Project for Algorithm and Complexity Course

Kai Liu ID:520021910017 Email: kai_liu@sjtu.edu.cn
Rong Shan ID:520021911331 Email: shanrong@sjtu.edu.cn
Zhengxiang Huang ID:520021910014 Email: huangzhengxiang@sjtu.edu.cn

May 28, 2022

Abstract

The course project focuses on resource scheduling problem on single or multiple hosts in Hadoop. Since the Load Balancing Problem is a NP-hard problem, existing algorithms can not solve the optimal solution with polynomial time complexity. Our group tried to design an algorithm with polynomial time complexity to approximate the optimal solution, based on the idea of greedy algorithm.

In Problem 1, we proved the NP hardness of our problem, we computed the approximation ratio of simple greedy approximation, explored some tricky cases, and finally, did some analysis and visualization jobs.

In Problem 2, we further employed the heuristic algorithm of Stochastic Greedy Local Search with Random Walk to made up the disadvantages when applying intuitive single-core allocation to deal with the much more complicated Comprehensive Version. We theoretically analyzed the performance of our algorithm and test it on the test cases we generated randomly. Finally, we visualized the results.

We, as a team, learned a lot in the very days of cooperation, thanks.

Keywords: Distributed Computing System, Resource Scheduling, Greedy Algorithm, Approximation Algorithm

1 A Simplified Version with Single Host

In this problem, we have only one single host with several cores to deal with multiple jobs. Each job has several vary-sized data blocks and specific calculation speed. We need to schedule the blocks with cores to achieve the minimum processing time.

1.1 Formalization of the problem

In this section, we will give the formalization of the scheduling program with single host.

1.1.1 Input

First, we will formalize the input of this programming problem as

$$\textit{Input} = (J, C, \alpha, N, S, \textit{size}),$$

in which $J = \{job_0, job_1, \cdots, job_{n-1}\}$ represents the jobs, $C = \{c_0, c_1, \cdots, c_{m-1}\}$ represents the cores, $0 < \alpha < 1$ represents the decay factor in function $g(e_i) = 1 - \alpha(e_i - 1)$, $N = \{n_0, n_1, \cdots, n_{n-1}\}$ represents the numbers of data blocks in each job, $S = \{s_0, s_1, \cdots, s_n\}$ represents the calculation speed of each job on one single core and *size* is the function to fetch the size of data blocks.

1.1.2 Output

Second, we will formalize the output, namely the scheduling scheme, as

$$Sol = \{T, Core\},$$

in which, $T = \{t_0, t_1, \cdots, t_{n-1}\}$ represents each job's starting time, $Core = \{Core^0, Core^1, \cdots, Core^{n-1}\}$, $Core^i = \{Core^i_0, Core^i_1, \cdots, Core^i_{n_i}\}$ and $Core^i_k$ represent the core ID of i-th job's k-th block. With the starting time and core allocation scheme, we can easily calculate everything we want.

1.1.3 Objective Function and Constrains

Third, we will formalize the calculation process, objective function and the constrains. The processing time and finishing time of job_i on $corej$ are

$$tp^i_j = \frac{\sum_{b^i_k \in B^i_j} size(b^i_k)}{s_i \cdot g(e_i)}$$
$$tf^i_j = t_i + tp^i_j$$

Therefore the processing time and finishing time of job_i are

$$tp^i = \max_{c_j} tp^i_j, \text{ for } c_j \in C$$
$$tf(job_i) = \max_{c_j} tf^i_j, \text{ for } c_j \in C$$

With all the definitions above, we can give the objective function

$$\min \max\nolimits_{job_i} tf(job_i), \text{ for } job_i \in J.$$

And the constrains are as follows.

$$\forall i, j \in J, (i \neq j) \wedge (A_i \cap A_j \neq \phi) \rightarrow t_i \geqslant tf\,(job_j) \vee t_j \geqslant tf\,(job_i)$$

A_i is the set of cores assigned to i-th job. This constrain means that if any two jobs are allocated the same core, then the time of starting of one job can not be smaller the time of ending of the other job.

1.2 NP-Hardness of the Simplified Version

1.2.1 NPH Proof by restricting to Multiprocessor Scheduling

There is a NP-Complete problem described in the Book [1], which is called Multiprocessor Scheduling, or Multi-identical Machine Scheduling in other handy books.

The decision problem of Multiprocessor Scheduling is formalized as given a set of jobs with different length, is there an assignment that makes the return time $t \leq D,\ D \in Z^+$.

We can show that Multiprocessor Scheduling problem can be polynomial time reduced to our Simplified Version by Cook Reducibility.

Cook Reducibility Multiprocessor Scheduling problem can be solved immediately by calling the oracle of our Simplified Version of Search Problem, by restricting our Simplified Version to $\alpha = 0$ and $n_i = 1,\ \forall i$, that is, no decay factor and each job only has one data block. After calling the oracle that solves our Simplified Version, in condition that $\alpha = 0$ and $n_i = 1,\ \forall i$, we get can the minimum time for the scheduling problem is T. Then we can compare T and D to return yes if $T < D$, which is the answer of Multiprocessor Scheduling problem.

$$Multiprocessor\ Scheduling \leq_p Simplified\ Version \tag{1}$$

Thus, because $Multiprocessor\ Scheduling \in NPC$, so

$$Simplified\ Version \in NPH \tag{2}$$

1.2.2 Beyond NPC

Now, the question becomes whether or not our Simplified Version is in NPC, or is in NP. Actually, we are now proving that our problem is not likely to be in NP.

co-NPC: The complementary problem of Multiprocessor Scheduling problem is stated as "For a given number of jobs, is it that no assignment exists whose time $T \leq D,\ D \in R^{+}$?" We know this complementary problem, co-Multiprocessor Scheduling Problem, is in co-NPC, which is unlikely to be in NP, because it's hard to determine if there is a solution even for a Non-deterministic Turing Machine.

