SACG: Simulated Annealing Combined with Greedy Algorithm

An Approximation Solution to Resource Scheduling Problem in Hadoop

Zhenran Xiao, Zichun Ye, Mabiao Long
Student ID:{520030910281, 520030910302, 520030910308}
{xiaozhenran, alchemist, albertlong007} @ sjtu.edu.cn

Department of Computer Science and Engineering,
Shanghai Jiao Tong University, Shanghai, China

摘要 The course project focuses on resource scheduling problem in Hadoop. This document first introduce the background and two versions of resource scheduling on single or multiple hosts. Then it designs an algorithm based on simulated annealing algorithm and greedy algorithm to give an approximate solution to the problems of two versions. It also does some experiments on parameters in the algorithm and analyses the complexity and other quality of the algorithm.
Keywords: Distributed Computing System, Resource Scheduling, Hadoop, Simulated Annealing Algorithm, Greedy Algorithm, NP-complete problem

1 Background and Motivation

Hadoop is an open-source software framework for storing data and running applications on clusters of commodity hardware. In Hadoop, the data of a running job can be divided into several data blocks, stored on different hosts. Each host has one or multiple CPU cores to process data blocks. In this project, our goal is to achieve effective parallel computing. That is to say, given a set of jobs with massive data, we need to design a resource scheduling algorithm to minimize the overall executing time of all jobs.

2 Problem Formulation and Explanation

In this section, we will formulate the problem and explain some definitions.

2.1 A Simplified Version with Single Host

First, let us consider a simple case with a single host storing all data blocks. To simplify the problem, we give the following assumptions and specifications.

1. There are n jobs that need to be processed by a single host, which has m CPU cores of the same computing capability. Let J be the set of jobs, and C be the set of cores for the host, where $J = \{job_0, job_1, \cdots, job_{n-1}\}$, and $C = \{c_0, c_1, \cdots, c_{m-1}\}$ (The labels of variables start from 0 because we use C/C++ source codes in the following tasks).

2. We treat data block as the smallest indivisible unit in our project, while a job can be divided into multiple data blocks with different sizes for storage and computing. Assume job_i is split into n_i data blocks, denoted by $B^i = \{b_0^i, b_1^i, \cdots, b_{n_i-1}^i\}$. For block b_k^i of job_i, define its size as $size(b_k^i)$.

3. Assume job_i is assigned to e_i cores for processing, and naturally $e_i \leq n_i$. That is to say, one core can process multiple data blocks of one job sequentially. Let $B_j^i \subseteq B^i$ denote the set of data blocks of job_i allocated to core c_j, and $B_j^i \cap B_{j'}^i = \varnothing$ if $j \neq j'$ (they should be disjointed).

4. For job_i, the processing speed of its data blocks is s_i when job_i is assigned to a single core. However, when multiple cores process job_i in parallel, the computing speed of each core all decays because of some complicated interactions. We formulate such speed decay effect caused by multi-core computation as a coefficient function $g(\cdot)$ with respect to core number e_i, as described in Equation (1):

$$g(e_i) = 1.00 - \alpha \cdot (e_i - 1), \quad \text{for } 1 \leq e_i \leq 10, \tag{1}$$

where α is a decay factor satisfying $0 < \alpha < 1$, and usually the number of cores for processing a single job is no more than 10. Then, the speed of each core can be rewritten as $s_i \cdot g(e_i)$ for job_i respectively. (Note that although the speed of each core decays, the overall processing time using e_i cores in parallel should be faster than that of using just one core. Otherwise we do not need to implement parallel computing. Thus the setting of α should guarantee this principle.)

Correspondingly, the processing time tp_j^i of core c_j for job_i can be expressed as Equation (2):

$$tp_j^i = \frac{\sum_{b_k^i \in B_j^i} size(b_k^i)}{s_i \cdot g(e_i)}. \tag{2}$$

5. For consistency issues, if we assign a job to multiple cores, all cores must start processing data blocks at the same time. If one or several cores are occupied by other affairs, then all other cores should wait for a synchronous start, and keep idle. It means that the processing of job_i for every core should all start at time t_i, whereas their processing duration might be different. Let tf_j^i be the finishing time of core c_j for job_i, which is calculated by Equation (3):

$$tf_j^i = t_i + tp_j^i. \tag{3}$$

However, the occupied cores of job_i are released synchronously when the computing process of the last data block is finished. Thus the finishing time $tf(job_i)$ of job_i is given as Equation (4):

$$tf(job_i) = \max_{c_j} tf_j^i, \text{ for } c_j \in C. \tag{4}$$

2.2 A Comprehensive Version among Multiple Hosts

In this section, we consider a more complex situation, where we need to schedule resources among multiple hosts. Now the data transmission process between pairwise hosts should be taken into consideration. The data blocks of jobs could be initially stored on different hosts, but one data block can only be initially stored on one specified host. If data block b_k^i and its assigned computing core c_j are not on the same host, b_k^i will be transmitted to the host containing c_j (Here we assume

that the bandwidth between hosts is sufficient for data transmission). The transmission process will influence the finishing time of jobs, further affecting the resource scheduling process.

Besides the descriptions and specifications of Section 2.1, here are more notations and explanations.

1. Assume we have q hosts $H = \{h_0, h_1, \cdots, h_{q-1}\}$, and host h_l has m_l cores (may have different number of cores). Let C_l be the set of cores on host h_l, $C_l = \{c_0^l, c_1^l, \cdots, c_{m_l-1}^l\}$. Easy to see, $\sum_{l=0}^{q-1} m_l = m$.

2. If core c_j^l on host h_l computes a data block b_k^i of job_i which is initially stored on another host h_l', then b_k^i needs to be transmitted from h_l' to h_l at a transmission speed s_t (this speed is fixed in our system). An example is shown in Figure 1, where hosts h_0 and h_1 both have two cores, and many jobs need to be processed. Core c_0^1 on host h_1 is assigned to compute the data block b_2^1 of job_1, which is initially stored on host h_0. In this case, b_2^1 needs to be transmitted from h_0 to h_1 at a transmission speed s_t first, and then be computed by c_0^1. Whenever b_2^1 starts transmission, other cores can work in parallel to process job_1.

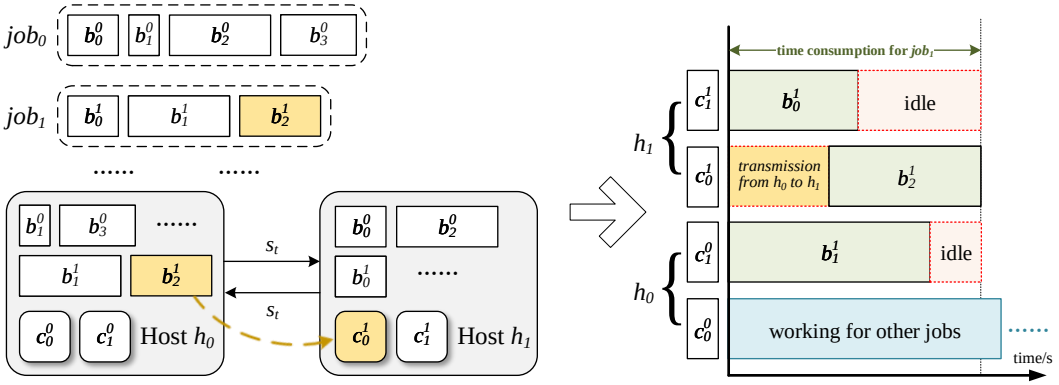

图 1. An example of data transmission between 2 hosts

3. Any core cannot call for data transmission when it is calculating other data blocks. Likewise, a core cannot start computing any data block until this block is ready, i.e. initially on the same host or transmitted from a remote host to the local host. For example, the core c_0^1 on host h_1 must wait for the data transmission of block b_2^1 from host h_0 to h_1, and then start computation. What is more, the transmission time of b_2^1 from h_0 to h_1 affects the finishing time of job_0, further affecting the finishing time of the whole system.

