Parallel Jacobi Iteration Poisson Approaches

Notes and Samples on OpenMP CPU/GPU

20th August, 2021

Ed Chen

1 Mathematical Formulations

1.1 Poisson Discretization

The poisson equation is the standard example of an elliptic PDE that has applications in various physics fields. It is given by the following general equation and in two-dimensions is expressed as

$$\begin{aligned} \Delta u &= f \\ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} &= f(x,y) \end{aligned} \tag{1}$$

We consider this problem across the simple domain of the unit square, $0 < x, y < 1$ and to necessitate a well defined solution, specify values of u on the boundary of Ω.

$$\Omega = \{(x,y) | 0 < x, y < 1\} \tag{2}$$

$$u(x,y) = 0 \text{ on } \partial\Omega \tag{3}$$

To discretize the problem, we use $(n+1)^2$ grid points, denoting the approximate solution as $u_{i,j}$ at the point $(i * h, j * h)$ where h is the grid spacing. To derive the stencil, use the FDM expression for second derivatives

$$\frac{d^2 u}{dx^2} \approx \frac{u(x+\Delta x) - 2u(x) + u(x-\Delta x)}{(\Delta x)^2} + \mathcal{O}(\Delta x) \tag{4}$$

which when applied to the two derivatives we are concerned with yields

$$\partial_{xx} u_{i,j} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \qquad \partial_{yy} u_{i,j} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} \tag{5}$$

and summing in accordance with the original PDE yields

$$\begin{aligned} \Delta u_{i,j} &= \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} &&= f_{i,j} \\ \Delta u_{i,j} &= \frac{-4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{h^2} &&= f_{i,j} \end{aligned} \tag{6}$$

which we notice can be expressed as a linear system of n^2 equations with n^2 unknowns. Expressed in the form $Au = b$, we have A expressing the coefficients generated by the laplacian operator, u as a vector of unknowns, and b as a vector of corresponding boundary terms. If we want to account for time, say t_m with timestep k, we can use the explicit Euler algorithm

$$\frac{u_{i,j}^{m+1} - u_{i,j}^m}{k} = \Delta u_{i,j} \tag{7}$$

$$u_{i,j}^{m+1} = \left(1 - \frac{4k}{h^2}\right) u_{i,j}^m + \frac{k}{h^2}(u_{i-1,j}^m + u_{i+1,j}^m + u_{i,j-1}^m + u_{i,j+1}^m) \tag{8}$$

To then formulate the system, we choose an ordering of the grid points for a well formed matrix A, using natural column/row ordering. In other words, we'd order the column vector u as $u_{1,1}, \ldots u_{n,1}, \ldots u_{1,n}, \ldots u_{n,n}$. This yields an $(n-1)^2$ by $(n-1)^2$ block tridiagonal matrix: the main diagonal blocks are identical and the offdiagonal blocks are appropriately scaled identity matrices.

$$\begin{bmatrix} -4 & 1 & & & 1 & & & \\ 1 & -4 & 1 & & & \ddots & & \\ & 1 & -4 & 1 & & & \ddots & \\ & & 1 & -4 & \ddots & & & 1 \\ 1 & & & \ddots & -4 & 1 & & \\ & \ddots & & & 1 & -4 & 1 & \\ & & \ddots & & & 1 & -4 & 1 \\ & & & 1 & & & 1 & -4 \end{bmatrix} \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{4,1} \\ u_{1,4} \\ u_{2,4} \\ u_{3,4} \\ u_{4,4} \end{bmatrix} = h^2 \begin{bmatrix} f_{1,1} \\ f_{2,1} \\ f_{3,1} \\ f_{4,1} \\ f_{1,4} \\ f_{2,4} \\ f_{3,4} \\ f_{4,4} \end{bmatrix} \tag{9}$$

For instance, the general structure for the $n = 4$ case can be seen by the linear system above. Notice whenever we are concerned with an $u_{i,j}$ on the boundary, where $i = 1, n$ or $j = 1, n$ some of the $u_{i\pm1,j\pm1}$ values will be on the boundary and thus is 0.

To solve this system, we could use any of the typical algorithms for dealing with matrices, but as with the case of numerical PDEs, problems that scale quickly are often enormous and cause bad runtimes and memory problems. Even a $n = 100$ problem for instance, will yield 10^4 equations and unknowns N, and 10^8 entries in A. Using numerical linear algebra, we can capitalize on the structure of the sparse matrix itself and employ an iterative method. Essentially, we are starting with a guess u_k, iterating it through a process to get u_{k+1} until we eventually get a residual change rate that is below some tolerance that we have preset.

1.2 Iterative Methods

We seek to use a simple iterative method to solve the linear system, Jacobi's method. Jacobi is an indirect iterative method in that every step of the algorithm reduces error by some factor ρ. In serial, Jacobi takes $\mathcal{O}(N^2)$ time, and theoretically takes $\mathcal{O}(N)$ on a parallel random access machines with N storage for N processors.

Indirect methods operate on each point's nearest neighbors, so we can only propagate a grid point's value over to its' neighbors in a successive iteration. Due to the solutions' dependency on the f values, it then takes n steps to propagate all values across the grid, necessitating roughly $n = \sqrt{N}$ steps to get a good solution. In other words, we need to make sure we are doing enough iterations to propagate the information in interior grid points across the whole grid. Other iterative methods such as FFT and multigrid move information in larger steps instead of just its neighbors and implicate interesting behaviors in parallelization.

For Jacobi's algorithm we can take the discrete formulation of the Poisson from (6) and use it towards an iterative approach, solving for each successive $u_{i,j}$. We use the standard initial guess of $u_{i,j} = 0$

$$\begin{aligned} u_{i,j} &= \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + h^2 f_{i,j}}{4} \\ u_{i,j}^{k+1} &= \frac{u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} + h^2 f_{i,j}}{4} \end{aligned} \tag{10}$$

To discern the convergence rate, we use the l^2 norm on the error values of each grid point

$$\text{error}(k) = \sqrt{\sum_{i,j}(u_{i,j}^k - u_{i,j})^2} \tag{11}$$

Working this out, it can be seen that error decreases by a factor of $\rho < 1$ dependent on dimension, $\text{error}(k) <= \rho(n) * \text{error}(k-1)$. So, convergence gets slower for larger problem size with the number of step sizes proportional to n^2, making serial complexity $\mathcal{O}(n^2)$.

