IN3200/IN4200 Spring 2024
Feedback about oblig 1 & comments about oblig 2

April 8, 2024

Outline

- General feedback about oblig 1
- Comments about oblig 2

Recap of oblig 1

- Serial and OpenMP implementations of Dijkstra's algorithm
- Comparison between dense (regular) and sparse (irregular) data structures
- Identification of parallelism
- OpenMP parallelization

Pseudo code of Dijkstra's algorithm

```
procedure DIJKSTRA(n, s, w, d)
begin
    Allocate a help array of length n to later mark if each vertex has been visited
    for i = 0, 1, 2, ..., n − 1 do
    begin
        if w[s][i]>0 then set d[i] = w[s][i];
        else set d[i] = a large enough positive integer (to represent ∞);
    end
    Set d[s] = 0;
    Mark vertex s as visited, the other vertices as unvisited;

    for i = 1, 2, ..., n − 1 do
    begin
        Find an unvisited vertex u such that the value of d[u]
         is the minimum among all the so-far unvisited vertices;
        Mark vertex u as visited;
        for each v of the remaining unvisited vertices do
        begin
            if w[u][v]>0 then set d[v] = min(d[v], d[u]+w[u][v]);
        end
    end
end DIJKSTRA
```

Serial implementation of dijkstra_serial_v1

```
#define infinity 1000000000

void dijkstra_serial_v1 (int n, int s, int **w, int *d)
{
  int i,u,v, min_d;
  char *visited = malloc(n*sizeof(char));

  for (i=0; i<n; i++) {
    visited[i] = 0;
    d[i] = (w[s][i]>0) ? w[s][i] : infinity;
  }

  d[s] = 0;
  visited[s] = 1;

  for (i=1; i<n; i++) {
    u = -1; min_d = infinity;

    for (v=0; v<n; v++)
      if (!visited[v] && d[v]<min_d) {
        u = v; min_d = d[v];
      }

    if (u>-1) {  // an important test
      visited[u] = 1;
      for (v=0; v<n; v++)
        if (!visited[v] && w[u][v]>0 && (d[v] > d[u]+w[u][v]))
          d[v] = d[u]+w[u][v];
    }
  }

  free(visited);
}
```

Why can it happen that we don't find a 'legal' u?

```
u = -1; min_d = infinity;

for (v=0; v<n; v++)
  if (!visited[v] && d[v]<min_d) {
    u = v; min_d = d[v];
  }

// u may still be -1 after the search
```

Discussion

If we indeed don't find a legal `u`, can we save some computing time?

Although very convenient, why is using a 2D array `w` inefficient?

```
if (u>-1) {
  visited[u] = 1;
  for (v=0; v<n; v++)
    if (!visited[v] && w[u][v]>0
       && (d[v] > d[u]+w[u][v]))
      d[v] = d[u]+w[u][v];
}
```

Discussion

How much memory is needed to store, respectively, a dense matrix and a sparse matrix in the CRS format?

Why do we pass `int *n` and `int ***w` to function `read_graph_from_file_v1`?

How to implement function `read_graph_from_file_v2` efficiently?

Where is the parallelism?

Easy to parallelize 1

```
for (i=0; i<n; i++) {
  visited[i] = 0;
  d[i] = (w[s][i]>0) ? w[s][i] : infinity;
}
```

Easy to parallelize 2

```
for (v=0; v<n; v++)
  if (!visited[v] && w[u][v]>0
      && (d[v] > d[u]+w[u][v]))
    d[v] = d[u]+w[u][v];
```

