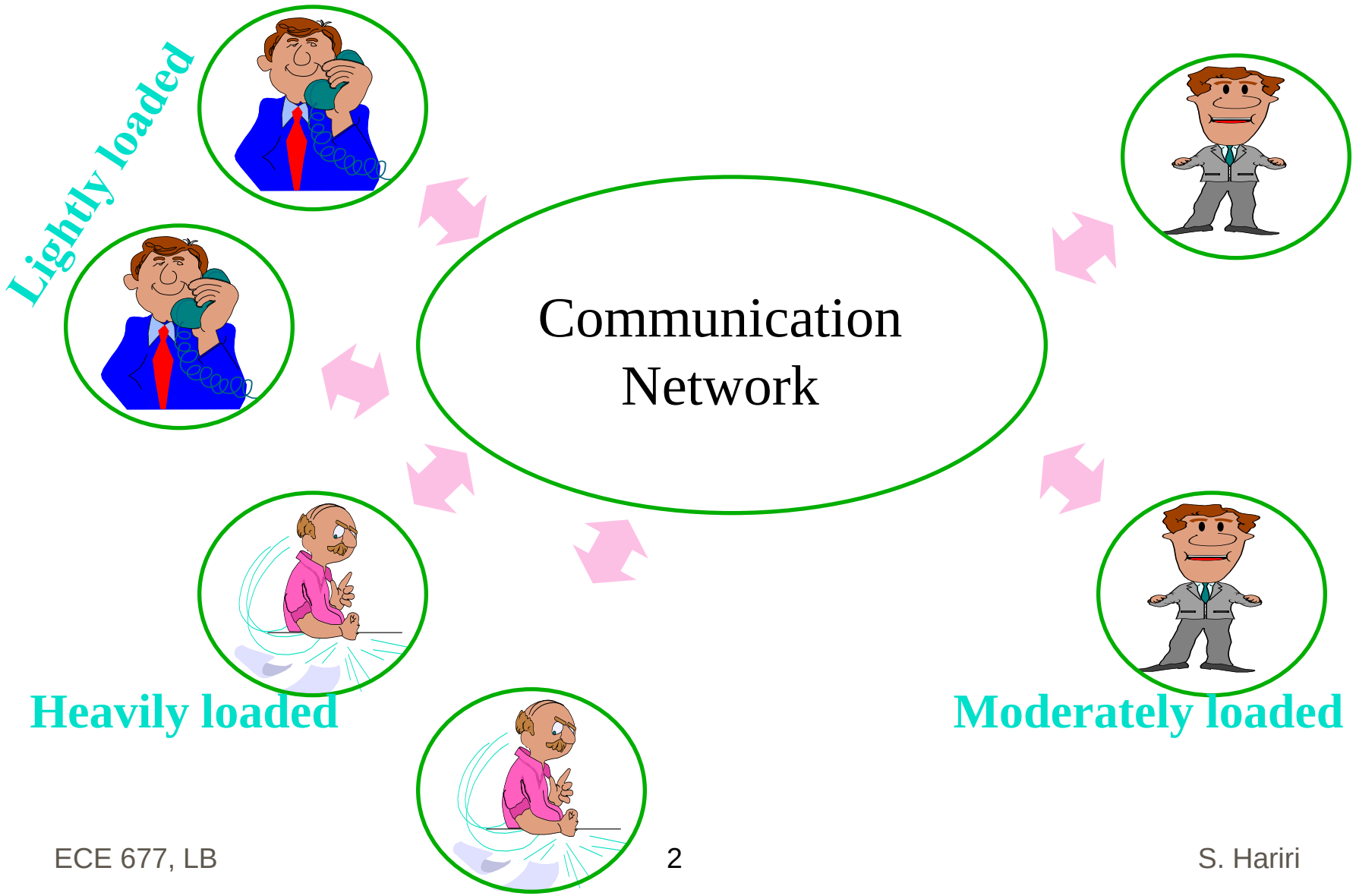
Load Balancing/Sharing/Scheduling

ECE 677

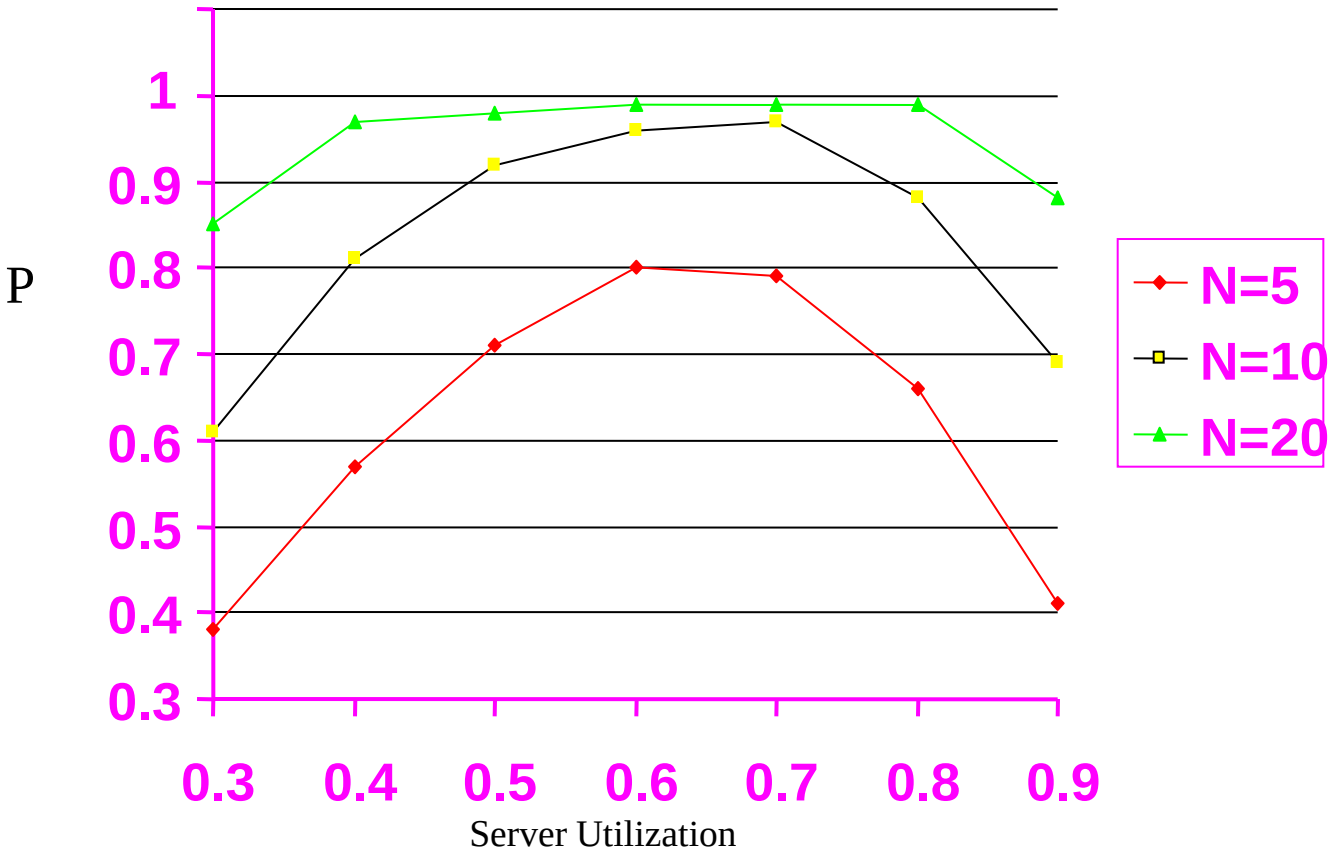
University of Arizona

Why Load Distribution?



Motivation

Let P is the probability that one task is waiting for service, while there is at least one computer is idle



Issues in Load Distributing/Scheduling: Load Definition

- it has been shown that a resource queue length is a good indicator.
  - For example, measuring the CPU queue length correlates to task response time and can be obtained with little overhead
- In interactive environment, it is suggested that queue length has little correlation with processor utilization.

Issues in Load Distributing/Scheduling: AlgorithmType

The basic function involves transferring load (tasks) from heavily loaded computers to idle or lightly loaded computers

Algorithms can be classified as: Static, dynamic or adaptive

Issues in Load Distributing: Load Balancing vs. Load Sharing

both algorithms try to reduce the likelihood of having one or more computers idle while tasks contend for service at another computer

load balancing goes a step further to equalize loads at all computers; this results in higher transfer rates and thus more overhead than load sharing algorithms

Issues in Load Distributing: Preemptive vs. Nonpreemptive Transfers

- Preemptive involves transferring a task that is partially executed
  - It is an expensive operation
  - Task state consists of a virtual memory images, a process control block, unread I/O buffers and messages, file pointers, timers that have been set, etc.
- Nonpreemptive transfers involve transferring of tasks that have not begun execution and thus do not require the transfer of the task state
- The execution environment of the transferred tasks need to be transferred to the receiving node
  - this includes user's current working directory, the privileges inherited by the task, etc.

Load Balancing in General

Enormous and diverse literature on load balancing

Computer Science systems

Computer Science theory

Operations research (OR)

Application domains

A closely related problem is scheduling, which is to determine the order in which tasks run

Understanding Load Balancing Problems

Load balancing problems differ in:

Tasks costs

Do all tasks have equal costs?

If not, when are the costs known?

Before starting, when task created, or only when task ends

Task dependencies

Can all tasks be run in any order (including parallel)?

If not, when are the dependencies known?

Before starting, when task created, or only when task ends

Locality

Is it important for some tasks to be scheduled on the same processor (or nearby) to reduce communication cost?

