# Parallelizing a CAD Model Processing Tool from the Automotive Industry

Luis Ayuso[1], Herbert Jordan[1], Thomas Fahringer[1],
Bernhard Kornberger[2], Martin Schifko[3],
Bernhard Höckner[1], Stefan Moosbrugger[1], and Kevin Verma[3]

[1] University of Innsbruck, Institute of Computer Science, Innsbruck, Austria
{luis,herbert,tf}@dps.uibk.ac.at,
{Bernhard.Hoeckner,Stefan.Moosbrugger}@student.uibk.ac.at
[2] Geom Softwareentwicklung, Graz, Austria
bkorn@geom.at
[3] ECS, Magna Powertrain, Steyr, Austria
Martin.Schifko@ecs.steyr.com

**Abstract.** Large industrial applications are complex software environments with multiple objectives and strict requirements regarding standards, architecture or technology dependencies. The parallelization and optimization of industrial applications can be an intrusive modification of the source code which increases the development complexity. In this paper we describe the analysis and modifications applied to an industrial code from the automotive industry, named *Merge*, with the goal to detect and exploit parallelism in order to reduce the resulting execution time for shared memory parallel architectures. As part of this effort we tried to maximize the potential for an effective parallelization, nevertheless preserving the original algorithm and the code features as far as possible. Reasonable speedup has been achieved on a shared memory parallel architecture. Furthermore, additional potential has been located for future parallelization and optimization work.

## 1 Introduction

Performance-oriented development of scientific and industrial applications for parallel architectures is a time-consuming and tedious process that involves many cycles of editing, compiling, executing, and performance analysis. This paper summarizes the experience acquired during the parallelization and optimization process of the *Merge* application which is being developed in C++ by Martin Schifko (ECS, Engineering Center Steyr GmbH & Co KG) and Bernhard Kornberger (Geom e.U.). ECS develops different software solutions for the automotive industry.

The *Merge* application is a fully automatic repair-, connect-, and re-meshing-tool for triangular meshes. It fulfills the high requirements in automotive industry and is of great interest because it can significantly reduce the development cycle of automotive industry products. The process of merging and possibly repairing

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part I, LNCS 8805, pp. 24–35, 2014.

and remeshing triangular meshes is nowadays still often performed manually. Involving an expert is time consuming and error-prone without an analysis tool like *Merge*.

The result of a *Merge* application execution – among other uses – is utilized by electrophoretic deposition simulation (e.g., *ALSIM* software[1] developed by ECS) which requires as input a single mesh object representing an entire car. In this scenario, the *Merge* application will operate on thousands of individual car body parts to produce a unique car mesh. The execution will also enforce some characteristics of the processed meshes: e.g. non-manifold meshes will be repaired, self-intersections will be avoided while maintaining the surface shape. The geometrical details of the merge process are covered by Schifko et al.[2].

Exact evaluation of geometric predicates, vital for the robustness of the Merge application, is achieved using the 3D linear geometry kernel of the Computational Geometry Algorithms Library (*CGAL*)[3], not only at the price of execution time overhead, it also prevents from applying GPGPU techniques.

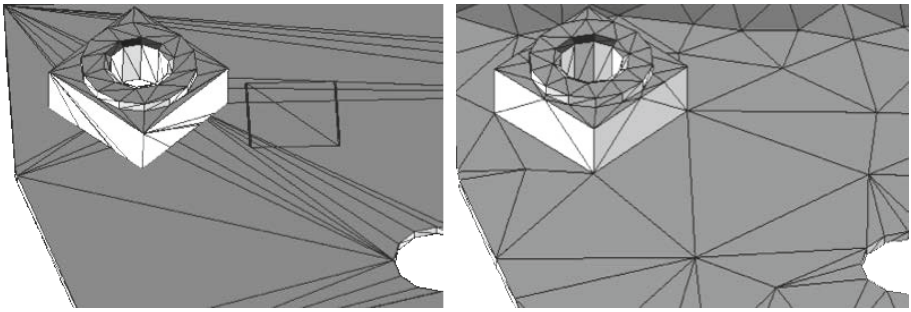
In this paper we start with a description of the *Merge* application and then describe the original parallelization and identify drawbacks. Based on this initial effort, we established clear principles for parallelization which have been applied in an improved parallelization of the *Merge* application. As a first step, we isolated code regions with potential for parallelism in the overall application which improved the modularity of the code and simplified ongoing development of *Merge*. The intended philosophy of any code modification was to interfere as little as possible with the core functionality of the program while reducing the size of code regions dedicated to parallelism management. Modifications applied to specific code regions should enable and expose parallelism without changing the original algorithmic properties.

