UrbanSim Parallel Programming Capstone Paper

Aneesa Awaludin and David Chen
CSE 481E
June 4, 2007

Table of Contents

Introduction

UrbanSim is a software-based simulation model for planning and analyzing urban planning choices. It is intended to be used by city planners and policy makers to simulate what sort of impact their decisions may have on land use, transportation, employment, and population. It is an open-source project under the GNU General Public License and is freely available online. For more information about UrbanSim in general, visit http://www.urbansim.org/

The parallel programming project's purpose was to investigate how UrbanSim may take advantage of parallelism to reduce the time required to perform a full simulation.

Motivation

UrbanSim is very computationally intensive and performs many operations on huge sets of data. Since urban policy planners look several decades ahead when making their decisions, it is necessary to simulate 20-30 years. Depending on the size of the system being modeled, this can take several days.

Currently, UrbanSim runs everything in sequence using only thread at a time[1], taking no advantage of parallelism. Multi-core machines are now fairly mainstream and many of the machines in the undergraduate labs have dual core processors. As networking speeds increase, the possibility for cluster-based computing also increase. UrbanSim, being single-threaded, can only take advantage of just a single core on a single machine.

By modifying UrbanSim to utilize parallelism, we hope to reduce the time needed for a full simulation drastically.

Goals

We had three main goals for the project:

1. Evaluate various parallel Python packages on their feasibility to be used with UrbanSim. Considerations include code maturity, ease of use in integrating with UrbanSim, what kinds of parallelism were supported, and their speed-up gains.

[1] UrbanSim does fork a process for each year of simulation, but then the parent process blocks until the child is finished. According to Professor Alan Borning, it does this to circumvent some behavior in Python's memory management.

2. Modify UrbanSim so that different models can be run in parallel. Each simulation can be broken up into different sections called models, which simulate one aspect of the system. Many of these models share datasets and/or depend upon another model being run first, but there are some that can be run in parallel. Since it requires significantly more knowledge of UrbanSim to make code run in parallel within the models, we are targeting running different models in parallel first.
3. Evaluate whether parallelism is promising enough to warrant incorporating it into the main trunk of UrbanSim in the future. This mostly involves running timing tests and weighing the speed-up gains against the trade-offs. This is the main purpose of the project.

Different Types of Parallelism

There are various different computing strategies all related to parallel programming. They are not mutually exclusive and can be used in conjunction with each other. Since the terms will be used later, they will be described here.

- Distributed computing – this is when the parallelism extends across multiple machines, usually communicating with each other on a network.
- Scatter/gather – a technique involving a central controller breaking up large datasets into smaller parts, "scattering" the parts out to different "workers" (which could just be different processes or could even be different computers), having each worker complete its portion of the calculation, and then the controller "gathers" the results and reforms the large datasets.
- Task farming – a central controller has a list of tasks that can be done independently of each other. Whenever a "worker" declares itself free, the controller will assign it a task.
- Shared memory – this is when multiple workers or threads do not need to copy data to pass each other information; instead, they rely on commonly shared memory. This has the advantage of not needing copying, but does require careful use of locks.

Ethical Considerations and Broader Impact

With any engineering project, it is necessary to evaluate how the project fits into society and will work in context. Since UrbanSim is used by policy makers and urban planners to make politically sensitive decisions, care must be taken that the project follows ethical guidelines.

Concerning UrbanSim in General

Policy makers and urban planners will be using the project to analyze the effects of different proposed choices. This includes analysis into traffic patterns, land property values, and measures of welfare and equity. It is vitally important that the engineers and scientists behind the construction of UrbanSim be up front about the limitation and accuracy of the simulation lest people put too much weight on its output.

UrbanSim does not model everything, though there are certainly many other things that would be useful and arguably have a large influence on urban planning decisions, such as education. However, many of these factors have do not have a consensus on what exactly the effects would be on a simulation of a city and are subject to much controversy. To use a controversial model and claim that it is indeed how the factor affects reality would be ethically dubious.

Similarly, it is important not to bias the system to favor policies the engineer or scientist may themselves favor.

UrbanSim's impact on society in general is mostly just that policy decisions can benefit from a little bit more scientific evidence about what would happen if one choice were made over another.

Relating Specifically to Parallel Programming

There is not too much to say about the ethical considerations of incorporating parallel programming into UrbanSim. It doesn't qualitatively change any of the outputs of the system; all it does it make it run faster, which allows it to be used more often. In that sense, it may heighten the other ethical considerations since it is more likely to be used if it can be used conveniently, but no other ethical considerations come to mind.

Methods

In this section, we'll discuss the major options that were looked at for enabling parallelism for UrbanSim. A lot of the useful results from evaluating the choices mostly comes in the form that the option would be inadequate, but we did get some positive results with process forking. Here are the different options we evaluated at length:

1. IPython (promising, but not quite there yet)

2. Native Python threads (not suitable)
3. Virtual Shared Memory (not suitable)
4. Process forking and disk communication (current choice)

There are many other parallel computing packages for Python in existence, but most of them only supported the "scatter/gather" style of parallelism in which a single dataset is broken up into small pieces, sent out to various workers, and then collected. Since we wanted to make distinct models run in parallel and each model performs different operations, this was not the type of parallelism suited for our purposes.

IPython

IPython stands for Interactive Python. While the default Python shell is already fairly interactive, it blocks on GUI applications. In addition, IPython is working on officially supporting parallel computing for version 1.0 (current version is 0.8.4) and has a relatively stable branch that already includes most of these capabilities.

Advantages

- It is part of the SciPy umbrella of packages and is thus designed to work well with other SciPy packages, including NumPy and Numeric. It has special support for the special array types that NumPy and Numeric use and which UrbanSim uses extensively.
- Actively maintained and currently being developed.
- Supports, or will eventually support, many types of parallelism: task farming, scatter/gather, distributed computing, shared memory.
- Supports MPI (a cross-language message passing protocol) which may be useful if we ever extend UrbanSim with non-Python libraries.
- Interactive shell would be useful for debugging and keeping track of the various parallel processes.
- Supports "workers" (different processes from the same computer or from a different one) dynamically leaving or entering the worker pool, though I'm unsure how it deals with a worker leaving if it has already been assigned a task. We believe you have to explicitly program recovery choices.

Disadvantages

- Parallel computing capabilities are not yet mature. Specifically, the shared memory model that we wished to use was not yet implemented, though they believe it should be "trivial" to do.

- In Windows, it requires the same compiler (MS Toolkit 6.0) that the Python binaries were built from to build some of the modules. This compiler, once freely available from Microsoft's website, seems to have disappeared when Microsoft introduced its new free Visual Studio compilers. Attempting to install it through Cygwin lead to a Cygwin Python bug where the Python process and various libraries do not map to the same base address. The documented fix is to use "rebase all" from the commandline to change the base addresses, but it and many variations did not work for us.
- Installing UrbanSim and IPython on Linux requires hunting down quite a bit of dependencies and a Fortran compiler, as well as the ability to interpret somewhat obscure errors when compiling from the source fails. This consumed a lot of our time. On machines without Fortran compilers and where the user does not have root access, it is non-trivial to install and is probably not practical to ask students to do so if IPython on Linux became a required dependency.

Verdict

We had a lot of trouble getting a system to install both IPython and UrbanSim. We eventually gave up on doing so for Windows because of the missing compiler necessary for IPython and because of the rebase fix not working for us in Cygwin. Thus, any UrbanSim installation utilizing IPython would not work on Windows unless solutions are found.

