**Abstract**

This paper compares two parallelization methods for data flow networks. A strict decoupling between the networks’ structural components and their evaluation is achieved using a pattern oriented architecture with generalized traverser objects. This architecture provides a clean basis for the implementation of parallel execution code using a) an explicit parallelization based on pthreads and b) an implicit parallelization using OpenMP. Both methods are then evaluated and compared to each other for different traversal heuristics.

**Index Terms:** D.1.3 [Software]: Concurrent Programming — Parallel programming

1 Introduction

Modern Central Processing Units (CPUs) no longer gain any performance benefits from increasing clock rates. Downsampling of chip layouts causes physical effects like Quantum tunneling to become a limiting factor. Increasing the operating voltage is not a durable solution since it results in an increased power consumption and heat generation. Hence, for some years now, improvements are more and more gained by utilizing well-known mainframe and special purpose server technologies. Pipelining, multiple cores and Single Instruction Multiple Data (SIMD) vector units are part of modern CPU’s (Central Processing Unit) and GPU’s (Graphics Processing Unit).

Furthermore, the strict distinction between specialized processing units and general purpose units disappears, as can be seen at Intel’s Larrabee project and NVIDIA’s GPGPU strategy. These new technologies are nowadays available on workstations, PCs, and video game consoles. This reduced the investment costs for Virtual Reality (VR) and Augmented Reality (AR) systems. Since performance increase is nowadays largely achieved using parallel hardware concepts instead of higher and higher clock rates, software design has to cope with this parallelization to utilize the available processing power.

Multiple parallelization methods and techniques exist. In general, they can be categorized in terms of the implied software development requirements and impact on the software design process. We call these two approaches implicit and explicit.

Implicit methods try to release developers from the burden of an explicit parallelization. They are often provided by the used programming model, programming language, or compiler itself. Possible solutions range from Message Passing like multi-agent systems (MAS), functional programming [11] as in the Actor-Model [8] of Erlang [2], the Software Transactional Memory approach [9], to compiler extensions like OpenMP [4].

Explicit methods based on Shared State Concurrency offer alternative methods for parallelization. Contrary to implicit methods, explicit ones need a detailed analysis of the used algorithms to prevent access violations and synchronize the different threads.

Semantic models offer one possible solution to reduce complexity on the task as well as on the underlying software systems’ layers [12, 15, 13, 14]. An example of a semantic model for typical data structures found in VR/AR systems can be seen in [16]. The main objective was to use a semantic network with the traverser or visitor pattern to decouple structural from process flow aspects and to utilize this method to represent the internals of a VR/AR system.

To evaluate the parallelization possibilities of the general semantic representation layer, we implemented a data flow network [17, 10, 18, 1] as a special VR/AR concept, similar to Figueroa [7] and Bues [5]. Parallel processing of data flow networks has already been considered in instantreality [3] and DLoVe [6]. Both systems split the data flow network in subgraphs and process them in parallel. DLoVe distributes the subgraphs not only on different threads, but also on different processes and machines.

Based on a strict separation between structure and process we compare the performance of serial and parallel traverser implementations in various test graphs. The parallel implementations compare an explicit pthread based approach to an implicit one using OpenMP. Both approaches fit into in the existing code base and only result in a few changes of the existing traverser implementations. Because data flow networks are a common structure in VR/AR-Tools, central concepts and results can be adapted to them without changing the central paradigm.

In this paper we will first illustrate the concept of structural components of the data flow network followed by a description of three different traverser concepts. Using this background we’ll discuss different techniques to parallelize the processing of data flow networks. The concepts will be evaluated by two different test scenarios and the results will be discussed. We’ll finish with a conclusion.

2 A Modular Pattern-Based Data Flow Network

2.1 Structural components

The basic concept of data flow networks is the directional propagation of values between value holders, called fields. The data flow network presented in this paper composes fields to containers, which are the central anchoring point for user defined functions. A field encapsulates a changeable value with a free selectable data type (Fig. 1).

