Pacific Northwest
NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965

Accelerating Graph Algorithms on Emerging Architectures

ANTONINO TUMEO AND MAHANTESH HALAPPANAVAR

Pacific Northwest National Laboratory

Overview

- Introduction
- Graph Matching
  - ½ approx matching
  - Implementations and results
- Community Detection
  - the Louvain method
  - Implementations and results
- Conclusions

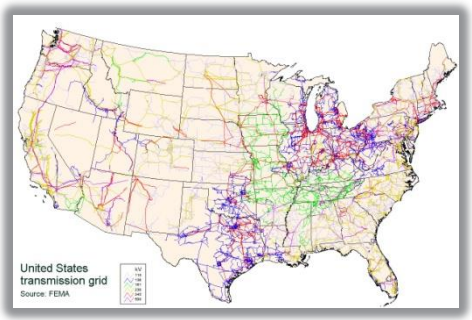
Applicability of Graph Algorithms

Big Science

Bioinformatics

Community Detection

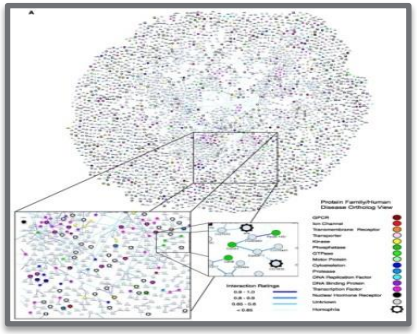
Graph Databases



Complex Networks

Knowledge Discovery

Language Understanding



Pattern Recognition

Graph algorithms as Irregular Kernels

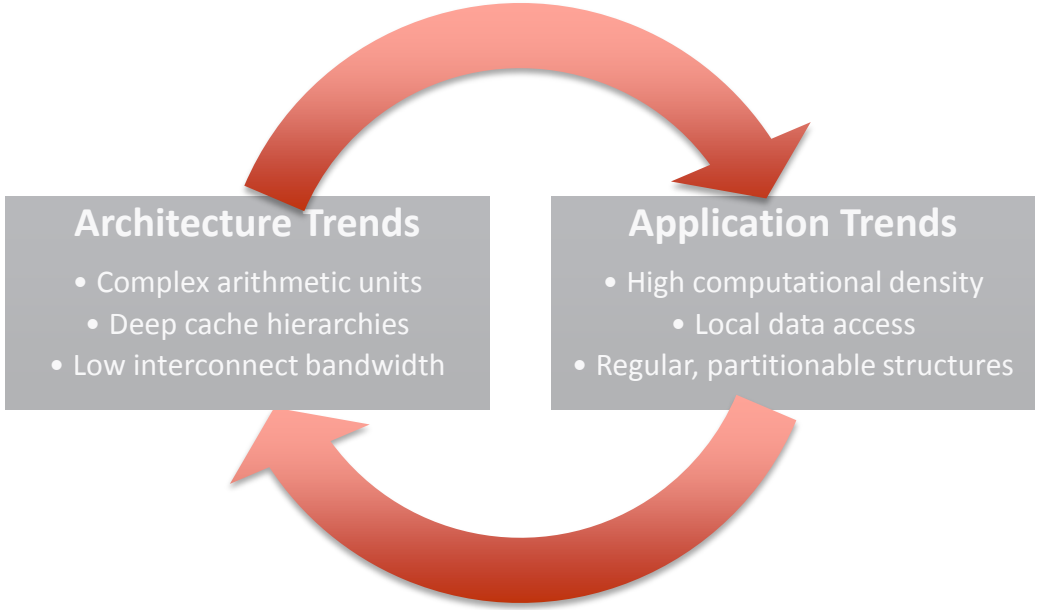
- Irregularity in data structures
  - Pointer- or linked-list based data structures very poor spatial and temporal locality
    - Unpredictable data accesses
    - Fine grained data accesses
- Irregularity in control
  - Divergent branches
    - If (vertex==x) do a; else do b
    - Foreach vertex v explore neighbors of v
- Irregularity in communication patterns
  - Unpredictable and fine grained communication
  - A consequence of irregularity in data structures and in control

Additional Characteristics

- Some datasets may be very large
  - Way more than what is currently available in a single cluster node
    - …and, obviously, a GPU
  - Very difficult to partition in a balanced way
- Large amounts of parallelism (e.g., each vertex, each edge in the graph)
- Usually, high synchronization intensity
  - Concurrent activities accessing the same elements of the data structures
- Datasets may be dynamically updated

Self-reinforcing Trend of FLOP-computing

- The HPC community builds systems for scientific simulations.



- We need systems for *data analysis, discovery, and inferencing.*

ARE THERE ARCHITECTURES MORE AMENABLE TO THESE WORKLOADS? WHAT DO WE REALLY NEED?

Breadth First Search

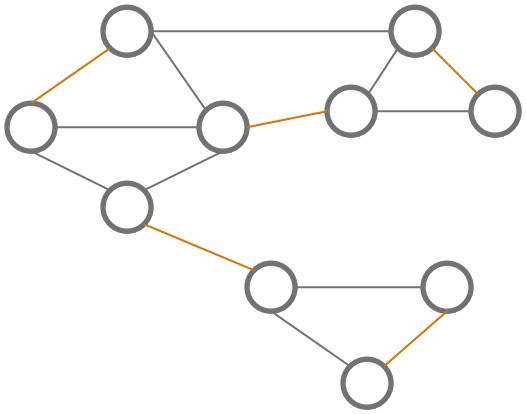
- Prototypical kernel for graphs
- Focus of the Graph500

- "You have to solve the BFS before looking at the other algorithms"

- True, but:
  - Graph algorithms may have different communication/computation balance than BFS
  - Over-optimizing for a single benchmark is not always good
  - Domain scientists still want to look at more efficient ways (i.e., lower computational complexity) to solve graph problems

½ Approximate Weighted Matching

- Problem: Given a graph G = (V,E) find a set (M) of pairwise non-adjacent edges
  - i.e., no two edges share a common vertex



- Maximum matching: maximizes cardinality (number of matched edges)
- Maximum weighted matching: maximizes sum of the weights of the matched edges
- Half-approximate weighted matching: guarantees that the sum of weights of matched edges is at least half of the optimal solution

Locally dominant algorithm for half-approximate weighted matching

Pacific Northwest NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965

```
1:  procedure SERIAL-QUEUE
      (G(V, E), mate)
2:    for each u ∈ V do                          ▷Initialization
3:      mate[u] ← ∅
4:      candidate[u] ← ∅
5:    Q ← ∅
6:    for each u ∈ V do                          ▷Phase 1
7:      PROCESS VERTEX(u, Q)
8:    while Q ≠ ∅ do                             ▷ Phase 2
9:      u ← FRONT(Q)
10:     Q ← Q\{u}
11:     for each v ∈ adj(u) \ mate(u) do
12:       if candidate[v] = u then                 ▷Process v only if u is
                                                     its candidate mate
13:         PROCESSVERTEX(v, Q)
```

Manne and Bisseling (2007)

ProcessVertex


```
1:  procedure ProcessVertex (s, Q)
2:      max_wt ← −∞
3:      max_wt_id ← ∅
4:      for each t ∈ adj(s) do
5:          if (mate[t] = ∅) AND                    ▷ Use vertex i.d. to
               (max_wt < w(e_{s,t})) then              break ties
6:              max_wt ← w(e_{s,t})
7:              max_wt_id ← t
8:      candidate[s] ← max_wt_id
9:      if candidate[candidate[s]] = s              ▷ Found a locally
           then                                        dominant edge
10:         mate[s] ← candidate[s]
11:         mate[candidate[s]] ← s
12:         Q ← Q ∪ {s,candidate[s]}                 ▷Only need to process
                                                       vertices in the queue.
```