To improve the overall parallelization and thereby reduce the computation time, we applied three optimizations comprising nested parallelism to mitigate load imbalance, reduction operations on demand to overlap map and reduce operations, and tuning OpenMP thread management to deal with the overhead of large number of threads. Experiments with several input data sets for the *Merge* application result in a speedup of up to 5.59 (when using 16 cores) compared to sequential executions.

Section 3 describes the original parallelization strategy for *Merge*. A new parallelization and optimization approach for *Merge* is explained in Section 4. Section 5 presents numerous experiments to demonstrate the effectiveness of the proposed parallelization and optimization techniques with several input data sets. Related work is discussed in Section 6 followed by Conclusion and Future Work (Section 7).

## 2 Architecture of the *Merge* Application

Fig. 1 shows a detail of a typical *Merge* application execution. Fig. 1a shows the superposition of different volumes, each of them defined by a different CAD file. Fig. 1b shows a single object which covers the volume corresponding to the union of Fig. 1a volumes.



**(a)** Example of Merge input meshes **(b)** Single output mesh

**Fig. 1.** Merge operation example

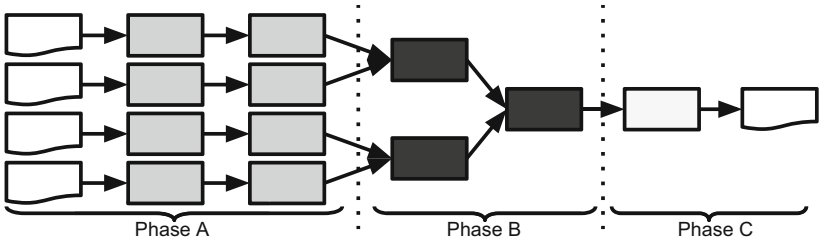



**Fig. 2.** *Merge*'s implicit task dependency graph

Fig. 2 shows the tree structure which intuitively describes the internal structure of the *Merge* application. This figure reveals that the internal structure fits a MapReduce pattern [4]. Each node of this tree corresponds to a certain task e.g. Repair, Offset, Merge, Export, IO. Tasks are internally organized as a sequence of algorithms, arranged and parametrized to solve a specific functionality. The term *task* refers to the execution of a sequence of algorithms on one or two data elements, and produces a single output element. Each of those data elements are loaded from a CAD file and referred to as *mesh* throughout the remainder of this paper. The *Merge* application operates on an input data set which contains a series of different CAD files. These data sets are processed by three phases:

- Phase A: Loading and preparing the input. Input meshes are read from files and then a series of algorithms is applied to each of them.
- Phase B: Merging. After the pre-process phase, meshes are pairwise merged, until a single resulting mesh is obtained. Notice that this is the actual merge operation. Although the application is called *Merge*, it actually executes several algorithms contributing additional services as part of the overall commercial solution.
- Phase C: Cleanup and export. The final resulting mesh is cleaned up such that parametrized geometrical constraints are enforced. In the end, the mesh is written into a file.

**Algorithm 1.** Baseline Merge Algorithm (proper synchronization implied)

**Input:** $M \ldots$ list of meshes

```
queue ← schedule(M)
begin parallel
  while (queue ≠ ∅) do
    t ← retrieve first runable task from queue
    process t
  end while
end parallel
```

As shown in Fig. 2, dependencies between tasks constrain their execution order.

## 3 Initial Parallel Structure

The initial *Merge* implementation, which provided the base line of our optimization and tuning efforts, has been utilizing the available parallelism on a task level, as summarized by Algorithm 1..

