Scheduling Multithreaded Computations by Work Stealing

Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Presentation: Pete Curry, COMP522, 2014-10-23

Outline

- Problem: efficiently schedule fine-grained tasks
- Computation model: Cilk-style tasks
- Scheduling strategies
  - Work-sharing scheduler
  - Greedy scheduling
  - Busy-leaves property
  - Work-stealing scheduler
- Conclusion

Reference: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

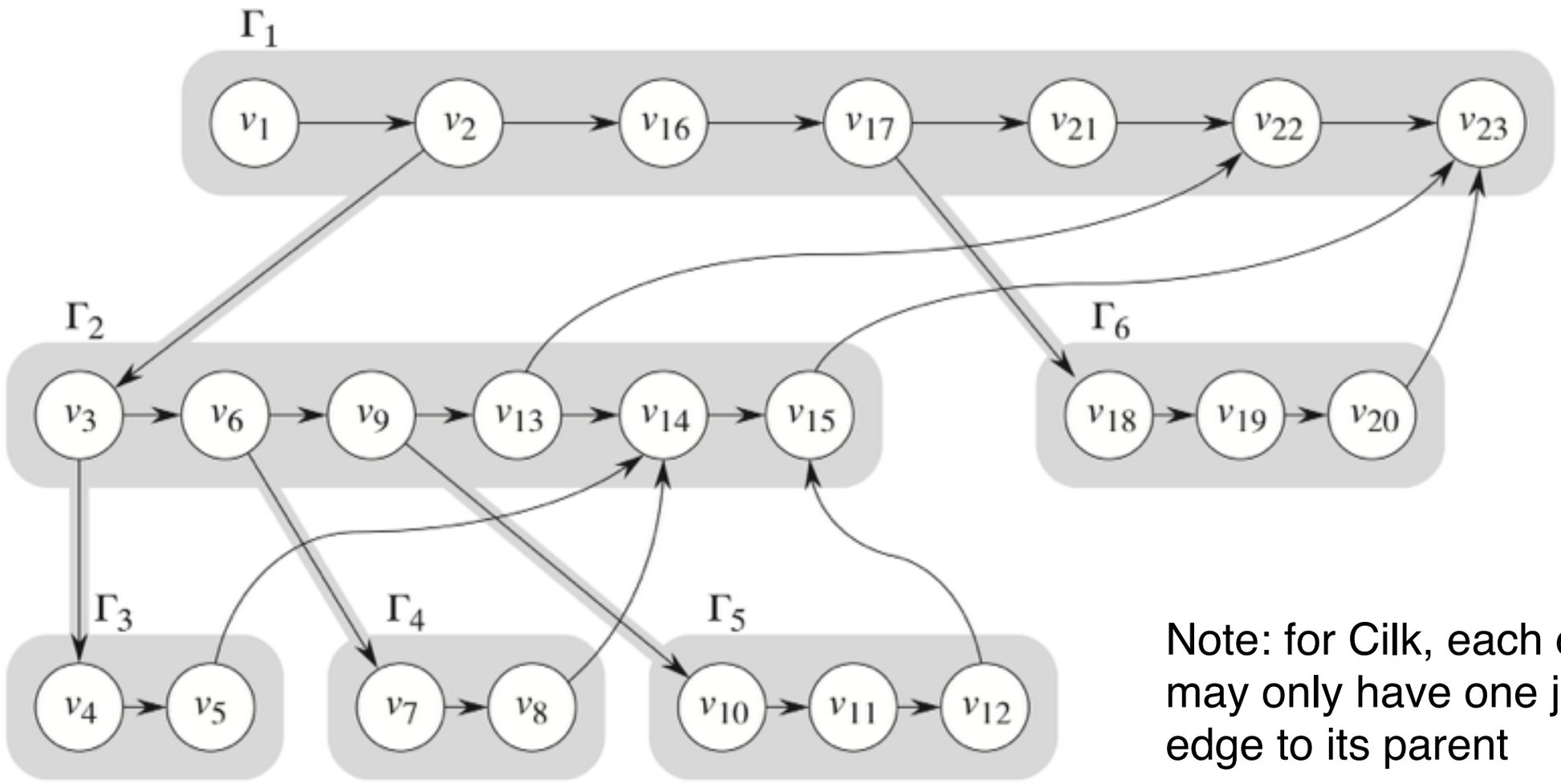
Problem overview

- Dynamic fine-grained tasks, such as in Cilk
- Efficiently schedule tasks across processors
- Avoid unnecessary scheduling overhead
- Too much parallelism creates issues
  - High memory consumption
  - Unnecessary data sharing across processors

- Goal: Linear speedup and linear growth in space, assuming sufficient parallelism

DAG computation model

- Nodes: instructions
- Edges: spawn, continue, join
- Tasks: disjoint subgraphs



Note: for Cilk, each child may only have one join edge to its parent

Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Computation model

- *Strict* parallel computations
  - Call graph with parallel children
  - All join edges from a task go to the task's parent
- *Fully strict* parallel computations
  - All children complete before their parent
  - Cilk follows this rule
- Note: not required for work stealing

Work-sharing scheduler

- Simple idea: when a task is spawned, find a processor to run it on
- Work queue implementation options
  - Global
  - Distributed per-processor

Problems with work-sharing

- Tasks move unnecessarily
  - Causes excessive non-local data accesses
- Can cause excessive growth in space
  - Too much parallelism
  - Breadth-first scheduling
- No formal analysis of bounds
- Inferior results in practice
  - Depends on application

Greedy-scheduling theorem

- Build scheduling model that allows us to prove tight upper bounds on time and space to meet the goal

- Greedy scheduler: perform computation in steps
  - Each step schedules instructions across processors
  - Complete step: all processors busy
  - Incomplete step: some processors idle

Greedy-scheduling theorem

- **Theorem 1**
  For any multithreaded computation with work T_1 and critical-path length T_∞, and for any number P of processors, any greedy P-processor execution schedule X achieves
  $T(X) \leq T_1/P + T_\infty$.

- Worst case: Fully serial along critical path takes T_∞, incomplete steps
- Best case: Fully parallel with total work takes T_1/P, complete steps
- Satisfies time goal, does not account for space requirement

Busy-leaves scheduler

- Keep a global ready task pool
- Depth-first scheduling: work on leaves first
  - When a task A spawns a task B, put A back in the work pool and switch to B
  - When a task completes, if it made the parent become ready switch to the parent, else processor becomes idle
  - When a processor becomes idle, it gets a task from the global ready task pool

- Problem:
  Global ready task pool has too much contention between the processors, not practical to implement

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

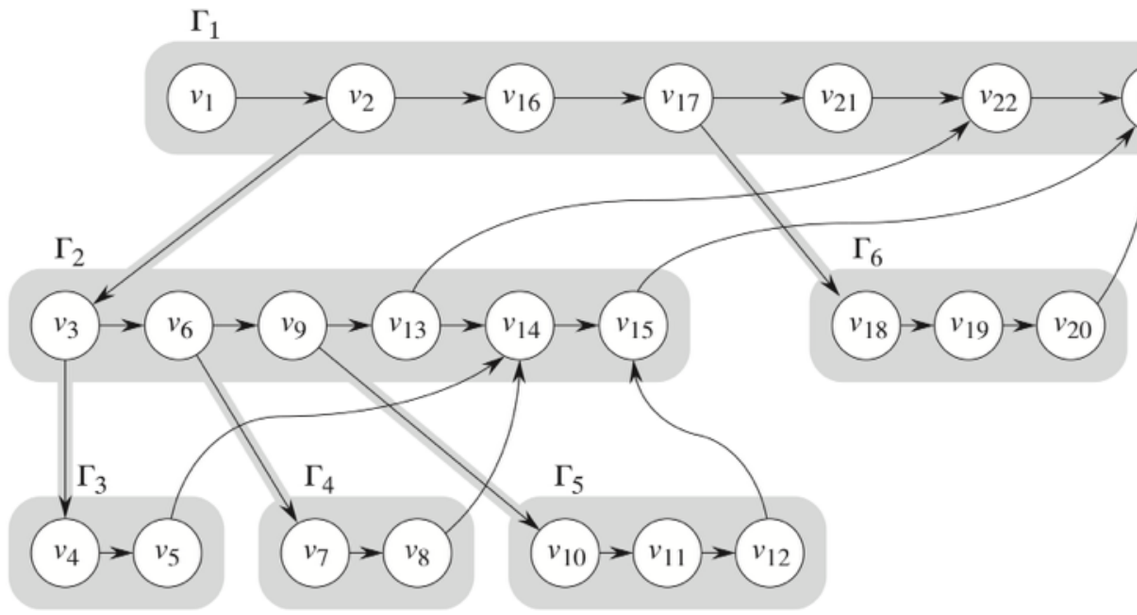


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		processor activity p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

