Comparing GPU and CPU implementations of agent based systems

Student Name: Adam Harries

Supervisor Name: Dr Tom Friedetzky

Submitted as part of the degree of BSc Computer Science to the

Board of Examiners in the School of Engineering and Computing Sciences, Durham University

Abstract —

Context/Background - In the last few years, graphical processing units have transitioned from being pure graphics production equipment to complex parallel processors. One potential application for the paralleism of GPGPU (General Purpose computation on Graphical Processing Units) is the implementation of multi-agent systems, software systems that are modelled as the interation of multiple independent software 'agents'. It therefore seems logical to investigate the possibility of applying GPGPU techniques to them, and evaluate the benefits from doing so.

Aims - This project aims to implement a three multi-agent systems, a flocking algorithm, a genetic algorithm, and an ant colony optimisation algorithm using GPGPU techniques, and compare their properties (e.g. speed, code size) with serialally implemented counterparts. The project will also aim to determine if any improvements are a result of parallelisation, or other factors.

Method - Three different multi-agent systems will be implemented, both using GPGPU techniques, and with traditional serial techniques. The systems will then be compared using a number of factors, including (but not limited to) code size, and time taken per iteration of the systems.

Results - The GPGPU implementations were in general slower than their serial counterparts, but scaled better as problem instances became more difficult and agent counts increased, implying that the parallelisation did in fact help to accellerate the algorithms.

Conclusions - The mixed results suggest that automatic parallelisation of agent based systems may be possible, but any system doing so would need substantial work on its optimisation capabilities in order to generate competitively fast code.

Keywords — GPGPU, CUDA, Multi-Agent Systems, Agent Based Systems, Parallel, Flocking algorithms, Genetic Algorithms, Ant Colony Optimisation

I INTRODUCTION

This project aims to look at the potential application of parallel graphics processors to the acceleration of agent based systems. In particular, this project investigates whether it would be possible to speed up *arbitrary* agent based systems, as opposed to ones specifically written for GPU acceleration.

A Context

General purpose GPU programming (GPGPU) - As many recent papers on parallel and distributed computing are quick to remark, it is currently generally accepted that Moore's law [1] is slowing down. A generally corollary of Moore's law is that the procesisng speed of computer processors will also similarly increase, and allow for more data to be processed in less time. However, given the slowdown in Moore's law, we can no longer rely on sheer speed to allow for more complex and time consuming calculations. One possibly way to reduce our dependence on fast hardware is to write far more parallel programs which instead run across multiple slower processing units. An increasingly common way of achieving this is to utilise the parallel nature of modern graphical processing units.

Dedicated graphical processing units (also known as GPUs) are a class of processors which are highly optimised for graphical computations, such as vertex transformation, or rasterisation. Early GPUs were only able to perform graphics operations, however within the last ten years, GPUs have transformed into highly parallel processors.

GPU parallelism is a variant of single instruction multiple data [2](SIMD) parallel classification. In 'normal' SIMD operations, a single instruction is applied to a vector of data to produce an equivalently sized vector of results. Within GPUs however, it is more common to find a number of individual SIMD processors (in CUDA terminology known as *streaming multiprocessors*), which allow the card to carry out a number of different operations at once on a much larger set of data.

Multi-agent systems - 'Multi-agent systems' is a wide term describing a large class of software systems, all of which can be conceptually divided into a system of discrete agents. Multi-agent systems are used for a wide variety of purposes across a large number of disciplines in computer science. Of particular interest to this project are the uses of multi-agent systems for simulation, and for providing approximations to hard problems.

In simulation, multi-agent systems have been used to simulate traffic flow (Doniec et al. 2008), trading behaviour in stock markets (Palmer et al. 1994), flocking algorithms (Reynolds 1987) (discussed further below), and many other behaviours. The usage of multi-agent systems as approximation and optimisation algorithms is also common. Well known search heuristics such as genetic algorithms, particle swarm optimisation algorithms and ant-colony optimisation algorithms are all either directly agent-based systems, or can be reasoned about as such.

This project specifically focusses on systems that can be represented using framework similar to the distributed map/reduce framework. Multi-agents systems that fit this framework can all be defined using the repeated application of two functions, dubbed `map` and `reduce`. In the `map` function a procedure `P` is applied to each of the agents independently. This function can read the state of the entire system, but can only modify the agent it is applied to. The `reduce` function acts across the entire system, and can read and update the environment based upon the state of the agents, or further modify the agents in the system.

From here on, such systems will be referred to as 'map/reduce multi-agent systems' or 'map/reduce agent-based systems'.

Combination - As multi-agent systems are formed of multiple discrete independent software systems, they would appear to be ideal candidates to attempt to parallelise. This is especially

[1] Moores law states that, roughly, the number of transistors on each integrated circuit is doubling every two years.

[2] One class of parallel architecture under Flynn's taxonomy, see (Flynn 1972) for more detail.

true of the category of agent based systems that this project focusses on, in particular the `map` operation. It would be therefore prudent to examine whether systems of this type can be easily, and productively parallelised using GPUs.

B Research question

This project aims to answer the following research question:

'Do map/reduce style agent based system intrinsically benefit from GPU parallelisation, without any further optimisations.'

In other words, given a simple multi-agent system that conforms to the map/reduce framework described above, does adapting it for GPU execution provide a speedup *without making any further modifications*. Further modifications refers to optimisations or modifications to the GPU implementation above and beyond those required to simply allow the system to run using the GPU. To attempt to answer the question, three different agent based systems have been implemented using both serial and parallel programming techniques, and their runtimes compared to determine if any speedup has occurred.

C Project Aims

The aim of this project is to measure what accelleration (if any) could be applied to map/reduce agent based systems through the use of GPU parallelism. This would be carried out through the comparison of serial (CPU) and parallel (GPU) implementations of 3 different agent based systems. Also envisioned was the implementation of a generic agent based system framework for generating both serial and parallel code, to control for any differences between the hand crafted code. The requirements below detail how these tasks were split into minimum, intermediate and advanced requirements.

Minimum requirement - The minimum requirement for the project was the serial implementation of the three algorithms selected. The three algorithms were: Reynolds flocking autonoma (Reynolds 1987), a genetic optimisation algorithm, and an ant colony optimisation algorithm.

Intermediate requirement - The intermediate requirement was the implementation of the same algorithms as the basic requirement, but utilising the CUDA toolkit to produce a parallel implementation of the algorithm.

Advanced requirement - The advanced requirement was the development of a generic library or language for the implementation of map/reduce agent based systems. The library would be able to generate either serial CPU code, or parallel GPU code.

Of the three requirements, both the first two have been completed, while the final (advanced) requirement was determined to be too difficult and time consuming for the constraints of the project. A more detailed exploration of the difficulties and issues faced during the attempted implementation can be found in the Solution, Evaluation and Results sections.

