A Thread Placement Simulator for Massively Many-core Processors

R. Colls

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

A dissertation presented in part fulfillment of the requirements of the Degree of Master of Science at the University of Glasgow

September 2014

Abstract

This report describes a software framework that can be used to simulate the performance of thread placement strategies. The simulations are intended to be quick rather than cycle accurate so that they can be applied to processors with many thousands of cores.

The simulator models the degradation of thread running speed caused by both time sharing of a c.p.u. and distant cache access. The characteristics of the process and thread profiles to be simulated are generated probabilistically from distributions of the user's choice.

The framework has been built in a modular style so that different thread placement strategies and processor models can be incorporated easily. The simulator incorporates both placement of newly created threads and migration of existing threads.

The software meets all of its functional requirements. It has been thoroughly tested and there are no known bugs. The software performs satisfactorily against its non-functional requirements, and this performance can be readily improved by future developments and extensions to the framework.

Education Use Consent

Acknowledgements

I would like to thank my project supervisor Dr. Wim Vanderbauwhede for his advice and support.

Contents

1 Introduction

An instance of a running computer program is often referred to as a *process*. A process will comprise one or more *threads*, which are activities within the process that can take place concurrently. A central processing unit, or *core*, can only execute one task at a time. In order for several threads to run apparently simultaneously the operating system must manage a time-sharing system, with each thread getting a slice of the core's operating time. This results in the runtime of a thread being longer than it would be if it alone was operating on the core.

The early 2000s saw the development of multi-core processors. The cores operate independently, meaning that different threads can be processed simultaneously (rather than apparently simultaneously). By distributing threads over the available cores the runtimes can be improved; the level of improvement will depend on how the threads need to access memory.

Computer main memory is large but core access is a comparatively slow operation. Cache memory is much smaller but can be accessed more quickly. The cache is used to store copies of recently used data from main memory. The data cache is typically divided into a hierarchy of levels (L1, L2 etc.); cache size usually increases with level, and access times get slower. The nature of the cache hierarchy will depend on the design of the chip.

Multi-core processors invariably implement sharing of higher level cache memory (L1 is commonly exclusive to a particular core); the specifics will again depend on the design of the chip. This means that any core should be able to reach data stored in the cache of another core, but the ease of access will depend on the relative locations of the cores in the cache hierarchy. As a further complication it is possible that caches may hold different local values for the same variable (i.e. the value that should be in main memory), and accesses to obsolete data must be prevented using a coherency protocol.

When distributing threads across a multi-core processor, the strategy of simply minimizing the number of threads per core may be counterproductive if it results in slower memory accessing activity. The implications of the thread location may not be obvious when the number of cores and threads is large. By using simulation the outcomes can be modeled for a range of input scenarios.

1.1 Statement of Problem

1.1.1 Overview

Optimal use of the capacity of a multi-core processor involves intelligent distribution of threads. However, relocation of an existing thread or inappropriate relative placement of coupled threads may have an adverse impact on the efficiency of cache memory accessibility. This becomes difficult to predict when the number of threads and cores gets large.

The development and testing of thread placement algorithms is time consuming. It is preferable to perform simulations of the effectiveness of the placement strategy for a range of inputs before implementation.

Existing simulations of the effects of thread placement on runtime are too detailed and slow, or impose unwanted restrictions (see Section 2).

1.1.2 Aims of the project

The project will produce the framework for a simulation program that can be used to assess the effectiveness of thread placement strategies. It will be capable of modelling the effects of both cpu time sharing and distant cache access.

The simulator will model processor usage at a high level of abstraction so that runtime will be feasible for processors with many thousands of cores. Instead of describing activity at the cycle level, the simulator will calculate the level of activity that would be expected over a longer time step.

The simulator will monitor the progress of generic processes and threads with properties determined from probability distributions.

The design needs to be easily extendable to allow the user to introduce alternative versions of system components.

2 Background Survey

2.1 Process simulations

MGSim [1] is an open source discrete event simulator for multi-core processor architectures, developed in C++. It is initially configured for the MicroGrid architecture [2] but is applicable to others. MGSim operates at the register level, meaning that the operations of a thread must be stipulated; additionally it will become computationally intense as the number of threads increases. The companion application HLSim [3] operates at a higher level of abstraction, but usually requires some output from MGSim.

Multi2Sim [4] is another open source simulator, this time developed in C. It is capable of modeling the cache coherence protocol; again it is cycle accurate, making it too slow for comparing thread placement strategies.

MCCSim [5] is a simulator used to investigate the placement of threads to minimize cache contention in multicore systems. It requires as input memory addresses generated by a specific process and captured using the Pin binary instrumentation tool [6]. CMP$im [7] and the work of Ratanaworabhan [8] provide similar functionality.

Genbrugge et al [9] describe an interval-based approach to architectural simulation. Instead of evaluation at every cycle, the intervals between cache miss events are analyzed as blocks. However, the analysis requires output from an emulator for the specific process.

Sniper [10] also uses an interval model. It is built using the Graphite framework [11], which is an extension of Pin.

AKULA [12] is a Java application described as "a toolset for experimenting and developing thread placement algorithms on multicore systems". It has functionality for the actual implementation of thread placement algorithms, and a so-called bootstrap method for rapid simulation/evaluation. It is the latter that is of interest. A system is modeled as follows.

- A Machine class, which contains
- Chip classes that represent the memory domains; these in turn contain
- Core classes.
- AKULAThread classes, which are mapped onto Cores.

The user specifies all of the threads that will be run during the simulation, along with the time that they start and the time increment of the simulation. The simulation then loops through the following steps until all threads have terminated. The procedure is:

- Calculate the progress of each thread over the time increment.
- Remove any completed threads and capture the performance data.
- Place any new threads generated at the time step.
- Relocate existing threads according to the directions of the placement strategy.

AKULA does not use actual performance data in its calculations which will make it significantly quicker than applications using the cycle-accurate approach. However, its method still requires data capture for a specific process and the approach used will become inconvenient when there are more than two threads on a core.

2.2 Thread Placement Strategies

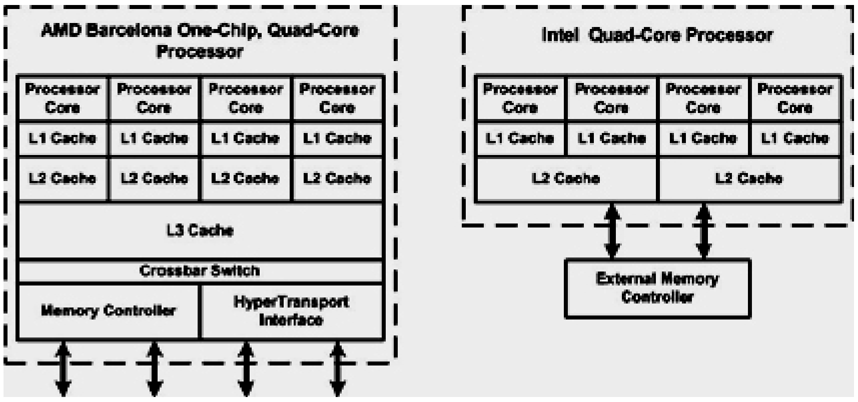
Tousimojarad and Vanderbauwhede [13] compared a placement strategy to the native Linux scheduler on a 64 core TilePro64 processor. The Extended Lowest Load (XLL) strategy places a thread on the core with the lowest activity (this is not the same as the core with the least threads, as some may be sleeping). The speed up is compared for repeated runs of 4 different benchmark problems. The XLL outperforms the Linux strategy for all tests. It is concluded that the Linux strategy performs well for a small number of threads, but does not scale well to a large number of threads and cores. It is also noted that increasing the number of threads does not always improve runtime.

Diener et al. [14] investigated the effect of placing threads according to how often they accessed the same memory addresses. Six benchmark tests were run with two identical processors based on the Intel Nehalem architecture (4 cores per processor; each core has a dedicated L1 and L2 cache while the L3 is shared by all). A performance improvement was observed for almost all tests when using 8 threads, but 16 threads resulted in performance degradation.

Rajagopalan et al. [15] suggest a strategy where the programmer can either specify the core for a thread, or identify which threads are likely to share data. The scheduler will try to group these threads to optimize shared memory, and run the threads at the same time. The method of implementation is suggested but no results are presented.

2.3 Example Cache Architectures

2.3.1 AMD Barcelona and Intel Quad Core

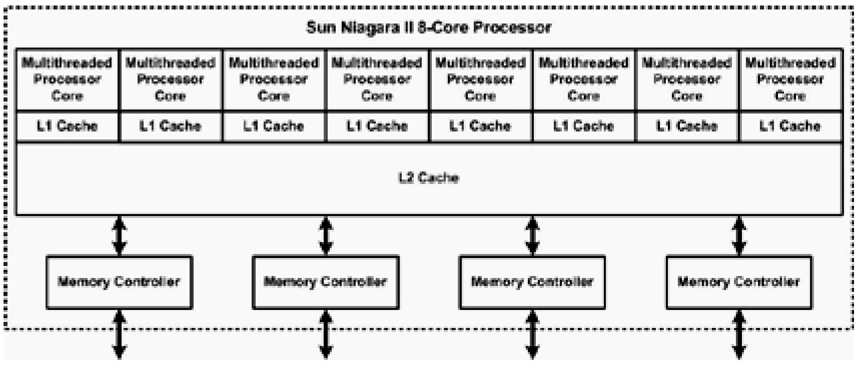


(from [16])

For the Intel Quad Processor each core has its own L1 cache and shares a L2 cache with 1 other core.

For the AMD Barcelona Processor each core has its own L1 and L2 cache, and shares a L3 cache with the other 3 cores.

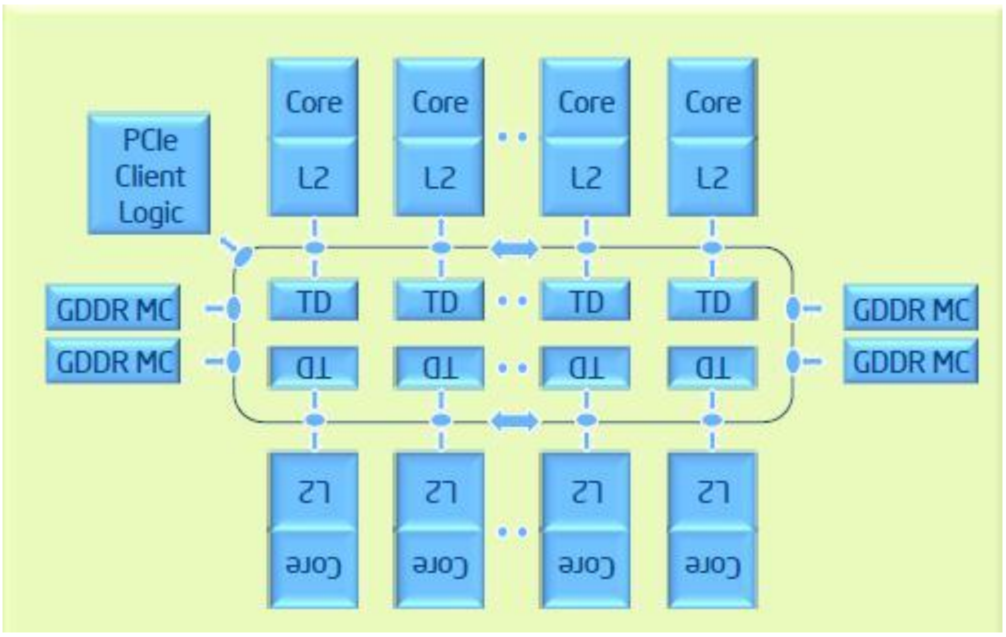
2.3.2 Sun Niagara II



(from [16])

For the Sun Niagara II Processor each core has its own L1 cache and shares a L2 cache with the other 7 cores.

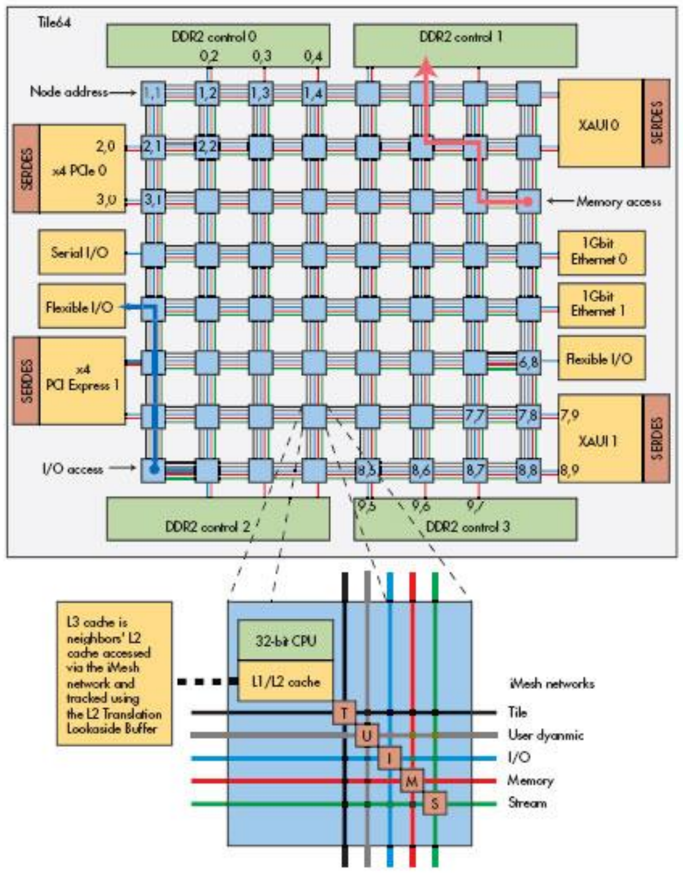
2.3.3 Xeon Phi



(from [17])

For the Xeon Phi Coprocessor each core has its own L1 cache and L2 cache, with the L2 caches connected in a ring.

2.3.4 Tilera



The Tile64 chip contains 64 32-bit cores connected by the iMesh interconnect, which has five independent networks carrying different types of data, including memory and I/O transfers. The L2 caches can be combined into a distributed L3 cache.

For the Tilera each of the 64 ‘tiles’ has its own L1 cache and L2 cache, with the L3 cache created by connection to the neighbouring L2 caches.

3 Requirements

3.1 Requirements Gathering

All requirements were established through discussions with Dr. Wim Vanderbauwhede, who is the proposer of the project and the leader of the research group that will use the simulator.

3.2 Functional Requirements

should indicates a mandatory requirement.

may indicates an optional requirement.

1. The simulator should calculate the running times of threads distributed over cores of a processor.
   - 1.1. The core time a thread receives should be inversely proportional to the number of threads on the core.
   - 1.2. The running time of a thread may be modified by its relationship to other threads in its associated process.
     - 1.2.1. The simulator may provide a means of coupling threads.
     - 1.2.2. The simulator may provide a means of modelling cache access time penalties.

2. The level of thread activity arising during the simulation should be probabilistic.
   - 2.1. The following parameters should be probabilistic:
     - the rate at which new processes are created,
     - the number of threads associated with a process, and
     - the nominal duration of the threads.

3. The simulator should use a thread placement strategy to decide the core upon which a new thread is placed.

4. The simulator may use a thread placement strategy to migrate threads to different cores.

5. The simulator should be flexible.
   - 5.1. The processor model should be defined at runtime
   - 5.2. The thread placement strategy should be defined at runtime.
   - 5.3. The probability distributions used to specify the process and thread profiles should be defined at runtime.
   - 5.4. Limits of the termination criteria should be specified at runtime

6. The simulator should be runnable from the command line.
   - 6.1. The simulator should not require a GUI.
   - 6.2. Input parameters should be read from a configuration text file.

7. Post processing of simulation results may be done using third party software.
   - 7.1. The simulator may output its basic data to text files.

3.3 Non-functional Requirements

8. The simulator should be scalable.
    8.1. The simulator should be able to able to work with processor models with many thousands of cores.

9. The source code should be runnable on Windows and Linux operating systems

10. The source code should be easily extendable.
    10.1. The source code should be well commented.
    10.2. The source code should be neat and easy to read.

11. The simulator should run quickly.
    11.1. Any algorithms should be simple.

12. The simulator should not require a large amount of run-time memory.
    12.1. The simulator will not store all data from each increment.

