UTS: A Portable Benchmark for Erlang/OTP

Mikael Östberg

Bachelor of Science Thesis
Stockholm, Sweden 2010

TRITA-ICT-EX-2010:118
UTS: A portable benchmark for Erlang/OTP

Author:
Mikael Östberg
ostber@kth.se

This paper is part of a bachelor thesis supported by the following institutions:
Swedish Institute of Computer Science (SICS)
The Royal Institute of Technology (KTH)

Examiner & Supervisor:
Mats Brorsson

Opponent: Israt Jahan
Abstract

In this paper the benchmark Unbalanced Tree Search (UTS) is ported and evaluated to the functional programming language Erlang. The purpose is to provide a portable benchmark that scales as the number of cores do in a system. Since Erlang is language built around concurrency language the speedup should prove to be interesting comparing to its competitors as the number of cores rise. This paper is written to describe how the algorithm works as well as what it has performed on a few different systems at SICS and presents the conclusions that can be drawn from them. Some questions remain unanswered however such as how well the benchmark performed on the Tilera64 because of some technical difficulties during the project. Also the results proved quite odd since there are possible bottlenecks in the performance making the speedup per added processor core somewhat limited. As a consequence of the strange behavior of the software some of the conclusions drawn from this thesis are mostly speculations.
7.2.2 One scheduler per core .................................................................................. 20
7.2.3 One scheduler .................................................................................................. 20
7.2.4 Graphical results ............................................................................................... 21
8  Discussion .............................................................................................................. 22
8.1 Bottlenecks .......................................................................................................... 22
8.2 Scalability ............................................................................................................. 22
9  Conclusions ........................................................................................................... 23
9.1 Some answers ....................................................................................................... 23
9.2 Speculations .......................................................................................................... 23
9.3 Recommendations ............................................................................................... 23
10 Future Work .......................................................................................................... 24
11 References ............................................................................................................. 25

Figures
Figure 1 – three recursive calls to the same function until a condition is met ............... 9
Figure 2 – Disambiguation of the tree data structure .................................................... 11
Figure 3 – A typical SHA-1 state/digest looks might look like this ............................... 13
Figure 4 – One recursive step of the UTS Algorithm .................................................... 14
Figure 5- A tree being traversed in parallel ................................................................. 14
Figure 6 - The C syntax of the recursive call in UTS .................................................... 15
Figure 7 - The Erlang syntax of the recursive call in UTS .......................................... 15
Figure 8 – Unbalanced Tree Search in Erlang ............................................................ 16
Figure 9 - Flowchart of how the C-node receives processes and returns data ............ 17
Figure 10 – The Tilera64 Chip ................................................................................... 18
Figure 11 - The performance of FET using 7 schedulers and N number of C-nodes .... 19
Figure 12 – Performance statistics of systems having run UTSE ............................... 21
Figure 13 - Relative speedup measured against the sequential version ...................... 21

Tables
Table 1 – A table of the most commonly used words throughout this paper ............... 5
Table 2 – Specifications of multicore processors that have performed UTSE ............ 18
Table 3 - Benchmark results based on one scheduler per core ................................ 20
Table 4 - Benchmark results based on one scheduler ................................................. 20
**Acronyms and terms used in this thesis**

For anyone not feeling really familiar with the terms used in this report this is a list of almost every abbreviation and common term used throughout the report. This can be useful for someone also into the subject if there is any doubt about what is meant by a certain term one can always go to this dictionary for a response about what that means.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTS</td>
<td>Unbalanced Tree Search</td>
</tr>
<tr>
<td>UTSE</td>
<td>Unbalanced Tree Search in Erlang</td>
</tr>
<tr>
<td>EVM</td>
<td>Erlang Virtual Machine</td>
</tr>
<tr>
<td>Node (Data structure)</td>
<td>A branch or leaf of the tree data structure</td>
</tr>
<tr>
<td>Node (Virtual Machine)</td>
<td>A standalone Virtual Machine in Erlang connected to another</td>
</tr>
<tr>
<td>Iteration</td>
<td>One executed run of a loop</td>
</tr>
<tr>
<td>Recursion</td>
<td>One executed run of a function calling itself until a condition is met</td>
</tr>
<tr>
<td>Process</td>
<td>A program executing in an operating system</td>
</tr>
<tr>
<td>Thread</td>
<td>A standalone thread of execution within a process</td>
</tr>
<tr>
<td>Syntax</td>
<td>Allowed words and expressions within a language</td>
</tr>
<tr>
<td>Multicore</td>
<td>Multiple cores in a processor working in parallel</td>
</tr>
<tr>
<td>Benchmark</td>
<td>A software used to measure performance of some component</td>
</tr>
<tr>
<td>Parallelism</td>
<td>The art of programming software running multiple threads</td>
</tr>
<tr>
<td>Algorithm</td>
<td>A problem solution expressed as a finite sequence of instructions</td>
</tr>
<tr>
<td>Erlang</td>
<td>A functional programming language</td>
</tr>
<tr>
<td>Atom</td>
<td>A constant in Erlang, can be a number, list, string or a tuple</td>
</tr>
<tr>
<td>List</td>
<td>A list of Elements in Erlang [Element, Element, Element]</td>
</tr>
<tr>
<td>Tuple</td>
<td>A data structure in Erlang. (Element, Element, Element)</td>
</tr>
<tr>
<td>C</td>
<td>An imperative programming language</td>
</tr>
<tr>
<td>Struct</td>
<td>A data structure consisting of one or more accessible elements.</td>
</tr>
<tr>
<td>Epoch</td>
<td>The time passed since January 1\textsuperscript{st} 1970 in seconds. The UNIX-clock.</td>
</tr>
<tr>
<td>Seed</td>
<td>An integer used to spawn a certain sequence</td>
</tr>
<tr>
<td>Bottleneck</td>
<td>A process or thread that works 100% and limits execution of others</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit (The “brain” of a computer)</td>
</tr>
</tbody>
</table>

