30.10.2014

Yunqing Dong

Mariusz Bujny

PROGRAMMING OF SUPERCOMPUTERS – ASSIGNMENT 1

1. Sequential Optimization – GCCG Solver

1.1.

Execution environment:

The Program is compiled and executed on SuperMUC. It composes of 18 thin code islands and 1 fat island. The computation should be conducted on thin island. On thin island Sandy Bridge-EP Intel Xeon E5-2680 8C processors are used. On every node there are 2 processors and there are 8 cores on each processor. When hyperthreading is activated, there could be 32 logical CPUs on one node. On every thin nodes island there are 9216 nodes in total, which could bring 3.185 PFlops as Peak Performance. More detailed information has been filled in the excel file in attachment. The code is compiled in Fat Island and then the job is submitted to the Thin Island. The parameters are specified in the the file job.cmd as follows:

```
#@ job_type = MPICH
#@ class = test
#@ node = 1
### schedule the job to 2 to 4 islands
#@ island_count=1, 1
#@ total_tasks= 16
## other example
##@ tasks_per_node = 16
#@ wall_clock_limit = 0:30:00
```

0 h 30 min 0 secs
#@ job_name = fire
#@ network.MPI = sn_all,not_shared,us

Since we know the job will be finished very soon, we always specify the class as test. Since we are doing single core optimization and measurements, it is reasonable to run the job only on one node. In case of use of just one core, job class could be specified as micro, as well. But the corresponding waiting time is considerably longer.

Formulas:

During the measurement we used the PAPI tool. In order to measure cache miss rate for both L2 and L3 level, we use counters PAPI_L2_TCM and PAPI_L3_TCM (to measure cache misses and counters) PAPI_L2_DCA and PAPI_L3_DCA (to measure data cache accesses). The formula to calculate cache miss rate is:

cache miss rate = number of cache misses/ number of total cache accesses

FLOPS stands for **FL**oating-point **O**perations **P**er **S**econd. Mflops is $10^{\mathbf{6}}$ FLOPS.
Theoretically we can compute the FLOPS by using:

$$\mathrm{FLOPS} = \mathrm{cores} \times \mathrm{clock} \times \frac{\mathrm{FLOPs}}{\mathrm{cycle}}$$

In our case, we use counters PAPI_SP_OPS and PAPI_DP_OPS to calculate the scaled single precision and double precision vector operations. Here are the definitions of these two counters:

PAPI_SP_OPS = FP_COMP_OPS_EXE:SSE_FP_SCALAR_SINGLE +
4(FP_COMP_OPS_EXE:SSE_PACKED_SINGLE) + 8*(SIMD_FP_256:PACKED_SINGLE)*

PAPI_DP_OPS = FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE +
2(FP_COMP_OPS_EXE:SSE_FP_PACKED_DOUBLE) +*
4(SIMD_FP_256:PACKED_DOUBLE)*

By summing up these two counters and dividing by the processing time, we get FLOPS:

FLOPS = (*PAPI_SP_OPS* + *PAPI_DP_OPS)* / process time

In case of the analyzed program, due to lack of single precision floating point operations, the above mentioned formula can be simplified to:

FLOPS = *PAPI_DP_OPS* / process time

We assume that we can treat both single and double precision operations as equivalent floating point operations.

1.2.1.

In this part we compare performance of the program for two input files: tjunc.dat and cojack.dat using -O1 and -O3 optimization flags. In order to obtain reliable results, we perform 3 runs for each case (what we specify explicitly in the job script) and avarage the results.

First of all, in both cases, real as well as process time is reduced by approximately 14% (from 5.0 to 4.3 seconds) when using -O3 (compared to -O1). This is the most important measure of the overall performance.

If we look at the Mflops, they are increasing by around 18% (from 1570 to 1852 Mflops) in case of -O3, as well. This could be explained by much better vectorization (more details in section 1.3). Namely, in the same amount of time, we are able to perform 2 times more (SSE) or 4 times more (AVX) double precision floating point operations. The extent to which the SIMD operations are used could be checked explicitly by using PAPI to count either just scalar or vector operations (PAPI_FP_INS , PAPI_VEC_DP, etc.). This method confirmed our hypotesis – there were considerably more vector operations in case of the use of -O3 flag. Also a decomposition of the program to an assembler code (objdump -d BINARY) showed more frequent use of xmm (SSE) registers. However, we did not notice use of any ymm (AVX) registers both in -O1 and -O3 case.

