Dynamic Fractional Resource Scheduling for HPC Workloads

Mark Stillwell[1] Frédéric Vivien[2,1] Henri Casanova[1]

[1]Department of Information and Computer Sciences
University of Hawai'i at Mānoa

[2]INRIA, France

Invited Talk, October 8, 2009

HPC Job Scheduling Problem

- $0 < N$ homogeneous nodes
- $0 < J$ jobs, each job j has:
  - arrival time $0 \leq r_j$
  - $0 < t_j \leq N$ tasks
  - compute time $0 < c_j$
- J not known
- r_j and t_j not known before r_j
- c_j not known until j completes

Schedule Evaluation

- make span not relevant for unrelated jobs
- flow time over-emphasizes very long jobs
- stretch re-balances in favor of short jobs
- average stretch prone to starvation
- max stretch helps with average while bounding worst case

Current Approaches

- Batch Scheduling, which no one likes
  - usually FCFS with backfilling
  - backfilling needs (unreliable) compute time estimates
  - unbounded wait times
  - poor resource utilization
  - No particular objective
- Gang Scheduling, which no one uses
  - globally coordinated time sharing
  - complicated and slow
  - memory pressure a concern

VM Technology

- basically, time sharing
- pooling of discrete resources (e.g., multiple CPUs)
- hard limits on resource consumption
- job preemption and task migration

Problem Formulation

- extends basic HPC problem
- jobs now have per-task CPU need α_j and memory requirement m_j
- multiple tasks can run on one node if total memory requirement $\leq 100\%$
- job tasks must be assigned equal amounts of CPU resource
- assigning less than the need results in proportional slowdown
- assigned allocations can change
- no run-time estimates
- so we need another metric to optimize

Yield

Definition

The *yield*, $y_j(t)$ of job j at time t is the ratio of the CPU allocation given to the job to the job's CPU need.

- requires no knowledge of flow or compute times
- can be optimized for at each scheduling event
- maximizing minimum yield related to minimizing maximum stretch
- How do we keep track of job progress when the yield can vary?

Virtual Time

Definition

The virtual time $v_j(t)$ of job j at time t is the subjective time experienced by the job.

- $v_j(t) = \int_{r_j}^{t} y_j(\tau) d\tau$
- job completes when $v_j(t) = c_j$

The Need for Preemption

- final goal is to minimize maximum stretch
- without preemption, stretch of non-clairvoyant on-line algorithms unbounded
  - consider 2 jobs
  - both require all of the system resources
  - one has $c_j = 1$
  - other has $c_j = \Delta$
- need criteria to decide which jobs should be preempted

Priority

Jobs should be preempted in order by increasing priority.

- newly arrived jobs may have infinite priority
- $1/v_j(t)$ performs well, but subject to starvation
- $(t - r_j)/v_j(t)$ time avoids starvation, but does not perform well
- $(t - r_j)/(v_j(t))^2$ seems a reasonable compromise
- other possibilities exist

Greedy Scheduling Heuristics

- **GREEDY–** Put tasks on the host with the lowest CPU demand on which it can fit into memory; new jobs may have to be resubmitted using bounded exponential backoff.
- **GREEDY-PMTN–** Like GREEDY, but older tasks may be preempted
- **GREEDY-PMTN-MIGR–** Like GREEDY-PMTN, but older tasks may be migrated as well as preempted

Connection to multi-capacity bin packing

For each discrete scheduling event:

- problem *similar to* multi-capacity (vector) bin packing, but has optimization target and variable CPU allocations
- can formulate as an MILP [Stillwell et al., 2009] (NP-complete)
- relaxed LP heuristics slow, give low quality solutions

Applying MCB heuristics

- yield is continuous, so choose a granularity (0.01)
- perform a binary search on yield, seeking to maximize
- for each fixed yield, set CPU requirement and apply heuristic
- found yield is the maximized minimum, leftover CPU used to improve average
- if a solution cannot be found at any yield, remove the lowest priority job and try again

MCB8 Heuristic

Based on [Leinberger et al., 1999], simplified to 2-dimensional case:

1. Put job tasks in two lists: CPU-intensive and memory-intensive
2. Sort lists by "some criterion". (MCB8: descending order by maximum)
3. Starting with the first host, pick tasks that fit in order from the list that goes against the current imbalance. Example:
   - current host tasks total 50% CPU and 60% memory
   - Assign the next task that fits from the list of CPU-intensive jobs.
4. When no tasks can fit on a host, go to the next host.
5. If all tasks can be placed, then success, otherwise failure.

