Parallel Dataflow Graph Coloring

Ahmet Erdem Sarıyüce[1], **Erik Saule**[2], and Ümit V. Çatalyürek[1,3]

esaule@uncc.edu, {aerdem,umit}@bmi.osu.edu

[1]Department of Biomedical Informatics, The Ohio State University
[2]Department of Computer Science, University of North Carolina at Charlotte
[3]Department of Electrical and Computer Engineering, The Ohio State University

Scheduling in Aussois^W Dagstuhl^W^W^W
Algorithms and Scheduling Techniques for Exascale Systems

Outline

1. Parallel Graph Coloring
2. Dataflow Graph Coloring
3. What's the link with scheduling?
4. Conclusion

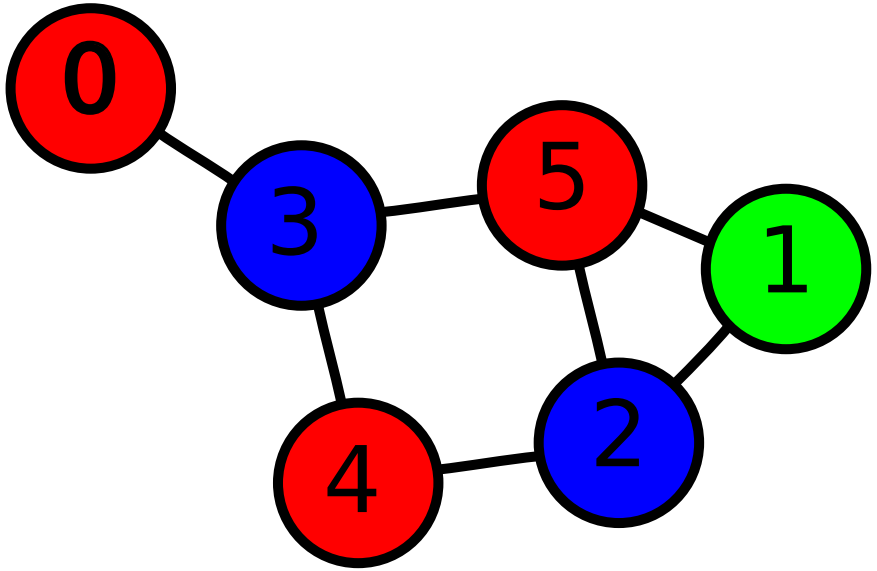
The Graph Coloring Problem

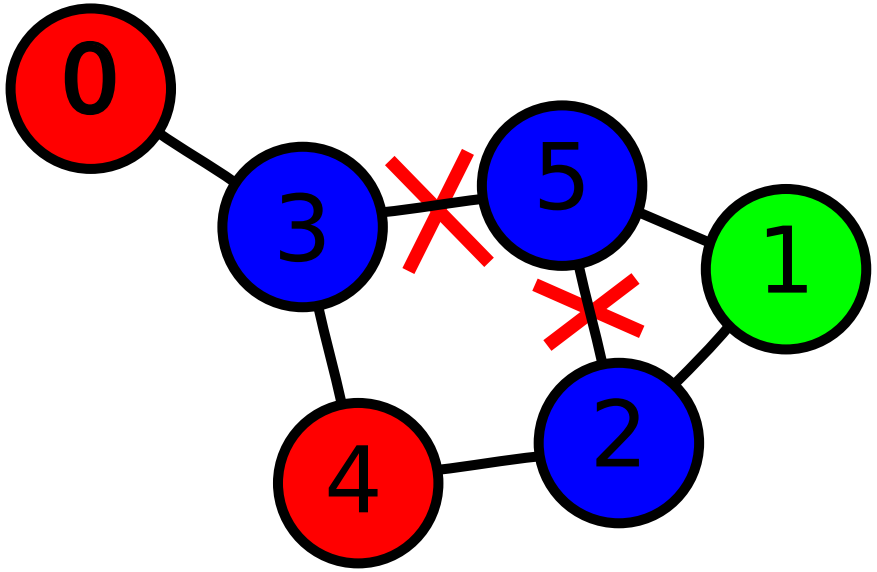
Definition

Coloring a graph consists in assigning a color (an integer) to each vertex so that no two adjacent vertices have the same color.

Complexity

The problem of finding the coloring with minimum number of colors is NP-Hard.
No approximation within $|V|^{1-\epsilon}$.
Greedy algorithm returns a solution with less than $1 + \Delta$ colors.





Graph Coloring Algorithm

First Fit algorithm

Pick a vertex and assign it the first available color. Then pick another one.
There exists a vertex ordering which leads to an optimal coloring.

Algorithm 1: Sequential greedy coloring.

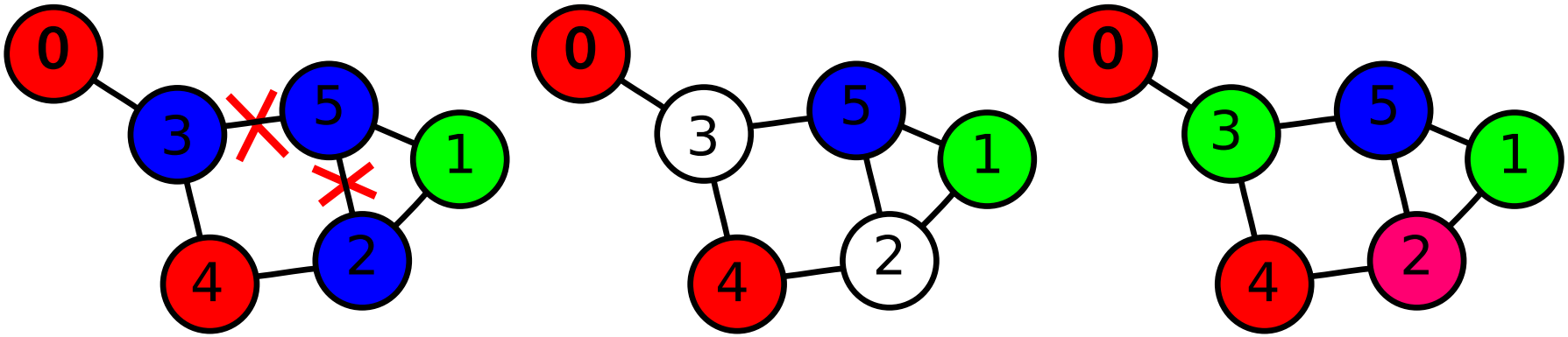
Data: $G = (V, E)$
for each $v \in V$ **do**
  for each $w \in adj(v)$ **do**
    forbiddenColors[color[w]] $\leftarrow v$
  color[v] $\leftarrow \min\{i > 0 : \text{forbiddenColors}[i] \neq v\}$

