H. Dwyer

Aug 2019

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

**EndoVantage Challenge Notes**

Received zip file Friday 8/02/19. Result due Monday morning, 8/04/19.

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

8/02/19

Unpacked zip file.

Printed the instructions and problem statement.

I understand the 2D triangulation, I think

* the grid is square in the examples, 200x200
* the grid has the same number of subdivisions vertically and horizontally. The number is controlled by a global constant DX\_GLOBAL, which is the dimension of a triangle in X.
* there is no DY\_GLOBAL ... we use DX\_GLOBAL in both directions
* the 2D projection of the elements will all be axis-aligned right triangles of the same size and with

What should we do if DX\_GLOBAL does not evenly divide the dimension of the grid?

Looking at the code, I see that there are implementations for most (all?) required procedures.

We will use a timer to compare performance of the original implementation and any changes I make.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

There does not seem to be any procedure to set Z values for the nodes.

If the Z values are all 0.0 then all the elements are the same shape and area, all the normals are vertical, and the area matrix is filled with identical values.

The problem description contains “upon arrival” ... but I have not heard that I am invited to visit (?)

I assume that I will send in a solution and they will evaluate it before making a decision on how to proceed.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

We are computing a matrix which contains the areas of the elements.

We then compute . I know that the matrix will be square and symmetric.

I note that the routine which computes this matrix product does not use the symmetry.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

It does not appear that the supplied procedures make any use of parallel processing.

I will explore this as time allows.

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

8/03/19

Installed VS 2019. Successfully created a “Hello world” console application.

VS 2019 runs very slowly on my machine

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

There does not seem to be a VS project included in the supplied files. The main routine seems to be a simple console application. The emphasis is clearly on run time, so I will not take the time to do even a simple user interface.

**The next step would seem to be to get the supplied code into a VS 2019 console app and get it to execute.**

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Added KPoint, Element, Node, and Vector to console app.

It builds, runs, says “Hello”.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Moved body of supplied main.cpp into VS solution

Builds, executes. Returns 441, 800

using

DX\_GLOBAL = 10

double dx = DX\_GLOBAL;

double width = 200;

double height = 200;

With 20 intervals there are 21 endpoints: 21\*21=441

With 20 intervals there are 20\*20 rectangles and 20\*20\*2=800 triangles.

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

Placed code and header files under version control in branch **EndoTestBranch**.

Added this file to version control.

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

Added progress messages.

* With DX\_GLOBAL = 10 we go into the MatrixMatrix procedure and do not come out for a while (I did not wait).
* With DX\_GLOBAL = 100 we get through all the code with a time of about 0.003 sec
* Array Y contains ridiculous values and the norm is ‘inf’

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

procedure **SetupModel**

Decreased the operation count slightly.

Found that two local dynamic arrays had not been deleted, so added two delete[] statements.

With DX\_Global = 10, time for **SetupModel** is approx. 0.045 sec.

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

procedure **AllocateArrays**

Decreased the operation count slightly.

With DX\_Global = 10, time for **SetupModel** and **AllocateArrays** is approx. 0.071 sec.

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

procedure **PopulateArrays**

Decreased the operation count slightly.

Moved call to get element area into the outer loop.

With DX\_Global = 10, time for **SetupModel**, **AllocateArrays** and **PopulateArrays** is approx. 0.078 sec.

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

class **Element**

Made the center, area and face normal into protected data fields and added booleans to indicate if they have been computed.

Modified **GetCenter**, **GetArea**, and **GetFaceNormal** so that the values are computed only as required.

Combined the computation of the face normal and the area into one routine.

With DX\_Global = 10, time for **SetupModel**, **AllocateArrays** and **PopulateArrays** is approx. 0.071 sec.

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

8/04/19

procedure **MatrixMatrix**

Created a debug procedure **Debug\_MockAreaMatrix** to allow me to create a matrix **area** of any size, for testing of **MatrixMatrix**.

Using size 3x3 it gives

The result from **MatrixMatrix** is

Calculation using a calculator confirms that the values in ATA are correct.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

We know that a matrix product will always be symmetric.

Procedure **MatrixMatrix** does not take advantage of this. By doing so, we can cut execution time in half.

Using the 3x3 test, we see that the rewrite produces the correct result.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Time test – average of 3 trials

|  |  |  |  |
| --- | --- | --- | --- |
| n | time for  **Debug\_MockAreaMatrix** | time for  **MatrixMatrix** (orig) | time for  **MatrixMatrix** (rewrite) |
| 300 | 0.020 | 0.355 | 0.157 |
| 600 | 0.077 | 6.174 | 2.775 |
| 1200 | 0.281 | 82.800 (one trial only) | 41.254 |

As expected, **Debug\_MockAreaMatrix** is very clearly .