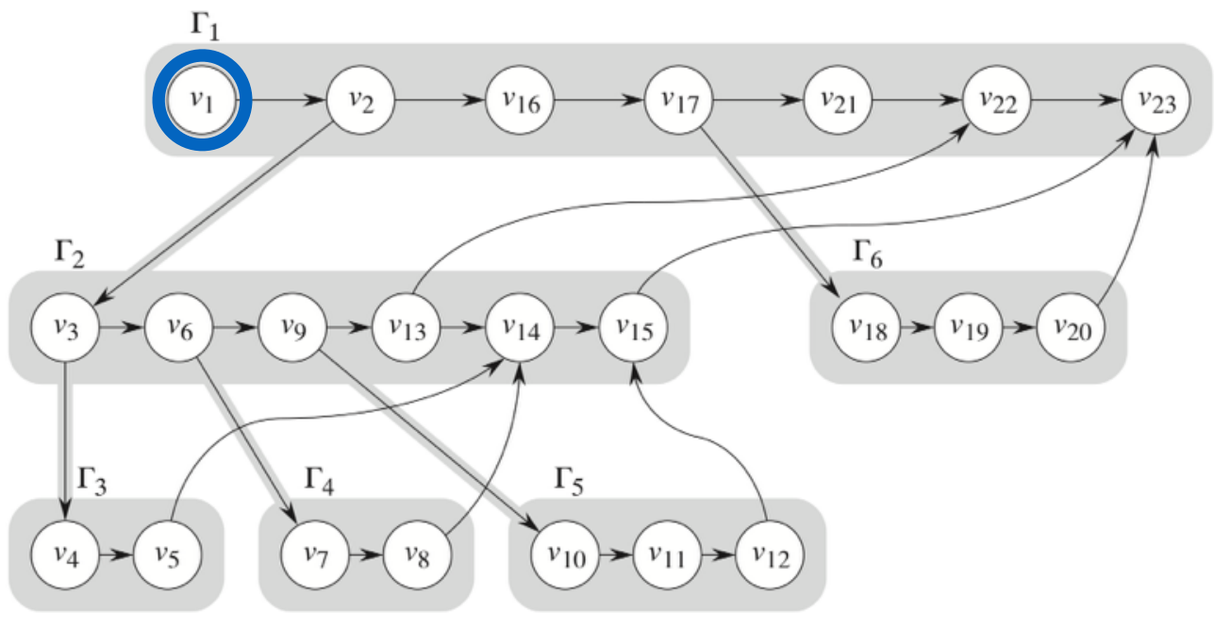


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

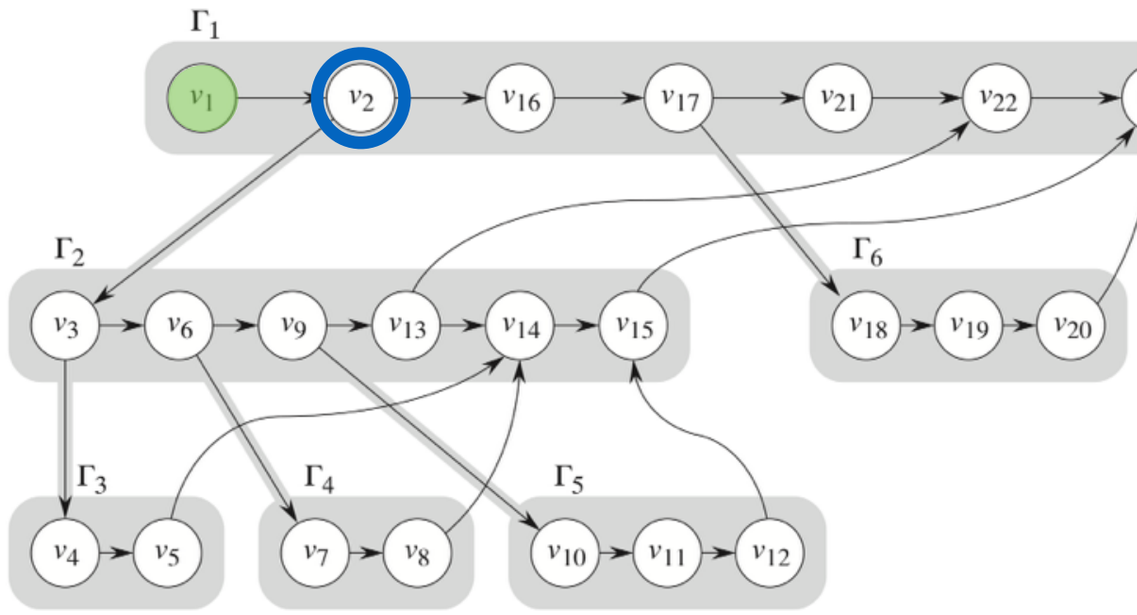


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

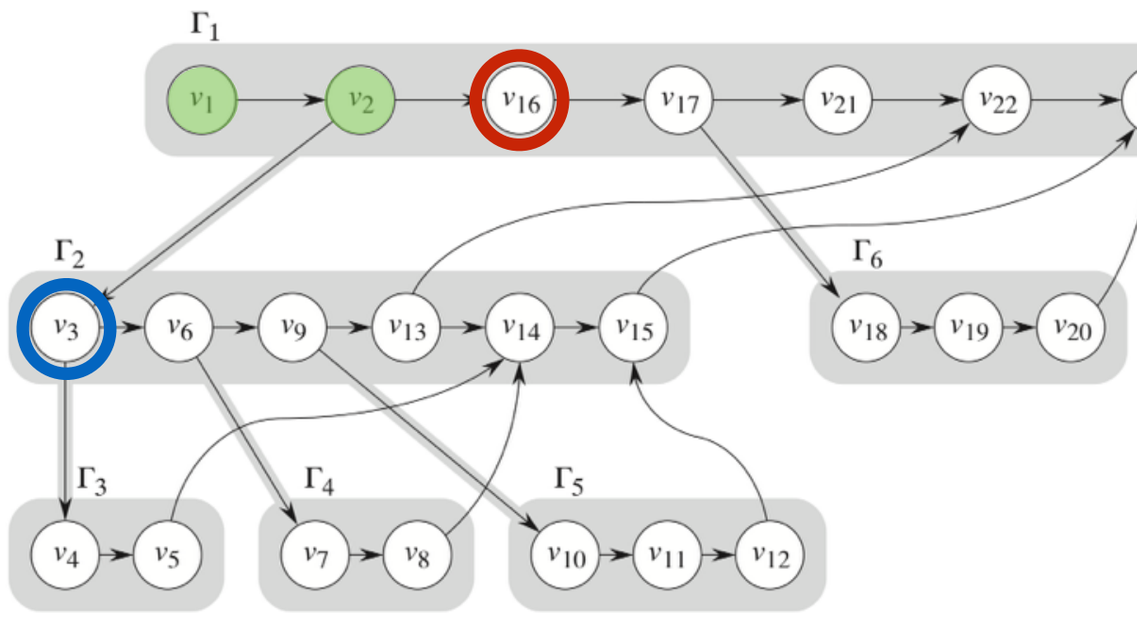


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		processor activity p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

