Stat250 W14 HW2

Karen Ng

February 13, 2014

Stat 250 HW 2 with Prof. Duncan Temple Lang

Author: (Karen) Yin-Yee Ng karenyng@ucdavis.edu

Github repository: `https://github.com/karenyng`

In [28]:
```
%autosave 60

# the following two lines allows the generating
# a floating table of content
# currently does not work with nbviewer online nor pdf
# I may fix it someday during my infinite spare time
%load_ext nbtoc
%nbtoc
from __future__ import division
%load_ext line_profiler
%load_ext memory_profiler
```

```
Autosaving every 60 seconds
The nbtoc extension is already loaded. To reload it, use:
  %reload_ext nbtoc



The line_profiler extension is already loaded. To reload it, use:
  %reload_ext line_profiler
The memory_profiler extension is already loaded. To reload it, use:
  %reload_ext memory_profiler
```

Part I

Disclaimer:

Both of the methods that I implemented share the same problem: The scheduling of the multiple concurrent process / threads >> # of cores.

A smarter way to do it is to use a *queue-like* implementation to schedule the processes / threads such that maximum # of process / threads run at a given time would match the number of cores. Then schedule the next batch of threads / processes matching the number of cores when the previous batch finishes running. Continue until all batches are done.

Join the results of the threads / processes when all threads / processes finish running.

This is the load balancing that parallel R / Open MPI probably has that my naïve implementation does not.

Part II

Background of this assignment

We parallelize previous methods for computing statistics of large csv methods with goals of:

- comparing different ways of parallelizing the code
- examining the speeding up of the code from different ways of parallelism

Discussion of the previous methods is available at:

http://nbviewer.ipython.org/github/karenyng/HW1_Stat250_Winter14/blob/master/writeup/hw1.ipynb?create=1

1 Possible difficulties / overhead

- handling data locality - avoid passing data around different workers
- overhead of combining data to compute the median

2 Notes for myself - Key concepts of parallelism

Data parallelism

handling multiple inputs concurrently

Task parallelism

if tasks are not dependent on each other, then they can be perform simultaneously with different workers

Data locality

Be careful about how to fork the data. Fine grained functions is hard to parallelize well since a lot of communication about the data will decrease the efficiency. Also have to be aware of how the language scope the functions / variables when sending them from master node to worker nodes

Race condition

Deadlock

3 My idea of how the most efficient algorithm in terms of both speed and memory usage should go:

use a compiled language:

- use threads to read in column data of each file concurrently line by line into shared memory (1st pass) while making a frequency table for the column data for each file
- combine frequency tables (this has to done be sequentially)

The following might not help but would be interesting to see how the runtime scales:

- make two more copies of the combined frequency table assuming the table is small enough that copying is fast
- compute mean, median and standard deviation from the 3 copies of the frequency tables with three threads

3.1 Performance metric of parallelizing code:

(From Scaling Up Machine Learning - Bekkerman R.)

- speedup - ratio of solution time for sequential algorithms vs. its parallel counterpart
- efficiency - measures the ratio of speed up to the # of processors
- scalability - tracks efficiency as a function of an increasing number of processors <- how are the lower two different!?!?!?!?!

Part III

Method 1: Parallelization in python

My code originally runs for ~3 mins. I am parallelizing it because I want to learn how to parallelize python code properly, most of the codes between me and my collaborators are in python (it's a community preference not a personal one).

I mainly make use of the book "Python high performance programming" as my reference for this method.

4 Profiling my python code to see where the speed / memory bottleneck is

Following a really nice tutorial at:

http://pynash.org/2013/03/06/timing-and-profiling.htmlimport a version of the code that is wrapped as a functionrun line profiler on it and dump the output to lpstat.txt

```
from profile_stat import run
%prun -T lpstat.txt run()
```

Educated guess: the reading in of files would take the most amount of time . . .

In [29]:
```
!head -10 lpstat.txt
```

```
         50247 function calls (50245 primitive calls) in 190.815
seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
       81  167.560    2.069  167.560    2.069 {method 'read' of
'pandas._parser.TextReader' objects}
      163   10.553    0.065   10.553    0.065
{numpy.core.multiarray.concatenate}
```

```
        1    2.619    2.619    2.619    2.619 {pandas.algos.median}
        1    1.483    1.483  190.694  190.694 profile_stat.py:8(run)
        4    1.387    0.347    1.387    0.347
common.py:128(_isnull_ndarraylike)
```

If we read the "cumtime" column we can find that most of the time is actually spent on reading the data the csv files. There is also half a minute spent on merging, appending and concatanating files - which is ~15% of the runtime… we may be able to assign different cores to join different data frames together. The reading of files is actually quite embarrassingly parallel but is also limited by the memory. Column data from each file is only around:

In [30]:
```python
print '{0:.0f} MB'.format(2.2 / 81. * 1000)
```

```
27 MB
```

Even if we read 4 at the same time it should only be ~0.1 GB for my quad core desktop with 16 GB of RAM… unless my code is doing something really stupid.

Let's do memory profiling to see if that's true!

```
%mprun -T mpstat.txt -f run run()
```

