How to deal with uncertainties and dynamicity ?

http://graal.ens-lyon.fr/~lmarchal/scheduling/

October 19, 2011.

1 The problem

The problem : the world is not perfect !

- Uncertainties
  - On the platforms' characteristics
    (Processor power, link bandwidth, etc.)
  - On the applications' characteristics
    (Volume computation to be performed, volume of messages to be sent, etc.)
- Dynamicity
  - Of network (interferences with other applications, etc.)
  - Of processors (interferences with other users, other processors of the same node, other core of the same processor, hardware failure, etc.)
  - Of applications (on which detail should the simulation focus ?)

Solutions : to prevent or to cure ?

To prevent
- Algorithms tolerant to uncertainties and dynamicity.

To cure
- Algorithms auto-adapting to actual conditions.

Leitmotiv : the more the information, the more precise we can statically define the solutions, the better our chances to "succeed"

2 Analyzing sensitivity and robustness

2.1 Analyzing the sensitivity

Question : we have defined a solution, how is it going to behave "in practice" ?

Possible approach

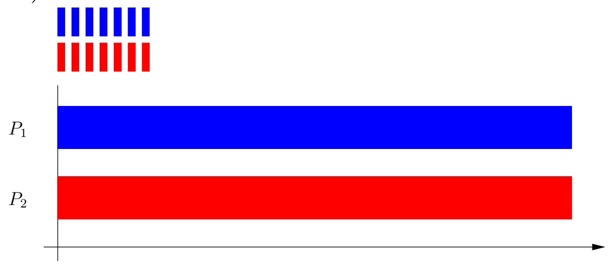
1. Definition of an algorithm $\mathcal{A}$.

2. Modeling the uncertainties and the dynamicity.
3. Analyzing the sensitivity of $\mathcal{A}$ as follows :
   - For each theoretical instance of the problem
     - Evaluate the solution found by $\mathcal{A}$
     - For each “actual”instance corresponding to the given theoretical instance, find the optimal solution and the relative performance of the solution found by $\mathcal{A}$.

   Sensitivity of $\mathcal{A}$: worst relative performance, or (weighted) average relative performance, etc.

Example

Problem
- Master-slave platform with two identical processors
- Flow of two types of identical tasks
- Objective function : maximum minimum throughput between the two applications (*max-min fairness*)



A possible solution... null if processor P_2 fails.

The case of Backfilling

Context :
- cluster shared between many users
- need for an allocation policy, and a reservation policy
- job request : number of processors + maximal utilization time
- (A job exceeding its estimate is automatically killed)

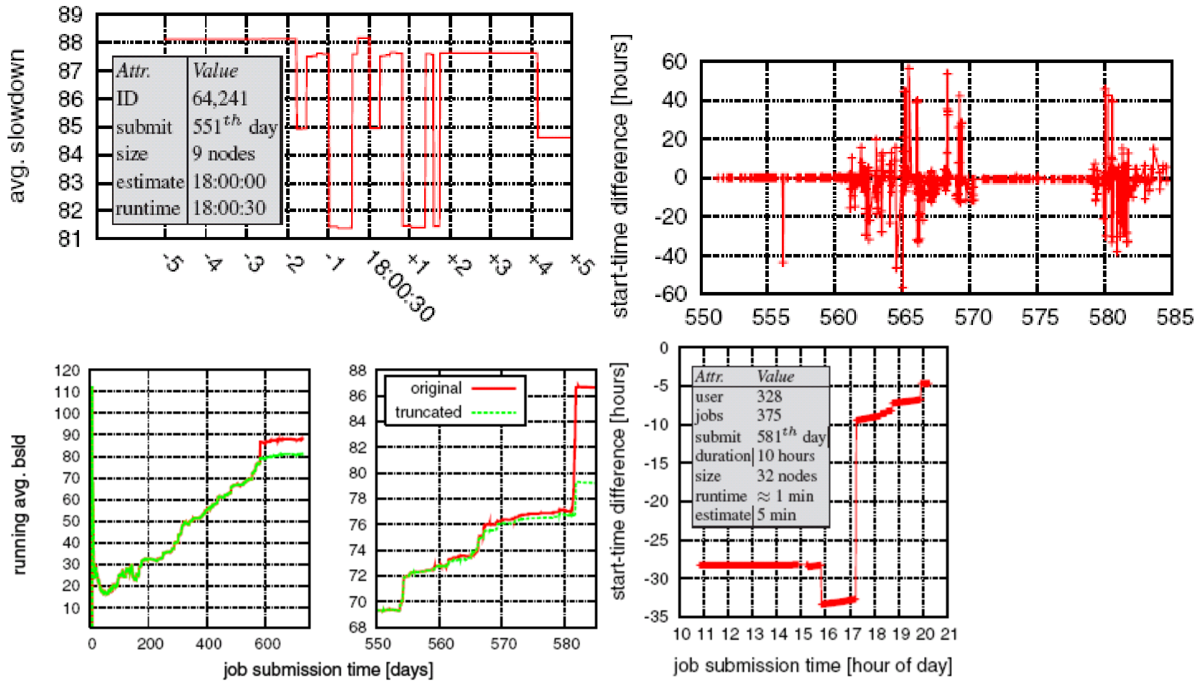
Simplistic policy : First Come First Served :
- lead to waste some resources

