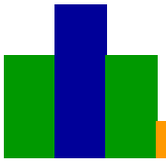


Lesson 7: Load balancing and termination detection

G Stefanescu — University of Bucharest

Parallel & Concurrent Programming
Fall, 2014

Topics

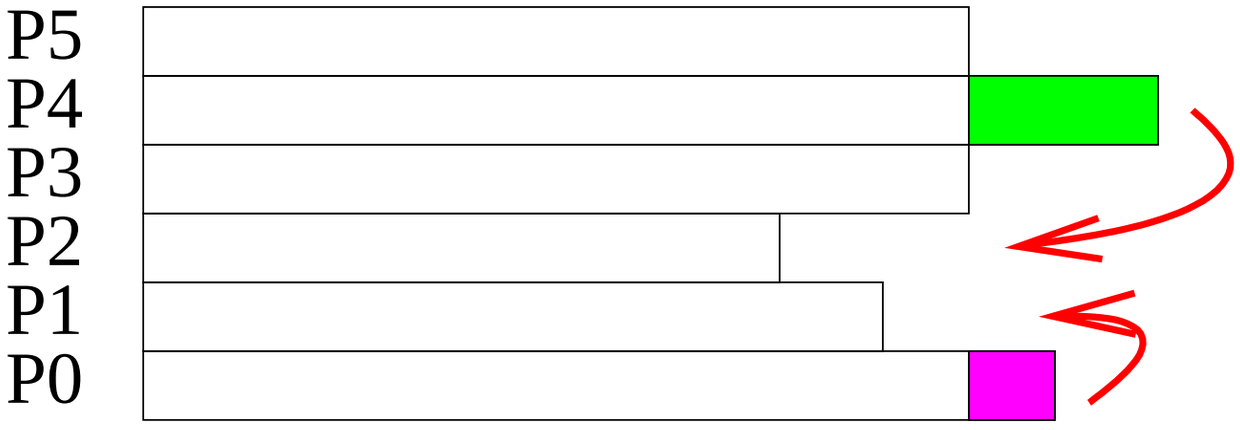
Load balancing:

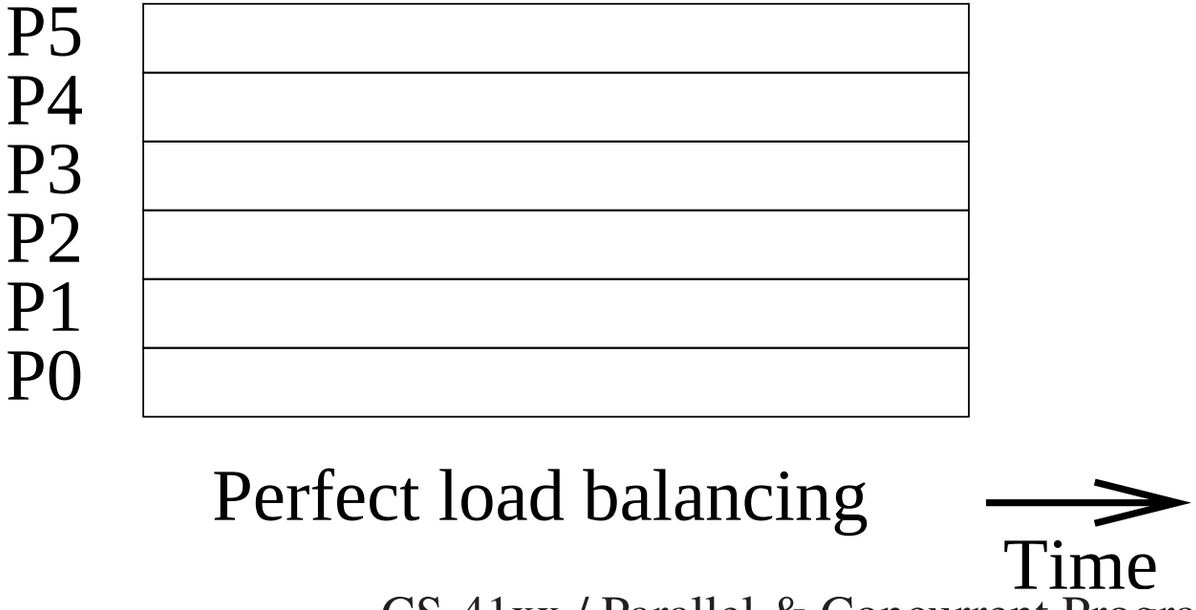
- a technique to make a *fair distribution of computation* across processors in order *to increase execution speed*

Termination detection:

- to detect when a computation has been completed
- it is usually more *difficult* when the computation is *distributed*

Load balancing



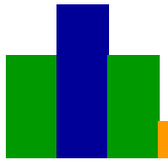


Static load balancing (SLB)

Static load balancing: in such a case the balancing is scheduled *before* the execution of any process.