*Table 1 – A table of the most commonly used words throughout this paper*
1 Introduction

Technology is constantly moving forward at a great velocity. Science is producing faster, more powerful and smaller devices that are becoming increasingly present in everyday life. It is fascinating at what rate science has developed over the latest 50 years. The transistor was a revolution in electronics around 1960s and two years ago Intel announced that they can fit at least 2 Billion of them on a single processor chip in a computer. There are several computers in each home that is more powerful than the supercomputers of yesterday. The performance of a processor has followed Moore’s law for the last 20 years in a surprisingly accurate manner. It tells us that performance of a processor should double every second year. This is no longer the case as physical boundaries are leading science in a new direction, possibly leading towards the next revolution in performance.

1.1 Multicore

Historically a processor has been a single unit processing instructions in a computer, and the development of components to increase the performance of these processors has mostly resulted in increasing the clock frequency of single powerful cores. Due to several factors, such as power and heat dispersion, we are at the physical limit of how efficient we can make these single CPUs. Multicore, however, allows two or more cores to process instructions simultaneously. These cores are slower and less expensive but can reach the same and better performance than one powerful single CPU. New technology spawns new problems however since a multicore processor faces problems with data synchronization and utilizing the processor to its full potential. Certain sequential applications might even run slower on a multi-core processor since it is only able to run on 1 of the slower cores. This is why it has become much more important now than before to teach programmers of today how to program multi-threaded applications.

1.2 Benchmarks

As processors have improved at a high rate it has become difficult to evaluate the relative performance of these multiple cores since they depend on many different factors. These factors might be cache size, clock frequency, busses and schedulers and so on. One reliable and trustworthy way to evaluate processor speed and efficiency of programming languages is through using a benchmark.

A benchmark is software with the purpose to test a specific factor of a component. It could be for the purpose of evaluating the performance of graphic circuits or how well the processor caches are used or how well the processors work in parallel. This software usually has a deterministic behavior. Determinism means that if the same settings are used, then the application will perform the exact same execution, allowing the user to measure performance across different platforms or systems. The measurement used could be time-based or based on the number of finished tasks during a set amount of time or some other factor that is easily measured. Normally the performance is translated into some kind of value or speedup factor.

A benchmark has the power to produce statistics that motivate changes or results and is widely used as a way to simulate architectures or systems before they are actually deployed or built to see if it is worth the money to do so.

Benchmarks can of course be written in all sorts of programming languages. Operations are scheduled and executed differently depending on the language used. The resources of a processor are thereby used slightly different between languages. Therefore it could be interesting to use a different language employing the same benchmark algorithm since it would show the difference in how these languages employ these resources.
2 Purpose

The project is part of a bachelor thesis which was approved by SICS (Swedish Institute of Computer Science). The purpose of SICS as an institution is to contribute to the competitive strength of industry by conducting research in computer science in which this thesis is a small contribution to Erlang and multicore research.

2.1 Why another benchmark?

As mentioned in the introduction multi-core technology is here to stay and the number of cores is only going to increase and therefore scalable benchmarks are needed for evaluating the performance of these many-core systems and more specifically how they can handle concurrent applications and languages.

2.2 Erlang

This projects prime purpose is to investigate how well Erlang scales with multiple processors as the number of processes become extremely many.

Erlang is built to be able to handle very high amounts of processes since they are very lightweight. Erlang is well known for its ability to handle concurrent applications without the need of synchronization through mutual exclusion locks. Evaluating the Erlang Virtual Machine scalability is of interest to anyone who would have the need to create a concurrent application on a modern computer.

Erlang also has an interface that has the ability to create a virtual Erlang machine that communicates via messages and thereby allows it to access C-libraries. It could be of value to the Erlang developers at Ericsson that the efficiency of this interface is tested in a stressful environment like this since it is not very commonly used judging from the lack of projects on the internet that used the library.

2.3 Questions to be answered

In the discussion part of this thesis, the following questions will be sought to be answered. It will likely be possible to draw some conclusions to what causes the results to look as they do in chapter 7.2.

- Does UTS scale well as the number of cores increase in Erlang?
- Does an 8-core processor really perform eight times better than a Single-core?
- How good is the speedup compared to the sequential variant when using N schedulers?
- How good does the speedup get in the Erlang Virtual Machine?
- Does process overhead on multicore processors become a bottleneck and limit speedup?
- How many concurrent C-nodes are optimal per the number of cores in a processor?
3 Project method

The project consists of three parts that can be summarized by section 3.1 to 3.3. Section 3.4 describes the limits of the project that has been laid down so that it will not become too large for a bachelor thesis.

