|  |
| --- |
| 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.

## 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 -elementigen Vektor nach der größtmöglichen Teilsumme. Eine Teilsumme kann folgendermaßen definiert sein:

Sei , mit , und . So ist eine Teilsumme .

Das Problem kann folgendermaßen umformuliert werden. Der Vektor wird auf eine Folga abgebildet. Dabei gilt . Zu dieser Folge wird die kumulierte Summe gebildet. Zu der neuen Folge wird nun sowohl das globale Minimum als auch Maximum bestimmt. Nun betrachtet man zunächst nur alle Werte rechts vom globalen Minimum und sucht dort nach dem Maximum . Analog suchen wir links vom globalen Maximum nach dem Minimum . Wir definieren und . Die größte Teilsumme ist nun .

## 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 in etwa gleich große Teilvektoren aufgeteilt. Zu jedem Teilvektor wird die Summe aller Elemente, sowie Minimum und Maximum berechnet. Nach Berechnung muss zu jeder Summe sowie auf Minimum und Maximum ein Offset addiert werden. Dieser wird folgendermaßen berechnet: , wobei . Aus allen Minima und Maxima wird nun das globale Minimum und Maximum berechnet: und . Wie oben beschrieben, bestimmt man nun links oder rechts der jeweiligen Extrema weitere Extrema und deren Differenzen.

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114 | nodes = number\_of\_nodes  [a, b] = get\_job  [sum, vmin, vmax] =  sum\_and\_extremes(v[a **to** b])  **switch** (role)  **case** not\_root:  send(sum, vmin, vmax);  **break**;  **case** root:  sums[nodes];  mins[nodes];  maxs[nodes];  sums[0] = sum;  mins[0] = vmin;  maxs[0] = vmax;  **while** (pending\_slaves) {  [sum, vmin, vmax, id] =  recv();  sums[id] = sum;  mins[id] = vmin;  maxs[id] = vmax;  }  offsets = calc\_offsets(sums)  mins += offsets;  maxs += offsets;  **print** valley\_hill(mins, maxs);  **break**;  }  [sum, vmin, vmax] =  sum\_and\_extremes(v[])  {  sum = 0;  vmin = MAX\_INT;  vmax = MIN\_INT;    **foreach** vi **in** v  **with** **index** i  {  sum += vi;  **if** (sum < vmin)  vmin = sum  **if** (sum > vmax)  vmax = sum  }  }  d = valley\_hill(mins[], maxs[]) {  min\_l = MAX\_INT;  max\_r = MIN\_INT;  min\_0 = MAX\_INT;  max\_0 = MIN\_INT;  imin = 0;  imax = 0;  **foreach** vmin **in**  mins **with** **index** i  {  **if** (vmin < min\_0) {  min\_0 = vmin;  imin = i  }  }  **foreach** vmax **in**  maxs **with** **index** i  {  **if** (vmax < max\_0) {  max\_0 = vmax;  imax = i  }  }  **foreach** vmin **in**  mins[0 **to** imax-1]  {  **if** (vmin < min\_l)  min\_l = vmin;  }  **foreach** vmax **in**  maxs[imin+1 **to** end]  {  **if** (vmax > max\_r)  max\_r = vmax;  }  d\_l = max\_0 - min\_l;  d\_r = max\_r - min\_0;  d = max(d\_l, d\_r);  }  offsets[] = calc\_offsets(sums[]) {  acc = 0;  **foreach** sum **in**  sums **with** **index** i  {  offsets[i] = acc;  acc += sum;  }  } | // every node works on a  // different range  // here the actual work is  // done  // now we gather the results  // and put everything  // together  // initialize result vectors  // put root’s result into  // vectors  // collect others’ results  // compensate offset error  // determine max sum and print  // finds minima and maxima of  // the given vector v  // calculates the greatest  // difference between minimum  // and maximum where the minimum  // is before the maximum  // find global minimum  // find global maximum  // minimum left from global  // maximum  // maximum right from global  // minimum  // we could actually avoid half  // the calculation if the global  // maximum is right from the  // global minimum  // calculates the offsets which  // are normalizing the partial  // vectors |

# 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.)

## 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 |  |  |  |

## 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;

}