Optimising the LINPACK 1000 using Parallelization

Sam Serrels
40082367@napier.ac.uk
Edinburgh Napier University
Concurrent and Parallel Systems (SET10108)

1 Introduction

The LINPACK benchmark The Linpack algorithm is a popular benchmark in the high performance computing field as it is uses as a floating point performance measure for ranking supercomputers. The Benchmark solves a large system of linear equations using LU decomposition and is typical of many matrix-based scientific computations. Linpack started as a Fortran maths processing application, the code for solving linear equations was extracted from the original program and turned into a benchmark.
The Linpack1000 algorithm first generates a random 1000x1000 element matrix, A, and 1000 element vector, b. The elements in A and b are all double precision floating point numbers. Processing then takes place to find the solution, a 1000 element vector, x, such that Ax = b.
This report documents the process of analysing a specific sequential Linpack implementation and then converting it to a parallel task.

Related Work As Linpack is used to benchmark large scale multiprocessor supercomputers, there are many parallel versions available. The main difference between implementations is the Gaussian elimination stage, for which there are many algorithms available, some of which can split the task up into separate easily parallelizable chunks of logic. For the scope of this project, changes to the algorithm were avoided wherever possible to keep a fair comparison to the original sequential code.

High-Performance Linpack The most commonly used implantation of Linpack is the HPL implementation, written by the Innovative Computing Laboratory at the University of Tennessee. HPL is an open-source project that aims to provide a toolbox for configuring, optimising, and running the benchmark over a network. It contains many version of the algorithm with plenty of configurable options and for tuning performance to a specific system.
The HPL project was used as a rough guide to how the reference algorithm could be modified for this project, however most of the optimisation were beyond the scope of this project.

Project Scope A quick optimisation would be to drop the precision of the algorithm from double, to single precision floating point values. Another method would be to swap the original Gaussian elimination algorithm for a different mathematical approach that would lend itself to parallel processing better. This project aimed to see how much the original Linpack code can be optimised with parallelization, without changing the core logic of the algorithm or data output so these routes for optimisation were ruled out.

OpenMP The technology for processing the application in parallel was chosen to be OpenMp, an API that abstracts the creation of threads from the user and therefore allows for easier development and better cross platform portability, assuming that the chosen platform has a complier that supports OpenMP. This was chosen over creating threads manually, mainly for ease of development reasons, but also because even in a situation that OpenMp is slower than Manual threads, there should still be a noticeable performance increase over the baseline results.

SIMD For an extra level of performance, SIMD instructions were used to gain performance in the most frequently executed parts of the program.

2 Linpack Gaussian elimination

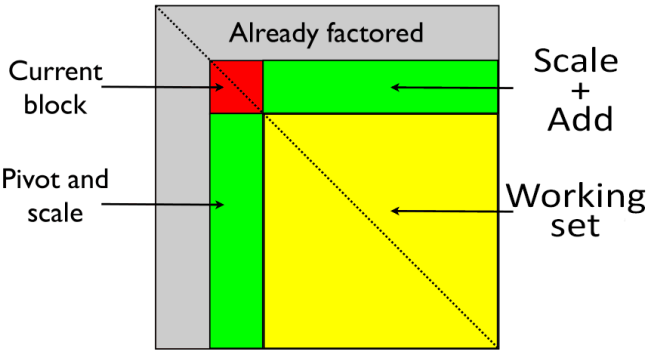

Figure 1: *Overview of GE progression* -

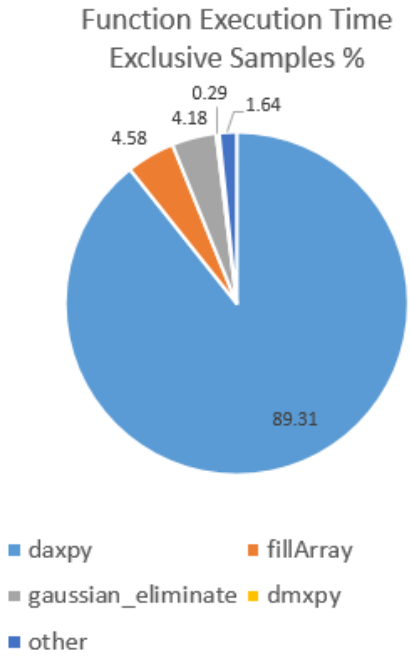

Figure 2: *Total program execution* -

Initial analysis The Gaussian elimination(GE) stage, work it's way through the array, starting in the "top left corner". It examines the entire first column(C) ("Pivot and scale", figure **??**) and finds the largest value (T). The row that contains the largest value(T) becomes the pivot, it is swapped with the topmost row. Then, each column is processed, by multiplying it by 1/T and then adding the value of the C column.

Once this is complete, the process restarts but in a subsection of the

Matrix A that is one row and column smaller than the previous iteration. This continues until the "Bottom right" corner is reached. This process transforms Matrix A into a upper triangular matrix that is in row echelon form.

On analysis of execution of the program , it was clear that the vast majority of the execution time was based in the Gaussian Elimination stage. On further examination, the "Daxpy" function, which his called many times during the GE stage takes up nearly 90% of the total execution time. Daxpy is the function that does the scaling and addition of each column, and is called 424166 times during the full execution of the program.

Daxpy While the Daxpy function is called at a high frequency it contains very few lines of code. It's function is to compute Y = S * X + Y, X and Y are elements of two arrays, and S is a scaler value. In the program, this is used in a loop to process each column in the A array.

This was the first part of the code to be examined for possible speed-up, as each iteration of the loop doesn't depend on any other iteration. Initially Paralleling the loop with OpenMp was attempted, but as the loop will only ever execute a maximum of 1000 times, the overhead time of creating and running threads was always greater than the time of running the loop without threads.

Listing 1: *daxpy Code*

```
1 void (int n,double scaler, double *dx, double *dy, int offset) {
2     double *const y = &dy[offset];
3     double *const x = &dx[offset];
4     for (int i = 0; i < n; ++i) {
5         y[i] += scaler * x[i];
6     }
7 }
8
```

Simd Daxpy As no increase of performance could be gained by using more threads, Simd instructions were investigated. Daxpy process N amount of numbers from the two input arrays, using a loop. The numbers to be processed are sequentially laid out in the input arrays, so the logic to convert the loop from processing one item at a time, to multiple, was a simple task. Additional lines of code were unavoidable, e.g if the size of N was not divisible by the amount of items that the loop processes in one pass, then the remainder would have to be processed at the end.
Simd instructions requires aligned memory, therefore the numbers form the input arrays had to be loaded into special aligned containers. This could be avoided if the whole program was converted to use aligned data arrays.