4 Design

4.1 Design Methodology

The simulator was developed using a phased strategy, with the feasibility of later phases determined by the time remaining. Each phase was an iterative process involving discussion and periodic presentation of source code.

The four phases were:

1. Developing the simulator engine to calculate thread times.
2. Interpreting configuration data from a text file.
3. Implementing a migration strategy.
4. Implementing thread coupling.

4.2 Design Architecture

The overall architecture of the simulator can be described as Model-View-Controller. In this instance the 'view' is a text file (!) as no GUI is required.

4.3 Detailed Design

The simulator development comprised two major stages (phases 1 and 2 in section 3.4). These can be described as

1. Integrating the various controller features to update the model classes appropriately (the simulator engine).
2. Reading input from file and interpreting it to create the requisite classes.

4.3.1 Simulator Engine

This will have many similarities to AKULA, described in Section 2.1.

Since this is the first version of the simulator, a simple advancement engine will be used. That is, the simulator will progress using a fixed time step.

The simulator will have two model classes.

- Thread, which will track running time.
- Core, which will determine how much of the time step a thread receives.

These will both be contained within controller classes.

- Process contains several Threads and retains the coupling relationships between them.
- MemModel contains all of the Cores. It specifies relative distances between Cores and the time penalty incurred when a Thread accesses data from another Core.

The Process classes are in turn contained within an OS class, representing the operating system.

OS uses SimProbabilities (a façade class) to generate processes and threads according to the required profile.

OS also links to a TPStrategy class, which decides how the Threads should be allocated to Cores.

Finally the OS contains an OutputResults object that is used to write the simulation output to file.

OS acts as the overall controller of the simulation. The whole simulation takes place within the function `OS::Run()`.Run increments the timestep, updates the thread runtimes via the cores, calls the migration strategy, decides whether to create new processes (and how many processes to create), and calls the output streams.

```
while(!Complete)
{
    timestep++;

    MemModel->UpdateCores();

    foreach(process)
       TPStrategy->MigrateThreads(process);

    if(Simprobs.CreateNewProcessDecision() == true)
    {
       noOfNewProcesses = SimProbs.GetNoOfProcesses();
      for(i = 1:noOfNewProcesses)
           CreateProcess();
   }

   output.WriteCoreInfo();
}
output.WriteThreadInfo();
```

Figure 4.1 Pseudocode for OS::Run()

The run will terminate on one of two criterion.

The simulation runtime reaches `maxRuntime` (specified in the configuration file).

The simulation has created the maximum number of processes (`maxProcesses` in the configuration file) and all of the threads within these processes have run to completion.

4.3.2 File Reading and Object Creation.

The requirements specify that MemModel, TPStrategy and SimProbabilities must all be defined at runtime. The conventional programming paradigm for handling this scenario is to use factory classes to create instances of objects that derive from a common base class.

As the different implementations are not expected to share much common functionality, both MemModel and TPStrategy will be implemented as abstract classes.

SimProbabilities is a façade class that will contain four instances of classes that define probability distributions. These will all be derived from the abstract base class ProbDist, allowing them to be used interchangeably.

ConfigFileReader stores all data from the configuration file and allocates it to the desired variable on request, calling the factory classes when required.

4.3.3 Abstract Base Classes

The simulator requires polymorphic behavior from three object types. Since the concrete implementations are not expected to share functionality the base classes are implemented as abstract.

MemModel

MemModel is an abstract base class from which all processor memory models should be derived. It provides the following pure virtual functions (i.e. no implementation is provided in the base class).

```
virtual void GetSize(int &noOfRows, int &noOfCols) = 0;
virtual void UpdateCores(double increment) = 0;
virtual double GetMemoryFactor(std::shared_ptr<Core> pSlaveCore,
                               std::shared_ptr<Core> pMasterCore) = 0;
virtual void FindNearestCores(std::list<std::shared_ptr<Core> > &coreList,
                              std::shared_ptr<Core> pCore, int level) = 0;
virtual void OutputThreadCount(OutputResults &res,double time) = 0;
virtual void IteratorReset() = 0;
virtual std::shared_ptr<Core> IteratorNext() = 0;
virtual std::shared_ptr<Core> IteratorFirst() = 0;
virtual bool IteratorComplete() = 0;
```

Figure 4.2 MemModel function declarations

`UpdateCores` is called to apply the input increment to all Cores in the memory model.

`GetMemoryFactor` is used to determine the cost of accessing data on the second core from the first.

`OutputThreadCount` is used to write core load data to file.

Iterator functionality is offered to allow sequential access to each core held by the memory model.

`FindNearestCores` provides a means of accessing a subset of cores specified by the integer *level*. It is left to the model developer to decide what *level* should mean (e.g. common cache level).

TPStrategy

TPStrategy is an abstract base class from which all thread placement strategies should be derived. It provides the following pure virtual functions.

```
virtual void AllocateThreadToCore(Process *process,
                                  std::shared_ptr<Thread> pThread, OS *os) = 0;

virtual void MigrateThreads(Process *process, OS *os) = 0;
```

Figure 4.3 TPStrategy function declarations

AllocateThreadToCore is used to find a core for a newly created thread.

MigrateThreads is used to move a thread already on a core.

Both calls receive a pointer to the calling Process, which will update the Thread memoryFactors.

MigrateThreads also receives a pointer to the OS, from which all processes and cores can be accessed.

ProbDist

ProbDist is an abstract base class from which all processor memory models should be derived. The distributions can be used interchangeably; it is left to the user to decide whether the distribution is appropriate for the task.

```
virtual int GetNextInt() = 0;
virtual double GetNextDouble() = 0;
virtual int GetNextIntGEZero() = 0;
virtual double GetNextDoubleGEZero() = 0;
virtual int GetNextIntGTZero() = 0;
virtual double GetNextDoubleGTZero() = 0;

int ProbDist::RounddoubleToInt(double f);
```

Figure 4.4 ProbDist function declarations

For some applications the value required from the distribution is an integer, while in other cases it is a double. In many instances the concrete instances will simply cast their natural output as required.

The class also includes a function[1] to round doubles to integers which, somewhat surprisingly, is not provided by the Standard Library.

[1] http://stackoverflow.com/questions/485525/round-for-double-in-c

4.3.4 Thread coupling and memoryFactor

In order to model the effect of cache distance the Thread objects need to be coupled. The coupling will be expressed as the proportion of time each thread spends accessing data from another thread (including itself). The way in which the threads are coupled is hard coded in the Process object and follows the simple pattern shown in Figure 4.1. The numbers in the boxes are the coupling ratios.

Thread 1 accesses its own data 100% of the time, i.e. does not require access to any other thread.

All other threads: accesses its own data 50% of the time and accesses Thread 1 the other 50%.

		Master Thread			
		1	**2**	**3**	**4**
Slave Thread	**1**	1	0	0	0
	2	0.5	0.5	0	0
	3	0.5	0	0.5	0
	4	0.5	0	0	0.5

Table 4.1 Coupling matrix for four threads

The Process object stores this information in a map, so that it is not dependent on the order in which the Threads are stored in the Process object.

```
std::map<std::pair<Thread*,Thread*>, double> coupling;
```

When a Thread is first created, or when it is migrated, it is necessary to calculate its memoryFactor. This is the penalty it incurs due to the relative positioning of its master thread.

memoryFactor = $\sum$ couplingRatio*memoryPenalty

where memoryPenalty is obtained from the MemModel based on the relative distance between cores.

5 Implementation

5.1 Choice of Language

The simulator will be developed in C++, which is a language commonly used by the research group for whom the application is being developed.

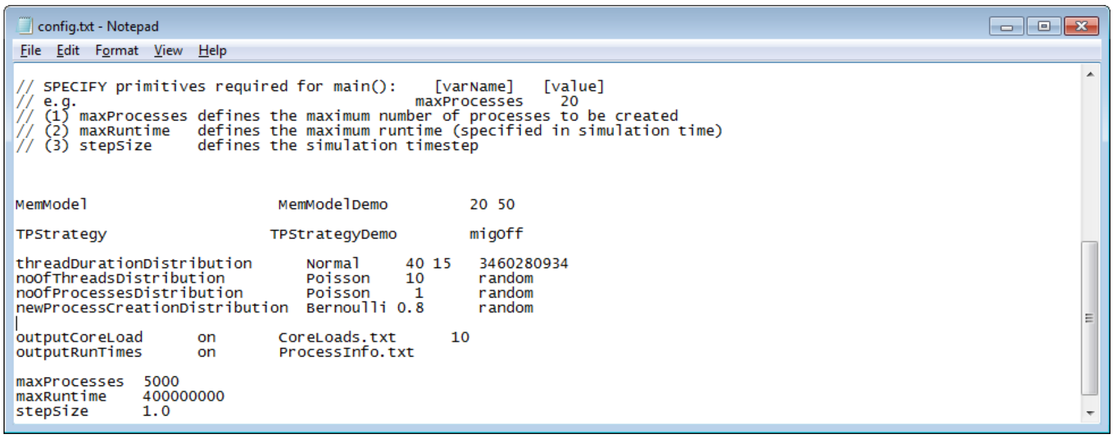
The code will us constructs from the C++ Standard Library, which is platform independent. In particular it makes use of *std::vector*, which is a managed array structure.

Unlike languages such as Java or C#, C++ does not provide automatic garbage collection for heap allocated memory. This creates a risk of memory leak, where pointers are reallocated before memory is released, and in some cases there is ambiguity about which part of the code should be responsible for deallocation. Since C++11[2] the standard library has offered *smart pointers*, which alleviate some of these difficulties. The simulator makes use of the *shared_ptr* variant where heap allocated objects are held as member variables. The raw memory-allocating pointer is wrapped with a *shared_ptr*; the *shared_ptr* keeps track of the number of copies of itself in existence and deallocates the memory when the last copy is destroyed.

5.2 File I/O

5.2.1 Configuration File

Configuration details are read from the text file *config.txt* located in the same directory as the application. An example file is shown in Figure 5.2



```
// SPECIFY primitives required for main():    [varName]   [value]
// e.g.                                       maxProcesses    20
// (1) maxProcesses defines the maximum number of processes to be created
// (2) maxRuntime   defines the maximum runtime (specified in simulation time)
// (3) stepSize     defines the simulation timestep


MemModel                     MemModelDemo        20 50

TPStrategy                  TPStrategyDemo       migOff

threadDurationDistribution      Normal     40 15   3460280934
noOfThreadsDistribution         Poisson    10      random
noOfProcessesDistribution       Poisson     1      random
newProcessCreationDistribution  Bernoulli 0.8      random

outputCoreLoad      on      CoreLoads.txt      10
outputRunTimes      on      ProcessInfo.txt

maxProcesses  5000
maxRuntime    400000000
stepSize      1.0
```

Figure 5.1: Example of config.txt

The file should contain one line of data for each entity in the simulator that requires configuring. The current implementation requires 11 lines:

- a memory model,
- a thread placement strategy,

[2] A version of the C++ standard approved in 2011.

- 4 probability distributions for creating processes and threads,
- 2 output filestreams, and
- 3 numerical values specifying the runtime and resolution of the simulation.

The first entry on the line is a string that will be used as an identifier by the program. Additional information required to construct the entity associated with the identifier is entered on the same line and separated with an arbitrary amount of white space.

For example, using the example in Figure 5.2, when tpsim tries to construct a MemModel it requires a specific class name ("MemModelDemo") and the constructor parameters required for that specific class (in this case two integers).

The constructor parameters required for each class are described in subsequent sections.

Lines beginning with double slash "//" and blank lines are ignored.

5.2.2 Results Output

The simulator offers two output streams, with parameters defined in the config file. Both can be deactivated and will write to the filename specified. Both, when activated, begin with a header that contains

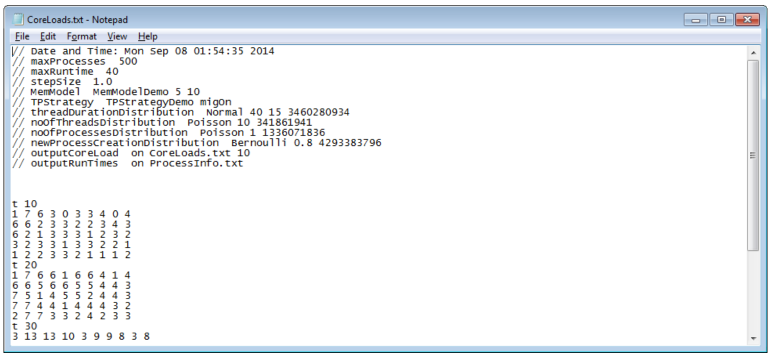
- the start time and date
- the simulation configuration data

The configuration data will be the same as that contained in the config file **except** that if a random seed has been specified for a probability distribution the actual seed value used will replace the string "random".

The last line in the file is the end time and date.

`outputCoreLoad` is the filestream for the number of threads on each core. The output interval (number of timesteps between outputs) must be specified. The format of the output is

- t timestep,
- number of threads per core, in a format specified by the `OutputThreadCount()` function of the selected MemModel.



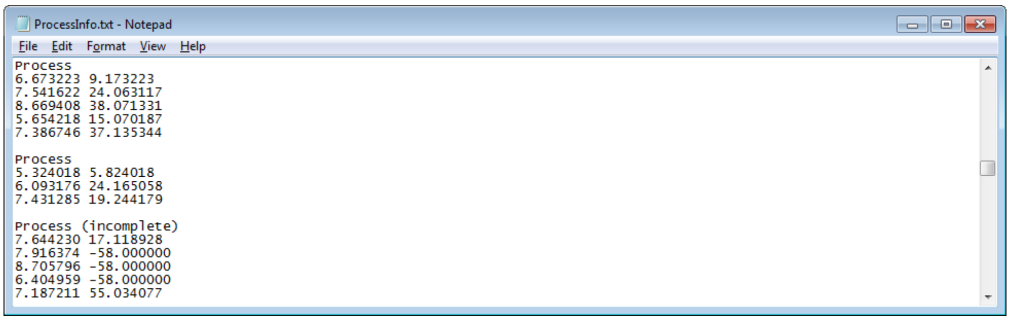
```
// Date and Time: Mon Sep 08 01:54:35 2014
// maxProcesses  500
// maxRuntime  40
// stepSize  1.0
// MemModel  MemModelDemo 5 10
// TPStrategy  TPStrategyDemo migOn
// threadDurationDistribution  Normal 40 15 3460280934
// noOfThreadsDistribution  Poisson 10 341861941
// noOfProcessesDistribution  Poisson 1 1336071836
// newProcessCreationDistribution  Bernoulli 0.8 4293383796
// outputCoreLoad  on CoreLoads.txt 10
// outputRunTimes  on ProcessInfo.txt



t 10
1 7 6 3 0 3 3 4 0 4
6 6 2 3 3 2 2 3 4 3
6 2 1 3 3 3 1 2 3 2
3 2 3 3 1 3 3 2 2 1
1 2 2 3 3 2 1 1 1 2
t 20
1 7 6 6 1 6 6 4 1 4
6 6 5 6 6 5 5 4 4 3
7 5 1 4 5 5 2 4 4 3
7 7 4 4 1 4 4 4 3 2
2 7 7 3 3 2 4 2 3 3
t 30
3 13 13 10 3 9 9 8 3 8
```

Figure 5.2 outputCoreLoad filestream

`outputRunTimes` is the filestream for the thread durations. The format of the output is

```
Process
Nominal duration1        Actual duration1
Nominal duration2        Actual duration2
…
Process
etc
```

Processes and threads are output in the order they were created. If a thread is incomplete at the end of the simulation its actual duration is listed as a negative number and the Process heading is followed by the string "(incomplete)".



```
Process
6.673223 9.173223
7.541622 24.063117
8.669408 38.071331
5.654218 15.070187
7.386746 37.135344

Process
5.324018 5.824018
6.093176 24.165058
7.431285 19.244179

Process (incomplete)
7.644230 17.118928
7.916374 -58.000000
8.705796 -58.000000
6.404959 -58.000000
7.187211 55.034077
```

Figure 5.3 outputRunTimes filestream

5.3 Class Descriptions

5.3.1 Main Function

The main function of the simulator is implemented in the file tpsim.cpp. It can be broken down into two sections.

- Creating objects based on the parameters in the configuration file.
- Using these objects to build a notional operating system (OS) which runs the simulation .