This is quite straightforward to implement in parallel, updating every grid point independently, giving a $\mathcal{O}(N)$ complexity. In practice, this would be done by blocking the grid and assigning appropriately to different processors, having only necessary communication on the block boundaries to its neighbors. Further variations on this standard iterative method are common, for instance SOR, successive overrelaxation, which takes advantage of already updated grid points to tile the grid in an alternating fashion.

2 Code Implementation

2.1 Iteration Loops

So in essence, the core Jacobi iteration process can be done in 3 major loops: one for storing the previous grid values, one to calculate the updated residuals and doing the actual iteration portion, and one to calculate the normed error to determine convergence. Without SOR, we will need to store both the previous and the current grid points. The primary data structures we need are then u^k, u^{k-1}, and f. All of these are column vectors with one entry per gridpoint, so with n and m points on x and y respectively, we have the necessary memory allocation of $3 * (n * m * 8\text{bytes})$. For reference, this is roughly 240gb at 100k gridpoints and 25gb at 32k gridpoints. Hence, this is allocated and appropriate macros created. The macros seem to mess up memory accesses however, and weren't used.

```
#define U(i,j) u[(i)*n+(j)]
#define Uprev(i,j) Uprev[(i)*n+(j)]
#define F(i,j) f[(i)*n+(j)]

double* __restrict Uprev = (double *) malloc(n*m*sizeof(double));
double* __restrict u = (double *) malloc(n*m*sizeof(double));
double* __restrict f = (double *) malloc(n*m*sizeof(double));
```

As described earlier, we initialize our guess to be $u_{i,j}^1 = 0$ and also initialize $f_{i,j}$ to its appropriate values for the b vector, accounting for the possibility of a different step size on the different axis. Above, we have it as h^2 but notice how this wouldn't be the case in (6) if Δx and Δy are unequal.

```
*dx = 2.0 / (n-1);
*dy = 2.0 / (m-1);

#pragma omp parallel for collapse(2)
for (j=0; j<m; j++){
    for (i=0; i<n; i++){
        xx = -1.0 + *dx * (i-1);
        yy = -1.0 + *dy * (j-1);
        u[i * n + j] = 0.0;
        f[i * n + j] = -(1.0 - xx*xx) * (1.0 - yy*yy) - 2.0 * (1.0 - xx*xx) - 2.0 * (1.0 - yy*yy);
    }
}
```

Immediately, notice we can introduce some OpenMP pragmas here to accelerate even the population of initial arrays. At scale, doing such simple loops in serial will still be costly. `#pragma omp parallel for collapse(2)` will then parallelize both loops, automatically making i, j private variables as to not interfere among the different threads. To affirm the number of threads that are in use, we call the dynamic runtime routine `omp_get_num_threads()`.

```
#pragma omp parallel
{
    #pragma omp master
        printf("\nNum Threads:      %d", omp_get_num_threads());
}
```

Inside the main iteration loop, we then need to do the 3 main loops in succession. To negotiate a more useful process when analyzing our performance, we also introduce a `maxit` upper bound on the number of iterations. Hence, we can choose to let the problem run its course with an appropriate tolerance or rely on the max iterations for profiling. Next, for storing the past values, this is relatively straightforward,

```
while (k < maxit && error > tol) {
    for (j=0; j<m; j++){
        for (i=0; i<n; i++){
            Uprev[i * n + j] = u[i * n + j];
        }
    }
    ...
}
```

The main loop are primarily then the residual calculation and error calculation. On the iteration or residual calculation loop, we need a reduction for the aggregated error, and use the `default(none)` construct to assist with a better understanding of private and shared variables, setting the variables that are redefined within the loop to private variables.

```
#pragma omp parallel for default(none) reduction(+:error) private(i, j, resid) shared(n, m, Uprev, u, f,
↪  ax, ay, b)
for (j=1; j<m-1; j++){
    for (i=1; i<n-1; i++){
        resid =(ax * (Uprev[(i-1) * n + j] + Uprev[(i+1) * n + j])
              + ay * (Uprev[i * n + (j-1)] + Uprev[i * n + (j+1)])
              + b * Uprev[i * n + j] - f[i * n + j]) / b;
        u[i * n + j] = Uprev[i * n + j] - resid;
        error = error + resid * resid;
    }
}
```

Likewise, a similar process can be used for the true error calculation, which directly uses the true solution of $(1-x^2)(1-y^2)$. This is a relegated to a separate function so it can be called afterwards.

```
#pragma omp parallel for private(i, temp) reduction(+:error) reduction(max:max_val)
↪  reduction(min:min_val) collapse(2)
for (j=0; j<m; j++){
    for (i=0; i<n; i++){
        xx = -1.0 + dx * (i-1);
        yy = -1.0 + dy * (j-1);
        temp = u[i * n + j] - (1.0 - xx*xx) * (1.0 - yy*yy);
        error += temp*temp;
        if (u[i * n + j] > max_val) { max_val = u[i * n + j]; }
        if (u[i * n + j] < min_val) { min_val = u[i * n + j]; }
    }
}
```

Note that `reduction(op:var)` can only accept one operation that is carried across a variable list, if different operators are needed in the same pragma they must each have their own reduction clause. In summary, we have the methods

```
void jacobi (int n, int m, double dx, double dy, double* __restrict u, double* __restrict f, double tol,
↪  int maxit);
void initialize (int n, int m, double *dx, double *dy, double* __restrict u, double* __restrict f);
void error_check(int n, int m, double dx, double dy, double *u, double *f, double* sol_error, double*
↪  sol_max, double* sol_min);
```

