Parallel Job Schedulings

Thoai Nam

Scheduling on UMA Multiprocessors

- Schedule:

  allocation of tasks to processors
- Dynamic scheduling
  - A single queue of ready processes
  - A physical processor accesses the queue to run the next process
  - The binding of processes to processors is not tight
- Static scheduling
  - Only one process per processor
  - Speedup can be predicted

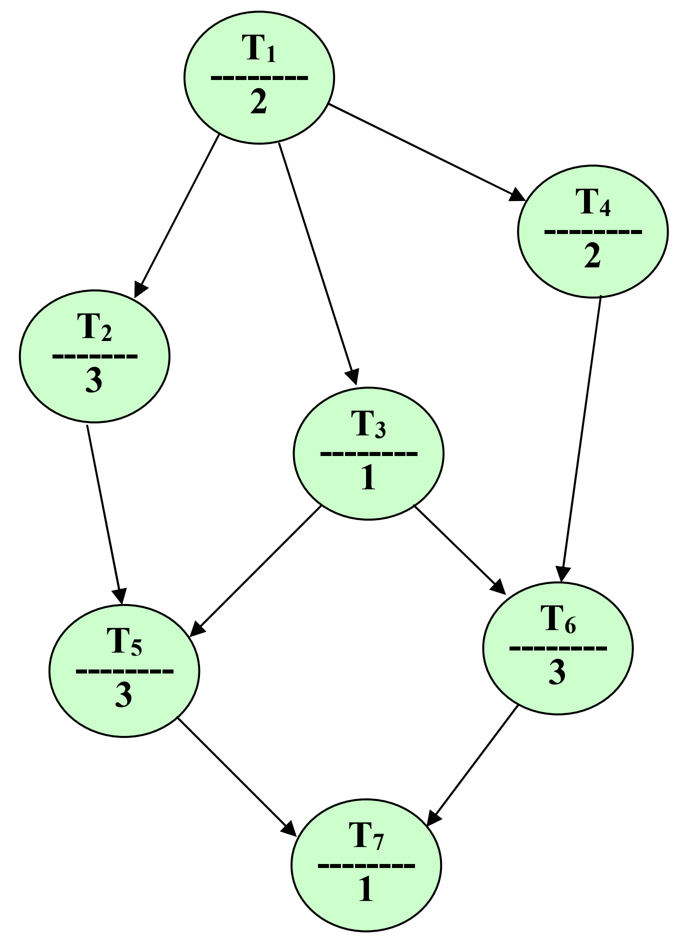
Classes of scheduling

- Static scheduling
  - An application is modeled as an directed acyclic graph (DAG)
  - The system is modeled as a set of homogeneous processors
  - An optimal schedule: NP-complete
- Scheduling in the runtime system
  - Multithreads: functions for thread creation, synchronization, and termination
  - Parallelizing compilers: parallelism from the loops of the sequential programs
- Scheduling in the OS
  - Multiple programs must co-exist in the same system
- Administrative scheduling