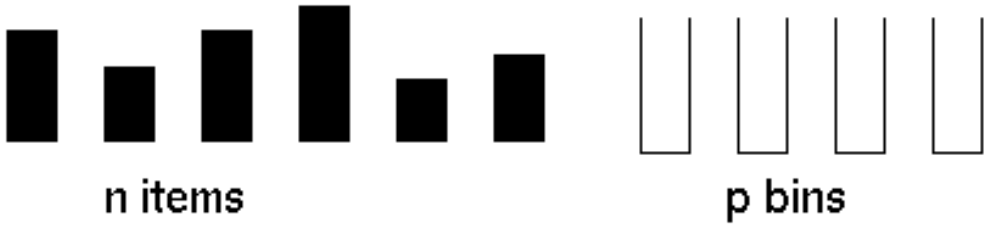
When is the information about communication between tasks known?

Task cost spectrum

Schedule a set of tasks under one of the following assumptions:

Easy: The tasks all have equal (unit) cost. branch–free loops

Harder: The tasks have different, but known, times. sparse matrix–vector multiply



Hardest: The task costs unknown until after execution. GCM, circuits

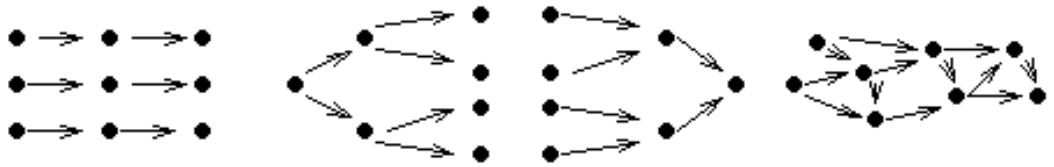
Task Dependency Spectrum

Schedule a graph of tasks under one of the following assumptions:

Easy: The tasks can execute in any order.

dependence free loops

Harder: The tasks have a predictable structure.



wave-front out-tree in-tree general dag

balanced or unbalanced

matrix computations (dense, and some sparse, Cholesky)

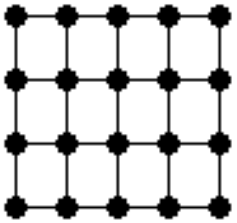
Hardest: The structure changes dynamically (slowly or quickly) search, sparse LU

Task Locality Spectrum (Data Dependencies)

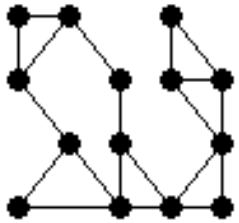
Schedule a set of tasks under one of the following assumptions:

Easy: The tasks, once created, do not communicate. — embarrassingly parallel

Harder: The tasks communicate in a predictable pattern. — PDE solver



regular



irregular

Hardest: The communication pattern is unpredictable. — discrete event simulation

Spectrum of Solutions

One of the key questions is when certain information about the load balancing problem is known

Leads to a spectrum of solutions:

All information is available to scheduling algorithm, which runs before any real computation starts. (Static/offline algorithms)

Semi-static scheduling. Information may be known at program startup, or the beginning of each timestep, or at other well-defined points. Offline algorithms may be used even though the problem is dynamic.

Dynamic scheduling. Information is not known until mid-execution. (online algorithms)

Load Balancing Algorithms

Early works focused on static placement techniques

- seeks optimal or near optimal solution to processor allocation

Recent works evolved to adaptive load balancing with process migration

A simple approach is to have a central allocation processor that receives periodically

- load information form all processors
- makes process placement decisions
- it has a single point of failure and can be a bottleneck

Distributed process allocation is another complex alternative

Static Load Balancing

Find an allocation of processes to processors to

- minimize execution cost
- minimize communication cost

Assumptions:

- program consists of a number of modules
- cost of executing a module on a processor
- volume of data flow between modules

Problem formulation:

- assign modules to processors in an optimal manner within their given cost constraints
- does not consider the current state of the system when making the placement decisions

Evaluation of load balancing

Efficiency

Communication

Partitioning Techniques

Regular grids (-: Easy :-)

striping

blocking

use processing power to divide load more fairly

Generalized Graphs

Levelization

Scattered Decomposition

Recursive Bisection

Levelization

Begin with a boundary

Number these nodes 1

All nodes connected to a level 1 node are labeled 2, etc.

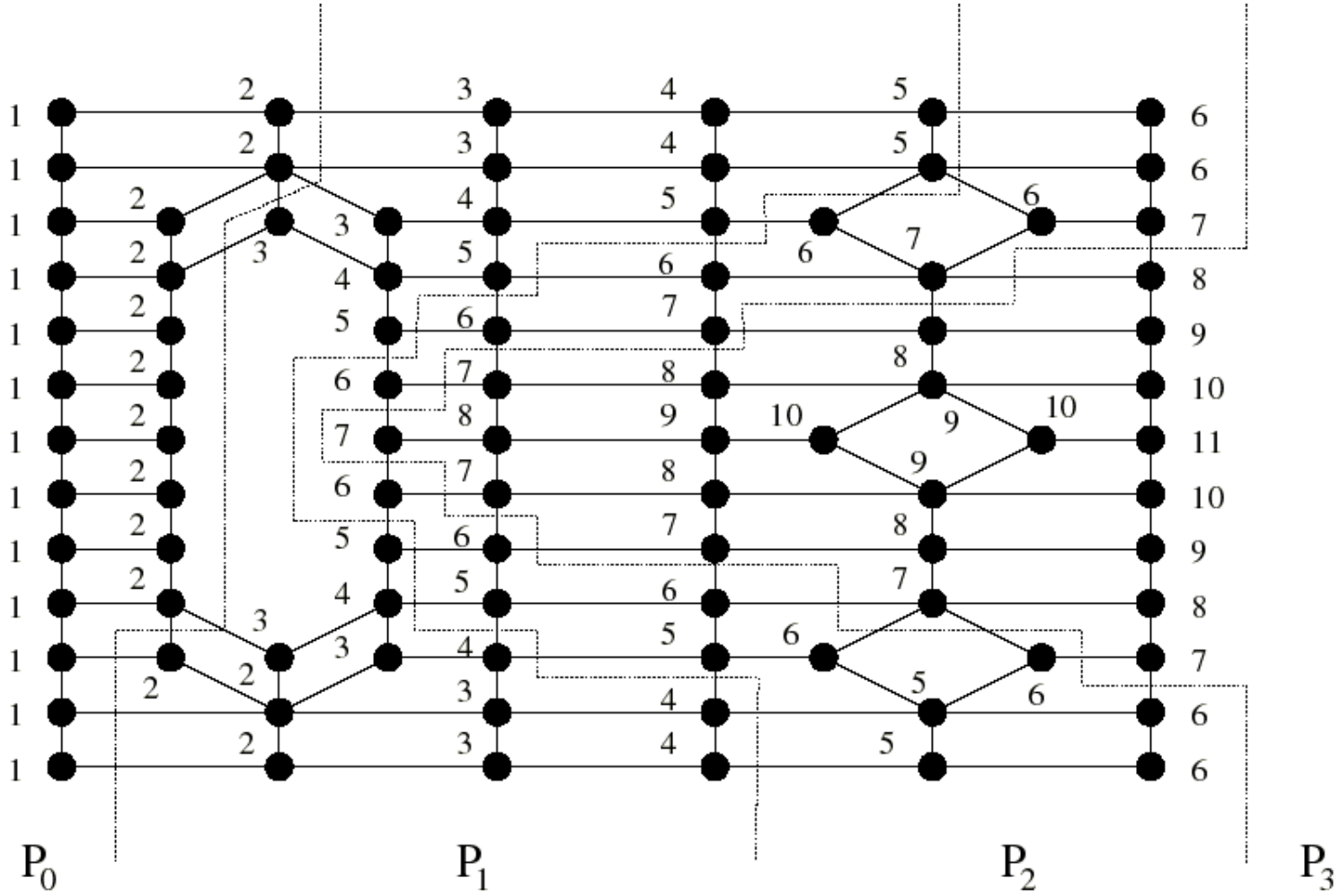
Partitioning is performed

determine the number of nodes per processor

count off the nodes of a level until exhausted

proceed to the next level

Levelization



Scattered Decomposition

Used for highly irregular grids

Partition load into a large number r of rectangular clusters such that $r >> p$

Each processor is given a disjoint set of r/p clusters.

Communication overhead can be a problem for highly irregular problems.

Greedy Bisection

Start with a vertex of the smallest degree

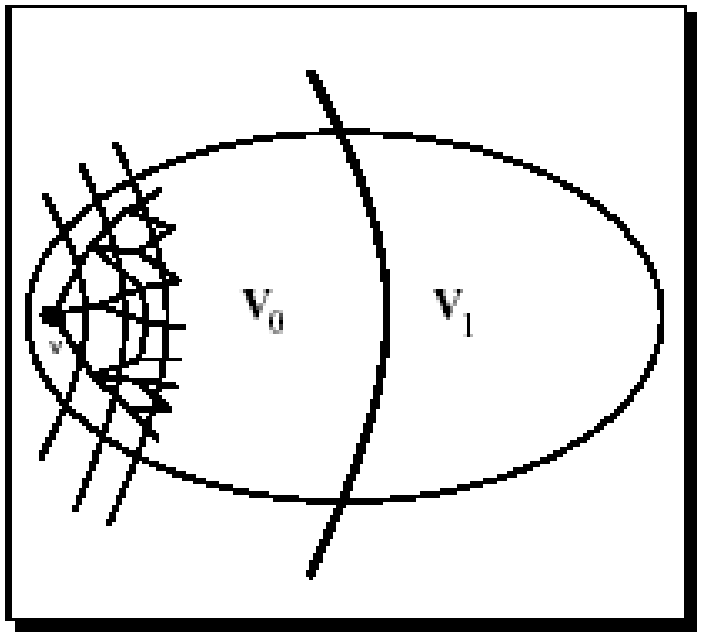
least number of edges

Mark all its neighbors

Mark all its neighbors neighbors, etc.