Listing 2: *Simd daxpy Code*

```
1 void (int n, double scaler, double *dx, double *dy,int offset) {
2     if ((n <= 0) || (scaler == 0)) {
3         return;
4     }
5     double *const y = &dy[offset];
6     double *const x = &dx[offset];
7
8     const __m256d scalers = _mm256_set1_pd(scaler);
9     const int remainder = n % 4;
10    const int nm1 = n - 3;
11
12    for (int i = 0; i < nm1; i += 4) {
13        // load X
14        const __m256d xs = _mm256_loadu_pd(&x[i]);
15        // load y
16        __m256d ys = _mm256_loadu_pd(&y[i]);
17        // mutliply X by scalers, add to Y
18        ys = _mm256_add_pd(ys, _mm256_mul_pd(xs, scalers));
19        // load back into y
20        _mm256_storeu_pd(&y[i], ys);
21    }
22
23    for (int i = n - remainder; i < n; ++i) {
24        y[i] += scaler * x[i];
25    }
26 }
```

Simd Daxpy Results Both 128bit and 256bit (Largest supported by available hardware) Simd versions of Daxpy were implemented and tested. The 128bit version provided no performance gains, due to the overhead of converting input data to aligned data. 256 bit instructions however provided a significant increase in performance, but only when used with an N parameter greater than around 100. As seen in Figure **??**.

The Pivot loop

The Collumn loop

Memory alignment

3 Optimisation Method

Code Simplification

Parallelised Daxpy no route was over the capacity of the truck.

SIMD Daxpy

MDaxpy

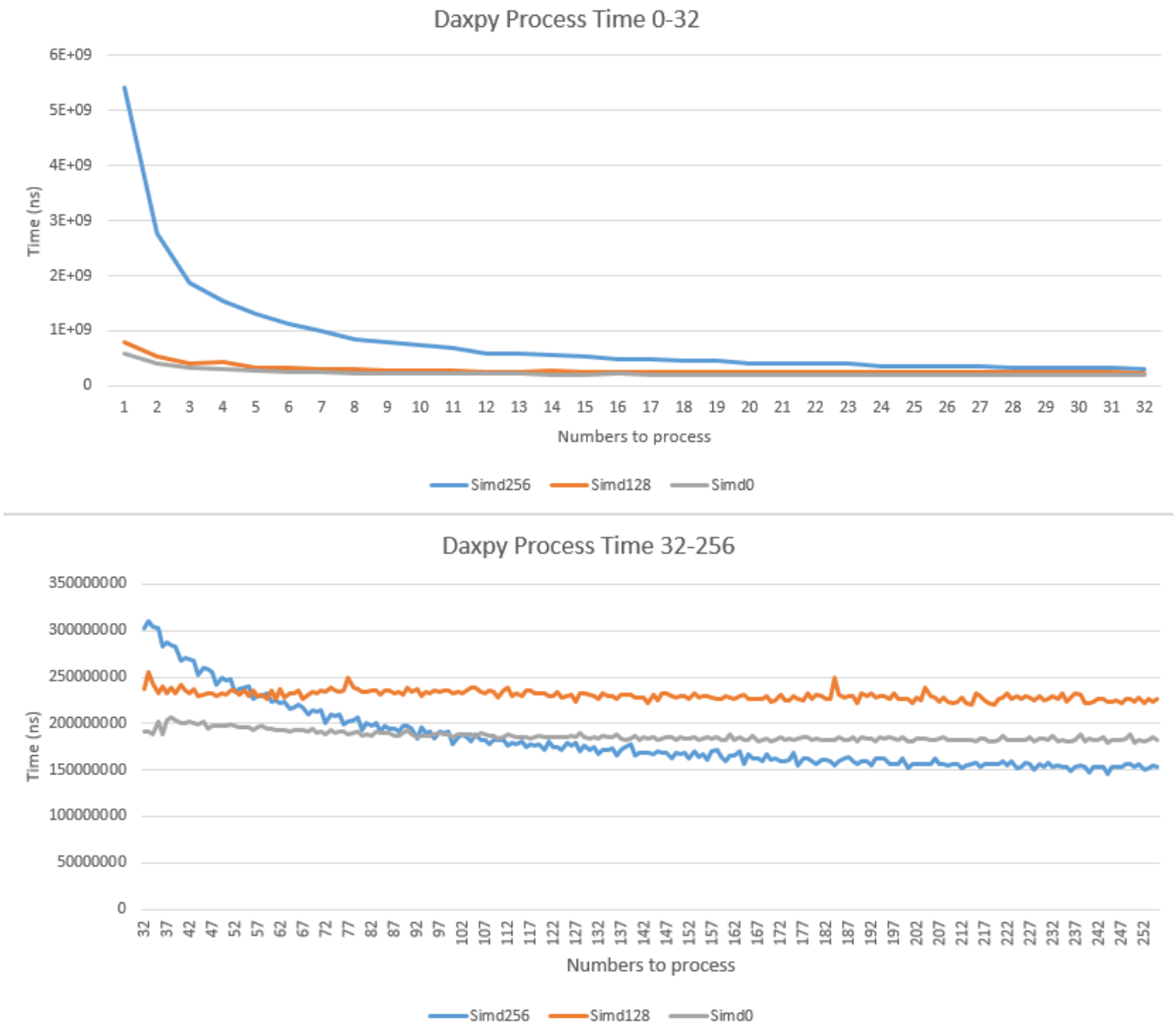

Figure 3: ***Daxpy Simd Comparisons*** *- Time taken to process 10'000 numbers, 10'000 times Each subsequent call to Daxpy increases the amount of number to calculate at once.*

Name	Allocate Memory (ms)	Create Input Numbers (ms)	gaussian eliminate (ms)	Solve (ms)	Validate (ms)	Total Time (ms)	Total Speedup	Speedup (With Simd)*
Threads: 1 No Simd	0.48	5.22	157.27	0.69	5.27	168.93	0%	0%
Threads: 1 Simd128	0.45	5.00	152.68	0.67	5.06	163.85	3%	0%
Threads: 1 Simd256	0.44	5.06	142.04	0.67	5.14	153.34	9%	0%
Threads: 2 No Simd	0.42	5.48	78.76	0.68	5.52	90.86	46%	46%
Threads: 2 Simd128	0.46	5.36	85.35	0.66	5.49	97.32	42%	41%
Threads: 2 Simd256	0.45	4.82	72.86	0.63	4.83	83.59	51%	45%
Threads: 3 No Simd	0.45	5.37	63.49	0.68	5.46	75.44	55%	55%
Threads: 3 Simd128	0.46	5.52	60.00	0.66	5.64	72.28	57%	56%
Threads: 3 Simd256	0.46	5.74	52.51	0.58	5.85	65.14	61%	58%
Threads: 4 No Simd	0.44	5.28	42.67	0.62	5.33	54.34	68%	68%
Threads: 4 Simd128	0.43	5.72	41.51	0.54	5.78	53.97	68%	67%
Threads: 4 Simd256	0.48	5.80	39.61	0.55	5.91	52.36	69%	66%
Threads: 5 No Simd	0.46	5.41	50.89	0.67	5.50	62.93	63%	63%
Threads: 5 Simd128	0.45	5.20	49.49	0.67	5.24	61.05	64%	63%
Threads: 5 Simd256	0.46	5.11	46.46	0.60	5.24	57.86	66%	62%
Threads: 6 No Simd	0.50	5.80	44.44	0.75	5.93	57.41	66%	66%
Threads: 6 Simd128	0.48	5.53	43.04	0.69	5.66	55.41	67%	66%
Threads: 6 Simd256	0.49	5.55	41.63	0.65	5.72	54.04	68%	65%
Threads: 7 No Simd	0.53	6.00	40.21	0.75	6.18	53.66	68%	68%
Threads: 7 Simd128	0.52	5.40	40.02	0.73	5.55	52.22	69%	68%
Threads: 7 Simd256	0.55	5.58	35.69	0.68	5.77	48.28	71%	69%
Threads: 8 No Simd	0.51	5.01	43.25	0.80	5.17	54.75	68%	68%
Threads: 8 Simd128	0.53	5.48	43.44	0.78	5.73	55.96	67%	66%
Threads: 8 Simd256	0.51	5.47	38.64	0.71	5.72	51.05	70%	67%

