Github repository with my code: https://github.com/savioarthu/HPCesga_Tools.git

Step 0. Check your serial implementation with Valgrind Memcheck to find possible memory issues. Have you found any? Which ones? Solve them and get a clean Valgrind run.

Process for Valgrind Memcheck:

module load gcccore/6.4.0

module load valgrind/3.15.0

gcc -g -o dgesv dgesv.c -lm

valgrind --leak-check=yes ./dgesv 512

```
==30087== Memcheck, a memory error detector
==30087== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==30087== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==30087== Command: ./dgesv 512
==30087==
Time taken by my implementation: 67.89s
==30087==
==30087== HEAP SUMMARY:
==30087==     in use at exit: 8 bytes in 1 blocks
==30087==   total heap usage: 12 allocs, 11 frees, 18,908,216 bytes allocated
==30087==
==30087== LEAK SUMMARY:
==30087==    definitely lost: 0 bytes in 0 blocks
==30087==    indirectly lost: 0 bytes in 0 blocks
==30087==      possibly lost: 0 bytes in 0 blocks
==30087==    still reachable: 8 bytes in 1 blocks
==30087==         suppressed: 0 bytes in 0 blocks
==30087== Reachable blocks (those to which a pointer was found) are not shown.
==30087== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==30087==
==30087== For lists of detected and suppressed errors, rerun with: -s
==30087== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Memcheck classifies memory leaks in 4 general types:

-**Still reachable:** Not especially relevant.

-**Definitely lost:** Symptom of problems. This is a memory leak.

-**Indirectly lost:** Problems usually caused by a previous definitely lost.

-**Possibly lost:** Inner pointers to allocated mem block.

At first, I just had a memory allocation problem (in still reachable) with aref and bref that I wasn't using, I was allocating memory for both but I only released memory for one. I put both in comments and the problems were solved.

```
-bash-4.2$ valgrind --leak-check=yes ./dgesv 512
==29351== Memcheck, a memory error detector
==29351== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==29351== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==29351== Command: ./dgesv 512
==29351==
Time taken by my implementation: 66.19s
==29351==
==29351== HEAP SUMMARY:
==29351==     in use at exit: 0 bytes in 0 blocks
==29351==   total heap usage: 10 allocs, 10 frees, 18,875,392 bytes allocated
==29351==
==29351== All heap blocks were freed -- no leaks are possible
==29351==
==29351== For lists of detected and suppressed errors, rerun with: -s
==29351== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Step 1. Parallelize your implementation of the routine dgesv using OpenMP. It is not required that you make a perfect parallelization, just, something parallel to work with. How did you parallelize your code? How much faster (speedup) is the parallelized code than the serial one?

First, I had the idea to use gprof to find out the weight of each of my functions. Thus, ReverseSys, ProdMat and QR were the three most consuming functions (like we can see on the picture).

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 51.76      3.82     3.82        1     3.82     3.82  ReverseSys
 31.84      6.17     2.35        2     1.18     1.18  ProdMat
 16.26      7.37     1.20        1     1.20     1.20  QR
  0.14      7.38     0.01        1     0.01     0.01  Transposed
  0.00      7.38     0.00        2     0.00     0.00  generate_matrix
  0.00      7.38     0.00        1     0.00     7.38  my_dgesv
```

So, I decided to parallel them. However, the algorithms I have built for QR and ReverseSys are too complex to be properly parallelized (loops nested with if conditions). I tried to modify the algorithms or change the way I parallelized, but I didn't get anything interesting. So, I turned to Transposed and ProdMat, which I managed to do, and which remain two important functions of the dgesv program. So, my speedup is necessarily bad because I only parallelized half of my functions. For example, on the supercomputer, for 4 threads and a size of 1024, I have an execution time of 55 seconds compared to 65 seconds in sequence. So, I get a negligible speedup of 1.2. It still allowed me to play with VTune Amplifier afterwards to find the hotspots of my code.

Step 2. Use Intel Vtune Amplifier to identify the hotspots of your parallelized code. Which are they?

I had to use VTune Amplifier locally because it didn't work on the supercomputer (not installed, among other things). Here are my results with my parallelized dgesv named dgesv_para (with 4 threads and size of 1024):

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
func@0x401c87	dgesv_para.exe	108.979s
ReverseSys	dgesv_para.exe	76.668s
QR	dgesv_para.exe	34.701s
NtWaitForMultipleObjects	ntdll.dll	0.029s
generate_matrix	dgesv_para.exe	0.020s
[Others]		0.047s

*N/A is applied to non-summable metrics.

So as expected, the non-parallelized functions are in the lead. The others don't appear here because they are very fast because they are parallelized. So, I made a good parallelization of my functions. We also see here the important hotspots in non-parallel functions.

In ReverseSys, two calculations are very time-consuming because they are embedded in many loops:

```
        // Dilatation                                                    
        stock[k*n+i]=rev[k*n+i]/rev[i*n+i];                    22.4%  49.360s
    }                                                                   
    // Outside the diagonal                                             
    if(j!=i && k!=i) {                                          0.5%   1.059s
        // Transvection composed of the previous computation            
        stock[k*n+j]=rev[k*n+j]-rev[i*n+j]*rev[k*n+i]/rev[i*n+i];  9.4%   20.657s
    }                                                                   
```

In QR, we have the same thing:

```
for (j=0;j<n;j++){                                              0.2%   0.441s
    SD += A[j*n+i] * q[j*n+k];                                  5.9%   13.024s
}                                                                       
r[k*n+i] = SD;                                                          
for (j=0;j<n;j++){                                              0.2%   0.472s
    A[j*n+i] -= r[k*n+i]*q[j*n+k];                              9.4%   20.715s
```

Step 3. Use Intel Advisor to improve the vectorization of your code. Which actions were recommended by Advisor, and in which parts of the code? How much performance gained after the vectorization was improved?

For this last step with Intel Advisor, I used auto-vectorization (on Windows) with Intel compiler to make Advisor work. The parameters and flags of the following line are mandatory for optimal Advisor operation with vectorization.

icl /debug:inline-debug-info -o dgesv /Qopenmp /MD /Qopt-report:1 /Qopt-report-phase:vec dgesv_para.c

Performances issues based on the results of Advisor:

- ➔ Potential underutilization of FMA instructions (depend just on my architecture, so it's not really a problem) in QR and ProdMat functions.
- ➔ Data type conversions present / system function call present / indirect function call present in generate_matrix.

Advisor has vectorized two loops in the ReverseSys function. The time saving is poor but enough to be annotated.

This is the results page displayed by Advisor: