Distributive and parallel problem solving - methods, data structures and algorithms

Antal Tátrai

Theses of PhD Dissertation
Department of Computer Algebra
Eötvös Loránd University

Supervisor: Dr. Attila Kovács PhD
PhD School of Informatics
Dr. András Benczúr D.Sc.
PhD Program of Basics and methodology of informatics
Dr. János Demetrovics
member of the Hungarian Academy of Sciences

2012
Introduction

Nowadays the multiprocessor computers become more and more conventional. Computers with hundreds or thousands of processors are common in the industry, and probably your desktop PC at home has more than one processor as well. On the other hand, there are several problems having only sequential solutions, which often do not satisfy the needed performance issues. One of the main challenge in the future’s computer science is developing efficient and safe parallel algorithms instead of the sequential ones.

In my thesis I analized and solved three difficult problems having no or only sequential solutions earlier.

1.1 Concurrent start-up of Erlang systems

A distributed system is a collection of processors that do not share memory or a clock. Each processor has its own local memory. The processors in the system are connected through a communication network. Communication takes place via messages. Erlang is a distributed programming language and it was designed by the telecommunication company Ericsson to support fault-tolerant systems running in soft real-time mode. Programs in Erlang consist of functions stored in modules. Functions can execute concurrently in lightweight processes, and communicate with each other through asynchronous message passing. The creation and deletion of processes require little memory and computation time. Erlang is an open source development system having a distributed kernel. Erlang is most often used together with the library called Open Telecom Platform (OTP). OTP consists of a development system platform for building, and a control system platform for running telecommunication applications. It has a set of design principles (behaviours), which together with middleware applications yield building blocks for scalable robust real time systems. The basic concept in Erlang/OTP is the supervision tree. This is a process structuring model based on the idea of workers and supervisors. The processes of an Erlang/OTP application can be structured in a (process) tree. Each inner node is a supervisor, and each leaf is a worker. The worker nodes perform the functionality of the application. The supervisor nodes monitor their children and can make decisions
in case of failures. This process structure is very helpful in case of high performance, fault-tolerant systems.

The start-up of these systems is a sequential process, because the nodes of the supervision tree communicate with each other. Supervisor nodes wait for acknowledge (ACK) messages from its children and starts its next child just after received the ACK message from the previous child. Consequently a supervisor node starts its children in left to right order one after the other.

The sequential start determines an order between the starting processes which will be vanished by the parallelization. Therefore an alternative parallel ordering is required for avoiding dead-locks and startup crashes.

The challenge:

For the solution of the concurrent Erlang systems’ start-up the following items must hold:

- The supervision tree structure, as well as other functionalities, must be preserved.
- The start-up must be reliable, and fast (faster than sequential).
- Only ”small” modifications are permitted in the Erlang/OTP stdlib.

1.2 Cache cleaning

Modern computer systems often use caches for reaching slowly accessible data much faster. One of the most typical example is the Domain Name Service. Usually, the response for a query arrives from a nearer server instead of the original name server of the target IP address, and the data is stored in the cache of the nearer server.

Because these data are just a copy of the original, it has to recheck the validity periodically. Usually each data that is saved into a cache has a special field called Time To Live (TTL). This is a time value and means the validity duration of the data. When an element expired it must drop out from the cache or get it again. Often this functionality is implemented with a special thread which starts periodically and checks the elements in the cache.

A cache is an abstract data type which can be implemented in several ways. In most cases trees and hash tables are the base of this data type. I focused on the linked hash table implementations. In the parallel case a large number of threads try to access the cache, and mutual exclusion become the most important aspect of the implementations. Usually, the cleaning thread has to read the whole data structure. Clearly knowing which
elements are expired the algorithms work faster.

The Challenge:

• Develop data structures on which the parallel cache cleaning algorithms work faster.

1.3 Classifications of $\mathbb{Z}^n$ through generalized number systems

Let $M \in \mathbb{Z}^{n \times n}$ be the base, and $0 \in D \subseteq \mathbb{Z}^n$ be a finite set of digits. The $(M, D)$ pair is called a *generalized number system* (GNS) if for each $x \in \mathbb{Z}^n$ there is a $k \in \mathbb{N}$ such that $x = \sum_{i=0}^{k} M^i d_i$, where $d_i \in D$. There is some necessary conditions for being the $(M, D)$ pair a GNS, namely $M$ must be expansive, $D$ must be a complete residue system modulo $M$, and $|M - I| \neq \pm 1$.

Let $\varphi : \mathbb{Z}^n \to \mathbb{Z}^n$, $\varphi(x) = M^{-1}(x - d)$ for the unique $d \in D$ satisfying $x \equiv d \pmod{M}$. The repeated application of $\varphi$ for an $x$ is called the orbit of $x$. A point $x$ is called periodic if $\varphi^k(x) = x$ for a $k > 0$. The orbit of a periodic point is called a cycle.

Since $M$ is expansive therefore there exists a norm in $\mathbb{Z}^n$ such that for the corresponding operator norm $\|M^{-1}\| < 1$. Let $S_k = \{x \in \mathbb{Z}^n | \|x\| < k\}$ for a $k \in \mathbb{R}$. Since $D$ is finite therefore there exists a $C \in \mathbb{R}$ such that the orbit of any $x \in \mathbb{Z}^n$ eventually enters into $S_C$ and remains there. Now the number system property can be reformulated as follows. The $(M, D)$ pair is a GNS iff the orbit of every $x \in \mathbb{Z}^n$ reaches 0. The decision problem means determining the number system property and the classification problem means computing all the disjoint cycles.

There are sequential algorithms for the decision and for the classification problem. Unfortunately, when some of the eigenvalues of $M$ are close to one then the finite set $S_C$ can be extremely large. This is the case in the generalized binary number systems (i.e. when $\det M = \pm 2$) in higher dimensions.

The Challenge:

• Develop a parallel algorithm for the decision and classification problem.
Theses

2.1 Concurrent start-up of Erlang systems

Thesis 1

I designed and implemented a method for concurrent starting of supervisors’ children without destroying the supervision trees. This solution does not need significant changes in the Erlang/OTP stdlib and provides a reliable dependency description between processes.

The dependency graph construction managing the starting order bears a resemblance to the mechanism used by Apple’s MacOSX StartupItems [5]. Information about Erlang and OTP can be found in [6,7,8,9,22].

In order to preserve the supervisor tree structure I introduced the notion of conditions. At the beginning of the start-up all conditions are false. The condition from that a process’ start depends on, is called precondition. A process can start after all its preconditions became true. When a process started up it sets its condition to true. We can represent these relations in a dependency graph.

Next, I proposed an Erlang trick, that enables starting processes in a concurrent way. Here we can set which nodes should start concurrently. When a supervisor process $s$ wants to start a child process $w_1$ the system starts a dummy (or wrapper) node $s_1$ instead.

![Figure 2.1: Inserting a dummy supervisor node for (1) preserving the supervisor’s restarting behaviour, and (2) enabling fast parallel start-up. Supervisors are denoted by squares, permanent processes by continuous border, and temporary processes by dashed border. In the middle of the tree one can see the living processes ($s_2,w_2$) after termination of the dummy functions.](image)
Then, the dummy process \( s_1 \) starts a simple function \( s_{1,f} \) (which just calls a \texttt{spawn} function) and sends an ACK message back immediately to its parent \( s \). Consequently the next child \( w_2 \) of the supervisor node \( s \) can start. Function \( s_{1,f} \) spawned function \( f \). The spawned function \( f \) starts process \( w_1 \) and attaches it into the dummy process \( s_1 \). The already started dummy process \( s_1 \) runs independently (parallel) from the other parts of the system. The advantage of this method is that if process \( w_1 \) has a blocking precondition then only \( w_1 \) will wait and not the whole system. Figure 2.1 shows the described supervision hierarchy after start.

After implementing the above mentioned mechanism in Erlang I created some example program for measuring the performance. Each worker process makes slow floating point calculations in its \texttt{init} section. My prototype is pessimistic in the sense that the telecommunication servers usually wait for a connection or a hardware, but waiting does not load the processors. Figure 2.2 shows the measured speed up.

### 2.2 Cache cleaning

**Thesis 2**

I developed and implemented secondary data structures to the hash table. These data structures hold only pointers and the expire time of the elements. The clearer thread has to examine only the secondary structure which results a faster availability of the expired elements.

In the eightees Carla Sch. Ellis described lockbased hash tables [11, 12] which make better performance than the simple locking. In 2002 a compare-and-set based hash table was introduced by Maged M. Michael [13, 14, 15, 16, 19, 20], that showed equal performance with Ellis hash table if the number of the threads is equal the number of the processors but was much faster if there are more threads than processors. A serious disadvantage of this hash table was that the backbone of the hash table was unresizable, and it produced
long buckets. Somewhat later, a high-performance resizable lock-free hash table was described by Ori Shalev and Nir Shavit [21]. During the implementation of my solution I used the Ellis locking mechanism showing a general solution to the problem of cache cleaning.

