



# Load Balancing and Scheduling

CS121 Parallel Computing  
Fall 2023



# Load balancing problem

- Goal is to finish a set of tasks as quickly as possible.
  - Requires resources don't idle, i.e. do similar amounts of work.
- Load balancing decides which tasks to perform on which processors.
- Scheduling decides the order of tasks, which affects fairness, responsiveness, etc.
- Load balancing and scheduling have a vast literature in parallel and distributed computing, operating systems, operations research, etc.
  - Many different models for computation and communication, precedence constraints, heterogeneous systems, etc.
- Many packages available, e.g. Cilk, ADLB, Zoltan, Chombo, ParMetis.
- For most scheduling problems, finding optimal solution is intractable.
  - Goal of load balancing is speed, so load balance algorithm itself needs to be fast.
  - Rely on fast heuristics that work well in practice or have approximate performance guarantees.



# Static vs dynamic

- Some applications are static, i.e. the set of tasks in the application, their sizes and communication pattern are known at the start of the execution.
  - Ex Dense and sparse linear algebra, FFT.
  - Load balancing can be done once at beginning of computation.
    - Can afford to spend more time to get higher quality solution.
- For semi-static problems, task information is known periodically at start of some phases.
  - Ex Particle simulations, grid computations.
  - Periodically rebalance when load changes substantially.
    - Requires relatively efficient algorithm.
- For dynamic problems, information is only known at runtime.
  - Ex Search problems.
  - Constantly rebalance on the fly. Need very lightweight methods.
- Can load balance at different granularities.
  - Fine grained task balancing gives best results, but may take too much time and memory.
  - Can group tasks together for coarse grained balancing.



# Centralized vs distributed

- Centralized load balancing gathers all information at one node.
  - Produces better result since global load info available.
  - Central node becomes performance bottleneck.
- Distributed load balancing lets nodes communicate and make own balancing decisions.
  - More scalable. Can react faster to load changes.
  - Hard to achieve globally optimal result.
  - May be slower than centralized if multiple balancing steps required.
- Hierarchical scheme uses centralized node for coarse grained load balancing, then uses distributed nodes for fine grained balance.
  - **Ex** First assign groups of tasks to nodes, then divide each group among the processors.

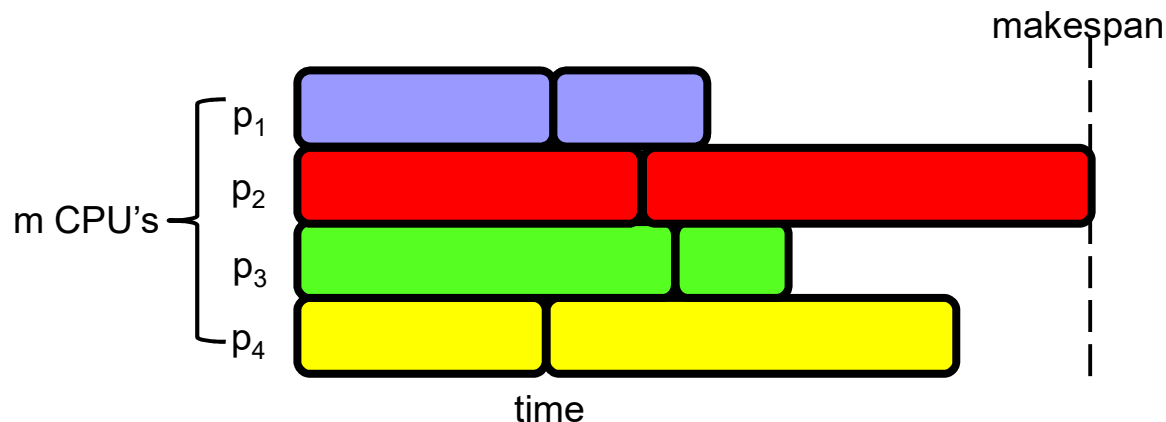


# Other issues

- Load estimation tries to predict the load and communication pattern of a task.
  - In best case, this can be inferred from the code.
  - Otherwise, can profile the task, and assume its future behavior matches the past.
    - Information gathered automatically, without user intervention.
    - Ex Works well for some scientific computations and simulations.
  - Alternatively, can build a model for the task behavior.
    - Labor intensive. Must update model if program changes.
- When load changes, can rebalance by migrating tasks from one node to another.
  - May be costly, because need to move code and possibly data.

# Static load balancing

- Start with a basic model where task sizes are known, there are no precedence constraints between tasks, and ignore communication costs.
  - Even in this model optimal scheduling is NP-hard.
- $n$  independent tasks with different sizes.
  - Tasks can be done in any order.
  - Any task can be done on any processor.
- $m$  processors with the same speed.
  - Each processors can do one task at a time.
- Minimize the makespan, i.e. time when last processor finishes.