$$co - Multiprocessor\ Scheduling\ Problem \in co - NPC \tag{3}$$

$$if\ co - NP \neq NP,\ co - Multiprocessor\ Scheduling\ Problem \notin NP \tag{4}$$

This means that the certifer of co-Multiprocessor Scheduling Problem is unlikely to be in P.

$$co - Multiprocessor\ Scheduling\ Certifier \notin P,\ co - NP \neq NP \tag{5}$$

Then let's :turn to our Simplified Version. Simplified Version can return the minimum time and assignment. The Certifier can certify a certificate whether yield a minimum time T. **Proof By Contradiction:** If this Certifier can run in polynomial time, than it can tell us the minimum T, which can directly be used for the Certifier of co-Multiprocessor Scheduling Problem, only one more step: *decide* $T < D$?. This means the co-Multiprocessor Scheduling Certifier is polynomial time reduced to our Simplified Version Certifier.

$$co - Multiprocessor\ Scheduling\ Certifier \leq_p Simplified\ Version\ Certifier \tag{6}$$

Here, based on Equation 5 and Equation 6, we can prove Equation 7.

$$Simplified\ Version\ Problem \notin NP,\ if\ co - NP \neq NP \tag{7}$$

So even our Simplified Version is unlikely to be in NP.

1.3 Algorithm Design and analysis

Observation We noticed that the influence of decay is really large from the Figure 1

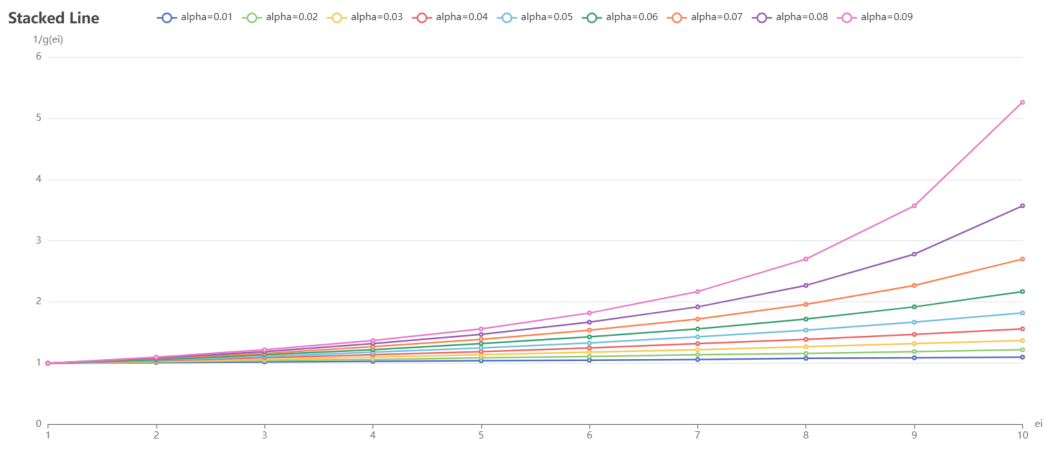

Figure 1: Relationship between $\dfrac{1}{g(e_i)}$ and e_i

Definition (*Load*) The *load* L_i of each core i is the total processing **time** of all job blocks executing on it.

Observation The decay factor α of executing speed can have a great influence on the executing time, so we must consider the number of cores we assign to a job with α. Here we list the relationship between α, the number of cores we assign to a job and the executing speed.

Single-Core Algorithm (*Greedy Algorithm Version*) Based on the observation,we design a greedy algorithm to assign each job one single core. First, we sort the jobs by their processing time on a single core from long to short.Then we assign jobs by order. Each time we assign a job, the algorithm chooses the core whose current load is smallest.

Algorithm 1: Single-host: Greedy Algorithm Version

```
Input: n,m,$\alpha$,S,size,b
Output: $L^* = max_job_itf(job_i)$
jobTime[] ← 0 ;
// every job's execution time
idx[i] ← $i$;
// record every job's executing order
for $j \leftarrow 0$ to $m$-1 do
    cmp[j].amt ← 0 ;
    // A struct to store the information of core;amt is current load
    cmp[j].id ← j ;
    push cmp[j] to CoreQueue ;
    /* A minimal priority queue to get the core whose current load is
       smallest                                                    */
end
for $i \leftarrow 0$ to $n-1$ do
    $jobTime[i] \leftarrow (\sum_b^i size(b_j^i)/s_i)$;
end
Sort jobTime[idx[i]] with decreasing order;
for $i \leftarrow 0$ to $n-1$ do
    $cmptemp \leftarrow CoreQueue.pop()$;
    $cmptemp.amt += jobTime[idx[i]]$;
    $CoreQueue.push(cmptemp)$;
end
$L_* \leftarrow max_{core_j} cmp.amt$;
```

1.3.1 Algorithm Analysis

Time complexity: Main time cost of our algorithm goes to sorting the n-size array *jobTime* and reorganization of priority queue *CoreQueue* after each pop and push. Sorting's time complexity is $O(n \log n)$. In the n-times *for-loop*, each push's and pop's time complexity is $O(\log m)$. So the total time complexity is $O(n \log n + (n+m) \log m)$.

Approximation Rate

Let's now analyze the approximation rate in the manner described in the textbook [2].

Denote t_j as the time consumes by the jth largest job. So, $t_1 \geq t_2 \geq \cdots t_n$. The processing time is T.

Lemma 1: The optimal processing time T^* is at least $\frac{1}{m}\sum_j tp^j$.

$$T^* \geq \frac{1}{m}\sum_j tp^j \tag{8}$$

Proof: Suppose at the best situation, the job can scatter evenly on all the cores, then we know that the OPT must be greater than this. Therefore, $T^* \geq \frac{1}{m}\sum_j tp^j$.

