Bachelor Thesis

Independent degree project - first cycle

Datateknik

Computer Engineering

Parallelism in Node.js applications
Data flow analysis of concurrent scripts

Linda Jansson
Abstract

To fully utilize multicore processors in Node.js applications, the applications must be programmed as multiple processes. Parallel execution can increase the throughput of data and hence lower data buffering for inter-process communication. Node.js’s asynchronous programming model and interface to the operating system make for convenient tools that are well suited for multiprocess programming. However, the run-time behavior of asynchronous processes results in non-deterministic processor load and data flow. That means the performance gain from increasing concurrency depends on both the application’s run-time state and the hardware’s capacity for parallel execution.

The objective of this thesis work is to explore the effects of increasing parallelism in Node.js applications by measuring the differences in the amount of data buffering when distributed processes run of a varying number of cores with a fixed rate of asynchronously arriving data. The goal is to simulate and examine the run-time behavior of three basic multiprocess Node.js application architectures in order to discuss and evaluate software parallelism techniques. The three architectures are: pipelined nodes for temporally dependent processing, a vector of nodes for data parallel processing, and a grid of nodes for one-to-many branched processing.

To simulate and visualize the run-time behavior, a simulation environment using multiple Node.js processes is created. The simulation is agent-based, where the agent is an abstraction for a specific data flow within the application. The simulation models and visualizes all of the data flows within a distributed application where processes communicate asynchronously via messages through sockets.

The results show that performance can increase when distributing Node.js applications across multiple processes running in parallel on multicore hardware. There are however diminishing returns as the number of active processes equal or exceed the number of cores. A good rule of thumb seem to be to distribute the decoupled logic across as many processes as there are cores. The interaction between asynchronous processes is on the whole made very simple with Node.js. Although running multiple instances of Node.js requires more memory, the distributed architecture has the potential to increase performance by nearly as many times as the number of cores in the processor.

Keywords: Node.js, parallelism, concurrent programming, multicore.
Table of Contents

Abstract ............................................................................................................................................. iii

1 Introduction ..................................................................................................................................... 1
  1.1 Background and problem motivation ..................................................................................... 1
  1.2 Overall aim .............................................................................................................................. 2
  1.3 Scope ......................................................................................................................................... 3
  1.4 Detailed problem statement ..................................................................................................... 3
  1.5 Outline ...................................................................................................................................... 3

2 Theory ............................................................................................................................................ 4
  2.1 Differences in sequential and parallel code. .............................................................................. 4
  2.1.1 An example: file listing of a file system hierarchy ............................................................. 4
  2.2 Concurrent programming ......................................................................................................... 5
  2.3 Node.js' single-threaded event-oriented concurrency model .................................................. 5

3 Model ............................................................................................................................................ 7

4 Implementation .............................................................................................................................. 9
  4.1 How the simulated system works ........................................................................................... 9
  4.2 How the simulation software works ....................................................................................... 11

5 Results .......................................................................................................................................... 13
  5.1 Pipelined processes ............................................................................................................... 13
  5.2 Vector processes .................................................................................................................... 13
  5.3 Grid processes ....................................................................................................................... 14

6 Discussion ...................................................................................................................................... 16
  6.1 Ethical considerations ............................................................................................................. 17
  6.2 Future work ............................................................................................................................ 17

References .......................................................................................................................................... 18
1 Introduction

To fully utilize the capacity for parallelism in multicore processors when writing Node.js applications, the application must be programmed for running across multiple processes. The asynchronous and event-oriented programming model in Node.js, along with its tools and interfaces to the operating system, make Node.js suitable for creating distributed software.

However, asynchronous processes that communicate introduce non-deterministic run-time behavior; the order of events cannot be determined because of various delays that are out of the programmer’s hands. This complex behavior is the result of numerous independent actions affecting a single environment. The result is that it is not obvious how the components of a distributed system can make use of multiple cores; each Node.js instance is single-threaded is will thus block and buffer any messages sent to it. Blocking will lower throughput and increase memory usage.

This thesis work explores the emerging behavior of multiprocess Node.js applications for hardware with limited capacity for parallelism.

To understand and reproduce complex systems, a useful method is agent-based modeling. In an agent-based model, independent actions are taken by autonomous agents that share a single environment that they both react to and affect. The logic within each agent is simple while the emerging behavior of the system is complex. This method is used to examine and visualize the data flows of a distributed system.

