Int J Parallel Prog (2007) 35:459–476
DOI 10.1007/s10766-007-0054-1

Nested Parallelization with OpenMP

Dieter an Mey · Samuel Sarholz · Christian Terboven

Received: 6 November 2006 / Accepted: 26 January 2007 / Published online: 20 July 2007

Abstract OpenMP is widely accepted as a de facto standard for shared memory parallel programming in Fortran, C and C++. Nested parallelization has been included in the first OpenMP specification, but it took a few years until the first commercially available compilers supported this optional part of the specification. We employed nested parallelization using OpenMP in three production codes: a C++ code for content-based image retrieval, a C++ code for the computation of critical points in multi-block CFD datasets, and a multi-block Navier-Stokes solver written in Fortran90. In this paper we discuss the opportunities as well as the deficiencies of the nested parallelization support in OpenMP.

Keywords OpenMP · Nested parallelization · ccNUMA · Shared memory parallelization

1 Introduction

OpenMP is widely accepted as a de facto standard for high-level shared memory parallel programming in Fortran, C and C++. Nested parallelization has already been included in the first OpenMP specification, leaving the implementer the flexibility to serialize parallel regions that are nested within parallel regions.

D. an Mey (✉) · S. Sarholz · C. Terboven
Center for Computing and Communication, RWTH Aachen University, Aachen, Germany
e-mail: anmey@rz.rwth-aachen.de

S. Sarholz
e-mail: sarholz@rz.rwth-aachen.de

C. Terboven
e-mail: terboven@rz.rwth-aachen.de

While compiler developers embraced OpenMP very quickly, it took a few years until the first commercially available compilers supported this optional part of the specification.

In fact, nested parallelization turned out to be quite useful for increasing the scalability of OpenMP codes, but still a few shortcomings were detected in the specification. These will hopefully be overcome with the upcoming version 3.0 of the OpenMP specification.

We describe experiences gained when employing nested parallelization using OpenMP in three production codes:

- FIRE is a C++ code for content-based image retrieval using OpenMP on two levels [1]. The nested OpenMP approach turned out to be easily applicable and highly efficient.
- NestedCP is written in C++ and computes critical points in multi-block CFD datasets by using a highly adaptive algorithm which profits from the flexibility of OpenMP to adjust the thread count on all parallel levels and to specify loop schedules on three parallel levels [2].
- The multi-block Navier–Stokes solver TFS written in Fortran90 is used to simulate the human nasal flow. OpenMP is employed on the block and on the loop level. This application puts a high load on the memory system and thus is quite sensitive to ccNUMA effects [3].

This paper is organized as follows: Section 2 describes the current support for nested parallelization in the actual OpenMP specification along with the deficiencies we recognized. In Sect. 3 we present our target hard- and software platforms. Sections 4–6 describe the parallelization of the three applications. In Sect. 7 we present some future perspectives of nested parallelization with OpenMP, and finally we draw our conclusions.

2 Nested Parallelization in the Current OpenMP Specification

Programming nested parallelization is as easy as just nesting multiple parallel regions using the standard OpenMP parallel construct plus activating nested parallelism by either setting the environment variable `OMP_NESTED` to true or evoking the runtime function `omp_set_nested()` correspondingly [4].

When the initial thread encounters a parallel region at the outer level, a team of threads is created and the initial thread becomes the master thread. When any or all threads of the outer team encounter another parallel region, and nested support is turned on, these threads create further (inner) teams of threads of which they become the master. So the one thread which is the master of the outer parallel region may also become the master of an inner parallel region, but also the slave threads of the outer parallel region may become the masters of inner parallel regions.

Once the support of nested parallelism is enabled, further parallelization levels can be evoked, and each of the master–master, master–slave, and slave–slave threads can create further teams of threads and become their master and so on. However,

specifying the number of threads of all those newly created teams has not been properly defined in the OpenMP API.

The initial version of OpenMP allowed to specify the number of threads using the environment variable `OMP_NUM_THREADS` and the runtime function `omp_set_num_threads()`. This runtime function was only allowed to be called outside of any active parallel region and thus only the same number of threads could be used on any level of parallelization.

Only with OpenMP V2.0 the `num_threads` clause was added to the parallel construct which can be used to explicitly specify the number of threads for any of the parallel regions at runtime. OpenMP V2.5 introduced the notion of the internal control variable *nthreads-var*, but left it unclear, whether there is one common version of this variable for all threads, or whether such a control variable is available for each individual thread.

Another shortcoming of the current OpenMP specification is the limited support of `threadprivate` data in combination with nested parallelism. Whereas threadprivate data persists between parallel regions when only one level of parallelization is employed, provided that dynamic adjustment of the number of threads is disallowed and that the number of threads remains constant, this is no longer the case with nested parallelization.

The specification of nested parallelization in OpenMP allows recursive nesting of parallel regions, which can be a very elegant solution for recursive algorithms. The specification however does not provide any means to control the level of nestedness nor the total number of threads. Furthermore the `omp_get_thread_num()` runtime function can only be used to inquire a thread's number within the current team, but there is no function to deliver any kind of global thread identification nor the parent thread's identification. Such a functionality can be explicitly programmed, but this is cumbersome and not very elegant.

Finally when employing nested parallelization, one is hit by the fact that OpenMP is not aware of the underlying machine architecture. Many modern symmetric multiprocessing (SMP) architectures have non-uniform memory access (NUMA), still providing cache coherency (ccNUMA), where the memory access time depends on the memory location relative to the accessing processor.

