# Design Study - Team 28

# Project1: Heated Plate – September 21, 2014

## A. Team 28:

|  |  |  |
| --- | --- | --- |
| **Team-28 Name** | **Team 28 -GT ID** | **Email** |
| Drew Evangelista | **gtg829m** | drewevangelista@gatech.edu |
| Prabhakar Kallakuri | **pkallakuri** | prabhakarkallakuri@gatech.edu |
| Michael Landolfi | **mlandolfi** | landolfi@gatech.edu |
| David Welker | **dwelker3** | dwelker3@gatech.edu |
| Jeffrey Wishnie | **jwishnie** | jwishnie@gatech.edu |

**(Paper is seven pages of text; without images and tables)**

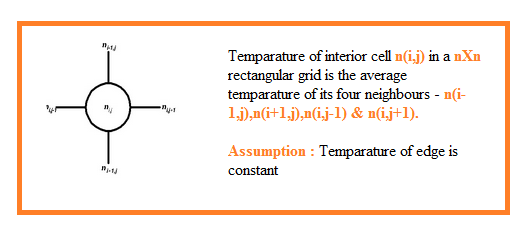
**B. Design Study Methodology**

**Introduction:**

This design study compares four algorithms with respect to their performance, memory usage, and the number of iterations they require to converge.

**Context:**

The phenomenon underlying this design study is the diffusion of heat through a uniform two dimensional metal plate. In order to better understand this process, we have developed four simulation programs in Java that compute the temperature of the each interior cell in the grid as the average of its four neighbors with the temperature of the plate’s edges held constant.



We will be analyzing and researching the behavior of a simulation program for various constraints such as precision, time, initial temperature of edges, size of grid, number of iterations, etc. The purpose is to compare the behavior of these programs in response to change in these conditions. Enough information is included in this report so that someone else could repeat our findings.

**Research Questions:**

Our design study addresses the following research questions:

1. How does processing time vary for each of the algorithms as the dimension of the rectangular plate grows from 100x100 to 1000x1000?
2. How does the memory use of each of the algorithms vary on a plate with dimensions 200x200?
3. How does increasing the dimensionality from 10x10 to 100x100 affect the number of iterations required for one of the four algorithms to complete?
4. How does the precision used to represent floating-point numbers (float versus double) impact performance, memory, and number of iterations to algorithm convergence?

# Subjects:

As part of design study and to fully understand the diffusion of heat through a uniform two-dimensional metal plate we have created five Java programs. The first four programs namely, Tpdahp, Tpfahp, Twfahp, and Tpdohp, produce textual output and the fifth program, Gallhp, is a GUI program that invokes these 4 programs and displays the results of executing them graphically.

|  |  |  |
| --- | --- | --- |
| **Naming convention for programs** | | |
|
| **Position 1-6** | **Value** | **Description** |
|
| 1 | T /G | **T** - Textual Output **G-** Graphical Output |
| 2 | p/d/w | **p** - Primitive Datatype **w** -Wrapper Class |
| 3 | d/f | **d** -Double Precision **f** -Float Precision |
| 4 | a/o | **a** -Array Indexing used to access neighboring elements **o** -Object References |
| 5 & 6 | hp | **hp** -Heated Plate |

# Experimental conditions:

The simulation programs were run on a machine with the characteristics described below:

|  |  |
| --- | --- |
| **Environment** | **Details** |
| Work station | MacBook Pro |
| Operating System | Mac OS X 10.9.4 |
| Java Version | 1.7.0\_45 |
| Virtual Machine | Java HotSpot(TM) 64-Bit Server VM |
| CPU /Processor | 2.7 GHz Intel Core i7 |
| Memory | 16 GB 1600 MHz DDR3 NVIDIA GeForce GT 650M 1024 MB |
| Default Parameters | Unless otherwise noted, our experiments were run with a dimension of 50x50 and an initial temperature of 50 for the left, right, top, and bottom edges. |
| Confounding Factors | The experiments were executed from Netbeans IDE 8.01. |

