

Parallel Programming with OpenMP

Dense Matrix Algorithms

The Stencil Pattern

Bibliography

- [Pacheco]: Peter Pacheco, Matthew Malensek, Introduction to Parallel Programming, 2nd Edition, Morgan Kaufmann Publisher, March 2020, Chapter 5
- <https://medium.com/@Styp/why-loops-do-matter-a-story-of-matrix-multiplication-cache-access-patterns-and-java-859111845c25>

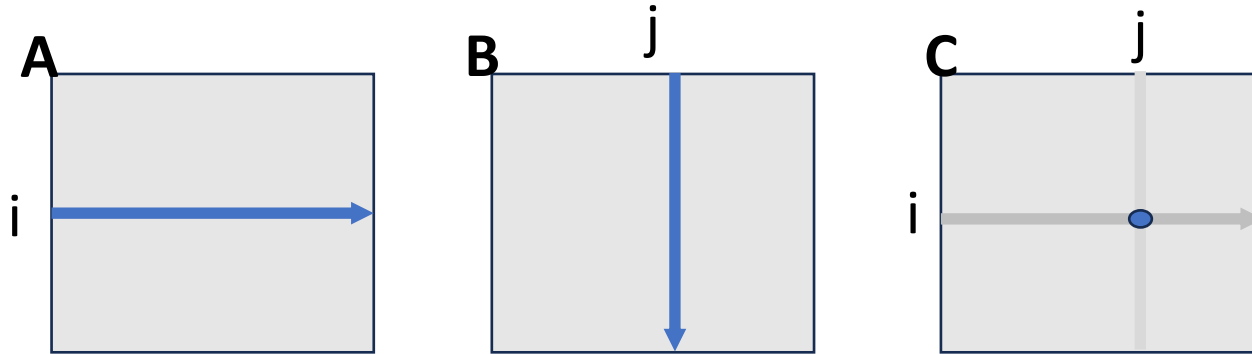
Dense Matrix Algorithms

Examples

- Matrix operations (Addition, Multiplication, Transposed, etc)
- Solving system of linear equations (LU decomposition, Gaussian elimination)
- “Stencil” pattern:
 - Applying the finite differences method – modelling various physical phenomena (Heat transfer model, atmospheric weather prediction model, etc)
 - Image processing

Matrix-Matrix Multiplication

$$C = A * B$$



Matrix-Matrix Multiplication – Classic Serial V0

```
#define N 1000 /* number of rows and column of matrices */

double a[N][N], /* matrix A to be multiplied */
       b[N][N], /* matrix B to be multiplied */
       c[N][N]; /* result matrix */

void serial_multiply(void)
{
    int i, j, k;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            c[i][j] = 0;
            for (k = 0; k < N; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
}
```

Matrix-Matrix Multiplication – Parallelization

- Every element $c[i][j]$ could be computed as an independent job
- Applying data partitioning of the result matrix: we partition the matrix C in stripes and assign a “stripe” to every processor
- With loop parallelism we assign the iterations on variable i to threads – each thread gets to compute one row of the result. We can vary the scheduling chunksize to assign wider “stripes”