D Project conclusions

The project concluded that GPU parallelisation of map/reduce agent based systems does produce some speedup, however that speedup is only realised in systems with highly complex map functions, or when system parameters (such as the number of agents, or the size of the problem instance) grow particularly large. This is encouraging for the prospected of automated parallelism parallelism, but also highlights the importance of GPU specific optimisations when porting programs.

II RELATED WORK

A Map/Reduce, and the GPU

Although GPU acceleration of specific algorithms is useful, this project focuses more in the implementation of the class of algorithms that can be implemented and reasoned about using a modified version of the map/reduce framework. (Dean & Ghemawat 2004) goes into detail about the structure of the original MapReduce framework, and its applicability and implementation details in a massively distributed environment. However, with more relevance to this project, map/reduce ideas have also been explored in a GPU context. Notable examples are the Mars framework (He et al. 2008) and (He et al. 2008). Although neither project deals specifically with agent based systems, both have reported good better performance than CPU implementations of the algorithms implemented.

There has been little obvious work in implementing agent based systems using a map/reduce framework on the GPU. However, there has been a large body of work in distributed map/reduce implementations. In terms of genetic algorithms, (Verma et al. 2009), (Jin et al. 2008), (Huang & Lin 2010) and (Zhou 2010) have all been successful in adapting evolutionary algorithms to a distributed MapReduce context. There has been less work on implementing ant colony optimisation algorithms using MapReduce techniques, possibly due to its younger age, and more complicated structure. However, a number of approaches to MapReduce parallelisation have been proposed. Two different approaches (similar to those described in (Sttzle 1998)) are described in (Wu et al. 2012). The first, similar to the approach taken in this project, processes a single ant to each 'Map' operation, and updates the environment in the 'Reduce' operation. The second proposes running a separate colony for a number of iterations in each 'Map' operation, and then selecting the best solution from all colonies in the 'Reduce' operation. As with the ant colony algorithm, there has been less work in applying MapReduce techniques to flocking algorithms. Despite this however, there is a large body of work in accellerating more generic agent based simulations and alternative agent based simulation algorithms using the MapReduce framework. The successful application of Hadoop (an open source alternative to the propriatory Google MapReduce framework) to a number of multi-agent systems including including circle forming, and spontaneous applause is described in (Sethia & Karlapalem 2011). In (Wang et al. 2010) a high level framework, BRACE (the Big Red Agent-based Computation Engine), and accompanying domain specific programming language, BRASIL (the Big Red Agent SImulation Language). Also described is the application of BRACE and BRASIL to a number of multi-agent simulation algorithms, including a traffic flow simulation algorithms, and a fish schooling algorithm (different to the flocking algorithm this project covers).

B GPU implementations of agent based systems

The potential for parallelisation of ant colony optimisation algorithms has been long recognised. Prior to the development of programmable graphics cards, (Sttzle 1998) suggested two possible parallelisation strategies. In the first, the algorithm is parallelised 'by ant', that is, the behaviour of each ant is run in parallel with the others. The second strategy involved running multiple separate colonies in parallel, using separate pheromone markers and edge probabilities. Since then, there has been much work in adapting the two parallel strategies to GPU execution, such as (Delvacq et al. 2013). Alternative strategies have also been explored, such as (Cecilia et al. 2011) which explores data parallel approaches to parallelisation, instead of thread parallel approaches. GPU parallelisation of genetic algorithms, as with ant colony optimisation algorithms, has been widely studied. (Johar et al. 2013) gives a good overview of some of the possible approaches and methods for GPU accelleration. As with our project, there has also been a lot of work in solving the travelling salesman problem (TSP) using genetic GPU techniques. Similar approaches to our project can be found in (Cekmez et al. 2013), which also uses an approach that maps a thread to each agent.

As with the previous two algorithms, there has also been a large body of work investigating the GPU parallel potential of flocking algorithms. Mirroring the evolution of GPU hardware, early work such as (Chiara et al. 2004) investigated the application of shader programs (a graphics focussed precursor to current GPU programming toolkits such as CUDA or OpenCL) to flocking algorithms, achieving very positive results. Later work such as (Husselmann & Hawick 2011) has also achieved very positive results with more modern GPGPU techniques.

C High level GPU frameworks

The final objective of this project is the implementation of a generic framework for implementing map/reduce agent based systems on the GPU, however there has been little visible academic work on this specific problem. The previous section noted a number of GPU MapReduce frameworks and implementations, however it is important to briefly note some of the other frameworks and languages that have been developed for automated GPU parallelism. Of interest to this project due to its functional foundations (and hence reliance on functional patterns such as maps and reductions) is the Haskell Accelerate library (described in (Chakravarty et al. 2011) and available at [3] and domain specific language. Accellerate defines an embedded domain specific language to allow programmers to perform high performance computing on a specific class of GPU portable arrays. The seperation of description from implementation in the Accelerate module is also particularly interesting for this project, as it allows the programmer to compile their program for either GPU execution, or for execution using an interpreter using the CPU. Finally, the Vector programming language (Mao 2014) defines a C-like syntax for the simple description of hybrid CPU/GPU programs. GPU execution is declared through the use of `pfor` blocks, which hint to the compiler that statements within the block can be executed in parallel on the GPU. Vector also includes higher order map and reduce functions which allow a further avenue to specifying code for GPU execution.

[3] `http://hackage.haskell.org/package/accelerate-0.14.0.0/docs/Data-Array-Accelerate.html`

III SOLUTION

To answer the research question of this project, a solution has been devised, which can be neatly split into three sections: serial algorithm implementations, parallel algorithm implementations and a generic auto-parallelising implementation. The final section of the solution was found to be beyond the scope of this project, and was therefore not completed. A discussion of the reasons why, and of the unsuccessful approaches taken for implementation can be found later in the Solution section and in the evaluation section.

The serial implementations section describe the specifics of how the chosen algorithms were implemented in a purely serial fashion, that is with a single thread on the GPU and without using the GPU at all. The parallel implementation section describes the methods used and changes made to implement the algorithms utilising the parallel nature of a graphical processing unit.

A Agent based systems

This project implements three different (though similar) agent based systems. The three systems were chosen for their ability to fit within the 'Map-Reduce' abstraction, as well as providing interesting additions to the concept. The systems implemented are:

- **Reynold's 'Boids' flocking algorithm** - a simulation algorithm modelling natural flocking and swarming behaviour
- **A genetic algorithm** - a technique for solving optimisation problems by mimicking the natural process of evolution
- **An ant colony optimisation algorithm** - a technique for solving optimisation problems by imitating the process colonies of ants use to search for food

On a general level, the three algorithms were chosen for their suitability for implementation using something akin to a 'Map/Reduce' operation. Each of the algorithms can be described (at a high level) as a 'Map operation' across all agents (i.e. a function applied to each agent) followed by one or more 'Reduce operations' across the agents, or the world. How each system fits to this abstraction is discussed below, within each system's subsection.

This particular abstraction was chosen as it fits well with the parallel model that GPUs use, which is similar to 'single instruction multiple data' (SIMD) model of parallelism. Within this model, and within each streaming multiprocessor in a GPU, a single instruction is carried out on a vector of data. This is ideally suited for simple functions that act identically and independently across a number of agents.