Many derivative algorithms:

- With Largest First
- With Smallest Last
- Dynamic orderings
- Least Used instead of First Fit.
- Iterated algorithm to do local descent.

Today, let's talk about the natural one.

Parallel Speculative Graph Coloring (Shared Memory)



Algorithm 2: 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 3: 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

At least two passes. More if unlucky (in practice $2 + \epsilon$)

Outline

1. Parallel Graph Coloring
2. Dataflow Graph Coloring
3. What's the link with scheduling?
4. Conclusion

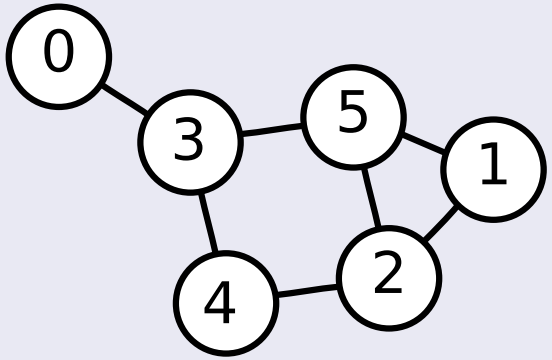
Parallel Dataflow Algorithm

Principle

The principle of Dataflow algorithm is that the generation of a result triggers the computation of the next tasks.

Dataflow coloring

The idea is to pick an absolute order of the vertices and each vertex only consider the color of the vertices with ID lesser than theirs.



- 0 and 1 can be executed concurrently
- 2 and 3 can be executed concurrently
- 4 and 5 can be executed concurrently

Not speculative, so only one pass.

Two approaches

Pick the vertices in some order. What happens when you pick a vertex with neighboors with high priority which haven't been allocated a color.

Recursive Dataflow

You recursively process the neighbor.

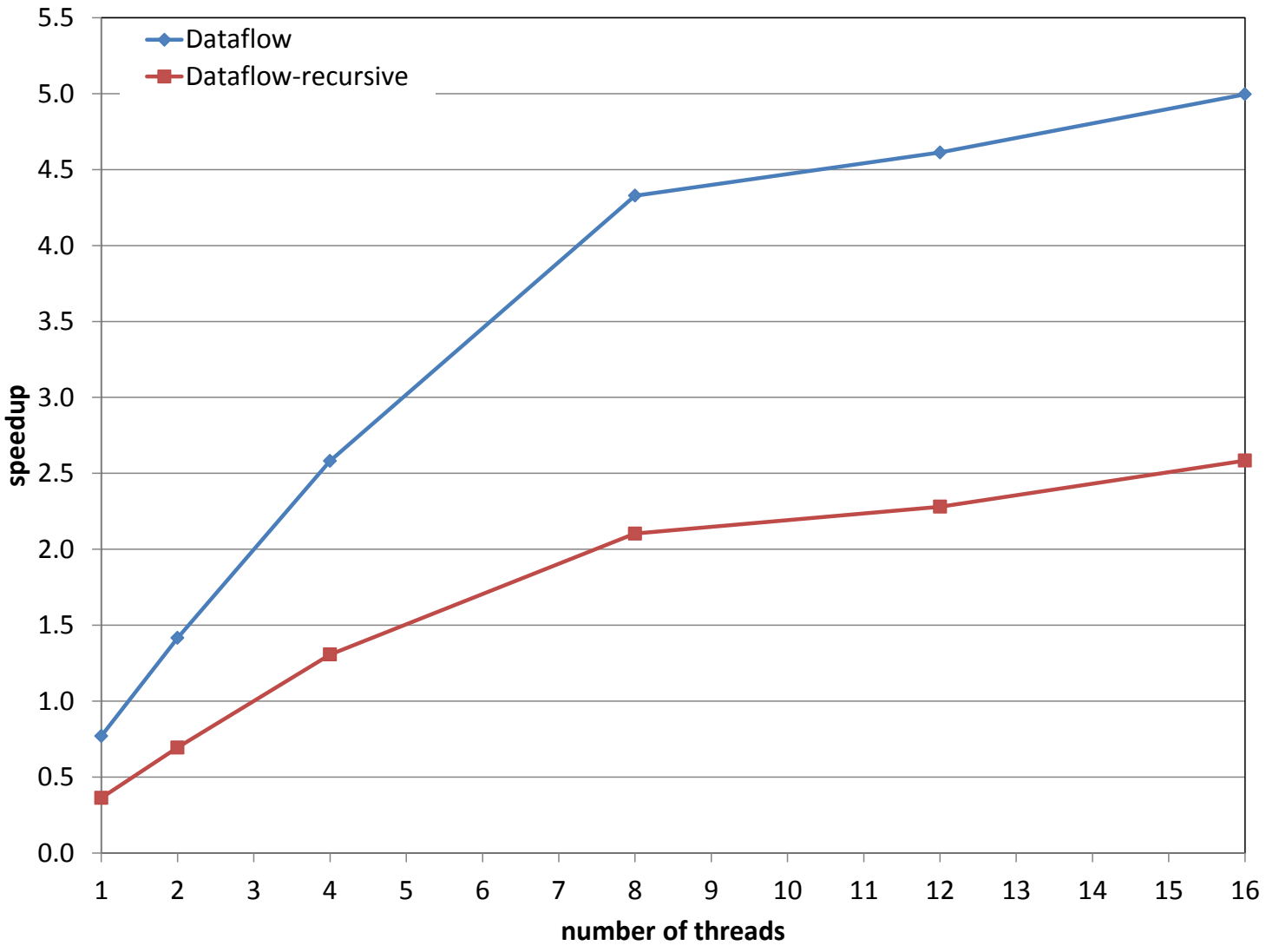
- No waiting time
- Some form of "workstealing" algorithm
- Complex synchronisation
- Higher memory allocation (or potentially redundant work)

