

Studienarbeit

Parallel Highway-Node Routing

Manuel Holtgrewe

Betreuer: Dominik Schultes, Johannes Singler
Verantwortlicher Betreuer: Prof. Dr. Peter Sanders

February 2, 2008

Abstract

Highway-Node Routing is a scheme for solving the shortest path problem showing excellent speedups over DIJKSTRA's algorithm. There also is a dynamic variant that allows changes to the cost function.

Based on an existing, sequential implementation, we present a parallel version of the precomputation required for Highway-Node Routing. We also present experimental results with the road network of Europe as the input.

During parallelization, problems with imbalanced load and memory bandwidth limitations were encountered. We evaluated several load balancing variants and also a modification that makes the problem more regular. We also considered a scheme that tries to exploit the shared cache structure of the used machine to lessen the memory bandwidth limitation.

The parallel version achieves a speedup of up to 3.1 with 4 threads and 4.5 with 8 threads over the original sequential implementation.

Contents

1	Introduction	3
2	Preliminaries	4
2.1	Highway-Node Routing Overview	4
2.2	OpenMP Overview	5
2.2.1	Fork/Join Model	5
2.2.2	OpenMP Memory Model	5
2.2.3	Parallelization Variants	6
2.3	MCSTL Overview	8
2.3.1	Parallel for_each	8
2.3.2	Work Stealing in for_each	8
3	Parallelization	9
3.1	Changes to the Original Data Structures	9
3.2	Original Sequential Implementation	10
3.2.1	Graph Structure	10
3.2.2	Shortest Path Tree Construction	11
3.2.3	Highway Edge Dispatching	11
3.2.4	Edge Reduction	12
3.3	Parallel Implementation	12
3.3.1	Shortest Path Tree Construction	12
3.3.2	Inserting Edges from the Overflow Buffers	13
3.3.3	Highway Edge Dispatching	14
3.3.4	Reduction	14
4	Experimental Results	15
4.1	Results With Simple Load Balancing	16
4.2	Load Balancing Comparison	18
4.2.1	Making The Edge Reduction More Regular	21
4.3	Examination of Memory Bandwidth Limitation	23
4.4	Improving Locality	25
4.5	Overall Performance	26
5	Discussion and Future Work	28

1 Introduction

The Shortest Path Problem (SPP) is a classic problem in algorithmics with many variations. In this paper, we consider the Single Source Single Target SPP (SSSTSP) variant: Given a graph G , a source node s and a target node t , the shortest path from s to t is to be computed in G .

One of the prominent applications is in road networks where it is of immediate value. The largest road networks available to us are those of Europe and North America. We performed experiments on the graph of Europe which has 18'029'721 nodes and 22'413'128 edges.

Asymptotically, DIJKSTRA's algorithm greatly solves the problem. Current research focuses on achieving speedups on the DIJKSTRA algorithm. The highest speedups can be reached with precomputation techniques. Highway-Node Routing (HNR), developed by Schultes and Sanders in [5], is one of the solutions to SSSTSP based on DIJKSTRA's algorithm. Lately, a variant of HNR has been introduced in [5] that allows changes to edge weights in the graph. The vision here is to change edge weights, e.g. to reflect traffic jams, without taking the shortest path computation system offline. Thus, the precomputation must be redone quickly after a change.

A high number of updates per time is desirable to keep the replies to the shortest path queries as up-to-date as possible. On a modern processor, the repeated precomputation step for HNR takes roughly two minutes.

The previously available program to perform the precomputation is sequential. We will not see big speedups on future computers for sequential programs, however. Future computers will feature multiple processors and/or cores instead.

Thus, it is desirable to provide a parallelized version of the HNR precomputation step. This way, it will be possible to either perform the precomputation step in less time, or to process more detailed or larger street networks in the same amount of time. For example, the "resolution" of streets could be increased to offer more detailed traffic reports, or the graph of Europe could be expanded by the roads in Asia. Ultimately, we could also consider searching for shortest path in a graph covering all roads and oversea connections world wide.

The main result of this student research project are shared-memory parallel variants of the multi-level overlay graph construction necessary for HNR. Differences in the variants are mainly respective to the load balancing since one of the steps is non-regular.

In Section 2, we will present some preliminaries: We give a high level view of (Dynamic) Highway Node Routing in Section 2.1, and an overview of the software tools we used for the parallel implementation in Sections 2.2 and 2.3. In Section 3, we describe the sequential implementation of the code that was parallelized, and the parallelized version. In Section 4, we present experimental results and explanations for them. Last, we give an outlook on open problems and future work in Section 5.

2 Preliminaries

In this paper, p is the number of threads that the parallel parts of the program is executed with.

Further, we only consider directed graphs $G = (V, E)$ with edges (u, v) from node u to node v . As usual, undirected graphs $G' = (V', E')$ are mapped to directed ones such that if $\{u, v\} \in E'$, then $(u, v) \in E$ and $(v, u) \in E$.

The edges represent roads and have a weight w so that $w(u, v)$ represents the time it takes to go from u to v . Let $d(u, v)$ denote the length of the shortest path from u to v .

In Section 2.2.3, we will introduce the so called *dynamic* load balancing variant. Note that this only refers to the *dynamic* load balancing variant for loops in OpenMP.

2.1 Highway-Node Routing Overview

Highway Node Routing (HNR) was developed by Sanders and Schultes and is described in [5]. We will give a short overview of HNR in this section as a preliminary.

Overlay Graphs. Let us assume that we could identify a set of “important” nodes V' , called *highway-nodes*, in a road network, i.e. the graph $G = (V, E)$. A graph $G' = (V', E')$ is an *overlay graph* of G with respect to $V' \subseteq V$ if for all $u, v \in V'$, the length of the shortest path between u and v is the same in both G and G' .

Shortest Path Search in Overlay Graphs. If we have constructed an overlay graph G' with respect to G , then we can perform shortest path queries for arbitrary nodes $s, t \in V$: Start a forward DIJKSTRA search from s and a backward DIJKSTRA search from t . Perform both searches in G .

DIJKSTRA searches grow shortest-path trees. Once all branches of these trees contain a highway-node, we can abort the search in G . We can then continue to search in the hopefully smaller graph G' from the reached highway-nodes.

This schema can then be extended to multiple levels of overlay graphs. An overlay graph hierarchy is constructed analogue to the above description from highway-nodes $V'_1 \supseteq V'_2 \supseteq \dots \supseteq V'_\ell$.

The query algorithm is extended in the obvious way.

Creating Overlay Graphs. The construction of an overlay graph hierarchy is done in a preprocessing step. We describe only the construction of one additional level, the extension to multiple levels is obvious.

An overlay graph is constructed with so called *local* DIJKSTRA searches: From each $v \in V'$, a DIJKSTRA search is started. The DIJKSTRA search is – roughly speaking – aborted if each path of the tree grown during the search contains at least one highway-node. The abortion of the searches is based on heuristics, called abortion criteria. See [5] for more details on these abortion criteria.

Say U is the set of these passed-through highway-nodes. For each node $u \in U$, the edge (v, u) is added to the edge set E' of the overlay graph.

2.2 OpenMP Overview

OpenMP [1, 4] is a specification for compiler extensions and a library that allow parallel programming in C++ (and Fortran) using shared memory. It provides library functions for timing, locks, and queries regarding the program's state. For example, there is a function returning the number of threads that are currently being executed, and a method returning the identifier of the current thread.

2.2.1 Fork/Join Model

These library functions support the parallelism OpenMP provides through compiler extensions (so called **#pragmas**). A programmer can use these pragmas to mark a section or a loop to be executed in parallel, limiting them to the fork/join parallelism model (see Figure 1).

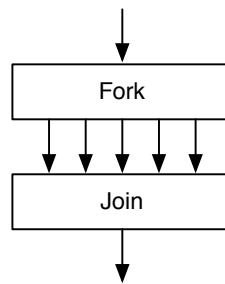


Figure 1: The Fork/Join Model

The fork/join model is a simple parallel model: On *fork*, several threads are created and executed in parallel. Each thread executes a part of the total work of the program. After completing its work, each thread waits for the other threads to complete their execution using barrier synchronization on *join*.