In [31]:
```
!cat mpstat.txt
```

```
Filename: profile_stat.py

Line #    Mem usage    Increment   Line Contents
================================================
     8    132.0 MiB      0.0 MiB   def run:
     9    132.0 MiB      0.0 MiB       import pandas as pd
    10    132.0 MiB      0.0 MiB       import numpy as np
    11    132.0 MiB      0.0 MiB       data_path = "../data/"
    12
    13                                 # First read in the data from
1987 to 2007
    14    132.0 MiB      0.0 MiB       year = [data_path + str(i) +
".csv" for i in range(1987, 2008)]
    15                                 # create empty dataframe
    16    132.0 MiB      0.0 MiB       delay1 = pd.DataFrame()
    17                                 # loop through the year-by-year
csvs
    18   1993.0 MiB   1861.0 MiB       for yr_file in year:
    19                                     # read in relevant column
from csv file using pandas
    20   1879.3 MiB   -113.7 MiB           temp = pd.read_csv(yr_file,
usecols=["ArrDelay"])
    21                                     # append the dataframes -
this is done by reference not by value
    22   1993.0 MiB    113.7 MiB           delay1 =
delay1.append(temp)
    23   1993.0 MiB      0.0 MiB           print 'appending ' +
yr_file + ' - total lines = ' + \
    24   1993.0 MiB      0.0 MiB
'{0}'.format(delay1.shape[0])
    25
    26                                 # create another empty
```

```
dataframe for handling month by month csv
    27   1993.0 MiB      0.0 MiB       delay2 = pd.DataFrame()
    28   1993.0 MiB      0.0 MiB       month = ['January', 'February',
'March', 'April', 'May', 'June', 'July',
    29   1993.0 MiB      0.0 MiB               'August', 'September',
'October', 'November', 'December']
    30   1993.0 MiB      0.0 MiB       year = [data_path + str(i) +
"_" +
    31   1993.0 MiB      0.0 MiB               mth + ".csv" for i in
range(2008, 2013) for mth in month]
    32                                 # loop through all the month-
by-month csv
    33   2399.7 MiB    406.7 MiB       for yr_file in year:
    34                                     # tell pandas to read only
the relevant column in the csv
    35   2392.2 MiB     -7.5 MiB           temp = pd.read_csv(yr_file,
usecols=["ARR_DELAY"])
    36                                     # append them to the
dataframe by reference
    37   2399.7 MiB      7.5 MiB           delay2 =
delay2.append(temp)
    38   2399.7 MiB      0.0 MiB           print 'appending ' +
yr_file + ' - total lines = ' + \
    39   2399.7 MiB      0.0 MiB
'{0}'.format(delay2.shape[0] + delay1.shape[0])
    40
    41                                 # hackish way to remove the
column name of the dataframe to append
    42                                 # the two types of csv columns
together
    43                                 # so I can compute the
statistics in one pass later on
    44   1510.7 MiB   -889.0 MiB       delay1 = np.array(delay1)
    45   1265.8 MiB   -244.8 MiB       delay2 = np.array(delay2)
    46   2399.7 MiB   1133.9 MiB       delay = np.append(delay1,
delay2)
    47   3533.6 MiB   1133.9 MiB       delay = pd.DataFrame(delay)
    48   3533.6 MiB      0.0 MiB       print 'total number of valid
lines = {0}'.format(delay.dropna().shape[0])
    49
    50                                 # note that pandas ignores nans
automatically while computing stats
    51   3533.6 MiB      0.0 MiB       print 'saving to results2.txt'
    52   3533.6 MiB      0.0 MiB       f = open('results2.txt', 'w')
    53   3533.6 MiB      0.0 MiB       f.write('mean =
{0}\n'.format(delay[0].mean()))
    54   3533.6 MiB      0.0 MiB       f.write('median =
{0}\n'.format(delay[0].median()))
    55   3533.6 MiB      0.0 MiB       f.write('std =
{0}\n'.format(delay[0].std()))
    56   3533.6 MiB     -0.0 MiB       f.close()
```

So the python memory profiler was reporting back that only ~3.5 GB of memory was used. Not sure if that is really true since the C backend of pandas might be by-passing the python intrepretter for memory usage. So I used valgrind to measure both the stack and heap usage. The memory use is consistent with what the python profiler found. The valgrind profiling results are stored in

5 Parallelize the bottleneck - different approaches within python

5.1 multi-threading (processing) in python?

Python has a global interpretted lock (GIL) which prevents more than one thread from being run at a time under usual circumstances (it's implemented like a mutex). Python users are advised to use multiple processes via the multiprocessing module

http://docs.python.org/dev/library/multiprocessing.html

that is part of the standard python library instead. The difference between the two is that memory aren't shared naturally between multiple processes in the case of multiple threads. Not sure if that is going to impact us here.

I tried using the default multiprocessing module but it does not like functions with optional arguments. Writing a wrapper function to parse the optional arguments also did not work properly.

5.2 parallel python packages

There are quite a number of powerful packages that can handle either parallelization on multiple core machine or over a distributed network of machines.

https://wiki.python.org/moin/ParallelProcessing

In this assignment, I use the "parallel python" package (pp). The package seems quite comprehensive, it supports multicore processing, distributed computing over a cluster etc. But after some use, I do not actually like this package, it has the worst (close to non-existent) python package documentation I have seen.