(Direct) Dataflow

You wait.

- No redudant work
- Simpler worksharing constraint.
- But maybe you waste time waiting.

Which is best?

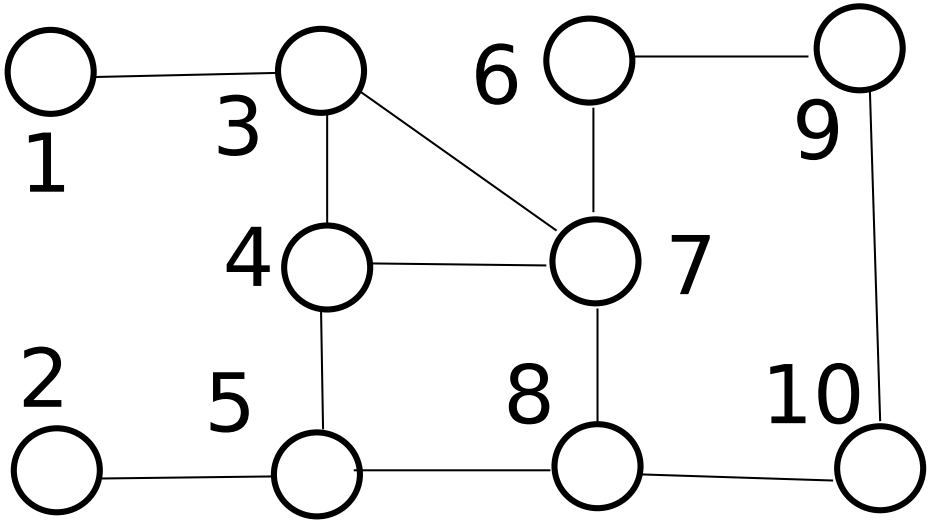


with lots of (yet) unexplained optimizations.

Outline

1. Parallel Graph Coloring
2. Dataflow Graph Coloring
3. What's the link with scheduling?
4. Conclusion

In practice, parallel speedup is 1

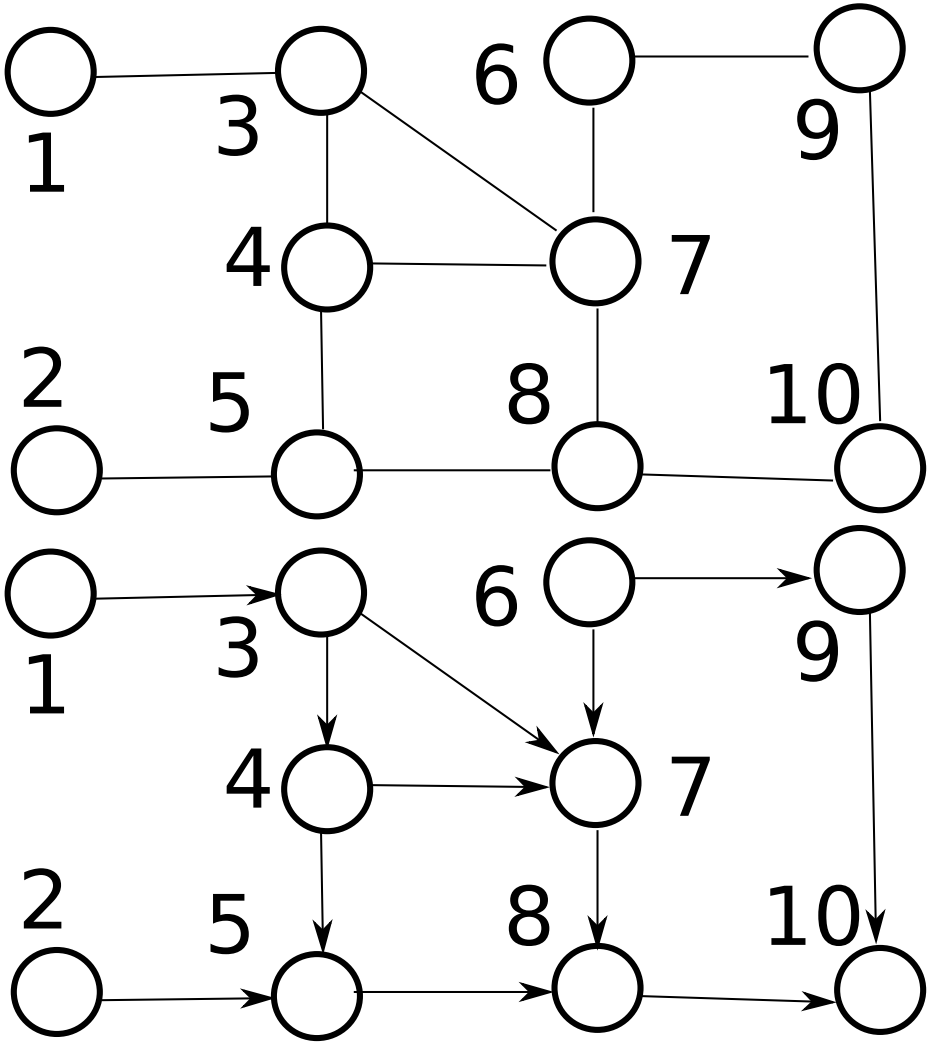


The graph is executed one vertex after another. So there are actually dependencies.