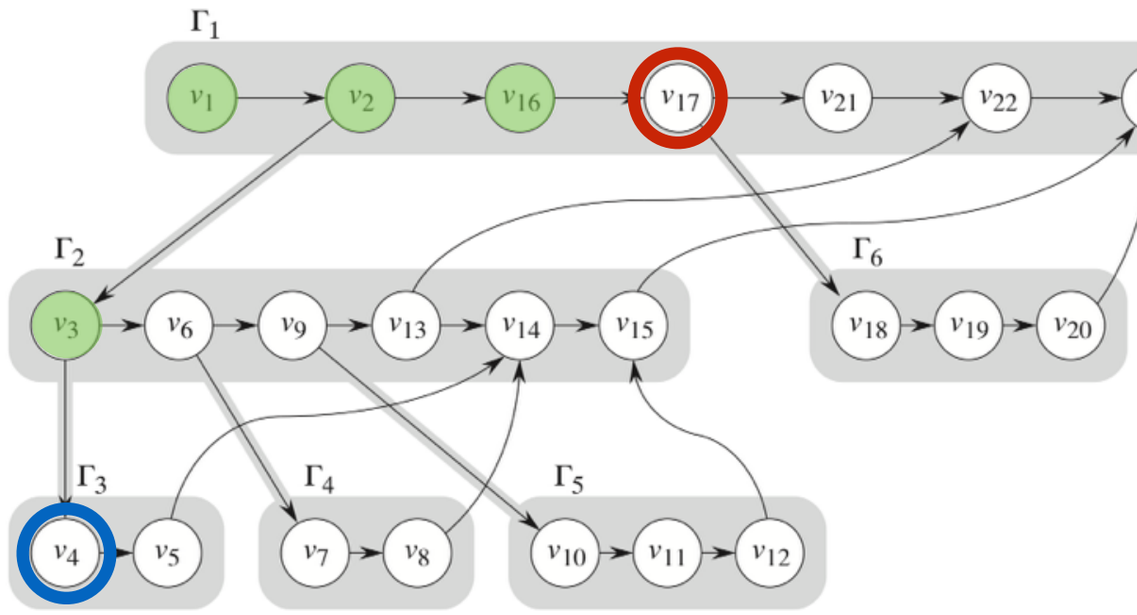


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

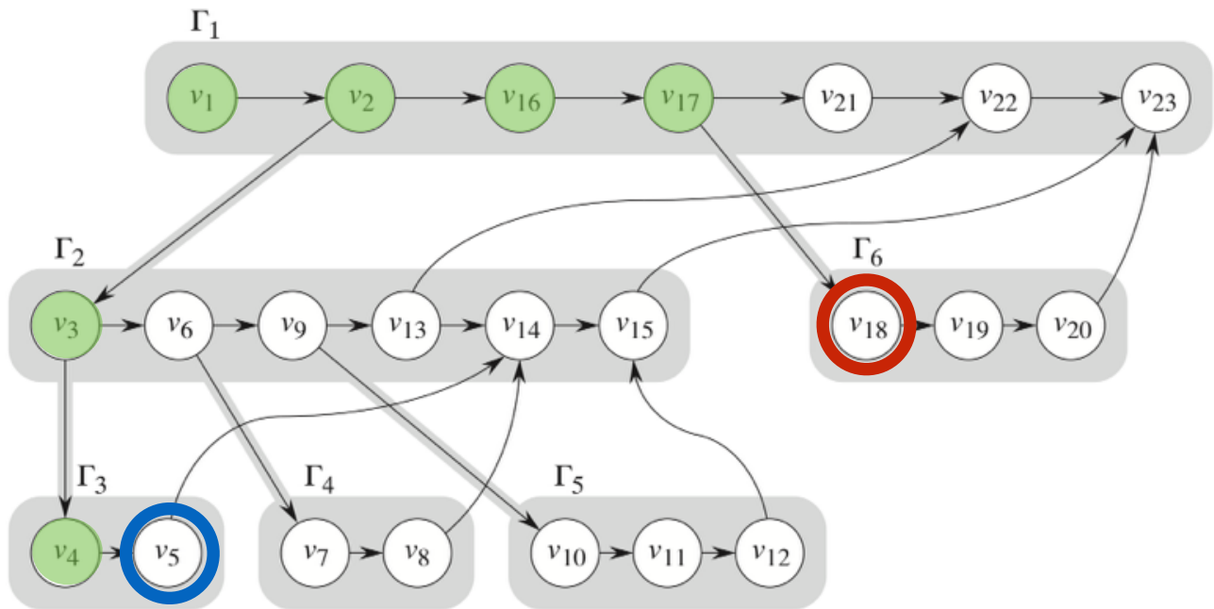


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

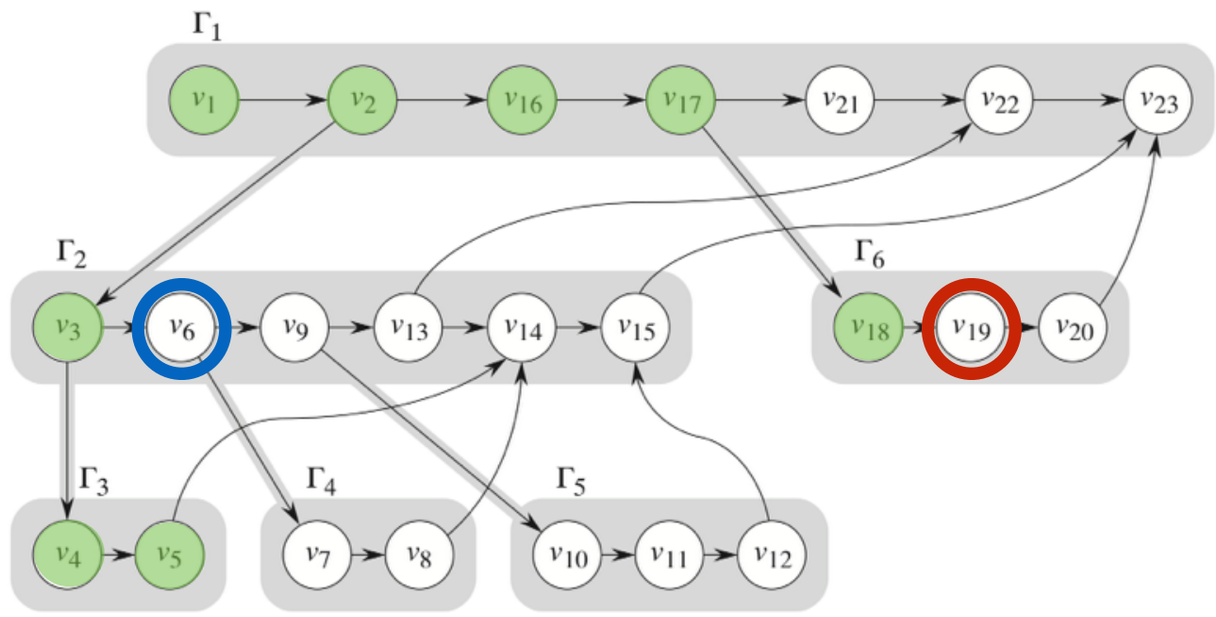


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

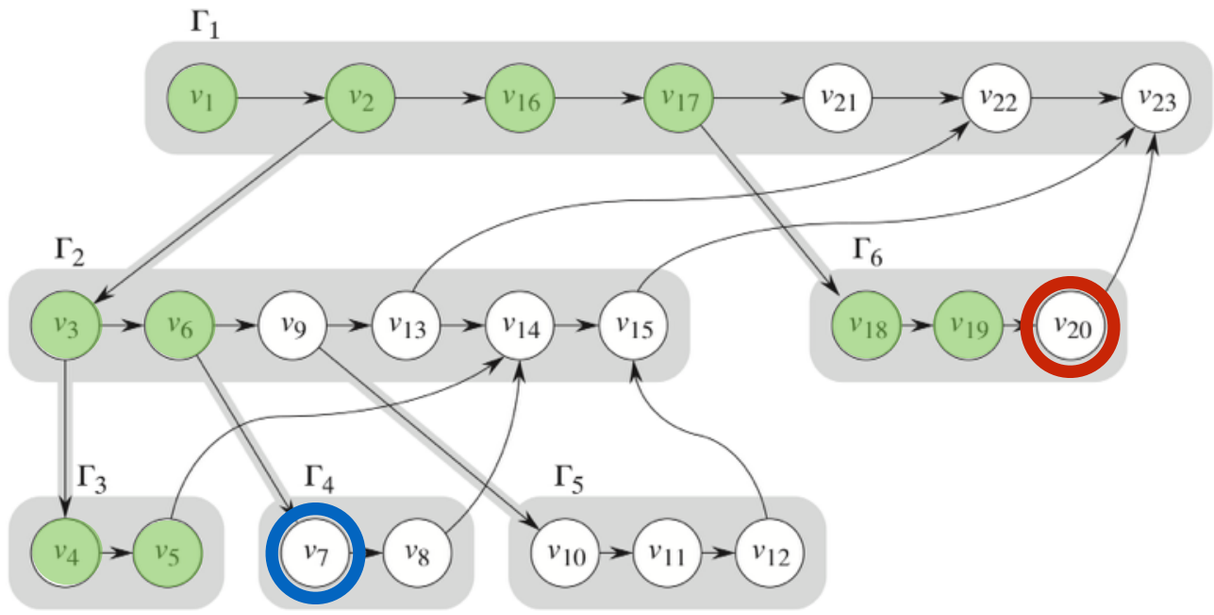


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

