Summer project for Martine

HPL

Jul 8, 2014

We ask the question: Is it possible to make a simple, "student-style", serial Python program and with simple means, quickly create a parallel, high-performance version of the code that can run on large clusters of multi-core/GPU nodes? We will explore the question in a specific problem domain: solution of partial differential equations by finite difference methods on uniform meshes.

1 Simpler model problems

Although partial differential equations (PDEs) are in focus, it can for learning and test purposes be advantageous to have some simpler models to work with.

1.1 Numerical differentiation

Given discrete values $f(x_i)$ of a function $f(x)$ at uniformly distributed mesh points $x_i = i\Delta x$, $i = 0, \ldots, N_x$, we want to approximate the derivative df/dx at the interior mesh points, $f'(x_i)$, $i = 1, \ldots, N_x - 1$, by a centered finite difference:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2\Delta x}.$$

At the end points we use one-sided differences:

$$f'(x_0) = \frac{f(x_1) - f(x_0)}{\Delta x}, \quad f'(x_{N_x}) = \frac{f(x_{N_x}) - f(x_{N_x-1})}{\Delta x}.$$

Serial implementations. A straightforward scalar implemention with explicit loops may look like

```
import numpy as np

def differentiate_scalar(f, a, b, n):
    """
    Compute the discrete derivative of a Python function
    f on [a,b] using n intervals. Internal points apply
    a centered difference, while end points apply a one-sided
```

```
    difference.
    """
    x = np.linspace(a, b, n+1)  # mesh
    df = np.zeros_like(x)       # df/dx
    f_vec = f(x)
    dx = x[1] - x[0]
    # Internal mesh points
    for i in range(1, n):
        df[i] = (f_vec[i+1] - f_vec[i-1])/(2*dx)
    # End points
    df[0]  = (f_vec[1]  - f_vec[0]) /dx
    df[-1] = (f_vec[-1] - f_vec[-2])/dx
    return df
```

A corresponding vectorized implementation takes the form

```
def differentiate_vec(f, a, b, n):
    """
    Compute the discrete derivative of a Python function
    f on [a,b] using n intervals. Internal points apply
    a centered difference, while end points apply a one-sided
    difference. Vectorized version.
    """
    x = np.linspace(a, b, n+1)  # mesh
    df = np.zeros_like(x)       # df/dx
    f_vec = f(x)
    dx = x[1] - x[0]
    # Internal mesh points
    df[1:-1] = (f_vec[2:] - f_vec[:-2])/(2*dx)
    # End points
    df[0]  = (f_vec[1]  - f_vec[0]) /dx
    df[-1] = (f_vec[-1] - f_vec[-2])/dx
    return df
```

Parallelization. Assume N_x is large and that we divide the mesh among processors. Think of $N_x = 10$ and three processors: processor 0 has x_0, x_1, x_2; processor 1 has x_3, x_4, x_5; and processor 2 has the rest, x_6, x_7, x_8, x_9, and x_{10}. To compute the derivative at the three points on processor 1 we need to access x_2 and x_6. We add these *ghost points* to the set of local mesh points on this processor. The other processors must also make use of ghost points.

1.2 Numerical integration

Integrals of a mathematical function $f(x)$ can be approximated by the trapezoidal rule:

$$I = \int_a^b f(x)dx \approx \Delta x \left(\frac{1}{2}f(a) + \frac{1}{2}f(b) + \sum_{j=1}^{N_x-1} f(a + j\Delta x) \right),$$

where Δx is the spacing between the $N_x + 1$ evaluation (mesh) points: $\Delta x = (b - a)/N_x$.

Serial implementations. A straightforward scalar implemention with explicit loops may look like

```
import numpy as np

def trapezoidal_scalar(f, a, b, n):
    """
    Compute the integral of f from a to b with n intervals,
    using the Trapezoidal rule.
    """
    h = (b-a)/float(n)
    I = 0.5*(f(a) + f(b))
    for i in range(1, n):
        x = a + i*h
        I += f(x)
    I = h*I
    return I
```

A corresponding vectorized implementation takes the form

```
def trapezoidal_vec(f, a, b, n):
    """
    Compute the integral of f from a to b with n intervals,
    using the Trapezoidal rule. Vectorized version.
    """
    x = np.linspace(a, b, n+1)
    f_vec = f(x)
    f_vec[0] /= 2.0
    f_vec[-1] /= 2.0
    h = (b-a)/float(n)
    I = h*np.sum(f_vec)
    return I
```

Parallelization. We now divide the mesh points among the processors. Each processor must sum its function values. The processors holding the first and last mesh points needs to adjust the function value at these points by a factor of one half. The one or all processors must collect the partial sums and form the final sum.

1.3 Random walk

- Perfectly parallelizable model (no communication before calculating statistics of the concentration of walkers).
- Demo programs from INF1100 can be used, but it is easier in 2D/3D to vectorize and compute in general if the walkers go SE, SW, NE, NW rather than N, S, E, W (i.e., the former can be computed by drawing two/three independent random integers -1 or 1, one in each space direction).

2 Key model problems

One of the most common time-consuming computing kernels when solving partial differential equations or running image processing algorithms is to move a (finite difference) stencil through a mesh. High computational efficiency of this kernel is what we want to study. A real physical problem where this kernel is basically the whole computation, is finite difference solution of the wave equation

$$u_{tt} = c^2\nabla^2 u + f,$$

where u is a function of space x and time t, c is a constant, and f is a function of x and t. The demo programs are therefore complete codes for solving such wave equations.

It can be wise to use a progressive set of test problems:

1. Simplest possible 1D wave equation code, `wave1D_u0.py`
2. Simplest possible 2D wave equation code, `wave2D_u0.py`
3. Variable coefficient 3D code for $u_{tt} = \nabla\cdot(c^2(x)\nabla u) + f$ for real performance tests

3 Technologies

In suggested order:

1. Numba
2. Cython: OpenMP loops
3. NumExpr for speeding up numpy expressions.
4. Migrating loops to C, parallelize with OpenMP
5. PyThran, paper
6. PiCloud gives access to supercomputing with Python and other languages.
7. Copperhead
8. Shedskin Python to C++ compiler
9. PETSc is a linear algebra library in C (with Python bindings via `petsc4py`), which can be used for parallel computing with vectors and matrices. We started a project on exploring petsc4py to solve PDEs.
10. Disco for parallel implementation of MapReduce algorithms. Once upon a time we started a tutorial for MapReduce in numerical computing.

Links:

- Comparison of Python, NumPy, Weave, Cython for Laplace operator (see the comments for NumExpr example)
- Ozsvald's 4h tutorial on speeding up Python with many tools

4 Tasks

Many of the tasks can be done in parallel.

1. Read parts of Finite difference methods for wave motion[1] to get a background of the algorithms for the wave equation. This document also explains how to migrate parts of the Python code to Fortran or C as well as the Cython technology. These are complementary tools to what is studied in the present project.
2. Read about Numba and try it on differentiation and integration.
3. Try Numba on wave equations (1D first, then 2D).
4. More to follow...

[1] If the numerics/mathematics in this document is not clear, it might help to look at a simple vibration ODE first (and that document builds on a basic introduction to finite difference methods, which starts absolutely from scratch).