![Figure 1: A container with three fields holding float values. The sum of the values of input1 and input2 is provided in result.](image)

Every field is always part of exactly one container and has a unique name inside of it. A field can have an optional amount of
receivers where a field can have only one source. The value gets propagated to the receiver if the value of the source and the receiver differs. A container marks itself as changed if a value of a field inside of it gets changed. Container flagged as changed needs to be evaluated by the evaluation method of the container. A evaluation method is defined by the user of the data flow network and describes the behavior of the container by making calculations on fields of the container or providing the results of API (Application Programming Interface) calls like system time or sensor data. This structure represents a directed graph where containers are nodes and the relations between the fields are edges. Loops in the graph are in general allowed, but must be supported by the processing traverser.

2.2 Processing components

Traversers encapsulate different data propagation heuristics. They implement a visitor pattern following a structure/process separation. A traverser moves over the data flow network starting from a given list of start container. It calls the evaluation method of a container and instructs the fields to propagate their values. In the next step the traverser collects as changed flagged parent container of the receiver fields. After all container in the current list are processed the traverser continues on the list of collected target container respectively terminates if no target containers are collected. No targets are collected if no container in the current list have a receiver or no receiver has changed. This fundamental traverser algorithm is described in algorithm 1.

Algorithm 1 simpleTraverser( startcontainers )
1: containers ← startcontainers
2: while |containers| > 0 do
3:   newcontainer ← ∅
4:   for all container in containers do
5:     container.evaluate()
6:   fields ← container.output fields
7:   for all field in fields do
8:     field.propagate()
9:     receiverfields ← field.receiverfields
10:   for all receiverfield in receiverfields do
11:     receivercontainer ← receiverfield.container
12:     if receivercontainer.haschanged then
13:       newcontainer ← newcontainer ∪ receivercontainer
14:     end if
15:   end for
16: end for
17: end while
18: containers ← newcontainer
19: end while

Structural components are implemented in a library which defines an abstract interface for the traversers. The library does not contain a specific traverser implementation which is implemented separately. This separation on the implementation level follows the strict separation between structural and processing components on the conceptual level. This provides the possibility to (ex)change the traverser implementation in already existing applications, e.g., to modify behavior or to tune performance. The library is implemented in C++.

For illustration purposes we’ll not show the fields in diagrams, but depict the graph as nodes and vertices, as can be seen in figure 3. Start containers are displayed as grey boxes.

3 TRAVERSER HEURISTICS

3.1 Simple traverser (ST)

A minimal traverser implementation to process a data flow network of the illustrated structure components matches algorithm 1. It provides multiple evaluation of containers, e.g., applied to the graph in fig. 3, container4 gets evaluated two times. Using this algorithm on graphs with loops results in an infinite loop.

3.2 Pre-sorting traverser (PST)

To prevent multiple evaluations of containers by algorithm 1, it is necessary to arrange the containers in an order where all sources of a container are recorded beforehand. The graph in figure 4 is used to illustrate this heuristic.

Figure 2: Two connected container with connected fields.

Figure 3: A data flow network represented as graph with relations between container without fields.

Figure 4: Example graph for pre-sorting. Container grouped as index assigned by the pre-sorting algorithm.

Pre-sorting is done by moving over the graph and assigning an index to each container. The start containers get the index 0, the target containers of the start containers get the index 1, and so on. If the traverser reaches a container that already has a lower index, the index is overwritten by the (new) higher index. This case occurs if the graph branches at a container and the number of containers between the branch and the merge point have different length (see c2-c6-c10 and c2-c4-c7-c10 in figure 4). The index assignment is described in line 5 - 19 in algorithm 2. This indices are used.
to build a list of a list of containers where containers are grouped by index. These (inner) lists are sorted according to the respective grouping index as can be seen in line 20 - 22 and table 1. The lists are evaluated one by one, beginning at the list with the index 0 (see line 23 - 24).

