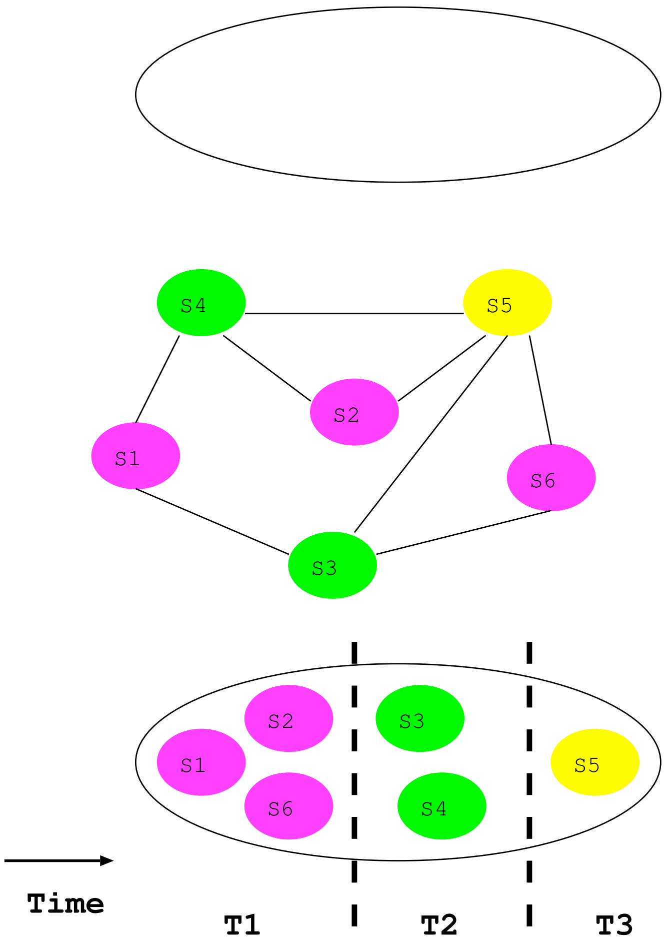
Parallelizing Graph Algorithms

- Challenges:
  - Runtime dominated by memory latency than processor speed
  - Little work is done while visiting a vertex or an edge
    - Little computation to hide memory access cost
  - Access patterns determined only at runtime
    - Prefetching techniques inapplicable
  - There is poor data locality
    - Difficult to obtain good memory system performance
- For these reasons, parallel performance
  - on distributed memory machines is often poor
  - on shared memory machines is often better

We consider here *graph coloring* as an example of a graph algorithm to parallelize on shared memory machines

Graph coloring

- *Graph coloring* is an assignment of colors (positive integers) to the vertices of a graph such that adjacent vertices get different colors
- The objective is to find a coloring with the *least* number of colors
- Examples of *applications*:
  - Concurrency discovery in parallel computing (illustrated in the figure to the right)
  - Sparse derivative computation
  - Frequency assignment
  - Register allocation, etc



A greedy algorithm for coloring

- Graph coloring is NP-hard to solve optimally (and even to approximate)
- The following Greedy algorithm gives very good solution in practice

Algorithm 1 Sequential greedy coloring.

procedure GREEDY($G(V, E)$)
    for each $v \in V$ **do**
        for each $w \in adj(v)$ **do**
            forbiddenColors[color[w]] $\leftarrow v$ $\triangleright$ mark color of w as forbidden to v
        color[v] $\leftarrow \min\{c > 0 : \text{forbiddenColors}[c] \neq v\}$ $\triangleright$ c is the *smallest* permissible color to v

color is a vertex-indexed array that stores the color of each vertex
forbiddenColors is a color-indexed array used to mark impermissible colors to a vertex

Complexity of GREEDY: $O(|E|)$ (thanks to the way the array forbiddenColors is used)

Parallelizing Greedy Coloring

- Desired goal: parallelize GREEDY such that
  - Parallel runtime is roughly $O(|E|/p)$ when p processors (threads) are used
  - Number of colors used is nearly the same as in the serial case
- Difficult to achieve since GREEDY is inherently sequential
- Challenge: come up with a way to create concurrency in a nontrivial way

A potentially "generic" parallelization technique

- "Standard" Partitioning
  - Break up the given problem into *p* independent subproblems of almost equal sizes
  - Solve the *p* subproblems concurrently

  Main work lies in the decomposition step which is often no easier than solving the original problem

- "Relaxed" Partitioning
  - Break up the problem into *p*, not necessarily entirely independent, subproblems of almost equal sizes
  - Solve the *p* subproblems concurrently
  - Detect inconsistencies in the solutions concurrently
  - Resolve any inconsistencies