Matrix-Matrix Multiplication – Parallel v0

```
void parallel_multiply(int nthreads, int chunk)
{
    int i, j, k;
    double temp;
#pragma omp parallel num_threads(nthreads), private(i, j, k, temp)
    {
#pragma omp for schedule(static, chunk)
        for (i = 0; i < N; i++)
        {
            for (j = 0; j < N; j++)
            {
                c[i][j] = 0;
                for (k = 0; k < N; k++)
                    c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

Matrix-Matrix Multiplication V0

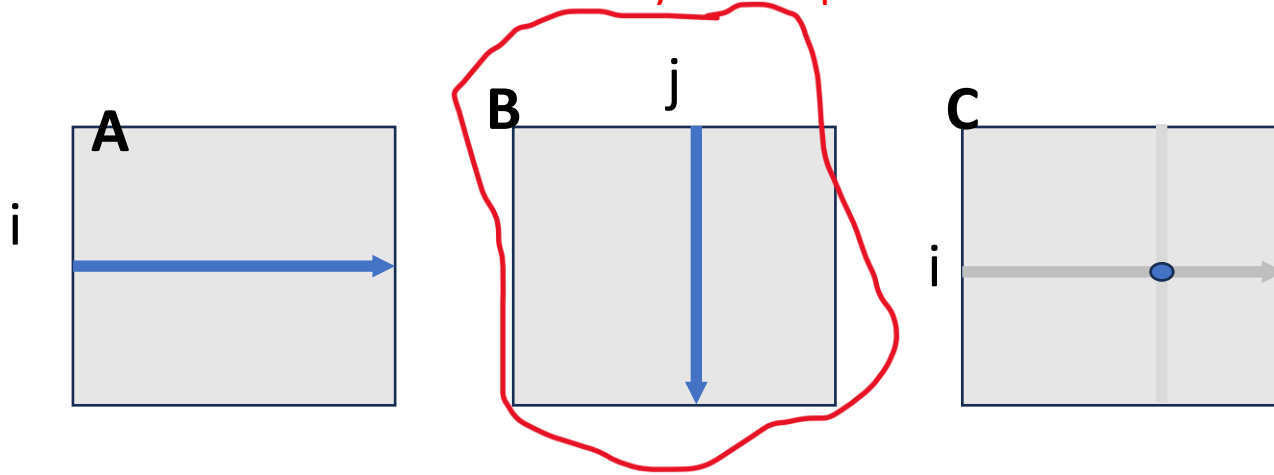
	N=1000	N=2000	N=3000
Serial Classic Time	4.19	48.16	188.4
Time Threads=2	2.07	22.98	85.03
Speedup Threads=2	2.02	2.09	2.21
Time Threads=4	1.07	11.55	42.8
Speedup Threads=4	3.91	4.16	4.39
Time Threads=8	0.62	7.04	25.5
Speedup Threads=8	6.75	6.84	7.38

Paralellization of the classical matrix multiplication algorithm appears to give good speedup. Even Supralinear Speedups (Speedup>nr threads) are measured!!!

Supralinear Speedups appear when cache miss rate is smaller in the parallel version than the serial version. *Can we do better?*

Matrix-Matrix Multiplication

Bad memory access pattern



Classical algorithm:

```
For i ...  
  For j ...  
    For k...  
       $c[i][j] = a[i][k] * b[k][j]$ 
```



Enhanced algorithm:

```
For i ...  
  For k ...  
    For j...  
       $c[i][j] = a[i][k] * b[k][j]$ 
```

Facultative additional reading

- [Why loops do matter — A Story of Matrix Multiplication, Cache Access Patterns and Java | by Martin Stypinski | Medium](#) (discusses cache access patterns in different versions of the serial matrix multiplication algorithm)

Matrix-Matrix Multiplication – Serial V2

```
void serial_multiply_v2(void)
{
    int i, j, k;
    double temp, aik;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            c[i][j] = 0;
    for (i = 0; i < N; i++)
        for (k = 0; k < N; k++)
        {
            aik = a[i][k];
            for (int j = 0; j < N; j++)
            {
                c[i][j] += aik * b[k][j];
            }
        }
}
```

Matrix-Matrix Multiplication – Parallel V2

