Full Name: Oliver Goldstein **Username**: OG14775

1) Introduction

The physical constraints of further increasing clock speeds has created an industry wide shift to multiple core parallel computing. Empirical and theoretical observations have been created in order to describe the implied increase of computation throughput as a result. One of these is called Amdahl's law, which gives a theoretical upper limit on computational speedup given the fraction of parallel work and the processor count. Fig 1.1 displays the speedup when 90% of the code is parallel and Fig 1.2 displays the speedup when 99.9% is parallel (x axis is processor count and y axis is factor speedup). This disproves the stigma that parallel computing is a simple solution to all intractable computational problems. Even with a code base which is 90% parallel, even with many thousands of cores, the speedup tends to a mathematical limit of a 10x improvement. With 100% parallel code the relative speedup would be linear with respect to the number of cores. Fig 1.2 shows the massive improvement of around 100 times Fig 1.1 with just an additional 9.9% parallel code. In addition to this Gunther's law quantifies the retrograde performance exhibited by code, including parameters not accounted for with Amdahl's Law, such as coherency delay and queuing contention. Gunther's law refutes the idea that a higher core count maps to a monotonically increasing function and actually describes how, after a certain point, performance levels off and decreases due to communication and coordination delays, especially when multiple threads need to share access to data structures. Note than when beta is 0 in Fig 1.0 the law is equivalent to Amdahl's law. Note that alpha represents the coherency delay. As we progress into the future and the relative gap between memory access speeds, communication between processors and computing power potentially grow, a great deal of computational power will be wasted unless it can be optimised for. I will test the code on the 128*256 grid initially as the time it took to test was far far shorter than for the 256*256 grid.

Fig 1.0 (Gunther's Law)

$$C(N) = \frac{N}{1 + \alpha((N-1) + \beta N(N-1))}$$

Fig 1.2 (Amdahls Law: 99.9%)

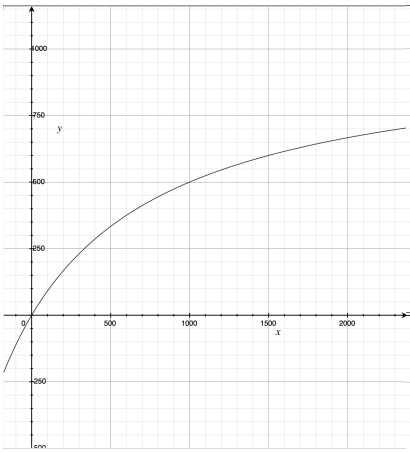


Fig 1.1 (Amdahls Law: 90%)

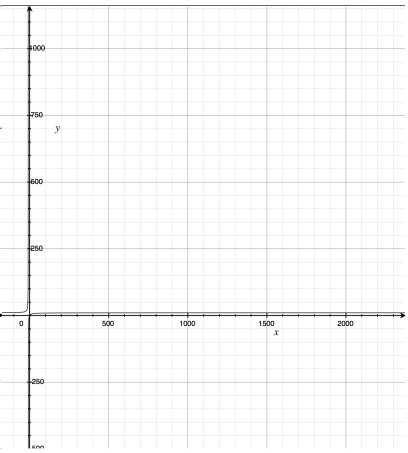


Table 1.0 (Initial Code Base) (128*256)

Function Name	Percentage Time (%)	Time (Seconds)	ms/call
collision	76.89	163.94	4.10
av_velocity	12.07	25.75	0.64
propagate	10.38	22.13	0.55
initialise	0.00	0.00	0.00
Total	99.34	211.82	5.29

2) Serial Code Optimisations

In light of the analysis above, producing serial code that can be parallelised in an efficient manner is crucial. Here, I will examine code efficiency with respect to the cache coherent NUMA architecture more specifically, the memory hierarchy, identify code that can be pipelined, and importantly, I will examine larger scale optimisations that the compiler would be unable to perform without appropriate knowledge. In light of Donald Knuth's philosophy I will focus on the larger optimisations which represent the bulk of the time spent in execution. I will profile the code initially with GPROF. The initial code base provided, compiled with **GCC 4.8.4**, has been profiled and can be seen in Table 1.0. I will spend the majority of time optimising the 3 functions collision, av_velocity and propagate as they constitute the clear majority (99.34%) of execution time. Initialise was placed there to demonstrate how irrelevant some functions were in terms of execution time. I considered pipelining functions that had dependencies, but decided against this after realising the data transfer cost between cores, later on during parallelisation, may potentially outweigh any performance gains. Generally, throughout the report I do not refer to specific functions, but more to specific optimisations as the procedure of loop fusion blurs the lines between what one would constitute the function. I also pay little attention to functions that do not constitute much of the code base. For consistency, all serial optimisations here were tested on the 128 * 256 grid size. The following optimisations are in chronological order:

Table 1.1 (Fusion Code Base) (128*256)

Total Time (s)	Compiler	Flags
196.01	GCC 4.8.4	-O3
181.79	GCC 6.1	-O3
185.39	ICC 6 U2	-O3

2.1) Optimisation One: Caching behaviour of collision function:
Currently each iteration of the nested for loop is larger than the cache size. Blue Crystal Phase 3 has a L1 cache size of 64 bytes. I initially aimed to optimise for this caching behaviour. After performing loop **fission** and observing no time difference, I realised that this strategy within functions was ineffective when trying to optimise the L1 cache, I also quadrupled the number of cache misses by having multiple loops over the same array which I used PAPI to confirm. I decided to try loop **fusion** instead. Upon performing fusion of all tilmestep functions including average velocity - exploiting the cache locality - code execution time decreased by 16 seconds (Table 1.1) with a ms/call of 4.91. From now on, I will refer only to loop_fusion.

2.2) Attempted Optimisation: Testing for the presence of obstacles in a data array, a task which branch prediction is not effective at, disrupts the compilers ability to vectorise instructions at the loop level. In addition the branch causes complication in temporal and spatial locality causing processor pipelines to flush each loop iteration. I removed the branch by computing everything no matter whether it was an obstacle or not. This assumed computation time was a less than the cost of a cache miss. I did this by, for instance, using the obstacle value multiplied by value A added to one minus the obstacle value multiplied by value B which would automatically choose relevant values dependent on the presence of an obstacle (1 or 0), thus removing the need for ternary operators/branches. I got the cells to choose neighbouring directional values (east, west etc.) oppositely if an obstacle was present there, before acting on the obstacles in a separate loop at the end. After completing this the code ran an extra 30 seconds slower at 210.30 seconds on O3 optimisation. I deduced that the computation time represents a longer time than the cost of a cache miss. An alternative method that I would have tried given more time is to

provide each data cell with a bit array representing the position of any neighbours if it had any. I would have also preprocessed the data array and separated the data from the obstacles. I tried a similar method to this, by emulating the computational cost of a for loop only over the data array and looping over the obstacles separately just flipping values. This took longer than the inclusion of the branch bringing the time to a total of 206.40 seconds on 128*256 with GCC 6.1. Later in this report, I discuss the problem of this branch again.

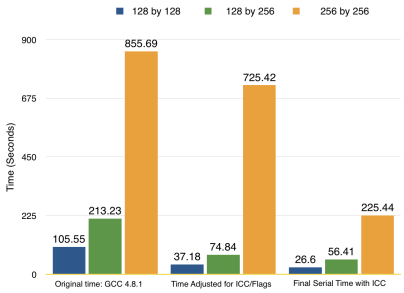
2.3) Minor Optimisation: I played around with different compilers in order to find the best compiler and compiler flags. I attempted to write a script to brute force the different options and find the best solution, however, it took intractably long to measure all of the different options. I noticed whilst searching that gcc 6.1 shaves an extra 15 seconds off execution time of 196 seconds. I saw more relevance in optimising compiler versions and flags at the end of the project, in case of tradeoffs beyond my comprehension.

2.4) Optimisation Two: Converting divide instructions to multiply instructions.
Divide instructions can often be far more complicated and expensive than multiply instructions, although they can be more accurate. I decided to factor them out. This was done by looking for common repeated divisions and then using simple algebraic manipulations to compute them once only. I changed divisions which used local_density, refactoring the d_equ array and removing the array named u. This may have filled the level one cache, which has a size of 64 bytes, and seeing as it just contained simple arithmetic expressions relating to u_x and u_y, could be removed. This optimisation alone reduced the time by over 100 seconds to 81.70 seconds for the 128 by 128 grid using GCC 6.1.

2.5) Optimisation with no net effect: I removed ternary operators which chose relevant cell locations for the southern and western cells in order that the processor pipelines would not flush and potentially compromising cache contents. I performed loop peeling; separating loop iterations where ii and jj were equal to zero and where ii and jj were known in advance. This dramatically increased the code size associated with the loop_fusion function. Upon testing, no speedup occurred. I deduce that perhaps I do not fully understand how GCC optimises the code or one can deduce that perhaps the cache misses were insignificant.