After successfully installing IPython and attempting to get it to do shared memory, we eventually discovered on the IPython wiki that the shared memory model has not yet been implemented: "[…] With that said, it would be nearly trivial to add VSM (Virtual Shared Memory) capabilities to our kernels." From the developer's mailing list archive, it seems the difficulty is getting a solution that works on both Windows and Linux, but we do not know how far along they are on that.

Still, IPython is very promising since it is pretty stable and has good support for some other forms of parallelism and NumPy arrays. We believe that it should be pursued further; even without shared memory, modifying UrbanSim to pass messages between processes is definitely a possibility.

Native Python Threads

This is the included threads package in Python. Python supports native threads, and this was the next option we explored after discovering that IPython did not support shared memory parallel computing.

Advantages

- No additional dependencies to install; no new third party packages to learn.
- Any person who has taken an operating systems class will understand how the thread model works.
- Supports shared memory, which will eliminate the need for unnecessary copies of datasets to be made.

Disadvantages

- Python uses a Global Interpreter Lock (GIL), which prevents more than one thread from actually executing at a time, per process.
- Would not support distributed computing or task farming styles of parallelism, and would require much more work to support a scatter/gather style of parallelism than IPython.

Verdict

Unfortunately, we did not know about the GIL when we first pursued this option. This effectively negates any gains made from introducing parallelism to computationally bound programs, which includes UrbanSim. It would be useful for I/O bound systems, but that's just not currently the problem for UrbanSim. It does not seem like they plan on addressing the GIL in CPython (the official Python) until version Python3000[2].

IronPython and Jython may be able to get around this earlier, since they are based upon systems that do not use a GIL, but they would introduce another dependency and it is unknown how quickly they would run UrbanSim.

Virtual Shared Memory

There are various third party packages that claim to support sharing Python objects between processes without the need for copying. Since we want to avoid copying data unnecessarily, we looked at these next. The two most promising were POSH (Python Object Sharing) and PyLinda. We'll skip straight to the verdict.

Verdict

Unfortunately, POSH was built on top of an earlier version of Python (2.3) and did not work for the current release (2.4.3). Running their sample tutorial code produced double-free errors or deadlocks.

[2] Python3000 is named such because it's a somewhat hypothetical release that Guido Van Rossum (creator of Python) would do if he felt the need to rebuild Python from scratch and not require backwards compatibility. The 3000 is the hypothetical version number or possibly the year in which it would be released.

Since double-free errors deal with memory management and garbage collection, we did not feel qualified to attempt to update POSH to work with the current version of Python. In addition, it would only work at all on *nix machines and not Windows, thus breaking UrbanSim's current multi-platform nature.

POSH is not currently being maintained and has not been for years, so it is not likely that an update will occur.

PyLinda is multi-platform and has been more recently maintained than POSH, but only supports built-in Python types, such as ints, lists, etc. It does not work with user-defined classes, such as the the datasets of UrbanSim, so it was not useful for our purposes.

Process Forking and Disk Communication

This was our last fall back and fortunately it worked reasonably well. Process forking involves copying everything associated with the process in memory to a new process, which will run independently from the first exactly where the first left off. We had the processes communicate via the flushing and loading functions built-in to UrbanSim, which writes and reads datasets to the disk.

Advantages

- No new dependencies; nothing new to learn.
- Seems to work decently well.
- The same model without too much work could be adapted to work with IPython and interprocess communication, enabling distributed computing.

Disadvantages

- Large amount of copying! Effectively doubles memory requirements.
- Writing and reading to disk is even slower than copying memory.
- Does not support distributed computing or task farming.

Verdict

This is the option we went with, mostly because the other options were not viable and by the time we got to this point, modifying UrbanSim to do interprocess communication using IPython would require too much time and had a chance of not working. Since we wanted some sort of positive result other than “this doesn't work and shouldn't be pursued”, we did our timing tests using this method.

Description of the Eugene System Models

The Eugene System is the set of data describing Eugene, Oregon. We chose this dataset because it was included in the default UrbanSim simulation and is fairly small, which makes it much more manageable for running repeated timing tests, particularly if we want to run it for many years. Considering the time constraints, we chose this over the Puget Sound Regional Council data. It is useful to describe what the models are and in what order they run in the single-process and multi-process versions, so we'll do that here.

Single Process

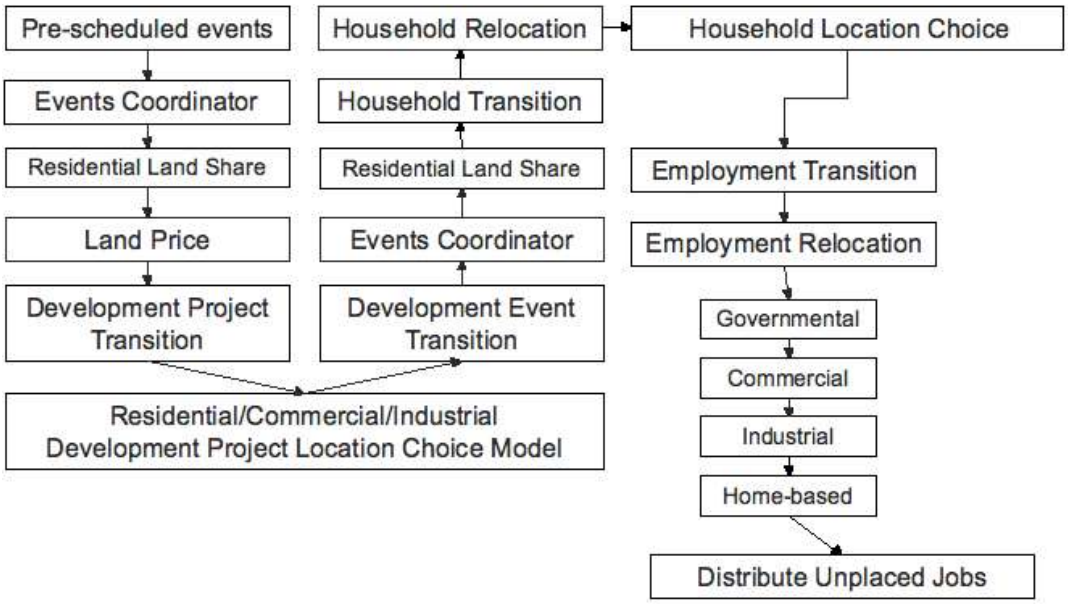


Illustration 1: Execution flow for the Eugene system using 1 process

It begins with "Pre-scheduled events", starting on the top-left of this diagram. The arrows indicate what is run next. The final model is "Distribute Unplaced Jobs".