```
void parallel_multiply_v2(int nthreads, int chunk)
{
    int i, j, k;
    double temp, aik;
#pragma omp parallel num_threads(nthreads), private(i, j)
#pragma omp for schedule(static, chunk)
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            c3[i][j] = 0;
#pragma omp parallel num_threads(nthreads), private(i, j, k, aik)
#pragma omp for schedule(static, chunk)
    for (i = 0; i < N; i++)
        for (k = 0; k < N; k++)
        {
            aik = a[i][k];
            for (int j = 0; j < N; j++)
            {
                c3[i][j] += aik * b[k][j];
            }
        }
}
```

Matrix-Matrix Multiplication Time

	N=1000	N=2000	N=3000	
Serial Classic	4.19	48.16	188.4	
Par V0 Threads=2	2.07	22.98	85.03	
Par V0 Threads=4	1.07	11.55	42.8	
Par V0 Threads=8	0.62	7.04	25.5	
Serial V2	2.32	18.67	63.48	
Par V2 Threads=2	1.42	12.07	44.06	
Par V2 Threads=4	0.75	6.72	23.52	
Par V2 Threads=8	0.46	3.92	14.57	

Matrix-Matrix Multiplication Time

- V0 (classic version - ijk): its parallelization *apparently* results in great speedup, if we take into account V0 serial as baseline
 - A parallel algorithm should consider as baseline for speedup the best serial algorithm for that problem!
- V2 (version ikj): improves cache miss rate for both serial and parallel versions.
 - Improves serial time, improves parallel time !

Code Examples

- [omp_matrix_mult.c](#) (V0 Serial+Par, V2 Serial+Par)

Parallel Algorithms on Dense Matrixes

- Need to make efficient use of cache
 - Memory access patterns should be predictable so cache re-ahead helps
 - Simplest if use unit stride
 - *Need to re-use data in cache as much as possible before replacing it*
- A concept that is useful in matrix algorithms is that of *block matrix operations*:
 - We can often express a matrix computation involving scalar algebraic operations on all its elements in terms of identical matrix algebraic operations on blocks or submatrices of the original matrix
 - For example, an $n \times n$ matrix A can be regarded as a $q \times q$ array of blocks $A_{i,j}$ ($0 \leq i, j < q$) such that each block is an $(n/q) \times (n/q)$ submatrix
 - *Size of a block (submatrix) must be chosen such as an entire block has place in the cache memory at once*

Blocked Matrix Algorithms

- **Blocked Matrix Algorithms: Instead of operating on elements of matrix, blocked algorithms operate on sub-matrices or blocks**
- **The key for improved performance** by blocked algorithms:
 - chose block size bs (submatrix size) so that it fits into cache
 - redesign the algorithm to reuses for a while only the data loaded into the cache (improve *temporal locality*)
 - matrix algorithms that **have potential for improvement** have a bigger computational complexity than memory complexity:
 - Matrix multiplication: $O(n^2)$ memory, $O(n^3)$ computations - this shows that data is used in multiple computations – can exploit *temporal locality* of caches

Blocked Matrix Multiplication

- All matrices $M_{i,j}$ are regarded as a $q \times q$ array of blocks $M_{i,j}$ ($0 \leq i, j < q$) such that each block is an $(n/q) \times (n/q)$ submatrix.
Block size $bs = n/q$
- For the matrix multiplication $C = A * B$:
- *All submatrices* C_{ij} (of size $bs * bs$) are computed as a sum of products of *submatrices* A_{ik} , B_{kj}

bs				
bs	A ₁₁	A ₁₂	A _{1n}
	A ₂₁	A ₂₂	A _{2n}
	⋮	⋮	⋮	⋮
	A _{n1}	A _{n2}	A _{nn}

×

bs				
bs	B ₁₁	B ₁₂	B _{1n}
	B ₂₁	B ₂₂	B _{2n}
	⋮	⋮	⋮	⋮
	B _{n1}	B _{n2}	B _{nn}

=

bs				
bs	C ₁₁	C ₁₂	C _{1n}
	C ₂₁	C ₂₂	C _{2n}
	⋮	⋮	⋮	⋮
	C _{n1}	C _{n2}	C _{nn}

The block matrix multiplication algorithm for $n \times n$ matrices with a block size of $(n/q) \times (n/q)$.