2.2 Driver Code

To make performance profiling easier, we'll put some options to run batches of problems and output the runtime. To run on various thread on problem sizes, we use a loop that increases problem sizes on powers of 2, and run a number of threads in multiples of 5 up to 40.

```
if (!single) {
    for (ps = 0; ps < 1; ps ++){
        n *= 2; m *= 2;
        printf("\nProblem Size:     %dx%d ===========\n", n, m);
        for (np = 1; np < 9; np ++){
            if (np != 0){
                omp_set_num_threads(np * 5);
            } else {
                omp_set_num_threads(1);
            }
            res[ps][np-1] = driver(ps);
        }
    }
} else {
    omp_set_num_threads(40);
    driver(n);
}
```

To prevent the need for constant recompilation we use parsed in command line arguments to toggle between batch and single runs, with an opt argument to print out the full progress statements or not. A straightforward memory bound calculation checks if the problem size will be too large.

```
double gb = ((float) n * (float) m * 8 * 3) / 1e9;
if (argc >= 3){
    opt = atoi(argv[1]);
    single = atoi(argv[2]);
} else {
    opt = 0;
    single = 0;
}
printf("%i, %i\n", opt, single);
printf("Needs %f GB of memory\n", gb);
if (gb > 150) { printf("Too large problem size"); return 1;}
```

The driver code then only needs to return the runtime calculated with the dynamic runtime routine `omp_get_wtime();`

```
double driver(int s) {
    double dx, dy;
    double runtime, sol_error, sol_max, sol_min;
    tol = 1e-15;
    maxits = 100;
```

```
    double* __restrict u = (double *) malloc(n*m*sizeof(double));
    double* __restrict f = (double *) malloc(n*m*sizeof(double));

    initialize(n, m, &dx, &dy, u, f);
    runtime = omp_get_wtime();
    jacobi(n, m, dx, dy, u, f, tol, maxits);
    runtime = omp_get_wtime() - runtime;
    printf("\nElapsed Time:     %g\n", runtime);

    error_check(n, m, dx, dy, u, f, &sol_error, &sol_max, &sol_min);
    if (opt) {
        printf("Sol Error:        %g\n", sol_error);
        printf("Sol Max:          %g\n", sol_max);
        printf("Sol Min:          %g\n", sol_min);
    }
    return runtime;
}
```

2.3 Compiling and Running

So, this is going to be run on the Electra Skylake nodes, which according to *https://www.nas.nasa.gov/hecc/support/kb/electra-configuration-details_537.html* have 40 cores per node, with the following cache sizes. Since we're only concerned with shared memory programming with OpenMP right now, and not using MPI, will cap our runs at 40 threads.

```
L1 Cache    Local to each core;
            Instruction cache: 32K
            Data cache: 32K;
            Associativity: 8;
            Cache line size: 64B

L2 Cache    1 MB per core;
            Associativity: 16;
            Cache line size: 64B
            .
            .

L3 Cache    27.5 MB shared non-inclusive by the 14 cores;
            Associativity: full;
            Cache line size: 64B
```

To have the three arrays fit into the 27MB, we would need to stick just around the 2^{10} by 2^{10} problem size. Furthermore, the total memory per node including DRAM comes to 192GB, which implicates a maximum problem size of just under $90k$ by $90k$.

To compile the script, we are testing both the intel (icpc), nvidia (nvc) and gnu (gcc) compilers. As expected corporations will optimize their compilers for their own products, hence the Intel compiler is best for CPUs, the Nvidia for GPUs, and gnu being a general use less performant compiler. For the intel compiler, we use the command

```
icpc -qopenmp input_file -O3 -xCORE-AVX512 -qopt-report=3 -o output_file
```

Here, `-qopenmp` includes the OpenMP library, `-O3` turns on a more aggressive level of optimization, `-qopt-report=3` tells the compiler to generate an 'optrpt' that details optimizations it has processed, and `-xCORE-AVX512` is an Intel Skylake specific flag that optimizes for various instruction sets used in that specific architecture. Similarly, as we cannot call the processor specific flag on gnu, we use

```
gcc -fopenmp -lm -O3 -o output_file input_file
```

with `-fopenmp -lm` denoting the OpenMP and math libraries respectively. Lastly, the Nvidia compiler we would be using to target GPUs. Hence we'd opt to enable the target offloading, which offloads OpenMP pragmas to be executed on GPU threads, only using multithreaded CPU as a fallback.

```
nvc -o output_file input_file -mp=gpu -Minfo=mp
```

Here, `-mp=gpu` instructs the compiler to enable the target offloading to GPUs. Alternatively, this can be switched to multicore as needed. `-Minfo=mp` is a flag to notify the compiler to produce standard error output as well.

In order to run scripts on the supercomputing cluster, we then need to submit a job which are dictated by either batch scripts (.pbs) or interactive sessions. Syntax is consistent either way, but to submit an interactive job we use

```
qsub -I -q devel -l select=1:ncpus=40:model=sky_ele,walltime=2:00:00
```

where `-I` denotes the interactive job, `-q devel` denotes the selection of the development queue, and the `-l select=1:ncpus=40:model=sky_ele,walltime=2:00:00` denotes a limit flag which specifies the settings and model of the requested job. To check status within a queue, we can call `qstat` and filter for jobs in the development queue

```
qstat -W o=-model -W o=+'Remwallt {model=-maxw 14} rank0 Group Pri' -W shares -a devel
```

When inside the job, modules will need to be explicitly loaded, `~/.bashrc` is not typically automatically called. Almost all necessary modules will be in the default modulefiles or `/nasa/modulefiles/testing` and `/nasa/nvidia/hpc_sdk/modulefiles` folders though it is also possible to use single network installed modules. In this case, the compilers `comp-intel/2020.4.304` and `nvhpc/21.5` are being used, with gnu already loaded. To do performance analysis, we'll also need Intel advisor, `advisor/2018`, Intel Vtune, `vtune/2020.1`, and python's package manager, `miniconda3/v4` inside `/swbuild/analytix/tools/modulefiles`.