Table 1: *Results of all tests*
**Speed-up with Simd, compares times against the Simd equivalent 1 Thread run*

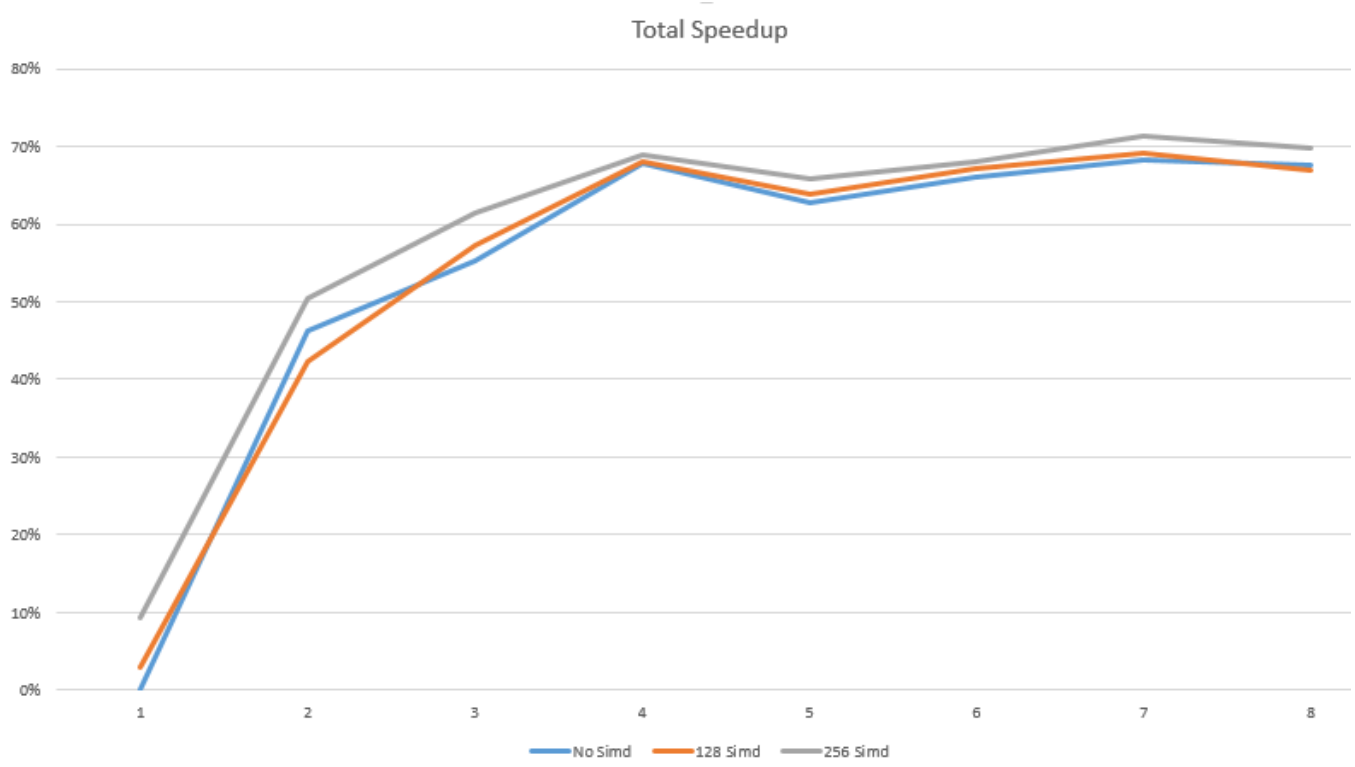

Figure 4: ***Total Speed-up percentage, for each number of threads*** -

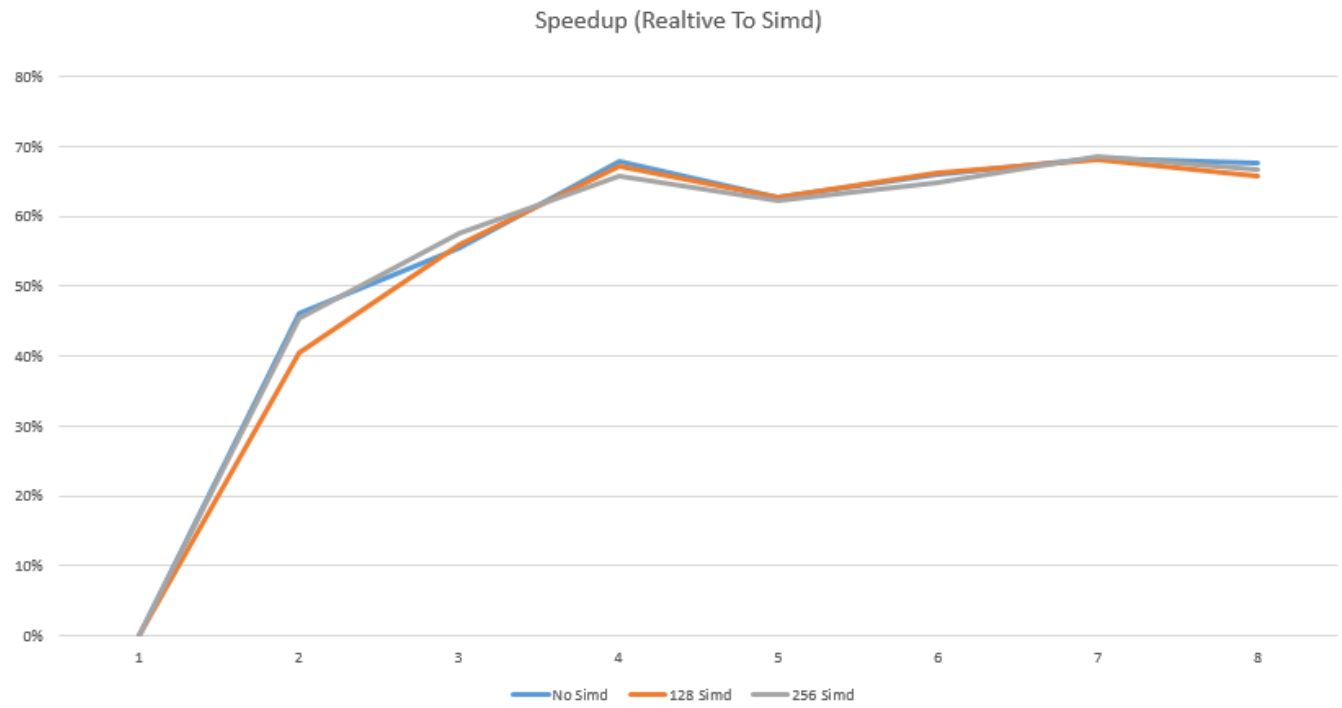

Figure 5: ***Total Speed-up (Relative to baseline Simd) percentage, for each number of threads*** -

4 Conclusions

Computation time Both Implementations of the Clarke-Wright algorithms produced expected results, with the parallel version producing larger and fewer routes. As for the time taken to calculate, the performance is roughly equal. The discrepancies shown in Figure **??** when the amount of customers increases beyond 800 is possibly due to optimisations carried out by the Java virtual machine. The total operations carried out is roughly the same for each algorithm, however the arrays are accessed and modified at different times, this is a possible cause for the difference in processing time.

