Technische Universität Berlin

School IV – Electrical Engineering and Computer Science
Chair Communication and Operating Systems (KBS)
http://www.kbs.tu-berlin.de

Integrated Course

Parallel Programming

J. Schönherr

in Summer Semester 2013

Assignment 4

Issued: Wednesday, 29th May 2013

Due: Wednesday, 19th June 2013

Information

Please upload your solution via the ISIS page of this course until 23:55 of the due date. Your upload should consist of a PDF document covering the theoretical parts and a zip/tar.gz archive with your source files. Please list the names and matriculation numbers of all group members inside your uploaded documents.

Exercise 1 – Going for Speed

Your goal is a parallel program for dense matrix-matrix multiplication that is as fast as you can get it, exploiting different levels of parallelism.

Some sources are provided as a common base at:

https://svn.kbs.tu-berlin.de/svn/ps/public/ss13/pp/a4/matrix

(Please do not change functions inside matrix.c. Instead, move individual functions out of matrix.c into a new c-file and modify them there.)

In no particular order (though the order below works fine), you should:

- increase ILP by
  - compiler optimizations,
  - loop optimizations (cache access patterns),
  - optionally cache size optimizations,
  - optionally other code optimizations;
- optionally increase DLP by
  - using SIMD instructions;
- increase TLP by
  - using OpenMP,
  - optionally taking care of NUMA while using OpenMP.

In your submission document your incremental optimizations along with benchmark results comparing the different steps. Also, describe your benchmark setup and include “optimizations” when they turn out not to optimize anything.

Remarks:

- You should make sure, that your optimized versions produce the same results as the basic version.
- Do not optimize towards one specific matrix size. Your benchmarks should include matrices of different characteristics.
- Include only the core calculation in your time measurements, i. e., no disk access and no memory allocation.
- Do not only report absolute times, but also achieved throughput, which we define as:

  Work Per Second (WpS) = (Work) divided by (seconds needed)

  Work = (rows of A) times (columns of A) times (columns of B) divided by (1000*1000)

  This should be something similar to MFLOPS.
- OpenMP benchmarks should be done on istanbul. Please try not to disturb the measurements of others (use `top` to see if someone is doing something). On the other hand, please keep your measurements short, e. g., less than ten minutes for a complete benchmark series. Also keep in mind, that undisturbed processing time might become rare close to the deadline.
- OpenMP benchmarks should include a speed-up curve. For that set the environment variable GOMP_CPU_AFFINITY to force a thread to CPU mapping. Then you get repeatable results on the NUMA system.

  The value below fully uses a NUMA node before putting a thread on the next node. This allows to measure speedup within a processor with 1, 2, 3, 4, 5, and 6 threads, and then NUMA scalability with 6, 12, 18, and 24 threads.

  ```
  export GOMP_CPU_AFFINITY=0,4,8,12,16,20,1,5,9,13,17,21,2,6,10,14,18,22,3,7,11,15,19,23
  ```
- Non-OpenMP benchmarks are initially best done on the pool workstations. Cache related optimizations are probably better verified on istanbul later.

Please discuss all aspects of this exercise in the discussion forum and also report WpS values as long as they were gathered in our pool or on istanbul.

Exercise 2 – Sparse Matrices

A common storage format for sparse matrices is the *compressed row storage* (CRS) format. While some variants and extended versions exist, that take advantage of special matrix layouts or aim at algorithms with lower overhead (e. g. block compressed row storage), we will stick to plain CRS.

In CRS, a *n x n* matrix *A* is defined by three vectors:

– `val` holds non-zero elements of *A* ordered row-wise,

– `col_ind` contains the column index for each value in `val`,

– `row_ptr` stores indices of `val`/`col_ind` where each row starts (and a terminal element).

For example:

$$A = \begin{pmatrix} 7 & 4 & 0 & 1 \\ 0 & 1 & -2 & 0 \\ 0 & 0 & 4 & 1 \\ 8 & 0 & 0 & 6 \end{pmatrix} \text{ results in } \begin{array}{l} \text{val} = (7\ \ 4\ \ 1\ \ 1\ \ -2\ \ 4\ \ 1\ \ 8\ \ 6) \\ \text{col\_ind} = (0\ \ 1\ \ 3\ \ 1\ \ \ \ 2\ \ 2\ \ 3\ \ 0\ \ 3) \\ \text{row\_ptr} = (0\ \ 3\ \ 5\ \ 7\ \ 9) \end{array}$$

Note, that this definition uses zero-based indices for both, `col_ind` and `row_ptr`.

(a) Sparse Matrix Vector Multiplication

For a *n x n* matrix *A* with *m* non-zero elements ($m \geq n$), CRS has a space complexity of *O(m)*. Even more importantly, a matrix vector multiplication can now be realized in *O(m)* time.

Think about how this is done and write commented/explained pseudo code, that calculates $c = A \cdot b$ for a (dense) vector *b* and a sparse matrix *A* given in CRS as above.

(Alternatively, you can look it up somewhere, but then you miss the fun. Anyway, do not forget to cite the reference that you used.)

(b) Parallelization

Considering parallel systems with distributed memory, *A* and *b* need to be distributed somehow. As with the dense version, one could consider a row blocked layout of *A* and a replicated vector *b*. This way, we get a blocked result vector *c*.

However, due to *A* being sparse, this distribution has at least two inconveniences. Which ones? (Give a short reasoning for each.)

Describe a distribution of *A* and *b*, that addresses at least one of the identified problems.

(c) Embedding into a FEM simulation

If we use this matrix vector multiplication as part of the Jacobi method in order to solve a linear system as part of a FEM simulation, we have to repeat the multiplication several times. Each time using the (slightly modified) output as input again. (The modification can be done on the distributed output as a local operation.)

That means, we have to redistribute the output to match the required input distribution. With a blocked output vector and replicated input vector, this is a global all-gather operation.

As the matrix A was created out of a meshed physical object it has some other properties beside being sparse (and being symmetric, which we ignore here). This can be used to improve the multiplication: the structure of A is roughly equivalent to the adjacency matrix of the mesh. (With sub-matrices instead of single elements; these sub-matrices have non-zero elements if the element in the adjacency matrix is non-zero.)

Knowing this, it is possible to order the elements of the mesh so that the maximum difference of indices of connected elements is minimized.

For example, consider two variants of a 2x4 mesh:

Variant A:

Mesh: $\begin{array}{ccccccc} 1 & \cdots & 2 & \cdots & 3 & \cdots & 4 \\ \vdots & & \vdots & & \vdots & & \vdots \\ 5 & \cdots & 6 & \cdots & 7 & \cdots & 8 \end{array}$

Maximum index difference: 4

Adjacency matrix: $\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}$

Variant B:

Mesh: $\begin{array}{ccccccc} 1 & \cdots & 3 & \cdots & 5 & \cdots & 7 \\ \vdots & & \vdots & & \vdots & & \vdots \\ 2 & \cdots & 4 & \cdots & 6 & \cdots & 8 \end{array}$

Maximum index difference: 2

Adjacency matrix: $\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$

So, we do not only have a sparse matrix, but instead a sparse banded matrix. The bandwidth of the matrix A is proportional to the maximum index difference. Minimizing that difference results in a matrix with a more pronounced band structure.

Now reconsider the parallel matrix vector multiplication. Analyze the implications of the band structure on the multiplication and describe an improved version!

Also, analyze and outline the effects on the Jacobi implementation described above when using this improved version.