The first n/p marked vertices form one subdomain

Apply the algorithm on the remaining



The O-1 Integer Programming Approach

C_{ij} : coupling factor = the number of data units transferred form module i to module j.

d_{kl} : interprocessor distance = the cost of transferring one data unit from processor k to processor l.

q_{ik} : execution cost = the cost of running module i on processor k.

The O-1 Integer Programming Approach

If i and j are resident on processors k and l respectively, then their total communications cost can be expressed as $C_{ij} * d_{kl}$.

In addition to these quantities the assignment variable is defined as:

$X_{ik} = 1$, if module i is assigned to processor k.

0, otherwise

Using the above notation, the total cost of processing a number of user modules is given as:

$$\Sigma_i \Sigma_k \left(q_{ik} X_{ik} + \Sigma_l \Sigma_j \left(c_{ij} * d_{kl} \right) X_{ik} X_{jl} \right)$$

The O-1 Integer Programming Approach

In this scheme, constraints can be added easily to the problem (Memory constraints)

$$\Sigma_i \; M_i X_{ik} <= S_k$$

M_i = memory requirements of module i

S_k = memory capacity of processor k

Non-linear programming techniques or branch and bound techniques can be used to solve this problem

The complexity of such algorithms is NP-complete

Main disadvantage is the need to specify the values of large number of parameters

Semi-Static Load Balance

If domain changes slowly over time and locality is important

- use static algorithm
- do some computation (usually one or more timesteps) allowing some load imbalance on later steps
- recompute a new load balance using static algorithm

Often used in:

- particle simulations, particle-in-cell (PIC) methods
  - poor locality may be more of a problem than load imbalance as particles move from one grid partition to another
- tree-structured computations
- grid computations with dynamically changing grid, where changes are slowly

Self-Scheduling

Self scheduling:

- Keep a centralized pool of tasks that are available to run
- When a processor completes its current task, it picks up a task from the pool
- If the computation of one task generates more tasks, add them to the pool

Originally used for:

- Scheduling loops by compiler (really the runtime-system)
- Original paper by Tang and Yew, ICPP 1986

When is Self-Scheduling a Good Idea?

Useful when:

A batch (or set) of tasks without dependencies

- can also be used with dependencies, but most analysis has only been done for task sets without dependencies

The cost of each task is unknown

Locality is not important

Using a shared memory multiprocessor, so a centralized pool of tasks is fine

Variations on Self-Scheduling

Typically, don't want to grab smallest unit of parallel work.

Instead, choose a chunk of tasks of size K.

- If K is large, access overhead for task queue is small
- If K is small, we are likely to have even finish times (load balance)

Four variations:

- Use a fixed chunk size
- Guided self-scheduling
- Tapering
- Weighted Factoring

Fixed Chunk Size

Apply a technique for computing the optimal chunk size

Requires a lot of information about the problem characteristics

- e.g., task costs, number

Results in an off-line algorithm. Not very useful in practice.

- For use in a compiler, for example, the compiler would have to estimate the cost of each task
- All tasks must be known in advance

Guided Self-Scheduling

Idea: use larger chunks at the beginning to avoid excessive overhead and smaller chunks near the end to even out the finish times.

The chunk size K_i at the ith access to the task pool is given by

$$\text{ceiling}(R_i/p)$$

where R_i is the total number of tasks remaining and

p is the number of processors

See Polychronopolous, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," IEEE Transactions on Computers, Dec. 1987.

Tapering

Idea: the chunk size, K_i is a function of not only the remaining work, but also the task cost variance

- variance is estimated using history information
- high variance => small chunk size should be used
- low variant => larger chunks OK

Distributed Task Queues

The obvious extension of self-scheduling to distributed memory is:

a distributed task queue (or bag)

When are these a good idea?

Distributed memory multiprocessors

Or, shared memory with significant synchronization overhead

Locality is not (very) important

Tasks that are:

known in advance, e.g., a bag of independent ones

dependencies exist, i.e., being computed on the fly

The costs of tasks is not known in advance

Engineering Distributed Task Queues

A lot of papers on engineering these systems on various machines, and their applications

If nothing is known about task costs when created

- organize local tasks as a stack (push/pop from top)
- steal from the stack bottom (as if it were a queue), because old tasks likely to cost more

If something is known about tasks costs and communication costs, can be used as hints. (See Wen, UCB PhD, 1996.)

- Part of Multipol (www.cs.berkeley.edu/projects/multipol)
- Try to push tasks with high ratio of cost to compute/cost to push
  - Ex: for matmul, ratio = $2n^3$ cost(flop) / $2n^2$ cost(send a word)

Goldstein, Rogers, Grunwald, and others (independent work) have all shown

- advantages of integrating into the language framework
- very lightweight thread creation

CILK (Leicerson et al) (supertech.lcs.mit.edu/cilk)

DAG Scheduling

For some problems, you have a directed acyclic graph (DAG) of tasks

- nodes represent computation (may be weighted)
- edges represent orderings and usually communication (may also be weighted)
- not that common to have the DAG in advance

Two application domains where DAGs are known

- Digital Signal Processing computations
- Sparse direct solvers (mainly Cholesky, since it doesn't require pivoting).
- The basic offline strategy: partition DAG to minimize communication and keep all processors busy
- NP complete, so need approximations
- Different than graph partitioning, which was for tasks with communication but no dependencies
- See Gerasoulis and Yang, IEEE Transaction on P&DS, Jun '93.

Mixed Parallelism

As another variation, consider a problem with 2 levels of parallelism

course-grained task parallelism

- good when many tasks, bad if few

fine-grained data parallelism

- good when much parallelism within a task, bad if little

Appears in:

Adaptive mesh refinement

Discrete event simulation, e.g., circuit simulation

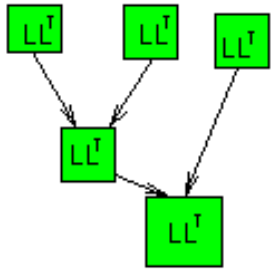
Database query processing

Sparse matrix direct solvers

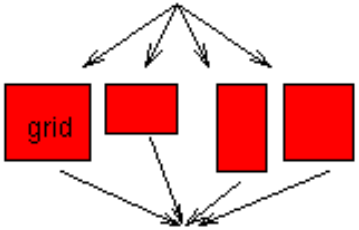
Mixed Parallelism Strategies

Many applications have course-grained task parallelism and fine-grained data parallelism

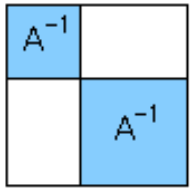
sparse cholesky



adaptive mesh refinement



sign function



blocks are data-parallel tasks within a task parallel execution

Questions:

Should the execution use only data parallelism, only task parallelism, or a mixture?

What is the relative benefit?

What is a good scheduling algorithm?

Approach:

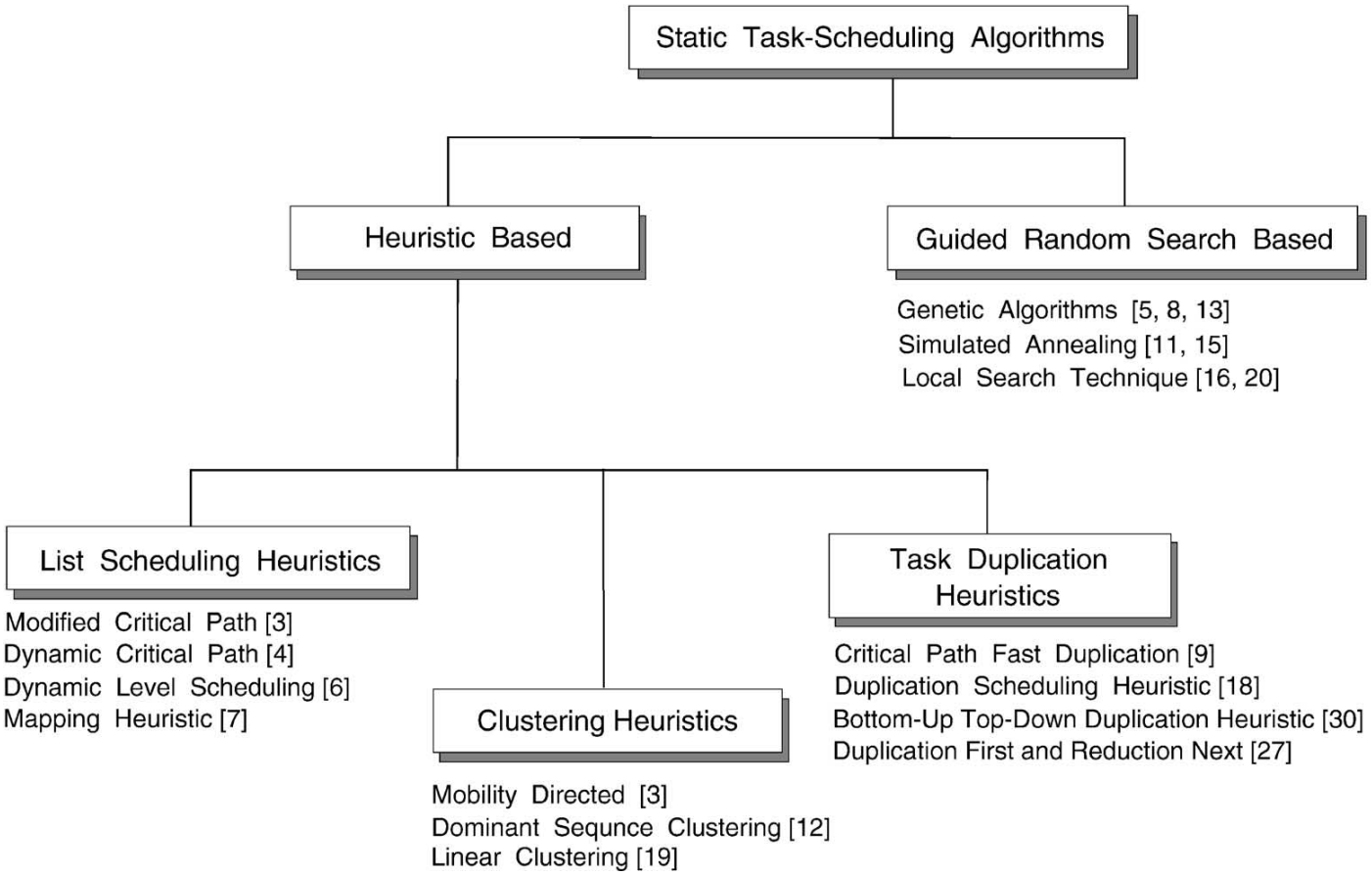
Use modeling, validated by experiments to predict performance

Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing

IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 3, March 2002, Topcuoglu, Hariri, Wu

- Scheduling Algorithms
  - Listing – order tasks based on priority, and then processor allocation to minimize objective function
  - Clustering – map a given graph to an unlimited number of clusters
  - duplication
  - Guided random search based on Genetic algorithms – long execution time

Classification of Static Scheduling Algorithms



- Task scheduling for heterogeneous processors is less studied due to the increase in complexity
  - Heterogeneous Earliest-Finish Time (HEFT) Algorithm
  - Critical Path on a Processor (CPOP) Algorithm

Task-Scheduling Problem

Assume that we know the execution time for each task, ni on each processor pj (computation cost matrix)

- Communication cost between tasks
- Define Earliest Start Time (EST) and Earliest Finish Time (EFT) for each task. We compute that for each task in the graph starting from the entry task.
- Actual Start Time AST (nm) and Actual Finish Time AFT (nm) determine the actual time to start a task and its finishing time on an assigned processor
- Makespan is the actual finish time after all graph tasks are scheduled and complete their execution
- Objective Function
  - Find the schedule of tasks that minimizes the makespace (schedule length is minimized)

Dynamic Level Scheduling (DLS) Algorithm

- each step, the algorithm selects (ready node, available processor) that maximizes the dynamic level: smallest time to start from a given task
- Mapping Heuristic (MH), estimate the cost of running each task on each processor, schedule tasks that minimizes the ready time (when the processor is ready to execute the next task)
- Levelized-Min Time (LMT) algorithm
  - It is a two phase Algorithm
  - Phase I: group all tasks that can execute in parallel using the level attribute
  - Phase 2: assign each task to fastest available processor
  - Each task is assigned to a processor that minimizes the sum of the task's computation cost and the total communication costs with the tasks in the previous levels

HEFT and CPOP Algorithms

They are based on upward and downward ranking

Upward rank represents the length of the critical path from task ni to the exist task, including the communication cost of task ni, computing from the exit node

-Downward ranking represents the length of the critical path from ni to the entry task of the

- Heterogeneous-Earliest Finish Time (HEFT) Algorithm

- Task Prioritizing Phase- for computing the priorities of all tasks

- Processor Selection Phase: for selecting the tasks in the order of their priorities and scheduling each task on its best processor, which minimizes the task's finish time

- Rank tasks based on upward rank value

HEFT Algorithm

- Set the computation costs of tasks and communication costs of edges
- Compute rank for all tasks by traversing graph upward, starting from the exit task
- Sort all tasks in a scheduling list by decreating order of rank
- While loop
  - - select first task, ni from the lsit
  - For each processor compute EFT (ni, pk)
  - Assign task ni to the processor pj that minimizes EFT of task Ni

HEFT Algorithm

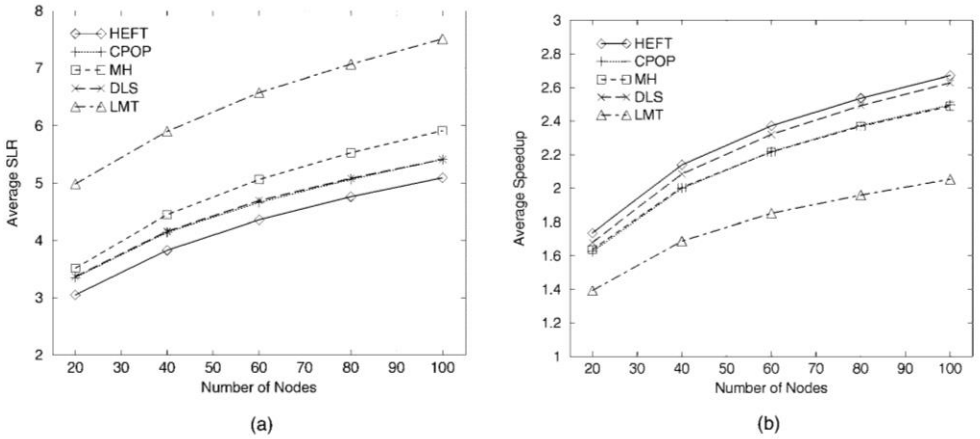
1. Set the computation costs of tasks and communication costs of edges with mean values.
2. Compute $rank_u$ for all tasks by traversing graph upward, starting from the exit task.
3. Sort the tasks in a scheduling list by nonincreasing order of $rank_u$ values.
4. **while** there are unscheduled tasks in the list **do**
5. Select the first task, n_i, from the list for scheduling.
6. **for** each processor p_k in the processor-set ($p_k \in Q$) **do**
7. Compute $EFT(n_i, p_k)$ value using the *insertion-based scheduling* policy.
8. Assign task n_i to the processor p_j that minimizes EFT of task n_i.
9. **endwhile**

Critical Path on a Processor (CPOP)

1. Set the computation costs of tasks and communication costs of edges with mean values.
2. Compute $rank_u$ of tasks by traversing graph upward, starting from the exit task.
3. Compute $rank_d$ of tasks by traversing graph downward, starting from the entry task.
4. Compute $priority(n_i) = rank_d(n_i) + rank_u(n_i)$ for each task n_i in the graph.
5. $|CP| = priority(n_{entry})$, where n_{entry} is the *entry* task.
6. $SET_{CP} = \{n_{entry}\}$, where SET_{CP} is the set of tasks on the critical path.
7. $n_k \leftarrow n_{entry}$.
8. **while** n_k is not the exit task **do**
9. Select n_j where $((n_j \in succ(n_k))$ **and** $(priority(n_j) == |CP|))$.
10. $SET_{CP} = SET_{CP} \bigcup \{n_j\}$.
11. $n_k \leftarrow n_j$.
12. **endwhile**
13. Select the critical-path processor (p_{CP}) which minimizes $\sum_{n_i \in SET_{CP}} w_{i,j}, \quad \forall p_j \in Q$.
14. Initialize the priority queue with the entry task.
15. **while** there is an unscheduled task in the priority queue **do**
16. Select the highest priority task n_i from priority queue.
17. **if** $n_i \in SET_{CP}$ **then**
18. Assign the task n_i on p_{CP}.
19. **else**
20. Assign the task n_i to the processor p_j which minimizes the $EFT(n_i, p_j)$.
21. Update the priority-queue with the successors of n_i, if they become ready tasks.
22. **endwhile**

Scheduling Metrics

1. Schedule Length Ratio (SLR). The ratio of the schedule length time over the shortest possible execution of the graph
2. Speedup: The sequential execution time of the graph over the parallel execution
3. Number of Occurrences of Better Quality of Schedules
    1. Number of time each algorithm produced better results
4. Running Time of the Algorithm
    1. How long it takes the algorithm to produce its results
5. Build a random graph generator to compare different algorithms (around 56K different graphs are used in the comparison)



a) Average SLR and (b) average speedup with respect to graph size.

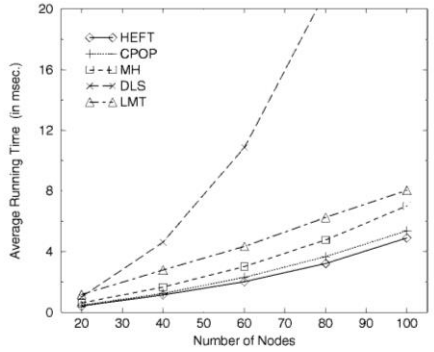


Fig. 7. Average running time of algorithms with respect to graph size.

TABLE 2
Pair-Wise Comparison of the Scheduling Algorithms

		HEFT	CPOP	DLS	MH	LMT	Combined
HEFT	better		45181	42709	49730	56059	86%
	equal	*	215	802	689	2	1%
	worse		10854	12739	5831	189	13%
CPOP	better	10854		24774	34689	53922	55%
	equal	215	*	108	76	3	< 1%
	worse	45181		31368	21485	2325	45%
DLS	better	12739	31368		44056	55873	64%
	equal	802	108	*	2170	1	1%
	worse	42709	24774		10024	376	35%
MH	better	5831	21485	10024		55342	41%
	equal	689	76	2170	*	6	1%
	worse	49730	34689	44056		902	58%
LMT	better	189	2325	376	902		2%
	equal	2	3	1	6	*	< 1%
	worse	56059	53922	55873	55342		98%

Dynamic Load Balancing

- Consider adaptive algorithms
- After an interval of computation
  - mesh is adjusted according to an estimate of the discretization error
    - coarsened in areas
    - refined in others
- Mesh adjustment causes load imbalance