Current operating systems typically support the first touch mechanism on ccNUMA machines to allocate data to the part of memory which is close to the thread which initializes it. Therefore the programmer is in charge of prudently initializing data by the same thread which will later on use it.

If the operating system's process scheduler succeeds in leaving all threads close to their initial location, the memory performance can be good as long as the program's behavior with respect to memory locality does not change over time. Now in the case of nested parallelization, the runtime system can hardly predict a program's future behavior when the first outer parallel region is hit. Typically all threads of the outer team will be placed close to each other with respect to the machine's memory architecture, such that all threads can access shared data efficiently. When later on the inner teams' threads are created, they will most likely be located apart from their master threads, thus causing unfavorable memory access for all shared data.

Table 1 List of the computer systems used for the performance studies

Machine model (abbreviated)	Processors	Operating system	Compiler	Remark
Sun Fire E25K (SFE25K)	72 UltraSPARC IV 1.05 GHz dual core	Solaris 10	Sun Studio 11	ccNUMA, each processor board has 8 cores+local memory
Sun Fire E6900 (SFE6900)	24 UltraSPARC IV 1.2 GHz dualcore	Solaris 9	Sun Studio 11	Flat memory
Sun Fire E2900 (SFE2900)	12 UltraSPARC IV 1.2 GHz dualcore	Solaris 10	Sun Studio 11	Flat memory
Sun Fire V40z (SFV40z)	8 Opteron 875 2.2 GHz dualcore	Solaris 10	Sun Studio 11	ccNUMA, each dualcore processor has has a local memory
Sun Fire X4600 (SFX4600)	8 Opteron 885 2.6 GHz dualcore	Solaris 10	Sun Studio 11	ccNUMA, each dualcore processor has a local memory
NEC SX-8 (NECSX8)	8 NEC SX-8 2.0 GHz vector unit	SX-OS	NEC	SMP vector system with flat memory

3 Nested Parallelization on the Sun Solaris Platform

The main target for our parallelization efforts was the 144-way Sun Fire E25K SMP machine (SFE25K) running the Solaris operating system. As the TFS code is ideally suited for vectorization we also investigated the combination of vectorization and OpenMP parallelization on the NEC SX-8 shared memory parallel vector system. For further performance studies a few more machines were used as listed in Table 1.

Unless otherwise mentioned, all the results quoted in this paper were obtained with the Sun Studio 11 Fortran 95 and C++ compilers on the SFE25K under control of the Solaris 10 operating system.

3.1 The ccNUMA Properties of the Machinery

Whereas the Sun Fire E6900 (SFE6900), the Sun Fire E2900 (SFE2900), and the NEC SX-8 (NECSX8) have a rather flat memory system, the Sun Fire E25K (SFE25K) and even more the Opteron-based Sun Fire X4600 (SFX4600) have a ccNUMA architecture.

On the SFE25K the two stage cache coherence protocol and the limited bandwidth of the backplane lead to a reduction of the global memory bandwidth and to an increased latency when data is not local to the accessing process. The machine has 18 processor boards with four dual-core processors and local memory, thus each locality domain consists of eight processor cores.

On the eight-socket dual-core Opteron based SFX4600 data and cache coherence information is transferred using the HyperTransport links. Each processor has three links so that the processor cores are up to three hops apart from each other. On the four-socket SFV40z the maximum distance is two hops. Whereas the local memory access is very fast, multiple simultaneous remote accesses can easily lead to grave congestions of the HyperTransport links.

3.2 The Solaris ccNUMA Support

Since version 9 update 1, the Solaris operating system provides the Memory Placement Optimization Facility (MPO) [5] which allows the use of first-touch or random placement strategies and also provides a low-level API to explicitly migrate pages to where they are used next (next-touch strategy). Furthermore, Solaris allows to explicitly bind threads to processors cores.

The combination of binding and memory placement control allows to deal with ccNUMA issues as described in Sect. 6, but none of these features is standardized in the context of OpenMP.

3.3 Nested OpenMP Support in the Sun Studio Compilers

The OpenMP runtime library of the Sun Studio compilers maintains a pool of threads that can be used as slave threads in parallel regions [6]. The size of this pool can be controlled by the environment variable `SUNW_MP_MAX_POOL_THREADS`. The default value is 1023.

The environment variable `SUNW_MP_MAX_NESTED_LEVELS` controls the maximum depth of nested active parallel regions that require more than one thread. Any active parallel region that has an active nested depth greater than the value of this environment variable will be executed by only one thread. A parallel region is considered active if it is an OpenMP parallel region whose `if` clause, if specified, evaluates to true. Only active parallel regions are counted. The default maximum number of active nesting levels is 4.

Calls to the OpenMP routines `omp_set_num_threads()`, `omp_set_dynamic()`, `omp_get_max_threads()`, `omp_get_dynamic()`, `omp_set_nested()` and `omp_get_nested()` within nested parallel regions deserve some discussion. The ‘set’ calls affect only the parallel regions at the same or inner nesting levels encountered by the calling thread. They do not affect parallel regions encountered by other threads, and they do not affect parallel regions the calling thread will later encounter in any outer levels. The ‘get’ calls will return the values set by the calling thread. When a team is created, the slave threads will inherit the values from the master thread.

4 Content-based Image Retrieval with FIRE

With the enormously growing amount of digitally available image data, the need for adequate methods to access, sort and store the data is heavily increasing. For example medical doctors have to access immense amounts of images daily [7] and home

users often have image databases of thousands of images [8]. Currently these images are usually accessed by meta data, e.g., the date when the image was taken. But content-based methods, though computationally more expensive, promise interesting possibilities [9].

