An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture

Erik Saule[1] and **Ümit V. Çatalyürek**[1,2]

{esaule,umit}@bmi.osu.edu

[1]Department of Biomedical Informatics
[2]Department of Electrical and Computer Engineering
The Ohio State University

MTAAP 2012

The Intel MIC Architecture

Features

- High Performance Computing with generic x86 cores.
- High core count.
- Large SIMD.
- Highly hyper-threaded.

The Knight Ferry prototype

32 cores (1 reserved for system purposes in our experiments).

The Knight Corner

50+ cores.

Graph Algorithms and Irregular Kernels

Many Applications where GPUs are holding back

Basically all applications based on indirection and pointer chasing: Sparse linear algebra (solvers, factorisation), Graph problem (Shortest Path, Travelling Salesman, Network Analysis), Text search (inexact pattern matching, indexing)

Graph Coloring

Given a graph, assign colors (integers) for each vertex so that two adjacent vertices have different colors.

Breadth First Search traversal

Given a graph and a particular vertex, build a list of all the vertices from the closest ones to the farthest ones.

Graph Algorithms and Irregular Kernels

Many Applications where GPUs are holding back

Basically all applications based on indirection and pointer chasing: Sparse linear algebra (solvers, factorisation), Graph problem (Shortest Path, Travelling Salesman, Network Analysis), Text search (inexact pattern matching, indexing)

Graph Coloring

Given a graph, assign colors (integers) for each vertex so that two adjacent vertices have different colors.

Breadth First Search traversal

Given a graph and a particular vertex, build a list of all the vertices from the closest ones to the farthest ones.

Programming Models

OpenMP

Pragma directives that allow parallel processing. Support for sections, locks, ...

Cilk Plus

Asynchronous function call powered by workstealing. Allows nested parallelism. Focus is on programmability by looking like sequential execution.

Intel TBB

Collection of tools for parallel processing. Object oriented programing paradigm. Versatile programming model supporting recursive decomposition, filter based parallel processing...

Outline

Speculative Coloring

- Each processor independently color some vertices.
- Conflicts might occur.
- They are detected in parallel; and some vertices are *uncolored*.
- The process repeats itself.

Algorithm 1: TentativeColoring

Data: $G = (V, E)$, Visit $\subset V$, color$[1 : |V|]$
maxcolor $\leftarrow 1$
localMC $\leftarrow 1$
for each $v \in$ Visit **in parallel do**
  for each $w \in adj(v)$ **do**
    localFC[color[w]] $\leftarrow v$
  color[v] $\leftarrow \min\{i > 0 : \text{localFC}[i] \neq v\}$
  if *color*[v] $>$ *localMC* **then**
    localMC $\leftarrow$ color[v]
maxcolor $\leftarrow$ **Reduce(max)** localMC
return maxcolor

Algorithm 2: DetectConflict

Data: $G = (V, E)$, Visit $\subset V$, color$[1 : |V|]$
Conflict $\leftarrow \emptyset$
for each $v \in$ Visit **in parallel do**
  for each $w \in adj(v)$ **do**
    if *color*[v] $=$ *color*[w] **then**
      if $v < w$ **then**
        atomic Conflict $\leftarrow$ Conflict $\cup\{v\}$
return Conflict

Variants

OpenMP

Implementation based on the *parallel for* construct. Three scheduling policies: **static**, **dynamic**, **guided**. Memory is allocated and indexed by threadIDs.