5.3.2 ConfigFileReader

This class reads the configuration file specified in main(). Each line of the file is tokenized using white space. The data is then stored in a map structure. The map key is the first string on the line. The map value is a vector of all other strings. Lines beginning with double slash "//" and blank lines are ignored.

The strings associated with (for example) the identifier varID can be obtained directly by calling the function *ExtractParameters*("varID", vectorOut).

The parameters can also be assigned directly. For example

```
double stepSize = 0;

config.AssignParameter("stepSize", stepSize, errMess)
```

will search for the map key stepSize, parse the corresponding value to a double and assign it to stepSize (passing by reference).

Where required, AssignParameter will call the necessary factory class.

AssignParameters returns a bool indicating whether or not the assignment was believed to be successful. Any generated error messages are appended to the string input parameter errMess (again returned by reference).

5.3.3 Factory Classes

The classes used to define the memory model, thread placement strategy and probability distributions are decided at runtime. The conventional programming paradigm for handling this scenario is to use factory classes. All three factory classes are of a traditional format which is easily extendable to include new classes.

New objects are created by calling the static Create function, which takes a vector of strings as input. The function matches the first string using *if* conditionals and then parses the remaining strings in accordance with expectations of the particular constructor. This requires tedious error checking that needs to be customized for each particular case.

The Create function returns a bool indicating whether or not the operation was believed to be successful. Any generated error messages are appended to the string input parameter errMess.

5.3.4 MemModelDemo

`MemModelDemo` is a concrete implementation of MemModel that has been developed for the purposes of demonstration and testing. It does not model a real processor and its characteristics do not represent those that would be observed when testing actual hardware. It uses what may be described as a Manhattan model (from the regular parallel-perpendicular street pattern of its namesake), where cache access is a simple function of the physical distance between cores.

In the config file `MemModelDemo` requires a line in the following format, where rows and columns pertain to the core arrangement.

```
MemModel          MemModelDemo      noOfRows   noOfColumns
```

All core pointers are held in a 1 dimensional vector. Each core is given an index number to simplify searching. For this model the index equals its array position + 1.

`MemModelDemo::FindNearestCores(coreList,p,level)` returns (by reference) a list of cores that are 'level' steps away from the input core p. For example, referring to figure 5.5, setting level to 1 returns the red cores in the order indicated by the numbering. A level value of 2 returns the green cores in numerical order. Level 3 returns the blue cores, for which there will be 2 less than expected due to the left and bottom edges.

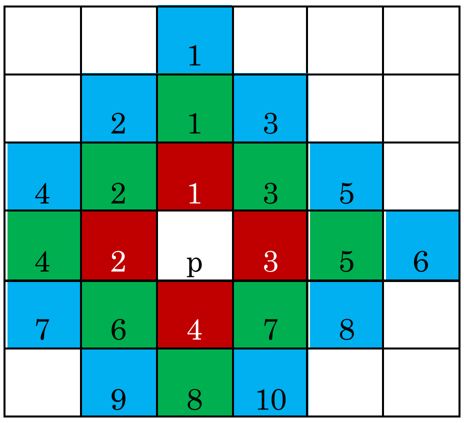


Figure 5.4 Cores returned by MemModelDemo for 3 different levels.

MemModelDemo calculates the memory access penalty using the algorithm shown in Figure 5.6 (cols is the number of columns of cores on the processor). The penalty is inversely proportional to the physical distance between cores. The primary attraction of modelling the memory factor in this way is infinite scalability.

```
int core = slaveCore->GetCoreNo();
slaveRow = (core-1)/cols;
slaveCol = (core-1)%cols;

int core = masterCore->GetCoreNo();
masterRow = (core-1)/cols;
masterCol = (core-1)%cols;

int dist = abs(slaveRow - masterRow) + abs(slaveCol - masterCol);

return (double)1.0/(dist + 1);
```

Figure 5.5 Calculation of MemoryFactor by MemModelDemo.

5.3.5 TPStrategyDemo

`TPStrategyDemo` is a concrete implementation of TPStrategy that has been developed for the purposes of demonstration and testing. It does not purport to be a good strategy (it clearly isn't) but it searches the processor in a way that a real strategy might. It also generates some interesting patterns in Section 6.

In the config file `TPStrategyDemo` requires a line in the following format, where the string `migOn` will enable the migration strategy and any other string will disable it.

```
TPStrategy       TPStrategyDemo        migOn
```

The implementation of the AllocateThreadToCore method is convoluted.

If the thread is not coupled to any other thread (i.e. it is a master thread) then it will be allocated to the idlest core. This is the core with the lowest load. That is, the lowest load on itself, then the lowest combined load on its level 1 surrounding cores (see Section 5.3.4). If there is more than one core with minimal loading then the first one encountered is selected. If the search finds a core where the core itself and its level 1 surrounding cores are all empty then the search is stopped.

If the thread is a slave thread then its dominant master thread is identified. The level 1 and level 2 cores surrounding the master thread core are then identified. The thread will then be allocated to the lowest loaded of these cores. If more than one core has equal low loading then it will be allocated to the first one according to the numerical order shown in Figure 5.4 (level 1 first then level 2).

The objective of the MigrateThreads method is, perversely, to move the Process master thread away from its other threads and onto the lightest loaded core closest to the last processor core. Ideally the thread should only be migrated once; this is done by checking to see if the memory penalty between the master and second thread is equal to 0.5 (i.e. they are on adjacent cores).

```
shared_ptr<Core> firstCore = (*firstThread)->GetCore();
Process::iterator nextThread = process->begin() + 1;
shared_ptr<Core> secondCore = (*nextThread)->GetCore();;
if(pMemModel->GetMemoryFactor(firstCore,secondCore) == 0.5)
{
     shared_ptr<Core> bestCore = pMemModel->IteratorFirst();
     // Iterate through all cores and find the latest one with low load
      while(!pMemModel->IteratorComplete())
      {
            nextCore = pMemModel->IteratorNext();
            double load = nextCore->GetActivityLevel();
            if(load <= minLoad)
            {
                 bestCore = nextCore;
                 minLoad = load;
            }
       }

       // Move the thread and update the memory penalties
       firstCore->RemoveThread(*firstThread);
       bestCore->AddThread(*firstThread,bestCore);
       process->UpdatePenalties(pMemModel);
 }
```

Figure 5.6 Pseudocode for TPStrategyDemo::MigrateThreads

5.3.6 Thread

The purpose of the thread model class is to track

- The amount of processing time the thread requires before completion (`remainingDuration`),
- The amount of (simulated) time the thread takes to reach completion (`runTime`).

```
1.  double UpdateThread(double cpuTime, double prevCpuTime, double timeStep)
2.  {
3.    // Reduce coreTime in accordance with memory penalty of thread
4.    double processTime = cpuTime*memoryFactor;
5.    double leftoverTime = 0;

6.    // if prevCpuTime > 0 need to correct values from previous visit
7.    if(prevCpuTime > 0)
8.    {
9.          runTime = runTime - timeStep;
10.         remainingDuration= remainingDuration + prevCpuTime*memoryFactor;
11.   }

12.   // Check if thread finishes in this visit. If so, rescale increment
13.   if(remainingDuration > processTime)
14.   {
15.          runTime = runTime + timeStep;
16.          leftoverTime = 0;
17.          remainingDuration = remainingDuration - processTime;
18.   }
19.   else
20.   {
21.          runTime = runTime + (remainingDuration/memoryFactor);
22.          leftoverTime = cpuTime - (remainingDuration/memoryFactor);
23.          remainingDuration = 0;
24.    }

25.    // If thread finished return leftover cpu time
26.    return leftoverTime;
27. }
```

Figure 5.7 Thread::UpdateThread method

At each `timeStep` an incomplete thread is allocated some `cpuTime` by the core on which it resides.

If the thread accesses data from a thread on a different core then its effective **`processTime`** is reduced by the **`memoryFactor`** (line 4).

At each function call the `remainingDuration` of the thread is reduced by the **`processTime`** and the `runTime` is increased by `timeStep` (lines 15-17).

Since this is the first implementation of the simulator it will be subjected to a high level of scrutiny and testing to increase confidence in its integrity.

If the thread finishes within its allocated `cpuTime` it will only increase the `runTime` by the time actually used, and return the unused time to the core (lines 21-23).

Further, the Core may choose to revisit its threads to redistribute time returned by other completed threads (lines 9 – 10).

Both of these if conditionals (though rarely engaged) will increase the amount of processing done by the simulator, and may be in excess of the level of accuracy required.

Each thread holds a pointer to the Core on which it resides. This is useful for two purposes.

- When migrating a thread to a new core it can be removed easily from its old core (the implemented migration strategy makes its decisions via the Process, which has no knowledge of the thread-core relationship).
- The memory penalty between coupled threads can be obtained simply.

5.3.7 Core

The purpose of the Core model is to update the status of all Threads in its List at each time step.

```
1.  void Core::UpdateCore(double increment)
2.  {
3.     // Core time is shared evenly between all threads on the core.
4.     double coreTime =
                    (GetNoOfThreads() > 0) ? increment/GetNoOfThreads() : 0;
5.     double prevCoreTime = 0;
6.     double leftoverTime = coreTime;
7.
8.     while(leftoverTime > 0 && GetNoOfThreads() > 0)
9.     {
10.        leftoverTime = 0;
11.      // Loop through all threads, removing them if they are complete
12.      for(list<shared_ptr<Thread> >::iterator thread = threads.begin();
                                                thread != threads.end();)
13.      {
14.         // If the thread completed this step, capture the unused time
15.         leftoverTime =leftoverTime +
                   (*thread)->UpdateThread(coreTime,prevCoreTime,increment);

16.         if((*thread)->IsComplete())
17.         {
18.               shared_ptr<Thread> deadThread = *thread;
19.               ++thread;
20.               RemoveThread(deadThread);
21.         }
22.         else
23.         {
24.               ++thread;
25.         }
26.      }
27.      // Loop again if there was unused time
28.      if((leftoverTime > 0) && (GetNoOfThreads() > 0))
29.      {
30.           prevCoreTime = coreTime;
31.           coreTime = coreTime + (leftoverTime/GetNoOfThreads());
32.      }
33.    }
34. }
```

Figure 5.10 Core::UpdateCore method

The Core distributes the step time evenly between all of its Threads.

Each Thread is Updated (line 15) and removed from the List if it is complete (lines 18-20).

The loop will reiterate if time is returned to the core by a completed Thread. This level of accuracy may be regarded as overkill, as discussed with regard to Thread.

5.3.8 ProbDist Variants

Most of the distributions require a random number generator. This in turn requires a seed value. In the configuration file the `seedval` can either be specified explicitly as an unsigned integer, or created by the system random device by entering the string “`random`”.

ProbDistBernoulli

This is a wrapper class for the template `std:: bernoulli_distribution`, which returns true or false (cast to 1 or 0).

In the config file `ProbDistBernoulli` requires a line in the following format.

```
targetDistribution  Bernoulli prob seedval
```

`targetDistribution` is the name of the distribution in the main function that the distribution is to be attached to.

`prob` is an input of type double that represents the probability of the distribution returning 1. `prob` must be 0 <= `prob` <= 1.

Function	return = 0
`GetNextIntGEZero()`	0
`GetNextDoubleGEZero()`	0
`GetNextIntGTZero()`	1
`GetNextDoubleGTZero()`	1

ProbDistNormal

This is a wrapper class for the template `std:: normal_distribution`, which returns a double. This class rounds the return to an integer rather than casting it.

In the config file `ProbDistNormal` requires a line in the following format.

```
targetDistribution  Normal mean stDev seedval
```

`targetDistribution` is the name of the distribution in the main function that the distribution is to be attached to.

`mean` and `stDev` define the properties of the distribution. `stDev` must be > 0.

Function	return < 0	return <= 0
`GetNextIntGEZero()`	Retry until valid	0
`GetNextDoubleGEZero()`	Retry until valid	0
`GetNextIntGTZero()`	Retry until valid	Retry until valid
`GetNextDoubleGTZero()`	Retry until valid	Retry until valid

The function will loop without limit until a valid return is obtained.

ProbDistPoisson

This is a wrapper class for the template `std::poisson_distribution`, which returns an integer.

In the config file `ProbDistPoisson` requires a line in the following format.

```
targetDistribution  Poisson mean seedval
```

`targetDistribution` is the name of the distribution in the main function that the distribution is to be attached to.

`mean` defines the properties of the distribution. `mean` must be >= 0

Function	return = 0
`GetNextIntGEZero()`	0
`GetNextDoubleGEZero()`	0

`GetNextIntGTZero()`	retry
`GetNextDoubleGTZero()`	retry

ProbDistConstant

This allows the user to specify a constant double output.

In the config file `ProbDistConstant` requires a line in the following format.

```
targetDistribution  Constant const
```

`targetDistribution` is the name of the distribution in the main function that the distribution is to be attached to.

`const` defines the value to be returned.

Function	const < 0	const <= 0
`GetNextIntGEZero()`	0	0
`GetNextDoubleGEZero()`	0	0
`GetNextIntGTZero()`	1	0
`GetNextDoubleGTZero()`	eps	0

6 Evaluation

6.1 Unit Testing

The source code has been tested at unit level using Microsoft Visual Studio's CppUnitTestFramework. It was subjected to (and passed) 167 unit tests. A breakdown of the tests by class is shown in Table 5.1.

Since the simulator is intended to be flexible (i.e. the models have variable parameters) and because it uses changing probability distributions, it cannot be tested exhaustively. Where possible, variables were changed between tests so that many different configurations were run and the chances of encountering issues were increased.

Class	**Number of Unit Tests**
ConfigFileReader	28
Core	11
FactoryMemModel	4
FactoryProbDist	24
FactoryTPStrategy	3
MemModelDemo	29
OutputResults	8
ProbDistBernoulli	6
ProbDistConstant	12
ProbDistNormal	6
ProbDistPoisson	6
Process	5
Thread	13
TPStrategyDemo	12

Table 5.1 Breakdown of unit tests by class

Classes OS and SimProbabilities are tested at system level.

6.2 System Testing

The system testing will serve two purposes.

1. To test features not covered by unit testing, such as file I/O
2. To demonstrate that the behaviour of the program corresponds to that of the individual tested units.

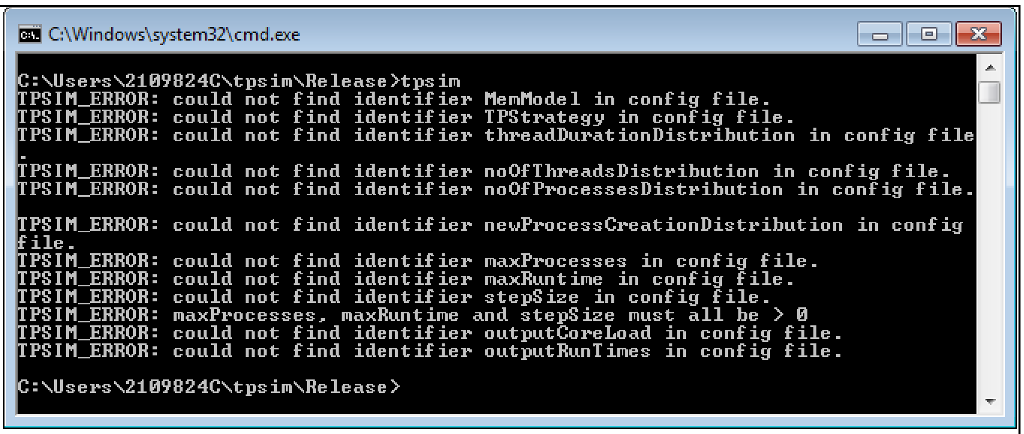
The simulator was subjected to (and passed) 30 system test.

As in the unit testing input variables were changed arbitrarily as much as possible to increase the chance of encountering errors.