Favourable Situation: In the Favourable Situation, the most heavily loaded core ends up with more than 1 job. We can show here that under this situation, our approximation rate is of constant level.

Proof: Denote i as the slowest core index, and k is the last job placed on the core i, which consumes t_k time. Because there is more than 1 job on core i, and the scheduling process follows the order from big to small, $k \geq m+1$. Denote T_j as the total workload. We have that $T_i - t_k \neq 0$ is smaller than all the other cores' workload at that time, therefore, $m(T_i - t_k) + t_k \leq \sum_j tp^j$.

$$\begin{aligned}
T^* &\geq \frac{1}{m}\sum_j tp^j \\
T^* &\geq \frac{1}{m}\sum_j t_j \\
T^* &\geq (T_i - t_k) + \frac{1}{m}t_k \\
T^* &\geq T_i - \frac{m-1}{m}t_k \\
T^* &\geq T_i - \frac{m-1}{m}t_{m+1},\ k \leq m+1
\end{aligned} \tag{9}$$

Under this circumstances, we shall also notice the following property.

$$\begin{aligned}
T^* &\geq \frac{1}{m}\sum_j t_j \\
T^* &\geq \frac{1}{m}(t_1 + \cdots + t_{m+1}) \\
T^* &\geq \frac{m+1}{m}t_{m+1}
\end{aligned} \tag{10}$$

Based on Equation 9 and 10, we can have below property.

$$\begin{aligned}
T_i &\leq T^* + \frac{m-1}{m}t_{m+1} \\
T_i &\leq T^* + \frac{m-1}{m}\cdot\frac{m}{m+1}T^* \\
T_i &\leq \frac{2m}{m+1}T^* \\
T_i &\leq (2 - \frac{2}{m+1})T^*
\end{aligned} \tag{11}$$

The approximation rate of $(2 - \frac{2}{m+1})$ is actually a rather tight bound for the *Favourable Situation*. So we prove here that when the last core has loaded at least 2 jobs, our approximation is very appealing.

Bad Situation: In the bad situation, where the slowest core ends up with only one job on it, however, can be arbitrarily bad.

Proof: This is quite obvious there is one job that should originally split into small pieces is only assigned one core and lagged behind.

Modification In most cases,our algorithm has the approximation rate exactly smaller than 2. However, as is mentioned before, there is bad cases that make our algorithm dumpy. According to our algorithm, every job is assigned to one single core. A single extremely large job with a handful data blocks can be catastrophic. So we do need some improvement of our algorithm to approach some extreme cases.

After the assignment of jobs to cores by our greedy algorithm, we judge the number of jobs on the core who has the maximum load. If the jobs number is 1, it means there exists an extremely large job, then we may need to change our algorithm to more advanced ones (talked about and analyzed in Problem 2).

1.4 Results and Discussions

In the previous analysis, we've already discussed that there are 2 situations for the single-core strategy greedy approximation.

1.4.1 Favourable Cases

The first situation is relatively good. There are lots of jobs, and their sizes are evenly distributed. It can be seen as a favourable case for our algorithm. We have a constant approximation rate, as discussed before. So let's visualize our results in these cases. Figure 2 shows our results on the test file of $task1_case1.txt$ given by TAs. In this example, all the jobs sizes are distributed evenly, and thus yields a really good result. Just as we have proved before, in such cases the approximation rate is appealing.

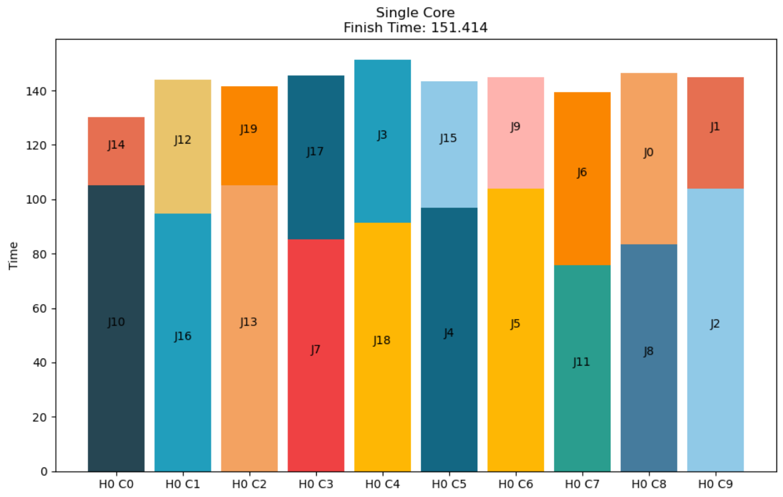

Figure 2: Single Host Greedy Approximation

The CPU utilization rate in this example is 95%, which means the approximation rate is close to 1, if this is not the OPT.

1.4.2 Bad Cases

In these cases, either the job number is small, even smaller than the number of cores, or the job sizes are biased. In these cases, we have a common problem that there are jobs that should originally be split up still maintained on a same core, which deteriorate the performance by large. We can see the bad performance below, in Figure 3. This is the toy example displayed in .pdf given to us.

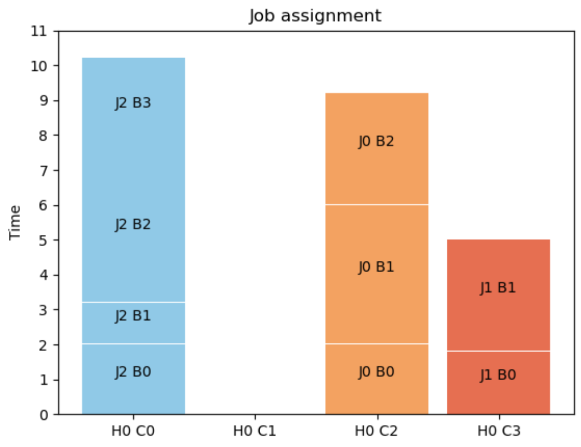

Figure 3: Single Host Bad Example

We can see here that this is definitely not a reasonable assignment. Actually we will revisit this problem again with our advanced algorithm we developed in the Comprehensive Version below. You can refer to Section 2.4 for more details on how we deal with this bad problem for our current simple greedy approximation.

