EECE7205 Project2

MCC Task Scheduling

Jingci Wang

MS in Data Science
College of Computer and Information Science
Northeastern University

December 3, 2018

Initial Scheduling Algorithm

Target

- **Primary assignment**: For each task, find the execution location which achieves its least running time
- **Task prioritizing**
- **Execution unit**: Consider the effects of the already scheduled tasks

Kernel Algorithm

Input: v_{tar}, k_{tar}
Output: T_{new}^{total}, E_{new}^{total}

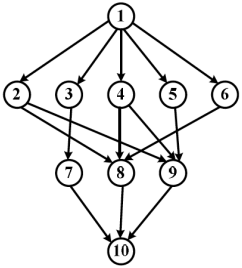
1: Remove v_{tar} from the original core
2: Insert v_{tar} into the right position on new location
3: Calculate new time and energy cost

Task Migration Algorithm

Input: T^{max}, Current Schedule
Output: New Schedule

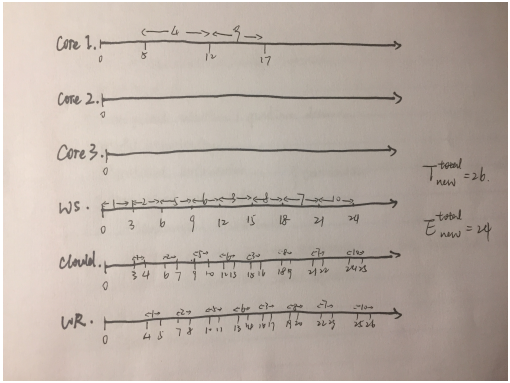
1: **while** $\delta E > 0$ && $T^{new} < T^{max}$ **do**
2: **for** v_{tar} *in* S_{local} **do**
3: **for** k in 0 to K **do**
4: $kernel(v_{tar}, k_{tar})$
5: **end for**
6: Get several migration option
7: Update schedule
8: **end while**

Examples



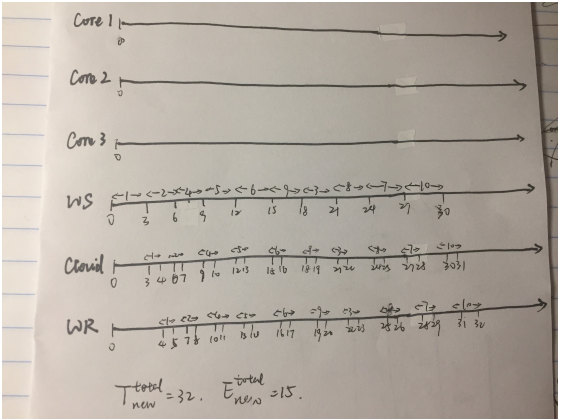
Task	Core1	Core2	Core3
1	9	7	5
2	8	6	5
3	6	5	4
4	7	5	3
5	5	4	2
6	7	6	4
7	8	5	3
8	6	4	2
9	5	3	2
10	7	4	2

$$1 \le i \le N, \begin{cases} T_i^s = 3 \\ T_i^c = 1 \\ T_i^r = 1 \end{cases}$$



Examples

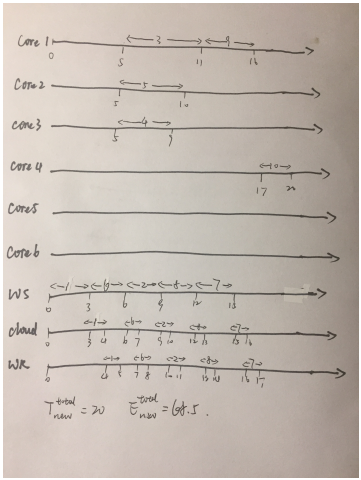
Set $T^{max} = 100$



Examples

Task	Core1	Core2	Core3	Core4	Core5	Core6
1	9	7	5	4	2	1
2	8	6	5	3	2	1
3	6	5	4	3	2	1
4	7	5	4	3	2	1
5	6	5	4	3	2	1
6	7	6	5	4	3	1
7	8	6	5	3	2	1
8	6	5	4	3	2	1
9	5	3	3	2	2	1
10	7	6	4	3	2	1

Examples



Thanks!

EECE7205 Project Report
Implementation of MCC Task Scheduling Algorithm

Jingci Wang

College of Computer and Information Science, Northeastern University

Introduction

In this paper, the implementation of MCC task scheduling algorithm proposed in [1] would be implemented using C++.

1 MCC Task Scheduling

Since [1] dose not present the whole process pseudo code explicitly, this paper would roughly state the one by one step in accord with the later implementation in each block. **The notation used in this report act in unison with [1]**.

1.1 Step One: Initial Scheduling Algorithm

$*$ Primary assignment

Algorithm 1 Primary assignment

Input:

The execution time on local cores: $T_{i,k}^{l}$, for $i = 1, \ldots, N$, and $k = 1, \ldots, K$
The time of sending task to the cloud: T_i^s, for $i = 1, \ldots, N$
The execution time on cloud: T_i^c, for $i = 1, \ldots, N$
The time of receiving task from the cloud: T_i^r, for $i = 1, \ldots, N$

Output:

The preliminary executing location for each task (whether on local core or cloud)

1: **for** v_i in V **do**
2: $T_i^{l,min} = \min_{1 \le k \le K} T_{i,k}^{l}$
3: $T_i^{re} = T_i^s + T_i^c + T_i^r$
4: **if** $T_i^{re} < T_i^{l,min}$ **then**
5: $v_i.cloud = true$

6: **end if**
7: **end for**

∗ Task prioritizing

Algorithm 2 Task prioritizing

Input:
The execution time on local cores: $T^l_{i,k}$, for $i = 1, \ldots, N$, and $k = 1, \ldots, K$
The time of sending task to the cloud: T^s_i, for $i = 1, \ldots, N$
The execution time on cloud: T^c_i, for $i = 1, \ldots, N$
The time of receiving task from the cloud: T^r_i, for $i = 1, \ldots, N$
Output:
The computation cost w_i, for $i = 1, \ldots, N$
The priority level $priority(v_i)$, for $i = 1, \ldots, N$
1: **for** v_i in V **do**
2: **if** $v_i.cloud == true$ **then**
3: $w_i = T^s_i + T^c_i + T^r_i$
4: **else**
5: $w_i = avg_{1 \le k \le K} T^l_{i,k}$
6: **end if**
7: **if** $v_i \in exist\ tasks$ **then**
8: $priority(v_i) = w_i$
9: **else**
10: $priority(v_i) = w_i + \max_{v_j \in succ(v_i)} priority(v_j)$
11: **end if**
12: **end for**

∗ Execution unit selection

For the convenience of implementing C++, $v_i.core = K + 1$ stands for that v_i is a cloud task.

Firstly, the process for updating the ready time and finish time for a task on specific location is defined

Algorithm 3 Update-time(v_i, available-time, k)

The execution time on local cores: $T^l_{i,k}$, for $i = 1, \ldots, N$, and $k = 1, \ldots, K$
The time of sending task to the cloud: T^s_i, for $i = 1, \ldots, N$
The execution time on cloud: T^c_i, for $i = 1, \ldots, N$
The time of receiving task from the cloud: T^r_i, for $i = 1, \ldots, N$
Available time for starting execution task v_i
Input/Output:
The ready time RT^l_i, RT^{ws}_i, RT^c_i, RT^{wr}_i, for $i = 1, \ldots, N$
The finish time FT^l_i, FT^{ws}_i, FT^c_i, FT^{wr}_i, for $i = 1, \ldots, N$

if $v_i.cloud$ = true **then**
  $RT_i^{ws} = available - time$
  $FT_i^{ws} = RT_i^{ws} + T_i^s$
  $RT_i^c = FT_i^{ws}$
  $FT_i^c = RT_i^c + T_i^c$
  $RT_i^{wr} = FT_i^c$
  $FT_i^{wr} = RT_i^{wr} + T_i^r$
  return FT_i^{wr}
else
  $RT_i^l = available - time$
  $FT_i^l = RT_i^l + T_{i,k}^l$
  return $FT_{i,k}^l$
end if

Algorithm 4 $Find - earliest - available - time(v_i, k)$

$precedence - satisfied - time$ = latest finish time for $v_j \in prec(v_i)$
if $k == K + 1$ **then**
  ws-available = earliest available time for v_i on ws channel
  return max{precedence-satisfied-time, ws-available}
else
  k-available = earliest available time for v_i on core k
  return max{precedence-satisfied-time, k-available}
end if

Algorithm 5 Execution unit selection

Input:
$priority(v_i)$, for $i = 1, \ldots, N$
The execution time on local cores: $T_{i,k}^l$, for $i = 1, \ldots, N$, and $k = 1, \ldots, K$
The time of sending task to the cloud: T_i^s, for $i = 1, \ldots, N$
The execution time on cloud: T_i^c, for $i = 1, \ldots, N$
The time of receiving task from the cloud: T_i^r, for $i = 1, \ldots, N$
Output:
The location arrangement $v_i.core$ for $i = 1, \ldots, N$
The ready time RT_i^l, RT_i^{ws}, RT_i^c, RT_i^{wr}, for $i = 1, \ldots, N$
The finish time FT_i^l, FT_i^{ws}, FT_i^c, FT_i^{wr}, for $i = 1, \ldots, N$
Initialize $RT_i^l = RT_i^{ws} = RT_i^c = RT_i^{wr} = 0$, for $i = 1, \ldots, N$
Initialize $FT_i^l = FT_i^{ws} = FT_i^c = FT_i^{wr} = 0$, for $i = 1, \ldots, N$
$V' = V$
for $j = 1$ *to* $|V|$ **do**
  $v_i = \arg\max_{v_j}\{priority(v_i), v_i \in V'\}$