Although each of the systems fits within the map/reduce abstraction, each have their own unique attributes which could potentially affect how they may be parallelised, and more importantly any resulting effects on performance. The flocking system brings the addition of visual feedback to its reduce operation, while the genetic algorithm requires a 'reduce' operation that acts across the entire set of agents, and the reduce operation of the ant colony optimisation algorithm modifies the environment that the agents are embedded in. All of the attributes of the three systems are substantially different from each other, and this presents interesting challenges to parallelisation.

A.1 'Boids' flocking algorithm

The flocking algorithm that has been implemented is Reynold's 'Boids' algorithm. The Boids system models a flock of agents, by considering three basic behaviours; collision avoidance, velocity matching, and flock centering (Reynolds 1987). To simulate each agent (termed a 'boid' in the algorithm), the algorithm derives its velocity at each iteration (or frame) by summing the directional vectors resulting from considering each of the three basic behaviours.

The first behaviour, collision avoidance, produces a vector which steers the agent away from potential collisions with other nearby agents. For an agent a, with position vector a_s, as part of a flock F, a collision neighbourhood indicator function N_{ab} which evaluates to 1 if another agent b is too close, and 0 otherwise, the collision avoidance vector can be described using the formula: $A = \sum_{b \in F} N_{ab} \times \left(\frac{a_s - b_s}{|a_s - b_s|} \right)$, where $|x|$ is the magnitude of a vector x.

The second behaviour, velocity matching, directs flocks towards the same heading. Again, for an agent a, with velocity vector a_v, as part of a flock F, a neighbourhood function N_{ab} which evaluates to 1 if another agent b is in the neighbourhood of agent a, and 0 otherwise, the velocity matching vector can be described as with the formula: $M = \sum_{b \in F} N_{ab} \times b_v$

The final behaviour, flock centering, creates a cohesive flock by directing boids towards other nearby boids. This function simply finds the mean location of all boids in the 'neighbourhood' of a boid a, decided as above with a neighbourhood function N_{ab}, with the final formula: $C = \sum_{b \in F} N_{ab} \times b_s$.

To calculate the final velocity vector, the three behaviour vectors are normalised, weighted, and summed. The weights associated with each of the behaviour vectors are defined by the parameters of the simulation.

The flocking algorithm fits easily into our map/reduce framework: the calculation of new velocity and location vectors is represented by the function called on each agent during the 'map' procedure, while the graphical display of the system of agents can be considered to be the 'reduce' procedure across the whole generation.

A.2 Genetic algorithms

Genetic algorithms are used to solve hard optimisation problems by drawing inspiration from the process of evolution, and thus 'evolving' better and better solutions to problems.

The algorithm can be thought of as an agent based system by considering each solution in the algorithm as a seperate agent in the system. Each agent has an update (map) method in which it mutates its solution, then calculates its new 'fitness', and the system has a 'reduce' method in which a new population of solutions is 'bred' from the old.

Procedure 1: 'Genetic algorithm' provides a fairly concise listing of the algorithm.

Procedure 1 Genetic algorithm

Input: Problem parameters P
  Generation $G \leftarrow$ RANDOMLY_INITIALISE_GENERATION()
  while Termination condition not reached **do**
    $G' \leftarrow$ BREED_GENERATION(G)
    $G'' \leftarrow$ MUTATE_GENERATION(G')
    $G \leftarrow G''$

The genetic algorithm fits nicely into our class of agent based systems. The mutation of each individual solution generation, followed by a calculation of each individual's fitness fits as the function invoked in the 'map' procudure, and breeding the previous generation to create the next can be thought of as the 'reduce' procedure.

A.3 Ant colony optimisation algorithms

The ant colony optimisation algorithm is an algorithm, developed by Marco Dorigo (Dorigo et al. 1996) for solving computationally hard optimisation problems (expressed as a graph) by mimicking the movements and interactions of a colony of ants. The algorithm is described in Procedure 2: 'Ant colony optimisation'.

Procedure 2 Ant colony optimisation

Input: Edge weights E, Pheromone weights P
  while Termination condition not reached **do**
    for all $ant\ a \in colony$ **do**
      $solution \leftarrow$ CONSTRUCT_SOLUTION(a, E, P)
      for all $edge\ e \in solution$ **do** $Deposit\_pheromone(e)$

In essence, each ant in the colony repeatedly traverses the graph representing the problem building a new solution each time. For each solution generated, the ant then deposits pheromone on the edges included in the solution.

To traverse the graph, the ant moves from a vertex v to another vertex according to a proportionality rule, defined for each edge from a vertex. For some ant k, the proportional rule for the ant moving from vertex v to some other vertex $u \in N$, where N is the neighbourhood of vertex v is defined via the probability:

$$p_{ik}^{k} = \frac{(\tau_{ij})^{\alpha} (\eta_{ij})^{\beta}}{\sum_{l \in N} (\tau_{il})^{\alpha} (\eta_{il})^{\beta}}$$

Where τ_{ij} is the amount of pheromone on edge $(i, j) \in E$ and η_{ij} is some weight factor for the edge relating to the problem. The probability is calculated by calculating how 'attractive' the edge is, based on the weight, and the pheromone deposited on it. The 'attractiveness' is then normalised (by dividing by the sum across all edges from the vertex) to provide a probability in the interval $[0, 1]$. For example, in the problem that this project tackles (the Travelling Salesman Problem), the weight factor would be inversely proportional to the distance between a pair of cities (the endpoints of the edge). α and β represent user defined weighting factors which decide the influence of the pheromone level and weight of the edge.

Before depositing the pheromone for each ant, the existing pheromone data is first decayed, according to a defined evaporation rate ρ so that each edge's pheromone level τ_{ij} is updated to: $\tau_{ij} \leftarrow (1-\rho)\tau_{ij}$, this allows edges that remain unvisited over a number of iterations to 'forgotten'.

The algorithm then deposits an amount of pheromone for each edge in its solution S^k, so each level is updated by:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

Where the deposited amount $\Delta\tau_{ij}^k$ for each ant k, with solution S^k is defined:

$$\Delta\tau_{ij}^k = \begin{cases} C^k & \text{if } (i,j) \in S^k \\ 0 & \text{otherwise} \end{cases}$$

And where C^k is some function of the solution produced by ant k, relating to the solution's quality. For example in our travelling salesman situation, C^k is inversely proportional to the length of the tour.

Again, this fits well into our class of agent based systems. The solution generation and pheromone depositing by each ant can be considered the function in the initial 'map' procedure, and pheremone evaporation and probability calculation as the 'reduce' procedure.