# Minimizing makespan is NPC

- Show that minimizing makespan on even two processors is NP-complete.
- Decision version of problem is in NP.
- **SUBSET-SUM problem**: Given a set of numbers  $S$  and target  $t$ , is there a subset of  $S$  summing to  $t$ ?
  - **Ex**  $S=\{1,3,8,9\}$ . For  $t=9$ , yes. For  $t=14$ , no.
  - SUBSET-SUM is NP-complete. Will reduce it to 2 processor makespan scheduling.
- Let  $(S,t)$  be an instance of SUBSET-SUM, and let  $s$  be sum of all elements in  $S$ .
- Make a set of tasks  $J = S \cup \{s-2t\}$ , and schedule them on 2 processors.
- Show that SUBSET-SUM reduces to min makespan, i.e. SUBSET-SUM has a solution iff min makespan has a certain solution.



# Minimizing makespan is NPC

- **Claim** If some subset of  $S$  sums to  $t$ , then min makespan is  $s-t$ .
- **Proof** Say  $S' \subseteq S$  sums to  $t$ . Schedule the tasks in  $S'$  and task  $s-2t$  on processor 1. So processor 1 finishes at time  $t+s-2t=s-t$ . Processor 2 does the tasks in  $S-S'$ , so it finishes at time  $s-t$  as well. Since processors finish at same time, the makespan is minimal.
- **Claim** If the min makespan is  $s-t$ , there exists a subset of  $S$  that sums to  $t$ .
- **Proof** Suppose WLOG processor 1 does the  $s-2t$  task. Since makespan is  $s-t$ , the other tasks processor 1 does must have total size  $s-t-(s-2t)=t$ .
- So  $(S,t)$  is yes instance of SUBSET-SUM iff minimum makespan =  $s-t$ , so minimizing makespan is NPC.





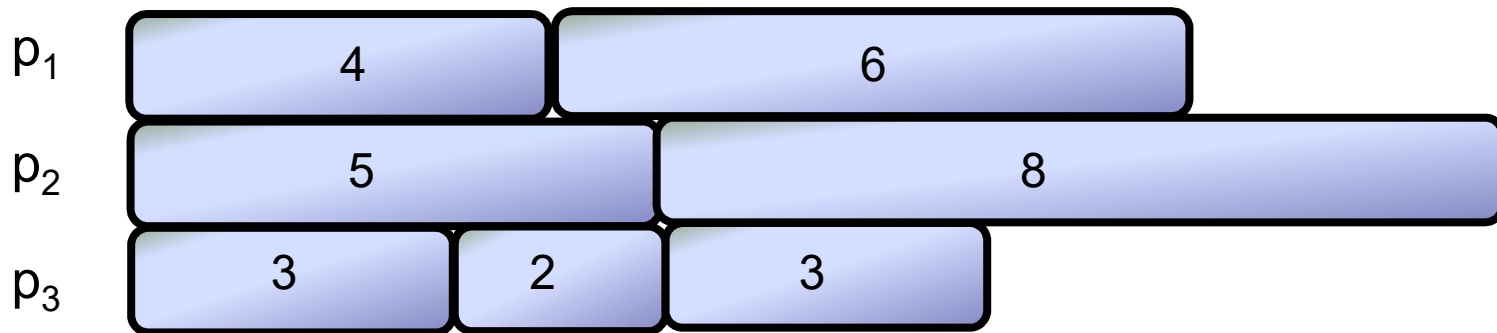
# List scheduling

- Since scheduling is NPC, it's unlikely we can find the min makespan in polytime.
- List scheduling (Graham) is a simple greedy algorithm that finds a schedule with makespan at most twice the minimum.
  - Call this a 2-approximation.
- If there are precedence constraints, we can modify list scheduling to allocate a task whenever it's available, i.e. all its preceding tasks are finished.
  - This still gives a 2-approximation, but we won't prove it.

- List the tasks in any order.
- While there are unfinished tasks.
  - If any processor is idle, give it the next task in the list.

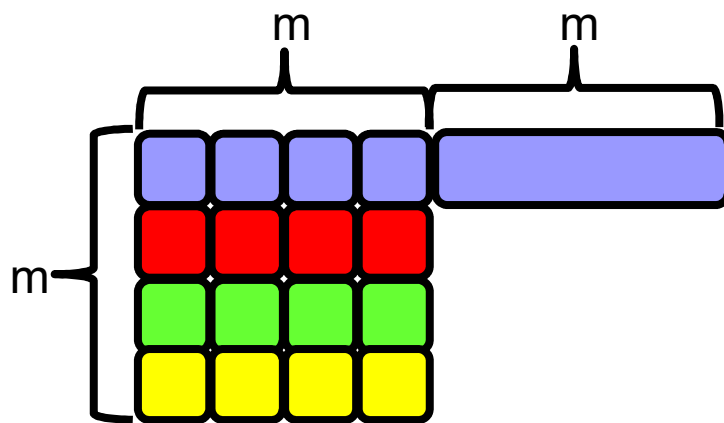
# Example

- 3 processors. The tasks have sizes 2, 3, 3, 4, 5, 6, 8.
- List tasks in any order. Say 4, 5, 3, 2, 6, 8, 3.
- All processors finishes by time 13, so makespan = 13.

