Distributed process scheduling

The primary objective of scheduling is to enhance overall system performance metrics such as process completion time and processor utilization. The existence of multiple processing nodes in distributed systems present a challenging problem for scheduling processes onto processors and vice versa.

A system performance model

Partitioning a task into multiple processes for execution can result in a **speedup** of the total task completion time. The speedup factor S is a function

$$S = F(Algorithm, System, Schedule)$$

S can be written as:

$$S = \frac{OSPT}{CPT} = \frac{OSPT}{OCPT_{ideal}} \times \frac{OCPT_{ideal}}{CPT} = S_i \times S_d$$

where

- $OSPT =$ optimal sequential processing time

- $CPT =$ concurrent processing time
- $OCPT_{ideal} =$ optimal concurrent processing time
- $S_i =$ the ideal speedup
- $S_d =$ the degradation of the system due to actual implementation compared to an ideal system

S_i can be rewritten as:

$$S_i = \frac{RC}{RP} \times n$$

where

$$RP = \frac{\sum_{i=1}^{m} P_i}{OSPT}$$

and

$$RC = \frac{\sum_{i=1}^{m} P_i}{OCPT_{ideal} \times n}$$

and n is the number of processors. The term $\sum_{i=1}^{m} P_i$ is the total computation of the concurrent algorithm where m is the number of tasks in the algorithm. S_d can be rewritten as:

$$S_d = \frac{1}{1+\rho}$$

where

$$\rho = \frac{CPT - OCPT_{ideal}}{OCPT_{ideal}}$$

RP is **Relative Processing**: how much loss of of speedup is due to the substitution of the best sequential algorithm by an algorithm better adapted for concurrent implementation. RC is the **Relative Concurrency** which measures how far from optimal the usage of the n-processor is. It reflects how well adapted the given problem and its algorithm are to the ideal n-processor system. The final expression for speedup S is

$$S = \frac{RC}{RP} \times \frac{1}{1+\rho} \times n$$

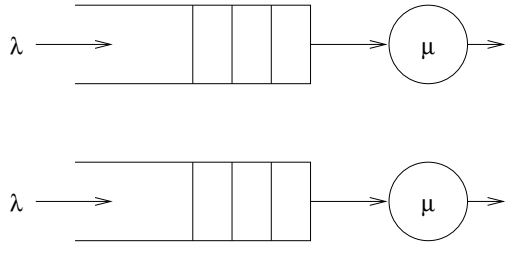
The term ρ is called *efficiency loss*. It is a function of scheduling and the system architecture. It would be decomposed into two independent terms: $\rho = \rho_{sched} + \rho_{syst}$, but this is not easy to do since scheduling and the architecture are interdependent. The best possible schedule on a given system hides the communication overhead (overlapping with other computations).

The **unified speedup model** integrates three major components

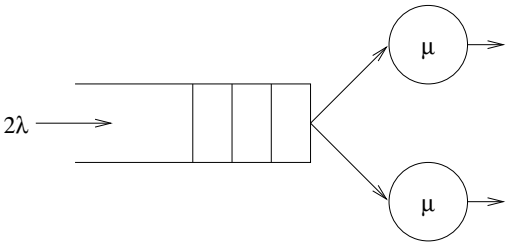
- algorithm development
- system architecture
- scheduling policy

with the objective of minimizing the total completion time (**makespan**) of a set of interacting processes. If processes are not constrained by precedence relations and are free to be redistributed or moved around among processors in the system, performance can be further improved by sharing the workload

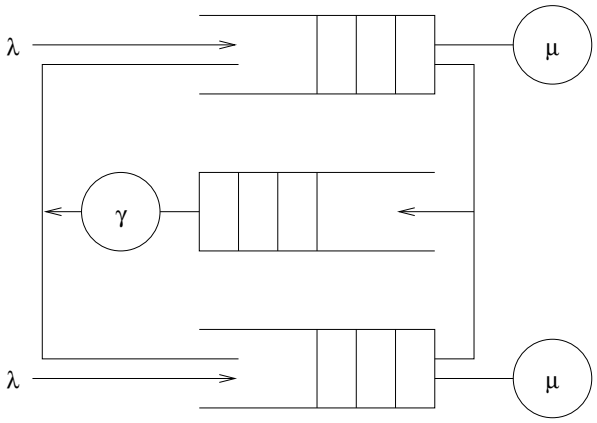
- statically - **load sharing**
- dynamically - **load balancing**



(a) M / M / 1 isolated workstations



(b) M / M / 2 processor pool model



(c) Migration workstation model

The standard notation for describing the stochastic properties of a queue is **Kendall's notation**. An $X/Y/c$ is one with an arrival process X, a service time distribution of Y and c servers. The processor pool can be described as an $M/M/2$, where M stands for a *Markovian distribution*.