3.1 Research

An in-depth study of papers on the subject is studied along with the source code of UTS. How trees as a data structure works and specifically unbalanced trees to get a deeper understanding of their usage in the software. Also reports on UTS (1) to get to know how the algorithm is built and thereby identify any problems that might surface when translating C to Erlang. The result of this preface consists of the chapters 4 and 5.1 which go in-depth about how UTS and all its components work together to create the benchmark.

3.2 Porting the software

The second process consists of translating parts of UTS that can be translated, parts that cannot; will be integrated with Erlang through the C-library erl_interface.h supplied with Erlang development packages. It has been decided that the porting will be limited to only porting the algorithm and concurrency to Erlang since the RNG-library is quite large and complex. The probability of making errors hard to detect by translating is far greater than that.

The results of this part are the finished software benchmark ready to be distributed and possibly uploaded to sourceforge.org for possible future work. The result of this process is chapter 6.1 to 5.2 which covers the functionality of the code.

3.3 Analysis of results

The finished software will undergo testing on three of SICS computers. Where the idea is to show how well the software scales as the number of cores increase as well as answer a few questions specified in chapter 2. Another interesting aspect is to see how well it scales as the number of schedulers of the Erlang virtual machine drop down to one. The analysis and results can be found in chapter 7.

3.4 Limitations of the project

Since the project has some scientific value it can be expanded to making very detailed benchmarks and specialized versions of the UTS-algorithm. So therefore some limitations to the project have been created so that it will not exceed the limits of a bachelor thesis.

The project is limited to porting the main algorithm to Erlang and avoids doing so with libraries in C that involve computation of SHA-1 hash digests to create pseudo-random numbers, instead the project focuses partially on creating some kind of interface to these libraries. This interface has potential of becoming a performance-bottleneck so that will be included when considering the results their analysis.

The project is limited to implementing and testing the software on chosen computers specified in section 7.1. Extensive testing and performance analysis is put on the development community since the source will be possibly be published on sourceforge for further development outside of this thesis.
4 Theoretical Background

This chapter will cover some of the basic theoretical background that is needed to digest the rest of the paper since it becomes very technical to someone not familiar with programming of parallel systems. Both the benchmark and Erlang as a programming language relies heavily on recursion which is why it has gotten a chapter of its own. It is quite important to have a firm grasp of how a recursive call works to be able to digest the rest of the content. Readers who feel they have a grasp of programming with recursion, trees and concurrency can ship chapter four as a whole and proceed to chapter five.

4.1 Recursion

In programming there are a few different programming paradigms to adhere to when creating software. A program can be built relying on loops that iterates a process of computing tasks, the problem is that this paradigm usually creates quite a bit of code and classes can become quite large and abstract. This is called iterative programming.

Recursion is an alternative to the iterative style of coding. Recursion is defined as a function making use of calling itself until a certain condition is met, just as in Figure 1. The advantage of using this paradigm depends on several factors such as how stack-intensive the calls are, in some cases recursion can cause memory overflow of the stack because too many calls to the own function has caused the process stack to become full of temporary variables. This depends very much of the implementation however, and some problems are inherently recursive and would become slower using iterative programming. Some of these algorithms are quicksort, fractal calculation and tree traversal as seen in Figure 2

![Figure 1](image-url) – three recursive calls to the same function until a condition is met.
4.2 Programming languages

In this project an algorithm called UTS (1) is ported from the source programming language C to the target language Erlang. So it is important to have an idea of some of the main differences between the two.

4.2.1 C

C is a robust general-purpose programming language dating back all the way to 1972. C is one of the most popular programming languages (2), originally intended for system software used to control system resources and functions. But it is also used for application software that can be run directly by the user. It has libraries and Application programming interfaces like OpenMP and Pthreads that allows a rather skilled programmer to parallelize software even though C was not intended to do so originally.

4.2.2 Erlang

Erlang is a general-purpose functional programming language and runtime system in the form of a virtual machine. The language was originally created by Ericsson in the 80s to handle their private branch exchange (PBX) stations. Since the programming languages of that day lacked certain aspects of concurrency and error recovery. Erlang adopted high level symbolism from Lisp, Prolog and Parlog and needed a concurrent granularity to symbolize one phone call as one process (3).

Erlang differs from C in one big fundamental manner; it is entirely functional and does for an example not make use of any loops or variables, instead it has constants and lists. Since Erlang is functional all loop-operations must be made into recursive calls and make use of pattern matching. While threads are considered complicated to program and prone to errors in other languages, Erlang has features for creating and managing processes with the aim of simplifying concurrency (4). One of those features is that it has no need of data synchronization whatsoever since there are no global variables.

Processes in Erlang are very cheap to create and communicate with each other via asynchronous message passing for synchronization. Asynchronous meaning that each process has its own mailbox of messages that it retrieves using the receive keyword, if no messages are to be found then the program blocks until a message is received unless specified otherwise.