Tricky to parallelize...

```
u = -1; min_d = infinity;

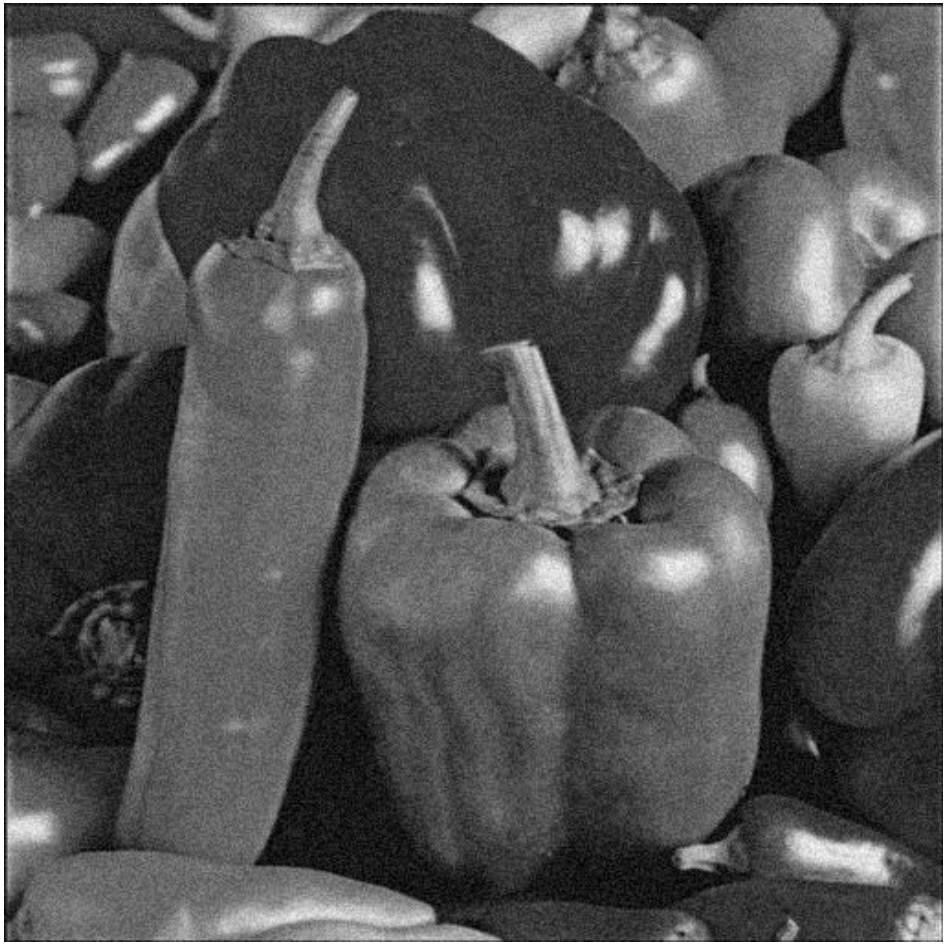
for (v=0; v<n; v++)
  if (!visited[v] && d[v]<min_d) {
    u = v; min_d = d[v];
  }
```

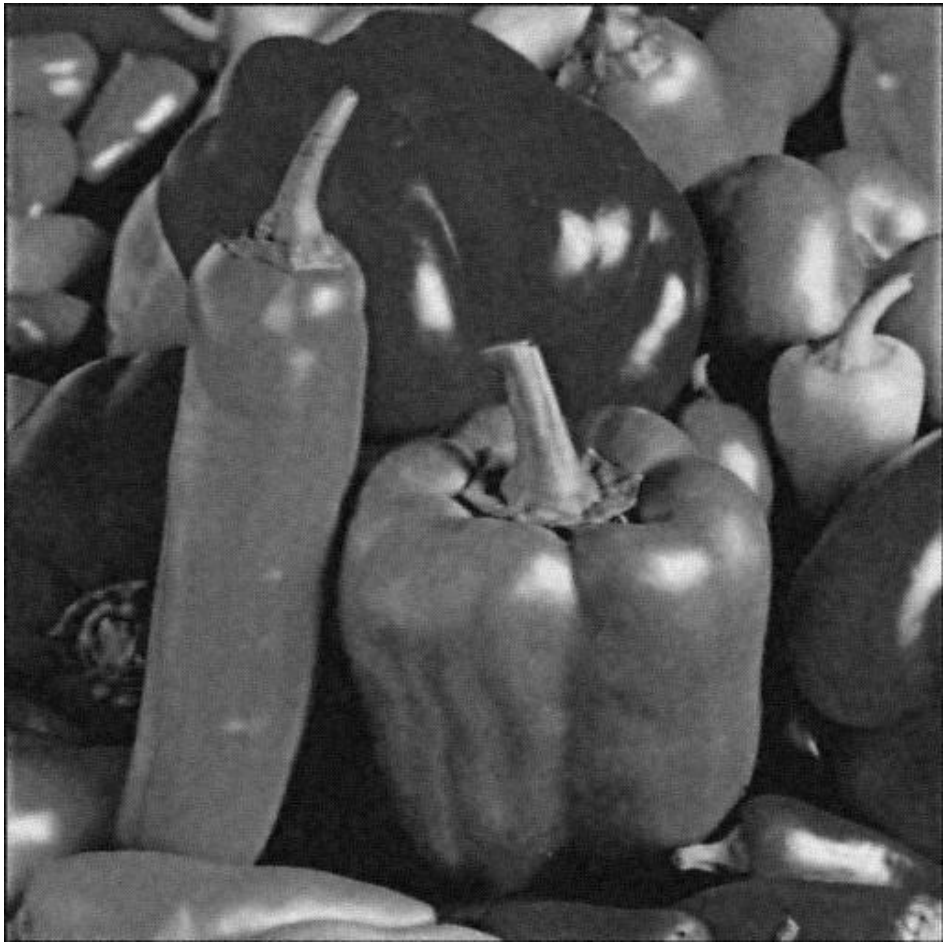
How many parallel regions?