2.6) Minor Optimisation: Optimisation is an NP hard problem, implying empirical analysis is sometimes the best one can do when searching for solutions. I discovered a tool for HPC clusters called the Periscope Tuning Framework which uses machine learning techniques similar to MILEPOST in order to tune compiler flags. Running it on my system, it recommended the flags -O3 -ffast-math -funroll-loops -Ofast, which successfully took 6 seconds off the total run time to 69.8 seconds when applied on BCP3 on the 128*128 grid with GCC 6.1.

Fig 2.1 (Serial Times So Far)



2.7) Optimisation Three: Vectorisation of loop operations.
I used icc's vector report to analyse which loops in my code were vectorised. In addition, I added the xHost flag which allows intel to test and use, the maximum level of instruction parallelism on the processor. On BlueCrystal each core is capable of doing 4 double precision binary operations at once. I compiled and executed the code base with icc v6 update 2 with the flag -vec-report3 (which reasons why vectorisation has not occurred). Inspecting the .optrpt file, I found that loop_fission - the most expensive function - nothing was being vectorised because of vector dependence, as well as this some loops were declared as too inefficient to vectorise with a potential speedup less than one. I removed dependencies by doing some loop fission at a minor time cost and removed if branches within the main for loop, which caused large masked strides, even if computation was being "wasted", **assuming** there are few obstacles in the given grid. Out of the three for loops here, I only managed to efficiently vectorise the main loop with an icc quoting a speedup of 2.560 times. The lack of vectorisation over other loops was due to the usage of an array of structures rather than a structure of arrays, causing loop level vectorisation to have disjoint rather than adjacent memory terms, resulting in inefficient memory access with many more load and stores with respect to the manipulation of vector registers.

2.8) Conclusion: Fig 2.1 shows the original time of each grid on the left, the reduction in speed with the flags recommended by Periscope in the middle and then the combined serial optimisations (thus far) with flags on the right compiled with ICC. We can see reductions because of the serial optimisations of a third for the two smaller grids and almost 350% in the case of the largest grid. I did not choose to preprocess the grid as the obstacles represented a very small proportion of the given grids and I managed to factor out the if branch effectively removing the cost of the obstacles existence. From this point onwards, **the only compiler I use** - as it is consistently faster for me - is **ICC**.

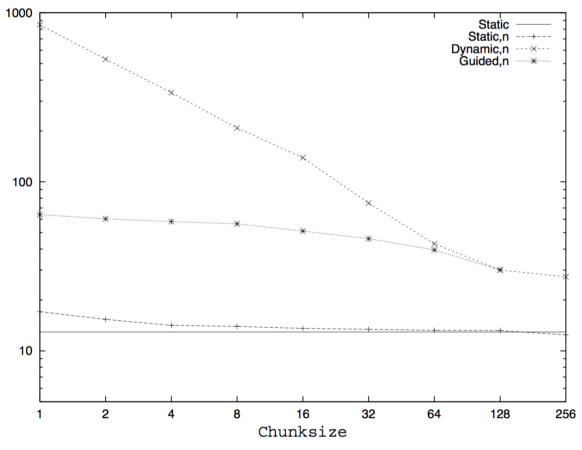
Table 3.1 (Simple Parallel Analysis ICC)

Grid Size	Initial Parallel Time	Reduction Factor (16 Cores) (1dp)
128 by 128	2.67 Seconds	9.7
128 by 256	4.51 Seconds	12.4
256 by 256	19.1 Seconds	11.8

3) Parallel Optimisations

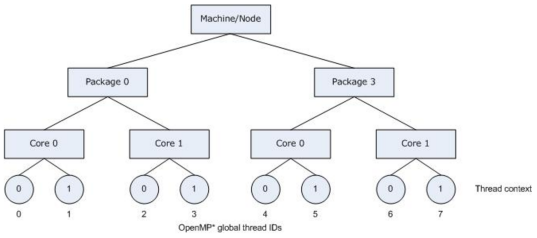
Initially, I added a pragma for, for each loop within loop_fusion, with reduction clauses over addition, for variables incremented over iterations such as tot_u and tot_cells. I also added the sims instruction, such that vectorisation would still occur. This gave me a speedup of around 12 and a half times in the best case - on the middle grid - and 9.7 in the worst. This is shown in Table 3.1. I tested the dynamic & guided thread allocation mechanisms, whereby when a thread is ready for more work, it accepts the next iteration. This has the advantage of reducing idle time in the presence of load imbalance, but incurs a cost of greater thread interaction. As a result all of the grids took 15% longer to compute. One can see the overhead of a dynamic schedule in figure 3.1, where the y axis is measured in ms and the x axis in chunk size. One can observe that the overhead for a static schedule is the invariant of the chunk size. I also tested nested parallelism with respect to the loops, which took far longer to compute, possibly due to the overhead of thread creation and

Fig 3.1 Scheduling Overheads (Log Scale)



destruction. I avoided critical or atomic bottlenecks.