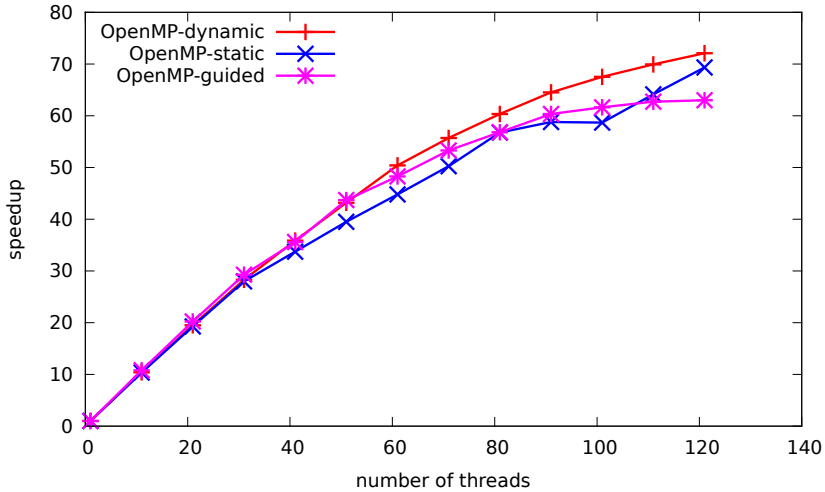
Cilk Plus

recursive decomposition of the iterations of the loop. Executed with workstealing. Allocating memory per thread is done by using Holders to allocate memory dynamically. Otherwise hack workerIDs and allocate memory first.

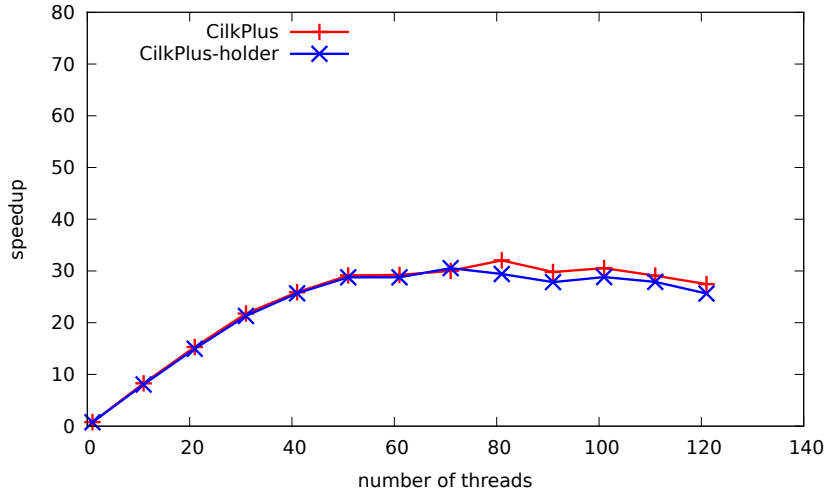
Intel TBB

tbb::parallel_for can use multiple types of partitioner: **simple** recursively divides the range up to a given size. **auto** uses workstealing event to decide when to stop. **affinity** tries to maximize cache reuse based on the index ordering.