1.1 Background and problem motivation

The speed of sequential computing, as opposed to parallel computing, relies on how much work can be pushed through the processor each second. The effectiveness of a sequential algorithm is therefore affected by the number of instructions it generates. Although it’s not irrelevant, this is an old model. Today, processor manufactures have stopped increasing clock speeds and instead turned to multicore processors as a solution to the heat produced from other architectures. With the advent of multicore processors, the responsibility for increasing software performance has shifted from hardware developer to software developer: program’s need effective architectural parallelism. The consequence of changing to a more parallel-oriented paradigm is that the idioms of sequential programming no longer apply.

Research into software parallelism [1] asks the question of what tools make best use of parallelism. Current suggestions of what problems must be solved and how to solve them differ, but provide useful insight into the problem domain.
An interesting point in the discussions is that parallelism is better suited for modeling real-world systems. Multi-agent systems behave more like biological systems where simple autonomous entities simultaneously follow similarly simple rules without global synchronization. The corresponding sequential model of such a system would in contrast be very complex. The most important thing to consider is then perhaps the mindset of programmer. The tools necessary to simplify parallelism may be very different from the sequential ones. The many commonly used programming languages are from times when there were no practical applications of parallel computing and this is reflected in their programming idioms and optimization techniques: sharing references, blocking on I/O, and using iteration instead of recursion.

In a webcast by David Ungar in 2011 [2], he spoke about how implicit determinism and correctness in sequential program results in unnecessary ordering dependencies that have a negative impact on performance since they cannot be run in parallel by the hardware. He talks about there being a trade-off between performance and synchronization in programs, as confirmed by Amdahl’s law which infers that speedup from parallelism will not result in programs being faster than its sequential part. By getting rid of synchronization entirely we are no longer restricted by Amdahl’s law, but instead have to deal with race conditions; i.e., non-determinism. We get the wrong answer more quickly, but it will be usable by the application. If applications are programmed to deal with non-determinism, then this relaxes the synchronization guarantees made by hardware, making optimization easier and more effective.

Node.js applications use lock-free task concurrency. While the program only has one thread of execution, the tasks it performs are independent and asynchronous. That means that the application can quickly react to events that were generated by parallel processes because Node.js applications are aware of external processes and can easily communicate with them asynchronously. Sandro Pasquali, author of the book Mastering Node.js [3], describes how to achieve higher performance through parallelism with Node.js applications. He writes:

Through architectural parallelism, our systems can manage increased data volume more efficiently. Specialized systems can be isolated when necessary, even independently scaled or otherwise clustered. [...] Node is particularly well suited to handle two key aspects of horizontally scaled architectures. First, Node enforces non-blocking I/O, [...] encouraging a decoupled architecture. Second, Node places great importance on supporting as many fast network communication protocols as possible.

The name Node.js is also idiomatic to the nodes of a network, further implying an inherent aptness for dealing with distributed communicating parallel processes.

1.2 Overall aim

The project's aim is to present tools to better understand the run-time data flows of multiprocess Node.js applications within an environment with limited parallelism. The work is meant to demonstrate the usefulness of Node.js as a tool to
achieve parallelism in desktop applications. If the tools for parallelism are simple the future may see more programmers focus on parallelism, resulting in software that is both more comprehensible and better suited for the hardware in use.

1.3 Scope

The focus of the project is the creation and usage of the simulation application. It’s results comes from modeling different but generic software architectures. The results are valid only for a general understanding of the used architectures, but these still represent the basic building blocks of multiprocess systems.

1.4 Detailed problem statement

While the overall problem domain is parallelism in Node.js applications, the detailed problem is the run-time behavior of concurrent tasks spread across processes. Specifically, the cost of data buffering. Because Node.js is single-threaded, all event handlers are queued in the same data structure. While the tools in Node.js are easy to use, misusing them can result in large memory usage.

To not misuse these tools, we must understand the system run-time behavior of the most basic building blocks of multiprocess applications. These are the structures that form the network of nodes where data flows:

- A pipeline of nodes for temporally dependent processing,
- A vector of replicated nodes for data parallel processing,
- A grid of nodes for one-to-many branched processing.

The objective is to simulate and measure the effects on buffering in each of the above structures when increasing the capacity for parallelism. By so doing, it enables the discussion of best practices regarding multiprocess Node.js applications.

1.5 Outline

Chapter 2 describes the field of study. Chapter 3 describes the simulation set-ups. Chapter 4 describes the construction of the simulation application and how it works. Chapter 5 presents the simulation results. Chapter 6 summarizes the thesis work and discusses future work.
2 Theory

Parallel computing is generally concerned with high performance computing and low-level hardware interfaces, which is not directly related to Node.js applications written in JavaScript. Nonetheless, the concepts and intuitions are relevant and help in adopting the right mindset to understand the benefits and drawbacks of concurrent programming.