Graham List Scheduling

When scheduling a dag, a greedy algorithm gets:
$C_{max} \leq \frac{W}{p} + (1 - \frac{1}{p})CP$

In practice, parallel speedup is 1



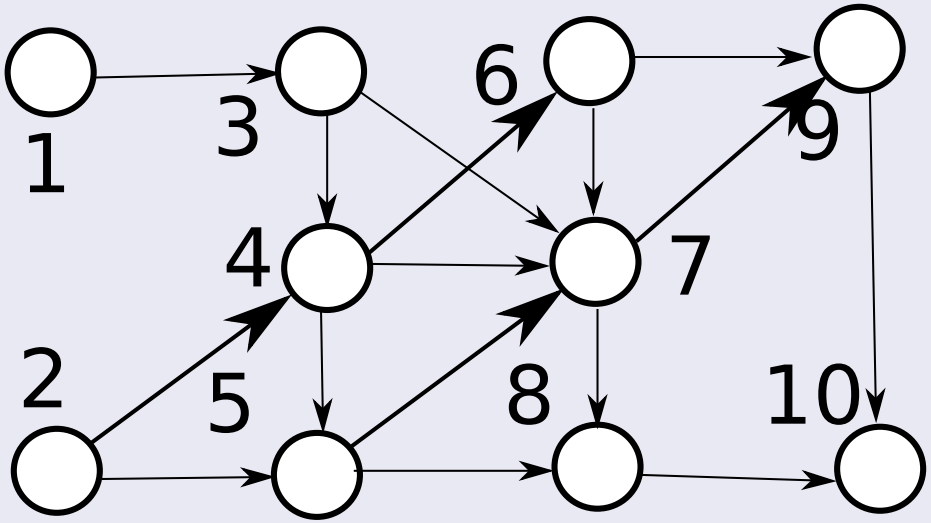
The graph is executed one vertex after another. So there are actually dependencies.

Graham List Scheduling

When scheduling a dag, a greedy algorithm gets:
$C_{max} \leq \frac{W}{p} + (1 - \frac{1}{p})CP$

It gets worse...

Static Scheduling

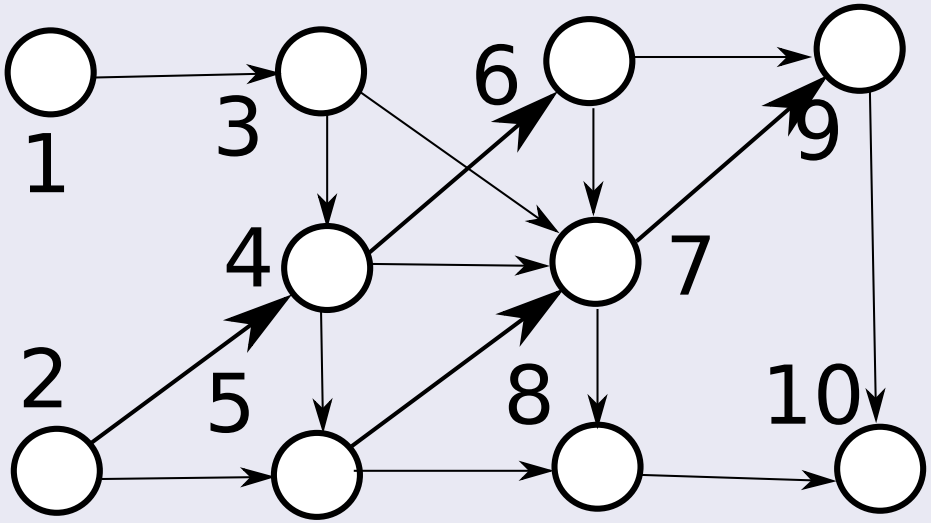


If you use a static OpenMP schedule, you add *de facto* dependencies in your graph. And the critical path increases significantly.

That's easy! Let's use dynamic instead!

It gets worse...

Static Scheduling

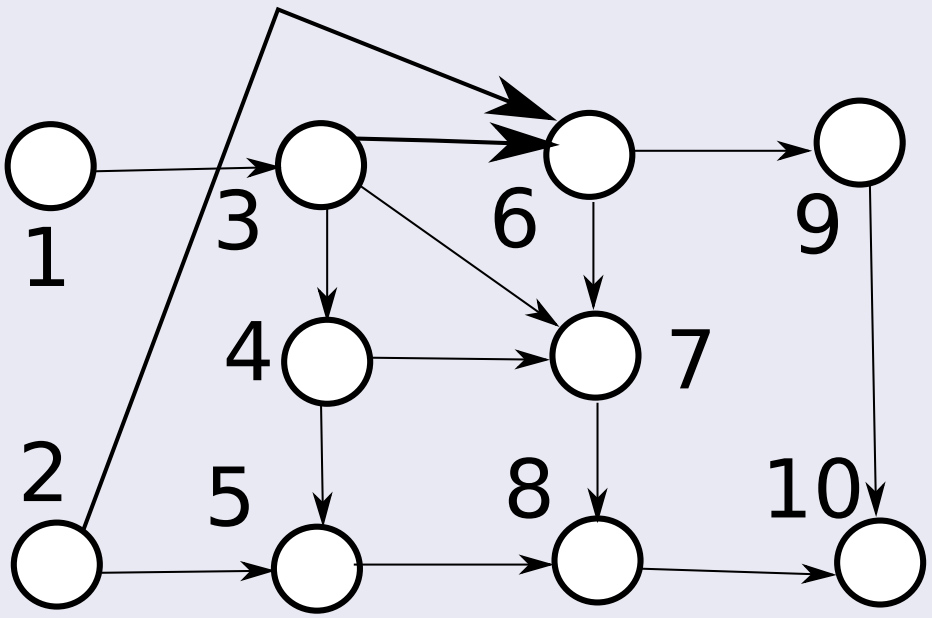


If you use a static OpenMP schedule, you add *de facto* dependencies in your graph. And the critical path increases significantly.