In the **migration workstation model**, the migration rate γ is a function of the channel bandwidth, process migration protocol, and context and state information of the process being transferred.

Static process scheduling

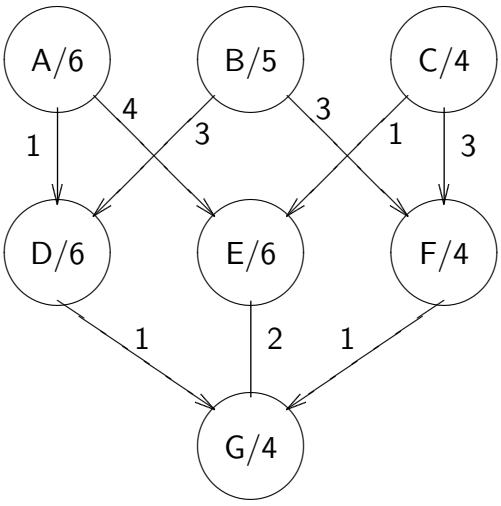
- Scheduling a set of partially ordered tasks on a nonpreemtive multiprocessor system of identical processors to minimize the overall finishing time (makespan)
- Except for some very restricted cases scheduling to optimize makespan is **NP-complete**
- Most research is oriented toward using approximate or heuristic methods to obtain a *near optimal* solution to the problem
- A good heuristic distributed scheduling algorithm is one that can best balance and overlap computation and communication

In static scheduling, the mapping of processes to processors is determined before the execution of the processes. Once a process is started, it stays at the processor until completion.

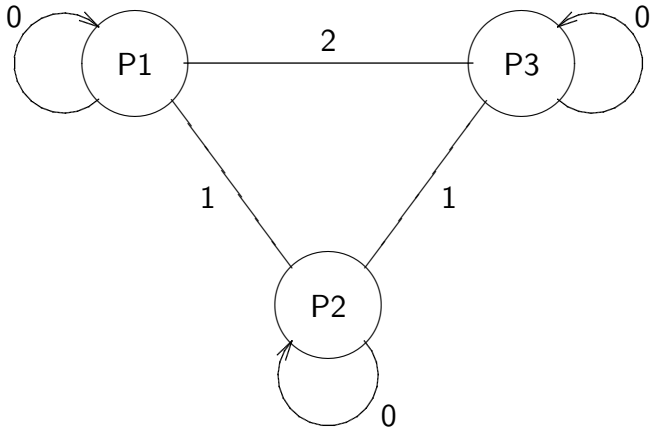
Precedence process model

- Program is represented by a *directed acyclic graph (DAG)*
- Computational model

- Primary objective of task scheduling is to achieve maximal concurrency for task execution within a program



