HPC Fall 2017 – Project 2
Gaussian Elimination Optimization

Robert van Engelen

Due date: April 11, 2017 (corrected date)

1 Introduction

1.1 HPC Account Setup and Login Procedure

See Project 1.

1.2 Download the Project Files

Next, download the project source:

```
[yourname@spear ~]$ wget http://www.cs.fsu.edu/~engelen/courses/HPC/Pr2.zip
```

The package bundles the following files:

- `Makefile`: a standard Makefile to build the project.
- `config.guess`: guess which platform
- `make.`*platform-comp*: platform- and compiler-specific files used by `Makefile`
- `cputime.h` and `cputime.c`: `cputime()` timer
- `rdtsc.h`: Intel RDTSC timer used by `cputime()` (optional)
- `gauss.c`: Gaussian elimination with pivoting

1.3 Getting Started

- Login to `spear-login.rcc.fsu.edu`
- run `module load sunstudio`
- Run `make` to build `gauss`, using the Sun compiler
- Run `./gauss` to solve a system of 1000 unknowns and print the average elapsed wall clock time for 4 runs of the solver.

1.4 Important - Do not Forget

You should `make clean` and rebuild the executables with `make` **every time** when

- you changed the content of the `Makefile`
- you changed the content of the `make.`*platform-comp* file(s)
- you change the `make` options, such as `make LAYOUT=JAGBYCOL` etc.
- you changed a `.h` header file

1.5 Project Aims

By completing this project you will be able to

1. investigate the impact of column/row-major and jagged matrix layouts on the performance of real-world algorithms, such as Gaussian elimination with pivoting (Section 2).
2. use advanced profilers to study and explain the impact of the matrix layout on the performance of the data cache (Section 3).
3. explain why array layouts impact the effectiveness of an advanced compiler auto-parallelizer and auto micro-vectorizer (Section 4).
4. parallelize the Gaussian elimination algorithm with OpenMP directives and study the impact of the OpenMP parallelization speedups (Section 5).

For this project you should write a report with your findings and submit this to the instructor for grading. Include explanations of your findings in your report. Your report must address the four parts and include the source code of the OpenMP code(s) that you wrote (you do not need to submit your source code(s) separately in an attachment).

2 Column/Row-Major and Jagged Matrix Layouts

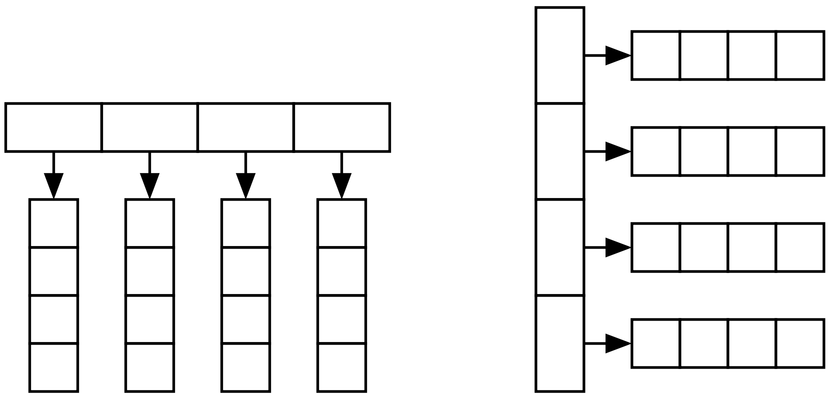
There are six different matrix layouts prepared for you. A specific matrix layout is enabled with the compiler option `make LAYOUT=`*layout*, where *layout* is

- `JAGBYCOL`: use jagged array layout (array of arrays) with heap-allocated columns, similar to arrays allocated in Java (Java has no true multidimensional arrays)
- `JAGBYROW`: use jagged array layout (array of arrays) with heap-allocated rows, similar to arrays allocated in Java (Java has no true multidimensional arrays)
- `COLMAJOR`: use column-major layout, similar to Fortran (w/o heap allocation)
- `ROWMAJOR`: use row-major layout, similar to C fixed-size arrays (w/o heap allocation)
- `COLARRAY`: use column-major layout mapped to a 1D array (w/o heap allocation)
- `ROWARRAY`: use row-major layout mapped to a 1D array (w/o heap allocation)

This command compiles `gauss` with the selected matrix layout.