MCB8 Heuristic

Based on [Leinberger et al., 1999], simplified to 2-dimensional case:

1. Put job tasks in two lists: CPU-intensive and memory-intensive
2. Sort lists by “some criterion”. (MCB8: descending order by maximum)
3. Starting with the first host, pick tasks that fit in order from the list that goes against the current imbalance. Example:
   - current host tasks total 50% CPU and 60% memory
   - Assign the next task that fits from the list of CPU-intensive jobs.
4. When no tasks can fit on a host, go to the next host.
5. If all tasks can be placed, then success, otherwise failure.

MCB8 Heuristic

Based on [Leinberger et al., 1999], simplified to 2-dimensional case:

1. Put job tasks in two lists: CPU-intensive and memory-intensive
2. Sort lists by "some criterion". (MCB8: descending order by maximum)
3. Starting with the first host, pick tasks that fit in order from the list that goes against the current imbalance. Example:
   - current host tasks total 50% CPU and 60% memory
   - Assign the next task that fits from the list of CPU-intensive jobs.
4. When no tasks can fit on a host, go to the next host.
5. If all tasks can be placed, then success, otherwise failure.

MCB8 Heuristic

Based on [Leinberger et al., 1999], simplified to 2-dimensional case:

1. Put job tasks in two lists: CPU-intensive and memory-intensive
2. Sort lists by "some criterion". (MCB8: descending order by maximum)
3. Starting with the first host, pick tasks that fit in order from the list that goes against the current imbalance. Example:
   - current host tasks total 50% CPU and 60% memory
   - Assign the next task that fits from the list of CPU-intensive jobs.
4. When no tasks can fit on a host, go to the next host.
5. If all tasks can be placed, then success, otherwise failure.

MCB8 Heuristic

Based on [Leinberger et al., 1999], simplified to 2-dimensional case:

1. Put job tasks in two lists: CPU-intensive and memory-intensive
2. Sort lists by "some criterion". (MCB8: descending order by maximum)
3. Starting with the first host, pick tasks that fit in order from the list that goes against the current imbalance. Example:
   - current host tasks total 50% CPU and 60% memory
   - Assign the next task that fits from the list of CPU-intensive jobs.
4. When no tasks can fit on a host, go to the next host.
5. If all tasks can be placed, then success, otherwise failure.

MCB8 Heuristic

Based on [Leinberger et al., 1999], simplified to 2-dimensional case:

1. Put job tasks in two lists: CPU-intensive and memory-intensive
2. Sort lists by "some criterion". (MCB8: descending order by maximum)
3. Starting with the first host, pick tasks that fit in order from the list that goes against the current imbalance. Example:
   - current host tasks total 50% CPU and 60% memory
   - Assign the next task that fits from the list of CPU-intensive jobs.
4. When no tasks can fit on a host, go to the next host.
5. If all tasks can be placed, then success, otherwise failure.

MCB8 Heuristic

Based on [Leinberger et al., 1999], simplified to 2-dimensional case:

1. Put job tasks in two lists: CPU-intensive and memory-intensive
2. Sort lists by "some criterion". (MCB8: descending order by maximum)
3. Starting with the first host, pick tasks that fit in order from the list that goes against the current imbalance. Example:
   - current host tasks total 50% CPU and 60% memory
   - Assign the next task that fits from the list of CPU-intensive jobs.
4. When no tasks can fit on a host, go to the next host.
5. If all tasks can be placed, then success, otherwise failure.

