Matrix Multiplication

1. Naive Approach

```
for(int i = 0; i < p; i++){
    for(int j = 0; j < r; j++){
        result->matrix[i][j] = 0;
        for(int k = 0; k < q; k++){
            result->matrix[i][j] += a->matrix[i][k]*b->matrix[k][j];
        }
    }
}
```

p, q, r all are equal to N

N	10	100	500	1000
Time	0.000084	0.007561	0.500851	4.488311

1.1. Caching

1. Value of result->matrix[i][j] is cached and stored in the variable.
2. This reduces the time as write to result->matrix[i][j] is done only once.

```
for(int i = 0; i < p; i++){
    for(int j = 0, val = 0; j < r; j++){
        for(int k = 0; k < q; k++){
            val += a->matrix[i][k]*b->matrix[k][j];
        }
        result->matrix[i][j] = val;
    }
}
```

N	10	100	500	1000
Time	0.000078	0.006202	0.345185	3.143836

2. Interchanging Loops

Interchanging the loops of j, k decreases the time in comparison to 1. as it improves spatial locality of matrix b.

```
for(int i = 0; i < p; i++){
    for(int j = 0; j < r; j++){
        result->matrix[i][j] = 0;
    }
}
for(int i = 0; i < p; i++){
    for(int k = 0; k < q; k++){
        for(int j = 0; j < r; j++){
            result->matrix[i][j] += a->matrix[i][k]*b->matrix[k][j];
        }
    }
}
```

N	10	100	500	1000
Time	0.000082	0.007025	0.398650	3.193811

2.1. Caching

1. Value of a->matrix[i][k] is cached and stored in the variable.
2. This reduces the time as read to a->matrix[i][j] is done only once.

```
for(int i = 0; i < p; i++){
    for(int j = 0; j < r; j++){
        result->matrix[i][j] = 0;
    }
}
for(int i = 0, x; i < p; i++){
```

```
        for(int k = 0, x; k < q; k++){
            x = a->matrix[i][k];
            for(int j = 0; j < r; j++){
                result->matrix[i][j] += x*b->matrix[k][j];
            }
        }
    }
```

N	10	100	500	1000
Time	0.000070	0.012780	0.368548	2.807992

2.2. Caching, Partial Loop Unrolling, Global temporary matrix

Caching decreases the time by 0.5 seconds, Partial loop unrolling decreases the time by 0.5 seconds and using temporary matrix instead of struct matrix decreases the time by 0.2 seconds.

```
    int rmat[1000][1000], bmat[1000][1000], amat[1000][1000];

    for(int i = 0; i < p; i++){
        for(int j = 0; j < r; j++){
            rmat[i][j] = 0;
        }
    }
    for(int i = 0; i < q; i++){
        for(int j = 0; j < r; j++){
            bmat[i][j] = b->matrix[i][j];
        }
    }
    for(int i = 0; i < p; i++){
        for(int j = 0; j < q; j++){
            amat[i][j] = a->matrix[i][j];
        }
    }
    for(int i = 0; i < p; i++){
```

```
    for(int k = 0, j; k < q; k++){
         x = amat[i][k]
        for(j = 0; j < r - 7; j += 8){
            rmat[i][0+j] += x*bmat[k][0+j];
            rmat[i][1+j] += x*bmat[k][1+j];
            rmat[i][2+j] += x*bmat[k][2+j];
            rmat[i][3+j] += x*bmat[k][3+j];
            rmat[i][4+j] += x*bmat[k][4+j];
            rmat[i][5+j] += x*bmat[k][5+j];
            rmat[i][6+j] += x*bmat[k][6+j];
            rmat[i][7+j] += x*bmat[k][7+j];
        }
        for( ; j < r; j++){
            rmat[i][j] += x*bmat[k][j];
        }
    }
}
for(int i = 0; i < p; i++){
    for(int j = 0; j < r; j++){
        result->matrix[i][j] = rmat[i][j];
    }
}
```