Manne and Bisseling (2007)

Parallel implementation


```
1:  procedure PARALLEL-QUEUE(G(V, E),
    mate)
2:    Q_C ← ∅
3:    Q_N ← ∅
4:    — — Phase-1 — —
5:    for each u ∈ V in parallel do        ▷OpenMP pragmas/
                                            CUDA kernels
6:      mate[u] ← ∅
7:      candidate[u] ← ∅
8:      max_wt ← −∞
9:      max_wt_id ← ∅
10:     for each v ∈ adj(u) do
11:       if (mate[v] = ∅) AND (max_wt      ▷Use vertex i.d.s to
            < w(e_{u,v})) then                break ties.
12:         max_wt ← w(e_{u,v})
13:         max_wt_id ← v
14:     candidate[u] ← max_wt_id
15:   for each u ∈ V in parallel do        ▷OpenMP pragmas/
                                            CUDA kernels
16:     if candidate[candidate[u]] = u      ▷Found a locally
          then                              dominant edge
17:       mate[u] ← candidate[u]
18:       Q_C ← Q_C ∪ {u}                   ▷Use atomic memory
                                            operation
```

Halappanavar et. al., “Approximate weighted matching on emerging manycore and multithreaded architectures“, IJHPCA 2012

Parallel implementation (2)

20: — — **Phase-2** — —
21: **while** $Q_C \neq \emptyset$ **do**
22: **for each** $u \in Q_C$ in `parallel` do ▷OpenMP pragmas/ CUDA kernels
23: **for each** $v \in adj(u) \setminus$ **do**
24: **if** candidate[v] $= u$ **then** ▷Process v only if u is its candidate mate
25: PROCESSVERTEX(v, Q_N)
26: $Q_C \leftarrow Q_N$ ▷The new set of matched vertices
27: $Q_N \leftarrow \emptyset$

Halappanavar et. al., "Approximate weighted matching on emerging manycore and multithreaded architectures", IJHPCA 2012

Challenges for GPU implementation

- Load balancing within a warp: thread processing the vertex with the largest degree is the slowest
  - Possible solution: block work into equally sized groups, and use different threads for blocks
  - Works for well balanced inputs and BFS, not suitable for our matching algorithm that needs synchronization between threads that process adjacency list of a vertex
- Coalescing memory accesses: vertices added in random order to the queue
  - Possible solution: reordering queues
  - May not be computationally cheap…
- Branch divergence: *if* statements in the algorithm
  - Possible solution: restructure the algorithm
  - Not trivial…

A very diverse set of platforms…

Pacific Northwest NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965

Platform:	Opteron 6176 SE (Magny-Cours)	Xeon E5540 (Nehalem)	Tesla C1060 (Tesla)	Tesla C2050 (Fermi)	ThreadStorm-2 (XMT)
Processing units					
Clock (GHz)	2.30	2.53 (base)	1.3	1.15	0.5
Sockets	4	2	30 SMs	14 SMs	128
Cores/socket	12	4	8 SPs/SM	32 SPs/SM	1
Threads/core	1	2	–	–	128
Total threads	48	16	240 SPs	448 SPs	16,384
Interconnect	HyperTransport-3	QPI	PCIe2	PCIe2	Seastar2
Interconnect (GB/s)	25.6	25.6	8	8	4.8
Memory system					
Cache structure	L1/L2/L3†	L1/L2/L3†	–	L1/L2†	cache-less
L1 (KB)/core: Inst/Data	64/64	32/32	–	48/SM*	–
L2 (KB)/core	512	256	–	768/SM	–
L3 (MB)/socket	12	8	–	–	–
Memory/socket (GB)	64	12	4	3	8
Total memory (GB)	256	24	3	3	1,024
Peak bandwidth (GB/s)	42.7 (DDR3)	25.6 (DDR3)	102 (GDDR3)	144 (GDDR5)	86.4 (DDR1)
Software					
C Compiler	GCC 4.1.2	Intel 11.1	NVCC (CUDA 4.0)	NVCC (CUDA 4.0)	Cray C 6.5.0
Flags	-O3	-fast	-O3	-O3	-par
Thread scheduling	static	static	dynamic	dynamic	block-dynamic
Reference	A	B	C	D	E

Cray XMT and Extended Memory Semantics…

- The power of full/empty bits:
  - purge: sets the full/empty bit to empty and the value to zero;
  - readff: reads a memory location only when the full/ empty bit is full and leaves the bit full when read finishes;
  - readfe: reads a memory location only when the full/ empty bit is full and leaves the bit empty when read finishes;
  - writeef: writes to a memory location only if the full/empty bit is empty, and flips the bit to full when the write finishes.

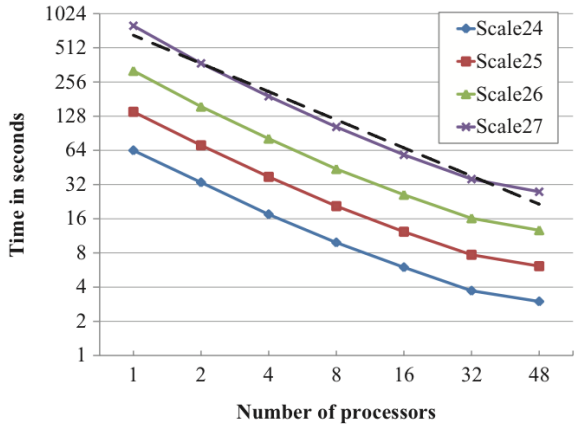
The XMT implementation


```
1:  procedure PARALLEL-DATAFLOW
        (G(V, E), mate)
2:      for each u ∈ V in parallel do
3:          mate[u] ← ∅
4:          state[u] ← 0
5:      for each e_{u,v} ∈ E in parallel do
6:          writeef(candidate [u → v], 0)      ▷ Set full/empty
                                                  bit to empty and
                                                  value to zero
7:          writeef(candidate [v → u], 0)
8:      for each u ∈ V in parallel do
9:          Processed ← int_fetch_add
                (state[u], 1)
10:         if Processed = 0 then              ▷ Process only if u
                                                  has not been
                                                  processed
                                                  before
11:             PROCESSVERTEXDF(u)
```

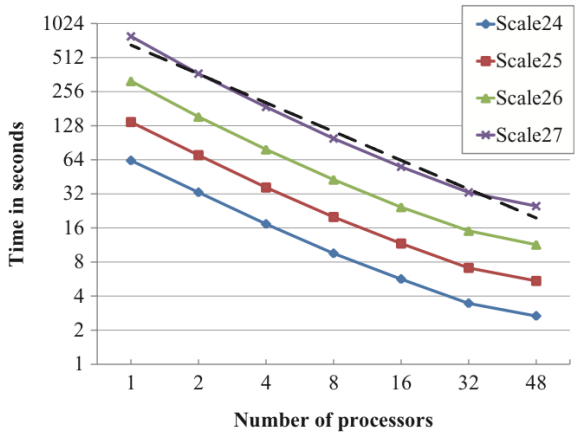
ProcessVertexDF


```
1:  procedure ProcessVertexDF(s)
2:      for each t ∈ adj(s) in non-
          increasing order of weights do
3:        writeef(candidate                ▷ Set full/empty bit to
            [s → t], 1)                      full and value to 1
4:        if int_fetch_add
            (state[t], 1) = 0 then
5:          ProcessVertexDF(t)             ▷ Proactively process
                                               the other end
6:        if readff (candidate             ▷ Wait until full/empty
            [t → s])=1 then                  bit becomes full
7:          mate[s] ← t                    ▷Found a locally
                                               dominant edge
8:          mate[t] ← s
9:          break
10:     for each t_np ∈ adj(s) not           ▷ Set the value to
          already processed do               zero for remaining
                                               neighbors of s
11:       candidate[s → t_np] ← 0
```