The EASY backfilling scheme
- The jobs are considered in First-Come First-Served order
- Each time a job arrives or a job completes, a reservation is made for the first job that cannot be immediately started, later jobs that can be started immediately are started.
- In practice jobs are submitted with runtime estimates.
  A job exceeding its estimate is automatically killed.

The set-up
- 128-node IBM SP2 (San Diego Supercomputer Center)
- Log from May 1998 to April 2000 log : 67,667 jobs *Parallel Workload Archive* (www.cs.huji.ac.il/labs/parallel/workload/)

- Job runtime limit : 18 hours.
  (Some dozens of seconds may be needed to kill a job.)
- Performance measure : average slowdown (=average stretch). Bounded slowdown :
  $\max\left(1, \dfrac{T_w + T_r}{\max(10, T_r)}\right)$

Execution is simulated based on the trace : enable to change task duration (or scheduling policy).

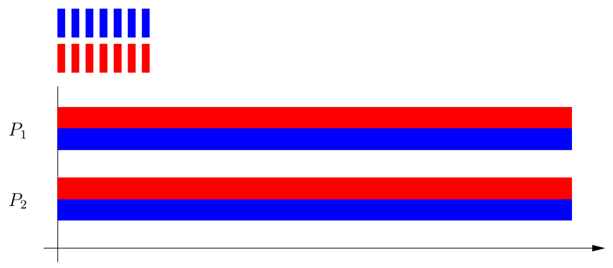


The length of a job running for 18 hours and 30 seconds is shorten by 30 seconds.

Complete study in the article “Instability in Parallel Job Scheduling Simulation : The Role of Workload Flurries” by Dan Tsafrir Dror G. Feitelson, IPDPS 2006, available at http://graal.ens-lyon.fr/~lmarchal/scheduling/feitelson-flurries-backfilling.pdf.

2.2 Robust solutions, an example of resolution

An algorithm is said to be robust if its solutions stay close to the optimal when the actual parameters are slightly different from the theoretical parameters.



This solution stays optimal whatever the variations in the processors’ performance : it is not sensitive to this parameter !

The problem

- A master which has an output bandwidth B.
- p slaves, slave P_i being link with a bandwidth b_i and having a computational speed of c_i.
- n flows of tasks, each flow being a set of identical tasks.
  A task from the flow k needs β_k units of communications and γ_k units of computation.
- $\rho_i^{(k)}$: throughput of application k on processor i.
  $\rho^{(k)} = \sum_i \rho_i^{(k)}$ is the throughput of application k on the whole platform.
- Objective : maximize $\min_k \rho^{(k)}$.

Dynamicity : a processor may fail (definitively).