Explicit barrier synchronization is also possible using OpenMP's pragmas.

2.2.2 OpenMP Memory Model

OpenMP uses a shared-memory model with relaxed consistency. This means that if multiple threads share global state, then each thread has its own *temporary view* of this state. When a thread changes this global state, the changes are not automatically visible to the other threads. This allows the compiler to apply several optimization strategies:

The compiler and the underlying machine can keep multiple copies of data and do not have to make sure that all views are coherent. For example, using relaxed memory consistency, the data can be kept in a register of the processor for a long time, saving accesses to the slower main memory. On non-uniform memory access (NUMA) systems, each computing node could keep a copy of the data, so coherence does not have to be enforced on every memory access.

The pragma *flush* allows the programmer to enforce the visibility of a certain variable in a thread. This pragma forces the variable to be loaded/stored directly out of/into the shared memory. The keyword *volatile* is a C++ language construct that fulfills a similar functionality. However, *flush* is more fine grained: *volatile* forces the variable to be accessed at its central location in main memory without exception. No compiler optimizations are allowed and thus no incoherence can appear. This affects every access of the variable marked as *volatile*. In contrast, *flush* only enforces consistency between a thread's temporary view and the actual memory when it is explicitly called.

OpenMP also provides limited access to atomic operations using the *atomic* pragma. It is possible to perform an increment, decrement one of the *+=*, **=*, ... operations atomically. An example of this is shown in Listing 1.

Listing 1 Atomic Operations in OpenMP

```
#pragma omp parallel for dynamic
for (int i = 0; i < 1000; ++i)
{
# pragma omp atomic
    sum += a[i];
# pragma omp atomic
    product *= a[i];
# pragma omp atomic
    j++;
}
```

2.2.3 Parallelization Variants

The two most important constructs OpenMP provides are *parallel regions* and *parallel loops*.

Parallel Regions. The simplest way to add parallel execution to a program is using *parallel regions*. In Listing 2, the block is executed by multiple threads (the number of threads is configurable using environment variables).

Listing 2 Example of `omp_get_thread_num()`

```
#pragma omp parallel
{
    int iam = omp_get_thread_num();

    // do work for thread with ID "iam"
}
```

Parallel Loops. A more refined way to add parallelization with OpenMP is to parallelize *for* loops. Each iteration of a loop must not depend on the result of any previous iteration for this to work. For example, the sum of two vectors a and b could be calculated as in Listing 3.

Listing 3 Example of a parallel loop with OpenMP

```
int sum *a = new int[100], *b = new int[100], *sum = new int[100];
#pragma omp parallel for schedule(variant, chunk_size)
for (int i = 0; i < 100; ++i) {
    sum[i] = a[i] + b[i];
}
```

The loop will be executed in parallel. How this is done exactly depends on the *schedule* setting and its parameters *variant* and *chunk_size*:

1. The variant *static* breaks the total number of iterations into pieces of size *chunk_size*. Then, the chunks are assigned to the threads in a round-robin fashion statically.
2. The variant *dynamic* breaks the total number of iterations into pieces of size *chunk_size*. Then, it assigns them to a thread as soon as this thread finishes its previous chunk.
3. The variant *guided* works as follows: If *chunk_size* is k , then the variant *guided* breaks the input into pieces whose size is proportional to the size of the work that still has to be processed divided by the number of threads. This number will decrease until it is k .

These chunks are assigned to the threads as the *dynamic* variant does.

These parameters allow the programmer to select from load balancing strategies and trade-off the amount of necessary synchronization against quality of load balancing. While the load balancing gets more refined from 1 to 3, the amount of necessary communication between the threads increases.

Necessary Communication. When two or more threads execute a loop in parallel using dynamic or guided assignment, they have to make sure that no iteration is processed twice or even skipped. Naïvely, this could be ensured using locks provided by the operating system. A more sophisticated way is to use atomic operations which are provided by modern shared memory parallel (SMP) computers. These are complex machine instructions that allow the programmer to access a machine word of the shared memory and guarantee that no other processor's instruction can access the part of memory at the same time.

For example, the x86-64 architecture we used provides a *fetch-and-add* command (called *XADD* for *exchange-and-add*, see [2]) which expects a source memory location, a target memory location, and an integer. It stores the value at the given source memory

location in the target memory location and adds the integer to the value at the given source location in main memory. This is faster than a lock which is implemented by the operating system. However, it is still slower than just accessing the memory normally.

Note that an atomic operation is required for each chunk. Thus, there is a trade-off between a large chunk size and few atomic calls and a small chunk size and many atomic calls. As always, this depends on how much the running time of the loop's body dominates the running time of the loop.

2.3 MCSTL Overview

The Multi Core Standard Template Library (MCSTL) [7] is a parallelized version of the C++ Standard Template Library (STL). It provides parallel versions of the STL *algorithm* module where a parallel version makes sense. At the moment of the writing, there is an effort to integrate the MCSTL into the GCC project as the “GNU libstdc++ parallel mode”.

2.3.1 Parallel for_each

The relevant part of the MCSTL for this paper is the parallel implementation of the `for_each` function. The function expects a begin and end iterator of a sequence and a functor. It then calls the functor on each item of this sequence in parallel.

An example is shown in Listing 4.

Listing 4 Example of the using the parallel `for_each`

```
struct functor {
    operator()(const int &x) {
        // process x
    }
};

// ...

// vector<int> items ... ;

functor f;
std::for_each(items.begin(), items.end(), f);
```

2.3.2 Work Stealing in for_each

The parallel implementation of the for each algorithm provides a load balanced variant using work stealing [7].

The work stealing algorithm works as follows (given an input sequence of length n , p threads and a chunk size k):

1. Separate the input sequence into p parts of equal length (plus/minus 1 element).
2. Assign each thread one of these parts.
3. Start all threads.
4. Each thread takes a chunk of size k out of his own part and processes it. The work taken out cannot be stolen. Note that k is a tuning parameter.
5. When a thread finishes its computation and other threads are still working, then it selects a random thread which still has unreserved work. The thread then steals half of the other thread's work, makes it its own part, and goes to 4. If all work is already reserved, the thread quits execution.

Taking work out of a thread's part – either from its own or stealing from another – requires atomic operations. The number of atomic operations depends on the chunk size. A large chunk size reduces the number of atomic operations since currently processed work cannot be stolen. This can also reduce the total running time since atomic operations are relatively expensive.

On the other hand, the running time of the functor on the input sequence can be very irregular. Thus, a large chunk size can increase the total running time: The first item of a large chunk could take a thread very long and idle threads cannot steal the rest of the chunk. Each thread has to wait for the others to complete their processing and so the longest-working thread dictates the total running time.

3 Parallelization

3.1 Changes to the Original Data Structures

We will first outline the general class structure of the sequential program and then describe the changes that were made to it to support parallelism.

The class *UpdateableGraph* provides an implementation of an updateable adjacency array based graph structure. It uses *UpdateableNode* objects for representing the nodes. One of the pieces of information these node objects store is the address of the node in the addressable priority queues for the DIJKSTRA variants.

The class *DynamicHwyNodeRouting* provides the state for the construction of the multi-level overlay graph. *DynamicHwyNodeRouting* objects have an instance of the DIJKSTRA variant for the shortest path tree construction and an instance of the DIJKSTRA variant for the edge reduction step described in Section 3.2.4. The two relevant DIJKSTRA variants for construction *DijkstraConstr* and reduction *DijkstraDynReduce* are instances of class template *DijkstraTemplate*.