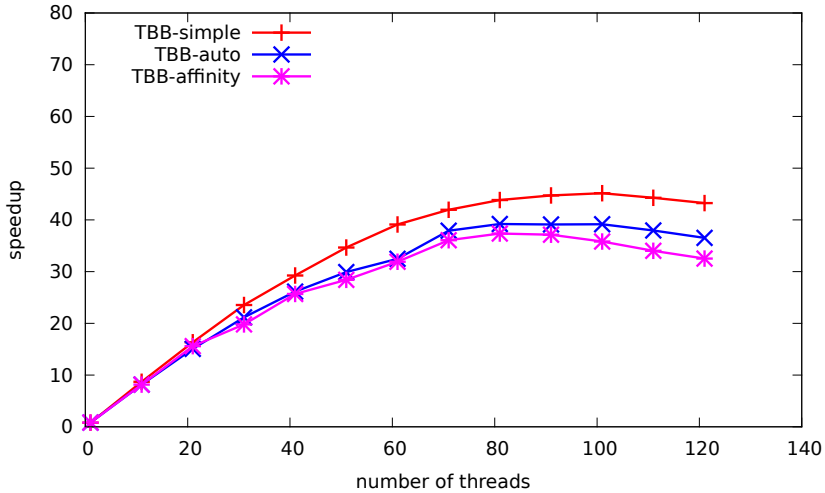
Experiments



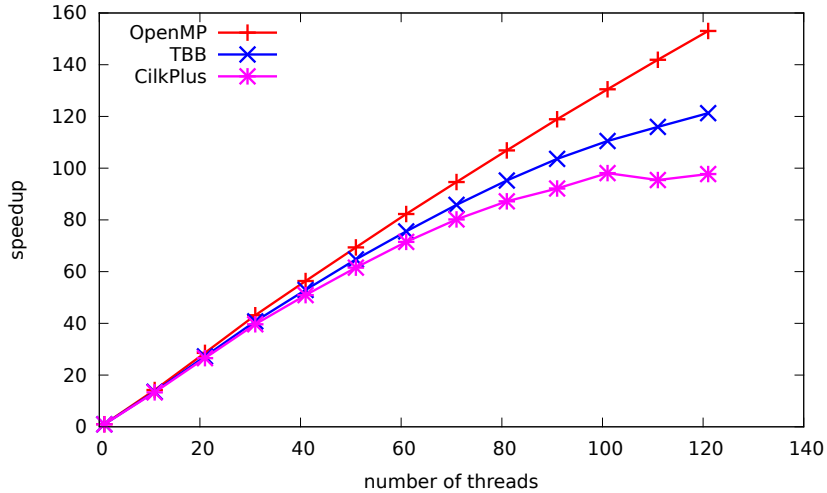
(a) OpenMP



(b) Cilk Plus



(c) TBB



(d) Randomly Ordered Graph

Outline

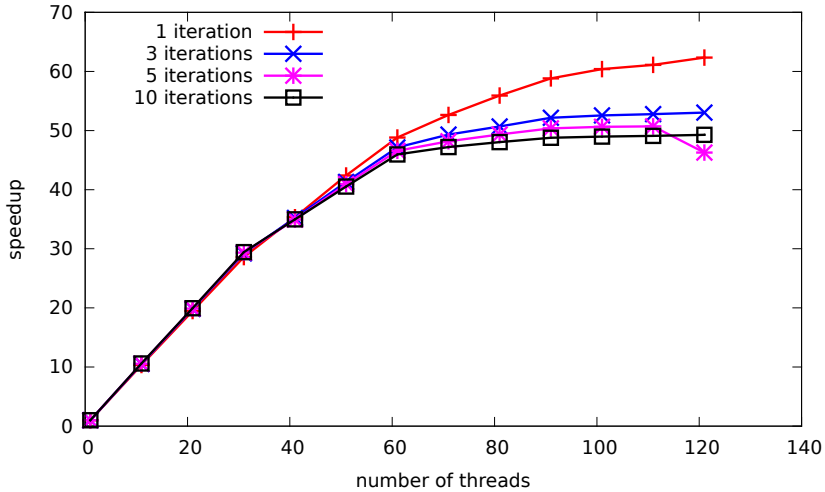
Loaded Computation

Algorithm 3: IrregularComputation

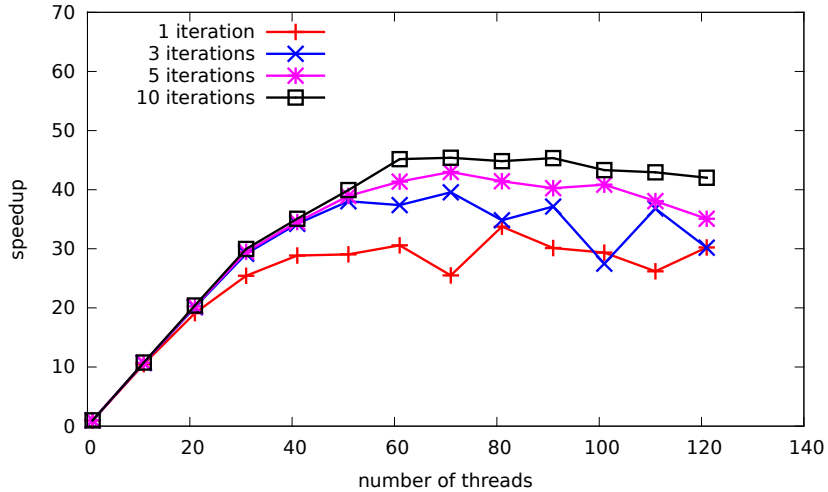
Data: $G = (V, E)$, $Visit \subset V$, state$[1 : |V|]$
for each $v \in V$ **in parallel do**
  for $i = 0$; $i < iter$; $i{+}{+}$ **do**
    sum $\leftarrow$ state$[v]$
    for each $w \in adj(v)$ **do**
      sum $\leftarrow$ sum + state$[w]$
    state$[v] \leftarrow \frac{\text{sum}}{|adj(v)+1|}$

