Models and Tools for Parallelism

Artificial Intelligence Programming

|  |  |  |
| --- | --- | --- |
|  | |  | | --- | | **Synopsis**  This report is a combination of two mini projects: a paper on parallel computing, and an assignment on artificial intelligence.  The first part of the report covers the basics of parallel computing, where the second part focuses on artificial intelligence, namely genetic algorithms.  The artificial intelligence portion of the report includes a software program that demonstrates the use of parallel processing. | |

Esbjerg Institute of Technology

Group: F6s-1-F11

Date: 27-05-2011

Supervisor: Dr. Akbar Hussain

Group Members: Adam Tulinius and Mikkel Madsen

# Table of Contents

[Table of Contents 2](#_Toc294523515)

[Preface 3](#_Toc294523516)

[Parallel Computing 3](#_Toc294523517)

[Levels of Parallelism 3](#_Toc294523518)

[Parallel Processing Methods 3](#_Toc294523519)

[Pipelining 3](#_Toc294523520)

[CPU Scheduling 4](#_Toc294523521)

[Multiple Core’s/CPU’s 4](#_Toc294523522)

[Distributed Computing 4](#_Toc294523523)

[Load Balancing 5](#_Toc294523524)

[Memory Architectures 5](#_Toc294523525)

[Shared Memory 5](#_Toc294523526)

[Distributed Memory 6](#_Toc294523527)

[Artificial Intelligence Programming 7](#_Toc294523528)

[Problem Description 7](#_Toc294523529)

[Program Documentation 7](#_Toc294523530)

[Server 8](#_Toc294523531)

[Client 9](#_Toc294523532)

[Clever Agent 10](#_Toc294523533)

[Ultimate DNA 10](#_Toc294523534)

[Testing 11](#_Toc294523535)

[Test procedure 11](#_Toc294523536)

[Test Results 11](#_Toc294523537)

[Conclusion 13](#_Toc294523538)

[Difficulty of Guessing a Word 13](#_Toc294523539)

[Artificial Intelligence 14](#_Toc294523540)

[Parallelization 14](#_Toc294523541)

[References 15](#_Toc294523542)

# Preface

The source code, test results and a digital copy of the report can be downloaded from [goo.gl/Iuvu6](http://goo.gl/Iuvu6)

The .zip file contains a readme.txt file with instructions for running the applications, e.g., required python libraries.

# Parallel Computing

*Sources: [1], [2]*

## Levels of Parallelism

In order for an application to utilize parallel processing, it must be possible to split the application into subtasks that can run in parallel. Applications can be classified according to how often subtasks must communicate or synchronize with each other. An application where the subtasks are very dependant on each other and are constantly communicating, is said to exhibit *fine-grained parallelism*. An application exhibits *coarse-grained parallelism* if the subtasks have a moderate amount of communication, and is *embarrassingly parallel* if the subtasks rarely have to communicate. Fine-grained applications are much harder to parallelize than coarse-grained or embarrassingly parallel applications.

## Parallel Processing Methods

### Pipelining

Pipelining is instruction-level parallelism. Modern processors have multi-stage instruction pipelines, each stage used for a different step of executing an instruction. A single instruction passes through all the stages in the pipeline, e.g.:

1. Fetch instruction
2. Decode instruction
3. Execute
4. Memory access
5. Register write back

Each processor clock executes one operation in each stage, so the processor can be working on 5 different instructions at the same time. The only restriction is that an instruction that uses data from a previous computation must wait to enter the pipeline until the data is ready. The result is that the first instruction is complete after 5 clock cycles, and another instruction will complete every subsequent clock cycle, assuming there are no data dependencies. In a best case scenario where data dependencies do not occur in instructions in close proximity to each other, pipelining decreases the number of clock cycles required to complete a set of instructions by a factor corresponding to the number of stages in the processor, minus the clocks needed for the first and last instructions where several stages of the pipeline must be idle.