Algorithm 2 preSortingTraverser( startcontainers )

1: sortedcontainers ← Ø
2: sortmap ← Ø
3: containers ← startcontainers
4: iteration ← 0
5: while |containers| > 0 do
6: newcontainer ← Ø
7: for all container in containers do
8: sortmap[container] ← iteration
9: fields ← container.outputfields
10: for all field in fields do
11: receiverfields ← field.receiverfields
12: for all receiverfield in receiverfields do
13: newcontainer ← newcontainer ∪ receiverfield.container
14: end for
15: end for
16: containers ← newcontainer
17: iteration ← iteration + 1
18: end while
19: for all entry in sortmap do
20: sortedcontainers[entry.index] = sortedcontainers[entry.index] ∪ entry.container
21: end for
22: for i ← 0 to |sortedcontainers| − 1 do
23: list ← sortedcontainers[i]
24: for all container in list do
25: if container.changed then
26: container.evaluate()
27: fields ← container.outputfields
28: for all field in fields do
29: field.propagate()
30: end for
31: end if
32: end for
33: end for
34: end for

The values of all containers in a list are independent from each other. All sources of fields of a container are already registered in a preceding list, hence every container is only evaluated one time. The pre-sorting can lead to increased performance if the overhead for pre-sorting is smaller than the overhead for multiple evaluations. This mainly relies on the structure of the data flow network and the work-load in the evaluation methods. This algorithm isn’t able to process graphs with loops.

3.3 Loop breaking traverser (LBT)

To process data flow networks with loops, algorithm 2 must be extended with a loop search algorithm before line 5. The loop search algorithm looks for edges that need to be broken. Values of fields get propagated over a broken edge but the target container isn’t evaluated. To find the loops, the algorithm recursively moves over the network as can be seen in algorithm 3. The algorithm begins with the start containers and two empty lists. The first empty list is used to save the current path. Edges that need to be broken are saved in the second empty list. The algorithm iterates over the given list of containers. For each container it collects the target container and checks if a target container is already in the path list. If it is, the edge is marked as broken and it is saved in the second list. Otherwise, the container is recorded in a list. After the exclusion of the already visited container, the algorithm puts the current container in the path list and calls itself with the collected target containers. The marked edges are used in the pre-sorting described in section 5.2 to prevent an infinite loop during the index assignment.

4 PARALLELIZATION

4.1 Parallelizable operations

In general, traversers provide two actions as potential parallelization targets: the traversal over the data flow network and the evaluation of the containers. Pre-sorting and loop searching are typical examples of traversal intensive operations which depends on movements between the nodes. The run-time of these operations largely depends on the number of edges. In contrast to the latter, the runtime of the evaluation depends on the number of containers and the specific operations performed during the evaluation method.

A parallelized evaluation of containers can be done on containers that have no data dependency during evaluation time. Both concepts can be combined by splitting the graph in independent sub-graphs and evaluate them in parallel like it’s done in DLoVe and instantreality.

To split the graph into subgraphs it’s necessary to move over the graph to find possible separation targets. For an advantageous payload distribution it’s advisable to collect statistical data about the run-time of the containers’ evaluation and use them for weighting the subgraphs. However, the pre-sorting algorithm in algorithm 2 already provides lists of independent containers. Only few changes are necessary to parallelize the processing of this lists: the evaluation in lines 26 - 32 must be executed in different threads.

After analyzing typical data-flow oriented systems it seems that most of the work is actually done during the evaluation processing phase and less time is spent for the traversals between containers. Hence, using the latter assumption, we expect a higher performance
improvement by parallelizing the evaluation. In addition, the evaluation methods encapsulate the functional behavior of containers and hence are good targets for a parallelization approach based on a functional programming approach which we will investigate later in future work.

The implementation of parallel processing of data independent containers in lists is straightforward given the existing architecture and much easier to implement as a segmentation into subgraphs. It rapidly provides first results about potential performance gains as well as drawbacks. We implemented this concept in three different versions.