Deterministic model

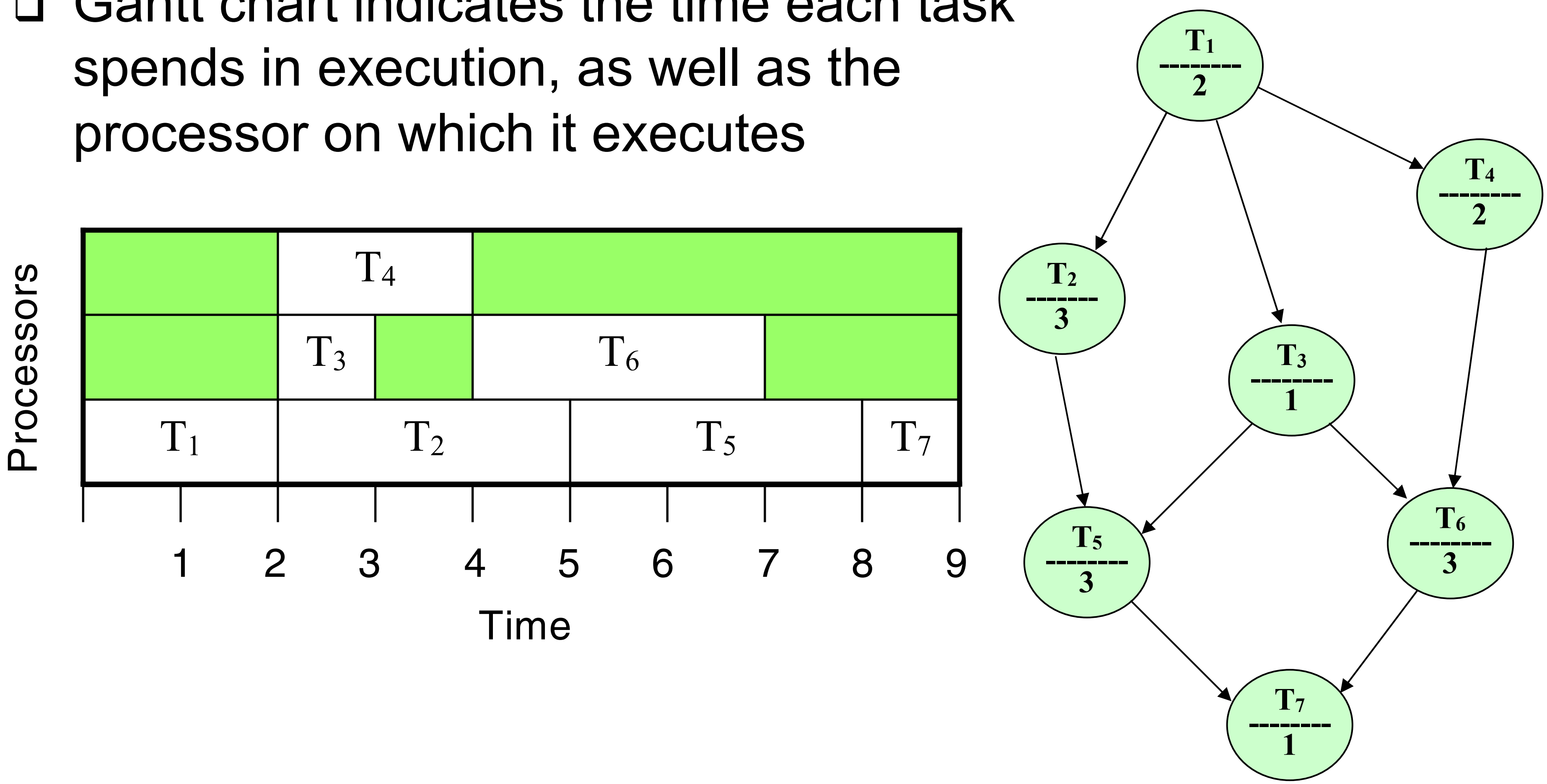
- A parallel program is a collection of tasks, some of which must be completed before others begin
- Deterministic model:

  The execution time needed by each task and the precedence relations between tasks are fixed and known before run time
- Task graph



Gantt chart

- Gantt chart indicates the time each task spends in execution, as well as the processor on which it executes