### CPU Scheduling

While not technically parallel processing, CPU scheduling can be used to share a processor between several processes. Even though there is only a single process being executed at any time (for a single-core CPU), to the user things will seem to be happening concurrently, e.g., listening to music while writing an e-mail. The music does not jitter every time the user pushes a key, so as far as the user is concerned the PC is doing two things concurrently. CPU scheduling is vital in computing, since even multi-core CPU’s will almost always need to service many more processes than they have cores.

### Multiple Core’s/CPU’s

True parallel processing requires more than one processor, either in the form of multiple physical processors, a processor with multiple cores, or multiple computers communicating over a network. A parallelized application will run much more efficiently on a multicore processor than a singlecore processor. On a singlecore processor the scheduler will switch the CPU between the application subtasks. A sequential program may run faster on a singlecore CPU than a parallelized version of the same program on the same CPU, since there is overhead involved in facilitating the creation of subtasks and communication between them.

#### Concurrency

The subtasks (threads) of an application share the same memory address space. Subtasks can have their own section of the address space, but for sharing data between subtasks, several threads will have to access the same memory. If a thread attempts to change a data structure in the memory, and the change requires more than one instruction, a second thread may access the same data structure while it is being changed (the possibility of this happening is called a *race hazard*), leading to undesired behavior in the application. To prevent this, the application can use mutual exclusion to prevent multiple threads from simultaneously accessing the same data structure, other threads will then have to wait their turn before accessing memory that is currently being modified by another thread. Simple read access can be facilitated for multiple threads simultaneously, since no data is being altered.

### Distributed Computing

Distributed computing, sometimes referred to as *horizontal scaling*, is when a workload is distributed among several machines.

#### Cluster Computing

When many otherwise ordinary computers are linked together to do computations in tandem, they are referred to as a cluster. The link is often done using fast local area networking, in contrast to traditional supercomputers where the computational units of the supercomputer are directly connected through a high-speed bus.

With a cluster it is possible to vastly increase computational power, while still making it programmable without having to manually load a program of every node of the cluster. They are often used for simulations, like weather foresights.

Clusters can be distinguished by the amount of communication between the nodes;

clusters requiring frequent communication between nodes are usually referred to as *Beowulf Clusters*, while clusters with very little inter-node communication are referred to as *Grid Computing.*

High-availability clusters contain redundant nodes, so that any single node can fail without decreasing availability. Clusters can also be used for load-balancing, where many nodes run the same software, making it possible to hand out computation jobs to any available node in the cluster.

#### Grid Computing

Grid computing networks often utilize unused computational resources, also known as *CPU scavenging*. This concept is known from projects such as *SETI@home*, where volunteers donate their unused CPU-time for searching for extraterrestrial lifeforms, and *Folding@home*, where volunteers donate CPU cycles for studying how protein folding works.

Networks like these have limited or no inter-node communication; computational tasks are sent from some sort of controller to each node, which then completes the computation and returns the result.

## Load Balancing

A common difficulty with load-balancing schemes is sharing data between clusters.

If data is stored on database-servers, a method known as ‘sharding’ can be utilized. Basically sharding splits data between servers, which also results in a way to scale available disc space.

Sharding can both be done manually, where you might have data from users stored on servers close to where the client originates, but automatic sharding is also possible, where the database cluster distributes data on nodes by itself. MongoDB, among others, support this.

Another difficulty is that it is easy to replicate a database-server so that read-only queries can be split across multiple servers, but writing to the cluster usually has to be done in a master-slave way, where data is written to the master which then replicates the data to the slaves.

Another method is what Google refers to as “BigTable”. BigTable is based on document-storage, where blobs of data are associated with a unique identifier. This allows the database data to be sharded across multiple machines.

## Memory Architectures

### Shared Memory

Shared memory is memory that can be simultaneously accessed by multiple CPU’s on the same machine. The CPU’s share the same memory address space, similarly to the way the threads of a process share address space. The CPU-to-memory bus can present a bottleneck for shared memory computers, posing a limit on the number of processors that are feasible to have in the machine. Shared memory computers rarely have more than ten processors for this reason. Another complication with shared memory computers is cache coherence; whenever the cache of one processor is updated with information that might be used by other processors, the change must be mirrored to all the other processors to ensure they are not working on incoherent data.