4.2 Parallelization without thread reusing (MT1)

The first implementation creates the required threads on demand. Every thread gets a chunk of the current list of containers and evaluates the containers in the chunk. After the processing is completed the thread terminates. The synchronization is done by the join function of the pthread library. This implementation is very simple and doesn’t need much changes of the code base.

4.3 Parallelization with thread reusing (MT2)

The second implementation is using a thread pool. A given number of threads is created and they are initially send to sleep state. On evaluation, a chunk of the current list of containers gets dispatched to each thread which is then started. After the thread has completed the evaluation of the chunk it goes to sleep. The synchronization is done with mutexes and condition variables. This implementation needs more changes than the first one because of the realization of the thread management and a reliable synchronization idiom.

4.4 Parallelization with OpenMP (MT3)

The third implementation uses OpenMP to parallelize the loop over the list of containers in lines 25 - 33 in algorithm 2. The thread management and synchronization is provided by OpenMP. Threads are reused by OpenMP. This implementation needs less changes than the other ones, because the loop is parallelized by using special annotation directives provided by OpenMP.

5 Evaluation

5.1 Test environment

Two test programs are used to compare the different traverser implementations. The first test program builds a data flow network typical for VR/AR applications: a scene animated by interpolators. This test program tests all six traversers. Every traverser processes the network several times. The total time is taken in microseconds after each traversal. The mean value is calculated out of all measured times for a given traverser after processing finishes.

The second test program creates 10 containers with an evaluation method that calculates prime numbers up to a given upper limit. All ten containers are start containers for the traverser. The traverser processes all ten containers several times. The elapsed time of each processing is used to calculate the average time for the overall processing. After the average time for each traverser has been calculated, the upper limit of the prime number calculation is increased and a new measurement cycle is performed for all traversers.

This test program is designed to answer three important questions:

- How much work-load is necessary to justify the overhead of parallelization and synchronization?
- Is there an advantageous ratio between number of cores and number of threads?
- Is it better to set an explicit affinity to a core for each thread or is it better to let the operating system’s scheduler do that job?

The test programs are started with highest priority to prevent side-effects during the measurements. In a second test they are started with normal priority and a second process is causing workload to test how the traversers behave under this non-exclusive situation. The test computer is a Lenovo ThinkPad T61p 6457-7XG with an Intel Core 2 Duo T7800 2.6 GHz CPU and 2 GB RAM. Debian GNU/Linux 5.0 is used as the operating system. The software load to test how the traversers behave under this non-exclusive situation. The test computer is a Lenovo ThinkPad T61p 6457-7XG with an Intel Core 2 Duo T7800 2.6 GHz CPU and 2 GB RAM. Debian GNU/Linux 5.0 is used as the operating system. The software is translated with the gcc 4.3.2 compiler with optimization level -O2.

5.2 Test case data flow network

The test case data flow network animates a scene where the elements of the scene are controlled by interpolator graph nodes. The interpolators are connected to a clock node that delivers the current system time in milliseconds at the output field. The interpolators are calculating a column vector that represents a position. The field with the calculated position is connected to the position field of a scene graph element illustrated in figure 5.

Two different types of interpolators are used for the test case. Both interpolate a value between a given minimum and maximum. One interpolator sets the value back to the minimum when the maximum is reached, the other interpolator reverses the interpolation direction for each interval.

![Figure 5: Test graph with clock node, interpolator nodes, and scene graph element nodes.](image)

5.3 Results

5.3.1 Data flow network

The results of the first test program are noted in table 2. In this case the parallel traverser implementations were predominantly slower.