$V'.pop(v_i)$
if $v_i.cloud == true$ **then**
$available - time = Find - earliest - available - time(v_i,\ K+1)$
$Update - time(v_i, available - time,\ K+1)$
$v_i.core = K+1$
else
$k^* = \arg\min_k Find - earliest - available - time(v_i, k)$
$v_i.core = k^*$
if $k^* == K+1$ **then**
$v_i.cloud = true$
$Update - time(v_i,\ Find - earliest - available - time(v_i, K+1))$
else
$Update - time(v_i,\ Find - earliest - available - time(v_i, k^*)$
end if
end if
end for

1.2 Step Two: Task Migration Algorithm

$*$ Kernel algorithm

Algorithm 6 kernel(v_{tar}, k_{tar})

Input:
Current-locally scheduled task v_{tar}
Targeted transiting location k_{tar}
Output:
Total time cost after migration T_{new}^{total}
Total energy consumption after migration E_{new}^{total}
$k_{ori} = v_{tar}.core$
$S_{k_{ori}} = S_{k_{ori}} - v_{tar}$
Find $RT(v_{(k_{tar},m)}) \leq RT(v_{tar}) \leq RT(v_{(k_{tar},m+1)})$
$S_{k_{tar}} = \{v_{(k_{tar},1)}, \ldots, v_{(k_{tar},m)}, v_{tar}, v_{(k_{tar},m+1)}, \ldots\}$
$v_{tar}.core = k_{tar}$
Re-schedule the new arrangement of the tasks by task-precedence requirement
calculate the total time cost T_{new}^{total} for new schedule
calculate the total energy cost E_{new}^{total} for new schedule
return T_{new}^{total} and E_{new}^{total}

$*$ Outer loop

Algorithm 7 Outer-loop

Input:
Total time cost for current schedule T^{total}
Total energy consumption for current schedule E^{total}
Maximal acceptable time cost T^{max}
Output:
New schedule with maximal reduction in Energy without violation on time constraint
New total energy consumption

while $\delta E > 0$ **do**
  Define a empty vector K storing migration which leads to energy reduction and satisfy the time constraint
  for v_{tar} in $\{v_i \in V \; for \; v_i.cloud = false\}$ **do**
    for $k = 1, \ldots, K+1$, and $k \neq v_{tar}.core$ **do**
      $kernel(v_{tar}, k)$
      if $E^{total}_{new} < E^{total}$ and $T^{total}_{new} < T^{max}$ **then**
        $K = K \bigcup \{migration(v_{tar}, k)\}$
      end if
    end for
    for $migration \in K$ with $T^{total}_{new} <= T^{total}$ **do**
      Find migration with maximum energy reduction (v_{opt}, k_{opt})
    end for
  end for
  if Find no migration with time cost no exceeding current time cost **then**
    Find migration with maximum $\frac{E^{total} - E^{total}_{new}}{T^{total}_{new} - T^{total}}$
  end if
  $T^{total}_{new} = kernel(v_{opt}, k_{opt}).T^{total}_{new}$
  $E^{total}_{new} = kernel(v_{opt}, k_{opt}).E^{total}_{new}$
  $\delta E = E^{total} - E^{total}_{new}$
  $E^{total} = E^{total}_{new}$
  $T^{total} = T^{total}_{new}$
  Re-schedule the tasks with $migration(v_{opt}, k_{opt})$
end while
return E^{total}, T^{total}

2 Analysis of the running time

According the the task scheduling process described in the previous section, the running time for each phase is as follows.

Primary assignment: $O(N)$

Task prioritizing: $O(N)$

Execution unit selection: $O(N^2 \log(N))$ (Used heapsort to find the task with maximum priority)

kernel: $O(N^2)$

Outer loop: $O(N \times K) \times O(kernel) = O(N^3 \times K)$

3 Results

This section would provide MCC task scheduling results of several input examples.

Example 1

- **Input:**
  The first example reproduces the case in paper [1] with $T^{max} = 27$
- **Output:**
  The task scheduling result after Step I:
  ——— Core1 scheduling——————
  Task4 from time 5 to time 12

  ——— Core2 scheduling——————
  Task6 from time 5 to time 11
  Task8 from time 12 to time 16

  ——— Core3 scheduling——————
  Task1 from time 0 to time 5
  Task3 from time 5 to time 9
  Task5 from time 9 to time 11
  Task7 from time 11 to time 14
  Task9 from time 14 to time 16
  Task10 from time 16 to time 18

  ————Cloud scheduling————
  —-Task2—-
  Wireless sending: from time 5 to time 8
  Could execution: from time 8 to time 9
  Wireless receiving: from time 9 to time10

  ————————Outer loop————————
  Set the upper bound for time cost to be 27

The final task scheduling result (after Step II):
——— Core1 scheduling——————
Task4 from time 5 to time 12
Task9 from time 12 to time 17

——— Core2 scheduling——————
No task scheduled in this core.

——— Core3 scheduling——————
No task scheduled in this core.

————Cloud scheduling—————
—-Task1——
Wireless sending: from time 0 to time 3
Could execution: from time 3 to time 4
Wireless receiving: from time 4 to time5

—-Task2——
Wireless sending: from time 3 to time 6
Could execution: from time 6 to time 7
Wireless receiving: from time 7 to time8

—-Task5——
Wireless sending: from time 6 to time 9
Could execution: from time 9 to time 10
Wireless receiving: from time 10 to time11

—-Task6——
Wireless sending: from time 9 to time 12
Could execution: from time 12 to time 13
Wireless receiving: from time 13 to time14

—-Task3——
Wireless sending: from time 12 to time 15
Could execution: from time 15 to time 16
Wireless receiving: from time 16 to time17

—-Task8——
Wireless sending: from time 15 to time 18
Could execution: from time 18 to time 19
Wireless receiving: from time 19 to time20

—-Task7——
Wireless sending: from time 18 to time 21
Could execution: from time 21 to time 22
Wireless receiving: from time 22 to time23

—-Task10——
Wireless sending: from time 21 to time 24
Could execution: from time 24 to time 25
Wireless receiving: from time 25 to time26
—-This is the presentation of total time————
$T^{total} = 26$
—-This is the presentation of total energy————-
$E^{total} = 24$
The running time for the code is 0.016986s.

Example 2

- **Input:**
  The second example still uses the input in the first one except setting $T^{max} = 100$. The expected outcome should be that all the tasks are offloaded to remote execution.

- **Output:**
  Set the upper bound for time cost to be 100

  ——— Core1 scheduling————
  No task scheduled in this core.

  ——— Core2 scheduling————
  No task scheduled in this core.

  ——— Core3 scheduling————
  No task scheduled in this core.

  ————Cloud scheduling————
  —-Task1——
  Wireless sending: from time 0 to time 3
  Could execution: from time 3 to time 4
  Wireless receiving: from time 4 to time5

  —-Task2——
  Wireless sending: from time 3 to time 6
  Could execution: from time 6 to time 7

Wireless receiving: from time 7 to time8

—-Task4—-
Wireless sending: from time 6 to time 9
Could execution: from time 9 to time 10
Wireless receiving: from time 10 to time11

—-Task5—-
Wireless sending: from time 9 to time 12
Could execution: from time 12 to time 13
Wireless receiving: from time 13 to time14

—-Task6—-
Wireless sending: from time 12 to time 15
Could execution: from time 15 to time 16
Wireless receiving: from time 16 to time17

—-Task9—-
Wireless sending: from time 15 to time 18
Could execution: from time 18 to time 19
Wireless receiving: from time 19 to time20

—-Task3—-
Wireless sending: from time 18 to time 21
Could execution: from time 21 to time 22
Wireless receiving: from time 22 to time23

—-Task8—-
Wireless sending: from time 21 to time 24
Could execution: from time 24 to time 25
Wireless receiving: from time 25 to time26

—-Task7—-
Wireless sending: from time 24 to time 27
Could execution: from time 27 to time 28
Wireless receiving: from time 28 to time29

—-Task10—-
Wireless sending: from time 27 to time 30
Could execution: from time 30 to time 31
Wireless receiving: from time 31 to time32
—-This is the presentation of total time————

$T^{total} = 32$
—-This is the presentation of total energy————-
$E^{total} = 15$
The running time for the code is 0.016986s.

Example 3

- **Input:**
  Set number of heterogeneous cores to be 6
  Set $P_1 = 1$, $P_2 = 2$, $P_3 = 4$, $P_4 = 8$, $P_5 = 16$, $P_6 = 32$, $P_s = 0.5$
  Set the upper bound for time cost to be 20
  The local running times are defined as follows:

Task	Core1	Core2	Core3	Core4	Core5	Core6
1	9	7	5	4	2	1
2	8	6	5	3	2	1
3	6	5	4	3	2	1
4	7	5	4	3	2	1
5	6	5	4	3	2	1
6	7	6	5	4	3	1
7	8	6	5	3	2	1
8	6	5	4	3	2	1
9	5	3	3	2	2	1
10	7	6	4	3	2	1

- **Output:**
  The task scheduling result after Step I:
  ——— Core1 scheduling——————
  No task scheduled in this core.

  ——— Core2 scheduling——————
  No task scheduled in this core.

  ——— Core3 scheduling——————
  No task scheduled in this core.

  ——— Core4 scheduling——————
  Task4 from time 1 to time 4
  Task9 from time 4 to time 6

——— Core5 scheduling———————
Task2 from time 1 to time 3
Task7 from time 3 to time 5

——— Core6 scheduling———————
Task1 from time 0 to time 1
Task6 from time 1 to time 2
Task3 from time 2 to time 3
Task5 from time 3 to time 4
Task8 from time 4 to time 5
Task10 from time 6 to time 7

————Cloud scheduling————————
No task offloaded to cloud.
The task scheduling result after Step II:
——— Core1 scheduling———————
Task3 from time 5 to time 11
Task9 from time 11 to time 16

——— Core2 scheduling———————
Task5 from time 5 to time 10

——— Core3 scheduling———————
Task4 from time 5 to time 9

——— Core4 scheduling———————
Task10 from time 17 to time 20

——— Core5 scheduling———————
No task scheduled in this core.

——— Core6 scheduling———————
No task scheduled in this core.