When it comes to L2 and L3 cache miss rates, they increase in case of -O3 optimization level. In principle, it should give worse performance. Nevertheless, thanks to better vectorization (what might be also the reason why we observe higher cache miss rates in case of -O3 flag, since bigger chunks of memory have to be loaded) the overall performance increases.

1.3.

Effect of the vectorization on the computation phase:

The use of the Intel SSE instruction set, which is an extension to the x86 architecture, is called vectorization. Vectorization try to utilize the full length of register in order to process more data for the same instruction in one cycle. For the loops or similar structures it will reduce the execution time. By default SSE2 supports the 128-bit SIMD floating-point register. If we turn on the compiler flag -O2 or -O3, compiler will also try to vectorize the code with SSE2 instructions.

In Sandy Bridge Processors we have 256-bit SIMD floating-point registers, and it supports AVX (Advanced Vector Extensions). They could proceed even more data for the same instruction at one cycle. By specifying -no-vec and -xhost together under Linux we tell compiler explicitly that we are now using AVX. In case just -xhost both SSE and AVX can be used.

By turning on the -no-vec flag we enforce the compiler not to use SSE vector instructions. By executing cojack.dat we need 11.7 seconds processing time. While using -vec and -vec-report we need only 3.6s. By introducing -xhost we need 3.8s as processing time. In order to understand the behavior behind, we need to check how the file was complied. By using **objdump -d** we can check which line is vectorized. Actually there is one interesting difference between using -xhost and -vec. In the function compute_solution the loop in line 59, by -vec it is not vectorized because it seems inefficient for the compiler. While by using -xhost it enforces it to vectorize.

```
for ( nc = nintci; nc <= nintcf; nc++ ) {
        direc2[nc] = bp[nc] * direc1[nc] - bs[nc] * direc1[lcc[nc][0]]
                - be[nc] * direc1[lcc[nc][1]] - bn[nc] * direc1[lcc[nc][2]]
                - bw[nc] * direc1[lcc[nc][3]] - bl[nc] * direc1[lcc[nc][4]]
                - bh[nc] * direc1[lcc[nc][5]];
    }
```

The computation here is very complicated and it requires a lot of data to calculate. By pre-fetching data it might waste resources. And the L3 cache miss rate explains it. For -vec the L3 cache miss rate is about 10.4% while by -xhost is 16.3%.

By vectorization we can utilize the extra length of floating-pointer register to do more calculations

in one cycle. This is the reason why the Mflops could raise more than 3 times by vectorization compared to SISD.

1.2.2.

It turned out that the highest Mflops value was achieved for -O3 optimization flag. The average value of Mflops in this case reached 1852 Mflops. Theoretical peak performance of a single core on SuperMUC thin island is 2.7 Gflops. Therefore, we conclude that a single core optimization with the use of different icc compiler flags can result even in 69% use of peak performance in the given program. This shows that the program is well-optimized for serial execution.

2. I/O Performance

The goal of the second exercie was to write a small tool to convert the input text files to the binary format. The GCCG source code had to be adopted accordingly, to be able to handle both text and binary input files.

2.1.

The „binconv" tool was partially based on the code provided for the lab course. We used the part of the code from the „util_read_files.c" file to load the data from the input text file into aproppriate data structures. The second part of the tool opens a binary file and saves the data structures with the use of the function from the "stdio" library:

*size_t fwrite (const void * ptr, size_t size, size_t count, FILE * stream);*

Where:

ptr

Pointer to the array of elements to be written, converted to a const void.*

size

Size in bytes of each element to be written.

count

Number of elements, each one with a size of size bytes.

stream

Pointer to a FILE object that specifies an output stream.

With a single call of this function we save the 1D arrays: su, bp, bh, bl, bw, bn, be, bs. The 2D array lcc is saved by calling the function multiple times in a loop – as a result we save many 1D arrays that could be used to recover the 2D array in the same way as the dynamic 2D array is allocated.

2.2.

The GCCG has been adopted to read the new binary files by use of the fread function:

*size_t fread (void * ptr, size_t size, size_t count, FILE * stream);*

Where the arguments are defined as above. This way binary files are read into the data structures mentioned above.

The correctness of the conversion has been tested with the xxd linux tool (displaying the content in binary format) as well as by writing and reading the binary files.

2.3.