<table>
<thead>
<tr>
<th>interpolators</th>
<th>ST</th>
<th>PST</th>
<th>LBT</th>
<th>MT1</th>
<th>MT2</th>
<th>MT3</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>123</td>
<td>191</td>
<td>291</td>
<td>345</td>
<td>412</td>
<td>354</td>
</tr>
<tr>
<td>1,000</td>
<td>1,245</td>
<td>1,941</td>
<td>2,947</td>
<td>2,809</td>
<td>3,364</td>
<td>2,970</td>
</tr>
<tr>
<td>10,000</td>
<td>24,121</td>
<td>35,618</td>
<td>52,372</td>
<td>47,242</td>
<td>52,745</td>
<td>52,026</td>
</tr>
</tbody>
</table>

Table 2: Arithmetic mean of 10,000 measurements of the first test program denoted in µs
Table 3: Arithmetic mean of 10,000 measurements of the second test program without affinity to a CPU denoted in µs. (2 threads/4 threads/8 threads)

<table>
<thead>
<tr>
<th>limit</th>
<th>LBT</th>
<th>MT1</th>
<th>MT2</th>
<th>MT3</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>46/46/46</td>
<td>81/114/209</td>
<td>89/164/308</td>
<td>67/79/125</td>
</tr>
<tr>
<td>200</td>
<td>92/92/92</td>
<td>114/149/244</td>
<td>118/190/353</td>
<td>82/128/151</td>
</tr>
<tr>
<td>300</td>
<td>142/142/142</td>
<td>148/191/287</td>
<td>139/206/367</td>
<td>113/182/202</td>
</tr>
<tr>
<td>1,000</td>
<td>630/630/629</td>
<td>503/565/687</td>
<td>422/515/689</td>
<td>406/544/569</td>
</tr>
<tr>
<td>2,000</td>
<td>1,603/1,623/1,603</td>
<td>1,233/1,323/1,439</td>
<td>981/1,177/1,198</td>
<td>945/1,258/1,242</td>
</tr>
<tr>
<td>4,000</td>
<td>4,216/4,215/4,215</td>
<td>3,077/3,319/3,433</td>
<td>2,350/2,760/2,742</td>
<td>2,422/3,474/3,182</td>
</tr>
<tr>
<td>8,000</td>
<td>11,326/11,328/11,346</td>
<td>11,448/8,324/8,815</td>
<td>6,352/7,226/7,317</td>
<td>9,072/8,632/7,838</td>
</tr>
</tbody>
</table>

Table 4: Arithmetic mean of 10,000 measurements of the second test program with affinity to a CPU denoted in µs. (2 threads/4 threads/8 threads)

<table>
<thead>
<tr>
<th>limit</th>
<th>LBT</th>
<th>MT1</th>
<th>MT2</th>
<th>MT3</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>44/44/44</td>
<td>71/100/182</td>
<td>85/111/236</td>
<td>69/65/98</td>
</tr>
<tr>
<td>200</td>
<td>90/90/90</td>
<td>115/138/219</td>
<td>137/140/246</td>
<td>93/92/122</td>
</tr>
<tr>
<td>300</td>
<td>140/140/140</td>
<td>165/188/269</td>
<td>191/203/295</td>
<td>122/125/166</td>
</tr>
<tr>
<td>500</td>
<td>257/257/257</td>
<td>282/305/386</td>
<td>309/258/409</td>
<td>189/227/256</td>
</tr>
<tr>
<td>1,000</td>
<td>626/626/626</td>
<td>651/673/754</td>
<td>679/496/852</td>
<td>391/470/492</td>
</tr>
<tr>
<td>2,000</td>
<td>257/257/257</td>
<td>225/277/343</td>
<td>1,651/1,106/1,823</td>
<td>912/1,081/1,106</td>
</tr>
<tr>
<td>4,000</td>
<td>4,207/4,202/4,202</td>
<td>4,228/4,251/4,331</td>
<td>4,257/2,739/4,392</td>
<td>2,308/2,719/2,742</td>
</tr>
<tr>
<td>8,000</td>
<td>11,295/11,304/11,291</td>
<td>11,322/11,354/11,446</td>
<td>11,350/7,196/11,500</td>
<td>6,117/7,187/7,197</td>
</tr>
</tbody>
</table>