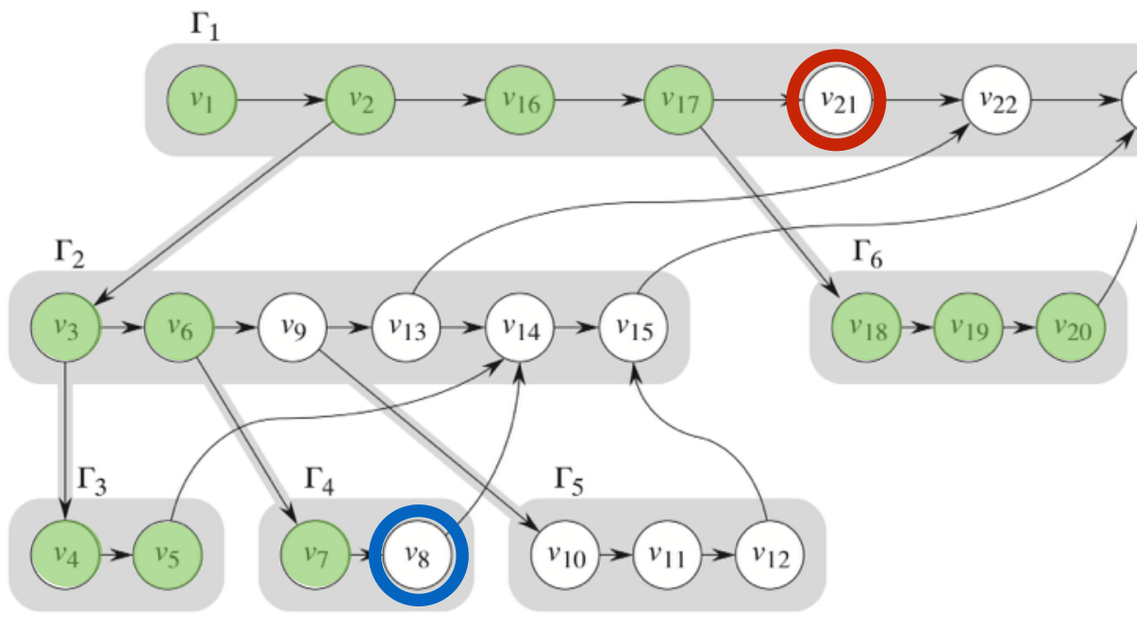


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

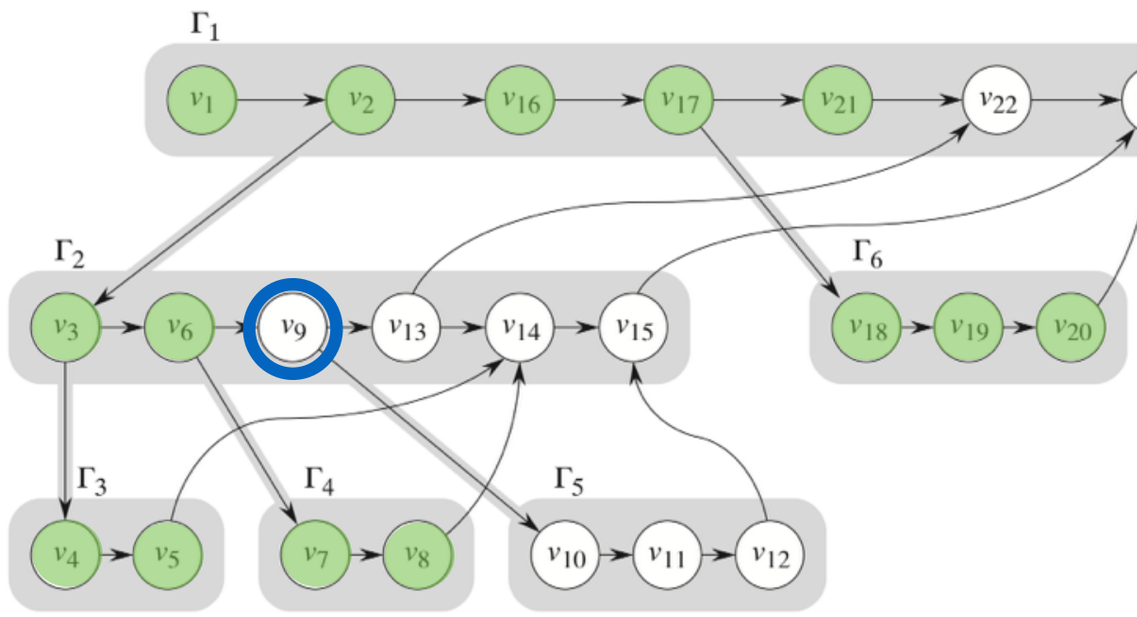


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

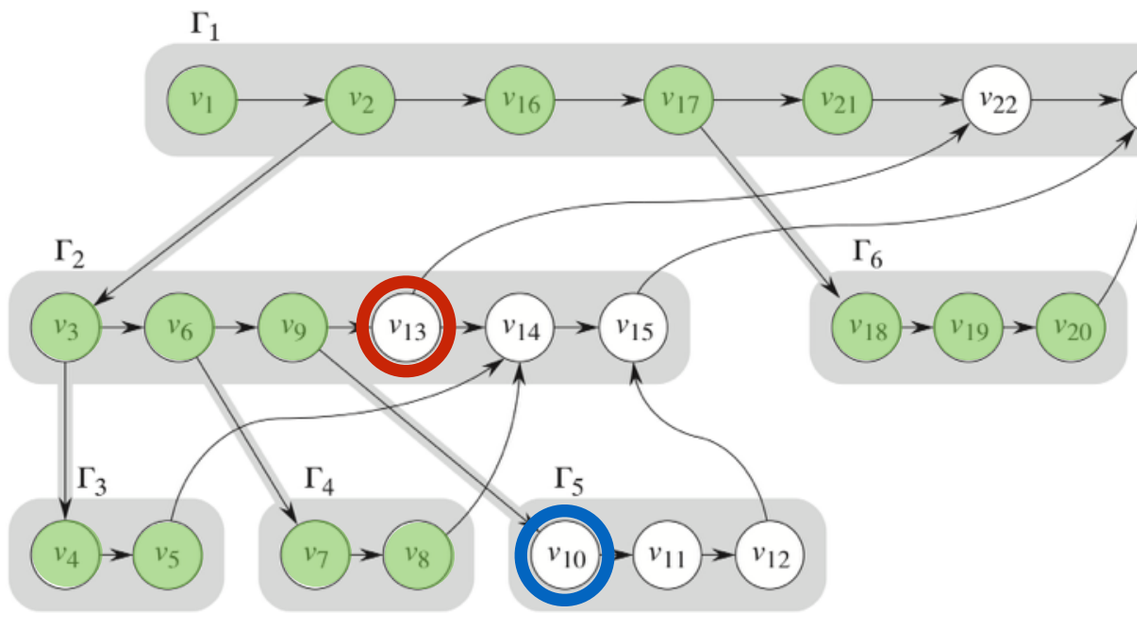


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

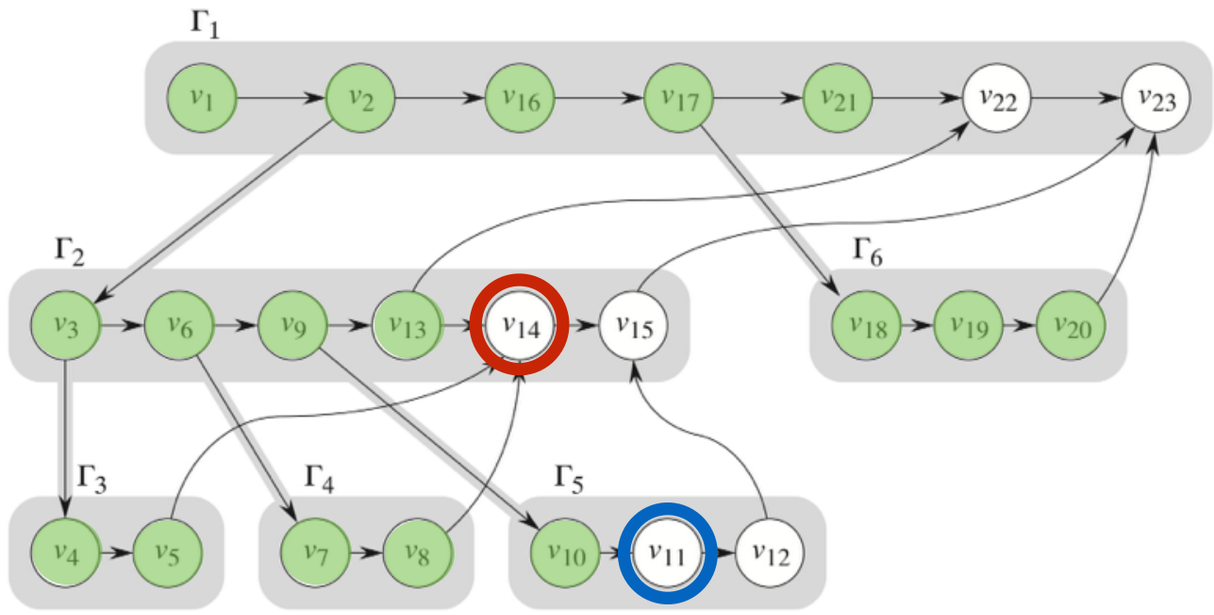


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

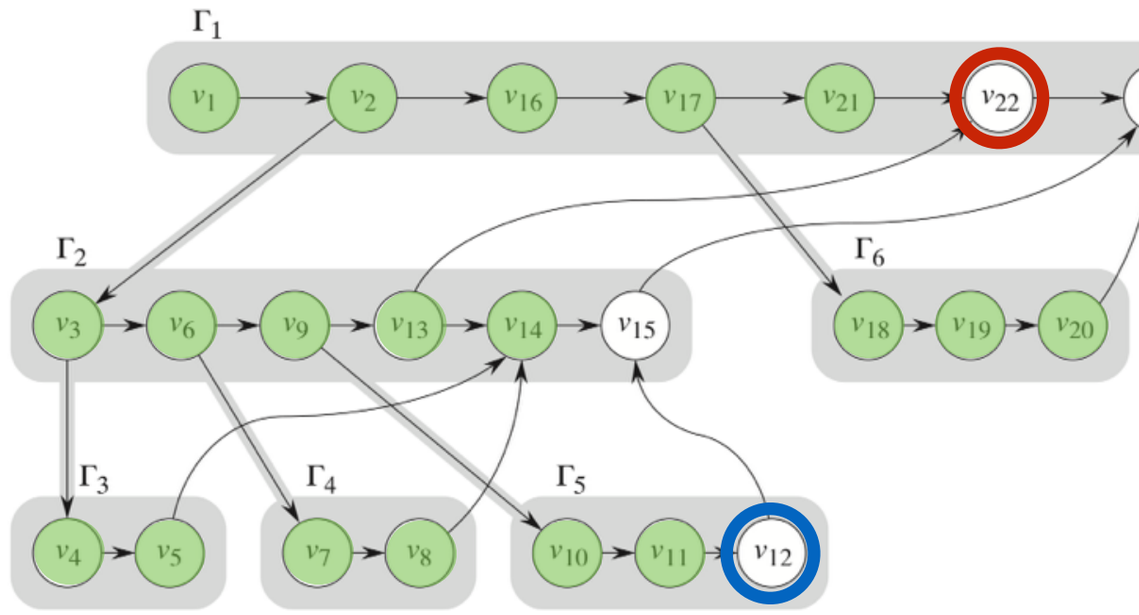