10	100	500	1000
0.000314	0.005435	0.267119	2.016880

2.3 Global Vector to cache result->matrix[i]

A vector is initialized globally and used in place of result->matrix[i][j] as i remains same for all i as j changes and vector is stored back to result->matrix[i] before the start of loop i. This decreases the time by 0.6 seconds as this improves the caching.

```
int rmat[1000][1000],bmat[1000][1000],amat[1000][1000],rtmp[1000];

for(int i = 0; i < p; i++){
```

```
    for(int j = 0; j < r; j++){
        rmat[i][j] = 0;
    }
}
for(int i = 0; i < q; i++){
    for(int j = 0; j < r; j++){
        bmat[i][j] = b->matrix[i][j];
    }
}
for(int i = 0; i < p; i++){
    for(int j = 0; j < q; j++){
        amat[i][j] = a->matrix[i][j];
    }
}
for(int i = 0; i < p; i++){
    for(int j = 0; j < r; j++)
        rtmp[j] = 0;
    for(int k = 0, j; k < q; k++){
         x = amat[i][k]
        for(j = 0; j < r - 7; j += 8){
            rtmp[0+j] += x*bmat[k][0+j];
            rtmp[1+j] += x*bmat[k][1+j];
            rtmp[2+j] += x*bmat[k][2+j];
            rtmp[3+j] += x*bmat[k][3+j];
            rtmp[4+j] += x*bmat[k][4+j];
            rtmp[5+j] += x*bmat[k][5+j];
            rtmp[6+j] += x*bmat[k][6+j];
            rtmp[7+j] += x*bmat[k][7+j];
        }
        for( ; j < r; j++){
            rtmp[j] += x*bmat[k][j];
        }
    }
    for(int j = 0; j < r; j++)
        rmat[i][j] = rtmp[j];
}
for(int i = 0; i < p; i++){
```

```
        for(int j = 0; j < r; j++){
            result->matrix[i][j] = rmat[i][j];
        }
    }
```

10	100	500	1000
0.000269	0.004427	0.192212	1.418374

2.4 ‘Register’ Keyword, Memset, Loop Unrolling, Caching

1. More frequently used variable is added to register which reduces the time by 0.6.
2. Base pointer of result, b matrix is stored in variable stored in register for faster access to pointers and pointer is incremented every time for faster access to its value.

```
    register int i, j, k, x, *pb = NULL, *pr = result->matrix[0];
    memset(pr, 0, p*r*sizeof(int));
    for(i = 0; i < p; i++){
        for(k = 0; k < q; k++){
            pb = b->matrix[k];
            x = a->matrix[i][k];
            pr = result->matrix[i];
            for(j = 0; j < r - 15; j += 16){
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
```

```
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
                *pr++ += x*(*pb++);
            }
            for( ; j < r; j++){
                *pr++ += x*(*pb++);
            }
        }
    }
```

10	100	500	1000
0.000121	0.003823	0.110865	0.781997

3. Transposing Matrix b and row to row multiplication

This also decreases the time in comparison to 1. as after taking the transpose of b, spatial locality is improved.

```
    int c[1000][1000];
    for(int i = 0; i < q; i++){
        for(int j = 0; j < r; j++){
            c[i][j] = b->matrix[j][i];
        }
    }
    for(int i = 0; i < p; i++){
        for(int j = 0; j < r; j++){
            result->matrix[i][j] = 0;
            for(int k = 0; k < q; k++){
                result->matrix[i][j]+=a->matrix[i][k]*c[j][k];
            }
        }
    }
```

N	10	100	500	1000
Time	0.000147	0.007294	0.401135	3.124750

4. Single Tiling

1. S = 16 gives the best result for single tiling.
2. Single tiling is not efficient here because we are not using parallelism.

```
for(int i = 0; i < p; i++){
    for(int j = 0; j < r; j++){
        result->matrix[i][j] = 0;
    }
}
int s = 16;
for(int ih = 0; ih < p; ih += s){
    for(int kh = 0; kh < r; kh += s){
        for(int jh = 0; jh < q; jh += s){
            for(int il = 0 ; il < s; il++){
                for(int kl = 0; kl < s; kl++){
                    for(int jl = 0; jl < s; jl++){
                        result->matrix[ih+il][jh+jl] +=
             a->matrix[ih+il][kh+kl]*b->matrix[kh+kl][jh+jl];
                    }
                }
            }
        }
    }
}
```