Repartitioning

Consider: dynamic situation is simply a sequence of static situations

Solution: repartition the load after each

some partitioning algorithms are very quick

Issues

scalability problems

how different are current load distribution and new load distribution

data dependencies

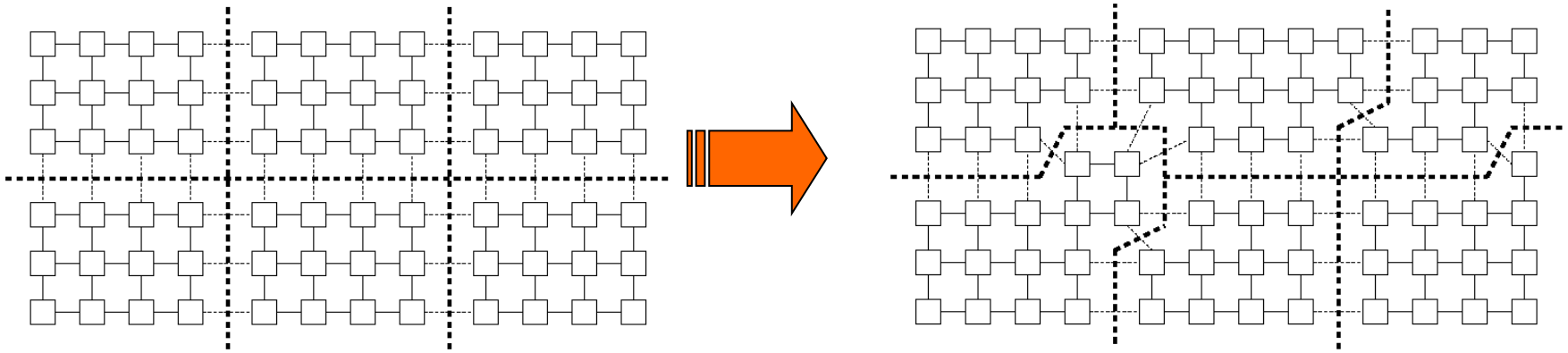
Dynamic Load Balancing

Load is statically partitioned initially

Adjust load when an imbalance is detected.

Objectives

- rebalance the load
- keep edge cut minimized (communication)
- avoid having too much overhead



Centralized DLB

Control of the load is centralized

Two approaches

Master-worker (Task scheduling)

- Tasks are kept in central location
- Workers ask for tasks
- Requires that you have lots of tasks with weak locality requirements.
  No major communication between workers

Load Monitor

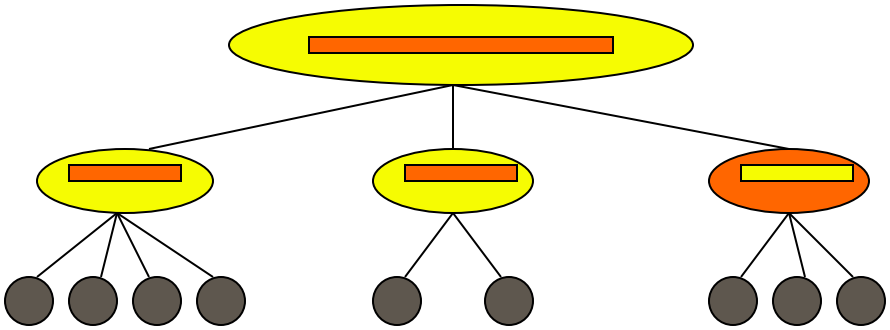
- Periodically, monitor load on the processors
- Adjust load to keep optimal balance

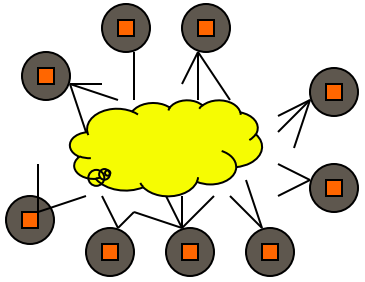
Decentralizing DLB

Generally focused on work pool

Two approaches

- Hierarchy
- Fully distributed





Adaptive Load Balancing

Need to avoid high communication overhead during load information exchange

Need to guard against instability

migrate many processes to lightly loaded processor that soon becomes heavily loaded

=> processor thrashing

How do you evaluate the Performance of LB Algorithms

Adaptive Load Balancing (cont)

DLB Algorithm components:

- Activation Policy – When we start the LB Algorithm
  - most of the techniques proposed are threshold policies.
  - When the load on one computer exceeds certain threshold, it becomes heavily loaded (sender)
- Selection Policy: Which Process/Task to migrate?
- Location Policy: Who can share load?
  - Polling is a common technique to find out whether a node is suitable for load sharing.
- Information Policy: What type of information?
  - it determines what information about the states of other nodes need to be collected, where it should be collected from, and when/how offten it should be collected.

Information Policy

Most of information policies are based on the following three types:

- Demand-driven. A node collects information about other nodes only when it becomes either a sender (busy) or a receiver node (idle)
- Periodic. Nodes exchange load every period T
- State-driven: whenever nodes state changes by a certain degree

Information Policy: Processor Load

We need to obtain from the OS the current load on each processor

The measure will be calculated frequently and thus should be done efficiently

It should adopt swiftly to changes in load state

One could use a load estimation program that constantly runs to determine the intervals between successive runs.

Information Policy: Processor Load

Use the Unix five-minute average which gives the length of the run queue

Use number of processes ready to run on the CPU at a given instant of time

We do need to maintain stability:

- cost of load balancing do not outweigh its benefits over a system using no load balancing

=> Load value should be averaged over a period at least as long as the time to migrate on average process

virtual load = actual load + migrated processes load

Processor Load Measurement (cont)

Bryant and Finkel (1981) used the remaining service time $R_E(t_K)$ to estimate the response time of a process arriving at one processor *J(P)* which has J jobs in its queue

R = $R_E(t_K)$; remaining time equal to current service time

for all *J* belong to *J(P)* Do

begin

if $R_E(t_J) < R_E(t_K)$

then $R = R + R_E(t_J)$

else $R = R + R_E(t_K)$

end

$RSP_E(K, J(P)) = R$

Where *J(P)*: set of jobs resident on processor P.

Processor Load Measurement (cont)

Ferrari (1985) proposed a linear combination of all main resource queues as a measure of load

This technique determines the response time of a Unix command in terms of resource Queue lengths

The analysis assumes steady-state system and certain queueing disciplines for resources

Information Policy: Frequency of Information Exchange

The state information will vary in its degree of accuracy since computers are loosely coupled

We need sufficient accuracy

to avoid instability, we need to increase frequent load exchange

=> This degrades performance

Information Policy: Demand Driven:

In Maitre d' systems (Bershad 1985), one daemon process examines the Unix five-minute load average. If the processor can handle more processes, it will broadcast this availability

Other alternative is to broadcast a message when the processor becomes idle ==> announcing willingness to accept migrating processes

This approach works efficiently if the network uses a broadcast communications medium

Information Policy: Demand Driven

Global System Load Approach

Processors calculate the load on the whole system and adjust their own load relative to this global value

When the processor load differs significantly from the average load, load balancing is activated

The difference should not be too small and also not too large

Information Policy: Demand Driven

Gradient model algorithm:

view global load in terms of a collection of distances from a lightly loaded processor

the proximity (W_i of a processor is calculated as its minimum distance form a lightly loaded processor

g_k is set to zero if the processor is lightly loaded

$W_i = min_K \{d_{iK}$, over K where $g_k = 0\}$ if $\exists K | g_K = 0$

$W_i = W_{max}$, if forall K, $g_k = W_{max}$,

$W_{max} = D(N) + 1$

$D(N) = max\{d_{iJ}$, forall i, J belong to N}; Diameter Distance

Information Policy - Demand Driven

global load is then represented by a gradient surface

$$GS = (W_1 , W_2 , \ldots W_n)$$

it gives a route to a lightly loaded processor with minimum cost

Information Policy: Periodic Exchange

each processor cyclically sends load information to each of its neighbors; to pair with a processor that differs greatly from its own

load information consists of a list of all local jobs, together with jobs that can be migrated to the sender of load message

Information Policy: State-change Driven

Load vector of a processor neighbors is maintained and updated when a state transition occurs: L_N, N_L, N_H, H_N

to reduce number of messages sent

N - L_load message is sent when N_L transition if previous state was heavy

- broadcast N_H transitions and only notify neighbors of H_N transition when process migration is negotiated

Activation Policy

Static threshold values:

when load goes beyond this threshold, processes should be off loaded

- this value is chosen experimentally

Under loaded processor could seek to accept processes form other peer processors

Activation Policy (cont)

Difference of a processor's load from that of its peers can be used to change node status (sender or receiver)

- When difference exceeds some bias, then migration is a viable proposition

- Examine periodically the response time of processes if moved to a remote processor

- If the response time is significantly better, the processes are migrated

Selection Policy

It selects a task for transfer once a node is identified as a sender

Which processes to be migrated?

- consider only newly-arriving processes
- we need to limit the number of times a process is permitted to migrate
- move the one that will benefit most from remote execution.

Selection Policy - Cont.

Kreuger and Finkel (1984) proposed the following:

1. Migration of a blocked process may not prove useful, since this may not effect local processor load
2. 2. Extra overhead will be incurred by migrating the currently scheduled and running process
3. The process with longer response time can better afford the cost of migration
4. Smaller processes put less load on the communications network
5. The process with the highest remaining service time will benefit most in the long-term from migration
6. Processes which communicate frequently with the intended destination processor will reduce communications load if they migrate
7. Migrating the most locally demanding process will be of greatest benefit to local load reduction

Location Policy

Define a method by which processors cooperate to find a suitable location for a migrating process

Methods can be categorized into two groups: sender-initiated and receiver initiated methods

initiating load-balancing from an overloaded processor is widely studied

Eager (1986) studied 3 simple algorithms

- activation policy is a simple static threshold

(a) choose a destination processor at random for a process migrating from a heavily-loaded processor.

Number of transfers is limited to only one

Location Policy

(b) choose a processor at random and then probe its load.

If it exceeds a static threshold,

another processor is probed and so on until one is found in less than a given number.