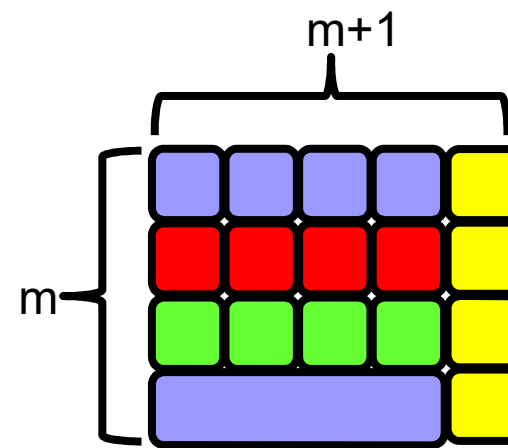


# Worst case for LS

- How badly can list scheduling do compared to optimal?
- Say there are  $m^2$  tasks with length 1, and one task with length  $m$ .
  - Suppose they're listed in the order  $1, 1, 1, \dots, 1, m$ .
  - LS has makespan  $2m$ . Optimal makespan is  $m+1$ .
  - $\text{makespan}(\text{LS}) / \text{makespan}(\text{opt}) = 2m/(m+1) \approx 2$ .
- This is worst possible case for list scheduling.
- **Thm** Suppose the optimal makespan is  $M^*$ , and LS produces a schedule with makespan  $M$ . Then  $M \leq 2M^*$ .



$\text{makespan}(\text{LS}) = 2m$



$\text{makespan}(\text{opt}) = m+1$



# LPT scheduling

- Worst case for LS occurred when longest job was scheduled last.
  - Large jobs are “dangerous” at end.
- Let's try to schedule longest jobs first.
- Longest processing time (LPT) schedule is just like list scheduling, except it first sorts tasks by nonincreasing order of size.
- **Ex** For three processors and tasks with sizes 2, 3, 3, 4, 5, 6, 8, LPT first sorts the jobs as 8,6,5,4,3,3,2. Then it assigns  $p_1$  tasks 8,3,  $p_2$  tasks 6,3,  $p_3$  tasks 5,4,2, for a makespan of 11.
- LPT has an approximation ratio of  $4/3$ .
- LS still has two advantages.
  - It can schedule tasks dynamically, as they're generated on the fly.
  - It doesn't need to know the sizes of the tasks.

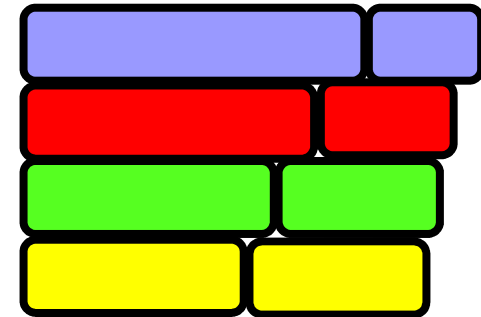


# LPT is a $4/3$ -approximation

- **Thm** Suppose the optimal makespan is  $M^*$ , and LPT produces a schedule with makespan  $M$ . Then  $M \leq 4/3 M^*$ .
- Let  $X$  be the last job to finish. Assume it starts at time  $T$  and has size  $t$ .
- Assume WLOG that  $X$  is the last job to start.
  - If not, then say  $Y$  starts after  $T$ .
  - $Y$  finishes before  $T+t$ . So we can remove  $Y$  without increasing the makespan.
- **Cor 1**  $X$  is the smallest job.
  - $X$  is the last job to start, so due to LPT scheduling it's the smallest.

# LPT is a $4/3$ -approximation

- **Claim 1** LPT's makespan =  $T+t \leq M^* + t$ .
  - No processor is idle up to time  $T$ .
  - So total work from all jobs is so  $W \geq T \cdot p$ , and  $M^* \geq W / p = T$ .
- **Case 1**  $t \leq M^*/3$ .
  - Then LPT's makespan  $\leq M^* + t \leq M^* + M^*/3 = 4/3 M^*$ .
- **Case 2**  $t > M^*/3$ .
  - Since  $X$  is the smallest task, all tasks have size  $> M^*/3$ .
  - So the optimal schedule has at most 2 tasks per processor. So  $n \leq 2m$ .
  - If  $1 \leq n \leq m$ , then LPT and optimal schedule both put one task per processor.
  - If  $m < n \leq 2m$ , then optimal schedule is to put tasks in nonincreasing order on processors  $1, \dots, m$ , then on  $m, \dots, 1$ .
    - LPT also schedules tasks this way, so it's optimal.