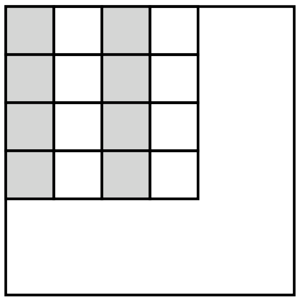
The choice of data layout can have a profound impact on the efficiency of memory references. In general, it is best to access consecutively stored array elements in sequence in loops (spatial locality) and reuse data (temporal locality). Since the algorithm is fixed in this case, we can only change the matrix layout.

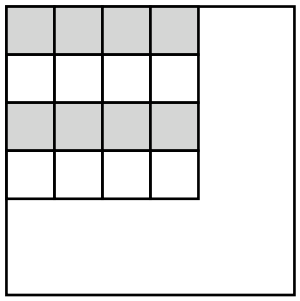
The two versions of the "jagged matrix" formats are shown below



That is, in jagged column format (`JAGBYCOL`), matrix element $a_{i,j}$ maps to column `a[j]` and element `a[j][i]`. In jagged row format (`JAGBYROW`), matrix element $a_{i,j}$ maps to row `a[i]` and element `a[i][j]`.

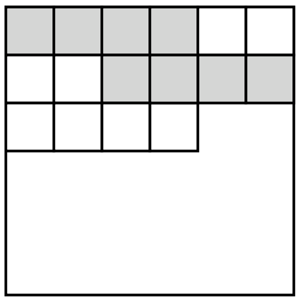
The column- and row-major matrix formats are shown below


Column major


Row major

where the matrix has a maximum size shown by the outer square. Either the row or column elements are mapped to consecutive memory locations. That is, in column major format (`COLMAJOR`), matrix element $a_{i,j}$ maps to memory $M[i + jN]$, where N is the maximum rank. Thus, row elements are stored consecutively. In row major format (`ROWMAJOR`), matrix element $a_{i,j}$ maps to memory $M[j + iN]$, where N is the maximum rank. Column elements are stored consecutively.

The "array" matrix format is shown below



where rows (`COLARRAY`) or columns (`ROWARRAY`) are placed to optimally fill the initial part of memory. That is, in `COLARRAY` format, matrix element $a_{i,j}$ maps to memory $M[i + jn]$, where n is the row rank (number of rows). in `COLARRAY` format, matrix element $a_{i,j}$ maps to memory $M[j + in]$, where n is the column rank (number of columns).

For convenience, `gauss.c` defines these matrix mappings using a macro `A(i,j,n)` to index the matrix element at $a_{i,j}$, where `n` is the matrix rank. The allocation of the matrix is automatically performed in `gauss` based on the selected layout.

Now let's move on to our algorithm! Gaussian elimination solves $Ax = b$ by reducing the system to upper triangular form $Ux = y$ and then applies backsubstitution to solve x. Pivoting is used for numerical stability (consult a textbook for more details).