A few static load balancing techniques are:

- *Round robin algorithm* - the tasks are passed to processes in a sequential order; when the last process has received a task the schedule continues with the first process (a new round)
- *Randomized algorithm*: the allocation of tasks to processes is random
- *Recursive bisection*: recursively divides the problem into sub-problem of equal computational effort
- *Simulated annealing* or *genetic algorithms*: mixture allocation procedure including optimization techniques



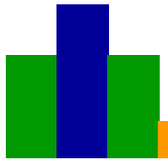
A theoretical result

A theoretical result:

- The problem of *mapping tasks to processes for arbitrary networks* is *NP-hard* (nondeterministically polynomial hard)

Hence

- by the common belief that $P \neq NP$, *no efficient polynomial time algorithm* may be found; one has to use various *heuristics*



Critics on static load balancing

Critics: - even when a good mathematical solution exists, static load balancing still have several flaws:

- it is very difficult to *estimate a-priori* [in an accurate way] the *execution time* of various parts of a program
- sometimes there are *communication delays* that vary in an uncontrollable way
- for some problems the *number of steps* to reach a solution is *not known* in advance

Dynamic load balancing (DLB)

Dynamic load balancing:

- the schedule of allocating tasks to processes is done *during the execution* of the processes

Features:

- the factors in the above critics are taken into account, improving the efficiency of the program
- there is an *additional overhead* during execution, but generally it is more effective than static load balancing
- *termination detection* is more *complicate* in dynamic load balancing

..DLB

(Features, cont.)

- computation is divided into *work* or *tasks* to be performed and *processes* perform these tasks
- processes are mapped onto *processors*; the ultimate goal is to keep processors busy;
- we will often map a single process onto each processor, so the terms *process* and *processor* are somehow interchangeably.

Type of DLB

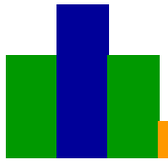
Dynamic load balancing can be classified as *centralized* or *decentralized*.

Centralized load balancing:

- tasks are distributed to processes from a centralized location
- a typical *master-slave* structure exists

Decentralized load balancing:

- (worker) processes interact among themselves to solve the problem, finally reporting to a single process
- tasks are passed between arbitrary processes: a worker process may receive tasks from any other worker process and may send tasks to any other worker process



Centralized DLB

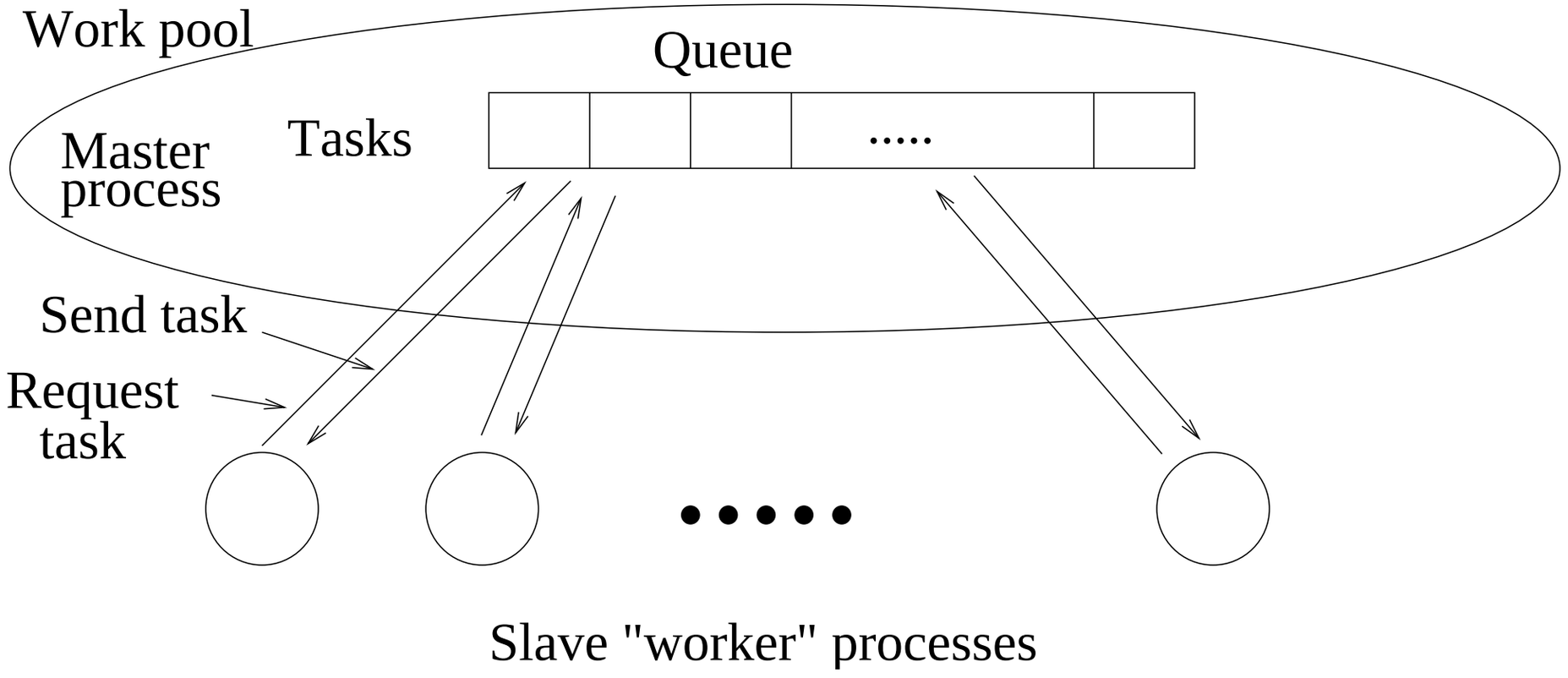
Good when: there is a *small number of slaves* and the problem consists of *computationally intensive tasks*