   For core c_j^l on host h_l, let $\widetilde{B}_{lj}^i$ be the set of data blocks of job_i allocated to c_j^l but not initially stored on host h_l. All the data blocks in $\widetilde{B}_{lj}^i$ need to be transmitted to host h_l before computing. Let B_{lj}^i be the set of data blocks of job_i allocated to core c_j^l. Then, the processing time tp_{lj}^i of core c_j^l for job_i can be reformulated as Equation (5):

$$tp_{lj}^i = \frac{\sum_{b_k^i \in \widetilde{B}_{lj}^i} size(b_k^i)}{s_t} + \frac{\sum_{b_k^i \in B_{lj}^i} size(b_k^i)}{s_i \cdot g(e_i)}. \tag{5}$$

4. If the processing of job_i starts at time t_i, then the finishing time of core c_j^l for job_i is

$$tf_{lj}^i = t_i + tp_{lj}^i.$$

Then the finishing time $tf(job_i)$ of job_i is formulated as:

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

3 Problem Soving

Based on the descriptions in Section 2, we want to design a resource scheduling algorithm to minimize the overall finishing time of all jobs, whose objective function is shown as:

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

And we hope the algorithm works for both versions of the problem.

3.1 SA: Simulated Annealing Algorithm

Simulated annealing algorithm was proposed by N. Tropolis scholar in 1953. It comes from the solid annealing principle, benefiting from the research results of statistical mechanics of materials. The algorithm is based on probability, mainly used to solve optimal solution problems, such as function extremum problems. In statistical mechanics of materials, different structures of particles in materials correspond to different energy levels of particles. At high temperature, particles have high energy and can move and rearrange freely. Low temperature particles have low energy. If the particles are cooled slowly from high temperature, namely annealed, they can reach thermal equilibrium at each temperature. When the system is completely cooled, a low-energy crystal is eventually formed.

The Principle of SA Algorithm Suppose the energy of the material in $state_i$ is $E(i)$, then the law of the material entering $state_j$ from $state_j$ at temperature T is as follows:

1. If $E(j) \leq E(i)$, we accept the $state_j$, namely the material enters $state_j$ from $state_i$.
2. If $E(j) > E(i)$, it is probable to accept the $state_j$. The probability is

$$e^{\frac{E(i)-E(j)}{KT}}$$

K is Boltzmann's constant, and we usually make K equal to 1 in fact. T is the temperature of the material.

When the temperature is very low, we think it is highly possible that the material is in the minimum energy state.

The Flow of SA Algorithm The flow of the algorithm is following:

1. Generating the initial state $state_0$. Set the initial temperature T_0 and the final temperature T_e. Set the rate of temperature fall β. Set the current number of iterations i and the maximum number of iterations M.
2. In each iteration, $T_i = \beta \cdot T_{i-1}$. Move the paticles to generating a new state $state_i$ by some scheme.
3. Calculate the change of the energy by an objective function:

$$\delta E = E(state_i) - E(state_{i-1})$$

   If $\delta E \leq 0$, accept the $state_i$. If not, accept the $state_i$ by probability $e^{\frac{-\delta E}{T_i}}$.
4. If the number of iterations arrives M or the temperature is lower than T_e, stop the iteration and obtain the final state as the result.

3.2 The Overview of Our Algorithm

In the resource scheduling problem, we can regard every feasible solution as a state, the finishing time as the energy, the blocks as the particles. We generate a feasible solution as the initial state at the beginning and set the initial temperature, final temperature and maximum number of iterations for the problem. Next, in each iteration we move the blocks by some schemes to generate a new solution as a new state. However, the moving scheme needs to be limited by the problem formulation in Section 2. According to the new state, we can calculate its energy and decide whether to accept it. When the iteration stops, we can get a final state as the approximation solution — we think it is highly possible that this solution is good enough.

Considering the following restrictions:

1. If we assign a job to multiple cores, all cores must start processing data blocks at the same time. If one or several cores are occupied by other affairs, then all other cores should wait for a synchronous start, and keep idle.
2. The occupied cores of a job are released synchronously when the computing process of the last data block is finished.

We can easily know that, for one job, if we have decided which cores will be allocated to it, it will be like a cube exisiting on the simulated image of the solution no matter where it is on the image. The width of it is the number of the cores allocated to the job. The length is the longest processing time on these cores. (Maybe the width of the cube is not connected together, but it doesn't matter too much.)

Therefore, we can use Greedy Algorithm for a job to make the length of its cube shorter, namely make the processing time for one job shorter. In both versions, Greedy Algorithm can give a good enough solution for one job.

When the cubes are fixed, we can just put them in the execution queues by some order, like the game *Tetris*.

