Computational performance with evalhf and Compile: A Newton fractal as case study

Dave Linder
Software Architect, Mathematical Software
Maplesoft

This follows up on an earlier article that discussed functionality differences amongst Maple's various modes for floating-point computation. The present article focuses on relative performance.

The driving example here is computation of a particular Newton fractal, which is easily parallelizable. We compute an image representation for this fractal under several computational modes, using both serial and multithreaded computation schemes.

This worksheet was executed using Maple 18.01 for 64bit Linux on a 4-core Intel Core i5 machine. Performance results will be platform dependent.

The iterative process of the computations done throughout this document gives rise to an image representation of a so-called Newton fractal.In Maple a grayscale image can be stored in a Matrix with hardware double-precision **float[8]** datatype.

The computational procedures each work by populating some Matrix with floating-point values calculated by a looped iteration.

The univariate polynomial example used in this document is taken as **n** as shown below. The computational procedures of this document work separately with the real and imaginary components of univariate polynomial **n** and it derivative **d**. The various real and imaginary components are computed here for illustration, and these formulae can be found within the computational procedures. No conversion to Horner form is made, and the same formulae are used throughout.

```
> n := z^3-2*z^2+2;
```

$$n := z^3 - 2\,z^2 + 2$$

```
> nri := subs(z = zr+I*zi, n);
```

$$nri := (zr + \mathrm{I}\,zi)^3 - 2\,(zr + \mathrm{I}\,zi)^2 + 2$$

```
> evalc(Re(nri));
```

$$-3\,zi^2\,zr + zr^3 + 2\,zi^2 - 2\,zr^2 + 2$$

```
> evalc(Im(nri));
```

$$-zi^3 + 3\,zi\,zr^2 - 4\,zi\,zr$$

```
> d := diff(n, z);
```

$$d := 3\,z^2 - 4\,z$$

```
> dri := subs(z = zr+I*zi, d);
```

$$dri := 3\,(zr + \mathrm{I}\,zi)^2 - 4\,zr - 4\,\mathrm{I}\,zi$$

```
> evalc(Re(dri));
```

$$-3\,zi^2 + 3\,zr^2 - 4\,zr$$

```
> evalc(Im(dri));
```

$$6\,zi\,zr - 4\,zi$$

The precise details of the algorithm are not all of central interest here.The central focus is rather that the algorithm involves intensive numeric operations only, and so is a good candidate for the various computational acceleration methods described in the earlier article. One detail of the algorithm that is key is that the calculations for each entry of the resulting image Matrix can be done independently, and the computation is parallelizable.

Click on the Code Edit Region button below to define the procedure. If you wish to see the procedure definition, right-click and select “Expand Code Edit Region”.

```
sourceNewton := proc
```

To allow us to focus on the performance of just the iterative process we construct all Vector and Matrices and pass then to the procedure which will act on them "in-place". This also allows a call to our procedure to be used easily under evalhf mode.

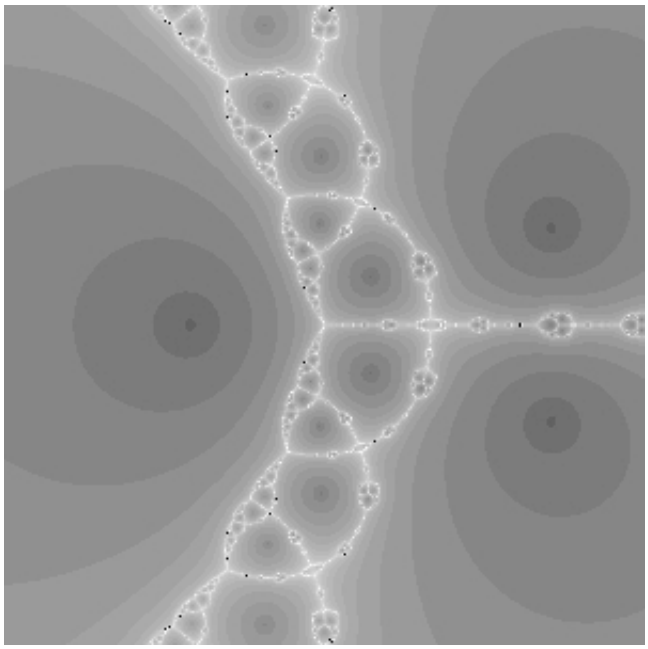
```
> N := 300:
  X := Vector(N, datatype=float[8]):
  Y := Vector(N, datatype=float[8]):
```