We expect a simple implementation of matrix multiplication to be but we are not seeing that here.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

As an experiment, I try modifying the routine to do AA rather than ATA.

The result will be numerically incorrect (result is not symmetric but is computed as if it was) but the time may tell us something.

Time test – average of 3 trials

|  |  |  |  |
| --- | --- | --- | --- |
| n | time for  **Debug\_MockAreaMatrix** | time for  **MatrixMatrix** (rewrite) | time for  **MatrixMatrix** (AA) |
| 300 | 0.020 | 0.157 | 0.134 |
| 600 | 0.077 | 2.775 | 1.910 |
| 1200 | 0.281 | 41.254 | 22.560 |

It appears that indexing does have something to do with this.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

When we compute the entry is the dot product of row i of with column j of

That is, the dot product of column i of with column j of .

If we compute the entry is the dot product of row i of with column j of

That is, the dot product of row i of with row j of .

Since is stored as an array of rows, we might be able to compute more quickly because we are working directly with rows. Note: In general ... I am just exploring ideas here.

Created a procedure **Degug\_MatrixMatrix\_AAT** which computes using local var to store row i and row j.

The 3X3 case shows that the computation is correct.

Time test – average of 3 trials

|  |  |  |  |
| --- | --- | --- | --- |
| n | time for  **Debug\_MockAreaMatrix** | time for  **MatrixMatrix** (rewrite) | time for  **MatrixMatrix\_AAT** |
| 300 | 0.020 | 0.157 | 0.088 |
| 600 | 0.077 | 2.775 | 0.671 |
| 1200 | 0.281 | 41.254 | 6.436 |
| 2400 | 1.041 |  | 44.035 |

These ratios are in the ballpark of , which is what we would expect.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

If we transpose the area matrix before we compute the result will be .

Since the matrix **area** seems to be used nowhere else, we can record it in transposed order during **PopulateArrays**.

Coded the modifications.

3x3 test produces the correct result. Times are the same (as they should be).

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

procedure **MatrixVector**

Time tests with **MatrixVector** added into the trial show times which are in the same range as the times we get without **MatrixVector**, The time for **MatrixMatrix** so dominates the total time that the variation in time between trials makes it impossible to see the time used by **MatrixVector** just by looking at time overall.

I will make a minor change to **MatrixVector**, but the real issue remains **MatrixMatrix**.

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

procedure **MatrixMatrix\_AAT**

The procedure is behaving in the mocked tests in an way.

This is usual for a general matrix-matrix multiplication.

(1) I am wondering if there is some sort of pattern (beyond symmetry) when we do which might be useful ... I have never heard that there was, but it is not something I have ever really thought about.

(2) Matrix-matrix multiplication is used as an example in the documentation of the parallel\_for loop.

<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-write-a-simple-parallel-for-loop#matrix-and-stopwatch-example>

Unfortunately it appears that this is strictly C#, at least as far as I can find.

I also referred to

<http://www.alecjacobson.com/weblog/?p=4544>

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

procedure **MatrixMatrix\_AAT\_Parallel**

When I run the program from VS in Release mode with **MatrixMatrix\_AAT**, the TaskManager shows that the task is using about 33% of the CPU.

With the threaded version, CPU peaked to about 95%, and seemed to average around 85%. At times, disk and memory use were quite high. Unfortunately, the values being computed are not correct. The non-threaded version will differ slightly for time between runs, but the norm is always exactly the same.

Non-Threaded Results

|  |  |  |  |
| --- | --- | --- | --- |
| **Simulation #** | **DX\_GLOBAL** | **L2 Norm** | **Execution time (sec)** |
| **1** | **50** | **4.20614e12** | **0.001** |
| **2** | **20** | **5.30094e10** | **0.040** |
| **3** | **10** | **1.75978e9** | **2.141** |
| **4** | **5** | **5.66047e7** | **109.927** |
| **5** | **1** | **not available** | **not available** |

The threaded version gives times which are a bit quicker, but every run produces a different norm value (they are not even close).

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

**Matrix size for area and ATA, as a function of DX\_GLOBAL ()**

The number of columns and rows is

The number of nodes is

The area matrix is square with dimension

The number of entries in the area matrix is approximately

Each entry is a double.

The ATA matrix is the same size as the area matrix.

The space required by these two matrices alone is approximately

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DX\_GLOBAL | 50 | 20 | 10 | 5 | 1 |
| size | 36 KB | 1.37 MB | 22.0 MB | 351 MB | 214 GB |

Even if we take advantage of the fact that ATA is symmetric, this is still a lot of memory.

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

**Rough Estimate for the time to run DX\_GLOBAL = 1**

A very rough estimate of time may be possible, assuming time is on the size of the area matrix.

We might expect that the time for DX\_Global = 1 is approximately