UNIVERSITY OF THESSALY

Performance evaluation using TeaLeaf mini app

Marios Angelis , Christos-Konstantinos Matzoros

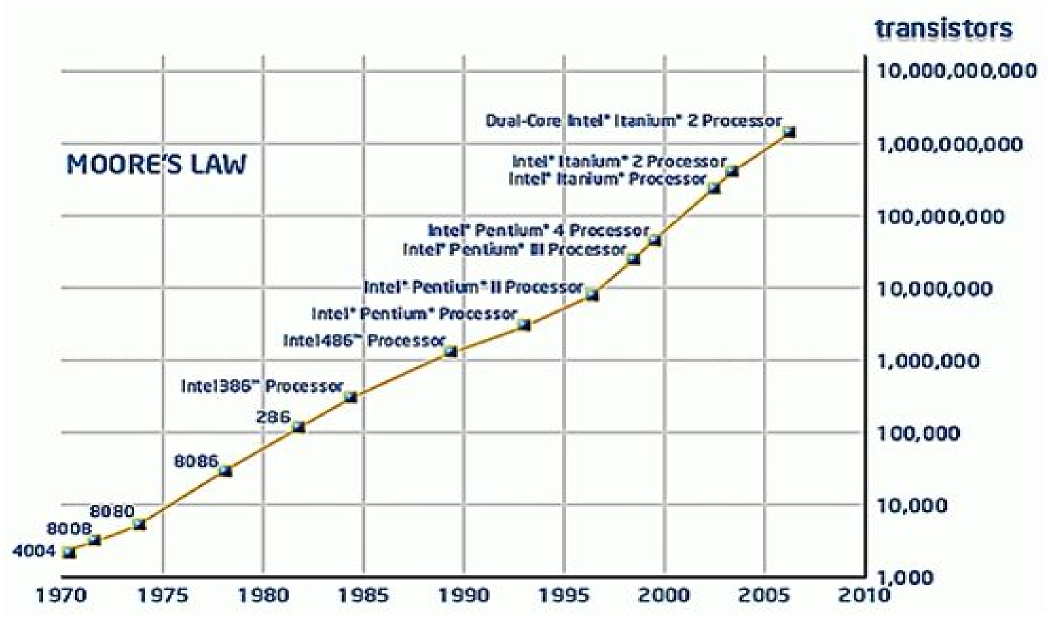
(mangelis@uth.gr) (cmatzoros@uth.gr)

01/08/2018 - 31/09/2018

Supervisor Professor : Christos D. Antonopoulos

INTRODUCTION

In 1965, Gordon Moore formulated what became known as Moore's law: "The number of transistors per silicon chip doubles each year". In 1975, as the rate of growth began to slow, Moore revised his time frame to two years. Moore was right, but after a lot of years, a problem begins to appear. The problem is power consumption. As we put in a chip more and more transistors, growth in power is unsustainable.



The solution to this problem is multicore systems which use less power. Using additional cores and decreasing the frequency may have a big impact on the power consumption of our system. The power consumption of a system is given by the simplified formula: $Power = Capacitance \cdot Voltage^2 \cdot Frequency$. The performance is being measured by the simplified formula: $Performance = Cores \cdot Frequency$. Provided that the reduction of the frequency allow us to achieve almost the same reduction in voltage (DVFS), decreasing by half the frequency and doubling the number of cores will have a quadratic reduction on power consumption. Using parallelism, cpu executes the computations of an application simultaneously on different cores, in order to complete a problem in less time.
An important performance metric in parallel computing is speedup. Speedup measures how much faster is a parallel execution of a given problem versus the best sequential execution of this particular problem. Speedup is equal to the sequential time divided by parallel time, known as the Amdahl's law: $Speedup = Sequential\ Time / Parallel\ Time$. If the number of processors tends to infinity, the maximum speedup tends to be: $Speedup = 1/(1-P)$, where P is the proportion of a system or program that can be parallelized. Speedup is limited by the total time needed for the serial part of the program. However, some of the assumptions of the law are 1) that the size of the problem is fixed, 2) that there are no discontinuities in the processor performance curve, 3) that the only resource that is varied is the number of processors 4) and that parallel overhead tends to zero.
These don't really apply to the today's problems and parallelization models. Is there any a solution to this problem? Gustafson-Barsis' Law posits that adding processors allows an application to solve bigger problems in the same time. If we scale up the problem size as processors increases, the serial fraction is being shrunk to the minimum. Theoretically, It is shown that if the number of processors tends to infinity the speedup tends to infinity too. In practice the big overhead from synchronization and communication of the threads causes a significant decrease in performance. Furthermore, the programmers must be aware of problems in parallel applications such as load imbalance on processors and false sharing on caches, that can affect the overall performance.