### Distributed Memory

In a multiple-processor computer utilizing distributed memory, every processor has its own private memory. Processors can only do computations on data in their own memory, and if remote data is needed the processor must communicate with the processors that have the needed data in their private memory. The processors in a distributed memory system can be in separate machines, with communication between machines done via a network protocol. Distributed memory computers do not suffer from the race hazards that shared memory computers face, since no memory can be accessed by more than one processor. A distributed memory system is ideal for applications that do not require too much communication between processes, i.e. coarse-grained or embarrassingly parallel algorithms.

# Artificial Intelligence Programming

*Source: [3]*

The below words are used several times throughout the following sections, and their meaning in this context is different than their usual meaning.

**Dictionary**: A text-file containing 58.112 words used for picking words for hangman simulations.

**DNA**: A string containing all letters of the alphabet (a-z) in some order; the DNA represents the order in which the letters will be guessed in a game of hangman.

## Problem Description

A secret word is picked from the dictionary, and it is the objective of the player to guess the secret word. Guesses are made by guessing one letter at the time. If the letter guessed is in the word, it’s position(s) will be revealed. E.g., if the secret word is “intelligence” the player will first see “\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_”. If the player guesses “e” he will now see “\_ \_ \_ e \_ \_ \_ \_e \_ \_ e”.

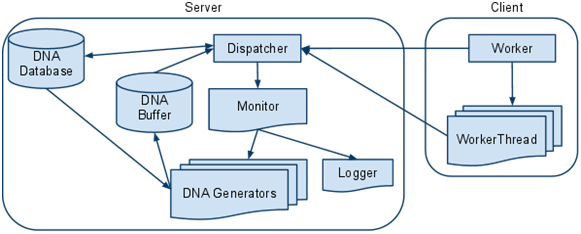
That way the player guesses one letter at a time, until the word is complete. However, when the player has guessed incorrectly 10 times, the game is lost.

The question is now: How good a player can we program?

We will be pursuing several options and compare the results of each. One of the options is based on the concept of “Genetic Algorithms”. Genetic algorithms work much in the same way as evolution. Algorithms are scored based on some criteria, and are then “mutated” by mixing several algorithms, to form new algorithms. By mainly mixing the algorithms that achieve a high score, the result should be that the algorithms evolve, gradually scoring higher and higher. Rather than actual algorithms, we are simply mutating the sequences of letters that indicate what order to guess letters in hangman.

## Program Documentation

The program is split into two separate applications, a server (*dispatcher*) and a client (*worker*). In short, the server tracks the results from simulations on different DNA’s and generates new DNA’s, while the client does the simulations needed to determine the win percentage of each DNA. The simulations require a lot of computing power, but splitting the application into server/client parts allows us to run the client application on multiple machines, forming a small-scale *grid computer*. The figure on the following page illustrates the different parts of the application.



Aside from the server and client portions of the application, we programmed a *Clever Agent* that will guess hangman words not based on a pre-defined string of letters, but with a more advanced method. We also take a guess at what we feel might be the *Ultimate DNA*. The results we get from our *Clever Agent* and *Ultimate DNA* will then be compared to the results of the “best and brightest” among the genetic mutation DNA’s.

### Server

Two interesting things happen in the server:

1. DNA is generated
2. DNA is sent to the client

The server tries to keep at least 100 new DNA strings ready to send out. This is done by spawning two threads that looks at the length of the *job\_buffer*. If the length is less than the one specified in *job\_buffer\_size* it will generate jobs until the buffer is full. For each job generated, the *job\_buffer\_lock* is acquired before writing to the *job\_buffer*. Each job is also added to the *results* dictionary (a Python dictionary is somewhat like a Java HashMap), with a score of *None*.

Job-generation is based on the following rules:

* The ratio between random DNA strings and mated DNA strings should be 1:10, so that new random DNA is added to the gene-pool frequently.
* No DNA strings will be mated before at least 10 results has been recieved.