Any speedup from the OpenMP parallelization?

Oblig 2: Image Denoising

Image denoising: removing (or decreasing) random noises in a "contaminated" image





Mathematical representation of an image

An image can be (logically) arranged as a 2D array, containing $m \times n$ pixels:

$$\mathrm{u} = \begin{bmatrix} u_{m-1,0} & u_{m-1,1} & \cdots & u_{m-1,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ u_{1,0} & u_{1,1} & \cdots & u_{1,n-1} \\ u_{0,0} & u_{0,1} & \cdots & u_{0,n-1} \end{bmatrix},$$

where each pixel has a scalar value for the case of a grey-scale image

A simple denoising algorithm

Isotropic diffusion:

$$\bar{u}_{i,j} = u_{i,j} + \kappa \left(u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i,j+1} + u_{i+1,j} \right).$$

ū is thus computed as a "smoother" version of u. (The procedure is typically repeated many iterations.)

Note: The above formula only computes the interior pixels of ū, that is, $1 \leq i \leq m-2$ and $1 \leq j \leq n-2$. The boundary pixels of ū can just copy the corresponding boundary pixels of u.

To read and write a JPEG image

We will use an existing library to read a JPEG image from file, and to write a denoised image to file. (The source code of this small library will be provided to you, no need to program yourself.)

Two functions will be used from this library:

```
void import_JPEG_file (const char* filename, unsigned char** image_chars,
                       int* image_height, int* image_width,
                       int* num_components);

void export_JPEG_file (const char* filename, const unsigned char* image_chars,
                       int image_height, int image_width,
                       int num_components, int quality);
```

An example of how to compile the external library and how to use the two functions will be provided.

Data stucture for image denoising

A 1D array of type *unsigned char* is used by the JPEG library for reading and writing an image.

However, a variable of type *unsigned char* only has an integer value between 0 and 255. This is not sufficient for accurate denoising computations.

We will use the following 2D data structure and floating-point values to represent the pixels while denoising (will need conversion from *unsigned char* to *float*).

```
typedef struct
{
  float** image_data;  /* a 2D array of floats */
  int m;               /* # pixels in vertical-direction */
  int n;               /* # pixels in horizontal-direction */
}
image;
```

Parallelization

For simplicity, we will only use 1D partition, that is, each MPI process is responsible for a consecutive block of rows in the image.

Each MPI process (except the first and last process) will have two neighbors: one from below and one from above.

“Halo rows”

Communication is needed between the neighbors before each parallelized denoising iteration. The communication pattern is the same as parallelizing the 2D Jaocbi solver (see Section 9.3 in the textbook).

Every process should thus allocate an extra “halo” row for each neighbor, to be used to place the incoming message from each neighbor.

Data partitioning

In the beginning only process 0 reads the entire image from file, with `m` rows.

Each process calculates the value of `my_m`, based on its unique rank. (The value of `my_m` is needed to allocate the local data structure per process.)

Note 1: `m` may not be divisible by the number of processes; Note 2: `my_m` should include the halo row(s) needed for communicating with the neighbor(s).

Process 0 then communicates the needed rows to each of the other processes. (Given the `m` value and the total number of processes, process 0 should be able to figure out which process needs what rows.)