Two Processes

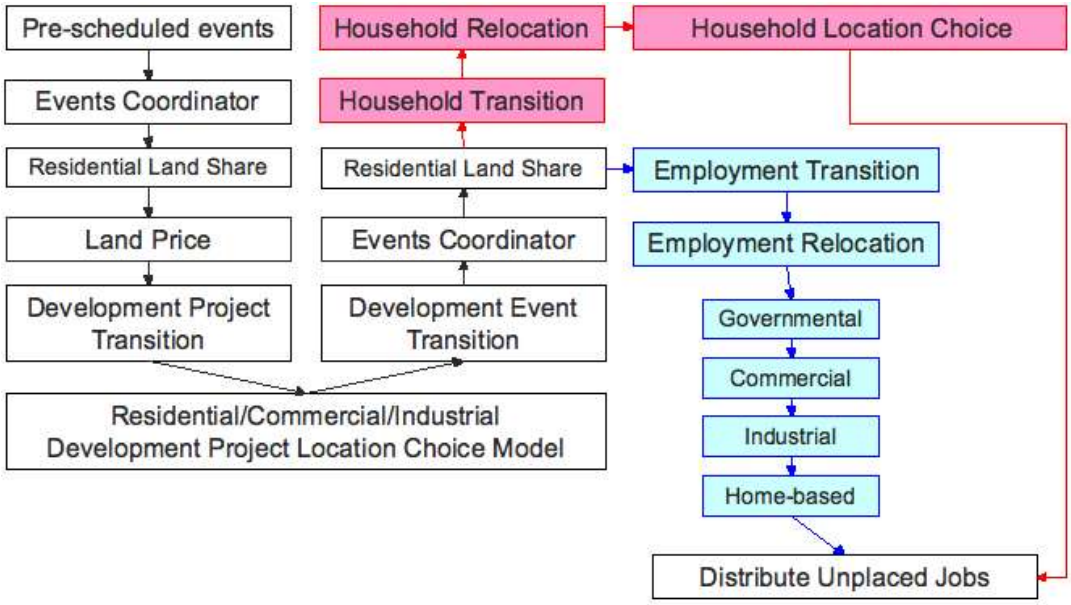


Illustration 2: Execution flow for the Eugene system using 2 processes

The change here is that after the second "Residential Land Share" model is run, an additional process is forked. The parent process continues on the red path, whereas the child process continues on the blue in parallel. They sync up before "Distribute Unplaced Jobs" is executed. The models up to "Residential Land Share" need to be run in sequence, and are thus cannot be run in parallel.

Five Processes

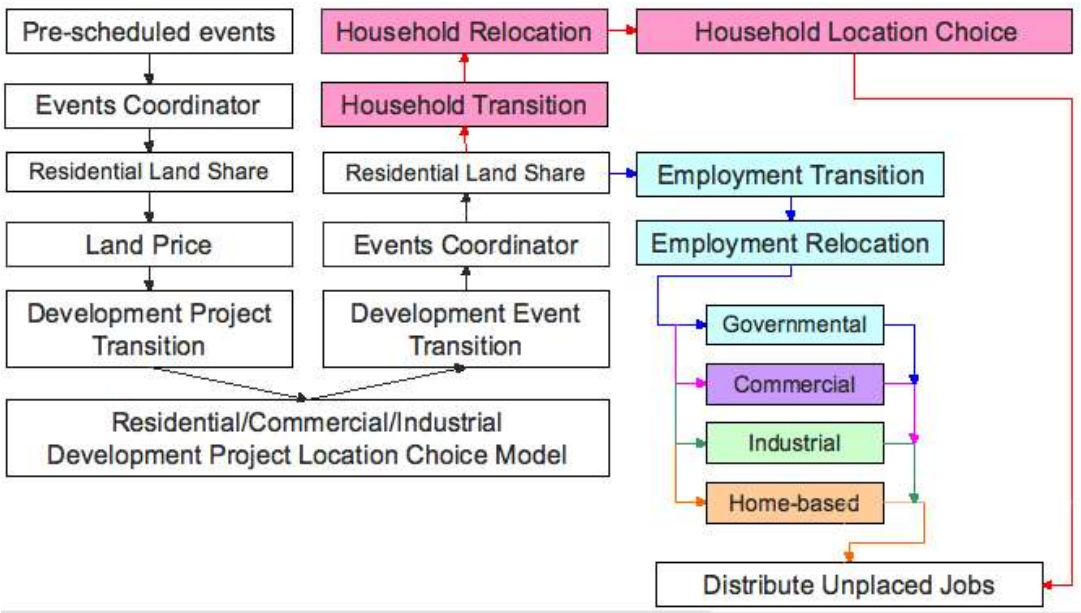


Illustration 3: Execution flow for the Eugene model using 5 processes

For five processes, the governmental, commercial, industrial, and home-based employment location choice models may be run in parallel.

Some Technical Details

We modified the class opus.urbansim.model_coordinators.model_system, which is in charge of reading the configuration file and running each of the models in order. We changed it so that it would fork a process upon reaching the "Household Transition" model. The parent process would skip the employment models and upon reaching the "Distribute Unplaced Jobs" model, wait for the child to die.

Meanwhile, the child would skip the household models and run until it reached "Distribute Unplaced Jobs" model, in which case it would use self.flush_dataset() to flush the job, gridcell, employment_sector, and employment_sector_group datasets to the cache (on the disk), then die.

Upon detecting the death of the child, the parent process will use dataset.unload_all_attributes() and dataset.load_dataset() to read the information from the cache before continuing on to "Distribute Unplaced Jobs".

The five-process version is a minor variation involving forking more processes upon reaching the "governmental employment location choice" model and again syncing at "distribute unplaced jobs".

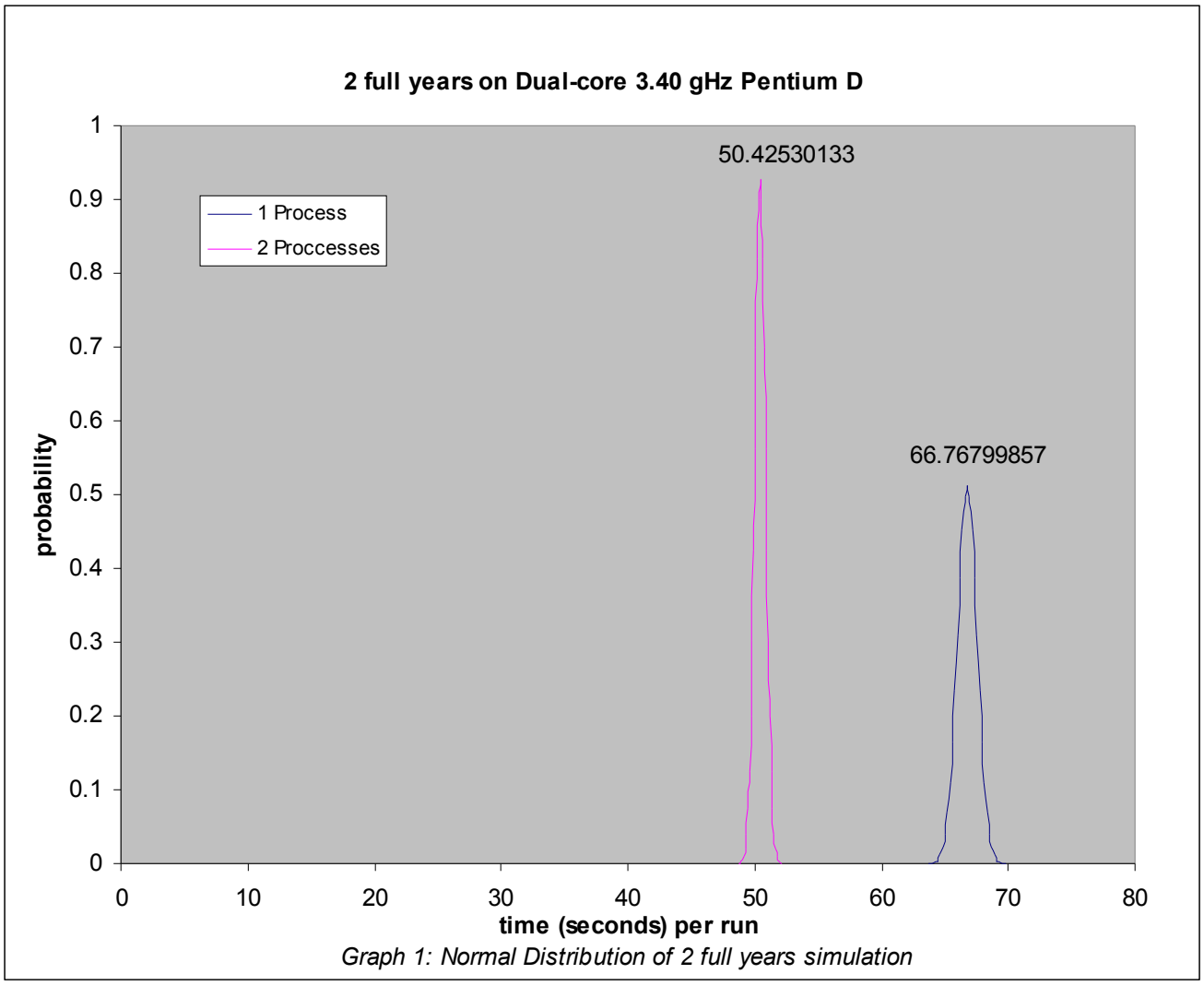
Timing Methodology

We ran simulations on two types of machines: Intel dual-core 3.40 gHz Pentium D and two Intel Xeon dual-core 3.0 gHz processors. To make our timing easier, we wrote a timing script in Ruby. We included this short script in Appendix B. Basically, each simulation is run a few times and the total time of each of the simulation is outputted in a comma-separated value (csv) file. This file is then can be easily imported into Microsoft Excel for further analysis (calculating standard deviations and producing graphs).