Test ID	1
Requirement	The program will trap bad input data and display a warning message to console
Preconditions	
Procedure	1. Remove config.txt from the working directory. 2. Run tpsim
Expected	Console displays `TPSIM_ERROR: failed to open config file.`
Actual	As expected
Comments	

Test ID	2
Requirement	The program will trap bad input data and display a warning message to console
Preconditions	
Procedure	1. Open config.txt and remove all data entries. 2. Run tpsim

Expected	Console displays TPSIM_ERROR message for each missing item:
	`TPSIM_ERROR: could not find identifier MemModel in config file.`
	`TPSIM_ERROR: could not find identifier TPStrategy in config file.`
	`TPSIM_ERROR: could not find identifier threadDurationDistribution in config file.`
	`TPSIM_ERROR: could not find identifier noOfThreadsDistribution in config file.`
	`TPSIM_ERROR: could not find identifier noOfProcessesDistribution in config file.`
	`TPSIM_ERROR: could not find identifier newProcessCreationDistribution in config file.`
	`TPSIM_ERROR: could not find identifier maxProcesses in config file.`
	`TPSIM_ERROR: could not find identifier maxRuntime in config file.`
	`TPSIM_ERROR: could not find identifier stepSize in config file.`

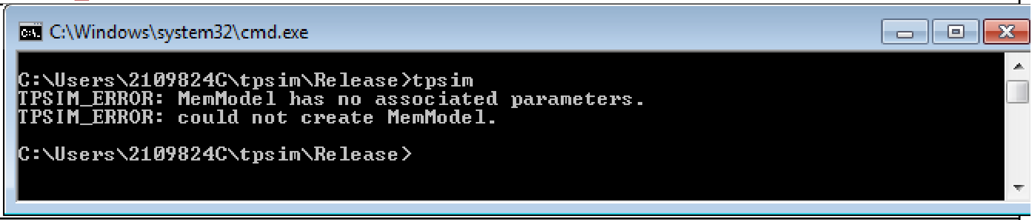
Actual	

```
C:\Users\2109824C\tpsim\Release>tpsim
TPSIM_ERROR: could not find identifier MemModel in config file.
TPSIM_ERROR: could not find identifier TPStrategy in config file.
TPSIM_ERROR: could not find identifier threadDurationDistribution in config file
.
TPSIM_ERROR: could not find identifier noOfThreadsDistribution in config file.
TPSIM_ERROR: could not find identifier noOfProcessesDistribution in config file.

TPSIM_ERROR: could not find identifier newProcessCreationDistribution in config
file.
TPSIM_ERROR: could not find identifier maxProcesses in config file.
TPSIM_ERROR: could not find identifier maxRuntime in config file.
TPSIM_ERROR: could not find identifier stepSize in config file.
TPSIM_ERROR: maxProcesses, maxRuntime and stepSize must all be > 0
TPSIM_ERROR: could not find identifier outputCoreLoad in config file.
TPSIM_ERROR: could not find identifier outputRunTimes in config file.

C:\Users\2109824C\tpsim\Release>
```

	Output files are not updated.
Comments	Displays extra TPSIM_ERROR message: `TPSIM_ERROR: maxProcesses, maxRuntime and stepSize must all be > 0` Acceptable nuisance bug

Test ID	3
Requirement	The program will trap bad input data and display a warning message to console
Preconditions	Set config to the following (MemModel has no parameters): MemModel TPStrategy TPStrategyDemo migOn threadDurationDistribution Normal 20 6 3401862510 noOfThreadsDistribution Constant 19 noOfProcessesDistribution Poisson 0.1 2951710052 newProcessCreationDistribution Bernoulli 0.567 84448526 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 20 maxRuntime 100 stepSize 1.0
Procedure	1. Run tpsim

Expected	Console displays TPSIM_ERROR message for missing MemModel data:
	`TPSIM_ERROR: MemModel has no associated parameters.`
	`TPSIM_ERROR: could not create MemModel.`

Actual	

```
C:\Users\2109824C\tpsim\Release>tpsim
TPSIM_ERROR: MemModel has no associated parameters.
TPSIM_ERROR: could not create MemModel.

C:\Users\2109824C\tpsim\Release>
```

Comments	This is a single system test used to demonstrate that the unit tests for error trapping are valid at system level.

Test ID	4
Requirement	The program will trap bad input data and display a warning message to console
Preconditions	Set config to the following (maxProcesses < 0): MemModel MemModelDemo 75 200 TPStrategy TPStrategyDemo migOn threadDurationDistribution Normal 20 6 3401862510

	noOfThreadsDistribution Constant 19 noOfProcessesDistribution Poisson 0.1 2951710052 newProcessCreationDistribution Bernoulli 0.567 84448526 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses -1 maxRuntime 100 stepSize 1.0
Procedure	1. Run tpsim
Expected	Console displays TPSIM_ERROR message for invalid maxProcesses data: `TPSIM_ERROR: maxProcesses, maxRuntime and stepSize must all be > 0`
Actual	As expected
Comments	

Test ID	5
Requirement	The program will trap bad input data and display a warning message to console
Preconditions	Same data as test 4 EXCEPT: change maxProcesses to 0
Procedure	1. Run tpsim
Expected	Console displays TPSIM_ERROR message for invalid maxProcesses data: `TPSIM_ERROR: maxProcesses, maxRuntime and stepSize must all be > 0`
Actual	As expected
Comments	

Test ID	6
Requirement	The program will trap bad input data and display a warning message to console
Preconditions	Same data as test 4 EXCEPT: change maxProcesses to 10 change maxRuntime to -5
Procedure	1. Run tpsim
Expected	Console displays TPSIM_ERROR message for invalid maxRuntime data: `TPSIM_ERROR: maxProcesses, maxRuntime and stepSize must all be > 0`
Actual	As expected
Comments	

Test ID	7
Requirement	The program will trap bad input data and display a warning message to console
Preconditions	Same data as test 4 EXCEPT: change maxProcesses to 10 change maxRuntime to 0
Procedure	1. Run tpsim
Expected	Console displays TPSIM_ERROR message for invalid maxRuntime data: `TPSIM_ERROR: maxProcesses, maxRuntime and stepSize must all be > 0`
Actual	As expected
Comments	

Test ID	8

Requirement	The program will trap bad input data and display a warning message to console
Preconditions	Same data as test 4 EXCEPT: change maxProcesses to 10 change stepSize to -0.001
Procedure	1. Run tpsim
Expected	Console displays TPSIM_ERROR message for invalid stepSize data: `TPSIM_ERROR: maxProcesses, maxRuntime and stepSize must all be > 0`
Actual	As expected
Comments	

Test ID	9
Requirement	The program will trap bad input data and display a warning message to console
Preconditions	Same data as test 4 EXCEPT: change maxProcesses to 10 change stepSize to 0
Procedure	1. Run tpsim
Expected	Console displays TPSIM_ERROR message for invalid stepSize data: `TPSIM_ERROR: maxProcesses, maxRuntime and stepSize must all be > 0`
Actual	As expected
Comments	

Test ID	10
Requirement	The program will terminate either when the maximum runtime is reached or when the maximum number of processes have run to completion
Preconditions	Set config to the following: MemModel MemModelDemo 8 43 TPStrategy TPStrategyDemo migOff threadDurationDistribution Constant 20 noOfThreadsDistribution Constant 8 noOfProcessesDistribution Constant 1 newProcessCreationDistribution Constant 10 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 100 maxRuntime 5 stepSize 1.0
Procedure	1. Run tpsim
Expected	Console displays : `tpsim complete.` Output file CoreLoads.txt contains data from 5 timesteps.
Actual	As expected
Comments	Thread runtimes will definitely exceed maxRuntime

Test ID	11
Requirement	The program will terminate either when the maximum runtime is reached or when the maximum number of processes have run to completion
Preconditions	Same data as test 10 EXCEPT: change stepSize to 2.0

Procedure	1. Run tpsim
Expected	Console displays : `tpsim complete.` Output file CoreLoads.txt contains data from 2 timesteps.
Actual	As expected
Comments	maxRuntime is not an exact multiple of stepSize

Test ID	12
Requirement	The program will terminate either when the maximum runtime is reached or when the maximum number of processes have run to completion
Preconditions	Set config to the following: MemModel MemModelDemo 5 9 TPStrategy TPStrategyDemo migOff threadDurationDistribution Constant 3 noOfThreadsDistribution Constant 2 noOfProcessesDistribution Constant 2 newProcessCreationDistribution Constant 1 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 8 maxRuntime 100 stepSize 1.0
Procedure	1. Run tpsim
Expected	Console displays : `tpsim complete.` Output file CoreLoads.txt contains data from 8 timesteps.
Actual	As expected
Comments	2 processes created every step; 8 processes after 4 steps Each process has 1 child thread with memoryfactor 0.75; all threads have ideal duration 3 so child thread takes 4 timesteps Last to finish will be child threads created at step 4 Hence duration should be 8 steps.

Test ID	13
Requirement	The program will allow the user to specify the probability distribution of the thread durations at runtime
Preconditions	Set config to the following: MemModel MemModelDemo 76 22 TPStrategy TPStrategyDemo migOff threadDurationDistribution Normal 20.5 3 `2515210734` noOfThreadsDistribution Poisson 8 `1907021853` noOfProcessesDistribution Poisson 1 `945653224` newProcessCreationDistribution Bernoulli 0.9 `2958845284` outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 200 maxRuntime 100000 stepSize 1.0

Procedure	1. Run tpsim 2. Perform a normal distribution fit to the data in ProcessInfo.txt (e.g. using Matlab)
Expected	Mean 20.5 stdev 3
Actual	Mean 20.5195 Stdev 2.93694
Comments	

Test ID	14
Requirement	The program will allow the user to specify the probability distribution of the thread durations at runtime
Preconditions	Same data as test 13 EXCEPT: change ThreadDurationDistribution to: Constant 5.55
Procedure	1. Run tpsim 2. Inspect ProcessInfo.txt
Expected	Thread durations should all be 5.55
Actual	As expected (with additional zeros padding)
Comments	

Test ID	15
Requirement	The program will allow the user to specify the probability distribution of the number of threads created at runtime
Preconditions	Set config to the following: MemModel MemModelDemo 9 44 TPStrategy TPStrategyDemo migOff threadDurationDistribution Normal 15.654 2.009 364154164 noOfThreadsDistribution Poisson 3 3970788114 noOfProcessesDistribution Poisson 7 2411825581 newProcessCreationDistribution Bernoulli 0.76 3902872731 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 350 maxRuntime 100000 stepSize 1.0
Procedure	1. Run tpsim 2. Perform a Poisson distribution fit to the data in ProcessInfo.txt (e.g. using Matlab)
Expected	Lambda 3
Actual	Lambda 3.09429
Comments	

Test ID	16
Requirement	The program will allow the user to specify the probability distribution of the number of threads created at runtime
Preconditions	Same data as test 15 EXCEPT: change noOfThreadsDistribution to: Normal 10.5 2.2 3314140055

Procedure	1. Run tpsim 2. Perform a Normal distribution fit to the data in ProcessInfo.txt (e.g. using Matlab)
Expected	Mean 10.5 stdev 2.2
Actual	Mean 10.4714 Stdev 2.26169
Comments	Demonstrates that continuous distributions can be used to generate discrete values

Test ID	17
Requirement	The program will allow the user to specify the probability distribution of the number of processes created at runtime
Preconditions	Set config to the following: MemModel MemModelDemo 22 7 TPStrategy TPStrategyDemo migOff threadDurationDistribution Constant 1000 noOfThreadsDistribution Constant 1 noOfProcessesDistribution Poisson 4.5 1965625379 newProcessCreationDistribution Bernoulli 0.77 2046550299 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 318 maxRuntime 900 stepSize 1.0
Procedure	1. Run tpsim 2. Perform a Poisson distribution fit to the changes in the thread numbers in CoreLoads.txt (e.g. using Matlab). Remember to remove zeros from the distribution.
Expected	Lambda 4.5
Actual	Lambda 4.6087
Comments	Number of threads is fixed at 1 so number of processes = number of threads. Also long thread lives ensures none finish before the end of the simulation

Test ID	18
Requirement	The program will allow the user to specify the probability distribution of the number of processes created at runtime
Preconditions	Same data as test 15 EXCEPT: change noOfProcessesDistribution to: Constant 5.8
Procedure	1. Run tpsim 2. Analyse CoreLoads.txt to verify that the number of processes (i.e. threads) increases by either 0 or 6 at each step
Expected	Number of threads increases by either 0 or 6 at each time step
Actual	As expected
Comments	

Test ID	19
Requirement	The program will allow the user to specify the probability distribution used to decide whether to create a new process at runtime

Preconditions	Set config to the following: MemModel MemModelDemo 21 80 TPStrategy TPStrategyDemo migOff threadDurationDistribution Constant 5000.876 noOfThreadsDistribution Constant 1 noOfProcessesDistribution Constant 1 newProcessCreationDistribution Bernoulli 0.25 `3390426004` outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 200 maxRuntime 1000 stepSize 1.0
Procedure	1. Run tpsim 2. Divide the number of processes (200) by the number of the timeste p when the last process was created.
Expected	p 0.25
Actual	P 0.2528
Comments	Inspection shows the last process is created on step 791.
Test ID	20
Requirement	The program will allow the user to specify the probability distribution used to decide whether to create a new process at runtime
Preconditions	Same data as test 19 EXCEPT: change noOfProcessesDistribution to: `Normal 0.5 0.25 3816716409` maxProcesses to: `300`
Procedure	1. Run tpsim 2. Divide the number of processes (300) by the numbe r of the timestep when the last process was created.
Expected	p 0.5
Actual	p 0.5282
Comments	Inspection shows the last process is created on step 568. The probability is biased high as the program rejects negative values from the left tail.

Test ID	21
Requirement	Threads will be allocated to cores in accordance with the placement strategy chosen at runtime
Preconditions	Set config to the following: MemModel MemModelDemo 5 7 TPStrategy TPStrategyDemo migOff threadDurationDistribution Constant 100 noOfThreadsDistribution Constant 17 noOfProcessesDistribution Constant 1 newProcessCreationDistribution Constant 1 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 1 maxRuntime 2 stepSize 1.0
Procedure	1. Run tpsim 2. Check the thread allocation of the first timestep in CoreLoads.txt

Expected	1	4	3	0	0	0	0
	3	3	0	0	0	0	0
	3	0	0	0	0	0	0
	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
Actual	As expected						
Comments	High level repeat of a unit test						

Test ID	22						
Requirement	Threads will be allocated to cores in accordance with the placement strategy chosen at runtime						
Preconditions	Set config to the following: MemModel MemModelDemo 5 7 TPStrategy TPStrategyDemo migOff threadDurationDistribution Constant 100 noOfThreadsDistribution Constant 16 noOfProcessesDistribution Constant 2 newProcessCreationDistribution Constant 1 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 100 maxRuntime 2 stepSize 1.0						
Procedure	1. Run tpsim 2. Check the thread allocation of the first time step in CoreLoads.txt						
Expected	1	3	3	3	1	2	2
	3	3	0	2	2	2	0
	3	0	0	0	2	0	0
	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
Actual	As expected						
Comments	Cannot replicate thread distribution of unit test						

Test ID	23
Requirement	Threads will be allocated to cores in accordance with the placement strategy chosen at runtime
Preconditions	Set config to the following: MemModel MemModelDemo 5 7 TPStrategy TPStrategyDemo migOff threadDurationDistribution Constant 100 noOfThreadsDistribution Constant 16 noOfProcessesDistribution Constant 3 newProcessCreationDistribution Constant 1 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 100 maxRuntime 2 stepSize 1.0
Procedure	1. Run tpsim

	2. Check the thread allocation of the first timestep in CoreLoads.txt
Expected	

1	3	3	3	1	2	2
3	3	3	2	2	2	0
3	2	1	2	2	0	0
0	2	2	2	0	0	0
0	0	2	0	0	0	0

Actual	As expected
Comments	Colours denote different processes

Test ID	24
Requirement	Threads will be allocated to cores in accordance with the placement strategy chosen at runtime
Preconditions	Set config to the following: MemModel MemModelDemo 5 7 TPStrategy TPStrategyDemo migOff threadDurationDistribution Constant 100 noOfThreadsDistribution Normal 25 8 1967522489 noOfProcessesDistribution Constant 1 newProcessCreationDistribution Constant 1 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 100 maxRuntime 3 stepSize 1.0
Procedure	1. Run tpsim 2. Check the threads added for the first 3 time steps in CoreLoads.txt corresponds with the number of threads in the new process in ProcessInfo.txt (1 process is added each time step).

t = 1 (18 threads added)

1	4	3	0	0	0	0
4	3	0	0	0	0	0
3	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

t=2 (37 threads added)

1	4	5	5	1	5	5
4	3	0	5	5	5	0
3	0	0	0	4	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

t = 3 (27 threads added)

1	4	5	5	1	5	5
4	4	4	5	5	5	0
3	4	1	4	4	0	0
0	3	4	3	0	0	0
0	0	3	0	0	0	0

Actual	As expected
Comments	Red boxes mark cores with threads from 2 different processes

Test ID	25
Requirement	Threads will be allocated to cores in accordance with the placement strategy chosen at runtime
Preconditions	Set config to the following: MemModel MemModelDemo 5 7 TPStrategy TPStrategyDemo migOff threadDurationDistribution Constant 100 noOfThreadsDistribution Constant 5 noOfProcessesDistribution Constant 1 newProcessCreationDistribution Constant 1 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 100 maxRuntime 8 stepSize 1.0
Procedure	1. Run tpsim 2. Check the threads added in time step 7 in CoreLoads.txt are allocated to the bottom right corner as expected.

