Technische Universität München
Lehrstuhl Informatik V
M. Bader, A.Pöppl

WS 2015/16
Example assignments
January, 18th 2016

HPC - Algorithms and Applications WS 15/16

Questions from previous exams

This worksheet contains exercises that have been asked in previous exams.

1 Parallel Sparse-Matrix Computation

In Figure 1, you see two suggestions to distribute the elements of a sparse matrix (a 13×13 matrix with 53 non-zeros) to four parallel partitions to be used for distributed-memory parallelisation.

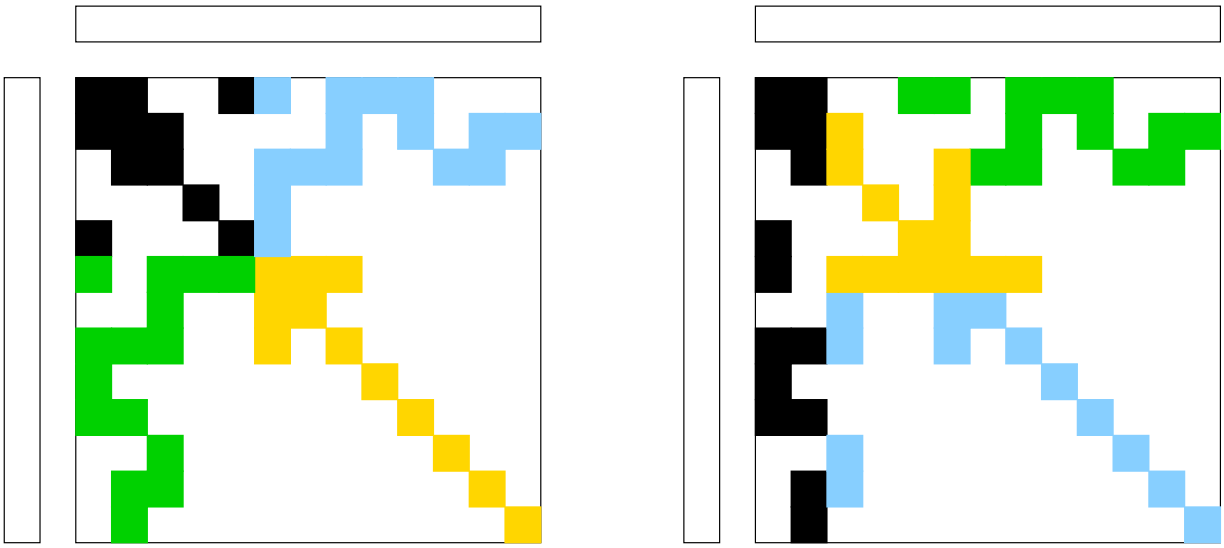


Figure 1: Two suggestions for the partitioning of a sparse matrix

a) For each of the matrix distributions, suggest an efficient(!) distribution of the elements of the input and output vectors – mark the elements in Fig. 1 by the characters 'k' (black), 'b' (blue), 'g' (green), and 'o' (orange/yellow), respectively. If necessary, state which assumptions you make on the underlying problem.

Criteria:

1. vector elements should be distributed (roughly equally) to all four processes

2. reduce communication in scatter and accumulate: vector elements of input/output vector should match one of the partitions of the respective non-zeros

Assumptions on the underlying problem particularly, whether output vector will be used as input vector (then: same distribution) in an iterative process.

b) State the main goals for efficient parallelisation of sparse-matrix computation. Discuss which of the two suggested partitionings achieves the respective goals more successfully.

1. load balancing: each partitioning should contain about the same number of non-zero elements; while the left partitioning varies between 11 and 15 non-zeros, the right matrix shows an optimal load distribution (13 or 14 elements).

2. reduce communication: the communication operations to distribute the elements of the object vector (scatter) and to accumulate partial sums in the object vector should be minimized. The Cartesian distribution in the left image requires communication with at most one other process in both steps. In the right distribution accumulation involves 2–3 processors; the scatter process involves 1–3 processes.

2 Spacetree Partitioning

Figure 2 shows a small quadtree grid. The underlying problem for this exercise is to create efficient partitionings for parallelisation with distributed memory.