N	10	100	500	1000
Time	0.000274	0.011374	0.601132	4.437947

5. Double Tiling

1. S = 16, T = 16 gives the best result on Double Tiling
2. Double Tiling is not efficient here as we are not using parallelism.

```
for(int i = 0; i < p; i++){
    for(int j = 0; j < r; j++){
        result->matrix[i][j] = 0;
    }
}
int s = 16, t = 16;
for(int ih = 0; ih < p; ih += s){
    for(int jh = 0; jh < q; jh += s){
        for(int kh = 0; kh < r; kh += s){
            for(int im = 0 ; im < s; im += t){
                for(int jm = 0; jm < s; jm += t){
                    for(int km = 0; km < s; km += t){
                        for(int il = 0 ; il < t; il++){
                            for(int kl = 0; kl < t; kl++){
                                for(int jl = 0; jl < t; jl++){
                             result->matrix[ih+im+il][jh+jm+jl] +=
        a->matrix[ih+im+il][kh+km+kl]*b->matrix[kh+km+kl][jh+jm+jl];
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

N	10	100	500	1000
Time	0.000259	0.014145	0.757729	5.649673

Result:

Loop interchanging, Loop unrolling, caching and use of Registers for frequently used variables gives the best result.

Cachegrind:

```
valgrind --tool=cachegrind ./a.out
==3918== Cachegrind, a cache and branch-prediction profiler
==3918== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==3918== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3918== Command: ./a.out
==3918==
--3918-- warning: L3 cache found, using its data for the LL simulation.
time 28.145928
==3918==
==3918== I   refs:    10,493,121,738
==3918== I1  misses:            1,079
==3918== LLi misses:            1,073
==3918== I1  miss rate:          0.00%
==3918== LLi miss rate:          0.00%
==3918==
==3918== D   refs:     3,141,067,439  (2,121,052,870 rd   + 1,020,014,569 wr)
==3918== D1  misses:      62,816,722  (   62,628,578 rd   +       188,144 wr)
==3918== LLd misses:         377,676  (      189,597 rd   +       188,079 wr)
==3918== D1  miss rate:          2.0% (          3.0%     +           0.0%  )
==3918== LLd miss rate:          0.0% (          0.0%     +           0.0%  )
==3918==
==3918== LL refs:         62,817,801  (   62,629,657 rd   +       188,144 wr)
==3918== LL misses:          378,749  (      190,670 rd   +       188,079 wr)
==3918== LL miss rate:           0.0% (          0.0%     +           0.0%  )
```

Gprof:

1. Flat profile:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 98.57      0.78     0.78        1   778.67   778.67  matrix_multiply
  1.27      0.79     0.01                             main
```

2. Call Graph

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.01	0.78		main [1]
		0.78	0.00	1/1	matrix_multiply [2]

		0.78	0.00	1/1	main [1]
[2]	98.7	0.78	0.00	1	matrix_multiply [2]

Perf:

Overhead	Command	Shared Object	Symbol
97.87%	a.out	a.out	[.] matrix_multiply
0.60%	a.out	libc-2.23.so	[.] __random
0.35%	a.out	libc-2.23.so	[.] __random_r
0.32%	a.out	a.out	[.] main
0.28%	a.out	libc-2.23.so	[.] rand
0.10%	a.out	[kernel.kallsyms]	[k] error_entry
0.08%	a.out	[kernel.kallsyms]	[k] page_add_file_rmap
0.07%	a.out	libc-2.23.so	[.] __memset_avx2

Merge Sort

1. Naive Merge Sort :
   a. N : 1e6 , Time : 0.23 sec.
   b. No optimization was applied

2. Using Loop Unrolling for 8 unrolls :
   a. N : 1e6 , Time : 0.20 sec
   b. Loop Unrolling helps to run various statements parallelly.

3. Using Loop Unrolling for 8 unrolls and Insertion Sort for small values of n :
   a. N : 1e6 , Time : 0.16

b. Insertion sort usually is a bit faster than other simple quadratic sort methods partially due to using of element chain shifting instead of full swapping.

4. Using Loop Unrolling for 16 unrolls and Insertion Sort for small values of n (<= 50):
    a. N : 1e6 , Time : 0.15
    b. On testing with various values of unrolls 16 unrolls for loop unrolling seem to be best.

5. Using Selection Sort instead of Insertion Sort in previous implementation:
    a. N : 1e6, Time : 0.18
    b. With Selection Sort for small values of n time got increased so we used Insertion Sort for small values of n along with merge sort.