Solution Quality The Parallel solution produced a large saving of up to a 600% increase against the baseline cost, shown in Figure **??**. The Sequential solution produced a constant saving of around 200%. These results are also shown in Figure **??**, showing the number of routes.

Edge Cases It is possible that a customer can be left out of all routes due to capacity constraints; this is checked for at the end of the calculation. If a customer is left over, it is seen if it would be possible to add it to any existing route and then if it would be more efficient than sending out a new truck. This can be seen in Figure **??**, the customer in the top left falls in to this edge case category.

Conclusion The implementation written for this report successfully computes optimised and usable data, the processing cost increases in a quadratic relation to the size of data. The specific implementation could be optimised to produce quicker results. One possible optimisation route could be a custom sort method, as profiling reported that 40% of the processing time is taken by the initial sort of the customer pairs.
Overall this report produced repeatable and meaningful data, and can be seen as a successful investigation into the Clark-Wright Algorithm.

References

CLARKE, G., AND WRIGHT, J. 1962. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research 12*, 4, 568–581.

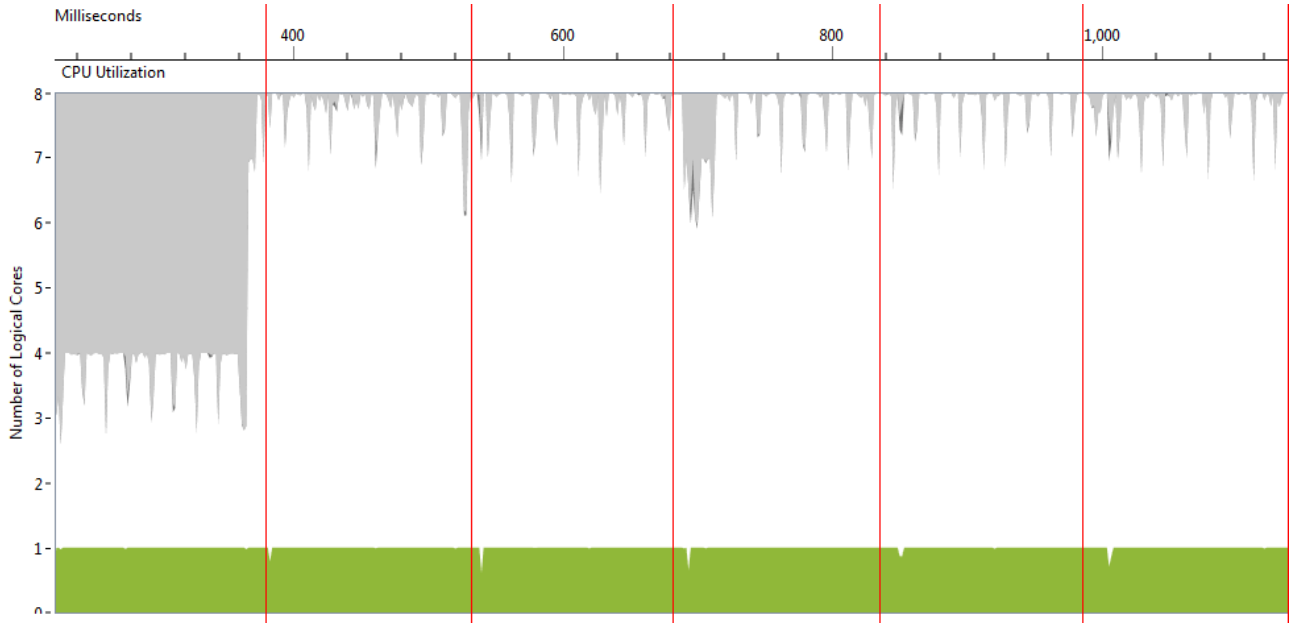

Figure 6: ***Single Threaded, 6 runs, simd256 Daxpy*** *- Overall system CPU utilisation*

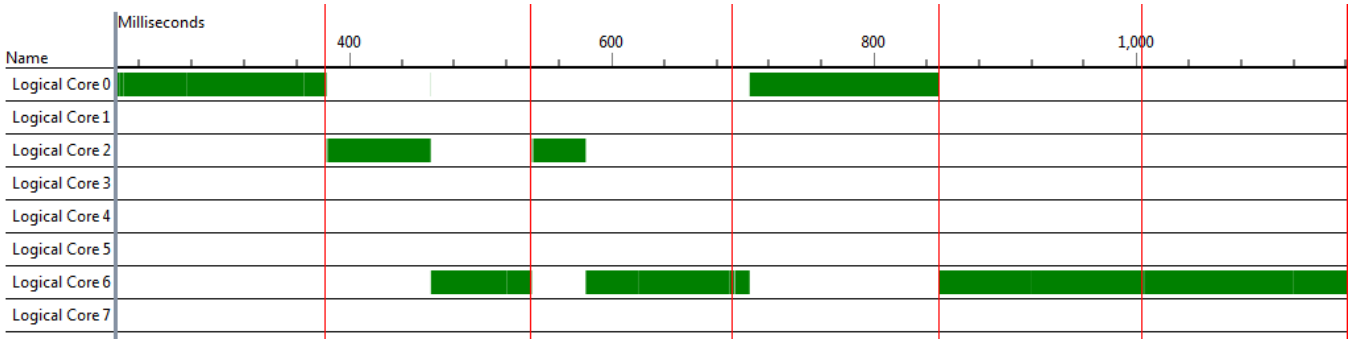

Figure 7: ***Single Threaded, 6 runs, simd256 Daxpy*** *- Thread to CPU Core allocation*