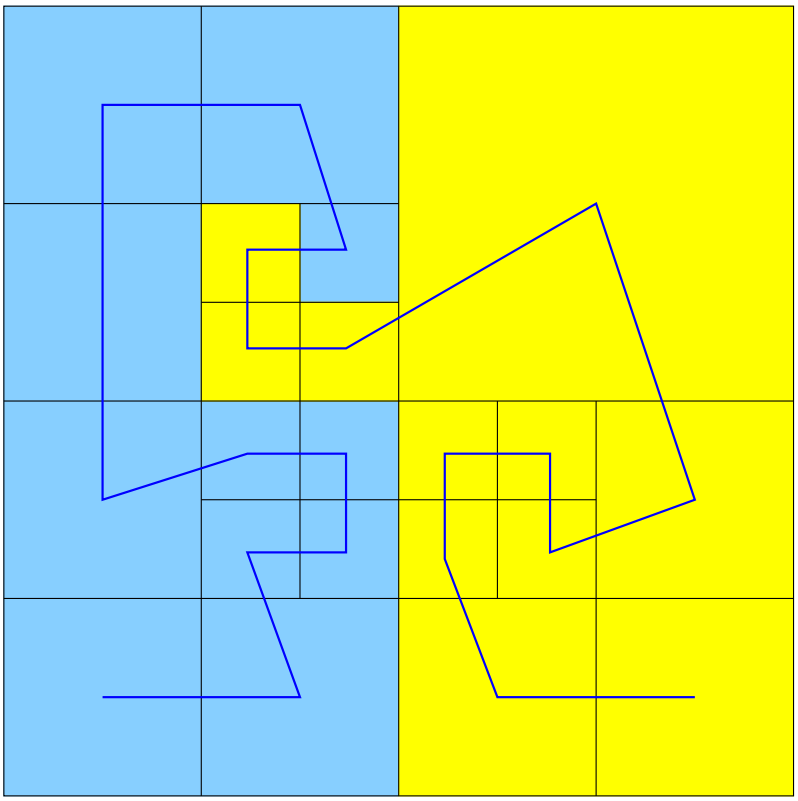
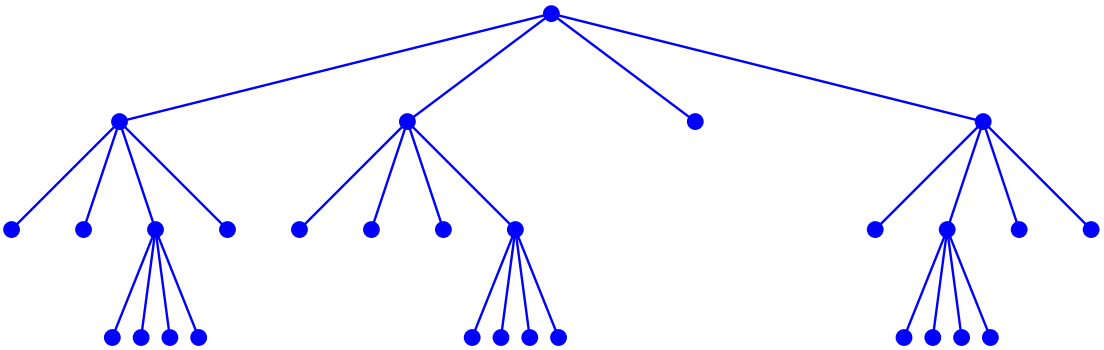


Figure 2: Adaptive quadtree grid for Exercise 2b) – sketch the REFTREE partitions here.

a) Draw the corresponding tree structure of the quadtree grid in Figure 2. The children shall be ordered in a way such that a depth-first traversal of the quadtree leads to a grid traversal in Hilbert order.



b) Shortly describe the main algorithmic steps of the REFTREE algorithm for the partitioning of tree-structured grids. Sketch the resulting partitioning into two partitions in Figure 2.

REFTREE algorithm:

1. annotate all quadtree nodes by the number of leaves (or nodes) of the tree that are descendents (children, grandchildren, etc.) of this node; requires a post-order traversal
2. determine desired size (or size range) or partitions
3. perform depth-first traversal of the node; keep track of the children(tree nodes) that should still be added to the current partition; add an entire subtree, if it contains fewer nodes; descend into tree, if it is partially contained in the current partition

c) Explain how the partitioning problem can be formulated as a graph partitioning problem. Sketch the respective graph in Figure 3.