The Flexible Image Retrieval Engine (FIRE) [10] has been developed at the Human Language Technology and Pattern Recognition Group of the RWTH Aachen University. It is designed as a research system with flexibility in mind and is easily extensible and highly modular. The FIRE system was successfully used in the ImageCLEF 2004 and 2005 content-based image retrieval evaluations [11,12] .

4.1 Image Retrieval

Given a query image and the goal to find images from a database that are similar to the given query image, we calculate a score for each image from the database. The database images with the highest score are returned.

Given the image retrieval system, three different layers can be identified that offer potential for parallelization:

- Queries tend to be mutually independent. Thus, several queries can be processed in parallel.
- The scores for the database images can be calculated in parallel as the database images are independent from each other.
- Parallelization is possible on the feature level, because the distances for the individual features can be calculated in parallel.

Only the first two layers are considered here, as the third may require larger changes in the code for some distance functions and we do not expect it to be profitable as the parallelization in the first two layers already leads to sufficient speedup in normal situations.

4.2 Parallelization

Shared memory parallelization is more suitable than distributed memory parallelization for the image retrieval task, as the image database can then be accessed by all threads and does not need to be distributed. Furthermore, it leads to better throughput on a parallel computer: If one process consumes the whole memory of a machine, the remaining processors cannot be used. In addition, several program instances loading the image database concurrently might put heavy pressure on the file server, as the image database can be several gigabyte in size. The object oriented programming paradigm as employed in the FIRE C++ code simplified the parallelization. The datatype encapsulation originating from the mathematical model of the image retrieval task prevents unintended data dependencies and supports the data dependency analysis as well.

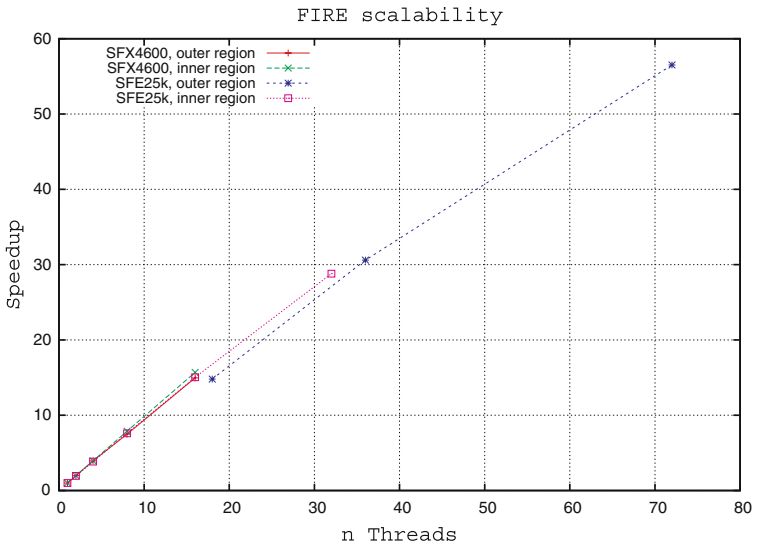

Fig. 1 Speedup of the FIRE code on the Sun Fire E25K and the Sun Fire X4600

4.3 Scalability

Experimental results with the FIRE code on a sixteen core Opteron Sun Fire SFX4600 and a 144 core UltraSPARC IV Sun Fire E25K show a very good speedup (Fig. 1).

The optimal thread distribution on the SFX4600 uses two threads on the outer level and eight on the inner level (speedup: 15.8, efficiency: 99%), and on the SFE25K the optimum uses eight threads on the outer level and 18 on the inner level (speedup: 133.3, efficiency: 93%). There are several reasons why such a high scalability can be reached, among these are:

- The computations on both levels are totally independent. The outer level contains a few synchronization constructs to ensure well formatted output for further processing.
- As for each query image the same number of database images has to be processed, and for each database image the comparison step always takes the same amount of time, the load is perfectly balanced.
- The required memory bandwidth is rather small, so the code scales reasonably well even on ccNUMA architectures.

5 Computation of 3D Critical Points in Multi-Block CFD Datasets

In order to interactively analyze results of large-scale flow simulations in a virtual environment, different features are extracted and visualized from the raw output data. When examining flow fields, one is particularly interested in the investigation of the velocity field, which is usually defined by vectors stored at the nodes of the grid. One

feature that helps describing the topology is the set of critical points, where the velocity is zero. [2].

5.1 Critical Point Algorithm

The algorithm for critical point extraction is organized in three nested loops, which are all candidates for parallelization, as there is no data dependency among their iterations. The outermost loop iterates over the time steps of unsteady datasets. The middle loop deals with the blocks of multi-block datasets and the inner loop checks all grid cells within the blocks. As soon as the number of threads is larger than the iteration count of the single loops, nested parallelization is obviously appropriate to improve speedup.

We use a heuristic on each cell to determine whether it may contain a critical point. If so, the cell is recursively bisected and the heuristic is applied again on each subcell. Non candidate (sub-) cells are discarded. After a certain recursion depth the Newton–Raphson iteration is used to determine the exact position. The time needed to check different cells may vary considerably as a result. If critical points are lined up within a single cell, the computational cost related to this cell may even increase exponentially. Furthermore, the size of the blocks as well as the number of cells per time step may vary considerably. Thus, this code can really profit from the flexible loop scheduling constructs provided in OpenMP.