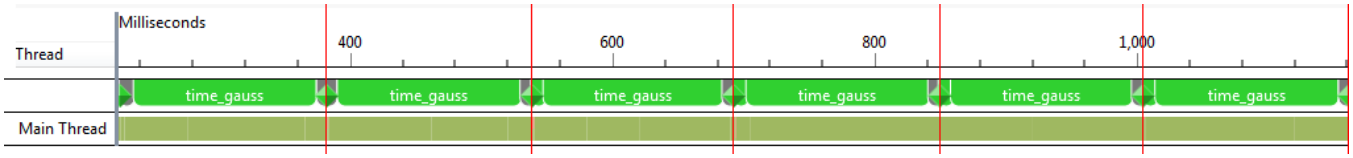

Figure 8: ***Single Threaded, 6 runs, simd256 Daxpy*** *- Thread Status*

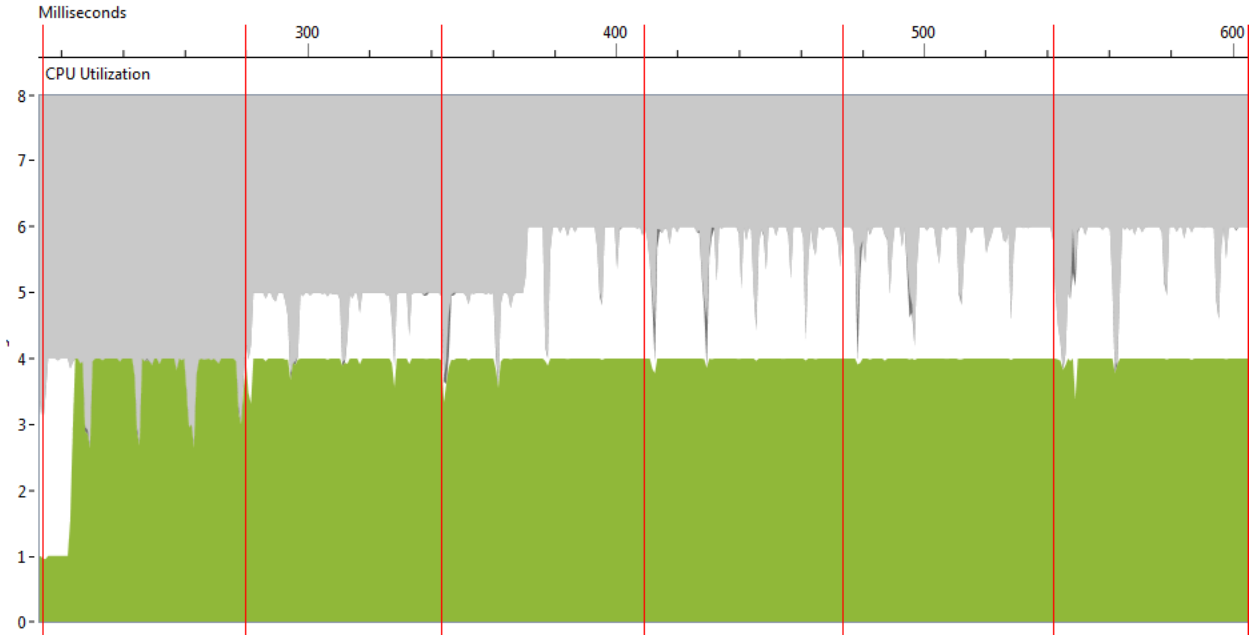

Figure 9: ***4 Threads, 6 runs, simd256 Daxpy*** *- Overall system CPU utilisation*

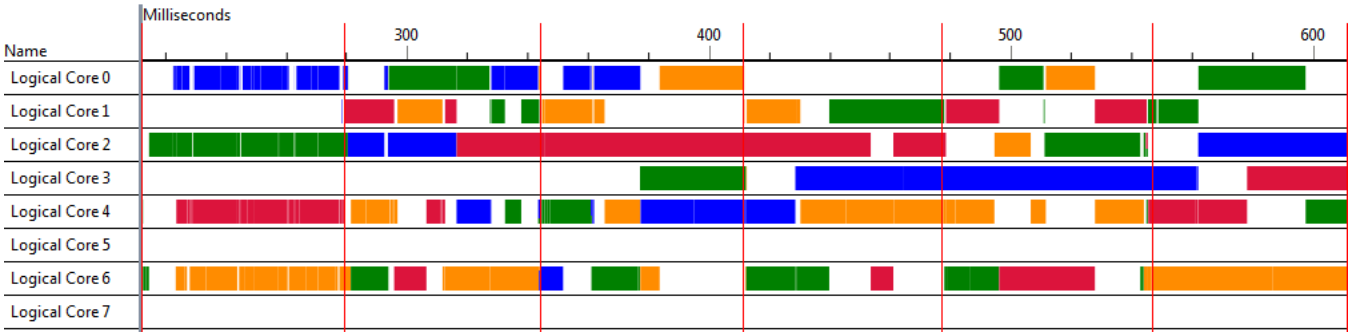

Figure 10: ***4 Threads, 6 runs, simd256 Daxpy*** *- Thread to CPU Core allocation*

