Implementing Exhaustive Search for the Coil-in-the-box Problem using MPI

Marcel Gehrke, Jan Winkelmann

Hamburg University of Technology
Institute for Software Systems

***Abstract*—This paper addresses parallelization of the exhaustive search for solving the coil-in-the-box problem. Since exhaustively solving the problem takes roughly 20 days, compairing different techniques to prune the search tree is unrealistic. However, every subtree of the search tree can be treated individually, an ideal setup for parallelization techniques. We present a simple client server architecture using MPI and analyze experimental data with respect to overhead; including work towards finding start parameters that minimize overhead.**

I. Introduction

This paper deals with the coil-in-the-box problem and how to parallelize it. To find a coil-in-the-box, we need to find the longest induced cycle in a hypercube. An induced cycle is a simple cycle, in which only adjacent vertices in the sequence are connected by an edge.

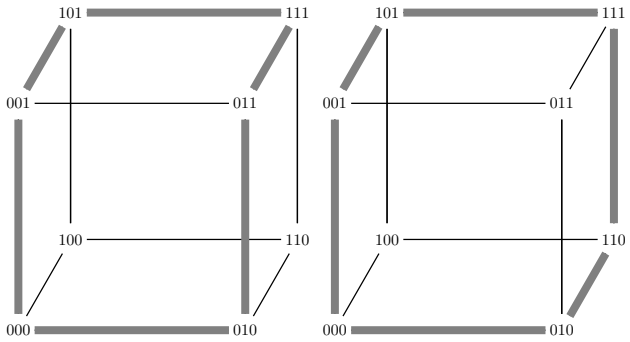

Fig. 1. One illegal coil (left) and one maximal coil (right) for a 3D hypercube

Traversing from 011 to 111, we have to mark 011 and 001 as unusable. Therefore, we can not, as shown in Fig. 1 on the left example, traverse from 101 to 000 over 001, but would have to go over 100.

A simple way to search for a cycle through the hypercube, we have to follow a few steps: when visiting a new vertex in the hypercube, we mark the previous vertex and its neighbors as unusable. The cycle through the hypercube is complete when reach the point of origin again. In order to be able to reach the origin again, we do not mark any vertex during the first traverse. There is also a snake-in-the-box, which is the longest induced path in a hypercube.

However, to find the longest induced cycle, all possible solutions have to be considered. Therefore, we need to exhaustively search the hypercube. The problem has been widely studied and is assumed to be NP-Hard [1].

Due to the fact that we need to exhaustively search the hypercube, the search space grows exponentially with the number of dimensions. Unfortunately, it is not possible to consider results of the 8th dimension, since it is computationally infeasible to calculate the coil-in-the-box problem for it [1]. Moreover, the computation for a 7 dimensional hypercube should take roughly 470 hours, based on our experiments.

Notwithstanding, to be able to compare different improvements on the algorithm, data up to the 7th dimension should be considered. Parallelizing the traverse of the search tree reduces the waiting time for the results. In order to parallelize it we present an implementation using MPI.

The paper consists of the following parts: In section 2, we explain the parallel process and MPI. We present the parallel architecture of our implementation in section 3. Finally, we conduct an analysis of overhead in section 4.

II. Parallel Processing

This section gives a short introduction on the problems of parallelizing the coil-in-the-box problem and the parallelization method used. First, we mention the difficulties of parallelizing the coil-in-the-box problem, followed by an introduction into MPI and its basic communications. Third, we discuss the specific types of MPI functions used in the implementation.

A. Challenges in the context of coils

The obvious approach to parallelizing exhaustive search problems is to delegate subtrees. We will call these chunks of the search space *subproblems* from now on.

The delegation of subtrees yields a good approach for parallelization since there is no computational overlap or dependency between nodes. Unfortunately, there is no known way to estimate the work needed for each subproblem. This difference between subproblems makes balancing workload difficult. It is, in fact, the most challenging part of the parallel architecture design.

Our implementation is based on the passing of subtrees, since exhaustive search divides into subproblems, which parallelize very well. The uneven workload distribution between subproblems calls for various measures to reduce workload imbalance.

B. MPI Primitives

Since we need minimal sharing between the copies of the program, message passing was an easy choice as a parallelization technique. And more specifically we chose *Message Passing Interface*, MPI for short. Another advantage to MPI is its high deployment rate on cluster systems. MPI implements advanced functionality including broadcast sends, grouping of nodes, and scatter-reduce. However, our simple approach does not use this functionality.

MPI starts one copy of the program per requested core. The copies pass messages through the interface provided by MPI.

The basic MPI functions allow one copy to communicate with exactly one other instance. The `send` and `recv` functions, allow to send and receive data from other instances.