  Can be used potentially successfully if the resolution in the fourth step involves only local adjustments

“Relaxed Partitioning” applied towards parallelizing Greedy coloring

- Speculation and Iteration:
  - Color as many vertices as possible concurrently, tentatively tolerating potential conflicts, detect and resolve conflicts afterwards (iteratively)

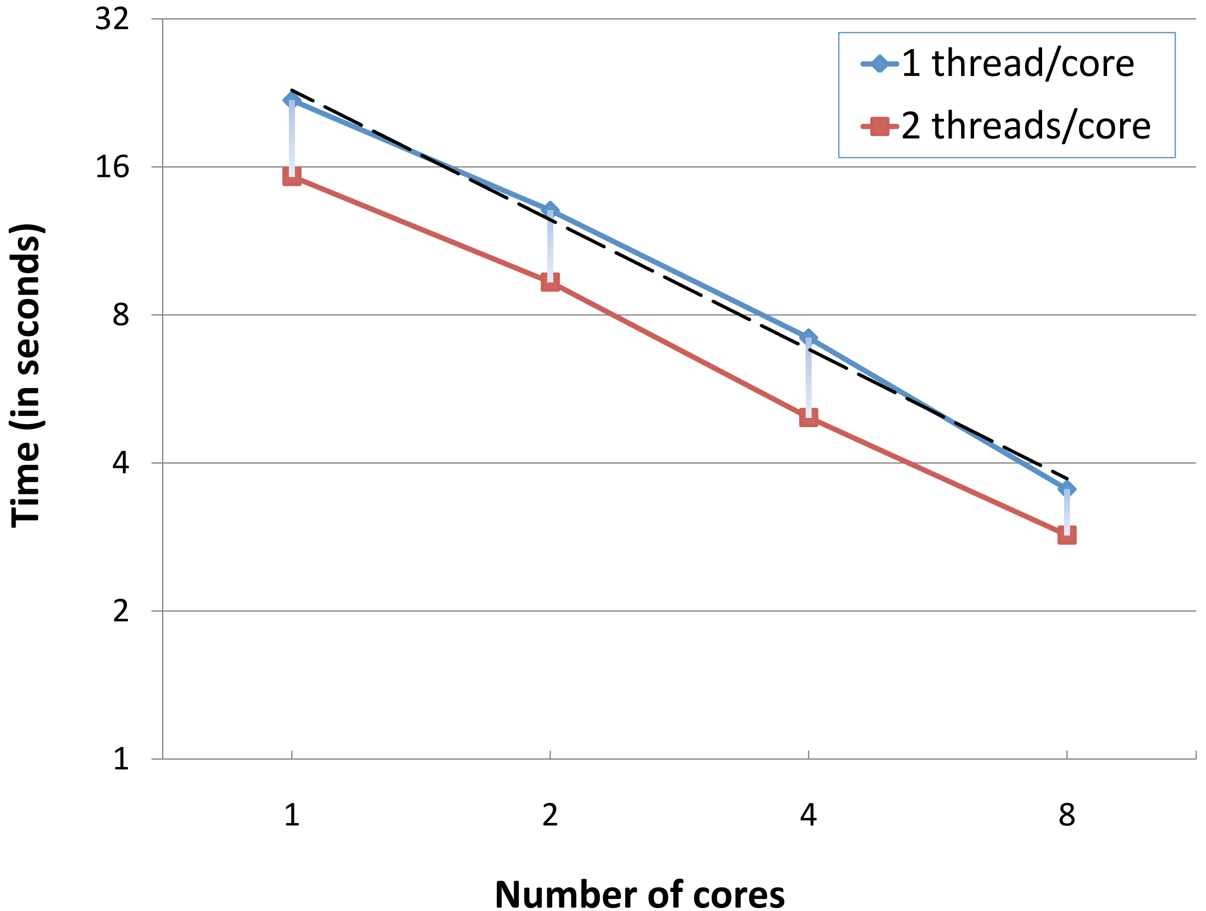
Parallel Coloring on Shared Memory Platforms (using Speculation and Iteration)

Algorithm 2 Iterative parallel greedy coloring.