(a) Precedence process model

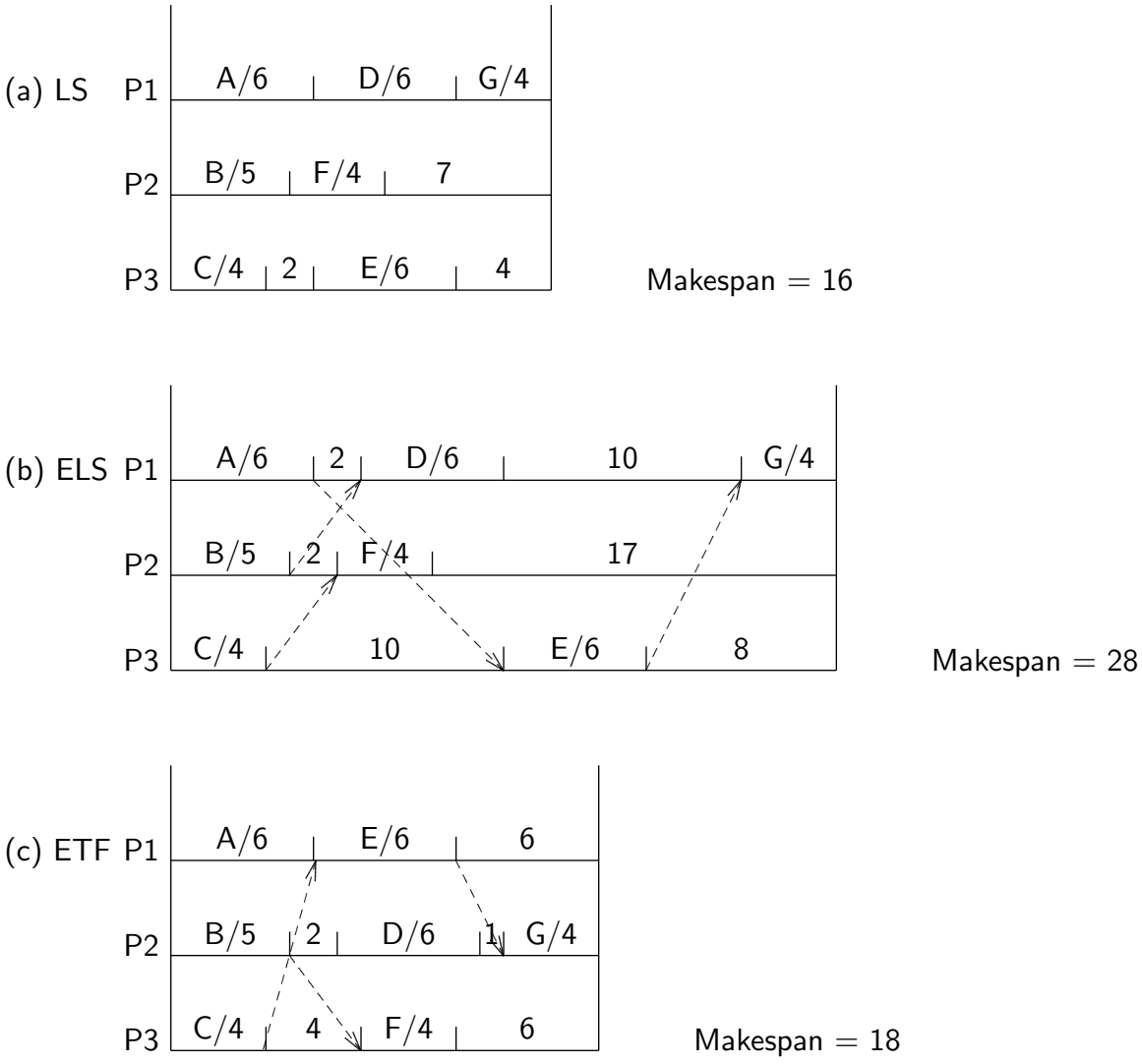


(b) Communication system model

Finding the minimum makespan is NP-complete, so we will rely on heuristic algorithms for finding good mapping of the process model to the system model. For precedence process graphs, the notion of **critical path** is useful - the longest execution path in the DAG, which is the lower bound of the makespan. Simple heuristic: map all tasks in a critical path onto a single processor.

1. **List Scheduling** (LS) strategy: No processor remains idle if there are some tasks available that it could process (without considering communication overhead).

2. **Extended List Scheduling** (ELS) strategy: Allocating tasks to processors according to LS and adding communication delays. communication overhead.
3. **Earliest Task First** (ETF): The earliest schedulable task is scheduled first (calculation includes communication overhead).