Fig 3.2 (Compact affinity thread placement)



At this point, I deduced that grid memory will be allocated to the core on which the master thread executes initialise. The initialise function will be accessed by one core. The layout of BCP3 consists of 16 Intel Xeon chips on two sockets, 8 cores on each. If a core on one socket initialises the grids, due to a first touch allocation policy, as soon as the parallel regions start, all other cores will try to access the memory region that has been localised around the first core in one of the sockets. The Infiniband interconnect will try to then distribute memory representing a bottleneck in this situation. In addition, when threads start and stop they may be reallocated by the operating system to different physical cores, meaning more time is wasted with respect to memory redistribution. In order to prevent this, I decided to parallelise the initialise function allocating the number of rows divided by the number of threads to each process. After this, I would use the concept of thread affinity to keep the thread in the same place, avoiding the need for memory redistribution. I will now test this theory empirically. Fig 3.2 shows how threads would be attached to cores using the compact assignment.

3.1) Optimising Thread Affinity:
I started by setting the environment variable KMP_AFFINITY to “verbose, granularity=fine,compact”. Verbose simply details that upon execution, it prints the processor-thread mapping, and fine indicates it maps each thread in a parallel construct in openMP to a core. The compact property refers to the placement of threads relative to each other and thus the distribution of memory. Compact has the property seen on the right such that the placement of a free thread context of n+1 is as close as possible to where the context of thread n was placed (see right). I also use the compiler flag opt-threads-per-core=1 which indicates there are at most 1 threads per core, allowing better scheduling decisions to be made. I will need all for loops which access all array data to be made parallel in order that the memory distribution problems do not occur. I also want the obstacles array to be split up in precisely the same way such that there is a consistent bijective mapping from the tmp_cells, cells and obstacles array to a particular core. I also noticed that physical memory allocation would occur on the first touch and not at the point of malloc, so parallelising the initialise function - where these first touches happen - would be critical.

3.2) Why static?
Fortunately openMP guarantees that a static schedule mechanism ensures a round robin execution fashion, such that if there exist two independent loops with the same iteration cardinality and these loops are executed with the same number of threads then each thread will receive exactly the same iteration ranges within the parallel regions. This is crucial for NUMA architectures as the same memory access may be faster if it is accessed later by the same node, removing the need for cross node data transferal. I also made sure that the number of iterations each thread executed, divided the number of rows of the grid meaning there was a reduced cost of a last iteration tracked. This could have introduced an unnecessary load balance problem. I anticipate that there could be the potential for false sharing to occur, so I wanted each thread to receive data in multiples of 64 bytes, the size of the cache line. I considered using array padding to get rid of false sharing, I did not get time to doing this. This phenomenon would have only occurred on the border regions, so I decided to leave optimising this until the end. I was also aware here of the total cache size was 20480KB for each core, so I made sure the grid data remained within the cache for each iteration. Currently the amount of data overall would be (256 columns * 256 rows * 9 floats per cell * 4 bytes per floats * 2 grids) + (256 columns * 256 rows 1 byte uchar for obstacles) gives 5,000,000 bytes or roughly 5 MB - well within the cache size of the shared L3 cache
which is 20MB. Each thread would receive 1/16th of that ideally, or around a quarter of a megabyte each. Empirical testing would be needed to confirm how many cache misses actually would occur in reality.