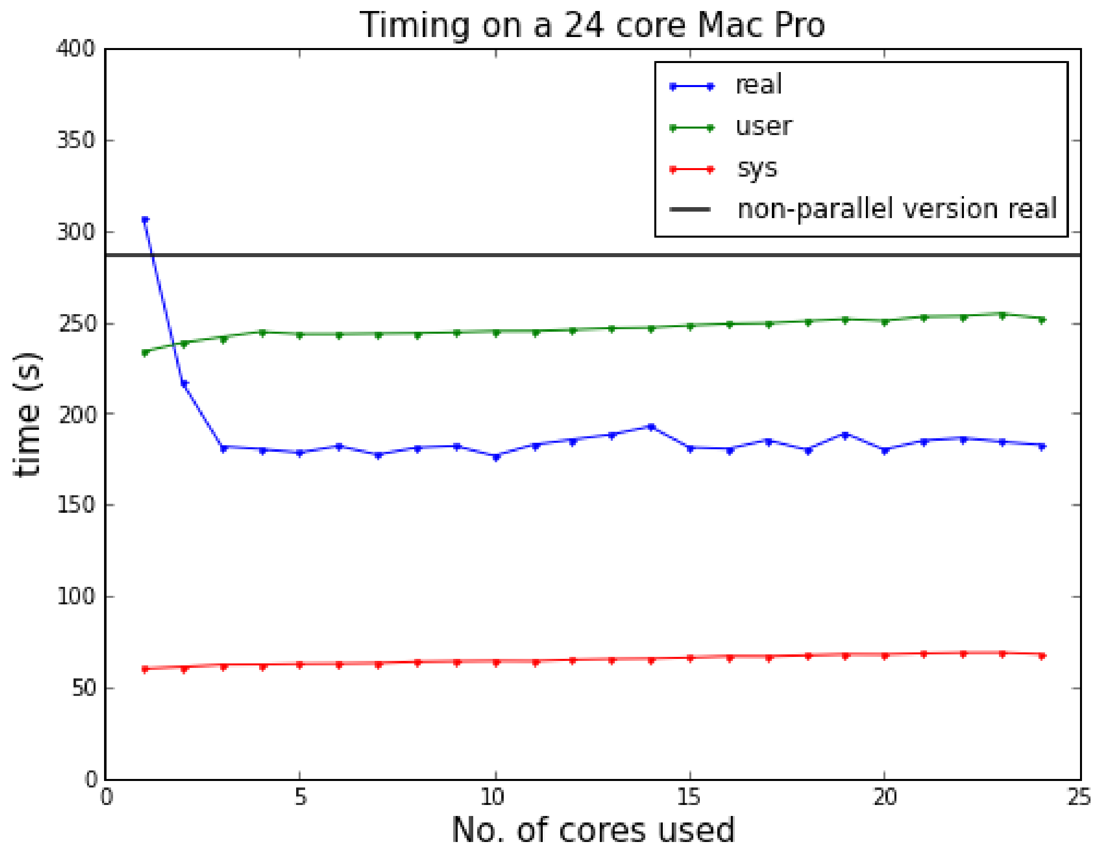
6 Overall timings

In [32]:
```
from parse_time import parse_time as parse_t

df = parse_t()
non_parallel_t = 4. * 60. + 46.597

fig = plt.figure(figsize = (8,6))
ax = fig.add_subplot(111)
ax.plot(range(1, 25), df["real"], ".-", label="real")
ax.plot(range(1, 25), df["user"], ".-", label="user")
ax.plot(range(1, 25), df["sys"], ".-", label="sys")
ax.axhline(non_parallel_t, label = "non-parallel version real",
           color = "k", linewidth = 1.5)
ax.set_xlabel("No. of cores used", size = 15)
ax.set_ylabel("time (s)", size = 15)
ax.legend(loc='best')
ax.set_title('Timing on a 24 core Mac Pro',size=15)
ax.set_ylim((0,400))
```

Out [32]:
```
(0, 400)
```



This plot tells us that my code does not scale well. The parallelized version has overhead of starting job and the speed up is marginal. Even though most of my research python codes (scipy, numpy, astropy) use fast C code as backend but the thought of my code not being able to scale is a pain.

7 Speed up

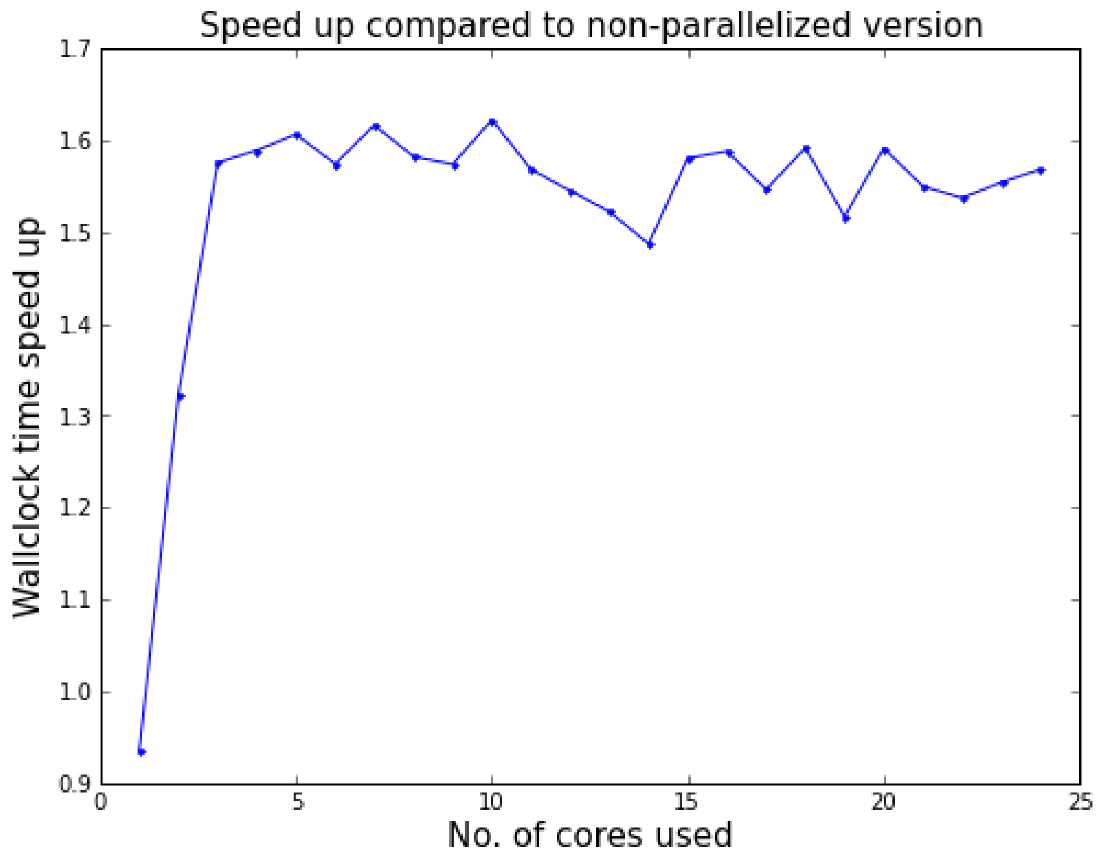
7.1 Speed up based on amount of data processed

too lazy to do but would be interesting to find out

7.2 Speed up using multiple cores

In [33]:
```
fig = plt.figure(figsize = (8,6))
ax = fig.add_subplot(111)
ax.plot(range(1, 25), non_parallel_t / df["real"], ".-")
ax.set_xlabel("No. of cores used", size = 15)
ax.set_ylabel("Wallclock time speed up ", size = 15)
ax.set_title('Speed up compared to non-parallelized version ', size=15)
```

Out [33]: <matplotlib.text.Text at 0x69f1850>



It just would not scale for more than 3 cores !!!