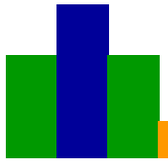
Basic features:

- a master process[or] holds the collection of tasks to be performed,
- tasks are sent to the slave processes
- when a slave process completes one task, it requests another task from the master process

One often uses the terms work pool, *or* replicated worker, *or* processor farm *in this context. Technically, it is more efficient to start with the larger tasks first.*

..centralized DLB





Termination in centralized DLB

Termination:

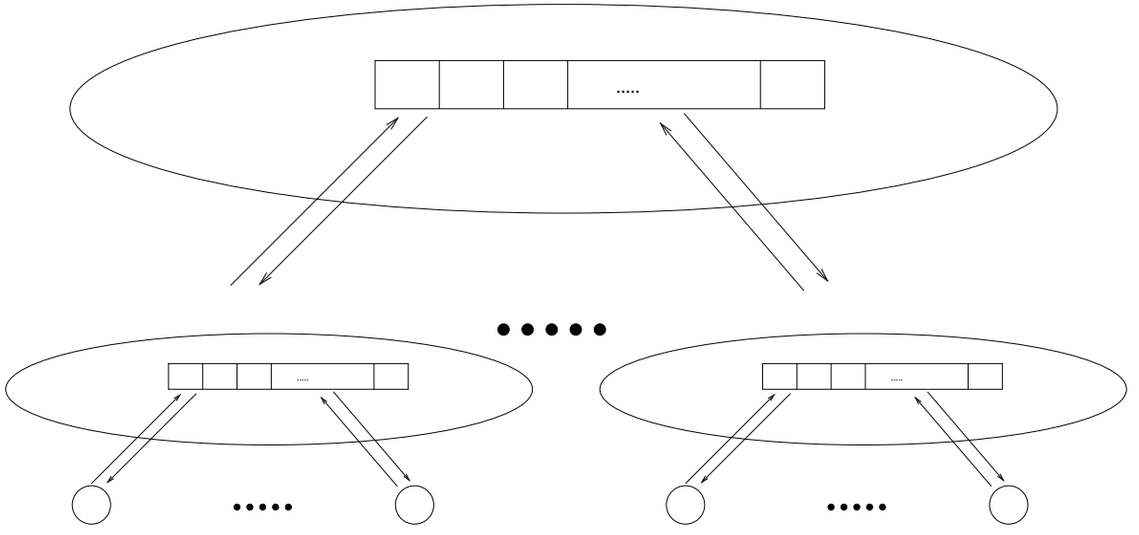
- stop the computation when the solution has been found
- when the tasks are taken from a task queue, computation terminates when
  - the task queue is empty *and*
  - every process has made a request for another task without any new tasks being generated

  [It is not sufficient to check if the queue is empty if running processes are allowed to put tasks in the task queue.]
- in some applications a slave may detect the program termination by some local termination, for instance finding an item in a search algorithm