```
procedure BLOCK_MAT_MULT (A, B, C)
begin
  for  $i := 0$  to  $q - 1$  do
    for  $j := 0$  to  $q - 1$  do
      begin
        Initialize all elements of  $C_{i,j}$  to zero;
        for  $k := 0$  to  $q - 1$  do
           $C_{i,j} := C_{i,j} + A_{i,k} \times B_{k,j};$ 
        endfor;
      end BLOCK_MAT_MULT
```

$C_{i,j}$ $A_{i,k}$ $B_{k,j}$ are matrices of $(n/q) \times (n/q)$

If we expand operations on submatrices, the algorithm will have 6 levels of nested loops

Performant Matrix Operations: State of art and links

- BLAS(Basic Linear Algebra Subprograms):
 - There are different BLAS libraries targeting different platforms
 - see about:
https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms
- Articles:
 - https://www.cs.utexas.edu/~flame/pubs/GotoTOMS_revision.pdf
 - https://www.cs.utexas.edu/~flame/pubs/blis3_ipdps14.pdf
- Discussions about implementation:
 - <https://cs.stanford.edu/people/shadjis/blas.html>
 - <https://salykova.github.io/matmul-cpu>

Stencils

The Stencil pattern

- *Stencils* compute each element of an output array as a function of neighbor elements from the input array
- For 2D array: the output element (x,y) is computed as a function of the input elements *above, below, to the left, and to the right*, that is, $(x,y-1)$, $(x,y+1)$, $(x-1,y)$, $(x+1,y)$, although any pattern is possible involving more neighbors