When generating a random string, the string *abcdefghijklmnopqrstuvwxyz* is simply shuffled, and when mating two strings (*A* & *B),* a letter is taken from the beginning of each string in turn, starting with *A*. If the next letter is already in the new DNA string, it is skipped. The two DNA sequences to mate are chosen from the set of results, such that it is more likely to mate DNA with good results, than bad.

Example: Mating “abc” and “cba” is done as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **a**bc | + | cba | = | **a** |
| bc | + | **c**ba | = | **c** |
| **b**c | + | ba | = | **b** |
| c | + | **b**a | = | nothing, since **b** is already used |
| **c** | + | a | = | nothing, since **c** is already used |
| \_ | + | **a** | = | nothing, since **a** is already used |

Resulting in the new DNA sequence “acb”.

#### DNA dispatching

The server is implemented as a WSGI service running on Tornado.

When clients access the path “/jobs/*job\_count*”, where *job\_count* is an integer, *job\_count* jobs will be sent to the client. This is done by acquiring the *job\_buffer\_lock* and popping one element from *job\_buffer* for each job to send out. If the lock was only acquired one time, a deadlock would happen if the client requested more jobs than were available. By acquiring the lock for every job to dispatch, the buffer fillers will have time to fill the buffer if necessary.

The result of the calculation is then sent to the server by opening “/dna/*dna\_string*/*guesses*/*wrongs*/*winrate*”, which then makes the server fetch *dna\_string* from *results*, and add the test data to it. *Guesses*, *wrongs* and *winrate* must be floats with a decimal point.

#### Other

Info about number of mated and random DNA strings and number of results can be seen on “/info”, while “/” shows the (probably) very long list of results sorted by win rate per default, optionally by number of total guesses or number of wrong guesses. “/repetitions” returns the number of random test each DNA string should be subjected to, and “/dictionary” returns the dictionary containing all possible words.

### Client

The client program has two major components, the main class *Worker*, and the class used for calculations, *WorkerThread*.

#### Worker

* Checks number of CPU cores, e.g. finds 4 CPU cores on the machine.
* Fetches the dictionary and 5 (CPU cores + 1) DNA strings from the server.
* Initializes 5 *WorkerThreads* (CPU cores + 1), each with it’s own DNA string.
* Monitors the number of active *WorkerThreads* 10 times a second, spawning new threads to fill the place of terminated threads (fetching new DNA from *Dispatcher* to do so).

The number of *WorkerThreads* is kept at the number of CPU cores plus one, decreasing idle time between one thread terminating and *Worker* spawning a new one and decreasing wasted CPU cycles while requesting new jobs from the server. Increasing the number of threads can decrease the overall speed of the client node, since the overhead involved with switching between threads will consume clock cycles that could have been used for other computations, so we have chosen to only use 1 extra thread.

#### WorkerThread

* Picks a random secret word from the dictionary.
* Simulates a game of hangman, with the order of the letters being guessed defined by the DNA.
* Checks if the letter is in the hangman word.
* Uses the dictionary to exclude words when they can be ruled out, i.e. if they are not the correct length, don’t contain a correctly guessed character at the correct position(s), or contains a letter that was not in the secret word.
* If all but one word has been excluded, or if all characters in the word have been guessed, the game is over.
* The total number of guesses and the number of incorrect guesses is stored before starting the next test, as well as noting if the DNA “won” the game, by guessing the secret word with less than 10 incorrect guesses.
* Once the test has been repeated the required number of times, the average number of incorrect guesses, average number of total guesses and the win percentage are sent to the *dispatcher* via HTTP.
* Thread terminates, making room for a new thread to be initialized by *Worker*

### Clever Agent

This agent is modeled after having the lowest risk of a wrong guess.

The agent keeps a list of all the possible words, and a list of the guesses made so far.

To begin with, all words with different length than the secret word are removed from the list of possible words.

For each guess it is calculated how many words contain each remaining letter, and the letter contained in most words will be chosen as the next guess. If the guess is correct, all words not containing the guessed letter will be removed, and if the guess was wrong, all the words containing the guessed letter will be removed.

This process is repeated until the word is guessed, or the maximum number of tries has been reached.

Source code for the Clever Agent is found in the clever\_agent.py file.