On Intel dual-core 3.40 gHz Pentium D, we ran two types of simulations: the single process simulations and the two processes simulations. On the two Intel Xeon dual-core 3.0 gHz processors, we ran three types of simulations: the single process simulations, the two processes simulations and the five processes simulations. For all the simulations, we ran only the simulations of the parallelized models for two, five and ten years.

Results

Below are our findings. All of our findings are graphed in normal distribution graphs, where the x-axis represents the time (second) per run and the y-axis represents the probability.



Graph 1: Normal Distribution of 2 full years simulation

One process:

$\mu = 66.7680$ s

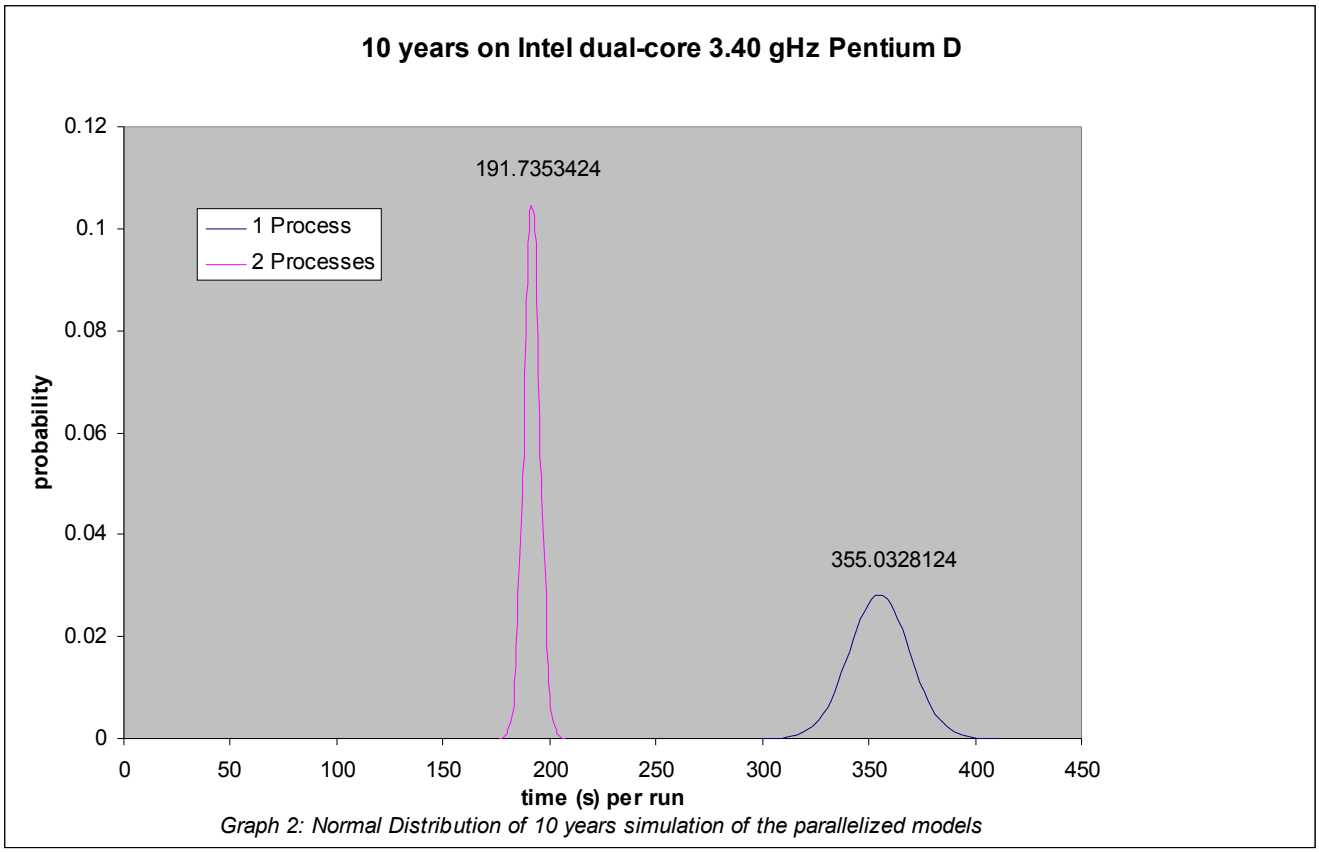
$\sigma = 0.7791$ s

Two processes:

$\mu = 50.4253$ s

$\sigma = 0.4297$ s

The graph above graphed the normal distribution of 2 full years of simulations. From the result above, we can see that the mean of the simulations using two processors was cut short more than 15 seconds, which means the simulations ran about 24.47% faster than their counterparts which uses only one processor.



Graph 2: Normal Distribution of 10 years simulation of the parallelized models

One process:

$\mu = 355.0328$ s

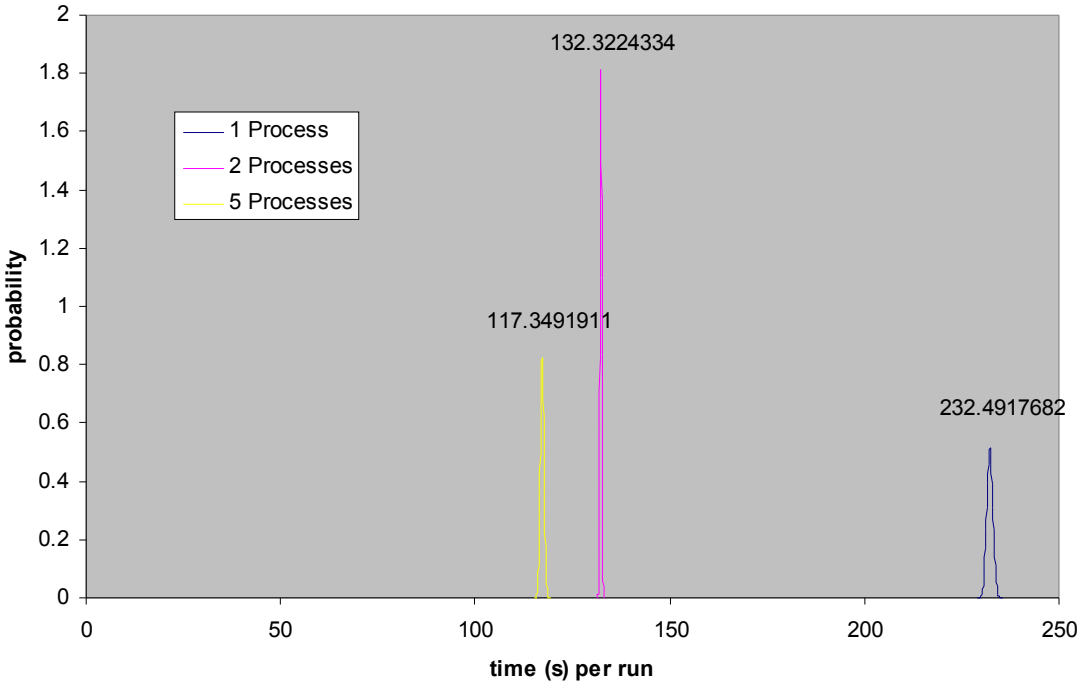
$\sigma = 14.1256$ s

Two processes:

$\mu = 191.7353$ s

$\sigma = 3.8144$ s

The graph above graphed the normal distribution of 10 years simulation of parallelized models on a dual-core machine. We can see that the two processes simulations ran about 46% faster than the single process simulations. Since we only ran the parallelized models, it is expected that we would gain almost 50% of the simulation time.