Table 1.3 (Improvement after affinity ICC)

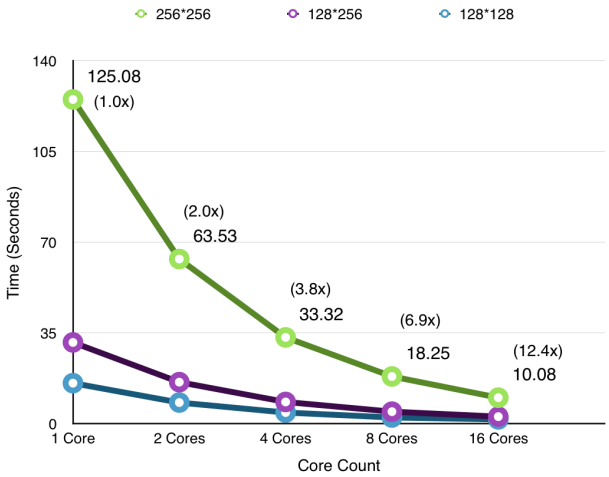
Grid Size	Current Parallel Time	Reduction Factor (16 Cores) (1dp)	Percent Change (%)
128 by 128	2.35 Seconds	11.3	-12
128 by 256	4.17 Seconds	13.5	-8
256 by 256	17.17 Seconds	13.1	-11

4) Retrospective Change

4.1) Are doubles needed? Upon further testing, I found that the conversion of double data types and data structures to float types and data structures was within the permitted error accuracy limits. On a memory bound task, this could effectively half the execution time, and could imply a two fold improvement in vectorisation. I converted the constant values to floats by adding a f to the end of the integers. I also converted the obstacles array to an array of unsigned chars, meaning that the amount of memory bandwidth they would occupy would reduce by a factor of 4. However, the obstacles, relatively did not represent a significant amount of memory and so did not affect the total time by anything worthwhile. At this point the vectorisation estimate given by intel rose by almost a factor of 3 to 5.060 factor speedup. This deducted one third of my time of execution on the largest grid through this conversion from approx 17 to approx 14 seconds.

4.2) One cell at a time. I also found that multiple store instructions to one cell turned out to be quicker than store instructions to multiple different areas. I suspect that this was due to the nature of the cache hierarchy and how it related to the memory accesses.

Figure 4.1 (Strong Scaling ICC)



Grid Size	Serial (1 core) Time (S)	1 Core (speedup)	2 Cores	4 Cores	8 Cores	16 Cores
128*128	**15.67**	1.0x	1.9x	3.61x	6.45x	9.22x
128*256	**31.38**	1.0x	1.95x	3.73x	6.71x	11.37x

4.3) Visualising it all. At this point, I was able to graph times for all 3 grid sizes and core counts of powers of 2 (see Figure 4.1) and the table beneath it plotting the relative speedups for the other grid sizes. I plotted initially a strong scaling which does not control the relative problem size, i.e. it simplifies how one can see the speedup on a fixed problem size from using multiple cores. For each core, I used a maximum of one thread, as the cost of multithreading especially with different cache elements can be quite expensive. To then account for amdahl's law, which describes how the speedup scales with respect to the proportion of parallel code against serial code i.e. , I scaled the problem size down in order to take into account the startup costs, for instance thread creation and destruction. This provides a type of analysis called a weak scaling. One can see that as the problem sizes grow larger, the fixed overheads represent a smaller proportion of the total overhead and so the relative speedup improves. This can be seen for instance with the 16 core speedup increasing from 9.22x on the smallest grid, 11.37x on the medium grid and 12.4x on the largest grid. Later on, I will test how, given a very large grid, it scales more appropriately. This heuristic fed into my decision to use a large chunk size when splitting up iterations in the static scheduler.

5) Additional attempts and further experiments

5.1) Thread pool creation frequency. In addition to this, I tested different thread pool creation frequencies within the main iteration from 3 times every iteration - in the iteration which was called from within the main function in the initial code base - to once only, within a parallel block. I observed no noticeable difference in runtime speed. This may be due to unforeseen compiler optimisations, that are in some sense, intractably complicated. Perhaps the compiler keeps track of the threads even after they are destructed, making them easier to reopen.

5.2) I attempted to remove read and write expressions within the bulk of the for loops (loop_fusion), hoping that if the compiler was unable to optimise them, there would be a corresponding performance increase. However, upon changing this around there was no performance benefit as clearly icc was able to optimise for reads or writes that are not modified between statements. One confusing feature that arose here is that adding an empty reduction clause in the second for loop in loop_fusion, sped the code up. I have no idea why, but is a testament to the obfuscated nature of compiler and hardware optimisations.