We first run the procedure without any special optimization techniques. When this version is executed serially the timing report indicates that the real (wall clock) time for the computation is the same as the total cpu time spent in the single computational thread.

```
> M := Matrix(N, N, datatype=float[8]):

  CodeTools:-Usage(
    sourceNewton( N, X, Y, M, -2.0, 2.0, -2.0, 2.0 )
                  ):
memory used=0.69GiB, alloc change=256.00MiB, cpu time=11.55s, real
time=11.06s, gc time=2.55s
>
```

In Maple 18, the Matrix **M** which represents a grayscale image can be viewed by directly embedding it into the current document. In Maple 17 the command **ImageTools:-View** can be used instead of **ImageTools:-Embed**.

```
ImageTools:-Embed( ImageTools:-FitIntensity(M) );
```



Now we will execute that same computation under the evalhf interpreter.For this example the total real time for the computation has been improved by about a factor of about seven.

```
> Mevalhf := Matrix(N, N, datatype=float[8]):

  CodeTools:-Usage(
    evalhf( sourceNewton( N, X, Y, Mevalhf, -2.0, 2.0, -2.0, 2.0 ) )
                  ):
memory used=1.19KiB, alloc change=0 bytes, cpu time=1.49s, real time=
1.47s, gc time=0ns
>
```

A quick numeric test shows that the two results are the approximately same.

```
  LinearAlgebra:-Norm( M - Mevalhf );
                                    0.
```

Now we will create a new version of the procedure which could be run using multiple threads. This is quite easily done after noticing that the entries **M[ii,jj]** are computed independently of each other. We can divide up the computation by operating on only a subset of adjacent rows of Matrix **M** each time we call the procedure.

In the following Code-Edit Region the **procedure sourceNewton_col** has loop index **ii** which runs only from index value **R_low** to **R_High**, which are new parameters of the procedure. If we called **sourceNewton_row** and passed **R_Low** a value of **1** and **R_High** a value of **N** then all entries of Matrix **M** would be computed as a single serial execution.

Click on the Code Edit Region button below to define the procedure. If you wish to see the procedure

definition, right-click and select “Expand Code Edit Region”.

```
sourceNewton_row := proc
```

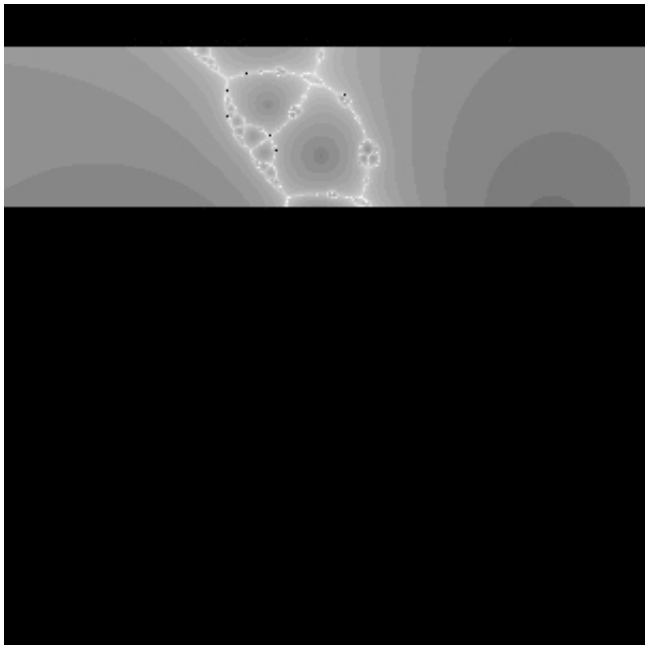
Let's compute over only rows 20 through 95, to illustrate a single instance of this procedure, computing serially. This worksheet has been executed on a 4-core machine.

```
> M_subrows := Matrix(N, N, datatype=float[8]):

  CodeTools:-Usage(
    evalhf( sourceNewton_row( 21, 95, N, X, Y, M_subrows, -2.0, 2.0,
  -2.0, 2.0 ) )
                  ):
memory used=1.19KiB, alloc change=0 bytes, cpu time=380.00ms, real
time=384.00ms, gc time=0ns
>
```