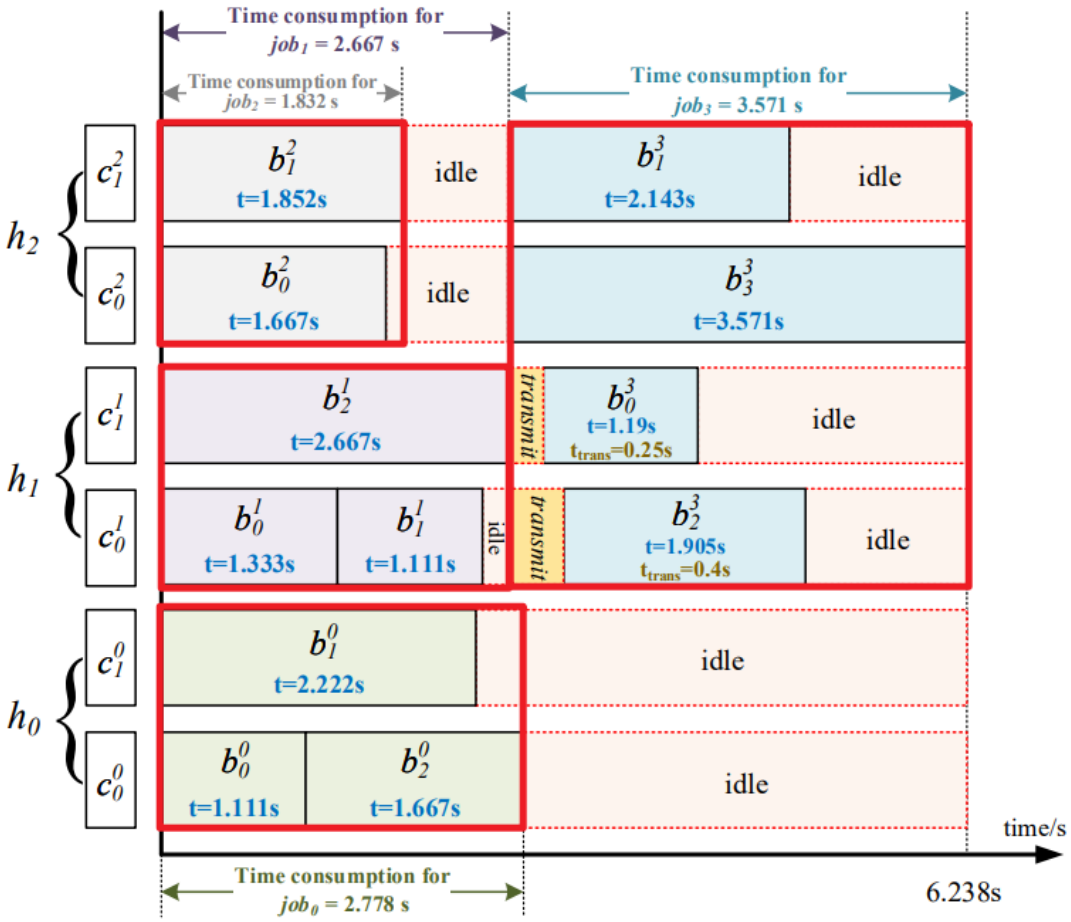

图 2. A simulated image of a solution

So the overview of our algorithm is:

1. Regard each job as a cube, generate the initial solution namely initial state in the Simulated Annealing Algorithm.
2. Move these cubes and change their width to generate a new state.
3. Considering the fixed width of cubes, use Greedy Algorithm to make their length shorter.
4. Simulate the calculating process to obtain the final finishing time, namely the energy of the state.
5. Perform the annealing, decide whether to accept the new state.
6. If the temperature and the number of iterations don't reach the standard, go to 2.
7. Otherwise, end the iteration, the final state is the final solution.

Here is the pseudo code:

Algorithm 1: *schedule*()

Input: all the information about hosts, cores, jobs, blocks

Output: the final state

```
1  init();
2  E ← 0;
3  tempE ← 0;
4  T ← 1000;
5  T_e ← 0.0001;
6  β ← 0.999;
7  δE ← 0;
8  M ← 100000;
9  repeat
10     move();
11     E ← finishTime();
12     tempE ← temp_finishTime();
13     δE ← tempE − E;
14     if δE < 0 then
15         Accept the new state;
16     end
17     else
18         There is a probability of exp(−δE/T) to accept the new state;
19     end
20     T ← T ∗ β;
21     if T < T_e then
22         break;
23     end
24     M ← M − 1;
25 until M = 0;
```

3.3 The Data Structure Mainly Used in the Program

We mainly use these data structure in the program:

```
1        vector<int> hostCore;              // The number of cores for each host
2        vector<int> jobBlock;              // The number of blocks for each job
3        vector<double> Sc;                 // Speed of calculation for each job
4        vector<vector<double>> dataSize;   // Job-> block number-> block size
5        vector<vector<int>> location;      // Job-> block number-> block location (host
             number)
6
7        vector<double> jobFinishTime;      // The finish time of each job
8
```

```
vector<vector<tuple<int, int, int>>> runLoc;
// 1. Block perspective: job number->block number->(hostID, coreID,rank), rank=1 means that block is the first task running on that core of that host

vector<vector<vector<tuple<int, int, double, double>>>> hostCoreTask;
// 2. Core perspective: host->core->task-> <job,block,startRunningTime, endRunningTime>

vector<vector<double>> hostCoreFinishTime; // host->core->finishTime

/* 3. This representation of a solution is used for SAA */
vector<int>jobCore;              // The number of cores allocated to each job.
vector<int>jobOrder;             //jobOrder[1] stores the first job
vector<vector<tuple<int, int, vector<int> > > > total_sol;
//the information of the number of cores are stored in jobCore
//job-> core ->(hostID, coreID, vector of blockID)
vector<vector<int> >blockLocation;
//job-> blockID-> block destination (hostID)

//modify the solution temporarily
vector<int>temp_jobCore;
vector<int>temp_jobOrder;
vector<vector<tuple<int, int, vector<int> > > > temp_total_sol;
vector<vector<int> >temp_blockLocation;
```

We will use the third representation of solutions during the annealing. It means different *jobOrder*, *jobCore* and *total_sol* together represent different states. When we generate a new state, we can use it to obtain the finishing time, namely the energy of the state by simulating the calculating process. When we get the final state, we can transform this representation to the block perspective and core perspective solution.

3.4 The Scheme to Generate the Initial State

We already know for one job, if we have decided which cores will be allocated to it, it will be like a cube exsiting on the simulated image of the solution no matter where it is on the image. So we only need to generate the *jobOrder* and the cores which are allocated to each job. We use random method to generate the two information above. The flow of the scheme is as followed:

1. Initialize the *jobOrder* to $1, 2, 3, .., n$.
2. For each job, randomly decide the number of cores allocated to it, but restrict that the cores can't more than its blocks.
3. For each job, randomly decide which cores will be allocated to it. The probability of each core that will be chosen is equal.

4. For all the cores allocated to one job, put its blocks into the queues of cores, one for each by order, until the blocks are all put in.

The step 4 above is meant to make the energy of the initial state higher, to enlarge the effect of moving at the beginning.

Here is the pseudo code:

Algorithm 2: *init*()

Input: all the information about hosts, cores, jobs, blocks

Output: the initial state

```
1  jobOrder ← (1, 2, ..., n) ;
2  for i ← 0 to n−1 do
3      jobCore[i] ← rand()%min(10, jobBlock[i], m) + 1 ;
4      for j ← 0 to jobCore[i] − 1 do
5          pick a core that hasn't been allocated to job_i;
6          total_sol[i][j] ← (host_id, core_id, process queue for this job);
7      end
8      for k ← 0 to jobBlock[i] − 1 do
9          total_sol[i][k%jobCore[i]].queue.push(k);
10         blockLocation[i][k] ← total_sol[i][k%jobCore[i]].host_id;
11     end
12 end
13 temp_jobOrder ← jobOrder ;
14 temp_jobCore ← jobCore ;
15 temp_total_sol ← total_sol ;
16 temp_blockLocation ← blockLocation ;
```

3.5 The Scheme to Move Blocks to Generate New State

The same reason, we only need to change the *job_order* and the cores allocated. We have two strategys to change them:

1. Randomly choose two adjacent jobs, change their order. This will happen at a probability of P_{co}. The probability of each job being chosen is equal.
2. Randomly choose one job, randomly decide the number of cores allocated to it, but restrict that the cores can't be more than its blocks and ten, then randomly decide which cores will be allocated to it. This will happen at a probability of P_{rc}. The probability of each core being chosen is equal.

