The platform used for this question is **COE System**. Below are the system details for **COE:**

Model name	Intel(R) Xeon(R) CPU X5650
Frequency	2.67GHz
Vendor ID	GenuineIntel
CPU cores	24
Architecture	x86_64
OS	CentOS Linux release 7.4.1708 (Core)
Cache	L1 data cache: 32K L1 instruction cache: 32K L2 cache: 256K L3 cache: 12288K
RAM	47.2 GB
CPU op-mode(s)	32-bit, 64-bit
Thread(s) per core	2
Core(s) per socket	6
Socket(s):	2
NUMA node(s)	2
CPU MHz	2660.023
BogoMIPS	5320.04
Virtualization	VT-x

The three single threaded benchmarks used for this question are as follows:

1. **Linpack Bench:**
   - The LINPACK benchmark is a test problem used to rate the performance of a computer on a linear algebra problem.
   - The test problem requires the user to set up a random dense matrix A with double floating point data type of size N = 1000, and a right-hand side vector B which is the product of A, and a vector X of all 1's. The first task is to compute an LU factorization of A. The second task is to use the LU factorization to solve the linear system **A*X = B.**
   - The number of floating point operations required for these two tasks is roughly

     ops = 2 * N*N*N / 3 + 2 * N * N
   - The source code was obtained from https://people.sc.fsu.edu/~jburkardt/cpp_src/linpack_bench/linpack_bench.html

2. **Memory Test:**
   - MEMORY_TEST, a C++ program that declares and uses a sequence of larger and larger arrays, to see what the memory limits are on a given computer. This program is a memory intensive program which uses three data types(integer, float, and double) and two data structures(Vector and Matrix).
   - The program tries an increasing series of values of N, using powers of 2, between limits that you set.
   - The source code was obtained from https://people.sc.fsu.edu/~jburkardt/cpp_src/memory_test/memory_test.html

3. **Sparse Matrix Multiplication:**
   - This is a C++ program which creates two sparse matrices with double floating point data type and multiplies them for a range of matrix dimensions.
   - The range of matrix dimension, step increment and fraction of sparsity should be provided as input.
   - This code was written from scratch by me.

Part A:

- Time taken to run Linpack benchmark on COE System with default optimization(O0) for 5 times is provided below:

Iteration number	1	2	3	4	5
Time Taken(sec)	1.17406	1.17375	1.17571	1.17402	1.17363

Average run time: **1.174234 sec** Maximum difference in time taken: 0.00208 sec ~ 2ms

- Time taken to run Memory test benchmark on COE System with default optimization(O0) for 5 times is provided below:
  N = 25

Iteration number	1	2	3	4	5	Average
Integer Vector Time(sec)	0.468	0.468	0.469	0.476	0.469	0.47
Float Vector Time(sec)	0.454	0.454	0.454	0.454	0.454	0.454
Double Vector Time(sec)	0.545	0.545	0.545	0.545	0.545	0.545
Integer Matrix Time(sec)	0.491	0.491	0.492	0.491	0.491	0.491
Float Matrix Time(sec)	0.480	0.480	0.480	0.480	0.480	0.480
Double Matrix Time(sec)	0.559	0.559	0.558	0.559	0.559	0.559

- Time taken to run Sparse Multiplication benchmark on COE System with default optimization(O0) for 5 times is provided below:

Iteration number	1	2	3	4	5
Time Taken(sec)	0.2811	0.2810	0.2810	0.2810	0.2810

Average run time: **0.2810 sec** Maximum difference in time taken: 0.0001 sec

- From the above recorded measurements, the difference in time taken across runs is significantly less for all the 3 benchmarks. The possible reasons for this difference are as follows:
    - There would be many background processes running like processing network packets, saving or logging data to the disk, and checking network time.
    - Initial conditions of the system like caches and branch predictors. When running the program for the first time, relevant data might not be there in the cache due to which there would be cache misses.

Part B:

The following are the set of optimization levels I used for optimizing the three benchmarks.