8 Profiling of how long each part of the computations took and intrepretation

8.1 Timing and memory profile of parallelized code using the parallel python package on a 24-core machine

Ok let's parse back the output of the "time" command from the shell.

```
from profile_method1 import run
%prun -T lpstat_parallelized_m1.txt run()
```

If I am not lazy I would convert the following text file into a plot

In [34]:
```
!head -10 lpstat_parallelized_m1.txt
```

```
         24731 function calls (24730 primitive calls) in 244.903
seconds

   Ordered by: internal time
```

```
   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
      450  223.675    0.497  223.675    0.497 {method 'acquire' of
'thread.lock' objects}
      164   11.006    0.067   11.006    0.067
{numpy.core.multiarray.concatenate}
        1    2.620    2.620    2.620    2.620 {pandas.algos.median}
        1    1.201    1.201  244.566  244.566
profile_method1.py:1(run)
       81    1.163    0.014    1.519    0.019 {cPickle.loads}
```

The rest of the file does not have any more process that takes up a big chunk of computing time. So the Python Global Interpreter Lock (GIL) is really messing things up here.

8.2 Profiling of a version that uses threads in python (...... can't believe I actually tried this after knowing about the GIL)

```
from profile_method1a import run
%prun -T lpstat_parallelized_m1a.txt run()
```

If I am not lazy I would convert the following text file into a plot

In [35]:
```
!head -10 lpstat_parallelized_m1a.txt
```

```
         4673 function calls in 249.731 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
      494  161.209    0.326  161.209    0.326 {method 'acquire' of
'thread.lock' objects}
       81   44.126    0.545  155.997    1.926 threading.py:726(start)
       81   22.865    0.282   22.865    0.282
{numpy.core.multiarray.concatenate}
        1    5.846    5.846  249.630  249.630
profile_method1a.py:22(run)
        1    5.201    5.201    5.201    5.201 {method 'sort' of
'numpy.ndarray' objects}
```

It gets even slower..... The overhead for starting the threads and thread locks are totally slowing things down !!! I am not quite sure if the thread locks are actually due to:

1. GIL,
2. stupid handling of race conditions from the parallel package which I do not have any control over
3. the memory bus / hard disk being a speed bottle neck

Number 1 and 2 seem more likely than 3 but 3 can be a good guess. I actually found an interesting article online at:

http://www.drdobbs.com/parallel/multithreaded-file-io/220300055?pgno=2

about how normal desktop file systems have limitations for using threads for I/O. The results from the article does not explain why my code does not scale but it is nonetheless interesting to know that there are other file systems designed for multi-threaded I/O from disk:

In [36]:
```python
from IPython.display import Image
i = Image(url="http://twimgs.com/ddj/images/" + \
          "article/2009/0909/090928wuth5_fg2a.gif")
i
```

Out [36]: <IPython.core.display.Image at 0x4984150>

where from web definition

> SCSI is especially designed as a parallel interface for I/O from hard disk
>
> RAID-5 The standard RAID levels are a basic set of RAID configurations that employ the techniques of striping, mirroring, or parity to create large reliable data stores from general purpose computer hard disk drives. The most common types today are RAID 0, RAID 1 and variants, RAID 5 and RAID 6. ...

Performance actually drops after more than 8 threads are used for a normal hard disk.

Of course if I/O is a huge issue there is also distributed file system like Hadoop which has better fault tolerance.

9 Verification of results

I ran the code and asked it to automatically adjust the # of cores and output the mean, median and std. dev. to different files called pp_{NUMBER_OF_CORES}.txt. Ran a shell loop of "diff" on those files and they agree:

In [37]:
```
!../tests/for_diff.sh
```

The lack of output indicates they are the same.

In [38]:
```
!cat ../tests/for_diff.sh
```

```
#!/bin/bash

for N in $(seq 24)
do
        diff "../writeup/runtime/pp_1.txt"
"../writeup/runtime/pp_"${N}".txt"
done
```

10 Not all hope is lost for python:

10.1 bypassing python GIL method 1.

wrap Duncan 's C code with Cython to do the I/O part. Thanks Duncan.

10.2 bypassing python GIL method 2. - use other existing python libraries with C backend

I came across

http://www.pytables.org/docs/LargeDataAnalysis.pdf and http://blosc.pytables.org/trac that talks about data compression technologies and related packages (PyTables, HDF5, BloSC) for dealing with memory bound / memory gap problems in python. Those libraries use C as a backend and avoid the GIL problem for I/O.

