WP / 23 / 93

GTAP Working Paper No. 93

Using Python for Parallelization

by Dominique van der Mensbrugghe

Global Trade Analysis Project
Center for Global Trade Analysis
Department of Agricultural Economics
Purdue University

First edition: May 2023

Please cite using the following reference:
van der Mensbrugghe, Dominique, 2023.
Using Python for Parallelization
GTAP Working Paper, WP/23/93
Center for Global Trade Analysis, Purdue University
West Lafayette, IN
https://www.gtap.agecon.purdue.edu/resources/res_display.asp?RecordID=6826

Center for Global Trade Analysis
Department of Agricultural Economics
Purdue University
West Lafayette, IN 47907
https://www.gtap.agecon.purdue.edu

Using Python for Parallelization

Dominique van der Mensbrugghe[1]

May 8, 2023

Abstract: This short note describes one way of taking advantage of the multiple cores on most desktop computers. It describes running one of the processes in the GTAP build procedure called 'FIT'. The input to 'FIT' is a balanced input-output table (IOT), which is adjusted to a number of exogenous elements including aggregate domestic absorption and import and export vectors. It is run for each of the countries/regions in the build, but there is no interaction across countries/regions and thus can be run in parallel. The procedure uses a Python script to run the 'FIT' procedure, either sequentially or in parallel. Most of the code is generic and thus can be easily adapted to other programs that can take advantage of parallelism, for example Monte Carlo simulations. For the tested 'FIT' procedure, it reduces the runtime from 75 minutes to 14 minutes on a relatively new desktop with a 12th Generation Intel Core I-9 CPU with 16 physical cores.

JEL Codes: C15, C53, C63
Keywords: Parallel processing, Monte Carlo, Maximum Entropy, GEMPACK, GAMS

[1] The Center for Global Trade Analysis, Purdue University, 403 Mitch Daniels Boulevard, West Lafayette, IN 47906. Corresponding e-mail: vandermd@purdue.edu.

Introduction

This short note describes one way of taking advantage of the multiple cores on most desktop computers. It describes running one of the processes in the GTAP build procedure called 'FIT'. The input to 'FIT' is a balanced input-output table (IOT), which is adjusted to a number of exogenous elements including aggregate domestic absorption and import and export vectors. It is run for each of the countries/regions in the build, but there is no interaction across countries/regions and thus can be run in parallel. The procedure uses a Python script to run the 'FIT' procedure, either sequentially or in parallel. Most of the code is generic and thus can be easily adapted to other programs that can take advantage of parallelism, for example Monte Carlo simulations. For the tested 'FIT' procedure, it reduces the runtime from 75 minutes to 14 minutes on a relatively new desktop with a 12^{th} Generation Intel Core I-9 CPU with 16 physical cores.[1]

In some sense this note is a follow-up to an earlier paper in the Journal Global Economic Analysis (Villoria (2017)) that highlighted the implementation of parallel processing to run Monte Carlo simulations with the GTAP model. In that article, R was used as the underlying scripting language. It also complements GEMPACK's internal parallel processing described in Horridge and Peason (2006).[2] The appendix includes a similar Python-based script that was used to run a suite of Monte Carlo simulations using GAMS—highlighting the script's potential in various domains.

Description of the code

The full code is provided in Listing 1. The preliminaries are on lines 1–26. There are the usual packages to import, lines 1–7. The ones specific to parallel processing are `subprocess` and `multiprocessing`. The code will set the number of processes to start equal to the number of physical cores, lines 9–12. Arguments specific to the 'FIT' procedure are provided on lines 16–23. Most relate to the folder locations of the input and output files, as well as the current working directory. The names of the countries/regions are contained in a simple text file, read in with the Pandas `read_csv` function. These are converted to a Python list called `TASKS`, which will be the core of the loop to process all of the countries/regions.[3] In the case of Monte Carlo simulations—the list `TASKS`—will be linked to the sample draw. If the draw is of size 10,000, `TASKS` will be of size 10,000 with labels consistent with the simulation code.

The main body of the program is in lines 125–161. If only running the parallel part of the program, the main program is very succinct—it simply calls the function `runQueue`, which runs all of the procedures using the parallel processing queuing mechanism. The rest of the main body allows running the entire process in sequential mode—and is used only for benchmarking purposes.

The function `runQueue` will not be described in any detail, leaving that up to the reader—suffice is to say that the only connection with the rest of the code is the use of the `TASKS` list for looping over all of relevant items—countries/regions in this case—and the pointer to the function, called `worker`, which is the user's responsibility to define for the task at hand.

Much of the effort of the programmer is thus to define the `worker` function, lines 53–72. It can be very succinct depending on which process will be called. Running GEMPACK programs requires some extra effort as GEMPACK code has to be run in a separate folder to avoid clashes. The `setup` function, lines 28–40 does much of the preliminary work. For each country/region it will create a temporary folder using the code of the country/region (line 32). It then copies two key input files from the working directory to the temporary directory (lines 33–34). The GEMPACK executables do not appear to be able to be called directly (with arguments) using Python's `subprocess` function. So the code creates a short batch file with the appropriate command and arguments (lines 36–40). The key function in the `worker` function is `subprocess.run`, line 64. The arguments are the name of the batch file, created in the `setup` function, and the relevant country/region code. The flags have two functions. The first, `CREATE_NEW_CONSOLE`, opens a console window in which the program will run. The other, `startupinfo`, is intended to avoid making the new console window the focus

[1] The author thanks Steven Dirkse from GAMS Corporation for the original Python code.

[2] See also https://www.copsmodels.com/parallel.htm.

[3] For debugging purposes, there is a much smaller set of countries in the text file labeled `iso.txt`.

of attention.[4] The `subprocess.run` returns a return code from the executed program which will be used to create a 'good' and a 'failed' lists, the latter of which will be displayed in the Python console when done. The function `cleanup`, lines 42–47, essentially removes the temporary folder.[5]

The remainder of the code in the main body of the program replicates essentially the `worker` function—but is used in the case of running all of the programs sequentially rather than in parallel.

Summary

This brief note introduces a useful Python script to run processes in parallel, taking advantage of the huge capabilities of modern desktop computers. It has been illustrated here for the case of the 'FIT' procedure, which is part of the GTAP build process, but can be readily adopted to many other uses, such as Monte Carlo simulations.

[4] Without this flag, the focal point, under Windows, is shifting constantly as console windows are created and destroyed.

[5] The instruction to remove the folder is in a conditional statement. This is for debugging purposes as it can be useful to see which files are in the subdirectory after execution.

Listing 1: Running 'FIT' in parallel using Python

```
import os
import shutil
import psutil
import time
import pandas as pd
import subprocess
from multiprocessing import Process, Queue, freeze_support

print("This computer has "+str(psutil.cpu_count(logical = False))+" physical CPU cores")
print("This computer has "+str(psutil.cpu_count(logical = True))+" logical CPU cores")

nProc   = psutil.cpu_count(logical = False)

#   User must set these variables

wDir = os.getcwd()
iDir = "Z:\\Output\\Build\FIT\\IN\\"
sDir = "Z:\\Output\\Build\FIT\\DMP\\"
oDir = "Z:\\Output\\Build\FIT\\OUT\\"

# Get the list of countries/regions
reg = pd.read_csv("regListV11_2017.txt",header=None,index_col=None)
# reg = pd.read_csv("iso.txt",header=None,index_col=None)

nReg = reg.shape[0]
TASKS = ['' + str(reg.loc[i][0]) for i in range(0,nReg)]

def setup(reg):
    #   Creates temporary directory for each run
    tDir = wDir+"/"+reg
    if(not os.path.exists(tDir)):
        os.mkdir(tDir)
        shutil.copy(wDir+"/"+"ISAM.HAR", tDir)
        shutil.copy(wDir+"/"+"FITNEW.cmf", tDir)
        #   Create the batch file
        filename = tDir+'/runFIT.cmd'
        print(filename)
        with open(tDir+'/runFIT.cmd','w') as f:
            f.write('..\\fitnew -cmf fitnew.cmf -p1='+reg+' -p2='+iDir+' -p3='+sDir+' -p4='+oDir)
        f.close()

def cleanup(reg):
    #   Delete temporary directory
    tDir = wDir+"/"+reg
    if True:
        if(os.path.exists(tDir)):
            shutil.rmtree(tDir)

#
# Function run by worker processes, i.e., this invokes the GEMPACK-based
#   'FIT' procedure for each country/region
#
def worker(input, good_output, fail_output):
    for case_id in iter(input.get, 'STOP'):
        #   Create and initialize the temporary directory
        setup(case_id)
        #   Change working directory
        tDir = wDir+'/'+case_id
        os.chdir(tDir)
        #   Run the program (i.e., batch file)
        args = [tDir+'\\runFit.cmd', case_id]
        startupinfo = subprocess.STARTUPINFO()
        startupinfo.dwFlags |= subprocess.STARTF_USESHOWWINDOW
        rc = subprocess.run(args,creationflags = subprocess.CREATE_NEW_CONSOLE, startupinfo=
            startupinfo)
        if 0 == rc.returncode:
```

```
66              good_output.put(case_id)
67          else:
68              fail_output.put(case_id)
69          #   Reset working directory
70          os.chdir(wDir)
71          #   Finish-up
72          cleanup(case_id)
73  #
74  #   This sets up and operates the queue
75  #
76  def runQueue():
77      global nProc, TASKS, nReg

79      NUMBER_OF_PROCESSES = nProc

81      # Create queues
82      task_queue = Queue()
83      good_queue = Queue()
84      fail_queue = Queue()

86      # fill up task queue
87      for task in TASKS:
88          task_queue.put(task)
89      # including stop signals so the worker processes terminate
90      for i in range(NUMBER_OF_PROCESSES):
91          task_queue.put('STOP')

93      print('Starting {} worker processes'.format(NUMBER_OF_PROCESSES))
94      w = []
95      # create and start worker processes
96      for i in range(NUMBER_OF_PROCESSES):
97          p = Process(target=worker, args=(task_queue, good_queue, fail_queue))
98          p.start()
99          w.append(p)

101     # and wait for them all to finish
102     for i in range(NUMBER_OF_PROCESSES):
103         w[i].join()

105     print('Worker processes are finished: all jobs are processed')
106     good_queue.put('STOP')

108     # Get and print good runs
109     if False:
110         print('Good runs:')
111         for case_id in iter(good_queue.get,'STOP'):
112             print('\t', case_id)
113         print('')

115     if fail_queue.empty():
116         print('There were NO failures')
117     else:
118         print('Failed runs:')
119         fail_queue.put('STOP')
120         for case_id in iter(fail_queue.get,'STOP'):
121             print('\t', case_id)

123 #   This is the entry point

125 if __name__ == '__main__':
126     freeze_support()

128     cStartTime = time.time()

130     #   Run all the simulations using the queue, i.e., in parallel mode
131     if True:
132         runQueue()
133         print('runQueue is finished')
```

```
134     else:
135         #   Run all simulations in sequence
136
137         good = []
138         bad = []
139
140         for r in TASKS:
141             setup(r)
142             tDir = wDir+'//'+r
143             os.chdir(tDir)
144             args = [tDir+'\\runFit.cmd', r]
145             rc = subprocess.run(args,creationflags = subprocess.CREATE_NEW_CONSOLE|subprocess.
                    SW_HIDE)
146             os.chdir(wDir)
147             if 0 == rc.returncode:
148                 good.append(r)
149             else:
150                 bad.append(r)
151             cleanup(r)
152
153         if len(bad) > 0:
154             print("The program failed for the following items:")
155             for r in bad:
156                 print(r)
157         else:
158             print("Program was successful for all items")
159
160     cStopTime = time.time()
161     print("Total time: "+str(cStopTime - cStartTime)+'\n')
```