2.1 Differences in sequential and parallel code

In Guy Steele's talk about parallelizing functional code [4], he brings up three important differences in sequential and parallel code. Firstly, sequential code paradigms minimize the total number of operations while parallel code will instead perform redundant operations to reduce communication. For example, in sequential code we use clever tricks to reuse previously computed results at a later time – memoization. This would cause synchronization dependencies when used in parallel code, resulting in less parallelism because processes must interact. The second point is that sequential algorithms try to minimize memory usage while parallel code uses extra memory space to permit temporal decoupling; i.e. separate memory spaces allow safe asynchronous memory access. The third point was that sequential idioms stress linear problem decomposition, which means that you compute a value by processing one part at a time and then accumulate the results. Parallel code requires multiway problem decomposition and multiway aggregation of results which generally means doing the same computation on different data simultaneously; this is also known as data parallelism.

2.1.1 An example: file listing of a file system hierarchy

As a more practical example, consider the sequential recursive search of a file system hierarchy. A single worker must search each directory by descending deeper into the hierarchy, and then backtrack until each file has been looked at.

With parallel programming, multiple workers are started and each will be given a file path from a collection of previously found directories. Each worker searches only the first level of the given path, without descending, and collects information on the data files. If it encounters a directory file, then it sends it back to the collection of directories. When the level has been searched, the worker then waits for a new directory path to search. Directory paths are only distributed to workers that are not already busy searching. This is illustrated in Figure 1.
2.2 Concurrent programming

Concurrent programming means using a model that handles several control flows at the same time. The concurrent control flows don’t have to run simultaneously – that would be parallelism – just be simultaneously managed. Hoare presents a formal language for interacting concurrent control flows called Communicating Sequential Processes [5] to describe the behavior of these systems.

Hoare also discusses the concept of coarse-grain concurrency and fine-grain concurrency [6]. Coarse-grain means that the concurrent control flows don’t share memory space and only interact through their input and output. Fine-grain concurrency involves control flows that interleave their access to shared memory at the fine granularity of single instruction execution – meaning no context switching.

When programming concurrent control flows, programmers will have to deal with the risks of race conditions, non-determinism, deadlocks and livelocks, and also consider the overhead of reliable planning by synchronizing control flows.

2.3 Node.js' single-threaded event-oriented concurrency model

Node.js is a command-line tool and a run-time system. It interprets JavaScript programs and provides a set of application programming interfaces accessible to the program. Underneath, Node consists of: a JavaScript engine, a multi-threaded library for asynchronous events and system interaction (libuv), and bindings written in C++ and JavaScript.
Node.js’s concurrency model uses an event-loop to support single-threaded event-driven JavaScript. Control flows are implemented as event-handlers. This means that while there are multiple active control flows accessing the same memory space, there is at most one event-handler running at any time and all event handlers will run to completion without being interrupted. Therefore, the programmer need not worry about concurrent control flows changing the shared memory state. It’s a fine-grain concurrency model without context switching, but the JavaScript programs are single-threaded and cannot utilize multicore parallel execution.

Event-driven programming is a programming style where the flow is determined by the occurrence of events. The events in Node.js are either external system events or internally generated events. Both invoke attached callbacks with closures by pushing them onto the event loop's queue. The event loop executes the callback that is the first in the queue.

To program applications for Node.js means mainly working with asynchronous function calls to not block the event loop. This is the de facto way to write programs with JavaScript which is also the language used in modern event-driven web browsers. Together with the native asynchronous facilities in JavaScript, Node.js provides many asynchronous library functions to make the event-oriented asynchronous paradigm easier to work with.
3 Model

Three multiprocess application architectures are simulated for a different number of cores. The collected data measures the pressure in the multiprocessor application, calculated by sampling the total amount of agents in the system each second throughout the simulation. The simulation set-ups are:

- **Pipeline**: a single-path task pipeline of eight nodes, having an input data rate of 64 agents every 100 milliseconds. Agents have the same set workload for each node. The set-up is illustrated in Figure 2.

- **Vector**: a vector of four parallel nodes are set up without any merges to represent replicated functionality. 64 agents are spawned every 100 milliseconds with the same workload and only moves through one node. All 64 spawned agents of each round are sent to one of the nodes, which is decided in a round-robin fashion. The set-up is illustrated in Figure 3.

