Single CPU Optimizations of SCEC AWP-Olsen Application

Hieu Nguyen (UCSD), Yifeng Cui (SDSC), Kim Olsen (SDSU), Kwangyoon Lee (SDSC)

Introduction

Reducing application's time-to-solution has been one of the grand challenges for scientific computing, especially at large-scale. This also applies to simulating earthquake using the Southern California Earthquake Center (SCEC) AWP-Olsen code. The SCEC 100-m resolution ShakeOut-D wave propagation simulations, conducted on Kraken at NICS using 64k cores with 14.4 billion mesh points for southern California region, consume hundreds of thousands of allocation hours per simulation. There has been an urgent need of reducing time spent on each individual processor since saving 1% execution time on each processor can save thousands of allocation hours per simulation. This work introduces single CPU optimization as a very effective solution.

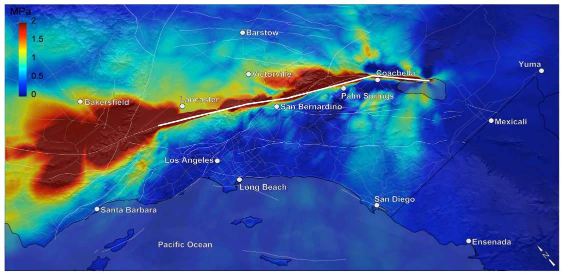

Figure 1. Peak dynamic Coulomb failure stress changes in Shakeout simulation of an Mw7.8 earthquake on the southern San Andreas Fault; 600 x 300 x 80 km domain, 100m resolution, 14.4 billion grids, 50k time steps. The latest kind of simulation has been performed in NICS Kraken with 64,000 processors.

General Performance Analysis

In the analysis of large-scale application in modern HPC environments, utilization of performance tool is essential as complexity often hinders information about program's "hot spots" where most of time is spent during execution, and critical hardware performance data. In our study, Cray Performance Analysis Tool (CrayPat) is used for performance profiling. The profiling results show that there are seven subroutines which account for about 40% of the execution time (see Figure 2) . These subroutines are our objects for optimizing.

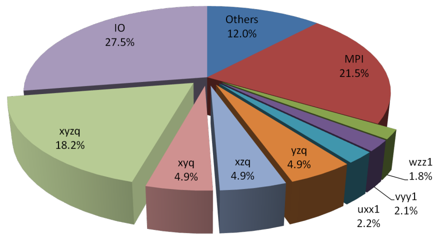

Figure 2: Time distribution of tasks and important subroutines for test problem on 200x200x40 km domain, resolution 100m using 1600 processors.

Optimal Mathematical Formulas

Cost to execute floating operations vary from microprocessor to microprocessor, however floating point division is always far more expensive than floating point addition and floating point multiplication (see table 1). Thus it can be our great advantage to reduce the use of division in frequently used formulas.

Operation	Addition	Multiplication	Division
Cost	3-6	4-8	32-45

Table 1: Cost of operations in clock cycle

In the four most expensive subroutines xyzq, xyq, xzq, and yzq, media parameters stored in two 3D arrays lam and mu are averaged using one of the two formulas below (only formulas for mu are shown, indices vary for different subroutines):

```
Formula 1: 1 addition, 3 divisions
xmu = 2./(1./mu(i,j,k)+1./mu(i,j,k-1))
Formula 2: 7 additions, 9 divisions
xl = 8./(1./mu(i,j,k)+1./mu(i+1,j,k)
      + 1./mu(i,j-1,k)+1./mu(i+1,j-1,k)
      + 1./mu(i,j,k-1)+1./mu(i+1,j,k-1)
      + 1./mu(i,j-1,k-1)+1./mu(i+1,j-1,k-1))
```

These formulas are considered expensive not only because of the cost of the multiple divisions but also because of the large difference in execution time between addition and division that degrade on-chip parallelism (additions have to wait long time for results from divisions).

$$\frac{1}{a}+\frac{1}{b}=\frac{a+b}{ab} \qquad \frac{1}{a}+\frac{1}{b}+\frac{1}{c}=\frac{bc+ac+ab}{abc}$$