That's easy! Let's use dynamic instead!

Is Dynamic Better?

Dynamic Scheduling



Even if you use a dynamic OpenMP schedule, similar effect still happen.

With two threads 4 and 5 need to be executed before 6 can start. So 1, 2 and 3 are implicit predecessor of 6. Because that is what the scheduler will do.

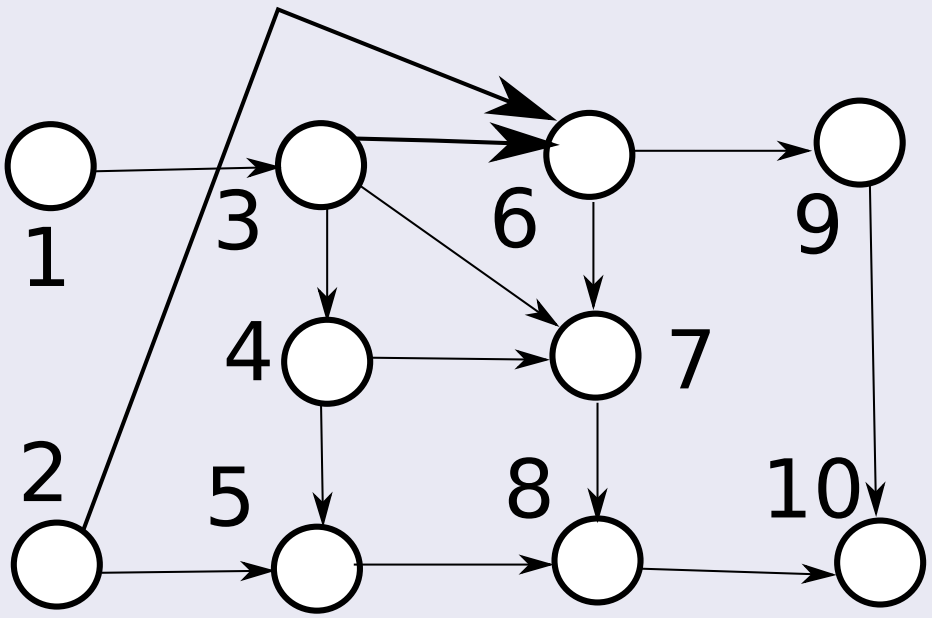
An easy solution

Compute a level by level order.

Well... That requires a graph traversal. The whole point was to traverse the graph only once.

Is Dynamic Better?

Dynamic Scheduling



Even if you use a dynamic OpenMP schedule, similar effect still happen.

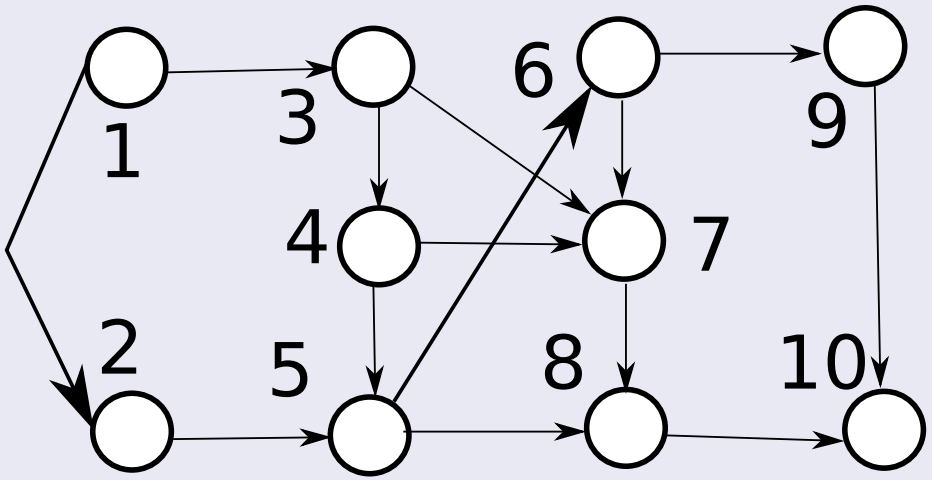
With two threads 4 and 5 need to be executed before 6 can start. So 1, 2 and 3 are implicit predecessor of 6. Because that is what the scheduler will do.

An easy solution

Compute a level by level order.
Well... That requires a graph traversal. The whole point was to traverse the graph only once.