Two relevant attributes to these basic modes of communication are synchronicity and whether they block or not. Blocking communications do not return until the communication has finished on both sides. Conversely non-blocking communications return immediately. Synchronicity of communication affects only send functions and guaranty that the matching receive also completed.

C. Synchronous, non-blocking communication

The implementation uses synchronous, non-blocking communication only.

Synchronous, non-blocking communication allows the sending of more than one message at a time and to ensure a message has arrived, which is desirable, especially in a client-server environment. Additionally, non-blocking receives make it possible to check for the arrival for more than one message at a time. The implementation employs non-blocking receives to monitor for delegated subproblems as well as the termination signal.

III. Parallel Architecture

A. Implementation Details

We decided that a simple client server architecture would be the best fit for us, since it is easy to implement, and allows for clear separation of computation and parallelization tasks. In our implementation, a simple server only delegates subproblems to the clients who solve them.

Both server and client are written in C++. A typical run has three phases: a *build-up*, a *computation* phase, and the *tear-down* phase.

During program start up, in the *build-up* phase, only the server works, while the clients are all idle. The server starts by traversing the search tree up to a certain depth, which we refer to as *initial depth*. While generating the subproblems, the server can already find some solutions that will then be written to an output file.

Then, the server sends out one subproblem to each client. Afterwards, the server has to *remove* the problem to make sure no problem gets sent out twice.

After the *build-up* phase, work is mainly on the client side, while the server delegates new subproblems on demand. Upon reception of a new subproblem, a client starts to calculate all possible solutions and writes the results to an output file. Having completed one subproblem the client immediately requests a new subproblem.

After a client requests the last subproblem, the server initiates termination in the *tear-down* phase.

Algorithm 1 Server Control Flow

```
P ← calculate subproblems until initial depth
send out a subproblem to each of the n clients
remove n subproblems from P
repeat
    wait for a request
    send out a subproblem
    remove subproblem from P
until P is empty
set up a termination message for every client
wait until every client received termination message
```

In the *build-up* phase, the client enters a loop where it waits for server delegations. Nonetheless, during the *build-up* phase, the client has to wait for the server to generates the subproblems. As soon as the client receives a subproblem, it enters the *computational* phase and calculates all possible solutions. Then the loop starts again and is repeated until the client receives the *termination message*. On receipt of the *termination message*, in the *tear-down* phase, the client leaves the loop and terminates.

Algorithm 2 Client Control Flow

```
loop
    receive from server
    if subproblem then
        calculate all possible solutions of subproblem
        writes all possible solutions to an output file
    else if termination message then
        terminate
    end if
end loop
```

B. Termination

After the last subproblem has been requested by a client and delegated to it by the server, the *tear-down* phase begins and the server sets up a *termination message* for every client. However, a client will only receive the message once it has finished its last subproblem. Receiving such a *termination message* will force the client to leave its normal working loop and to terminate. In order to be sure the termination process succeeded, the server waits until every client received the *termination message*.

To demonstrate the problem of balancing the workload between clients, consider the following example:

The server has one subproblem left and all clients are nearly done calculating their subproblem. In the worst case, to compute the last subproblem takes the longest. Now one

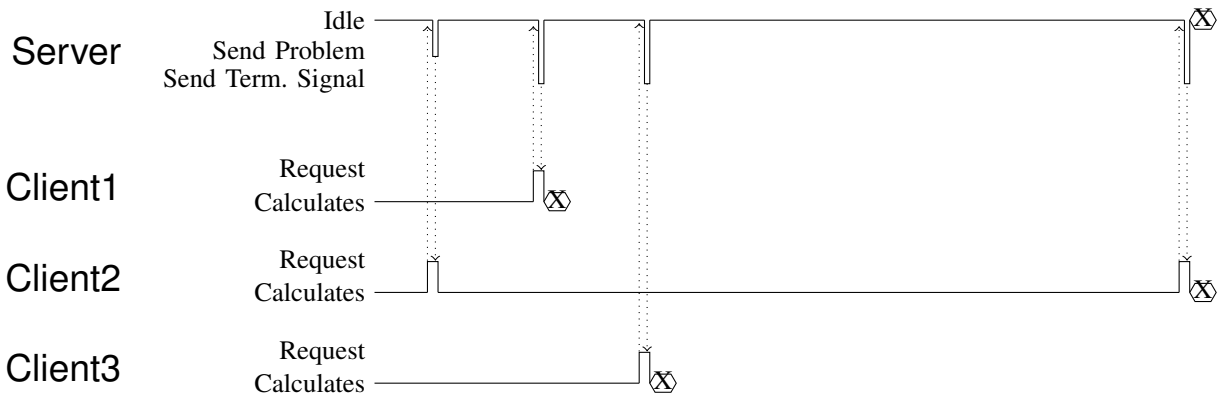

Fig. 2. Termination

client is occupied with the last subproblem while all the other clients have already terminated. Ideally, the workload would be split, so that all clients terminate simultaneously.