Algorithm 1. consists of a scheduling and an execution step. In the first step, the input data set is mapped to a list of tasks, ordered according to their dependencies, and stored in a queue. In the second step, a team of concurrently running threads retrieve tasks from the queue and executes them in parallel. When all tasks are done, the process is complete.

The basic idea of the *schedule* procedure is to instantiate the necessary tasks, determine dependencies and compute (implicitly) a topological order of the resulting task-dependency graph – as illustrated in Fig. 2. The main source of influence on the performance is to determine the order in which meshes are combined in Phase B. The corresponding operations are associative and commutative. The meshes produced by Phase A may therefore be reduced to a single mesh using an arbitrary order.

In the intention to keep the critical path length short, the scheduler of the base implementation realizes a binary reduction resembling a balanced, binary tree. Furthermore, since the computational costs of the operations in Phase B are (partially) depending on the number of triangles of the input meshes, a heuristic keeping the size of intermediate results low is utilized for its generation.

### 3.1 Performance Analysis

To gain insides on the behavior of the utilized algorithms and scheduling policies we traced the processing of different data sets on a parallel architecture. Fig. 3 illustrates a typical pattern that is observable when processing 6 meshes on a 4 core system. The CPU time required by tasks of Phase A are varying by up to an order of magnitude. Naturally, the longest execution of those tasks delay tasks

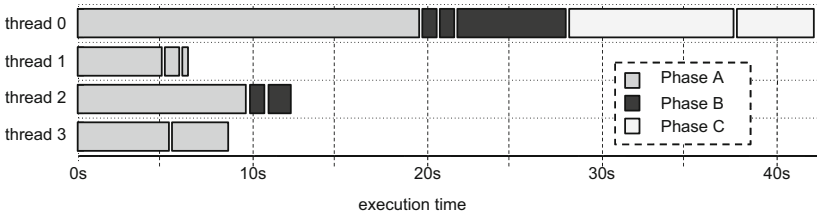



**Fig. 3.** Example task distribution of the baseline *Merge* implementation

of Phase B, which is made worse by the balanced nature of the reduction tree. Finally, the tasks of Phase C, whose constraints demand a sequential execution, are responsible for roughly a fifth of the sequential execution time. The best experimentally observed speedups have been $\sim 2.5$ for a large number of input files on an 8 core system.

### 3.2 Improvement Potential

The information gained from the initial analysis lead us to focus on two approaches for increasing *Merge*'s performance:

- *Nested Parallelism* – the low resource utilization due to the high load imbalance in Phase A and the sequential Phase C could be improved by unveiling and utilizing nested parallelism within the individual tasks
- *Dynamic Reduction* – the constraint imposed of a balanced binary reduction tree could be dropped by merging temporary results according to the (dynamic) order they are completed, thereby reducing the time threads are stalled due to unfinished temporary results

The implementation and application of corresponding techniques is the topic of the following section.

## 4 New Parallelization and Optimization Strategy

To improve the performance of the code by increasing its resource utilization we applied a series of steps as covered in the following subsections.

### 4.1 Step 1: Nested Parallelism

Nested parallelism can effectively help reducing the load imbalance by lowering the granularity of the workload to be distributed among the available resources. Unfortunately, to benefit from this, parallelism within the individual tasks has to be identified and utilized.

Essentially there are two potential approaches for speeding up a single task being applied on a single input mesh. Either the operations to be applied themselves are distributed among parallel resources or the input mesh is partitioned

into smaller meshes, which then are processed concurrently. However, due to the nature of the applied algorithms, the integrity of the processed meshes is essential. Consequently, only the first approach could be pursued.

Fortunately, meshes themselves are essentially just collections of triangles. Hence, many of the applied procedures within the involved tasks offer good opportunities for data parallelism. The profile of the execution helped to identify loops dominating the execution time to obtain worthwhile candidates for parallelization. Each of those were inspected for dependencies preventing their iterations to be processed concurrently and if present, the alternatives were studied. In general, three types of loops were found:

- *embarrassing parallel loops* – where each iteration is working fully independently of any other iteration. For instance, frequently an operation has to be applied on all triangles of a mesh or all the edges or points referenced by those. An extended, yet almost equally simple to handle, variant is a reduction loop aggregating values computed by iterations into an overall result utilizing a commutative and associative operator. The parallelization of this kind of loops is straightforward.
- *parallel loops with false dependencies* – where each iteration is conducting operations on a shared data structure, e.g. the processed mesh, while those operations have no effect in the operations of the remaining iterations. The dependency between loop iterations is only introduced by the utilized containers and index structures. Those dependencies are not essential for the algorithm, yet prohibit a parallel execution. Typically, adding fine grained locking and atomic operations to those data structures would help resolving the problem, yet a modification of elements producing this dependencies, and implemented on third-party codes was not feasible.
  For the loops fitting in this category, the parallelization was implemented by separating the identification of manipulations to be conducted on the shared data structure from the actual application. The first step can so be conducted in parallel and results in a list of closures encapsulating manipulation operations. Those operations are then applied in a second step sequentially. Occasionally, identified update operations where sorted such that parts of them could be applied concurrently.
- *inherently sequential loops* – where each iteration is depending on the results of the immediate previous iteration. In those cases no parallelization could be done and other code sections dominating the execution time were considered.

The scheduling policies to be utilized for the parallelized loops were also studied. In general, the identified loops exhibit a rather irregular workload distribution among its iterations and even for regular loops a *dynamic loop scheduling scheme* proved to be most profitable due to the fact that their computation happens in a nested context including threads of other tasks competing for resources.

**Algorithm 2.** Improved Merge Algorithm (proper synchronization implied)

**Input:** $M \ldots$ list of meshes

```
stack ← ϵ
for all m ∈ M do
  r ← phaseA(m)
  while (stack ≠ ϵ) do
    r ← phaseB(r, pop(stack))
  end while
  push(stack, r)
end for
return phaseC(pop(stack))
```

In general, those parallelization steps have been rather straightforward. The biggest lesson that could be drawn from this process is the fact that OpenMP and C++ idioms are sometimes in conflict. In particular OpenMP's restriction on the structure of for loops, including the requirement of *random access iterators* not provided by all types of collections and the (current) exclusion of the C++11[5] for-each loop are minor, yet cumbersome restrictions. More severe issues, however, are implied by object oriented design patterns and the associated principle of *information hiding* making it difficult to identify dependencies and potential race conditions in manipulated code.

Those issues are not unknown in general, yet our experience provides further evidence for those and a motivation to provide improved tool support and/or increased flexibility in the future.

### 4.2 Step 2: Reduction on Demand

This section focuses on the top-level structure of the parallel implementation which has been summarized in Algorithm 1.. The separation of the scheduling step and the execution is prohibiting the consideration of dynamically obtained information into the organization of the reduction operation of Phase B. Furthermore, load management responsibilities have been explicitly included in the application code by the utilization of the included (global) *queue* and the imposed work-fetching scheme. Both activities could and should be delegated to the runtime system.

The baseline implementation was utilizing *boost threads* for the top level parallelism, resulting in a mixture of runtime systems when combined with the nested OpenMP based parallel codes. Hence, in a first step the top level parallelism has been restructured to utilize OpenMP as well – thereby providing the OpenMP runtime system the full control on thread spawning and management decisions.

In a second step *schedule* function and the *queue* were replaced by an eager reduction scheme summarized by Algorithm 2. based on a parallel for-all loop. In this approach meshes that are ready for being included in the reduction step

of Phase B are put on a *stack*. Whenever a mesh $r$ is yield by Phase A, the current result $r$ is combined with all available meshes stored in the stack. Thus constituting a dynamic reduction scheme combining meshes as soon as they are available.

Unfortunately, like the baseline implementation, our approach encodes an explicit scheduling policy for the reduction operations. Ideally, this responsibility would also be delegated to the runtime system by utilizing an *OpenMP custom reduction* incorporated in version 4.0. However, since the *Merge* application is targeted to be compiled by the *Microsoft* C++ compiler and this version is not yet supported an ad hoc implementation was required.

On the other hand, explicit scheduling policies do provide the opportunity to integrate heuristics to manage the reduction process. The base implementation aimed for minimizing the size of temporary results while our approach is an eager reduction not considering any algorithm specific traits and hence risking and increasing the computational complexity of the overall process. Yet, as will be demonstrated in the experimental section, the latter approach resulted in faster execution times for the investigated scenarios.