5.2 Results

We measured the runtime of the parallel feature extraction on the Sun Fire E25K, varying the loop schedules and the number of threads assigned to each of the parallelization levels. Let n be the number of threads involved. The amount of threads for the time level is denoted as ti and for block level as bj. The remaining $n - i \cdot j$ threads, denoted as ck, are assigned to the cell level.

The achievable speedup heavily depends on the selected dataset. Datasets which do not cause severe load imbalance display almost perfect scalability.

An extremely well-tempered case is the output dataset of a supersonic shock simulation. A speedup of 115 can be reached by just using all 144 threads on the outer level and a `static` loop schedule. The speedup can be increased to 119.9 with nested parallelization (t24 b1 c6) and `static` scheduling.

The situation is quite different on another dataset simulating the inflow and compression phase of a combustion engine. When the valves start to close and suddenly move in the opposite direction of the inflowing air, plenty of critical points close to the valve can be detected in the output data of the corresponding simulation. Applying static schedules on this heavily imbalanced dataset limits the speedup to 11 at best (t12 b12 c1 threads, see Fig. 2).

By choosing an appropriate schedule on all parallelization levels, the speedup can be considerably increased. The `dynamic` schedule with a chunk size of one turned out to work best on both outer parallelization levels. However it was not possible to find a suitable chunk size for the `dynamic` schedule on the cell level for all datasets: it

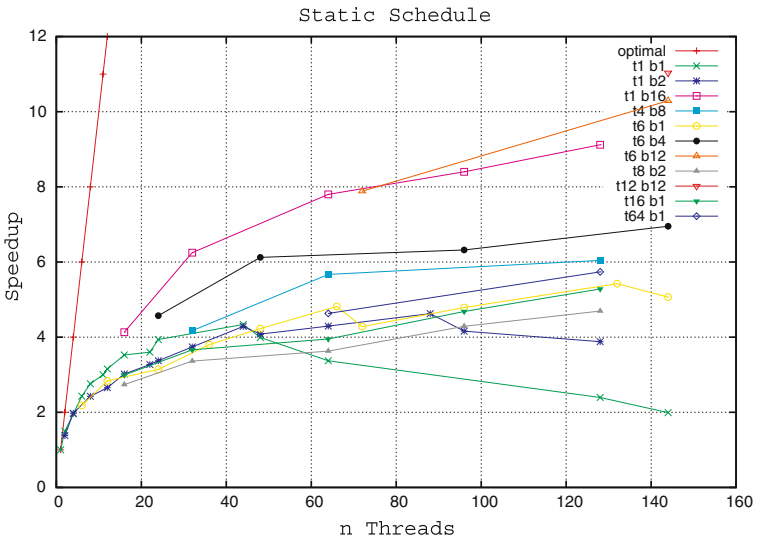

Fig. 2 Computing the critical points for a combustion engine, static schedule

eather caused too much overhead or the chunks were too large to satisfactorily improve the load balance. The `guided` schedule offers a nice compromise by reducing the chunk size over time according to the formula 1.

$$chunk\_size = \#unassigned\_iterations/(c \cdot \#threads) \tag{1}$$

With the above setting the critical point computation for the engine dataset scaled up to 33.6 using 128 threads (t4 b4 c8). It turned out, however, that the weight parameter c in Eq. 1 which defaults to 2 on Sun's OpenMP implementation is not an optimal choice. Fortunately it is possible to change the weight parameter c using the Sun specific environment variable `SUNW_MP_GUIDED_WEIGHT`.

We reached best results using the dynamic schedule with a chunk size of one on both outer loops and the guided schedule with a chunk size of 5 and weight parameter $c = 20$, as shown in Fig. 3. The speedup improved for all thread combinations reaching a maximum of 70.2 with t6 b4 c6 threads.

Even the speedup for the supersonic shock dataset profits from these scheduling parameters: it increases to 126.4 with t144 c1 b1 threads and to 137.6 with t12 b1 c12 case, which corresponds to an efficiency of 96%.

6 The TFS Flow Solver

The Navier–Stokes solver TFS developed by the Institute of Aerodynamics of the RWTH Aachen University is currently used in a multidisciplinary project to simulate the air flow through the human nose [13,14]. The numerical method is second order accurate and uses a multi-block structured grid with general curvilinear coordinates.

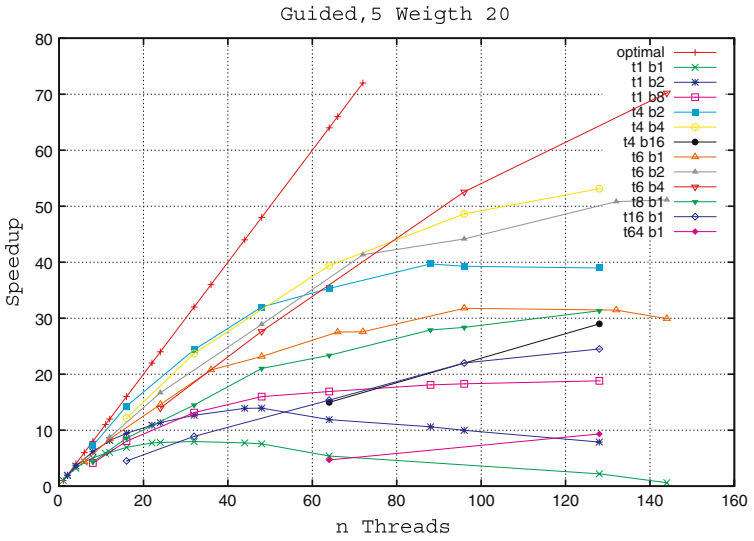

Fig. 3 Computing the critical points for a combustion engine, dynamic/guided schedules

The goal is to get a better understanding of the functioning of the human nose and then to provide a work flow for computer assisted surgery allowing the physician to first perform a virtual surgery in a virtual reality environment. This would then give the opportunity to verify the success of such an operation by another computer simulation before the actual surgery on the patient.

