

Multi-mode parallelism of a scientific code

by Jack Hinton

FILE	1568_JACK_HINTON_MULTI-MODE_PARALLELISM_OF_A_Scientific_CODE_113617_1250497586.DOCX		
TIME SUBMITTED	04-MAY-2018 08:15PM (UTC+0100)	WORD COUNT	739
SUBMISSION ID	88415743	CHARACTER COUNT	3559

Multi-mode parallelism of a scientific code

Jack Hinton - p147676

1.1 Optimisation

1.1.1. Techniques

The techniques trialled in the optimisation of the provided code were focus primarily around reducing the quantity of computation that was performed in looped sections. To this end, a series of processes were performed upon the code including :

1.1.1.1. Multiplication operations moved

Multiplication operations by the loop iterator were simplified to be iterative summation operations within the loop declaration

```
(
...
for (size_t L = 0; L < arrSize; L++)
{
    for (size_t I = 0; I<arrSize; I++)
    {
        size_t b = L * arrSize + I;
    }
}
...
⇒
...
size_t b = 0;
for (size_t L = 0; L < arrSize; L++)
{
    b = L*arrSize;
    for (size_t I = 0; I<arrSize; I++, b++)
    {
    }
}
...
)
```

1.1.2. Use of inbuilt library functions

Loops that performed operations such as zeroing an array have been replaced with their more efficient [1][2] functional counterparts.

```
(  
...  
for (size_t l = 0; l<arrSize; l++)  
{  
    size_t b = L * arrSize + l;  
    nC[b] = 0.0;  
    nD[b] = 0.0;  
...  
⇒  
...  
    memset(nC, 0, sizeof(nC));  
    memset(nD, 0, sizeof(nC));  
...  
)
```

1.1.3. Prior computation of multiplications

Where multiplications could not be replaced with incremental summations, they were instead precomputed in a separate loop. Whilst this method does not entail as great a speed increase as the above (see 1.1.1), it nevertheless can reduce repeated multiplication and thus can provide some benefit.

```
(  
...  
for (size_t l = 0; l < arrSize; l++)  
{  
    size_t b = L * arrSize + l;  
...  
}
```



```
...  
for (int loop = 0; loop < arrSize; loop++)  
{  
    arrMult[loop] = loop * arrSize;  
}  
for (size_t l = 0; l < arrSize; l++)  
{  
    b = arrMult[L] + l;  
...  
}
```

1.1.4. Removal of unnecessary loops

In some areas of the provided code, nested loops were used to obtain data that could have been obtained with less code repetition.

```
(  
for (size_t L = 0; L < arrSize; L++)  
{  
    for (size_t I = 0; I < arrSize; I++)  
    {  
        size_t b = L * arrSize + I;  
        nC[b] = 0.0;  
        nD[b] = 0.0;  
  
        for (size_t J = 0; J < arrSize; J++)  
        {  
            size_t k1 = L * arrSize + J;  
            size_t k2 = I + J * arrSize;
```

...



```
...  
for (int L = 0; L < arrSize; L++)  
{  
    k10 = arrMult[L];  
  
    for (size_t I = 0; I < arrSize; I++)  
    {  
        b = arrMult[L] + I;  
  
        k2 = L + arrMult[I];  
        k1 = k10 + (k2 - I) / arrSize;  
  
    ...  
    )
```

1.2. Results

The combination of the above optimisation methods actually resulted in code that ran significantly slower than in the un-optimised instance. This could be due to the added complexity inherent in some of the time-saving measures employed - As this program is not that large, the saving in runtime obtained may be outweighed by the overhead caused by declaration of extra terms[3]. In recognition of this, the parallelism employed in the next section is applied to the original code bereft of much of the previously described optimisation measures.

Parallelism

Multithreading

Parallelism using multithreading was achieved by splitting the input list into equal segments and distributing amongst the threads used. Each thread then had its own ComputationEngine object with which to perform the necessary calculations. The resulting data is then written to file in sequence using a loop performed subsequent to the completion of the threaded processes. This measure entailed a speed increase of between 600% and 13333% depending on the number of threads utilised.

```
(  
...  
for (int outerloop = 0; outerloop < threadnums; outerloop++)  
    {  
        arrArrays[outerloop] = new double[size2];  
        sizeMultOuter = size2 * outerloop;  
  
        for (int loop = 0; loop < size2; loop++)  
        {  
            arrArrays[outerloop][loop] = arr[sizeMultOuter+loop];  
        }  
        ceArr[outerloop] = new ComputationEngine(&arrArrays[outerloop], size2);  
  
        threadArr[outerloop] = thread(runCE, &ceArr[outerloop]);  
    }  
  
    for (int loop = 0; loop < threadnums; loop++)  
    {  
        threadArr[loop].join();  
    }  
...  
)
```

References:

- [1] S. Falke, F. Merz, and C. Sinz, "Extending the Theory of Arrays: memset, memcpy, and Beyond," in *Verified Software: Theories, Tools, Experiments*, 2014, pp. 108–128.
- [2] H. O. Kultala *et al.*, "Exposed datapath optimizations for loop scheduling," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2017 International Conference on*, 2017, pp. 171–178.
- [3] D. Detlefs, A. Dosser, and B. Zorn, "Memory allocation costs in large C and C++ programs," *Softw. Pract. Exp.*, vol. 24, no. 6, pp. 527–542, Jun. 1994.

Multi-mode parallelism of a scientific code

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

28/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

GRADING FORM: SMU GENERIC_GJ

JACK HINTON

28

RESEARCH-INFORMED LITERATURE



A modest number of academically appropriate sources used.

3

KNOWLEDGE AND UNDERSTANDING



While I suspect you understand the topic well, by your own admission, this assignment is a long way from finished. As such, the mark in this, and other sections are rather limited.

9.5

ANALYSIS



Some evidence of analysis, but it would've been nice to see some timing data or meaningful conclusions drawn, even from the limited technical work.

6

PRACTICAL APPLICATION AND DEP.



Some threading completed, on the correctly identified bottleneck. Don't love the series of if statements in calculating the thread limits at all, however!

6

SKILLS FOR PROFESSIONAL PRAC.



Your writing style is good and the report is appropriately presented.

3.5