————Cloud scheduling————————
—-Task1——
Wireless sending: from time 0 to time 3
Could execution: from time 3 to time 4
Wireless receiving: from time 4 to time5

—-Task6——
Wireless sending: from time 3 to time 6

Could execution: from time 6 to time 7
Wireless receiving: from time 7 to time8

—-Task2—-
Wireless sending: from time 6 to time 9
Could execution: from time 9 to time 10
Wireless receiving: from time 10 to time11

—-Task8—-
Wireless sending: from time 9 to time 12
Could execution: from time 12 to time 13
Wireless receiving: from time 13 to time14

—-Task7—-
Wireless sending: from time 12 to time 15
Could execution: from time 15 to time 16
Wireless receiving: from time 16 to time17
—-This is the presentation of total time————
$T^{total} = 20$
—-This is the presentation of total energy————-
$E^{total} = 68.5$
The running time for the code is 0.022928s.

Example 4

- **Input:**
  Set the previous time constraint to be 10, which is close to T^{min}

- **Output:**

  The task scheduling result after Step I:
  ——— Core1 scheduling——————
  No task scheduled in this core.

  ——— Core2 scheduling——————
  No task scheduled in this core.

  ——— Core3 scheduling——————
  No task scheduled in this core.

  ——— Core4 scheduling——————

Task4 from time 1 to time 4
Task9 from time 4 to time 6

——— Core5 scheduling———————
Task2 from time 1 to time 3
Task7 from time 3 to time 5

——— Core6 scheduling———————
Task1 from time 0 to time 1
Task6 from time 1 to time 2
Task3 from time 2 to time 3
Task5 from time 3 to time 4
Task8 from time 4 to time 5
Task10 from time 6 to time 7

————Cloud scheduling————————
No task offloaded to cloud.
The task scheduling result after Step II:

——— Core1 scheduling———————
Task3 from time 1 to time 7

——— Core2 scheduling———————
Task5 from time 1 to time 6
Task9 from time 6 to time 9

——— Core3 scheduling———————
No task scheduled in this core.

——— Core4 scheduling———————
Task4 from time 1 to time 4

——— Core5 scheduling———————
Task2 from time 1 to time 3
Task7 from time 7 to time 9

——— Core6 scheduling———————
Task1 from time 0 to time 1
Task10 from time 9 to time 10

————Cloud scheduling————————
—-Task6——

Wireless sending: from time 1 to time 4
Could execution: from time 4 to time 5
Wireless receiving: from time 5 to time6

—-Task8—-
Wireless sending: from time 4 to time 7
Could execution: from time 7 to time 8
Wireless receiving: from time 8 to time9
—-This is the presentation of total time————
$T^{total} = 10$
—-This is the presentation of total energy————-
$E^{total} = 177$
The running time for the code is 0.019498s.