The main idea behind parallelizing the precomputation technique is executing multiple DIJKSTRA searches in parallel since they are independent: For each node u of the graph, a DIJKSTRA search is started. As described in Section 2.1, the searches only occur on the current level ℓ . These DIJKSTRA searches do not consider the edges that have been

added to the currently constructed level $\ell + 1$. Thus, an arbitrary number of threads could be started, each executing a DIJKSTRA search.

To support the parallel execution of DIJKSTRA searches, the *UpdateableNode* class has been changed. It does not store the identifier (an integer) for the addressable priority queues any more. Instead, *DijkstraTemplate* has been augmented with a mapping from nodes to their address in the priority queue. The mapping is implemented as a simple STL vector of addresses in the priority queue indexed by the node ID. The priority queues have been adjusted accordingly, too. We checked for possible performance problems because of this change, but none was found.

Additionally, the *DynamicHwyNodeRouting* now stores an array of the objects implementing the DIJKSTRA variant – one for each of the p thread. Because these objects each store a mapping from node ID to the position of the node in the priority queue, constructing and freeing them is expensive.

3.2 Original Sequential Implementation

This section describes the relevant parts of the original sequential implementation. First, the graph data structure of the sequential implementation is described in Section 3.2.1. Then, Sections 3.2.2, 3.2.3 and 3.2.4 outline the three steps for the level construction. These are executed iteratively to construct each level from the previous one.

Section 3.3 describes the changes made to the sequential implementation.

3.2.1 Graph Structure

The graph is implemented as an adjacency array (aka forward-star) representation (see Figure 2). Technically, we are using an STL vector which essentially is an array that can grow dynamically. However, the data structure is augmented to accommodate for the hierarchical overlay graphs. The extension relevant to the parallelization is as follows:

The elements in the node array have references (indices) into the edge array but also to *level nodes*.

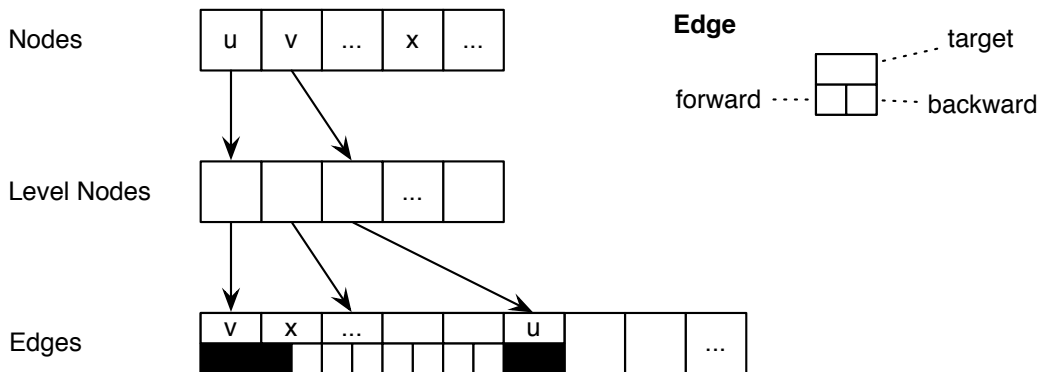


Figure 2: Relevant parts of the graph structure.

For each node, the edges are sorted by the highest highway level they belong to. Additionally, each edge stores the lowest level it belongs to (the level it was created on). The level nodes point to the first edge of the level they represent. The highest level of the multi-level overlay graph, an edge belongs to, is implicitly given by the level node with the smallest level that the node belongs to. Note that reusing edges is only done to save memory, but it is not required for the correctness. The implementation could simply create a new edge on every level.

Because edges are added to the graph, the two edge sequences belonging to two different nodes are not continuous. Instead, there always is space for 2^k edges in an *edge bucket*. In the original implementation, this information is stored implicitly:

If there are ℓ edges stored in the edge bucket, the space is the next power of two, so that $\ell \leq 2^k$. When a bucket overflows, the original implementation simply allocates a new bucket at the end of the edge array with the size of 2^{k+1} . Finally, the edge is inserted.

The adjacency array implementation represents directed graphs. The query algorithm for HNR uses a bidirectional DIJKSTRA search. For directed graphs, for each directed edge (u, v) , there has to be a backward edge which is conceptually not the same as the edge (v, u) that exists if u and v are connected by a road that can be used in both directions. Thus, the original implementation stores a flag for both the forward and the backward search direction. For the common case where a conceptually undirected edge is represented as two directed edges, both edges have the forward and the backward flag set, in order to save memory.

3.2.2 Shortest Path Tree Construction

The shortest path tree construction is used to find highway edges for each node u . Thus, for each node u , a unidirectional DIJKSTRA search is started. The search continues until a certain abortion criterion is fulfilled (see [5] for a detailed explanation).

For each reached highway-node v of this shortest path tree, the edge (u, v) is considered: If the edge already exists, it is marked for a level upgrade later on. If the edge does not exist yet, it is added to node u for the level we are currently constructing. Note that as described in Section 3.2.1, this might move the edge bucket for node u .

3.2.3 Highway Edge Dispatching

After the shortest path tree construction, the existing edges that are to be upgraded to the next level have been marked. Upgrading an edge consists of moving the edge to a higher index in the edge array and adjusting the pointers of the level nodes into the edge array.

There is a special case that has to be taken care of: There might be an undirected edge $\{u, v\}$ in the graph and only one direction is to be upgraded. As described in Section 3.2.1, conceptually, there are four directed edges representing this undirected edge.

The undirected edge has to be “split”, i. e. the backward flag has to be removed from both edges. The backward edges are later added again to the graph. This happens in a separate step when exporting the constructed multi-level overlay graph. They are only relevant for the query phase but not for the construction itself.

3.2.4 Edge Reduction

When a new level of the overlay graph is created, it might contain superfluous edges. An edge (u, v) in the next level is superfluous if there already is an alternative path from u to v in the next level whose total weight is less than or equal to the weight of (u, v) .

These superfluous edges can be determined by performing a DIJKSTRA search from each node u of the graph. The search can be aborted when all neighbors of u have been settled. Then, the algorithm can check for edges to be removed. An edge can be removed if $d(u, v) \leq w(u, v)$ and the path does not only consist of (u, v) . d is the distance of two nodes and w is the weight of an edge between two nodes. See [5] for details.

Note that because of the nature of road networks, the DIJKSTRA search for reduction can be aborted very quickly for almost all nodes. Experiments show that there are a only few (35 in our graph of Europe) nodes where the reduction step takes very long. See Section 4.2.1 and Table 4 for details.

The reason for the long search times are long-distance ferry connections: Relatively many other nodes have to be settled until the target node of a “long” edge describing a ferry connection has been settled. Section 3.2 of [6] describes a similar problem with local DIJKSTRA searches and long-distance ferry connections.

3.3 Parallel Implementation

3.3.1 Shortest Path Tree Construction

The most time-consuming part in the sequential shortest path tree construction are the DIJKSTRA searches. They are all independent of each other and can be executed in parallel.

The parallel implementation separates the sequence of nodes into a number of chunks proportional to the number of threads. Then, p independent threads are started. Each one takes a chunk (or slab, as chunks are sometimes called) and then processes each node in this chunk as described in Section 3.2.2:

For each node u , the DIJKSTRA search tree is grown. For each leaf v of this tree, it is checked whether the edge (u, v) already exists in the graph. If so, then the node is marked as a highway edge. This marking does not affect any other DIJKSTRA search that might relax the upgraded edge. If (u, v) does not already exist in the graph then it is added *iff there still is space in the edge bucket*.

When constructing level ℓ , the DIJKSTRA searches are only performed on level $\ell - 1$. The edges in the buckets are sorted by the highest level of the multi-level overlay graph they belong to. Thus, adding an edge to the bucket will only insert it at the right end of the bucket and the DIJKSTRA searches on level $\ell - 1$ are not affected by the insertion.

If there is not enough space in the edge bucket, then the to-be-added edge is added to an *overflow edge buffer*. Each thread has its own overflow edge buffer. Each node is processed once by one thread so there cannot be any duplicate edge (u, v) in two edge buffers. These “overflow” edges are then later added to the graph as described in Section 3.3.2. They belong to the next level of the graph and only the current level is considered for the construction of the next one.