The timing of that last computation is much as expected. It computed results for only **95-21+1=75** rows, which is 1/4 of the total **N=300** rows of **M**. And it took close to 1/4 of the time taken for computing the full Matrix results serially.

We can view the partial result.

```
ImageTools:-Embed( ImageTools:-FitIntensity(M_subrows) );
```



Next we use Maple's Task parallel execution mechanism to run multiple instances of our new procedure in parallel. Each of the calls to the procedure will compute over a different subset of the rows of Matrix **M**.

Note that on MS-Windows the performance under parallel execution may not improve upon that of the serial execution.

The method works by using a "task" which examines the number of rows specified (which is **R_High-RLow+1**) and if that is greater than some given value then it splits into a pair of new tasks which each act only half of that subset of rows.

If the number of rows specified for any particular instance of a task is less than the given value then the procedure which does the actual computation is called, rather than splitting yet again.

The Task mechanism allows the instances of actual computation (over various subsets of the rows, after splitting) to be run in parallel, i.e.. concurrently.

Click on the Code Edit Region button below to define the procedure. If you wish to see the procedure definition, right-click and select “Expand Code Edit Region”.

```
newtonTask := proc
```

The multithreaded Task process is next executed in full, without using evalhf. The computational subtasks run in parallel.

```
> M_row := Matrix(N, N, datatype=float[8]):

  CodeTools:-Usage(
    Threads:-Task:-Start( newtonTask, 1, N, N, X, Y, M_row, -2.0,
  2.0, -2.0, 2.0 )
                    ):
memory used=0.69GiB, alloc change=0 bytes, cpu time=13.88s, real time=
5.24s, gc time=3.05s
```

The results for all entries of the Matrix M were computed, and the real (wall clock) time to execute was reduced to about 1/2 of the unaccelerated serial version.

Note that the *cpu time* reported by the CodeTools:-Usage command is the aggregate of cpu time resources used by all cores running in parallel. For that last computation the aggregate cpu time was 13.88 seconds, but the computation completed in only 5.24 seconds of real time.

The multithreaded Task process is next executed in full, but using **evalhf**.

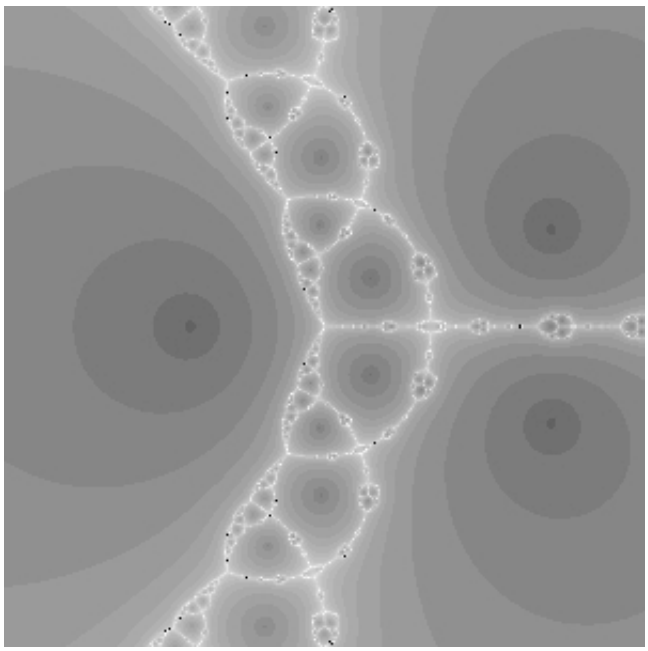
```
> M_row := Matrix(N, N, datatype=float[8]):

  CodeTools:-Usage(
    Threads:-Task:-Start( newtonTask_evalhf, 1, N, N, X, Y, M_row,
  -2.0, 2.0, -2.0, 2.0 )
                    ):
memory used=51.72KiB, alloc change=0 bytes, cpu time=1.66s, real time=
453.00ms, gc time=0ns
```

The results for all entries of the Matrix M were computed, and the real (wall clock) time to execute was reduced to about 1/3 of the serial version also run under evalhf.