Optimal schedule

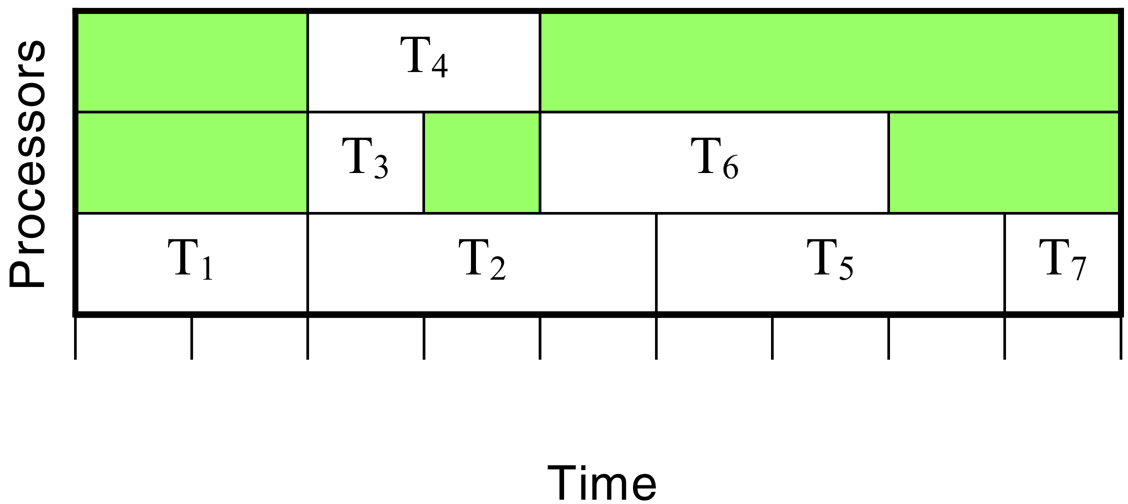
- If all of the tasks take unit time, and the task graph is a forest (i.e., no task has more than one predecessor), then a polynomial time algorithm exists to find an optimal schedule
- If all of the tasks take unit time, and the number of processors is two, then a polynomial time algorithm exists to find an optimal schedule
- If the task lengths vary at all, or if there are more than two processors, then the problem of finding an optimal schedule is NP-hard.

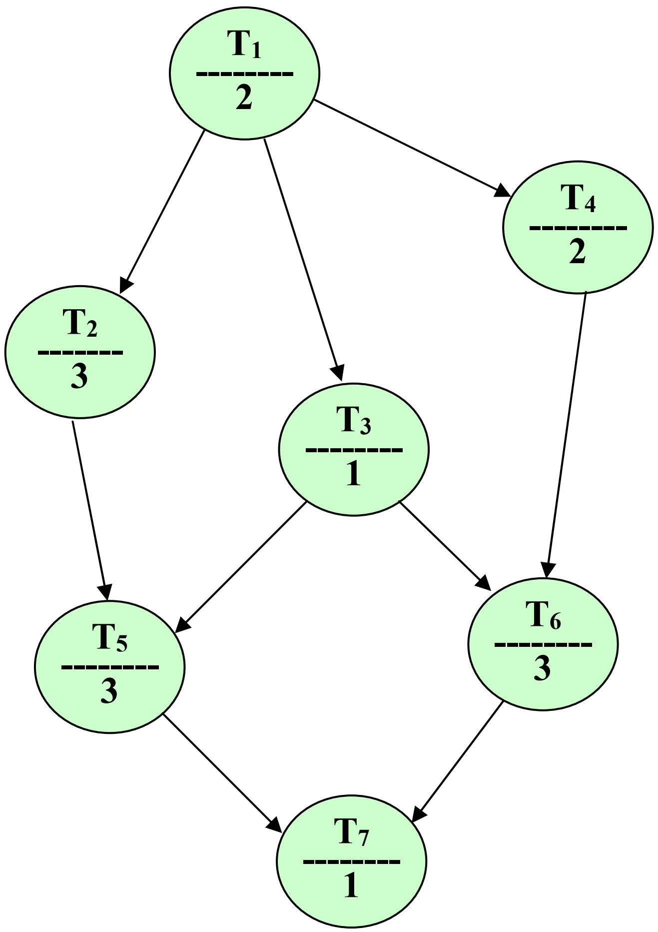
Graham's list scheduling algorithm

- **T** = $\{T_1, T_2, \ldots, T_n\}$
  a set of tasks
- μ: **T** $\rightarrow (0,\infty)$
  a function associates an execution time with each task
- A partial order $\prec$ on **T**
- **L** is a list of task on **T**
- Whenever a processor has no work to do, it instantaneously removes from **L** the first ready task; that is, an unscheduled task whose predecessors under $\prec$ have all completed execution. (The processor with the lower index is prior)