1. **O0**: This optimization flag reduces compilation time and makes debugging produce the expected results. This optimization flag is used by default.
2. **O1**: The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time. This set turns on additional optimization flags along with O0 which optimizes looping(*-fmove-loop-invariants*), constant declaration(*-fmerge-constants*) and branch prediction(*-fguess-branch-probability*).
3. **O2:** This option increases both compilation time and the performance of the generated code. It turns on all optimization flags specified by -O1. This optimizes more than O1. This set turns on additional optimization flags like -falign-functions which align the start of functions to the next power of two.
4. **O3:** This option optimizes further. It turns on all optimizations specified by -O2 and also turns on the optimization flags like -floop-interchange, -floop-unroll-and-jam which does further loop optimizations.
5. **Os:** This option optimizes for size. -Os enables all -O2 optimizations except those that often increase code size like -falign-functions, and -falign-loops.
6. **Ofast:** -Ofast enables all -O3 optimizations. It also enables optimizations that are not valid for all standard-compliant programs. It turns on -ffast-math, -fallow-store-data-races and the Fortran-specific -fstack-arrays, unless -fmax-stack-var-size is specified, and -fno-protect-parens.
7. **Og:** -Og enables all -O1 optimization flags except for those that may interfere with debugging like -fbranch-count-reg, -fmove-loop-invariants

Linpack Benchmark:

- The below table shows the execution time and size of the executable for each optimization applied.

Optimization	**O0**	**O1**	**O2**	**O3**	**Os**	**Ofast**	**Og**
Time(sec)	1.49744	0.448628	0.387236	0.391007	0.387044	0.390663	0.481809
Size(KB)	22.4	18.2	18.4	22.6	18.1	22.6	18.3

- This benchmark solves the real system A*X=B using the factors computed by LU Factorization(DGEFA). The code mainly contains for loops, if-else conditions and daxpy. The code also contains many constants.
- With O1 optimization, flag *-fguess-branch-probability* predicts branch probability using heuristics which speeds up the execution. -fmove-loop-invariants and -fmove-loop-stores move invariant stores to after the end of the loop in exchange for carrying the stored value in a register across the iteration. This increases execution speed as it reads from register and not from cache.
- With O2 optimization, flag -falign-loops align loops to a power of two boundaries. As the loops are executed many times, this makes up for any execution of the dummy padding instructions. Due to this aligning, the size of the executable increased from 18.2KB to 18.4 KB.
- With O3 optimization, there is not much difference compared to O2 optimization in terms of execution speed. This may be due to the presence of a certain degree of loop unrolling and handling of variable stride in the code.
- As Os optimization is the same as O2 optimization except for the increase in code size, the execution time is same for both and the code size for Os is less than O2.
- As Ofast optimization is similar to O3, the execution time and code size is the same.
- As Og is similar to O1 except for a few flags like loop optimization which interferes with debugging, the execution time is higher compared to O1.

Memory Benchmark:

- The below table shows the execution time and size of the executable for each optimization applied.

Optimization	**O0**	**O1**	**O2**	**O3**	**Os**	**Ofast**	**Og**
Size	18.7	18.3	18.5	22.8	18.2	26.9	18.4
Integer Vector Time(sec)	0.59723	0.44638	0.44631	0.44633	0.46035	0.40284	0.46045
Float Vector Time(sec)	0.57755	0.41779	0.41796	0.41779	0.43206	0.38746	0.41821
Double Vector Time(sec)	0.69085	0.48583	0.51423	0.48664	0.48608	0.42427	0.48621
Integer Matrix Time(sec)	0.57809	0.46053	0.44681	0.44642	0.46315	0.40299	0.50331
Float Matrix Time(sec)	0.55202	0.40971	0.39516	0.39837	0.4101	0.38772	0.41819
Double Matrix Time(sec)	0.64247	0.44215	0.44189	0.44197	0.44561	0.38724	0.44319

- This benchmark primarily tests memory limits on a given computer. The benchmark performs a simple looping on all the elements of a vector or matrix. As the matrix elements are arranged similar to a vector, accessing matrix elements from cache is optimized.
- As there is little computation and little scope for optimization, there is no significant difference in execution time across O1, O2, O3, Os and Og. There is a slight reduction in execution time when compared to O0.
- There is a reduction in execution time in Ofast optimization compared to other optimization sets across the above data types and data structures. This may be due to the enabling of flag *-ffast-math* which reduces math operation computation time.