Bibliography

Dietz, S., J. Rising, T. Stoerk, and G. Wagner. 2021. "Economic impacts of tipping points in the climate system." *Proceedings of the National Academy of Sciences*, 118(34). doi:10.1073/pnas.2103081118.

Horridge, J., and K. Peason. 2006. "Running Simulations Faster on Multi-Processor or 64-Bit PCs via GEMPACK." Center of Policy Studies (CoPS), Melbourne, Australia, Computing Document No. C15-01, April. https://www.copsmodels.com/ftp/workpapr/c15.pdf.

van der Mensbrugghe, D. 2023. "The META 21 Integrated Assessment Model in GAMS and LHS Sampling." Global Trade Analysis Project (GTAP), Department of Agricultural Economics, Purdue University, West Lafayette, IN, GTAP Working Paper No. 95. https://www.gtap.agecon.purdue.edu/resources/res_display.asp?RecordID=7036.

Villoria, N.B. 2017. "R Meets GEMPACK for a Monte Carlo Walk." *Journal of Global Economic Analysis*, 2(2): 128–154. doi:10.21642/JGEA.020204AF.

Appendix:
Running Monte Carlo simulations in parallel with GAMS

We briefly outline more or less the same Python script to undertake Monte Carlo simulations with a specific model in GAMS.[6] The key to parallelization is the labeling of the sample. The GAMS code has labeled each sample from `S1` to `S10000`. This is converted to the `TASKS` list in line 58—and each sample label will become an input argument to the GAMS code.

Unlike GEMPACK, GAMS does not need a separate directory to run, so there is no equivalent to the `setup` and `cleanup` functions that were useful to run the main script in the case of GEMPACK. The passing of arguments also appears to work naturally with the invocation of GAMS and thus there is also no need to create a batch file. The key process is defined on lines 31–49 of the `worker` function, which invokes GAMS, using the GAMS code `META21Stoch.gms` and a number of arguments—including `OBS`, which is set to the relevant sample number, identified with `case_id`. One key element of the setup is that each sample run starts from an existing solution, hence the use of GAMS' `restart` option. This in principle minimizes the start-up costs such as reading the core input files, calibrating parameters, etc. Parameters that are sample-dependent will of course be re-calibrated. The main program, lines 104–131, contains the GAMS execution of the base simulation, called `META21.gms`, which becomes the starting point for the Monte Carlo simulations.[7] One additional element of this version of the code is that the observation number for failed simulations is stored in a text file—named by the user on line 17. The remainder of the code is very similar to the code used for running the GEMPACK programs.