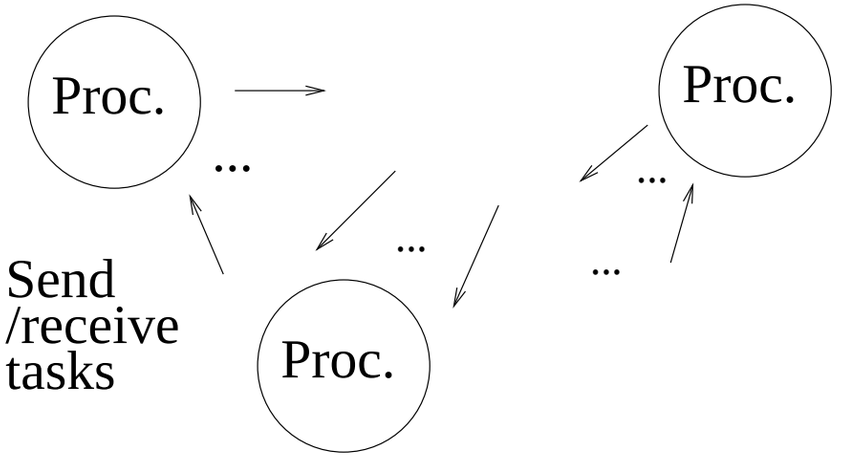
Decentralized DLB

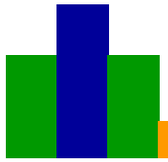
Distributed work pool:

Tree structure



General (fully distributed work pool):





Task transfer mechanisms

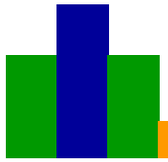
Task transfer mechanisms:

(1) *receiver-initiated method* or

(2) *sender-initiated method*

Receiver-initiated method:

- *a process requests tasks* from other processes it selects; typically this is done when the process has few or no tasks to perform
- this method work *well* when there is a *high system load* [but, as we have seen, it can be expensive to determine process load]



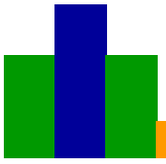
..Task transfer mechanisms

Sender-initiated method:

- *a process sends tasks* to other processes it selects; typically this is done when the process has a heavy load and may pass some tasks to other processes that are willing to accept them
- this method work *well* when there is a *light system load*

Final comments:

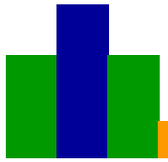
- the above "pure" approaches may be mixed
- however, whatever method one may use, in very heavy system loads, load balancing can be difficult to achieve due to the lack of available processes.



Process selection in DLB

Process selection: Possible algorithms for selecting a process may be

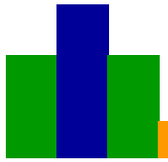
- *Round robin algorithm:* process P_i requests tasks from process P_x, where x is given by a counter that is incremented modulo n, [if there are n processes], excluding $x = i$
- *Random polling algorithm:* process P_i requests tasks from process P_x, where x is a number that is randomly selected from the set $\{0, \ldots, i-1, i+1, \ldots, n-1\}$ [assuming there are n processes $P_0, \ldots P_{n-1}$]



DLB on a line structure

Procedure: (informal)

- the *master* process *feeds the queue with tasks* at one end; the tasks are shifted down the queue
- *when a worker* process $P_i (1 \leq i < n)$ *detects a task* and it is *idle*, it *takes the task* from the queue
- then the *tasks* to the left *shuffle* down the queue so that the space held by the task is filled; a *new task is inserted* into the left side end of the queue
- eventually, all processes will have a task and the queue is filled with new tasks
- for better results, high priority or larger tasks could be placed in the queue first

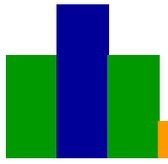


Code for DLB on a line structure

Shifting actions: - based on left and right communications with adjacent processes and handling of current task

Code (master):

```
for (i=0; i < noTasks; i++){
    recv(P1, requestTag);
    send(&task,P1,taskTag);
}
recv(P1,requestTag);
send(&empty,P1,taskTag);
```



..Code for DLB on a line structure

Code $P_i (1 \leq i < n)$):

```
if (buffer == empty){
    send(P_{i-1},requestTag);
    recv(&buffer,P_{i-1},taskTag);
}
if ((buffer == full) && (!busy)){
    task = buffer;
    buffer = empty;
    busy = TRUE;
}
nrecv(P_{i+1}, requestTag, request)
if (request & (buffer == full)){
    send(&buffer, P_{i+1});
    buffer = empty;
}
if (busy) {
    do some work on task;
    if task finished, set busy to false;
}
```

Notice: `nrecv` *denotes a nonblocking receive routine.*

Nonblocking routines

PVM:

- the nonblocking receive routine, `pvm_nrecv()`, returns a value that is zero if no message has been received
- a probe routine, `pvm_probe()`, may be used to check whether a message has been received without actual reading of this message [later, a normal `recv()` has to be used to accept and unpack the message]

PVM:

- the nonblocking receive routine, `MPI_Irecv()`, returns in a parameter a request "handle", which is used later to complete the communication (using `MPI_Wait` and `MPI_Test`)
- actually, it posts a request for a message and return immediately

Distributed termination detection (TD)

Termination conditions: - the following conditions have to be fulfilled:

- the [application specific] *local termination* conditions are satisfied by all processes
- there are no messages in transit between processes

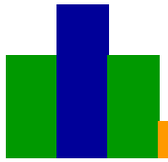
Notice: The second condition is necessary to avoid a situation where a message in transit may restart *an already terminated process. It is not easy to check, as the communication time is not known in advance.*

TD using acknowledgment messages

Termination detection using acknowledgment messages:

- each process P is either *inactive* (no task to handle) or *active*
- a process P' which sends a message to make an idle process P active becomes its *parent*
- whenever P *receives a task*, it immediately *sends an acknowledgment* message [except if the task is received from its parent]

(cont.)