4.3 Threads & Concurrency

A thread is a set of instructions that executes in parallel with other threads within a process and may be scheduled to do so in different cores depending on the scheduler of the system. Using threads is a vital paradigm to programming for multicore processors since it allows threads to execute instructions in true parallel on multiple cores. It is important however that those threads have some kind of data synchronization between them so that one thread does not write to a variable at the same time as another thread reads it. Most synchronization techniques cause threads to go into a blocked state until the variable is not being used by another thread.

The efficiency of parallel execution is limited by the part of the execution that is controlled by synchronization. This factor is often highly dependent on the programming language used and how it has chosen to implement concurrency.
4.4 Trees

A tree is a data structure built on linked nodes. A node is an object that stores some kind of data, keeps track of how many children it has, and keeps references to its parent and children. Nodes can at most have one parent and nodes who do not have any children are called leafs or leaf-nodes. The node at the top of the hierarchy is called the Root-node. The depth of a tree is the length from the root node to the child which is furthest away from the root. The tree in Figure 2 is of depth 2 because the nodes 5 and 6 at the bottom have the distance 2 to the root node.

This kind of data structure greatly simplifies searching for a specific object if you compare it to a list of linked elements where one must traverse the entire list to sort or find a certain element. The time it takes traversing a large tree of N elements is one of the most common and compared to other operations quite an expensive operation to do. Also it is an operation easily divided into several separate tasks due to its divide and conquer (5) nature which allows parallelism.

An unbalanced tree is a tree that has each node inserted in order which causes the tree to have one very heavy branch on one side of the parent and simply a leaf on the other. The worst case of traversing this tree of n nodes causes traversal to take $O(n)$ operations which is sub-optimal if you are using a tree-structure for storing data.

This means that several sub trees can be traversed concurrently proving an interesting computation for a benchmark, if you somehow have a way of ensuring that the same tree is generated and traversed every time the benchmark runs.

![Figure 2 – Disambiguation of the tree data structure](image-url)
5 Introduction to UTS

The Unbalanced Tree Search (UTS) benchmark is a parallel benchmarking algorithm that returns the performance achieved when performing an exhaustive search on an unbalanced tree. The tree is generated on the fly using a random number generator library (RNG) that allows the random stream to be split and processed in parallel while keeping the tree deterministic. RNG is constructed using a secure hash algorithm (SHA1). Generating child-nodes requires multiple runs of the SHA1 hash algorithm to generate hash-digests for each child.

In its unaltered state the unbalanced tree search benchmark algorithm hereby referred to as “the algorithm” is recursive, meaning it is initially called as a function and calls itself as a function before returning to the calling thread. The benchmark is designed to measure performance of a multicore processor by implementing the algorithm with some sort of parallel library or ported as a whole to a concurrent language such as Erlang.

5.1 The sequential algorithm in ten steps

Each of these steps has a visual representation in Figure 4. Also note that this is the sequential algorithm and has not been modified to execute in parallel, which will be covered in section 0.

1. **Start a wall-clock**
   Measure the time passed since Epoch.

2. **Create a Root node**
   Call the initiate function from RNG based on a Seed.

3. **Call the tree-search function**
   Call the function with a reference to the root and the height 0.

4. **Compare the current max depth**
   Compare the global maximum depth to the current local depth, if the current max is lower than the current depth then it must be updated with the local depth.

5. **Call the random function from RNG**
   Call the random function with a reference of the parent state which retrieves a number N between zero and one.

6. **Did this number N become larger than the branching-limit?**
   If so, increment the number of leaves and return to the parent recursion. If not then skip until step 9.

7. **Start spawning children digests**
   If N got smaller than the branching limit then it is time to start creating SHA-1 digests based on the parents digest.

8. **Recursively continue the search on each child**
   Continue retrieving each subsequent sub-tree from each child and summarize them to get the size of the subtree from this node. Meaning a regular node will go back to step 3.

9. **Return the local subtree to parent**
   Return the statistics gathered about the subtree such as its number of leaves and the total number of nodes to the parent. A leaf node returns having one node and one leaf.

10. **Stop the clock and print statistics**
    If the depth has become 0 and all children have been evaluated then it means that the current node is the root node. The clock should be stopped, the benchmark has finished.
5.2 The RNG C-library

The Random Number Generator (RNG) C-Library is a library created by Brain Gladman [dead Ref] specifically for creating pseudo random numbers using a Secure Hash Algorithm (SHA-1). That is a commonly used cryptographic hash function to encrypt data into a 160 bit digests. Before this chapter it has been referenced to something called a State in UTS. A State is a data structure in the RNG Library that consists of 20 8-bit unary integers that make up a Sha-1 hash key. An 8 –bit unary integer means that it can have 256 different values, 0 <= x <=255. Together the 20 x 8-bit values become the 160 – bit digest.

<255, 123, 142, 77, 0, 12, 199, 242, 212, 87, 90, 255, 201, 188, 13, 56, 89, 22, 122, 1>

Figure 3 – A typical SHA-1 state/digest looks might look like this.

This report will not go any more in-depth on how sha-1 works than this since it is a quite long and complicated operation to be explained here and the important part is how the RNG library uses it to generate random numbers.