### Ultimate DNA

The “Ultimate DNA” is what we imagine would be a near optimal order to guess the letters in the alphabet, in the event that this order must be defined before the game begins. This DNA is found by counting how many words in the dictionary each letter occurs in, and then listing the letters in order from most common to least common. In contrast to the way the Clever Agent works, this DNA can be tested in the exact same way as the other DNA strings. The “Ultimate DNA” isn’t necessarily the absolute best order to guess the letters in, but we are expecting it to be very efficient and it serves as a great comparison for the DNA strings generated with mutations. It is entirely possible for this “Ultimate DNA” (or maybe even a better one) to be generated by the server node, but due to the number of possible strings, it is unlikely that this exact string would be generated within a reasonable time with the limited computation power available for our tests.

The “Ultimate DNA” was found using the ultimate\_dna.py script.

## Testing

For ultimate and genetic DNA tests, all tests were done twice, once with “skip” enabled, and once without “skip”. With “skip” enabled, the DNA would not guess a letter that it knows can‘t be in the word; this is done by examining the known words where any non-possible words are excluded, and skipping the next letter in the DNA string if the letter does not exist in any of the words. The genetic and ultimate DNA scores with skip enabled give a much higher success rate, since this method is more intelligent.

### Test procedure

#### Clever Agent

The clever agent was run against the entire dictionary (58,112 words) and a win-rate was found.

#### Genetic DNA

The server/client applications were run for more than 80.000 iterations of DNA, where each DNA was tested against 1000 random words from the dictionary (to save time: the tests take quite a lot of time to run).

This procedure was repeated a second time without “skip” enabled.

The top 4 of each type of genetic DNA (skip and non-skip) was then run against the entire dictionary to get results comparable to that of the clever agent.

**Ultimate DNA**

The “Ultimate DNA” - *eisarntolcdugpmhbyfvwkxqjz* - was tested against all the words in the dictionary, first with “skip” enabled, then without “skip”.

### Test Results

#### With Skip

|  |  |  |
| --- | --- | --- |
| **DNA (Tested WITH skips)** | **Win % (1000 tests)** | **Win % (All words)** |
| nrdieatlogscbuwzkmxyphvfqj | 91.6 | 87.7 |
| neditrasclobhgpwuzkmyfxvjq | 91.2 | **89.9** |
| tmnasiledourgwhpfqczbjxvyk | 90.9 | 87.9 |
| entgsliradchowyuxvmbzpqfkj | 90.4 | 87.6 |
| *Ultimate DNA* | - | *88.6* |

It is interesting to note that the best DNA from genetic mutations performs better than the Ultimate DNA, in the tests where the “skip” feature was used.

#### Without Skip

|  |  |  |
| --- | --- | --- |
| **DNA (Tested WITHOUT skips)** | **Win % (1000 tests)** | **Win % (All words)** |
| dtinlrsuavecopbgjyzmxhwkfq | 68.1 | 66.6 |
| tdloerishmnuygacfwbzvjpkqx | 68.0 | 62.8 |
| mdytiglesorunacvpbhkzfjqwx | 67.6 | 67.0 |
| tyislmeondcaurgbjzhvwkpfxq | 67.5 | 64.6 |
| *Ultimate DNA* | - | ***69.9*** |

In this test the Ultimate DNA performs better than the best genetic DNA, however the genetic DNA could easily be surpassed by another DNA if the test had been allowed to run for a longer time, or if the top 20 genetic DNA’s had been tested against the entire dictionary, while the Ultimate DNA result can not improve. Furthermore, it is likely that the tests without skip would have gotten better if the mating method made it easier to place “bad” letters in the end of the DNA string.

#### Clever Agent

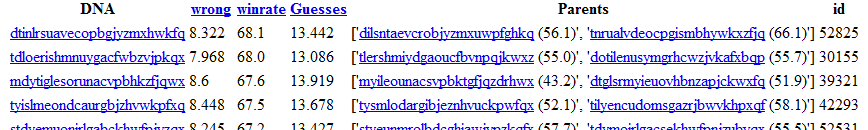
Finally, the Clever Agent managed to win **98.3%** when tested on the entire dictionary. We expected this agent to perform better than any of the static DNA’s, so the result is not surprising.