For that last computation the aggregate cpu time was 1.66 seconds, but the computation completed in only 0.453 seconds of real time.

The resulting image can be viewed and seen to agree with the original result.

```
> ImageTools:-Embed( ImageTools:-FitIntensity(M_row) );
```



Let's now turn our attention now to the Compiler. In order to use the Compiler effectively it is often prudent and sometimes necessary to place type information on the parameters in the procedure definition.

New versions of both the serial procedure **sourceNewton** and the parallelizable procedure **sourceNewton_row**, with type information on parameters and local variables, are given in the Code Edit Region below.

By giving the a source procedure **option threadsafe** the Compiler returns a compiled procedure which can run in parallel.

Click on the Code Edit Region button below to define the procedures. If you wish to see the procedure definitions, right-click and select “Expand Code Edit Region”.

```
sourceNewton_row_typed := proc
```

The procedure **sourceNewton_row_typed** is now compiled, and the resulting new procedure is assigned as **csourceNewton_row_typed**.

```
> csourceNewton_row_typed := Compiler:-Compile
```

```
  (sourceNewton_row_typed):

>
```

The full result is once again computed, running the compiled version serially. This executes about twenty times faster than the parallel computation run under **evalhf**, on a 4-core machine.

```
  M_row := Matrix(N, N, datatype=float[8]):

  CodeTools:-Usage(
    csourceNewton_row_typed( 1, N, N, X, Y, M_row, -2.0, 2.0, -2.0,
  2.0 )
                   );
memory used=1.27KiB, alloc change=0 bytes, cpu time=20.00ms, real
time=25.00ms, gc time=0ns
>
```

As the fastest choice of all, the full result is now computed by running the compiled version in parallel. This possibility is new to Maple 18, for which the runtime environment of the Compiler has been made thread-safe. The result is twice as fast again as for the compiled version running serially.

```
  M_row := Matrix(N, N, datatype=float[8]):

  CodeTools:-Usage(
    Threads:-Task:-Start( newtonTask_row_typed, 1, N, N, X, Y,
  M_row, -2.0, 2.0, -2.0, 2.0 )
                   ):
memory used=51.53KiB, alloc change=0 bytes, cpu time=20.00ms, real
time=12.00ms, gc time=0ns
>
```

We could view the result, if we wished.

```
  #ImageTools:-Embed( ImageTools:-FitIntensity(M_row) );
```

Here is a tabulated summary of the timings.

	Serial	Multithreaded
unaccelerated	11.06	5.34
evalhf mode	1.47	0.453
Compiled	0.025	0.012

Timings in seconds: N=300

It's clear that for intensive computations the effort of adjusting code to run in one of the accelerated floating-point computational environments may well be worthwhile.

An aspect of the timing results worth noting is that the serial run of the Compiled version performed much better than the multithreaded version running under **evalhf.** Similarly the serial run under **evalhf**

performed better than the multithreaded unaccelerated version.For a particular application it may be more beneficial to change the computation mode than to compute in parallel, unless the multithreaded performance scales very well with the number of cores and a large number of cores are available.

There are several more interesting aspects to consider which can affect performance. For example, these computational procedures might perform better if the order of nesting of the double loop over the Matrix entries, and the Task-splitting across rows, matched the storage in memory (**C_order** versus **Fortran_order**). It would also be worthwhile to see whether calculating with complex **z** rather than separated real and imaginary components **zr** and **zi** performs better or worse, for each method. There is also the practical problem of how to get arbitrary formulae (such as used here for the polynomial and its derivative) into the body of a looping or iterating procedure. These are good topics for further investigation.