Expected	`t 6`
	`1 1 1 1 1 1 1`
	`1 1 1 0 1 1 0`
	`1 1 1 1 1 1 1`
	`1 1 1 1 1 1 0`
	`1 1 1 1 1 0 0`
	`t 7`
	`1 1 1 1 1 1 1`
	`1 1 1 0 1 1 0`
	`1 1 1 1 1 1 1`
	`1 1 1 1 1 1 2`
	`1 1 1 1 1 2 1`

Actual	As expected
Comments	

Test ID	26
Requirement	Threads will be allocated to cores in accordance with the placement strategy chosen at runtime
Preconditions	Set config to the following: MemModel MemModelDemo 5 7 TPStrategy TPStrategyDemo migOff threadDurationDistribution Constant 100 noOfThreadsDistribution Constant 1 noOfProcessesDistribution Constant 1 newProcessCreationDistribution Constant 1 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 100 maxRuntime 40 stepSize 1.0
Procedure	1. Run tpsim 2. Check the threads added in time steps 36, 37, 38 and 39 in CoreLoads.txt are allocated as expected.
Expected	`t 35`

	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 t 36 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 t 37 2 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 t 38 2 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 t 39 2 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 2
Actual	As expected
Comments	When the chip has even but non-zero load the placement strategy targets the corners because the 'surrounding load' is lowest (core positions off the edge contribute no load).

Test ID	27
Requirement	Threads will be allocated to cores in accordance with the placement strategy chosen at runtime
Preconditions	Set config to the following: MemModel MemModelDemo 5 15 TPStrategy TPStrategyDemo migOn threadDurationDistribution Constant 100 noOfThreadsDistribution Constant 15 noOfProcessesDistribution Constant 1 newProcessCreationDistribution Constant 1 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 5 maxRuntime 10 stepSize 1.0
Procedure	1. Run tpsim 2. Check the master thread is migrated to the bottom right the step after it is created. 3. Check that the migrated threads do not move again
Expected	t 1 1 3 3 0 0 0 0 0 0 0 0 0 0 0 0

	3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 t 2 0 3 3 2 1 2 2 0 0 0 0 0 0 0 0 3 3 0 2 2 2 0 0 0 0 0 0 0 0 0 2 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 t 3 0 3 3 2 0 2 2 2 1 2 2 0 0 0 0 3 3 0 2 2 2 0 2 2 2 0 0 0 0 0 2 0 0 0 2 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 t 4 0 3 3 2 0 2 2 2 0 2 2 2 1 2 2 3 3 0 2 2 2 0 2 2 2 0 2 2 2 0 2 0 0 0 2 0 0 0 2 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 t 5 0 3 3 2 0 2 2 2 0 2 2 2 0 2 2 3 3 2 2 2 2 0 2 2 2 0 2 2 2 0 2 2 1 2 2 0 0 0 2 0 0 0 2 0 0 0 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 1 1 1 1 t 6 0 3 3 2 0 2 2 2 0 2 2 2 0 2 2 3 3 2 2 2 2 0 2 2 2 0 2 2 2 0 2 2 0 2 2 0 0 0 2 0 0 0 2 0 0 0 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 1 1 1 1 1
Actual	As expected
Comments	

Test ID	28
Requirement	
Preconditions	Set config to the following: MemModel MemModelDemo 11 31 TPStrategy TPStrategyDemo migOff threadDurationDistribution Constant 10 noOfThreadsDistribution Constant 9 noOfProcessesDistribution Constant 1 newProcessCreationDistribution Constant 1 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 2 maxRuntime 4000 stepSize 1.0
Procedure	1. Run tpsim 2. Check the thread runtimes are consistent with the core loading and memory factor
Expected	t 2 1 2 2 2 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

	2 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... Process 10.000000 10.000000 = 10*1*1 10.000000 26.333333 = floor(10*4/3)*2 + [10*4/3-floor(10*4/3)] 10.000000 26.333333 = floor(10*4/3)*2 + [10*4/3-floor(10*4/3)] 10.000000 30.000000 = 10*2*(6/4) 10.000000 15.000000 = 10*1*(6/4) 10.000000 15.000000 = 10*1*(6/4) 10.000000 26.333333 = floor(10*4/3)*2 + [10*4/3-floor(10*4/3)] 10.000000 26.333333 = floor(10*4/3)*2 + [10*4/3-floor(10*4/3)] 10.000000 30.000000 = 10*2*(6/4) Process 10.000000 10.000000 = 10*1*1 10.000000 26.333333 = floor(10*4/3)*2 + [10*4/3-floor(10*4/3)] 10.000000 13.333333 = 10*1*(4/3) 10.000000 13.333333 = 10*1*(4/3) 10.000000 15.000000 = 10*1*(6/4) 10.000000 15.000000 = 10*1*(6/4) 10.000000 15.000000 = 10*1*(6/4) 10.000000 15.000000 = 10*1*(6/4) 10.000000 26.333333 = floor(10*4/3)*2 + [10*4/3-floor(10*4/3)]
Actual	As expected
Comments	Memory factor is 1, (1/2 + ¼) or (1/2 + 1/6)

Test ID	29
Requirement	
Preconditions	Set config to the following: MemModel MemModelDemo 5 7 TPStrategy TPStrategyDemo migOff threadDurationDistribution Constant 5 noOfThreadsDistribution Normal 25 8 1967522489 noOfProcessesDistribution Constant 1 newProcessCreationDistribution Constant 1 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 3 maxRuntime 100 stepSize 1.0
Procedure	1. Run tpsim 2. Check the runtimes where the core is shared by threads from different processes (orange and purple blocks below).
Expected	1 4 5 5 1 5 5 4 4 4 5 5 5 0 3 4 1 4 4 0 0 0 3 4 3 0 0 0 0 0 3 0 0 0 0 Process 5.000000 5.000000 5.000000 26.166667 5.000000 26.166667 5.000000 36.166667

```
5.000000 29.083333
5.000000 22.166667
5.000000 26.166667
5.000000 26.166667
5.000000 36.166667
5.000000 29.083333
5.000000 22.166667
5.000000 26.166667
5.000000 26.166667
5.000000 36.166667
5.000000 29.083333
5.000000 22.166667
5.000000 26.166667
5.000000 26.166667

Process
5.000000 5.000000
5.000000 33.066667
5.000000 33.066667
5.000000 33.066667
5.000000 37.100000
5.000000 37.100000
5.000000 37.100000
5.000000 30.000000
5.000000 33.066667
5.000000 33.066667
5.000000 33.066667
5.000000 37.100000
5.000000 37.100000
5.000000 37.100000
5.000000 30.000000
5.000000 33.066667
5.000000 33.066667
5.000000 33.066667
5.000000 37.100000
5.000000 37.100000
5.000000 37.100000
5.000000 30.000000
5.000000 33.066667
5.000000 33.066667
5.000000 33.066667
5.000000 36.250000
5.000000 37.100000
5.000000 37.100000
5.000000 37.100000
5.000000 30.000000
5.000000 33.066667
5.000000 33.066667
5.000000 33.066667
5.000000 36.250000
5.000000 37.100000
5.000000 37.100000
5.000000 37.100000

Process
5.000000 5.000000
5.000000 26.166667
5.000000 26.166667
5.000000 26.166667
5.000000 26.166667
5.000000 22.166667
```

	5.000000 22.166667
	5.000000 22.166667
	5.000000 26.166667
	5.000000 26.166667
	5.000000 26.166667
	5.000000 26.166667
	5.000000 22.166667
	5.000000 22.166667
	5.000000 22.166667
	5.000000 26.166667
	5.000000 26.166667
	5.000000 26.166667
	5.000000 26.166667
	5.000000 22.166667
	5.000000 22.166667
	5.000000 22.166667
	5.000000 26.166667
	5.000000 26.166667
	5.000000 26.166667
	5.000000 26.166667
	5.000000 28.000000

Actual	As expected
Comments	Detailed calculations shown in Appendix A

Test ID	30
Requirement	
Preconditions	Set config to the following: MemModel MemModelDemo 8 8 TPStrategy TPStrategyDemo migOn threadDurationDistribution Constant 7 noOfThreadsDistribution Constant 9 noOfProcessesDistribution Constant 1 newProcessCreationDistribution Constant 1 outputCoreLoad on CoreLoads.txt 1 outputRunTimes on ProcessInfo.txt maxProcesses 2 maxRuntime 4000 stepSize 1.0
Procedure	1. Run tpsim 2. Check the thread runtimes are consistent with the core loading and memory factor after the thread migration.
Expected	0 2 2 2 0 1 1 0 2 1 0 1 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 Process 7.000000 7.000000 7.000000 25.366667 7.000000 25.366667 7.000000 25.380952 7.000000 12.761905

	`7.000000 12.761905`
	`7.000000 25.366667`
	`7.000000 25.366667`
	`7.000000 25.380952`
	
	`Process`
	`7.000000 7.000000`
	`7.000000 25.145833`
	`7.000000 12.250000`
	`7.000000 12.250000`
	`7.000000 12.259259`
	`7.000000 12.515152`
	`7.000000 12.259259`
	`7.000000 12.259259`
	`7.000000 25.145833`

Actual	As expected
Comments	Detailed calculations shown in Appendix B

6.3 Application Demonstration

This section will show how the simulator could be used to evaluate a thread placement strategy. TPStrategyDemo, created for the purpose of demonstrating and testing the simulator, will be used as the test case.

Thread placement strategies are commonly evaluated by comparison with alternatives. TPStrategyDemo can be used in two modes:

- Placement of new threads only (migOff)
- Placement of new threads and migration of existing threads (migOn).

The simulator will be run for each option using an identical thread profile (seed values replicated) and the results contrasted.

All post-processing of output data is performed using the commercial analysis and graphics package Matlab.

6.3.1 Simulation Set-up

```
MemModel           MemModelDemo        20 50
TPStrategy         TPStrategyDemo      migOff       // migOn

threadDurationDistribution       Normal    40 15   3460280934
noOfThreadsDistribution          Poisson   10      341861941
noOfProcessesDistribution        Poisson    1      1336071836
newProcessCreationDistribution   Bernoulli 0.8     4293383796

outputCoreLoad      on       CoreLoads.txt      10
outputRunTimes      on       ProcessInfo.txt

maxProcesses   5000
maxRuntime    10000000
stepSize        1.0
```

The memory model specifies a processor that is non-square and has 1000 cores.

The simulator is set to generate 5000 processes with (nominally) 10 threads each.

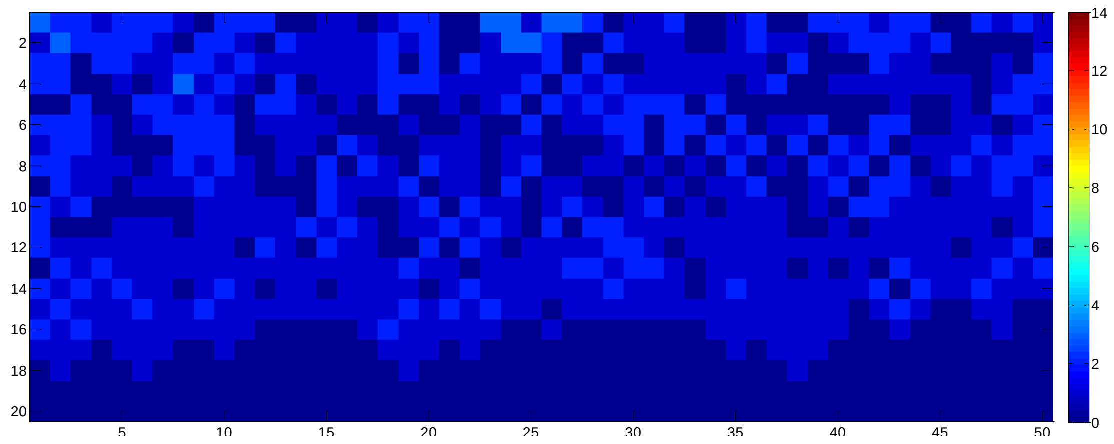
For the ideal scenario where all threads were independent and incurred no memory access penalty the objective would be to place them so that each core held one thread. If processes with 10 threads are generated at a rate of approximately 1 per time step then the processor would be fully occupied after 100 time steps. Hence if the processes have a nominal runtime of 100 time steps the first should be finishing as the 101st is starting, maintaining a core utilization rate of close to 100%. Since both the initial placement and subsequent migration strategies are sub-optimal and many of the threads incur memory access penalties, cores become loaded with multiple threads and the actual process runtimes will be extended. Experimentation shows that when migration is activated a nominal thread duration of 100 time steps results in loadings of 50-60 threads on some cores. This is not a problem for the simulator, but maintaining consistent heat map scaling for the purposes of report presentation would result in a loss of detail when comparing the effects of enabling/disabling the migration strategy. So for primarily aesthetic reasons the process generation profile will be relaxed from the nominal ideal.

The process generation profile for the specified configuration has the following expected values:

- 1 new process is created 4 out of every 5 time steps,
- a new process contains 10 threads,
- a new thread has an ideal runtime of 40 steps.

Note that the number of new processes created will be biased upward as the Poisson distribution will be retried if it returns 0 (at least 1 process will be created once a positive decision has been made). Similarly the simulator will not generate negative thread runtimes from the left tail of the normal distribution.

6.3.2 Comparison of core loading



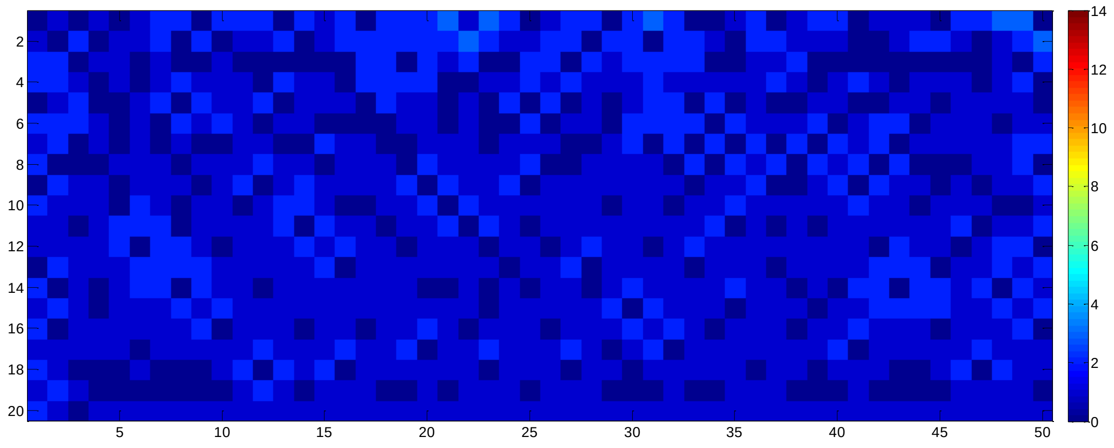
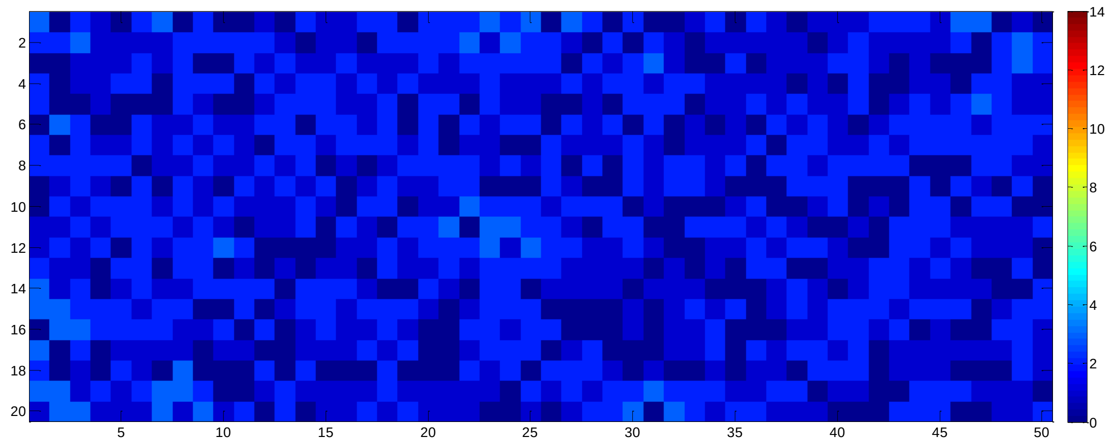


Figure 5.1: Step 100. Comparison of number of threads per core with migration strategy (i) enabled (ii) disabled.