It gets EVEN worse

Nobody should use "dynamic,1"

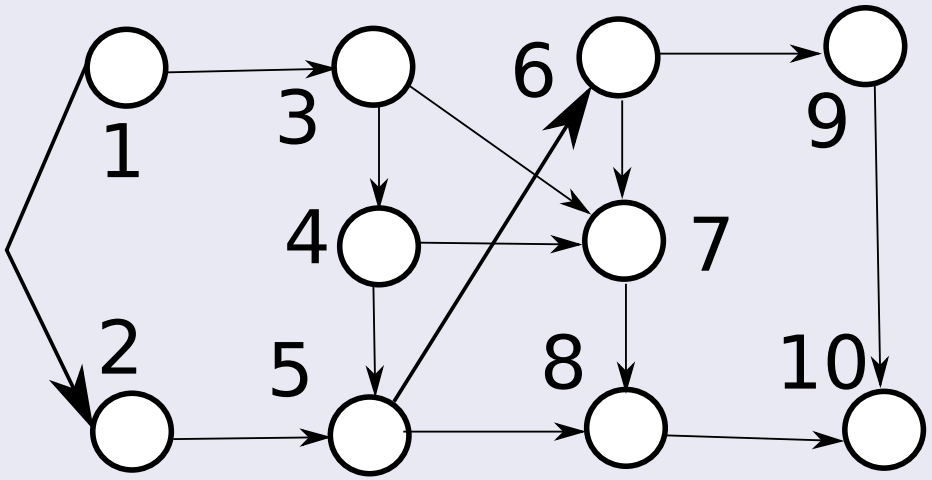


Since vertices are grouped together, by openmp's granularity, you have implicit edges between the vertices of each group.
There is an implicit edge between 1 and 2, and between 5 and 6.

In this type of kernel, you should at least use groups of 32 vertices.

It gets EVEN worse

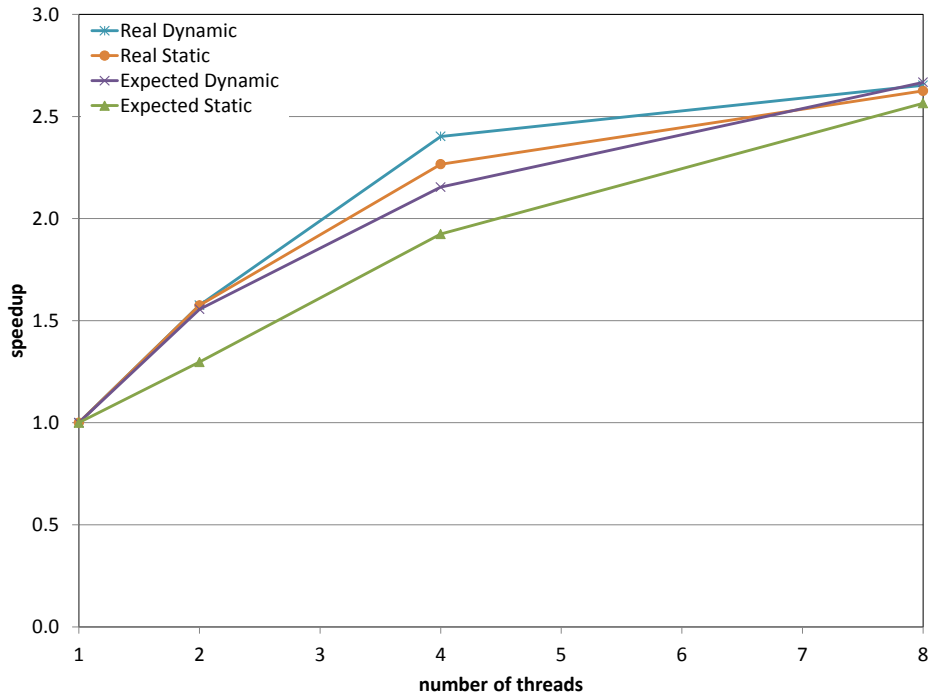
Nobody should use "dynamic,1"



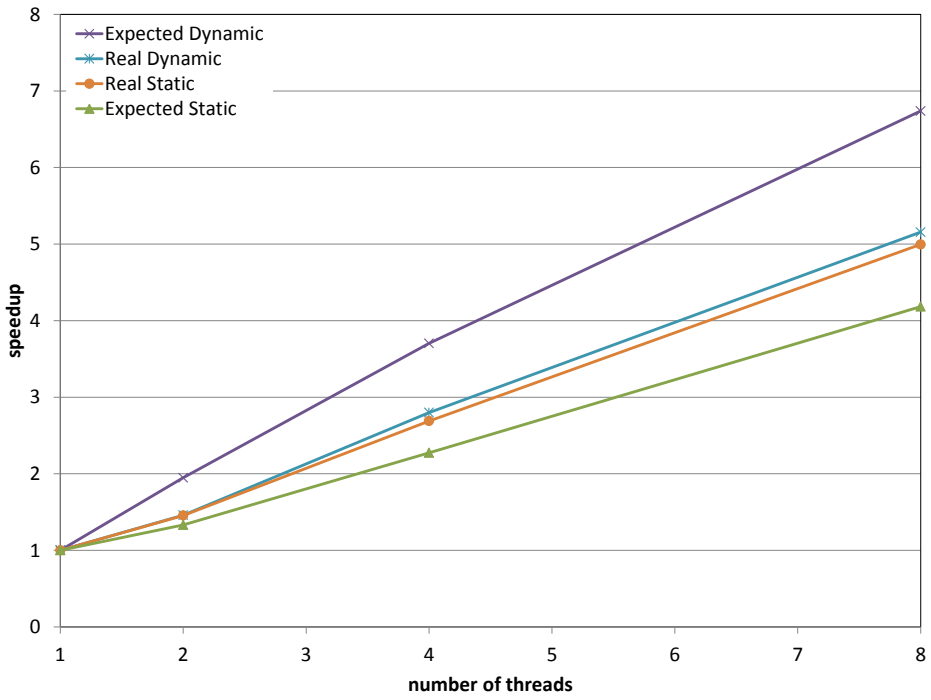
Since vertices are grouped together, by openmp's granularity, you have implicit edges between the vertices of each group.
There is an implicit edge between 1 and 2, and between 5 and 6.

In this type of kernel, you should at least use groups of 32 vertices.