Graph 3 : Normal Distribution of 10 years simulation of parallelized models

One process:	Two processes:	Five processes:
$\mu = 232.4918$ s	$\mu = 132.3224$ s	$\mu = 117.3491$ s
$\sigma = 0.7742$ s	$\sigma = 0.2203$ s	$\sigma = 0.4853$ s

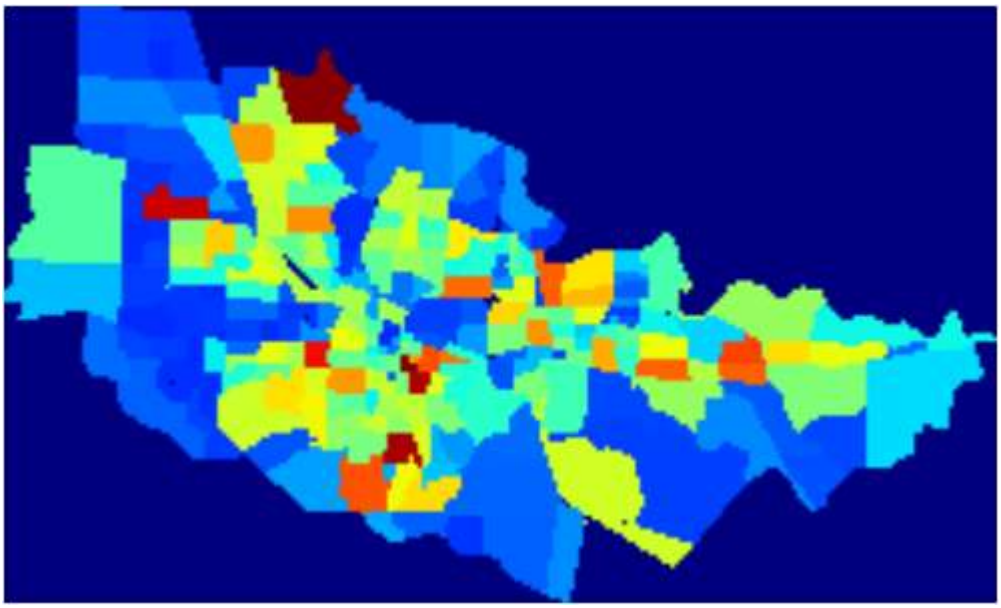
Above is the normal distribution graph of 10 years parallelized models simulations on a quad-core machine. While the two processes simulations gain about 43.1% from the single process simulations, the five processes simulations did not gain very much from the two processes simulations. The five processes simulations ran only about 15 seconds faster than the two processes simulations. This result is not unexpected because in the five processes simulations, the parallelized models (employment location choice models) take only a while to be simulated. In addition, the overhead cost of flushing and reloading from disk is greater than the time gained from these simulations.

In general, the simulations ran faster on the quad-core machine compare to the simulations ran on the dual-core machine. The single process simulations ran about 34.5 % faster in the quad-core machine, while the two processes simulations ran about 31% faster.

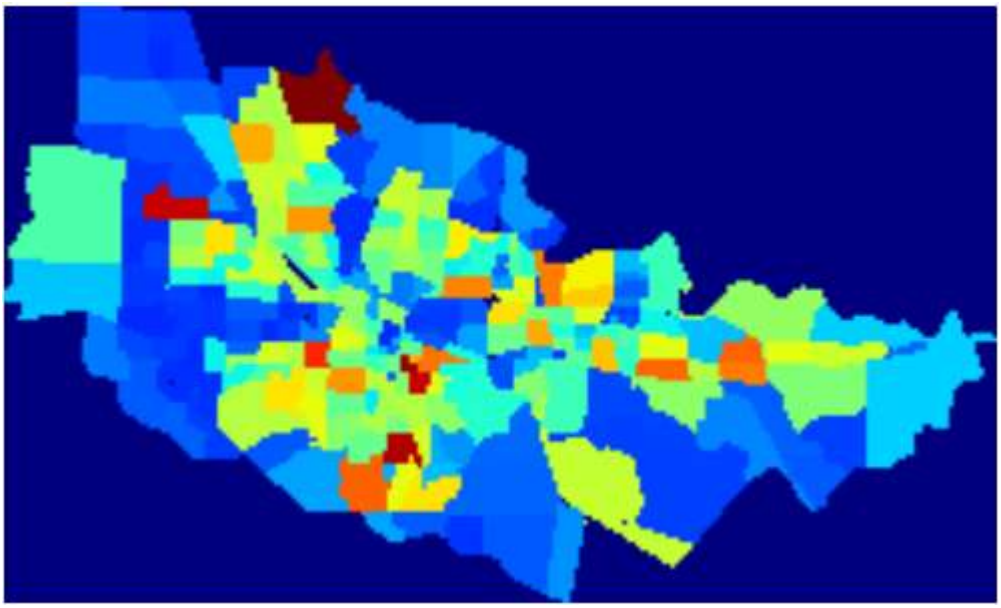
In most cases, note that the standard deviation of single process simulations is larger than the standard deviation of two processes simulations. We are not entirely sure of why it is so.

More of our findings are included in Appendix A.

Now that we managed to get the simulations to run faster, we have to validate our results. On the advice of Liming Wang, we decided to produce maps of the population and number of jobs indicator.



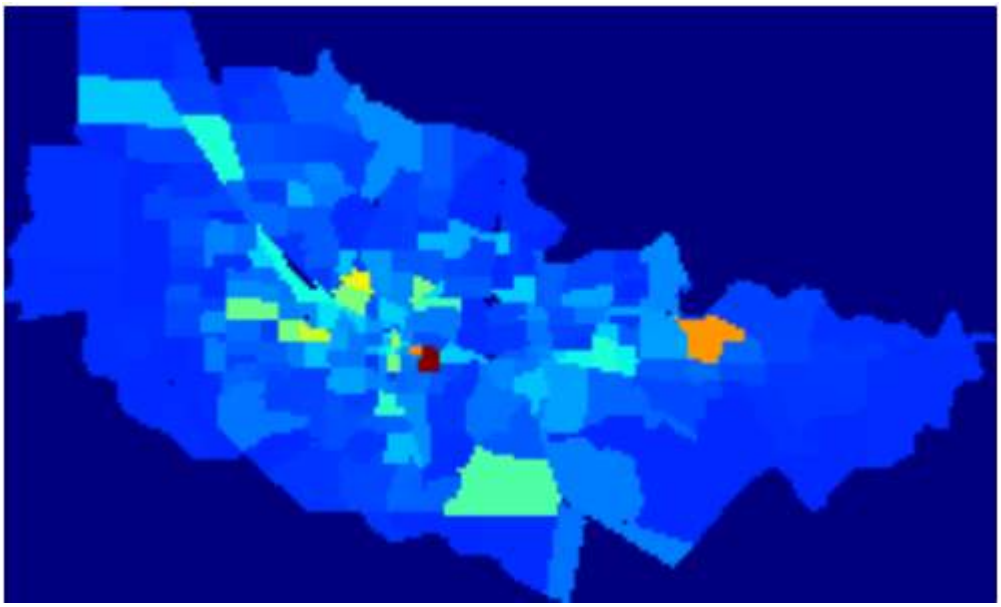
Map 1: Population in 1990 for single process simulation