To actually run the script, we also need to use the NAS-developed `mbind` utility to bind processes and threads to CPUs, which works for both OpenMP and MPI. This is critical for performance, and currently supports both the Intel and GNU OpenMP libraries, but not the Nvidia one.

```
/u/scicon/tools/bin/mbind.x -t num_threads executable_file
```

Some other useful commands that might come in handy for working on the cluster: `ls -la -Slh` lists everything with readable sizes in decreasing order, `quota -vs` displays readable size quotas of the home directory, `lfs quota -hu ehchen /nobackup/ehchen` displays readable size quotas for the `/nobackup/` user directories, `lscpu` or `top` gives information on system CPU cores and current usage, `free -h` shows free memory in readable sizes, `nproc --all` outputs the number of cores, and `nvidia-smi` displays information on the GPUs assigned.

2.4 Preliminary Runs

So to affirm everything works as intended, we run a few small problem sizes, which seems to be consistent under various granularities and both gcc and intel compilers. Note these are all extremely small grids, falling under 1000 points per grid, well under the L3 cache on the environments we are running it on.

```
On Iter:          1000
Total Iters:      1000
Residual:         6.09530626970886e-10
Elapsed Time:     104.746
Sol Error:        0.000266013

On Iter:          1000
Total Iters:      1000
Residual:         4.68886540132381e-07
Elapsed Time:     0.944634
Sol Error:        0.0021727

On Iter:          1000
Total Iters:      1000
Residual:         4.68886540132381e-07
Elapsed Time:     0.694824
Sol Error:        0.0021727

On Iter:          1000
Total Iters:      1000
Residual:         6.09530626970903e-10
Elapsed Time:     50.1038
Sol Error:        0.000266013
```

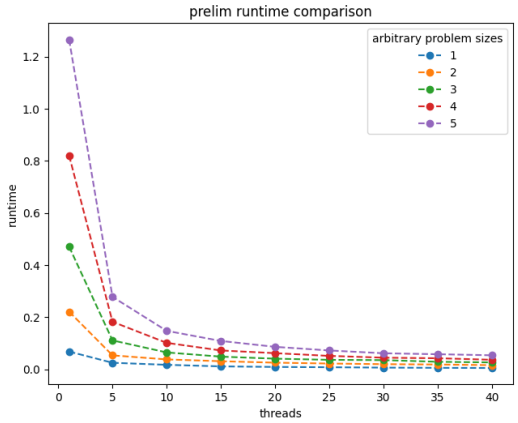
We'll also try out a batch case, running from 5 to 40 threads in multiples of 5 for problem sizes of 2^{10} to 2^{15} which take 24mb and 25gb in memory respectively. The following is truncated for brevity, but ultimately we get the runtimes in a python array format which can then be plotted and analyzed in python.

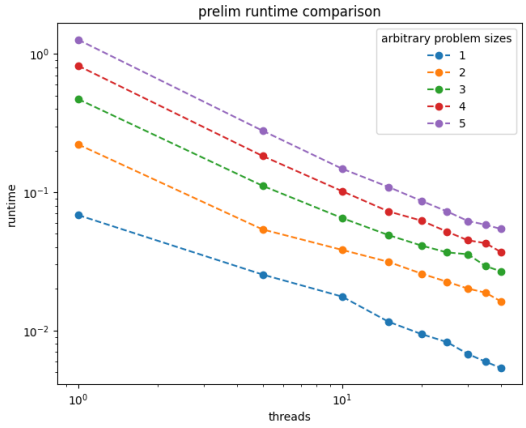
```
PBS r147i4n23:~/poisson> mbind 40
↪  ./exec_new/icpc_Aug6_new2.out
OMP: Warning #181: GOMP_CPU_AFFINITY: ignored
↪  because KMP_AFFINITY has been defined
Starting

Problem Size:     1024x1024 ===========

Num Threads:      5
Elapsed Time:     0.585538

Num Threads:      10
Elapsed Time:     0.299825

Num Threads:      15
Elapsed Time:     0.220528

Num Threads:      20
Elapsed Time:     0.176391

Num Threads:      25
Elapsed Time:     0.144266

Num Threads:      30
Elapsed Time:     0.121617

Num Threads:      35
Elapsed Time:     0.107057

Num Threads:      40
Elapsed Time:     0.0965991

Problem Size:     2048x2048 ===========

Num Threads:      5
Elapsed Time:     2.31943

Num Threads:      10
Elapsed Time:     1.16004

Num Threads:      15
Elapsed Time:     0.833148
...
Final Elapsed Times
6 problem sizes, 8 thread counts
[[0.585538, 0.299825, 0.220528, 0.176391, 0.144266,
↪  0.121617, 0.107057, 0.0965991],[2.31943,
↪  1.16004, 0.833148, 0.652833, 0.531123,
↪  0.448852, 0.385771, 0.33643],[9.2557, 4.63372,
↪  3.31336, 2.59914, 2.13118, 1.77554, 1.52153,
↪  1.3251],[37.0157, 18.5432, 13.2559, 10.4621,
↪  8.57877, 7.15679, 6.13412, 5.28984],[148.166,
↪  74.2027, 53.0499, 41.8902, 34.3909, 28.6554,
↪  24.5717, 21.2403],[591.53, 298.324, 213.84,
↪  177.059, 142.339]]
```