Here is the pseudo code:

Algorithm 3: *move*()

Input: the last state

Output: the new state

```
p ← rand()%2;
if n ≥ 2 and p != 0 then
    i ← rand()%(n − 1) + 1;
    swap(temp_jobOrder[i], temp_jobOrder[i + 1]);
end
p ← rand()%2;
if p != 0 then
    i ← rand()%n;
    temp_jobCore[i] ← rand()%min(10, jobBlock[i], m) + 1 ;
    for j ← 0 to temp_jobCore[i] − 1 do
        pick a core that hasn't been allocated to job_i;
        temp_total_sol[i][j] ← (host_id, core_id, process queue for this job);
    end
    greedy_allocation(job_i);
end
```

3.6 Make the length of Cubes Shorter by Greedy Algorithm

For one job, when the cores allocated to it have been decided, namely the width of the cube has been fixed, we use Greedy Algorithm to make its length shorter:

1. Sort all blocks of the job in descending order of data size.
2. From the biggest block, compare which core to place it on can minimize the current processing time. Put it on that core.
3. Decide all blocks position by 2.
4. During the process, we can update the *blockLocation*.

Here is the pseudo code:

Algorithm 4: *greedy_allocation*(job_i)

Input: job_i

Output: the schedule scheme for this job

```
1  sort all blocks of job_i in descending order of data size, use
   jobBlock_sorted.push((block_id, datasize));
2  coreLength ← (0, 0, .., 0);
3  for k ← 0 to jobBlock[i] do
4      b ← jobBlock_sorted[k].block_id;
5      min ← MAX_LENGTH;
6      for j ← 0 to temp_jobCore[i] do
7          if temp_total_sol[i][j].host_id= location[i][b] then
8              ratio ← 1
9          end
10         else
11             ratio ← 1 + Sc[i] * g(jobCore[i])/St;
12         end
13         if coreLength[j]+dataSize[i][b] * rate <min then
14             min ← coreLength[j] + dataSize[i][k] * ratio;
15             index ← j;
16         end
17     end
18     coreLength[index] ← min;
19     temp_total_sol[i][index].queue.push(b);
20     temp_blockLocaton[i][b] = temp_total_sol[i][index].host_id;
21 end
```

3.7 The Scheme to Calculate the Energy

We will simulate the calculating process to obtain the final time, namely the energy. The simulating scheme is following:

1. Initialize a data structure tf. It is a two-dimensional array, meaning $host- > core- > releaseTime$.
2. According to $jobOrder$ and $total_sol$, put the cube of each job into the execution queue one by one, like playing $Tetris$.
3. For every next job, first find the core whose end time is the latest among all the cores allocated to this job and set its end time as the start time of this job.
4. Then add this job's cube length, namely the total processing time of the job, to the start time, obtain finish time of the job, update tf for all the cores allocated to this job.
5. Go to 3.
6. If all the jobs are scheduled, end the cycle.

After simulating, we obtain the final finish time as the energy of the state.

Here is the pseudo code:

Algorithm 5: $finishTime()$

Input: $jobOrder$, $jobCore$, $total_sol$

Output: $finishtime$

```
for i ← 1 to n do
    jobi ← jobOrder[i];
    for j ← 0 to jobCore[jobi] − 1 do
        host_id ← total_sol[jobi][j].host_id;
        core_id ← total_sol[jobi][j].core_id;
        block_id ← total_sol[jobi][j].queue;
        tp_j i ← 0;
        for k = 0 to block_id.size() − 1 do
            if location[jobi][block_id[k]] = blockLocation[jobi][block_id[k]] then
                tp_j i+ = dataSize[jobi][block_id[k]]/(Sc[jobi] · g(jobCore[jobi]));
            end
            else
                tp_j i+ = dataSize[jobi][block_id[k]]/St +
                  dataSize[jobi][block_id[k]]/(Sc[jobi] · g(jobCore[jobi]));
            end
        end
        tp_i[j] ← tp_j i;
        if t_i < tf[host_id][core_id] then
            t_i ← tf[host_id][core_id];
        end
    end
    tf_jobi ← t_i + max(tp_i[]);
    update tf[host_id][core_id] for all cores allocated to jobi;
    if finishtime < tf_jobi then
        finishtime ← tf_jobi;
    end
end
return finishtime;
```

4 Results and Parameter Experiments

4.1 Final Results

We first give a sample result of our algorithm:

For $Task1$:

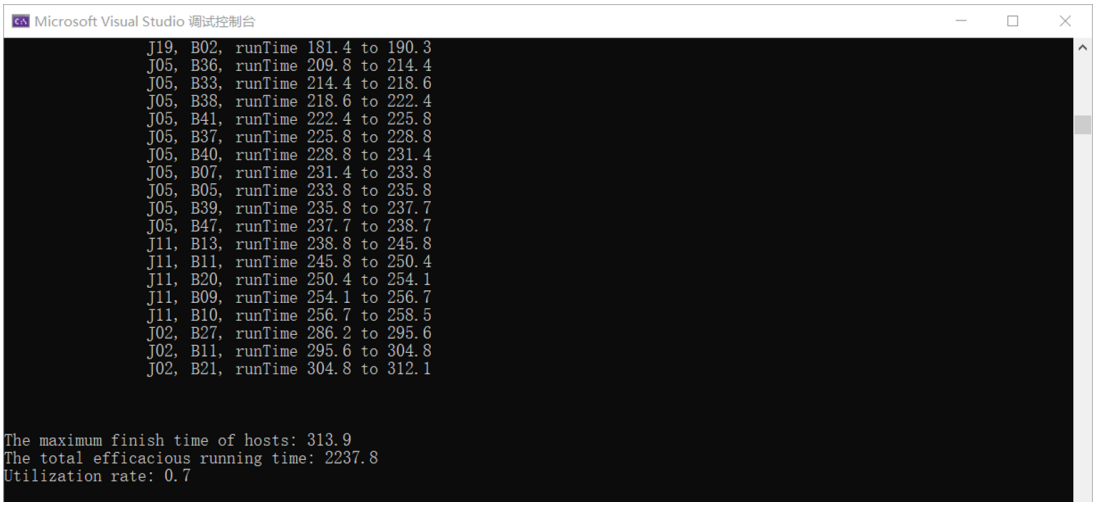

图 3. The sample result of Task1

For $Task2$:

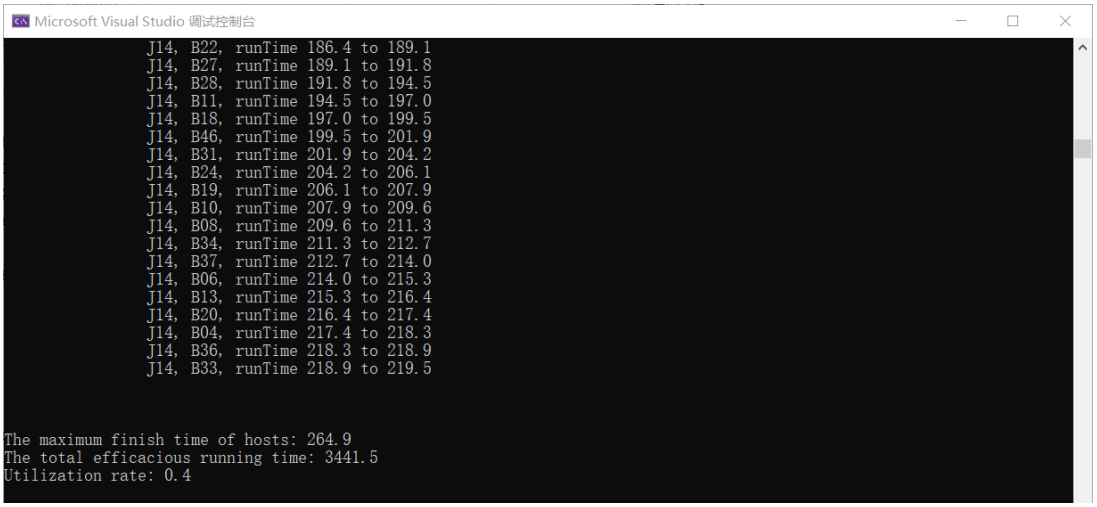

图 4. The sample result of Task2

The complete output can be checked in `task1_case1_01.txt` and `task2_case1_01.txt`.