Magny Cours



(a) RMAT-ER

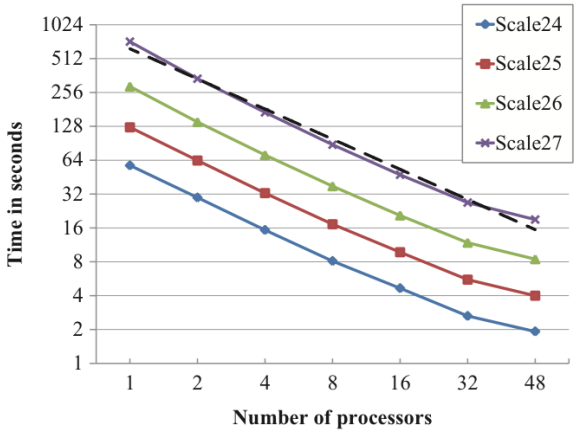


(b) RMAT-G

RMAT- ER: (0.25, 0.25, 0.25, 0.25)
RMAT-G: (0.45, 0.15, 0.15, 0.25)
RMAT-B: (0.55, 0.15, 0.15, 0.15)

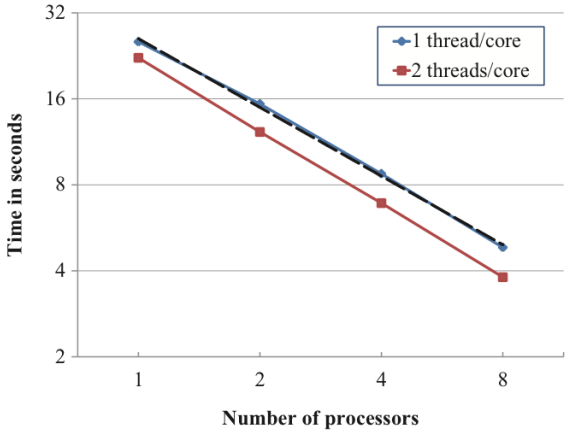
RMAT-ER: Erdös–Rényi random graphs.

RMAT-G and RMAT-B: real-world graphs with skewed normal distributions for vertex degrees and small-world phenomenon of short average distance between pairs of vertices.