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

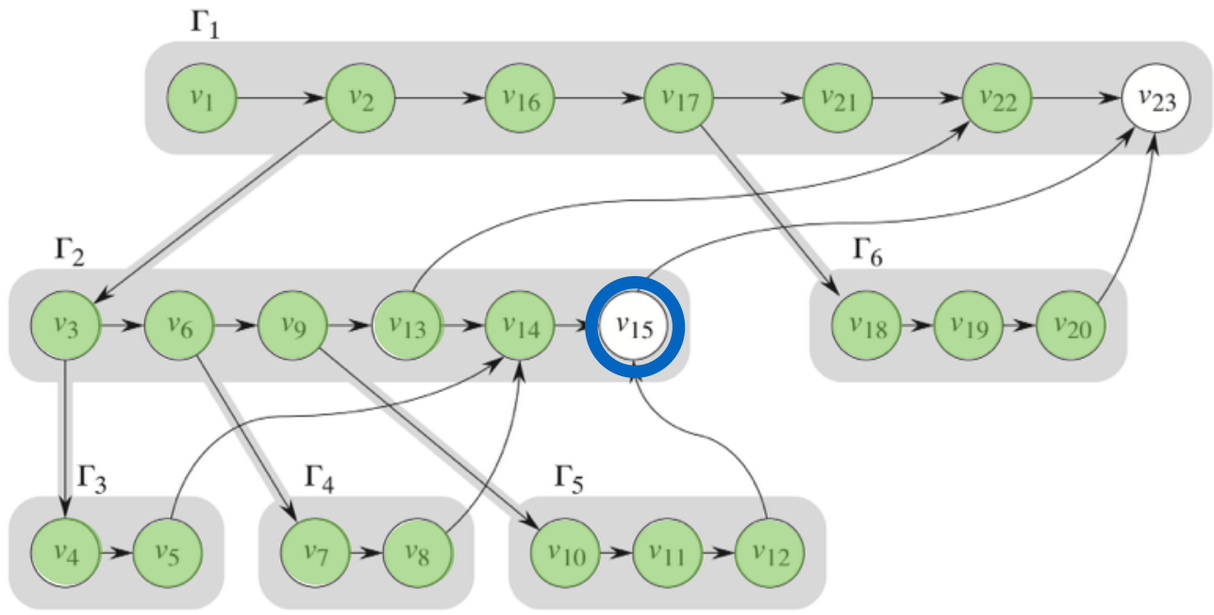


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

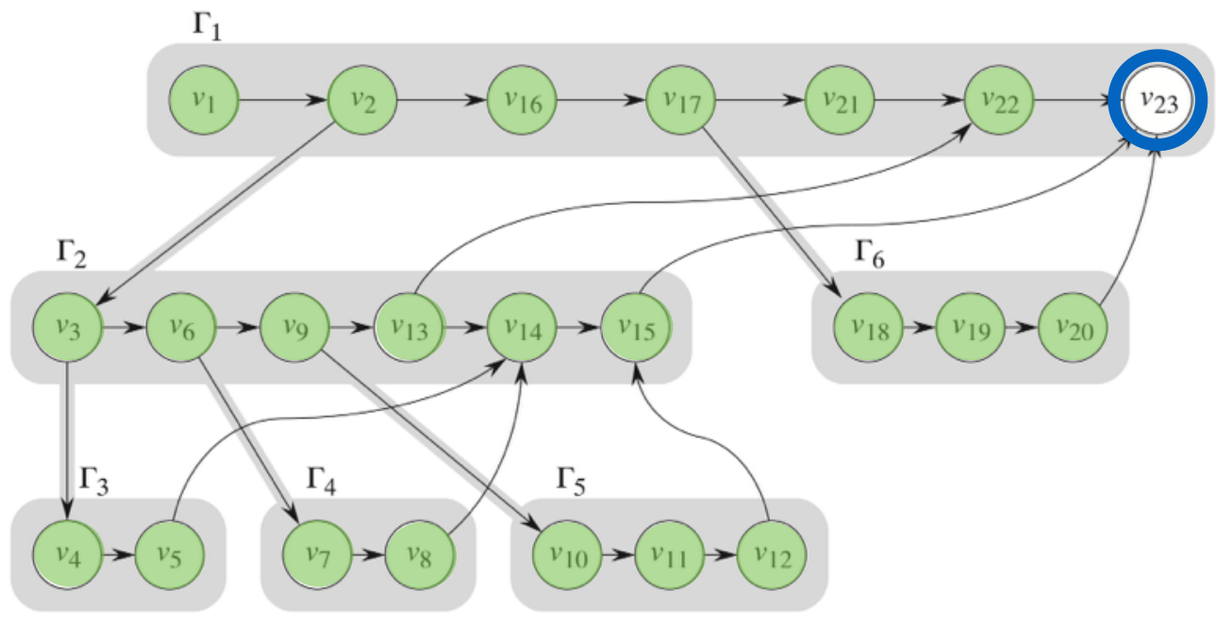


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves scheduler

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

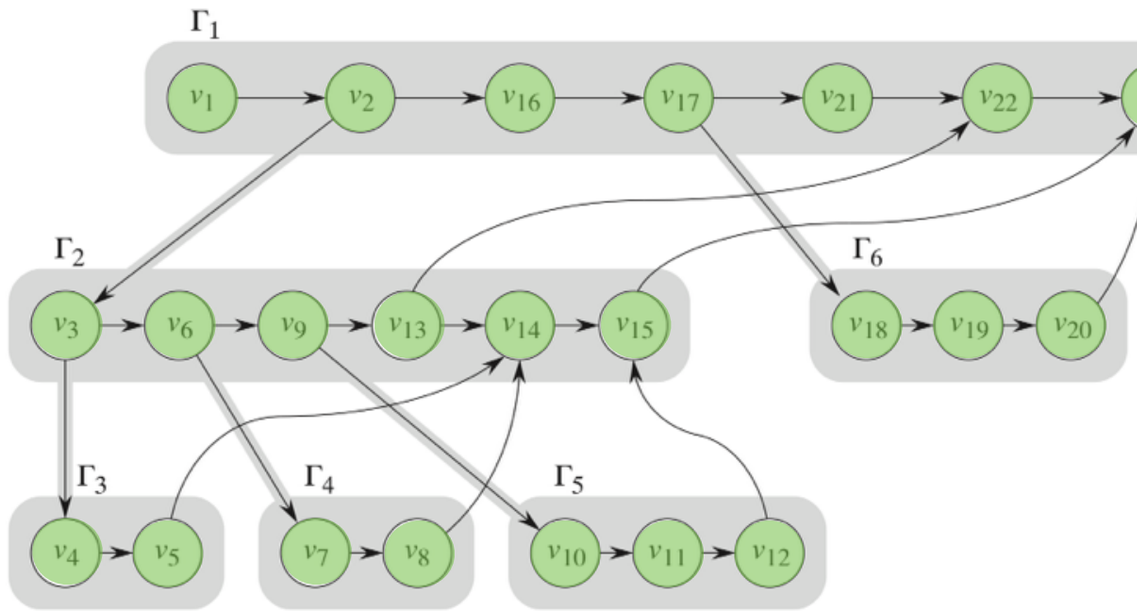


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Busy-leaves analysis

- **Lemma 2**

  For any multithreaded computation with stack depth S_1, any P-processor execution X that maintains the busy-leaves property uses space bounded by $S(X) \leq S_1 P$.

- Going down call chain, stack space bounded by sequential space usage times number of processors
- All leaves are active, can only have as many leaves as processors