1. use the dual grid of the quadtree grid: cells are vertices of the graph; two cells/vertices are connected by an edge, if they share a common cell edge.
2. We then require a partitioning that minizes the edge-cut (to reduce communication) while maintaining a certain balance on the number of nodes per partition (for load balancing)

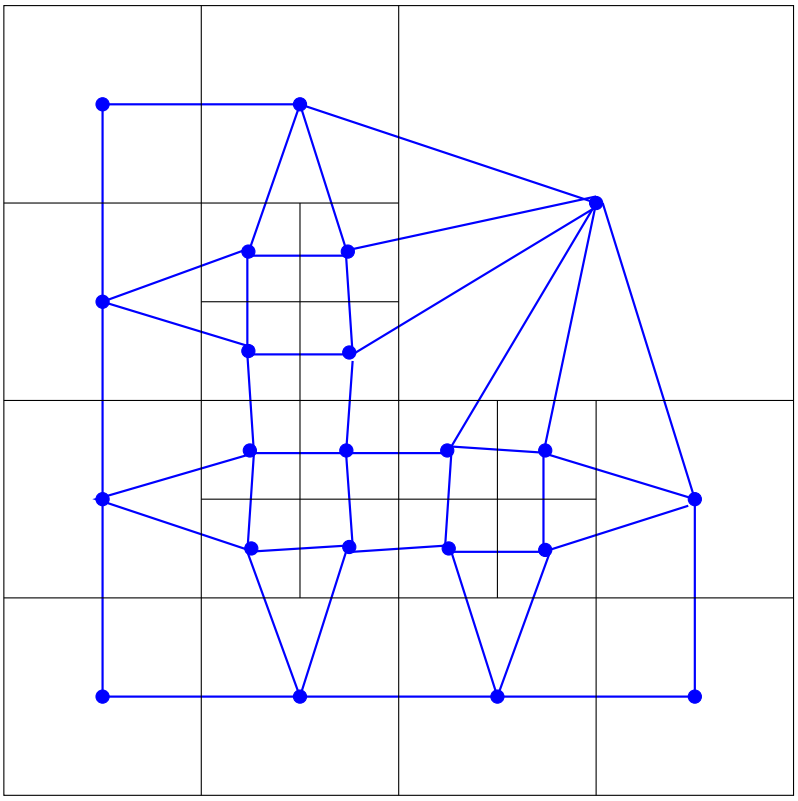


Figure 3: Adaptive quadtree grid for Exercise 2c) – sketch the graph here.

d) Give a short overview of the main steps of the multilevel graph partitioning algorithm discussed in the lecture (roughly two sentences for each of the main algorithmic phases are sufficient).

1. graph coarsening phase: pairs of connected edges, found by a heuristic matching algorithms, are combined to produce coarser and coarser graphs. Both the vertices and the edges of the coarsened graphs are weighted by the vertices/edges they represent in the original graph.

2. on a sufficiently coarse graph, the partitioning problem is solved via a suitable heuristics

3. uncoarsening of the graph: nodes are step by step inflated; the partitionings for the finer graphs are defined by first propagating the partitioning information of the respective coarser vertex but then trying to locally improve the load balancing and edge-cut. For that purpose, vertices on the edges are examined whether they would lead to a balancing or edge-cut gain, when moved to another partition.

3 Structured Grids (≈ 5+2+3 = 10 points)

Given is the following relaxation scheme on a 2D Cartesian mesh – the unknowns are located on the Cartesian grid points and stored in an array `u(i,j)` (using column-major order for the storage scheme; `i` is the column index).

```
for (int i=1; i<n; i++)
   for (int j=1; j<n; j++) {
      u(i,j) = u(i,j) - 0.125*( u(i+1,j) + u(i-1,j) +
               u(i+1,j-1) + u(i,j-1) + u(i-1,j-1) +
               u(i+1,j+1) + u(i-1,j+1) + u(i,j+1)
              );
  };
```

a) Analyse the performance of this relaxation scheme in the Roofline Model. Consider a small cache in comparison to the problem size n (i.e., cache size $M \ll n$). Assume that you are running the code on a machine with a memory bandwith of 10 GB/s and a peak floating-point performance of 20 GFlop/s (assume single precision for all variables and Flop/s) .