1. **Independent variables**
2. Precision: The number of significant digits that can be stored in a floating point number. In this study, precision is determined by whether an algorithm that uses floats versus doubles is used.
3. Dimension: An integer that determines the size of the grid. For example, if n = 10, a 10x10 grid will be used.
4. Temperature: These are integers that determine the temperatures at the edges of the plate which then propagate. Four temperatures can be set, one for each edge.
5. Relative Change Stopping Criterion: The percentage change threshold that must be exceeded for the simulation to continue. In all of our experiments, the threshold used was 0.1% and this value was stored in a floating-point number.
6. **Dependent Variables:**
7. Memory: The amount of additional memory needed to execute the program above and beyond that needed to execute the JVM by itself. This was measured in bytes.
8. Performance: The time needed to execute each of the four programs. This was measured in milliseconds.
9. Iterations: The number of times a new plate is calculated using values from an old plate. This was measured using an integer.
10. **Metrics:**

|  |  |  |
| --- | --- | --- |
| **Variable** | **Unit** | **Description** |
| Precision | bits | The number of bits used to represent the floating point number. For float, 32-bits, and for double, 64-bits. |
| Dimension | integer | The size of the grid. |
| Temperature | integer | The temperature applied to the grid edges. |
| Stopping Criterion | float | A ratio that indicates whether execution should continue. A change of 0.1% is represented by floating point number 0.001. |
| Memory | bytes | The number of bytes that an algorithm requires above the number of bytes required to run the JVM. |
| Performance | milliseconds | The number of millisecond that an algorithm runs before stopping. |
| Iterations | integer | The number of times the plate is recalculated before stopping. |

**Method:**

The four programs Tpdahp, Tpfahp, Twfahp, and Tpdohp can be invoked using commands with the following format: java <algorithm>.Demo -d <int> -t <int> -b <int>-l <int> -r <int>

For Example: java Tpdahp.Demo -d 9 -t 50 -b 50 -l 50 -r 50

**Number of trials:**

The number of trials used varied based on whether performance, memory or iterations were measured and is evident in the results section.

**Measurement devices and tools:**

The tool was used to measure outputs varied based on the target. For performance, the System class was used, for memory, the Runtime class was used, and for iterations, the custom GUI we developed was used.

**Significant digits:**

The heated plate algorithm generates textual output with 2 significant digits to the right of the decimal.

**B. Design Study Methodology**

**Initial conditions:**

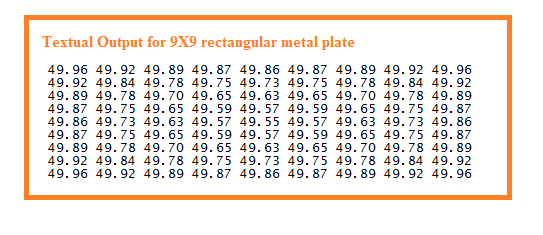
The temperature of edge is kept constant as shown in above example. By default, we used a dimension of 50 and a uniform temperature of 50 for all edges, though we have varied that for some experiments as indicated.

**Coarseness of the lattice**:

The size of grid is varied from 100x100 to 1000x1000 and a comparison report of processing times for all the four subjects or programs was studied.

**Results:**

The results of a sample execution is displayed as a textual output below:



**Performance:**

Fixed:

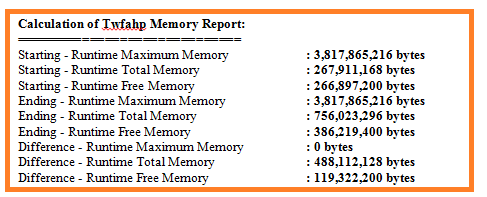
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Fixed - Algorithm comparison** | | | | |
| **Algorithm** | **Size of rectangular plate** | **Temperature of edges** | **Processing time in milliseconds** | **Percent Change (%)** |
|
| Tpdahp | 500X500 | 50L 50R 50T 50B | 11596 | 0.1 |
| Tpfahp | 500X500 | 50L 50R 50T 50B | 12577 | 0.1 |
| Twfahp | 500X500 | 50L 50R 50T 50B | 36593 | 0.1 |
| Tpdohp | 500X500 | 50L 50R 50T 50B | 17641 | 0.1 |