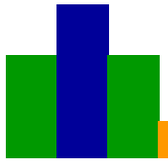
..TD using acknowledgment messages

(cont.)

- it sends an *acknowledgment* message *to its parent* process when
  - its *local termination* condition holds [all its tasks have been completed]
  - it has *transmitted acknowledgment* messages for all *tasks* it has *received*
  - it has *received acknowledgment* messages for all *tasks* it has *sent out*

  [hence, this process becomes inactive before its parent]

- the full *computation is finished* when the *first process* [which has started the computation] becomes *idle* (inactive)

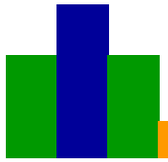


Single-pass ring termination algorithm

Termination detection using ring termination algorithms:
Single-pass ring termination algorithm:

- When P_0 *has terminated* its computation, it *generates a token* that is *passed to* P_1.
- When $P_i (1 \leq i < n)$ *receives the token* and has already *terminated*, it passes the token onward *to* P_{i+1}. Otherwise, it waits for its local termination condition and then passes the token onward. [P_{n-1} passes the token to P_0.]
- When P_0 *receives a token*, it knows that *all* processes in the ring have *terminated*. Finally, a message can be sent to all processes informing them on global termination.

To insure the correctness of this algorithm, it is supposed that *a process cannot be reactivated* after reaching its termination condition.

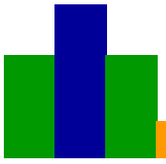


Dual-pass ring termination algorithm

Dual-pass ring termination algorithm:

- can handle the case when *processes may be reactivated after their local termination*, but it requires a repeated passing of the token around the ring.
- technically, we use *colored tokens*: a token may be *black* or *white*
- subsequently, *processes* are also *colored*: a process is either *black* or *white*
- roughly speaking, a black color means global termination may have not occurred

The algorithm is as follows (starting with P_0):



..dual-pass ring termination

(..dual-pass ring termination algorithm):

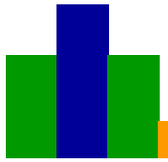
- P_0 becomes white when it has terminated; it generates a white token that is passed to P_1
- The token is passed through the ring form one process P_i to the next when P_i has terminated.
- However, the color of the token may be changes: if a process P_i passes a task to a process P_j with $j < i$, then it becomes a *black process*; otherwise it is a *white process*.
- A black process will color a token black and pass on; a white process will pass on the token in its original color
- After P_i has passed on a token, it becomes a white process.
- When P_0 receives a black token, it passes on a white token; if it receives a white token, all processes have terminated

Fixed energy distributed termination algorithm

Fixed energy distributed termination algorithm: - based on a *fixed quantity* within the system, called *energy*:

- the system starts with all the energy being held by one process, the master process
- master process passes out portions of the energy with the tasks to processes making requests for tasks
- if these processes receive requests for tasks, the energy is divided further and passed to these processes
- when a process becomes idle, it passes the energy it holds back to master before requesting a new task

(cont.)



..fixed energy distributed termination

(cont.)

- (it can return the energy to the process which has activated it, but than it has to wait until all the energy it handed out is back to him, i.e., we have a tree structure)
- when all energy is returned to the master and the master becomes idle, all the processes must be idle and the computation can terminate

Significant disadvantage:

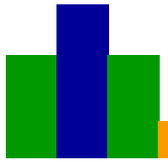
- the process of dividing and adding energy may not have 100% precision [if the number of processes is large, the energy may become very small]
- hence, the termination may not always be correctly detected.

Case studies: Shortest path problem (SPP)

Shortest path problem: - the problem is to find the shortest distance between two points of a graph; more precisely,

SPP Given a set of interconnected nodes where the links between the nodes are marked with *weights*, find the path from one specific node to another specific node that has the smallest accumulated weights.

Terms: the interconnected nodes can be described by a *graph*; the nodes are also called *vertices* and the links *edges*; if the edges have directions [i.e., edges can only be traversed in one direction], then the graph is called *directed*.



