Int J Parallel Prog (2009) 37:306–323
DOI 10.1007/s10766-009-0098-5

Parallelism and Scalability in an Image Processing Application

Morten S. Rasmussen · Matthias B. Stuart · Sven Karlsson

Received: 10 March 2009 / Accepted: 31 March 2009 / Published online: 23 April 2009

Abstract The recent trends in processor architecture show that parallel processing is moving into new areas of computing in the form of many-core desktop processors and multi-processor system-on-chips. This means that parallel processing is required in application areas that traditionally have not used parallel programs. This paper investigates parallelism and scalability of an embedded image processing application. The major challenges faced when parallelizing the application were to extract enough parallelism from the application and to reduce load imbalance. The application has limited immediately available parallelism and further extraction of parallelism is limited by small data sets and a relatively high parallelization overhead. Load balance is difficult to obtain due to the limited parallelism and made worse by non-uniform memory latency. Three parallel OpenMP implementations of the application are discussed and evaluated. We show that with some modifications relative speedups in excess of 9 on a 16 CPU system can be reached.

Keywords OpenMP · Image processing · Parallelization · Performance evaluation

M. S. Rasmussen (✉) · M. B. Stuart · S. Karlsson
DTU Informatics, Technical University of Denmark, Richard Petersens Plads,
DTU - Building 322, 2800 Kgs. Lyngby, Denmark
e-mail: msr@imm.dtu.dk

M. B. Stuart
e-mail: ms@imm.dtu.dk

S. Karlsson
e-mail: ska@imm.dtu.dk

1 Introduction

To reach higher performance, processor designers have in the last few decades focused on clock frequency and elaborate designs that can extract instruction level parallelism from sequential code. However, that approach presently leads to diminishing returns and high power consumption. As a consequence, vendors have turned their focus to multi-core architectures where several processors are placed on a single silicon chip to increase system performance [1,2]. Such architectures are inherently explicitly parallel and trends indicate that this evolution will continue towards massively parallel many-core architectures [3].

So far the programming models for multi-core architectures have been very similar to those for shared memory multiprocessors. OpenMP [4,5] is one of these shared memory programming models. OpenMP offers a multi-threaded programming model based on a set of compiler directives and library calls. Instead of explicit thread management, OpenMP controls threads, synchronization and work distribution implicitly based on the parallelism exposed in the code through the use of compiler directives. Thus, explicit thread management and lock-based synchronization, which are both complex and error-prone, are largely avoided. One example of the parallel constructs supported by OpenMP is automated parallel execution of for-loops without dependencies between the iterations.

Increasing parallelism in processor architecture is not limited to processors for high performance systems. Embedded systems follow a similar trend where multi-processor system-on-chip solutions with advanced interconnection networks have been proposed [6–8]. Increasing amounts of available on-chip transistor resources allows system architectures with multiple low-power processor cores. Furthermore, chip production costs have increased the focus on designing embedded platform architectures using more general processor cores, which can be used for a range of applications [9]. Thus, there is a need to explore parallel processing in the context of embedded systems where parallel programmers are faced with new types of applications. Thereby new challenges are exposed as embedded systems have requirements different from those of high performance computing systems. One example is applications with user interaction, where fast processing time is more important than throughput to ensure the necessary responsiveness.

Since embedded systems are an emerging area for parallel programming, efficient programming models for these systems has yet to be found. Shared memory has been used for decades in traditional multiprocessor systems and is therefore an obvious starting point for exploring parallelism in applications for embedded systems.

In this paper, we explore an embedded image processing application for object identification using multi-spectral images and we investigate its parallel behavior using OpenMP 2.5 [4]. In short, our contributions are: (i) The analysis of an embedded image processing application; (ii) a thorough performance evaluation of the parallel properties of the application using OpenMP.

The major challenges faced when parallelizing the application were to extract enough parallelism from the application and to reduce load imbalance. The experimental results show that, with some tuning, relative speedups in excess of 9 on a 16 CPU system can be reached.

The rest of the paper is organized as follows. This section is concluded with a discussion of related work. In Sect. 2, we describe the application and in Sect. 3 we explain how the application has been parallelized. Section 4 discusses parallelization using OpenMP and experimental results are presented in Sect. 5. The paper ends with conclusions in Sect. 6.