Variable:

|  |  |  |  |
| --- | --- | --- | --- |
| **Tpdahp Algorithm Performance Report** | | | |
|
| **Size of rectangular plate** | **Temperature of edges** | **Processing time in Milliseconds** | **Percent Change (%)** |
|
| 100X100 | 50L 50R 50T 50B | 290 | 0.1 |
| 200X200 | 50L 50R 50T 50B | 1028 | 0.1 |
| 300X300 | 50L 50R 50T 50B | 3074 | 0.1 |
| 400X400 | 50L 50R 50T 50B | 6988 | 0.1 |
| 500X500 | 50L 50R 50T 50B | 11596 | 0.1 |
| 600X600 | 50L 50R 50T 50B | 17142 | 0.1 |
| 700X700 | 50L 50R 50T 50B | 22710 | 0.1 |
| 800X800 | 50L 50R 50T 50B | 28892 | 0.1 |
| 900X900 | 50L 50R 50T 50B | 37183 | 0.1 |
| 1000X1000 | 50L 50R 50T 50B | 45742 | 0.1 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Tpfahp Algorithm Performance Report** | | | | | | |
|
| **Size of rectangular plate** | | **Temperature of edges** | | **Processing time in Milliseconds** | | **Percent Change (%)** |
|
| 100X100 | | 50L 50R 50T 50B | | 288 | | 0.1 |
| 200X200 | | 50L 50R 50T 50B | | 1101 | | 0.1 |
| 300X300 | | 50L 50R 50T 50B | | 3288 | | 0.1 |
| 400X400 | | 50L 50R 50T 50B | | 7397 | | 0.1 |
| 500X500 | | 50L 50R 50T 50B | | 12577 | | 0.1 |
| 600X600 | | 50L 50R 50T 50B | | 17881 | | 0.1 |
| 700X700 | | 50L 50R 50T 50B | | 24415 | | 0.1 |
| 800X800 | | 50L 50R 50T 50B | | 31480 | | 0.1 |
| 900X900 | | 50L 50R 50T 50B | | 42385 | | 0.1 |
| 1000X1000 | | 50L 50R 50T 50B | | 52425 | | 0.1 |
|  | | | | | | | |
| **Twfahp Algorithm Performance Report** | | | | | | | |
|
| **Size of rectangular plate** | **Temperature of edges** | | **Processing time in Milliseconds** | | **Percent Change (%)** | | |
|
| 100X100 | 50L 50R 50T 50B | | 551 | | 0.1 | | |
| 200X200 | 50L 50R 50T 50B | | 2960 | | 0.1 | | |
| 300X300 | 50L 50R 50T 50B | | 11160 | | 0.1 | | |
| 400X400 | 50L 50R 50T 50B | | 24668 | | 0.1 | | |
| 500X500 | 50L 50R 50T 50B | | 36593 | | 0.1 | | |
| 600X600 | 50L 50R 50T 50B | | 52538 | | 0.1 | | |
| 700X700 | 50L 50R 50T 50B | | 68859 | | 0.1 | | |
| 800X800 | 50L 50R 50T 50B | | 98456 | | 0.1 | | |
| 900X900 | 50L 50R 50T 50B | | 115529 | | 0.1 | | |
| 1000X1000 | 50L 50R 50T 50B | | 148708 | | 0.1 | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Tpdohp Algorithm Performance Report** | | | |
|
| **Size of rectangular plate** | **Temperature of edges** | **Processing time in milliseconds** | **Percent Change (%)** |
|
| 100X100 | 50L 50R 50T 50B | 244 | 0.1 |
| 200X200 | 50L 50R 50T 50B | 1635 | 0.1 |
| 300X300 | 50L 50R 50T 50B | 3722 | 0.1 |
| 400X400 | 50L 50R 50T 50B | 9489 | 0.1 |
| 500X500 | 50L 50R 50T 50B | 17641 | 0.1 |
| 600X600 | 50L 50R 50T 50B | 29881 | 0.1 |
| 700X700 | 50L 50R 50T 50B | 40396 | 0.1 |
| 800X800 | 50L 50R 50T 50B | 52975 | 0.1 |
| 900X900 | 50L 50R 50T 50B | 67073 | 0.1 |
| 1000X1000 | 50L 50R 50T 50B | 83442 | 0.1 |