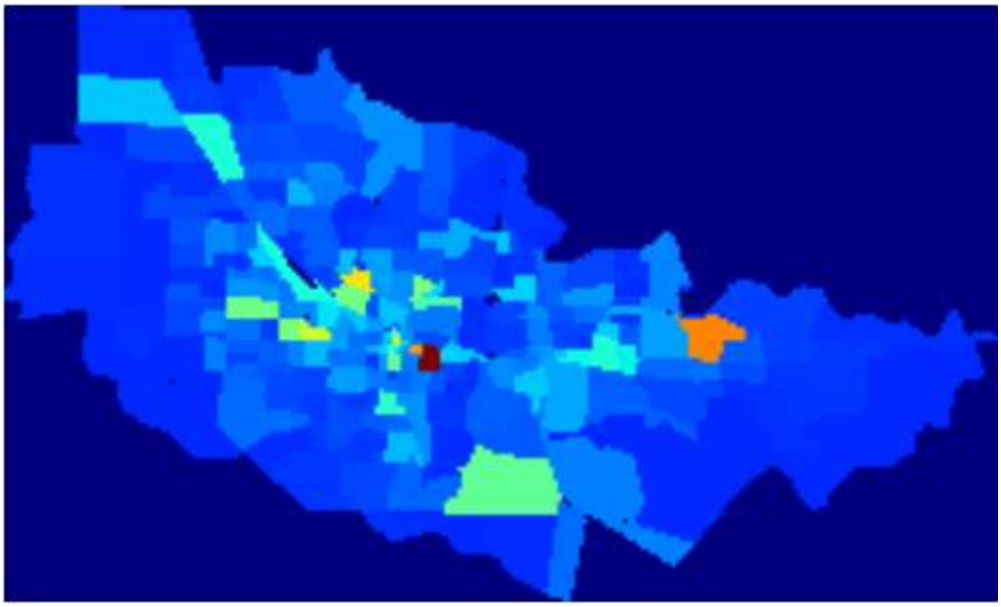
Map 2: Population in 1990 for two process simulation

The above maps are the population at Eugene, Oregon in 1990 based on the simulations. The top map is from a single process simulation while the bottom one is from a two processes simulation. We can see that the map produced by the two processes simulation is similar to the other one. The reason why these maps cannot be exactly the same is because UrbanSim simulations are non-deterministic. Even with the

same configurations, we will not get the exact same map for each run.



Map 3: Number of jobs in 1990 for single process simulation



Map 4: Number of jobs in 1990 for two processes simulation

Similarly, the maps above indicate the number of jobs in 1990 in Eugene. The top map is produced by a single process simulation while the bottom one is produced by a processes simulation.

Based on these maps, we believed that we have reasonable results out of the two processes simulations.

Conclusion

We feel that the results are very promising and that further effort should be made in utilizing parallel computing in UrbanSim. The speed-up gains are very impressive even considering the inefficiencies of memory copying and reading and writing to disk. While not all the models can be made to run in parallel, in the case of the Eugene system, those that could were the ones that took the most time to run and so the system overall benefited noticeably.

Caveats

There are some rather large caveats.

- We did not have the time to test PSRC or any other system. Eugene is fairly small, and it is on large datasets that we really care about speeding the process up. It's not yet known whether larger system would experience the same speed-up, though we are pretty confident that they would, especially if the system was modified to not require forking the existing process and to utilize inter-process message-passing with IPython.
- On larger systems, forking the process is not really feasible since it doubles the memory requirements. Some of the systems already require special measures to ensure a lower memory footprint, and this would definitely not help. One such measure we had to disable to get the parallelism working: the parallel version no longer flushes the gridcell, job, and household datasets to cache after each model runs, because doing so would overwrite the results of either the child or the parent process.
- Our current code is very brittle and essentially depends on certain models being specified in the configuration file in a certain order. It could take a lot of work to make this flexible and configurable inside the file.

Recommendations for Future Work

1. Performing timing tests on larger datasets, like PSRC would be one of the first things to do, to see whether or not the large gains seen in for the Eugene system scale up.
2. Modifying the UrbanSim to use inter-process message passing instead of flushing to disk and reading from disk would theoretically be faster, changing it to be a memory copying operation.
3. Modifying UrbanSim to not depend upon a fork; instead create a new process with just the model code in memory and use inter-process message passing to give it the datasets necessary to run the model. This would reduce the memory consumption requirements considerably and also

allow the system to be integrated with IPython, opening up possibilities to use distributed computing and its other supported parallel options.

Bibliography

IPython Wiki. 23 May 2007. <http://ipython.scipy.org/moin/>

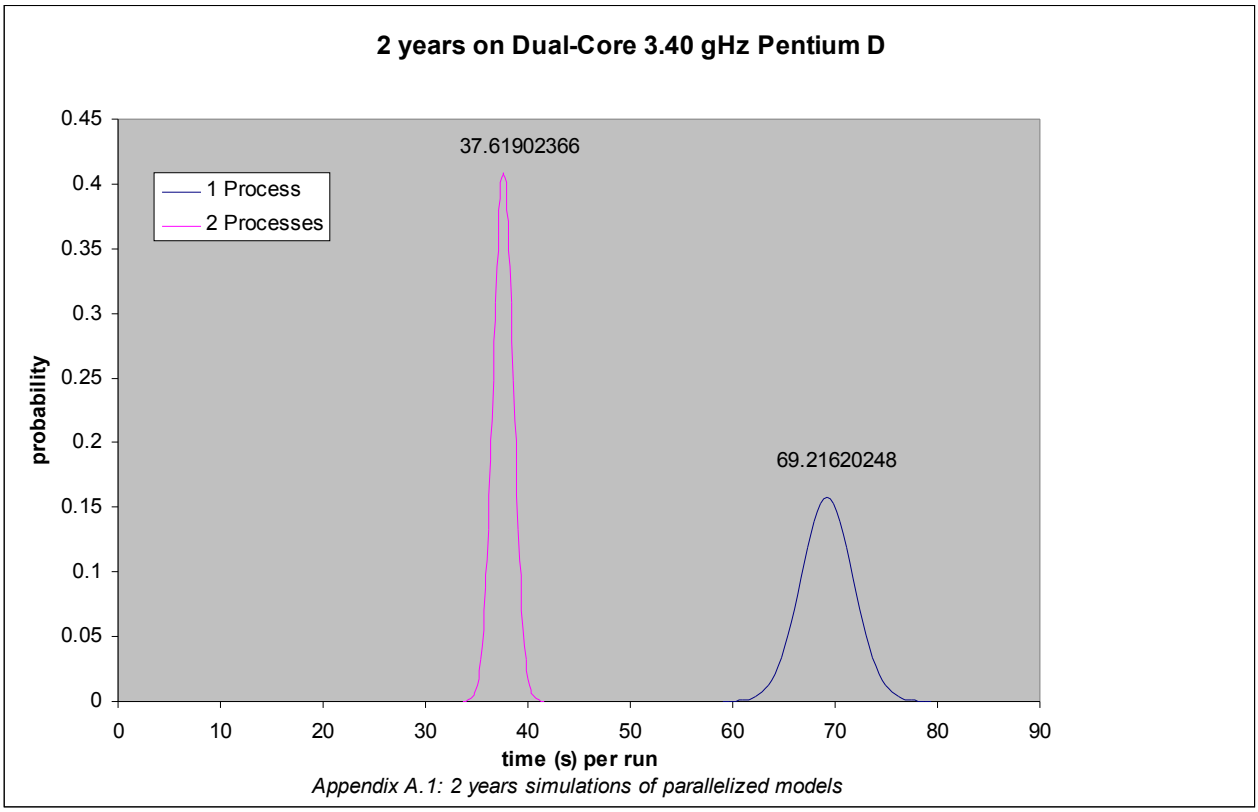
POSH: Python Object Sharing. Stefen Valvag, Age Kvalnes, Kjetil Jacobsen. Mar 25 2003.
< http://poshmodule.sourceforge.net/>