IV. Evaluation

This section analyzes the behavior of the implementation, in order to evaluate its scalability and efficiency. In particular, the parallelization technique. We will need the following terms throughout this section:

Wall-time is the difference between the Unix time at start of the program and at its end for a single core.

Core hours is wall-time multiplied with the number of cores.

Computation time is whenever a client works on a subproblem. The amount of computation time for the entire exhaustive search is constant throughout runs. For our set of runs it is at about 470 hours.

The *size* of a subproblem refers to the amount of computation time needed to exhaustively search it.

First, we will discuss some specifics on gathering data. Second, is a discussion on the adjustable parameters of the algorithm, followed by the presentation of the results. Last is the analysis of the results, which includes works towards minimizing overhead.

A. Gathering Data

Gathering data for the evaluation requires running the program on a cluster and collecting profiling data. However, our implementation uses non-blocking sends on the server side and non-blocking receives for the client, only. These facts render conventional profiling libraries mostly useless.

The primary source for statistics is the accounting report provided by the task scheduler used on the cluster. The report includes the wall-time and the core hours of the run, but unfortunately it reports the maximum wall-time and not individually per client.

Additionally, the implementation itself measures the computation time per subproblem using the MPI `Wtime()` function.

For the data gathered during these runs see Subsection C.

B. Adjustable parameters

The two main parameters that impact performance are the depth up to which the server pre-calculates the tree before delegating them to clients, called *initial depth*, and the *number of cores* running the program in parallel.

The initial depth changes number of subproblems that the server delegates and the size of the subproblems, which directly determines the computation time necessary per subproblem.

The number of cores impacts the run-time of the program. Since the implementation is a client server architecture the number of cores should be at least three, and certainly no more than the number of subproblems. Ideally, the execution time would scale linearly with the number of cores used. Unfortunately this is not the case, since the parallelization is not perfect, especially in a client-server architecture.

C. Experiment Results

It stands to reason, that some combinations of the initial depth and the number of cores are more efficient than others. We want to get a value for the initial depth that causes the least overhead for a small range of cores, in this case 8-64 cores. To achieve this goal we did two sets of experiments. The first experiments investigates the initial depth, and the second one the number of cores.

1) Initial depth experiments: We first investigated the distribution of sizes for given initial depths, arbitrarily choosing an initial depth of 8 for our experiments and compare that with one level deeper and shallower.

Consider Table I, which shows properties of the subproblems delegated at the initial depths of 7 to 9. The data gathered on a certain initial depth refers to the properties of the exhaustively searched subproblems at that level. Indicated on the table are the number of subproblems at this level and the wall-time necessary to execute the implementation at that level and with 32 cores. Additionally, the minimum, maximum and average computation time per subproblem.

2) Number of cores experiments: Table II is the data set for runs with a varying number of cores and the chosen initial

TABLE I
TABLE WITH PROPERTIES OF THE COIL-IN-THE-BOX SUBPROBLEM AT DIFFERENT INITIAL DEPTHS

Initial Depth	# Subproblems	Wall-time (h)	min. Time (h)	avg. Time (h)	max. Time (h)
7	131	–	1.6	3.5	9.4
8	483	18.1	0.2	1.0	3.94
9	2043	15.9	0.03	0.2	1.1

TABLE II
TABLE OF EXPERIMENT DATA AT INITIAL DEPTH 8; ALL TIMES IN HOURS

Cores	Wall-time	Core Hours	Overhead	Overhead per Client
8	67.2	538	68.0	0.1
16	33.7	539	70	1.5
32	18.1	579	110	2.5
64	9.8	627	158	2.1

depth of 8. Provided are values for the wall-time, the core hours, the overhead produced, and adjusted overhead, all of which are in hours. Overhead is calculated by subtracting the ideal computation time, 470 hours, from the core hours. Overhead per client is the total overhead adjusted for the server (by subtracting the wall-time from the core hours) and divided by the number of clients (number of cores -1).

D. Discussion

The first Subsection is the analysis of the initial depth. Second, is the discussion of the number of cores. Lastly, a discussion on finding optimal values for the adjustable parameters.

1) Initial depth: The following section will discuss Table I. The number of subproblems generated per initial depth level increases drastically from 131 at level 7 to 2043 in level 9. This growth is to be expected, since the subproblems are essentially generated by breadth-first search of a tree and the number of nodes per depth of a tree increases exponentially.

The wall-time is also dependent on the initial depth. Increasing the initial depth from 8 to 9 resulted in a speedup of 14% from 18.1 hours to 15.9 hours. We will discuss the influence of the initial depth on overhead later on.