- Stencil loop: the stencil is applied to all elements of the array
- Stencil loops are often embedded in nested loops (array is processed in many successive generations)
- Stencil loops are very frequent in a number of applications:
 - scientific computing: for solving PDEs discretized with Finite Difference (FD) or Finite Volume (FV) methods.
 - Image processing: blur, edge detection
- Stencils are easily parallelized with shared memory approaches

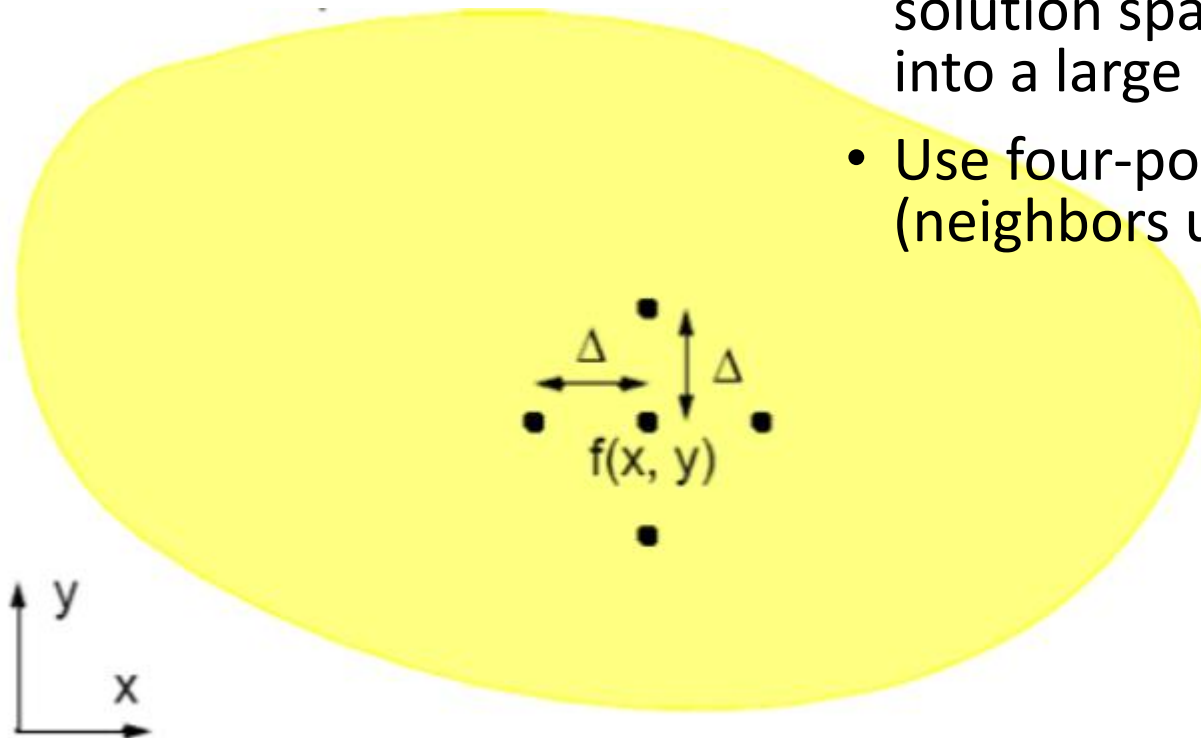
The Heat Distribution Problem

- We have a 2D rectangular piece
- We hold the piece's boundaries at some constant temperature values (apply heat in specific points, or freeze, etc) (boundary conditions)
- Determine what will be the temperature in every point of the piece
- Intuitively:
 - Heat flows in objects according to local temperature differences, as if seeking local equilibrium
 - Because of heat diffusion, temperature of neighboring patches of the area is bound to equalize, changing the overall distribution
- Mathematically:
 - Heat transfer in the system above is governed by the partial differential equation(s) describing local variation of the temperature field in time and space