Because each node is processed by exactly one thread, the insertion does not have to be synchronized.

3.3.2 Inserting Edges from the Overflow Buffers

After growing the DIJKSTRA search trees from Section 3.3.1, the edges from the overflow buffers have to be processed. Optimally, this would also be done in parallel but a naïve approach is not feasible: Two threads would have to reserve space at the end of the graph’s edge array. The resizing could require the array to allocate new memory and copy the old values over. Thus, multiple requests to reserve space at the end of the edge array would have to be sequentialized using locking. Since the number of edges to be added to the buckets is relatively small (see Table 1) this would degrade performance greatly.

level	bucket size				
	16	32	64	128	256
1	21'711	1'166'457	7'963	3	
2	31	85'350	4'758		
3	1	6'639	10'322	112	
4		386	3'265	810	
5		19	251	1'018	
6			25	393	4
7			1	64	5
8				6	1

Table 1: Distribution of the moved buckets’ sizes after resizing for all levels

Instead, the following approach has been taken:

1. Determine the total size that would be required if the edges were added sequentially from the overflow buffers.
2. For each overflow buffer, determine the offset in the edge array where the first to-be-moved bucket would be placed at.
3. Resize the edge array to this size.
4. Add edges from the overflow buffers in batches, and in parallel.

Step 1 is simple to parallelize: For each bucket affected by an edge in the overflow buffers, add the number of edges in the buffer to the current size of the bucket and round up to the next power of 2. This can be done in one linear pass because the edges in the overflow buffers are sorted by their source node.

Step 2 is not worth the overhead to be parallelized: Only the sizes of the p overflow buffers need to be added.

Step 3 has not been parallelized either. Instead the sequential implementation of the STL *vector* class has been used. Allocating memory is strictly sequential and on the uniform memory access machines (UMA) we used, since one core can saturate the memory bus with streaming.

Step 4 can then again be parallelized in a simple manner: Each thread processes its own overflow buffer. For each affected edge bucket, move it to the correct position and add the edges from the overflow buffer.

After freeing the overflow buffers, the parallel implementation has a small advantage over the sequential implementation in memory footprint: Normally, when a buffer overflows, it is only resized to the next power of 2. In the parallel implementation, this waste of memory does not occur.

Note that this waste of memory is relatively small: Judging by the numbers from Table 1, 48 MiB is a rough upper limit for the amount of wasted memory. The program uses a maximum of 2.6 GiB main memory on an 8 core machine when performing the precomputation for the graph of Europe. So there is a waste of less than 2% of memory in the sequential program.

3.3.3 Highway Edge Dispatching

The parallelization of the sequential edge dispatching described in Section 3.2.3 is simple:

When making a bidirectional edge unidirectional, the processing of one edge is completely independent of the processing of all other edges. Edges can only be iterated through the node they belong to. Thus, the node array is iterated in parallel, and from each node u , each outgoing edge (u, v) is considered. The program searches for the reverse edge (v, u) , and if this edge is no highway edge, then edges (u, v) and (v, u) are marked to be unidirectional edges.

“Iterating parallel” means that the node ids are iterated with a parallel for loop and then processed in the loop.

Because the reverse edge is also considered, a thread could process the node u and edge (u, v) while another thread considers v and (v, u) . However, since the edge is made unidirectional iff only one edge belongs to the next highway level, only one thread can touch the direction markers. No synchronization or atomic operation needs to be performed for this step.

3.3.4 Reduction

The sequential edge reduction described in Section 3.2.4 collects the reducible edges of the reduction DIJKSTRA search and then immediately removes them from the next level.

Again, the parallelization of this step executes the DIJKSTRA searches in parallel. However, when removing the edges directly after the search, another thread’s DIJKSTRA search might access one of the edges which another thread tries to remove from the graph. Instead of introducing book keeping to prevent this, the two tasks *search reducible edges* and *remove edges* are separated.

First, all nodes are considered in parallel and for each node u , a reduction DIJKSTRA search is executed. The search does not collect the reducible edges in a buffer as the sequential variant does. Instead, it marks the superfluous edges.

Then, in a second step, all edges are considered again in parallel and the superfluous ones are removed. This causes no conflicts since deleting edges does not move an edge buffer.

4 Experimental Results

For our experiments, we used a stable snapshot of the GNU GCC C++ compiler’s 4.3 development branch. At the moment of the writing, no stable version was available. The 4.3 branch is the first one which has the MCSTL built-in as the “libstdc++ parallel mode”.

All experiments were conducted on a 8-core Intel Xeon 5345 system with 2.33 GHz each and 16 GiB of shared memory. The 8 cores are distributed on 2 processors, each processor has 4 cores. Each processor also has two 4 MiB L2 caches that are shared by two cores each. Each core has its own L1 cache. Figure 3 shows the memory hierarchy of the machine we used.

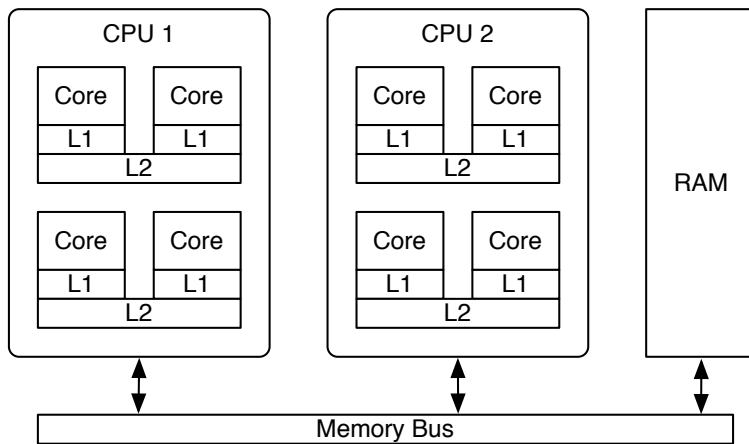


Figure 3: The memory hierarchy of the machine we used.

The machine is a UMA system, both processors share a common memory bus. Section 4.3 examines possible performance limitations caused by the shared singular memory bus.

The machine runs a Linux Kernel version 2.6.18.8.

4.1 Results With Simple Load Balancing

Because road networks are irregular graphs, optimal performance cannot be expected without any load balancing. Thus, we will first consider the results of the parallelization variant that uses OpenMP loop parallelization with the load balancing variant *dynamic*. Table 2 shows the running time of the construction phase.

The initialization of one DIJKSTRA construction object and one DIJKSTRA reduction object takes 0.12s. For each thread, such a DIJKSTRA construction object has to be initialized and this creates a total overhead of roughly 1 second.

section		number of threads							
		1	2	3	4	5	6	7	8
level 1	running time	73.9	46.7	35.0	31.3	26.9	25.1	23.6	22.6
	speedup		1.58	2.11	2.36	2.75	2.94	3.13	3.27
level 2	running time	28.3	15.3	11.0	8.7	7.1	6.2	5.6	5.2
	speedup		1.85	2.57	3.25	3.99	4.56	5.05	5.44
level 3	running time	6.7	3.7	2.8	2.3	2.0	1.8	1.7	1.6
	speedup		1.81	2.39	2.91	3.35	3.72	3.94	4.19
level 4	running time	2.6	1.6	1.3	1.1	1.0	1.0	1.0	0.9
	speedup		1.62	2.00	2.36	2.60	2.60	2.60	2.89
level 5	running time	1.1	0.8	0.7	0.7	0.7	0.7	0.7	0.7
	speedup		1.38	1.57	1.57	1.57	1.57	1.57	1.57
levels 6-8 ^a									
total	running time	116.6	72.2	54.9	48.3	41.9	39.1	37.0	35.5
	speedup		1.61	2.12	2.41	2.78	2.98	3.15	3.28

^aNot shown. running time is less than 1s and the speedup ranges between 1 and 0.8.