Variants are the same that in speculative coloring. Allows to change the computation intensivity by tuning the number of iterations.

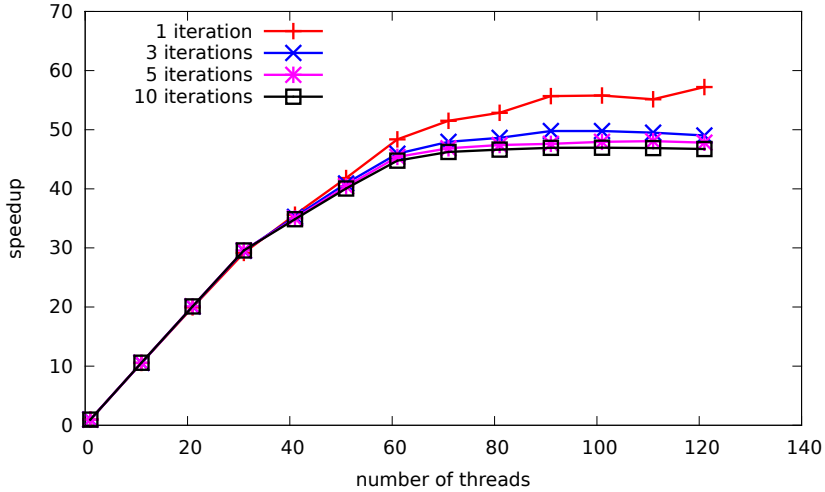
Experiments



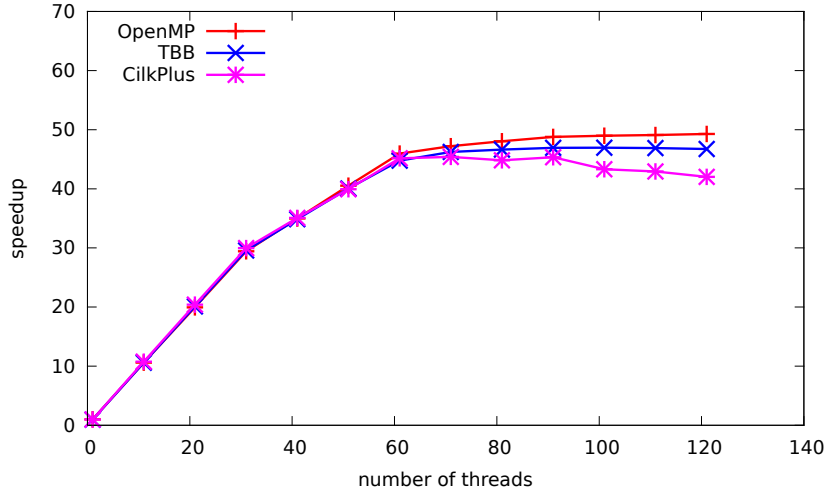
(e) Using OpenMP



(f) Using Cilk



(g) Using TBB



(h) 10 iterations

Outline

Parallel Layered Breadth First Search

Algorithm 4: ParLayeredBFS

```
Data: G = (V, E), source ∈ V
for v ∈ V in parallel do
    bfs[v] ← −1
bfs[source] ← 0
cur.add(source)
level ← 1
while ! cur.empty() do
    for v ∈ cur in parallel do
        for each w ∈ adj(v) in parallel do
            if bfs[w] = −1 then
                bfs[w] ← level
                uniquely next.add(w)
    SWAP (cur, next)
    level ← level + 1
return bfs
```