Otherwise, the process is executed locally

(c) poll a fixed number of processors, requesting their current queue lengths and selecting the one with the shortest queue

Location Policy (cont)

Stankovic (1984) proposed three algorithms which are based on relative difference between processor loads

- information exchange is through periodically broadcasting local values

(a) choose least-loaded processor if load difference is larger than a given bias

(b) if difference > bias 1, migrate one process

(c) if difference > bias 2, migrate two processes

(d) similar to (a), except no further migration to that processor for a given period 2t

Location Policy (cont)

When a processor becomes overloaded, it broadcasts the fact

under-loaded processor responds and indicates number of processes that can be accepted and adjusts its load

if no response, it assumes that the average value is too low and increases this global value and then broadcasts it

it adopts fast to fluctuations in system load

Receiver-Initiated Approaches

(a) when the load on one processor falls below the static threshold (T), it polls random processors to find one where if its process is migrated would not cause its load to be below T.

Receiver-Initiated Approaches - cont.

(b) to avoid migrating an executing process,

a reservation is made to migrate the

next newly-arriving process

simulation results showed that it does not perform as well as (a) approach

Location Policy (cont)

For broadcast networks, Livny and Melman (1982) proposed two receiver-initiated policies:

(a) A node broadcasts a status message when it becomes idle

Location Policy (cont)

A node broadcasts a message when it becomes idle

Receivers carry out the following actions:

i. If $n_i > 1$ continue to step ii, else terminate algorithm.

ii. Wait D/n time units, where D is a parameter depending on the speed of the communications; by making this value dependent on processor load, more heavily-loaded processors will respond more quickly.

iii. Broadcasting a reservation message if no other processor has already done so (if this is the case terminate algorithm).

iv. Wait for reply

v. If reply is positive, and $n_i > 1$, migrate a process to the idle processor.

Location Policy (cont)

(a) method might overload the communication medium, so a second method is to replace broadcasting by polling when idle. The following steps are taken when a processor's queue length reaches zero.

i. Select a random set of R processors ($a_{i,} \ldots a_N$) and set a counter j=1.

ii. Send a message to processor a_j and wait for a reply.

iii. The reply from a_j will either be a migrating process or an indication that it has no processes.

iv. If the processor is still idle and $j<R$, increment j and go to step ii else stop polling.

Load Balancing Algorithms

- Sender-Initiated Algorithms
- Receiver-Initiated Algorithms
- Symmetrically Initiated Algorithms
- Adaptive Algorithms
  - A Stable Symmetrically Initiated Algorithm
  - A Stable Sender-Initiated Algorithm

Sender-Initiated Algorithms

We study three simple algorithms presented by Eager et al

Transfer policy: All three algorithms use a threshold policy based on CPU queue length

- a node is identified as a sender if the originating task at that node makes the queue length > T
- a node is identified as a receiver for a remote task if its queue length is still < T when it accepts that task

Selection Policy: consider only newly arrived tasks for transfer

Location Policy: Random, Threshold, Shortest

Sender-Initiated Algorithms- Cont.

Location Policy: Random Policy

a task is simply transferred to a node selected at random

useless task transfers can occur

it is instable algorithm

it provides performance improvement over no load sharing if the load is moderately low to average

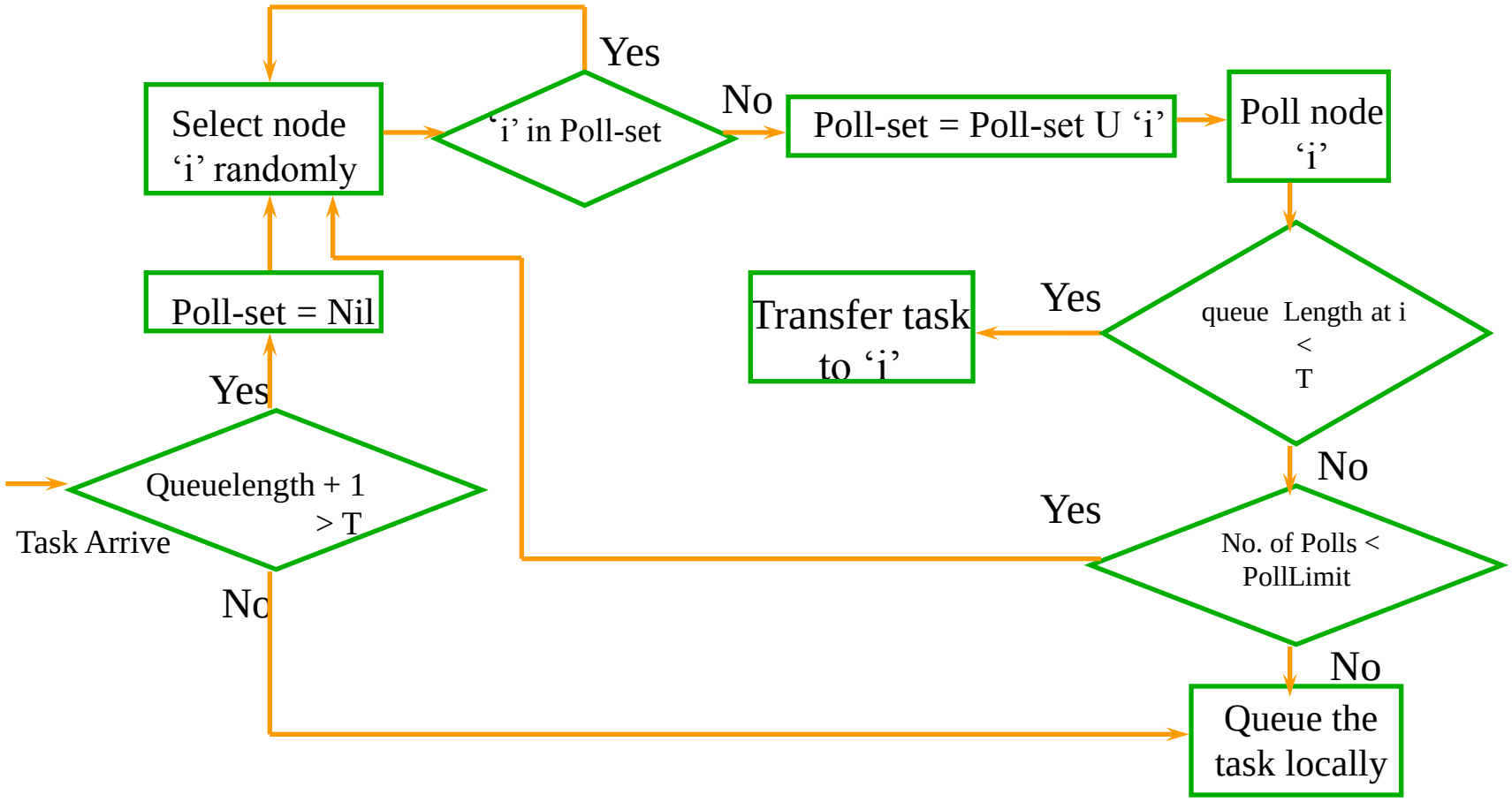
Sender-Initiated Algorithms- Cont

Location Policy: Threshold

useless task transfers can be avoided by polling a node (selected at random) to determine if it is a receiver

if so, the task is transferred to the selected node

Sender-Initiated Load Sharing with Threshold Location Policy



Sender-Initiated Algorithms - Cont.

Location Policy: Shortest

a number of nodes are selected at random and are polled to determine their queue length

the node with the shortest queue length is selected as the destination for task transfer unless its

queue length >= T

the performance improvement of shortest location policy over threshold location policy was shown to be marginal

Information Policy: It is based on demand-driven policy

Receiver-Initiated Algorithms

the load distributing activity is initiated from an under-loaded node (receiver) trying to obtain a task from a sender node

Activation (Transfer) Policy: It is based on a threshold policy using CPU queue length. The transfer policy is triggered when a task departs. If the local queue length falls below T, the node is identified as a receiver

Selection Policy:

select the newly arrived tasks

the overhead incurred by transferring the task is less than the expected performance

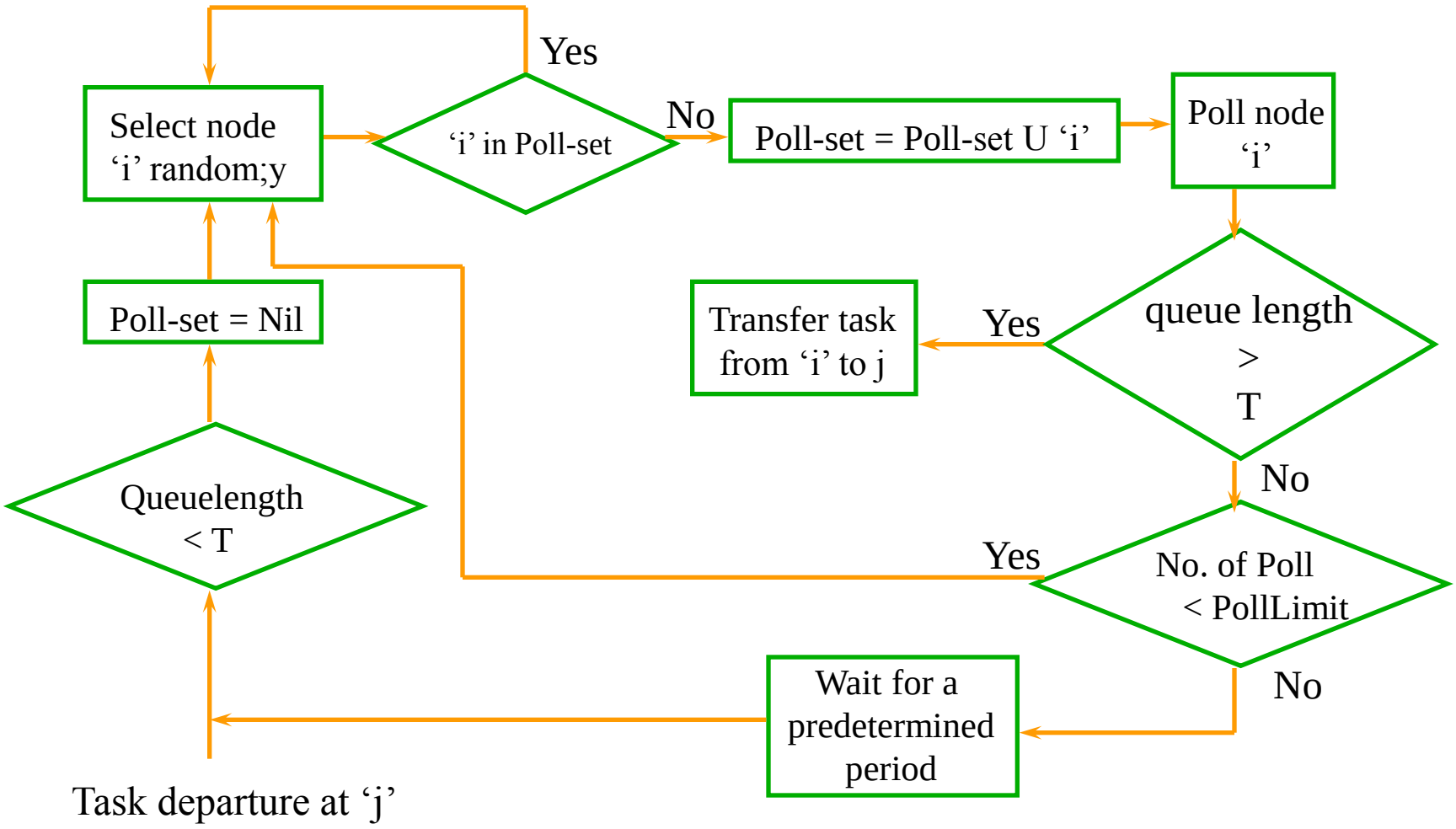
a task is selected if response time will be improved upon transfer

