CS 5220 Project 3
All-Pairs Shortest Paths

Bob(Kunhe) Chen [kc853]
Cornell University

Abstract

In this project, our goal is to improve the performance of a naive implementation of the Floyd-Warshall algorithm that finds the shortest path in a graph. We used the Intel VTune Amplifier to profile the reference code and identify its performance bottlenecks. The bottlenecks largely come from frequent memory access due to inefficient utilization of cache memory. We have encountered simliar problem in project 1 where we optimized matrix multiplication. Therefore, we employed similar strategies to optimize this code using vectorization, blocking and compiler flags. By doing this, we achieved over 20 times better performance in the final code. We report here our final results and possible future improvement to the code.

1 Previous Report and Work Summary

We refer our readers to the previous report by Xiangyu Zhang [xz388], in which he reported our preliminary assessment of the code and bottlenecks. The report included profiling results and brief summary of some of our work. We hence assume readers' familiarity with the problem setup in this report and focus on technical details of our changes to the code.

We provide a list of changes that we have made to the code that turned out to be effective in speeding up the program:

- We used blocking strategy to take advantage of cache memory.
- We optimized our kerne function for fast computation of a basic block.
- We used aligned memory blocks for data storage and access.
- We used compiler flags to tell the compiler how to optimize our code.

2 Blocking and Kernel function

The Floyd-Warshall algorithm has a similar computational pattern as matrix multiplication, thus permitting us to adapt the blocking code from matrix multiplication project. We studied all the reports from class project 1 and picked one implementation from Scott_wu that yielded good results. We modified Scott_wu's matrix multiplication code for this project.

The idea of Scott_wu's code remained the same but we have made various changes to make it working for the parallelized Floyd-Warshall algorithm. One major change is an additional omp critical section to update the `done` flag in the computation. Moreover, we made a transposed copy of original matrix for aligned memory access.

The kernel function to compute a basic square block is modified from original code to yield best performance. In the matrix multiplication project, we used AVX intrinsics to get best performance but it would be more complicated to use in this case. Since we are able to fit more `int` variables in one vector register and consequently we will have to deal with a larger matrix (8-by-8 instead of 4-by4). The cost to implement this does not justify the potential performance boost and we therefore haven't attempted it.

Nevertheless, we are able to improve the performance of kernel function by a factor of 2 at least by changing the looping order and inner loop operation. We changed the looping order to maximize the usage of loaded memory and avoid unnecessary memory reading and writing. In the inner loop, we replaced `if` statement with two ternary operations. This allowed compiler to interpret the operation without ambiguity and further improved the performance of kernel function. A hotspots report from Amplifier shows that the ternary operators yield better performance than an if statement.

```
int basic_square(const int* restrict l, const int* restrict l_transpose,        0.179s
                 int* restrict lnew, int done){
    // Kernel routine to compute small problem that fits into memory
    __assume_aligned(l, 64);
    __assume_aligned(l_transpose, 64);
    __assume_aligned(lnew, 64);
    int oi, oj, ok;
    int ta, tb, tc;
    int temp;
        for (int j = 0; j < BLOCK_SIZE; ++j) {
        oj = j * BLOCK_SIZE;
        #pragma vector aligned
        for (int k = 0; k < BLOCK_SIZE; ++k) {                                   0.130s
            ok = k * BLOCK_SIZE;
            tb = l_transpose[oj+k];
            #pragma vector aligned
            for (int i = 0; i < BLOCK_SIZE; ++i) {
                temp = l[ok+i] + tb;                                             0.190s
                lnew[oj+i] = (lnew[oj+i] < temp)?temp:lnew[oj+i];                0.140s
                done = (lnew[oj+i] < temp)?0:done;                               0.200s
            }
        }
    }
    return done;
}
```

3 Memory Storage and Access

The original code used `malloc` to allocate memory blocks for variable storage. `malloc` becomes inadequate when compiler tries to vectorize loops that uses these variables, so we changed it to `_mm_malloc` to allocate aligned memory blocks for better access. The `free()` functions are also replaced by `_mm_free()`.