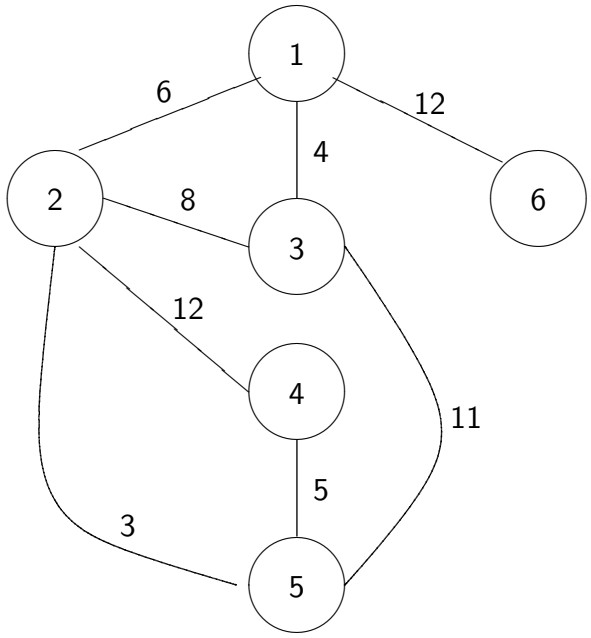


Communication process model

- Process scheduling for many system applications has a perspective very different from precedence model - applications may be created independently, processes do not have explicit completion time and precedence constraints
- Primary objectives of process scheduling are to maximize resource utilization and to minimize interprocess communication
- Communication process model is an undirected graph G with node and edge sets V and E, where nodes represent processes and the weight on an edge is the amount of interaction between two connected processes

Process	Cost on A	Cost on B
1	5	10
2	2	infinity
3	4	4
4	6	3
5	5	2
6	infinity	4

(a) Computation set



(b) Communication cost

Objective function called **Module Allocation** for finding an optimal allocation of m process modules to P processors:

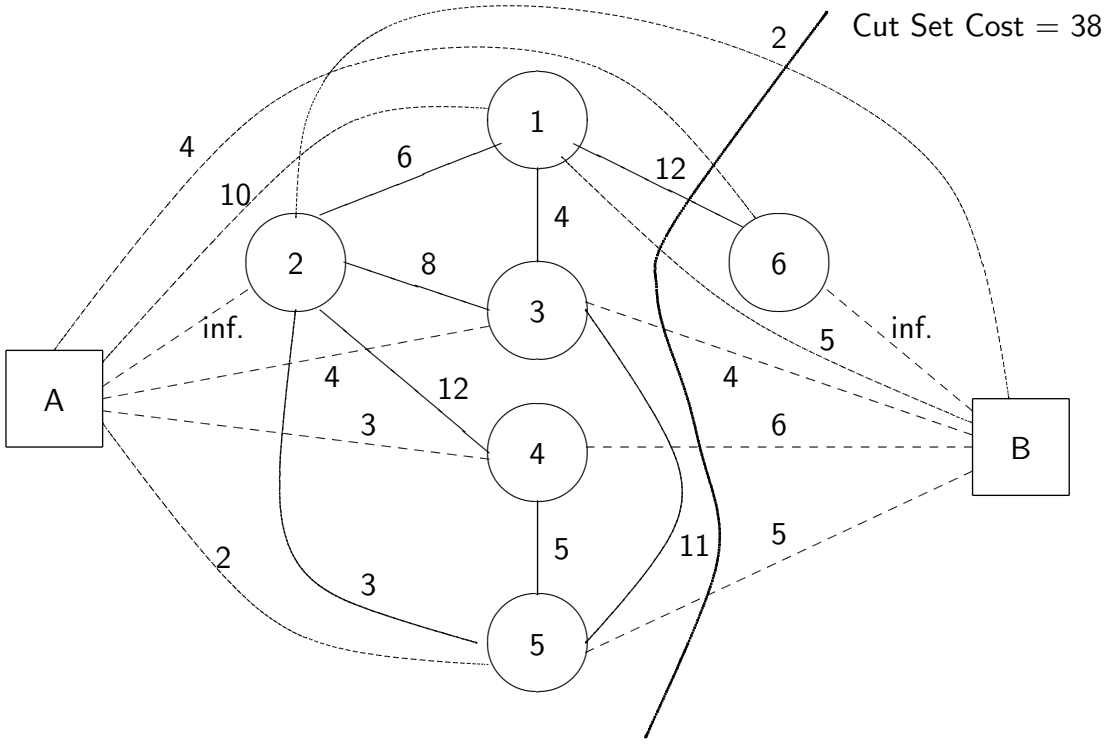
$$Cost(G, P) = \sum_{j \in V(G)} e_j(p_i) + \sum_{(i,j) \in E(G)} c_{i,j}(p_i, p_j)$$

where $e_j(p_i)$ is the execution cost of process j on p_i, which is the processor allocated to process j and $c_{i,j}(p_i, p_j)$ is the communication cost between two processes i and j, allocated to two different processors p_i and p_j.