**Table 1.** Experiment inputs

| | orbit | cubes | carPart | cylParts |
|---|---|---|---|---|
| Number of input files | 8 | 44 | 32 | 4 |
| Total # triangles | 3.1MB | 9MB | 37.24MB | 0.24MB |
| Sequential time | 75.8 | 780.99 | 1354.53 | 21.22 |

**Table 2.** Original MapReduce (seconds)

| | Original | | | | Step 1 | | | |
|---|---|---|---|---|---|---|---|---|
| Cores | Orbit | Cubes | carPart | CylParts | Orbit | Cubes | carPart | CylParts |
| 2 | 56.00 | 456.97 | 1191.92 | 19.00 | 43.3 | 469.0 | 780.1 | 13.5 |
| 4 | 48.84 | 342.05 | 1021.23 | 18.54 | 28.7 | 318.5 | 610.5 | 11.9 |
| 8 | 46.51 | 310.96 | 860.11 | 18.48 | 20.1 | 243.9 | 374.9 | 6.7 |
| 16 | 46.90 | 315.35 | 992.96 | 18.61 | 18.6 | 535.7 | 361.6 | 11.6 |

**Table 3.** Reduction on demand (seconds)

| | Step 2 | | | | Step 3 | | | |
|---|---|---|---|---|---|---|---|---|
| Cores | Orbit | Cubes | carPart | CylParts | Orbit | Cubes | carPart | CylParts |
| 2 | 35.19 | 431.64 | 671.32 | 10.32 | 34.0 | 450.1 | 741.4 | 9.2 |
| 4 | 26.49 | 271.29 | 479.19 | 9.11 | 26.1 | 308.3 | 397.1 | 6.8 |
| 8 | 25.50 | 263.07 | 454.28 | 8.35 | 15.4 | 217.6 | 269.7 | 5.0 |
| 16 | 25.78 | 269.24 | 456.99 | 9.1 | 16.8 | 289.3 | 238.2 | 6.8 |

### 4.3 Step 3: Tuning OpenMP Thread Management

For the code version exploiting nested parallelism, the analysis of the experiments conducted on a variety of configurations – exhibiting a different number of cores – shown diminishing results for larger scale systems (see Section 5). An investigation of those results revealed that this effect could not only be attributed to higher synchronization overhead caused by increased lock contention. Instead the nested parallel code led to an increase in the number of concurrently active threads competing for CPU time – a number that grows exponentially with the nesting level. The default policy of OpenMP[6] – to create thread groups consisting of the same number of threads as physical cores available on the system – amplifies this effect on larger scale systems. As an example, a 3-level nested parallel code section leads to the creation of up to 64 threads on a 4-core system, 512 threads on an 8 core system, and up to 4096 threads on a 16 core system. Despite having a large number of threads is beneficial for overcoming load-balancing issues, a too large number of threads leads to congestion and hence overhead introduced by the OS-level scheduler.

Consequently, a fine-tuning of the thread spawning policy was required. One widely utilized possibility is to integrate the thread spawning control into the application code in order to determine the number of threads – during runtime – to be utilized to execute concurrent regions [7]. Yet this would have led to dependencies between otherwise unrelated parts of the code, solely introduced for the thread management. Also, it would require the integration of an internal mechanism to monitor the parallel execution of the application.

Fortunately, an alternative is provided by OpenMP by offering parameters for tuning thread-spawning policies. While limiting the maximum number of nested parallel levels is less applicable, since the required nesting level is depending on the current state of the execution, parameters limiting the maximum number of simultaneously active threads have turned out to be most beneficial. Preliminary investigations resulted in acceptable performance improvements for larger systems when fixing this value to twice the number of available cores. This threshold value corresponds to the best speedup for most of the tests inputs. Yet, we suspect that smarter policies may lead to even higher performance. This particular problem, regarding on how to manage thread spawning policies in highly unbalanced, irregular codes is the topic of future investigations.

## 5 Evaluating Results

The experiments described in this section have been executed on a two socket Intel Xeon Processor E5-4650 Sandy Bridge EP. A fill-socket-first affinity policy was adopted. The source code was compiled with GCC 4.8.2 with optimization level -O3 and executed on Fedora 19 with Linux Kernel 3.12.11.