Potential drawback:

- overhead for using PyTables / HDF5 / BLOSC, we have to first compress ASCII files into HDF5 (binary) format

Part IV

Method 2 - AirlineDelay package

I could use of Duncan 's AirlineDelay package since he did most of the hardwork of putting it together. Remaining steps include:

- unit test the code
- run valgrind to make sure there is no memory leak!
- add code for handling the month-by-month csv files
- run the sequential version first to make sure everything work???
- run the threaded version - test if it returns the values correctly
- do global test...
- implement a time-profiler for profiling each part of the code
- test runtime for different number of files (threads) / cores used

11 Summary of fixes that I implemented in the AirlineDelay package:

1. wrote function to catch problematic inputs, this includes error handling for :
   - problematic input file paths
   - number of files < number of threads
2. catch all NaN values that matches "NA" or empty string ""
3. change return type of readRecord function in order to detect NAN correctly, it seems impossible to return NAN from a C routine without it being converted to be 0
   - I implemented a version that does not require calling NAN values since NAN might not be defined for all C compilers
4. fix the off-by-1-count bug from the merge table function
   - the code initially would join the first table to itself in the for loop
   - have to change i=0 to i=1 to for starting the for loop to prevent the first table being joined to itself
5. documentations about the functions, learned how to run roxygen2 to generate *.Rd files
   - it bugs me that there is no proper documentation for the R code since I am so used to documenting all inputs and outputs for functions of intrepretted languages which are not type safe
6. global test code in

12 Overall timing for quad core machine 1 :

4 threads (files) per core on average: real time 3.443 s (this is very fast actually)

8 threads (files) per core on average: real time 49 s (hyperthreading is slower)These timings look good but if the overall timing suck if I naïvely just start 81 threads for all 81 files on quad core machine. I cannot be more amused when

the runtime went from the sequential version of 3 min to my python parallel version of 3+ min to the multithreaded version of 13 mins. This is the runtime when run on quad core machine:

real 13m18.151s

user 0m52.840s

sys 0m4.772s

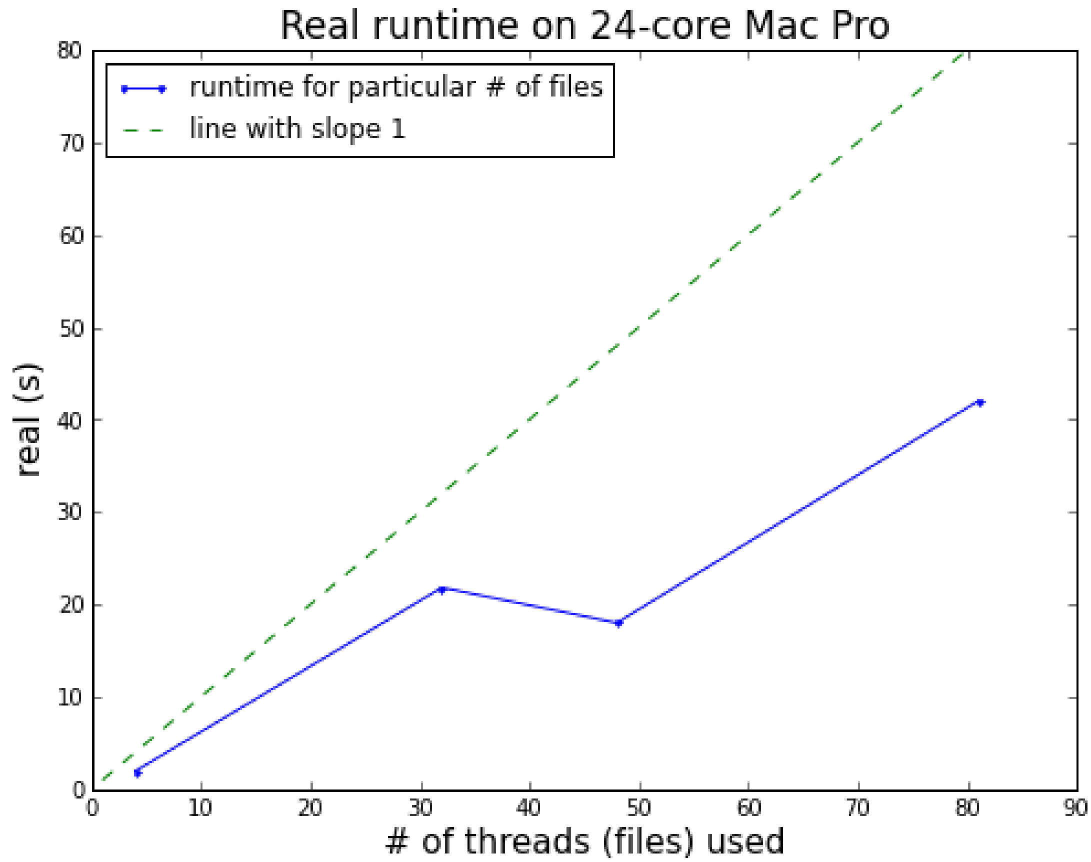
This is not that surprising given the figure of read speed vs # of thread figure that I posted above.

12.1 Overall timing on 24-core machine :

In [39]:
```
from __future__ import division
files = np.array([4, 32, 48, 81] )
real = np.array([1.961, 21.815, 18, 41.996]) # in seconds
compare = np.arange(1,81)
time_per_file = real / files

fig = plt.figure(figsize = (8,6))
ax = fig.add_subplot(111)
ax.plot(files, real, '.-', label = 'runtime for particular # of files')
ax.plot(compare, compare, '--', label = 'line with slope 1')
ax.legend(loc = 'best')
ax.set_xlabel("# of threads (files) used", size = 15)
ax.set_ylabel("real (s)", size = 15)
ax.set_title("Real runtime on 24-core Mac Pro", size = 17)
```