#### Summary

The combined test results are shown below, ordered from highest to lowest win percentage.

|  |  |
| --- | --- |
| **Agent** | **Win % (All words)** |
| Clever Agent | 98.3 |
| Best Genetic DNA (With skip) | 89.9 |
| Ultimate DNA (With skip) | 88.6 |
| Ultimate DNA (No skip) | 69.9 |
| Best Genetic DNA (No skip) | 67.0 |

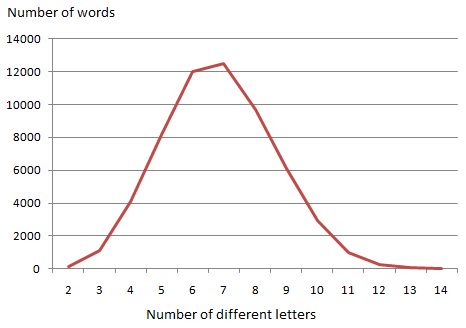
The generated genetic DNA’s and their scores can be seen by opening the skip.html and no\_skip.html files in the accompanying material (they should look like the figure below). The scoreboards clearly show how randomly generated DNA scores are much lower than for DNA strings that have been evolved over several generations. Random DNA strings can be recognized as those on the list with no parents. The “id” number to the far right in each entry is indicates how many DNA’s were generated before that DNA. For example, in the figure below the top 4 DNA’s all have id’s over 30.000. It is worth noting that higher ID’s doesn’t necessarily equal better scores.



# Conclusion

It is clearly an advantage to be able to skip letters not in the possible remaining words, which can be seen by the skipping agents winning about 75% more games than the non-skipping agents.

If the dictionary had had words with at least 17 or more different letters they should have been removed, since it is impossible to lose with them (26 letters in the alphabet - 9 wrong guesses without losing = 17). However, that is not the case, as seen in the graph in the following page, illustrating the distribution of words with varying numbers of letters:



### Difficulty of Guessing a Word

The difficulty is dictated by several factors: Clearly it is easier to avoid dying as the number of different letters in the word increases, but 80% of the words have 8 different letters or less, so the number of easy-to-guess words is rather low.

Word length is not a hugely important factor, only when it is first used to eliminate all possibilities with a different length. The reason for this is that a word might contain more than one of a given letter. This can both be an advantage and a disadvantage, since guessing a letter with several occurrences in the word helps eliminate more words, but fewer different letters also makes it more difficult to actually guess a letter contained in the word.

### Artificial Intelligence

It is possible that the mating method used could be changed in a way that either resulted in better performing DNA strings, or achieving similar results in fewer iterations.

The results achieved by the genetic DNA’s are similar or better than the “Ultimate DNA” that we assumed would be a near optimal sequence to guess characters in a game of hangman. This indicates that our initial idea about making guesses based on the probability of existence of the letter in the remaining words isn’t the optimal way of gessing. The results also show that the DNA sequences were getting better and better the longer we allowed the server/client applications to test new DNA’s, and that randomly generated DNA’s generally performed very poorly compared to the genetic DNA’s.

The Clever Agent can also be considered an artificial intelligence, and is extremely efficient at “winning” games of hangman with a win rate of over 98%.

### Parallelization

By splitting the program into two separate applications (server/client) and having them communicate via http, we formed a small-scale grid computer. By making the calculations threaded we achieved further speed gains proportional to the number of CPU cores on the client machines. The DNA sequences were allowed to evolve over approximately 24 hours for each testing, running a total of 8 threads at the time. Without parallelization the tests would probably have taken vastly more time to run.

# References

**[1]** *Computer Architecture: A Quantitative Approach, Volume 1* (2007) - John Hennessy and David Patterson

**[2]** *Distributed Computing: Principles, Algorithms, and Systems* (2008) - Ajay Kshemkalyani and Mukesh Singhal

**[3]** *Artificial Intelligence: A Modern Approach* (2010) - Stuart Russel and Peter Norvig