1.1 Related Work

Parallelization of image processing algorithms for image classification using OpenMP has previously been presented [10]. The work investigated an algorithm used for identifying forest areas in satellite images. The algorithm is parallelized to achieve better performance by running individual processing steps in parallel and by decomposing the data set into smaller parts. A high performance 64 processor computer was used as test platform to process the 1.2 GB images. Our work differs in that we present experiences with a potential embedded application with images two orders of magnitude smaller, which means that parallelization overhead is more pronounced.

Parallelization approaches for image processing on smaller images has previously been proposed by Meerwald et al. [11]. Two reference implementations of the JPEG2000 image coding standard are analyzed to identify stages where parallelism can be exploited for increased performance. The JPEG2000 implementations have high memory bandwidth requirements. Cache performance is also an issue for the JPEG2000 implementations. The scalability is found to be limited, reaching a speedup of 2 on 16 processors. The application studied in our paper exhibit similar problems with memory bandwidth requirements and cache performance.

Content-based image retrieval is another application of automated image classification which can benefit significantly from parallelization. Content-based image retrieval allows advanced image database queries based on image content. A database query thereby involves processing every image in the database in order to examine its content. A shared memory parallelization of this application has been presented previously [12]. The application is parallelized by processing individual queries and images involved in each query in parallel. In contrast, we strive to minimize processing latency of a single image by parallelizing the processing of the individual image, which requires more fine-grained parallelization.

In many algorithms, tasks may be decomposed into subtasks which can be processed in parallel. This is also the case for the application analyzed in this work, where all possible parallelism must be exploited to ensure scalability. OpenMP supports this form of nested parallelism, but load balancing among nested threads is limited. an Mey et al. [13] discuss issues in nested parallelization of three production codes using OpenMP. Similar to our work, selecting the parallelization strategy for each nested level of parallelism and load balancing are major challenges. However, in contrast to our work, abundant parallelism is available at each level and the parallelization overhead is low for two of the codes examined.

An algorithm for finding good mappings of tasks to threads when using nested parallelism has previously been proposed [14]. It is based on the assumption that the outer-level layer of parallelism consists of coarse-grained independent tasks of unequal sizes. The algorithm seeks to distribute the available threads among the tasks while balancing the load. Large tasks may have multiple threads assigned, in which case the algorithm is applied again to distribute its subtasks among the assigned threads. In our work, the source of load imbalance is not unequally sized tasks, but rather equally sized tasks that do not match the number of available processors. In this case, load balancing equally sized tasks requires either sharing of nested threads among coarse level tasks or automatic run-time management of the number of nested treads. None of these features are supported by OpenMP 2.5 [4].

Automatic run-time thread distribution for nested parallelism has previously been proposed to eliminate the need for hand optimization [15,16]. In the first proposal, all available parallelism is exposed to the environment and leaves the parallelization strategy to the run-time environment [15]. Instead of static parallelization, which can not take run-time conditions into account, the run-time environment seeks to minimize overhead and balance load by first exploiting as much coarse level parallelism as possible and then balance the load by exploiting fine-grained parallelism. This approach would be beneficial for the application in our work, as it has the ability exploit coarse level parallelism first and then reduce the slack by utilizing any available fine-grained parallelism.

In the latter approach, the performance of each coarse level task is monitored at run-time and threads are dynamically transferred between coarse level tasks to obtain the best overall performance [16]. This approach could improve performance of our nested implementation as the performance issue of thread dedication to groups of nested threads is avoided by managing their assignment at run-time. However, none of these two improvements have been included in the OpenMP specification yet and thus they are not given further consideration in this work.

2 Image Processing Application

In this paper, we are focusing on an image processing application developed at DTU [17] and written in Matlab. The application is used for object identification based on multi-spectral imaging and can be used for many different purposes. One example is identifying the species of a *Penicillium* fungus in a petri dish from a multi-spectral image [17]. The object identification is based on information extracted from the images in the form of scalar values, called *features*, that each describes some aspect of the input image. Features are grouped into *feature sets*, based on extraction method used for the particular features.