3 Performance Profiling

3.1 Fundamentals

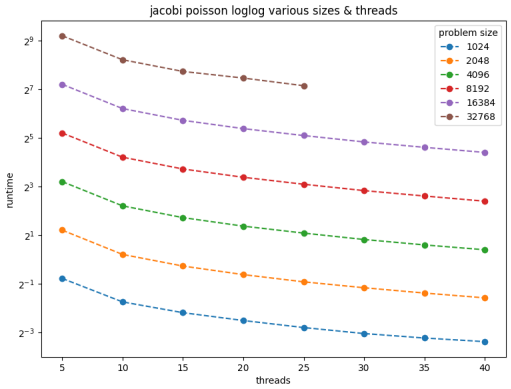
To evaluate the performance of both the compilers and algorithms used, we have a variety of methods available. On the simple side, we can directly manually calculate problem sizes and implicate various performance metrics based on their respective runtimes. On the other hand, we can also use softwares like Intel Vtune and Intel Advisor to further dissect which loops are performing at which capacities and investigate metrics on the roofline diagram. Using the preliminary values from before, we first plot metrics for the smaller problem sizes. Plotting the values on both a normal and log scale yields

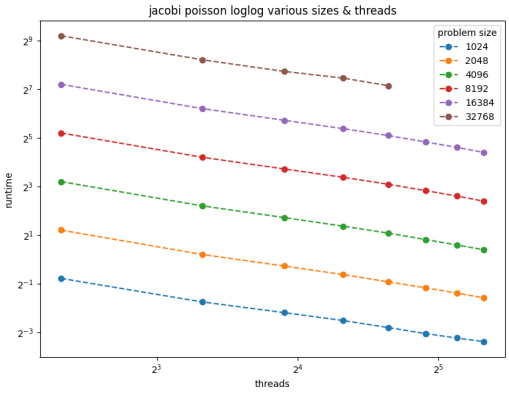




reasonable trend lines for the runtime. As observed, due to the small problem sizes and relatively straightforward algorithm, speedup is roughly linear here. The small abnormalities should mainly just be due to the slight changes as all these runtimes are exceedingly small.

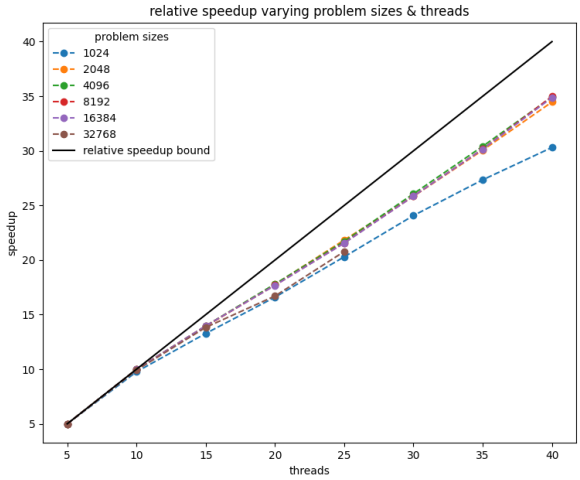
To yield some more useful results we run the batch from 5 to 40 threads in multiples of 5 and run problem sizes from 2^{10} to 2^{15}. The latter two $32k$ problem size cases were omitted just due to an expired job, but as evident the trendlines follow a pretty linear speedup in the same factor by the increased problem sizes, a sign of good scalability.



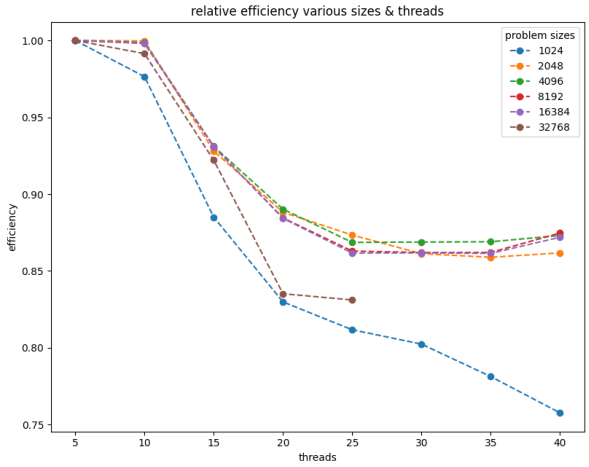


3.2 Speedup and Efficiency

Now, to further analyze performance we can draw out values of speedup and efficiency. Speedup is defined as the factor to which a parallel implementation speeds up the script as it was run to parallel, and hence defined as $S_p = T_1/T_p$, the speedup on p processors is the best serial time divided by the time taken in parallel. Evidently, no parallel implementation will have a speedup $S_p > p$, so we can treat it as an upper-bound for speedup. However, due to the large problem sizes we are dealing with, it is impractical to run this as a truly serial script, so we are benchmarking against the lowest thread count which we have data for, 5 threads.

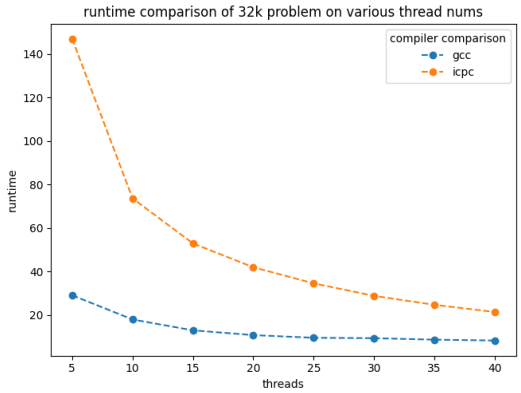


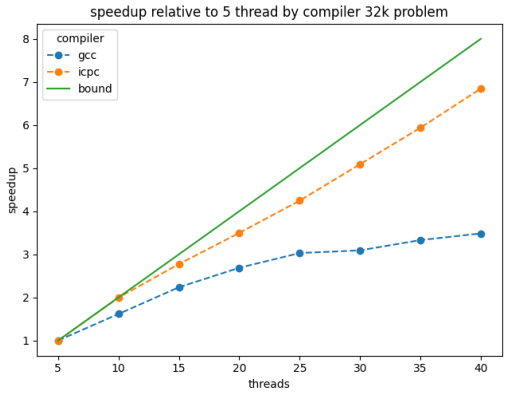
In a similar vein, we can also calculate the efficiency to see how far we are from ideal speedup, defined by $E_p = S_p/p$, in other words how efficient each processor is being when put up against the serial implementation. Again, we are using the same referential calculation with accordance to the 5 thread runtime.