4 Source Code

```
1
2  #include <iostream>
3  #include <vector>
4  #include <algorithm>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <cmath>
8  using namespace std;
9
10 #define Max_time 999999
11 #define Max_value 999999
12 // define a struct to represent one specific task
13 typedef struct Task{
14     int index;
15     vector<int> prec;
16     vector<int> succ;
17     double w_i;
18     double priority;
19     int core = 0;
20     bool cloud = false;
21     double Energy = 0;
22 }Task;
23 // define the time consumption and energy struct
24 typedef struct Localtime{
25     vector<double> T_k;
26 }Localtime;
27 typedef struct Localenergy{
28     vector<double> E_k;
29 }Localenergy;
30
```

```
typedef struct Cloudtime{
    double T_s;
    double T_c;
    double T_r;
}Cloudtime;
typedef struct Cloudenergy{
    double E_c;
}Cloudenergy;
// define the finish and ready time struct
typedef struct Finishtime{
    double FT_l;
    double FT_ws;
    double FT_c;
    double FT_wr;
}Finishtime;
typedef struct Readytime{
    double RT_l;
    double RT_ws;
    double RT_c;
    double RT_wr;
}Readytime;

// function to order the tasks according to their priorities
vector<Task> max_heapify(vector<Task> V, int i, int heap_size) {
    vector<Task> v = V;
    int l = i * 2 + 1;
    int r = i * 2 + 2;
    int largest;
    if (l <= heap_size - 1 && (v[l].priority > v[i].priority)) {
        largest = l;
    }else{
        largest = i;
    }
    if (r <= heap_size - 1 && (v[r].priority > v[largest].priority)) {
        largest = r;
    }
    if (largest != i) {
        swap(v[i], v[largest]);
        v = max_heapify(v, largest, heap_size);
    }
    return v;
}
vector<Task> build_max_heap(vector<Task> V, int heap_size){
    vector<Task> v = V;
    for (int i = (heap_size - 1) / 2; i >= 0; i--) {
        v = max_heapify(v, i, heap_size);
    }
    return v;
}
// function to check task-precedence requirements
double check_precedence(Task v_i, vector<Finishtime> FT){
```

```
82      vector<int> prec = v_i.prec;
83      double last_finishtime = 0;
84      if (v_i.prec[0] == 0) {
85          return last_finishtime;
86      }
87      if (v_i.cloud == false) {
88          for (int i = 0; i < prec.size(); i++) {
89              int prec_node = prec[i] - 1;
90              double this_finishtime = max(FT[prec_node].FT_wr, FT[prec_node].
    FT_l);
91              if (this_finishtime > last_finishtime) {
92                  last_finishtime = this_finishtime;
93              }
94          }
95      }else {
96          for (int i = 0; i < prec.size(); i++) {
97              int prec_node = prec[i] - 1;
98              double this_finishtime = max(FT[prec_node].FT_ws, FT[prec_node].
    FT_l);
99              if (this_finishtime > last_finishtime) {
100                 last_finishtime = this_finishtime;
101             }
102         }
103     }
104     return last_finishtime;
105 }
106 double check_successors(Task v_i, vector<Readytime> RT, vector<Task> v){
107     vector<int> succ = v_i.succ;
108     double first_readytime = Max_time;
109     if (succ[0] == 0) {
110         return first_readytime;
111     }
112     for (int i = 0; i < succ.size(); i++) {
113         int succ_node = succ[i] - 1;
114         if (v[succ_node].cloud == true) {
115             if (RT[succ_node].RT_ws < first_readytime) {
116                 first_readytime = RT[succ_node].RT_ws;
117             }
118         }else{
119             if (RT[succ_node].RT_l < first_readytime) {
120                 first_readytime = RT[succ_node].RT_l;
121             }
122         }
123
124     }
125     return first_readytime;
126 }
127
128 // Calculate the total energy consumption
129 double E_total(vector<Task> v){
130     double e_total = 0;
```

```
131     for (int i = 0; i < v.size(); i++) {
132         e_total += v[i].Energy;
133     }
134     return e_total;
135 }
136 // Calculate the total time
137 double T_total(vector<Finishtime> FT, vector<Task> v){
138     int t_total = 0;
139     for (int i = 0; i < v.size(); i++) {
140         int id = v[i].index - 1;
141         if (v[i].succ[0] == 0) {
142             if (v[i].cloud == false) {
143                 if (FT[id].FT_l > t_total) {
144                     t_total = FT[id].FT_l;
145                 }
146             }else {
147                 if (FT[id].FT_wr > t_total) {
148                     t_total = FT[id].FT_wr;
149                 }
150             }
151         }
152     }
153     return t_total;
154 }
155 // Order the tasks on the same core
156 void ordertask_local(vector<Task> &S, vector<Finishtime> FT) {
157     for (int j = 1; j < S.size(); j++) {
158         Task key = S[j];
159         int i = j - 1;
160         int key_ind = S[j].index - 1;
161         int ele_ind = S[i].index - 1;
162         while (i >= 0 && FT[ele_ind].FT_l > FT[key_ind].FT_l) {
163             swap(S[i + 1], S[i]);
164             i = i - 1;
165             ele_ind = S[i].index - 1;
166         }
167         swap(S[i + 1], key);
168     }
169 }
170 void ordertask_cloud(vector<Task> &S, vector<Finishtime> FT) {
171     for (int j = 1; j < S.size(); j++) {
172         Task key = S[j];
173         int i = j - 1;
174         int key_ind = S[j].index - 1;
175         int ele_ind = S[i].index - 1;
176         while (i >= 0 && FT[ele_ind].FT_wr > FT[key_ind].FT_wr) {
177             swap(S[i + 1], S[i]);
178             i = i - 1;
179             ele_ind = S[i].index - 1;
180         }
181         swap(S[i + 1], key);
```

```
182     }
183 }
184 // define a struct to store the return value of kernel function
185 typedef struct T_E_ori_tar{
186     double T_new_total;
187     double E_new_total;
188     int tar;
189     int k_tar;
190     vector<Finishtime> ft;
191     vector<Readytime> rt;
192 }T_E_ori_tar;
193 
194 // define a class to perform task scheduling
195 class TaskSchedule {
196 public:
197     int K;
198     int N;
199     double R_s;
200     double R_r;
201     double P_s;
202     double Wireless_sending;
203     vector<double> Core_operating;
204     vector<double> f_k;
205     vector<double> P_k;
206     vector<double> alpha_k;
207     vector<double> gamma_k;
208     vector<Task> v;
209     vector<Localtime> T_l;
210     vector<Localenergy> E_l;
211     vector<Cloudtime> T_cloud;
212     vector<Cloudenergy> E_cloud;
213     vector<Finishtime> FT;
214     vector<Readytime> RT;
215     double T_max;
216     TaskSchedule(int K, double R_s, double R_r, double P_s, vector<double> f_k
        , vector<double> alpha_k, vector<double> gamma_k, vector<Task> v, vector<
        Localtime> T_l, vector<Cloudtime> T_cloud){
217         this->N = (int)v.size();
218         this->K = K;
219         this->R_s = R_s;
220         this->R_r = R_r;
221         this->P_s = P_s;
222         this->T_max = Max_time;
223         this->Wireless_sending = 0;
224         vector<double> core_operating(K, 0);
225         this->Core_operating = core_operating;
226         this->f_k = f_k;
227         this->alpha_k = alpha_k;
228         this->gamma_k = gamma_k;
229         this->v = v;
230         this->T_l = T_l;
```

```
231         this->T_cloud = T_cloud;
232         for (int i = 0; i < K; i++) {
233             this->P_k.push_back(alpha_k[i] * pow(f_k[i], gamma_k[i]));
234         }
235         for (int i = 0; i < T_l.size(); i++) {
236             Localenergy E_l_taski;
237             for (int k = 0; k < K; k++) {
238                 E_l_taski.E_k.push_back(this->P_k[k] * T_l[i].T_k[k]);
239             }
240             this->E_l.push_back(E_l_taski);
241         }
242         for (int i = 0; i < T_cloud.size(); i++) {
243             Cloudenergy E_c_taski;
244             E_c_taski.E_c = P_s * T_cloud[i].T_s;
245             this->E_cloud.push_back(E_c_taski);
246         }
247 
248         for (int i = 0; i < N; i++) {
249             Finishtime FT_i;
250             Readytime RT_i;
251             FT_i.FT_l = 0;
252             FT_i.FT_ws = 0;
253             FT_i.FT_c = 0;
254             FT_i.FT_wr = 0;
255             this->FT.push_back(FT_i);
256             RT_i.RT_l = 0;
257             RT_i.RT_ws = 0;
258             RT_i.RT_c = 0;
259             RT_i.RT_wr = 0;
260             this->RT.push_back(RT_i);
261         }
262     }
263     // Step 1
264     // Phase I: Primay assignment
265     void primary_assign(){
266         for (int i = 0; i < N; i++) {
267             double T_l_min = *min_element(T_l[i].T_k.begin(), T_l[i].T_k.end()
    );
268             double T_re = T_cloud[i].T_s + T_cloud[i].T_c + T_cloud[i].T_r;
269             if (T_re < T_l_min) {
270                 v[i].cloud = true;
271                 v[i].core = this->K;
272                 v[i].w_i = T_re;
273                 v[i].Energy = this->E_cloud[i].E_c;
274             }else{
275                 double wi = 0;
276                 for (int k = 0; k < K; k++) {
277                     wi = wi + T_l[i].T_k[k];
278                 }
279                 v[i].w_i = wi / K;
280             }
```

```
281         }
282     }
283     // Phase II: Task prioritizing
284     void task_prior(){
285         for (int i = N - 1; i >= 0; i--) {
286             double max = 0;
287             for (int j = 0; j < v[i].succ.size(); j++) {
288                 if (v[i].succ[j] == 0) {
289                     max = 0;
290                     break;
291                 }
292                 int succ_index = v[i].succ[j] - 1;
293                 if (v[succ_index].priority > max) {
294                     max = v[succ_index].priority;
295                 }
296             }
297             v[i].priority = v[i].w_i + max;
298         }
299     }
300     // find the schedule
301     vector<vector<Task>> schedule() {
302         vector<vector<Task>> S;
303         // push local tasks into S
304         for (int k = 0; k < this->K; k++) {
305             vector<Task> S_k;
306             for (int i = 0; i < this->v.size(); i++) {
307                 if (this->v[i].core == k) {
308                     S_k.push_back(this->v[i]);
309                 }
310             }
311             ordertask_local(S_k, this->FT);
312             S.push_back(S_k);
313         }
314         // push cloud tasks into S
315         vector<Task> S_c;
316         for (int i = 0; i < this->v.size(); i++) {
317             if (this->v[i].cloud == true) {
318                 S_c.push_back(this->v[i]);
319             }
320         }
321         ordertask_cloud(S_c, this->FT);
322         S.push_back(S_c);
323         return S;
324     }
325     // Phase III: Execution unit selection
326     void execution_unit_selection(){
327         vector<Task> V = v;
328         for (int i = 0; i < N; i++) {
329             V = build_max_heap(V, N - i);
330             Task v_i = V[0];
331             swap(V[0], V[N - i - 1]);
```

```
332             int key = v_i.index - 1;
333             double available_time = check_precedence(v_i, FT);
334             if (v_i.cloud == true) {
335                 if (available_time <= this->Wireless_sending) {
336                     RT[key].RT_ws = this->Wireless_sending;
337                     this->Wireless_sending = T_cloud[key].T_s + this->
    Wireless_sending;
338                 }else{