Out [39]:
```
<matplotlib.text.Text at 0x6757bd0>
```

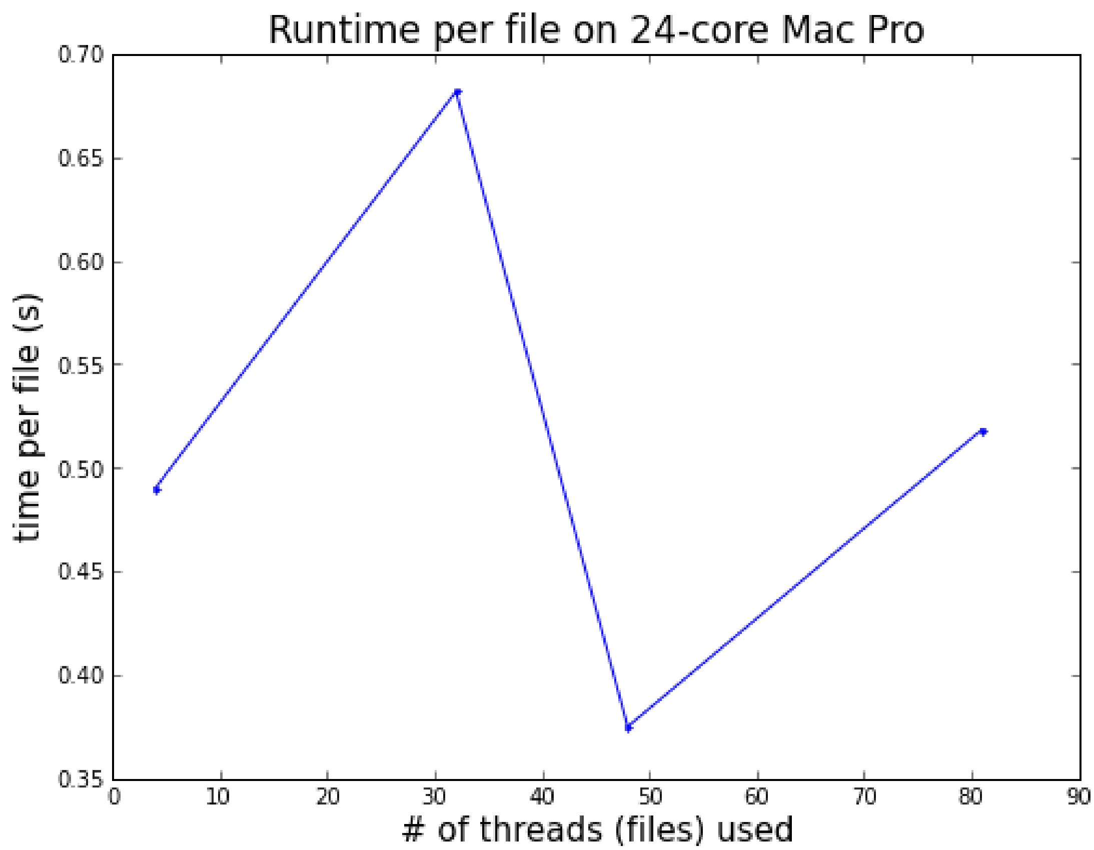


If I am not lazy I would use # of lines on the x-axis instead to make it a fairer comparison.

In [40]:
```
fig = plt.figure(figsize = (8,6))
ax = fig.add_subplot(111)
ax.plot(files, time_per_file, '.-')
ax.set_xlabel("# of threads (files) used", size = 15)
```

```
ax.set_ylabel("time per file (s)", size = 15)
ax.set_title("Runtime per file on 24-core Mac Pro", size = 17)
```

Out [40]: <matplotlib.text.Text at 0x6f81d10>



Good that it the runtime per file is approximately constant.

12.2 Speed up from sequential python code: 6.8x

In [41]:
```
(4. * 60. + 46.597 )/  41.996
```

Out [41]: 6.8243880369559

which is really a marginal speed up per additional core . . .

13 Pending modification to C code for realizing the promise of huge speed up with multithreads:

- have numThreads = numFiles
- start # of threads = # of core at one time
- wait for the those threads to finish
- start more threads that matches the # of core, then wait etc. until all of numThreads are finished

- need to modify the C code to do this
  - can use a queue to store the file names to start the threads

14 Verification of results

> "mean = 6.56650421703496"
>
> "median = 0"
>
> "std dev. = 31.5563262622735"

This method agrees up to all the printable digits from previous methods for all 81 filesThis also passes the global tests that I wrote for comparing thet statistics for individual files.

15 Memory usage:

I am alarmed that some 7.440 MB of swap memory was used! Really not sure why because at a given time only a small % of memory is shown up to be used from the output of the "top" command.

And of course if I am not lazy about doing profiling, this can be explained. . . . . .

Part V

Logistics

16 Dependencies:

- put csv files in ${GIT_REPOSITORY}/data/
- install the R package inside Git repository called AirlineDelays

17 How to run the code

17.1 Method 1

For running the python code specifying a certain number of cores, at a terminal run:

```
>./method1.py ${NUMBER_OF_CORES_TO_USE}
```

For running the python code for a range of cores, modify method1_wrapper.py then run:

```
>./method1_wrapper.py
```

17.2 Method 2

call script from within the git repository

```
>Rscript
```

18 Machine Specifications

18.1 Machine 1

- Corsair Vengeance 2x8GB DDR3 1600 MHz Desktop Memory,
- Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
- GeForce GTX 770 SuperClocked
- Samsung 840 Pro 256 GB SSD
- Western Digital HDD 2TB Intellipower adjustable RPM 5400 - 7200
- Motherboard: Asus Z87-Deluxe DDR3 1600 LGA 1150
- Linux Mint 15 Olivia (GNU/Linux 3.8.0-19-generic x86_64)

18.2 Software package dependencies on machine 1

Python package dependencies:

- numpy 1.7.1
- pandas 0.10.1
- pp 1.6.4

R - 3.0.2

gcc (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3

18.3 Machine 2

- Mac Pro 5.1
- Processor Name: Intel 6-Core Xeon @ 2.66 GHz
- Number of Processors: 2
- Total Number of Cores per processor: 12
- L2 Cache (per Core): 256 KB
- L3 Cache (per Processor): 12 MB
- Memory: 24 GB
- Processor Interconnect Speed: 6.4 GT/s
- OSX Maverick 10.9.1

18.4 Software package dependencies on machine 2

Python package dependencies:

- numpy 1.8.0
- pandas 0.13.0
- pp 1.6.4 (not the best python package!!!)