[6] The model is the META 21 model developed by Dietz et al. (2021)—originally in Excel and converted to GAMS (van der Mensbrugghe, 2023).

[7] The main program also has the possibility of running a single run of the Monte Carlo simulation—this has been useful for debugging purposes.

Listing 2: Runinng Monte Carlo simulations in parallel with GAMS

```
import os
import sys
import subprocess
from multiprocessing import Process, Queue, freeze_support
import psutil

#   User must set these variables
#   Simulation code ('wTP' or 'xTP')
sim  = 'wTP'
#   Core directory to hold simulation results
baseodir = "z:/Output/META21/"
#   Simulation specific directory
odir = baseodir + sim + "/"
#   Name for 'save/restart' file
base = "MetaRCP4_5SSP2" + sim
#   Name of file with list of 'bad' simulation(s)
badFile = "badFile_"+sim+".txt"
#   Set the sample size
nSample = 10000

#   Get the number of physical cores
nProc  = psutil.cpu_count(logical = False)

#   Check to make sure the output folder(s) exists
if(not os.path.exists(baseodir)):
    os.mkdir(baseodir)

if(not os.path.exists(odir)):
    os.mkdir(odir)

#
# Function run by worker processes, i.e., this invokes GAMS for each
#   of the observations
#
def worker(input, good_output, fail_output):
    for case_id in iter(input.get, 'STOP'):
        filename = odir+case_id+'.lst'
        gamsCommand = ['gams', 'META21Stoch.gms', '-restart=' + base, \
                       '--sim=' + sim, '-output=' + filename, \
                       '--OBS=' + case_id, '-lo=0']
        r = subprocess.run(gamsCommand)
        if 0 == r.returncode:
            good_output.put(case_id)
        else:
            fail_output.put(case_id)
            #   Save the observation number if the simulation fails
            fp = open(badFile, "a")
            fp.write(str(case_id)+'\n')
            fp.close()

#
#   This sets up and operates the queue
#
def runQueue():
    global nProc, nSample

    NUMBER_OF_PROCESSES = nProc
    TASKS = ['S' + str(i) for i in range(1,nSample+1)]

    # Create queues
    task_queue = Queue()
    good_queue = Queue()
    fail_queue = Queue()

    # fill up task queue
    for task in TASKS:
```

```
67          task_queue.put(task)
68      # including stop signals so the worker processes terminate
69      for i in range(NUMBER_OF_PROCESSES):
70          task_queue.put('STOP')
71
72      print('Starting {} worker processes'.format(NUMBER_OF_PROCESSES))
73      w = []
74      # create and start worker processes
75      for i in range(NUMBER_OF_PROCESSES):
76          p = Process(target=worker, args=(task_queue, good_queue, fail_queue))
77          p.start()
78          w.append(p)
79
80      # and wait for them all to finish
81      for i in range(NUMBER_OF_PROCESSES):
82          w[i].join()
83
84      print('Worker processes are finished: all jobs are processed')
85      good_queue.put('STOP')
86
87      # Get and print good runs
88      if False:
89          print('Good runs:')
90          for case_id in iter(good_queue.get,'STOP'):
91              print('\t', case_id)
92          print('')
93
94      if fail_queue.empty():
95          print('There were NO failures')
96      else:
97          print('Failed runs:')
98          fail_queue.put('STOP')
99          for case_id in iter(fail_queue.get,'STOP'):
100             print('\t', case_id)
101
102 #   This is the entry point
103
104 if __name__ == '__main__':
105     freeze_support()
106
107     # Delete the 'badfile' if it exists
108     if os.path.exists(badFile):
109         os.remove(badFile)
110
111     #   Initialize the GAMS' run (META21.gms) and 'save' the output
112     gamsCommand = ["gams", "META21.gms", "--sim=" + sim, "-save=" + base]
113     try:
114         r = subprocess.run(gamsCommand,check=True,shell=True)
115     except:
116         print("Process initialization failed...")
117         print("Verify that the PATH to GAMS is correct and/or that the file 'META21.gms' is in
                the working directory")
118         sys.exit(12)
119
120     #   Run all the simulations using the queue, i.e., in parallel mode
121     if(True):
122         runQueue()
123         print('runQueue is finished')
124
125     #   Optionally run the stochastic program in sequential mode
126     if False:
127         oname = os.path.join(odir,'metaStoch.lst')
128         gamsCommand = ["gams", "metaStoch.gms", "-restart="+base, "-o=" + oname, "-lo=2", \
129                        "--outdir=" + odir]
130         print('  gamsCommand = ', gamsCommand)
131         subprocess.run(gamsCommand,check=True)
```