PARALLEL PROGRAMMING

Assignment 2

MPI – Parallization

Tammo Johannes Herbert (319391), Rico Jasper (319396), Erik Rudisch (343930)

15.05.2013

Exercise 1 – Sequential Overspecification

Consider the following sequential algorithm that calculates max_sum:

```
int a[N] = { -1, 5, -2, 0, 1, 4, 6, -4, 3, -6, ... };
int max_sum = 0;
int cur_sum = 0;
for (i = 0; i < N; i++) {
      cur_sum += a[i];
      if (cur_sum < 0) cur_sum = 0;
      if (cur_sum > max_sum) max_sum = cur_sum;
}
```

There is no straightforward way to parallelize the loop as each iteration depends on the previous iteration. Yet, the underlying problem can be parallelized.

(a) Analysis

Analyze the algorithm in order to extract the original problem. What is the purpose of the given algorithm?

Solution

Der gegebene Code sucht in einem n-elementigen Vektor v nach der größtmöglichen Teilsumme. Eine Teilsumme kann folgendermaßen definiert sein:

Sei $a, b \in \mathbb{N}$, mit $a \leq b$, $a \geq 1$ und $b \leq n$. So ist eine Teilsumme $\sum_{i=a}^{b} v_i$.

Das Problem kann folgendermaßen umformuliert werden. Der Vektor v wird auf eine Folga $a(k)$ abgebildet. Dabei gilt $a(k) = v_k$. Zu dieser Folge wird die kumulierte Summe $A(k) = \sum_{k=1}^{n} a(k)$ gebildet. Zu der neuen Folge $A(k)$ wird nun sowohl das globale Minimum min_0 als auch Maximum max_0 bestimmt. Nun betrachtet man zunächst nur alle Werte rechts vom globalen Minimum min_0 und sucht dort nach dem Maximum max_r. Analog suchen wir links vom globalen Maximum max_0 nach dem Minimum min_l. Wir definieren $d_r \coloneqq max_r - min_0$ und $d_l \coloneqq max_0 - min_l$. Die größte Teilsumme ist nun $\max(d_l, d_r)$.

(b) Parallel Algorithm

Devise a parallel algorithm based on your problem description!

Write down your algorithm in (commented/explained) pseudo code using collective operations when necessary!

Solution

Das Problem kann nach dem Teile-und-herrsche-Prinzip gelöst werden. Entsprechend wird der Vektor v in m etwa gleich große Teilvektoren u_i aufgeteilt. Zu jedem Teilvektor v_i wird die Summe s_i aller Elemente, sowie Minimum min_i und Maximum max_i berechnet. Nach Berechnung muss zu jeder Summe sowie auf Minimum und Maximum ein Offset c_i addiert werden. Dieser wird folgendermaßen berechnet: $c_i = \sum_{k=1}^{i-1} s_i$, wobei $c_1 \coloneqq 0$. Aus allen Minima und Maxima wird nun das globale Minimum und Maximum berechnet: $gmin \coloneqq \min_{i\in\{1,\dots,m\}}\{min_i\}$ und $gmax \coloneqq \max_{i\in\{1,\dots,m\}}\{max_i\}$. Wie oben beschrieben, bestimmt man nun links oder rechts der jeweiligen Extrema weitere Extrema und deren Differenzen.

```
1  nodes = number_of_nodes
2
3  [a, b] = get_job                        // every node works on a
4                                          // different range
5
6  [sum, vmin, vmax] =                     // here the actual work is
7      sum_and_extremes(v[a to b])         // done
8
9  switch (role)                           // now we gather the results
10 case not_root:                          // and put everything
11     send(sum, vmin, vmax);              // together
12     break;
13 case root:
14     sums[nodes];                        // initialize result vectors
15     mins[nodes];
16     maxs[nodes];
17
18     sums[0] = sum;                      // put root's result into
19     mins[0] = vmin;                     // vectors
20     maxs[0] = vmax;
21
22     while (pending_slaves) {            // collect others' results
23         [sum, vmin, vmax, id] =
24             recv();
25
26         sums[id] = sum;
27         mins[id] = vmin;
28         maxs[id] = vmax;
29     }
30
31     offsets = calc_offsets(sums)        // compensate offset error
32     mins += offsets;
33     maxs += offsets;
34
35     print valley_hill(mins, maxs);      // determine max sum and print
36
37     break;
38 }
39
40 [sum, vmin, vmax] =                     // finds minima and maxima of
41     sum_and_extremes(v[])               // the given vector v
42 {
43     sum = 0;
44     vmin = MAX_INT;
45     vmax = MIN_INT;
46
47     foreach vi in v
48         with index i
49     {
50         sum += vi;
51
52         if (sum < vmin)
53             vmin = sum
54         if (sum > vmax)
55             vmax = sum
56     }
57 }
58
59 d = valley_hill(mins[], maxs[]) {       // calculates the greatest
60     min_l = MAX_INT;                    // difference between minimum
61     max_r = MIN_INT;                    // and maximum where the minimum
62     min_0 = MAX_INT;                    // is before the maximum
```

```
63      max_0 = MIN_INT;
64
65      imin = 0;
66      imax = 0;
67                                          // find global minimum
68      foreach vmin in
69          mins with index i
70      {
71          if (vmin < min_0) {
72              min_0 = vmin;
73              imin = i
74          }
75      }
76
77      foreach vmax in                     // find global maximum
78          maxs with index i
79      {
80          if (vmax < max_0) {
81              max_0 = vmax;
82              imax = i
83          }
84      }
85
86      foreach vmin in                     // minimum left from global
87          mins[0 to imax-1]               // maximum
88      {
89          if (vmin < min_l)
90               min_l = vmin;
91      }
92
93      foreach vmax in                     // maximum right from global
94          maxs[imin+1 to end]             // minimum
95      {
96          if (vmax > max_r)
97              max_r = vmax;
98      }
99
100     d_l = max_0 - min_l;                // we could actually avoid half
101     d_r = max_r - min_0;                // the calculation if the global
102                                         // maximum is right from the
103     d = max(d_l, d_r);                  // global minimum
104 }
105
106 offsets[] = calc_offsets(sums[]) {      // calculates the offsets which
107     acc = 0;                            // are normalizing the partial
108                                         // vectors
109     foreach sum in
110         sums with index i
111     {
112         offsets[i] = acc;
113         acc += sum;
114     }
    }
```