OpenMP

OpenMP is a language for writing parallel (multithreaded) applications. The OpenMP Architecture Review Board (ARB) published its first API specifications, OpenMP for Fortran 1.0, in October 1997. In October the following year they released the C/C++ standard. The main instruction construct of openmp is : #pragma omp parallel [construct]. The pragma omp parallel is used to fork additional threads to carry out the work enclosed in the construct in parallel. The original thread will be denoted as master thread with thread ID 0. The main advantage of

OpenMp is that there is no need of dealing with message passing as MPI does. Also with simple usage of <omp.h> library, code can be very efficient. On the other side, there are a lot of problems such as race conditions, synchronization bugs or false sharing.

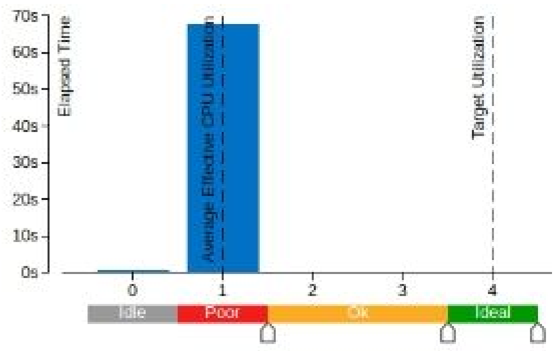
TeaLeaf

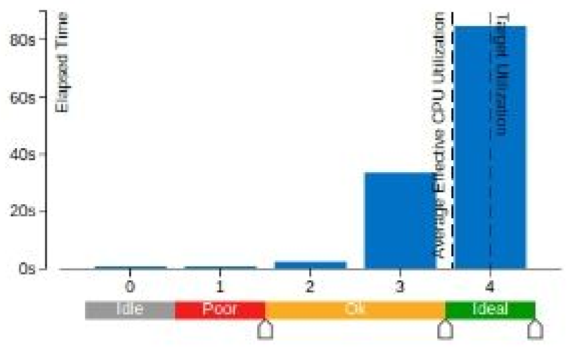
TeaLeaf is a mini-app that solves the linear heat conduction equation on a spatially decomposed regularly grid with implicit solvers. TeaLeaf solves the equations in three dimensions. The computation in TeaLeaf has been broken down into "kernels", low level building blocks with minimal complexity allowing maximum optimisation by the compiler. Solvers such as the Jacobi method(default method) are used to invert the system of linear equations, because of the complexity of advanced methods. In addition there is an option to use a Conjugate Gradient or a Chebyshev solver. The advantage of these is that they are matrix free and independent of any library dependencies.
Tealeaf-master-dist/3d is the main directory. Inside it, there are 3 directories (c_kernels, drivers and build). Each time a user types the instruction “make”, inside the “build” directory there are created files with suffix “.o”. Also, inside the c_kernels directory, there are the serial code directory and the “omp” directory which contains the parallel code. Before compiling the program with icc compiler, user must connect the location of directory “TeaLeaf-master-dist" with the intel compilers directory simply by typing “source ~/intel/compilers_and_libraries_2018/linux/bin/compiler/vars.sh intel64”[1]. Then, by typing “make” an executable file called tealeaf is being created. To run tealeaf, type “./tealeaf”. To remove old executable, type “make clean”. In the main directory, there is a file called tea.in. From tea.in, a user can set the size of problem, the solver method (cg, Cheby, Jacobi , ppcg) and other metrics such as geometry, time, iterations, energy and density settings. Also, user can compile the program with gcc, icc or mpicc compiler by selecting one of three options inside the Makefile. If compiler option is gcc compiler, delete “#” from line 10 to activate “–lm” suffix. In addition, compile the application with the ”-g” suffix in order to generate information for debugging, that allows Vtune to associate timing information with specific locations in our source code.