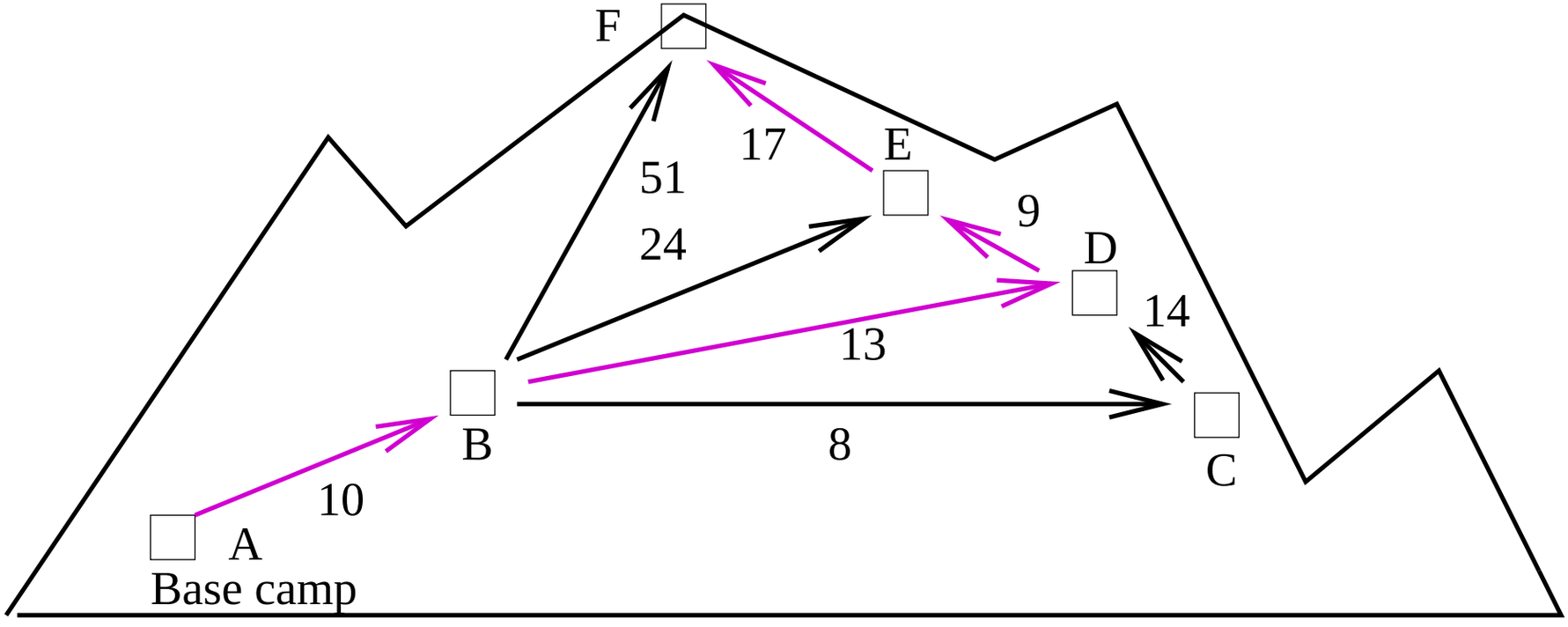
Particular interpretations of SPP

Particular interpretations of the problem:

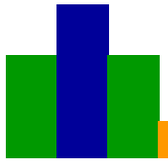
- The *shortest distance* between two towns or other points on a map, where the weights represent distance;
- The *quickest route to travel*, where the weights represent time [different from "the shortest" if different modes of travel are available: plane, train, etc.];
- The *least expensive way to travel* by air, where the weights represent the cost of flights between cities [vertices];
- The *best way to climb* a mountain given a terrain map with contours
- The *best route through a computer network* for minimum message delay
- Etc.

Best way to climb a mountain

Example: The best way to climb a mountain:



- The weights of the graph indicate the amount of effort taking a route between two connected camp sites
- The effort in one direction may be different from the effort in the opposite direction, hence we have a *directed graph*