339                     RT[key].RT_ws = available_time;
340                     this->Wireless_sending = T_cloud[key].T_s + available_time
    ;
341                 }
342                 FT[key].FT_ws = Wireless_sending;
343                 RT[key].RT_c = FT[key].FT_ws;
344                 FT[key].FT_c = RT[key].RT_c + T_cloud[key].T_c;
345                 RT[key].RT_wr = FT[key].FT_c;
346                 FT[key].FT_wr = FT[key].FT_c + T_cloud[key].T_r;
347                 this->v[key].Energy = this->E_cloud[key].E_c;
348 
349             }else{
350                 double min_finishtime = Max_time;
351                 int core = 0;
352                 double RT_k = 0;
353                 for (int k = 0; k < K; k++) {
354                     double core_available = Core_operating[k];
355                     double ft;
356                     if (available_time <= core_available) {
357                         ft = core_available + T_l[key].T_k[k];
358                         if (ft < min_finishtime) {
359                             min_finishtime = ft;
360                             core = k;
361                             RT_k = core_available;
362                         }
363                     }else {
364                         ft = available_time + T_l[key].T_k[k];
365                         if (ft < min_finishtime) {
366                             min_finishtime = ft;
367                             core = k;
368                             RT_k= available_time;
369                         }
370                     }
371                 }
372                 double if_cloud_RT_ws;
373                 double if_cloud_RT_c;
374                 double if_cloud_RT_wr;
375                 double if_cloud_FT_ws;
376                 double if_cloud_FT_c;
377                 double if_cloud_FT_wr;
378                 if (Wireless_sending > available_time) {
379                     if_cloud_RT_ws = Wireless_sending;
380                     if_cloud_FT_ws = if_cloud_RT_ws + T_cloud[key].T_s;
```

```
                    if_cloud_RT_c = if_cloud_FT_ws;
                    if_cloud_FT_c = if_cloud_RT_c + T_cloud[key].T_c;
                    if_cloud_RT_wr = if_cloud_FT_c;
                    if_cloud_FT_wr = if_cloud_FT_c + T_cloud[key].T_r;
                }else{
                    if_cloud_RT_ws = available_time;
                    if_cloud_FT_ws = if_cloud_RT_ws + T_cloud[key].T_s;
                    if_cloud_RT_c = if_cloud_FT_ws;
                    if_cloud_FT_c = if_cloud_RT_c + T_cloud[key].T_c;
                    if_cloud_RT_wr = if_cloud_FT_c;
                    if_cloud_FT_wr = if_cloud_FT_c + T_cloud[key].T_r;
                }
                if (if_cloud_FT_wr < min_finishtime) {
                    this->v[key].cloud = true;
                    this->v[key].core = this->K;
                    this->v[key].Energy = this->E_cloud[key].E_c;
                    RT[key].RT_c = if_cloud_RT_c;
                    RT[key].RT_ws = if_cloud_RT_ws;
                    RT[key].RT_wr = if_cloud_RT_wr;
                    FT[key].FT_ws = if_cloud_FT_ws;
                    FT[key].FT_c = if_cloud_FT_c;
                    FT[key].FT_wr = if_cloud_FT_wr;
                    Wireless_sending = FT[key].FT_ws;
                }else {
                    RT[key].RT_l = RT_k;
                    this->v[key].core = core;
                    this->v[key].Energy = this->E_l[key].E_k[core];
                    FT[key].FT_l = min_finishtime;
                    this->Core_operating[core] = min_finishtime;
                }
            }
        }
    }
    // Set T_max
    void set_T_max(double T_max) {
        this->T_max = T_max;
    }
    //Step 2
    // Kernel Algorithm

    // re-initialize ready and finish time
    void initial_readytime(){
        for (int i = 0; i < this->RT.size(); i++) {
            this->RT[i].RT_l = 0;
            this->RT[i].RT_ws = 0;
            this->RT[i].RT_c = 0;
            this->RT[i].RT_wr = 0;
        }
    }
    void initial_finishtime(){
        for (int i = 0; i < this->FT.size(); i++) {
```

```
            this->FT[i].FT_l = 0;
            this->FT[i].FT_ws = 0;
            this->FT[i].FT_c = 0;
            this->FT[i].FT_wr = 0;
        }
    }
    // kernel algo
    T_E_ori_tar kernel(int tar, int k_tar){
        T_E_ori_tar kernel_return;
        vector<vector<Task>> Schedule = this->schedule();
        vector<Task> V = this->v;
        int k_ori = V[tar].core;
        // update the location and energy variable in v_tar struct
        V[tar].core = k_tar;
        if (k_tar == this->K) {
            V[tar].cloud = true;
            V[tar].Energy = this->E_cloud[tar].E_c;
        }else {
            V[tar].Energy = this->E_l[tar].E_k[k_tar];
        }

        double RT_tar = check_precedence(V[tar], this->FT);
        // erase v_tar from S_ori
        vector<Task>::iterator iter;
        for (iter = Schedule[k_ori].begin(); iter != Schedule[k_ori].end();) {
            if (iter->index == tar + 1) {
                iter = Schedule[k_ori].erase(iter);
            }else {
                ++iter;
            }
        }
        // insert v_tar to new location on S_tar
        if (V[tar].prec[0] == 0) {
            Schedule[k_tar].insert(Schedule[k_tar].begin(), V[tar]);
        }else if ((int) Schedule[k_tar].size() == 0){
            Schedule[k_tar].push_back(V[tar]);
        }else if ((int) Schedule[k_tar].size() == 1) {
            if (k_tar == this->K) {
                if (RT_tar <= this->RT[Schedule[k_tar][0].index - 1].RT_ws) {
                    Schedule[k_tar].insert(Schedule[k_tar].begin(), V[tar]);
                }else {
                    Schedule[k_tar].push_back(V[tar]);
                }
            }else {
                if (RT_tar <= this->RT[Schedule[k_tar][0].index - 1].RT_l) {
                    Schedule[k_tar].insert(Schedule[k_tar].begin(), V[tar]);
                }else {
                    Schedule[k_tar].push_back(V[tar]);
                }
            }
        }else {
```

```
483             bool middle = false;
484             for (int i = 0; i < (int)Schedule[k_tar].size() - 1; i++) {
485                 int prec_id = Schedule[k_tar][i].index - 1;
486                 int succ_id = Schedule[k_tar][i + 1].index - 1;
487                 if (k_tar == this->K) {
488                     if (this->RT[prec_id].RT_ws <= RT_tar && RT_tar <= this->
    RT[succ_id].RT_ws) {
489                         middle = true;
490                         if (this->RT[prec_id].RT_ws == RT_tar && V[tar].w_i >
    V[prec_id].w_i) {
491                             Schedule[k_tar].insert(Schedule[k_tar].begin() + i
    , V[tar]);
492                         }else if (this->RT[succ_id].RT_ws == RT_tar && V[tar].
    w_i < V[succ_id].w_i) {
493                             Schedule[k_tar].insert(Schedule[k_tar].begin() + i
     + 2, V[tar]);
494                         }else{
495                             Schedule[k_tar].insert(Schedule[k_tar].begin() + i
     + 1, V[tar]);
496                         }
497                         break;
498                     }
499                 }else {
500                     if (this->RT[prec_id].RT_l <= RT_tar && this->RT[succ_id].
    RT_l >= RT_tar) {
501                         middle = true;
502                         if (this->RT[prec_id].RT_l == RT_tar && V[tar].w_i > V
    [prec_id].w_i) {
503                             Schedule[k_tar].insert(Schedule[k_tar].begin() + i
    , V[tar]);
504                         }else if(this->RT[succ_id].RT_l == RT_tar && V[tar].
    w_i < V[succ_id].w_i){
505                             Schedule[k_tar].insert(Schedule[k_tar].begin() + i
     + 2, V[tar]);
506                         }else{
507                             Schedule[k_tar].insert(Schedule[k_tar].begin() + i
     + 1, V[tar]);
508                         }
509                         break;
510                     }
511                 }
512                 }
513             if (middle == false) {
514                 Schedule[k_tar].push_back(V[tar]);
515             }
516         }
517         /*
518         // check schedule
519         cout << "This is to check the schedule in each core after transition:
    " << endl;
520         for (int i = 0; i <= this->K; i++) {
```

```
521             cout << "Core" << i << " ";
522             for (int k = 0; k < Schedule[i].size(); k++) {
523                 cout << Schedule[i][k].index;
524             }
525             cout << endl;
526         }
527         */
528         // re-schedule
529         // initialize ready1 and ready2
530         vector<int> ready1;
531         vector<int> ready2;
532         for (int i = 0; i < V.size(); i++) {
533             if (V[i].prec[0] == 0) {
534                 ready1.push_back(0);
535             }else {
536                 ready1.push_back((int)V[i].prec.size());
537             }
538             int k_i = V[i].core;
539             int operating_core = 0;
540             for (int m = 0; m < Schedule[k_i].size(); m++) {
541                 if (Schedule[k_i][m].index == V[i].index) {
542                     break;
543                 }
544                 operating_core++;
545             }
546             ready2.push_back(operating_core);
547         }
548 
549         // initialize new tables recording ready times and finish times for
    new schedule
550         vector<Finishtime> ft;
551         vector<Readytime> rt;
552         for (int i = 0; i < V.size(); i++) {
553             Finishtime fti = {0, 0, 0, 0};
554             Readytime rti = {0, 0, 0, 0};
555             ft.push_back(fti);
556             rt.push_back(rti);
557         }
558         // schedule according to ready1 and ready2
559         vector<Task> readytask;
560         vector<vector<int>> scheduled_arrange;
561         // set a table to record those already been scheduled tasks on each
    core
562         for (int k = 0; k <= this->K; k++) {
563             vector<int> k_core_scheduled;
564             scheduled_arrange.push_back(k_core_scheduled);
565         }
566         // set a bool vector to record whether a task has been schedule
567         vector<bool> scheduled;
568         for (int i = 0; i < V.size(); i++) {
569             bool vi_scheduled = false;
```

```
570             scheduled.push_back(vi_scheduled);
571         }
572         // push v_i with ready1_i == 0 && ready2_i == 0 into the ready stack
573         for (int i = 0; i < V.size(); i++) {
574             if (ready1[i] == 0 && ready2[i] == 0) {
575                 readytask.push_back(V[i]);
576                 // tag the chosen task so that it won't be pushed into stack
    again
577                 scheduled[i] = true;
578             }
579         }
580         // schedule the tasks in ready stack one by one
581         while (readytask.empty() == false) {
582             Task inprocesstask = readytask[(int)readytask.size() - 1];
583             int location = inprocesstask.core;
584             int task_id = inprocesstask.index - 1;
585             double available_time = check_precedence(inprocesstask, ft);
586             //cout << "insert task" << inprocesstask.index << "and the least
    satisfied time is " << available_time << endl;
