Astronomy and Computing 2 (2013) 1–10

Contents lists available at SciVerse ScienceDirect

Astronomy and Computing

journal homepage: www.elsevier.com/locate/ascom

Full length article

Parallel astronomical data processing with Python: Recipes for multicore machines

Navtej Singh *, Lisa-Marie Browne, Ray Butler

Centre for Astronomy, School of Physics, National University of Ireland - Galway, University Road, Galway, Ireland

HIGHLIGHTS

- We propose three recipes for parallelizing long running tasks on multicore machines.
- Native Python multiprocessing module makes it trivial to write parallel code.
- Parallel performance can be optimized by carefully load balancing the workload.
- The cross-platform nature of Python makes the code portable on multiple platforms.

ARTICLE INFO

Article history:
Received 25 July 2012
Accepted 30 April 2013

Keywords:
Astronomical data processing
Parallel computing
Multicore programming
Python multiprocessing
Parallel Python
Deconvolution

ABSTRACT

High performance computing has been used in various fields of astrophysical research. But most of it is implemented on massively parallel systems (supercomputers) or graphical processing unit clusters. With the advent of multicore processors in the last decade, many serial software codes have been re-implemented in parallel mode to utilize the full potential of these processors. In this paper, we propose parallel processing recipes for multicore machines for astronomical data processing. The target audience is astronomers who use Python as their preferred scripting language and who may be using PyRAF/IRAF for data processing. Three problems of varied complexity were benchmarked on three different types of multicore processors to demonstrate the benefits, in terms of execution time, of parallelizing data processing tasks. The native multiprocessing module available in Python makes it a relatively trivial task to implement the parallel code. We have also compared the three multiprocessing approaches—Pool/Map, Process/Queue and Parallel Python. Our test codes are freely available and can be downloaded from our website.

1. Introduction

In 1965, Gordon Moore predicted that the number of transistors in integrated circuits would double every two years (Moore, 1965). This prediction has proved true until now, although semiconductor experts[1] expect it to slow down by the end of 2013 (doubling every 3 years instead of 2). The initial emphasis was on producing single core processors with higher processing power. But with increasing heat dissipation problems and higher power consumption, the focus in the last decade has shifted to multicore processors—where each core acts as a separate processor. Each core may have lower processing power compared to a high end single core processor, but it provides better performance by allowing multiple threads to run simultaneously, known as thread-level parallelism (TLP). At present, dual and quad core processors are common place in desktop and laptop machines and even in the current generation of high end smart phones. With both Intel (Garver and Crepps, 2009) and AMD[2] working on next generation multicore processors, the potential for utilizing processing power in desktop machines is massive. However, traditional software for scientific applications (e.g. image processing) is written for single-core Central Processing Units (CPU) and does not harness the full computational potential of multicore machines.

Traditionally, high performance computing (HPC) is done on supercomputers with a multitude of processors (and large memory). Computer clusters using commercial off the shelf (COTS) hardware and open source software are also being utilized (Szalay, 2011). And recently graphical processing unit (GPU) based clusters have been put to use for general purpose computing (Strzodka et al., 2005; Belleman et al., 2008). The advent of multicore processors

* Corresponding author. Tel.: +353 91 492532; fax: +353 91 494584.
E-mail addresses: n.saini1@nuigalway.ie, reachnavtej@gmail.com (N. Singh), l.browne1@nuigalway.ie (L.-M. Browne), ray.butler@nuigalway.ie (R. Butler).

[1] From the 2011 executive summary of International Technology Roadmap for Semiconductor (http://www.itrs.net/links/2011itrs/2011Chapters/2011ExecSum.pdf).

[2] Advanced Micro Devices.

2213-1337/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.ascom.2013.04.002

provides a unique opportunity to move parallel computing to desktops and laptops, at least for simple tasks. In addition to hardware, one also needs unique software protocols and tools for parallel processing. The two most popular parallel processing protocols are Message Passing Interface (MPI) and OpenMP. MPI is used on machines with distributed memory (for example—clusters) whereas OpenMP is geared towards shared memory systems.

Parallel computing has been used in different sub-fields of astrophysical research. Physical modeling and computationally intensive simulation code have been ported to supercomputers. Examples include N-Body simulation of massive star and galaxy clusters (Makino et al., 1997), radiative transfer (Robitaille, 2011), plasma simulation around pulsars, galaxy formation and mergers, cosmology, etc. But most of the astronomical image processing and general time consuming data processing and analysis tasks are still run in serial mode. One of the reasons for this is the intrinsic and perceived complexity connected with writing and executing parallel code. Another reason may be that day to day astronomical data processing tasks do not take an extremely long time to execute. Irrespective of this, one can find a few parallel modules developed for astronomical image processing. The cosmic ray removal module CRBLASTER (Mighell, 2010) is written in C and based on the MPI protocol, and can be executed on supercomputers or cluster computers (as well as on single multicore machines). For co-addition of images, Wiley et al. (2011) proposed software based on the MapReduce[3] algorithm, which is geared towards processing terabytes of data (for example—data generated by big sky surveys like the SDSS[4]) using massively parallel systems.