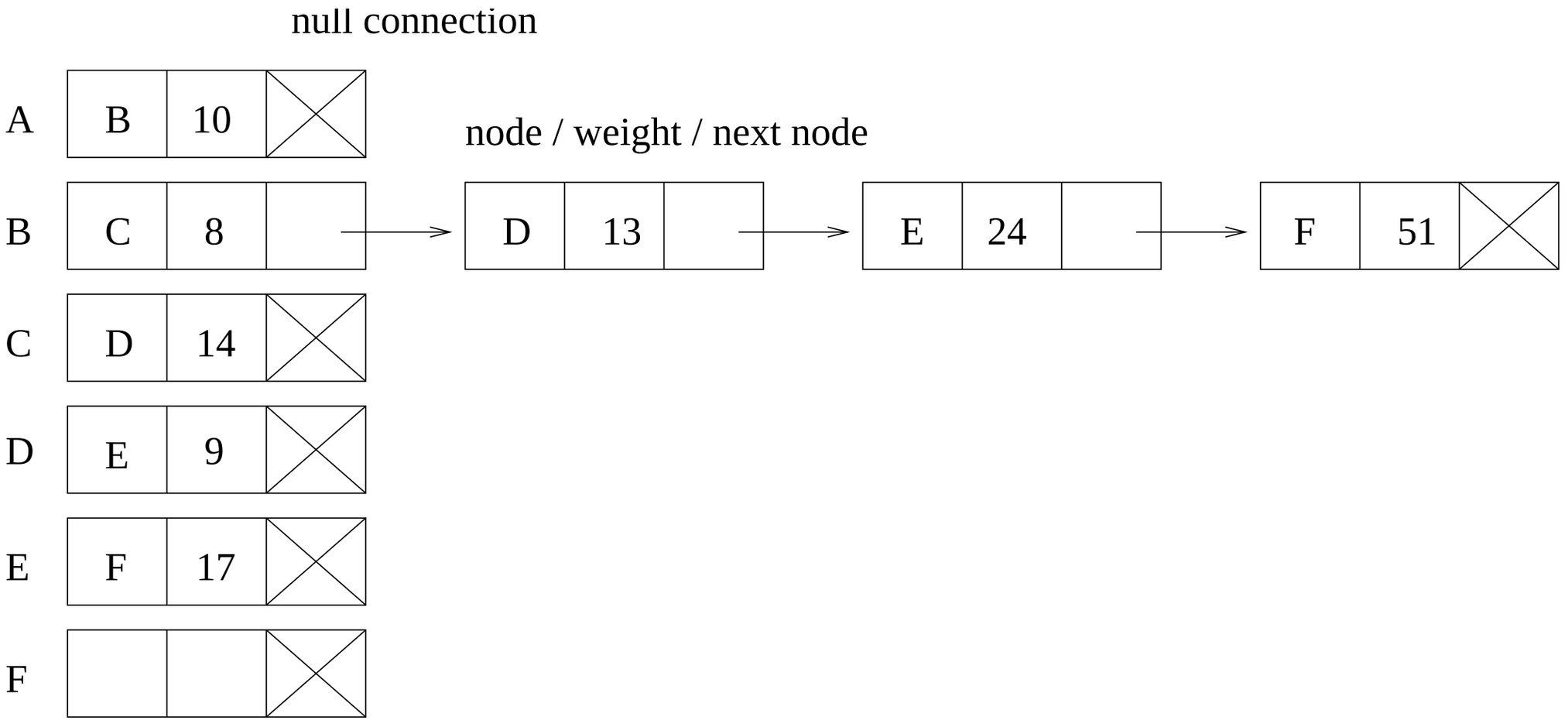
Graph representations

Graph representation - matrix: - that is, a two-dimensional array `a`, in which `a[i][j]` holds the weight associated to the edge between vertex *i* and vertex *j* [good for *dense* graphs]

Source \ *Destination*	*A*	*B*	*C*	*D*	*E*	*F*
A	∞	10	∞	∞	∞	∞
B	∞	∞	8	13	24	51
C	∞	∞	∞	14	∞	∞
D	∞	∞	∞	∞	9	∞
E	∞	∞	∞	∞	∞	17
F	∞	∞	∞	∞	∞	∞

..graph representations

Graph representation - lists: to a vertex v we attach a list containing all directly connected vertices from v (by an edge) and the corresponding weights of those edges [good for *sparse* graphs]



Sequential algorithms for SPP

Searching a graph: Two well-known algorithms

- Moore's single-source shortest path algorithm [Moore 1957; any order for search]
- Dijkstra's single-source shortest path algorithm [Dijkstra 1959; nearest vertex first]

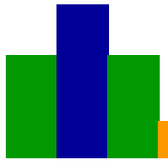
Here Moore algorithm is chosen because it is more amenable to parallel implementation, although it may do more work.

[Notice: The weights must be *positive* values; there are variations of the algorithms dealing with negative values.]

Moore algorithm for SPP

Moore algorithm: - based on $d_j := min(d_j, d_i + w_{i,j})$

- Start with the source vertex;
- Suppose d_i is an [updated] current minimum distance from source vertex to a vertex i; for an arbitrary vertex j, update the minimum distance d_j with $d_i + w_{i,j}$, if the latter is smaller
- Repeat the above for all vertices with updated distance, till there are no more updating.



