

HPC Fall 2008 – Project 2

Parallel Gaussian Elimination

Robert van Engelen

Due date: November 18, 2008

1 Introduction

1.1 Account and Login Information

For this assignment you need an SCS account. The account gives you access to `pamd`, `gp004`, and `tempest`. Contact the instructor if you do not have an SCS account.

1.2 Setup

If you use `tcsh` as your main shell then edit your `.tcshrc` file in your home directory, or edit the `.cshrc` file. If you use `bash` then edit `.bashrc`. Add the following lines:

```
# Set paths to Sun compilers
setenv MANPATH ${MANPATH}:/opt/sun/sunstudio12/man:/opt/SUNWspro/man
set path = ( /opt/sun/sunstudio12/bin $path )
```

1.3 Download

Next, download the project source code from

```
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`: determines the 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()`
- `gauss.c`: Gaussian elimination algorithm with pivoting

1.4 Getting Started

To get started with this assignment and try out the package content, follow these steps:

- Open a terminal and login to `gp004` with `qlogin -q gp-int@gp004 -l sunstudio`
- Run `make COMP=suncc` to build `gauss`, using the Sun compiler
- Run `./gauss` which solves a system of 1000 unknowns and prints the average elapsed wall clock time (for 4 runs of the solver).

1.5 An Important Note

When one or more of the following conditions apply, you must `make clean` and rebuild the executables with `make` before testing

- when you change compilers with the `make COMP=compiler` option
- when you change to another platform to run the experiments (e.g. `gp004` or `tempest`)
- when you changed the content of the `Makefile`
- when you changed the content of the `make.platform-comp` files
- when you changed a `.h` header file

1.6 Project Aims

This assignment is subdivided in several parts:

- The first part is to investigate the effect of column/row-major and jagged matrix layouts in memory on the performance of Gaussian elimination (Section 2). This will familiarize you with the impact of array layouts and iterations over these arrays on performance.
- In the second part we will use advanced profilers to study and explain the impact of the matrix layout on cache performance (Section 3). You will learn how to use profiling techniques to determine memory/cache hotspots.
- In the third part we will use auto-parallelization and auto-vectorization to speed up the performance of the solver. We also try to explain why the matrix layout impacts the effectiveness of the auto-parallelizer/vectorizer (Section 4).
- Finally in the fourth part we will explicitly parallelize the solver with OpenMP directives and study the speedup (Section 5).

For this project you should write a single report with your findings and submit this to the instructor for grading. Include explanations of your findings in your report. Your report must address all tasks listed in the sections below and also list the source code of the OpenMP version that you wrote. You do not need to submit your source code(s).

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 COMP=compiler`, where *layout* is

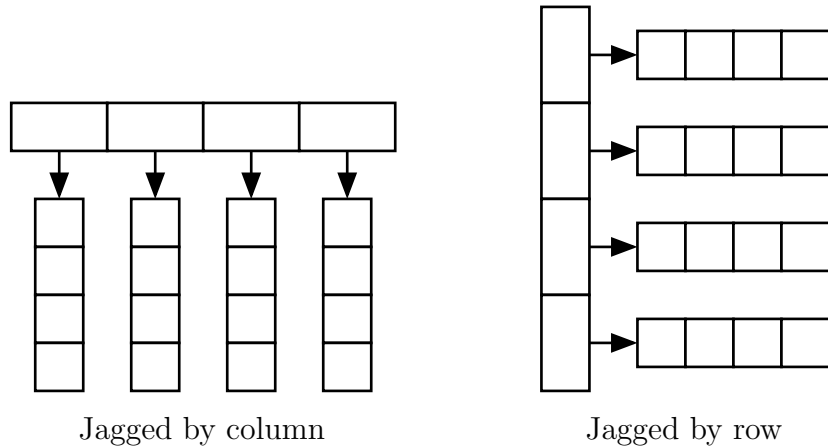
- JAGBYCOL: use jagged array layout (array of arrays) with heap-allocated columns, similar to arrays allocated in Java (Java has no true multidimensional arrays)
- JAGBYROW: use jagged array layout (array of arrays) with heap-allocated rows, similar to arrays allocated in Java (Java has no true multidimensional 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 and compiler of choice. For the `compiler` option you must use `suncc` to ensure consistency in the optimizations and timings obtained.

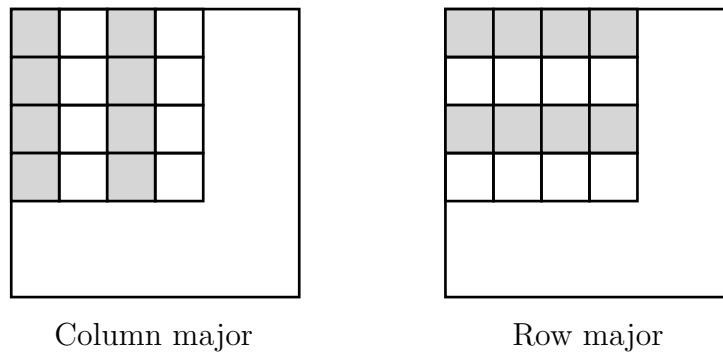
The choice of data layout can have a profound impact on the efficiency of memory references. In general, it is best to access consecutively stored array elements in sequence in loops (spatial locality) and reuse data (temporal locality). Since the algorithm is fixed in this case, we can only change the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

The two versions of the “jagged matrix” formats are shown below



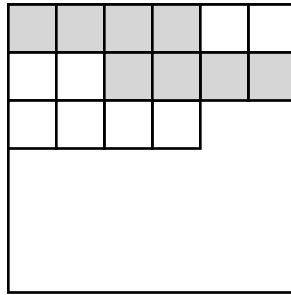
That is, in jagged column format (JAGBYCOL), matrix element $a_{i,j}$ maps to column `a[j]` and element `a[j][i]`. In jagged row format (JAGBYROW), matrix element $a_{i,j}$ maps to row `a[i]` and element `a[i][j]`.

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



where the matrix has a maximum size shown by the outer square. The shaded areas are elements that 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. In row major format (ROWMAJOR), matrix element $a_{i,j}$ maps to memory $M[j + iN]$, where N is the maximum rank.

The “array” matrix format is shown below



where the shaded area are consecutively stores rows (COLARRAY) or columns (ROWARRAY). 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 ROWARRAY 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).

For your project we implemented 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 (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.

The question is, how do these layouts impact the performance of the solver?