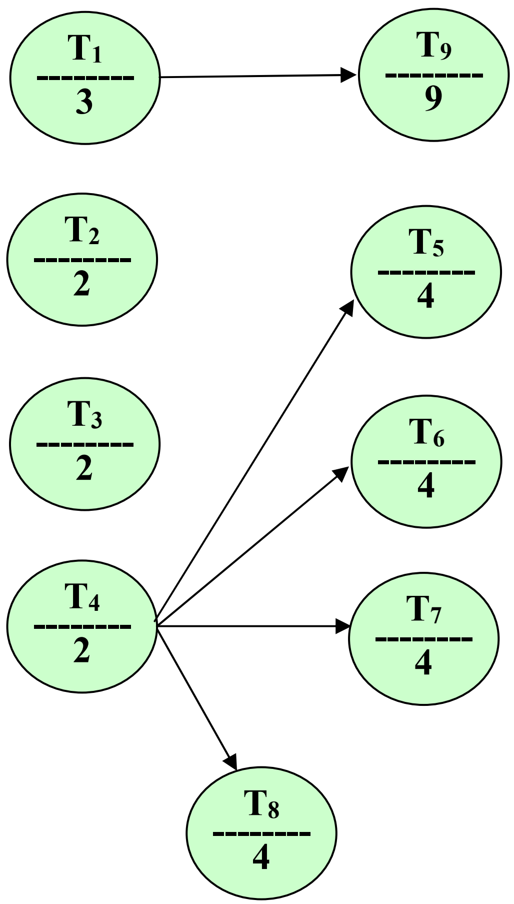
Graham's list scheduling algorithm - Example

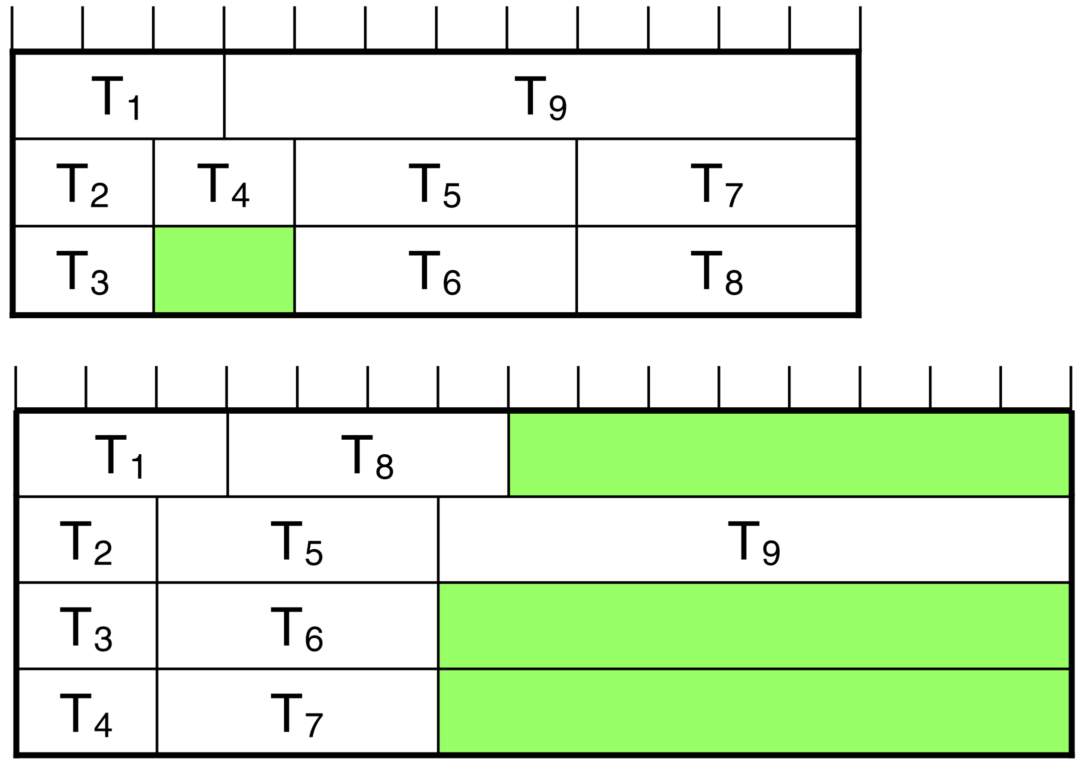
$L = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7\}$





Graham's list scheduling algorithm - Problem





$L = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9\}$

Coffman-Graham's scheduling algorithm (1)

- Graham's list scheduling algorithm depends upon a prioritized list of tasks to execute
- Coffman and Graham (1972) construct a list of tasks for the simple case when all tasks take the same amount of time.