In `shortest_paths()` function, the two pointers `l` and `lnew` are passed to `square()` function to compute the shortest path. Then a memcpy is used to update the value in the memory block pointed by `l`. Notice that if we switch the role of these two variables, we can save the work to call the extra `memcpy` function.

With better memory access, we spend most of the time in kernel function:

Function	Module	CPU Time	CPU Time:Idle	CPU Time:Poor
`basic_square`	path.x	0.040s	0s	0.040s
`gen_graph`	path.x	0.021s	0.011s	0.010s
`fletcher16`	path.x	0.020s	0s	0.020s
`__intel_memset`	path.x	0.020s	0s	0.020s
`infinitize`	path.x	0.010s	0s	0.010s

This is a hotspots report generated by Amplifier using $N = 2000,\ p = 0.05$ and using 1 thread.

4 Compiler Flags

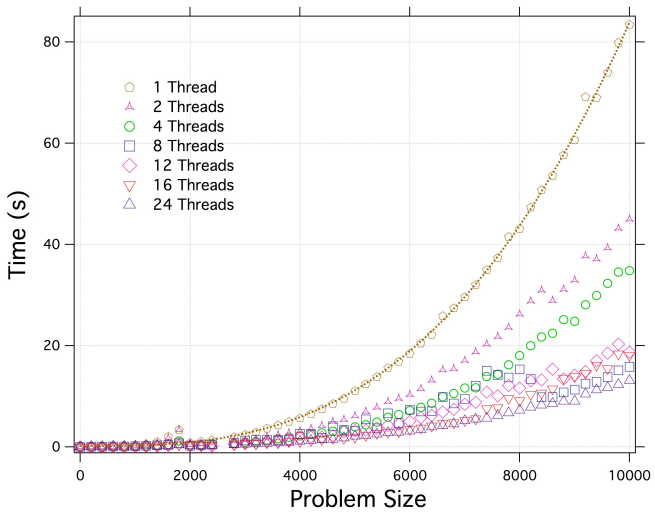
On the totient node, we tried some compiler flags and these ones improved the performance of our code:

```
-O3 -no-prec-div -xcore-avx2 -ipo -restrict
-unroll-aggresive -ftree-vectorize -opt-prefetch
-ansi-alias -ip
```

Also, we use `BLOCK_SIZE=64` as it gives the best performance on large problems.

5 Scaling Study

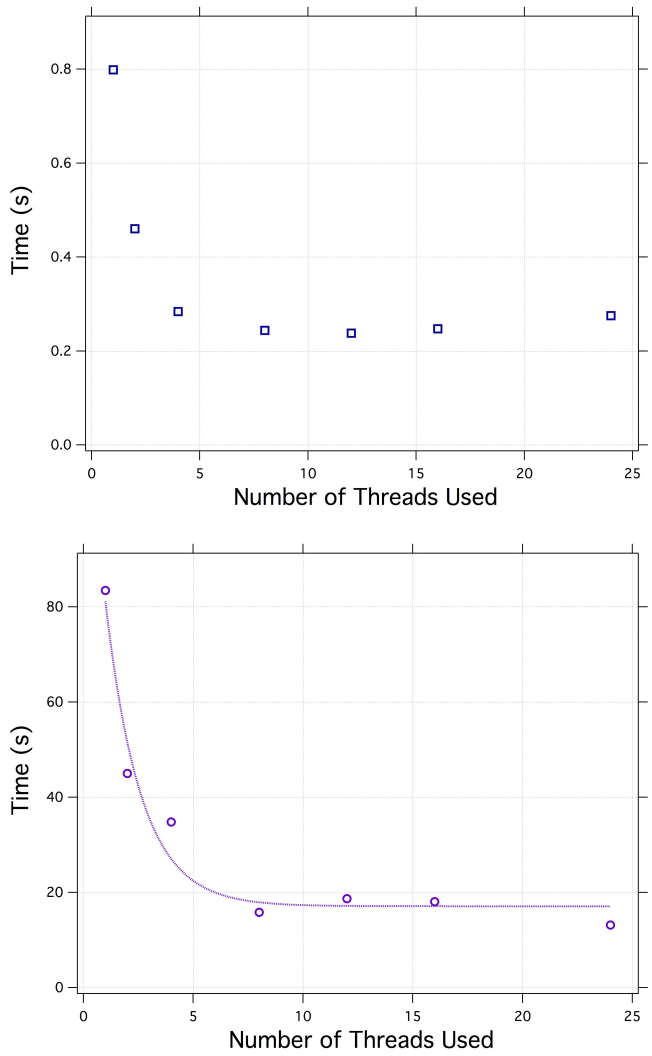
We measure the performance of our algorithm by averaging each run four times to reduce the effect of random initial condition. For a small case, $N = 2000$, we look at the time vs. number of threads in