In this paper, we have explored the other end of the spectrum—single multicore machines. We are proposing a few recipes for utilizing multicore machines for parallel computation, to perform faster execution of astronomical tasks. Our work is targeted at astronomers who are using Python as their preferred scripting language and may be using PyRAF[5] or IRAF[6] for image/data processing and analysis. The idea is to make the transition from serial to parallel processing as simple as possible for astronomers who do not have experience in high performance computing. Simple IRAF tasks can be re-written in Python to use parallel processing, but re-writing the more lengthy tasks may not be straightforward. Therefore, instead of re-writing the existing optimized serial tasks, we can use the Python multiprocessing modules to parallelize iterative processes.

In Section 2, we introduce the concept of parallel data processing and the various options available. Python multiprocessing in discussed in Section 3 with emphasis on native parallel processing implementation. Three different astronomical data processing examples are benchmarked in Section 4. In Section 5, we discuss load balancing, scalability, and portability of the parallel Python code. Final conclusions are drawn in Section 6.

2. Parallel data processing

Processors execute instructions sequentially and therefore, from the initial days of computers to the present, most of the applications have been written as serial code. Generally coding and debugging of serial code is much simpler than parallel code. However, debugging is an issue only for parallel programs where many processes depend on results from other processes—whereas it is not an issue while processing large datasets in parallel. Moving to parallel coding not only requires new hardware and software tools, but also a new way of tackling the problem in hand. To run a program in parallel, one needs multiple processors/cores or computing nodes.[7] The first question one asks is how to divide the problem so as to run each sub-task in parallel.

Generally speaking, parallelization can be achieved using either *task parallelization* or *data parallelization*. In task parallelism, each computing node runs the same or different code in parallel. Whereas, in data parallelism, the input data is divided across the computing nodes and the same code processes the data elements in parallel. Data parallelism is simpler to implement, as well as being the more appropriate approach in most astronomical data processing applications, and this paper deals only with it.

Considering a system with N processors or computing nodes, the speedup that can be achieved (compared to 1 processor) can be given as:

$$S = \frac{T_1}{T_N}, \tag{1}$$

where T_1 and T_N are the code runtime for one and N processors respectively. T_N depends not only on the number of computing nodes but also on the fraction of code that is serial. The total runtime of the parallel code using N processors can be expressed using Amdahl's law (Amdahl, 1967):

$$T_N = T_S + \frac{T_P}{N} + T_{\text{sync}} \tag{2}$$

where T_S is the execution time of the serial fraction of the code, T_P is the runtime of code that can be parallelized, and T_{sync} is the time for synchronization (I/O operations, etc.). The efficiency of the parallel code execution depends a lot on how optimized the code is, i.e. the lower the fraction of serial code, the better. If we ignore synchronization time, theoretically unlimited speedup can be achieved as $N \to \infty$ by converting the serial code to completely parallel code. More realistically, T_{sync} can be modeled as $K * \ln(N)$, where N is number of processors and K is a synchronization constant (Gove, 2010). This means that at a particular process count, the performance gain over serial code will start decreasing. Minimization of Eq. (2) gives:

$$N = \frac{T_P}{K}. \tag{3}$$

This means that the value of N for which the parallel code scales is directly proportional to the fraction of code that is parallel and inversely proportional to synchronization. In other words, by keeping N constant, one can achieve better performance by either increasing the fraction of parallel code or decreasing the synchronization time, or both.

We have used multiprocessing instead of multi-threading to achieve parallelism. There is a very basic difference between threads and processes. *Threads* are code segments that can be scheduled by the operating system. On single processor machines, the operating system gives the illusion of running multiple threads in parallel but in actuality it switches between the threads quickly (time division multiplexing). But in the case of multicore machines, threads run simultaneously on separate cores. Multiple processes are different from multiple threads in the sense that they have separate memory and state from the master process that invokes them (multiple threads use the same state and memory).

The most popular languages for parallel computing are C, C++ and FORTRAN. MPI as well as OpenMP protocols have been

[3] Model to process large datasets on a distributed cluster of computers.

[4] SDSS: Sloan Digital Sky Survey [http://www.sdss.org/].

[5] PyRAF is a product of the Space Telescope Science Institute, which is operated by AURA for NASA.

[6] IRAF is distributed by the National Optical Astronomy Observatories, which are operated by the Association of Universities for Research in Astronomy, Inc., under cooperative agreement with the National Science Foundation.

[7] The terms processors and computing nodes will be used interchangeably in the rest of the paper.