We can visualize the annealing process:

For $Task1$:

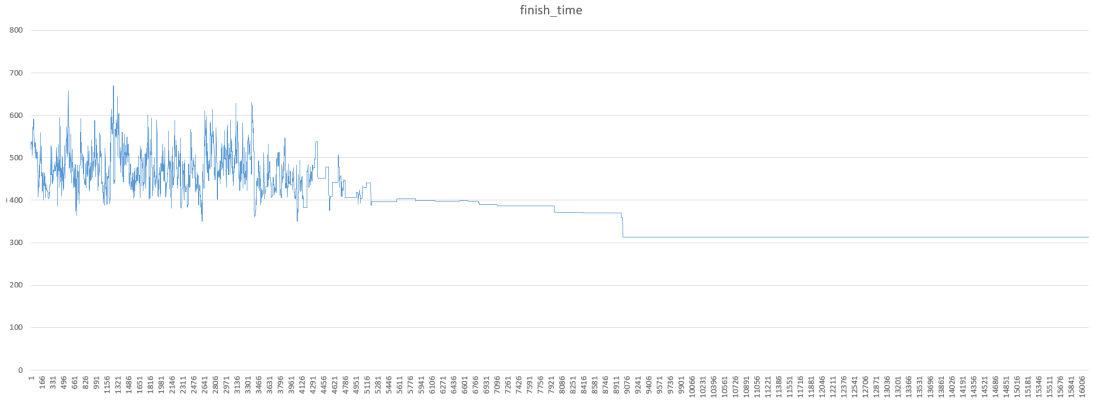

图 5. The annealing process of Task1

For $Task2$:

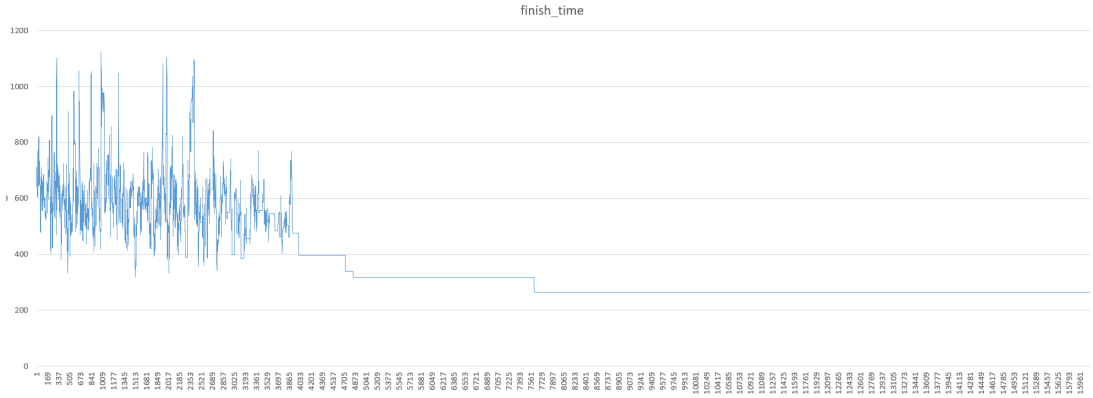

图 6. The annealing process of Task2

4.2 Parameter Experiments

In this section, we will use different parameters to test the simulated annealing algorithm and compare the different results.

For $Task1$:

表 1. The average result of Task1 for different parameters

	$P_{co} = 0.25$	$P_{co} = 0.5$	$P_{co} = 0.75$
$P_{rc} = 0.25$	361.10	356.70	353.16
$P_{rc} = 0.5$	360.54	358.16	349.98
$P_{rc} = 0.75$	344.74	354.98	354.90

For $Task2$:

表 2. The average result of Task2 for different parameters

	$P_{co} = 0.25$	$P_{co} = 0.5$	$P_{co} = 0.75$
$P_{rc} = 0.25$	352.64	350.56	354.18
$P_{rc} = 0.5$	332.76	315.92	331.48
$P_{rc} = 0.75$	316.10	324.28	323.02

However, the parameters will be affected by the test case itself. The results we get may not be universal.

5 Algorithm Analysis

In this section, we will analyze the time complexity of our algorithm. And we will justify whether this problem is NP-complete.

5.1 Time Complexity

1. The times of annealing is
$$min(M, log_\beta \frac{T_e}{T_0})$$
2. In $move()$, the time complexity of changing order is $O(1)$. As for choosing cores, the max number of cores allocated to one job is 10. Assume that we need X times to pick out 10 different cores for one job, X_i is how many times it needs to pick a new core when it has already picked $i-1$ cores. X_i follows the a geometric distribution.
$$E(X_i) = \frac{1}{p_i} = \frac{m}{m-i+1}$$

p_i means the probability of choosing a new core when it has already picked $i-1$ cores.
Because $X = \sum_{i=1}^{10} X_i$,

$$E(X) = \sum_{i=1}^{10} \frac{m}{m-i+1} \to 10$$

3. The time complexity of $greedy_allocation(job_i)$ is $O(jobBlock[i] \cdot jobCore[i])$. Because $jobCore[i] \leq jobBlock[i]$, the time complexity is actually $O(jobBlock[i]^2)$. Therefore, the time complexity of $move()$ is $O(10 \cdot max(jobBlock[])^2) \sim O(max(jobBlock[])^2)$.
4. $finishTime()$ essentially iterates through all the blocks and then finds the max finish time on cores. So the time complexity is $O(\sum_{i=0}^{n-1} jobBlock[i] + m)$.
5. The other functions just have the same time complexity of the functions above or smaller.

So the time complexity of the whole algorithm is

$$O(min(M, log_\beta \frac{T_e}{T_0}) \cdot [max(jobBlock[])^2 + \sum_{i=0}^{n-1} jobBlock[i] + m])$$

5.2 Explanation to Greedy Thoughts

First of all, we would like to explain how we figure out the greedy algorithm used for shortening the length of cubes of jobs.