- due to the small cache, less than one row of the grid fits into cache; hence, from the 9-point stencil the three elements involving a j+1 element have not been accessed in the previous execution of the j-loop, and will thus not be in cache. Hence, we need to read 3 floats per iteration, and write one flop, leading to 16 bytes of memory transfer.
- for each execution of the inner loop body, we perform 7 additions, 1 subtraction and 1 multiplication, i.e. 9 Flops. Hence, we have an arithmetic intensity of 9/16.
- Exploiting the full memory bandwidth of 10 GB/s, we can thus execute at most $90/16 \approx 5.5$ GFlop/s. Our code is memory-bound on the given machine.

b) What is the best performance you could achieve by using a (perfect) loop blocking implementation for this relaxation scheme?

In the best case, we only need to load and store each u(i,j) once, which leads to 1 read and 1 write access, hence 8 bytes. Thus, the arithmetic intensity doubles, such that we can reach up to $180/16 \approx 11$ GFlop/s

c) Explain shortly how the cache line transfers could be reduced for the case of performing multiple relaxations by using the Cache oblivious algorithm by Frigo et al.

A blocking of the time loop, similar to b), leads to problems with outdated values of variables. Unless this is allowed for the applied methods, we need to reduce the size of the blocks by one layer per iteration step (hence, obtain trapezoidal blocks).

Providing a sketch of the solution is helpful (and considered OK to describe the cache-oblivious approach).

4 Band matrices in CUDA

A sparse matrix $\mathbf{B} \in \mathbb{R}^{N \times N}$ is called *band matrix* if its sparsity pattern fulfills the following rule: For certain k_{start}, k_{end} with $-N \ll k_{start} < k_{end} \ll N$:

$$\mathbf{B}_{ij} \neq 0 \text{ only if } j \geq i + k_{start} \text{ and } j < i + k_{end}$$

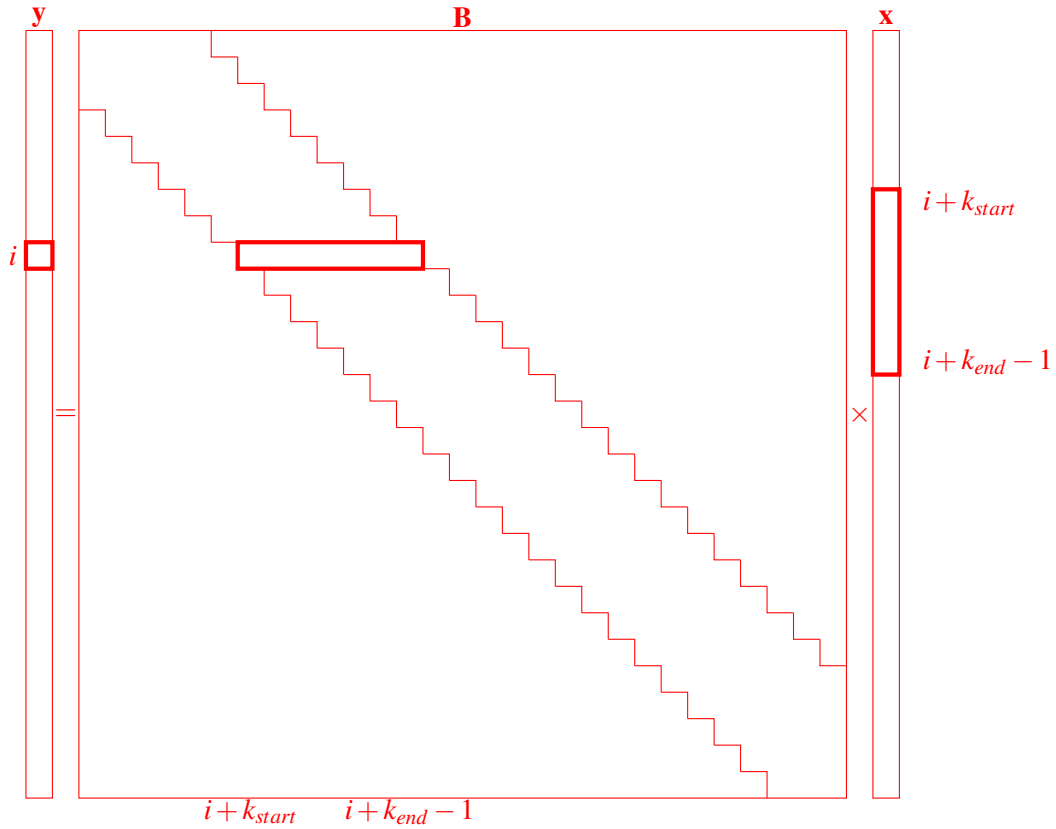
Here, a band matrix is compressed efficiently into a 1D-array `b[]` of size $N \cdot (k_{end} - k_{start})$, that contains the successive elements $(\mathbf{B}_{ij})_{j=i+k_{start},\ldots,i+k_{end}-1}$ for each row $i \in \{0, 1, \ldots, N-1\}$. Any array element with an invalid column index < 0 or $\geq N$ is assumed 0.
We implement a band matrix-vector product $\mathbf{y} = \mathbf{B}\mathbf{x}$ with $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$ in CUDA. The target architecture supports CUDA Compute Capability 2.0 or higher and contains a shared memory but no cache:

```
__global__ void band_matvec(int N, int k_start, int k_end,
                            float* b, float* x, float* y) {

  int i = TILE_SIZE * blockIdx.x + threadIdx.x;

  if (i < N) {
    float dot = 0;

    for (int k = k_start; k < k_end; k++) {
      float val = b[(k_end - k_start) * i + k - k_start];
      int j = i + k;

      if (val != 0) dot += val * x[j];
    }

    y[i] = dot;
  }
}
```