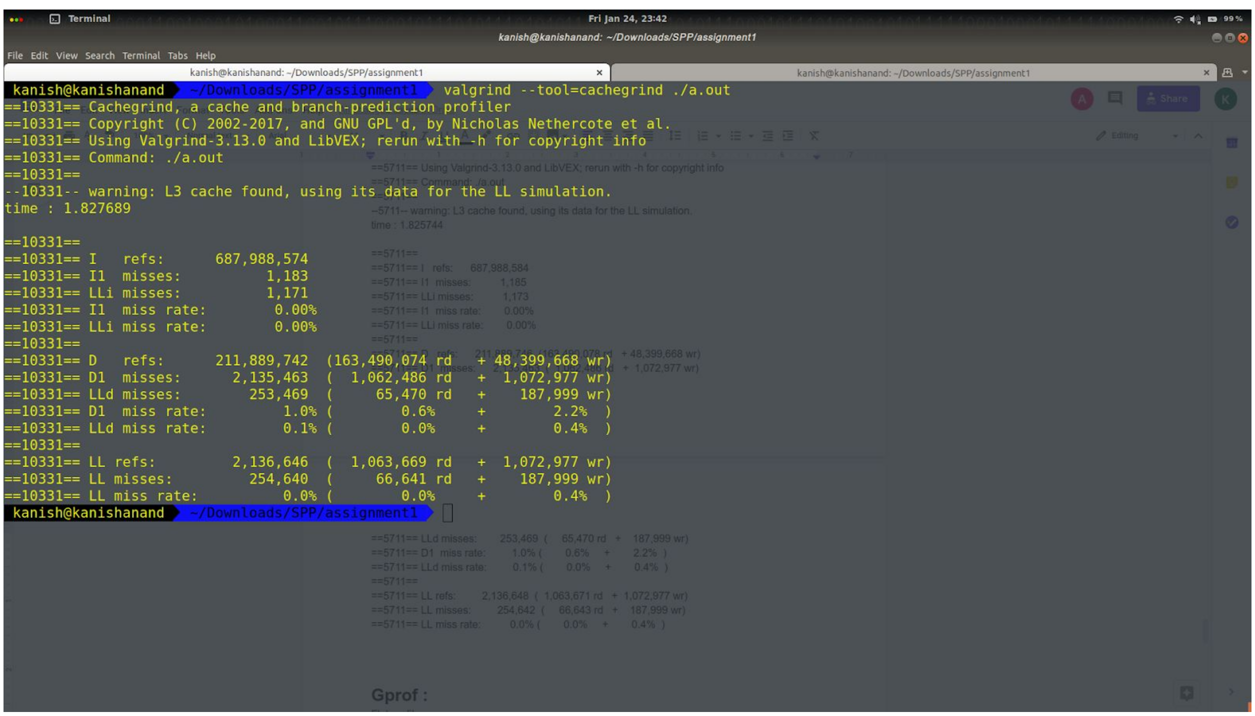
6. Using Loop Unrolling for 16 unrolls , Insertion Sort for small values of n and using register int instead of int :
    a. N : 1e6, Time : 0.10
    b. On taking register int for some values those values gets stored directly in registers so it takes very less time to access these values.

Cachegrind :

```
==5711== Cachegrind, a cache and branch-prediction profiler
==5711== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==5711== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==5711== Command: ./a.out
==5711==
--5711-- warning: L3 cache found, using its data for the LL simulation.
```

time : 1.825744

==5711==
==5711== I   refs: 687,988,584
==5711== I1 misses: 1,185
==5711== LLi misses: 1,173
==5711== I1 miss rate: 0.00%
==5711== LLi miss rate: 0.00%
==5711==
==5711== D   refs: 211,889,746 (163,490,078 rd   + 48,399,668 wr)
==5711== D1 misses: 2,135,463 (1,062,486 rd   + 1,072,977 wr)
==5711== LLd misses: 253,469 (65,470 rd   +   187,999 wr)
==5711== D1 miss rate: 1.0% (0.6%   +   2.2%)
==5711== LLd miss rate: 0.1% (0.0%   +   0.4%)
==5711==
==5711== LL refs: 2,136,648 (1,063,671 rd   + 1,072,977 wr)
==5711== LL misses: 254,642 (66,643 rd   +   187,999 wr)
==5711== LL miss rate: 0.0% (0.0%   +   0.4%)