587             double pre_finishtime;
588             if (scheduled_arrange[location].empty() == true) {
589                 pre_finishtime = 0;
590             }else{
591                 int pre_arranged_num = (int) scheduled_arrange[location].size
    ();
592                 int pre_id = scheduled_arrange[location][pre_arranged_num - 1]
     - 1;
593                 if (location == this->K) {
594                     pre_finishtime = ft[pre_id].FT_ws;
595                 }else {
596                     pre_finishtime = ft[pre_id].FT_l;
597                 }
598             }
599             if (location == this->K) {
600                 rt[task_id].RT_ws = max(available_time, pre_finishtime);
601                 ft[task_id].FT_ws = rt[task_id].RT_ws + this->T_cloud[task_id
    ].T_s;
602                 rt[task_id].RT_c = ft[task_id].FT_ws;
603                 ft[task_id].FT_c = rt[task_id].RT_c + this->T_cloud[task_id].
    T_c;
604                 rt[task_id].RT_wr = ft[task_id].FT_c;
605                 ft[task_id].FT_wr = rt[task_id].RT_wr + this->T_cloud[task_id
    ].T_r;
606             }else {
607                 rt[task_id].RT_l = max(available_time, pre_finishtime);
608                 ft[task_id].FT_l = rt[task_id].RT_l + this->T_l[task_id].T_k[
    location];
609             }
610 
611             // record the scheduled task on the specific core
612             scheduled_arrange[location].push_back(inprocesstask.index);
```

```
613             //update vector ready1 and ready2
614             vector<int> succ_task = inprocesstask.succ;
615             if (succ_task[0] != 0) {
616                 for (int i = 0; i < succ_task.size(); i++) {
617                     int suc_id = succ_task[i] - 1;
618                     ready1[suc_id] = ready1[suc_id] - 1;
619                 }
620             }
621             if (Schedule[location][Schedule[location].size() - 1].index - 1 !=
    task_id) {
622                 for (int i = (int)Schedule[location].size() - 1; i >= 0; i--)
    {
623                     if (Schedule[location][i].index - 1 == task_id) {
624                         break;
625                     }else{
626                         int s_succ_id = Schedule[location][i].index - 1;
627                         ready2[s_succ_id] = ready2[s_succ_id] - 1;
628                     }
629                 }
630             }
631             // pop the scheduled task from ready task stack
632             readytask.pop_back();
633             for (int i = 0; i < ready1.size(); i++) {
634                 if (ready1[i] == 0 && ready2[i] == 0 && scheduled[i] == false)
    {
635                     readytask.push_back(V[i]);
636                     scheduled[i] =true;
637                 }
638             }
639         } // end task-in-stack while loop
640         double T_new_total = T_total(ft, V);
641         double E_new_total = E_total(V);
642         kernel_return = {T_new_total, E_new_total, tar, k_tar, ft, rt};
643         return kernel_return;
644     }
645     // Outer Loop
646     void outer_loop() {
647         double T_tot = T_total(this->FT, this->v);
648         double E_tot = E_total(this->v);
649         double D_E = Max_value;
650         int g = 1;
651         while (D_E > 0) {
652             vector<Task> S_local;
653             vector<T_E_ori_tar> kerneloutcome;
654             double T_new = T_tot;
655             double E_new = E_tot;
656             for (int i = 0; i < this->v.size(); i++) {
657                 if (this->v[i].cloud == false) {
658                     S_local.push_back(this->v[i]);
659                 }
660             }
```

```
661         /*    cout << "Round" << g << ": local tasks are: ";
662             for (int i = 0; i < S_local.size(); i++) {
663                 cout << S_local[i].index << " ";
664             }
665             cout << endl; */
666             g++;
667            // cout <<  "Total time cost is  " << T_tot << endl;;
668             for (int i = 0; i < S_local.size(); i++) {
669                 int id = S_local[i].index - 1;
670                 int k_ori = S_local[i].core;
671                 for (int k = 0; k <= this->K; k++) {
672                     if (k != k_ori) {
673                         T_E_ori_tar ker = kernel(id, k);
674                      //    cout << "task " << id + 1 << " migrate from " <<
    k_ori + 1 << " to " << k + 1 << ", t_total = " << ker.T_new_total << " and
     e_total = " << ker.E_new_total << endl;
675                         if (ker.T_new_total <= this->T_max) {
676                         //    cout << "the migration not exceed t_max is task
    :" << this->v[id].index << " to " << k << endl;
677                             kerneloutcome.push_back(ker);
678                         }
679                     }
680                 }
681             }
682             if (kerneloutcome.empty() == true) {
683              //   cout << "Any migration will cause total time exceeding T_max
     " << endl;
684                 break;
685             }
686
687             // check if any migration can reduce E (D_E > 0) while keeping
    total time unincreased
688             vector<T_E_ori_tar> no_time_increase;
689             for (int i = 0; i < kerneloutcome.size(); i++) {
690                 if (kerneloutcome[i].T_new_total <= T_tot && (kerneloutcome[i
    ].E_new_total - E_tot) < 0) {
691                     no_time_increase.push_back(kerneloutcome[i]);
692                 }
693             }
694             T_E_ori_tar opt_migration;
695             double max_D_E = 0;
696             double E_reduction;
697             double T_increase;
698             double max_E_T = 0;
699             if (no_time_increase.empty() == false) {
700                 for (int i = 0; i < no_time_increase.size(); i++) {
701                     E_reduction = E_tot - no_time_increase[i].E_new_total;
702                     if (E_reduction > max_D_E) {
703                         opt_migration = no_time_increase[i];
704                         max_D_E = E_reduction;
705                     }
```

```
706                 }
707             }else{
708                 for (int i = 0; i < kerneloutcome.size(); i++) {
709                     E_reduction = E_tot - kerneloutcome[i].E_new_total;
710                     T_increase = kerneloutcome[i].T_new_total - T_tot;
711                     if ((E_reduction / T_increase) > max_E_T) {
712                         opt_migration = kerneloutcome[i];
713                         max_E_T = E_reduction / T_increase;
714                     }
715                 }
716             }
717             // cout << "this round, the chosen task for migration is: " <<
    opt_migration.tar + 1 << " to " << opt_migration.k_tar + 1 << endl;
718             E_new = opt_migration.E_new_total;
719             T_new = opt_migration.T_new_total;
720             D_E = E_tot - E_new;
721           //  cout << " Reduction of Energy: " << D_E << endl;
722             if (D_E > 1e-3) {
723                 E_tot = E_new;
724                 T_tot = T_new;
725                 int opt_tar = opt_migration.tar;
726                 int opt_k_tar = opt_migration.k_tar;
727                 int opt_k_ori = this->v[opt_tar].core;
728                 vector<vector<Task>> Schedule = this->schedule();
729 
730                 // update the location and energy variable in v_tar struct
731                 this->v[opt_tar].core = opt_k_tar;
732                 //double old_e = this->v[opt_tar].Energy;
733                 if (opt_k_tar == this->K) {
734                     this->v[opt_tar].cloud = true;
735                     this->v[opt_tar].Energy = this->E_cloud[opt_k_tar].E_c;
736                 } else {
737                     this->v[opt_tar].Energy = this->E_l[opt_tar].E_k[opt_k_tar
    ];
738                 }
739                 //cout << "Energy of task" << opt_tar + 1 << " is changed from
     " << old_e << " to " << this->v[opt_tar].Energy << endl;
740                 double RT_tar = check_precedence(this->v[opt_tar], this->FT);
741                 // erase v_tar from S_ori
742                 vector<Task>::iterator iter;
743                 for (iter = Schedule[opt_k_ori].begin(); iter != Schedule[
    opt_k_ori].end();) {
744                     if (iter->index == opt_tar + 1) {
745                         iter = Schedule[opt_k_ori].erase(iter);
746                     }else {
747                         ++iter;
748                     }
749                 }
750                 // insert v_tar to new location on S_tar
751                 if (this->v[opt_tar].prec[0] == 0) {
```

```
752                     Schedule[opt_k_tar].insert(Schedule[opt_k_tar].begin(),
    this->v[opt_tar]);
753                 }else if ((int) Schedule[opt_k_tar].size() == 0){
754                     Schedule[opt_k_tar].push_back(this->v[opt_tar]);
755                 }else if ((int) Schedule[opt_k_tar].size() == 1) {
756                     if (opt_k_tar == this->K) {
757                         if (RT_tar <= this->RT[Schedule[opt_k_tar][0].index -
    1].RT_ws) {
758                             Schedule[opt_k_tar].insert(Schedule[opt_k_tar].
    begin(), this->v[opt_tar]);
759                         }else {
760                             Schedule[opt_k_tar].push_back(this->v[opt_tar]);
761                         }
762                     }else {
763                         if (RT_tar <= this->RT[Schedule[opt_k_tar][0].index -
    1].RT_l) {
764                             Schedule[opt_k_tar].insert(Schedule[opt_k_tar].
    begin(), this->v[opt_tar]);
765                         }else {
766                             Schedule[opt_k_tar].push_back(this->v[opt_tar]);
767                         }
768                     }
769                 }else {
770                     bool middle = false;
771                     for (int i = 0; i < (int)Schedule[opt_k_tar].size() - 1; i
    ++) {
772                         int prec_id = Schedule[opt_k_tar][i].index - 1;
773                         int succ_id = Schedule[opt_k_tar][i + 1].index - 1;
774                         if (opt_k_tar == this->K) {
775                             if (this->RT[prec_id].RT_ws <= RT_tar && RT_tar <=
     this->RT[succ_id].RT_ws) {
776                                 middle = true;
777                                 if (this->RT[prec_id].RT_ws == RT_tar && this
    ->v[opt_tar].w_i > this->v[prec_id].w_i) {
778                                     Schedule[opt_k_tar].insert(Schedule[
    opt_k_tar].begin() + i, this->v[opt_tar]);
779                                 }else if (this->RT[succ_id].RT_ws == RT_tar &&
     this->v[opt_tar].w_i < this->v[succ_id].w_i) {
780                                     Schedule[opt_k_tar].insert(Schedule[
    opt_k_tar].begin() + i + 2, this->v[opt_tar]);
781                                 }else{
782                                     Schedule[opt_k_tar].insert(Schedule[
    opt_k_tar].begin() + i + 1, this->v[opt_tar]);
783                                 }
784                                 break;
785                             }
786                         }else {
787                             if (this->RT[prec_id].RT_l <= RT_tar && this->RT[