1: **procedure** ITERATIVE($G(V, E)$)
2: $U \leftarrow V$ ▷ U is the current set of vertices to be colored
3: **while** $U \neq \emptyset$ **do**
4: **for each** $v \in U$ in **parallel do** ▷ Phase 1: tentative coloring
5: **for each** $w \in adj(v)$ **do**
6: mark $\mathsf{color}[w]$ as forbidden to v
7: Pick the smallest permissible color c for vertex v
8: $R \leftarrow \emptyset$ ▷ R is a set of vertices to be recolored
9: **for each** $v \in U$ in **parallel do** ▷ Phase 2: conflict detection
10: **for each** $w \in adj(v)$ **do**
11: **if** $\mathsf{color}[v] = \mathsf{color}[w]$ **and** $v > w$ **then**
12: $R \leftarrow R \cup \{v\}$
13: $U \leftarrow R$

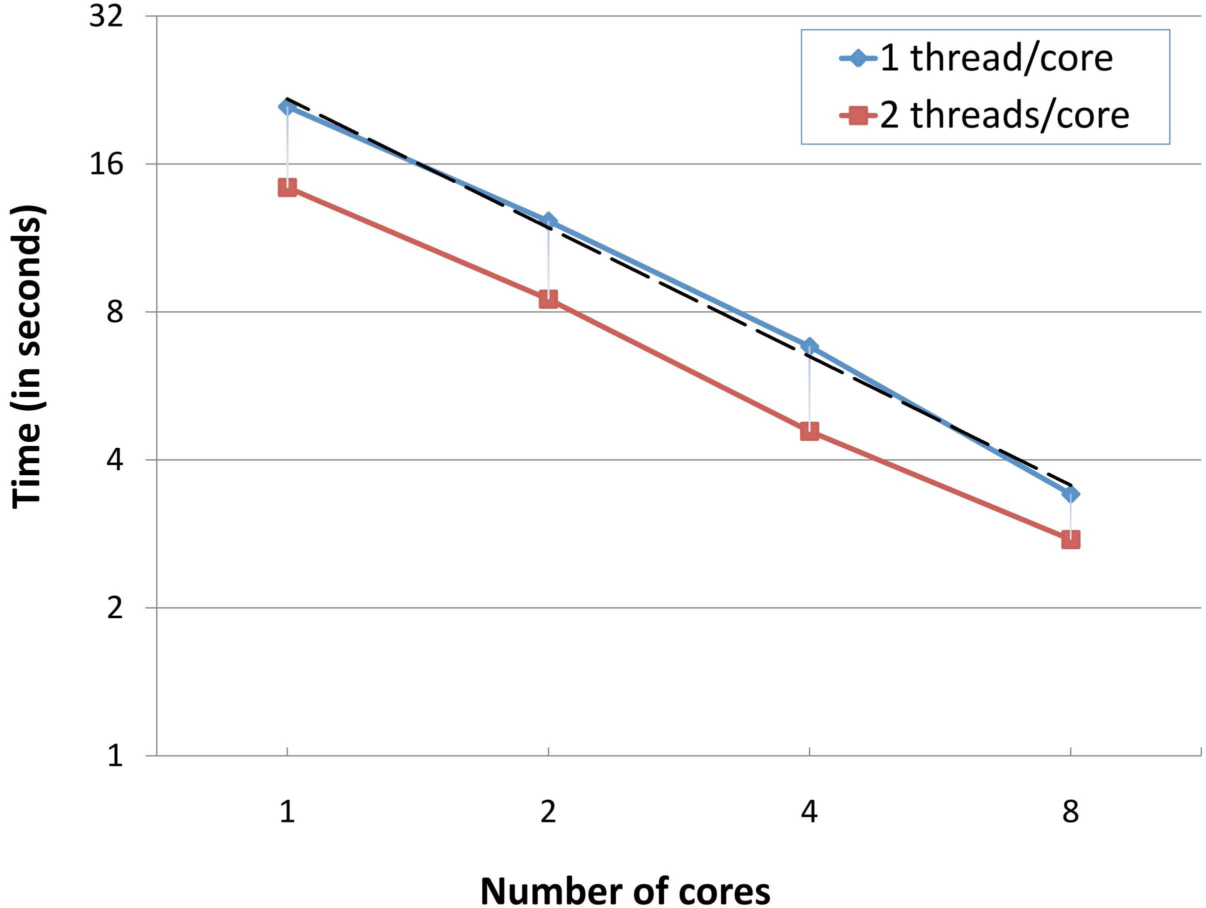
Lines 4 and 9 can be parallelized using the OpenMP directive
#pragma omp parallel for

Sample experimental results of Algorithm 2 (Iterative) on Nehalem: I



Graph (RMAT-G): 16.7M vertices; 133.1M edges;
Degree (avg=16, Max= 1,278, variance=416)

Sample experimental results of Algorithm 2 (Iterative) on Nehalem: II



Graph (RMAT-B): 16.7M vertices; 133.7M edges;
Degree (avg=16, Max= 38,143, variance=8,086)