3.3 Compiler Comparison

Now, we can also compare the performance of the different compilers with the same script. Since Nvidia won't really be used for multicore compilation in the near future, the main two compilers of focus would be the intel and gcc compilers. Using the same thread counts, but only on one of the largest problem size yields





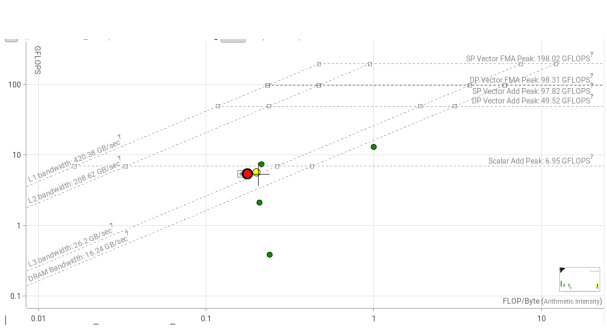
for the runtime and speedup graphs, respectively. Notice, however, that the Intel compiler is almost an order of magnitude worse than the gnu compiler. As the Intel compiler is optimized for the architecture that we are running these problems on, this is certainly an issue and not supposed to be happening. Nonetheless, speedup and runtime trends still look to be somewhat in order. To further identify the issues slowing down the intel compiler, we can use more specialized performance profiling softwares, advisor and vtune.

3.4 Intel Advisor and Vtune

The purpose of Intel Advisor and Vtune are both to collect performance data on scripts and determining where better threading and vectorization can be achieved to yield better performance. To use advisor, its recommended that code is compiled at a minimum of an `-O2` optimization level and are targetting the appropriate processor types. To collect data on the script, use

```
mbind -t 40 advixe-cl -collect survey -project-dir result_dir -- ./executable_file
mbind -t 40 advixe-cl -collect tripcounts -flop -project-dir result_dir -- ./executable_file
```

where the first command collects a survey of the code and timings of the overall and self time of each loop, and the second command collects specific flop counts and makes more granular analysis possible. So starting from the preliminary small case run on one thread, we have pretty well vectorized loops with the loops hovering around the L3 bandwidth on the roofline diagram since all the needed arrays can fit within the 25MB L3 cache at that stage.



Program metrics

Elapsed Time	69.40s		
Vector Instruction Set	AVX512, AVX2, AVX	Number of CPU Threads	1
Total GFLOP Count	368.02	Total GFLOPS	5.30
Total Arithmetic Intensity	0.19746		

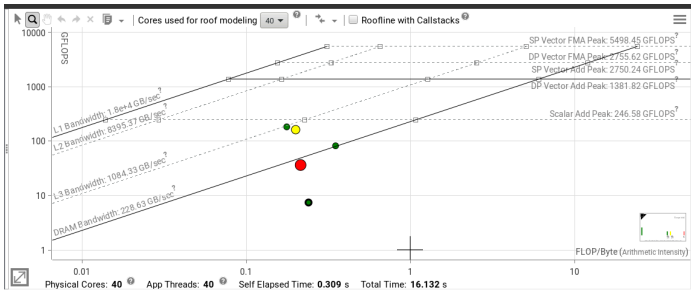
Loop metrics

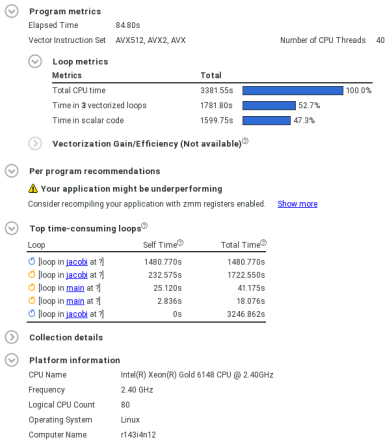
Metrics	Total	
Total CPU time	69.39s	100.0%
Time in **4** vectorized loops	68.78s	99.1%
Time in scalar code	0.61s	
Total GFLOP Count	368.02	100.0%
Total GFLOPS	5.30	

Vectorization Gain/Efficiency (Not available)

Now, looking into the parallel larger test case where we are seeing the aforementioned unexpected performance difference between the Intel and gnu compilers we can see that the roofline looks roughly as expected; the two system processes are slightly above the three main loops, which

are bounded by the DRAM bandwidth as the problem size is larger than can fit in the L3 cache. However, notice that only 3 of the 4 intended parallelized loops are shown on this roofline diagram. Looking at the survey confirms this, with the most costly loop in jacobi being run as scalar code and significantly dampening performance. Evidently, this is not intended and we can use the opt reports to look into it.





3.5 Forcing Vectorization

So the loop in question is the driving loop of the iteration, calculating the residual and updating in accordance to the tolerances. Looking at the opt report generated by the compiler, we can see that the remarks detail the potential speedup was below 1, in other words the compiler thought it would be faster in serial and adapted it as such.

```
LOOP BEGIN at jacobi.c(48,13)
   remark #15335: loop was not vectorized: vectorization possible but seems inefficient. Use vector
↪   always directive or -vec-threshold0 to override
   remark #15462: unmasked indexed (or gather) loads: 1
   remark #15463: unmasked indexed (or scatter) stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 54
   remark #15477: vector cost: 88.500
   remark #15478: estimated potential speedup: 0.600
   remark #15482: vectorized math library calls: 2
   remark #15487: type converts: 2
   remark #15488: --- end vector cost summary ---
LOOP END
```

A couple reasons this might be the case were fixed. Firstly, the initial macros defined to access the arrays were accessing vastly different sections of memory in quick succession, opting to directly access the term using $i * n + j$ instead. Next, using the `#pragma omp simd` construct, OpenMP is able to force vectorization directly on the inner loop, ignoring the remarks in the opt report. Lastly, the data structures were amended with a restrict keyword which restricts the pointer aliasing, helping with memory access.