- **Grid**: A grid of nine nodes set up 3-by-3. Agents arrive at the top-left node and travel to the bottom-right node, only moving right or down. At each node where the path branches to the right and downwards, the agent spawns a new agent that takes the path not taken. One agent is spawned at the top-left node every 100 milliseconds. The set-up is illustrated in Figure 4.

![Pipeline of nodes. Input enters the first node and output leaves the last node in the pipeline.](image-url)
The data collected and presented in this report is from sampling the sum of all the nodes' queue lengths using 1, 2, 4, 8 and 16 simulated cores. This is the amount of concurrent activity in the application and is a measure of pressure generated for the amount of incoming data. The relation of interest is between pressure and parallelism. Parallelism is derived from the number of cores available for simultaneous processing.

The number of cores is plotted against the resulting distribution of concurrent agents. The difference in activity from doubling the number of cores is given as a percentage using the average total activity throughout the simulations. Maximum and minimum activity, as well as standard deviation, is also presented.
4 Implementation

There are two parts to the simulations performed in this project: the simulated model and the simulation software itself. The simulated model is of a multicore computer running multiple Node.js processes. The Node.js processes are called nodes and the system is a network of concurrent nodes. The simulation software itself is a multiprocess Node.js application that runs in a distributed environment using Node.js and web browser tools.

4.1 How the simulated system works

The simulated system is a network of nodes, and each node receives tasks to perform. Tasks are buffered in a first-in-first-out queue. An agent consists of an ordered sequence of tasks, and is an abstraction for a distributed control flow. Nodes request processing resources in the form of cores to perform the received tasks. Only as many nodes as there are cores can execute in parallel. A scheduler determines which nodes can execute at any time by distributing the available cores. After a certain amount work cycles, all cores are reclaimed so that they can be re-distributed to active nodes. Active nodes are nodes that have non-empty queues. An illustration of the system is given in Figure 5.

![Figure 5: Network of eight asynchronous node processes that communicate. The green hexagons are nodes and the stacks next to them are their task buffers. The dots are agents in flight, going from one node to the next in order to enqueue tasks.](image)

The simulation is agent-based to capture the complex interactions between the asynchronous nodes that make up the system. The abstraction for all the distributed control flows is an agent. These agents are small re-enterable routines that maintain a single sequence of tasks. Internally, an agent keeps an array of node IDs to represent how it flows through the system of nodes. It also has a corres-
pondering array of workloads to determine how much work is to be performed at each node. For each task in the path, an agent arrives at the end of a node's queue. When the agent is at the front of the queue it can begin to have its work processed. When the work is done, the agent moves to the next node or terminates if all of the agent's tasks are done. The structure of agents and how they move is illustrated in Figure 6.

![Figure 6: Structure of the agent. The top part shows how an agent consists of a path through nodes along with the corresponding workloads to be performed there. The bottom part shows how the agent will flow through the system of nodes.](image)

The nodes compete for cores and the scheduler is fair in its distribution. Nodes can make use of any core to execute its tasks one by one. The cores represent the capacity for parallelism in the processor, where each core can be used by one node at a time to process tasks. The concept of how cores are brokered to nodes is illustrated below in Figure 7.
4.2 How the simulation software works

There are four communicating processes running the simulation. One process manages the concurrent agents and nodes, one process regulates the distribution of cores, one process collects data and sends it to the fourth process which renders a graphical presentation of the current system state. The view of the system is a web browser application that communicates with the software through a WebSocket. All other processes are implemented in Node.js. The messaging infrastructure is implemented using ZeroMQ sockets, which is made available through a Node.js module. Figure 8 is a diagram of the simulation application.

Figure 7: Diagram demonstrating fair queueing as performed in the simulation. In this figure, there are four node queues and two available cores. Each queue contains buffered tasks, waiting to be processed at the corresponding node. The scheduler dynamically distributes the core resources fairly.

Figure 8: Diagram for the simulation software. Each box is a separate process. All except the view process is running a Node.js instance. Interprocess communication is implemented using ZeroMQ sockets.
An initial handshaking communicates the parameters of the simulation and throughout the simulation the queue lengths at each node are sampled once every second. The simulation parameters are:

- the number of cores,
- the number of nodes,
- the spawn-rate and number of agents,
- the paths that each agent takes through the node network,
- the workload of each task for all agents,
- how fast the nodes can execute one unit of work,
- how often the scheduler re-distributes cores.