Intel Vtune Amplifier

Vtune is a performance profiling tool for Fortran, C and C++. It selects data about the amount of concurrency and identifies possible bottlenecks by synchronization primitives. VTune Amplifier uses the hardware event-based sampling collector to collect data for many types of analysis such as “Hotspots”, “Advanced Hotspots”, “Memory Access”, “HPC Performance Characterization” and several other. The major analysis that we used is basic hotspot analysis. It identifies

functions taking the most time to be executed and provides information about the thread activity as well as the memory consumption that takes place during runtime. After analysis is completed there are many options of viewing and interpret the results.
To run VTune Amplifier, user must execute the following instructions: 1) Follow upper instructions[1] to connect compilers_and_libraries with the location of the TeaLeaf directory and from the same terminal go to intel/vtune_amplifier_2018 directory. Type “source amplxe-vars.sh” and run amplifier by typing “amplxe-gui”. Create a new project and connect it with the tealeaf executable by Application textfield. Also give the path to file tea.in in the Application Parameters textfield. In each of the diagrams below, we present a poor serial performance and a better parallel performance for the same application.





Padding

After analyzing concurrency analysis results, we found that TeaLeaf-application did not have false sharing data problems.However,we tried to put padding technique to our code.First of all,we added an extra dimension to element vec_w inside TeaLeaf-master-dist/3d/chunk.h file.So,we transformed the old statement: double * vec_w to new statement : double ** vec_w. After that,we added a new function in file TeaLeaf-master-dist/3d/c_kernels/kernel_init_chunk.c In this function,we allocate dynamically memory for a double ** as follows :
a = (double**)malloc(x*y*z*sizeof(double *)),so a can be considered as an array of pointers to double with length equals to x*y*z.We allocate memory for each pointer to double inside a for loop statement as follows:

```
for(int ii=0;ii<x*y*z;ii++){
   a[ii]=(double *)malloc(sizeof(double)*PAD);
}
```

Each line of the array is a pointer to double with length equals to PAD.In our system,cache line is 64 bytes ,so each line of the array is equal to a cache line,only if PAD=8(because PAD*sizeof(double)=64).We called this function only to initialize vec_w and in each solver method,we used vac_w as a 2D array.For the method to be applied correctly,we used only the first element of each line.So, if the program instruction before padding was :
vec_w[index]=SMVP() , after using padding method the same instruction becomes :
vec_w[index][0]=SMVP(),emphasizing that the second dimension is always 0.

Parallelism methods:

We used a dual cpu system, with 14 cores per cpu, 2-way hyperthreaded, with 56 threads giving maximum performance .First of all, we made changes to directory c_kernel's files. Inside each file, there is a defined variable called NUM_THREADS which symbolizes the total number of threads used to parallelize each for loop. Before each loop, we put the OpenMP instruction "#pragma omp parallel for set_threads(NUM_THREADS)". This is called SPMD method. Instead of giving to the master thread all the amount of the calculations inside the loop, the master thread creates some threads(the number of created threads is NUM_THREADS) and the cpu gives them work to do. Exactly, these statements are same:

Statement1:

```
#pragma omp parallel {            /*codeblock to parallelize*/
        int id=omp_get_thread_num(), threads=omp_get_num_threads();
        for(int ii=x+id; ii<=1000; ii+=threads){
                ...
        }
}
```