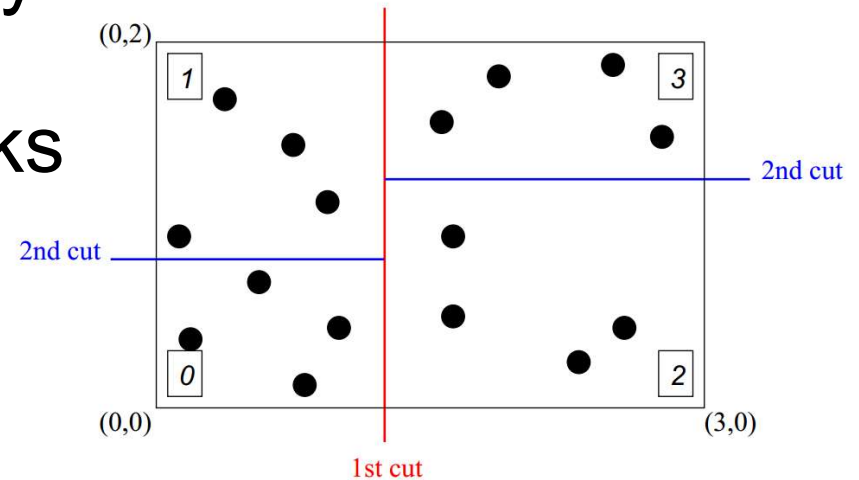


# Geometric load balancing

- In many parallel applications, tasks have geometric coordinates, and nearby tasks communicate with each other.
  - **Ex** In a particle simulation, nearby particles interact.
  - Assume a static or semi-static setting, where tasks have same size.
- We want to load balance and also minimize communication.
  - Want to place nearby tasks on same processor.
  - Still assume the task sizes are known.
- Represent each task by a point at some coordinates.
- Partition the points into  $m$  sets. Assign tasks in each set to one processor.

# Recursive bisection

- First partition tasks evenly in the x direction.
- In each half, partition tasks evenly in the y direction.
- In each quarter, partition tasks evenly in the x direction. Etc.
- This might lead to partitions with large aspect ratios, causing many communicating tasks to lie in different partitions.

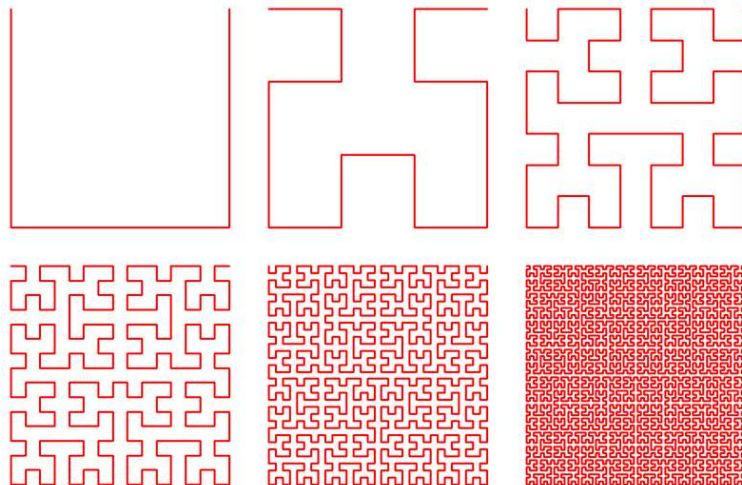
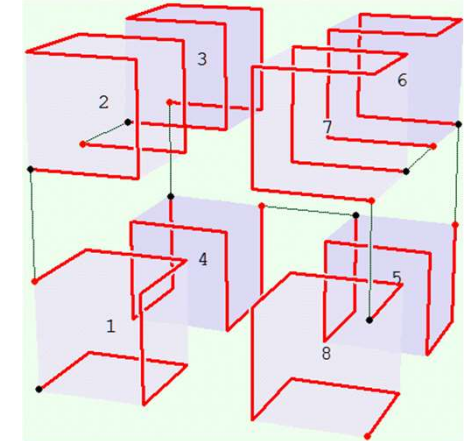
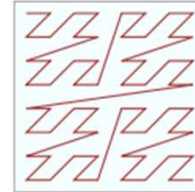
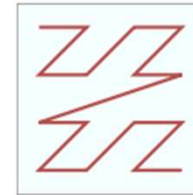
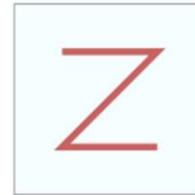


Source: New Challenges in Dynamic Load Balancing, Devine et al.