Table 2: Running time of the construction of the levels in seconds and speedups over construction with one thread.

As can be seen in Table 2, the construction of level 1 and 2 dominates the running time. Thus, any detailed analysis should be focused on these levels. We focus on the construction of the first level with 1, 2, 4 and 8 threads. See Table 3 and Figure 4 for the running time in seconds and the speedup when constructing the first level with the dynamic load balancing variant.

For 2 threads, the speedup is in the range that can be expected. The speedup of 6.43, with 8 threads, for the shortest path tree construction is also acceptable.

The speedup for the parts taking less than one second does not increase with 4 or 8 threads, but this is explainable with memory bus limits: The steps performed for highway edge dispatching and edge reduction, for example, do not do much more than

section		number of threads						
		1	2	1.85	4.17	3.55	2.3	6.43
shortest path tree		14.8	8.0	<i>1.85</i>	4.17	<i>3.55</i>	2.3	<i>6.43</i>
overflow edges	bucket sizes	0.05	0.03	<i>1.67</i>	0.02	<i>2.50</i>	0.02	<i>2.50</i>
	resize ^a	0.73	0.73	<i>1.00</i>	0.72	<i>1.01</i>	0.71	<i>1.03</i>
	move edges	0.22	0.13	<i>1.69</i>	0.12	<i>1.83</i>	0.11	<i>2.00</i>
hwy edge dispatching	split bidir edges	0.64	0.38	<i>1.68</i>	0.23	<i>2.78</i>	0.12	<i>5.33</i>
	upgrade edges	0.70	0.38	<i>1.84</i>	0.21	<i>3.33</i>	0.12	<i>5.83</i>
edge reduction	marking	55.9	36.3	<i>1.54</i>	25.4	<i>2.20</i>	18.5	<i>3.02</i>
	edge removal	0.43	0.24	<i>1.79</i>	0.14	<i>3.07</i>	0.10	<i>4.30</i>

^aSequential; see Section 3.3.2 for details.

Table 3: Running time in seconds and speedups of the construction steps for level 1 with *dynamic* balancing. Speedups are in italics. Initialization times are not shown; They are significantly smaller than 0.1s. Compare Figure 4.

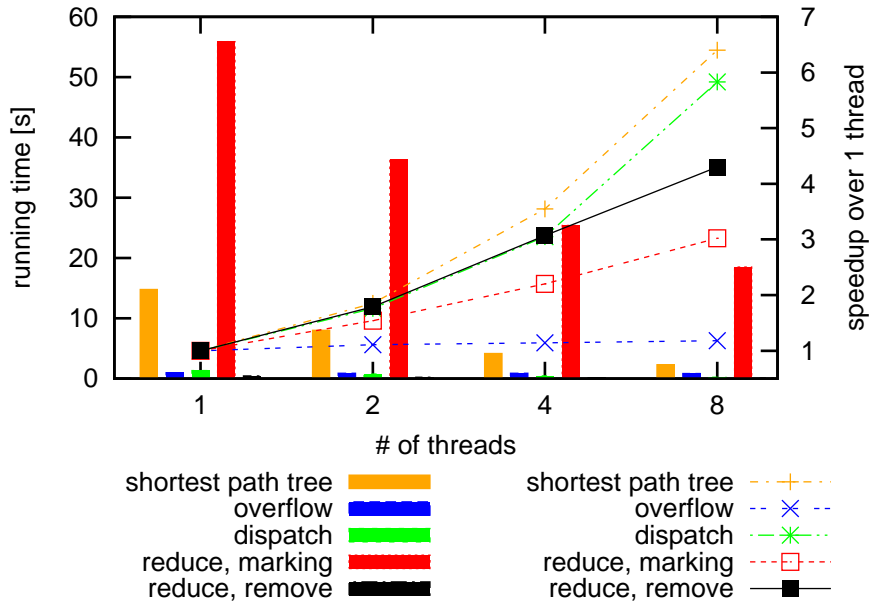


Figure 4: Running time shown as boxes on left axis and speedups shown as lines and points on the right axis. Speedups are in italics. Initialization times are not shown; They are significantly smaller than 0.1s. Compare Table 3.

iterate over all nodes and their outgoing edges, and do some simple operations on the edges.

The edge marking for the edge reduction step takes both a long time *and* only shows a low speedup of 3.02 with 8 threads. From here, we will focus on the load balancing of the edge marking reduction step.

4.2 Load Balancing Comparison

We measured the running time of the reduce edges marking step from individual nodes. The frequencies can be found in Table 4 and are visualized in Figure 5. This shows that the reduce edges marking step is relatively *easy* – taking less than 0.01 s – from more than 99.9% of all nodes. The number of *hard* nodes, where the marking step takes more than 0.1 s, is many orders of magnitude lower.

running time	absolute frequency	settled nodes limit	
		max	average
0 - 0.001	2'504'098	3'074	35
0.001 - 0.01	2'159	21'310	4'525
0.01 - 0.1	126	197'815	54'619
0.1 - 1.0	18	1'095'639	406'895
1.0 - 2.4	17	2'490'221	1'897'406

Table 4: Running times in seconds of the reduce edges marking step with frequencies and average number of settled nodes. Compare Figure 5.

If the hard nodes were distributed equally over the array of nodes this would not be a big problem. However, the node numbering is locality-sensitive. For most nodes, the identifiers (numbers) of the neighbor’s nodes are close to the number of the node itself. The node array is sorted ascendingly by the node numbers and this allows a cache efficient iteration over the nodes.

The hard nodes are mostly nodes on the coast, with adjacent edges representing long-distance ferry connections. Thus, there are a few “clusters” in which hard nodes are placed “close” to each other in the node array. Additionally, their node identifier is relatively high. Note that this is, by chance, a property of the input graph.

Thus, the hard nodes are not distributed equally among the threads but assigned to few of them. The dynamic load balancing separates the nodes into slabs of equal size and the largest cluster of hard nodes is assigned to the same slab.

Possible solutions to this problem are:

- *Continue using the dynamic load balancing but reduce the slab size.* While this solution circumvents the worst case, lots of atomic operations have to be executed. While these operations are relatively cheap compared to the work required for the hard nodes, they are relatively expensive in relation to the work for the easy nodes.

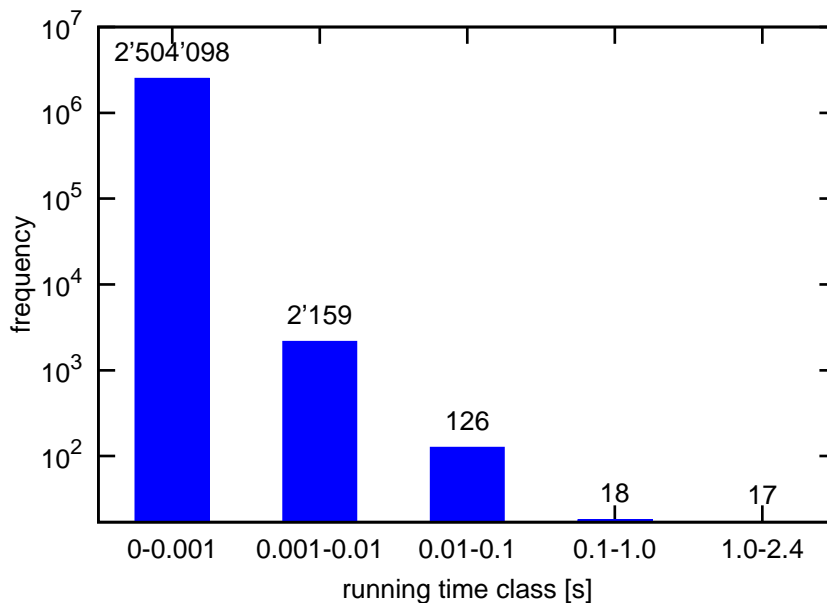


Figure 5: Distribution of the moved buckets' sizes after resizing for all levels. Note that the vertical axis is logarithmic. Compare Table 4.