B Serial and parallel implementation details

B.1 'Boids' flocking algorithm

The parallel and serial implementations of the flocking algorithm were very similar, in many ways differing only in the function signatures of the agents methods. This is due to the fact that CUDA functions must be declared with either `__device__` or `__global__` keywords to signify that they are to be compiled into GPU code. Aside from this detail, the only other major difference is in how the 'reduce' portion of the code is run. In the serial implementation, each agent is rendered using an OpenGL call, which implicitly transfers data to the GPU and renders it. With the parallel implementation, as the agent data already resides on the GPU, a single call to OpenGL is made to render the entire flock. This could be considered an optimisation beyond the scope of adapting the algorithm for GPU execution, however this is a common technique for rendering arrays of vertices using OpenGl, but without a specific call to copy the vertices to the GPU (as they already reside there).

B.2 Genetic algorithm

As with the flocking autonoma the serial and parallel implementations of the genetic algorithm were very similar. Again, as with the flocking algorithm, the main difference was in the 'reduce' stage of the algorithm. In the case of the genetic algorithm this was reflected in the how the ranking of the population of solutions was achieved prior to breeding of a new solution. With the serial implementation this was implemented using an optimised C standard library function `qsort`. For the parallel implementation however, a CUDA specific variant of the 'Bitonic sort' (see (Leighton 1992) for more information) algorithm was developed.

Figure 1 provides a good visualisation of a bitonic sorting network with 16 inputs. Our CUDA implementation of the algorithm first iterates (using a for loop) over possible 'sub-list sizes' of the input, shown using pale yellow and blue boxes in Figure 1. Within that loop, it then iterates over 'swap distances', shown as the lengths of the comparator arrows in Figure 1. Finally, within the loop the implementation then launches $n/2$ comparisons in parallel, using a simple CUDA kernel to perform the swap. The set sub-list sizes, comparison distances and thread index are given as parameters to the kernel to determine which elements to perform the swap with and direction (i.e. whether to sort using the '$<$' or '$>$' comparison) in which to perform the comparison and swap.

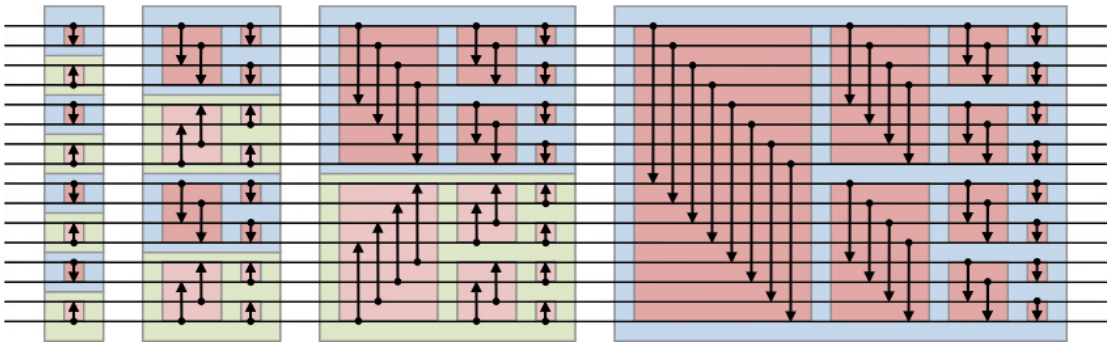


Figure 1: A visual representation of the bitonic sort algorithm (Commons 2011)

B.3 Ant colony optimisation algorithm.

Again, the major differences in implementation between the serial and parallel versions of the ant colony optimisation algorithm resided in the 'reduce' section of the algorithm.

B.4 Technologies used

All the serial and parallel implementations were written using either the C++ language or the CUDA C++ dialect. The source code was compiled to executables using the GCC (for serial code) and NVCC (for parallel code) compilers. GNU Make was used to automate and standardise the compilation of the implementations. While the algorithm implementation for each set of implementations was carried out separately, as set of libraries for common functions (such as timing, or argument gathering) was also developed.

C Generic implementation

To better explain reasons for its failure, discussion of the advanced objective, the generic implementation has been split into two sections. The first discusses the purpose of the objective, and the potential technical challenges associated with it. The second section discusses the implementation strategies attempted and the technical reasons for their failure.

C.1 Overview of the generic framework

To realise the Advanced objective, work was begun on a 'generic' framework for describing and implementing map/reduce agent based systems. Such a framework would need to provide two main services for transforming system descriptions to code:

- Transformation and possible parallelisation of agent based system functions (i.e. the map and reduce functions) to GPU specific code, or CPU specific code
- Automatic generation of memory management code for either GPU or CPU

The reasoning for the first service is clear - any generic system for describing and compiling an agent based system must also be able to generate code for execution. In the case of the generic system, it would need to be able to produce code for both serial implementations and for GPU parallel implementations. The second service is required to support the activities of the first.

As CUDA (before version 5.5) works using a separated memory model (that is, GPU accessible memory is distinct from CPU accessible memory), all the parallel implementations would require completely different memory management code, which it would be the framework's responsibility to generate.

A number of approaches for implementing the generic agent framework were investigated, before it was decided that further work would be beyond the scope of the project. The first two approaches aimed to use tools already built into the main implementation language of the project (CUDA C/C++), while the latter two approaches used custom tools written separately in alternative languages. The inbuilt tool approaches aimed to use the C preprocessor and C++ template functions to automatically generate serial and parallel code. The custom tool approaches attempted to, respectively, implement a custom C++ preprocessor to transform descriptions of agent based systems into C++ code, and implement a new domain specific programming language for describing map/reduce style agent based systems. As discussed later, It was found that the former two techniques were not powerful enough to achieve the aims of the generic purpose. While for the latter two techniques, one was found to be insufficiently powerful by itself and would require enough hinting as to not achieve the aims of the system, and the other was found to be too complex to implement within the scope of the project.

C.2 Approaches attempted

The initial attempt at implementing the generic framework utilised the C preprocessor to take agents described as C structures and generate code for automatic memory management. Preliminary work into this approach was explored, including the creation of a 'generic' list (i.e. one able to be used with any type) using the C preprocessor, however it quickly became apparent that this approach would not be powerful enough for the full requirements of the objective.

Initial work with the preprocessor approach focussed on the second service of the generic framework: providing automatic memory management for the system across either the GPU or CPU. Using the preprocessor initially seemed to promising. It allowed for the definition of generic data structures without the use of `void*` pointers, as well as defining custom memory management methods for such structures. The preprocessor also allowed for limited type introspection with the use of the `typeof` macro.

It was however quickly realised that it would be possible to get gain similar power with the use of C++ templates, but with a number of benefits. Firstly, using C++ template provided a much easier, and more natural, method for defining generic lists. Secondly, it provided a far more intelligent type system. Using template types allowed for memory location information to be embedded within the type itself, so (for example) a list of integers stored on the graphics card could have the type signature `List<int, device>`. Finally, C++ templates also allowed for the overloading of assignment operators, so that the semantics of the operator can change depending on the types involved.

Combining operator overloading with location aware types then allowed the C++ approach to solve one of the difficulties with the pure preprocessor technique: determining when and where to copy memory to/from the GPU. This was possible by defining multiple overloaded assignment operators, one for each possible copy direction, and then use location aware types to have the compiler automatically select the appropriate one.