Coffman-Graham's scheduling algorithm (2)

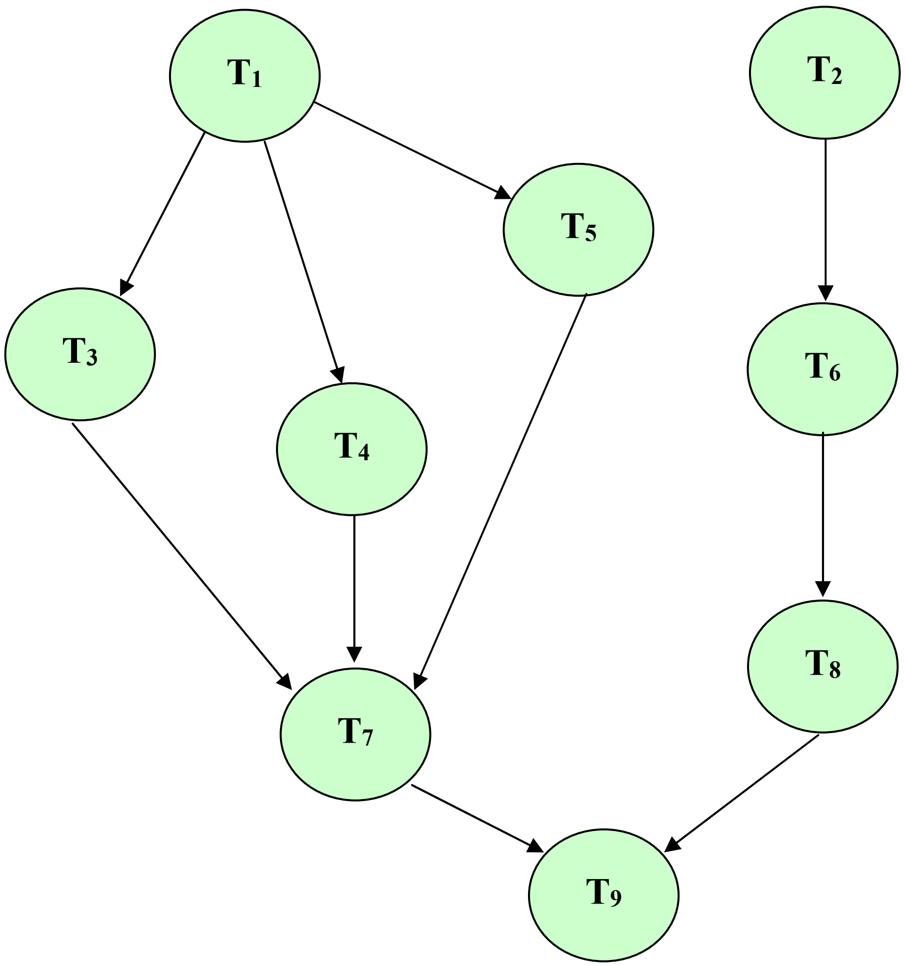
- Let **T** = $T_1, T_2, \ldots, T_n$ be a set of n unit-time tasks to be executed on p processors
- If $T_i \prec T_j$, then task is T_i an immediate predecessor of task T_j, and T_j is an immediate successor of task T_i
- Let $S(T_i)$ denote the set of immediate successor of task T_i
- Let $\alpha(T_i)$ be an integer label assigned to T_i.
- N(T) denotes the decreasing sequence of integers formed by ordering of the set $\{\alpha(T') | T' \in S(T)\}$

Coffman-Graham's scheduling algorithm (3)