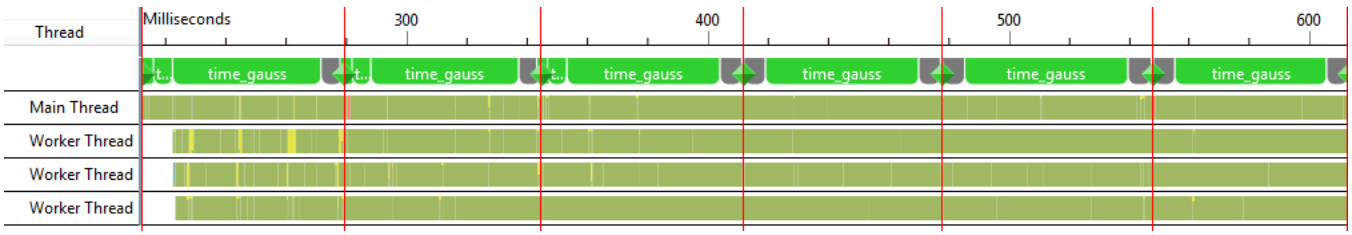

Figure 11: ***4 Threads, 6 runs, simd256 Daxpy*** *- Thread Status*

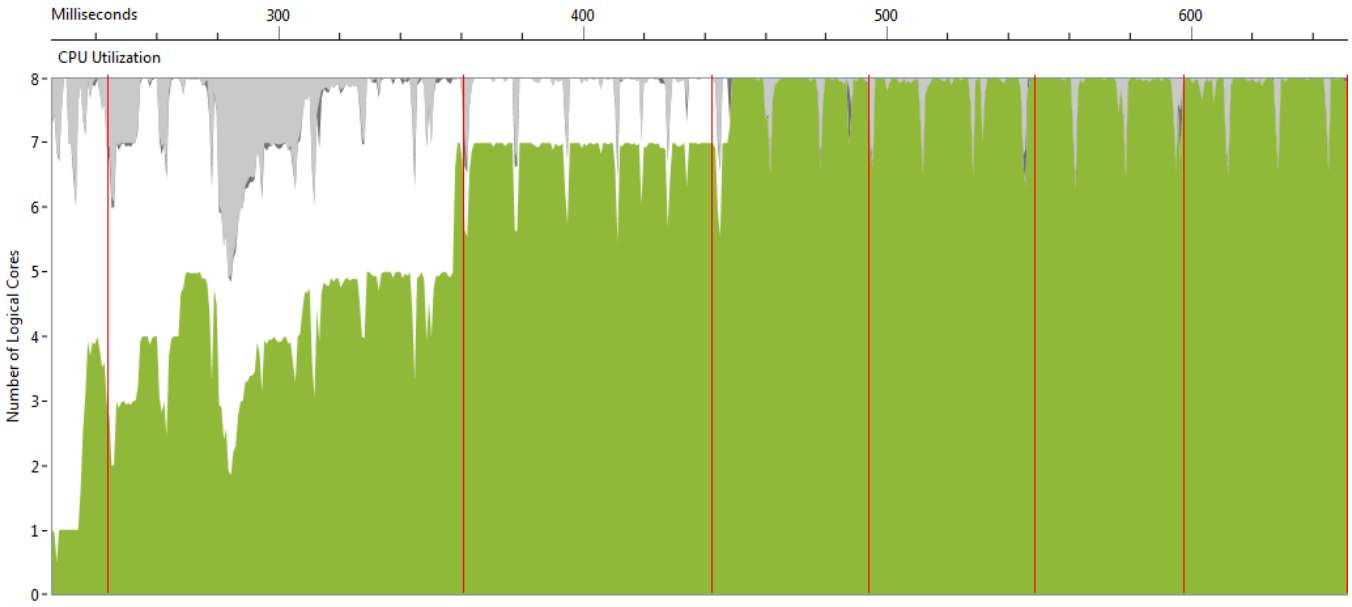

Figure 12: ***8 Threads, 6 runs, simd256 Daxpy*** *- Overall system CPU utilisation*

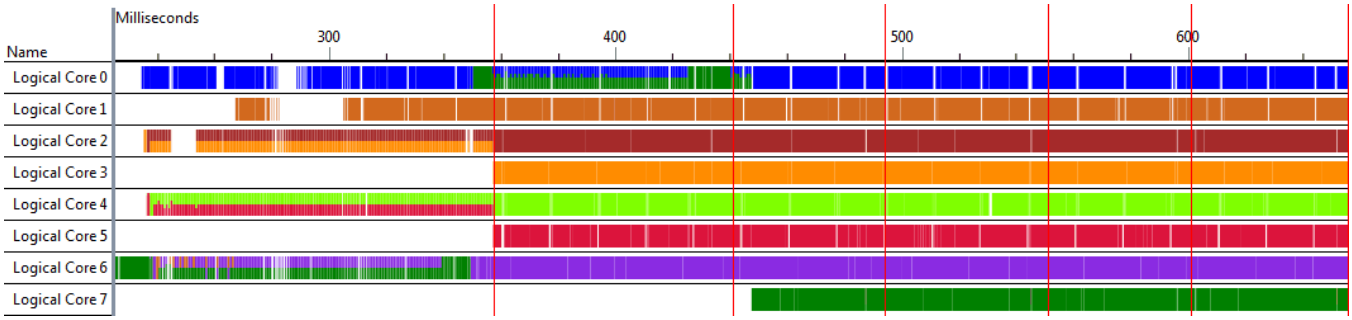

Figure 13: ***8 Threads, 6 runs, simd256 Daxpy*** *- Thread to CPU Core allocation*