R - 3.0.2

Apple LLVM version 5.0 (clang-500.2.79) (based on LLVM 3.3svn)

Configured with: –prefix=/Applications/Xcode.app/Contents/Developer/usr

–with-gxx-include-dir=/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.9.sdk/

Target: x86_64-apple-darwin13.0.0

Thread model: posix

19 Actual code

19.1 Method 1: parallelizing python with pp

In [42]:
```
!cat ../method1.py
```

```
#!/usr/bin/env python
import numpy as np
import pp
import sys

data_path = "../data/"
max_cpus = 24

# catch exception
assert len(sys.argv) == 2, "Requires user to specify number of cpu " +
\
    "used for parallelization"
assert int(sys.argv[1]) <= max_cpus, "Max number of cpu used = 24" + \
                " problematic input = ".format(argsv[0])

# have the job server use the specified number of CPU!
ncpus = int(sys.argv[1])
job_server = pp.Server(ncpus=ncpus)


def kludgy_read_csv_wrapper1(files):
    import pandas as pd
    import numpy as np
    return np.array(pd.read_csv(files, usecols=["ArrDelay"]))


def kludgy_read_csv_wrapper2(files):
    import numpy as np
    import pandas as pd
    return np.array(pd.read_csv(files, usecols=["ARR_DELAY"]))

#yr_by_yr_files = [data_path + "1987.csv"]
yr_by_yr_files = [data_path + str(i) + ".csv" for i in range(1987,
2008)]
month = ["January", "February", "March", "April", "May", "June",
         "July", "August", "September", "October", "November",
         "December"]
mnth_by_mnth_files = \
    [data_path + str(yr) + "_" + mnth +
     ".csv" for yr in range(2008, 2013) for mnth in month]

##print "going to read in the following # of files:" + \
##    " {0}".format(len(yr_by_yr_files) + len(mnth_by_mnth_files))
```

```
# looping in python
jobs1 = [(input1,
          job_server.submit(kludgy_read_csv_wrapper1,
                            (input1,))) for input1 in yr_by_yr_files]

jobs2 = [(input2,
          job_server.submit(kludgy_read_csv_wrapper2,
                            (input2,))) for input2 in
mnth_by_mnth_files]

# wait for all jobs to finish
job_server.wait()
delay = np.array([])
for input, job1 in jobs1:
    delay = np.append(delay, job1())

for input2, job2 in jobs2:
    delay = np.append(delay,job2())

# one can parallelize the following but it's not worth the overhead
delay = delay[~np.isnan(delay)]
mean = np.mean(delay)
sd = np.std(delay)
median = np.median(delay)
#print "number of valid lines {0}".format(delay.size())

f = open("pp_{0}.txt".format(ncpus), "w")
f.write('mean = {0:1.7}\n'.format(mean))
f.write('median = {0:1.7}\n'.format(median))
f.write('std = {0:1.7}\n'.format(sd))
f.close()
```

Wrapper for calling my python code with a range of cpu processors:

In [43]:
```
!cat ../method1_wrapper.py
```

```
#!/usr/bin/env python
import os

for i in range(20,24):
    os.system("(time ./method1.py {0}) 2>> time_cpus.txt".format(i))
```

19.2 Version 2 with python threads (that failed miserably)

In [44]:
```
!cat ../profile_method1/profile_method1a.py
```

```
import pandas as pd
import numpy as np
import thread
import threading
```

```
class myThread(threading.Thread):
    def __init__(self, threadID, filename, colname):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.filename = filename
        self.colname = colname
    def run(self):
        print "reading " + self.filename
        self.data = np.array(pd.read_csv(self.filename,
                                         usecols=self.colname))


def run():
    data_path = "../data/"
    yr_by_yr_files =\
        [data_path + str(i) + ".csv" for i in range(1987, 2008)]
    month = ["January", "February", "March", "April", "May", "June",
             "July", "August", "September", "October", "November",
             "December"]
    mnth_by_mnth_files =         [data_path + str(yr) + "_" + mnth +
            ".csv" for yr in range(2008, 2013) for mnth in month]
    print "going to read in the following # of files:" +         "
{0}".format(len(yr_by_yr_files) + len(mnth_by_mnth_files))


# Out[3]:

#     going to read in the following # of files: 81
#

# In[24]:

    threads = []
    threadID = 1
    for yr_f in yr_by_yr_files:
        thread = myThread(threadID, yr_f, ["ArrDelay"])
        thread.start()
        threads.append(thread)
        threadID += 1
    for mth_f in mnth_by_mnth_files:
        thread = myThread(threadID, mth_f, ["ARR_DELAY"])
        thread.start()
        threads.append(thread)
        threadID += 1

    data = np.array([])


    for t in threads:
        t.join()
        data = np.append(data, t.data)

    mean = np.mean(data[~np.isnan(data)])
    median = np.median(data[~np.isnan(data)])
```

```
    sd = np.std(data[~np.isnan(data)])

    print mean
    print median
    print sd
```

19.3 Method 2: AirlineDelay package

I will not post all the C code here. The summary of the modifications that I made are summarized above.