Classical solution

1. Resource constraint : processors' capacities
$$\forall i, \qquad \sum_k \rho_i^{(k)} \gamma_k \leq c_i$$
2. Resource constraint : processors' bandwidths
$$\forall i, \qquad \sum_k \rho_i^{(k)} \beta_k \leq b_i$$
3. Resource constraint : the master output bandwidth
$$\sum_{i,k} \rho_i^{(k)} \beta_k \leq B$$
4. Objective
$$\text{Maximize} \min_k \sum_i \rho_i^{(k)}$$

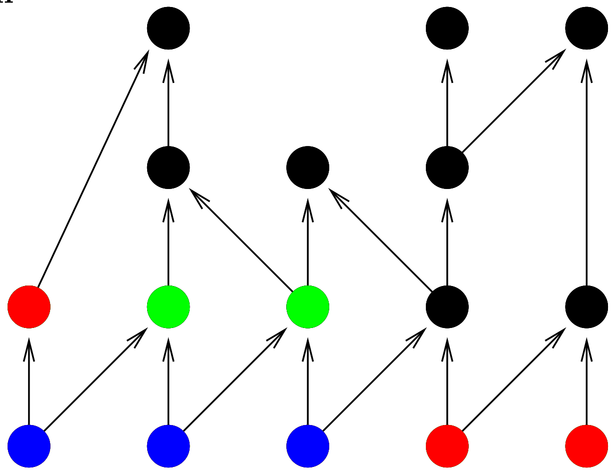
Robust solution

1. $\forall i, \qquad \sum_k \rho_i^{(k)} \gamma_k \leq c_i$
2. $\forall i, \qquad \sum_k \rho_i^{(k)} \beta_k \leq b_i$
3. $\sum_{i,k} \rho_i^{(k)} \beta_k \leq B$
4. Objective if processor P_p fails :
$$\rho_{\bar{p}} = \min_k \sum_{i \neq p} \rho_i^{(k)}$$
5. Objective : $\text{Maximize} \min \left\{ \min_p \frac{\rho_{\bar{p}}}{\rho_{\bar{p}}^{(\text{opt})}}, \min_k \sum_i \frac{\rho_i^{(k)}}{\rho^{(\text{opt})}} \right\}$

2.3 Internet-Based Computing

Giving priority to robustness : an extreme solution

Problem motivation



A possible schedule
(enabled, in process, completed)

Framework
– Internet-Based computing : processor capacities are unknown, running times are thus unpredictable.
– Execution of a task graph (aka workflow) containing n tasks.

The principle
– Motivation : lessening the likelihood of the “gridlock” that can arise when a computation stalls pending computation of already allocated tasks.
– IC-optimal schedule : after t tasks have been executed, the number of eligible (=executable) tasks is maximal, for any $t \in [1, n]$

More details in the article “Toward a Theory for Scheduling Dags in Internet-Based Computing” by Grzegorz Malewicz, Arnold L. Rosenberg, and Matthew Yurkewych in IEEE TC, available at http://graal.ens-lyon.fr/~lmarchal/scheduling/internet-computing.pdf.

3 Dynamic load-balancing

To cure (and no longer preventing) : the algorithm balance the load to take into account uncertainties and dynamicity.

General scheme
– From time to time, do :
(Each invocation has a cost : the invocations should only take place at “useful” instants)

- Compute a good solution using the observed parameters.
  (How do we predict the future from the past ?)
- Evaluate the cost of balancing the load
- If the gain is larger than the cost : load-balance
  (If the objective is to minimize the running time, the comparison is obvious. How do we compare a time and some QoS ?)

Network Weather Service

Distributed system which periodically monitors/records network and processor performance, and allows to predict the future performance of the network and of the processors http://nws.cs.ucsb.edu/.

Does the past enable to predict the future ?

4 How useful is old information ?

From the article "How Useful Is Old Information ?" by Michael Mitzenmacher in IEEE TPDS, available at http://graal.ens-lyon.fr/~lmarchal/scheduling/Mitzenmacher.pdf.

The problem
- The values used when taking decisions have already "aged".
- Is it a problem ? Should we take this ageing into account ?

Framework : platform and information
- A set of n servers.
- Tasks arrive according to a Poisson law of throughput λn, $\lambda < 1$ (the exponential law $f(x) = \lambda e^{-\lambda x}$ is the probability density function of the inter-arrival times)
- Task execution time : exponential law of mean 1.
- Each server executes in FIFO order the tasks it receives.
- We look at the time each task spent in the system (= flow).

There is a *bulletin board* on which are displayed the loads of the different processors.

This information may be wrong or approximate.

We only deal with the case in which this information is *old*.

This is the only information available to the tasks : they cannot communicate between each other and have some coordinated behavior.

Obvious strategies

- Random and uniform choice of the server.
  - Low overhead, finite length of queues.