Columns 4 to 6 on Table I refer to the computation time per subproblem. (**This is not obvious from the Table, is that ok?**) The columns indicate that the average, minimum and maximum computation time per subproblems decreases with an increase in the initial depth. From a constant amount of computation time for the entire tree and an increase in subproblems follows the decrease in average computation time per subproblem. Especially note that the maximum computation time per subproblem decreases by a factor of 3 for the levels 7 to 9, since it is of importance for a later section.

From the knowledge gained in the findings and the architecture of the program we can draw some conclusions on the influence of the initial depth on overhead. The implementation has 3 phases, the build-up, the computation and the tear-down phase, all of which are influenced by the initial depth in different ways. During the build-up phase the server need to traverse the search tree up to the initial depth. If the initial depth is higher, the server takes longer. Since the clients are not able to do anything but wait in this phase, a higher initial depth increases overhead in the build-up phase.

Overhead is generated during the computation phase by the passing of messages. Higher values for the initial depth generate more subproblems, and thus more overhead, because more messages require more communication.

As already discussed, the length of the tear-down phase is bound from above by the maximum of the computation times of the subproblems. Therefore a small value for this maximum can decrease the length of the tear-down phase and thus decrease overhead.

2) Number of cores: Table II indicates that wall-time decreases when cores are added in the range of 8 to 64. However, the decrease in wall-time is not strictly linear, from 67.2 hours at 8 cores down to 9.8 hours at 64 cores, which can be explained by a loss of efficiency. The table also shows that overhead increases linearly per additional core running the program. Core hours increase from 538 for 8 cores up to 627 hours for 64 cores.

From the data we can conclude, that increasing the number of cores can significantly decrease the wall-time necessary to run the program. The decrease in wall-time comes at the cost of more overhead. Generally the overhead increases with an increased number of cores. This is due to the fact, that the increase in communication and the wait times inherent in the model do not make up for the better client to server ratio.

There are exceptions to that pattern. Table II shows remarkably small values for 8 and 64 cores, namely 0.1 and 2.1 hours of adjusted overhead. These exception could be caused by lucky breaks in the timing of delegations to clients. If the subproblems are delegated in such a manner, that client wait times are reduced during the tear-down phase, this decreases overhead. Conversely, one of the major sources for the increase in overhead when adding more cores is probably ill-timing during the tear-down phase. Other sources may include wait times in the build-up phase and communication related waiting times during the computation phase.

3) Towards an optimal value for the initial depth: Given the two parameters values that minimize overhead are desirable.

From an evaluation point of view the number of cores is of little significance, since it does not immediately relate to the program and because in real-world usage the number of cores

is limited by external factors. Ideally one would determine values that lead to a global minimum in overhead. This might not be practical since reducing the wall-time might be more desirable than reducing overhead when running a program. More helpful would be a value for initial depth that is nearly optimal for a reasonable number of cores, which in this work is between 8 and 64.

It is certain that the initial depth of 8 is too small to be optimal. Unfortunately, determining the optimum value is not trivial, because increasing the depth decreases the length of the tear-down at the cost of the length of the build-up phase. Determining this optimum depth is beyond the scope of this work, since the only way to determine the optimum value is by experiment. For very large values for the initial depth the communication overhead and the load on the server will become a bigger source of overhead than the workload imbalance. The optimal value is most likely not much bigger than 9.

V. Future Work

To even further improve the execution time needed to exhaustively search the hypercube for the coil-in-the-box, there are a few approaches. One could implement the before-mentioned backwards delegation or split up the last subproblems in a few more subproblems. Focusing on the server, using it to do some work as well, after the pre-calculation of subproblems, is also a possibility. However, the more cores you add the less improvement will be done by the server's improvement.

Another approach would be to use heuristics on the algorithm to calculate the coil-in-the-box for a hypercube. Doing that some of the mentioned improvements on the parallelized architecture might not work anymore, but maybe others will be needed. Therefore, our implementation is a good start, but there might still be some need for improvement, depending on what you want to do with the implementation.

VI. Conclusion

We presented a way to parallelize the coil-in-the-box problem, the simple client server architecture of the implementation and evaluated its behavior. We analyzed overhead, using experimental data, and speculated as to possible sources of the overhead.

Overall our simple approach yields a working parallel implementation. This approach comes with some inherent overhead. But a better performance can be reached if a better value for the initial depth is know. Finding this value, however, requires experimental runs. Since the number of cores is almost always limited by external constraints the initial depth is worth quite some attention when running the coil-in-the-box problem in parallel by delegation of subtrees.

Acknowledgment

Thanks to Professor Schupp for giving us the opportunity to work as a student help in the institute for software systems and to Gustav Munkby for overseeing and helping us at every step.

References

[1] Technical Report TR-CIS-UG-2008-001, Department of Computing and Information Science, University of Guelph, Canada. http://www.cis.uoguelph.ca/department/technical_reports.html, 2008.