Busy-leaves analysis

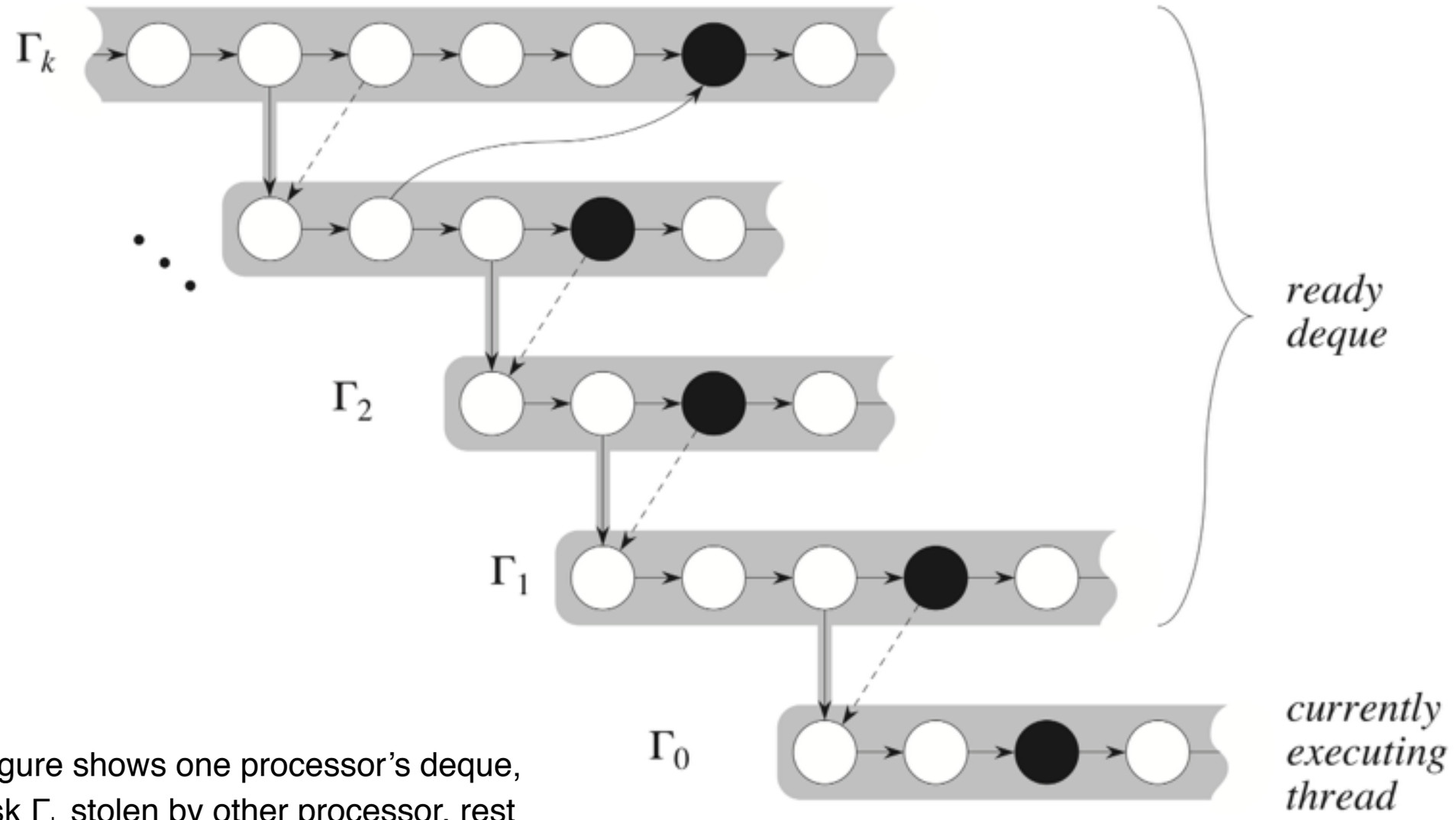
- **Theorem 3**
  For any number P of processors and any strict multithreaded computation with work T_1, critical-path length T_∞, and stack depth S_1, the Busy-Leaves Algorithm computes a P-processor execution schedule X whose execution time satisfies $T(X) \le T_1/P + T_\infty$ and whose space satisfies $S(X) \le S_1 P$.

- Space bound from Lemma 2
- Time bound from Theorem 1 (greedy scheduling)
- Does not account for task pool contention
- Excludes communication overhead

Work-stealing scheduler

- Maintains busy-leaves property
  - Keeps time and space bounds
- Reduces work pool contention
  - Distributes the work pool
  - Keeps local ready task deque on each processor

Work-stealing scheduler

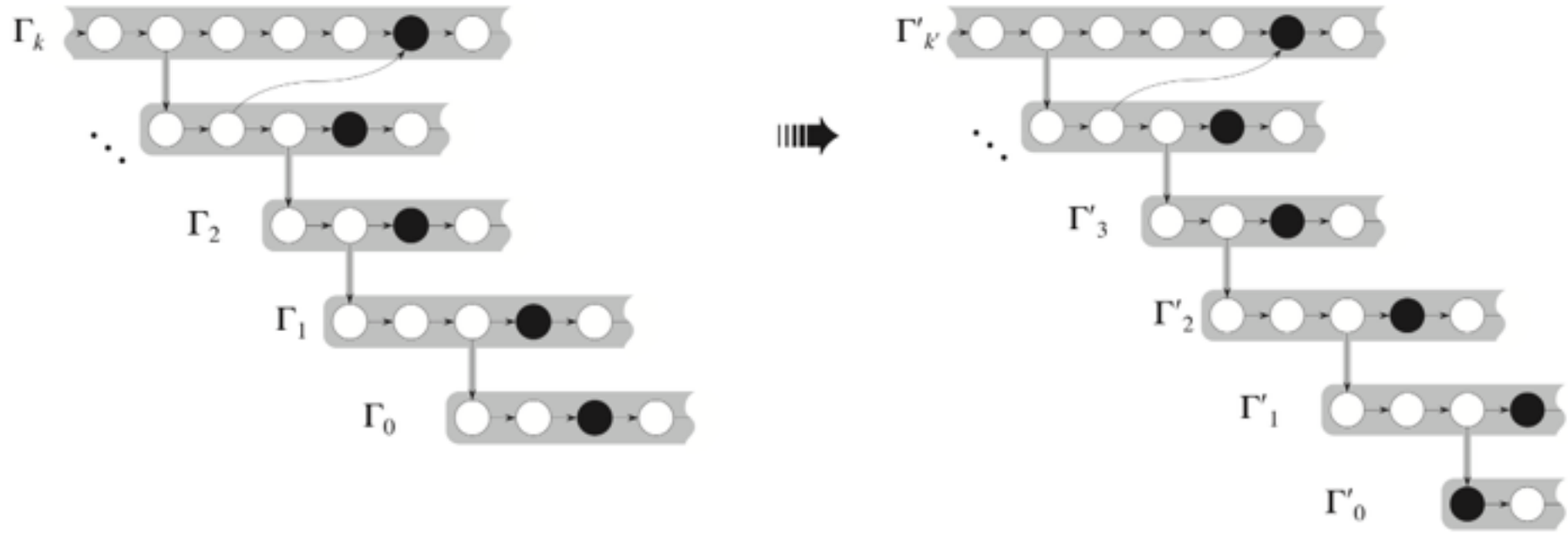


Note: figure shows one processor's deque, with task Γ_k stolen by other processor, rest of ready instructions are right after spawn

Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Work-stealing scheduler

- **Spawns:** When a task is spawned, put the current task on the top of the local deque and then start the new task



(a) Before spawn. (b) After spawn.

Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Work-stealing scheduler

- **Stalls/Dies:** When a processor is idle, get a ready task from the local deque, or if empty steal a task from the top of a random processor's deque