To simulate a certain multiprocess software architecture, the rules for how agents flow are programmed into the agent routine. An agent is similar to a co-routine: a re-enterable procedure that yields execution control at certain steps in the routine.
5 Results

The resulting queue buffers for a different number of cores are presented as graphs for each multiprocess setup within the following subsections. The measurement for pressure is the average sum of all node queues throughout the simulation, as well as standard deviations, maxima and minima.

5.1 Pipelined processes

The statistics for the measure of pressure for a pipeline of eight nodes is presented in Figure 9.

![Figure 9: The pressure in a pipeline of eight processes as the number of cores increase. Blue dots denote average value, thin dashes are standard deviation markers, thick dashes are maxima and minima.](image)

The results show increased performance as the number of cores increase with diminishing returns; as the number of cores increase, the performance gain goes toward no gain in performance.

5.2 Vector processes

The statistics for the measure of pressure for a pipeline of eight nodes is presented in Figure 10.
Figure 10: The pressure in a vector of four processes as the number of cores increase. Green dots denote average value, thin dashes are standard deviation markers, thick dashes are maxima and minima.

The results show increased performance as the number of cores increase with diminishing returns. As the number of cores equal the number of processes, there is no more performance gain.

5.3 Grid processes

The statistics for the measure of pressure for a pipeline of eight nodes is presented in Figure 11.
The results show increased performance as the number of cores increase with diminishing returns. As the number of cores nears the number of processes, the drop in performance gain is immediately noticeable.
6 Discussion

What motivated this thesis work in the first place was the perceived preconception some have of Node.js; many consider it to be nothing more than a scripted web server. However, as a run-time system with a large set of interfaces to the underlying operating system, Node.js applications are capable of more. This work has demonstrated the usefulness of Node.js as a tool to achieve parallelism in desktop applications.

For this work, Node.js was used to implement a simulation environment to understand the behavior of multiprocess applications through agent-based simulations. The simulation was distributed over multiple Node.js processes and the lock-free asynchronous programming model in Node.js made it easy to both build and also communicate with a web application.

The results show that for all the simulations, performance can increase multiple times if the hardware allows, simply by distributing logic across multiple processes. Increasing parallelism by doubling the number of cores even resulted in more than a doubling in average performance as a result of the increased throughput and thus reduction in memory usage because buffers don’t grow as fast. There were however diminishing returns as the number of cores equal or exceed the number of processes, since the application cannot use the increased capacity for parallelism – it cannot be simultaneously spread across any more cores.

The cost of data buffering in a multiprocess environment feeds back into the data flow system, hence lowering throughput and increasing buffer growth when data input arrives externally at a fixed rate. A system that feeds back its own output data would either stabilize at a steady throughput, or – due to overloading – would spiral into a system crash.

As a rule of thumb, to better utilize a multicore processor without crashing, distributing the decoupled logic across as many processes as there are cores should achieve increased performance even if it isn’t optimal utilization. This avoids excessive growth of unbound buffers and also has the potential to speed up execution. This applies to any software architecture that use parallelized pipelines, vectors or grids where there is a high enough activity in the processes. It is probable that low activity processes that don’t overly tax the processor can be spawned without a noticeable decrease in performance if the working memory is large enough. Actually, higher distribution increases concurrency so that – even with increased memory usage and without speedup – the benefits from a decoupled software architecture is that it allows better scaling if moving the software to other hardware or distributing processes over more computers connected over a network.
6.1 Ethical considerations

The ethical considerations is about the programming model that run-time systems like Node.js provide. Specifically, how it affects the work of programmers and system developers.

The advantage of creating a run-time system that provides a programmable concurrency model is that it encapsulates all of the multithreading primitives and provides automatic optimization. Programs are more declarative which results in correctness being easier, software structure being more comprehensible and development-times being shorter.

The possible disadvantage of adopting a new programming model is that it takes time getting used to. If tools are misused then it can damage software quality instead. Developers must consider the learning curve and whether their problem domain benefits from concurrent programming and parallelism. A major port of source code may be infeasible and building interfaces may increase complexity, but it will probably increase software quality as well as make the programmers’ jobs simpler if the alternative is multithreaded programming.

6.2 Future work

The simulation provided a general view of the run-time behavior of multiprocess Node.js applications. The same measurements could be performed on actual multiprocess applications on actual hardware. It could be built as a testing tool suite or framework. If the number of cores continue to increase in modern desktop processors, this would become more relevant testing tools.

The execution model in Node.js is simple, but powerful. If a decoupled software architecture is easier to use than multithreaded programming – that rely on context-switching – in order to achieve parallelism, then the future of high-level programming should be more concerned with dealing with communication tools. This includes communication channels, real-time data streams and message protocols.
References