The flowchart in Fig. 1 gives an overview of the application. It consist of three major parts: pre-processing, analysis and a statistical model. The application input is a multi-spectral image of the object that has to be identified. The multi-spectral image is a set of spectral images, where each spectral image shows the object exposed to single colored source of light. Different wave lengths of light reveal different

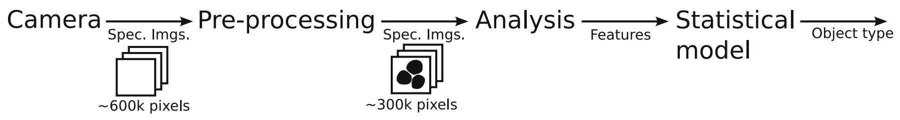

Fig. 1 Overview of the entire application

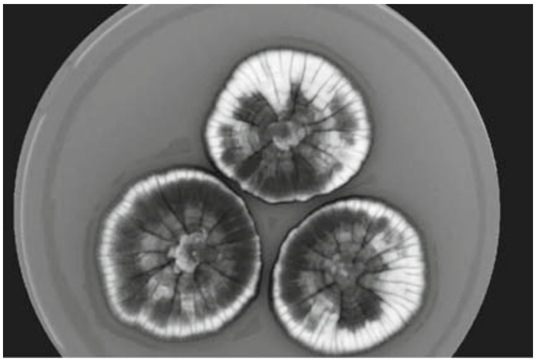

Fig. 2 Spectral image of fungi colonies

elements in the object. Figure 2 shows an example of a spectral image of fungi colonies.

The pre-processing part involves preparing the raw input image for processing, which means removing unnecessary information in the image and normalizing the image. The analysis part is feature extraction based on arithmetic and morphological operations and scale space analysis. The extracted feature sets are used in the last part, the statistical model, to classify the object using known statistical characteristics of the object types to be identified.

In this paper, we focus on the pre-processing and analysis based on features from arithmetic operations as these are the most computationally intensive parts of application. Furthermore, the case of identification of fungi is based on features extracted using these operations. The statistical methods for classifying the contents of images are outside the scope of this paper and are described elsewhere [17].

The remaining parts of this section will describe the application in more detail.

2.1 Pre-Processing and Mask Generation

The pre-processing of the multi-spectral input image involves two steps: (i) the actual pre-processing and (ii) the mask generation.

The pre-processing step produces a noise-filtered normalized image. First, the pixel-wise average intensity across spectral bands in the multi-spectral input image is found. The mean of the resulting single-channel image is found and subtracted from each pixel. Following this, each pixel is then divided by the standard deviation to produce the normalized image. Finally, a 3×3 median filter is used to filter noise. These steps are illustrated in the more detailed overview of the application in Fig. 3.

The mask is used to select the interesting parts of the image, thus its generation varies depending on what information is extracted. For the input images used