# Space filling curve

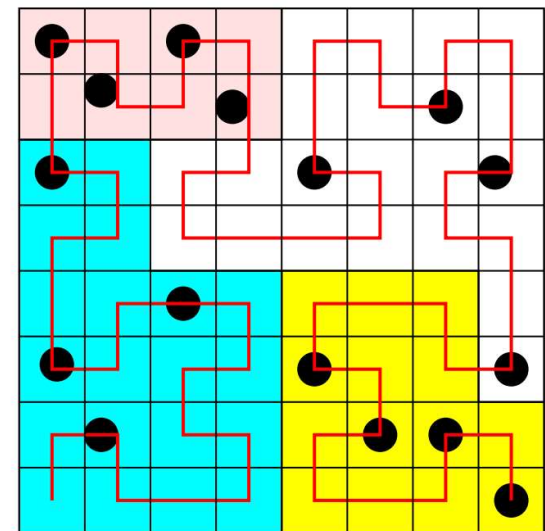
- A space filling curve (SFC) is a 1-D curve that passes through all the points in a discrete / continuous space.
- SFC's have good locality properties, i.e. nearby points in the SFC are nearby in space, and usually vice versa.
- Many types of SFC's, e.g. Morton (Z-order) and Hilbert curves.
- SFC's can be generalized to higher dimensions.



	Starting curve	Transformation 1	Transformation 2	Transformation 3	Transformation 4
$H_1$					
$H_2$					
$H_3$					
$H_4$					

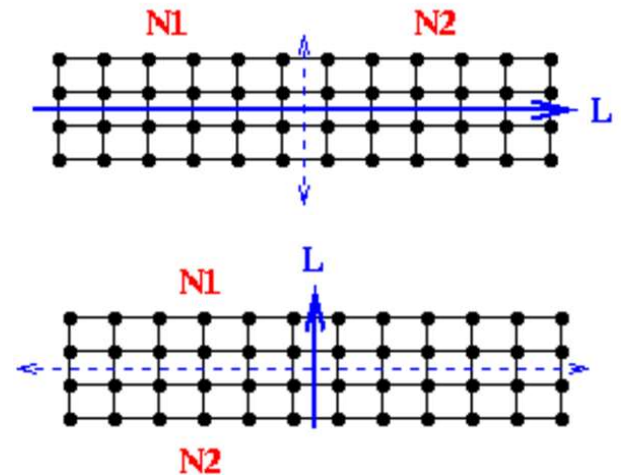
# Space filling curve partitioning

- We can use an SFC to map nodes onto a 1D line.
- Then we partition nodes along the line evenly.
- Given a node, there are efficient algorithms to determine which partition it lies in.
- Given a box, can also efficiently enumerate all nodes lying inside the box.
- These operations are useful for particle simulations and collision detection.

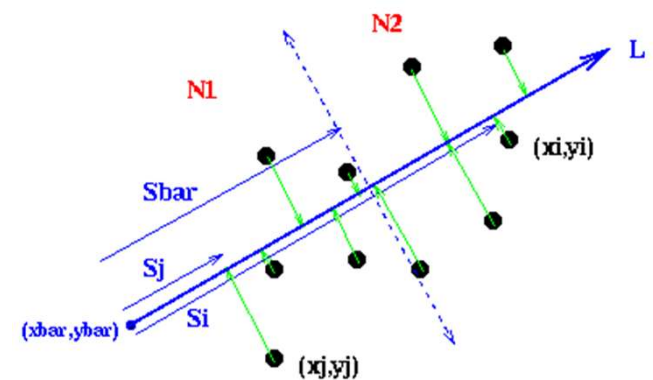


# Inertial partitioning

- The top partitioning (across the dotted line) is better than the bottom one, because it cuts fewer communicating pairs of nodes.
- Intuitively, we want to find a line  $L$  that minimizes the moment of inertia of the nodes rotating around  $L$ .
  - A closed form solution exists for the optimal  $L$ .
- Once we have  $L$ , project all the nodes onto  $L$ , then find the median.
  - Partition nodes based on which side of the median their projection lies.
- This produces two partitions. For  $m$  partitions, apply the partitioning recursively.



Inertial Partitioning in 2D



Source: <https://people.eecs.berkeley.edu/~demmel/cs267/lecture18/lecture18.html>