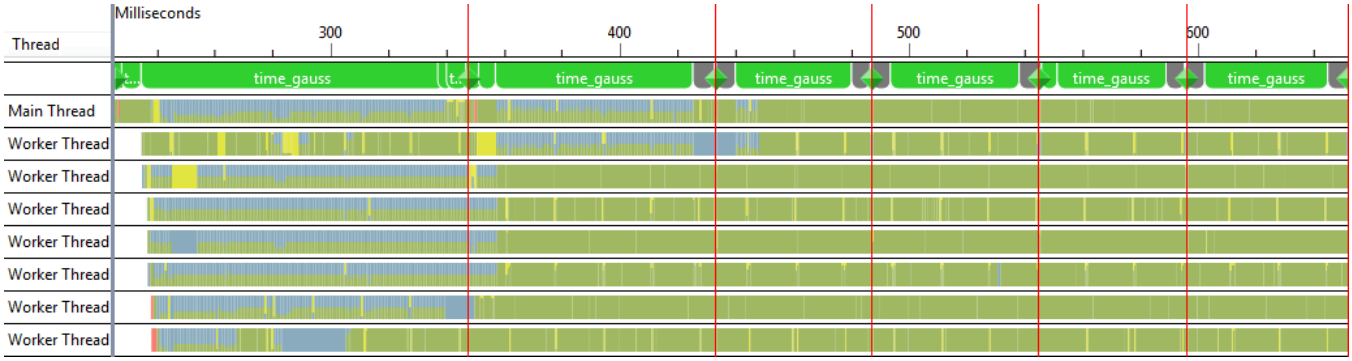

Figure 14: ***8 Threads, 6 runs, simd256 Daxpy*** *- Thread Status*

5 Appendix: Code

```
#include <iostream>
#include <string>
#include <assert.h>
#include <omp.h>
#include "sequentialOMP.h"
#include "Timer.h"
#include <algorithm>
#include <cvmarkersobj.h>

#define MAX_THREADS 8
#define PAR_DAXPY 0
#define SIMD_DAXPY256 0
#define SIMD_DAXPY128 0
#define SIMD_L_Loop 1
using namespace std;
using namespace Concurrency::diagnostic;

namespace seqOMP {

void (*cdaxpy)(unsigned int, const double, double *, double *, unsigned int);
int (*cgaussian)(double **, int, int *);
// Fills A with random doubles, ppopulates b with row sums, returns largest val
double fillArray(double **a, int n, double *b) {
  double largestValue = 0.0;
  int init = 1325;

  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      init = 3125 * init % 65536; // cheap and nasty random generator
      a[j][i] = (static_cast<double>(init) - 32768.0) / 16384.0;
      largestValue = (a[j][i] > largestValue) ? a[j][i] : largestValue;
    }
  }
  // fill b with 0
  for (int i = 0; i < n; ++i) {
    b[i] = 0.0;
  }
  // add every element of each row of A to each row of B
  for (int j = 0; j < n; ++j) {
    for (int i = 0; i < n; ++i) {
      b[i] += a[j][i];
    }
  }

  return largestValue;
}

/* Purpose : Find largest component of double vector dx
n    : number of elements in input vector
dx   : double vector with n+1 elements, dx[0] is not used
dx_off : offset in reading dx
*/
int indexOfLargestElement(int n, double *dx, int dx_off) {
  double dmax, dtemp;
  int itemp = 0;

  if (n < 1) {
    itemp = -1;
  } else if (n == 1) {
    itemp = 0;
  } else {
    itemp = 0;
    dmax = abs(dx[0 + dx_off]);
    for (int i = 0; i < n; ++i) {
      dtemp = abs(dx[i + dx_off]);
      if (dtemp > dmax) {
        itemp = i;
        dmax = dtemp;
      }
    }
  }

  return itemp;
}

// Scales a vector by a constant
void scaleVecByConstant(int n, double da, double *dx, int dx_off, int incx) {
  if (n > 0) {
    if (incx != 1) {
      int nincx = n * incx;
      for (int i = 0; i < nincx; i += incx)
        dx[i + dx_off] *= da;
    } else {
      for (int i = 0; i < n; ++i)
        dx[i + dx_off] *= da;
    }
  }
}

/* Constant times a vector plus a vector
Purpose : To compute dy = da * dx + dy
--- Input ---
n       : number of elements in input vector(s)
scaler  : double scalar multiplier
dx      : double vector with n+1 elements
dy      : double vector with n+1 element
--- Output ---
dy = da * dx + dy, unchanged if n <= 0
*/
void daxpyS128(unsigned int n, const double scaler, double *dx, double *dy,
               unsigned int offset) {
```

```
  if ((n <= 0) || (scaler == 0)) {
    return;
  }
  double *const y = &dy[offset];
  double *const x = &dx[offset];
  const __m128d scalers = _mm_set1_pd(scaler);
  const int remainder = n % 2;
  const int nm1 = n - 1;

  for (int i = 0; i < nm1; i += 2) {
    // load X
    const __m128d xs = _mm_loadu_pd(&x[i]);
    // load y
    __m128d ys = _mm_loadu_pd(&y[i]);
    // mutliply X by scalers, add to Y
    ys = _mm_add_pd(ys, _mm_mul_pd(xs, scalers));
    // load back into y
    _mm_storeu_pd(&y[i], ys);
  }

  if (remainder != 0) {
    y[n - 1] += scaler * x[n - 1];
  }
}
void daxpyS256(unsigned int n, const double scaler, double *dx, ↩
    double *dy,
           unsigned int offset) {
  if ((n <= 0) || (scaler == 0)) {
    return;
  }
  double *const y = &dy[offset];
  double *const x = &dx[offset];

  const __m256d scalers = _mm256_set1_pd(scaler);
  const int remainder = n % 4;
  const int nm1 = n - 3;

  for (int i = 0; i < nm1; i += 4) {
    // load X
    const __m256d xs = _mm256_loadu_pd(&x[i]);
    // load y
    __m256d ys = _mm256_loadu_pd(&y[i]);
    // mutliply X by scalers, add to Y
    ys = _mm256_add_pd(ys, _mm256_mul_pd(xs, scalers));
    // load back into y
    _mm256_storeu_pd(&y[i], ys);
  }

  for (int i = n - remainder; i < n; ++i) {
    y[i] += scaler * x[i];
  }
}
void daxpyPar(unsigned int n, const double scaler, double *dx, ↩
    double *dy,
           unsigned int offset) {
  if ((n <= 0) || (scaler == 0)) {
    return;
  }
  double *const y = &dy[offset];
  double *const x = &dx[offset];
#pragma omp parallel
  {
    const int thread_id = omp_get_thread_num();
    const int thread_count = omp_get_num_threads();
    const int per_thread = n / thread_count;

    for (int i = (thread_id * per_thread); i < per_thread; ++i) {
      y[i] += scaler * x[i];
    }
  }
}

void daxpy(unsigned int n, const double scaler, double *dx, double *↩
    dy,
           unsigned int offset) {

  double *const y = &dy[offset];
  double *const x = &dx[offset];
  for (int i = 0; i < n; ++i) {
    y[i] += scaler * x[i];
  }
}

// Performs Gaussian elimination with partial pivoting
int gaussian_eliminate(double **a, int n, int *ipivot) {
  // Pointers to columns being worked on
  double *col_k;
  int nm1 = n - 1;
  int info = 0;

  if (nm1 >= 0) {
    int kp1, l;
    for (int k = 0; k < nm1; ++k) {
      // Set pointer for col_k to relevant column in a
      col_k = &a[k][0];
      kp1 = k + 1;

      // Find pivot index
      l = indexOfLargestElement(n - k, col_k, k) + k;
      ipivot[k] = l;

      // Zero pivot means that this column is already triangularized
      if (col_k[l] != 0) {
        double t;
        // Check if we need to interchange
        if (l != k) {
          t = col_k[l];
          col_k[l] = col_k[k];
          col_k[k] = t;
        }

        // Compute multipliers
        t = -1.0 / col_k[k];
        //Multiply collum by t
        scaleVecByConstant(n - kp1, t, col_k, kp1, 1);

        // Row elimination with column indexing
        for (int j = kp1; j < n; ++j) {
          // Set pointer for col_j to relevant column in a
          double *col_j = &a[j][0];

          double t = col_j[l];
          if (l != k) {
            col_j[l] = col_j[k];
            col_j[k] = t;
          }
          cdaxpy(n - kp1, t, col_k, col_j, kp1);
        }

      } else
        info = k;
    }
  }

  ipivot[n - 1] = n - 1;
  if (a[n - 1][n - 1] == 0) {
    info = n - 1;
  }

  return info;
}
// Performs Gaussian elimination with partial pivoting
int gaussian_eliminatePAR(double **a, int n, int *ipivot) {
  // Pointers to columns being worked on
  double *col_k;
  int nm1 = n - 1;
  int info = 0;

  if (nm1 >= 0) {
    int kp1, l;
    for (int k = 0; k < nm1; ++k) {
      // Set pointer for col_k to relevant column in a
      col_k = &a[k][0];
      kp1 = k + 1;

      // Find pivot index
      l = indexOfLargestElement(n - k, col_k, k) + k;
      ipivot[k] = l;
```

```
257
258     // Zero pivot means that this column is already triangularized
259     if (col_k[l] != 0) {
260       double t;
261       // Check if we need to interchange
262       if (l != k) {
263         t = col_k[l];
264         col_k[l] = col_k[k];
265         col_k[k] = t;
266       }
267
268       // Compute multipliers
269       t = -1.0 / col_k[k];
270       scaleVecByConstant(n - kp1, t, col_k, kp1, 1);
271
272 // Row elimination with column indexing