Sources of parallelism

- parallel vertex traversal
- parallel edge traversal (inefficient)

Synchronizations

- At the end of each level
- Management of `next`

Existing Implementations

Snap [BM08]: a queue based implementation with OpenMP

- Keeps a local queue per thread in its TLS.
- Merge the queues at the end of the level in $O(n)$.
- Locks the vertices to change their state (visited or not) to avoid a race condition which inserts in `next` the same vertex multiple times.

Leiserson and Schardl[LS10]: a bag based implementation with Cilk

Observed first that the race condition on `next` is harmless.

- A vertex can be added twice to the list.
- It increases the runtime but the algorithm stays correct.
- And it is unlikely to happen multiple times.

The method is designed to be used with a workstealing scheduler and represents `next` as a *Bag* of vertices. It is a data structure that supports split and merge operations in $O(\log n)$.

Existing Implementations

Snap [BM08]: a queue based implementation with OpenMP

- Keeps a local queue per thread in its TLS.
- Merge the queues at the end of the level in $O(n)$.
- Locks the vertices to change their state (visited or not) to avoid a race condition which inserts in `next` the same vertex multiple times.

Leiserson and Schardl[LS10]: a bag based implementation with Cilk

Observed first that the race condition on `next` is harmless.

- A vertex can be added twice to the list.
- It increases the runtime but the algorithm stays correct.
- And it is unlikely to happen multiple times.

The method is designed to be used with a workstealing scheduler and represents `next` as a *Bag* of vertices. It is a data structure that supports split and merge operations in $O(\log n)$.

A blocked queue based implementation

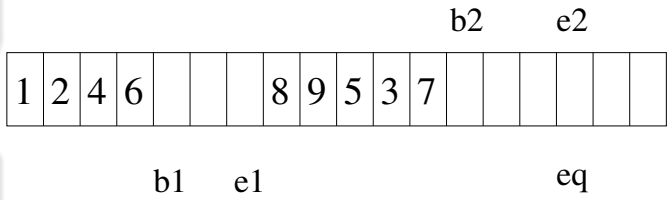
Goals

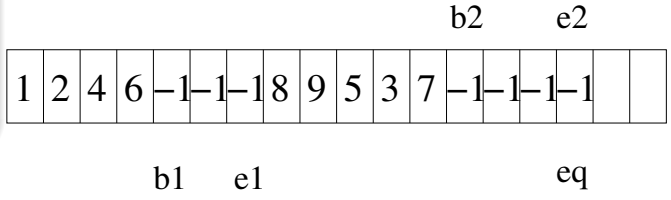
- Scheduling overhead in $O(1)$: no bags.
- End of level overhead in $O(1)$: no merge.

Blocked Queues

The threads:

- fill up *concurrently* a single queue.
- reserve a part with an atomic operation.
- fill up the block with a sentinel value at the end of the level.





Variants

Implemented in OpenMP and TBB. With or without locks on `next`.

A performance model

Observation

The parallelism of the algorithm depends on the shape of the graph. If the graph is a chain, there is no parallelism. Moreover, every single vertex can not be scheduled independently.

