Using Python Open Source Modules with Star-P:

A Task-Parallel Computing Example

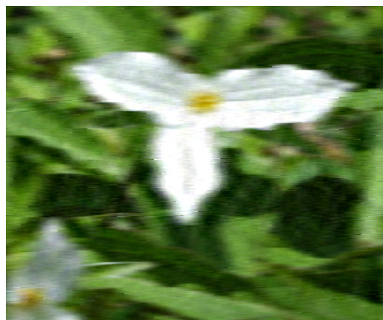
Introduction

One of the powerful aspects of Python is its open source nature, and availability of a wide array of Python modules for technical computing. These modules can be readily parallelized using the Star-P's task-parallel engine.

As an example, we use the Python Imaging Library for image compression. We use tools for image file I/O and accessing pixel data in different color bands. The numerical library NumPy is utilized to perform the image compression by singular value decomposition (SVD).

In this example, we process an array of tif-formatted image files. We build a module imageprocess where we define a function compress(imno,rank) The imno determines the index of the image in the array, and quality of the compression is determined by the variable rank. A larger rank implies that more data is kept.

Original Figure



compress(0,20)

compress(0,50)

The results of the compress function for two different compression levels are shown. The results illustrate the lossy compression algorithm; resolution is lost when the image is reconstructed from the compressed data.

Image Processing in Serial Python

Let's look at the module imageprocess in more detail. We begin by importing the Image module form the library, as well as the numerical module, NumPy for computing the singular value operation.

We use the Image.open() function to load the tif-formatted data to the memory. The image is divided to red, green and blue color bands which are processed individually. We use the Imaging Library's getdata function to extract the pixel values from each color band which are then converted into a NumPy 2-D array. The NumPy svd and matrix operations are used to compress the data and to reconstruct the image from compressed data. Finally, the three color bands are merged and saved as a new image.

```
import Image, numpy

def compress(imno,rank):
#
# Load data
#
    imagefile = '/mydir/flower%d.tif' % imno
    theimage = Image.open(imagefile)

    rank = int(rank)
    nx = theimage.size[0]
    ny = theimage.size[1]
#
# Process the color bands individually
#
    colors = theimage.split()

    for i in range(0,3):
#
# Extract pixel values
#
        colorarray = numpy.array(list(colors[i].getdata()))
        colorarray = numpy.resize(colorarray,theimage.size)
#
# Perform the SVD and compress the data
#
        (u,s,v) = numpy.linalg.svd(colorarray)

        uc = u[0:nx,0:rank]
        sc = numpy.diag(s[0:rank])
        vc = v[0:rank,0:ny]
#
# Rebuild a new image from the compressed data
```

```
#
        colorarray = numpy.dot(uc,numpy.dot(sc,vc))

        colorarray = numpy.floor(colorarray)
        extval = colorarray > 255
        colorarray[extval] = 255
        extval = colorarray < 0
        colorarray[extval] = 0

        colorarray = numpy.resize(colorarray,(nx*ny,))
        colorarray = colorarray.tolist()
        colors[i].putdata(colorarray)
#
# Merge the color bands, save a new image
#
    newimage = Image.merge('RGB',colors)

    newimafile = '/mydir/compressed%d_%d.tif' % (imno,rank)
    newimage.save(newimfile)
    return 0
```

Processing an Array of Images in Parallel Using Star-P

The function compress can be used to process individual images. To compress an array of images using serial computation we build another function which executes the function compress in a for-loop

```
def serial():

    for i in range(0,15):
        compress(i,50)

    return 0
```

To execute the serial loop from the Python client, we call:

```
>>> imageprocess.serial()
```

Let's look at processing the same set of image files in parallel. Independent iterations of in for loops such as above are straightforward to parallelize by using Star-P Python's starp.ppeval We create an array i that

holds the image indexes , instead of looping over the range. The values in this array tell each processor which images to load. The starp.ppeval is used to execute the compress – it automatically imports the Python Imaging Library installed on the server, and executes the compression of different images in parallel. Note that there are no changes to the function compress.

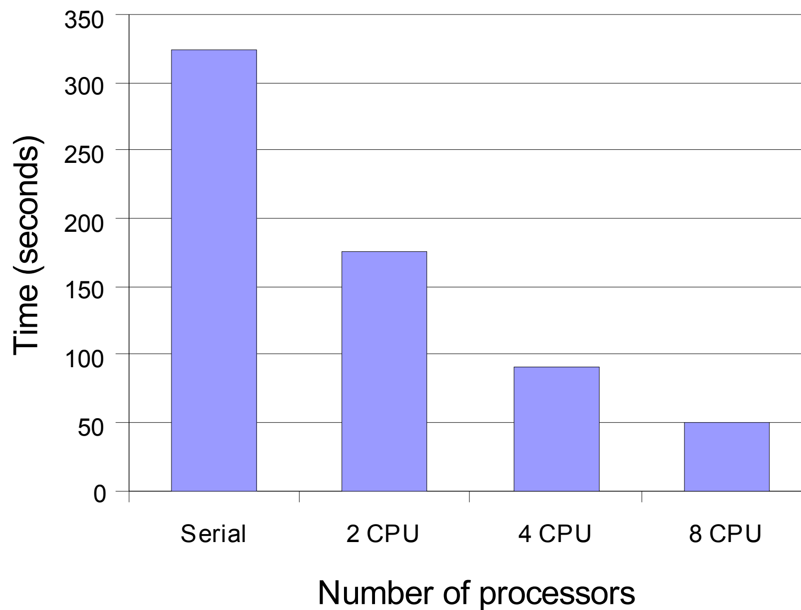
```
def parallel():

   i = starp.array(range(0,15))
   starp.ppeval(compress,i,50)

   return 0
```

Results

To lauch the parallel execution from the Python client, we simply call the parallel function:

```
>>> imageprocess.parallel()
```

The scaling is nearly linear, with a speedup factor 6.4 on 8 AMD Opteron™ processors, when processing a set of 16 images of 3.4 Mb size each.



This example illustrates how functions from Open Source modules can be parallelized in an easy-to-use fashion, while Star-P handles the low-level tasks such as managing client-server communications and distributing the input data. In this way, Star-P's task-parallel engine makes a broad range of Open Source modules available for parallel computation.