..Moore algorithm for SPP

Moore algorithm, sequential code:

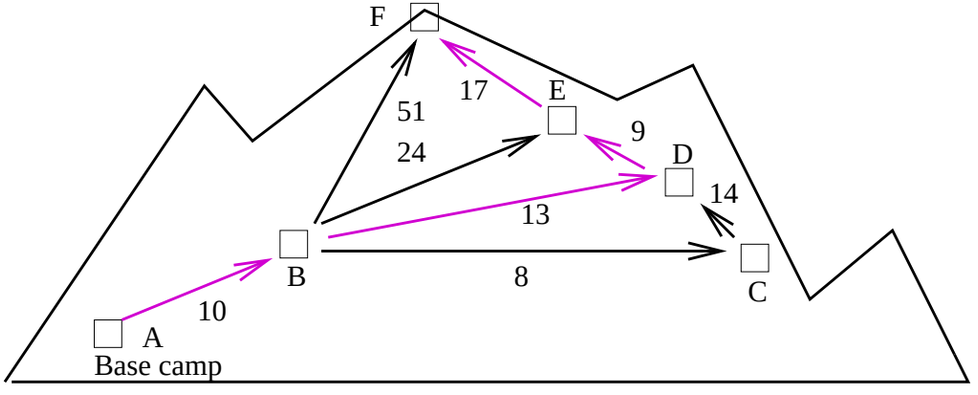
```
while(vertexQ != empty){
    i = getVetex(vertexQ);
    for (j=1; j<n; j++){
        if (w[i][j] != infinity){
            newDist = dist[i]+w[i][j];
            if(newDist < dist[j]){
                dist[j] = newDist;
                put(j,vertexQ);
            }
        }
    }
}
```

..Moore algorithm for SPP

Moore algorithm: example

Use a FIFO queue `vertexQ` holding the vertices with updated distances; `dist[i]` holds the current distance to vertex *i*. A running, using the above climbing example, is

`vertexQ`	`dist` *to*					
	A	*B*	*C*	*D*	*E*	*F*
A	0	∞	∞	∞	∞	∞
B	0	10	∞	∞	∞	∞
E,*D*,*C*[,*F*]	0	10	18	23	34	61
D,*C*	0	10	18	23	34	51
C,*E*	0	10	18	23	32	50
E[,*D*]	0	10	18	23	32	49
[,*F*]						



Centralized parallel Moore algorithm

Centralized parallel Moore algorithm:

- vertices from `vertexQ` are used as tasks
- each slave takes vertices from the vertex queue and return new vertices
- the master holds an array with current distances; it updates the array when shorter distances are received
- as the graph structure is fixed, we assume the adjacency matrix is copied to each slave

Centralized parallel Moore algorithm

Master code

```
while ((vertexQ != empty) || (more messages to come)){
     recv(j, newDist, P_any, source = P_i, tag);
     if (tag == requestTag){                          /* request task */
          v = get(vertexQ);
          send(v,P_i);                              /* send next vertex */
          send(&dist,&n,P_i);                         /* and dist array */
     } else {                                 /* got vertex and distance */
          if(newDist[j] < dist[j]){
               put(j,vertexQ);                  /* put vertex in queue */
               dist[j] = newDist;                        /* update dist */
          }
     }
}
for (j=1; j<n; j++){
     recv(P_any, source = P_i);
     send(P_i, terminationTag);
}
```

..Centralized parallel Moore algorithm

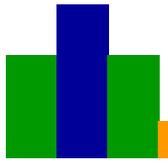
Slave code

```
send(P_master)                                  /* send request for task */
recv(&v,P_master, tag)                          /* get vertex/tag */
while (tag != termination tag){
     recv(&dist,&n,P_master);                   /* get distances */
     for (j=1; j<n; j++)
          if(w[v][j] != infinity){
               newDist = dist[v]+w[v][j]
               if (newDist < dist[j]){
                    send(&j,&newDist,P_master);
                                   /* send vertex and new distance */
               }
          }
     send(P_master)                             /* send request for task */
     recv(&v,P_master, tag)                     /* get vertex/tag */
}
```

Decentralized parallel Moore algorithm

Informal algorithm

- Process i handle vertex i; it holds the [current] minimum distance `dist` from source to vertex i [initially this is ∞]; it also holds a list with its neighbors
- When process i receives a new, smaller value for the distance from the source to itself computed by a different process, then
  —it updates its current minimum distance `dist`
  —computes all new distances `dist+w[j]` to all vertices j it is connected to, and
  —send these values to the appropriate neighbors
- Normally, a process is idle; it is activated by receiving a distance sent by another process; initially only the source is active;
- The termination is checked using a distributed termination procedure.



Decentralized parallel Moore algorithm

Slave code

```
recv(newDist, P_any);
if (newDist < dist){
    dist = newDist;
    for (j=1; j<n; j++){
        if (w[j] != infinity){
            d = dist + w[j];
            send(&d, P_j);
        }
    }
}
```

Notice: This is the basic step for the slave code; it has to be placed in a loop and integrated with a procedure for termination detection.