- *Use guided load balancing with decreasing slab sizes.* It is not very bad if some threads get hard nodes in their first slabs. The processing time of the nodes that are behind the first hard node is relatively small compared to compared to the total outstanding work. If hard nodes are placed close to the end, it is relatively improbable that they are concentrated on few slabs. However, these considerations are only true if not all hard nodes are concentrated at the beginning. “Luckily,” for the guided variant, this is the case on our specific graph of Europe.
- *Use load balancing with work-stealing.* If the granularity is small enough, this should yield a good performance since only “few” nodes are reserved from the part, and cannot be stolen. Our experiments showed a best overall performance with a granularity of 1 for the graph of Europe.

Table 5 shows the running time of the four variants in the edge reduction marking step. For example, the static and dynamic load balancing variants only achieve a speedup of 1.61/1.54 with two threads while the guided and work stealing variant allow a speedup of 1.96/1.92.

While workstealing is a more sophisticated load balancing scheme than the guided variant, the latter shows a better speedup for 8 threads. The reason for this probably lies in the number of performed atomic operations. The guided variant begins with large chunk sizes and lets them shrink towards the end.

The work stealing variant has to do a lot of (atomic) work stealing: The work is first equally distributed between the threads but one thread gets most of the hard nodes.

load balancing	number of threads						
	1	2	4	8			
OpenMP static	55.8	34.6	<i>1.61</i>	22.5	<i>2.48</i>	18.9	<i>2.95</i>
OpenMP dynamic	55.9	36.3	<i>1.54</i>	25.2	<i>2.22</i>	18.5	<i>3.02</i>
OpenMP guided	56.4	28.8	<i>1.96</i>	15.1	<i>3.74</i>	9.6	<i>5.88</i>
MCSTL work stealing	56.3	29.3	<i>1.92</i>	15.3	<i>3.68</i>	10.2	<i>5.52</i>

Table 5: Running times for the reduce edges marking step on level 1 in seconds. Speedups are shown in italics. Compare Figure 6.

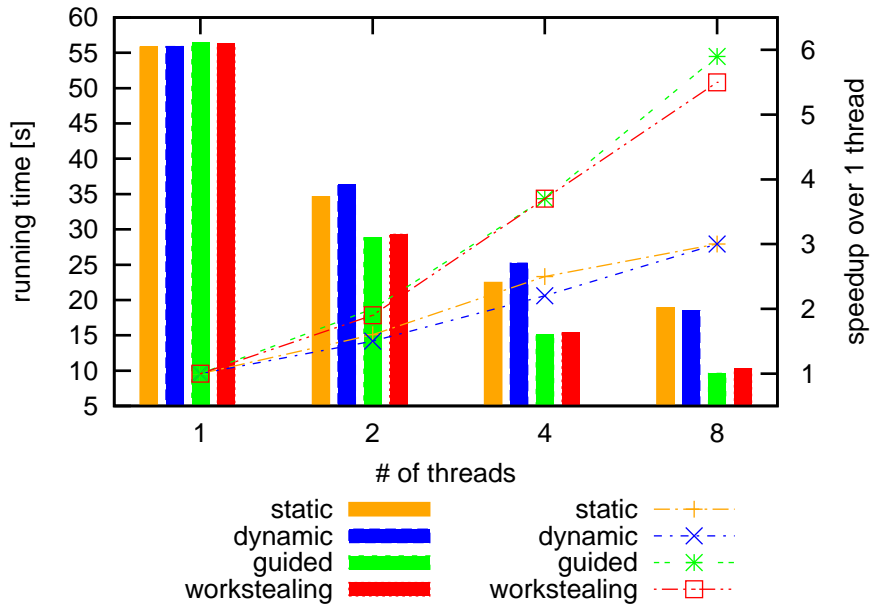


Figure 6: Running times for the reduce edges marking step on level 1 as boxes on the left axis and speedups as lines and markers on the right axis. Compare Table 5.

Parts of its work queue’s content will be stolen by other threads. These threads will then also process a hard node and again, their work is stolen.

Note that it is coincidence that the hard nodes have a high identifier and are thus processed in small blocks for the guided variant. If they were processed in earlier blocks, the guided variant would show no improvement over the dynamic variant. Instead, because of the higher number of atomic operations, it would show worse performance.

Nevertheless, even the highest speedup of 5.88 achieved with 8 threads is not very satisfactory for the problem at hand: The sequential portion of the marking step is limited to the atomic operations and a better speedup should be achievable.

Section 4.2.1 describes a technique to make the problem easier to load balance and increase speedups. Section 4.3 examines the role of the memory bus for the speedups. Section 4.4 describes an approach to improve the locality of the program in order to increase cache efficiency.

4.2.1 Making The Edge Reduction More Regular

Besides introducing load balancing for an irregular problem, it might also be worth trying to make it more regular. However, care has to be taken that changing the problem does not degrade the solution – in this case the highway hierarchies and the resulting query times.

For the edge reduction marking step, this means making the DIJKSTRA search from “hard” nodes easier. We modified the parallel variants to count the number of settled nodes in the DIJKSTRA search. When a certain limit is reached, the search is aborted. This might lead to reducible edges still being present in the graph. As a consequence, this could change the construction time of higher levels and increase the query time. However, the reduction itself does not change shortest path since it only removes superfluous edges. Even if not all of them are removed, the query results will still be correct.

We did experiments with limits of 100’000, 10’000 and 1’000. A limit of 100’000 effectively cuts off DIJKSTRA reduction searches taking longer than 0.1s. The other limits were chosen arbitrarily to examine how the construction varies with the settled nodes limit.

Effects on Construction Time. Table 6 shows the running times of the edge reduction marking step with different limits for the number of settled nodes. The speedups over one thread are shown in italics.

The data shows that when aborting reduction early on the hard nodes, the reduction times decrease. For one thread, it decreases by roughly 33s for all variants and for 8 threads, it decreases by 15s for the static and dynamic variant and by roughly 7s for the guided and work stealing variant.

Additionally, the problem is more regular and the load balancing gets better. The guided and work stealing load balancing variants get an almost linear speedup up to 4 threads. For 8 threads, a speedup of almost 7 is reached with the guided and with the work stealing variant. Lowering the limitation to 100’000 settled nodes is enough for the work stealing variant to achieve a speedup of 6.80.

variant	limit	number of threads						
		1	2	4	8			
OpenMP static	none	55.8	34.6	<i>1.61</i>	22.5	<i>2.48</i>	18.9	<i>2.95</i>
	100'000	29.7	16.4	<i>1.81</i>	9.2	<i>3.23</i>	6.5	<i>4.57</i>
	10'000	25.1	13.2	<i>1.90</i>	7.1	<i>3.54</i>	4.5	<i>5.58</i>
	1'000	22.3	11.5	<i>1.94</i>	6.1	<i>3.66</i>	3.9	<i>5.72</i>
OpenMP dynamic	none	55.9	36.3	<i>1.54</i>	25.2	<i>2.22</i>	18.5	<i>3.02</i>
	100'000	30.3	16.9	<i>1.79</i>	9.8	<i>3.09</i>	6.1	<i>4.97</i>
	10'000	25.1	13.5	<i>1.86</i>	7.5	<i>3.35</i>	4.2	<i>5.98</i>
	1'000	21.9	11.7	<i>1.87</i>	6.3	<i>3.48</i>	3.5	<i>6.26</i>
OpenMP guided	none	56.4	28.8	<i>1.96</i>	15.1	<i>3.74</i>	9.6	<i>5.88</i>
	100'000	29.8	15.1	<i>1.97</i>	7.9	<i>3.77</i>	4.5	<i>6.62</i>
	10'000	25.6	12.9	<i>1.98</i>	6.7	<i>3.82</i>	3.7	<i>6.92</i>
	1'000	21.8	11.0	<i>1.98</i>	5.7	<i>3.82</i>	3.1	<i>7.03</i>
MCSTL work steal	none	56.3	29.3	<i>1.92</i>	15.3	<i>3.68</i>	10.2	<i>5.52</i>
	100'000	29.9	15.3	<i>1.95</i>	7.9	<i>3.78</i>	4.4	<i>6.80</i>
	10'000	25.4	12.9	<i>1.97</i>	6.7	<i>3.79</i>	3.6	<i>7.06</i>
	1'000	21.9	11.1	<i>1.97</i>	5.7	<i>3.84</i>	3.1	<i>7.06</i>

Table 6: Running time of the edge reduction marking step in seconds for different limits for settled nodes when constructing the first level. Speedups are shown in italics.