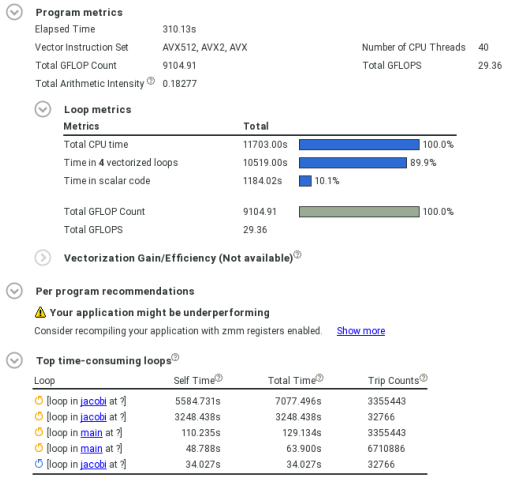
```
#pragma omp parallel for default(none) reduction(+:error) private(i, j, resid) shared(n, m, Uprev, u, f,
↪   ax, ay, b)
for (j=1; j<m-1; j++){
    #pragma omp simd
    for (i=1; i<n-1; i++){
```

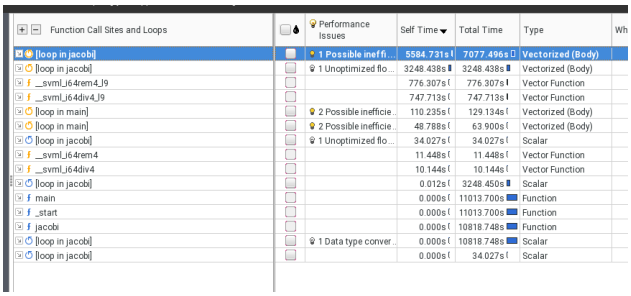
```
        resid =(ax * (Uprev[(i-1) * n + j] + Uprev[(i+1) * n + j])
              + ay * (Uprev[i * n + (j-1)] + Uprev[i * n + (j+1)])
              + b * Uprev[i * n + j] - f[i * n + j]) / b;
        u[i * n + j] = Uprev[i * n + j] - resid;
        error = error + resid * resid;
    }
}
```

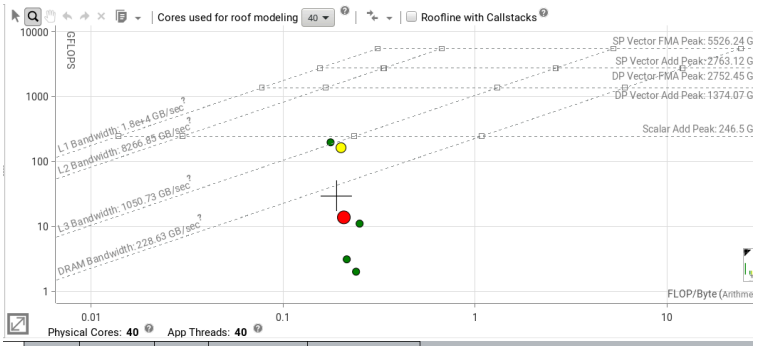
All in all, revisiting the opt report after these changes yielded hopeful results, having had the loop in question parallelized with an estimated speedup > 1.

```
LOOP BEGIN at jacobi.c(70,9)
   remark #15301: SIMD LOOP WAS VECTORIZED
   remark #26013: Compiler has chosen to target XMM/YMM vector. Try using -qopt-zmm-usage=high to override
   remark #15450: unmasked unaligned unit stride loads: 7
   remark #15451: unmasked unaligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 62
   remark #15477: vector cost: 18.250
   remark #15478: estimated potential speedup: 3.270
   remark #15486: divides: 1
   remark #15488: --- end vector cost summary ---

   LOOP BEGIN at jacobi.c(72,13)
      remark #25460: No loop optimizations reported
   LOOP END
LOOP END
```

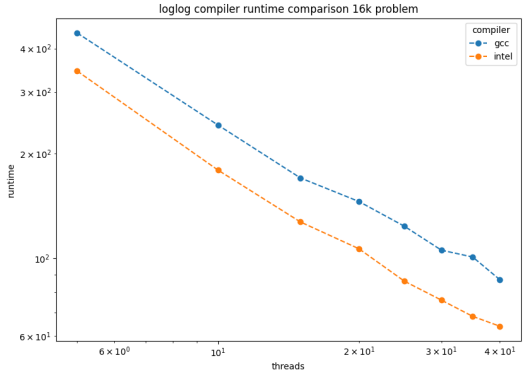
Likewise, running the revised code through Intel advisor gave similar results, with the vectorized code making up 89% of runtime. The roofline diagram also looked more natural, with the two system processes slightly above the other loops, all four of which were bounded by the DRAM bandwidth as the problem size exceeded the L3 cache. Furthermore, checking the trip counts with the expected values also works out.

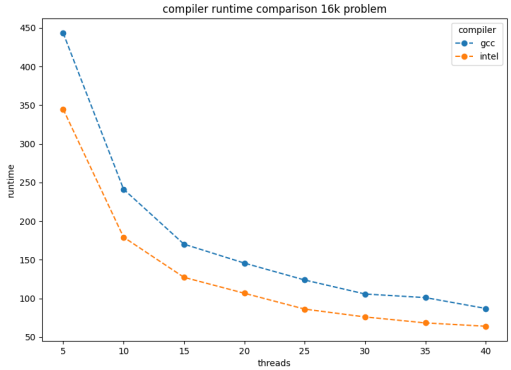






However, note that using the forced SIMD pragma seems to have cut the performance of the main four loops by a decent margin. Revisiting the original runtime and speedup graphs between the Intel and gnu compilers, we get a more expected trendline. Nonetheless, still note that a more stark runtime improvement for the Intel compiler should have been expected.





4 GPU Offloading

The Nvidia compiler, starting with version 21.5 has supported a new flag where the OpenMP implementation can be toggled between multicore and gpu, and applicable loops can be offloaded to execute on Nvidia GPUs with fallback on multicore. This feature is only supported on Nvidia GPUs V100 or newer. The GPU referenced here is the V100.