5.3) AOS vs SOA vs AOSOA. Vectorisation works on the loop level and often fails to vectorise when it is inefficient to do so or there exist data dependencies, perhaps in the case of non-unit masked strides or multiple load and store instructions due to non contiguous data at the loop level. This implies that whilst an array of structures has good spatial locality, it has a poor standing with vectorisation. I therefore implemented a structure of arrays, the structure having 9 arrays of length params.nx*params.ny, given the knowledge that the two grid sizes divided amongst 16 threads would fit within each cores cache, simple maths shows that the size required to fit is in the order of tens of thousands of bytes well within the L2 cache. Surprisingly, after passing make check, this approach, which took many hours to create, took far longer to compute by a factor of eight. I can not pin down the reason why this was the case. I decided to attempt getting the "best of both worlds" and achieve spatial locality with the ability to vectorise. This was through an array of struct of arrays, such that there is an array of size (params.nx*params.ny)/number of cells in each struct. Each struct contains 9 arrays of a pre defined length. I made some headway with this progress finding it took a third off my time for the largest grid on sixteen cores from 17 seconds to 13, when I first tried it. Unfortunately due to bureaucracies within C, I found it difficult to dynamically size the structs and scale them in parallel.

5.4) How far could I go? I wanted to theorise about the limits of the speed of my computation. I looked in /proc/cpuinfo and researched the core Intel® Xeon® Processor E5-2670. I found that the memory bandwidth of the core is 6.4 gigabytes (51.4 gigabits) per second. The memory bandwidth I have is only perhaps a megabyte per second or so. Assuming the computation speed is only a fraction of the total speed which is often true then it would be the case that there is a vast under utilisation of the memory architecture in place.

5.5) Vectorising functions Using Intel VTune's profiler, I noticed the square root operation took a lot of time and that intel had a vectorised copy. The C intrinsic math function was not being vectorised and so if I had more time I would have replaced this, as it represents a bottleneck.

5.6) What about really really big grids? If you take a grid 16 times larger than the biggest grid, you would expect it to run 16 times slower (160 seconds). In actual fact Figure 5.1 depicts the actual running time. The 16 times larger is only 10 times slower. We can see the fixed costs are diminishing in their relative importance. The 64 times largest grid is only 38 times slower which is a massive relative amortisation.

5.7) Weak scaling. As you can see from Figure 5.2, the time it takes per worker per area drops dramatically the moment you start scaling the problem size upwards. For instance, it would start from 6.8 seconds with one worker on one core to 1.8 seconds for one worker when the problem grid is larger and shared amongst more workers.

Conclusion: Overall, good serial optimisations are key. Parallelisation with openMP should be a last finishing touch to a good memory hierarchy & vectorisation performance. The best performance was 10.08 seconds on the largest grid (256*256). I used ICC update 6 V2.

Figure 5.1 Amortisation

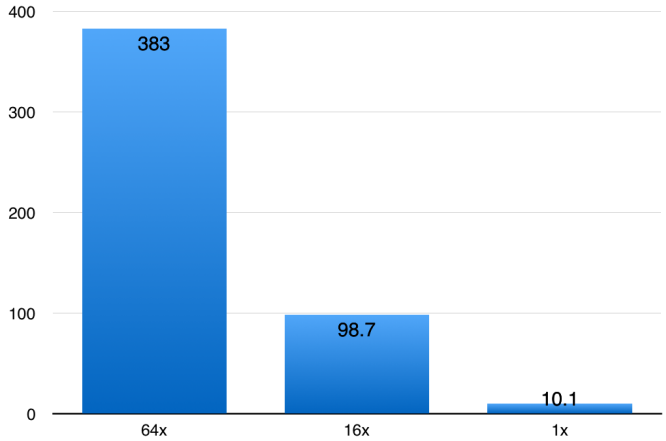
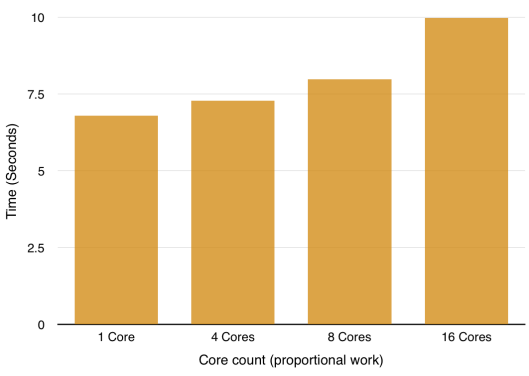


Figure 5.2 Proportional Work Size (Weak Scaling)



References:

Wikipedia: https://en.wikipedia.org/wiki/Neil_J._Gunther#Universal_Law_of_Computational_Scalability
(Fig 3.1) Using OpenMP Portable Shared Memory Parallel Programming