1.4.3 A Little Conclusion

In Problem 1, we prove that the simplified version is already a NPH problem, so a polynomial time solution is not likely to be possible. Therefore, we introduced the greedy approximation algorithm, and displayed the results as before. There are some cases that our simple greedy approximation algorithm goes bad. Let's see the Comprehensive Version Problem and our advanced algorithm.

2 A Comprehensive Version among Multiple Hosts

In this problem, we need to consider the possibility of transferring the blocks to other hosts to speed up calculation. At the same time, there exists the transmission cost. This problem is even harder that it is NP-hard problem. We will give the formalization, NP-hard proof, algorithm(greedy and random walk) and results.

2.1 Formalization of the problem

Based on the former formalization of the problem, we give the formalization of scheduling program with multiple hosts.

2.1.1 Input

First, we will formalize the input as

$$\textit{Input} = (J, B, \widetilde{B}, C, \alpha, N, S, \textit{size}),$$

in which $J = \{job_0, job_1, \cdots, job_{n-1}\}$ represents the jobs, B^i_{lj} represents the set of data blocks of job i allocated to core c_j, $\widetilde{B^i_{lj}}$ represents the set of data blocks of job_i allocated to core c^l_j but not initially stored on h_l , B^i represents the set of data blocks of job i, $C_i = \{c^i_0, c^i_1, \cdots, c^i_{m_i-1}\}$ represents the cores on host i, $0 < \alpha < 1$ represents the decay factor in function $g(e_i) = 1 - \alpha(e_i - 1)$, $N = \{n_0, n_1, \cdots, n_{n-1}\}$ represents the numbers of data blocks in each job, $S = \{s_0, s_1, \cdots, s_n\}$ represents the calculation speed of each job on one single core and *size* is the function to fetch the size of data blocks.

2.1.2 Output

Second, we will formalize the output, namely the scheduling scheme, as

$$Sol = \{T, Core\},$$

in which, $T_i = \{t^i_0, t^i_1, \cdots, t^i_{|B^i|-1}\}$ represents each job i's block's starting time, $Core_i = \{Core^i_0, Core^i_1, \cdots, Core^i_{|B^i|}\}$ and $Core^i_k$ represents the core ID on which i-th job's k-th block runs. With the starting time and core allocation scheme, we can easily calculate everything we want.

2.1.3 Objective Function and Constrains

Third, we will formalize the calculation process, objective function and the constrains. The processing time and finishing time of job_i on core j of host l are

$$tp^i_{lj} = \frac{\sum_{b^i_k \in B^i_{lj}} size(b^i_k)}{s_i \cdot g(e_i)} + \frac{\sum_{b^i_k \in \widetilde{B^i_{lj}}} size(b^i_k)}{s_t}$$
$$tf^i_{lj} = t_i + tp^i_{lj}$$

Therefore the finishing time of core c_j^l for job_i is

$$tf(job_i) = \max_{c_j^l} tf_{lj}^i, \text{ for } c_j^l \in C$$

With all the definitions above, we can give the objective function

$$\min \max_{job_i} tf(job_i), \text{ for } job_i \in J.$$

And the constrains are as follows.

$$\forall i, j \in J, (i \neq j) \wedge (A_i \cap A_j \neq \phi) \rightarrow t_i \geqslant tf\,(job_j) \vee t_j \geqslant tf\,(job_i)$$

A_i is the set of cores assigned to i-th job. This constrain means that if any two jobs are allocated the same core, then the time of starting of one job can not be smaller the time of ending of the other job.

2.2 NP-Hardness of the Comprehensive Version

2.2.1 NPH Proof by restriction

We still use the simple proof by restriction to prove the NP-hardness of the Comprehensive Version.

Cook Reducibility: The Simplified Version can be solved by calling the oracle of the Comprehensive Version by restricting (host) q=1, and all the data blocks are on the very single core. Therefore, the Simplified Version can be polynomial time reduced to the Comprehensive Version.

$$Simplified\ Version \leq_p Comprehensive\ Version \tag{12}$$

The simplified version is NPH as we've proved before in Section 1.2, so the comprehensive version is NPH.

$$Comprehensive\ Version \in NPH \tag{13}$$

2.2.2 Beyond NPC

Just as what we've proved in Section 1.2, the Comprehensive Version is beyond NPC if $co-NP \neq NP$. The details are memtioned before.

$$co-NP \neq NP \Longrightarrow Comprehensive\ Version \notin NP \tag{14}$$

2.3 Algorithm Design and analysis

Redefinition(*Load*) In our multi-host version, the load L_i of one core i is the total transmission time plus the processing time of the job blocks.

Assumption In real life case, the transmission speed through Internet can largely exceed the processing speed of jobs. So in our multi-host version, we make a similar assumption that transmission speed exceeds processing speed(e.g. ten times faster).

