Term report – CSS 600

Introduction

There are many scheduling algorithms out there developed for a distributed system or to get more fancy "cloud" or grid computing. Most of these algorithms are based on an advance knowledge or user-specified knowledge about the job duration.
The proposed scheduling algorithm assumes no information about a submitted job. It generates information about a job's run at runtime and makes decisions on how to best run this job to completion. For example, the preemption frequency, and migration frequency of a job are generated across job preemptions locally and job migrations across different computing nodes.

Related work

Condor

Condor allows any machine to simultaneously execute jobs and serve as a submission point. Every machine in the condor system can submit and run batch jobs including the central manager, which is the central scheduler for the whole system. Condor matches idle jobs with available machines by using ClassAds. ClassAds are advertised by both machines and jobs and is a flexible representation of the characteristics and constraints of machines and jobs in the Condor system.

Open PBS

OpenPBS uses a central scheduler that gets resource requests from a server (on the same machine or different machine), and then forwards the resource requests to the MOM (machine oriented mini-server) component. After a resource has been found and returned by MOM to the scheduler, the scheduler requests job information from the server, makes a policy decision to run, and sends a run request to the server. The server eventually sends the job to MOM to run.

Baseline schedulers

Condor and Open PBS would be used as the baseline schedulers for comparison with the proposed scheduling algorithm. This is because both Condor and Open PBS are made for batch jobs running in a distributed system. One drawback of these schedulers is that there is no support for interactive jobs like GUI applications like a text editor, or word processor.

Problem

Job scheduling in distributed systems like Condor migrate jobs based on finding the best computing nodes to execute batch jobs, however, the frequency migration of these jobs are not taken into consideration. This migration overhead is unaccounted for but adds up with disproportionately higher compute-bound batch jobs than others causing frequent migrations across the pool as the current computing node become "less fit" for execution.

Preliminary design of scheduler

Hypothesis

Given a set of jobs J, with considerably higher CPU-intensive jobs, scheduling the higher CPU-intensive jobs using their migration frequency across a pool of computing nodes would improve their overall runtime efficiency.

Proof by induction

A list of n jobs submitted to the head node of the proposed scheduler would all get a chance to execute and eventually complete their CPU-burst cycle without risk of starvation.
Solution: Let *P(n)* be the proposition that n jobs would complete their CPU-burst
BASIS STEP: *P(1)* is true, because a job submitted to the head node would get its chance to execute without preemption if there are no other jobs present. However, if this job exceeds the maximum preemption frequency, it would be migrated to the remote pool for scheduling by the migration frequency scheduler. At the worst case, if this job's migration count is greater than or equal to the maximum allowable for any job in the remote pool, it is migrated to the fastest node in the pool for execution. An assumption in this system is that a job that reaches the remote pool has the highest available resources and hence, is guaranteed to run to completion.
INDUCTIVE STEP: Prove for P(k), P(k+1), then P(n) – **in progress**

Algorithms

Preemption Frequency Scheduler, PFS

The PFS schedules jobs on a round robin order to be executed within a specified time window. A job is preempted if execution is not completed with the time quantum and this preemption frequency PF, is used to determine local execution versus sending the job to the remote execution pool.
Variables:
PF: Preemption frequency is the running count of the number of times a job has been preempted

Algorithm ***PFS*** (L, n)
Input: L (a list of jobs of size n)
Output: L (a list of completed interactive jobs)

```
Begin
    LocalQuantum := MAX_QUANTUM;
    LocalPF := MAX_PREEMPTION;
    if n = 0 then do nothing
    else
        for i := 0 to n do
            job := L[i];
            unpreempted_queue.addToBack( job );
        {check for jobs waiting in the unpreempted_queue}
        while unpreempted_queue.length > 0 do
            job := unpreempted_queue.removeFromFront();
            execute( job );
            while job.is_still_executing & ElapsedTime < LocalQuantum do
                if LocalQuantum >= job.RemainingExecutionTime then
                    job.PF = job.PF + 1;
                    preempt( job );
                    preempted_queue.addToBack( job );
                else if job.RemainingExecutionTime = 0 then
                    job.PF = 0;
        {check for jobs waiting in the preempted_queue}
        while preempted_queue.length > 0 do
            job := preempted_queue.removeFromFront();
            if job.PF >= LocalPF then
                remote_queue.addToBack( job );
            else if LocalQuantum * 1.1 >= job.RemainingExecutionTime then
                job.PF = job.PF + 1;
                preempt( job );
                preempted_queue.addToBack( job );
                check (unpreempted_queue);
            else if job.RemainingExecutionTime = 0 then
                job.PF = 0;
            check (unpreempted_queue);
        {check for jobs waiting in the remote_queue}
        while remote_queue.length > 0 do
            job := remote_queue.removeFromFront();
            if unpreempted_queue.length = 0 then
                if preempted_queue.length = 0 then
                    unpreempted_queue.addToBack( job );
            else if preempted_queue.length = 0 then
                preempted_queue.addToBack( job );
            else
                job.MF := job.MF + 1;
                preempt( job );
                mobile_agent := job
                checkpoint(mobile_agent);
                remote := findMostAvailableRemote();
                send(mobile_agent, remote);
            check (unpreempted_queue);
End
```