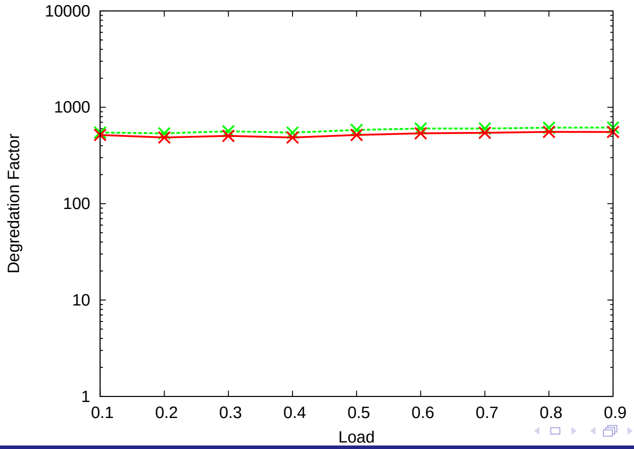
MCB8 Scheduling Heuristics

- **DYNMCB8–** Apply heuristic on every event
- **DYNMCB8-PER–** Apply heuristic periodically
- **DYNMCB8-ASAP-PER–** like DYNMCB8-PER, but try to greedily schedule incoming jobs
- **DYNMCB8-STRETCH-PER–** like DYNMCB8-PER, but try to optimize worst-case max stretch

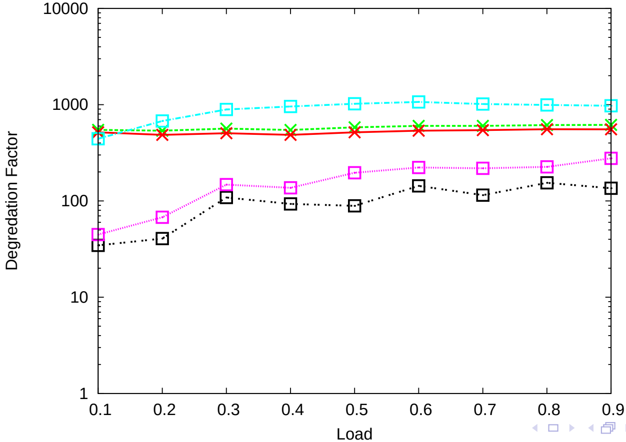
Methodology

- discrete event simulator takes list of jobs and returns stretch values
- workloads based on synthetic and real traces
- synthetic workload arrival times scaled to show performance on different load conditions
- algorithms evaluated by per-trace *degradation factor*
- experiment with “free” preemption/migration and experiment where preemption/migration costs job a constant amount of wall clock time.

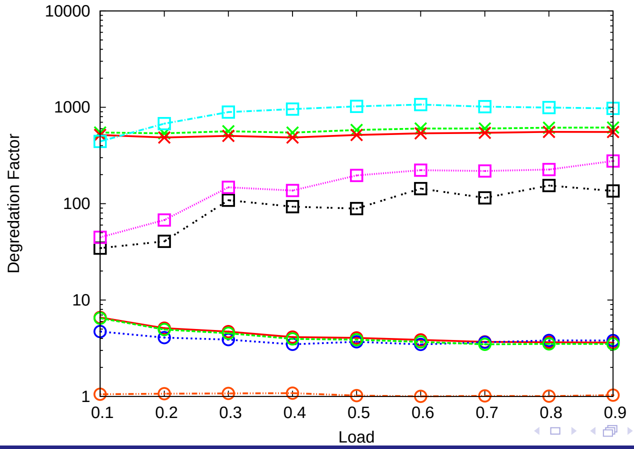
Average Maximum Yield, No preemption/migration penalty



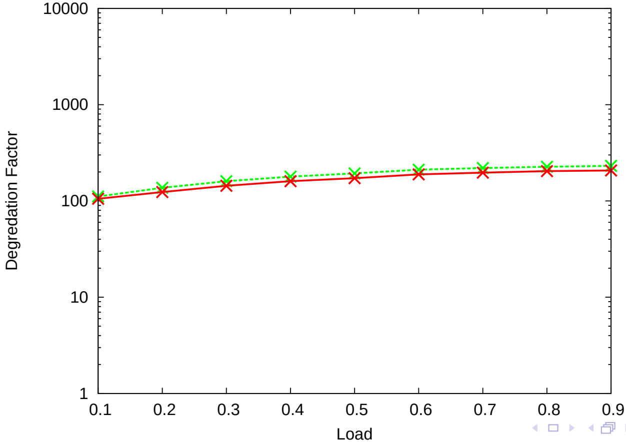
Average Maximum Yield, No preemption/migration penalty



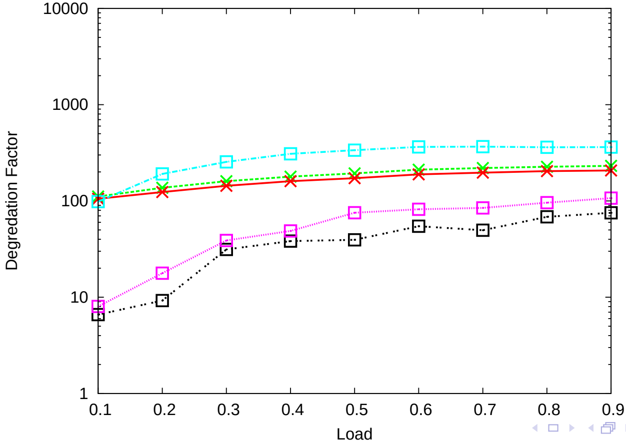
Average Maximum Yield, No preemption/migration penalty



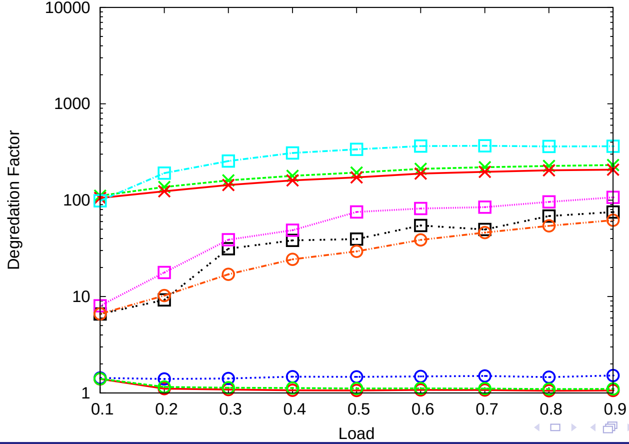
Average Maximum Yield, 5 minute preemption/migration penalty



Average Maximum Yield, 5 minute preemption/migration penalty



Average Maximum Yield, 5 minute preemption/migration penalty



Comparison of Synthetic vs. Real workload results

Algs	Scaled synth. Deg. factor		Unscaled synth. Deg. factor		Real-world Deg. factor	
	avg.	max	avg.	max	avg.	max
EASY	167	560	139	443	94	1476
FCFS	186	569	154	476	118	2219
greedy	294	1093	249	1050	153	1527
greedyp	41	875	35	785	9	147
greedypm	62	835	37	773	17	759
mcb	32	162	11	162	11	231
mcbp	1	12	2	21	3	20
gmcbp	1	9	2	22	2	20
mcbsp	1	12	2	21	3	23

Computation Times

- Most scheduling events involve 10 or fewer jobs and require negligible time for all schedulers.
- Even when there are about 100 jobs, the time for MCB8 is under 5 seconds on a 3.2Ghz machine

Costs

- Greedy approaches use significantly less bandwidth than MCB approaches (<1GB/s in the worst case)
- MCB approaches cause jobs to be preempted around 5 times on average.
- DYNMCB8 uses 1.3GB/s on average, 5.1GB/s maximum
- periodic algorithms 0.6GB/s on average, 2.1GB/s maximum

Conclusions

- DFRS potentially much better than batch scheduling
- multi-capacity bin packing heuristics perform best
- targeting yield does as well as targeting worst case max stretch
- periodic MCB approaches perform nearly as well as aggressive ones when there is no migration cost and much better when there is a fixed migration cost
- adding an opportunistic greedy scheduling heuristic to DYNMCB8-PER gives no real benefit to max stretch
- MCB approaches can calculate resource allocations reasonably quickly
- MCB approaches need to try to mitigate migration/preemptions costs

References I

Leinberger, W., Karypis, G., and Kumar, V. (1999).
Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints.
In *Proc. of the Intl. Conf. on Parallel Processing*, pages 404–412. IEEE.

Stillwell, M., Shanzenbach, D., Vivien, F., and Casanova, H. (2009).
Resource Allocation using Virtual Clusters.
In *Proc. of CCGrid 2009*, pages 260–267. IEEE.