2.3.1 Multi-host Algorithm 1 (*Simple One-Core Greedy Algorithm*)

According to the assumption, the transmission time has a smaller impact on the executing time. So when considering this problem, we can temporarily ignore the transmission progress and pay more attention to the processing progress. Then the multi-host problem can be degraded into a similar problem to single core problem. So our Algorithm 1 takes the same idea. We sort jobs by their single core processing time and assign them one single core to run. Each time we assign a job, we choose the core whose current load is smallest and then transmit all the blocks of the job to the core to run.

```
Algorithm 2: Multi-host: Greedy Algorithm Version
   Input: n,m,q,$\alpha$,S,size,B,C,b,$s_t$
   Output: $L^* = max_job_itf(job_i)$
 1 jobTime[] ← 0 ;
   // every job's execution time
 2 idx[i] ← $i$;
   // record every job's executing order
 3 for $i \leftarrow 0$ to $q$-$1$ do
 4     for $j \leftarrow 0$ to $m[i]$ do
 5         cmp.amt ← 0 ;
 6         cmp.id ← j ;
 7         cmp.idh ← i ;
           // cmp is a struct to store information of cores
 8         push cmp to CoreQueue ;
           /* A minimal priority queue to get the core whose current load is
              smallest                                                      */
 9     end
10 end
11 for $i \leftarrow 0$ to $n-1$ do
12     $jobTime[i] \leftarrow (\sum_b^i size(b_j^i)/s_i)$;
13 end
14 Sort n jobs by their jobTime so that jobTime[idx[0]] ≥ jobTime[idx[1]] ≥ ... ≥
    jobTime[idx[n-1]];
15 for $i \leftarrow 0$ to $n-1$ do
16     $cmptemp \leftarrow CoreQueue.pop()$;
17     $hid \leftarrow cmptemp.idh$;
18     $cid \leftarrow cmptemp.id$;
19     for $j \leftarrow 0$ to $|B^i|$ do
20         if $b_j^i \in B_{hid,cid}^i$ then
21             $cmptemp.amt+ = size(b_j^i)/s_i$;
22         else
23             $cmptemp.amt+ = size(b_j^i)/s_i + size(b_j^i)/s_t$;
24         end
25     end
26     $CoreQueue.push(cmptemp)$;
27 end
28 $L_* \leftarrow max_{core_j} cmp.amt$;
```

Time complexity: The main time cost of our algorithm goes to the sorting of the n-size array *jobTime*, pop and push action of the priority queue *CoreQueue* in the for-loop. The sorting's time complexity is $O(n \log n)$. The pop and push cost $O(\log m)$ in the n-times for-loop. So the total time complexity is $O(n \log n + (n+m) \log m)$.

2.3.2 Multi-host Algorithm 2 (*SLS + Greedy Approximation + Random Walk*)

In order to search the feasible solution space with a better coverage, we can turn to heuristic algorithms. A very appealing method applied in many fields is the **Stochastic Greedy Local Search** with or without the reinforcement of the **Random Walk** [3]. Here, we try to implement this idea on our problem.

A local search algorithm starts from a randomly chosen complete instantiation and moves from one complete instantiation to the next. The search is guided by some cost function related to the task, and at each step the value of the variable that leads to the greatest reduction of the cost function is changed. Then the algorithm may be restarted from a different initial random assignment. The algorithm stops either when the cost is minimized or when when there is no way to improve the current assignment by changing one variable.

To reduce the chance of reaching a local minimum, when randomizing the elements, we use another technique **Random Walk**. Random Walk starts with a random assignment and randomly flip the value of its variables.

Definition(*Solution Space*) The Solution Space of our algorithm denotes the order jobs executing and the numbers of cores assigned to the jobs. Typically, it's a struct Sol {vector<int> order, vector<int> core_num }.

Definition(*flip*) A **flip** is an action of swapping two neighbouring jobs' executing order or a increase/decrease of the number of cores assigned to one job. A flip causes a local change in out Solution Space.

Then we can get our **Multi-host Algorithm 2**. For *MAX_TRY* times, we randomly produce a Solution Space, and in every Solution Space, we do *MAX_FLIP* times of actions. Each action has two choices. The first choice, with probability p to be chosen, does a flip. And the other, with probability *1-p*, does a random walk. In our algorithm, the random walk simply takes no action, namely avoid doing a flip.

In each step, given the Solution Space, namely we know the order jobs should execute and the number of cores assigned to jobs. The problem is degraded into several sub-problems of minimizing the executing time of jobs. Here we use Greedy Algorithm. For a certain job j which is assigned k cores, we find k cores which has the k smallest load of all cores. Since each block of the same job has the same processing speed, the size of block is proportional to processing time. So we sort blocks of the same job from big size to small size. And each time we assign a block to a core, we choose the core which has the smallest current load in the selected k cores.

Algorithm 3: Multi-host: Greedy Algorithm+Greedy Search+Random Walk Version

```
Input: n,m,q,$\alpha$,S,size,B,C,b,$s_t$
Output: $L^* = max_job_itf(job_i)$
1  blockTime[] ← 0,idx[i] ← $i$ ;
2  for $i \leftarrow 0$ to $q$-1 do
3      for $j \leftarrow 0$ to $m[i]$ do
4          cmp.amt ← 0,cmp.id ← j,cmp.idh ← i ;
           // cmp is a struct to store information of cores
5          push cmp to CoreQueue ;
           /* A minimal priority queue on current workload           */
6      end
7  end
8  repeat
9      randomly initialize sol;
       // Greedy Algorithm to solve sub-problem
10     forall job do
11         normalize job's execution time;
12         sort blockTime[idx[i]] to decreasing order ;
13         pick next job and update core time in queue ;
14     end
       // Greedy Search:  take flips
15     repeat
16         randomly yield a choice for order or core number;
17         if change order then
18             randomly change two near job's order;
19             calculate the time cost;
20             randomly choose to become better or not;
21         else if core number then
22             randomly change the core number of a job;
23             calculate the time cost;
24             randomly choose to become better or not;
25         end
26     until repeat $MAX\_FLIP$ times;
27 until repeat $MAX\_TRY$ times;
```

Time complexity: In the *MAX_TRY*-times for-loop, each time we need to use greedy algorithm to solve the sub-problem like **Algorithm 1** and **Algorithm 2**, this step's time complexity can be analyzed similarly: for each job i, it needs $O(|B^i|\log|B^i|)$ to sort and $O((|B^i|+e_i)\log e_i)$ to do priority queue's work, so this step's time complexity is $O(n\cdot\max_i(|B^i|\log|B^i|))$. Then in the *MAX_FLIP*-times for-loop, local change will lead to another computation of the sub-problem. So in total, the time complexity is $O(n\cdot MAX\_TRY\cdot MAX\_FLIP\cdot\max_i(|B^i|\log|B^i|))$.