TFS has been particularly well prepared for vectorization by employing one-dimensional arrays to store three-dimensional geometries. Many loops with a high number of iterations are as well suited for loop level parallelization.

The ParaWise/CAPO automatic parallelization environment [15,16] has been used to assist in the OpenMP parallelization of the TFS multi-block code accomplished by the coauthors of [3] from Parallel Software Products and the University of Greenwich. A parallel version of TFS that can scale to large numbers of processors targeted at Sun Microsystems Sun Fire E25K shared memory parallel systems (SFE25K) was the ultimate goal of this work [17].

6.1 The Initial Intra-block Version

The initial version was produced without any user interaction, just using the ParaWise interprocedural, value based dependency analysis and the ParaWise OpenMP code generator [16,18]. The aim of ParaWise is to put as much application code as possible into parallel regions, so if all outer loops in a loop nest have dependencies that inhibit parallelism, any parallelism from inner loops in the loop nest is exploited. Parallelism exploited at an inner loop level can have a detrimental effect on performance as the OpenMP runtime overhead can become significant. Automatic

parallelization included addition of privatization and reduction clauses, nowait clauses to avoid loop end synchronization when legal and generation of parallel regions containing as many parallel loops as possible in an interprocedural context.

6.2 The Improved Intra-block Version

ParaWise was specifically designed with the understanding that user interaction is desirable and, often, essential for the production of effective parallel code. The ParaWise and CAPO browsers were used to investigate, add to and alter information to improve the initial parallelization. In the TFS code a large work array is dynamically allocated in the beginning of the program and then frequently passed to subroutines through parameter lists and used throughout the program by indirect addressing. It is obviously not possible by any static program analysis to assure that there is no data race in accessing this work array, so the user must provide further information. The user can exploit his knowledge of the program to address the parallelization inhibitors determined by ParaWise and displayed in its browsers. Furthermore runtime information provided by the Sun Performance Analyzer can be fed into ParaWise to guide further user efforts for improving the scalability. Additional manual code modifications lead to increased parallel regions and a reduction of the number of synchronizations.

6.3 The Inter-block Version

Then a version where parallelism is exploited at the block level was generated with the previous parallel version as a starting point. The geometry data is read into the application code at runtime, preventing dependence analysis from determining independencies between the iterations of the block loops. As a result, the CAPO browsers were applied to produce most of the parallel code with additional manual code changes to complete the parallel version.

All the relevant loops involved the above mentioned large work array and revealed data dependencies. Those sections of the work array used for mesh data must be scoped `shared` as they are used throughout the whole code. Other sections of the same work array are used for temporary workspace and need to be privatized as they are reused for every block. For other variables inhibiting automatic parallelization by ParaWise, many inhibitors were removed as either dependencies between iterations or loop in/out dependencies are known not to exist. A few variables were involved in reduction style operations and these can be set in the ParaWise GUI.

Some block loops containing I/O instructions were not considered for parallelism, they contained inner loops which were considered instead. Additionally, the loop dealing with block–block interaction has dependencies that force serial execution.

6.4 The Nested Parallel Version

The nested parallel version was created by merging the two previous inter-block and intra-block parallel versions and using nested OpenMP parallelism to allow a more

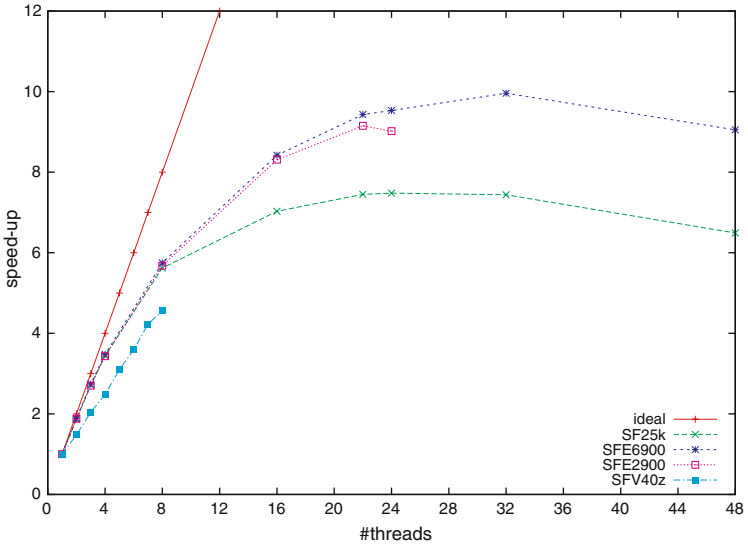

Fig. 4 Speedup of the intra-block version of TFS

efficient parallel execution, each level using a smaller number of processors. In this new version the use of `threadprivate` directives for common blocks was not possible as explained in Sect. 2. Instead, the affected variables were passed into the necessary routines as additional arguments and then defined as `private` for the outer (inter block) parallel region, but shared by the subsequent inner (intra block) parallel region. Finally in the nested parallel version 7 major parallel outer block loops out of 11 include inner parallel loops. This version contains some 16,000 lines of Fortran code with 534 OpenMP directives in 79 parallel regions.