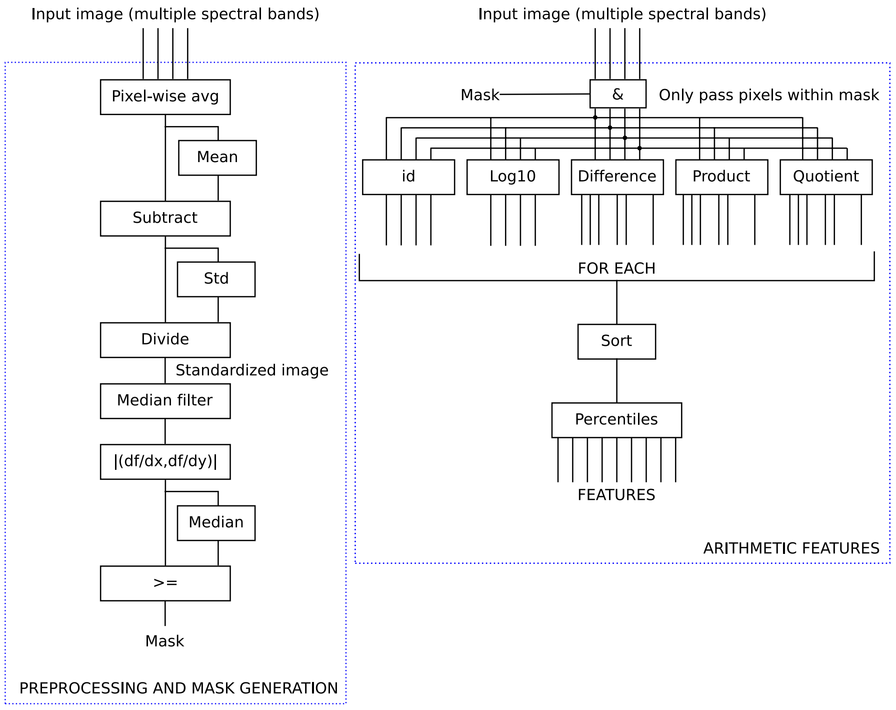

Fig. 3 Overview of immediately available parallelism in the application

in this paper, edge detection is used to find areas of interest in the images. For each pixel in the single-channel image previously constructed by pixel-wise average of the spectral images, the magnitude of the numerical gradient $|(\frac{\delta f}{\delta x}, \frac{\delta f}{\delta y})|$ is calculated where f describes the pixel values as function of coordinates (x, y). The median of the gradient values is found and all pixels whose gradient are greater than or equal to the median are included in the mask. They correspond to interesting areas in the image. The mask can be seen as a bit field where each bit corresponds to a pixel in the image. Each bit indicates if the pixel should be considered or not.

2.2 Arithmetic Feature Extraction

The mask is applied to each spectral band in the input multi-spectral image by discarding all pixels *not* in the mask. Five feature sets are extracted from the masked spectral bands of the input image, using five different arithmetic operations. Two operations take a single band at a time, while the other three operate on all pairs of bands. The two single-operand operations are the identity function, which just pass the image data through, and the pixel-wise base-10 logarithm. The other three operations find the pixel-wise difference, product and quotient of all pairs of spectral images. Each pair is considered only once, e.g. if $I_a - I_b$ is calculated, $I_b - I_a$ is not. If the input image has n spectral bands, the operations produce $2n + 3\frac{n(n-1)}{2}$ data sets.

The features of each feature set are extracted from the data sets produced by the arithmetic operations by finding the 1st, 5th, 10th, 30th, 50th, 70th, 90th, 95th and 99th percentiles of the pixel values. Determining the percentiles requires the data sets to be sorted individually.

3 Parallelization

We will now discuss the parallelization and the OpenMP implementation of the algorithm described in Sect. 2. The image processing algorithm differs from traditional high performance computing applications, such as matrix multiplication and physics simulation by having a significantly smaller data set and shorter execution time. Thus, the parallelization overhead can not be neglected.

The algorithm has two main parts as illustrated in Fig. 3. The pre-processing and mask generation part is governed by data dependencies, while the arithmetic feature extraction has parallelism immediately available between the feature sets, but also within the individual sets.

Profiling a sequential implementation of the algorithm revealed that 95% of the execution time is spent in feature extraction. Thus, it is the target for parallelization.

To summarize the task parallelism illustrated in Fig. 3, five independent feature sets are computed, which each produce n or $n(n-1)/2$ data sets for which the features are extracted by finding certain percentiles in the data sets. This means that the processing required for each feature set differs significantly. The feature extraction within each feature set should, in theory, be possible to split into parallel and equally sized workloads. However, non-uniform memory latencies caused by the target architecture may cause the execution time of each such parallel workload to differ. Scaling properties are discussed in Sect. 3.1 without considering architectural effects which are discussed in Sect. 3.2.

3.1 Scaling Properties

The running times of the feature sets differ by up to a factor of $(n-1)/2$ leading to load imbalance problems if different feature sets are run in parallel. In this paper, we therefore concentrate on extracting parallelism of each individual feature set.

As mentioned earlier, each feature set has n or $n(n-1)/2$ equally sized workloads immediately available, which can be run in parallel. But if n is less than the number of available processors $|P|$, in processor set P, more parallelism must be extracted from these workloads. This is also advantageous to reduce the imbalance slack for the feature sets containing $n(n-1)/2$ workloads, as this may not match a multiple of $|P|$.

Additional parallelism can be extracted by splitting data sets into subsets that can be computed independently and then recombined. This means adding an extra nested level of parallelism. The arithmetic operations of all feature sets have no inter-pixel dependencies, which mean that the processing of spectral bands into data sets can be split without creating any subset border synchronization issues. The sorting involved in the percentile calculation can be done on each subset separately followed by a merge of the sorted subsets before the percentiles are found. This allows the arithmetic operations to scale further, but with the overhead of merging the sorted subsets. It should be noted that the execution time of sorting each subset decreases by $d \times \log(d)$, where d is the number of pixels in the subset. The execution time of merging the sorted subsets increases proportionally with the number of subsets generated by the data set decomposition. This means that the amount of parallel work decreases and the sequential part increases with an increasing number of subsets. Thus, the gain of increasing parallelism is diminishing. In addition, the parallelization overhead, such as spawning threads and synchronization, may be significant at this level as the subsets are small.

