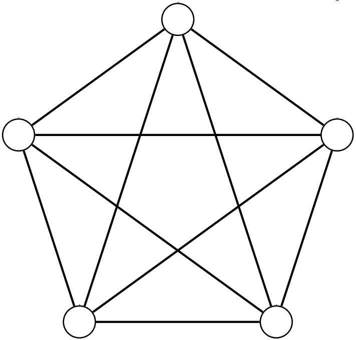
  - Cilk Plus

[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization
  - Auto-Vectorization
  - Cilk Plus

```
float a[MAX] __attribute__((aligned(64)));
float b[MAX] __attribute__((aligned(64)));
float c[MAX] __attribute__((aligned(64)));
c[i:MAX] = a[i:MAX] + b[i:MAX];
```

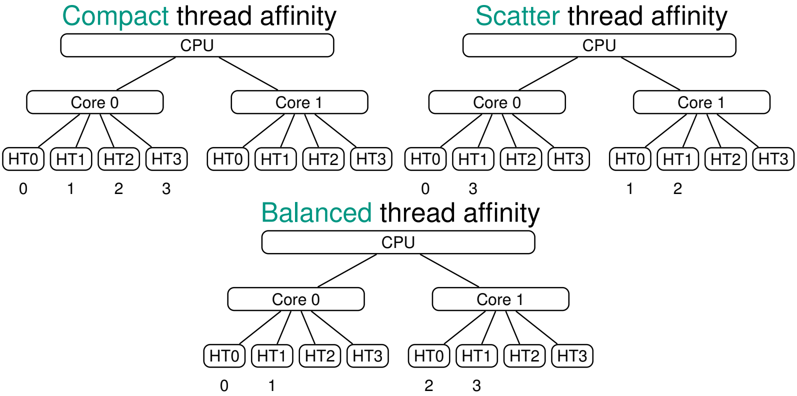
[Appendix] Preliminaries: Graphs



- Graph $G = (V, E)$ consists of
  - Set of vertices V
  - Set of edges $E \subseteq V \times V$
- Edge $e = (u, v) \in E$: connection from source u to target v
- Number of vertices $n = |V|$
- Number of edges $m = |E|$
- Undirected graphs only: $(u, v) \in E$ iff $(v, u) \in E$
- Neighborhood $N(u) = \{v : (u, v) \in E\}$
- Loop $(u, u) \in E$
- Degree $deg(v)$: number of incident edges, counting loops twice
- Distance: number of edges in shortest path connecting vertices
- Diameter d: greatest distance between any pair of vertices

[Appendix] Network Generation: Results

- OpenMP Scheduling:



⇒ Scatter and balanced 1.4 times faster