For each of the six layouts compile `gauss` and run the code on `gp004`, compiled as usual with `suncc`. Repeat the experiment on `tempest` using `suncc`. Remember to do `make clean` and recompile for each new layout with `make COMP=suncc LAYOUT=[layout]`. Create a six-by-two table with the observed elapsed wall-clock times for the six layouts for the two machines. There will be some interesting differences in the table. We will find out more about the differences in the next task.

Note: for this task do not change the compiler optimization options, only use the preset options defined in the makefiles of the Pr2 package.

For all experiments, we will use a matrix with 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 the matrix layouts impact the performance at the memory level.

To find out what the standard sampling-based profiler `gprof` reports, run `make gprof COMP=suncc LAYOUT=layout` on `tempest`, where *layout* is one of the six layouts. This special `make gprof` command compiles and then runs `gauss` with `gprof`.

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 total cumulative running time of `gauss`?
- Do you think there any way with `gprof` to find out what is causing the timing differences of `gauss` for the six different matrix layouts?

Repeat this experiment and answer the same questions, but now using the Sun performance tools on `tempest`. Run `make prof COMP=suncc LAYOUT=layout` for each layout. Use `analyzer prof.er` to browse the sampling results by selecting the *timeline*.

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 `tempest`. Modify the `Makefile` for the `prof` rules to run `collect` with option `-h`. First find out how to profile D-cache and E-cache stall cycles and D-TLB miss counts (Hint: run `collect` without arguments on `tempest` and read the Sun Profiler documentation). Use `make clean` and then `make prof COMP=suncc LAYOUT=layout` to run `collect`. Use `analyzer prof.er` to browse the results. Repeat this for each hardware counter settings for `collect` you need to use to derive statistics for D-cache, E-cache, and D-TLB for each of the six matrix layouts. Create a six-by-three table that lists the D-cache stall cycles, E-cache stall cycles, and D-TLB miss results of the `gauss` routine for each layout. Important: you may have to increase the `RUNS` to more than 4 to get more accurate results. You may also want to change the interval of the HW counter (the counter overflow value) to a smaller value to get more accurate

results, especially for infrequent events such as D-TLB miss (use `hi` or a numeric value with option `-h` as explained in `man collect`).

- Conduct your investigation with rank $n=1000$ and once again with the maximum rank $n=N-1$, thus create a second six-by-three table for $n=N-1$.
- Explain your findings by referring to the algorithm's parts and the effect of the matrix layout. Explain your choices for the profiling settings and options.

Note: hardware counter profiling currently only works on the UltraSparc machine `tempest`. Unfortunately, the installation of the Sun tools is incomplete on the `gp00x` machines and we currently cannot do any testing on these general-purpose machines.

4 Advanced Compiler Options for Parallelization and Vectorization

For this and the following parts of the assignment you need to use the Sun Studio 12 compiler documentation:

<http://developers.sun.com/sunstudio/documentation/product/compiler.jsp>

Select the C users guide, then Appendix A and B to peruse the compiler options.

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).

- Use `suncc` with auto-parallelization options `-xautopar` `-xdepend` and `-xloopinfo` and auto-vectorization option `-xvector=simd` to optimize `gauss` on `gp004` using automatic parallelization and vectorization. Use `er_src gauss` 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, and 4 processors by setting environment variable `OMP_NUM_THREADS` to 1, 2, and 4 respectively (e.g. using `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.

5 OpenMP Parallelization

In this final part of the assignment we will explicitly parallelize the function `gauss` and report the speedups.

- 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 the Gaussian Elimination HPF example should be used as guidance to deal with the parallelization of backsubstitution. Use `suncc` option `-xopenmp` and `-xloopinfo`. 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 cores. To verify how many threads are running your solver, use:

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

    #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 `gp004` to try all six matrix layouts with your explicitly OpenMP parallelized solver and create a table with performance results for 1, 2, and 4 threads (set `OMP_NUM_THREADS` and `n=1000`). Increase `RUNS` if needed to improve accuracy of the timings. Compare relative speedups. Compare the speedup to the optimal sequential version (this version may use a different matrix layout).

6 Problem-shooting

- 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.

End.