4.1 Mapping Process

Analagous Portion from OpenMP Parallelism Notes

In order to use OpenMP constructs on Nvidia GPUs the following mapping is used. So in a GPU parallelization structure, we need target to start offload, teams to take care of first-level parallelism, and parallel to take advantage of second-level parallelism. In the CPU side, we use parallel and simd because we can only use the different threads as parallel. The question becomes how we maintain portability between these two systems.

`#pragma omp target`	Starts Offload
`#pragma omp teams`	[GPU] CUDA Thread Blocks in grid [CPU] num_teams(1)
`#pragma omp parallel`	[GPU] CUDA Threads within thread block [CPU] CPU threads
`#pragma omp simd`	[GPU] simdlen(1) [CPU] Provides hints for vectorizations

Using the construct `#pragma omp target teams num_teams(X) thread_limit(Y)` we can then initiate offload and enable parallelism. The `target` clause starts the offload while the `teams` clause creates the teams X is the number of CUDA thread blocks, which does not affect the CPU. Y is the number of CUDA threads per thread block, corresponding to the number of CPU threads on that end. Using this we can see the Nvidia compiler generate both Tesla (GPU) and Multicore (CPU) code. In the GPU implementation each team has a master thread, and here the compute region is run by these master threads *sequentially*; in other words, we have offloaded our code but we have not enabled parallelism yet. There are two main ways of enabling parallelism in target offloading, following the prescriptive and descriptive models. They are described in more detail in *OpenMP Parallelism Notes* or the AMS Seminar *https://www.nas.nasa.gov/pubs/ams/2021/05-04-21.html*.

4.2 Compiling and Results

Accordingly, we use the descriptive model to allow the Nvidia compiler to decide on areas where best to offload to GPUs, and whether it should be done across both or just the innermost loops. In specific, the following pragmas are replaced so that they follow the syntax dictated by their hpc_sdk. All in all, with 1 GPU it yields roughly the same performance as 30 CPU threads on the 16k problem size, 28s.

```
#ifdef offload
    #pragma omp target teams loop collapse(2)
#else
    #pragma omp parallel for collapse(2)
#endif

#ifdef offload
    #pragma omp target teams loop reduction(+:error) private(i, resid) collapse(2)
#else
    #pragma omp parallel for default(none) reduction(+:error) private(i, j, resid) shared(n, m, Uprev, u,
    ↪  f, ax, ay, b)
#endif

#ifdef offload
    #pragma omp target teams loop reduction(+:error) reduction(max:max_val) reduction(min:min_val)
    ↪  collapse(2)
#else
    #pragma omp parallel for private(i, temp) reduction(+:error) reduction(max:max_val)
    ↪  reduction(min:min_val) collapse(2)
#endif
```

Poking through the compiler output, we can see the intended offloading sections, casting the OpenMP pragmas to parallelize across teams of GPU threads. Note this sometimes messes up reductions and may need to be fine tuned.

```
jacobi:
     33, #omp parallel
         34, Begin master region
         35, End master region
     52, #omp target teams loop
         52, Generating "nvkernel_jacobi_F1L52_2" GPU kernel
             Generating Tesla code
           52, Loop parallelized across teams, threads(128) collapse(2) /* blockIdx.x threadIdx.x */
           53,   /* blockIdx.x threadIdx.x collapsed */
         52, Generating Multicore code
           52, Loop parallelized across threads
     52, Generating implicit copyout(Uprev[:]) [if not already present]
         Generating implicit copyin(u[:]) [if not already present]
     71, #omp target teams loop
         71, Generating "nvkernel_jacobi_F1L71_3" GPU kernel
             Generating Tesla code
           71, Loop parallelized across teams, threads(128) collapse(2) /* blockIdx.x threadIdx.x */
           73,   /* blockIdx.x threadIdx.x collapsed */
           71, Generating reduction(+:error)
         71, Generating Multicore code
           71, Loop parallelized across threads
     71, Generating implicit copyout(u[:]) [if not already present]
         Generating implicit copyin(f[:],Uprev[:]) [if not already present]
         Generating map(tofrom:error)
initialize:
    108, #omp target teams loop
        108, Generating "nvkernel_initialize_F1L108_4" GPU kernel
             Generating Tesla code
          108, Loop parallelized across teams, threads(128) /* blockIdx.x threadIdx.x */
          109, Loop run sequentially
        108, Generating Multicore code
          108, Loop parallelized across threads
    108, Generating implicit copyin(dx) [if not already present]
         Generating implicit copyout(f[:],u[:]) [if not already present]
         Generating implicit copyin(dy) [if not already present]
    109, Loop carried dependence of f-> prevents parallelization
         Loop carried backward dependence of f-> prevents vectorization
error_check:
    135, #omp target teams loop
        135, Generating "nvkernel_error_check_F1L135_5" GPU kernel
             Generating Tesla code
          135, Loop parallelized across teams, threads(128) collapse(2) /* blockIdx.x threadIdx.x */
          136,   /* blockIdx.x threadIdx.x collapsed */
          135, Generating reduction(+:error)
               Generating reduction(min:min_val)
               Generating reduction(max:max_val)
               Generating reduction(min:min_val)
               Generating reduction(+:error)
        135, Generating Multicore code
          135, Loop parallelized across threads
    135, Generating implicit copyin(u[:]) [if not already present]
```

5 Code Reference

After compilation with the aforementioned commands, use `mbind -t 40 ./a.out` to run the whole batch from 5 to 40 threads across 2^{10} to 2^{15} problem sizes. Alternatively, use `mbind -t 40 ./a.out A B` where command line input A is 1 or 0 to display progress text, and input B is 1 or 0 for single

(40 threads, 16k problem size) or batch respectively.

To modify the thread counts or problem sizes check lines 186-187, 207-208, and 221 accordingly. To compile for GPU offloading uncomment line 9.