use plot: Notice there is a slight increase in the time at the end of plot. As we increase the number of threads, the overhead work for OpenMP outweighs the benefit of using more threads. However, for a larger case, $N = 10000$, The speed keeps going up as we increase the number of threads in use. The final code scales well with large problems.

6 Conclusion and Future Work

For $N = 2000,\ p = 0.05$ test case, the original code takes about $3.3s$ with 24 threads. Our code can do it under $0.27s$ using the same number of threads. As explained in the last section, $N = 2000$ is too



small a problem size to fully take advantage of 24 threads. In addition, our code scales well with large problems, especially when the problem size gets above $N = 8000$.

The results come from using blocks to maximize the arithmetic intensity and using vectorized loops for fast computation. Future work can include more in-depth study in using a different computation pattern to speed up the code in an algorithmetic level.

7 Acknowledgement

We referred to other people's work in our attempt to speed up the code. In particular, we used Scott_wu's blocking strategy and kenlimmj's idea of using `#pragma vector aligned`.

8 Attachment

A cleaned up version of the final code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <getopt.h>
#include <unistd.h>
#include <getopt.h>
#include <omp.h>
#include "mt19937p.h"

#ifndef BLOCK_SIZE
#define BLOCK_SIZE ((int) 64)
#endif

int basic_square(const int* restrict l, const int* restrict l_transpose,
                      int* restrict lnew, int done){
    // Kernel routine to compute small problem that fits into memory
    __assume_aligned(l, 64);
    __assume_aligned(l_transpose, 64);
    __assume_aligned(lnew, 64);

    int oi, oj, ok;
    int ta, tb, tc;
    int temp;
    for (int j = 0; j < BLOCK_SIZE; ++j) {
        oj = j * BLOCK_SIZE;
        #pragma vector aligned
        for (int k = 0; k < BLOCK_SIZE; ++k) {
            ok = k * BLOCK_SIZE;
            tb = l_transpose[oj+k];
            #pragma vector aligned
            for (int i = 0; i < BLOCK_SIZE; ++i) {
                temp = l[ok+i] + tb;
                lnew[oj+i] = (lnew[oj+i] < temp)?temp:lnew[oj+i];
                done = (lnew[oj+i] < temp)?0:done;
            }
        }
    }
   return done;
}

int square_block(int n,               // Number of nodes
           int* restrict l,     // Partial distance at step s
```

```
            int* restrict lnew)  // Partial distance at step s+1
{
    int* restrict l_transpose __attribute__((aligned(64)))= (int*) _mm_malloc(n*n*sizeof(int),64

    for( int i = 0; i < n; i++)
      for( int j = 0; j < n; j++){
        l_transpose[j*n+i] = l[i*n+j];
      }

    int done = 1;
    int M    = n; // Change the variable to M so it is consistent
    // Number of blocks total
    const int n_blocks = n / BLOCK_SIZE + (M % BLOCK_SIZE? 1 : 0);
    const int n_size = n_blocks * BLOCK_SIZE;
    const int n_area = BLOCK_SIZE * BLOCK_SIZE;
    const int n_mem = n_size * n_size * sizeof(int);
    // Copied A matrix
    int * restrict L __attribute__((aligned(64)))= (int *) _mm_malloc(n_mem*sizeof(int),64);
    memset(L, 0, n_mem);
    // Copied B matrix
    int * restrict Lt __attribute__((aligned(64)))= (int *) _mm_malloc(n_mem*sizeof(int),64);
    memset(Lt, 0, n_mem);
    // Copied C matrix
    int * restrict Ln __attribute__((aligned(64)))= (int *) _mm_malloc(n_mem*sizeof(int),64);
    memset(Ln, 0, n_mem);