    succ_id].RT_l >= RT_tar) {
788                                 middle = true;
```

```
789                                 if (this->RT[prec_id].RT_l == RT_tar && this->
    v[opt_tar].w_i > this->v[prec_id].w_i) {
790                                     Schedule[opt_k_tar].insert(Schedule[
    opt_k_tar].begin() + i, this->v[opt_tar]);
791                                 }else if(this->RT[succ_id].RT_l == RT_tar &&
    this->v[opt_tar].w_i < this->v[succ_id].w_i){
792                                     Schedule[opt_k_tar].insert(Schedule[
    opt_k_tar].begin() + i + 2, this->v[opt_tar]);
793                                 }else{
794                                     Schedule[opt_k_tar].insert(Schedule[
    opt_k_tar].begin() + i + 1, this->v[opt_tar]);
795                                 }
796                                 break;
797                             }
798                         }
799                     }
800                     if (middle == false) {
801                         Schedule[opt_k_tar].push_back(this->v[opt_tar]);
802                     }
803                 }
804                 // re-schedule
805                 // initialize ready1 and ready2
806                 vector<int> ready1;
807                 vector<int> ready2;
808                 for (int i = 0; i < this->v.size(); i++) {
809                     if (this->v[i].prec[0] == 0) {
810                         ready1.push_back(0);
811                     }else {
812                         ready1.push_back((int)this->v[i].prec.size());
813                     }
814                     int k_i = this->v[i].core;
815                     int operating_core = 0;
816                     for (int m = 0; m < Schedule[k_i].size(); m++) {
817                         if (Schedule[k_i][m].index == this->v[i].index) {
818                             break;
819                         }
820                         operating_core++;
821                     }
822                     ready2.push_back(operating_core);
823                 }
824                 // clear ready & finish time storage and re-calculate
825                 this->initial_readytime();
826                 this->initial_finishtime();
827                 // schedule according to ready1 and ready2
828                 vector<Task> readytask;
829                 vector<vector<int>> scheduled_arrange;
830                 // set a table to record those already been scheduled tasks on
    each core
831                 for (int k = 0; k <= this->K; k++) {
832                     vector<int> k_core_scheduled;
833                     scheduled_arrange.push_back(k_core_scheduled);
```

```
834                 }
835                 // set a bool vector to record whether a task has been
    schedule
836                 vector<bool> scheduled;
837                 for (int i = 0; i < this->v.size(); i++) {
838                     bool vi_scheduled = false;
839                     scheduled.push_back(vi_scheduled);
840                 }
841                 // push v_i with ready1_i == 0 && ready2_i == 0 into the ready
     stack
842                 for (int i = 0; i < ready1.size(); i++) {
843                     if (ready1[i] == 0 && ready2[i] == 0) {
844                         readytask.push_back(this->v[i]);
845                         // tag the chosen task so that it won't be pushed into
     stack again
846                         scheduled[i] = true;
847                     }
848                 }
849                 // schedule the tasks in ready stack one by one
850                 while (readytask.empty() == false) {
851                     Task inprocesstask = readytask[(int)readytask.size() - 1];
852                     int location = inprocesstask.core;
853                     int task_id = inprocesstask.index - 1;
854                     double available_time = check_precedence(inprocesstask,
    this->FT);
855                     //calculate the last finish time in the same core
856                     double pre_finishtime;
857                     if (scheduled_arrange[location].empty() == true) {
858                         pre_finishtime = 0;
859                     }else{
860                         int pre_arranged_num = (int) scheduled_arrange[
    location].size();
861                         int pre_id = scheduled_arrange[location][
    pre_arranged_num - 1] - 1;
862                         if (location == this->K) {
863                             pre_finishtime = this->FT[pre_id].FT_ws;
864                         }else {
865                             pre_finishtime = this->FT[pre_id].FT_l;
866                         }
867                     }
868                     if (location == this->K) {
869                         this->RT[task_id].RT_ws = max(available_time,
    pre_finishtime);
870                         this->FT[task_id].FT_ws = this->RT[task_id].RT_ws +
    this->T_cloud[task_id].T_s;
871                         this->RT[task_id].RT_c = this->FT[task_id].FT_ws;
872                         this->FT[task_id].FT_c = this->RT[task_id].RT_c + this
    ->T_cloud[task_id].T_c;
873                         this->RT[task_id].RT_wr = this->FT[task_id].FT_c;
874                         this->FT[task_id].FT_wr = this->RT[task_id].RT_wr +
    this->T_cloud[task_id].T_r;
```

```
875                     }else {
876                         this->RT[task_id].RT_l = max(available_time,
    pre_finishtime);
877                         this->FT[task_id].FT_l = this->RT[task_id].RT_l + this
    ->T_l[task_id].T_k[location];
878                     }
879                     // record the scheduled task on the specific core
880                     scheduled_arrange[location].push_back(inprocesstask.index)
    ;
881                     //update vector ready1 and ready2
882                     vector<int> succ_task = inprocesstask.succ;
883                     if (succ_task[0] != 0) {
884                         for (int i = 0; i < succ_task.size(); i++) {
885                             int suc_id = succ_task[i] - 1;
886                             ready1[suc_id] = ready1[suc_id] - 1;
887                         }
888                     }
889                     if (Schedule[location][Schedule[location].size() - 1].
    index - 1 != task_id) {
890                         for (int i = (int)Schedule[location].size() - 1; i >=
    0; i--) {
891                             if (Schedule[location][i].index - 1 == task_id) {
892                                 break;
893                             }else{
894                                 int s_succ_id = Schedule[location][i].index -
    1;
895                                 ready2[s_succ_id] = ready2[s_succ_id] - 1;
896                             }
897                         }
898                     }
899                     readytask.pop_back();
900                     for (int i = 0; i < ready1.size(); i++) {
901                         if (ready1[i] == 0 && ready2[i] == 0 && scheduled[i]
    == false) {
902                             readytask.push_back(this->v[i]);
903                             scheduled[i] =true;
904                         }
905                     }
906                 } // end of while loop for scheduling task stack
907             }
908         } // end while loop
909     }
910     void local_schedule(vector<Task> core_k, int k) {
911         cout << "---------- Core" << k + 1 <<  " scheduling----------------" <<
    endl;
912         if (core_k.empty() == true) {
913             cout << "No task scheduled in this core." << endl;
914         }else {
915             for (int i = 0; i < core_k.size(); i++) {
916                 int id = core_k[i].index - 1;
```

```
                cout << "Task" << core_k[i].index << " from time " << this->RT[id].RT_l << " to time " << this->FT[id].FT_l << endl;
            }
        }
    }
    void cloud_schedule(vector<Task> core_c) {
        cout << "----------Cloud scheduling----------" << endl;
        if (core_c.empty() == true) {
            cout << "No task offloaded to cloud." << endl;
        }else {
            for (int i = 0; i < core_c.size(); i++) {
                cout << "----Task" << core_c[i].index << "-----" << endl;
                int id = core_c[i].index - 1;
                cout << "Wireless sending: from time " << this->RT[id].RT_ws << " to time " << this->FT[id].FT_ws << endl;
                cout << "Could execution: from time " << this->RT[id].RT_c << " to time " << this->FT[id].FT_c << endl;
                cout << "Wireless receiving: from time " << this->RT[id].RT_wr << " to time" << this->FT[id].FT_wr << endl;
                cout << endl;
            }
        }
    }

    void print_scheduling_result() {
        vector<vector<Task>> Schedule = this->schedule();
        for (int k = 0; k < this->K; k++) {
            this->local_schedule(Schedule[k], k);
            cout << endl;
        }
        this->cloud_schedule(Schedule[this->K]);
    }
};

// check local running time setting
void print_localtime(vector<Localtime> T){
    cout << "Task";
    for (int k = 0; k < T[0].T_k.size(); k++) {
        cout << "\t Core" << k + 1;
    }
    cout << "\n";
    for (int i = 0; i < T.size(); i++) {
        cout <<" " << i + 1;
        for (int k = 0; k < T[i].T_k.size(); k++) {
            cout <<"\t\t " << T[i].T_k[k];
        }
        cout << "\n";
    }
}
// check cloud running time setting
void print_cloudtime(vector<Cloudtime> T){
```

```
964     cout << "Task "  << "T_s " << "T_c " << "T_r" << endl;
965     for (int i = 0; i < T.size(); i++) {
966         cout << i << "\t " << T[i].T_s << "\t " << T[i].T_c << "\t " << T[i].
        T_r ;
967         cout << "\n";
968     }
969 }
970
971 int main() {
972     clock_t begin, end;
973     double t;
974     // initialize the task graph
975     // dependent relationship for the tasks
976     vector<Task> V;
977     Task v1;
978     v1.index = 1;
979     v1.prec = {0};
980     v1.succ = {2, 3, 4, 5, 6};
981     v1.w_i = 0;
982     v1.priority = 0;
983     V.push_back(v1);
984     Task v2;
985     v2.index = 2;
986     v2.prec = {1};
987     v2.succ = {8, 9};
988     v2.w_i = 0;
989     v2.priority = 0;
990     V.push_back(v2);
991     Task v3;
992     v3.index = 3;
993     v3.prec = {1};
994     v3.succ = {7};
995     v3.w_i = 0;
996     v3.priority = 0;
997     V.push_back(v3);
998     Task v4;
999     v4.index = 4;
1000    v4.prec = {1};
1001    v4.succ = {8, 9};
1002    v4.w_i = 0;
1003    v4.priority = 0;
1004    V.push_back(v4);
1005    Task v5;
1006    v5.index = 5;
1007    v5.prec = {1};
1008    v5.succ = {9};
1009    v5.w_i = 0;
1010    v5.priority = 0;
1011    V.push_back(v5);
1012    Task v6;
1013    v6.index = 6;
```

```
    v6.prec = {1};
    v6.succ = {8};
    v6.w_i = 0;
    v6.priority = 0;
    V.push_back(v6);
    Task v7;
    v7.index = 7;
    v7.prec = {3};
    v7.succ = {10};
    v7.w_i = 0;
    v7.priority = 0;
    V.push_back(v7);
    Task v8;
    v8.index = 8 ;
    v8.prec = {2, 4, 6};
    v8.succ = {10};
    v8.w_i = 0;
    v8.priority = 0;
    V.push_back(v8);
    Task v9;
    v9.index = 9;
    v9.prec = {2, 4, 5};
    v9.succ = {10};
    v9.w_i = 0;
    v9.priority = 0;