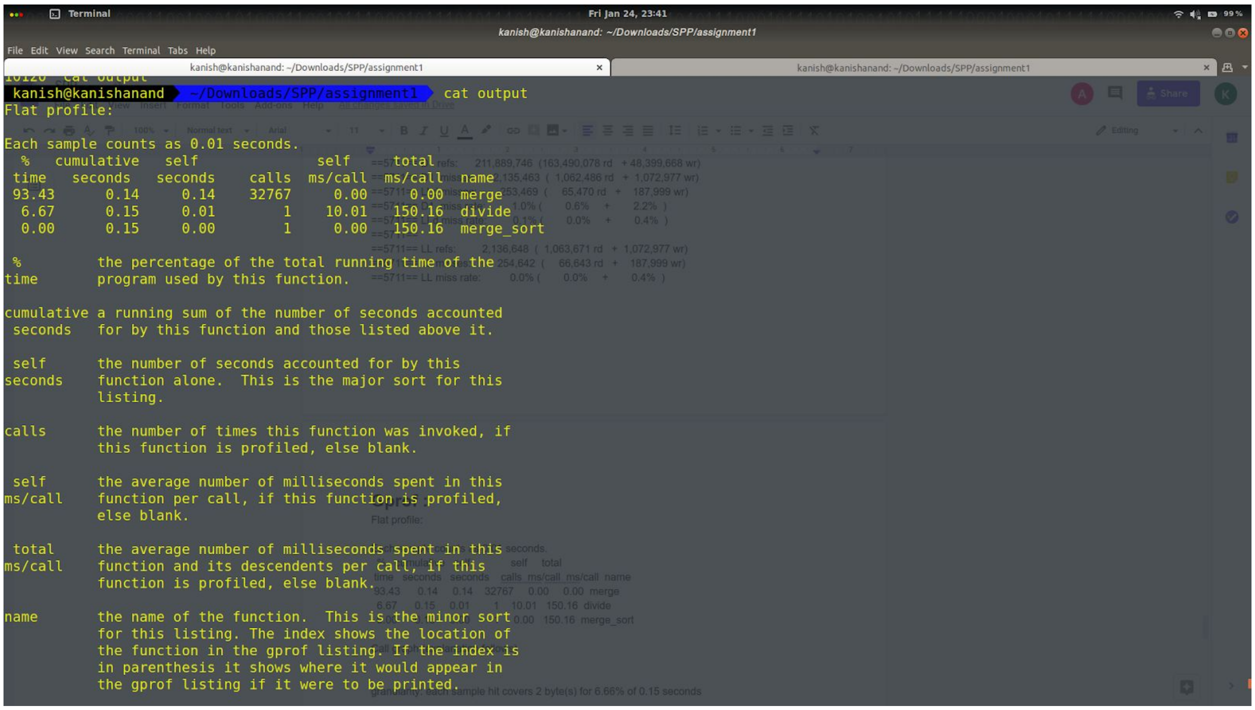


Gprof :

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
93.43	0.14	0.14	32767	0.00	0.00	merge
6.67	0.15	0.01	1	10.01	150.16	divide
0.00	0.15	0.00	1	0.00	150.16	merge_sort