We implement Gaussian elimination with pivoting in file `gauss.c` as follows

```
void gauss(MATRIX a, VECTOR x, int n)
{
  int i, j, k, maxloc;
  double maxval;
  INDEX idx;

  /* Init row index array */
  for (i = 0; i < n; i++)
    idx[i] = -1;

  for (k = 0; k < n; k++)
  {
    /* Find pivot element in k'th column */
    maxval = 0;
    maxloc = -1;
    for (i = 0; i < n; i++)
    {
      if (idx[i] == -1 && maxval < fabs(A(i,k,n)))
      {
        maxval = fabs(A(i,k,n));
        maxloc = i;
      }
    }

    /* Singular? */
    if (maxval < DBL_EPSILON)
    {
      fprintf(stderr, "Singular matrix\n");
      exit(1);
    }

    /* Relabel row of the k'th pivot element */
    idx[maxloc] = k;

    /* Reduce the rows, except pivot row and previous rows */
    for (i = 0; i < n; i++)
    {
      if (idx[i] == -1)
      {
        double fac = A(i,k,n)/A(maxloc,k,n);
        for (j = k; j < n+1; j++)
          A(i,j,n) = A(i,j,n) - fac * A(maxloc,j,n);
      }
    }
  }
  /* Row exchanges for A[][] and b[] */
#ifdef JAGBYROW
  {
    MATRIX tmp;

    /* Simply exchange the pointers to the rows, no data movement needed! */
    for (i = 0; i < n; i++)
      tmp[idx[i]] = a[i];
    for (i = 0; i < n; i++)
      a[i] = tmp[i];
  }
#else
  /* Iterate per column to exchange elements */
  for (j = 0; j < n+1; j++)
  {
    VECTOR tmp;

    for (i = 0; i < n; i++)
      tmp[idx[i]] = A(i,j,n);
    for (i = 0; i < n; i++)
      A(i,j,n) = tmp[i];
  }
#endif

  /* Solve x by backsubstitution */
  for (i = n-1; i >= 0; i--)
  {
    /* Note: b[i] is stored in A(i,n,n) */
    double sum = A(i,n,n);

    for (j = i+1; j < n; j++)
      sum = sum - A(i,j,n) * x[j];

    x[i] = sum/A(i,i,n);
  }
}
```

Note that in this code:

- macro `A(i,j,n)` refers to array `a` using an index mapping based on the selected matrix layout (selection of a layout is a compile-time option)
- vector b is stored in the column-augmented matrix a at column `n`, i.e. b_i=`A(i,n,n)`
- the matrix and vectors are indexed from element 0 to element `n-1`
- for `JAGBYROW` we used the array of pointers to our advantage: instead of moving data we can simply reorder the rows by changing the pointers in the array of pointers to rows.

For each of the six layouts compile and run `gauss` on `spear`. Remember to do `make clean` and recompile for each new layout with `make LAYOUT=`*layout* where *layout* is one of the six layouts listed at the beginning of this section. Create a six-by-two table with the observed elapsed wall-clock times for the six layouts. There will be some interesting differences in the table. We will find out more about the differences in the next task.

For these experiments we will use a matrix rank `n=1000` and 4 runs (`#define RUNS 4`), unless explicitly stated otherwise.

3 Advanced Profiling

In this part of the assignment you will determine how a matrix layout impacts the performance at the memory level.

First, to find out what the standard sampling-based profiler `gprof` reports, run `make gprof LAYOUT=`*layout*, where *layout* is one of the six layouts you want to study. This special `make gprof` command compiles and then runs `gauss` with `gprof`, see `Makefile` actions for more details.

Answer these questions about the `gprof` results for each of the six matrix layouts:

- How many times is the `gauss` solver invoked?
- What is the average running time of `gauss`?
- Is there any way with `gprof` to find out what is causing the timing differences at the cache/memory level?

Repeat this experiment and answer the same questions, but now using the Sun performance tools. Run `make prof LAYOUT=`*layout* for each layout. Ignore the warnings if there are any. Use `analyzer prof.er` to browse the sampling results by selecting the *timeline*. As above, record and compare the total cumulative time of `gauss` for the layouts.

Next, we will use hardware counter profiling to determine the cause of the performance impact of all six matrix layouts.

- Determine the cache utilization with the Sun performance analyzer's hardware counter profiling on `spear`:
    1. Modify the `Makefile` for the `prof` rules to run `collect` with option `-h`. Select L1 D-cache and L2 D-cache hit and miss counters (hint: run `collect -h` for a list of options).

2. Execute `make clean`.
3. Execute `make prof LAYOUT=`*layout* to rebuild `gauss` with the layout and run `collect`.
4. Use `analyzer prof.er` to browse the results.
5. Repeat steps 1 to 4 for each of the six matrix layouts, and write down the results in a six-by-three table showing the L1 D-cache and L2 D-cache hits and misses and the cache miss ratio.

- Usually we want to profile an application with a small, medium, and large data set. The value `n=1000` will be considered medium size. We will skip the small data set testing. Conduct your investigation once again with a large rank `n=4096` (by the way, you do **not** need to change the value of the constant `N`). Create a second six-by-three table for the large size with `n=4096`. For this experiment with large rank, set `RUNS` to `2` to reduce the time of the experiment. A run with this rank can take several minutes.
- Explain your findings by referring to the algorithm's parts and the effect of the matrix layout on the memory hierarchy and observed timings for both problem sizes. Also explain your choices for the profiling settings and options.