 - The Steady State Heat Equation = Laplace Equation
 - Jacobi Iterative Method.

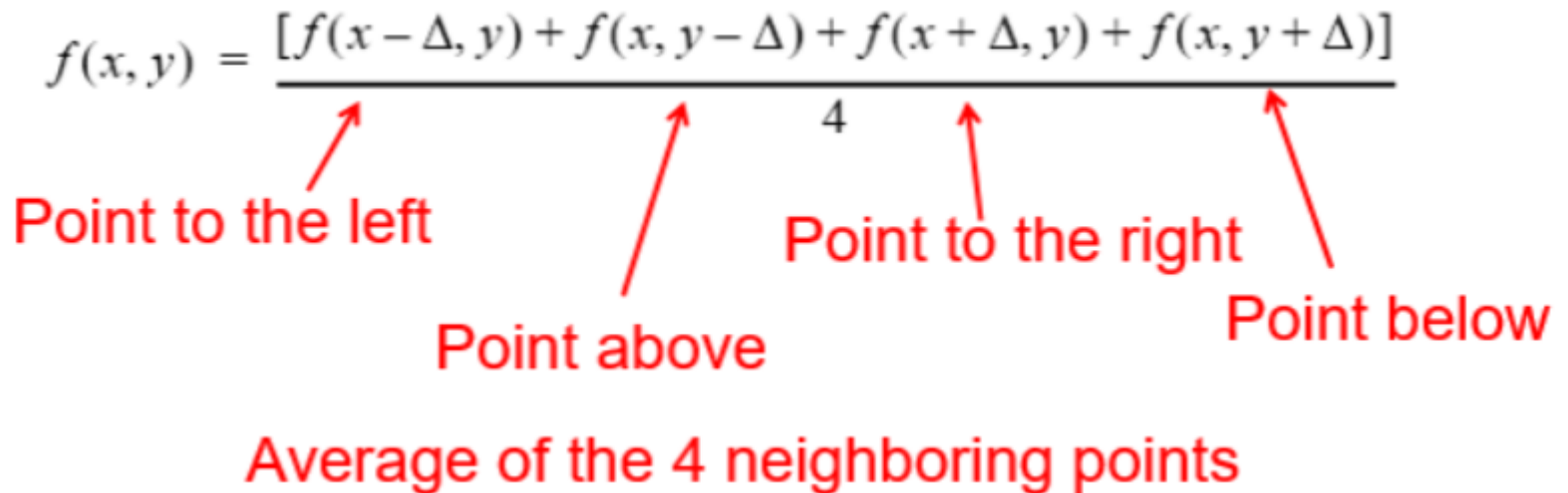
Laplace Equation

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

- Solve for f over the 2D x - y space
- For numerical solution, use ***finite difference method***: the solution space is discretized into a large number of points
- Use four-point stencil (neighbors up, down, left, right)



Laplace Solution

$$f(x, y) = \frac{[f(x - \Delta, y) + f(x, y - \Delta) + f(x + \Delta, y) + f(x, y + \Delta)]}{4}$$


Point to the left

Point above

Point to the right

Point below

Average of the 4 neighboring points

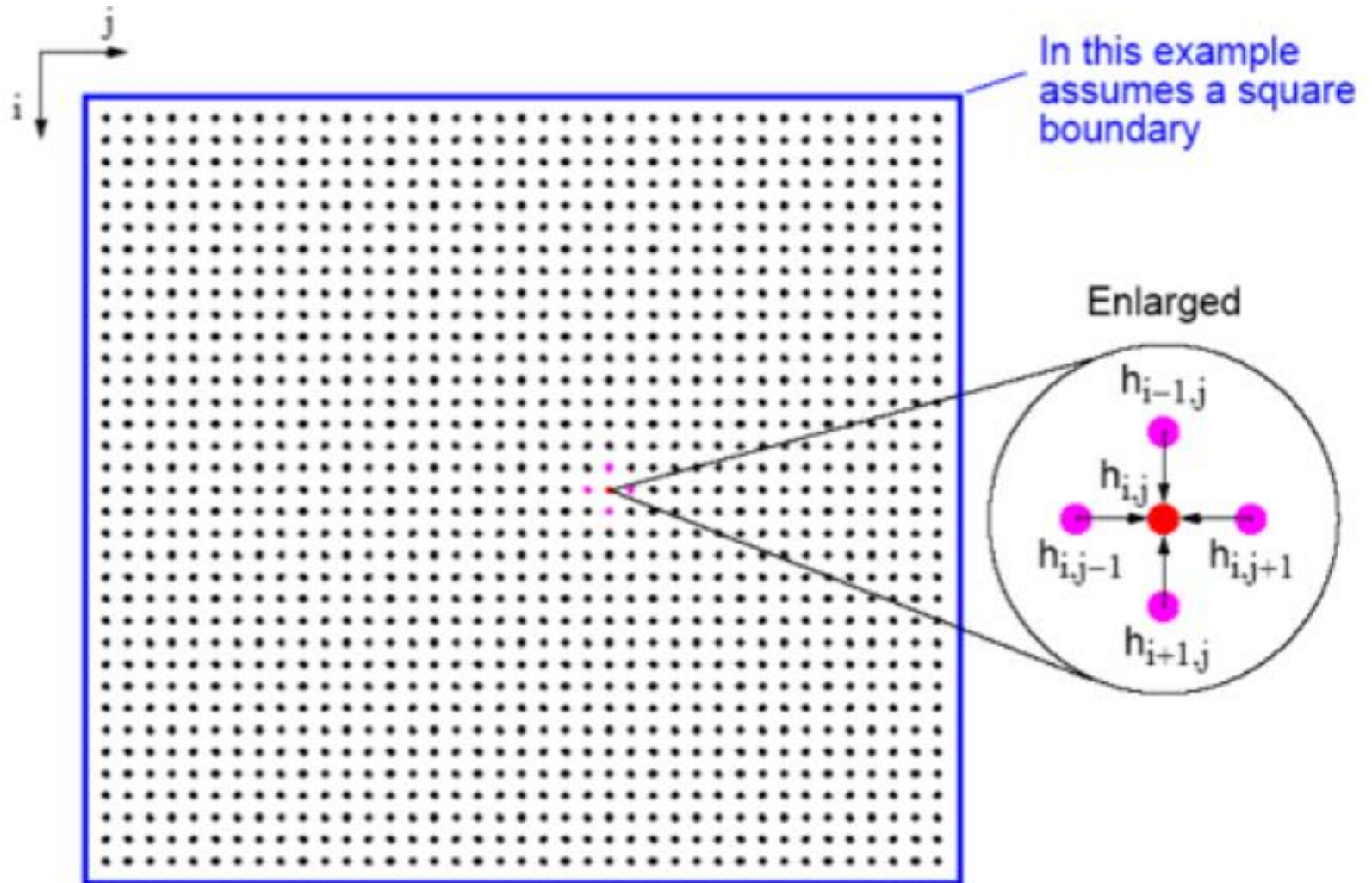
Jacobi Iteration

$$f^k(x, y) = \frac{[f^{k-1}(x - \Delta, y) + f^{k-1}(x, y - \Delta) + f^{k-1}(x + \Delta, y) + f^{k-1}(x, y + \Delta)]}{4}$$

$f^k(x, y)$ - k th iteration, $f^{k-1}(x, y)$ - $(k - 1)$ th iteration.

- If the number of iterations is big enough, the difference between successive estimates of the solution will go to zero
- In practice:
 - Set a fixed large number of iterations, or
 - Set a *epsilon* value representing the maximum allowed difference between successive estimates

Heat 2D



Heat 2D - Serial