2.4 Results and Discussions

2.4.1 Comparison between Simple Greedy and SLS + Random Walk

Differences: For our Simple Greedy Algorithm, each job can only be assigned to a single core, which can be problematic though it escapes decay factors and may or may not avoid transmission overhead. On the other hand, SLS + Random Walk allows random issues (more like a NTM), for which reason it can cover a much larger region in our feasible space. It searches for local optimum solution in local regions and search for global optimum by random initialization. Thus, in SLS + Walk algorithm, we can optimize multi-core assignments, that is, a single job can be put on different cores to run. This is the major difference.

Common Points: The common point is that all our algorithm didn't take transmission speed into account when scheduling, which may or may not deteriorate the performance of our algorithm to some extent. On other aspects, our algorithm possess a solid theoretical foundation.

Case 1: Simple Greedy performs better

There are some cases when simple single-core processor Greedy Algorithm performs better than our SLS + Walk, in which cases, jobs are distributed evenly, or in another words, the *Favourable Cases* in Section 1.4. Why? It's because that in single-core Simple Greedy Approximation, the decay factor problem is reduced to the minimum scale, while SLS + Walk suffers.

Here goes our example, the *Favourable Cases* in Section 1.4. The following is the SLS + Walk result, where 217s is larger than 151s shown in Figure 2. Compare Figure 2 and Figure 4.

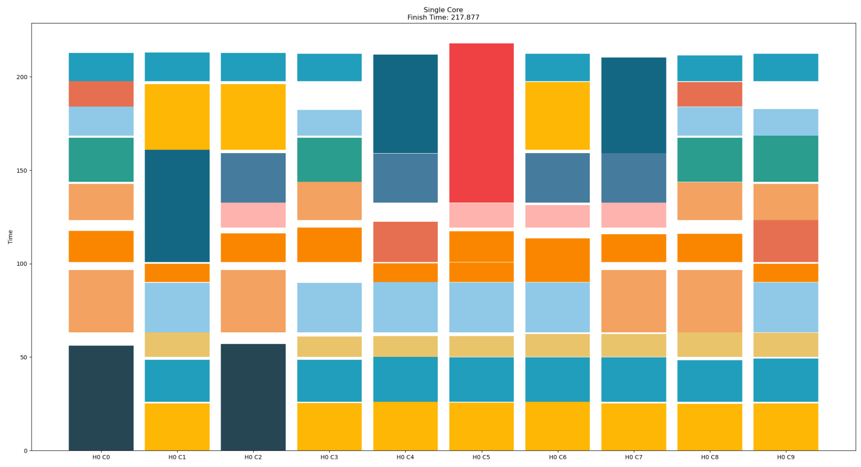

Figure 4: SLS + Walk Scheduling

We can actually find out in the so-called *Favourable Cases* Simple Greedy Algorithm outperforms our SLS + Walk.

Case 2: Other Cases Where SLS + Walk Stands Out

Actually, in virtually all the other cases, SLS + Walk gives out a far more reasonable answer than that given out by the Simple Greedy Algorithm. Let's first see some small cases.

The first example is the toy example in the .pdf, the single-host version, also refer to Figuer 3 we've discussed before. In the Figure 5, we can see that the SLS + Walk successfully solved the previously so-called *Bad Cases*. By permuting all possible combination, we can also find that our SLS + Walk actually gives out OPT in this case.

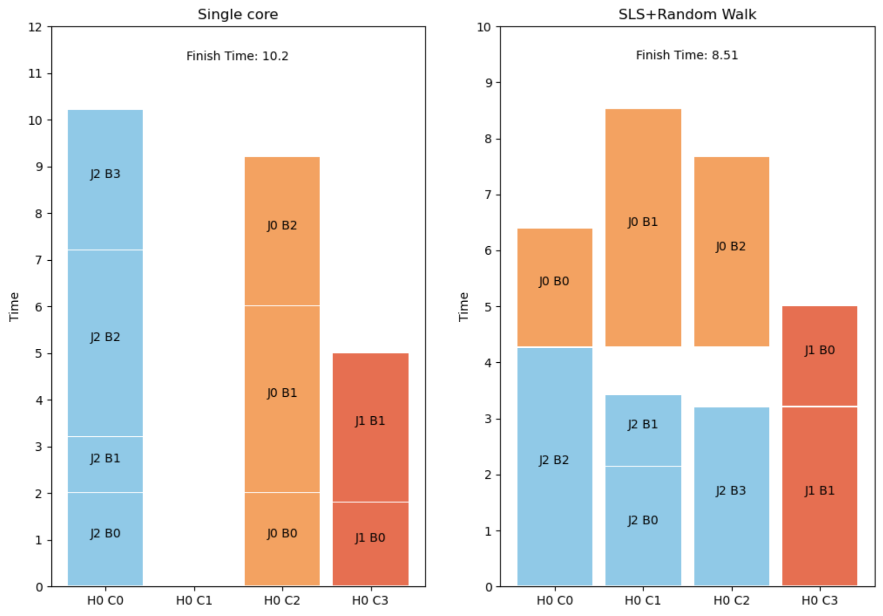

Figure 5: Single-Core Comparison

Let's see also the multi-host toy example given in the .pdf. In Figure 6, we can see that SLS + Walk again solves the *Bad Cases* astonishingly. Our SLS + Random Walk performs outstandingly here.