- Random and uniform choice of d servers, the task being sent on the least loaded of the d servers.
  - Better than random, practical in distributed settings (poll a small number of processors)

- Task sent on the least loaded server.
  - Optimal in a variety of situations, need for centralization.

4.1 First model : periodic updates

- Each T units of time the bulletin board is updated with correct information.
- $P_{i,j}(t)$: fraction of queues with true load j but load i on the board, at time t
- $q_i(t)$ rate of arrivals at a queue with size i on the board at time t

System dynamics ($t \neq kT$, $j > 0$) :

$$\frac{dP_{i,j}(t)}{dt} = \underbrace{P_{i,j-1}(t) \times q_i(t)}_{\text{task arriving from } (i,j-1)} + \underbrace{P_{i,j+1}(t)}_{\text{task processed in } (i,j+1)} - \underbrace{P_{i,j}(t) \times q_i(t)}_{\text{task arriving from } (i,j)} - \underbrace{P_{i,j}(t)}_{\text{task processed in } (i,j)}$$

When $t = kT$, the system "jumps" to $P_{i,j}(t) = 0$ for all $ti \neq j$, and $P_{i,i}(t) = \sum_j P_{j,i}(t^-)$.

It is possible to prove that this equation describe the limiting behavior of the system (when $n \to \infty$).

Specific strategies

fractions of servers with (apparent) load i : $b_i(t) = \sum_j Pi, j(t)$

- choose the least loaded among d random servers

$$q_i(t) = \lambda \frac{\left(\sum_{j\geq i} b_j(t)\right)^d - \left(\sum_{j>i} b_j(t)\right)^d}{b_i(t)}$$

- choose the shortest queue (assume there is always a server with load 0)

$$\begin{aligned} q_0(t) &= \frac{\lambda}{b_0(t)} \\ q_i(t) &= 0 \qquad i \neq 0 \end{aligned}$$

Two possible resolutions

1. Theoretical :

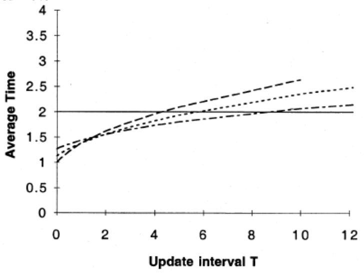
- fixed point when $\frac{dP_{i,j}(t)}{dt} = 0$? ? No, because of the jump when $t = kT$
- fixed cycle on $[kT, (k+1)T]$
- can be solved using waiting queue theory (close form, but complex)

2. Practice with the above differential system :
– simulations, on truncated version of the system (bounding i and j)

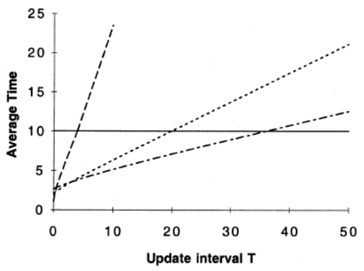
3. Practice without the differential system :
– simulate 100 queues
– can use every distribution you want

After using 2 and 3 : comparable results on the same set of parameters.

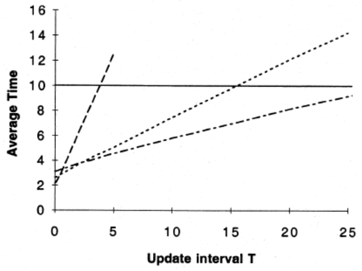
Results



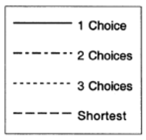
$n = 100$ and $\lambda = 0.5$



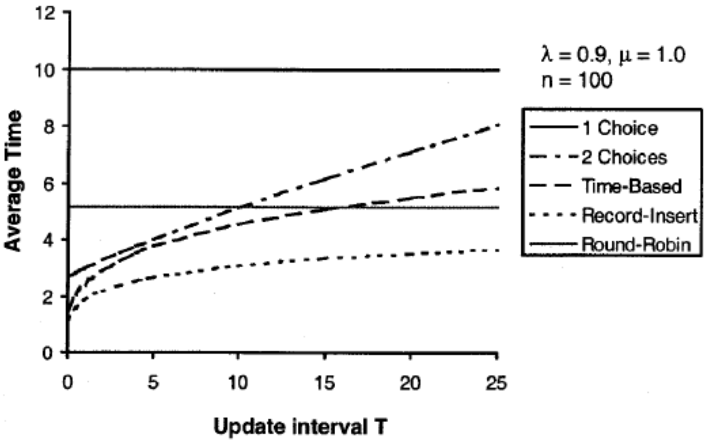
$n = 100$ and $\lambda = 0.9$



$n = 8$ and $\lambda = 0.9$



More elaborated strategies
– *Time-based* : random choice among the servers which are supposed to be the least loaded.
– *Record-Insert* : centralized service in which each task updates the bulletin board by indicating on which server it is sent.