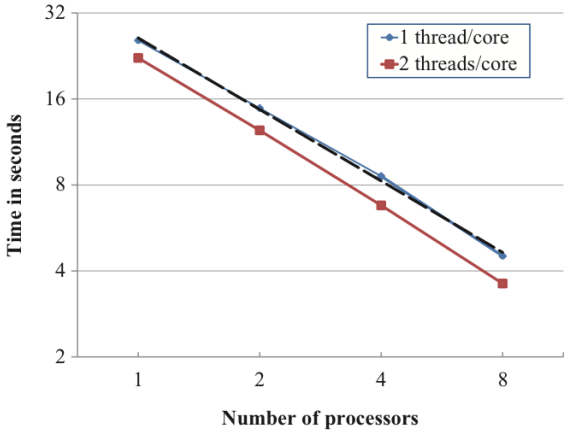


(c) RMAT-B

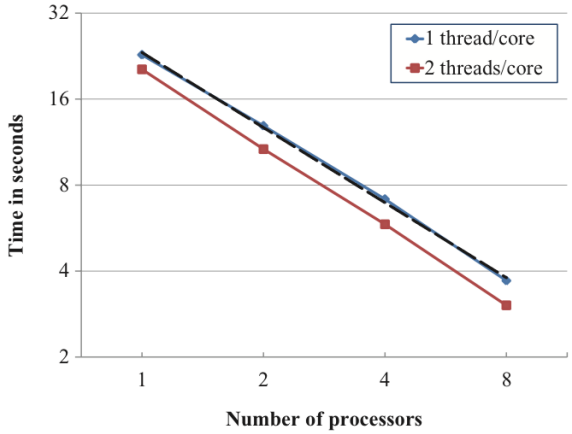
Nehalem



(a) RMAT-ER

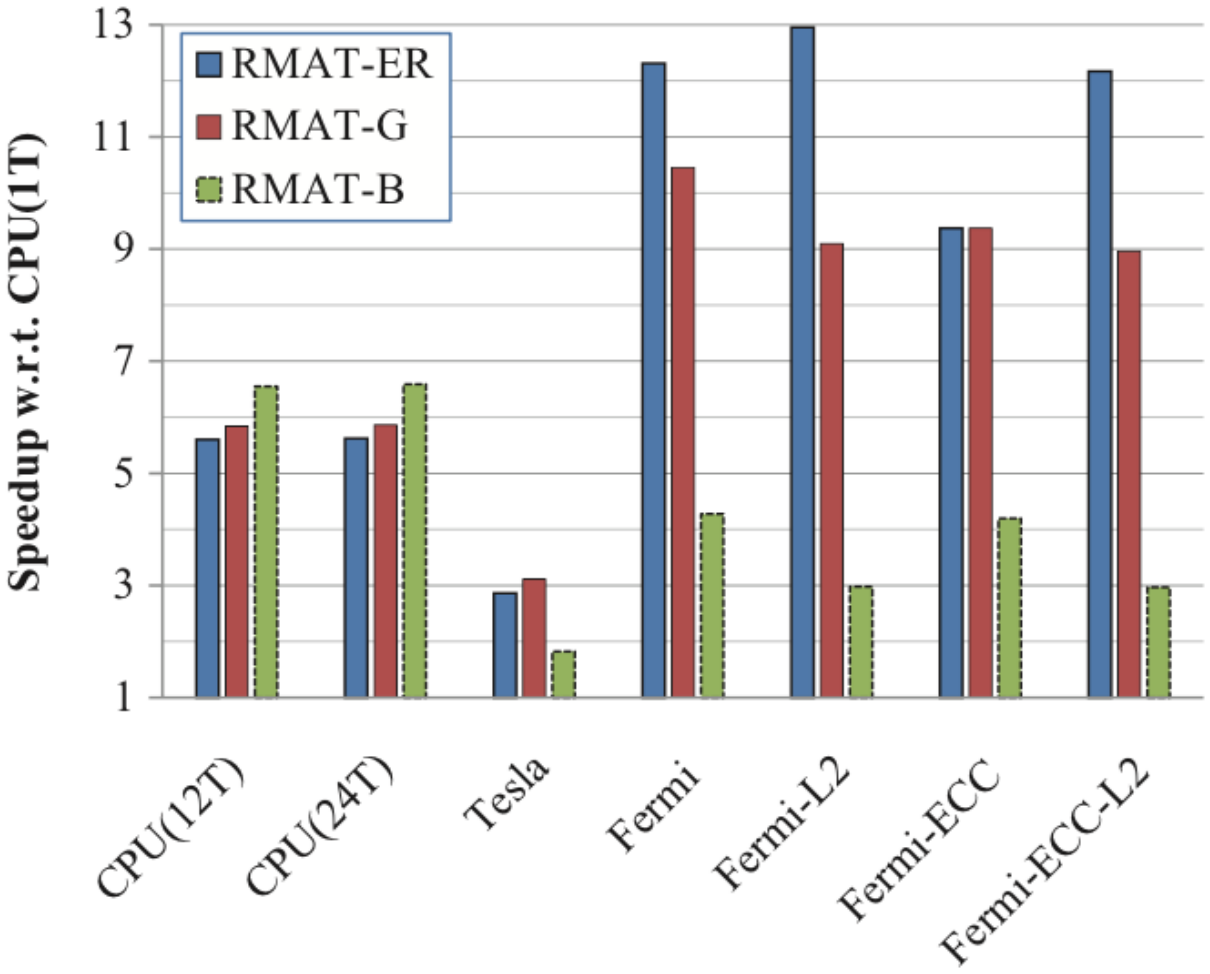


(b) RMAT-G

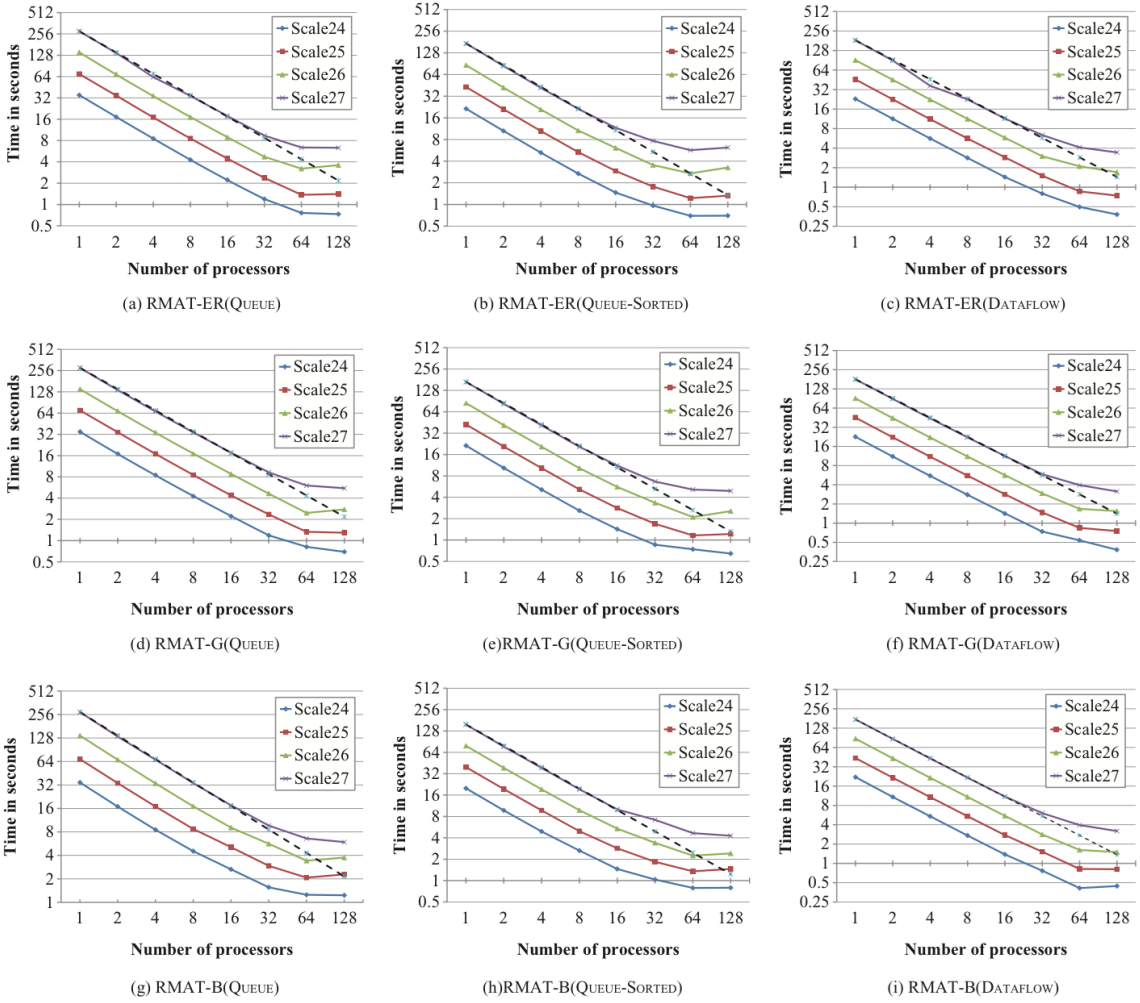


(c) RMAT-B

Tesla & Fermi (SCALE 23)



Cray XMT



(a) RMAT-ER(QUEUE) (b) RMAT-ER(QUEUE-SORTED) (c) RMAT-ER(DATAFLOW)

(d) RMAT-G(QUEUE) (e)RMAT-G(QUEUE-SORTED) (f) RMAT-G(DATAFLOW)

(g) RMAT-B(QUEUE) (h)RMAT-B(QUEUE-SORTED) (i) RMAT-B(DATAFLOW)

The Suitor Algorithm

Algorithm 2 The Sequential Suitor Algorithm

```
1: for each u ∈ V do
2:     suitor(u) = NULL
3:     ws(u) = 0
4: for each u ∈ V do
5:     current = u
6:     done = False
7:     while not done do
8:         partner = suitor(current)
9:         heaviest = ws(current)
10:        for each v ∈ N(current) do
11:            if w(current, v) > heaviest and
               w(current, v) > ws(v) then
12:                partner = v
13:                heaviest = w(current, v)
14:        done = True
15:        if heaviest > 0 then
16:            y = suitor(partner)
17:            suitor(partner) = current
18:            ws(partner) = heaviest
19:            if y ≠ NULL then
20:                current = y
21:                done = False
```

Manne and Halappanavar, "New Effective Multithreaded Matching Algorithms", IPDPS 2014

Parallelizing the Suitor Algorithm

Algorithm 2 The Sequential Suitor Algorithm

```
1:  for each u ∈ V do
2:      suitor(u) = NULL
3:      ws(u) = 0
4:  for each u ∈ V do                                   <-- In parallel for each vertex
5:      current = u
6:      done = False
7:      while not done do
8:          partner = suitor(current)
9:          heaviest = ws(current)
10:         for each v ∈ N(current) do
11:             if w(current, v) > heaviest and         <-- if w(current,v) = ws(v) and
                w(current, v) > ws(v) then                  current < suitor(v) but line 17
12:                 partner = v                             executed -> could be different
13:                 heaviest = w(current, v)                vertices
14:         done = True
15:         if heaviest > 0 then                        <-- Lock(v) to execute test
16:             y = suitor(partner)                     <-- Lock partner
17:             suitor(partner) = current
18:             ws(partner) = heaviest
19:             if y ≠ NULL then
20:                 current = y
21:                 done = False                        <-- Unlock partner
```

Manne and Halappanavar, "New Effective Multithreaded Matching Algorithms", IPDPS 2014

GPU challenges

- A thread per vertex
  - Load imbalance
  - Locks!