Figure 3: Formulas for collecting group of 2 and group of 3

Mathematically equivalent formulas are introduced to eliminate these problems. Terms in the denominators of the formulas are collected in groups of 2 , 3 or 4 (see Figure 3). Multiple experiments shows that for formula 2, two groups of 4 gives the best performance. However, since values of the parameter arrays can be as large as 10^{10} , it is unsafe, in single precision, to have product of four of such values. The optimal choice is two groups of 3 and one group of 2. This ways, not only the number of divisions is reduced in both formulas but the on-chip parallelism is also improved as the execution time of two operands in each operator are more balanced.

```
(To save spaces elements from array mu are replaced by a,b,c,…)
Revised Formula 1: 1 addition, 2 multiplications, 1 division
      xmu = 2.*a*b/(a+b)
Revised Formula 2: 7 additions, 11 multiplications, 4 divisions
      xl = 8./((b*c+a*c+a*b)/(a*b*c)
           +  (e*f+d*f+d*e)/(d*e*f)
           +  (g+h)/(g*h))
```

Optimal parameter storage

Further study shows media parameter arrays mu and lam are computed once and remain unchanged during the whole simulation. In addition, even thought elements in the arrays mu and lam are used in both forms mu(i,j,k) and 1./mu(i,j,k) only the later one is used in frequently called subroutines. This suggests that the code should store the reciprocals of mu and lam original values. This way, we use costly operations (divisions to compute reciprocal values), however it is one-time calculation only. The results can be used in every time step. Improvement is significant since the number of time step in real simulation is usually larger than 40k .

Loop unrolling

All of the critical subroutines in the code share the same structure of three nested loop, looping over all nodes in the local mesh of each processor. CrayPat hardware counter performance ported on these loops demonstrate they have good memory access in general. However, cache utilization is very low. This is mainly due to the requirement of assessing values of multiple 3D arrays with the second or last indices varied. When one of these values is needed the whole cache line, which contain the value, is fetched in to L1 cache. Since the numbers of variables in the inner loops are large the cache line is usually evicted just after one reference. To improve cache utilization we need to use as many values per one fetch as possible.

```
do i= nxb,nxe
      …
  vyz=c1*(v1(i,j,k+1)-v1(i,j,k))+c2*(v1(i,j,k+2)-v1(i,j,k-1))
      …
enddo
do i= nxb,nxe,2
      …
  vyz=c1*(v1(i,j,k+1)-v1(i,j,k))+c2*(v1(i,j,k+2)-v1(i,j,k-1))
  vyz1=c1*(v1(i+1,j,k+1)-v1(i+1,j,k))+c2*(v1(i+1,j,k+2)-v1(i+1,j,k-1))
      …
enddo
```

Loop unrolling is introduced as a very effective way to improve cache utilization. In loop unrolling of depth k, the loop counter is reduce by k times and the amount of work inside the loop is k times bigger. Above is an example of loop unrolling to depth 2. In this example, at least 2 values are referenced for every fetch of a cache line for values of array v1.

	Original Version	Optimized Version
Time	4.279909 secs	2.330380 secs
REQUESTS_TO_L2:DATA	13.540M/sec	25.170M/sec
DATA_CACHE_REFILLS	4.430M/sec	8.197M/sec
PAPI_L1_DCA	507.742M/sec	1577.473M/sec
D1 cache hit ratio	97.4% hits	98.4% hits
D2 cache hit ratio	33.4%	33.9%
D1+D2 cache utilization	56.29 refs/miss	94.84 refs/miss
System to D1 refill	9.019M/sec	16.633M/sec
System to D1 bandwidth	550.495MB/sec	1015.198MB/sec

Table 2: CrayPat API profiling results of subroutine xzq before and after loop unrolling to depth 2

Loop unrolling clearly improve the cache utilization (see table 2) however it might also increase the calculation complexity in both number of variables and number of operations. As the number of registers on a machine is limited over unrolling could degrade loop performance. Empirical study prove that unrolling to depth 2 give the best performance for subroutines xyq and xzq.

Experimental Results

Improvements are measured by comparing the average computing time per step (this includes both communication, which is untouched, and calculation time) of modified code and the original one. With all the changes discussed in this work and some other small changes, we was able to gain a speed up of up to 31.25%.

Number of Processors	Computing Time per Step			Elapsed Time		
	Original	Optimized	Improvement	Original	Optimized	Improvement
2K	3.148s	2.174s	30.94%	3162.784s	2182.335s	30.99%
4K	1.584s	1.089s	31.25%	1596.44s	1123.424s	29.63%
8K	0.820s	0.571s	30.37%	832.936s	583.69s	29.92%
16K	0.432s	0.305s	28.47%	453.642s	334.518s	26.26%

Table 3: Improvements from optimized version of problem of 600 x 300 x 80 km domain, 100m resolution, 1k time step.