Table 5: Arithmetic mean of 10,000 measurements of the second test program with affinity to a CPU and without OpenMP traverser denoted in µs. (2 threads/4 threads/8 threads)

<table>
<thead>
<tr>
<th>limit</th>
<th>LBT</th>
<th>MT1</th>
<th>MT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>45/44/44</td>
<td>72/100/181</td>
<td>82/106/234</td>
</tr>
<tr>
<td>200</td>
<td>90/90/90</td>
<td>105/139/208</td>
<td>101/146/254</td>
</tr>
<tr>
<td>300</td>
<td>140/140/140</td>
<td>141/180/249</td>
<td>143/173/255</td>
</tr>
<tr>
<td>500</td>
<td>257/257/257</td>
<td>225/277/343</td>
<td>182/280/414</td>
</tr>
<tr>
<td>1,000</td>
<td>627/626/626</td>
<td>495/565/654</td>
<td>448/488/555</td>
</tr>
<tr>
<td>2,000</td>
<td>1,597/1,597/1,597</td>
<td>1,622/1,645/1,725</td>
<td>924/1,119/1,183</td>
</tr>
<tr>
<td>4,000</td>
<td>4,207/4,203/4,202</td>
<td>3,501/3,243/3,404</td>
<td>2,335/2,753/2,804</td>
</tr>
<tr>
<td>8,000</td>
<td>11,295/11,295/11,294</td>
<td>11,390/8,045/8,610</td>
<td>6,376/7,274/7,254</td>
</tr>
</tbody>
</table>

than the serial implementations. The approaches without reusing threads and OpenMP showed a small performance increase, but only for a very high work-load. Reusing of threads produced the worst results.

5.3.2 Prime numbers
The results of the second test program are noted in table 3, table 4, and table 5. Table 3 illustrates the results without an affinity to a core, table 4 shows the results with an affinity set, and table 5 illustrates the results with affinity and without the OpenMP traverser in the same process. The loop breaking traverser matches the functionality of the parallelized implementations, except for the parallelization of the container evaluation, hence the results of the easy and loop braking traversers are not in the table.

Table 6, table 7, and table 8 compare the results as ratios. The measurement using an explicitly set affinity has been made two times, because using the OpenMP traverser and the other parallelized traverser in the same process will lead to side-effects as can be seen by comparing table 7 and table 8.

Parallelization initially increases processing time. Using a parallelized traverser implementation at low work-load tends to result in a performance decrease. The OpenMP implementation is the first one that reaches the break-even with a ratio of 0.89 as can be seen in table 3 and 6 for the prime calculation limit of 200. The traverser with reusing of threads reaches the break-even at 300 as the upper limit, whereas the traverser without reusing threads is still a little bit slower than the serial reference implementation. Given an upper limit of 500 and a resulting overall time consumption of about 300µs, all parallel traversers reach the break-even point. The OpenMP implementation produces less overhead than the other ones.

6 Discussion
The thread reusing traverser produces the worst results given a very low work-load, but gains performance on higher work-loads. It’s ambiguous if it’s better to set an affinity to a core for each thread. The best results are achieved with an explicit affinity, the worst without affinity.

Parallelized traversers run slower if the number of threads doesn’t match to number of cores. The thread reusing traverser produces a vast bulk of overhead whereas the OpenMP solutions produces the fewest. The results support the obvious assumption that the number of threads should match the number of cores. Still, this assumption should be verified on another test system with more
Table 6: Results from table 3 as ratio.