1. Choose an arbitrary task T_k from **T** such that $S(T_k) = 0$, and define $\alpha(T_k)$ to be 1
2. for $i \leftarrow 2$ to n do
   a. R be the set of unlabeled tasks with no unlabeled successors
   b. Let T* be the task in R such that N(T*) is lexicographically smaller than N(T) for all T in R
   c. Let $\alpha(T^*) \leftarrow i$

   endfor
3. Construct a list of tasks $L = \{U_n, U_{n-1}, \ldots, U_2, U_1\}$ such that $\alpha(U_i) = i$ for all i where $1 \le i \le n$
4. Given (**T**, $\prec$, L), use Graham's list scheduling algorithm to schedule the tasks in T

Coffman-Graham's scheduling algorithm – Example (1)



T_2	T_6	T_4	T_7	T_9
T_1	T_3	T_8		
	T_5			

Coffman-Graham's scheduling algorithm – Example (2)

Step1 of algorithm

task T_9 is the only task with no immediate successor. Assign 1 to $\alpha(T_9)$

Step2 of algorithm

- i=2: R = $\{T_7, T_8\}$, $N(T_7)= \{1\}$ and $N(T_8)= \{1\}$ $\Rightarrow$ Arbitrarily choose task T_7 and assign 2 to $\alpha(T_7)$
- i=3: R = $\{T_3, T_4, T_5, T_8\}$, $N(T_3)= \{2\}$, $N(T_4)= \{2\}$, $N(T_5)= \{2\}$ and $N(T_8)= \{1\}$ $\Rightarrow$ Choose task T_8 and assign 3 to $\alpha(T_8)$
- i=4: R = $\{T_3, T_4, T_5, T_6\}$, $N(T_3)= \{2\}$, $N(T_4)= \{2\}$, $N(T_5)= \{2\}$ and $N(T_6)= \{3\}$ $\Rightarrow$ Arbitrarily choose task T_4 and assign 4 to $\alpha(T_4)$
- i=5: R = $\{T_3, T_5, T_6\}$, $N(T_3)= \{2\}$, $N(T_5)= \{2\}$ and $N(T_6)= \{3\}$ $\Rightarrow$ Arbitrarily choose task T_5 and assign 5 to $\alpha(T_5)$
- i=6: R = $\{T_3, T_6\}$, $N(T_3)= \{2\}$ and $N(T_6)= \{3\}$ $\Rightarrow$ Choose task T_3 and assign 6 to $\alpha(T_3)$

Coffman-Graham's scheduling algorithm – Example (3)

- i=7: R = {T_1, T_6}, N(T_1)= {6, 5, 4} and N(T_6)= {3} $\Rightarrow$ Choose task T_6 and assign 7 to $\alpha(T_6)$
- i=8: R = {T_1, T_2}, N(T_1)= {6, 5, 4} and N(T_2)= {7} $\Rightarrow$ Choose task T_1 and assign 8 to $\alpha(T_1)$
- i=9: R = {T_2}, N(T_2)= {7} $\Rightarrow$ Choose task T_2 and assign 9 to $\alpha(T_2)$

Step 3 of algorithm

L = {T_2, T_1, T_6, T_3, T_5, T_4, T_8, T_7, T_9}

Step 4 of algorithm

Schedule is the result of applying Graham's list-scheduling algorithm to task graph **T** and list L

Issues in processor scheduling

- Preemption inside spinlock-controlled critical sections

P_0	P_1	P_2
Enter	→ Enter	→ Enter
→Critical Section	Critical Section	→Critical Section
Exit	Exit	Exit

- Cache corruption
- Context switching overhead

Current approaches

- Global queue
- Variable partitioning
- Dynamic partitioning with two-level scheduling
- Gang scheduling

Global queue

- A copy of uni-processor system on each node, while sharing the main data structures, specifically the run queue
- Used in small-scale bus-based UMA shared memory machines such as Sequent multiprocessors, SGI multiprocessor workstations and Mach OS
- Autonamic load sharing
- Cache corruption
- Preemption inside spinlock-controlled critical sections

Variable partitioning

- Processors are partitioned into disjoined sets and each job is run only in a distinct partition

Scheme	Parameters taken into account		
	User request	System load	Changes
Fixed	no	no	no
Variable	yes	no	no
Adaptive	yes	yes	no
Dynamic	yes	yes	yes