    // Initialize matrices
    int bi, bj, bk, i, j, k;
    int copyoffset;
    int offset, offsetL, offsetLt;

    for (bi = 0; bi < n_blocks; ++bi) {
        for (bj = 0; bj < n_blocks; ++bj) {
            int oi = bi * BLOCK_SIZE;
            int oj = bj * BLOCK_SIZE;
            copyoffset = (bi + bj * n_blocks) * BLOCK_SIZE * BLOCK_SIZE;
            for (j = 0; j < BLOCK_SIZE; ++j) {
                for (i = 0; i < BLOCK_SIZE; ++i) {
                    offsetL  = (oi + j) + (oj + i) * M;
                    offsetLt = (oi + i) + (oj + j) * M;
                    // Check bounds
                    if (oi + j < M && oj + i < M) {
                         L[copyoffset] = l[offsetL];
                        Lt[copyoffset] = l_transpose[offsetLt];
                    }
                    if (oi + i < M && oj + j < M) {
                         L[copyoffset] = l[offsetLt];
```

```
                        Lt[copyoffset] = l_transpose[offsetLt];
                    }
                    Ln[copyoffset] = 0;
                    copyoffset++;
                }
            }
        }
    }

#pragma omp parallel shared(L, Lt, Ln, done)
{
    #pragma omp for
    for (bi = 0; bi < n_blocks; ++bi) {
        for (bj = 0; bj < n_blocks; ++bj) {
            // #pragma vector aligned learned from kenlimmj's code
            #pragma vector aligned
            for (bk = 0; bk < n_blocks; ++bk) {
                int td = basic_square(
                     L + (bi + bk * n_blocks) * n_area,
                    Lt + (bk + bj * n_blocks) * n_area,
                    Ln + (bi + bj * n_blocks) * n_area,
                    done);
                if (done == 1 && td == 0){
                    #pragma omp critical
                    done = 0;
                }
            }
        }
    }
}

     // Copy results back
     for (bi = 0; bi < n_blocks; ++bi) {
         for (bj = 0; bj < n_blocks; ++bj) {
            int oi = bi * BLOCK_SIZE;
            int oj = bj * BLOCK_SIZE;
            copyoffset = (bi + bj * n_blocks) * n_area;
            for (j = 0; j < BLOCK_SIZE; ++j) {
                 for (i = 0; i < BLOCK_SIZE; ++i) {
                    offset = (oi + i) + (oj + j) * M;
                    // Check bounds
                    if (oi + i < M && oj + j < M) {
                          lnew[offset] = Ln[copyoffset];
                    }
                    copyoffset++;
                 }
            }
```

```
        }
     }
    _mm_free(l_transpose);
    return done;
}
int square(int n,               // Number of nodes
           int* restrict l,     // Partial distance at step s
           int* restrict lnew)  // Partial distance at step s+1
{
    // Make a copy of transposed matrix to speed up vectorization
    int* restrict l_transpose __attribute__((aligned(64)))= (int*) _mm_malloc(n*n*sizeof(int),64
    for( int i = 0; i < n; i++){
      for( int j = 0; j < n; j++){
        l_transpose[j*n+i] = l[i*n+j];
      }
    }

    __assume_aligned(l, 64);
    __assume_aligned(l_transpose, 64);
    __assume_aligned(lnew, 64);

    int done = 1;
    #pragma omp parallel for shared(l, lnew, l_transpose) reduction(&& : done)
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            int lij = lnew[j*n+i];
            for (int k = 0; k < n; ++k) {
                int lik = l_transpose[i*n+k];
                int lkj = l[j*n+k];
                if (lik + lkj < lij) {
                    lij = lik+lkj;
                    done = 0;
                }
            }
            lnew[j*n+i] = lij;
        }
    }
    _mm_free(l_transpose);
    return done;
}

static inline void infinitize(int n, int* restrict l)
{
    for (int i = 0; i < n*n; ++i)
        if (l[i] == 0)
            l[i] = n+1;
}
```