We do the following qualitative analysis:

1. The length of a job's cube must be larger than the max size of its blocks.
2. Every time we put a new block into the cores' queue, the increase of the length is not bigger than the size of this block.
3. We want the idle part to be as small as possible. And the max idle part is the gap between the shortest queue and the longest queue.
4. If adding a block makes the longest queue changes, we want the increase of the length of the cube is the smallest.

Based on them, we designed the $greedy_allocation()$. It obtain a good enough soltion for each job in the simplfied version. Because the processing time for each block is proportional to its size. But it isn't optimal. We will explain it in Section 5.3.

In the comprehensive version, the transmission time will affect the actual processing time of a job. Because some blocks have no need to transmit, while the others need. The actual processing time is still proportional to its size, but the ratio is different for each block. So the Greedy Algorithm can't work better than it does in the simplified version. However, because the transmission time of each job is one tenth of the processing time or less, the greedy algorithm can still give a good solution.

5.3 NP-complete Proof

NP If we know the minimum finish time and the list of places of all blocks, we use function $finishtime()$ to certificate, which leads to the polynomial time complexity.

Partition Problem The partition problem, or number partitioning, is the task of deciding whether a given multiset S of positive integers can be partitioned into two subsets S_1 and S_2 such that the sum of the numbers in S_1 equals the sum of the numbers in S_2. The partition problem is NP-complete[4].

NP-complete of Simplest Case We consider the simplest case that there is only one job allocated onto 2 cores.

Consider the partition problem about the set of blocks' working time A. If we find a solution of the problem that there is no idle time ($finishtime = \frac{1}{2}\sum_{a\in A} a$), we get a solution of partition problem about A, and if there is idle time existing ($finishtime \neq \frac{1}{2}\sum_{a\in A} a$), we know that there is no solution of this partition problem.

The reduction takes $O(n)$ time complexity to sum all elements in A and $O(1)$ time of the "black box" algorithm, so it is a polynomial reduction, which means the simplest case is NP-complete.

NP-complete of General Case It is obvious that the general case, with more jobs and more blocks, is harder than the simplest case. The discussion below is based on the assumption that we have an algorithm $f(m, n)$ to solve m job case on n cores, where m and n are arbitrary. Single-host or multi-host doesn't matter, because we only need to add a special parameter on the size of blocks.

Considering the simplest case, we can set $m = 1$ and $n = 2$, then $f(1, 2)$ is an algorithm to solve it, which leads to a polynomial reduction. Since the simplest case is NP-complete, the general case is NP-complete as well.

5.4 Experiment on Greedy Quality

In Section 5.3, we prove that minimizing the length of one job cube with fixed core number is an NP-complete problem. If $P \neq NP$, we can't find a optimal solution in polynomial time. The greedy algorithm we used for the simplified version is not optimal. (For the comprehensive version it is obviously not the optimal.) Therefore, in the simplified version, we compare the solution which our greedy algorithm gives to the theoretical optimal solution, in order to test whether it gives a good enough solution. The theoretical optimal cube length for job_i is

$$\frac{\sum_{k=0}^{jobBlock[i]-1} datasize[i][k]}{jobCore[i]}$$

It is easy to know that the actual optimal cube length must not be shorter than the theoretical one.

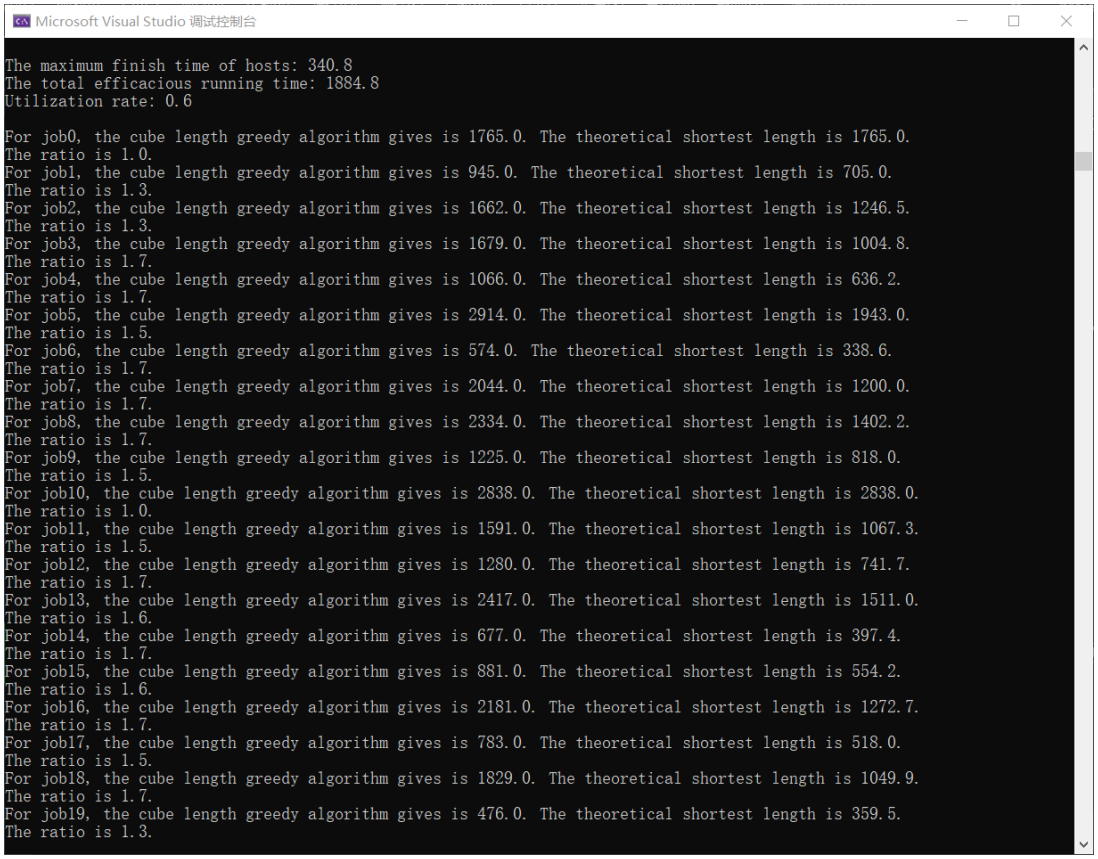

图 7. Comparation between greedy solution and theoretical optimal solution

The complete output can be checked in `task1_case1_01 (2).txt`.

We can see the cube length greedy algorithm gives is not more than 2 times the theoretical optimal one. For the actual optimal solution this ratio may be smaller. And sometimes it gives a result as good as the theoretical optimal one. Therefore, we think our greedy algorithm gives a good enough solution.

6 Experimemt Based on A Completely Random Move

In Section 5, we explain and test the quality of the greedy algorithm we used. And we can know for multiple host version greedy algorithm is not optimal. So we design another move method which not only moves the job cubes, but also moves the blocks in each job cube. Namely, we will use a completely random strategy to generate new states and check if it can give us a better result. The new move strategy is following:

1. Randomly choose two adjacent jobs, change their order. This will happen at a probability of P_{co}. The probability of each job that will be chosen is equal.
2. Randomly choose one job, randomly choose a core that has been allocated to it, change it to another core that hasn't been allocated to it. This will happen at a probability of P_{rc}. The probability of each core that will be chosen is equal.