The C++ template approach, C preprocessor approach and combined approach worked well for simple memory management, e.g. managing a list of integers. Unfortunately however, both

approaches failed when faced with more difficult (and realistic challenges) such as nested memory management, or management of more complex types. This was due to the inability of either approach to automatically enumerate the member variables of a memory managed class or agent, and thus neither approach could automatically generate the memory management code needed. Because of this, it would have been necessary for the programmer to manually implement all the memory management routines, effectively losing the benefits an automated system/framework for implementing agent based systems.

To attempt to solve this, the next stage and method was the implementation of a custom preprocessor on top of the C preprocessor and C++ template metaprogramming. The custom preprocessor was written using Python, using the CppHeaderParser package [4] to automatically parse descriptions of agents written as C++ classes. The preprocessor would then identify the methods and variables, and generate the correct methods, calls and memory management code for the system (depending of course on whether it was to be a serial or parallel implementation).

This solved the majority of the problems with the first pair of approaches, in particular the automated memory management code generation, but it too soon encountered problems. Instead of on the memory management side of the framework, the difficulties now shifted towards the implementation of automatic code generation when parallelising systems described under the framework. This was due to the large range of different 'reduce' functions that would need to be supported, and the difficulty in automatically detecting parallelism within such sections. The difficulty with this approach (as with the template approach and memory management) was that to be useful, it would require a great deal of 'hinting' from the programmer describing the system. This again, would reduce the usefulness of the framework, as the programmer would again be left (essentially) describing exactly how to parallelise the system, similarly to writing raw CUDA code.

The final approach taken was the specification and implementation of a full domain specific language for describing map/reduce agent based systems. This was the most promising approach of all four. By being able to provide a custom description of an agent, as opposed to relying on the syntax and semantics of a more general purpose language, it would theoretically be easier to determine how the system's logic could be parallelised. For example, language constructs such as parallel for statements (e.g. as in the Vector language (Mao 2014)) However, despite initial work beginning on the compiler and specification, it was soon realised that the full implementation of novel language and compiler was likely beyond the scope of the project.

IV RESULTS

This sections presents the raw results of experiments run to examine and compare the parallel and serial implementations. An in depth analysis of the results can be found in the 'Evaluation' section.

A Methodology

Although all the systems implemented within this project fall under the map/reduce agent based system banner, it was important to consider them more closely during the testing phase, as each system had different characteristics and properties that would need to be measured. The

[4] Available at `https://pypi.python.org/pypi/CppHeaderParser/2.4.1`

systems were therefore divided into two categories for testing: optimisation systems (encompassing the genetic and ant colony optimisation algorithms) and graphical systems (consisting solely of the flocking algorithm).

A.1 Optimisation systems

The parallel optimisation algorithms were each compared with their serial counterparts on two metrics: time taken, and tour length. The time taken was measured by each program internally, using timing functions from the POSIX C library `<sys/time.h>`, and measuring at the start and end of the `main` function. This method was chosen instead of operating system timing as it was able to measure to millisecond accuracy, as well as measuring the 'clock' time taken (i.e. the actual time taken, not time spent using the CPU), and thus would not be affected by different CPU usage patterns between parallel and serial implementations. Four conditions were altered for each run of each implementation: the problem instance, the number of agents, the number of iterations. The number of agents were varied over a range of 8 to 4096 agents, iterations over a range of 10 to 10000 and problem instances over a range of 12 to 535 cities. For each combination of conditions, each implementation was run ten times using ten different seed files for random number generation. As well as varying conditions and seeds, each run was performed twice using two different test machines. The first was a DELL XPS15 Laptop with a four core Intel Core i5-2410M processor running at 2.3 GHz, and a NVIDIA GeForce GT540M graphics card. The second was a server with a dual core Intel Core 2 Duo processor running at 2.66 GHz and a NVIDIA Quadro FX 1700 graphics card. For convienence, from here on the first test computer shall be referred to as 'machine 1' and the second as 'machine 2'.

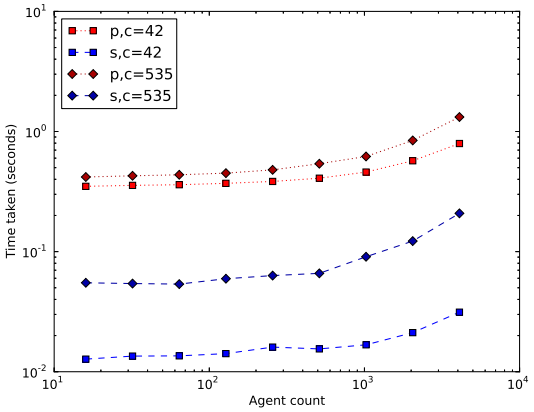
A.2 Graphical systems

The graphical system, Reynolds flocking algorithm, was measured using two related metrics: the time taken to render 1000 frames, and derived from that, the average number of frames rendered per second. This was measured by allowing the algorithm to run for 1000 frames, to allow the frame rate to stabilise, then measuring the time taken to produce a further 1000 frames. This was repeated over a range of agents between 256 and 8192. To ensure fair testing, the display area was kept at a constant size of 640 pixels wide by 480 pixels high. All tests of the graphical system were performed using the computer named 'machine 2' as described above.

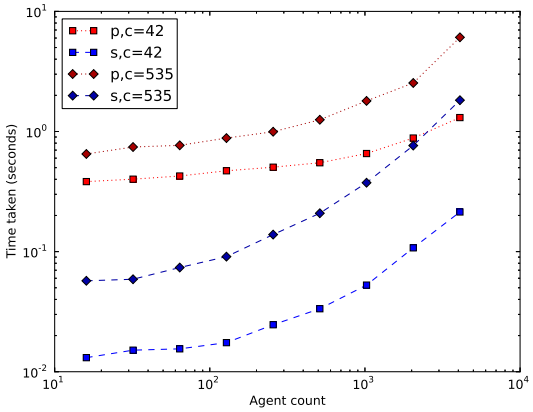
B Experimentatal Results

B.1 Optimisation systems

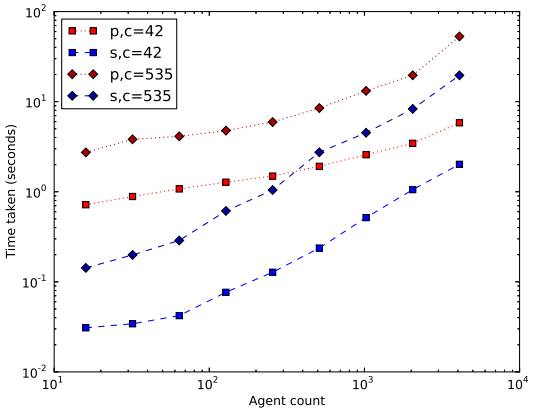
Genetic algorithms Figure 2 shows the average results across all runs of the genetic algorithms, across both test computers. The average times taken for two different instances is shown, with 42 and 535 cities respectively. The input sizes chosen for the genetic algorithm are over an exponential range (in that each input is either double (for the agent count) or ten times (for the iteration count) the size of the previous input). To allow this to be more easily analysed, all axes in figure 2 operate on a logarithmic scale.