Receiver-Initiated Algorithms

Location Policy: a node selected at random is polled to determine if transferring a task from it would not place its queue length below T

Information Policy: it is based on demand-driven because the polling activity starts only after a node becomes a receiver

Receiver-Initiated Load Sharing



Receiver-Initiated Algorithm

Stability: do not cause system instability

at high loads, there is a high probability to find a sender

at low loads, there are few senders, but more receiver-initiated polls

Receiver-Initiated Algorithm

Drawbacks:

under most of CPU scheduling techniques, newly arrived tasks are provided quickly quantoms of service

consequently, most of task transfers are preemptive and thus are expensive

sender-initiated algorithms can make a greater use of non-preemptive transfers

Symterically Initiated Algorithms

- both senders and receivers search for receivers and senders, respectively
- at low system loads, sender-initiated component is more successful in finding underloaded nodes
- at high system loads, receiver-initiated component is more successful in finding overloaded nodes
- this scheme has the disadvantages of both schemes.

The Above-Average Algorithm

- proposed by Krueger and Finkel, it tries to maintain the load at each node within an acceptable range of the system average
- transfer policy: it is a threshold policy that uses two adaptive thresholds:

  upper and lower thresholds that are equidistant from the node's estimate of the average load across all nodes

  nodes above the upper threshold are considered senders

  while those less than the lower thresholds are considered receivers

  nodes between these two thresholds are considered to be acceptable

- location policy: it has two components: sender-initiated component

The Above-Average Algorithm-Cont.

⑩ Sender-Initiated Component

a sender broadcasts a TooHigh messsage, set TooHigh timeout alarm, and listens for an Accept message

a receiver that receivers a TooHigh message cancels its TooLow timeout, sends an Accept message, increases its load value, and sets an AwaitingTask timeout

on receiving an Accept message, the sender node chooses the best task to transfer

on expiration of TooHigh timeout, if no Accept message has been received, the sender broadcasts a ChangeAverage message to increase the average load estimate at the other nodes

The Above-Average Algorithm-Cont

⑩ Receiver-Initiated Component

when a node becomes a receiver, it broadcasts a TooLow message, sets a TooLow timeout alarm, and starts listenning for a TooHigh message

if a TooHigh message is received, the receiver sends an Accept message, increases its load, and sets a timeout alarme

if the TooLow timeout expires before receiving any TooHigh messages, the receiver broadcasts a ChangeAverage message to decrease the average load estimate at the other nodes

The Above-Average Algorithm-Cont

- Selection Policy: Can use any of the techniques discussed before.
- Information Policy: it is based on demand-driven policy.
  - the system load average is determined individually at each node without exchanging many messages.
  - the acceptable range determines the responsiveness of the algorithm
  - when the communication network is heavily (lightly) loaded, the acceptable range can be inccreased (decreased) by each node individually

Adaptive Algorithms- A Stable Symetrically Initiated Algorithm

- the main instability in the previous algorithms is caused by the indiscriminate polling by the sender's initiated component
- this scheme utilizes the information gathered during polling to classify nodes as senders, receivers, or OKs
- each node maintains a data structure that includes Sender List, Receiver List and OK List
- initially, each node assumes that every other node is a receiver
- Transfer Policy: it has two components:sender-initiated and receiver-initiated components

A Stable Symetrically Initiated Algorithm-Cont.

Sender-Initiated Component:

- the sender node polls the node at the head of the Receivers List
- the polled node puts the sender node at the head of its Senders List, and informs the sender about its status
- At the sender, if it is not a receiver, it is moved from the Receivers List and put in the proper list
- the polling process stops if a suitable receiver is found

A Stable Symetrically Initiated Algorithm-Cont

Receiver-Initiated Component

The nodes are polled in the following way:

- head to tail in the Senders List; use up-to-date information first
- tail to head in the OK List; use out-to-date information first
- tail to head in the Receivers List; use out-to-date information first

it stops when it found a sender node

A Stable Symetrically Initiated Algorithm-Cont

Selection Policy: sender initiated component considers only newly arrived tasks, while receiver-initiated component can use a variety of techniques

Information Policy: it is a demand-driven policy since the polling activity starts when a node becomes a sender or a receiver

A Stable Symetrically Initiated Algorithm-Cont

Discussion:

at high system loads, many unsuccessful polls for nodes result in their removal from the Receivers list

- thus future sender-initiated polls will terminate at high system loads

at low system loads, receiver-initiated polling generally fail, do not affect system performance since there is plenty CPU capacity available in the system

- with update information, the Receivers List is accurately reflecting the system state, and sender-initiated activity will succeed with a few polls

A Stable Sender-Initiated Algorithm

- this algorithm does not cause instability
- load sharing is only due to nonpreemptive task transfers
- it has the same sender-initiated component but modified receiver-initiated component

A Stable Sender-Initiated Algorithm

- Receiver-Initiated Component

  maintain a state vector about system state which it lets each node to keep track of which list (sender, receiver, OK) it belongs to at each node in the system

  when a node becomes a receiver, it informs all nodes that are misinformed about its current state

- no preemptive transfers because the sender-initiated component performs the load sharing

Performance Comparison

- **Assumptions:**

  the average service demand for tasks is one time unit

  task interarrival times and service demands are independently exponentially distributed

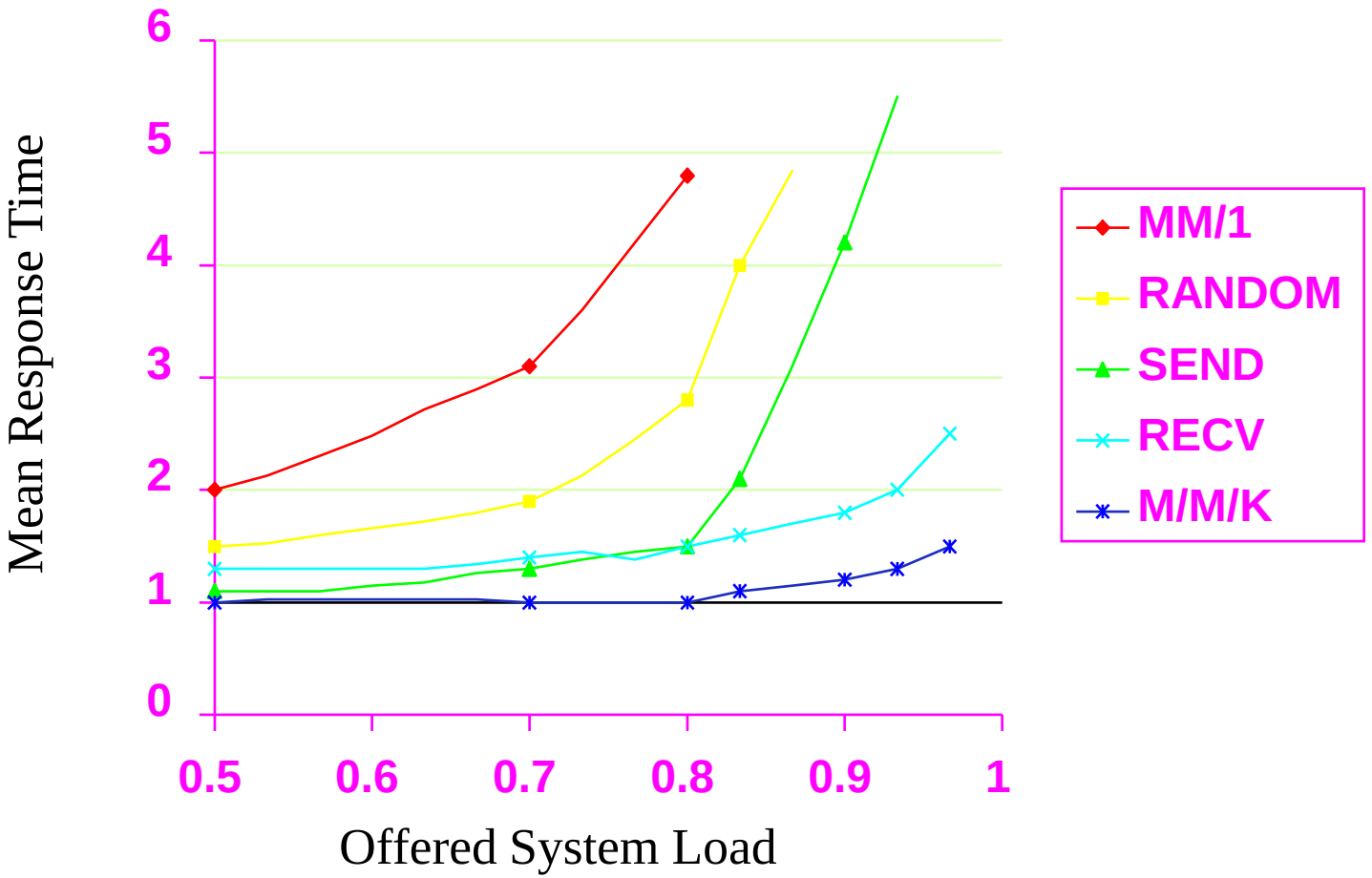
  the system load is homogeneous and has 40 identical nodes