First Results



(a) auto

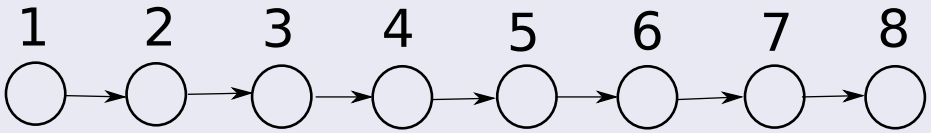


(b) ldoor

Ouch!

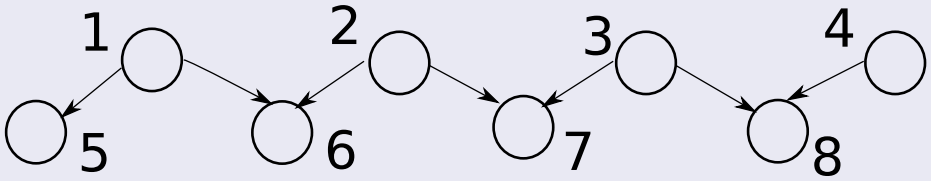
Reordering of vertex IDs

Just a chain



The critical path can be quite long in a natural ordering.

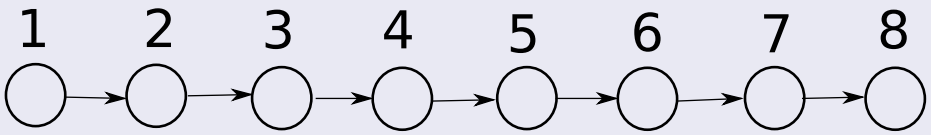
At best



Need to traverse the graphs...

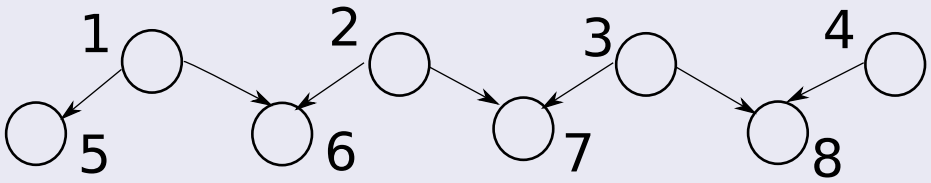
Reordering of vertex IDs

Just a chain



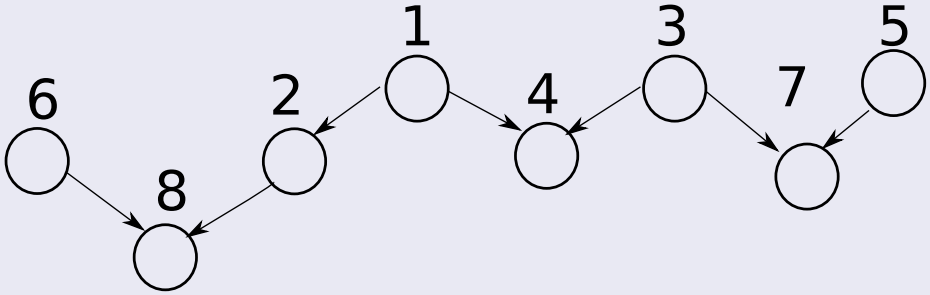
The critical path can be quite long in a natural ordering.

At best



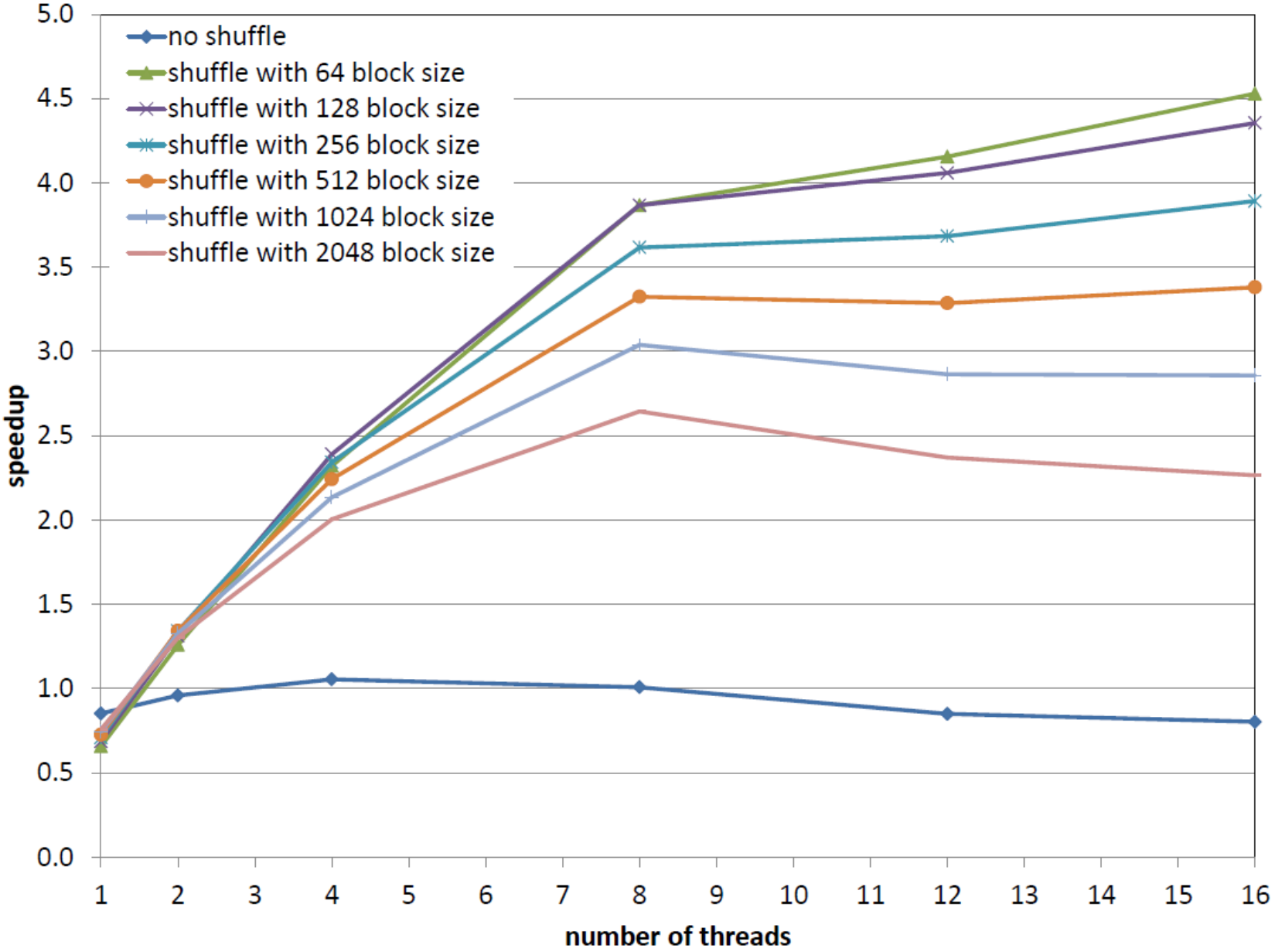
Need to traverse the graphs...