After 100 steps there are still some visible similarities in the distributions of threads across the cores. The effects of the migration can be seen as the solid line of threads along the bottom row in the lower figure.

In both cases the level of core loading is reasonable, with the busiest cores supporting four or five threads.



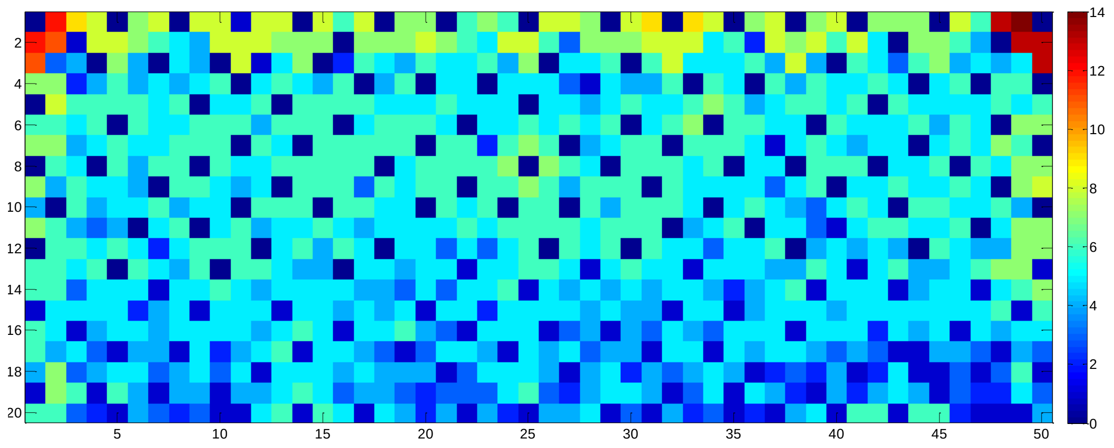


Figure 5.2: Step 2700. Comparison of number of threads per core with migration strategy (i) disabled (ii) enabled.

By step 2700 there is a clear difference in the core loading, with the migration strategy resulting in the chip having to support more threads. Since the thread generation profile is identical for the two cases this indicates threads are running longer.

This step shows the highest loading of a single core when using migration (14). The highest observed loading when migration is not used is 5.

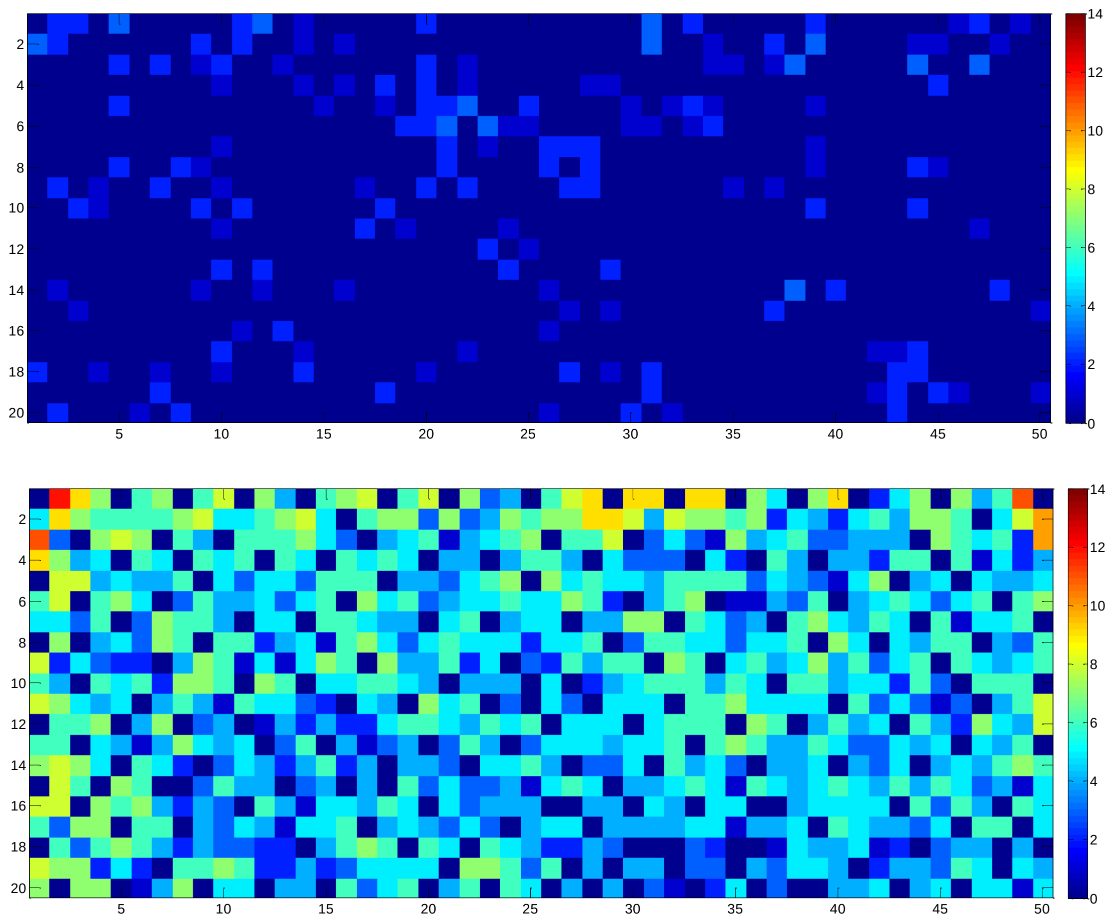
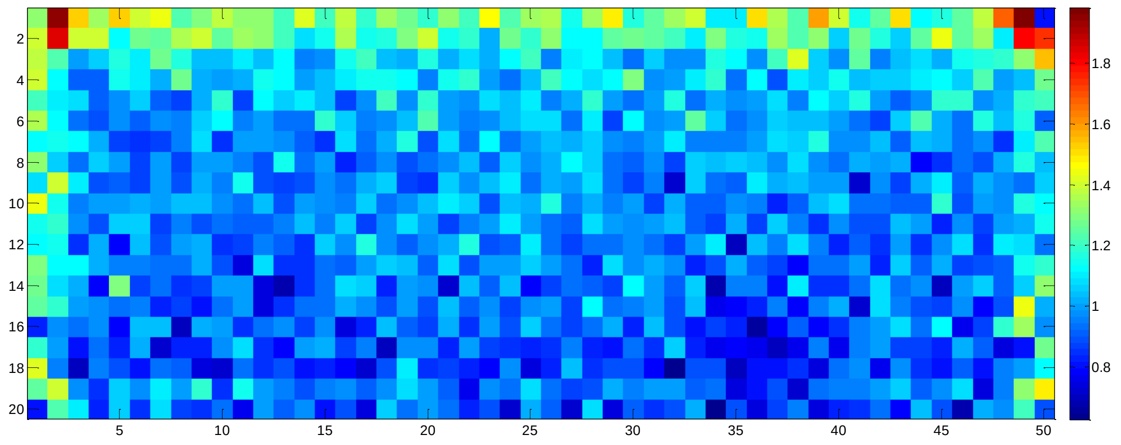


Figure 5.3: Step 4000. Comparison of number of threads per core with migration strategy (i) disabled (ii) enabled.

By step 4000 the simulation with migration disabled is approaching the end of its run, which occurs between steps 4120 and 4130. With migration enabled the chip is still working hard with numerous heavily loaded cores; this strategy extends the runtime of the simulation by a little over 400 steps.

Both simulations finish well before the expected termination time (the last process was expected to be created around step 6250). This is attributable to the upward bias on the Poisson distribution generating more processes per step than was expected.



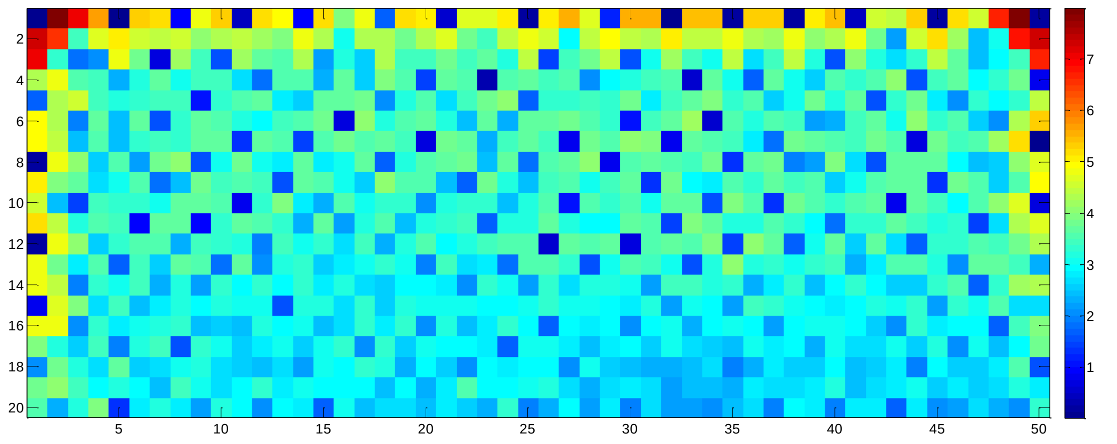


Figure 5.4: Comparison of number of threads per core averaged over the run with migration strategy (i) disabled (ii) enabled. Note the different heat map scaling.

Comparison of the average core loads over the duration of the simulation shows both similarities and differences.

In both cases the core loading tends to decrease as the row number (shown on the left) increases. This is a consequence of the initial placement strategy for the master thread. The search moves from top left to bottom right, and in cases where the loading is equal it is the one that is found first that is selected. Distribution is more even when migration is enabled as the core preference for the single migrated thread is reversed, moving from bottom right to top left.

Core loading is heaviest along the edges. This occurs because all slave threads are placed within two core distances of the master thread, and cores on the edges of the chip have fewer surrounding cores over which to distribute threads. This is exacerbated in the corner positions, where the highest loading is observed to occur. The higher edge loading highlights the regular pattern of near-empty cores created by migration of single master threads (lower figure).

The most striking difference between the two heat maps is the scaling. With migration disabled even the busiest cores have an average loading of two threads. When threads are migrated this increases to eight; the four factor increase is also representative of the lower loaded cores.

6.3.3 Thread and Process Runtimes

Heat maps are useful for analyzing the efficiency of core usage, but the metrics of primary interest are the running times of the threads and processes.

The thread and process data is summarized below. The duration of a process is deemed to be the runtime of the longest lived thread.

Total number of threads = 50162;

Average ideal thread duration = 40.1953;

Summary of thread data		
	MigOff	MigOn
Average duration (s)	82.4910	284.8295
Average overrun (s)	42.2957	244.6342
Average overrun(% of ideal)	107.9819	613.5621
Maximum overrun (s)	256.1877	1521.8
Maximum overrun (% of ideal)	569.1749	2402.1
Threads within 10% of ideal (%)	9.7464	8.1117
Threads within 50% of ideal (%)	31.5857	8.5623
Threads within 100% of ideal (%)	47.0316	10.3764

Table 5.2 Comparison of thread metrics with migration enabled and disabled (thread generation profiles identical).

Total number of processes = 5000;

Average ideal process duration (based on length of longest thread) = 62.6417;

Summary of process data (based on lengths of longest thread)		
	MigOff	MigOn
Average duration (s)	131.3461	480.3761
Average overrun (s)	68.7044	417.7343
Average overrun (% of ideal)	109.6392	668.7999
Maximum overrun (s)	256.1877	1521.8
Maximum overrun(% of ideal)	423.3075	2201.2
Processes within 10% of ideal (%)	0.88	0.04
Processes within 50% of ideal (%)	16.86	0.08
Processes within 100% of ideal (%)	41.1	0.5

Table 5.3 Comparison of process metrics with migration enabled and disabled (thread generation profiles identical).

Summary analysis of the thread runtime data supports what could be inferred from inspection of the heat map data: the TPStrategyDemo strategy works better with the migration disabled. migOff can be expected to outperform migOn on basic metrics by a factor of between 3 and 6.

After inspection of the migration strategy this conclusion is unsurprising. Moving a master thread from an adjacent to a distant core reduces the thread runtime multiplier from 0.75 to around 0.5, a factor of 1/3.

In this implementation of the framework the processes are output to file in the order that they are created. Since the rate of process creation is expected to be reasonably even his means the process runtimes can be plotted as a (loose) time series to see if there are any trends.

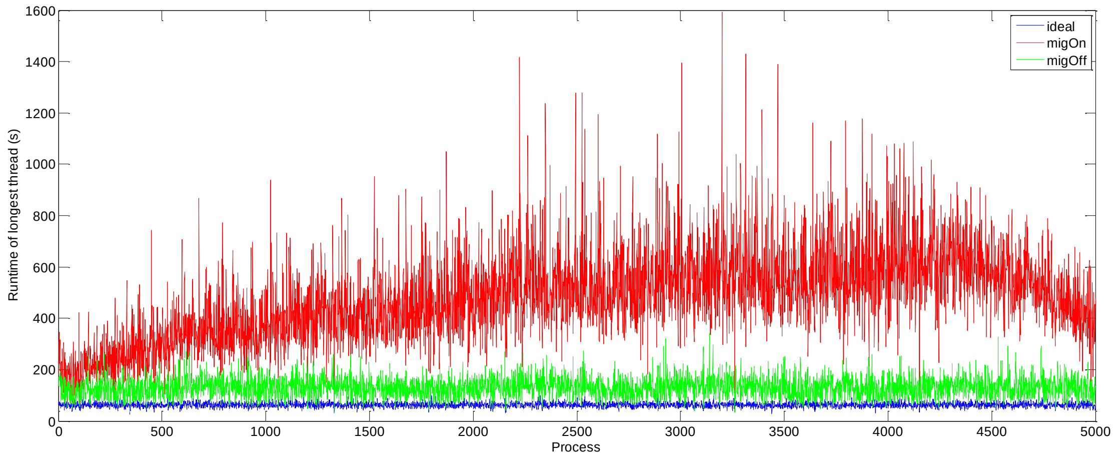


Figure 5.5 Process runtimes plotted in order of creation.

The runtimes of processes from the migOn simulation show a clear upward trend until around 4500 have been created, after which there is a downward trend. The runtimes of migOff show no obvious relation to the point at which they were created.