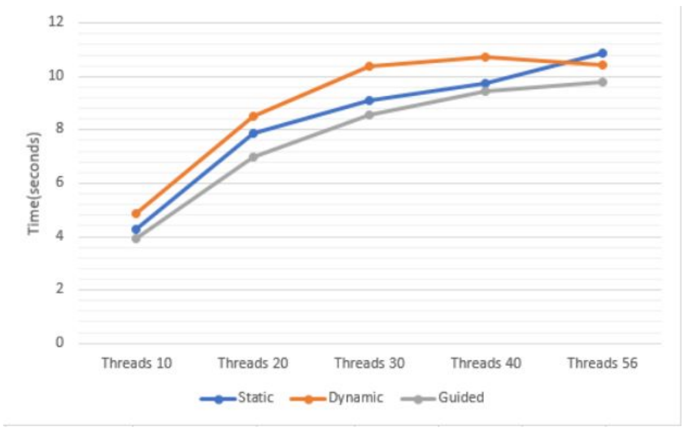
Statement 2:

```
#pragma omp parallel for        /*for loop  to parallelize*/
for(int ii=x; ii<=1000; ii++){
                ...
}
```

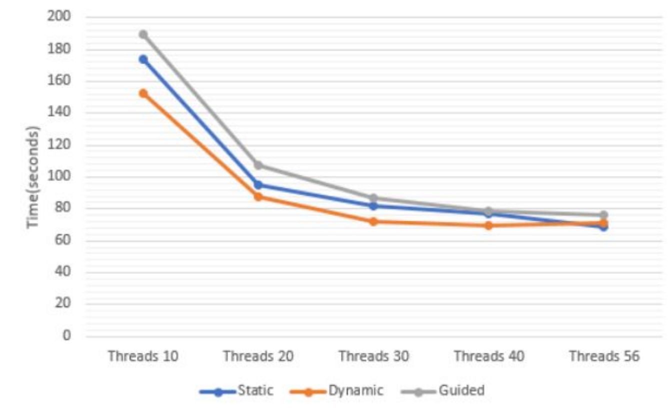
Assume that our system has 4 threads and we have a loop with 5 iterations. Thread with id=0, will make the calculations about iterations: 0 and 4, thread with id=1 will make the calculations about iterations: 1 and 5, thread with id=2 will work on second iteration and thread with id=3, will make calculations about the third iteration. All threads work inside the same loop, and make the same amount of calculations in less time. Inside TeaLeaf application solver's files, there are a lot of for loops with total amount of iterations close to 25*10^9 each. For nested loops, we put the #pragma instruction only in the first for-loop. Also, inside the codeblock of a for-loop, each iteration calculates an index and writes inside an array at the position index. (buffer[index]=...). There is no conflict problem because all threads write to the same array but in different positions. Secondly, each variable which is declared inside the parallel for loop, is a local variable and a variable which is declared outside of the parallel loop, is a global variable.Only one thread has

access to his local variable ,but all threads can change a global variable. So what will happen when all threads increase a global variable each time +1 and after the loop statement we print this variable? Each time the result will be different. This is a typical example of a critical section that needs to be executed without interference.The first problem was that each time the program was raising a global variable visible to all threads inside the loop, at the end of the loop statement, the result was different and of course it was wrong. So, to solve this problem, we used reduction construct, by adding the construct in the same instruction "#pragma omp parallel for reduction(+:pw_temp)". Reduction statement, creates a local variable with the same name for each thread , which is visible only by this thread. So, each thread makes changes to his local variable and at the end of the loop, compiler undertakes to collect the result and put it in a global variable with the same name.
As you can see at the metrics below,we tested 3 different types of **scheduling,**static ,dynamic and guided.For the **Static** method,compiler divides the loop into equal-sized chunks (if chunk is not declared).Exactly, chunk=iteratios_number/number_of_threads.For example ,if we have a for loop with 500 iteration and a system with 4 threads,chunk will be 500/4=125 .So,first thread will run the for loop iterations from 0 to 125,second thread will execute iterations from 126 to 250,etc.Static scheduling is very important on NUMA systems like our system: if you touch some memory in the first loop, it will reside on the NUMA node where the executing thread was. Then in the second loop the same thread could access the same memory location faster since it will reside on the same NUMA node,so the problem of false sharing is eliminated .**Dynamic** scheduling uses a queue .When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue.However,dynamic scheduling maybe has overhead,because after finishing,each thread must stop and receive a new value of the loop iteration variable to to use for the next iteration.Generally,increasing the chunk makes the scheduling more static and decreasing it makes the scheduling more dynamic.Also,if all iterations have the same amount of work,static scheduling is better ,but when each iteration has different amount of work to execute,splitting across threads by dynamic scheduling is faster.
Guided scheduling method is similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. By default the chunk size is approximately loop_count/number_of_threads.In the table below,you can see that static method is more efficient when running TeaLeaf application .As we decrease the amount of threads,the program is getting all over and slower.