Algorithm ***findMostAvailableRemote****()*

Input:

Output: *N (most available computing node or least busy node)*

```
begin
    QuickSort_By_CPU_Availability(computing_nodes)
    return computing_nodes[0];
end
```

Migration Frequency Scheduler, MFS

The **MFS** schedules a job for execution if it has a higher migration count than other potential jobs in the queue of jobs; selecting a job at random breaks ties, and a job that exceeds the maximum migration frequency, MF, is sent to the fastest computing node and banned for further migrations.

Variables:

MF: Migration frequency is the running count of the number of times a job has been migrated

PF: Preemption frequency is the running count of the number of times a job has been preempted

K: $K_C + K_{C/R}$

K_C is the communication overhead; $K_{C/R}$ is a job's checkpoint/restart overhead

Algorithm ***MFS*** *(M, n, R)*

Input: *M (List of mobile agents M of size n, to be assigned to nodes R)*

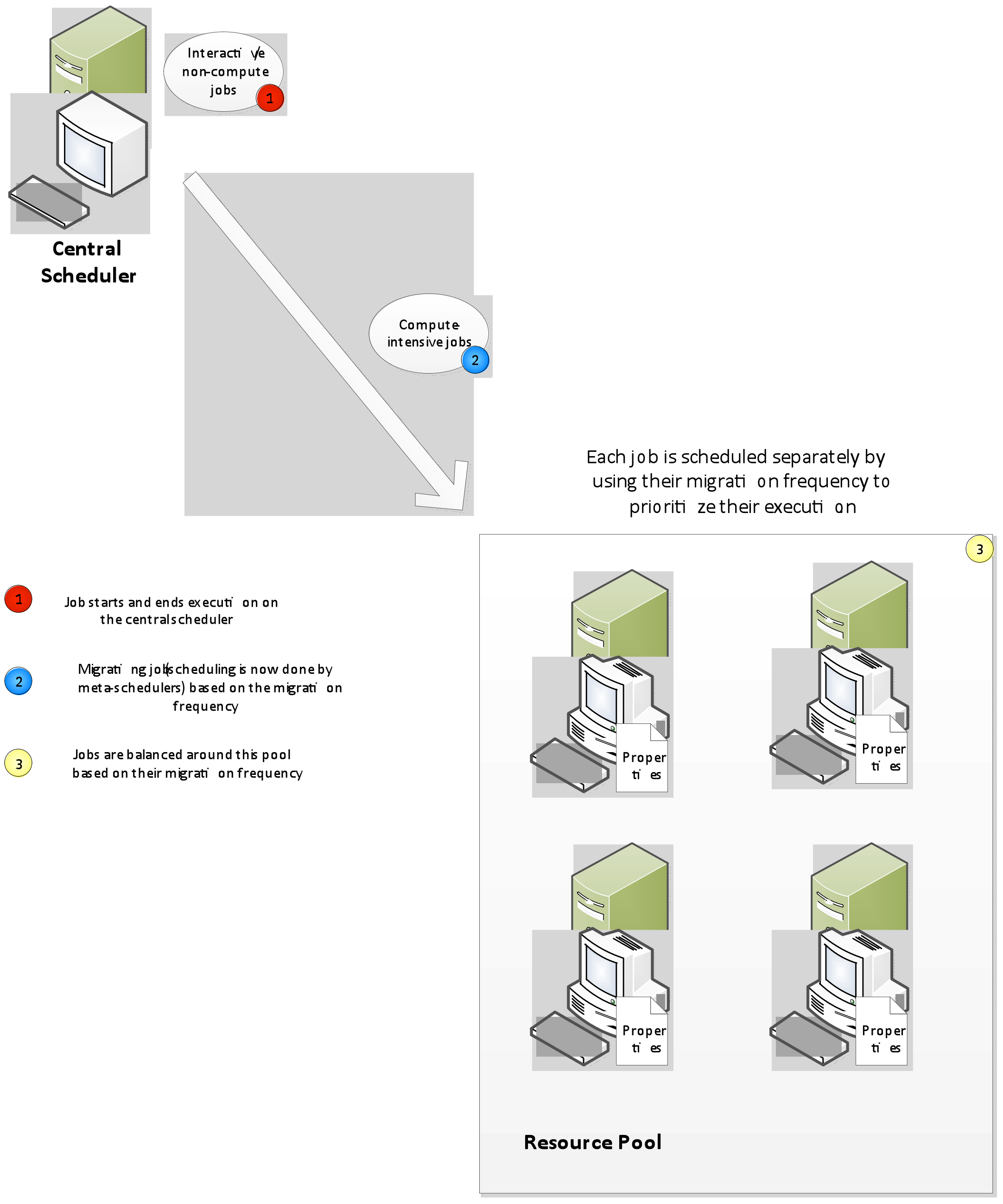
Output: *Completed status of mobile agent jobs*

```
begin
    available_nodes := R;
    assigned_nodes := 0;
    QuickSort_By_CPU_Availability( available_nodes );

    for i := 0 to n do
        if available_nodes.size > 0 then
            node := available_nodes.removeFromFront();
            job := extract_job( M[i] );
            assigned_nodes.addToBack( node );
            if current_node != node then
                if current_node.CPU_Avail < node.CPU_Avail then
                    {attempt to send for another remote node}
                    if job.MF <= localMF then
                        PFS(job, 1);
                    else
                        job.MF := job.MF + 1;
                        send(M[i], node);
                else
                    PFS(job, 1);
            else
                PFS(job, 1);
end

Algorithm send(mobile_agent, node)
Input: mobile_agent
Output: Status of send
begin
    QuickSort_By_CPU_Availability( total_nodes );
    if job.MF > MAX_MF then
        job.MF := job.MF + 1;
        send_to_fastest_node(mobile_agent);
    else
        execute_at(mobile_agent, node);
end
```

Mathematical comparison with related schedulers