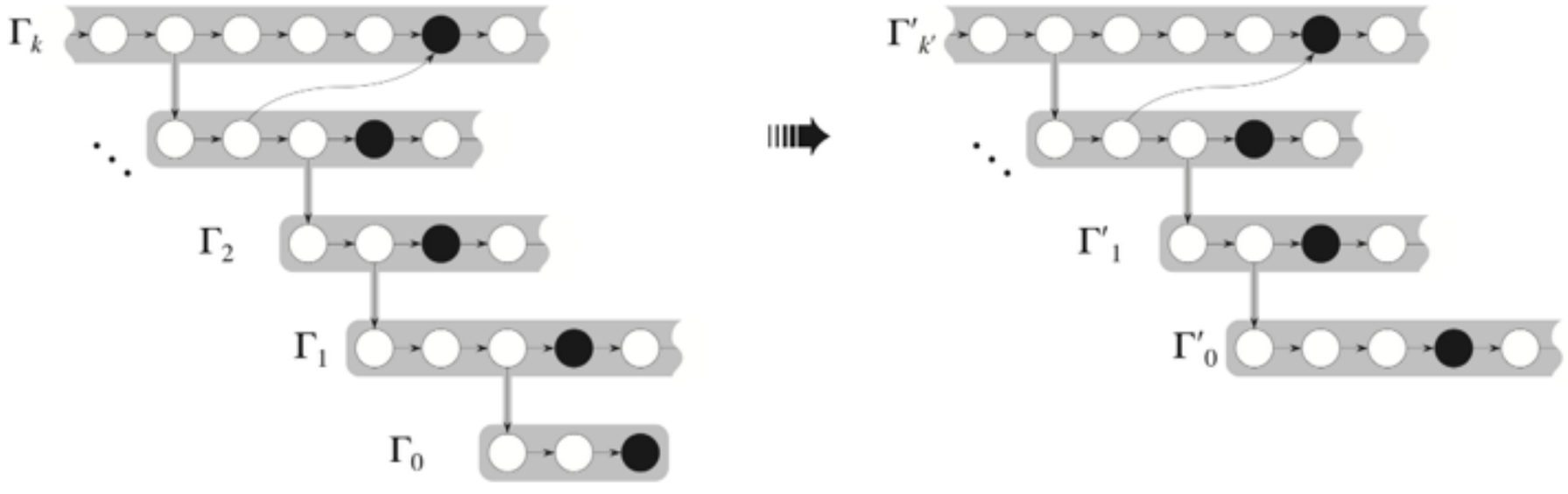


Figure: Scheduling multithreaded computations by work stealing, Blumofe, Robert D. and Leiserson, Charles E. Journal of the ACM 46, 5 (Sep. 1999), 720-748. DOI=http://doi.acm.org/10.1145/324133.324234

Work-stealing scheduler

- **Enables:** When a task becomes ready (i.e., when all of its children have completed), place it on the bottom of the deque of the processor which made it become ready (i.e., the deque of the processor that completed its last child)

Work-stealing space analysis

- **Lemma 2**

  For any multithreaded computation with stack depth S_1, any P-processor execution X that maintains the busy-leaves property uses space bounded by $S(X) \leq S_1 P$.

- Work-stealing maintains busy-leaves property, so its space bound holds

Work-stealing time analysis

- **Theorem 13**
  Consider the execution of any fully strict multithreaded computation with work T_1 and critical-path length T_∞ by the Work-Stealing Algorithm on a parallel computer with P processors. The expected running time, including scheduling overhead, is
  $T_1/P + O(T_\infty)$.
  Moreover, for any $\varepsilon > 0$, with probability at least $1 - \varepsilon$, the execution time on P processors is
  $T_1/P + O(T_\infty + \log P + \log(1/\varepsilon))$.

- Build up proof of bounds by analogy of
  - balls-and-bins “recycling game” to model steal requests and bound their delays, and
  - buckets-and-dollars with work, steal, and wait buckets to model actions executed by each processor and bound time and communication

Work-stealing communication analysis

- **Theorem 14**
  Consider the execution of any fully strict multithreaded computation with work T_1 and critical-path length T_∞ by the Work-Stealing Algorithm on a parallel computer with P processors. Then, the total number of bytes communicated has expectation
  $O(PT_\infty(1+n_d)S_{max})$
  where n_d is the maximum number of join edges from a thread to its parent and S_{max} is the size in bytes of the largest activation frame in the computation. Moreover, for any $\varepsilon > 0$, the probability is at least $1 - \varepsilon$ that the total communication incurred is
  $O(P(T_\infty + \log(1/\varepsilon))(1+n_d)S_{max})$.

- Maximum number of times a task can be stolen in Cilk is twice the number of spawns it executes
- Steals only occur along the critical path because of busy-leaves strategy

Practical results

- Implemented in Cilk, TBB, Habanero-Java others
- Good results, not really shown in paper
- Needs sufficient parallel slackness (available parallelism for given number of processors)
- Need sufficient task granularity to outweigh overhead

Practical results

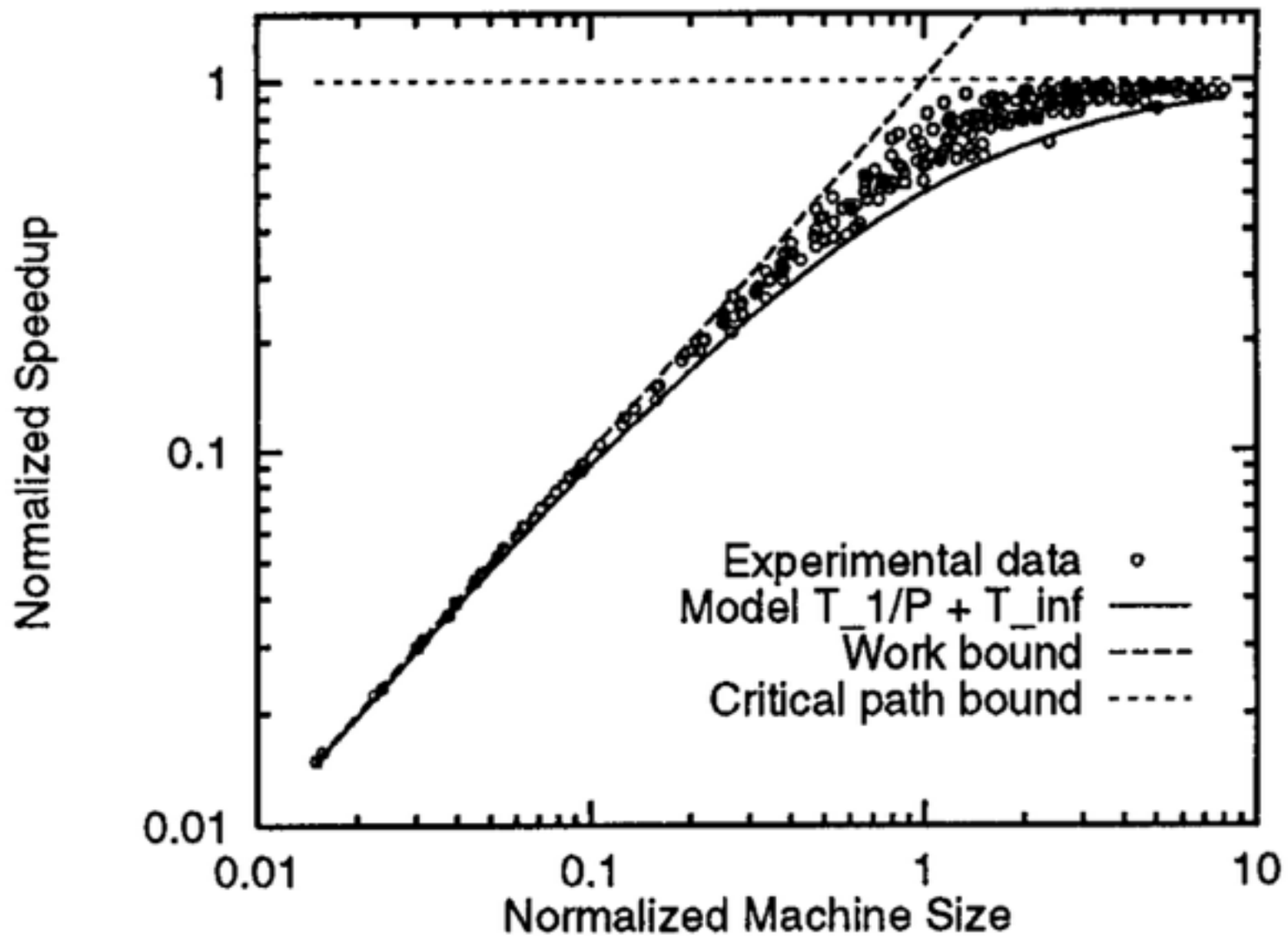


Figure: The Implementation of the Cilk-5 Multithreaded Language by Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 1998.

Conclusion

- Work-stealing very efficient
- Practical to implement with good results
- Linear speedup with only linear growth in space
  - Space bound: S_1P
  - Expected time: $T_1/P + O(T_\infty)$
- Reasonable communication: $PT_\infty(1+n_d)S_{max}$