a) Draw a sketch of the band matrix $\mathbf{B}$ and the vector $\mathbf{x}$. Mark the matrix and vector elements, which are read by the kernel for computation of a single vector entry $\mathbf{y}_i$.
You can assume that the condition in line 12 of the kernel always evaluates to true.

(i) How many read accesses to $\mathbf{x}$ does the kernel execute for each block of size `TILE_SIZE`?

(ii) Is this number optimal? What's the smallest number of elements that must be read from $\mathbf{x}$ in order to compute a block of `TILE_SIZE` elements?



There are $(k_{end}-k_{start})\cdot$`TILE_SIZE` read accesses to global memory by each block in the kernel. Only $(k_{end}-k_{start}-1)+$`TILE_SIZE` entries of **x** are required though.

b) Describe shortly how usage of shared memory would reduce the amount of read accesses to global memory in the kernel. Write a line of code where you allocate the amount of shared memory required for this improvement.

We could prefetch the array **x** into shared memory instead of accessing it from global memory in each call. In the loop we would only access shared memory, which is faster than global memory, thus speeding up memory transfer.

Using a compile-time constant c with $k_{end}-k_{start}-1 \leq c \ll N$, we have to allocate an array of size at least

```
__shared__ float xs[TILE_SIZE + c];
```

c) What is coalesced memory access? Are read accesses to the array `b[]` coalesced in the original kernel? Explain your answer.

Coalesced memory access means memory accesses by a warp are executed concurrently and not serialized. Certain conditions must be met for that: all threads in a warp have access to the same 32B, 64B or 128B memory chunk. Access had to be aligned with stride 1 in the first CUDA architectures (cc 1.0, cc 1.1, ..), later architectures (cc 2.0, ..) allow partial coalescing for overlapped read access, small strides and misalignment.
Since the array `b[]` in the kernel is accessed by neighboring threads with stride $k_{end}-k_{start}$, coalescing will not happen if $k_{end}-k_{start} \gg 1$.

d) We assume, that the order of elements in the array `b[]` can be chosen freely. Describe in two sentences, how under these circumstances coalesced memory access to `b[]` is achieved. Which order would you have to choose for `b[]`? Explain, how line 9 in the original code has to be changed accordingly:

```
float val = b[(k_end - k_start) * i + k - k_start];
```

The simplest solution is to change indexing of `b[]` to `b[N * (k - k_start) + i]`. This requires that the matrix elements in `b[]` are stored in diagonal-wise order rather than row-wise order.

e) In order to measure the compression quality of a sparse matrix format, we count the number of entries that are zero and not stored explicitly by the format. Describe a sparse matrix $\in \mathbb{R}^{N \times N}$, that cannot be compressed by the band matrix format, but is compressed optimally by the ELLPACK format.

A permutation matrix with an entry in the lower left and an entry in the top right corner will always degenerate in the band matrix format. The ELLPACK format stores only 1 non-zero per row however and handles it efficiently.