These states are used to identify each particular node in UTS. The main reason for using these sha-1 keys is to generate a pseudo random number between 0 and 2147483648 to be used in UTS to decide whether a node should have children or is a leaf node. The second reason is so that each child is created based on its parents state using the RNG spawn function and a number from zero to the maximum number of children.

A state is initially created by running the rng_init(int) function which will process the data of the integer and create and return a State. This function is primarily only used for creating the root-node. Running rng_spawn(State1, State2, int) will create a new State based on State1 and an integer and put it in State2. rng_rand(State) generated a random number between 0 and 2147483648 based on the state and returns it to the caller.

The reason spawning states depending on earlier states is that a state generated from the same integer will always become the same state every time you encrypt it. And each call to spawn will generate the same B each time state A is sent to the spawn-function with the same integer. This is inherently because of its cryptology nature and highly usable for the type benchmark UTS is. It will generate the exact same tree as long as the same seed-integer for the root is used. This is crucial when randomly generating a tree because you might want to generate the same tree on several times disregarding what platform or system is used.

As this is a quite complex library it was decided to limit the project to not translate it into Erlang-code. It is integrated to the Erlang Virtual Machine through a small interface called a C-node that solves translation and parsing of Erlang messages so that it is possible to call functions from this library.
5.3 Recursive parallelism

As a first approach of making this algorithm execute in parallel one should reason as to where the algorithm is most easily split into several independent tasks. Most languages uses different ways to parallelize the software but most of them share the fact that they use parallel recursion to do so.

In the step 8 of chapter 6.2 lies the recursive call to recursively traverse a child-subtree. This is the perfect piece of code to implement concurrency into this code. It is easy to spawn multiple processes that each take care of one child and its subsequent sub-trees, just as in Figure 5 but this is only a valid solution if processes are cheap. It is then important to implement communication with the parent process so that the children may alert and resume execution of the parent when all of them are finished with traversing their sub-trees. This communication should be implemented with asynchronous message passing to ensure that no deadlocks will occur.

One way to handle the large amount of processes is through making the tasks bigger. This means that each process has responsibility for a greater number of tasks. For an example take Figure 5 and assume that every second level spawns a process per child while each level in between spawns children sequentially. This allows the parent in the figure to run sequentially until it has created Child 1 through 4. As child 1 creates its children it will spawn a new process to take care of Child 1#1 through 1#4.

![Figure 4 – One recursive step of the UTS Algorithm](image-url)

![Figure 5 – A tree being traversed in parallel](image-url)
6 Porting the Unbalanced Tree Search Benchmark

This chapter covers the practical result of the project that this thesis is built upon; an explanation on how the benchmark explained in section UTSE works in all its modules.

6.1 From an imperative to a functional language

When porting an algorithm it is important to consider the differences between the two languages before implementing functions straight off. As mentioned in section 4.2.1 C is a general-purpose language and has a standard way of implementing loops, global and local variables, threads very alike many other languages like Java, C++, python et. c. In Erlang there is no such thing as loops or variables. There are only functions, constants and recursive calls with news Atoms.

Row 6 of the C-algorithm is the subject to recursive parallelism where C must apply concurrent libraries such as OpenMP or Unified Parallel C (UPC) to cause the recursion to execute in parallel. All Erlang has to do is to pass on a reference to the parent-process and create a number of child-processes to run the function again. The Child will then send a message to the parent Process ID (PID) with the results using the “!” operator when it is finished to tell the parent that it may collect the results from the children. Then it is allowed to send the compiled results to its parent. In the figures 5 and 6 one can clearly see the differences between the languages. Both code-boxes show the exact same piece of execution, but with a few twists. For an example the increment of the variable nLeaves on row 8 can never be done in Erlang since that is a global variable and global variables does not exist there. Instead it was solved so that the node statistics being passed on to parent on Figure 5 row 6 includes the number of Leaves in the subtree of that node or if that node is a leaf it only has 1 as in row 8.

```
1. if(numChildren > 0){
2.   int i;
3.   for(i = 0; i < numChildren; i++){
4.     rng_spawn(parent->state.state, child.state.state, i);
5.   }
6.   subtreeSize += treeSearch(depth+1, &child); //can be parallelized
7. }else {
8.   ++nLeaves; //increase number of leaves globally
9. }
10. return subtreeSize
```

Figure 6 - The C syntax of the recursive call in UTES

```
1. case (NumChildren > 0) of
2.   true ->
3.     Children = spawnChildren(State, NumChildren, Options),
4.     launchChildren(Children, Depth, Options),
5.     Statistics = waitChildren(NumChildren),
6.     ParentPid ! {stats, {MaxDepth, Subtree+1, Leaves}};
7.   _Else ->
8.     ParentPid ! {stats, {Depth, 1, 1}} %leaf node
```

Figure 7 - The Erlang syntax of the recursive call in UTES
6.2 The benchmark in Erlang

The first goal was for the program to implement all the functionality that the C-variant offers. Since it was needed that the hashing algorithm was identical to the C-variant a conclusion was drawn that the C-library RNG was not to be translated since it might generate undeterministic trees. Instead an interface with the help of a specific C-library (erl_interface.h) is used to create a link from Erlang to the RNG-library and back. This allows the creation of a C-Node (6). A C-Node initializes a way of communicating between the two languages through message passing. More in depth about how the C-node works is explained in chapter 6.3. As seen in Figure 8 UTSE has a separate module that can handle the communication with C so that the UTS - algorithm in itself will become smaller and easier to understand. Therefore the RNG module was created in Erlang to handle that abstraction level. In the figure it shows that it is used as a link between the C-node and the Erlang VM sending messages back and forth as a kind of data synchronization.

The main algorithm has undergone few but important changes mostly handling the absence of global variables and implementing concurrency. At first the idea of spawning one process per node can be quite startling for a pragmatic programmer since they are usually very expensive to use both memory and processor-wise due to high overhead. In Erlang, however, processes are very cheap and can be spawned for very small tasks. A process is very likely spawned just to make one call to RNG and realize that it is a leaf-node and send one message about this. Then the parent is alerted and then the process dies. Thousands of processes are doing this work simultaneously and thereby hopefully maximizing the processor resources.

![Figure 8 – Unbalanced Tree Search in Erlang](image-url)
6.3 The C-node

Having looked into the possibilities of Erlang interoperability solutions three methods were found. Interoperability was found accessible using three different methods; ports, drivers and nodes. Due to the ease of debugging and good documentation of the C-nodes, it was chosen as the only means of communication with the RNG library.

The creation of a virtual Erlang node in a C-program allows for a very simple form of communication and makes it easy to parse and call the sought functions. It works in the manner that it connects to an existing Erlang Virtual Machine (EVM) through its short local name as well as a secret cookie that must be specified at startup of an EVM and the C-node. Then it initiates a few pointers to so called Erlang Terms (ETERMs) found in the interface. These are pointers to general data structures able to hold any Erlang data structures or constants. From them it is a small task to parse any data which has been received from the EVM without the hassle of decoding the messages received.

After receiving a message from the EVM it is connected to it parses all relevant data into the predefined ETERMs and decides what function that has been called through looking at the header of the ETERM.Parsed data are then translated into States that the RNG library can understand and process. After processing has been finished the structs are translated back into data that Erlang can understand and sent back to the EVM.

On the Erlang side a module called RNG is used which is not to be confused with the RNG library in C. This is simply an Erlang module built to send messages to a specified node and consists of no more than 25 lines of code.

There may be any number of C-nodes attached to the EVM that is running UTSE, but one must keep in mind that a C-node demands some CPU-performance to perform its tasks and multiple C-nodes might actually limit the benchmark results greatly, which is tested in 7.2.1. The C-node is run externally so it doesn’t live under the same conditions as the Erlang processes and be scheduled to different cores than the EVM processes.

![Figure 9 - Flowchart of how the C-node receives processes and returns data](image-url)
7 Benchmark Results

In this chapter the results of using the benchmark on specified hardware is presented based on criteria and questions specified in chapter 2.3.

7.1 Hardware & specifications

The software will be run on several systems featuring different specifications. The finished software is tested on several of SICS computers, which are listed in the table below along with their respective performance. Since the most interesting processor to investigate is the Tilera64. It has gotten its own subchapter with more detailed specifications.

<table>
<thead>
<tr>
<th>Host</th>
<th>Processor</th>
<th>Clock frequency</th>
<th>L1 Caches</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sony</td>
<td>Intel core 2 duo</td>
<td>2x2.26 GHz</td>
<td>3Mb</td>
<td>4Gb</td>
</tr>
<tr>
<td>Small</td>
<td>Quad core Intel i7</td>
<td>4x2 GHz</td>
<td>3Mb</td>
<td>8Gb</td>
</tr>
<tr>
<td>Fet</td>
<td>Dual Quad core Opteron</td>
<td>8x2.3 Ghz</td>
<td>512kb</td>
<td>16Gb</td>
</tr>
</tbody>
</table>

Table 2 – Specifications of multicore processors that have performed UTSE

7.1.1 The Tilera 64 Processor

[THIS IS CURRENTLY LEFT OUT DUE TO TECHNICAL DIFFICULTIES]

This processor was big news in 2009 as it was the first processor to break the barrier of 64 (8x8) identical processor cores interconnected with an on-chip network called iMesh. This means that every tile has the potential to run an entire operating system or multiple tiles can be grouped together and run operating systems like SMP Linux. Tiles can also be grouped together like this and run single applications. (7)
7.2 The Results

The Benchmark tested is UTSE as specified in 3.3 and the results between sequential and parallel execution has been compared. The relative speedup is measured and compared the C–version of the benchmark. The relative number under parallel and sequential execution is the amount of seconds passed from starting the benchmark until it finishes. In the graphical representation of the tables the performance of each core is considered to be approximately the same to simplify the graphs. All of the benchmarks are executed with the settings for a tree called T3 which is specified in the original paper about the UTS benchmark (1).

7.2.1 Optimal number of C-nodes

To try and decide how many multiple C-nodes should be attached to a Virtual machine a series of test runs was performed. Each run had a different number of C-nodes attached to the Virtual Machine to decide how many nodes that needs to be attached in contrast to number of cores. A possible bottleneck was seemingly eliminated as can be seen in Figure 11. A sufficient amount of C-nodes needs a minimum of \( \frac{3}{8} \) of the maximum number of usable cores for the parallel access to not be limited by the nodes other than by the asynchronous message passing. In the following results that number of C-nodes that are used in contrast to the number of cores on that system. sony gets one C-node, the Small gets two and the fet gets four. What cannot be read from the graph but that was shown in the tests was that it doesn’t affect the performance much at all if you add more nodes than is required to a system.

![Figure 11 - The performance of FET using 7 schedulers and N number of C-nodes](image)

Unbalanced Tree Search: A portable benchmark for Erlang/OTP
7.2.2 One scheduler per core

Launching the EVM without any special flags other than those needed to communicate between the C-node and EVM will make the virtual machine create one scheduler per available core in the system. Also several tests has led to the conclusion that approximately three cores can be handled by one single C-node since requests does not come that regularly. So each system that has 3 or more cores has one C-node attached every three cores. Sequential always run with one single C-node.

<table>
<thead>
<tr>
<th>System</th>
<th>Parallel</th>
<th>Sequential</th>
<th>performance</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fet</td>
<td>225</td>
<td>397</td>
<td>8 x 2.3 Ghz</td>
<td>2,00</td>
</tr>
<tr>
<td>Sma1</td>
<td>106</td>
<td>384</td>
<td>4 x 2.0 Ghz</td>
<td>3,62</td>
</tr>
<tr>
<td>Sony</td>
<td>212</td>
<td>662</td>
<td>2 x 2.0 Ghz</td>
<td>2,75</td>
</tr>
</tbody>
</table>

Table 3 - Benchmark results based on one scheduler per core

The tendency seems to be that the speedup is a lot greater on Sma1 even though it runs on fewer cores. It remains to be discovered whether this depends on the probable bottleneck of the C-nodes for the other systems or if it depends on the number of context switches being performed on Fet.

7.2.3 One scheduler

Launching the EVM with the flag +S N, where N is a number, tells the system that it can use N different schedulers and thereby uses N cores when scheduling processes to execute instructions. For the benchmark results in Table 4 have been run with only one scheduler. That means that EVM will only be able to schedule Erlang processes on one of these many cores this should make the sequential software run smoother but give a penalty to the parallel execution which should run a little slower than the sequential version. It shows from the figure however that parallel execution is not penalized by the occasion of one scheduler. It becomes quite clear that the parallel execution is still running a lot faster with only one scheduler than several. This is discussed in section 0. Each EVM has only had one C-node attached to them in these results since the EVM is running “Sequentially” on one core, it will need no more than one.

<table>
<thead>
<tr>
<th>System</th>
<th>Parallel</th>
<th>Sequential</th>
<th>performance</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fet</td>
<td>122</td>
<td>397</td>
<td>8 x 2.3 Ghz</td>
<td>3,25</td>
</tr>
<tr>
<td>Sma1</td>
<td>147</td>
<td>372</td>
<td>4 x 2.0 Ghz</td>
<td>2,53</td>
</tr>
<tr>
<td>Sony</td>
<td>212</td>
<td>662</td>
<td>2 x 2.2 Ghz</td>
<td>3,12</td>
</tr>
</tbody>
</table>

Table 4 - Benchmark results based on one scheduler

It can be stated however that something is limiting the parallel execution but it is currently unclear as to what it might be. It can be narrowed down to a few factors such as: process overhead, the C-node becomes a bottleneck in processors with many cores, the message passing between processes on multiple cores might be slowing down execution because of the high amount. It might also be that the C-node is scheduled by the OS to a different core than the Erlang processes.
7.2.4 Graphical results

These results are the same as in 7.2.1 and 7.2.3 but put into a graph with each other to be more easily compared. Figure 12 shows the performance of each benchmark as it was limited by the +S flag of the EVM. Figure 13 shows the relative speedup compared to the sequential UTS run on the same system.

![Figure 12 - Performance statistics of systems having run UTSE](image)

![Figure 13 - Relative speedup measured against the sequential version](image)
8 Discussion

This section reasons about the results in the previous section so it is quite important to have read them and reasoned about them first. Otherwise this part might become hard to digest because it is will focus on reasoning around the results and the structure of UTSE. The benchmark has quite a few possible bottlenecks but as there are multiple possibilities that the speedup might depend on. It is however hard to point out any particular one.

8.1 Bottlenecks

As seen in Figure 12 there is no difference at all between the two blue sony-columns meaning that the benchmark performed just as well with one scheduler as with two. Normally this should not be the case if everything had run within the EVM. But because of the limitations in section 3.4 the communication with the RNG library goes through the detached C-node. In the case of the sony it is a dual core processor which means that no difference is achieved if Erlang is scheduled on one processor or two. The same performance is clearly made in both cases.

In the case of sequential execution the C-node normally gets scheduled on the same core as UTSE is running on, this gives quite good performance actually. When it comes to parallel execution the overhead of multiple C-nodes seems to cause the speedup to deteriorate steadily.

The figure is actually quite misleading because when only one core is used that means that the virtual machine is only using one scheduler which is only scheduling Erlang processes on one lonely core. But in reality it is using two cores since the C-node gets scheduled to its own core by the OS when executing UTSE in parallel. This is important to keep in mind when looking at these statistics.