1. **Runtime memory use:**
2. Snapshot of calculation of memory using java Runtime class
3. ****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Size of rectangular plate** | **Temperature of edges** | **Type -Tpdahp** | **Processing time** | **Memory usage in bytes** |
|
| 200X200 | 50L 50R 50T 50B | Primitive Double | 1028 | Runtime Total Memory: 14,155,776 bytes |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Size of rectangular plate** | **Temperature of edges** | **Type-Tpfahp** | **Processing time** | **Memory usage in bytes** |
|
| 200X200 | 50L 50R 50T 50B | Primitive  Float | 1101 | Runtime Total Memory: 13,631,488 bytes |
|  |  |  |  |  |
| **Size of rectangular plate** | **Temperature of edges** | **Type-Twfahp** | **Processing time** | **Memory usage in bytes** |
|
| 200X200 | 50L 50R 50T 50B | Wrapper classes-  Float | 2960 | Runtime Total Memory: 488,112,128 bytes |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Size of rectangular plate** | **Temperature of edges** | **Type-Tpdohp** | **Processing time** | **Memory usage in bytes** |
|
| 200X200 | 50L 50R 50T 50B | Object reference | 1635 | Runtime Total Memory: 15,204,352 bytes. |
|  |  |  |  |  |

1. **Model Representation:**

By observing the memory report above, we could conclude that wrapper classes program **Twfahp** takes 488,112,128 bytes i.e. almost thirty five time times that of primitive data types. In addition, we can determine that primitive double algorithm has optimum performance and optimum memory usage. Although it is more computationally expensive, the wrapper class is slightly more elegant.

1. **Precision:**
2. By comparing processing time and memory usage of 200X200 rectangular metal grid for primitive float and double data types, we can conclude that Tpfahp program uses less run time memory but more processing time than that of Tpdahp program.
3. **Number of iterations:**
   * + 1. Number of iterations would be more for large sized grids when compared to smaller sized grids. In addition, for a constant grid size and edge temperature there would be zero or minimal difference in count of iterations for all four version of programs i.e. for a 3X3 grid, the count of iterations is same for Tpdahp and Tpfahp programs.

|  |  |  |
| --- | --- | --- |
| Program Type | Size of rectangular plate | Number of Iterations |
|
| Tpdahp | 10X10 | 119 |
| Tpdahp | 20X20 | 225 |
| Tpdahp | 30X30 | 566 |
| Tpdahp | 40X40 | 821 |
| Tpdahp | 50X50 | 1081 |
| Tpdahp | 60X60 | 1340 |
| Tpdahp | 70X70 | 1597 |
| Tpdahp | 80X80 | 1852 |
| Tpdahp | 90X90 | 2105 |
| Tpdahp | 100X100 | 2356 |

1. **Usability:**
2. **Controls:**
3. We actually implemented two versions of the GUI program before settling on one for this assignment. The main difference between them consist of their controls. In the first program, we used radio buttons to select an algorithm and kept all parameters grouped together. For the second program, we used a drop down list to select the algorithm and put the laid out the parameters or left, right, top, and bottom visually.
4. **Presentation:**

With respect to presentation, two different options would be to represent the hotness of the plate with one color only, such as red, where bright red represents a very hot plate and black represents a very cold plate. Another, more colorful alternative would be to use the colors of the rainbow, where colors on the rainbow associated with cooler temperatures would be used.

In order to select between these options, usability studies where users are recruited and are given an opportunity to express their preferences would be extremely helpful. Resources permitting, prototypes would be developed, although it is also possible to survey them on their conceptual preferences using mockups short of a prototype.

**Adaptability:**

|  |  |  |
| --- | --- | --- |
| **Functional Requirements** | | |
| **Scenario** | **Description** | **Effect on Design** |
| Add Algorithm | Adding another algorithm implementation that simulates a heating plate. | The combo box would have to be expanded to include the new algorithm. The code which runs simulations would have to be modified to execute the new algorithm. |
| 3D Heat Plate | Add an algorithm that functions on a three dimensional grid instead of a two dimensional grid. | The GUI would have to be adapted to show a cube instead of a square. The user would need to set temperatures for heating six sides rather than four edges. |
| Run Multiple Algorithms | Currently, the user must execute the algorithms one at a time. This feature would enable the user to specify multiple algorithms to run. Each algorithm would run with the same parameters and would output a relevant report that compares the performance of the algorithms. | Instead of a drop down, the GUI would need to allow the user to add the algorithms they would like to execute in a list. The run button would then have to execute all of the algorithms in the list, instead of only the currently selected algorithm as it does currently. Optionally, these algorithms could execute in parallel to better leverage multithreading processors, only to combine their results into a single report. |
| Parameter Sweep | Currently, if the user wants to run the same algorithm with different parameters, they must run the algorithm, manually change parameters, and run it again. Instead of requiring this, this feature would allow the user to specify that a parameter sweep that would automatically run the same algorithm with different parameter values. The results would be aggregated in a single report that would allow the user to compare the results. | GUI would have to be modified to specify a parameter sweep. For the simplest case, a sweep would include only one parameter while holding everything constant. But, in a more advanced version, multiple parameters could be varied at once, although the danger here is that number of runs of the program would grow exponentially. In addition, the run method would have to be modified to run the same algorithms multiple times with different parameters. Again, these could be run in parallel to exploit multithreading architectures. |