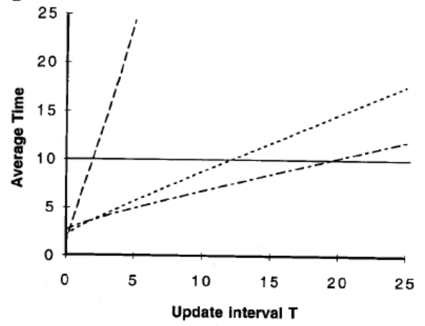


$n = 100$ and $\lambda = 0.9$

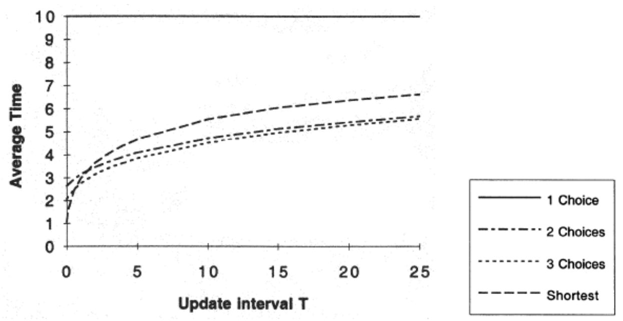
4.2 Other models : continuous & de-synchronized updates

Second model : continuous updates

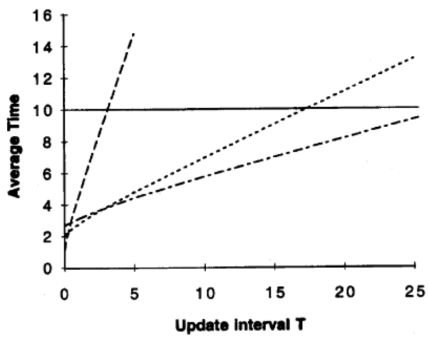
Model : continuous updates, but the information used is T units of time old.
$n = 100$ and $\lambda = 0.9$
Age of information :



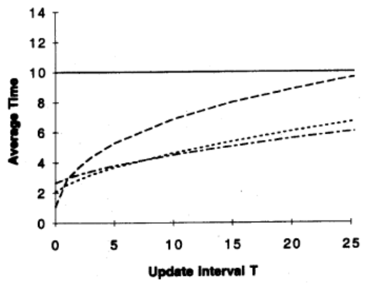
exactly T



exponential distribution, average T



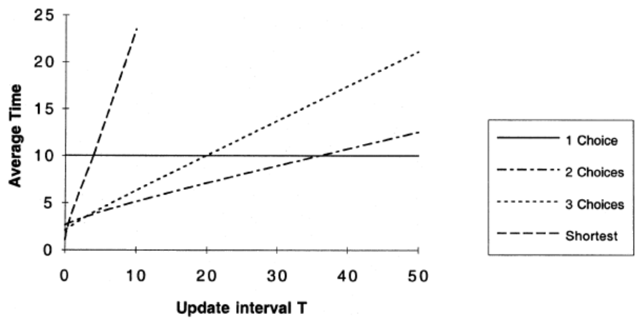
uniform distribution on $[\frac{T}{2}; \frac{3T}{2}]$



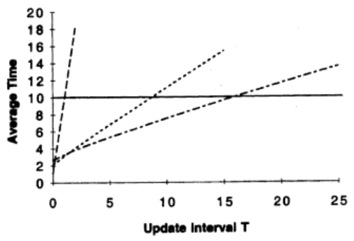
uniform distribution on $[0; 2T]$

Third model : de-synchronized updates

The different servers updates their information in a de-synchronized manner, each following an exponential law of average T.
$n = 100$ and $\lambda = 0.9$