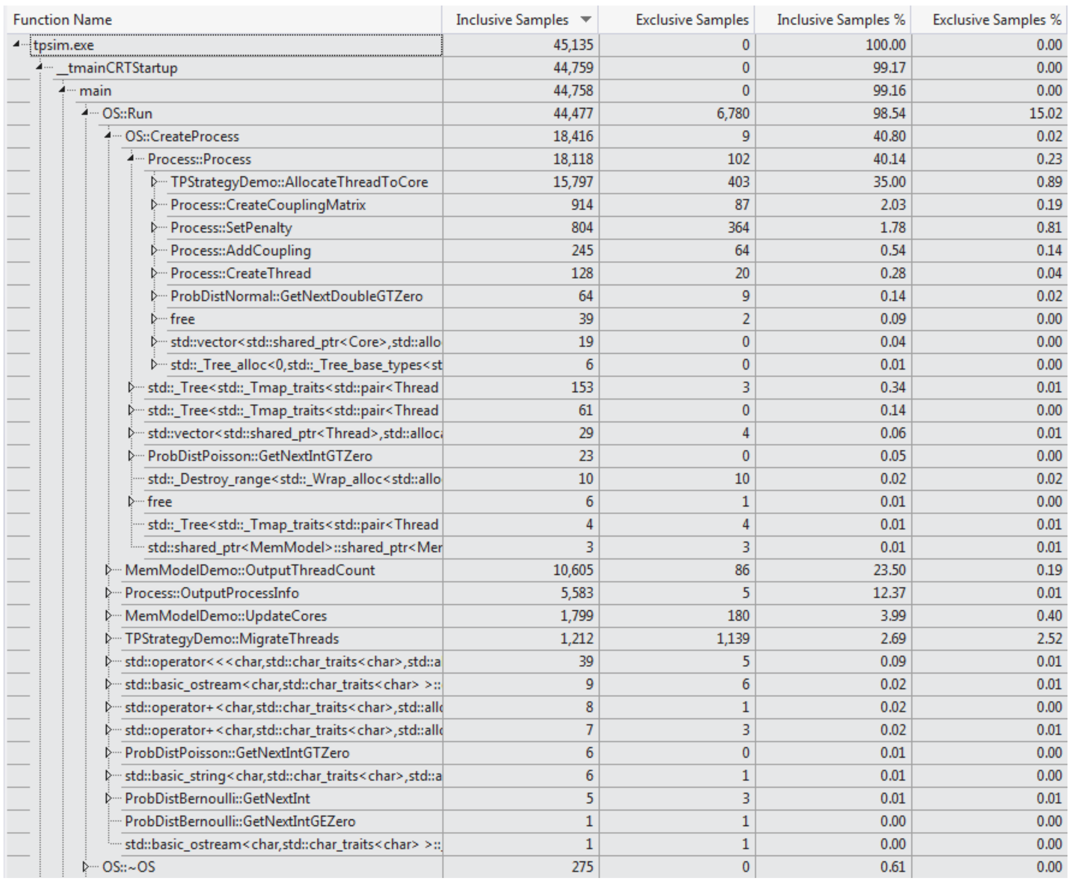
Using the core loading information in conjunction with the thread runtime data it can be concluded that:

1. Moving the master thread a long way from its slaves slows running significantly.
2. This creates a knock-on effect. Longer living threads increase the loading on the cores which slows runtime even further.

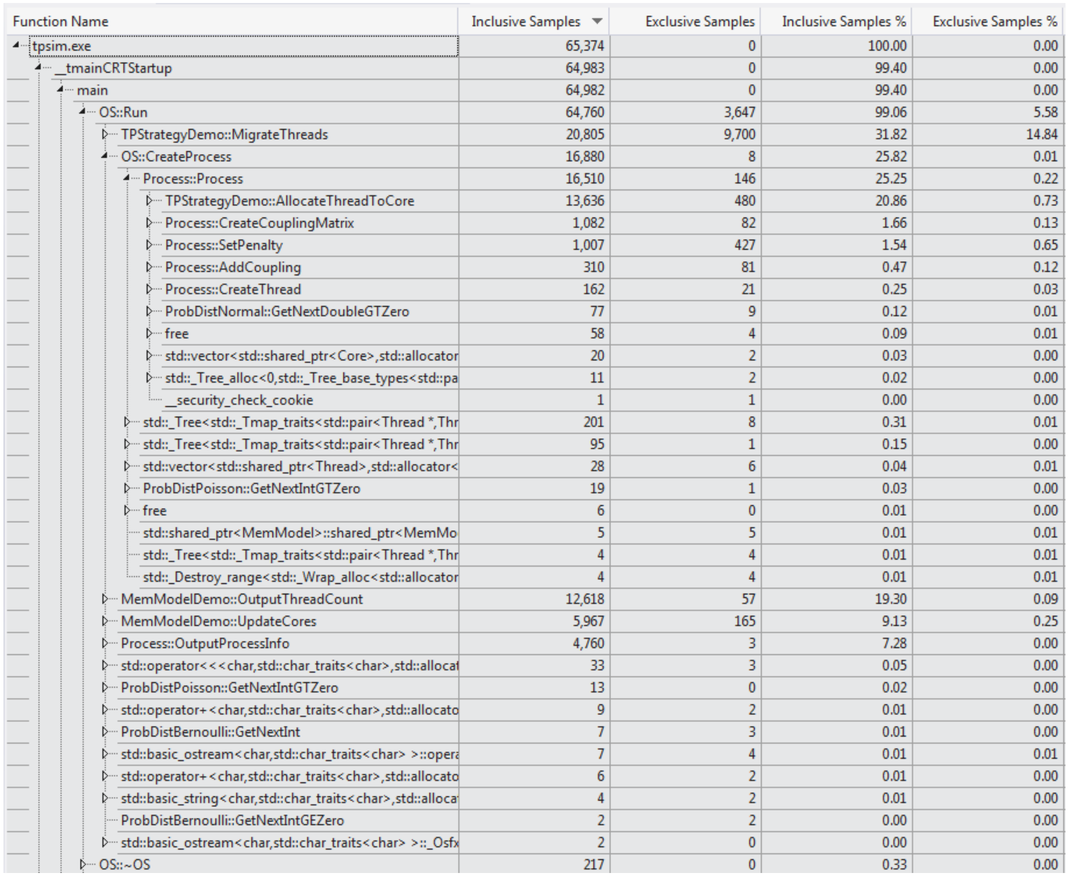
6.4 Code Profiling

The runtimes for both simulations described in Section 6.3 were trivial (approximately 1 and 2 seconds for migration off and on respectively: using an Intel i5 running Windows 7).

Microsoft Visual Studio 2012 offers code profiling tools to investigate the bottlenecks and key functions. The following figures show the results (sampling rate: every 50000 clock cycles).



Function Name	Inclusive Samples	Exclusive Samples	Inclusive Samples %	Exclusive Samples %
tpsim.exe	45,135	0	100.00	0.00
__tmainCRTStartup	44,759	0	99.17	0.00
main	44,758	0	99.16	0.00
OS::Run	44,477	6,780	98.54	15.02
OS::CreateProcess	18,416	9	40.80	0.02
Process::Process	18,118	102	40.14	0.23
TPStrategyDemo::AllocateThreadToCore	15,797	403	35.00	0.89
Process::CreateCouplingMatrix	914	87	2.03	0.19
Process::SetPenalty	804	364	1.78	0.81
Process::AddCoupling	245	64	0.54	0.14
Process::CreateThread	128	20	0.28	0.04
ProbDistNormal::GetNextDoubleGTZero	64	9	0.14	0.02
free	39	2	0.09	0.00
std::vector<std::shared_ptr<Core>,std::allo	19	0	0.04	0.00
std::_Tree_alloc<0,std::_Tree_base_types<st	6	0	0.01	0.00
std::_Tree<std::_Tmap_traits<std::pair<Thread	153	3	0.34	0.01
std::_Tree<std::_Tmap_traits<std::pair<Thread	61	0	0.14	0.00
std::vector<std::shared_ptr<Thread>,std::alloc	29	4	0.06	0.01
ProbDistPoisson::GetNextIntGTZero	23	0	0.05	0.00
std::_Destroy_range<std::_Wrap_alloc<std::allo	10	10	0.02	0.02
free	6	1	0.01	0.00
std::_Tree<std::_Tmap_traits<std::pair<Thread	4	4	0.01	0.01
std::shared_ptr<MemModel>::shared_ptr<Mer	3	3	0.01	0.01
MemModelDemo::OutputThreadCount	10,605	86	23.50	0.19
Process::OutputProcessInfo	5,583	5	12.37	0.01
MemModelDemo::UpdateCores	1,799	180	3.99	0.40
TPStrategyDemo::MigrateThreads	1,212	1,139	2.69	2.52
std::operator<<<char,std::char_traits<char>,std::a	39	5	0.09	0.01
std::basic_ostream<char,std::char_traits<char> >::	9	6	0.02	0.01
std::operator+<char,std::char_traits<char>,std::all	8	1	0.02	0.00
std::operator+<char,std::char_traits<char>,std::all	7	3	0.02	0.01
ProbDistPoisson::GetNextIntGTZero	6	0	0.01	0.00
std::basic_string<char,std::char_traits<char>,std::a	6	1	0.01	0.00
ProbDistBernoulli::GetNextInt	5	3	0.01	0.01
ProbDistBernoulli::GetNextIntGEZero	1	1	0.00	0.00
std::basic_ostream<char,std::char_traits<char> >::	1	1	0.00	0.00
OS::~OS	275	0	0.61	0.00



Function Name	Inclusive Samples	Exclusive Samples	Inclusive Samples %	Exclusive Samples %
tpsim.exe	65,374	0	100.00	0.00
__tmainCRTStartup	64,983	0	99.40	0.00
main	64,982	0	99.40	0.00
OS::Run	64,760	3,647	99.06	5.58
TPStrategyDemo::MigrateThreads	20,805	9,700	31.82	14.84
OS::CreateProcess	16,880	8	25.82	0.01
Process::Process	16,510	146	25.25	0.22
TPStrategyDemo::AllocateThreadToCore	13,636	480	20.86	0.73
Process::CreateCouplingMatrix	1,082	82	1.66	0.13
Process::SetPenalty	1,007	427	1.54	0.65
Process::AddCoupling	310	81	0.47	0.12
Process::CreateThread	162	21	0.25	0.03
ProbDistNormal::GetNextDoubleGTZero	77	9	0.12	0.01
free	58	4	0.09	0.01
std::vector<std::shared_ptr<Core>,std::allocator	20	2	0.03	0.00
std::_Tree_alloc<0,std::_Tree_base_types<std::pa	11	2	0.02	0.00
__security_check_cookie	1	1	0.00	0.00
std::_Tree<std::_Tmap_traits<std::pair<Thread *,Thr	201	8	0.31	0.01
std::_Tree<std::_Tmap_traits<std::pair<Thread *,Thr	95	1	0.15	0.00
std::vector<std::shared_ptr<Thread>,std::allocator<	28	6	0.04	0.01
ProbDistPoisson::GetNextIntGTZero	19	1	0.03	0.00
free	6	0	0.01	0.00
std::shared_ptr<MemModel>::shared_ptr<MemMo	5	5	0.01	0.01
std::_Tree<std::_Tmap_traits<std::pair<Thread *,Thr	4	4	0.01	0.01
std::_Destroy_range<std::_Wrap_alloc<std::allocator	4	4	0.01	0.01
MemModelDemo::OutputThreadCount	12,618	57	19.30	0.09
MemModelDemo::UpdateCores	5,967	165	9.13	0.25
Process::OutputProcessInfo	4,760	3	7.28	0.00
std::operator<<<char,std::char_traits<char>,std::allocat	33	3	0.05	0.00
ProbDistPoisson::GetNextIntGTZero	13	0	0.02	0.00
std::operator+<char,std::char_traits<char>,std::allocato	9	2	0.01	0.00
ProbDistBernoulli::GetNextInt	7	3	0.01	0.00
std::basic_ostream<char,std::char_traits<char> >::opera	7	4	0.01	0.01
std::operator+<char,std::char_traits<char>,std::allocato	6	2	0.01	0.00
std::basic_string<char,std::char_traits<char>,std::alloca	4	2	0.01	0.00
ProbDistBernoulli::GetNextIntGEZero	2	2	0.00	0.00
std::basic_ostream<char,std::char_traits<char> >::_Osfx	2	0	0.00	0.00
OS::~OS	217	0	0.33	0.00

Figure 5.6 Comparison of the busiest functions when migration is disabled and enabled.

When migration is disabled the run is dominated by the Process::CreateProcess() call. Drilling down further, 35% of the runtime is due to TPStrategyDemo::AllocateThreadToCore(). As the processor fills with threads this function call involves a visit to each core. Recall that TPStrategyDemo is not a real placement strategy, only a demonstrator. However this does indicate a potential weakness in the MemModel interface, which does not provide an efficient means to access cores based on loading.

Of further concern is the fact that the next two time-consuming activities both involve output to file (23.5% for core loading and 12% for thread runtime data). Output of thread runtime data involves printing two double precision values for each thread created; if detailed analysis of runtimes is required there is little scope for improvement. Core loading data, on the other hand, could certainly be released more efficiently. It could be reduced by a factor of 5 simply by changing the output interval from 10 to 50.

The fourth busiest function is MemModel::UpdateCores(), which is the heart of the simulator engine. This only accounts for 4% of the runtime.

When migration is enabled the most time consuming function is TPStrategyDemo::MigrateThreads() at 32%, which determinedly examines every core. This is an indicator that the simulator runtime will inevitably be influenced by the thoroughness of the placement strategy.

The ordering of the subsequent active functions is similar to that when migration is inactive. An interesting point is that the activity level of MemModel::UpdateCores() more than doubles to 9% as the runtimes of the threads increases. This indicates the sensitivity of the simulation runtime to the thread generation profile: if a comparatively small number of threads with very long durations were created then updating thread times would be the busiest function.

In order to provide a sense of scale, the time spent in MemModel::UpdateCores() is dominated by its calls to Thread::UpdateThread(). The most common route through this function comprises

double * double

double + double

double – double

i.e. three FLOPS.

7 Conclusion

7.1 Status of the Software

7.1.1 Functional Requirements

The software meets all of its functional requirements. It has been thoroughly tested and there are no known bugs.

The software provides a framework for assessing thread placement strategies on different processor memory models using a runtime-specified thread generation profile. The thread runtime is calculated as a function of both core loading and cache distance. The thread placement strategy may be used to place newly created threads and migrate existing threads. The usefulness of the simulation output in assessing strategies has been demonstrated.

Examples of a processor memory model and a thread placement strategy are implemented. These do not represent real world examples but act as demonstrators and facilitate testing.

The abstract interfaces for the memory model and thread placement strategy are sufficient to provide the functionality required, but a means of accessing cores based on load would be beneficial.

7.1.2 Non-Functional Requirements

The objects modelling the cores and threads require only a small amount of memory so the simulator is scalable to many thousands of cores. However, the current implementation holds all created thread objects in memory until the end of the simulation. This may become a burden if the number of threads created over the simulation run becomes very large.

The simulator was built and tested using Microsoft Visual Studio 2012 on a machine running Windows 7. The source code has been shown to build (with some minor modifications) with GCC on a Linux OS, but has not been tested.

Subjectively, the style of the source code (readable and appropriately commented) has been adjudged to be of a suitable standard for future modification and development.

The central algorithm to the simulator engine, which updates the status of the threads, will for most calls take 3 FLOPS. The algorithm also contains less visited branches that increase accuracy at the cost of speed; these were implemented primarily to increase confidence in testing and a stripped down variant could be used for a small performance gain.

7.2 Further Work

7.2.1 Implementation issues

The Thread::UpdateThread() and Core::UpdateCore() methods include additional processing to redistribute cpu time returned by completed threads. This was introduced primarily to make the arithmetic more challenging for the purpose of accuracy testing. However it will incur a small overhead. Since high speed is one of the primary requirements for the simulator it may be preferable to remove this additional processing (the thread runtime results would still be expected to have accuracy close to one time step).

At present all Threads are retained in memory until the end of the run, when the runtime data is printed out. Memory usage could be improved by printing out Thread data when the parent Process is completed, then deleting the Process. (Forcibly deleting Threads before the Process is complete is inadvisable as coupling information may still be required). However, this will mean that processes are not printed out in the order they were created; this proved a useful feature when analyzing data in Section 6.3.3.

The MemModel base class does not provide an obvious interface for finding the lowest (or indeed highest) loaded cores, and the TPStrategy is forced to iterate over the whole processor to determine the information itself. Including a facility to return cores based on loading information efficiently is likely to improve simulation runtime for a range of placement strategies. It could be implemented in a similar way to MemModel::FindNearestCores(), with the caller specifying a level. It does, however, present a difficulty in that in that the number of cores to return will often be very large.

Coupling of threads is achieved by a crude, inflexible and hard-coded implementation in the Process class. A simple improvement to this would be to make the couplings probabilistic by adding extra functionality to the SimProbabilities class. Other approaches could be to implement variants of the Process class with different coupling characteristics, or to choose the functionality at runtime.