|  |  |  |
| --- | --- | --- |
| **Non -Functional Requirements** | | |
| **Scenario** | **Description** | **Effect on Design** |
| Increase Usability of GUI | Currently, the GUI does not provide any method of determining specific cell values. One possibility for addressing this would be to show the value of a cell in a tooltip when the user hovers over it. Also, feedback from user studies could be obtained to make the GUI otherwise more user friendly. | Adding tooltips or otherwise changing the GUI ought to usually be fairly straightforward and not have deep architectural implications. Most of the effort would go into getting feedback from users and reconciling conflicting advice. |
| Increase Extensibility Via Plugins | Currently, to add a new algorithm to the list of selections for a user, manual code changes are needed. The proposal here is to increase extensibility by having new algorithms implement a common interface and performing a standard installation procedure. | To enable this, the current interface for algorithms (currently called Simulator) would have to be carefully scrutinized to ensure that all required methods are present. It currently appears that this interface is already good enough for the purpose. A second change would involve how algorithms are placed in the combo selection box and how they are retrieved from it. Changes would have to be made so that if a JAR file is in a conventionally recognized location, any algorithms provided by that JAR would be loaded into the combo box. The run functionality then would need to retrieve a reference to the algorithm from a registry rather than depending on built-in knowledge of the algorithm based on the String value in the combo box. |

**Discussion:**

The experiment involved the creation of small custom Java programs and it was necessary to prevent other programs from running as much as possible to avoid interference with the results. Considering performance first, interestingly, using primitive doubles instead of primitive floats improved performance. While this effect was only about 10 to 15 percent, a much more significant performance effect occurred when moving from primitive floats to wrapped floats. Primitive floats were between two and three times as fast. As would be expected, there was also a small performance penalty for moving from double arrays to double objects; but this was not nearly as dramatic as the penalty for using wrapped floats.

Moving next to consider memory, as expected, floats take less memory as double. However, the effect was much less dramatic than expected. Although doubles take 64-bits instead of 32-bits, the memory improvement for using floats was only 10 to 15 percent. In contrast, moving from using primitive floats to wrapped floats consumed over 35 times as much memory. Using objects instead of arrays for doubles created a minor memory penalty, similar to the penalty for preferring doubles rather than floats. Overall, moving from doubles to floats has a small memory benefit, but a small performance penalty. Moving from arrays to objects has a small negative effect on both memory and performance. Finally, moving from primitives to wrapped objects has a very large negative effect on performance and a huge negative effect on memory.

While we did not observe differences between the algorithms, we did observe a linear increase in the number of iterations required for convergence as dimension increased with respect to the number of iteration.

The study process could be improved by creating tools that allow for multiple algorithms to run in parallel, and the same program to be run multiple times, as suggested by the change scenarios in the adaptability section. One interesting anomaly is that doubles had better performance than floats. There does not seem to be any principled explanation for why a larger data type should be faster to process than a smaller data type. We speculate that the Java Virtual Machine is better optimized for handling 64-bit doubles than 32-bit floats.

**Conclusion:**

Our study successfully enabled us to answer the four research questions we posed. First, we found that processing times grew exponentially for all four algorithms, but that using wrapped floats was most expensive. Second, with respect to memory, there was a huge penalty for using wrapped values, but only a surprisingly small penalty for using doubles instead of floats. Third, with respect to iterations, we found that as the number of dimensions increased, the number of iterations required for the algorithm to converge increased linearly. Fourth, we found that doubles were best for maximizing performance while floats were best for minimizing memory use. As with so many things in computer science, there are unavoidable trade-offs here. To maximize performance and minimize memory, the algorithms Twdahp and Tpdohp would certainly be rejected, and the choice would be between Tpdahp and Tpfahp depending on whether the priority was maximizing performance or minimizing memory use.