Exercise 2 – Conway's Game of Life

The goal of this exercise is an implementation of a distributed memory version of Conway's Game of Life. See for instance http://en.wikipedia.org/wiki/Conway%27s_game_of_life for an introduction.

Your implementation should not focus on absolute speed, but instead on correct interaction patterns and a bottleneck-free design. There are two variants presented here: A (simple) and B (advanced). (If you are interested in speed, have a look at *Hash Life*.)

Variant A: Regular Parallelism

A matrix consisting of 1s and 0s is used as a representation of the playing field.

Variant B: Irregular Parallelism

Storing a complete matrix is not necessary as (normally) most cells are dead and do not change. Therefore, it is possible to store only coordinates of living (or interesting) cells, as only living cells or dead cells with living neighbors may change. (Another side-effect is that one can get rid of borders as coordinates may have arbitrary values.)

(a) Problem decomposition

For *both* variants, consider different strategies to distribute work and data across multiple processes – basically the last two steps in Foster's design methodology.

Use the discussion forum in ISIS to present *one* way to decompose the problem (for A *or* B). Present a decomposition that was not yet given. In your suggestion, include the data and work distribution and the necessary data exchanges between nodes.

For your submission, compare different suggestions (for A *and* for B) and highlight their advantages and disadvantages – also considering different target systems.

Solution

Variant A		Variant B	
Advantages	Disadvantages	Advantages	Disadvantages
Suitable for small grids	Bad performance by large amount of data, because the whole array has to be iterated	Just storing the living cells reduces the amount of needed storage	The complexity to apply conways rules increases
Simple arrays fit the data parameters of the MPI-methods directly	To use complete arrays large amount of storage is needed		The implementation becomes more complex
Intuitive data usage – every cell is directly located in the grid			

(b) Implementation

Select one variant, A *or* B, and implement a strategy with MPI, which is suitable for our cluster!

Solution

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>
#include <math.h>
#include "map.h"

void calc_next_tick(map_t* map, int rank, int size);

int main(int argc, char** argv)
{
	int rank, size;

	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &size);

	if(size < 2)
	{

		map_t* map = calloc(1, sizeof(map_t));
		map_init(map, 16, 16);
		map_fill_pulsar(map);
		while(true)
		{
			map_print(map);
			calc_next_tick(map, rank, size);
			sleep(1);

		}
	} else
	{
		int map_width = (int) ceil((float)16/(float)(size-1));

		if(rank == 0)
		{
			// Initialize the globale map.
			map_t* map = calloc(1, sizeof(map_t));
			map_init(map, map_width*(size-1), 16);
			map_fill_pulsar(map);

			// Distribute the globale map to working processes.
			for(cell_t* cell_i = map_get_next(map); cell_i != NULL;
cell_i = map_get_next(map))
			{
				int dot[2] = {cell_i->x, cell_i->y};
				int segment = ((cell_i->x)/map_width)+1;
				MPI_Send(dot, 2, MPI_INT, segment, 0,
MPI_COMM_WORLD);
			}
			int dot[2] = {-1, -1};
			for(int i = 1; i < size; i++)
				MPI_Send(dot, 2, MPI_INT, i, 0, MPI_COMM_WORLD);

			// Reveive the composed map and print it.
			int count;
```

```
                  while(true)
                  {
                        count = 0;
                        while(count < size-1)
                        {
                              MPI_Recv(dot, 2, MPI_INT, MPI_ANY_SOURCE, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                              if(dot[0] == -1 && dot[1] == -1)
                                    count++;
                              else
                                    map_add(map, dot[0], dot[1]);
                        }

                        map_print(map);

                        map_free(map);
                  }
            } else
            {
                  map_t* map = calloc(1, sizeof(map_t));
                  map_init(map, map_width, 16);

                  // Receive the disributed map from root.
                  int dot[2] = {0, 0};
                  while(dot[0] != -1 && dot[1] != -1)
                  {
                        MPI_Recv(dot, 2, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                        if(dot[0] != -1 && dot[1] != -1)
                              map_add(map, dot[0]%map_width, dot[1]);
                  }

                  // Calculate the map and send it to root.
                  while(true)
                  {
                        int offset = (rank-1)*map_width;
                        for(cell_t* cell_i = map_get_next(map); cell_i !=
NULL; cell_i = map_get_next(map))
                        {
                              int dot[2] = {offset+cell_i->x, cell_i->y};
                              MPI_Send(dot, 2, MPI_INT, 0, 0,
MPI_COMM_WORLD);
                        }
                        int dot[2] = {-1, -1};
                        MPI_Send(dot, 2, MPI_INT, 0, 0, MPI_COMM_WORLD);

                        calc_next_tick(map, rank, size);
                        sleep(1);
                  }
            }
      }

      MPI_Finalize();

      return EXIT_SUCCESS;
}

// Send borders to neighbors.
void sendBorders(map_t* map, int rank, int size)
{
      int map_border_left[map->height];
      int map_border_right[map->height];
```

```
	memset(map_border_left,  0, sizeof(map_border_left));
	memset(map_border_right, 0, sizeof(map_border_right));

	for(cell_t* cell_i = map_get_next(map); cell_i != NULL; cell_i =
map_get_next(map))
	{
		int y = cell_i->y;
		int x = cell_i->x;

		if(x == 0)
			map_border_left[y] = 1;
		if(x == map->width-1)
			map_border_right[y] = 1;
	}

	// Send to right neighbor.
	if(rank < size-1)
		MPI_Send(map_border_right, map->height, MPI_INT, rank+1, 0,
MPI_COMM_WORLD);

	// Send to left neighbor.
	if(rank > 1)
		MPI_Send(map_border_left,  map->height, MPI_INT, rank-1, 0,
MPI_COMM_WORLD);
}

// Receive borders from neighbors.
void recvBorders(map_t* map, int rank, int size)
{
	int map_border_left[map->height];
	int map_border_right[map->height];

	// Receive from left neighbor.
	if(rank > 1)
	{
		MPI_Recv(map_border_left,  map->height, MPI_INT, rank-1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		for(int y = 0; y < map->height; y++)
		{
			if(map_border_left[y] == 1)
				map_add(map, -1 ,y);
		}
	}

	// Receive from right neighbor.
	if(rank < size-1)
	{
		MPI_Recv(map_border_right, map->height, MPI_INT, rank+1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		for(int y = 0; y < map->height; y++)
		{
			if(map_border_right[y] == 1)
				map_add(map, map->width ,y);
		}
	}
}

// Apply the rules of conway's game-of-life.
void calc_next_tick(map_t* map, int rank, int size)
{
	if(rank % 2 == 0)
	{
```

```
			sendBorders(map, rank, size);
			recvBorders(map, rank, size);
		} else
		{
			recvBorders(map, rank, size);
			sendBorders(map, rank, size);
		}

		// Count the neighboring cells. Added an extra border around.
		int  map_count[map->width+4][map->height+2];
		memset(map_count, 0, sizeof(map_count));

		for(cell_t* cell_i = map_get_next(map); cell_i != NULL; cell_i =
map_get_next(map))
		{
			int x = cell_i->x+2;
			int y = cell_i->y+1;

			map_count[x][y+1]   += 1;
			map_count[x][y-1]   += 1;
			map_count[x+1][y]   += 1;
			map_count[x+1][y+1] += 1;
			map_count[x+1][y-1] += 1;
			map_count[x-1][y]   += 1;
			map_count[x-1][y+1] += 1;
			map_count[x-1][y-1] += 1;
		}

		// Don't consider the extra border around!
		memset(map_count[0], 0, sizeof(map_count[0]));
		memset(map_count[1], 0, sizeof(map_count[1]));
		memset(map_count[map->width+2], 0, sizeof(map_count[map->width+2]));
		memset(map_count[map->width+3], 0, sizeof(map_count[map->width+3]));
		for(int x = 0; x < map->width+4; x++)
		{
			map_count[x][0] = 0;
			map_count[x][map->height+1] = 0;
		}

		map_t* map_new = calloc(1, sizeof(map_t));
		map_init(map_new, map->width, map->height);

		for(cell_t* cell_i = map_get_next(map); cell_i != NULL; cell_i =
map_get_next(map))
		{
			int x = cell_i->x+2;
			int y = cell_i->y+1;

			for(int dx = -1; dx < 2; dx++)
			{
				for(int dy = -1; dy < 2; dy++)
				{
					int neighbors = map_count[x+dx][y+dy];

					// Is this cell considered already?
					if(neighbors < 1)
						continue;

					// Is this cell newborned or alived?
					if((dx == 0 && dy == 0 && neighbors == 2) ||
(neighbors == 3))
					{
```

```
                                map_add(map_new, x+dx-2, y+dy-1);
                                map_count[x+dx][y+dy] = -1;
                            }
                        }
                    }
                }

                map_free(map);

                *map = *map_new;
}
```