- Can implement locks with compare-and-swap, but need to disalign threads in a warp
- Different chunking (e.g., assign a thread-block per vertex and evaluate neighbors in parallel)
  - Sizing of thread-blocks to maximize utilization
  - Redistribution of workload

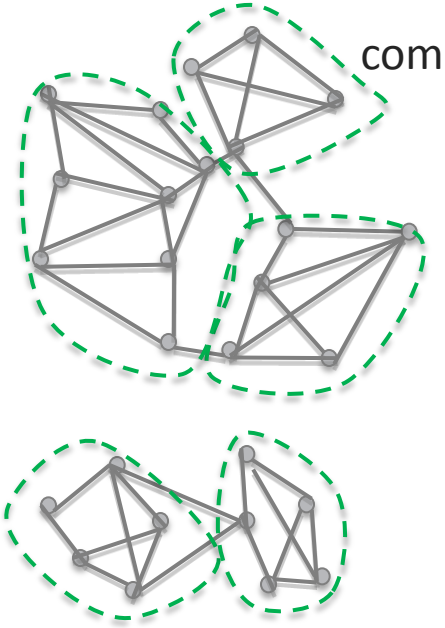
Preliminary results (seconds)

Graph	X86 - 8 threads	Tesla K40	Speed up
af_1_k101.mtx	60.457764	26.304256	2.3
af_2_k101.mtx	69.582275	25.323263	2.7
af_3_k101.mtx	67.461426	25.777952	2.6
af_4_k101.mtx	61.180908	29.906528	2.0
af_5_k101.mtx	61.199951	29.084831	2.1
pkustk10.mtx	13.904053	8.650304	1.6
pkustk11.mtx	15.187988	10.354752	1.4
pkustk12.mtx	20.045166	16.716225	1.2
pkustk14.mtx	36.321045	27.059135	1.3
roadNet-CA.mtx	107.801025	7.95504	13.5
roadNet-PA.mtx	75.349854	4.3848	17.2
roadNet-TX.mtx	95.695068	5.393344	17.7

Naim et al., in progress

Community Detection in Graphs

- Problem: Given $G(V,E,\omega)$, identify tightly knit groups of vertices that strongly correlate to one another within their group, and sparsely so, outside.



Input :

- $V = \{1,2,\ldots n\}$
- E: a set of M edges
- $\omega(e)$: weight of edge e (non-negative)
- $m = \sum_{\forall e \in E} \omega(e)$

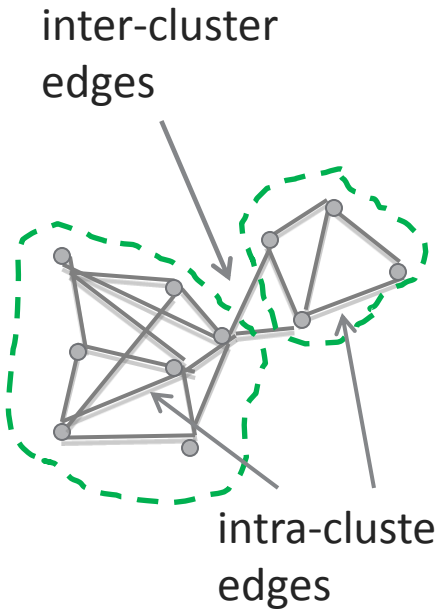
Output :

- A partitioning of V into k mutually disjoint communities $P = \{C_1, C_2, \ldots C_k\}$ such that modularity is optimized.

Numerous applications spanning various scientific and social domains of computing

Modularity (Newman 2004)

- A statistical measure for assessing the quality of a given community-wise partitioning P of the vertices V:



$$Q = \frac{1}{2m} \sum_{\forall i \in V} e_{i \to C(i)} - \sum_{\forall C \in P} \left(\frac{a_C}{2m} \cdot \frac{a_C}{2m} \right)$$

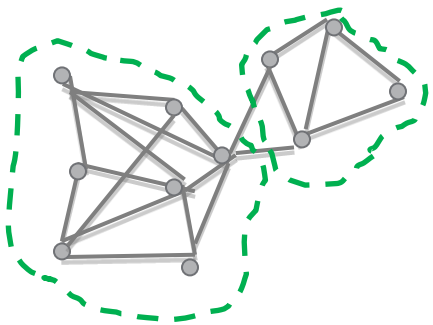
Fraction of intra-cluster edges

Equivalent fraction in a random graph

Community Detection

Modularity optimization (Newman & Girvan, 2004)

Compute a partitioning *P* of *V* into an arbitrary number (*k*) of clusters such that the *modularity score (Q)* of the partitioning is maximized.



$$Q = \frac{1}{2m} \sum_{\forall i \in V} e_{i \to C(i)} - \sum_{\forall C \in P} \left(\frac{a_C}{2m} \cdot \frac{a_C}{2m} \right)$$

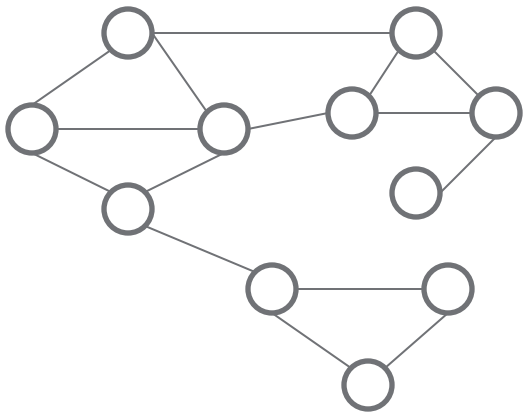
Q = 2(5+13)/40 - (28+12)/40*40
= 0.875

Notation	Definition
C(i)	Cluster containing vertex i
$e_{i->C(i)}$	Number of edges from i to vertices in C(i)
a_C	Sum of the degree of all vertices in cluster C

Modularity optimization is NP-Hard (Brandes et al. 2006).

Louvain method (Blondel et al. 2008)

Input: *G(V,E)*
Goal: *Compute a partitioning of V that maximizes modularity (Q)*
Init: *Every vertex starts in its own community (i.e., C(i)={i})*



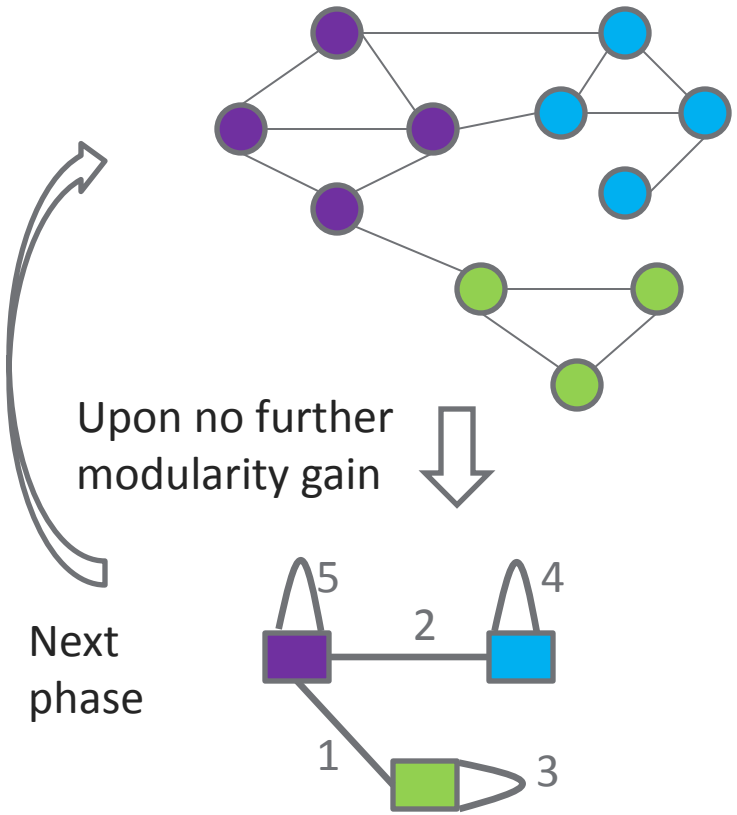
Multi-phase iterative heuristic

Within each iteration:

- **For** every vertex $i \in V$:
  1. Let $C(i)$: current community of i
  2. Compute modularity gain (ΔQ) for moving i into each of i's neighboring communities
  3. Let C_{max} : neighboring community with largest ΔQ
  4. If ($\Delta Q > 0$) { Set $C(i) = C_{max}$ }

Louvain method (Blondel et al. 2008)

Input: G(V,E)
Goal: Compute a partitioning of V that maximizes modularity (Q)
Init: Every vertex starts in its own community (i.e., *C(i)={i}*)



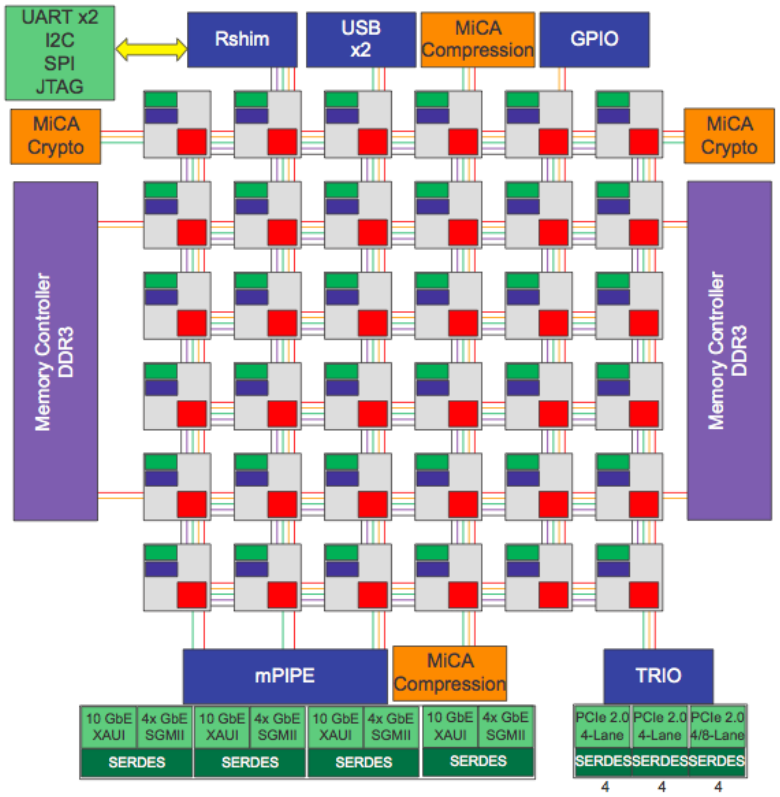
Multi-phase iterative heuristic

Within each iteration:

- **For** every vertex $i \in V$:
  1. Let $C(i)$: current community of i
  2. Compute modularity gain (ΔQ) for moving i into each of i's neighboring communities
  3. Let C_{max} : neighboring community with largest ΔQ
  4. If ($\Delta Q>0$) { Set $C(i) = C_{max}$ }

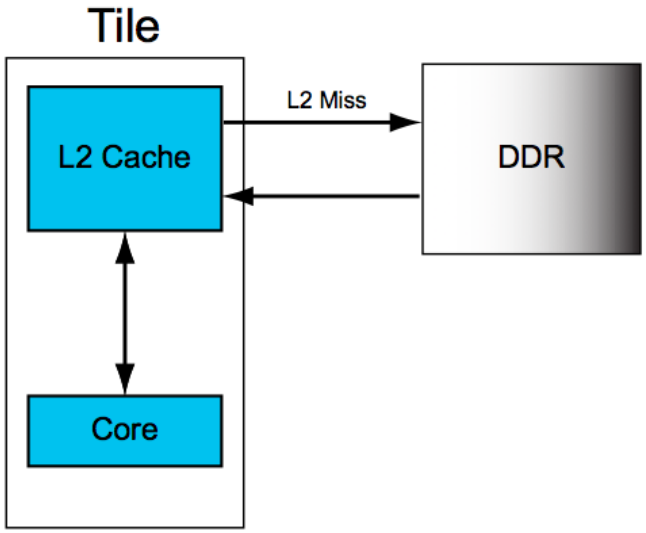
Another cool architecture…

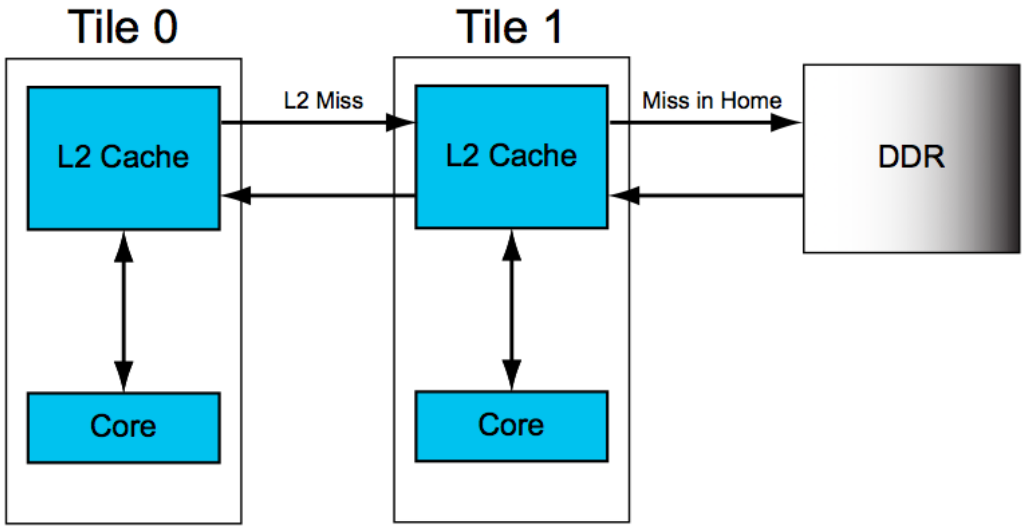
TILE-Gx 8036 processor



- 36 Tiles
- 6 NoCs
  - User Level, Static, I/O, Memory and Cache (3)
- Processor Engine:
  - VLIW – 64 bits – 1.2 GHz
- Cache Engine:
  - 32 KB L1, 256 KB L2
  - Dynamic Distributed L2 (Configurable homing for memory pages)
- Switch Engine:
  - Connects to the NoCs
- 2 DDR3 memory controllers
  - Configurable striping size
- HW accelerators
  - Cryptography, compression, packet inspection
- Power instrumentation
  - Outputs to the IPMI Voltage, Current, Power for processor and memory