The dynamic and static variant do not reach speedups that are better than 6.3 with 8 threads. This indicates that the problem still is very irregular and thus requires more sophisticated load balancing.

This data also shows that if we make the problem more regular using only a limit of 1'000 settled nodes then the work stealing and the guided variant yield equally good results.

The guided variant is relatively “lucky” on our graph of Europe since the hard nodes have high numbers and they are in the smaller chunks. If the nodes were numbered differently, it would show worse performance.

Effects on Query Time. The different load balancing variants only affected the construction time. The constructed multi-level overlay graph was the same for all of them. The settled nodes limitation, however, changes the created overlay graph: Some superfluous edges are still present in the constructed graph. However, there is no significant change in query time when introducing a settled nodes count limitation to either 100'000, 10'000 or 1'000.

4.3 Examination of Memory Bandwidth Limitation

The speedups described in Section 4.1 indicate that the resulting parallel code on the used machine does not scale well to more than four threads even with sophisticated load balancing. Such scaling problems can have multiple sources, e. g. false sharing. After checking the program for possible problems on the software side, we assumed that the small speedups beyond four threads were caused by limitations of the memory bus.

Test Software. To check whether our assumption is valid, another parallel program was written: Instead of collaborative threads, we considered the multiple parallel execution of the sequential program.

The main program created p threads. Each thread loaded the graph into memory so there was no shared data between the threads. Each thread waited for all other threads to finish loading the graph. Then, the threads started the construction of the graph and the time for the construction was stopped.

Test Results. The test software was run six times: With one thread, with two threads on different processors, two threads on the same processor but on different L2 caches, two threads on the same processor and the same L2 cache and four threads, one thread on each L2 cache. Finally, it was run with 7 threads (the machine did not have enough memory to run 8 copies).

The results are shown in Table 7.

Explanation. First, consider the third and the fourth run: Two threads on the same processor but on different L2 caches versus two threads on the same L2 cache. The performance degradation from different cache to one shared cache is less than 5s. The only difference between the two runs is that they now share the same L2 cache; they still have the same connection to the memory bus.

Note that the slowdown only is around 1-4%. Thus, the influence of sharing the L2 cache are very low. Additionally, the time lost due to cache effects because of sharing L2 caches will not increase when using more cores.

Now, consider the running times from one thread to two threads on different processors. The degradation is around 5s, i. e. 4-5%. When the two threads are run on the same processor but different L2 caches, the performance gets worse a bit again. An explanation for this is that the bus arbitration works better for two cores on different processors than for two cores on the same processor.

The performance gets worse again when considering one or two threads on different L2 caches to 4 threads on different L2 caches. The total running time is 22% higher than with one thread. Since the only resource they share is the bus, it must be the bottleneck for the program.

We also considered the case where a maximum number of cores executes the sequential program. The program only fits seven times into main memory. As can be seen in this data, the performance degrades again and the total running time is 58% longer.

description		running time
one thread	thread 1	109.3
2 threads, different processors	thread 1	114.1
	thread 2	115.3
2 threads, same processor, different L2 caches	thread 1	115.9
	thread 2	118.4
2 threads, same processor, same L2 cache	thread 1	119.8
	thread 2	119.9
4 threads, different L2 caches	thread 1	126.2
	thread 2	127.6
	thread 3	131.3
	thread 4	133.8
7 threads ^a	thread 1	162.2
	thread 2	164.6
	thread 3	166.6
	thread 4	167.5
	thread 5	169.5
	thread 6	170.8
	thread 7	173.0

^aEight copies of the graph do not fit into the main memory of the machine we did the experiments on.

Table 7: Experimental results for the memory bus testing. Running times are in seconds.

This degradation can only partly be explained by cache effects. The individual steps executed by the sequential program are the same as the ones of the parallel program for most of the running time. Because the threads in the parallel program collaborate on the same data, the cache contention will not get worse from the multiple instances of the sequential program. Thus, these experiments support our assumption that memory bus saturation is the reason for the relatively low speedups observed in Section 4.1. Section 4.4 shows an approach to improve the program’s locality and thus the construction time.

We can also extrapolate an educated guess for the speedup with collaborative threads: 7 parallel programs take a total of 173.0s to run, the sequential program takes 109.3s. We could interpret “7 times the work in 173.0s” as “1 times the work in $(173.0/7)s = 24.7s$ ” which means an overall speedup of $109.3/24.7 = 4.43$. Of course, we also have to expect some overhead for initializing the DIJKSTRA search object and some speedup due to cache effects since the threads collaborate on the same data. The speedup of 4.43 with 7 threads is equivalent to an efficiency of $4.43/7 = 0.63$.

4.4 Improving Locality

Since Section 4.3 indicates that memory bandwidth limits the program’s performance, it is worthwhile to consider improving memory locality, i. e. to promote cache usage and to reduce memory access.

Thus, we implemented a “striped” load balancing variant. This variant works like the dynamic load balancing variant but each chunk is assigned to two threads. The threads run on the two cores sharing the same L2 cache. The first thread works on all elements of the chunk with an even index, the other thread processes the elements with an odd index.

Since the nodes are sorted by their index in the node array and the numbering is local in a certain sense, there is the hope that the search space of the DIJKSTRA searches overlap and the overlapping memory only has to be loaded once.

Table 8 shows the resulting running times. For one thread, the striped variant is identical to the dynamic load balancing.

limit	number of threads						
	1	2	4	8			
none	55.9	36.4	<i>1.54</i>	23.1	<i>2.42</i>	16.9	<i>3.31</i>
100'000	30.3	17.0	<i>1.78</i>	9.5	<i>3.19</i>	5.8	<i>5.22</i>
10'000	25.1	13.4	<i>1.87</i>	7.1	<i>3.54</i>	4.1	<i>6.12</i>
1'000	21.9	11.5	<i>1.90</i>	6.0	<i>3.65</i>	3.5	<i>6.26</i>

Table 8: Running times of the striped load balancing variant for the reduce edges marking step in seconds. Speedups over 1 thread are shown in italics.

Compare these these running times and speedups with the ones from Table 6: As can be seen, the running times are slightly better than with the dynamic load balancing. However, the running times are still not better than the more sophisticated load balance variants guided and work stealing.

Note that future chips will no longer have two cores sharing one L2 cache. Instead, each core will have its own L2 cache (as they already have their own L1 cache) and its own memory controller.

However, the *striped* load balancing variant can still make sense when considering *simultaneous multithreading* (SMT). Different from providing multiple complete cores on a chip, processors using SMT only appear as two different processor cores. Internally, they only have one set of execution units but provide multiple duplicates of the architectural state of the processor (registers, instruction buffers etc).

Whenever one virtual processor is stalled, the other one can be activated. Reasons for stalling include cache misses (i. e. accessing a higher level of the memory hierarchy) and synchronization. Intel’s Hyperthreading is an example for SMT. See [3] for more information regarding SMT.