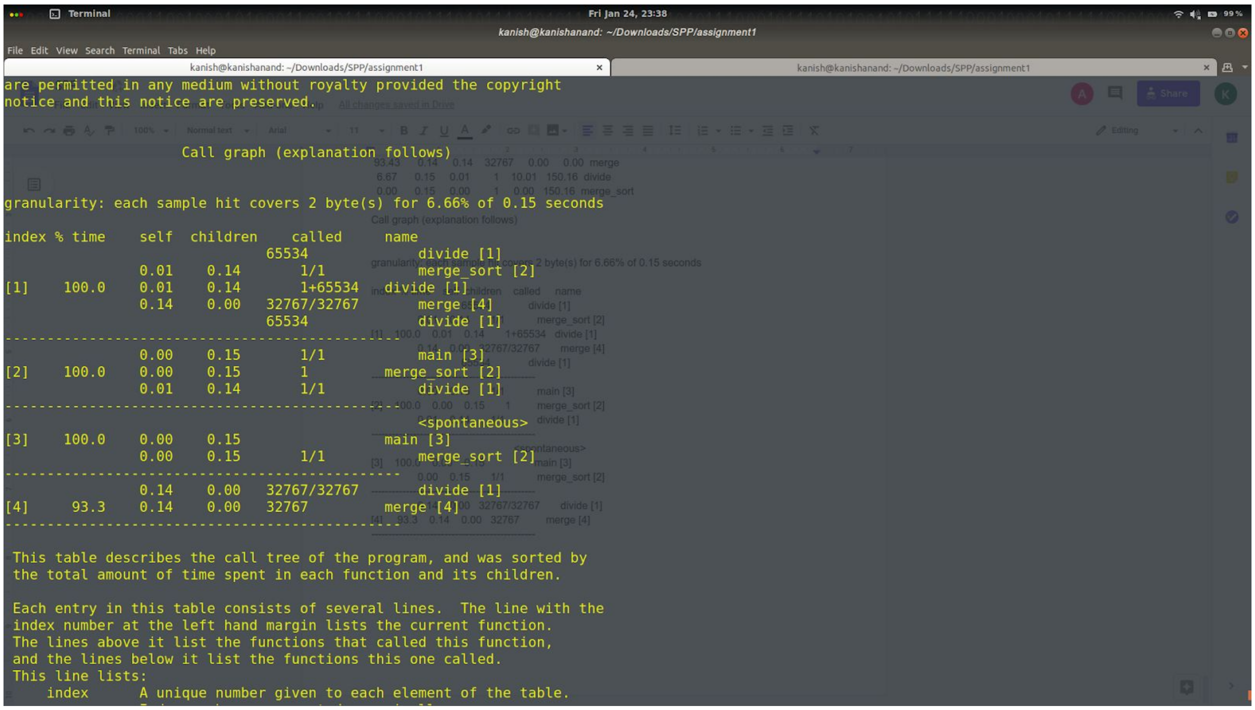


Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 6.66% of 0.15 seconds

```
index % time    self  children    called     name
                               65534             divide [1]
                0.01    0.14       1/1           merge_sort [2]
[1]    100.0    0.01    0.14       1+65534   divide [1]
                0.14    0.00   32767/32767       merge [4]
                               65534             divide [1]
-----------------------------------------------
                0.00    0.15       1/1           main [3]
[2]    100.0    0.00    0.15       1         merge_sort [2]
                0.01    0.14       1/1           divide [1]
-----------------------------------------------
                                                 <spontaneous>
```

```
[3]    100.0    0.00    0.15                main [3]
                0.00    0.15       1/1          merge_sort [2]
-----------------------------------------------
                0.14    0.00   32767/32767       divide [1]
[4]     93.3    0.14    0.00   32767          merge [4]
-----------------------------------------------
```



```
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.

                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 6.66% of 0.15 seconds

index % time    self  children    called     name
                               65534             divide [1]
                0.01    0.14       1/1           merge_sort [2]
[1]    100.0    0.01    0.14       1+65534   divide [1]
                0.14    0.00   32767/32767       merge [4]
                               65534             divide [1]
-----------------------------------------------
                0.00    0.15       1/1           main [3]
[2]    100.0    0.00    0.15       1         merge_sort [2]
                0.01    0.14       1/1           divide [1]
-----------------------------------------------
                                                 <spontaneous>
[3]    100.0    0.00    0.15                 main [3]
                0.00    0.15       1/1           merge_sort [2]
-----------------------------------------------
                0.14    0.00   32767/32767       divide [1]
[4]     93.3    0.14    0.00   32767         merge [4]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each entry in this table consists of several lines.  The line with the
 index number at the left hand margin lists the current function.
 The lines above it list the functions that called this function,
 and the lines below it list the functions this one called.
 This line lists:
     index      A unique number given to each element of the table.
```

Perf :

```
75.14%  k.out   k.out                [.] merge
15.83%  k.out   k.out                [.] divide
 2.74%  k.out   k.out                [.] main
 2.02%  k.out   libc-2.23.so         [.] rand
 1.17%  k.out   libc-2.23.so         [.] __random_r
 1.16%  k.out   libc-2.23.so         [.] __random
 0.60%  k.out   ld-2.23.so           [.] strlen
 0.57%  k.out   [kernel.kallsyms]    [k] get_page_from_freelist
 0.24%  k.out   [kernel.kallsyms]    [k] lru_cache_add_active_or_unevictable
 0.24%  k.out   [kernel.kallsyms]    [k] __mod_zone_page_state
 0.24%  k.out   [kernel.kallsyms]    [k] swapgs_restore_regs_and_return_to_usermode
 0.04%  k.out   [kernel.kallsyms]    [k] perf_event_comm_output
 0.00%  perf    [kernel.kallsyms]    [k] native_sched_clock
 0.00%  perf    [kernel.kallsyms]    [k] native_write_msr
```