1. Problem

There are N jobs in the list. Each jobs requires certain CPU utilization and takes certain amount of time to complete (job’s length). There are M workers that are physically the same (e.g. same memory, same CPU speed, etc). We are assuming the following for the time being:

* No more new job is coming in.
* Any job once started always complete with no failures (failure handling is not required).
* Jobs do not have deadline, maximum waiting time, or priority. A job can be scheduled to run at any time.
* Any worker always has enough resources to run any number of jobs as long as their total CPU utilization does not exceed 100%. In other words, CPU utilization is the only constraint in this problem. There is no constraint over number of cores, total amount of memory, context switching cost, etc.

Our goal is to design a scheduling algorithm that can maximize the average utilization. Although this problem can be reduced to the KNAPSACK problem for which a polynomial-time solution is not possible, we will try a few heuristic approaches and see how they perform in various input sets.

1. Approaches

The simplest approach is to try assigning jobs to workers whenever possible. We do not hope this approach to perform well since it does nothing to optimize the average utilization.

|  |
| --- |
| Repeat:  For each job I that has not been scheduled:  For each worker J that job I can fit in:  Assign job I to worker J;  Wait 1 second;  Until all jobs has been scheduled; |

Figure : Approach #1 – Simplest approach

Heavy Jobs First

As stated, Approach #1 does nothing for optimization. It is not difficult to come up with a test case for which this approach performs poorly e.g. Figure 2.

2 workers

|  |  |  |
| --- | --- | --- |
| Job Id | CPU | Length |
| 1 | 10% | 10 sec |
| 2 | 10% | 10 sec |
| 3 | 10% | 10 sec |
| 4 | 80% | 10 sec |
| 5 | 80% | 10 sec |

|  |  |  |
| --- | --- | --- |
| Approach #1 | Worker 1 | Worker 2 |
| First 10 sec | Job 1, Job 2, Job 3 | Job 4 |
| Next 10 sec | Job 5 |  |

Approach #1’s Average Utilization: 0.475

|  |  |  |
| --- | --- | --- |
| Optimal | Worker 1 | Worker 2 |
| First 10 sec | Job 4, Job 1, Job 2 | Job 5, Job 3 |

Optimal Average Utilization: 0.95

Figure : Prioritizing “light jobs” causes poor performance.

Figure 2 shows that assigning *light jobs* (jobs with small CPU usages) first tends to cause poor performance in the long run. The explanation here is that by assigning light jobs first, we will be eventually facing the situation where only heavy jobs are left. Heavy jobs usually produce wasteful CPUs since they cannot be scheduled to run together on the same worker.

This leads us to Approach #2 which is essentially Approach #1 with prioritizing heavy jobs.

|  |
| --- |
| Sort jobs according to the descending order of CPU usages; // add this line to Approach #1  Repeat:  For each job I that has not been scheduled:  For each worker J that job I can fit in:  Assign job I to worker J;  Wait 1 second;  Until all jobs has been scheduled; |

Figure : Approach #2 – Heavy jobs first

Long Jobs First

Another way to priority jobs is give preference to their length (Approach #3). The reason behind this approach is that a long jobs if started late will most likely cause wasteful CPU since there are not many jobs left to run with it. Figure 4 gives a test case for which this approach achieve optimal utilization while Approach #2 does not.

2 workers

|  |  |  |
| --- | --- | --- |
| Job Id | CPU | Length |
| 1 | 10% | 30 sec |
| 2 | 30% | 10 sec |
| 3 | 70% | 10 sec |
| 4 | 100% | 10 sec |

|  |  |  |
| --- | --- | --- |
| Approach #2 | Worker 1 | Worker 2 |
| First 10 sec | Job 4 | Job 3, Job2 |
| Next 30 sec | Job 1 |  |

Approach #1’s Average Utilization: 0.286

|  |  |  |
| --- | --- | --- |
| Optimal | Worker 1 | Worker 2 |
| First 10 sec | Job 1 | Job 2, Job 3 |
| Next 10 sec | Job 1 | Job 4 |
| Last 10 sec | Job 1 |  |

Optimal Average Utilization: 0.383

Figure : The optimal schedule gives priority to long jobs and achieve optimal utilization. Approach #2 does not perform well in this case.

|  |
| --- |
| Sort jobs according to the descending order of job length; // add this line to Approach #1  Repeat:  For each job I that has not been scheduled:  For each worker J that job I can fit in:  Assign job I to worker J;  Wait 1 second;  Until all jobs has been scheduled; |

Figure : Approach #3 – Long jobs first

Combined approaches

|  |
| --- |
| Sort jobs according to the descending order of CPU usages, in case of equal CPU usages, sort according to the descending order of job length;  Repeat:  For each job I that has not been scheduled:  For each worker J that job I can fit in:  Assign job I to worker J;  Wait 1 second;  Until all jobs has been scheduled; |

Figure : Approach #4

|  |
| --- |
| Sort jobs according to the descending order of job length, in case of equal length, sort jobs according to the descending order of CPU usages;  Repeat:  For each job I that has not been scheduled:  For each worker J that job I can fit in:  Assign job I to worker J;  Wait 1 second;  Until all jobs has been scheduled; |

Figure : Approach #5

General approach with tuning parameter

This approach prioritizes jobs according to a weight function: , where is a tuning parameter that defines the relationship between CPU usage and length to priority. This approach is called general since it can be reduced to other approach by setting an appropriate value:

* Set , this approach becomes Approach #2.
* Set , this approach becomes Approach #3.
* Set , this approach becomes Approach #4.
* Set , this approach becomes Approach #5.

assuming CPU usages and job length are integer values within [0, 100].

Figure 8 shows how utilization varies upon in data set **tests/5/2\_128\_20\_20\_in.txt**.

Figure : Effect of tuning parameter to utilization in a data set

In order to determine the best value of , various algorithms can be used such as: sampling, hill climbing, simulated annealing, etc. In the implementation, we used sampling.

|  |
| --- |
| Define job weight = CPU usage + theta \* length;  Sort jobs according to the descending order of job weight;  Repeat:  For each job I that has not been scheduled:  For each worker J that job I can fit in:  Assign job I to worker J;  Wait 1 second;  Until all jobs has been scheduled; |

Figure : Approach #6 – General approach with tuning parameter

1. Evaluations

We implemented 6 approaches and run them against 540 generated data sets. Each of our data sets are labeled using the format *aa\_bbb\_cc\_dd\_in.txt* where:

* *aa* is the number of workers. It is either 2 or 20.
* *bbb* is the number of jobs. It varies between 2 to 200.
* *cc* is the percentage of heavy jobs. 100 – *cc* is the percentage of light jobs. If *cc* is 00, we generate job’s CPU usage uniformly randomly. *cc* is either 00, 70, 80 or 90.
* *dd* is the percentage of long jobs. 100 – *dd* is the percentage of short jobs. If *dd* is 00, we generate job length uniformly randomly. *dd* is either 00, 70, 80 or 90.

Detailed results are attached in file *Evaluations.xlsx*. Approach #6 achieves the maximum utilization among all approaches in 539/540 data sets. Approach #5 and #3 comes second and third with 383 and 334 data sets respectively. Approach #1 and #2 have the worst performance with only 112 and 122 sets.

There are 106 data sets (19.63%) in which Approach #6 is the only approach that produces the best utilization.