The experiments were conduced using 4 different data sets (Table 1), each of them covers a different number of inputs. The data sets were designed to stress certain regions of the code during development and to expose different behaviors regarding performance.

Table 2 shows runtime measurements for two configurations of the original MapReduce. The columns named *Original* correspond to the original parallel implementation. Step 1 stands for the *OpenMP* implementation of the approach using nested parallelism. Step 1 column exhibits lower execution times up to 8 cores, while for a higher number of cores, competition between threads deteriorate performance (as detailed in Section 4.3). E.g. the *Cubes* data set shows a longer execution time for Step 1 (16 cores) than the original approach.

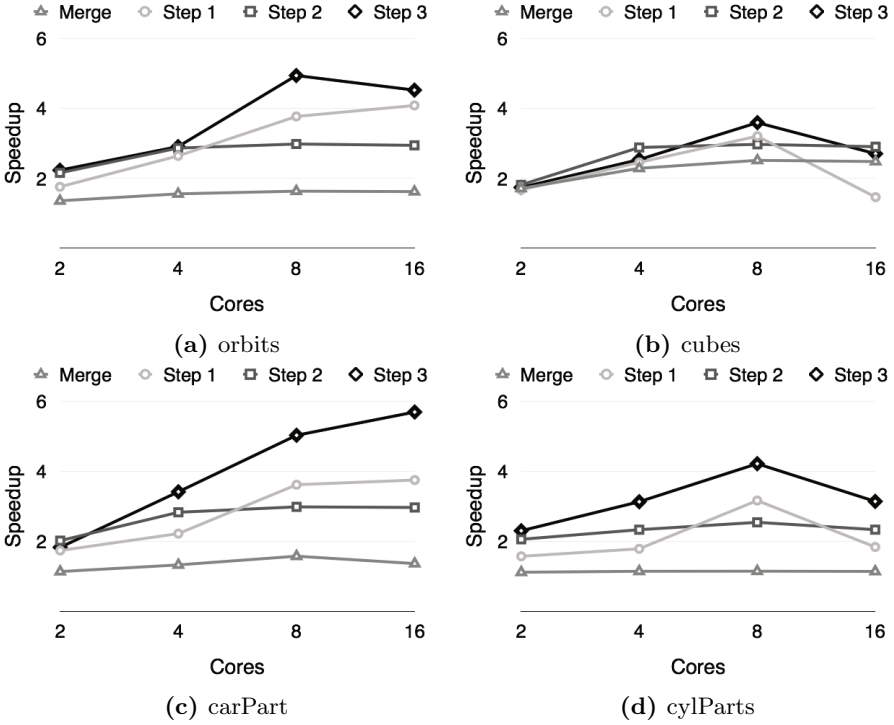
Execution time measurements for the described reduction on demand approach are shown in Table 3. Step 2 column presents the measurements for the nested parallel implementation using dynamic reduction. The column Step 3 presents data corresponding to executions of the same implementation as in Step 2 with a different configuration: the maximum number of spawned threads was bounded to two times the amount of cores used. Step 3 implementation excels at scenarios with bigger input file sizes (like *carPart* data set), those executions can benefit of a less congested system where long time executing tasks do not need to compete with others. Fine grained scenarios (like *Cubes* data set) consist of short time running tasks which are less vulnerable to congestion, therefore – for these data sets – step 3 technique will present a non-monotonic behavior, leading to poor results on larger systems.

Fig. 4 illustrates the achieved speedup of each approach by comparing it to a sequential execution, derived from execution times shown in Tables 2 and 3. The speedup measurements show that the thread limited execution of the dynamic reduction approach (step 3) excels for most of the input data sets. A different behavior is shown for the *cubes* data set: a large number of uniformly sized small meshes (compared to the other data sets) benefits the original implementation. As described in Section 3, the original implementation of Phase A can benefit of a balanced workload when utilizing equally distributed mesh sizes. This feature of the *cubes* data set leaves little room for improvement, although speedup was achieved for all configurations with the nested parallel approach that uses a dynamic reduction (step 2).