PyLinda – Distributed Computing Made Easy. Andrew Wilkinson. December 7 2005
< http://www-users.cs.york.ac.uk/~aw/pylinda/>

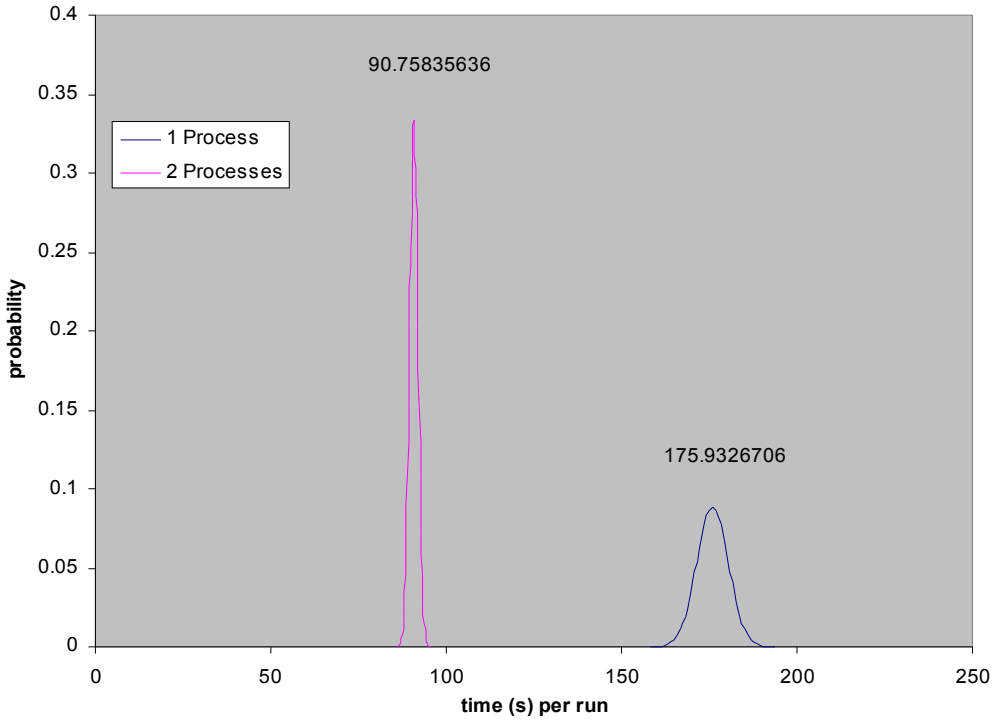
Understanding Threading in Python. Krishna G Pai. October 2004.
< http://mirrors.techiesabode.com/linuxgazette/107/pai.html>

Thread State and the Global Interpreter Lock. September 19 2006.
<http://docs.python.org/api/threads.html>