The speedup of the nested version was superior to either of the previous parallel versions, but was still not satisfying. Increased overhead of OpenMP runtime library for a larger numbers of processors, a severe load imbalance between different blocks, as well as ccNUMA effects delimited scalability. Contributions to this include the poorly performing block interaction loop that must execute in serial, with the additional impact of one or two small code sections that were left serial because the OpenMP overheads led to slowdown when parallelism was exploited. Seven out of eleven code sections which exploit inter-block level parallelism, but no intra-block level parallelism only represented a tiny proportion of runtime in serial, but on larger numbers of processors their impact on speedup is more significant (Amdahl's law).

6.5 Improving Scalability of the Nested Parallel Program

Figures 4 and 5 depict the best effort speedup of the intra-block and the inter-block versions respectively on the machines listed in Table 1.

It can clearly be seen in Fig. 6 that the nested approach leads to an increased speedup for all of the larger SMP machines. The maximum is close to 20 when using 64 threads

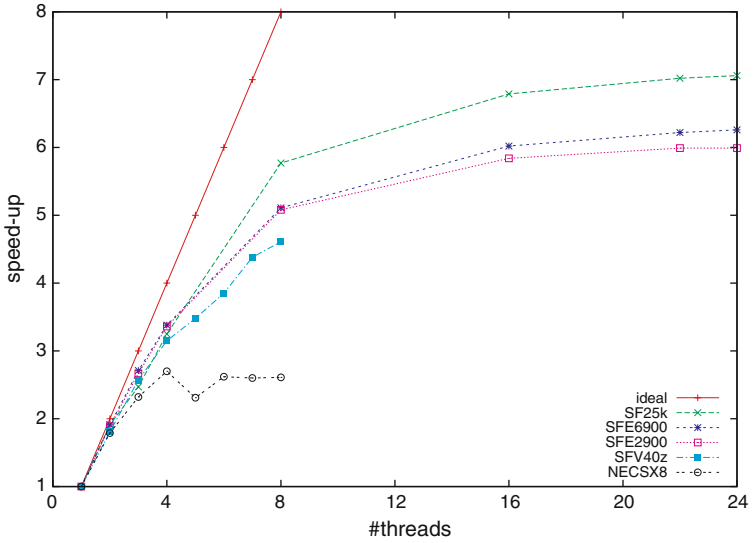

Fig. 5 Speedup of the inter-block version of TFS

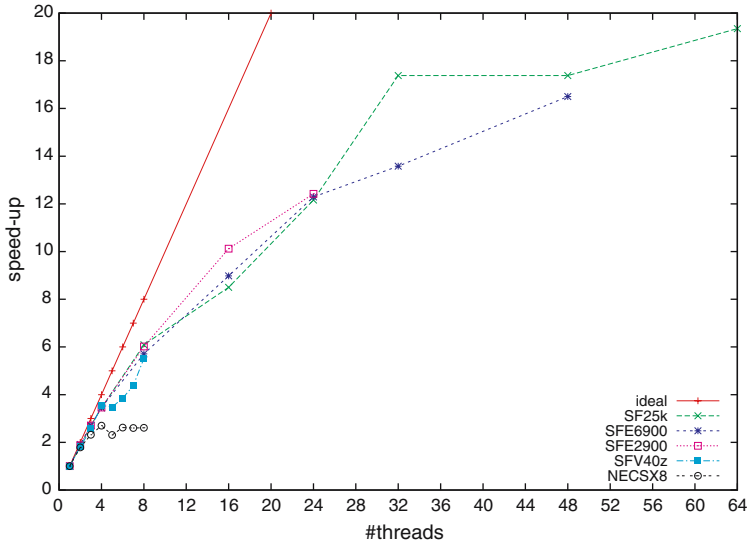

Fig. 6 Speedup of the nested parallel version of TFS

on the SFE25K machine, whereas the speedup of both single-level approaches is less than 10 in all cases. On the NECSX8 vector machine, vectorization replaces the intra-block level OpenMP parallelization and the inter-level version delivers a speedup of 2.7 when using 8 threads, yet displaying a very high absolute speed.

6.5.1 Sorting Blocks by Size

Because of the complex geometry of the human nose, the 32 blocks of the computational grid are considerably varying in size: The largest block has about 15 times more grid points than the smallest block and accounts for about 10% out of the 2,200,000 grid points. This fact limits the attainable speedup on the block level to 10 at most. Splitting the larger blocks was not considered at this point. The first approach of selecting a dynamic schedule for all of the block-level loops in order to handle the resulting load imbalance already works reasonably well. But if a relatively large block is scheduled to one thread at the end of the loop, the other threads might finally be idle. Sorting the blocks in decreasing order, such that the smallest block is scheduled last, leads to a first slight improvement in runtime of 5–12% percent for nine or more threads on the SFE25K and 13.5% for eight threads on the SFV40z.

6.5.2 Thread Balancing

The idea of the dynamic thread balancing scheme has previously been used to solve load imbalance of hybrid (MPI + OpenMP) programs [19]. Here the number of threads of the inner teams were adjusted to the size of the corresponding blocks. This leads to a slight improvement of more than 10% on the SFE25K when using 121 threads, as the scalability of the loop level parallelization is limited and cannot overcome the difference in size of the blocks.