The two levels of parallelism within each feature set, among data sets and among subsets, are denoted as l_0 and l_1 respectively. In our implementation, the parallelism at each level s_0 and s_1, can be adjusted independently, though the parallelism at l_0 is limited. The total number of subsets across all data sets w is given by $w = s_0 \times s_1$ and constitutes the total number of workloads in the application. Subset processing time is defined as the wall clock time spent performing arithmetic operations on the parts of the spectral band data that corresponds to the subset and time spent sorting the subset.

In order to avoid load imbalance, s_0 and s_1 should be determined such that w is equal to or slightly less than a $m \times |P|$, where m is a multiple of the number of available processors $|P|$. If w is slightly larger than $m \times P$, only one or a few processors will be involved in processing the last remaining subsets while the majority of processors are idle, causing a large slack. The slack can be reduced by increasing w. But as mentioned earlier, s_0 is limited by n or $n(n-1)/2$ and s_1 is limited by the merge sort overhead, which causes diminishing parallelization gain. Hence, determining s_0 and s_1 is a trade off between load imbalance and parallelization overhead.

3.2 Non-Uniform Memory Latency

The discussion in the previous section holds under the assumption that the execution time of equally sized workloads do not differ. This assumption will not hold for architectures with non-uniform memory latencies. Threads running on processors which have long memory latency will have longer subset processing times than threads with short memory latency.

In this application, all spectral bands of the image are loaded into memory sequentially and then processed in parallel. Assuming a first touch memory placement policy in a hierarchical memory system, all image data will be located in the part of main memory local to the processor loading in the images, e.g. in the local memory on the Uniboard processor board, in a Sun Fire architecture system. A thread running on a processor associated with a different branch of the memory hierarchy, e.g. a processor on a different Uniboard than the one holding the main memory containing the image data, will access all data through the global memory interconnect and therefore have a significantly longer memory latency. This is not easily solved through parallel loading

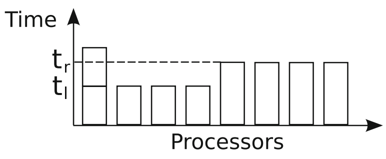

Fig. 4 Different workload execution times caused by non-uniform memory latency

of the spectral images due to the fact that the data set processing requires all combinations of spectral bands. Thus, the effective subset processing time depends on the location of the processor.

Combining this effect with the scaling properties means that even though the total number of subsets w matches the number of available processors, linear speedup can not be obtained. Consider a system with $|P|$ processors, where $P_l \subset P$ is the subset of processors having local memory access to the image data and $P_r \subset P$ is the subset of processors having remote memory access to the image data through global memory interconnect. The execution times of a subset on $p_i \in P_l$ and $p_j \in P_r$ are t_l and t_r respectively, where $t_r > t_l$.

In the case of uniform memory latency, where $P = P_l$ and $w = m \times |P_l|$, the total execution time is given by $T = m \times t_l$, ignoring the parallelization overhead. In the non-uniform case where $P = P_l \cup P_r$, T depends on the workload scheduling. Consider the case where w equals the number of processors $|P|$. In this case, every processor will process one subset each. Thus the total execution time is given by $T = max(t_l, t_r) = t_r$, if the parallelization overhead is assumed to be negligible. The processors in P_l finish before the processors in P_r, but the final result is not available until all processors have finished processing their subset. In the case where $w = 2 \times |P_l| + |P_r|$, assuming dynamic scheduling, $T = max(2t_l, t_r)$ as the processors in P_l will finish two subsets. If $2t_l > t_r$ the remote memory access of P_r, will not influence T. This is illustrated in Fig. 4. As a consequence of these two cases, resolving load imbalance may not result in the speedup outlined in Sect. 3.1. This applies to scaling both the number of processors and subsets, as these are both parameters that influence the load imbalance. Increasing the number of processors, such that $w = 2 \times |P_l| + |P_{r1}|$ becomes $w = |P_l| + |P_{r2}|$, where $|P_{r2}| = |P_{r1}| + |P_l|$, results in $T = t_r$. Thereby the total execution time reduction is only $2 \times t_l - t_r$, and not t_l.