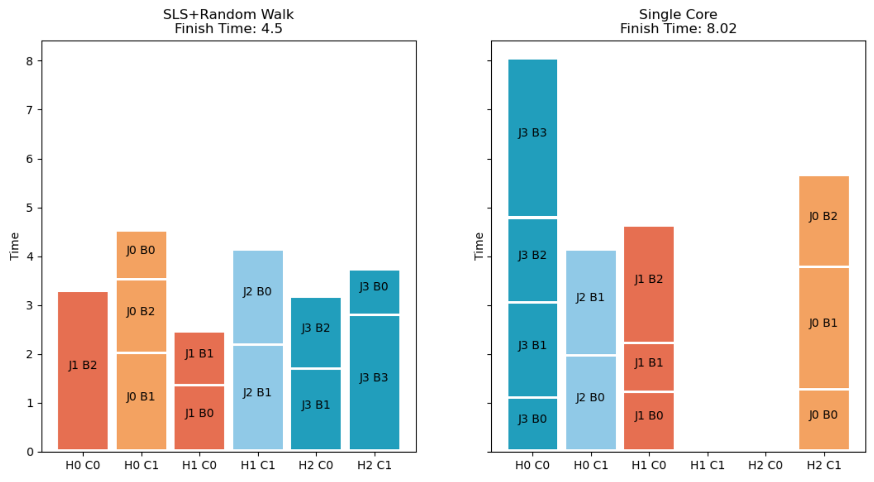

Figure 6: Scheduling upon Multi-Host Simple Problem

Let's see the final example for this section. This is the test problem in "test2_case2.txt" given by our adorable TAs. In Figure 7, we can see that SLS + Walk again performs really good. For Simple Greedy, it can never resolve such problems that involve enormous job with a handful of data blocks. Here, SLS + Walk reinforce it!

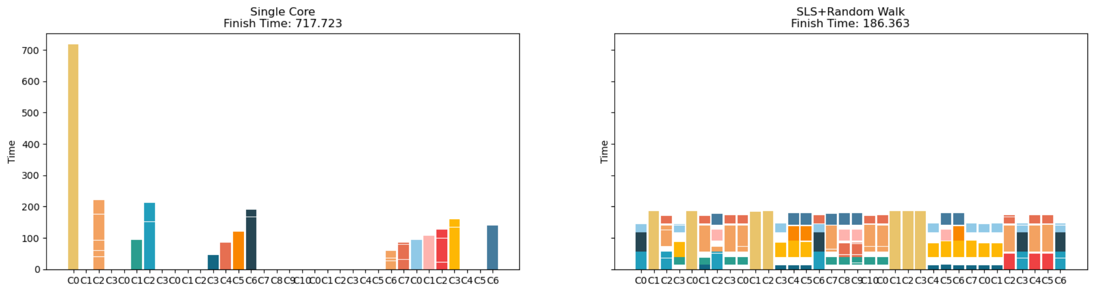

Figure 7: Scheduling upon Multi-Host Complicated Problem

186s is significantly better than 717.7s.

2.4.2 Further Analysis for SLS + Walk

Case 3: Performance Issue for Large Cases

In these large cases, we have mountainous jobs to finish, each with dozens of blocks, and accordingly, we have even more hosts and cores that is handy to use.

Let's do some experiments on SLS + Random Walk to observe the results on large cases.

The following Table 2.4.2 contains the comparison and the key performance issue of communication decay and CPU utilization rate. This Table can give us some idea of the scalability of our algorithms.

		1	2	3	4	5
simple greedy	utilization rate	0.15	0.97	0.37	0.27	0.34
	efficacious running time	14087	257	170	276	51
SLS + Walk	utilization rate	0.75	1	0.77	0.82	0.75
	efficacious running time	16050	259	177	300	53
	decay penalty rate	1.139348	1.007782	1.041176	1.086957	1.039216
		6	7	8	9	10
simple greedy	utilization rate	0.12	0.19	0.15	0.07	0.21
	efficacious running time	82913	67876	80907	49239	74798
SLS + Walk	utilization rate	0.57	0.78	0.7	0.53	0.75
	efficacious running time	86949	112700	100282	64645	106353
	decay penalty rate	1.048678	1.660381	1.239472	1.312882	1.42187

Table 1: small and large cases

We can tell from here that SLS + Walk can almost always have a good CPU utilization Rate, which is always welcomed. Besides, we can see from each cases, the decay rate, that is, the extra amount of work one must suffer from the benefit of more task-level parallelism. However, we can see that, almost always, the benefit from CPU utilization rate outweighs the overhead of decay rate.

Finally, we generate a massive example with hundred of jobs, thousands of blocks. We can take it as an extreme case, in Table 2.4.2.

		11
simple greedy	utilization rate	0.99
	efficacious running time	1965194
SLS + Walk	utilization rate	/
	efficacious running time	/
	decay penalty rate	/

Table 2: Extreme Case

In such an extreme example, our SLS + Walk ceased to work because of the time complexity. However, our Simple Greedy loves it, whose CPU utilization rate is 99%, almost OPT.

Now we have a more comprehensive overview upon the performance of our Simple Greedy Algorithm and SLS + Random Walk Algorithm.

Case 4: Transmission Rate and Decay Factor Matters

What if the transmission overhead is large that cannot be overlooked in scheduling processes? This is also a key issue we shall experiment on.

We actually observed that all the 3 algorithms perform worse than ever when transmission speed is unfavourable, because none of them have taken it into account. Figure 8 can gives us some hints on it.