4 Advanced Compiler Options for Parallelization and Vectorization

In this part of the assignment we study automatic parallelization and vectorization and determine how the matrix layouts impact the ability of a compiler to automatically parallelize and vectorize the loops in the solver code. Simple loop structures with linear array access patterns can be analyzed for dependences for loop parallelization, but array accesses via pointers and indexing arithmetic can prevent a compiler from disproving dependence (assumed dependences prohibit parallelization) and prevent SIMD vectorization when array indexing is not done with a unit stride.

- Reset `RUNS` to `4` and `n=1000` in `gauss.c` (the original values).
- Use `suncc` with auto-parallelization options `-xautopar -xdepend` and `-xloopinfo` and auto-vectorization option `-xvector=simd` to optimize `gauss` on `spear` using automatic parallelization and vectorization.
- Use the `er_src gauss` command to find out more about the resulting optimizations.
- For each of the six matrix layouts, determine how many loops are parallelized and how many are vectorized in function `gauss`. Also for all six matrix layouts determine the

timing of the code using 1, 2, 4, 6, 8, 10, 12, 14, and 16 threads by setting environment variable `OMP_NUM_THREADS` (e.g. using the command e.g. `setenv OMP_NUM_THREADS 4`).

- What are the speedups (or slowdowns), i.e. how much faster/slower does that code run compared to the best serial version? Note: the optimal serial version may not have the same matrix layout as the best parallel version!

5 OpenMP Parallelization

In this final part of the assignment we will explicitly parallelize the function `gauss` and report the speedups. Please take the time to study the code and think carefully about the placement of OpenMP constructs to parallelize the code. Speedups will be achieved, but require some effort to change the algorithm's part(s).

- Annotate the `gauss` function with OpenMP directives to parallelize the algorithm. Note that you may have to rewrite parts of the function to support parallel execution.
- The `gauss` function should use a single parallel region, that is, one fork-join. Use the lecture notes on *OpenMP* and *Synchronous Computing Examples. Particularly study the Gaussian Elimination HPF example which can be used as guidance to rewrite backsubstitution by interchanging the loops for parallelization* (therefore not requiring a parallel inner loop that causes performance reductions.)
- Use `suncc` option `-xopenmp` and `-xloopinfo` to enable OpenMP and to show loop parallelization info.
- If necessary, use `er_src` to inspect the optimizations.
- Verify that your implementation works correctly by running it on small matrices of rank `n=10`. If necessary, print the matrix in each reduction stage for debugging and to see what happens. Note: you must use `setenv OMP_NUM_THREADS 4` to use four threads.
- To verify how many threads are running your solver, use the following code fragment:

```
void gauss(MATRIX a, VECTOR x, int n)
{
  int i, j, k, maxloc;
  double maxval;
  INDEX idx;

  /* start all threads */
  #pragma omp parallel shared(...) private(...)
  {
```

```
#ifdef _OPENMP
    #pragma omp master
    printf("Number of threads = %d\n", omp_get_num_threads());
#endif
```

- Use `make` with `suncc` on `spear` to try all six matrix layouts with your explicitly OpenMP parallelized solver and create a table with performance results for 1, 2, 4, 6, 8, 10, 12, 14, and 16 threads (set `OMP_NUM_THREADS` and `n=1000`). Compare relative speedups. Compare the speedup to the optimal sequential version (this version may use a different matrix layout).

6 Troubleshooting

- Always run `make clean` before compiling the code with new compiler options and settings.
- Use `suncc` option `-g` to enable `er_src` and Analyzer source code views.
- Use OpenMP wisely, don't forget barriers and critical sections. Incrementally expand your parallel region to eventually encompass all of the code in the `gauss` function. By gradually extending the scope and testing each time you make a change, you will know when and where things start to go wrong. Use the examples we discussed in class.
- Use `setenv OMP_NUM_THREADS 4` to set the number of OpenMP threads to 4 for example.
- If you are having trouble accessing man pages, try `setenv MANPATH ${MANPATH}:`

– *End.*