```
static inline void deinfinitize(int n, int* restrict l)
{
    for (int i = 0; i < n*n; ++i)
        if (l[i] == n+1)
            l[i] = 0;
}

void shortest_paths(int n, int* restrict l)
{
    // Generate l_{ij}^0 from adjacency matrix representation
    infinitize(n, l);
    for (int i = 0; i < n*n; i += n+1)
        l[i] = 0;
    // Repeated squaring until nothing changes
    int* restrict lnew __attribute__((aligned(64)))= (int*) _mm_malloc(n*n*sizeof(int),64);
    memcpy(lnew, l, n*n * sizeof(int));
    for (int done = 0; !done; ) {
        if ( n < 1801) done = square(n, lnew, l);
        else done = square_block(n, lnew, l);
    }
    _mm_free(lnew);
    deinfinitize(n, l);
}

int* restrict gen_graph(int n, double p)
{
    int* l __attribute__((aligned(64)))= (int*) _mm_malloc(n*n*sizeof(int),64);
    struct mt19937p state;
    sgenrand(10302011UL, &state);
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i)
            l[j*n+i] = (genrand(&state) < p);
        l[j*n+j] = 0;
    }
    return l;
}

int fletcher16(int* data, int count)
{
    int sum1 = 0;
    int sum2 = 0;
    for(int index = 0; index < count; ++index) {
          sum1 = (sum1 + data[index]) % 255;
          sum2 = (sum2 + sum1) % 255;
    }
    return (sum2 << 8) | sum1;
}
```

```
void write_matrix(const char* fname, int n, int* a)
{
    FILE* fp = fopen(fname, "w+");
    if (fp == NULL) {
        fprintf(stderr, "Could not open output file: %s\n", fname);
        exit(-1);
    }
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j)
            fprintf(fp, "%d ", a[j*n+i]);
        fprintf(fp, "\n");
    }
    fclose(fp);
}

/**
 * # The `main` event
 */

const char* usage =
    "path.x -- Parallel all-pairs shortest path on a random graph\n"
    "Flags:\n"
    "  - n -- number of nodes (200)\n"
    "  - p -- probability of including edges (0.05)\n"
    "  - i -- file name where adjacency matrix should be stored (none)\n"
    "  - o -- file name where output matrix should be stored (none)\n"
    "  - t -- set number of threads (max threads)\n";

int main(int argc, char** argv)
{
    int n    = 200;            // Number of nodes
    double p = 0.05;           // Edge probability
    const char* ifname = NULL; // Adjacency matrix file name
    const char* ofname = NULL; // Distance matrix file name
    int num_threads_used = omp_get_max_threads();

    // Option processing
    extern char* optarg;
    const char* optstring = "hn:d:p:o:i:t:";
    int c;
    while ((c = getopt(argc, argv, optstring)) != -1) {
        switch (c) {
        case 'h':
            fprintf(stderr, "%s", usage);
            return -1;
        case 'n': n = atoi(optarg); break;
```

```
        case 'p': p = atof(optarg); break;
        case 'o': ofname = optarg;  break;
        case 'i': ifname = optarg;  break;
        case 't': num_threads_used = atoi(optarg); break;
        }
    }

    // Graph generation + output
    int* restrict l = gen_graph(n, p);
    if (ifname)
        write_matrix(ifname,  n, l);

    if (num_threads_used <= omp_get_max_threads()){
      omp_set_num_threads(num_threads_used);
    }

    // Time the shortest paths code
    double t0 = omp_get_wtime();
    shortest_paths(n, l);
    double t1 = omp_get_wtime();

    printf("== OpenMP with %d threads\n", num_threads_used);
    printf("n:     %d\n", n);
    printf("p:     %g\n", p);
    printf("Time:  %g\n", t1-t0);
    printf("Check: %X\n", fletcher16(l, n*n));

    // Generate output file
    if (ofname)
        write_matrix(ofname, n, l);

    // Clean up
    _mm_free(l);
    return 0;
}
```