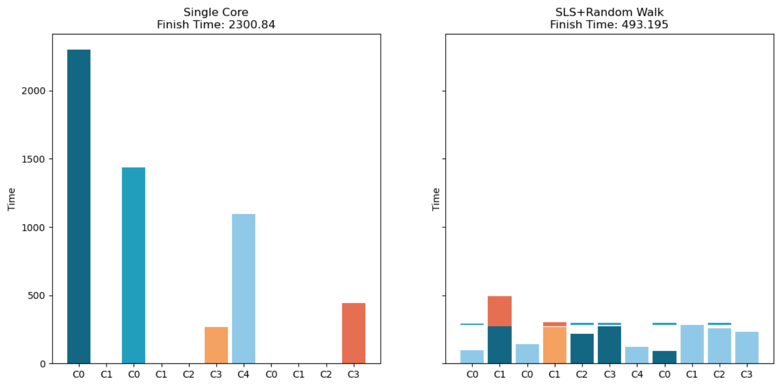

Figure 8: Large Transmission Overhead

In modern Hadoop, if the transmission bandwidth is enough, then this problem will never occur. That's why we aim to improve transmission bandwidth.

Anyway, taking into account the transmission speed is a possible improving field for future researches.

2.4.3 A Little Reflection

We can see that in most cases, SLS + Random Walk Algorithm performs astoundingly well. That's why we introduced this algorithm in our project report.

3 Conclusion

To sum up, our Simple Greedy Algorithm performs well for some *Favourable Cases* while the data blocks are distributed evenly. However, the performance of this trivial algorithm can degrade heavily when some *Bad Cases* happen.

To compare with it, when the decay factor is not too large, no matter how the data blocks or cores on hosts distribute, our SLS + Walk algorithm can always give a pretty good result which is a very close approximation of the optimal solution, since the algorithm can nearly traverse through the feasible solution space.

However, the decay factor and big-size data can challenge our SLS + Random Walk Algorithm a lot. From our analyzed time complexity, *MAX_TRY* and *MAX_FLIP* influence the performance a lot. With larger randomization iteration times, we may get better results, but the time cost goes up. Moreover, with larger decay factor, the performance of parallel processing suffers a lot and sequential running is better.

From the analysis above, it hinds on us that in real world, when handling similar load-balancing problem or resource scheduling in distributed system, we should take factors like transmission time and processing speed decay when doing parallel computing into consideration, which may influence a lot to our decision whether parallel computing is better. And the trade-off between time complexity and performance is another question we must consider.

Last but not least, there isn't a cure-all algorithm for this problem in the foreseeable future. However, the basic concept of greedy approximation and random/heuristic algorithm is always an option to us for an acceptable good solution.

4 Acknowledgements

In writing this paper, we have benefited from the presence of our teacher and our classmates a lot. They generously helped us figure out some hard problems and made many invaluable suggestions. We hereby extend out grateful thanks for their kind help, without which the paper would not have been what it is.

To be honest, the beginning is the most difficult. Because there is not a clear direction, we tried many possible ways but all failed. Later, we just jumped out of the limitation of the methods that we had already learnt, tried to learn some new methods to solve the problem. And that how we come up these fascinating algorithm. We have to admit that the most powerful algorithm sometimes comes from simple ideas, like the random walk algorithm and greedy search algorithm. Also, we learnt how to analyse the approximation rate and the NP related problem. In sum, we really benefited a lot from this project.

As for suggestions, we hope that there could be some useful reference or possible methods to solve the problem, otherwise we really have to spend a lot of time on designing the algorithm. It feels bad when an algorithm doesn't work. But the designing process and see the excellent output is really attractive.

Thank you very much! Let it be our ending.

5 Appendix

References

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences).* W. H. Freeman, first edition ed., 1979.

[2] J. Kleinberg and E. Tardos, *Algorithm Design.* USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[3] R. Dechter, "chapter 7 - stochastic greedy local search," in *Constraint Processing* (R. Dechter, ed.), The Morgan Kaufmann Series in Artificial Intelligence, pp. 191–208, San Francisco: Morgan Kaufmann, 2003.

Table 3: Symbols and Definitions

Symbols	**Definitions**
n	The number of jobs
m	The number of cores
q	The number of hosts
A_i	The set of cores assigned to i-th job
job_i, J	job_i is the i-th job. The job set is $J = \{job_0, \cdots, job_{n-1}\}$.
h_l, H	h_l is the l-th host. The host set is $H = \{h_0, \cdots, h_{q-1}\}$.
m_l	The number of cores on host h_l
c_j^l, C_l	c_j^l is the j-th core on host h_l. C_l is the set of cores on host h_l.
C	The set of cores. $C = \{c_0, \cdots, c_{m-1}\}$ for single-host. $C = \cup_{l=0}^{q-1} C_l$ for multi-host.
$Core$	The set of core allocation results
$Core^i$	The set of core allocation result of i-th job
$Core_k^i$	The core id of i-th job's k-th block
b_k^i	The block of job_i whose id is k
B_j^i	The set of data blocks of job_i allocated to core c_j
B^i	The set of data blocks of job_i
B_{lj}^i	The set of data blocks of job_i allocated to core c_j^l
$\widetilde{B}_{lj}^i$	The set of data blocks of job_i allocated to core c_j^l but not initially stored on h_l
$size(\cdot)$	The size function of data block
$size^i$	The sum of size of i-th job's data blocks
$size_k^i$	The size of i-th job's k-th data block
L_j	The sum of size of data blocks processed by j-th core(Load of j-th core)
$g(\cdot)$	The computing decaying coefficient caused by multi-core effect
s_i	The computing speed of job_i by a single core
s_t	The transmission speed of data
e_i	The number of cores processing job_i
t_i	The time to start processing job_i
tp_j^i, tf_j^i	The processing time / finishing time of core c_j for job_i
tp_{lj}^i, tf_{lj}^i	The processing time / finishing time of core c_j^l for job_i
$tf(job_i)$	The finishing time of job_i