Memory homing strategies

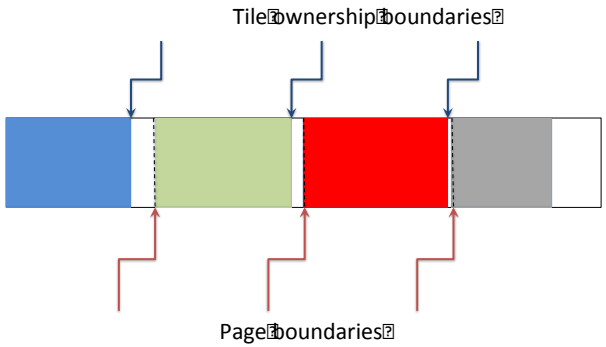




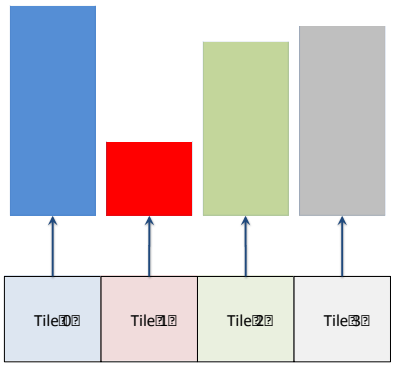
- ***Local homing***
  - Page homed in local cache
- ***Remote homing***
  - Page homed in remote cache
- ***Hash-for-home***
  - Each page has a line homed in different caches, using a round robin distribution

Memory layout schemes explored

Scheme	Description	Comments	Locality
Hashed	Memory distributed in round-robin blocks of 64 bytes across tiles	Limited data locality	Unaware
Local	Local homed pages used for private data, hashed for shared	Exploits locality where its clearly available	Partially-aware
Padded	Principal arrays partitioned into n/p chunks, private data is local	Chunks are rounded up to x64 KB page size, memory accesses are explicitly aligned to match page boundaries	Partially-aware
Partitioned	Vertex and edge lists are partitioned according to an external partitioner	Use PaToH and METIS as partitioners for initial graphs, local memory is used for each partition	Fully aware



Padded



Partitioned

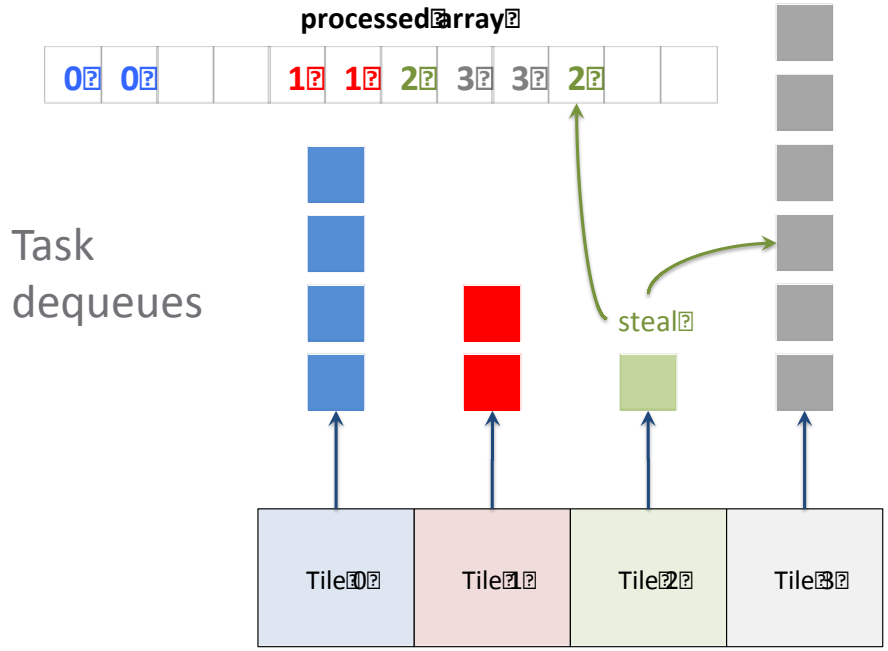
Chavarría et al., “Scaling Graph Detection on the Tilera Many-core Architecture”, HiPC 2014

Mapping the Louvain Method on Tilera: Task Parallelization

Pacific Northwest NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965

- Two schemes: Load balance-centric and locality-centric
- Must take into account varying costs of processing different vertices (depends on their neighborhood communities)
- **Load balance-centric:** use OpenMP `guided` scheduling, does not exploit locality information from the data layout
- **Locality-centric:** custom task-based work-stealing, execute local tasks first before load balancing across threads

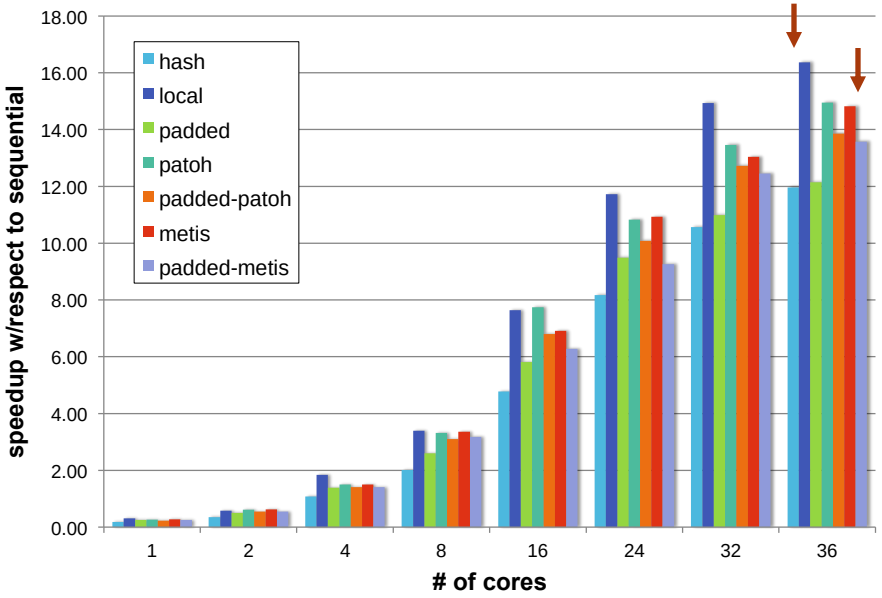
Locality-centric scheme w/ work-stealing



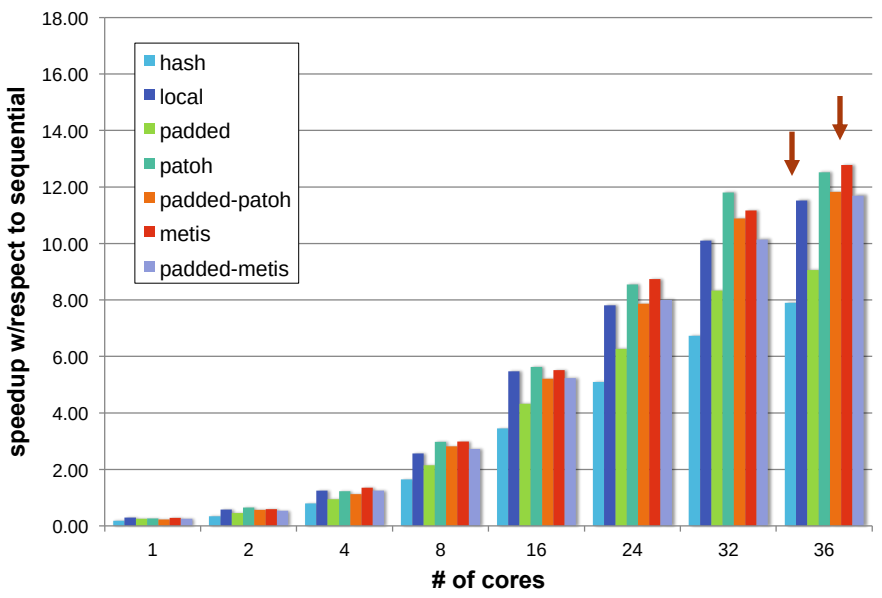
Chavarría et al., "Scaling Graph Detection on the Tilera Many-core Architecture", HiPC 2014