Regular updates

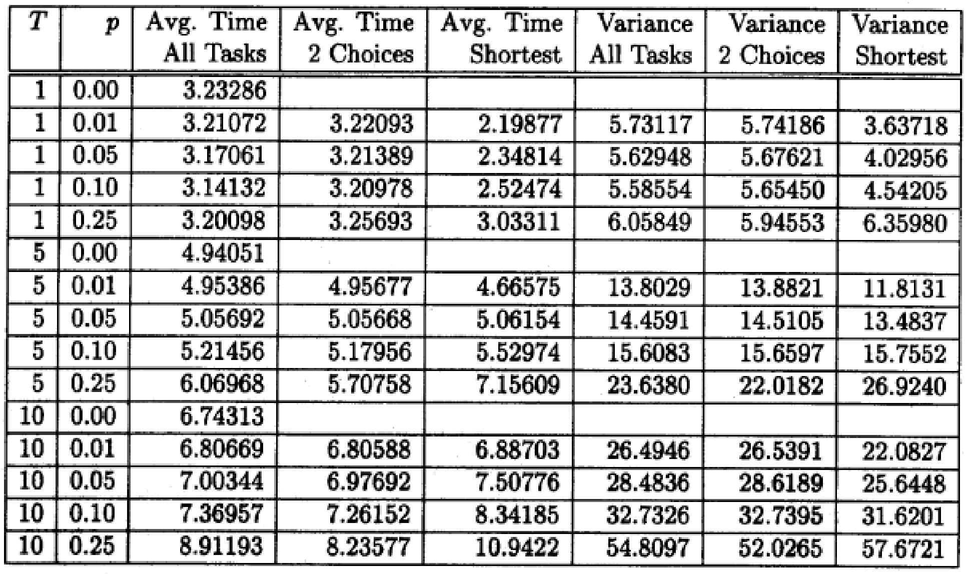


De-synchronized updates.

4.3 Changing the rules

And if some were cheating ?

With a probability p a task does not choose between two randomly determined servers, but takes the least loaded of all servers.



T	p	Avg. Time All Tasks	Avg. Time 2 Choices	Avg. Time Shortest	Variance All Tasks	Variance 2 Choices	Variance Shortest
1	0.00	3.23286					
1	0.01	3.21072	3.22093	2.19877	5.73117	5.74186	3.63718
1	0.05	3.17061	3.21389	2.34814	5.62948	5.67621	4.02956
1	0.10	3.14132	3.20978	2.52474	5.58554	5.65450	4.54205
1	0.25	3.20098	3.25693	3.03311	6.05849	5.94553	6.35980
5	0.00	4.94051					
5	0.01	4.95386	4.95677	4.66575	13.8029	13.8821	11.8131
5	0.05	5.05692	5.05668	5.06154	14.4591	14.5105	13.4837
5	0.10	5.21456	5.17956	5.52974	15.6083	15.6597	15.7552
5	0.25	6.06968	5.70758	7.15609	23.6380	22.0182	26.9240
10	0.00	6.74313					
10	0.01	6.80669	6.80588	6.88703	26.4946	26.5391	22.0827
10	0.05	7.00344	6.97692	7.50776	28.4836	28.6189	25.6448
10	0.10	7.36957	7.26152	8.34185	32.7326	32.7395	31.6201
10	0.25	8.91193	8.23577	10.9422	54.8097	52.0265	57.6721

Some memory always help

– Studied scenario : a task is allocated to the “best” of two randomly determined servers.

– New scenario : a task is allocated to the “best” of two servers, one being randomly chosen and the other one being the least-loaded one —after the previous task was allocated— of the two processors considered by the previous task.

– The problem : the memorization requires some communications and centralization.

Complete vs. incomplete information

Complete information
– Requires some centralization (or total replication) ;
– Communications of the most remote elements to the “center” ;
– Obsolescence of the information.

Decentralized schedulers
– The local data are more up-to-date ;
– A local optimization does not always lead to a global optimization...

5 Conclusion

Conclusion

– An obvious need to be able to cope with the dynamicity and the uncertainties.
– Crucial need to be able to model the dynamicity and the uncertainty.
– The static world is already complex enough !
– Where is the trade-off between the precision of the models and their usability ?
– Trade-off between static and dynamic approaches ?