TBD

Solution


Interactive non-compute jobs
1
Central Scheduler
Compute intensive jobs
2
Each job is scheduled separately by using their migration frequency to prioritize their execution
3
1
Job starts and ends execution on the central scheduler
2
Migrating jobs (scheduling is now done by meta-schedulers) based on the migration frequency
3
Jobs are balanced around this pool based on their migration frequency
Properties
Properties
Properties
Properties
Resource Pool

Applications

Some applications of this solution is dependent on the fact that

1. General-purpose commodity machines can be used to execute batch jobs, as well interactive application programs like: word processors, DVD/music player, and streaming a YouTube video online.
2. A stream of compute-intensive scientific applications can benefit from the computational capacity of this scheduling system.
3. Image processing applications would also benefit from this system. The image processing job can start out as an interactive application with minimal CPU-

usage and eventually morph into more CPU-intensive computation which has the potential of remotely being executed.

Table comparing related schedulers

Feature	**Scheduler**	**MF Scheduler**
Batch jobs	Condor, Open PBS	Supported
Interactive jobs		Supported
Migration-frequency based		Supported
Central resource manager	Condor, Open PBS	N/A
Multiple resource managers		Supported – each node has pool resource information

Conclusion

The proposed algorithm is focused on establishing a correlation between the migration frequency of CPU-bound (batch) jobs and the average turnaround time of the total jobs in the system. It is important to note that the interactive (low CPU-bursts) jobs are relatively unaffected by the migration frequency based scheduling because they do not participate in migration, however, the preemption based scheduling would also impact average turnaround time of the jobs in the system. This is because two separate schedulers – the preemption frequency scheduler, and the migration frequency scheduler would schedule CPU-bound jobs; this extra overhead would be investigated and analyzed.