<table>
<thead>
<tr>
<th>limit</th>
<th>LBT</th>
<th>MT1</th>
<th>MT2</th>
<th>MT3</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>46/46/46</td>
<td>1.76/2.48/4.54</td>
<td>1.93/5.57/6.7</td>
<td>1.47/1.72/2.72</td>
</tr>
<tr>
<td>200</td>
<td>92/92/92</td>
<td>1.3/1.62/2.65</td>
<td>1.28/2.07/3.84</td>
<td>0.89/1.39/1.64</td>
</tr>
<tr>
<td>300</td>
<td>142/142/142</td>
<td>0.98/1.45/2.58</td>
<td>1.1/0.73/0.75</td>
<td>0.59/0.78/0.77</td>
</tr>
<tr>
<td>500</td>
<td>259/260/260</td>
<td>0.91/1.12/1.6</td>
<td>1.01/0.71/0.84</td>
<td>0.57/0.82/0.75</td>
</tr>
<tr>
<td>1,000</td>
<td>630/630/629</td>
<td>0.87/1.19/1.5</td>
<td>1.02/1.12/1.6</td>
<td>0.55/0.78/0.75</td>
</tr>
<tr>
<td>2,000</td>
<td>1,603/1,623/1,603</td>
<td>0.77/0.82/0.9</td>
<td>0.61/0.73/0.75</td>
<td>0.59/0.78/0.77</td>
</tr>
<tr>
<td>4,000</td>
<td>4,216/4,215/4,215</td>
<td>0.73/0.79/0.81</td>
<td>0.56/0.65/0.65</td>
<td>0.57/0.82/0.75</td>
</tr>
<tr>
<td>8,000</td>
<td>11,326/11,328/11,346</td>
<td>1/0.73/0.78</td>
<td>0.56/0.64/0.64</td>
<td>0.50/0.76/0.69</td>
</tr>
</tbody>
</table>

Table 7: Results from table 4 as ratio.

<table>
<thead>
<tr>
<th>limit</th>
<th>LBT</th>
<th>MT1</th>
<th>MT2</th>
<th>MT3</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>44/44/44</td>
<td>1.61/2.27/4.14</td>
<td>1.93/2.52/5.36</td>
<td>1.57/1.47/2.22</td>
</tr>
<tr>
<td>200</td>
<td>90/90/90</td>
<td>1.28/1.53/2.43</td>
<td>1.52/1.56/2.73</td>
<td>1.03/1.02/1.36</td>
</tr>
<tr>
<td>300</td>
<td>140/140/140</td>
<td>1.18/1.34/1.92</td>
<td>1.36/1.45/2.11</td>
<td>0.87/0.89/1.19</td>
</tr>
<tr>
<td>500</td>
<td>257/257/257</td>
<td>1.1/1.91/1.3</td>
<td>1.2/1.59</td>
<td>0.74/0.88/1</td>
</tr>
<tr>
<td>1,000</td>
<td>626/626/626</td>
<td>1.04/1.08/1.2</td>
<td>1.08/1.12/1.36</td>
<td>0.62/0.75/0.79</td>
</tr>
<tr>
<td>2,000</td>
<td>1,597/1,557/1,597</td>
<td>1.02/1.06/1.08</td>
<td>1.03/0.71/1.41</td>
<td>0.57/0.69/0.69</td>
</tr>
<tr>
<td>4,000</td>
<td>4,202/4,202/4,202</td>
<td>1/1.01/1.03</td>
<td>1.01/0.65/0.64</td>
<td>0.55/0.65/0.65</td>
</tr>
<tr>
<td>8,000</td>
<td>11,294/11,304/11,291</td>
<td>1/0.71/0.76</td>
<td>1/0.64/0.64</td>
<td>0.54/0.64/0.64</td>
</tr>
</tbody>
</table>

Table 8: Results from table 5 as ratio.