The effect of load imbalance due to non-uniform memory latency also decreases significantly when w becomes much larger than the number of processors. Then again, the amount of parallelism available in the application may be limited and comes at a high cost in terms of parallelization overhead. The optimum solution is a trade off between parallelization overhead and load imbalance, where load imbalance is caused both by the algorithm itself, but also the architecture of the target execution platform. It should be noted that this is based on dynamic workload scheduling. Static workload scheduling will perform worse, due to varying execution times among the workloads.

4 OpenMP Implementation

The application was originally implemented in Matlab, and then ported to C using standard libraries only and without OpenMP parallelization in mind. All Matlab functions used in application were re-implemented using standard C-libraries to allow verification of the C-implementation by direct comparison to the results obtained using the original Matlab implementation. Subsequently, it was modified to meet the requirements for OpenMP parallelization.

In the sequential algorithm implementation, arithmetic feature extraction is implemented as a loop, where each iteration performs the arithmetic operation on a spectral band or pair of spectral bands, to form a new data set from which features are extracted. Unary arithmetic operators are applied to each individual spectral band in feature sets 1 and 2. These are implemented by a single loop through all the pixels in the spectral band. The feature sets 3, 4 and 5 are based on binary arithmetic operations between two spectral bands. First a list of pairs to be processed is generated. Then all pairs are processed using a loop. Similarly to the unary operations, the binary arithmetic operations are applied pixel by pixel in a single loop. Hence, the features sets are implemented using two nested loops.

Three different parallel versions of the application have been implemented using OpenMP. One implementation uses nested parallelism, while the two other variants do not make use of nested parallelism.

4.1 Nested Implementation

The nested version exploits the two levels of nested parallelism discussed in Sec. 3.1. The first level of parallelism, l_0, consists of the aforementioned loop over the data sets, which is already present in the sequential implementation. This loop is parallelized using the OpenMP [4] for workload sharing construct with dynamic scheduling, which is illustrated as the first thread fork in Fig. 5a.

Within each l_0-thread the data set is further split into subsets processed by another loop, which adds an extra loop to the implementation and forms the nested parallelism

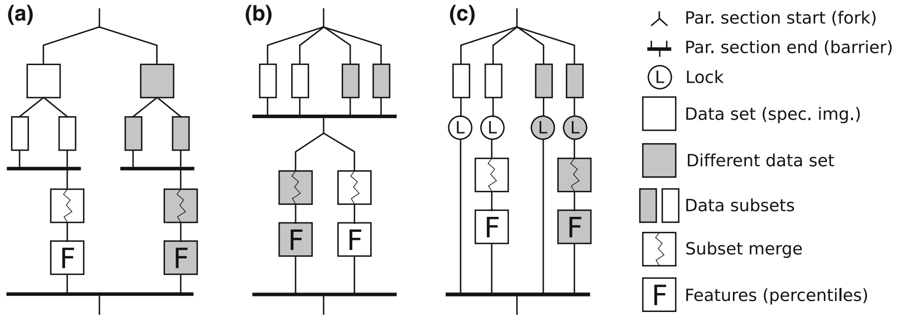

Fig. 5 Thread utilization in the three OpenMP implementations: **a** Nested parallelism, **b** non-nested parallelism, **c** improved non-nested parallelism using locks. Only two data sets are shown

level l_1. This is illustrated as the second thread fork in Fig. 5a. Sorting each individual subset before they are merged as described in Sec. 3.1 requires complete control over the subset partitioning, which prevents the use of the existing pixel loop for this purpose. When all nested threads have finished and reached the implicit barrier of the OpenMP work sharing construct, the l_0-thread will continue by merging the subsets and extracting the features.

Since the assignment of the nested threads can not be managed dynamically as proposed in Duran et al. [16] and it is generally not possible to assign the same number of threads to each nested parallel section while having one thread per processor, one thread is created for every subset without considering the total number of threads. Balancing the load optimally may require one thread to process iterations from two different nested loops, which is not possible in OpenMP 2.5. Thus, creating more threads than processors will enable operating system schedulers capable of dynamic thread migration to load balance the processors. However, spawning more threads than processors may also induce a large scheduling overhead in the operating system.