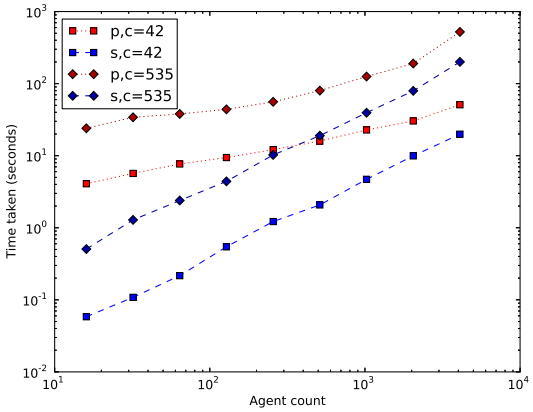
(a) 10 iterations



(b) 100 iterations



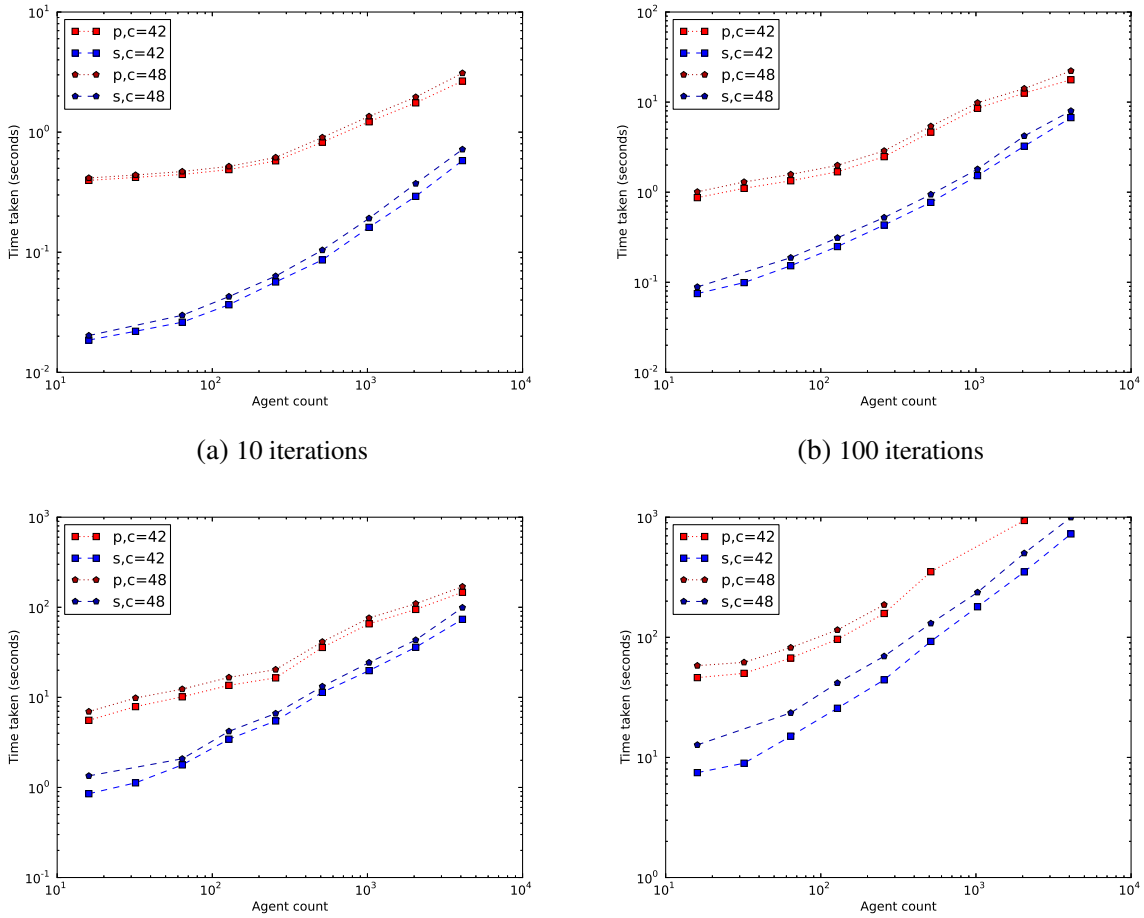
(c) 1000 iterations



(d) 10000 iterations

Figure 2: Comparison of genetic algorithm implementations

Ant colony optimisation algorithm Gathering and analysing results for the ant colony algorithm was more difficult than for the Genetic algorithm. This was due to the time contraints of the project, as there was insufficent time between the finalisation of the system's code, and analysis of results, to run both implementatons for all the possible parameters. However, the majority of parameters have had results gathered for them. As with the genetic algorithm, figure 3 shows the results for two different problem instances, with 42 and 48 cities respectively. For the same reason as the genetic algorithm, figure 3 uses logaritmic axes throughout.



(a) 10 iterations

(b) 100 iterations

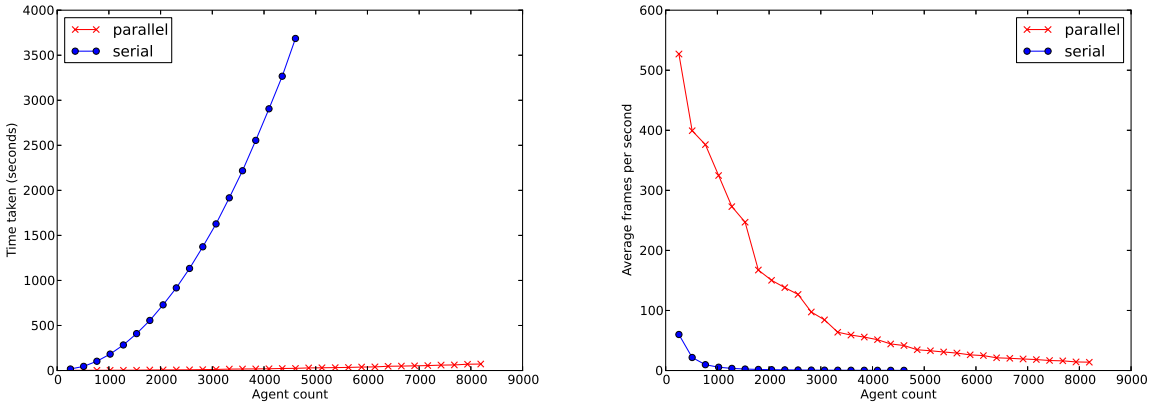
(c) 1000 iterations

(d) 10000 iterations

Figure 3: Comparison of ant colony optimisation algorithm implementations

B.2 Graphical systems

The results for the tests on the graphical system can be found in Figure 4. Figure 4a shows the relationship between number of agents in the simulation, and the time taken to run the simulation, while 4b shows the relationship between agent count, and the average number of frames rendered per second, across 1000 frames. As with the ant colony algorithm, some results were impossible to gather within the time constraints of the project, however clear trends are visible in the successfuly gathered results. Unlike the presentation of the optimisation system results, figure 4 uses linear axes, again to reflect the inputs.