The formatting of the data output to file is crude and will benefit from a protocol that eases post-processing. Further, code profiling shows that a significant proportion of runtime is spent outputting core activity data. An improvement may be to only output when a change is detected. Extending this further, each thread could track its core history and this could be output instead. This would also allow analysis of relative distance between coupled threads, which cannot be performed with the current output. This implementation would increase the memory footprint of the Thread object, but the history will be short provided the thread is not moved too often.

In the present architecture, identifying the parent Process of a Thread is possible but non-trivial (it would involve exhaustive search of all Processes via the OS). This is not an issue for the demonstration thread placement strategy, but may be desirable for a migration strategy that (for example) chose its threads via the Cores rather than the Processes. The Thread object could hold a pointer back to its Process, but these will be stack variables (created in the Run() method of OS). This is feasible provided OS::Run is only called once during a simulation (as in the current implementation), or the stack position does not shift between successive calls (i.e. no additional memory is allocated).

At present the Normal distribution will loop indefinitely if a positive value is requested from a negative based distribution; a loop limit followed by an error message is required.

Since the MemModel is created by a factory and is necessarily heap allocated the Core objects that it contains could be stack allocated regardless of whether theRun() function is exited or not. This would result in the Thread objects holding raw pointers to Core objects which may be regarded as undesirable.

7.2.2 Extending the Framework

Clearly the effectiveness of the simulator depends on the strength and efficiency of the processor models and thread placement strategies. Code profiling shows that a placement strategy that requires extensive searching can dominate the simulation runtime, and this will be compounded if searching the memory model becomes laborious.

Further, testing the simulator results against hardware will require the ability to describe and model meaningful process and thread probability distributions.

With regard to increasing the extending the capability of the components, and added complexity would be to make threads wait for each other as often occurs in multithreaded applications. This can be added simply, though the accuracy of each wait may be of the order of a time step. The Thread object would be required to hold a boolean that indicates its waiting status to the Core. This would be tracked and updated by the parent Process at the end of each time step.

8 References

1. Lankamp M, Poss R, Yu Q, Fu J, Uddin M. I. & Jesshope C. R. (2013): *MGSim - Simulation tools for multi-core processor architectures*; In: CoRR abs/1302.1390.

2. Poss R, Lankamp M, Yu Q, Fu, J, van Tol M W & Jesshope C. R. (2012): *Apple-CORE: Microgrids of SVP Cores - Flexible, General-Purpose, Fine-Grained Hardware Concurrency Management*; in 'DSD' , IEEE, , pp. 501-508.

3. Uddin I, Poss R, Jesshope C (2014): *Analytical-based high-level simulation of the microthreaded many-core architectures*; In: 22nd Euromicro International Conference on Parallel, Distributed and Network-based Processing.

4. Ubal R, Sahuquillo J, Petit S and Lopez P (2007): *Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors*; In: 19th International Symposium on Computer Architecture and High Performance Computing; p 62-68.

5. Zwick M, Durkovic M, Obermeier F, Bamberger W and Diepold K. (2009): *Mcccsim - a highly configurable multi core cache contention simulator*; Technical report, Technische Universitat Munchen.

6. http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

7. Jaleel A, Cohn R.S, Luk C-K, and Jacob B (2008): *CMP$im: A Pin-based on-the-fly multi-core cache simulator*; In: Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), p 28-36.

8. Ratanaworabhan P (2012): *Functional cache simulator for multicore*, In: 2012 9th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, ECTI -CON

9. Genbrugge D, Eyerman S and Eeckout L (2010): *Interval simulation: Raising the level of abstraction in architectural simulation*; In: International Symposium on High-Performance Computer Architecture; p 1-12.

10. Carlson TE, Heirman W and Eeckout L (2011): *Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation;* In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Article No 52.

11. Miller J.E., Kasture H, Kurian G, Gruenwald C, Beckmann N, Celio C, Eastep J and Agarwal A (2010): *Graphite: A distributed parallel simulator for multicores*; In: IEEE 16th International Symposium on High Performance Computer Architecture, p 1-12.

12. Zhuravlev, S, Blagodurov, S. and Fedorova A (2010): *AKULA: a toolset for experimenting and developing thread placement algorithms on multicore systems*; In: Proceedings of the 19th international conference on Parallel architectures and compilation techniques, p 249-260.

13. Tousimojarad A. and Vanderbauwhede W. (2014): *An Efficient Thread Mapping Strategy for Multiprogramming on Manycore Processors*; In: Advances in Parallel Computing, Vol. 25, p 63-71.

14. Diener M, Madruga FL, Rodrigues ER, Alves MAZ, Schneider J, Navaux POA and Heiss H-U (2010): *Evaluating Thread Placement Based on Memory Access Patterns for Multi-core Processor*; In: 12th IEEE International Conference on High Performance Computing and Communications, p 491-496.

15. Rajagopalan M, Lewis BT and Anderson TA (2007): *Thread Scheduling for Multicore Platforms*, In: Proc. 11th Workshop on Hot Topics in Operating Systems, USENIX, pp. 7–12

16. http://www.embedded.com/design/mcus-processors-and-socs/4007135/Multicore-microprocessors-and-embedded-multicore-SOCs-have-very-different-needs

17. http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

18. http://electronicdesign.com/embedded/64-core-chip-spins-smp-design-higher-performance-levels

Appendix A Calculations for Test 29

Process1 (orange)

NoOfThreads	**memFactor**	**cpuTime**	**Remainder (cpu)**	**Remainder (Real)**	**Thread time**	**Leftover *3 threads**
3	0.66666667	0.222222			1	
5	0.66666667	0.355556			2	
5	0.66666667	0.488889			3	
5	0.66666667	0.622222			4	
5	0.66666667	0.755556			5	
5	0.66666667	0.888889			6	
5	0.66666667	1.022222			7	
5	0.66666667	1.155556			8	
5	0.66666667	1.288889			9	
5	0.66666667	1.422222			10	
5	0.66666667	1.555556			11	
5	0.66666667	1.688889			12	
5	0.66666667	1.822222			13	
5	0.66666667	1.955556			14	
5	0.66666667	2.088889			15	
5	0.66666667	2.222222			16	
5	0.66666667	2.355556			17	
5	0.66666667	2.488889			18	
5	0.66666667	2.622222			19	
5	0.66666667	2.755556			20	
5	0.66666667	2.888889			21	
5	0.66666667	3.022222			22	
5	0.66666667	3.155556			23	
5	0.66666667	3.288889			24	
5	0.66666667	3.422222			25	
5	0.66666667	3.555556			26	
5	0.66666667	3.688889			27	
5	0.66666667	3.822222			28	
5	0.66666667	3.955556			29	
5	0.66666667	4.088889			30	
5	0.66666667	4.222222			31	
5	0.66666667	4.355556			32	
5	0.66666667	4.488889			33	
5	0.66666667	4.622222			34	
5	0.66666667	4.755556			35	
5	0.66666667	4.888889			36	
			0.1111111	0.1666667	0.1667	0.1
		Total			**Total**	
		5			36.167	

Process2 (orange)

NoOfThreads	memFactor	Leftover Correction	cpuTime	Remainder (cpu)	Remainder (Real)	Thread time
5	0.6666667		0.13333			1
5	0.6666667		0.26667			2
5	0.6666667		0.4			3
5	0.6666667		0.53333			4
5	0.6666667		0.66667			5
5	0.6666667		0.8			6
5	0.6666667		0.93333			7
5	0.6666667		1.06667			8
5	0.6666667		1.2			9
5	0.6666667		1.33333			10
5	0.6666667		1.46667			11
5	0.6666667		1.6			12
5	0.6666667		1.73333			13
5	0.6666667		1.86667			14
5	0.6666667		2			15
5	0.6666667		2.13333			16
5	0.6666667		2.26667			17
5	0.6666667		2.4			18
5	0.6666667		2.53333			19
5	0.6666667		2.66667			20
5	0.6666667		2.8			21
5	0.6666667		2.93333			22
5	0.6666667		3.06667			23
5	0.6666667		3.2			24
5	0.6666667		3.33333			25
5	0.6666667		3.46667			26
5	0.6666667		3.6			27
5	0.6666667		3.73333			28
5	0.6666667		3.86667			29
5	0.6666667		4			30
5	0.6666667		4.13333			31
5	0.6666667		4.26667			32
5	0.6666667		4.4			33
5	0.6666667		4.53333			34
5	0.6666667		4.66667			35
5	0.6666667	0.1	4.83333			36
				0.1666667	0.25	0.25
			Total			**Total**
			5			36.25

Process1 (purple)

NoOfThreads	memFactor	cpuTime	Remainder (cpu)	Remainder (Real)	Thread time	Leftover *3 threads
3	0.6666667	0.222222			1	
3	0.6666667	0.444444			2	
4	0.6666667	0.611111			3	
4	0.6666667	0.777778			4	
4	0.6666667	0.944444			5	
4	0.6666667	1.111111			6	
4	0.6666667	1.277778			7	
4	0.6666667	1.444444			8	
4	0.6666667	1.611111			9	
4	0.6666667	1.777778			10	
4	0.6666667	1.944444			11	
4	0.6666667	2.111111			12	
4	0.6666667	2.277778			13	
4	0.6666667	2.444444			14	
4	0.6666667	2.611111			15	
4	0.6666667	2.777778			16	
4	0.6666667	2.944444			17	
4	0.6666667	3.111111			18	
4	0.6666667	3.277778			19	
4	0.6666667	3.444444			20	
4	0.6666667	3.611111			21	
4	0.6666667	3.777778			22	
4	0.6666667	3.944444			23	
4	0.6666667	4.111111			24	
4	0.6666667	4.277778			25	
4	0.6666667	4.444444			26	
4	0.6666667	4.611111			27	
4	0.6666667	4.777778			28	
4	0.6666667	4.944444			29	
			0.05555556	0.0833333		0.5
		Total			**Total**	
		5			29.083	

Process3 (purple)

NoOfThreads	**memFactor**	**Leftover Correction**	**cpuTime**	**Thread time**
4	0.666666667		0.166667	1
4	0.666666667		0.333333	2
4	0.666666667		0.5	3
4	0.666666667		0.666667	4
4	0.666666667		0.833333	5
4	0.666666667		1	6
4	0.666666667		1.166667	7
4	0.666666667		1.333333	8
4	0.666666667		1.5	9
4	0.666666667		1.666667	10
4	0.666666667		1.833333	11
4	0.666666667		2	12
4	0.666666667		2.166667	13
4	0.666666667		2.333333	14
4	0.666666667		2.5	15
4	0.666666667		2.666667	16
4	0.666666667		2.833333	17
4	0.666666667		3	18
4	0.666666667		3.166667	19
4	0.666666667		3.333333	20
4	0.666666667		3.5	21
4	0.666666667		3.666667	22
4	0.666666667		3.833333	23
4	0.666666667		4	24
4	0.666666667		4.166667	25
4	0.666666667		4.333333	26
4	0.666666667		4.5	27
4	0.666666667	0.5	5	28
			Total	**Total**
			5	**28**

Appendix B Calculations for Test 30

Process1 Threads 2,3,7,8

memFactor	Threads	cpuTime	RealTime
0.75	2	0.375	1
0.53571429	2	0.64286	2
0.53571429	2	0.91071	3
0.53571429	2	1.17857	4
0.53571429	2	1.44643	5
0.53571429	2	1.71429	6
0.53571429	2	1.98214	7
0.53571429	2	2.25	8
0.53571429	2	2.51786	9
0.53571429	2	2.78571	10
0.53571429	2	3.05357	11
0.53571429	2	3.32143	12
0.53571429	2	3.58929	13
0.53571429	2	3.85714	14
0.53571429	2	4.125	15
0.53571429	2	4.39286	16
0.53571429	2	4.66071	17
0.53571429	2	4.92857	18
0.53571429	2	5.19643	19
0.53571429	2	5.46429	20
0.53571429	2	5.73214	21
0.53571429	2	6	22
0.53571429	2	6.26786	23
0.53571429	2	6.53571	24
0.53571429	2	6.80357	25
		0.19643	0.366667
			Total
			25.36667

Process1 Threads 4 and 9

memFactor	**Threads**	**cpuTime**	**RealTime**
0.6666667	2	0.33333	1
0.5384615	2	0.60256	2
0.5384615	2	0.87179	3
0.5384615	2	1.14103	4
0.5384615	2	1.41026	5
0.5384615	2	1.67949	6
0.5384615	2	1.94872	7
0.5384615	2	2.21795	8
0.5384615	2	2.48718	9
0.5384615	2	2.75641	10
0.5384615	2	3.02564	11
0.5384615	2	3.29487	12
0.5384615	2	3.5641	13
0.5384615	2	3.83333	14
0.5384615	2	4.10256	15
0.5384615	2	4.37179	16
0.5384615	2	4.64103	17
0.5384615	2	4.91026	18
0.5384615	2	5.17949	19
0.5384615	2	5.44872	20
0.5384615	2	5.71795	21
0.5384615	2	5.98718	22
0.5384615	2	6.25641	23
0.5384615	2	6.52564	24
0.5384615	2	6.79487	25
		0.20513	**0.380952**
			Total
			25.38095

Process1 Threads 5 and 6

memFactor	Threads	cpuTime	RealTime
0.6666667	1	0.66667	1
0.5384615	1	1.20513	2
0.5384615	1	1.74359	3
0.5384615	1	2.28205	4
0.5384615	1	2.82051	5
0.5384615	1	3.35897	6
0.5384615	1	3.89744	7
0.5384615	1	4.4359	8
0.5384615	1	4.97436	9
0.5384615	1	5.51282	10
0.5384615	1	6.05128	11
0.5384615	1	6.58974	12
		0.41026	0.761905
			Total
			12.7619

Process2 Threads 2 and 9

memFactor	Threads	cpuTime	RealTime
0.75	2	0.375	1
0.54545455	2	0.64773	2
0.54545455	2	0.92045	3
0.54545455	2	1.19318	4
0.54545455	2	1.46591	5
0.54545455	2	1.73864	6
0.54545455	2	2.01136	7
0.54545455	2	2.28409	8
0.54545455	2	2.55682	9
0.54545455	2	2.82955	10
0.54545455	2	3.10227	11
0.54545455	2	3.375	12
0.54545455	2	3.64773	13
0.54545455	2	3.92045	14
0.54545455	2	4.19318	15
0.54545455	2	4.46591	16
0.54545455	2	4.73864	17
0.54545455	2	5.01136	18
0.54545455	2	5.28409	19
0.54545455	2	5.55682	20
0.54545455	2	5.82955	21
0.54545455	2	6.10227	22
0.54545455	2	6.375	23
0.54545455	2	6.64773	24
0.54545455	2	6.92045	25
		0.07955	0.145833
			Total
			25.14583

Process2 Threads 3 and 4

memFactor	Threads	cpuTime	RealTime
0.75	1	0.75	1
0.55555556	1	1.30556	2
0.55555556	1	1.86111	3
0.55555556	1	2.41667	4
0.55555556	1	2.97222	5
0.55555556	1	3.52778	6
0.55555556	1	4.08333	7
0.55555556	1	4.63889	8
0.55555556	1	5.19444	9
0.55555556	1	5.75	10
0.55555556	1	6.30556	11
0.55555556	1	6.86111	12
		0.13889	0.25
			Total
			12.25

Process2 Threads 5, 7,8

memFactor	Threads	cpuTime	RealTime
0.6666667	1	0.66667	1
0.5625	1	1.22917	2
0.5625	1	1.79167	3
0.5625	1	2.35417	4
0.5625	1	2.91667	5
0.5625	1	3.47917	6
0.5625	1	4.04167	7
0.5625	1	4.60417	8
0.5625	1	5.16667	9
0.5625	1	5.72917	10
0.5625	1	6.29167	11
0.5625	1	6.85417	12
		0.14583	0.259259
			Total
			12.25926

Process2 Thread 6

memFactor	Threads	cpuTime	RealTime
0.6666667	1	0.66667	1
0.55	1	1.21667	2
0.55	1	1.76667	3
0.55	1	2.31667	4
0.55	1	2.86667	5
0.55	1	3.41667	6
0.55	1	3.96667	7
0.55	1	4.51667	8
0.55	1	5.06667	9
0.55	1	5.61667	10
0.55	1	6.16667	11
0.55	1	6.71667	12
		0.28333	0.515152
			Total
			12.51515