## 6 Related Work

The MapReduce pattern provides the foundation for the *Hadoop* framework. The porting procedure of a C++ development into *Hadoop* is analyzed in Gudmundsson et al.[8]. C++ codes needed to be wrapped into the *Hadoop* Java framework loosing the opportunity to expose finer granularity parallelism. Distributed MapReduce developments lack of the capability of exploiting nested parallelism[9] which is our main contribution.

C++11 [7] provides of native mechanisms to exploit concurrency which on the one hand can lead to efficient and portable parallel implementations. On the other hand, managing parallelism and concurrent tasks synchronization needs to be addressed by the programmer. The solutions provided along this paper (specially nested parallelism) require a global management of concurrent tasks, which according to the C++11 model would require an ad hoc implementation.




**(a)** orbits **(b)** cubes

**(c)** carPart **(d)** cylParts

**Fig. 4.** Speedup: comparison of different approaches

Regarding congestion experienced with nested parallelism, Tanaka et al.[10] substitute the standard implementation of *OpenMP* by a *lightweight process* model which mitigates the impact of thread competition. The *Insieme* compiler [11] benefits of nested recursive parallel regions – generating multiple versions of the nested tasks – and maximizes the utilization of cores.

## 7 Conclusion and Future Work

This paper discussed the incremental process of improving the performance of an industrial application from the automotive industry that intersects and repairs meshes generated by a CAD application. Starting from the original code version that was already parallelized, we identified the requirement of unveiling, exposing and utilizing nested parallelism to provide enough parallel workload to overcome initial load-balancing issues. Moreover, a statically scheduled binary reduction operation has been re-factored to a dynamically orchestrated reduction schema whose structure depends on the actual execution time of the described steps to mitigate additional load balancing issues. Finally, tuning runtime parameters influencing the management of nested parallelism provided an additional performance increase, improving the speedup from 2.9 – when executing the nested

parallel approach with reduction on demand – to a total of 5.69 for the best scenario.

Future work will address a more sophisticated API for exploiting parallelism of C++ codes.

**Acknowledgements.** This research has been funded by the Austrian Research Promotion Agency under contract 834307 (AutoCore).

## References

[1] Engineering Center Steyr: ALSIM, a simulation tool for the dynamical dip painting process (June 2014), `http://alsim.ecs.steyr.com`
[2] Schifko, M., Jüttler, B., Kornberger, B.: Industrial application of exact boolean operations for meshes. In: Proceedings of the 26th Spring Conference on Computer Graphics, SCCG 2010, pp. 165–172. ACM, New York (2010)
[3] Wein, R., Fogel, E., Zukerman, B., Halperin, D.: Advanced programming techniques applied to cgal's arrangement package. Computational Geometry 38(1-2), 37–63 (2007); Special Issue on CGAL
[4] Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
[5] ISO: ISO/IEC 14882:2011 Information technology — Programming languages — C++. International Organization for Standardization, Geneva, Switzerland (February 2012)
[6] OpenMP Architecture Review Board: OpenMP application program interface version 3.0 (May 2008)
[7] Williams, A.: C++ Concurrency in Action. Manning, Pearson Education (2012)
[8] Gumundsson, G.Ó., Amsaleg, L., Jónsson, B.Ó.: Distributed High-Dimensional Index Creation using Hadoop, HDFS and C++. In: CBMI - 10th Workshop on Content-Based Multimedia Indexing, Annecy, France (2012); Quaero
[9] Hindman, B., Konwinski, A., Zaharia, M., Stoica, I.: A common substrate for cluster computing. In: Proceedings of the 2009 Conference on Hot Topics in Cloud Computing, HotCloud 2009. USENIX Association, Berkeley (2009)
[10] Tanaka, Y., Taura, K., Sato, M.: Performance evaluation of openmp applications with nested parallelism. In: Dwarkadas, S. (ed.) LCR 2000. LNCS, vol. 1915, pp. 100–112. Springer, Heidelberg (2000)
[11] Thoman, P., Jordan, H., Fahringer, T.: Compiler multiversioning for automatic task granularity control. Concurrency and Computation: Practice and Experience (2014)