4.5 Overall Performance

So far, we have only considered the creation of the first level and then focused on improving the edge reduction marking step. We will now consider the overall speedup with the different load balancing variants. Table 9 shows the total construction time of the multi-level overlay graph and speedups with a varying number of threads and different load balancing methods. Figure 7 visualizes the numbers for the variant with a settled nodes limitation of 1'000 and the variant without any limitations. The total construction time also includes all initialization required for the multi-level overlay graph creation.

load balancing	limit	number of threads						
		1	2	4	8			
OpenMP static	none	116.7	70.6	<i>1.65</i>	45.7	<i>2.55</i>	36.2	<i>3.22</i>
	100'000	91.2	52.4	<i>1.74</i>	32.5	<i>2.81</i>	23.7	<i>3.85</i>
	10'000	84.0	47.7	<i>1.76</i>	28.7	<i>2.93</i>	20.4	<i>4.12</i>
	1'000	81.0	45.1	<i>1.80</i>	27.4	<i>2.96</i>	19.7	<i>4.11</i>
OpenMP dynamic	none	116.6	72.2	<i>1.61</i>	48.3	<i>2.41</i>	35.5	<i>3.28</i>
	100'000	93.2	53.9	<i>1.73</i>	32.7	<i>2.85</i>	22.8	<i>4.09</i>
	10'000	83.8	48.3	<i>1.73</i>	29.0	<i>2.89</i>	19.9	<i>4.21</i>
	1'000	80.1	45.4	<i>1.76</i>	27.5	<i>2.91</i>	19.0	<i>4.22</i>
OpenMP guided	none	117.3	65.5	<i>1.79</i>	37.4	<i>3.14</i>	26.3	<i>4.46</i>
	100'000	91.2	50.6	<i>1.80</i>	30.0	<i>3.04</i>	20.6	<i>4.43</i>
	10'000	85.6	46.8	<i>1.83</i>	28.0	<i>3.06</i>	19.3	<i>4.44</i>
	1'000	79.9	44.4	<i>1.80</i>	27.0	<i>2.96</i>	18.6	<i>4.30</i>
OpenMP striped	none	116.9	72.6	<i>1.61</i>	53.6	<i>2.18</i>	37.9	<i>3.08</i>
	100'000	92.2	54.2	<i>1.70</i>	40.5	<i>2.28</i>	26.6	<i>3.47</i>
	10'000	83.7	48.2	<i>1.74</i>	36.6	<i>2.29</i>	23.9	<i>3.50</i>
	1'000	79.7	45.9	<i>1.74</i>	35.2	<i>2.26</i>	24.3	<i>3.28</i>
MCSTL work stealing	none	116.4	63.8	<i>1.82</i>	37.3	<i>3.12</i>	26.5	<i>4.39</i>
	100'000	91.4	51.4	<i>1.78</i>	29.6	<i>3.09</i>	20.2	<i>4.52</i>
	10'000	84.0	46.3	<i>1.81</i>	27.6	<i>3.04</i>	18.9	<i>4.44</i>
	1'000	78.2	44.3	<i>1.77</i>	26.5	<i>2.95</i>	18.3	<i>4.27</i>

Table 9: Construction times in seconds and speedups over 1 thread for creating the whole multi-level overlay graph. Compare Figure 7.

We can observe the following: As expected, the work stealing and guided variants achieve the best speedups. With a settled-nodes limitation of 1'000, the differences in speedup become smaller. However, the problem is still not regular and investing overhead into load balancing still pays off.

The settled nodes limitation does not show a big influence on the speedup for the

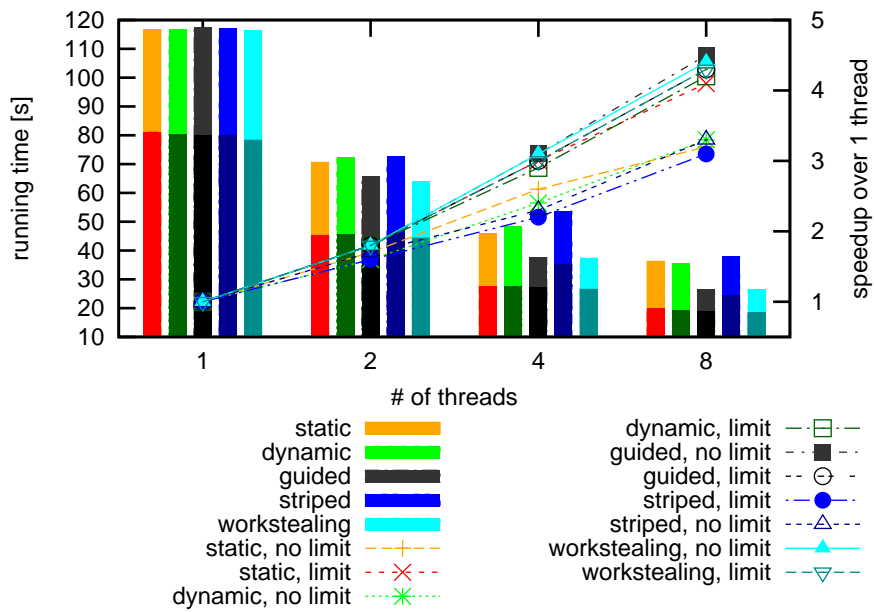


Figure 7: Total construction time shown as boxes on left axis and speedups shown as lines and dots on right axis. The darker parts of the bars represent the running time of the variant with a limitation of 1'000 settled nodes. The whole bars represent the running time without any limitation. Compare Table 9.

guided and the work stealing variant. The reason for this probably is that the percentage of time spent reducing edges gets lower as the reduction DIJKSTRA searches get aborted earlier. Instead, other parts of the construction get more dominant. These parts do not show much more potential to get parallelized. Also note that the levels above level 4 contribute 2s to the running time with 8 threads which is 7.5% of the parallel running time. These levels are relatively small. For such small problem instances, parallelization is not as likely to show good speedups as with larger instances.

In Section 4.3, we estimated an efficiency of 0.63 (for 7 threads). The expected speedup based on the efficiency estimate is $0.63 \cdot 8 = 5.04$. This is above the achieved speedup but the difference can be explained with initialization times, memory bus limitations, irregular load balancing and synchronization overhead for load balancing.

5 Discussion and Future Work

We presented a shared-memory parallel version of the precomputation for Highway-Node Routing.

The irregularity of the problem was experimentally examined for the road network graph of Europe and a simple solution was presented to make it more regular: The number of settled nodes is limited, which yields better speedups and does not significantly change the query time.

Multiple variants of load balancing were considered: OpenMP's built-in variants for parallelizing loops and the parallel *for_each* provided by the MCSTL.

We also presented the *striped* load balancing variant which tries to take advantage of shared caches (the L2 cache on our machine).

The achieved speedups were 3.14 for 4 threads and 4.46 for 8 threads.

Some problems are still open:

Parallelization. A combination of work stealing and striped processing would be interesting.

At the time of writing, shared memory systems with more than eight cores are very expensive. Although future machines will have more cores and NUMA machines will make the memory bottleneck less problematic, a parallelization to multiple machines is also of interest. As a shared memory parallel version exists, a distributed shared memory version seems to be the simplest extension. A message passing based variant would make sense, too.

Functionality. The parallelization is especially interesting for the dynamic variant and there already is a sequential dynamic implementation. However, the current parallel version only works for static Highway-Node Routing. Thus, the dynamic variant should be incorporated into the parallelized program.

References

- [1] OpenMP Website (<http://www.openmp.org>), 12 2007.
- [2] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, Order Number: 253666-025US*, November 2007.
- [3] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, May 2002.
- [4] OpenMP Architecture Review Board. *OpenMP Application Program Interface, 2.5 edition*, May 2005.
- [5] D. Schultes and P. Sanders. Dynamic highway-node routing. In *6th Workshop on Experimental Algorithms (WEA)*, LNCS 4525, pages 66–79. Springer, 2007.
- [6] Dominik Schultes. Fast and exact shortest path queries using highway hierarchies. Master's thesis, Universität des Saarlandes, 2005.
- [7] Johannes Singler, Peter Sanders, and Felix Putze. The Multi-Core Standard Template Library. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 682–694. Springer, 2007.