At random



Not best but probably not the worst you can get.
For cache purposes, you need to keep some locality, so you shuffle blocks of vertices.
Any guarantee on that?

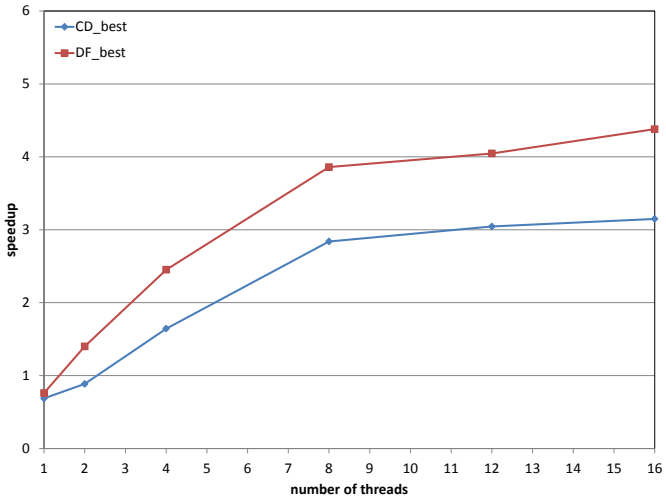
Results


no shuffle
shuffle with 64 block size
shuffle with 128 block size
shuffle with 256 block size
shuffle with 512 block size
shuffle with 1024 block size
shuffle with 2048 block size
speedup
number of threads
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

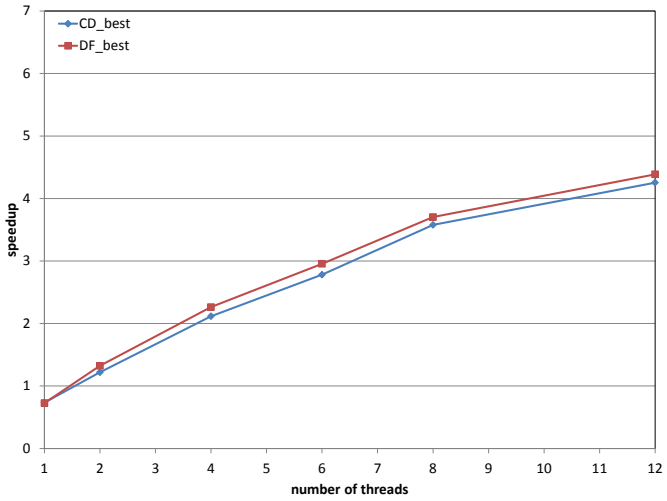
Outline

1. Parallel Graph Coloring
2. Dataflow Graph Coloring
3. What's the link with scheduling?
4. Conclusion

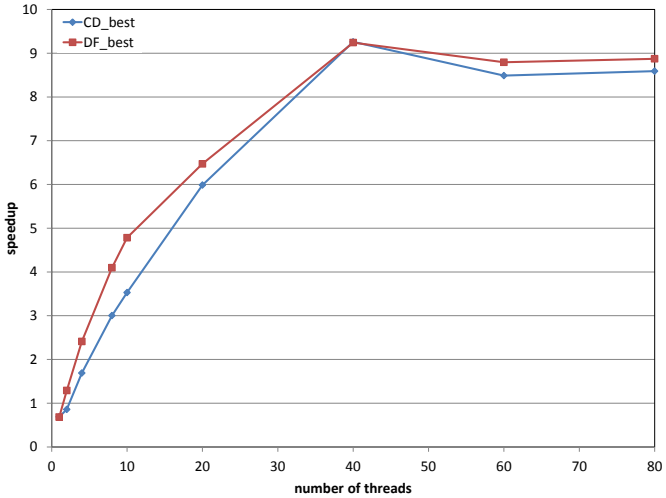
Faster than speculative coloring?



Owens (8 cores)



Oakley (16 cores)



Mirasol (40 cores)

Works on small machines.
Impact seem to decrease with core count.

- There is an other limiting factor

Wrap up

Conclusions

- designed and tested dataflow coloring algorithms on multicore architectures.
- analyzed their performance.

Classical scheduling helps understanding performance issues in not-so-related problems. Similar analysis performed on BFS.

Future works

- how to pick a better order?
- can we get rid of the computation and lookup of permutation?

Thank you

More information

Contact : esaule@uncc.edu
Visit: http://webpages.uncc.edu/~esaule