- Distributed memory machines: Intel and nCube hypercudes, IBM PS2, Intel Paragon, Cray T3D
- Problem: fragmentation, big jobs

Dynamic partitioning with two-level scheduling

- Changes in allocation during execution
- Workpile model:
  - The work = an unordered pile of tasks or chores
  - The computation = a set of worker threads, one per processor, that take one chore at time from the work pile
  - Allowing for the adjustment to different numbers of processors by changing the number of the wokers
  - Two-level scheduling scheme: the OS deals with the allocation of processors to jobs, while applications handle the scheduling of chores on those processors

Gang scheduling

- Problem: Interactive response times $\Rightarrow$ time slicing
  - Global queue: uncoordinated manner
- Observartion:
  - Coordinated scheduling in only needed if the job's threads interact frequently
  - The rare of interaction can be used to drive the grouping of threads into gangs
- Samples:
  - Co-scheduling
  - Family scheduling: which allows more threads than processors and uses a second level of internal time slicing

Several specific scheduling methods

- Co-scheduling
- Smart scheduling [Zahorijan et al.]
- Scheduling in the NYU Ultracomputer [Elter et al.]
- Affinity based scheduling
- Scheduling in the Mach OS

Co-Scheduling

- Context switching between applications rather then between tasks of several applications.
- Solving the problem of “preemption inside spinlock-controlled critical sections”.
- Cache corruption???

Smart scheduling

- Advoiding:

  (1) preempting a task when it is inside its critical section

  (2) rescheduling tasks that were busy-waiting at the time of their preemption until the task that is executing the corresponding critical section releases it.
- The problem of “preemption inside spinlock-controlled critical sections” is solved.
- Cache corruption???.

Scheduling in the NYU Ultracomputer

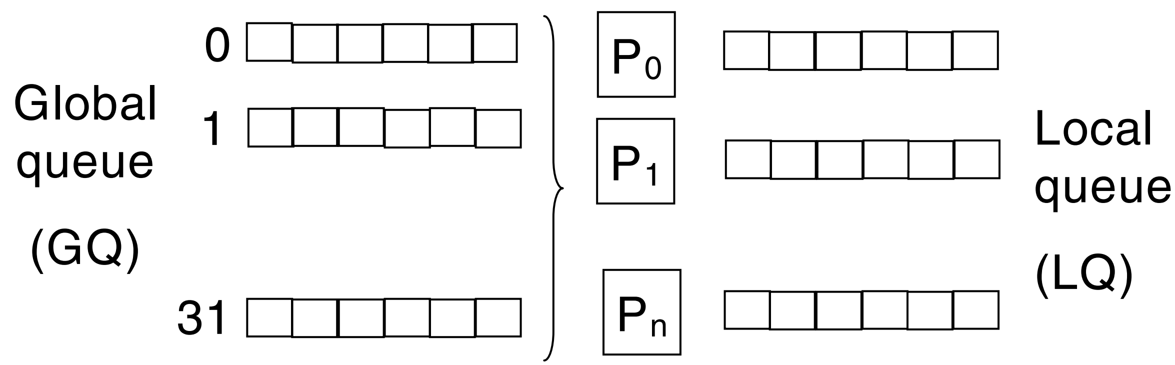
- Tasks can be formed into groups
- Tasks in a group can be scheduled in any of the following ways:
  - A task can be scheduled or preempted in the normal manner
  - All the tasks in a group are scheduled or preempted simultaneously
  - Tasks in a group are never preempted.
- In addition, a task can prevent its preemption irrespective of the scheduling policy (one of the above three) of its group.

Affinity based scheduling

- Policity: a tasks is scheduled on the processor where it last executed [Lazowska and Squillante]
- Alleviating the problem of cache corruption
- Problem: load imbalance

Scheduling in the Mach OS

- Threads
- Processor sets: disjoin
- Processors in a processor set is assigned a subset of threads for execution.
  - Priority scheduling: LQ, GQ(0),…,GQ(31)



  - LQ and GQ(0-31) are empty: the processor executes an special *idle* thread until a thread becomes ready.
  - Preemption: if an equal or higher priority ready thread is present