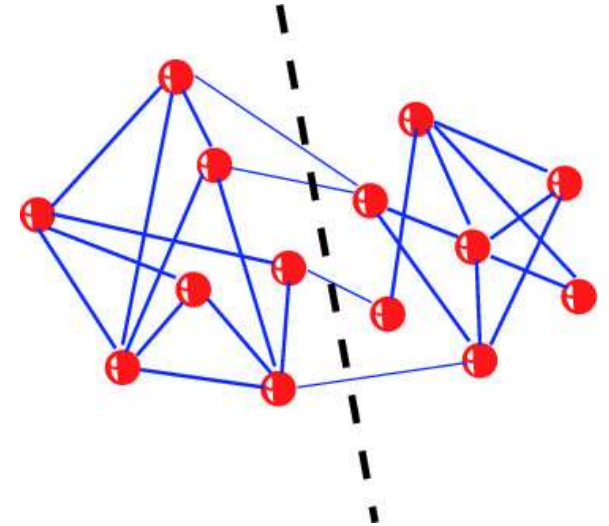


# Graph based partitioning

- In geometric partitioning, the communicating tasks were defined implicitly based on proximity.
- In graph based partitioning, we're given an explicit graph showing the pairs of communicating nodes.
  - Graph nodes can be weighted based on size of the task.
  - Graph edges can be weighted based on amount of communication.
- Want to partition nodes of graph in two parts, and map each part onto a processor.
  - If we have more than two processors, do the partitioning recursively.
  - Want each part to have approximately same number / weight of nodes, for load balance.
  - Want to minimize number of edges cut (i.e. crossing between partitions), because these represent communication between processors.

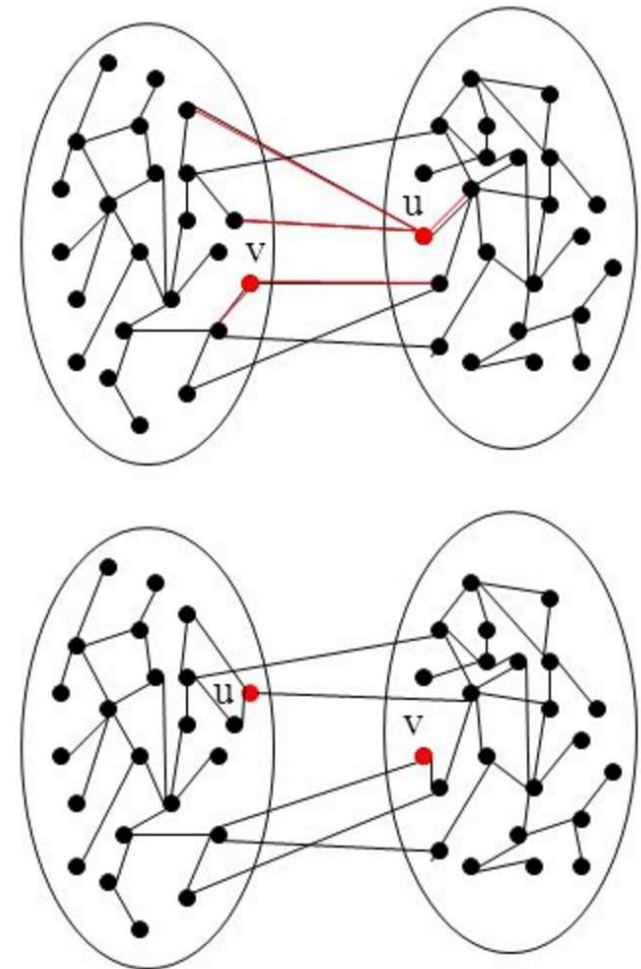
# Graph based partitioning

- Computing optimal partitioning is NP-complete, so we have to settle for heuristics.
  - Local search (e.g. Kernighan-Lin).
  - Spectral.
  - Multilevel.



# Kernighan-Lin partitioning

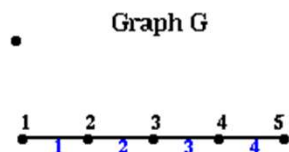
- Greedily improve a partition by swapping nodes until no improvement possible.
- Start with an arbitrary partition  $A, B$ , each with half the nodes.
- For  $i=1, \dots, n/2$  iterations.
  - Choose  $a_i \in A$  and  $b_i \in B$  s.t.  $a_i$  and  $b_i$  have never been swapped before, and swapping  $a_i$  and  $b_i$  results in smallest cut.
  - Let  $C_i$  be the cost of the partition after swapping  $a_i$  and  $b_i$ .
- Choose the  $C_i$  with the lowest cost.
- If  $C_i$ 's cost is lower than cost of  $(A, B)$ , restart the algorithm with partition  $C_i$ .
- Otherwise return  $C_i$ .
- In practice KL is fast and returns reasonably good partitions.



# Graph Laplacian

- Given an undirected graph  $G$ , define the Laplacian matrix  $L(G)$ 
  - For each edge  $(i,j)$  in  $G$ , set entry  $(i,j)$  to  $-1$  in  $L(G)$ .
  - For each node  $i$ , set entry  $(i,i)$  of  $L(G)$  to  $d$ , where  $d$  is the degree of  $i$ .
  - Set all other entries to 0.
- Similar to adjacency matrix, but has interesting spectral properties.