In [45]:
```
!sed -n  '83,183p' ../AirlineDelays/R/getDelayFreqTable.R
```

```
#' @name getListOfFiles
#' @title return a list of filenames
#' @param filepath
#'   string that contains the path to the directory containing the
files
#' @param  pattern
#'   string that denotes the regular expression for matching file
names
#' @param full.names
#' @return FILES
#'   R list of filenames
#' @seealso \code{\link[base]{list.files}}
#' @export
getListOfFiles =
function(filepath, pattern = NULL, full.names = TRUE)
{
  # list all the files in the relevant diretory
  files = list.files(filepath, pattern = pattern, full.names = TRUE)
  if(length(files) == 0)
  {
    stop(paste("Failed to read in files at", filepath))
  }

  # function that Duncan wrote only likes lists
  # so have to rearrange the file paths as lists
  FILES <- list()
  for(i in 1:length(files))
  {
    FILES <- append(FILES, list(files[i]))
  }
  FILES
}


#' @name checkInputsForErrors
#' @title check for input errors
#' @param FILES
#'   R list of files
#' @param numCores
#'  an integer that denotes the number of cores to be used
```

```
#' @note this suppresses a possible memory error
#' @return numCores
#'  an integer that denotes numCore that will not cause memory error
#' @export
checkInputsForErrors =
function(FILES, numCores)
{
 if(length(FILES) < as.integer(numCores)){
   print("Number of files supplied < number of threads!!")
   print("setting numCores = number of files")
   numCores <- length(FILES)
 }
 if(length(FILES) > as.integer(numCores)){
   stop("Number of files supplied > number of threads! \n
        Increase the number of threads via numCores!")

 }
 numCores
}


#' @name freq_mean
#' @title compute mean from frequency table
#' @param tt
#'   vector with count as field value, column name as delay
#' @param w.total
#'   integer denoting total frequency count
#' @return list of
#'   total freq. count, mean
#' @export
freq_mean =
function(tt)
{
  print(tt)
  df <- as.data.frame(tt)
  # store them as double to avoid numerical instabilities
  delay <- sapply(rownames(df), as.double)
  w.total <- sum(df[,c('tt')])
  # would there be overflow? or underflow for the following line?
  t.mean <- sum(df[,c('tt')] * ( delay / w.total), na.rm = TRUE)

  c(w.total, t.mean)
}


#' @name freq_median
#' @title compute median from frequency table
#' @param w.total
#'   integer denoting total frequency count
#' @param tt
#'   vector with count as field value, column name as delay
#' @return median
#'   double
#' @export
```

```
freq_median =
function(w.total, tt)
{
  i <- 1
  df <- as.data.frame(tt)
  delay <- sapply(rownames(df), as.double)
  Sum <- df[['tt']][i]
```

In [46]:
```
!sed -n '184,242p' ../AirlineDelays/R/getDelayFreqTable.R
```

```
  medianFreqCount <- floor(w.total / 2)
  ## sorry don't know better than to write a loop...
  while(Sum < medianFreqCount) {
    i <- i + 1
    # this vectorized operation
    Sum <- sum(df[['tt']][1:i], na.rm = TRUE)
    # is faster than
    ## if ( !is.na(DF[['freq']][i]) ) {
    ##   Sum <- Sum + DF[['freq']][i]
    ## }
  }

  ## check for corner case:
  ## or else there the median will may be off
  if( Sum == medianFreqCount &&  w.total %% 2 == 0){
    #print("going through special case")
    t.median <- (delay[i] + delay[i+1])/2
  }else{
    t.median <- delay[i]
  }
  t.median
}

#' @name freq_sd
#' @title compute sd from frequency table
#' @param t.mean
#'   double denoting the mean
#' @param w.total
#'   integer denoting total frequency count
#' @param tt
#'   vector with count as field value, column name as delay
#' @return median
#'   double
#' @export
freq_sd =
function(t.mean, tt, w.total)
{
  df <- as.data.frame(tt)
  delay <- sapply(rownames(df), as.double)
  std.dev <- sqrt(sum(df[,c('tt')] * (delay - t.mean) ^ 2 /
(w.total-1)))
}
```

19.4 code for global test of individual files:

In [47]:
```
!cat ../AirlineDelays/tests/test_stat.R
```

```
require(AirlineDelays)
filepath <- "/mnt/Winter14Stat250/HW2_Stat250_W14/data/"
pattern <- "^2008_May.csv$"
filename <- "2008_May.csv"
datapath <- paste(filepath, filename, sep='')
#wantedCol <- 15L
wantedCol <- 43L
tolerance <- 1e-12

getStat =
function(datapath, wantedCol)
{
  # just wanna grab a particular column
  #numLines <- getNumLines(datapath)
  df <- read.csv(datapath, nrow = 1)
  colNum <- length(colnames(df))
  colClasses <- vector("list", colNum)
  colClasses[[wantedCol]] <- "numeric"

  df <- read.csv(datapath, colClasses = colClasses, fill = T)#, nrow =
numLines)
  colName <- colnames(df)
  mean_ans <- mean(df[,c(colName[[1]])], na.rm = T)
  median_ans <- median(df[,c(colName[[1]])], na.rm = T)
  sd_ans <- sd(df[,c(colName[[1]])], na.rm = T)
  #print(paste("total sum from read.csv=",sum(df[,c(colName[[1]])],
na.rm =
  #                                           T)))
  c(mean_ans, median_ans, sd_ans)
}
stat_ans <- getStat(datapath, wantedCol)
print(stat_ans)



# this numCore variable should be input from the console instead

FILES <- getListOfFiles(filepath, pattern = pattern)
print(paste("Setting numThreads = number of files input =",
length(FILES)))
tt <- getDelayTable_thread(FILES, numThreads = length(FILES))
ans <- freq_mean(tt)
ad_mean <- ans[[2]]
ad_std <- freq_sd(ans[[2]], tt, ans[[1]])
ad_median <- freq_median(ans[[1]], tt)

#sprintf("%1.10f", ad_mean, stat_ans[[1]])
#sprintf("%1.10f %1.10f", ad_mean, stat_ans[[1]])
#sprintf("%1.10f %1.10f", ad_median, stat_ans[[2]])
```

```
#sprintf("%1.10f %1.10f", ad_std, stat_ans[[3]])

stopifnot(ad_mean - stat_ans[[1]] < tolerance)
stopifnot(ad_median[[1]] -  stat_ans[[2]] < tolerance)
stopifnot(ad_std -  stat_ans[[3]] < tolerance)

print(paste("Passed global test with tolerance of stat precision =",
tolerance))
```

Part VI

References:

[1] http://cyrille.rossant.net/profiling-and-optimizing-python-code/