3. Randomly choose one job job_i, randomly choose a core $core_j$ that has been allocated to job_i, randomly choose a block B_k that on this core. Then randomly generate a core index. The core index will be in the range of $[0, temp_jobCore[i]]$, but it can't be the $core_j$. This will happen at a probability of P_{mb}.

   If the core index is equal to $temp_jobCore[i]$, it means we will give a new core to job_i. We randomly choose this new core from the cores which haven't been allocated to job_i. If there is only one block on $core_j$, we will directly change $core_j$ to this new core. If there are more than one block on $core_j$, we add this new core to job_i and move B_k from $core_j$ to this new core.

   If the core index is smaller than $temp_jobCore[i]$, it means we will move B_k from $core_j$ to another core of job_i. If there is only on block on $core_j$, we will delete $core_j$ from job_i after moving.

However, in experiment, though we amplify the probability of each move stategy and the iteration times, we can't get a better result than that the greedy algorithm gives.

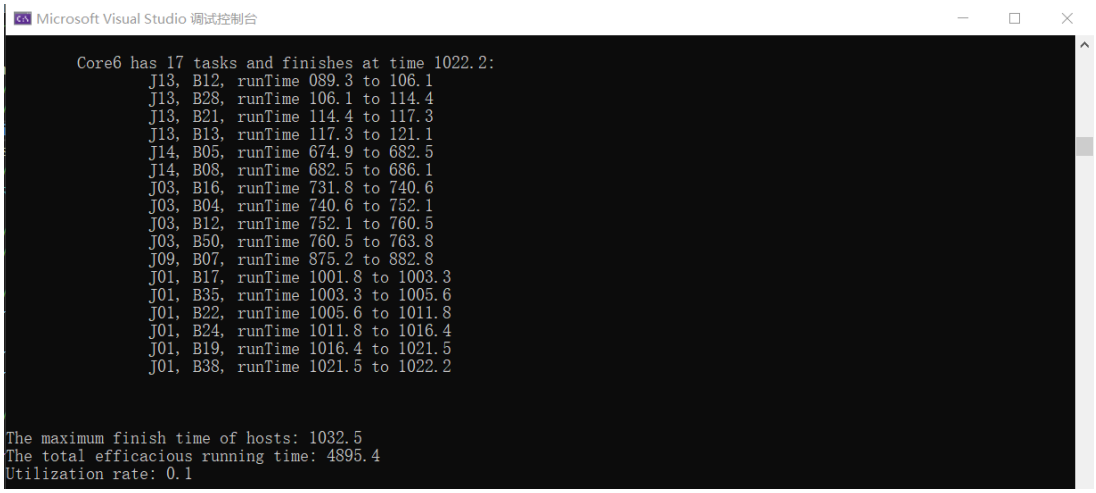

图 8. The result of Task2 given by completely random move strategy

This is because each movement of this strategy is too small. If we change the Simulated Annealing Algorithm, let it only accepts the state with lower energy, the result will be improved a little. But it is still much worse than the greedy.

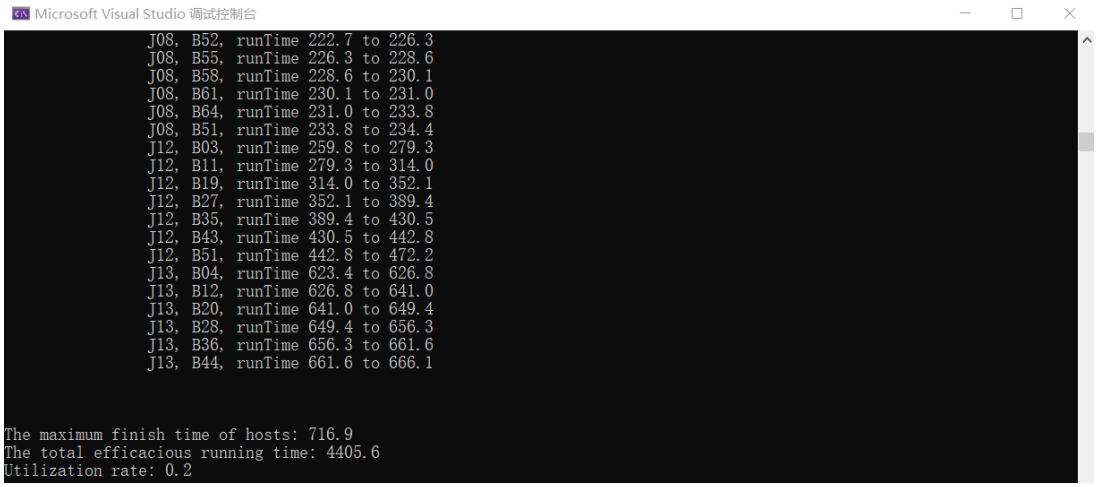

图 9. The result of Task2 given by completely random move strategy if SAA only accepts states with lower energy

If we apply this changed SAA on the greedy version, sometimes it can give a better solution than the average result of the original SAA. But it can hardly give a solution better than the best result the original SAA ever gives. The reason is that it may cause the state to stay at the locally optimal solution. When the solution is good enough, it is hard for the changed SAA to move anymore.

7 Summary

In this article, we focus on designing an algorithm for solving resource scheduling problems in Hadoop. The algorithm we design is based on simulated annealing algorithm and greedy algorithm to give an approximate solution to the problems of both simplified and comprehensive versions. We also do some experiments on parameters in the algorithm but the results we get are for the special test cases and not universal. During analyzing the complexity and other quality of our algorithm, we found that the problem is NP-complete. Though we hope our greedy algorithm is optimal for one job in the simplified version, it is an NP-complete problem, we can’ t find its optimal solution in polynomial time. So we compare the greedy solution to the theoretical optimal solution to prove the greedy one is good enough. Finally, we use a completely random move strategy to try to find a better solution for the comprehensive version. But the result is that it can’ t work better than the greedy.

In general, the SACG is our feasible but imperfect answer to resource scheduling problem in Hadoop.

Acknowledgements

This problem is motivated from a real-world cloud service corporation. It is formulated by Prof. Xiaofeng Gao (gao-xf@cs.sjtu.edu.cn) from Department of Computer Science and Engineering at Shanghai Jiao Tong University. Yue Ma scribed and modified the project. Yangyang Bao provided the example in Task 1. Zhen Sun provided the example in Task 2. Wanghua Shi provided the

test data and source code. Jiale Zhang, Tianyao Shi helped on proofreading and provided many suggestions.

Thank assistants Fang Hongjie, Lv Jialin and Shi Wanghua. They gave so many instructive suggestions to our group.

Thank professor Gao Xiaofeng. The problem she assigned helps us understand the actual case in real world better and improves our abilities.

Every group member has learned a lot from this project.

Zhenran Xiao: Figuring out this problem is more and more interesting during the whole process. At the beginning, I just feel anxious and nervous. I was worried whether we can give a result. But as the project progressed, I had fun, especially when I drew the picture of the annealing process. I feel the charm of the Simulated Annealing Algorithm.