Incidence and Laplacian Matrices

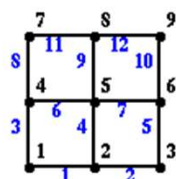


Incidence Matrix  $In(G)$

$$\begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} -1 & & & \\ 1 & -1 & & \\ & 1 & -1 & \\ & & 1 & -1 \\ & & & 1 \end{bmatrix} \end{matrix}$$

Laplacian Matrix  $L(G)$

$$\begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 1 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -1 & 1 \end{bmatrix} \end{matrix}$$



$$\begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{matrix} & \begin{bmatrix} -1 & & & & 1 & & & & & & & \\ 1 & -1 & & & & 1 & & & & & & \\ & 1 & -1 & & & & 1 & & & & & \\ & & 1 & -1 & & & & 1 & & & & \\ & & & -1 & 1 & -1 & & & 1 & & & \\ & & & & -1 & 1 & -1 & & & 1 & & \\ & & & & & -1 & 1 & & & & -1 & \\ & & & & & & -1 & 1 & -1 & & & \\ & & & & & & & -1 & 1 & -1 & & \\ & & & & & & & & -1 & 1 & & \end{bmatrix} \end{matrix}$$

$$\begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{matrix} & \begin{bmatrix} 2 & -1 & & & & & & & \\ -1 & 3 & -1 & & & & & & \\ & -1 & 2 & & & & & & \\ & -1 & & 3 & -1 & & & & \\ & & -1 & -1 & 4 & -1 & & & \\ & & & -1 & -1 & 3 & & & -1 \\ & & & & -1 & & 2 & -1 & \\ & & & & & -1 & -1 & 3 & -1 \\ & & & & & & -1 & -1 & 2 \end{bmatrix} \end{matrix}$$

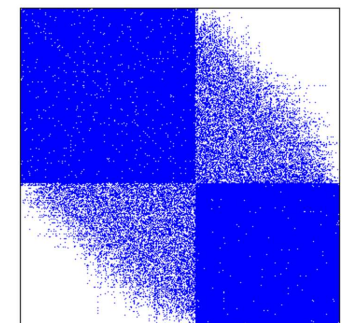
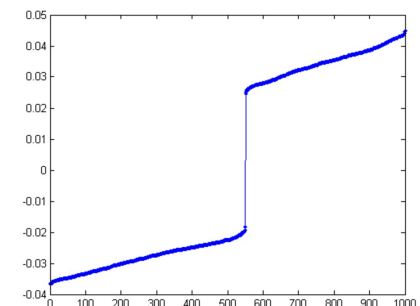
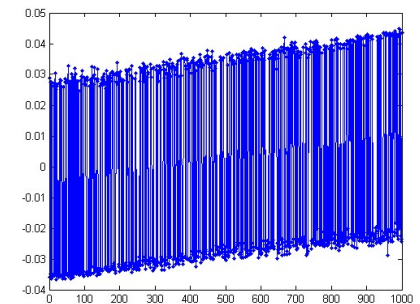
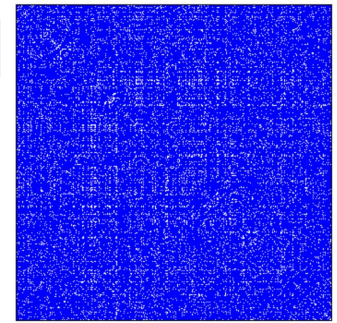
Nodes numbered in black  
Edges numbered in blue

Source: <https://people.eecs.berkeley.edu/~demmel/cs267/lecture18/lecture18.html>



# Spectral partitioning

- **Fact**  $L(G)$  is positive semidefinite, and all the eigenvalues of  $L(G)$  are real and nonnegative.
- Let  $(x_1, x_2, \dots, x_n)$  be the eigenvector of  $L(G)$  corresponding to the second smallest eigenvalue.
- Partition  $G$  as  $A = \{i : x_i \leq 0\}$  and  $B = \{i : x_i > 0\}$ .
- Usually produces better partitions than Kernighan-Lin.
- But finding second eigenvector is quite expensive.
  - Suffices to find approximate eigenvector. But this is still costly.



Source: <https://www.cs.purdue.edu/homes/dgleich/demos/matlab/spectral/spectral.html>