Performance Comparison

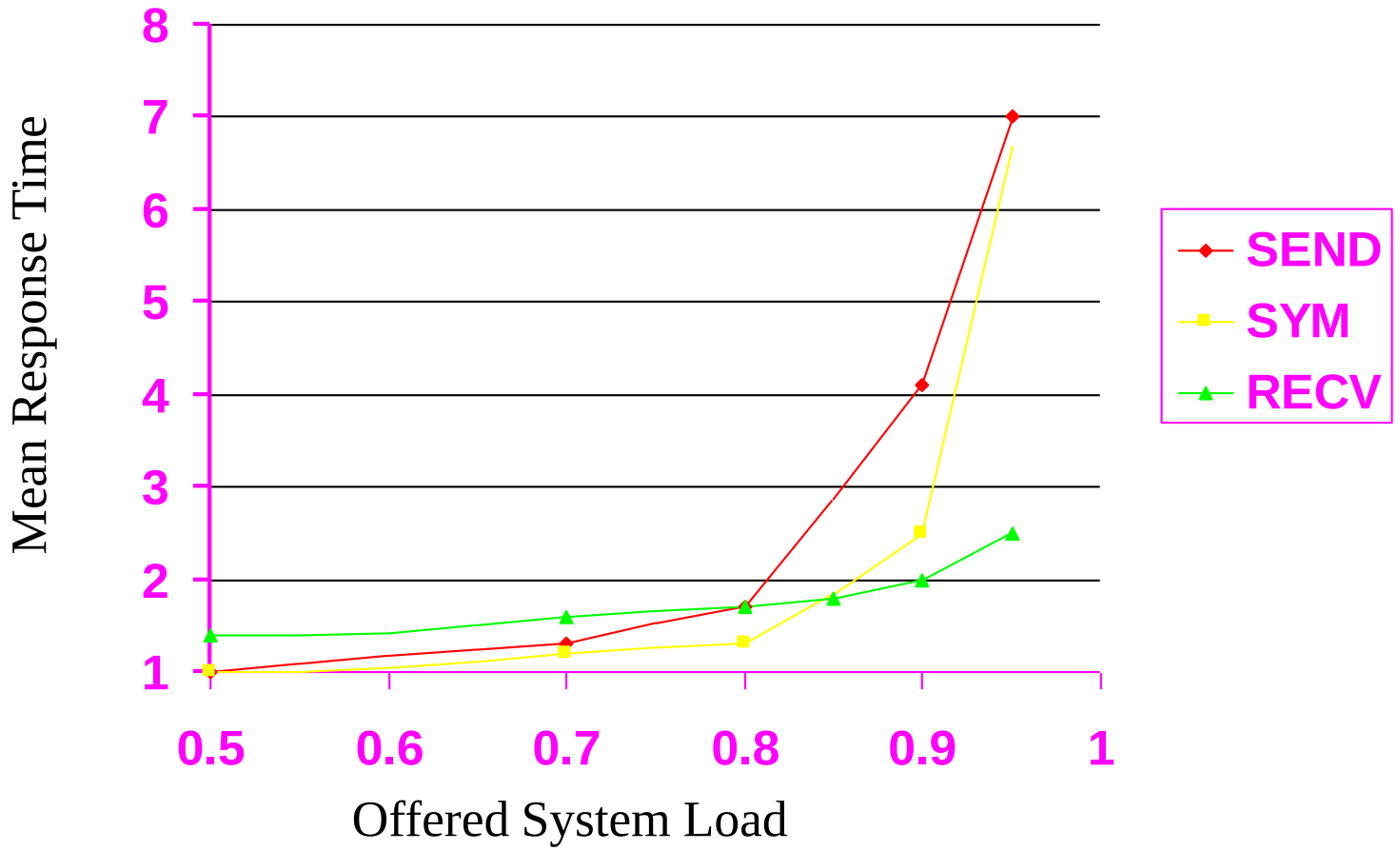
- Algorithms Studied are:

M/M/1	A distributed system that performs no load distributing
RECV	Receiver-initiated algorithm
RAND	Sender-initiated algorithm with random location policy
SEND	Sender-initiated algorithm with threshold policy
ADSEND	Stable sender-initiated algorithm
SYM	Symetrically initiated algorithm
ADSYM	Stable symemetrically initiated algorithm
M/M/K	A distributed system that performs ideal load distributing

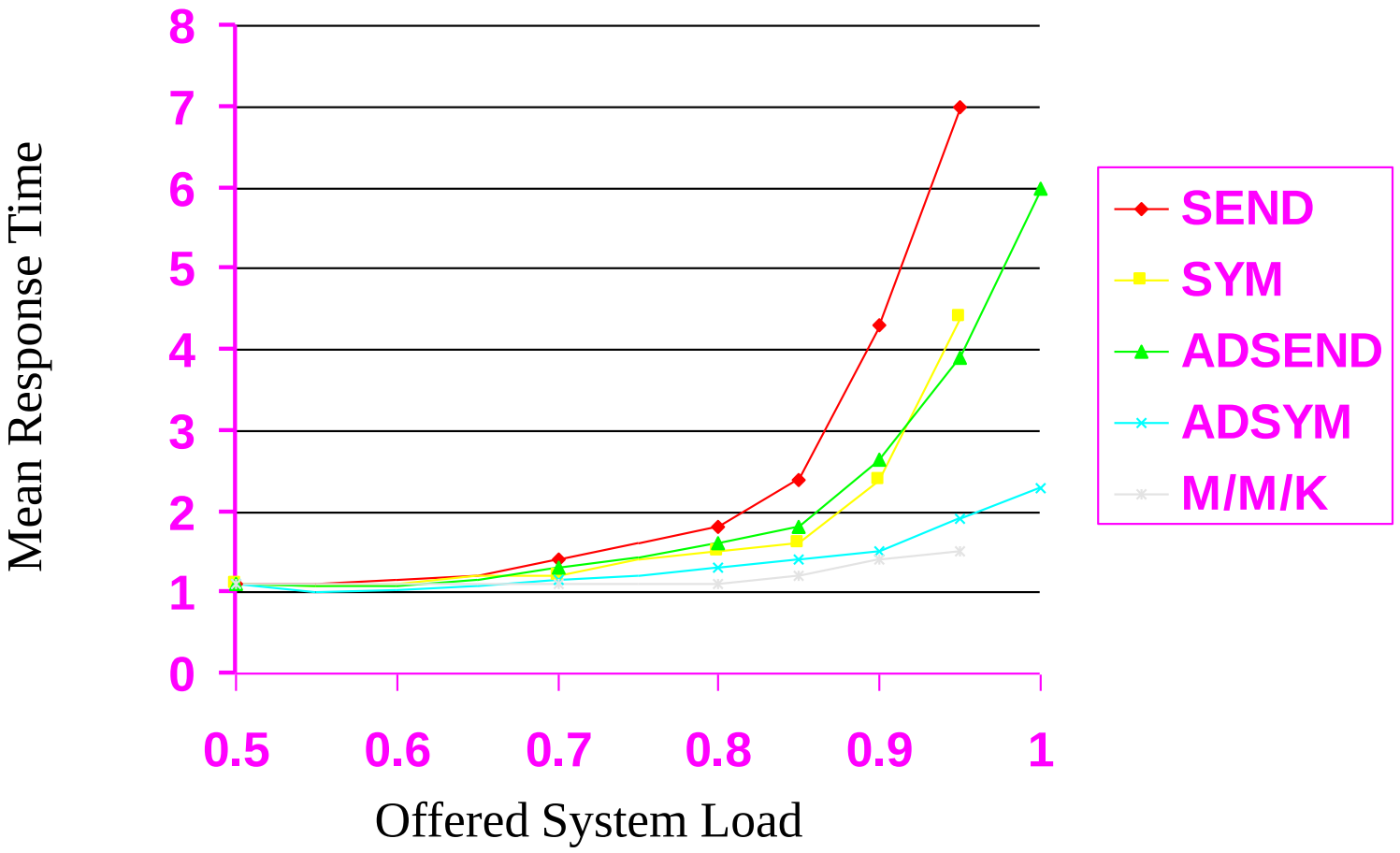
Receiver-initiated vs. Sender-initiated Load Balancing



Symmetrically Initiated Load Sharing



Stable Load Sharing Algorithms



Selecting A Suitable Load Sharing Algorithm

- If the system never attains high loads, sender-initiated algorithms give acceptable performance improvement
- If the system can reach high loads, use stable scheduling algorithms
- If the system experience a wide range of load fluctuations, the stable symmetrically initiated scheduling algorithm is recommended

Selecting A Suitable Load Sharing Algorithm

- For systems with wide range of load fluctuations and has a high cost for migration partially executed tasks, stable sender-initiated algorithms are recommended
- For systems that experience heterogeneous work arrival, adaptive stable algorithms are recommended