<table>
<thead>
<tr>
<th>limit</th>
<th>LBT</th>
<th>MT1</th>
<th>MT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>45/44/44</td>
<td>1.62/2.37/4.11</td>
<td>1.82/2.41/5.32</td>
</tr>
<tr>
<td>200</td>
<td>90/90/90</td>
<td>1.16/1.54/2.31</td>
<td>1.12/1.62/2.82</td>
</tr>
<tr>
<td>300</td>
<td>140/140/140</td>
<td>1/1.29/1.78</td>
<td>1.02/1.24/1.82</td>
</tr>
<tr>
<td>500</td>
<td>257/257/257</td>
<td>0.86/1.08/1.33</td>
<td>0.71/0.79/1.61</td>
</tr>
<tr>
<td>1,000</td>
<td>627/626/626</td>
<td>0.79/0.9/1.05</td>
<td>0.71/0.78/0.89</td>
</tr>
<tr>
<td>2,000</td>
<td>1,599/1,597/1,596</td>
<td>0.78/0.83/0.84</td>
<td>0.58/0.70/0.74</td>
</tr>
<tr>
<td>4,000</td>
<td>4,207/4,203/4,202</td>
<td>0.83/0.77/0.81</td>
<td>0.56/0.66/0.67</td>
</tr>
<tr>
<td>8,000</td>
<td>11,295/11,295/11,294</td>
<td>1/0.71/0.76</td>
<td>0.56/0.64/0.64</td>
</tr>
</tbody>
</table>

than two cores. Still, there exist multiple reasons which influence threading behavior, from hardware layout to operating systems’ scheduling strategies.

A remarkable anomaly occurs given a very high work-load where the OpenMP traverser and the traverser without thread reusing show a worse performance than before when using two threads. This anomaly doesn’t occur if more than two threads are used.

If other processes are running on the system, the parallelized traversers show heavy performance losses. Even if the program stress is started with a low priority and the test program with high priority the ratios are around 29.6. This additional work-load does only have a small effect on the serial traversal. Parallelization using threads depends on many parameters, like the thread scheduling and even the design of the CPU and the chipset. The used hardware is a consumer system. It seems advisable to individually evaluate the concepts taking the target platform in account.

The difference between the result of the first and the second test program is remarkable. To analyze this behavior, the second test program was modified so that 1000 containers were checked with a lower prime calculation limit. Both test programs produced bad results under this condition. A possible reason for this effect are cache misses caused by memory consumption or frequent calls of virtual methods which will play havoc with the jump prediction heuristics of the CPU.

It’s best to have containers with high work-load to take benefit from parallelization. If only parts of a data flow network show this characteristic, the performance improvement for this part can be negated by other parts with many containers consisting of low work-loads.

7 Conclusion

The modular architecture with a strict separation between structural and process components has proven to be beneficial during the analysis of different parallelization approaches. The architecture provided a reliable foundation for the necessary evaluations. Typical side-effects that often occur when changing central behaviors—like the order of evaluation—could be avoided. The chosen parallelization techniques were implemented with only minor modifications which were all targeted at the traverser layer. The underlying structural components and their semantics were kept untouched.

The performance benefit was varying. Using the first test scenario, parallelization mainly lead to a decreasing performance. A small benefit occurred a very high work-load. The second test program showed an asymptotic behavior to the theoretical gain limit of 100% as should be ideally provided by a system with two CPU cores. This result confirms that the performance gain depends on the graph structure and especially on the function and work-load of the containers.

The essential criterion for parallelization is the granularity of the
parallelization target. Threads seem to be inappropriate to parallelize a typical data flow network with many containers having a low processing profile. It seems advisable to evaluate the concepts with data flow networks with a high work-load on every container, e.g., like gesture recognition etc. A degree of granularity seems to be helpful, but is hard to find due to the heterogeneous target platforms. Generally, coarse structures with higher work-loads are more appropriate.

The second test program produced much better results than the first one and the best results of all were produced with reused resources. This supports the obvious assumption that the resources for parallelization should be reused if a qualified data flow network is processed. The modular approach illustrated in this paper can be easily adjusted to current requirements by using different traversers adaptively. The results can in either case be transferred to a semantic structure, as proposed in prior work [16], because this layer also represents task-local data. Different traversers can operate simultaneously on different tasks like input processing, physics, or rendering depending on an available graph structure. Furthermore, this high-level layer has a very beneficial granularity for parallelization.

ACKNOWLEDGEMENTS

The authors wish to thank Dennis Wiebusch and Martin Fischbach.

REFERENCES