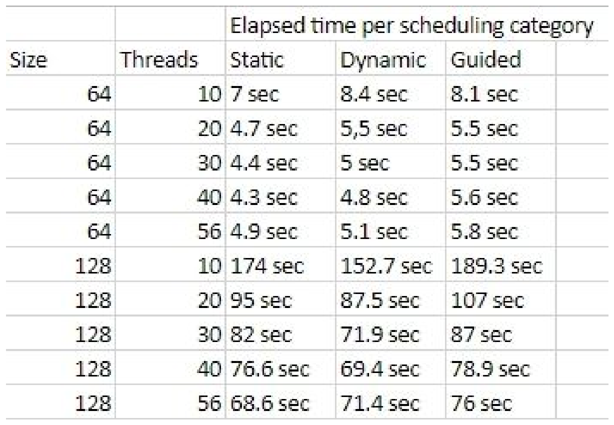
First diagram: It shows the speedup for a tealeaf problem with size 128 as we increase the total number of threads .



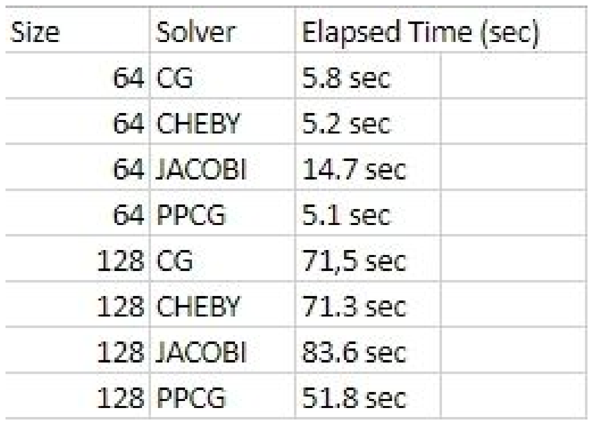
Second diagram: It shows the elapsed time for a tealeaf problem with size 128 as we increase the total number of threads.



Time table: It shows the elapsed time for all problem sizes threads and solver method combinations.



		Elapsed time per scheduling category		
Size	Threads	Static	Dynamic	Guided
64	10	7 sec	8.4 sec	8.1 sec
64	20	4.7 sec	5,5 sec	5.5 sec
64	30	4.4 sec	5 sec	5.5 sec
64	40	4.3 sec	4.8 sec	5.6 sec
64	56	4.9 sec	5.1 sec	5.8 sec
128	10	174 sec	152.7 sec	189.3 sec
128	20	95 sec	87.5 sec	107 sec
128	30	82 sec	71.9 sec	87 sec
128	40	76.6 sec	69.4 sec	78.9 sec
128	56	68.6 sec	71.4 sec	76 sec



Size	Solver	Elapsed Time (sec)
64	CG	5.8 sec
64	CHEBY	5.2 sec
64	JACOBI	14.7 sec
64	PPCG	5.1 sec
128	CG	71,5 sec
128	CHEBY	71.3 sec
128	JACOBI	83.6 sec
128	PPCG	51.8 sec

REFERENCES:

[1]Introduction to openMP by Tim Mattson
(https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf)

[2]OpenMp tutorials by intel
(https://www.youtube.com/watch?v=nE-xN4Bf8XI)

[3]Intel Vtune Amplifier description by Lawrence Livermore National Lab site:
(https://hpc.llnl.gov/software/development-environment-software/intel-vtune-amplifier)

[4]Uses and abuses of Amdahl's law
S. Krishnaprasad,Mathematical, Computing, and Information Sciences
Jacksonville State University:
http://www.di-srv.unisa.it/~vitsca/SC-2011/DesignPrinciplesMulticoreProcessors/Krishnaprasad2001.pdf

[5]Lectures from HPC(HY421) course https://courses.e-ce.uth.gr/CE421/lectures.php