6.5.3 Grouping Blocks

As the block sizes remain constant during the whole runtime of the program, the blocks can be explicitly grouped and accordingly distributed to a given number of threads on the outer parallel level in order to reduce the overhead of the dynamic schedule and to avoid idle threads. Surprisingly this did not lead to a measurable performance improvement on the SFE25K. Further investigations using hardware counters to measure the number of L2 cache misses revealed that threads working on smaller blocks profit more from the large size (8 MB) of the external L2 caches of the UltraSPARC IV-based machines than larger blocks and therefore ran at a much higher speed. When employing a single thread on the loop level, the thread working on the smallest block ran at 351 MFlop/s and the one working on the largest block ran at 225 MFlop/s. This of course aggravates the load imbalance.

Grouping and distributing the blocks was profitable on the SFV40z as the varying block size did not impact the MFlop/s rate, because of the smaller L2 cache (1 MB) of the Opteron processor. The performance improved by 6.7% when using eight threads.

6.5.4 Memory Locality

Further hardware counter measurements indicated that L2 cache misses lead to a high percentage of remote misses and that the global memory bandwidth consumption of the code on the SFE25K was close to the maximum value, which we observed when stressing the memory system with the Stream benchmark [20] in earlier experiments

with disadvantageous memory placement. We concluded that an improvement in the memory locality would have a positive impact on the performance of TFS on the SFE25K.

In order to improve memory locality we bound threads to processors and also used the `madvise()` Solaris system call after a warm-up phase of the program in order to explicitly migrate pages to where they are used (next-touch mechanism). Surprisingly, this was only profitable when applied to a single level of parallelism. It lead to an improvement of roughly 10% on the SFE25K for higher thread counts, but was particularly beneficial on the SFV40z—the improvement was up to a factor of 1.9 for the intra-block version.

Unfortunately, applying these techniques to the nested parallel version was not profitable at all, because the current implementation of nested OpenMP parallelization in the Sun Studio compilers employs a pool of threads as described in Sect. 3. These threads are dynamically assigned whenever an inner team is forked. Therefore the threads of the inner teams frequently loose their data affinity.

There is no single strategy which performs best in all cases. Figure 6 depicts the best effort speedup of the nested parallel versions on the machines listed in Table 1. On the SFE25K, our target platform, the speedup was still below 20 with 64 threads.

6.5.5 Applying an Experimental Threading Library

In theory, if all threads are bound to processor cores and a static schedule on the block level is used and blocks are explicitly grouped and assigning to the (master) threads and their inner teams and data pages related to the blocks are allocated (migrated to) where they are used, performance should be better.

Compiler Engineers from Sun Microsystem provided an experimental version of the threading library `libmtsk` which improves the thread affinity by maintaining the mapping between threads of the pool and the members of the inner teams. The combination of thread affinity, processor binding and explicit data migration finally lead to another improvement in scalability of about 25% on the SFE25K. A speed-up of 25 for 64 threads finally is a satisfying result taking into account

- That each locality group (processor board) has eight cores and thus using eight threads is a sweet spot for the intra-block version (see Fig. 4) delivering a speed-up of 5–6,
- That the largest block dominates the inter-block version with more than eight threads (see Fig. 5) thus limiting the speed-up to about six
- That there are some serial parts and some parts only suited for one level of parallelization.

7 Conclusion and Outlook

We were able to efficiently apply nested parallelization with OpenMP to three production codes to increase their scalability on large SMP machines allthough there are some deficiencies in the current support of nested parallelization in OpenMP.

7.1 FIRE

We presented our approach to using shared-memory based parallelization techniques in content-based image retrieval. Using OpenMP the performance increase is nearly linear with the number of the processors with minimal modifications to the source code, thus not impacting either the algorithmic structure nor the portability of the code. It is clearly shown that shared-memory parallelization is a suitable way to achieve better runtime performance for applications in computer vision.

7.2 NestedCP

The parallelization of the critical point computation for the visualization of CFD simulation results in virtual environments using OpenMP was very successful. Three levels of parallelization offer several degrees of freedom to choose the number of threads and the loop schedule on each level. Based on numerous experiments, we developed a heuristic to determine a reasonable setting for all investigated datasets. With 144 threads, we obtained a speedup of between 70 and 137. The measurements indicate that even larger shared memory systems than a 144-way Sun Fire E25K would have been beneficial.

7.3 TFS

The ParaWise environment was used to perform most of the parallelization of the multi–block TFS Navier–Stokes solver, and greatly assisted in the subsequent manual tuning. Intra-block and inter-block parallel versions were generated first and then merged into a nested version improving the total speedup for more than eight threads. The number of threads on each parallelization level and the assignment of blocks to the threads of the outer team had to be carefully choosen to overcome the load imbalance on the block level. As the code consumes a high memory bandwidth, the parallel versions is sensitive to ccNUMA effects. In spite of thread binding and explicite data migration an experimental version of the Sun Studio threading library was necessary to increase the speedup from about 20 to a factor of 25 when using 64 threads on the Sun Fire E25K.

7.4 OpenMP Version 3.0

In [21] the issues which are currently under consideration for the upcoming version 3.0 of the OpenMP specification have been disclosed. Some will improve the support for nested parallelism:

- The definition of multiple internal control variables will permit to call the `omp_set_num_threads()` runtime function inside a parallel region for an improved control of the number of threads of inner teams, a feature which is already provided by the Sun Studio compilers.

- Additional library routines will be available to determine the depth of nesting and the thread identification of ancestor threads.
- Threadprivate variables will persist across inner parallel regions under certain conditions.
- Describing the nesting structure up-front will allow the runtime to make intelligent thread placement decisions.

The ccNUMA support can already be important for a single level of parallelization, but it may be even more performance critical for nested parallelism, as has been exposed in Sect. 6. Solaris already provides efficient tools to deal with data affinity and there are activities in the Linux community to catch up [22]. Both operating systems provide sophisticated policies to keep threads close to their home location. We consider a `migrate_next_touch` directive to be an adequate tool to deal with situations in which a first touch mechanism is not sufficient.

Acknowledgements We like to thank Steve Johnson, Peter Leggett, and Constantinos Ierotheou from Parallel Software Products and University of Greenwich for their remarkable work generating the initial nested parallel version of TFS with the ParaWise/CAPO automatic parallelization toolkit. We also want to thank Andreas Gerndt from Louisiana Immersive Technologies Enterprise at the University of Louisiana who guided the work on the critical point computation while he was still at the RWTH Aachen University. We thank Nawal Copty and Yuan Lin from Sun Microsystems for providing an experimental version of their threading library.

References

1. Terboven, C., Deselaers, T., Bischof, C., Ney, H.: Shared-memory parallelization for content-based image retrieval. In: ECCV 2006 Workshop on Computation Intensive Methods for Computer Vision
2. Nested OpenMP for Efficient Computation of 3D Critical Points in Multi-Block CFD Datasets; Supercomputing (2006) (to appear)
3. Johnson, S., Leggett, P., Ierotheou, C., Spiegel, A., an Mey, D., Hoerschler, I.: Nested parallelization of the flow solver tfs using the parawise parallelization environment; IWOMP (2006); http://iwomp.univ-reims.fr/cd/papers/JLI+06.pdf
4. OpenMP Architecture Review Board: OpenMP application program interface, v2.5. (2005) http://www.openmp.org or http://www.compunity.org
5. Solaris Memory Placement Optimization and Sun Fire Servers, Technical White Paper, http://www.sun.com/servers/wp/docs/mpo_v7_CUSTOMER.pdf
6. Sun Studio 11: OpenMP API User's Guide, Chapter 2, Nested Parallelism, http://docs.sun.com/source/819-3694/2_nested.html
7. Müller, H., Michoux, N., Bandon, D., Geissbuhler, A.: A review of content-based image retrieval systems in medical applications–clinical benefits and future directions. Int. J. Med. Inform. (73)1–23 (2004)
8. Sun, Y., Zhang, H., Zhang, L., Li, M.: Myphotos a system for home photo management and processing. In: ACM Multimedia Confernce, pp. 81–82 Juan-les-Pins, France, (2002)
9. Smeulders, A.W.M., Worring, M., Santini, S., Gupta, A., Jain, R.: Content-based image retrieval: the end of the early years. IEEE T. Pattern Anal. **22**(12), 1349–1380 (2000)
10. Deselaers, T., Keysers, D., Ney, H.: Features for image retrieval—a quantitative comparison. In: DAGM 2004, Pattern Recognition, 26th DAGM Symposium, pp. 228–236 Number 3175 in Lecture Notes in Computer Science, Tübingen, Germany (2004)
11. Clough, P., Müller, H., Sanderson, M.: The CLEF cross language image retrieval track (ImageCLEF) 2004. In: Fifth Workshop of the Cross–Language Evaluation Forum (CLEF 2004). Volume 3491 of LNCS, pp. 597–613 (2005)

12. Clough, P., Mueller, H., Deselaers, T., Grubinger, M., Lehmann, T., Jensen, J., Hersh, W.: The clef 2005 cross-language image retrieval track. In: Workshop of the Cross–Language Evaluation Forum (CLEF 2005). Lecture Notes in Computer Science, Vienna, Austria (2005) (in press)
13. Hörschler, I., Meinke, M., Schröder, W.: Numerical simulation of the flow field in a model of the nasal cavity. Comput. Fluids **32** 3945 (2003)
14. Hörschler, I., Brücker, C., Schröder, W., Meinke, M.: Investigation of the impact of the geometry on the nose flow, Eur. J. Mech. B/Fluids (In Press) http://dx.doi.org/10.1016/j.euromechflu.2005.11.006
15. ParaWise automatic parallelisation environment, PSP Inc. http://www.parallelsp.com
16. Jin, H., Frumkin, M., Yan, J.: Automatic generation of OpenMP directives and it application to computational fluid dynamics codes. International Symposium on High Performance Computing, p. 440 Tokyo, Japan, (2000)
17. Johnson, S., Ierotheou, C.: Parallelization of the TFS multi-block code from RWTH Aachen using the ParaWise/CAPO tools, PSP Inc, TR-2005-09-02, (2005). http://www.parallelsp.com/downloads/TechnicalReports/TR-2005-09-02.pdf
18. Johnson, S., Cross, M., and Everett, M.: Exploitation of symbolic information in interprocedural dependence analysis. Parallel Comput. **22**, 197–226 (1996)
19. Spiegel, A., an Mey, D., Bischof, C.: Hybrid parallelization of CFD Applications with Dynamic Thread Balancing, PARA04. In: Dongarra J., Madsen K., Wasniewski J. (eds.) Applied Parallel Computing State of the Art in Scientific Computing: 7th International Conference, PARA 2004, vol. 3732, pp. 433–441. Lyngby, Denmark (2006)
20. McCalpin, J.D.: STREAM: sustainable memory bandwidth in high performance computers, http://www.cs.virginia.edu/stream/
21. Bull, M.: The status of OpenMP 3.0, SC06, OpenMP BoF http://www.compunity.org/futures/Mark_SC06BOF.pdf
22. SUSE Linux 10.1 NUMA Policy Control, http://www.novell.com/products/linuxpackages/../suselinux/numactl.html