(a) Time to run 1000 iterations

(b) Average frames per second

Figure 4: Comparison of flocking implementations across a run of 1000 iterations

V EVALUATION

A Analysis of results

Analysis of the optimisation systems, and graphical systems present two seemingly conflicting results. To examine why, it is best to discuss each class of algorithms individually before discussion any overall trends.

A.1 Optimisation algorithms

Initial analysis of the optimisation algorithms showed a far more negative picture than expected. The parallel implementaitons of both algorithms were significantly, and consistently slower than their serial counterparts. However, analysing and comparing the overall trends of each implementation, it appears that despite starting far slower, the runtime of the parallel implementations grows at a far slower rate than their serial counterparts.

The initial poor performance of the parallel implementations is likely due to a number of factors. First, and most frustratingly, is the issue of driver and memory management overheads. With the serial implementations, all code is run 'bare', that is, relying on the minimum of other libraries and tools, with the exception of core operating system code (for example for file IO or memory management). For the parallel implementations however, as well as relying on operating system code, the program also needed to use a large amount of driver code to communicate with the GPU, and spend time copying problem data to the GPU and results data back. As an example of where this may be a problem, consider the (common) pattern of applying a function to a list of items. With a serial implementation, this comprises a for loop over the list, along with a function call for each item. For the parallel implementation, it requires launching a copy of kernel function (in parallel) for each item. Comparing the two implementations, the serial code is fairly simple, and involves (likely) no more than a number of `jmp` and `call` statements to evaluate the loop, and call the function (depending of course on the processor architecture compiled for). With the parallel implementation however, evaluating the kernal launch involves the invocation of a number of driver functions to copy memory, and send instructions to the

GPU. Given that utilising driver code typically involves executing a substantial amount of code (for small vairables, potentially more than the entirety of the serial code!) this introduces a large, but constant, overhead. For small problem instances, agent counts and iterations, this negates any speedups potentially gained by the parallelisation, however as the varibles grow larger, this constant overhead becomes less of a factor in determining the time for the implementation to run.

A.2 Graphical systems

In stark contrast to the optimisation systems, the parallel implementation of the graphical system (Reynold's flocking algorithm) performed far better than its serial counterpart, both in terms of raw results, and asymptotically. In the case of the graphical systems, it is likely that driver overheads actually worked in its favour, as both implementations were required to use similar driver calls for common tasks such as rendering. As the majority of time taken for a rendering operation is in simply copying data to the GPU, such calls would have been significantly faster for the parallelised implementation, as all agent data already resided on the GPU.

In addition to driver based speedups, the flocking algorithm is more complex than either of the other algorithms when measured asymptotically in terms of the number of agents. Our nave implementation of the flocking algorith, is roughly $O(n^2)$, where n is the number of agents. For a constant number of cities c, and ignoring the number of iterations, our implementation of the genetic algorithm is roughly $O(cn)$, and ant colony, $O(c^2n)$. Because of this, after parallelisation in terms of the agents (i.e. $O(n^2)$ becomes $O(n)$), constant factors (such as driver overheads) are far more of an issue in the optimisation algorithms, while asymptotically mattering less with the flocking algorithm.

B Analysis of project as a whole

As well as examining the successes and results of the project, it is also important (and possibly more useful) to examine its limitations, and reflect on ways they could be improved and circumvented. Within the scope of the deliverables, the limitations are best examined as two seperate areas: implementation limitations, and testing limitations.

B.1 Testing limitations

Despite producing clear results, the testing methodology of the project has a number of areas in which it could be improved. The lack of results for some combinations of algorithms and inputs, as well as the inability to establish trends in the data with a large amount of certainty point to two main possible areas of improvement. Firstly, the range of values that variables took would have benefited from extension, to allow greater ranges of tests to be run, and results analysed over a greater range. A prime candidate for possible extension would be the number of iterations, which for the tests listed above only took on four different values: 10,100,1000 and 10000. Extending the number of iterations further would hopefully allow the difference asymptotic complexity of the two implementations to be made clearer, and ideally negate the impact of the fixed overheads (e.g. drivers, memory management) of the parallel implementations. Secondly, and paradoxically, the range of problem sizes for the optimisation algorithms would benefit from streamlining, and limiting. For this project, results were gathered over a large range of problem sizes, as well as a range of agent numbers, and iteration counts. This made analysing the results

more difficult, as controlling for multiple variables and visualising the results was difficult by the range of problem instances, and the range of difficulty of each instance. An alternate solution may be to have a single fixed size of instance, and retain the current system of running each test with a different seed for the random number generators. This would allow the results of each test to be compared more simply, and would allow for a far greater variation in other variables while still conducting the same number of tests.

B.2 Implementation limitations

Although good results were gathered from the 'hand coded' serial and parallel system implementations, a clear limitation of the project was its inability to control for any possible programmer introduced biases. This was solely due to the failure to implement the proposed generic system for automatically generating parallel and serial implementations from high level descriptions of agent based systems. Although results gathered from experimentation with the 'hand coded' implementations did appear to suggest no functional difference between the two, it would have been useful to verify that the implementations behaved identically. In addition to improving the validity of results gathered, a high level system for parallelising agent based systems would serve as a good basis for futher investigation into parallelism of agent based system. However this is not particularly within the scope of the project, but would serve as a useful basis for future work.

C Organisation of the project

Outside of the failure to complete the advanced objective, there are two main areas within the organisation and execution of the project which could have been improved. Firstly, the timing of the project should have allowed more time for testing and results gathering for all implementations. As discussed above, the range of tests run was in many ways too limited to easily draw firm conclusions from the resulting data. Allowing a longer period of testing would allow for a wider range of tests, and therefore more concrete results.

VI CONCLUSIONS

A Main findings

The results of the comparisons of serial and parallel implementations for the optimisation algorithms generally showed that as problem instances, agent counts, and iteration counts grew, GPU parallel implementations of map/reduce agent based systems tended to perform better. This result was however tempered by the finding that for smaller inputs and simpler algorithms the GPU specific overheads tended to negate any improvement gained by the parallelism. For the more complex graphical systems however, GPU parallelism massively benefited the system, as overheads such as host/device memory copying worked against the serial implementation instead.

Despite the GPU overheads however, this shows that map/reduce agent based systems are a potential target for more investigation into parallelisation, and especially automated parallelisation. Given that 'dumb' GPU implementations of the systems (i.e. ones that did not try and account for any GPU specific optimisations) performed asymptotically better than equivilant parallel systems, the results suggest that it is likely that any parallelisation, even without particularly intelligent parallelisation, would still produce similarly good results.