Results on Tilera

Channel – ***guided*** (load balance-centric)



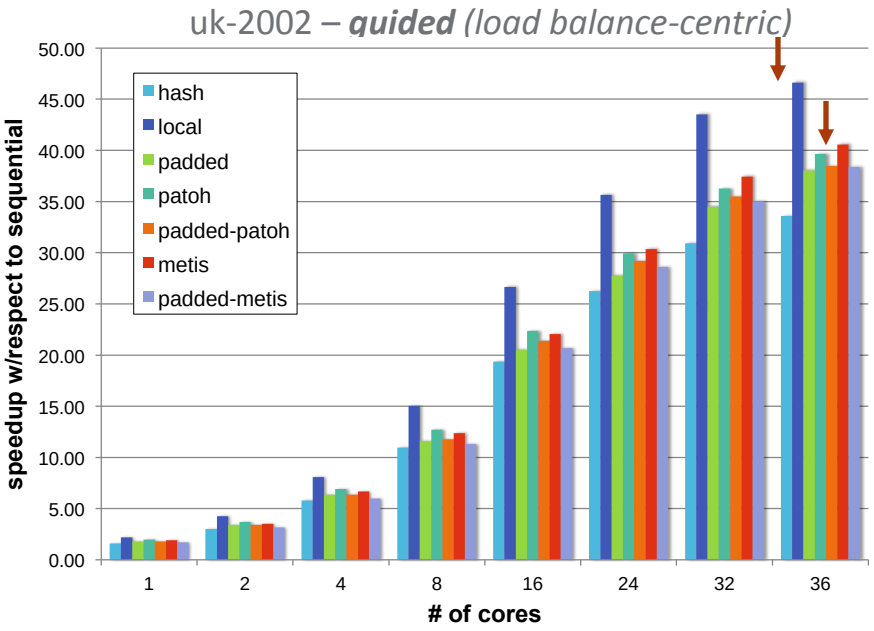
Channel – ***tasks*** (locality-centric)

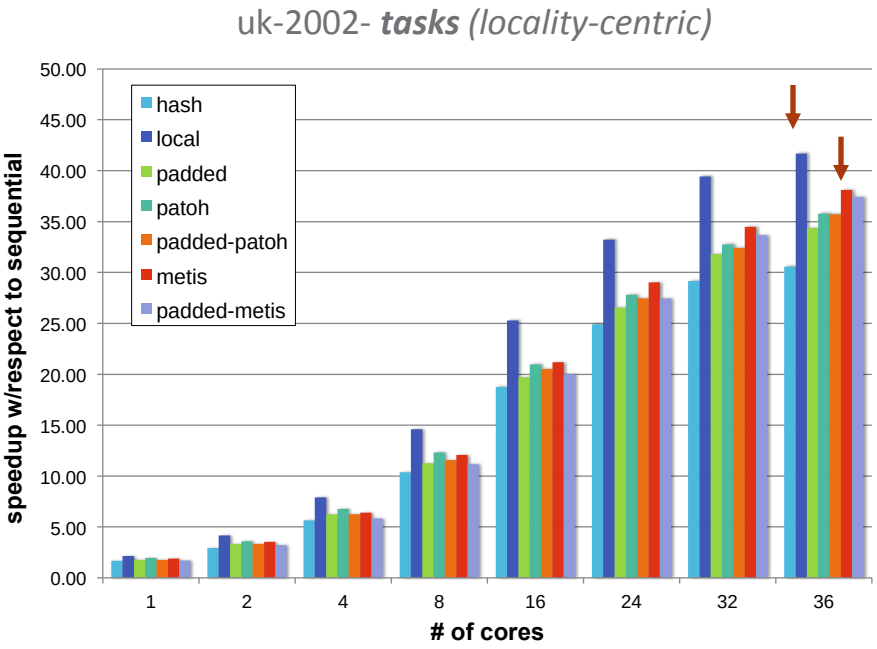


- The **local-guided** scheme performed the best (16x speedup)
- Task parallelism w/ work-stealing preserves locality better but also introduces more overheads in generating the local task queues
- The Channel input is a banded graph => slower convergence, more phases

Chavarría et al., "Scaling Graph Detection on the Tilera Many-core Architecture", HiPC 2014

Results on Tilera (2)





- The **local-guided** scheme performed the best (45x speedup)
- Task parallelism w/ work-stealing preserves locality beter but also introduces more overheads in generating the local task queues
- The uk-2002 input is power-law with a high community structure => faster convergence, lower parallel overhead

Chavarría et al., "Scaling Graph Detection on the Tilera Many-core Architecture", HiPC 2014

Run-time and Modularity Results: Tilera vs. x86

Pacific Northwest NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965

Input	Serial (Tilera)		Parallel (Tilera/36)		Serial (Intel)	Parallel (Intel/32)	
	Modularity	Time (s)	Modularity	Time (s)	Time (s)	Modularity	Time (s)
CNR	0.912784	37.89	0.912497	**0.87**	4.36	**0.912626**	1.25
Channel	0.849672	287.65	0.934461	**17.57**	30.92	**0.934671**	25.58
Europe	-	-	0.998843	335.00	-	**0.998846**	**163.03**
Uk-2002	0.989700	2340.16	0.989526	**50.20**	335.99	**0.989532**	52.18
MG2	0.998426	4011.61	0.998416	159.63	1313.74	**0.998426**	**101.96**

- Single core runs on Tilera were generally much slower than the corresponding x86 single core runs
- However, the timings on 36 Tilera cores were comparable and at times faster than on 32 x86 cores => better scaling
- Modularities achieved by both systems were highly comparable with the x86 figures marginally better than Tilera's

Chavarría et al., "Scaling Graph Detection on the Tilera Many-core Architecture", HiPC 2014

What about GPUs?

- Main challenge is finding neighboring communities of a vertex
  - Perform an initial search, generating a map (community, degree)
  - After, calculate gains in modularity by moving the vertex to each one of those communities
- Possible solution for GPUs
  - Instead of generating a map, iterate on each neighbor of a vertex
  - For each neighbor, check the other neighbors in the same community…
  - And calculate gain in modularity
    - Flops are cheap!
  - Store only the maximum
- Still refining the algorithm…

Conclusions

- Not only Breadth First Search
  - A reference kernel - for other algorithms to work, BFS must work first…
  - ..but there are other graph algorithms
- Matching and Graph Clustering
  - Relevant graph kernels for many computing problems
  - Lot of parallelism, but hard to implement on architectures optimized for regular computation…
  - …and scaling is even harder!
  - Discussed some of the algorithms we are exploring

Thank you for your attention!

Pacific Northwest NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965

- Questions?
  - antonino.tumeo@pnnl.gov
  - mahantesh.halappanavar@pnnl.gov
  - All our collaborators!

- Call for papers! - Special issue of the Journal of Parallel Computing (PARCO) on Theory and Practice of Irregular Applications (TaPIA)
  - http://www.journals.elsevier.com/parallel-computing/call-for-papers/parallel-computing-on-theory-and-practice-of-irregular-appli/
  - Deadline: March 30
  - Guest editors: Antonino Tumeo, Oreste Villa, John Feo