273 #pragma omp parallel for
274       for (int j = kp1; j < n; ++j) {
275         // Set pointer for col_j to relevant column in a
276         double *col_j = &a[j][0];
277
278         double t = col_j[l];
279         if (l != k) {
280           col_j[l] = col_j[k];
281           col_j[k] = t;
282         }
283         cdaxpy(n - kp1, t, col_k, col_j, kp1);
284       }
285
286     } else
287       info = k;
288   }
289 }
290
291 ipivot[n - 1] = n - 1;
292 if (a[n - 1][n - 1] == 0) {
293   info = n - 1;
294 }
295
296 return info;
297 }
298
299 // gaussian_eliminate
300 void dgesl(double **a, int n, int *ipivot, double *b) {
301   int k, nm1;
302   nm1 = n - 1;
303
304   // Solve a * x = b.  First solve l * y = b
305   if (nm1 >= 1) {
306     for (k = 0; k < nm1; ++k) {
307
308       int l = ipivot[k];
309       double t = b[l];
310
311       if (l != k) {
312
313         b[l] = b[k];
314         b[k] = t;
315       }
316
317       cdaxpy(n - (k + 1), t, &a[k][0], b, (k + 1));
318     }
319   }
320
321   // Now solve u * x = y
322   for (int kb = 0; kb < n; ++kb) {
323     k = n - (kb + 1);
324     b[k] /= a[k][k];
325     double t = -b[k];
326     cdaxpy(k, t, &a[k][0], b, 0);
327   }
328 }
329
330 // Multiply matrix m times vector x and add the result to vector y
331 void dmxpy(int n1, double *y, int n2, double *x, double **m) {
332   for (int j = 0; j < n2; ++j) {
333     for (int i = 0; i < n1; ++i) {
334       y[i] += x[j] * m[j][i];
335     }
336   }
337 }
338
339 // Runs the benchmark
340 void run(double **a, double *b, int n, int *ipivot) {}
341
342 // Validates the result
343 void validate(double **a, double *b, double *x, int n) {
344   // copy b into x
345   for (int i = 0; i < n; ++i) {
346     x[i] = b[i];
347   }
348
349   // reset A and B arrays to orignal rand values
350   double biggestA = fillArray(a, n, b);
351
352   for (int i = 0; i < n; ++i) {
353     b[i] = -b[i];
354   }
355
356   // multipy a*x, add to b
357   dmxpy(n, b, n, x, a);
358
359   double biggestB = 0.0;
360   double biggestX = 0.0;
361   for (int i = 0; i < n; ++i) {
362     biggestB = (biggestB > abs(b[i])) ? biggestB : abs(b[i]);
363     biggestX = (biggestX > abs(x[i])) ? biggestX : abs(x[i]);
364   }
365
366   double residn =
367       biggestB / (n * biggestA * biggestX * (2.2204460492503131e↩
            -016));
368   assert(residn < CHECK_VALUE);
369 }
370
371 int start(const unsigned int runs, const unsigned int threadCount,
372           const bool simd128, const bool simd256) {
373
374     double *a = new double [10000];
375     double *b = new double [10000];
376     double *c = new double [10000];
377
378     int init = 1325;
379     for (int j = 0; j < 10000; ++j) {
380       init = 3125 * init % 65536; // cheap and nasty random generator
381       a[j] = (static_cast<double>(init)-32768.0) / 16384.0;
382     }
383     for (int j = 0; j < 10000; ++j) {
384       init = 3125 * init % 65536; // cheap and nasty random generator
385       b[j] = (static_cast<double>(init)-32768.0) / 16384.0;
386     }
387     for (int j = 0; j < 10000; ++j) {
388       init = 3125 * init % 65536; // cheap and nasty random generator
389       c[j] = (static_cast<double>(init)-32768.0) / 16384.0;
390     }
391
392
393     for (int q = 1; q < 256; ++q) {
394       auto start = chrono::high_resolution_clock::now();
395       for (int j = 0; j < 10000; ++j) {
396         for (int i = 0; i < 10000; i += q) {
397           daxpy(q, c[i], a, b, i);
398         }
399       }
400       cout << q << ",\t" << chrono::duration_cast<chrono::↩
            nanoseconds>(chrono::high_resolution_clock::now() - start).↩
            count() << endl;
401     }
402
403
404     return 0;
405
406     ResultFile r;
407     r.name = "Sequential LinPack OMP" + to_string(runs);
408
409     if (simd256) {
410       cdaxpy = &daxpyS256;
411       r.name += "Simd256";
```

```
412  } else if (simd128) {
413   cdaxpy = &daxpyS128;
414   r.name += "Simd128";
415  } else {
416   cdaxpy = &daxpy;
417  }
418  if (threadCount > 1) {
419   unsigned int t = min((unsigned int)omp_get_max_threads(), ↩
         threadCount);
420   cgaussian = &gaussian_eliminatePAR;
421   omp_set_num_threads(t);
422   r.name += "ParLloopT" + to_string(t);
423  } else {
424   cgaussian = &gaussian_eliminate;
425  }
426  cout << r.name << endl;
427  r.headdings = {"Allocate Memory", "Create Input Numbers",
428                 " gaussian_eliminate", "Solve", "Validate"};
429
430  marker_series series;
431  // span *flagSpan = new span(series, 1, _T("flag span"));
432  // series.write_flag(_T("Here is the flag."));
433  // delete flagSpan;
434
435  Timer time_total;
436  for (size_t i = 0; i < runs; i++) {
437   // series.write_flag(0, (to_string(i).c_str()));
438   series.write_flag(_T("Here is the flag."));
439   span *flagSpan = new span(series, 1, _T("time_allocate"));
440   cout << i << endl;
441   // Allocate data on the heap
442   Timer time_allocate;
443   double **a = new double *[NSIZE];
444   for (size_t i = 0; i < NSIZE; ++i) {
445    // a[i] = new double[SIZE];
446    a[i] = (double *)_aligned_malloc(NSIZE * sizeof(double), sizeof(↩
          double));
447   }
448
449   double *b =
450       (double *)_aligned_malloc(NSIZE * sizeof(double), sizeof(↩
           double));
451
452   double *x =
453       (double *)_aligned_malloc(NSIZE * sizeof(double), sizeof(↩
           double));
454
455   int *ipivot = (int *)_aligned_malloc(NSIZE * sizeof(int), sizeof(int↩
         ));
456   // double *b = new double[SIZE];
457   // double *x = new double[SIZE]
458   ;
459   // int *ipivot = new int[SIZE];
460   delete flagSpan;
461   time_allocate.Stop();
462
463   // Main application
464   flagSpan = new span(series, 1, _T("time_genRnd"));
465   Timer time_genRnd;
466
467   auto aa = fillArray(a, NSIZE, b);
468
469   delete flagSpan;
470   time_genRnd.Stop();
471
472   flagSpan = new span(series, 1, _T("time_gauss"));
473   Timer time_gauss;
474
475   cgaussian(a, NSIZE, ipivot);
476
477   delete flagSpan;
478   time_gauss.Stop();
479
480   flagSpan = new span(series, 1, _T("solve"));
481   Timer time_dgesl;
482
483   dgesl(a, NSIZE, ipivot, b);
484
485   delete flagSpan;
486   time_dgesl.Stop();
487
488   flagSpan = new span(series, 1, _T("time_validate"));
489   Timer time_validate;
490   validate(a, b, x, NSIZE);
491
492   delete flagSpan;
493   time_validate.Stop();
494
495   r.times.push_back({time_allocate.Duration_NS(), time_genRnd.↩
         Duration_NS(),
496                      time_gauss.Duration_NS(), time_dgesl.Duration_NS(),
497                      time_validate.Duration_NS()});
498   // Free the memory
499   for (size_t i = 0; i < NSIZE; ++i) {
500    _aligned_free(a[i]);
501   }
502   _aligned_free(b);
503   _aligned_free(x);
504   _aligned_free(ipivot);
505   // delete[] b;
506   // delete[] x;
507   // delete[] ipivot;
508  }
509  r.CalcAvg();
510  r.PrintToCSV(r.name);
511  time_total.Stop();
512  cout << "Total Time: " << Timer::format(time_total.Duration_NS↩
         ());
513  return 0;
514 }
515 }
```