One fascinating thing with Figure 13 is that Smal1 has the overall greatest relative speedup. This is quite strange since smal1 has the least clock frequency of its single cores. It is however one standalone Quad core which means that it shares all resources internally in one core, in difference to the Fet system which has Dual Quad cores which does not share as equally since its two large quad cores and not a unified system as smal1. One theory is that since they are two separate processors the message passing between the C-node and EVM cross-processor becomes a bottleneck on its own. Along with multiple context-switches where the OS decides that one C-node should block while the EVM is waiting for a message from it.

8.2 Scalability

As far as the test results go it seems that the benchmark does not scale very well as the number of cores go up, the speedup does not follow, it decreases. This could depend on a combination of the overhead of processes along with an increasing number of needed C-nodes that gets scheduled by the operating system possibly taking performance from the EVM by causing many context switches.

From what we can tell from Figure 13 the Dual quad core scales terribly with a speedup of merely factor 2 with 8 schedulers and factor 3, 25 with one scheduler. It is also a possibility that the schedulers are too busy scheduling the processes among the eight processors that the overhead cost of those context-switches and jumps back and forth between cores without really doing any computation.

It is hard to say if the Erlang interface in C limits the execution so much that it limits the scaling but it is a possibility that the problem lies solely within the scheduling. This can be said since the Sony in Figure 13 gets a speedup of approximately factor 3. This is quite strange since it only has two cores. It could be explained with that there are only one scheduler used in all the major speedups excluding smal1 which gets the best results at two schedulers. This allows for one to speculate whether the internal OS scheduler picks fights for the cores with the EVM since no problems occur when using only one.
9 Conclusions

This section presents some of the conclusions and speculations that can be drawn from the results and discussion of this thesis. Conclusions are drawn from results and discussion in chapters 7 and 0. It’s hard to draw any definite conclusions without being speculative. Some of these conclusions, especially those about bottlenecks and performance, should thereby be considered as speculations.

9.1 Some answers

Did UTS scale well as the number of cores increased with the Erlang Virtual Machine? Considering the results in previous chapters one it cannot be concluded that Erlang scales poorly in such a message passing intensive application. Since the software is forced to communicate internally there is so much overhead and waiting over the cores that limits the use of the processor cores. This might depend on the schedulers fighting over the cores or the OS scheduling C-nodes poorly. It can be concluded, however, that the speedup does get a lot worse as the number of cores increase. This means that the system performance does not scale as the amount of cores increase which it was hoped to do originally.

Does an 8-core processor perform eight times better at UTS than a singlecore of the same clock frequency? Certainly not, it is limited by several factors previously presented in chapter 8.2.

9.2 Speculations

Does the Erlang interface in C scale well when using multiple cores? Having tested it out it never hits the limits of the processor capacity possibly due to the constant internal messaging. It goes up to a maximum of 50% of performance on the systems that it was run on. After that another C-node has to be attached to make the EVM perform well around 90-95% using one or two schedulers.

Process overhead however does not seem to be a problem in Erlang which can have the explanation in that Erlang has such ligh-weight processes in general so that it never becomes a problem that each process has too little work to do. This is only a speculation and should be properly tested by extending UTSE to be able to handle heavier workloads.

9.3 Recommendations

By the basis of what has been stated in this report it be recommended to use UTSE for scientific purposes as long as one takes it’s quirks into account. it is a great measurement tool to evaluate the relative concurrent power of a strength of EVM.

Erlang as a programming language for concurrent applications is definitely to be recommended since the concurrent part of the programming, message passing and the synchronization techniques makes it well worthwhile to consider in the future as applications will need to be more parallel than ever before to perform well on multicore processors. This report shows that it is really hard to make use of all of the cores even in the most concurrent of applications such as the UTSE benchmark.

It is to prefer that the Erlang interface to C is not to be used for extremely work-heavy applications needing a lot of accesses concurrently unless they can be parallelized somehow using a parallelization library in C like the ones mentioned in section 4.2.1.
10 Future Work

Having finished this project there are of course a few things that would be interesting to continue working on. The following suggestions are far from finished ideas but could well solve some of the fundamental problems that UTSE currently faces.

- It would be interesting to compare the EVM to the OpenMP variant of UTS.
- The C-node might be possible to parallelize into handling several threads at once instead of one separate process. This might speed it up and possibly replace the somewhat messy procedure of launching a separate screen or terminal per process.
- It would be interesting to actually port the random number generator to Erlang since that would leave out the message passing between the virtual C-node and EVM as long as the RNG performs the exact same operations.
- One idea is to make or force the C-nodes to launch internally from the EVM to make it easier to start. This was attempted during the project but no success was reached to make the processes launch within the EVM.
- A Manual needs to be written for UTSE if it is to go public on sourceforge and as it looks today that has not been created. Permission is also currently pending from the authors.
11 References


Acknowledgements

This has been a very interesting project to be working on, probably more so since these are such interesting times for multicore technology.

I would like to thank professor Mats Brorsson for the generous support and allowing me to work with the Tilera64 chip, and various other systems freely at SICS in Kista. I would like to thank the other researchers and thesis workers at SICS for the feedback I have gotten in the process. It has been really valuable to the success of the project to have worked in such a creative environment.