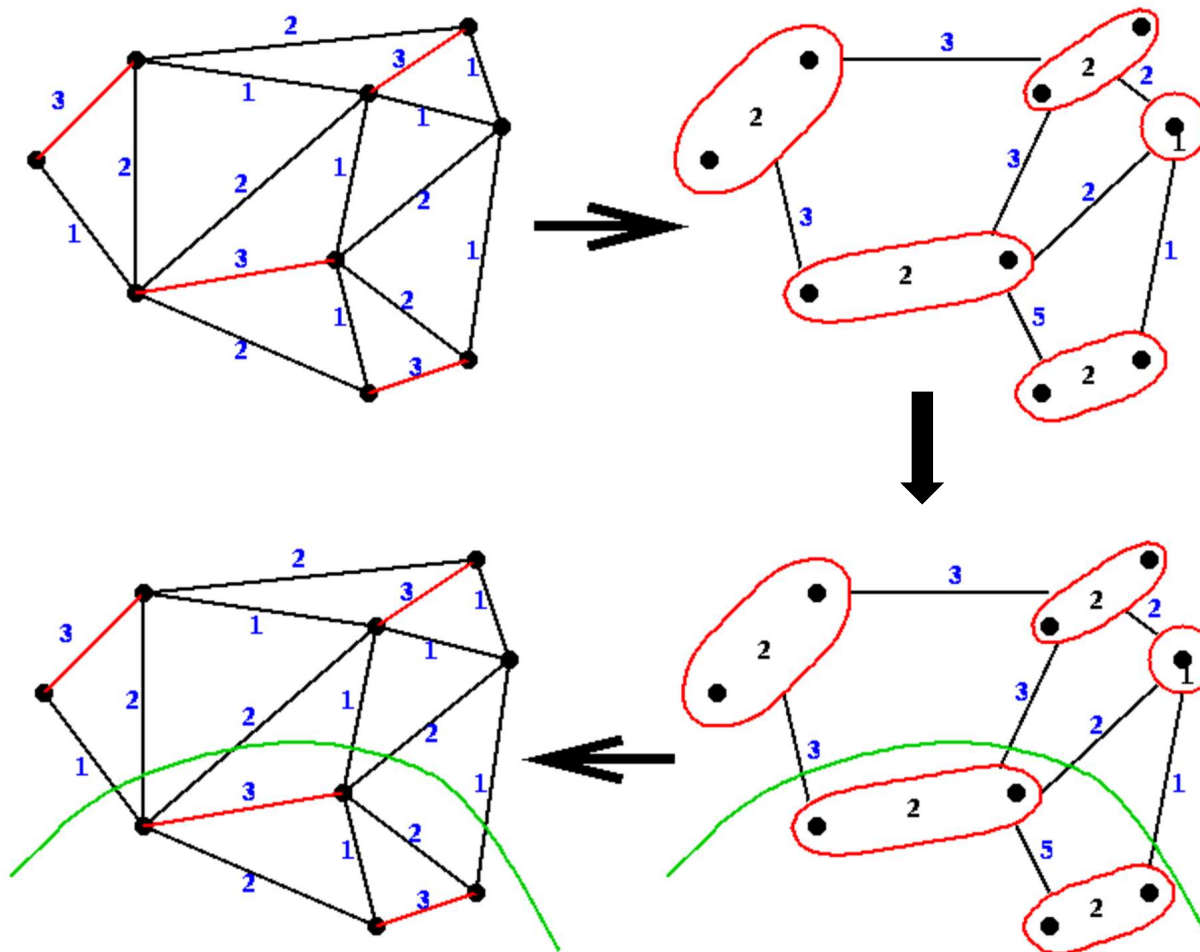




# Multilevel partitioning

- Spectral, and even KL partitioning are too slow on very large graphs.
- To speed them up we run them on coarsened versions of the task graph.
- In fact, we coarsen the graph several times, until the number of nodes is small. Then we partition the coarse graph. Finally, we recover a partition on the fine graph using the coarse partition.
  - During the recovery, we can refine the coarse partitioning, by e.g. using it as the starting guess for Kernighan-Lin.
- Multilevel schemes achieve good quality and speed in practice.

# Multilevel partitioning



- ❑ One way to coarsen a graph is based on matchings.
- ❑ First, find a maximal weight matching greedily.
- ❑ Collapse matched nodes.
- ❑ Merge edges connected to matched nodes.
- ❑ After partitioning the coarse graph, expand the merged edges to recover partition in the original graph.
- ❑ Can refine the partition using e.g. Kernighan-Lin.



# Dynamic load balancing

- In some applications tasks are created by processes dynamically.
  - Ex Search algorithms. Recursive algorithms.
- Ideally do distributed load balancing, since tasks are created by distributed processes.
- One method is diffusion (aka push).
  - If a process has too many tasks, it sends some to its neighbors. If a neighbor becomes overloaded, it does the same thing.
  - Eventually load spreads out and equalizes.
  - But might take a long time and cause lots of communication.



# Dynamic load balancing

- Another technique is work stealing (aka pull).
  - Processes without work steal work from processes with work.
- In work stealing, each process maintains a double-ended queue (deque).
- Process performs task on the top of the deque.
- If process creates a task, it pushes it onto top of the deque.
- If the process's deque is empty, it needs to load balance.
  - It picks a random other process and steals a task from the bottom of that process's deque.
  - This minimizes (but doesn't completely avoid) contention between the two processes, since one takes tasks from top and one from bottom.
- Work stealing doesn't incur any overhead when processes have tasks.
- Overhead when stealing is also borne by idle processes.
  - In contrast, for work pushing busy processes incur overhead for load balancing.
- Work stealing is used in the Cilk parallel runtime.