Assumptions

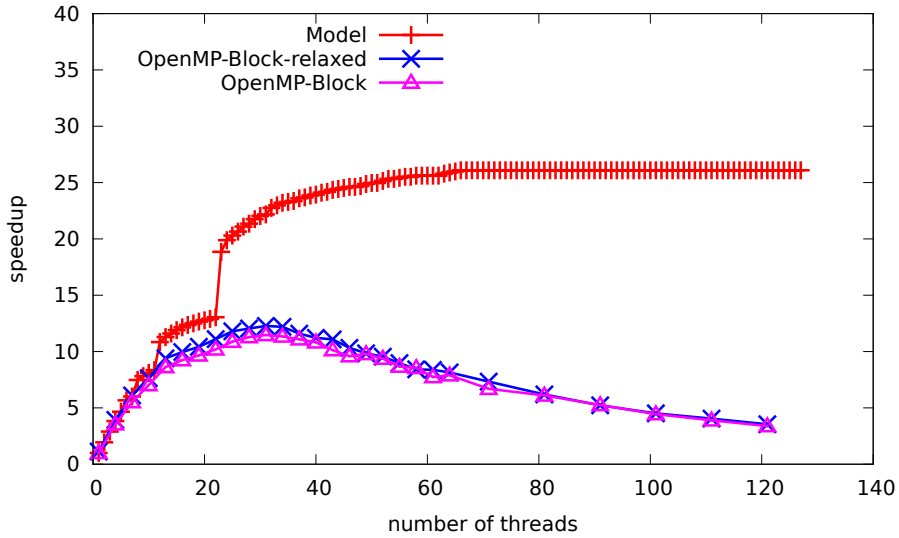
There is a synchronization at the end of each level. There are x_l vertices in level l. t threads are used. Computations are performed by blocks of b vertices. Processing each vertex takes the same time. No other scheduling overhead.

Model

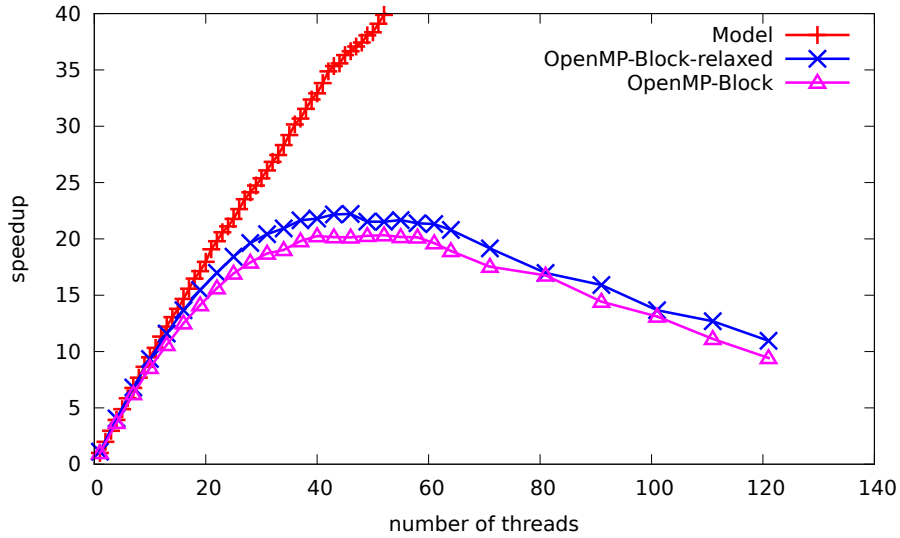
The computation time of level l is then: $c(l) = x_l$ if $x_l < b$ and $c(l) = \left\lceil \frac{x_l}{tb} \right\rceil * b$ otherwise.

Maximum speedup: $\frac{\sum_{l=1}^{L} x_l}{\sum_{l=1}^{L} c(l)}$.