```
void serial_temp()
{
    int i, j, time;
    int current, next;
    current = 0;
    next = 1;

    for (time = 0; time < MAXITER; time++)
    {
        for (i = 1; i < N - 1; i++) // iterate grid but skip boundary
            for (j = 1; j < N - 1; j++)
                temp[next][i][j] = (temp[current][i + 1][j] +
                                     temp[current][i - 1][j] +
                                     temp[current][i][j + 1] +
                                     temp[current][i][j - 1]) *
                                     0.25;

        current = next; // toggle between current and next grid
        if (current == 0) next = 1; else next = 0; //avoid copying arrays!!
    }
}
```

Heat 2D - Parallel

```
void parallel_temp()
{
    int i, j, time;
    int current, next;
    current = 0;
    next = 1;
    for (time = 0; time < MAXITER; time++)
    {
#pragma omp parallel for num_threads(NTHREADS) \
                        default(none) private(i, j) shared(temp, next, current)
        for (i = 1; i < N - 1; i++)
            for (j = 1; j < N - 1; j++)
                temp[next][i][j] = (temp[current][i + 1][j] +
                                     temp[current][i - 1][j] +
                                     temp[current][i][j + 1] +
                                     temp[current][i][j - 1]) *
                                     0.25;

        current = next;
        if (current == 0) next = 1; else next = 0;
    }
}
```

Code Examples

- [omp_heat.c](#)

Image Processing

- Digital images are represented as 2D arrays of pixels
- In color images, each pixel contains 3 values, the different amounts of the 3 fundamental colors RGB
- Many image transformations can be carried out by applying different stencils to each point of the original image
- Examples of transformations: Blur, smooth, edge detection
- <https://www.damtp.cam.ac.uk/user/hf323/M21-II-NA/demos/stencil/stencil.html>

Box Blur

- Each pixel in the resulting image has a value equal to the average value of its neighboring pixels in the input image.
- A 3 by 3 box blur ("radius 1") can be written as matrix $\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
- Apply transformation for every pixel(i,j) of the image; If blurring degree is not enough, then repeat

```
image[next][i][j] = (image[current][i - 1][j - 1] +  
    image[current][i - 1][j] +  
    image[current][i - 1][j + 1] +  
    image[current][i][j - 1] +  
    image[current][i][j] +  
    image[current][i][j + 1] +  
    image[current][i + 1][j - 1] +  
    image[current][i + 1][j] +  
    image[current][i + 1][j + 1])  
    / 9;
```


Assignment 1 variants

- “Bacteria colony” is a very typical Stencil pattern, the grid of positions is partitioned
- “Epidemics simulation” is not a Stencil pattern because of persons mobility ! It is best approached by data decomposition - partitioning of the set of persons, not the grid of positions