ASCII files can be viewed as special binary files, where each character is coded as a single byte. As a result, for instance number „100” is coded with the use of 3 bytes in ASCII format and 1 byte (01100100) in binary format. Therefore, in many cases much more storage is needed to code a text file. This was also the case of the provided input files. As shows the table below, writing the same data into a binary files might result even in 65% reduction of the file size.

	cojack	**pent**	**tjunc**
size_text [B]	75058440	30636050	2882967
size_bin [B]	30666672	10587472	1552200
size_bin / size_text	**40,86%**	**34,56%**	**53,84%**

Below reading times for text and binary files is presented:

	T(text)	T(bin)	T(bin) / T(text)
cojack.dat			
1	1,61662	0,04061	
2	1,61900	0,04171	
3	1,61798	0,04028	
average	**1,61787**	**0,04086**	**2,53%**
tjunc.dat			
1	0,07461	0,00267	
2	0,07489	0,00243	
3	0,07433	0,00239	
average	**0,07461**	**0,00249**	**3,34%**

Since the size of data to be read is considerably smaller and the fread function is able to read a whole array at once, the program reads the input files even 30-40 times faster. Such result is reasonable – unlike in case of text input files, where we have to read word by word and deal with data conversions, binary files allow to copy at once the whole data structure directly to the memory. What is more, the results shown above could be even improved if some changes to the data structures were made – for instance by ensuring a contiguous allocation of the lcc array (first allocate a 1D array, and then an array of pointers pointing to appropriate elements).

3. Visualisation using ParaView

In order to make the program capable of saving VTK files, we made use of the functions vol2mesh and write_result_vtk. The arguments of the functions calling them had to be extended accordingly. Finally, the program is saving 3 VTK files (plots of VAR, CGUP and SU) at each call.

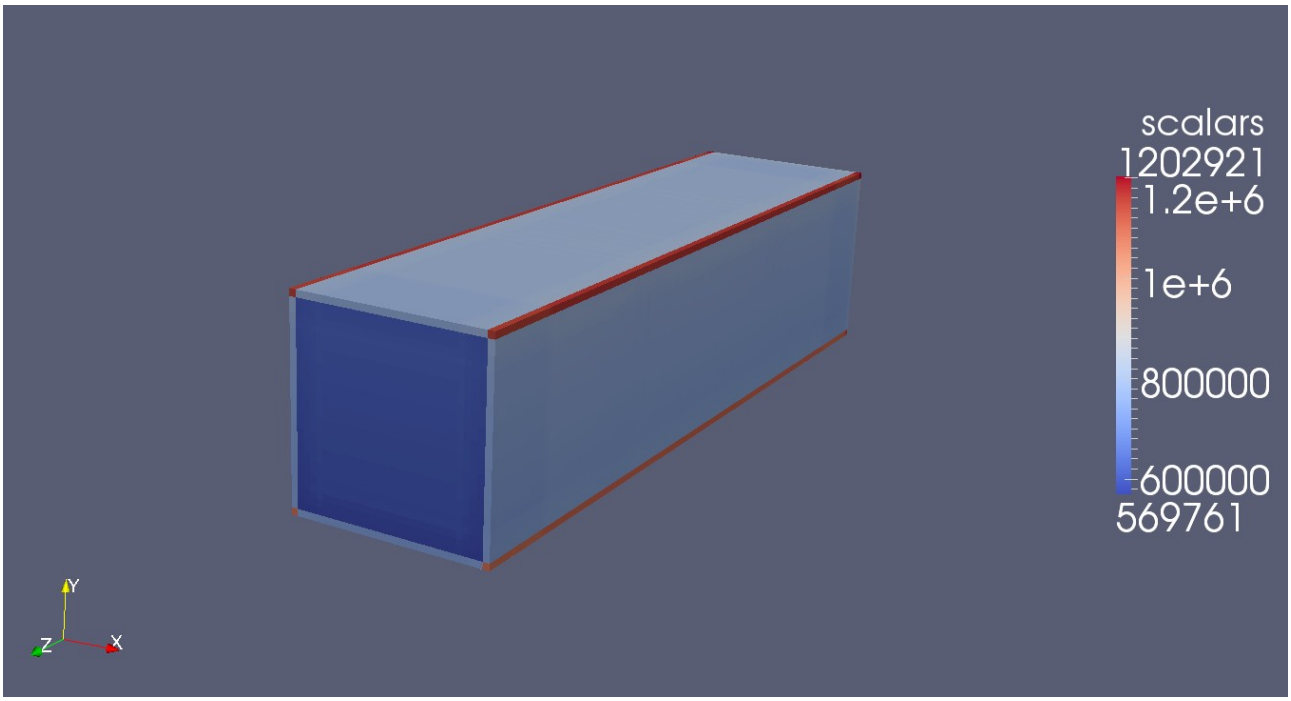

Figure 1. Values of the CGUP field for the pent.dat input file.