Impact of the synchronizations

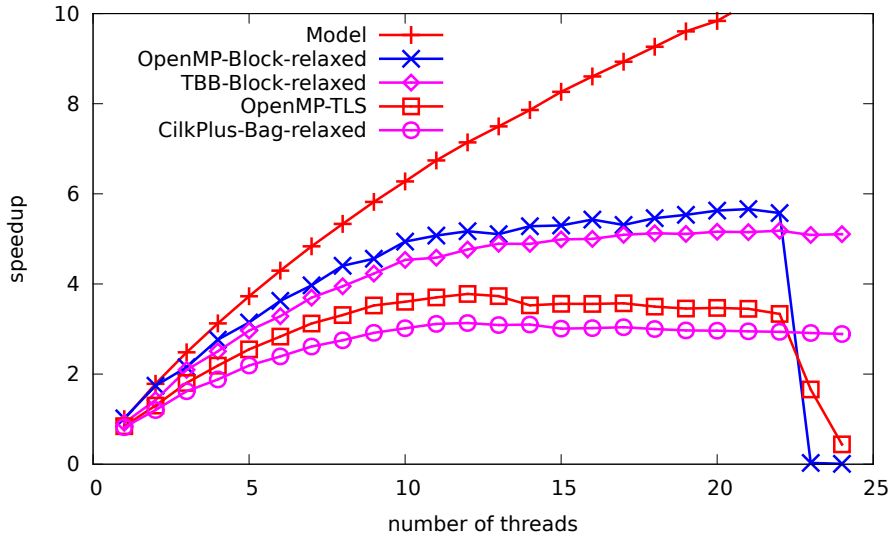


(i) pwtk

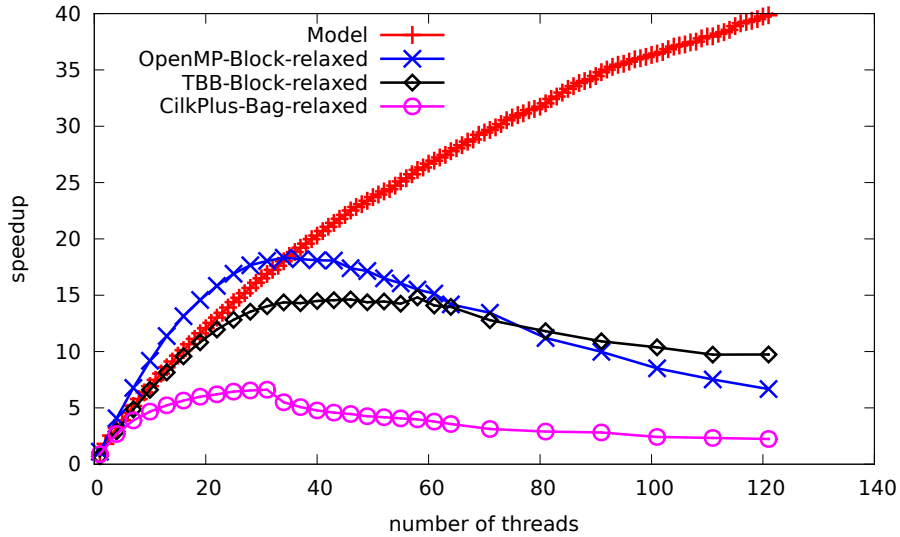


(j) inline_1

Comparisons



(k) All graphs on CPU



(l) All graphs on Intel MIC

Outline

Conclusions

Hyperthreading

In all experiment the behavior of the algorithm changed when hyperthreading is used.

- Coloring speeds up with different slopes up to 4 threads per core.
- Loaded computation peeked at 2 threads per core.
- None of the BFS kernel improved with more than 1 thread per core.

Simple scheduling policies

- Simple dynamic scheduling policies appear to be the best since they keep scheduling overhead low allowing to pump as much data as possible.
- Difference disappear quickly when the amount of computation increases.

Perspective

Knight Corner

All the experiments were run on prototype KNF cards. Looking forward to perform analysis on production KNC cards.

Comparison with GPUs

Will the Intel MIC architecture allow to perform graph analysis kernels faster than GPUs? Performing fair comparisons.

Programming models

We used simple parallelism. But the cores are independent; Pipelined computing should be efficient. Integration in Dataflow middleware such as DataCutter.

Thank you

Support

Intel for allowing us to use Knight Ferry prototypes.
OSC for providing computation infrastructure.

More information

contact : umit@bmi.osu.edu

visit: `http://bmi.osu.edu/hpc/` or `http://bmi.osu.edu/~umit`

Research at HPC lab is funded by


David A. Bader and Kamesh Madduri.
Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks.
In *International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2008.


Charles L. Leiserson and Tao B. Schardl.
A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers).
In *Symposium on Parallel Architectures and Algorithms (SPAA)*, pages 303–314, 2010.