    V.push_back(v9);
    Task v10;
    v10.index = 10;
    v10.prec = {7, 8, 9};
    v10.succ = {0};
    v10.w_i = 0;
    v10.priority = 0;
    V.push_back(v10);
    // Local running time for each task on each core
    vector<Localtime> T_local;
    Localtime T_k_1;
    T_k_1.T_k = {9, 7, 5};
    Localtime T_k_2;
    T_k_2.T_k = {8, 6, 5};
    Localtime T_k_3;
    T_k_3.T_k = {6, 5, 4};
    Localtime T_k_4;
    T_k_4.T_k = {7, 5, 3};
    Localtime T_k_5;
    T_k_5.T_k = {5, 4, 2};
    Localtime T_k_6;
    T_k_6.T_k = {7, 6, 4};
    Localtime T_k_7;
    T_k_7.T_k = {8, 5, 3};
    Localtime T_k_8;
    T_k_8.T_k = {6, 4, 2};
```

```
    Localtime T_k_9;
    T_k_9.T_k = {5, 3, 2};
    Localtime T_k_10;
    T_k_10.T_k = {7, 4, 2};
    T_local.push_back(T_k_1);
    T_local.push_back(T_k_2);
    T_local.push_back(T_k_3);
    T_local.push_back(T_k_4);
    T_local.push_back(T_k_5);
    T_local.push_back(T_k_6);
    T_local.push_back(T_k_7);
    T_local.push_back(T_k_8);
    T_local.push_back(T_k_9);
    T_local.push_back(T_k_10);
    cout << "The local running times are defined as follows: " << endl;
    print_localtime(T_local);
    // Cloud runnig time on sending, cloud, receiving process for each task
    vector<Cloudtime> T_cloud;
    Cloudtime T_c_1;
    T_c_1.T_s = 3;
    T_c_1.T_c = 1;
    T_c_1.T_r = 1;
    Cloudtime T_c_2;
    T_c_2.T_s = 3;
    T_c_2.T_c = 1;
    T_c_2.T_r = 1;
    Cloudtime T_c_3;
    T_c_3.T_s = 3;
    T_c_3.T_c = 1;
    T_c_3.T_r = 1;
    Cloudtime T_c_4;
    T_c_4.T_s = 3;
    T_c_4.T_c = 1;
    T_c_4.T_r = 1;
    Cloudtime T_c_5;
    T_c_5.T_s = 3;
    T_c_5.T_c = 1;
    T_c_5.T_r = 1;
    Cloudtime T_c_6;
    T_c_6.T_s = 3;
    T_c_6.T_c = 1;
    T_c_6.T_r = 1;
    Cloudtime T_c_7;
    T_c_7.T_s = 3;
    T_c_7.T_c = 1;
    T_c_7.T_r = 1;
    Cloudtime T_c_8;
    T_c_8.T_s = 3;
    T_c_8.T_c = 1;
    T_c_8.T_r = 1;
    Cloudtime T_c_9;
```

```
1116     T_c_9.T_s = 3;
1117     T_c_9.T_c = 1;
1118     T_c_9.T_r = 1;
1119     Cloudtime T_c_10;
1120     T_c_10.T_s = 3;
1121     T_c_10.T_c = 1;
1122     T_c_10.T_r = 1;
1123     T_cloud.push_back(T_c_1);
1124     T_cloud.push_back(T_c_2);
1125     T_cloud.push_back(T_c_3);
1126     T_cloud.push_back(T_c_4);
1127     T_cloud.push_back(T_c_5);
1128     T_cloud.push_back(T_c_6);
1129     T_cloud.push_back(T_c_7);
1130     T_cloud.push_back(T_c_8);
1131     T_cloud.push_back(T_c_9);
1132     T_cloud.push_back(T_c_10);
1133     cout << "The cloud running times are defined as follows: " << endl;
1134     print_cloudtime(T_cloud);
1135     cout << ">>>>>>Example 1<<<<<<<<<<" << endl;
1136     // construct a new task schedule
1137     TaskSchedule TS = TaskSchedule(3, 0, 0, 0.5, {1, 2, 4}, {1, 1, 1}, {1, 1,
        1}, V, T_local, T_cloud);
1138     //cout << "----------primary assignment----------" << endl;
1139     begin = clock();
1140     TS.primary_assign();
1141    // cout << "----------task priority----------" << endl;
1142     TS.task_prior();
1143    /* cout << "The priority relationship:  " << endl;
1144     for (int i = 0; i < TS.v.size(); i++) {
1145         cout << TS.v[i].priority << " " ;
1146     }
1147     for (int i = 0; i < TS.v.size(); i++) {
1148         cout << TS.v[i].cloud << " ";
1149     }
1150     cout << endl;*/
1151    // cout << "----------execution unit selection----------" << endl;
1152
1153     TS.execution_unit_selection();
1154     cout << "The task scheduling result after Step I: " << endl;
1155     TS.print_scheduling_result();
1156
1157     /*
1158     cout << endl;
1159
1160     for (int i = 0; i < TS.v.size(); i++) {
1161         cout << i + 1 << " " << TS.v[i].core + 1 << endl;
1162     }
1163
1164     cout << "-----This is to check ready time-------" << endl;
1165     for (int i = 0; i < TS.RT.size(); i++) {
```

```
1166        cout << TS.RT[i].RT_l << " " << TS.RT[i].RT_ws << " " << TS.RT[i].RT_c
    << " " << TS.RT[i].RT_wr << endl;
1167    }
1168    cout << "------This is to check finish time-----" << endl;
1169    for (int i = 0; i < TS.FT.size(); i++) {
1170        cout << TS.FT[i].FT_l << " " << TS.FT[i].FT_ws << " " << TS.FT[i].FT_c
    << " " << TS.FT[i].FT_wr << endl;
1171    }
1172    cout << "------This is to check core-----------" << endl;
1173    for (int i = 0; i < TS.v.size(); i++) {
1174       // cout << TS.v[i].core + 1 << " ";
1175        cout << i+ 1 << " "  << TS.v[i].priority << endl;
1176    }
1177    cout << endl;
1178    cout << "----This is the presentation of total time----------" << endl;
1179    cout << T_total(TS.FT, TS.v) << endl;
1180    cout << "----This is the presentation of total energy---------" << endl;
1181    cout << E_total(TS.v) << endl;
1182
1183    */
1184
1185
1186    cout << "-------------------Outer loop-------------------------" << endl;
1187    cout << "Set the upper bound for time cost to be 27" << endl;
1188    TS.set_T_max(27);
1189    TS.outer_loop();
1190    end = clock();
1191    t = (double)(end - begin) / CLOCKS_PER_SEC;
1192    /*
1193    cout << "-----This is to check ready time-------" << endl;
1194    for (int i = 0; i < TS.RT.size(); i++) {
1195        cout << TS.RT[i].RT_l << " " << TS.RT[i].RT_ws << " " << TS.RT[i].RT_c
    << " " << TS.RT[i].RT_wr << endl;
1196    }
1197    cout << "------This is to check finish time-----" << endl;
1198    for (int i = 0; i < TS.FT.size(); i++) {
1199        cout << TS.FT[i].FT_l << " " << TS.FT[i].FT_ws << " " << TS.FT[i].FT_c
    << " " << TS.FT[i].FT_wr << endl;
1200    }
1201    cout << "------This is to check core-----------" << endl;
1202    for (int i = 0; i < TS.v.size(); i++) {
1203         cout << TS.v[i].core << " ";
1204    }
1205    cout << endl;
1206     */
1207    cout << "----This is the presentation of total time----------" << endl;
1208    cout << T_total(TS.FT, TS.v) << endl;
1209    cout << "----This is the presentation of total energy---------" << endl;
1210    cout << E_total(TS.v) << endl;
1211    cout << "The running time for the code is " << t << "s." << endl;
1212
```

```
1213
1214      /*
1215      cout << "----------kernel----------" << endl;
1216      T_E_ori_tar te =  TS.kernel(0, 3);
1217      cout << "T_new = " << te.T_new_total << endl;
1218      cout << "E_new = " << te.E_new_total << endl;
1219      vector<Finishtime> ft = te.ft;
1220      vector<Readytime> rt = te.rt;
1221      cout << "---this is to check the kernel arrangement for ready time---" <<
         endl;
1222      for (int i = 0; i < rt.size(); i++) {
1223          cout << rt[i].RT_l << " " << rt[i].RT_ws << " " << rt[i].RT_c << " "
         << rt[i].RT_wr << endl;
1224      }
1225
1226      cout << "--this is to check the kernel arrangement for finish time-------"
         << endl;
1227      for (int i = 0; i < ft.size(); i++) {
1228          cout << ft[i].FT_l << " " << ft[i].FT_ws << " " << ft[i].FT_c << " "
         << ft[i].FT_wr << endl;
1229      }
1230       */
1231      cout << "The final task scheduling result (after Step II): " << endl;
1232      TS.print_scheduling_result();
1233      cout << "----------" << endl;
1234
1235      cout << ">>>>>>Example 2<<<<<<<<<<" << endl;
1236      cout << "Set the upper bound for time cost to be 100" << endl;
1237      TaskSchedule TS1 = TaskSchedule(3, 0, 0, 0.5, {1, 2, 4}, {1, 1, 1}, {1, 1,
         1}, V, T_local, T_cloud);
1238      begin = clock();
1239      TS1.primary_assign();
1240      TS1.task_prior();
1241      TS1.execution_unit_selection();
1242      TS1.set_T_max(100);
1243      TS1.outer_loop();
1244      end = clock();
1245      cout << "----This is the presentation of total time----------" << endl;
1246      cout << T_total(TS1.FT, TS1.v) << endl;
1247      cout << "----This is the presentation of total energy----------" << endl;
1248      cout << E_total(TS1.v) << endl;
1249      cout << "The running time for the code is " << t << "s." << endl;
1250      TS1.print_scheduling_result();
1251      cout << "----------" << endl;
1252
1253      cout << ">>>>>>Example 3<<<<<<<<<<" << endl;
1254      cout << "Set number of heterogeneous cores to be 6" << endl;
1255      cout << "Set P_1 = 1, P_2 = 2, P_3 = 4, P_4 = 8, P_5 = 16, P_6 = 32, P_s =
         0.5" << endl;
```

```
1256      cout << "Set the upper bound for time cost to be 20" << endl;
1257      // Local running time for each task on each core
1258      vector<Localtime> T1_local;
1259      Localtime T1_k_1;
1260      T1_k_1.T_k = {9, 7, 5, 4, 2, 1};
1261      Localtime T1_k_2;
1262      T1_k_2.T_k = {8, 6, 5, 3, 2, 1};
1263      Localtime T1_k_3;
1264      T1_k_3.T_k = {6, 5, 4, 3, 2, 1};
1265      Localtime T1_k_4;
1266      T1_k_4.T_k = {7, 5, 4, 3, 2, 1};
1267      Localtime T1_k_5;
1268      T1_k_5.T_k = {6, 5, 4, 3, 2, 1};
1269      Localtime T1_k_6;
1270      T1_k_6.T_k = {7, 6, 5, 4, 3, 1};
1271      Localtime T1_k_7;
1272      T1_k_7.T_k = {8, 6, 5, 3, 2, 1};
1273      Localtime T1_k_8;
1274      T1_k_8.T_k = {6, 5, 4, 3, 2, 1};
1275      Localtime T1_k_9;
1276      T1_k_9.T_k = {5, 3, 3, 2, 2, 1};
1277      Localtime T1_k_10;
1278      T1_k_10.T_k = {7, 6, 4, 3, 2, 1};
1279      T1_local.push_back(T1_k_1);
1280      T1_local.push_back(T1_k_2);
1281      T1_local.push_back(T1_k_3);
1282      T1_local.push_back(T1_k_4);
1283      T1_local.push_back(T1_k_5);
1284      T1_local.push_back(T1_k_6);
1285      T1_local.push_back(T1_k_7);
1286      T1_local.push_back(T1_k_8);
1287      T1_local.push_back(T1_k_9);
1288      T1_local.push_back(T1_k_10);
1289      cout << "The local running times are defined as follows: " << endl;
1290      print_localtime(T1_local);
1291
1292      TaskSchedule TS2 = TaskSchedule(6, 0, 0, 0.5, {1, 2, 4, 8, 16, 32}, {1, 1,
          1, 1, 1, 1}, {1, 1, 1, 1, 1, 1}, V, T1_local, T_cloud);
1293      begin = clock();
1294      TS2.primary_assign();
1295      TS2.task_prior();
1296      TS2.execution_unit_selection();
1297      cout << "The task scheduling result after Step I: " << endl;
1298      TS2.print_scheduling_result();
1299
1300      TS2.set_T_max(10);
1301      TS2.outer_loop();
1302      end = clock();
1303      cout << "The task scheduling result after Step II: " << endl;
1304      cout << "——This is the presentation of total time————————" << endl;
1305      cout << T_total(TS2.FT, TS2.v) << endl;
```

```
1306     cout << "------This is the presentation of total energy----------" << endl;
1307     cout << E_total(TS2.v) << endl;
1308     cout << "The running time for the code is " << t << "s." << endl;
1309     TS2.print_scheduling_result();
1310
1311     return 0;
1312 }
```

Listing 1: MCC Task Scheduling

Conclusion

The key point for implementing MCC task scheduling is to figure out which items are invariant (e.g., the local running time for each task, the precedent relationship) and which are not (e.g., the time and energy cost for the whole process). At each time a task is scheduled, the corresponding states for this task should be updated immediately in the record (e.g., the location where it is executed, the ready time and finish time).

References

[1] X. Lin, Y. Wang, Q. Xie and M. Pedram, "Energy and Performance-Aware Task Scheduling in a Mobile Cloud Computing Environment," *2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*, . Anchorage, AK, USA, 2014, pp. 192-199..