B Possible extensions to the project

Although the main purpose of the project was achieved, the difficulty in implementing the advanced requirement, and its usefulness for verifying the results, highlights it as a possible area in which the project could be extended. This would likely take the form of a library or language for describing agent based systems using a high level notation. From this, the library/language would then be extended to allow for code generation for parallel implementations.

An alternative extension, spurred by the success of this projects implementation, would be to investigate potential further speedups to the flocking system. Despite the speed increase, the parallel system used no GPU optimised data structures or methods, so it would be interesting to see what further speedup could be acheived using such optimisations.

References

Cecilia, J. M., Garcia, J. M., Ujaldon, M., Nisbet, A. & Amos, M. (2011), Parallelization strategies for ant colony optimisation on gpus, *in* 'Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum', IPDPSW '11, IEEE Computer Society, Washington, DC, USA, pp. 339–346.
URL: *http://dx.doi.org/10.1109/IPDPS.2011.170*

Cekmez, U., Ozsiginan, M. & Sahingoz, O. (2013), Adapting the ga approach to solve traveling salesman problems on cuda architecture, *in* 'Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium on', pp. 423–428.

Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L. & Grover, V. (2011), Accelerating haskell array codes with multicore gpus, *in* 'Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming', DAMP '11, ACM, New York, NY, USA, pp. 3–14.
URL: *http://doi.acm.org/10.1145/1926354.1926358*

Chiara, R. D., Erra, U., Scarano, V. & Tatafiore, M. (2004), Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance., *in* B. Girod, M. A. Magnor & H.-P. Seidel, eds, 'VMV', Aka GmbH, pp. 233–240.
URL: *http://dblp.uni-trier.de/db/conf/vmv/vmv2004.html#ChiaraEST04*

Commons, W. (2011), 'A bitonic sorting network with arrows showing the direction of the comparators.'. File:BitonicSort1.svg.
URL: *https://commons.wikimedia.org/wiki/File:BitonicSort1.svg*

Dean, J. & Ghemawat, S. (2004), Mapreduce: Simplified data processing on large clusters, *in* 'Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6', OSDI'04, USENIX Association, Berkeley, CA, USA, pp. 10–10.
URL: *http://dl.acm.org/citation.cfm?id=1251254.1251264*

Delvacq, A., Delisle, P., Gravel, M. & Krajecki, M. (2013), 'Parallel ant colony optimization on graphics processing units', *Journal of Parallel and Distributed Computing* **73**(1), 52 – 61. Metaheuristics on {GPUs}.
URL: *http://www.sciencedirect.com/science/article/pii/S0743731512000044*

Doniec, A., Mandiau, R., Piechowiak, S. & Espi, S. (2008), 'A behavioral multi-agent model for road traffic simulation', *Engineering Applications of Artificial Intelligence* **21**(8), 1443 – 1454.
URL: *http://www.sciencedirect.com/science/article/pii/S0952197608000456*

Dorigo, M., Maniezzo, V. & Colorni, A. (1996), 'Ant system: optimization by a colony of cooperating agents', *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on* **26**(1), 29–41.

Flynn, M. (1972), 'Some computer organizations and their effectiveness', *Computers, IEEE Transactions on* **C-21**(9), 948–960.

He, B., Fang, W., Luo, Q., Govindaraju, N. K. & Wang, T. (2008), Mars: A mapreduce framework on graphics processors, *in* 'Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques', PACT '08, ACM, New York, NY, USA, pp. 260–269.
URL: *http://doi.acm.org/10.1145/1454115.1454152*

Huang, D.-W. & Lin, J. (2010), Scaling populations of a genetic algorithm for job shop scheduling problems using mapreduce, *in* 'Butt Computing Technology and Science (ButtCom), 2010 IEEE Second International Conference on', pp. 780–785.

Husselmann, A. & Hawick, K. (2011), Simulating species interactions and complex emergence in multiple flocks of boids with gpus, *in* 'Proc. IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011)', pp. 100–107.

Jin, C., Vecchiola, C. & Buyya, R. (2008), Mrpga: An extension of mapreduce for parallelizing genetic algorithms, *in* 'eScience, 2008. eScience '08. IEEE Fourth International Conference on', pp. 214–221.

Johar, F., Azmin, F., Suaidi, M., Shibghatullah, A., Ahmad, B., Salleh, S., Aziz, M. & Md Shukor, M. (2013), A review of genetic algorithms and parallel genetic algorithms on graphics processing unit (gpu), *in* 'Control System, Computing and Engineering (ICCSCE), 2013 IEEE International Conference on', pp. 264–269.

Leighton, F. T. (1992), *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes*, Morgan Kaufmann Publishers, San Mateo, California, chapter 3.5, Sorting.

Mao, H. (2014), 'The vector programming language.'.
URL: *http://zhehaomao.com/project/2013/12/20/vector.html*

Palmer, R., Arthur, W. B., Holland, J. H., LeBaron, B. & Tayler, P. (1994), 'Artificial economic life: a simple model of a stockmarket', *Physica D: Nonlinear Phenomena* **75**(13), 264 – 274.
URL: *http://www.sciencedirect.com/science/article/pii/0167278994902879*

Reynolds, C. W. (1987), 'Flocks, herds and schools: A distributed behavioral model', *SIGGRAPH Comput. Graph.* **21**(4), 25–34.
URL: *http://doi.acm.org/10.1145/37402.37406*

Sethia, P. & Karlapalem, K. (2011), 'A multi-agent simulation framework on small hadoop cluster', *Engineering Applications of Artificial Intelligence* **24**(7), 1120 – 1127. Infrastructures and Tools for Multiagent Systems.
URL: *http://www.sciencedirect.com/science/article/pii/S0952197611001096*

Sttzle, T. (1998), Parallelization strategies for ant colony optimization, *in* 'Proceedings of PPSN-V, Fifth International Conference on Parallel Problem Solving from Nature', Springer-Verlag, pp. 722–731.

Verma, A., Llora, X., Goldberg, D. & Campbell, R. (2009), Scaling genetic algorithms using mapreduce, *in* 'Intelligent Systems Design and Applications, 2009. ISDA '09. Ninth International Conference on', pp. 13–18.

Wang, G., Salles, M. V., Sowell, B., Wang, X., Cao, T., Demers, A., Gehrke, J. & White, W. (2010), 'Behavioral simulations in mapreduce', *Proc. VLDB Endow.* **3**(1-2), 952–963.
URL: *http://dl.acm.org/citation.cfm?id=1920841.1920962*

Wu, B., Wu, G. & Yang, M. (2012), A mapreduce based ant colony optimization approach to combinatorial optimization problems, *in* 'Natural Computation (ICNC), 2012 Eighth International Conference on', pp. 728–732.

Zhou, C. (2010), Fast parallelization of differential evolution algorithm using mapreduce, *in* 'Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation', GECCO '10, ACM, New York, NY, USA, pp. 1113–1114.
URL: *http://doi.acm.org/10.1145/1830483.1830689*