Several secondary structures was implemented. First, the well-known binary heap. The disadvantage of this structure is that (in contrast to Dijkstra algorithm) the cleaning usually will erase more than one elements. Erasing the elements from the cache implies an $O(\log(n))$ computational time reordering of the secondary structure for each item. Moreover, the insertion costs the same overhead. I tried and get similar results in case of search trees.

Finally, I developed a special data structure, called bucket-list. (Figure 2.3 shows this data structure.) The base idea behind this structure is time slicing. The time is devided into parts containing cleaning periods. The parts in which cache elements expire are stored in a linked list. Each element of the list (bucket) represents a time slice and contains pointers to the items of the hash table which will expire in this time slice. In each execution of the cleaning thread only a whole bucket will be erased. The list contains the time slices in linear order. Consequently the cleaning thread finds the proper time slice at the end of the list in $O(1)$ computation time. In worst case, the insertion takes $O(ts)$ time, where $ts$ is the number of time slices in the list. However if we assume that TTL values are constants, then the insertion has $O(1)$ computational time. This restriction is usually held in case of the DNS.

I implemented the Ellis hash table with a cleaning thread, and with the three secondary structures above. In case of the bucket-list I found speed up only if the time slices were very sort (1 second).
2.3 Classifications of $\mathbb{Z}^n$ through generalized number systems

Thesis 3.1

*I developed and implemented two grid solutions for the classification problem using master-slave grid technology.*

It is known that all the periodic elements in the system $(M, D)$ are inside the compact set $-H$, where the set $H$ contains the elements with zero integer parts in the systems (the fractions). There is a method by which set $H$ can be enclosed in an $n$-dimensional rectangle [17]. The earlier (sequential) algorithm decides the periodicity of points of this rectangle by analyzing the points’ orbits. My simple grid solution enumerates the points of the rectangle and cuts this sequence into slices. The server application sends these slices to the worker machines. Clearly, if during the computation of the function $\varphi$ the orbit arrives at an already examined point then no more iteration needed. My first type of enumeration creates slices in a back-tracking order. In this case all workers receive the start point and the size of a slice.

When the orbit of a point returns to a point which occurs earlier in the enumeration than the slice’s first point then further iteration of $\varphi$ is unnecessary, consequently the algorithm can start with the next point. This traversing of the rectangle is similar to a box (in 3 dimension), in that we fill water. In general, function $\varphi$ is tending to zero and the water is in the bottom of the box therefore the already checked points are mostly far from the orbits.

It is possible to improve the previous algorithm with a different rectangle division. The division contains small rectangles (called bricks). One worker computes bricks one after the other. The first brick encloses 0. Bricks, which are being computed or have been computed form a prism if they constitute an $n$-dimensional rectangle. The master generates the bricks in spiral order. A worker gets a brick and a prism at a time. It checks all points of the brick and stops an iteration if it arrives into the prism. In this way the program checks the points nearer to 0 first. The advantage of this method is that during the computation of the orbits the iterations come to points near to 0 more often. Consequently, these points will be accessed essentially only once. Figure 2.4 shows the factors of the running time of the GRID algorithms.

Thesis 3.3

*I created parallel solution for the Brunotte algorithm that were optimized to two different*
The Brunotte algorithm [10, 18] also applies function $\varphi$ to locate a nonzero periodic point or to conclude that the system is a GNS. But in contrast of the previously mentioned solutions in this case the algorithm starts apply the function $\varphi$ on a small set that is growing in each iteration. This growing is finite. If the set finished the growing and there is no other periodic elements only 0 then the $(M, D)$ pair is GNS.

I produced two parallel solution for this problem. The first version runs on computers that have multicore chips and shared memory (SMP platform) and the second version is optimized to the Cell Broadband Engine Architecture. The two versions were compared and this shows that the Brunotte algorithm runs faster on SMP platforms. Figure 2.5 shows the running time of the two parallel Brunotte algorithms.

![Figure 2.4: The factors of the running time in case of some small dimension cases](image)

![Figure 2.5: The running time of the SMP and the CBEA optimized Brunotte algorithms](image)
Publication of the author


Bibliography

[5] Mac OS X Startup Items