For $P = 2$ there is an efficient polynomial-time solution using Ford-Fulkerson's maximum-flow algorithm - equivalent to the *minimum cut* set in the graph.

Heuristic solution: separate optimization of computation and communication into two independent phases.

- Processes with higher interprocess interaction are merged into clusters
- Each cluster is then assigned to the processor that minimizes the computation cost



Minimum-cost cut

Dynamic load sharing and balancing

The assumption of prior knowledge of processes is not realistic for most distributed applications. The disjoint process model, which ignores the effect of the interdependency among processes, is used. Objective of scheduling: *utilization* of the system (has direct bearing on throughput and completion time) and *fairness* to the user processes (difficult to define).

If we can designate a controller process that maintains the information about the queue size of each processor:

- Fairness in terms of equal workload on each processor (join the shortest queue) - migration workstation model (use of **load sharing** and **load balancing**, perhaps **load redistribution** i.e. **process migration**)
- Fairness in terms of user's share of computation resources (allocate processor to a waiting process at a user site that has the least share of the processor pool) - processor pool model

Solutions without a centralized controller: **sender-** and **receiver-initiated** algorithms.

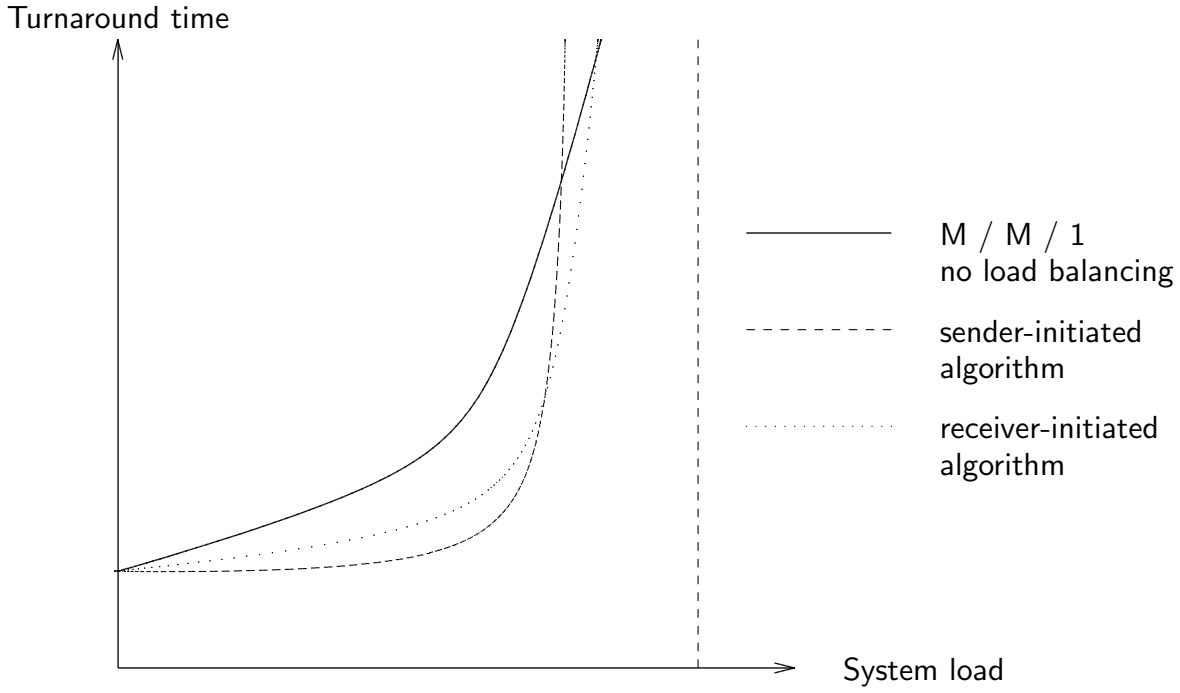
Sender-initiated algorithms:

- *push* model
- includes probing strategy for finding a node with the smallest queue length (perhaps multicast)
- performs well on a lightly loaded system

Receiver-initiated algorithms:

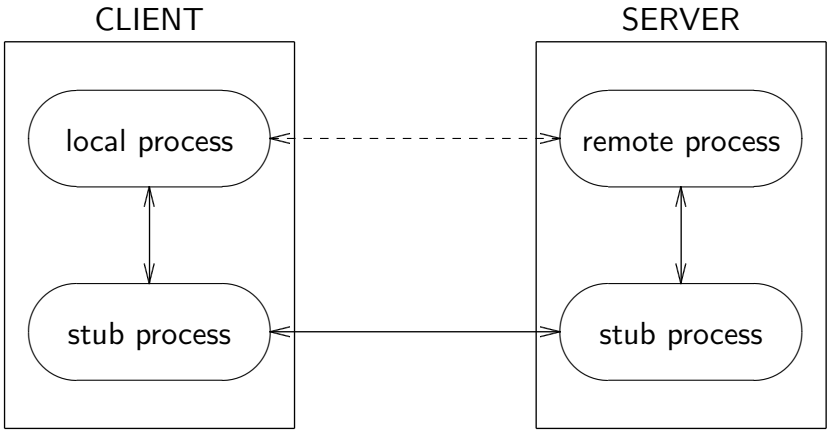
- *pull* model
- probing strategy can also be used
- more stable
- perform on average better

Combinations of both algorithms are possible: choice based on the estimated system load information or reaching threshold values of the processing node's queue.



Performance comparison of dynamic load-sharing algorithms

Distributed process implementation



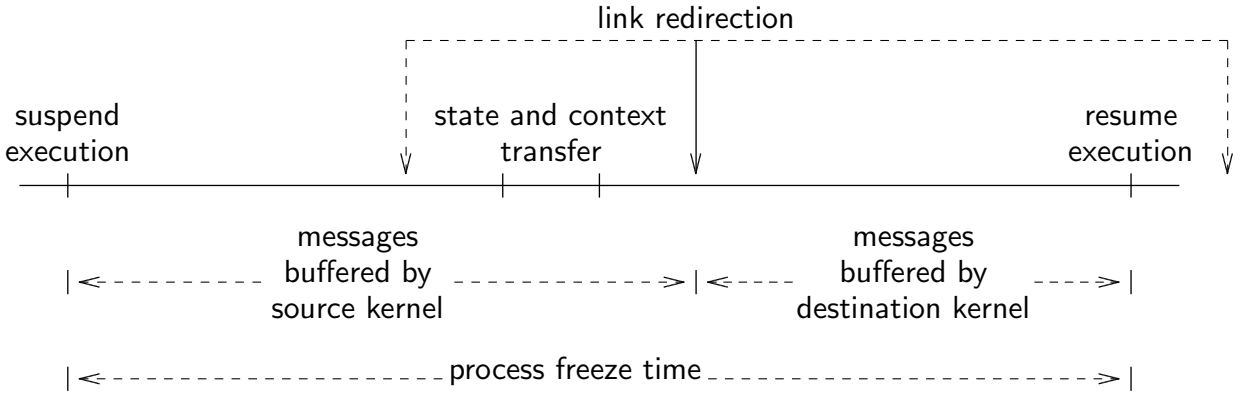
Logical model of local and remote processes

Three significant application scenarios:

- **Remote service**: The message is interpreted as a request for a known service at the remote site (constrained only to services that are supported at the remote host)
  - remote procedure calls at the language level
  - remote commands at the operating system level
  - interpretive messages at the application level
- **Remote execution**: The messages contain a program to be executed at the remote site; implementation issues:
  - load sharing algorithms (sender-initiated, registered hosts, broker...)

  - location independence of all IPC mechanisms including signals
  - system heterogeneity (object code, data representation)
  - protection and security
- **Process migration**: The messages represent a process being migrated to the remote site for continuing execution (extension of load-sharing by allowing a remote execution to be preeemted)

State information of a process in a distributed systems consists of two parts: **computation state** (similar to conventional context switching) and **communication state** (status of the process communication links and messages in transit). The transfer of the communication state is performed by *link redirection* and *message forwarding*.



Link redirection and message forwarding

Reduction of *freeze time* can be achieved with the transfer

of minimal state and leaving *residual computation dependency* on the source host: this concept fits well with distributed shared memory.