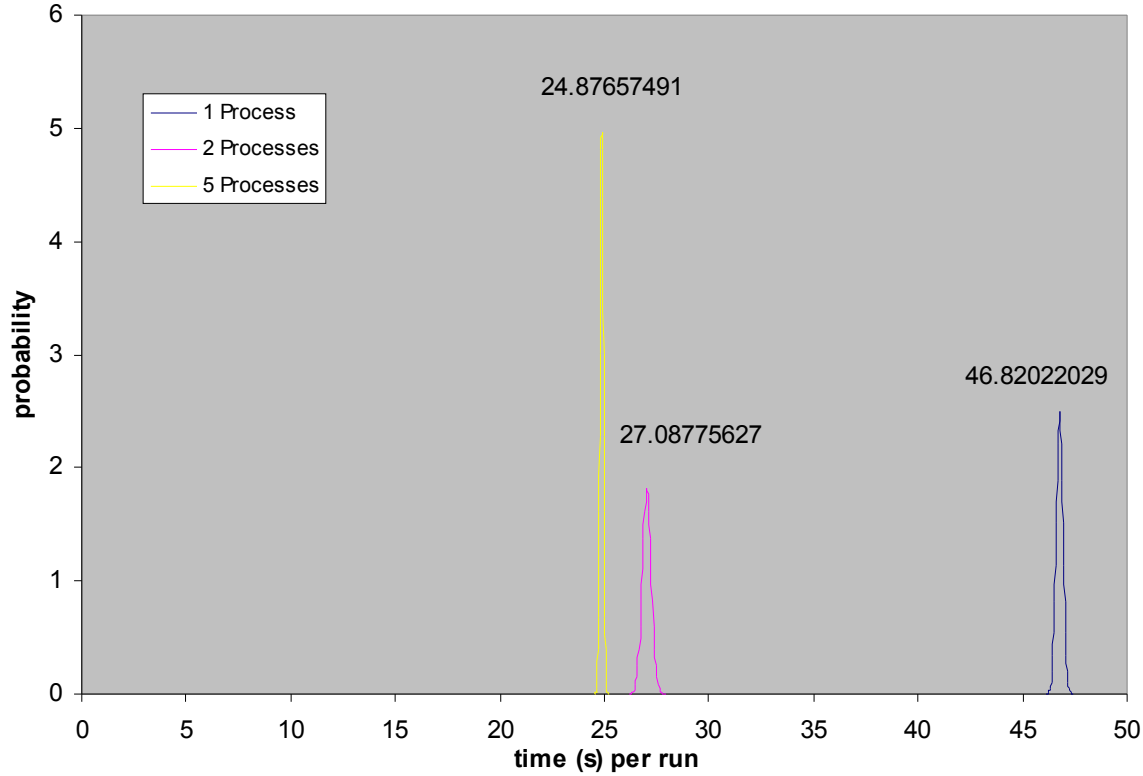
Appendix A – Data



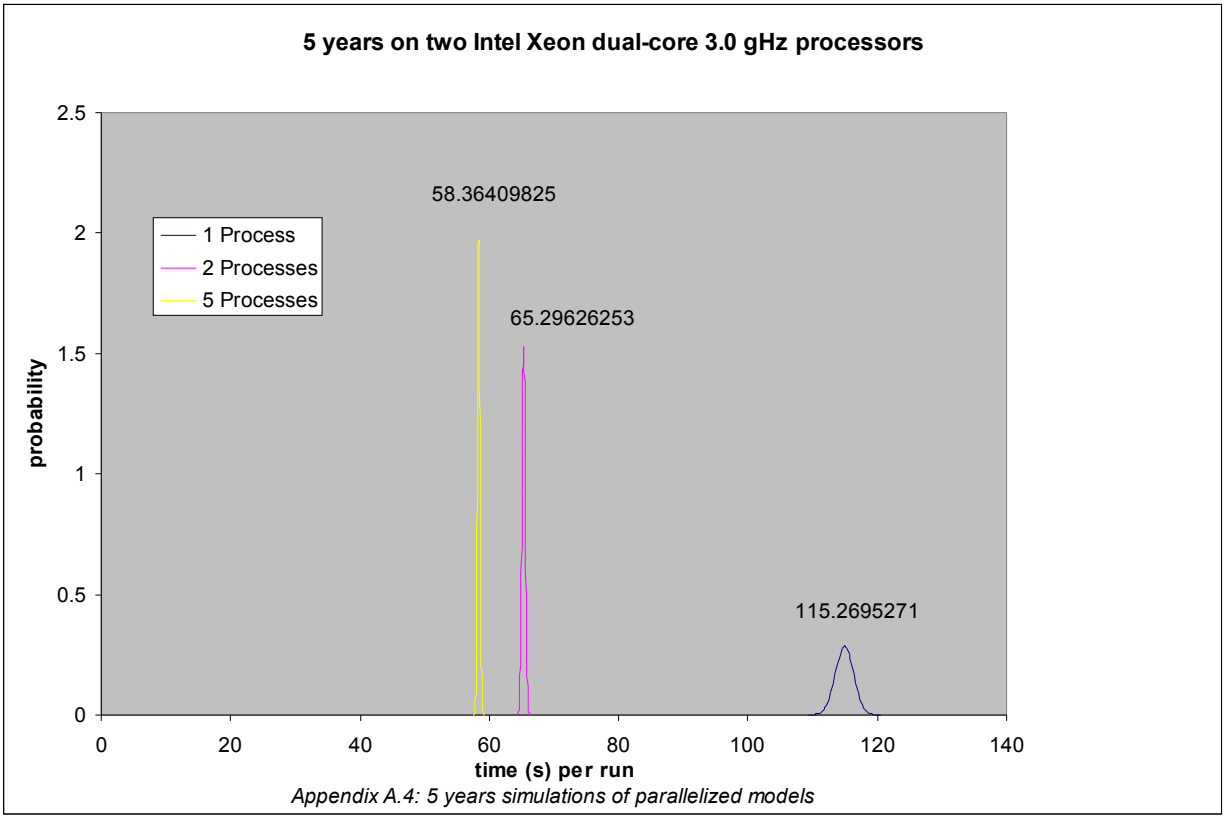
Appendix A.1: 2 years simulations of parallelized models



Appendix A.2: 5 years simulaion of parallelized models



Appendix A.3: 2 years simulations of parallelized models



Appendix A.4: 5 years simulations of parallelized models

Run #	Single Process (2 years)	Two Processes (2 years)	Single Process (5 years)	Two Processes (5 years)	Single Process (10 years)	Two Processes (10 years)
1	78.191391	37.19854808	173.633189	92.91269088	399.2960551	193.1278272
2	69.282624	39.37048197	172.8684571	92.89072204	350.008697	188.3642569
3	69.022647	37.10362697	174.812139	91.16971111	344.6624498	187.179522
4	68.578117	37.17818093	174.1421671	90.9341321	351.2968299	187.536411
5	68.874138	37.38737702	172.2034109	91.55451894	362.449862	187.10027
6	69.151218	36.65210199	175.252985	92.17336917	345.3440142	190.6882591
7	68.407275	37.37913704	174.3920729	90.99650002	343.967078	189.4561732
8	68.509737	37.10212684	174.1990929	89.9049542	351.8592939	189.3782601
9	68.858366	38.20426297	172.463114	90.21305609	351.387768	189.0103369
10	67.634602	36.88245893	172.483618	89.30836082	345.5125711	192.8922241
11	68.090453	37.52284503	173.6767149	89.6839292	345.7961762	196.4079669
12	69.301983	37.66346002	179.9754999	90.07067013	363.3915448	195.7441001
13	67.759035	37.15847492	177.401227	89.97713685	358.1170912	194.1842561
14	68.211846	37.20336413	184.4049208	89.21881485	347.9942749	197.533483
15	68.369605	40.27890801	187.0814509	90.36677909	364.4084799	197.42679
Mean	69.216202	37.61902366	175.9326706	90.75835636	355.0328124	191.7353424
Std Dev	2.5358952	0.977380899	4.491938602	1.19434275	14.12557871	3.814411208

Appendix A.5: Timing results from simulations run on the dual-core machine

Run #	Single Process (2 years)	Two Processes (2 years)	Five Processes (2 years)	Single Process (5 years)	Two Processes (5 years)	Five Processes (5 years)	Single Process (10 years)	Two Processes (10 years)	Five Processes (10 years)
1	46.89812	26.9871	24.8347	119.5722	65.53707	58.09289	231.8228	132.2904	116.3684
2	46.60313	26.87442	24.93088	114.084	65.0579	58.43142	232.6805	132.5548	117.8504
3	46.88233	26.89151	25.0414	114.8915	65.116	58.13801	232.7632	132.3794	117.1929
4	46.85828	27.17055	25.00353	114.7302	65.3337	58.29467	233.3325	132.4956	117.9431
5	46.75219	26.94351	24.81877	114.5469	65.38667	58.38885	231.752	132.7306	117.4449
6	46.67308	26.9167	24.72525	114.9689	65.25727	58.28594	231.3479	132.1948	117.5858
7	46.64219	27.0038	24.88176	114.1845	65.34334	58.25691	232.4597	132.5633	116.8574
8	46.57344	26.94296	24.90317	114.8072	65.30817	58.20706	231.6505	132.136	117.589
9	46.80994	27.77664	24.9039	114.1409	64.91564	58.37537	231.4491	132.212	117.7265
10	46.71795	27.02754	24.88593	114.5079	65.40603	58.26597	232.875	132.2131	117.2505
11	46.56183	26.95399	24.91571	115.6418	65.04791	58.5843	234.0002	132.6166	116.4069
12	47.14911	27.05848	24.82827	115.5891	65.08096	58.22757	232.6007	132.0101	116.8535
13	46.82833	26.94873	24.85913	113.9994	65.15057	58.87788	232.5217	132.0682	117.2677
14	46.95473	26.9583	24.78672	114.0197	65.53649	58.4519	231.7553	132.2383	116.8407
15	46.7994	27.04214	24.82951	114.1432	65.9662	58.58272	231.4622	132.1332	117.2402
Mean	46.78027	27.03309	24.87657	114.9218	65.29626	58.3641	232.2982	132.3224	117.2279
Std Dev	0.159804	0.218661	0.080292	1.390802	0.260968	0.202354	0.77418	0.220326	0.48528

Appendix A.6: Timing results from simulations run on the quad-core machine

Run #	Single Process	Two Process
1	67.84297	51.34334
2	67.51328	50.63018
3	66.46696	50.33828
4	66.73616	50.36653
5	68.13353	50.68755
6	67.5955	50.03528
7	66.34144	50.26436
8	67.65055	50.84474
9	66.36268	50.32332
10	66.3708	49.95345
11	65.78634	50.68612
12	66.09404	50.22257
13	66.50952	50.03401
14	65.6282	50.93756
15	66.48801	49.71223
Mean	66.768	50.4253
Std Dev	0.779064	0.429674

Appendix A.7: Timing results from the full simulations run on the dual-core machine

Appendix B – Code

Timing Script

```
avg = 0
runs = 0
aFile = File.new("output_file.csv", "w")
aFile.write "Run # , Ave time\n"
15.times do
      runs = runs + 1
      before = Time.now.to_f
      system("python start_run.py -c eugene.configs.baseline > trash")
      after = Time.now.to_f
      aFile.write runs.to_s + " , " + (after - before).to_s + " \n"
      avg = avg + after - before
      puts "run # : " + runs.to_s
end
aFile.close
```