Sparse Multiplication Benchmark:

- The below table shows the execution time and size of the executable for each optimization applied.

Optimization	**O0**	**O1**	**O2**	**O3**	**Os**	**Ofast**	**Og**
Size	14.9	14.4	14.6	14.7	14.2	14.6	14.6
Time (sec)	1.44304	0.358567	0.374178	0.365952	0.538785	0.365898	0.540765

- This benchmark performs matrix multiplication. The code mainly contains a nested for loop to parse through the matrix.
- With loop optimizations present in the O1 optimization set, there is reduction in execution time compared to O0.
- As the code contains a simple matrix parsing for loop, there wouldn't be many instances of branch prediction. As a result, the execution time of O1, O2, O3 and fast are similar.
- As the flag *-falign-loops* is not enabled for optimization set Os, the execution time is higher compared to O1, O2 and O3.

Part C:

Parallelizing Linpack Benchmark using Pthreads:

1. **Program Overview:**
    - **a.** The first task is to compute an LU factorization of A. The second task is to use the LU factorization to solve the linear system A*X = B**.**
    - **b.** LU Factorization:
        - i. For each column k in a matrix dimension N*N:
            1. Calculate the row position of maximum value in the current column and get pivot indices.
            2. Scale the elements in the current column with the inverse of the first element in the column.
            3. Loop through i=k+1 to N and j=k+1 to N and update data[i][j] as data[i][j] -= data[k][j]*data[i][k]
    - **c.** Solve A*X = B**:**
        - i. This function loops through pivot indices(calculated from the above step) and performs daxpy operations.

2. **Parallelizing using M Pthreads:**
    a. LU Factorization:
        i. For each column k in a matrix dimension N*N:
            1. Divide the column by M threads using the start and end indices and find row index of local maximum value. Then do a maximum of all these local maximum values and find the row position of the global maximum.
            2. Divide the column by M threads using the start and end indices and scale the elements between these indices parallely.
            3. Divide the outer loop such that each thread has (N-k-1)/M elements and loop through j=k+1 to N and update data[i][j] as data[i][j] -= data[k][j]*data[i][k]

        As we have to execute each step only after the completion of previous steps, we can't divide the outermost for-loop and run parallely.

    b. Solve A*X = B**:**
        i. The number of elements in the pivot indices will be divided by M threads and daxpy operations will be performed parallely.

Parallelizing Memory Benchmark using Pthreads:

1. **Program Overview:**
    a. This benchmark primarily tests memory limits on a given computer.
        i. This program loops through a large number of elements in a vector or a matrix and scales them by a constant factor.
        ii. It loops again to calculate the average of all the elements in a vector or a matrix.

2. **Parallelizing using M Pthreads:**
    a. For vector,
        i. Divide the vector containing a large number of elements by M threads and perform scalar multiplication parallely.
        ii. Again divide the vector by M threads and add the elements locally. Then add all these locally calculated summations to get the global sum. Finally divide by the number of elements to get the global average of elements in the vector.

b. For matrix,
    i. In the program, as the matrix elements are arranged similar to a array(arr[i + j*num_cols]), the above parallelization method applies here.

Parallelizing Sparse Matrix Multiplication using Pthreads:

1. **Program Overview:**
    a. The program randomly generates two sparse matrices taking *frac_sparse* (fraction of sparsity) as input and multiplies them.
    b. The program executes the above step for a range of matrix dimensions provided.

2. **Parallelizing using M Pthreads:**

Parallelizing matrix multiplication can be done in many ways like elementwise parallelism in Matrix C, row wise parallelism in Matrix C. Here we do row wise parallelism in Matrix C.

a. For a matrix multiplication A X B = C where A and B are square matrices with dimension N,
    i. Take the transpose of matrix B (column major order) to optimize for cache memory access.
    ii. Divide the rows in Matrix A by M threads such that each thread performs the following:

```
Loop through i in (k, k+N/M):
    Loop through j in B:
        Loop through elements in Bj:
            Perform multiplication and addition
```