When I gave out the greedy algorithm used for one job on single host(I need to state I think out this by myself instead of searching it out on the Internet), I actually don't know how to prove whether it is optimal. Finally, we found out that the problem is an NP-complete problem. We can't find a optimal solution in polynomial time.

Really thankful to my groupmates, they did much difficult work in the project and answered many questions I asked. Trying to express everyone's awesome ideas clearly in the article is a challenge as well. I really learned a lot.

Zichun Ye: As a member of the group, my job is implementing the main part of the code, debugging and providing the test answer. During this process, I faced many difficulties including setting the variables, finding errors through hundreds of lines and reading others' code. I took notes of all these variables to make coding easier. In the end, the program runs smoothly without annoying bugs.

I learned that before writing the code, I shall have clearheaded logic, otherwise it' s destructive to modify the code. Writing notes in the code is also essential because your teammates also need to understand what you write.

Thank Xiao Zhenran and Long Mabiao. They supported me a lot and did an excellent job in this project.

Mabiao Long: Since we haven' t learned about the NP problem at first, we learned from other similar problems on the Internet and finally chose the SA algorithm, instead of dynamic programming, to deal with the case that a large amount of data may appear. I believe that learning from others will be a shortcut while facing engineering problems, though it may be a hard problem or even an unsolvable problem now, there may be some tips or thoughts that lead to the correct way.

I worked on two functions, $finishtime()$ and $write_back()$. Based on a concise and effective data structure, it is easier to work out. I also worked on the proof of NP-complete, which deepened my understanding of computational complexity and taught me that some seemingly simple questions can be difficult.

Thanks for my teammates, Zhenran Xiao and Zichun Ye, they inspired me and corrected my mistakes, which taught me a lot during these weeks.

参考文献

1. shiwanghua/SharedFiles, https://github.com/shiwanghua/SharedFiles/tree/Project-CodeDemo. Last accessed 15 May 2022
2. yunzhe99/Hadoop_Scheduling, https://github.com/yunzhe99/Hadoop_Scheduling. Last accessed 11 Dec 2021
3. 模拟退火 (simulated annealing) 算法详解, https://blog.csdn.net/weixin_60737527/article/details/123936955?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-0.pc_relevant_default&spm=1001.2101.3001.4242.1&utm_relevant_index=3. Last accessed 3 Apr 2022
4. Hayes, Brian (March–April 2002), "The Easiest Hard Problem" (PDF), American Scientist, Sigma Xi, The Scientific Research Society, vol. 90, no. 2, pp. 113–117, JSTOR 27857621

Appendix A

表 3. Symbols and Definitions

Symbols	Definitions
n	The number of jobs
m	The number of cores
q	The number of hosts
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.
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
$g(\cdot)$	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
$state_i$	The ith state of the material or solution
$E(i)$	The energy of $state_i$
K	Boltzmann's constant
T	The temperature of current state
T_i	The temperature of $state_i$
T_e	The final temperature
β	The rate of temperature fall
M	The maximum number of iterations
δE	The change of energy between two states
P_{co}	The probability changing two job's order
P_{rc}	The probability re-choosing cores for one job
X	Times to pick out 10 different cores for one job
X_i	Times it needs to pick a new core when it has already picked $i-1$ cores
p_i	The probability of choosing a new core when it has already picked $i-1$ cores
P_{mb}	The probability move blocks for one job

Appendix B

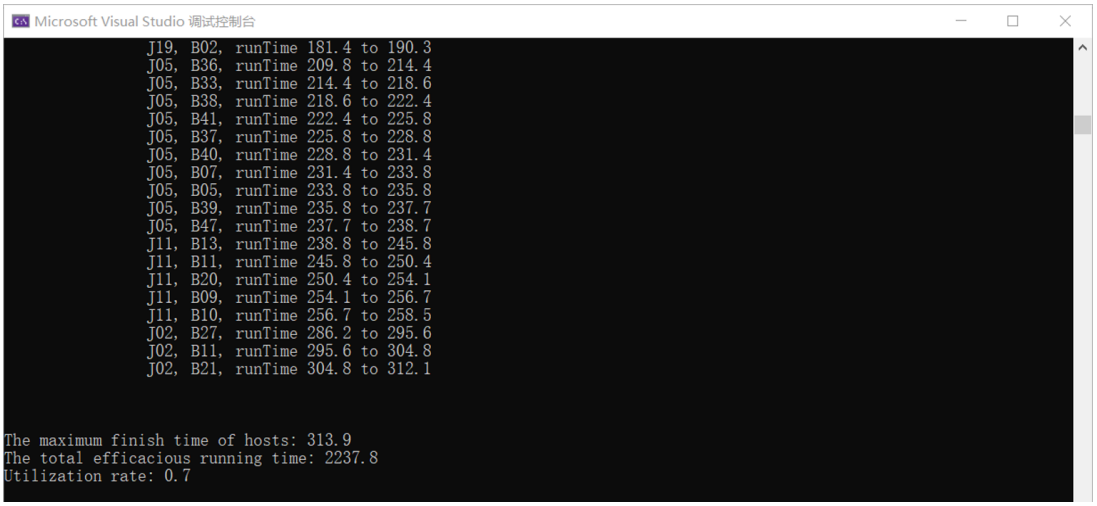

图 10. The best result of Task1 we have ever gotten

The complete output can be checked in `task1_case1_01.txt`.

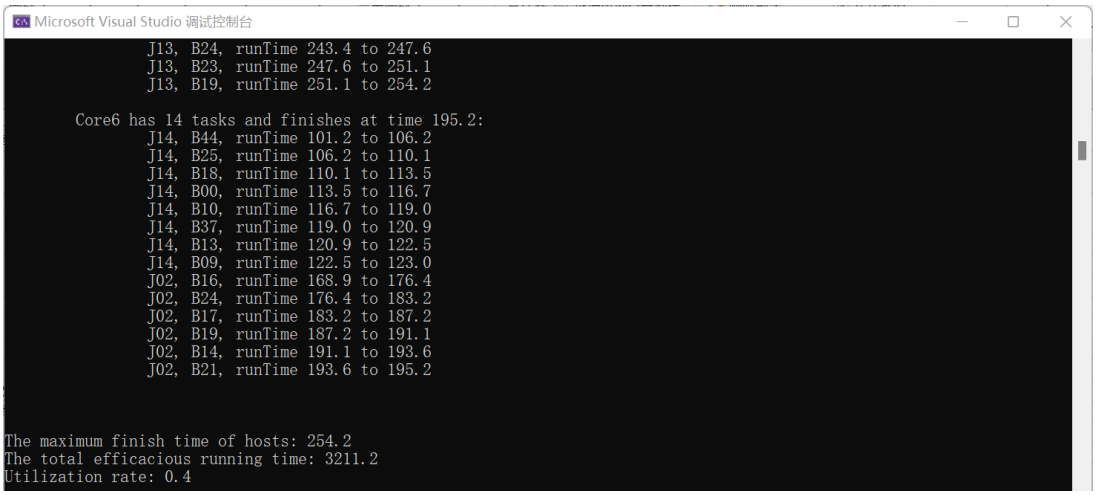

图 11. The best result of Task2 we have ever gotten

The complete output can be checked in `task2_case1_01 (2).txt`.