4.2 Non-Nested Implementation

To avoid relying on the operating system thread load balancing capabilities a non-nested version has been made. To flatten the two levels of parallelism, all $s_1 \times n$ or $s_1 \times n(n-1)/2$ subsets are enumerated and then processed in a single parallelized for loop, as shown in Fig. 5b. The number of threads is thereby completely independent of how many subsets the data sets are split into.

However, removing the two level hierarchy from the implementation and allowing subsets to be processed in any order means that it is no longer known when all subsets of each data set has been processed. Thus, merging can only take place when all subsets of all data sets have been processed. This is in contrast to the nested implementation where subsets are merged as soon as all nested threads belonging to a given data set have finished. Hence, merging and percentile determination must be implemented as a second parallelized loop executed afterwards. This is illustrated by the second thread fork in Fig. 5b after all threads in the first thread fork have finished.

This implementation has the advantage of having only one thread per processor, but separating the subset processing and merging in two parallel sections has great influence on the application memory access pattern and thus also the cache performance.

The processed subsets of a given data set can to a great extent be found in the caches of the processors that processed them. But implementing merging as a second loop, Fig. 5b, means that there is no guarantee that any of these processors will perform the merging of the data set and exploit that one subset is located in its local cache. In the nested implementation illustrated in Fig. 5a, however, the l_0-thread will be one of the nested threads and thus cache performance will be better.

Furthermore, by first processing all subsets and then merge them later, some of the subset data may be evicted from the cache due to limited cache capacity before they are merged. Merging the data set as soon as its subsets have finished, improves the probability of finding the subset data in the caches. However, this is very dependent of the cache size.

4.3 Improved Non-Nested Implementation

To avoid the cache performance disadvantages of the initial non-nested implementation, a second improved non-nested version has been implemented. This implementation merges subsets as soon as all subsets of a data set have been completed and improves locality and thus cache performance.

Subsets are enumerated like in the first non-nested implementation, but instead of merging the data sets in a second loop, it is integrated into the subset processing. The thread finishing the last subset of a data set is responsible for merging all the subsets of the data set before it can process another subset as illustrated in Fig. 5c. This is implemented by assigning an OpenMP lock and a counting variable to each data set, illustrated by the “L” in Fig. 5c, which keeps track of how many subsets of the data set that have been processed.

A drawback of this version is that the subset processing times are not equal. Though, dynamic scheduling is used for the workload sharing construct, it can not be expected to counter this effect completely.

A similar implementation could be done using tasks, which were introduced in OpenMP 3.0 [5]. Subset processing can be implemented using the new task construct. However, the OpenMP 3.0 specification does not support task data dependency notation. Thus the *taskwait* construct must be used to determine when the subset processing has finished and the subsets can be merged, which is similar to using the lock in our implementation. The number of tasks to be processed is known, so the application will not generate tasks dynamically. In all, we envision that the net advantage of using tasks is slightly simpler code.

5 Results and Discussion

This section presents results obtained by running the nested and the two non-nested parallelized algorithm implementations using 16 processor cores on the test platform and compares these with the scalability issues discussed in Sect. 3.

5.1 Test Setup

In the presented results, the algorithm has been used to calculate all arithmetic feature sets of the input images. The input images are ten images, each containing nine spectral bands in a resolution of 777×776 pixels. The light intensity of each pixel is represented by a double precision floating point number.

The test platform used for producing the results in this paper is a Sun Fire E6900. The machine has 48 UltraSPARC IV CPUs. Each processor has two cores running at 1200 MHz and has 8 MB L2 cache per core. The machine is running Solaris 10. Compilation has been done using the Sun C compiler version 5.9 patch 124867-01 using these options: `-fast -xarch=sparcvis2 -m32 -xopenmp=parallel -lm`.

The image loading time has been excluded from the measurements by loading all ten images, one by one, into main memory before they are processed. Warm up is done by processing all ten images once. To increase the accuracy of the measurements the presented results are based on the average execution times of ten or twenty consecutive runs of each feature set, where all ten images are processed. The number of runs is determined by the execution time of the particular test case. Using larger input images is not representative for the practical use of the algorithm and will lead to unrealistic results.

The average sequential execution times for feature sets 2 and 3 are 35 s and 127 s respectively, processing all ten multi-spectral image.

5.2 Parallel Efficiency

All tests have been limited to a maximum of 16 processor cores. Several parallelization approaches have been tested to investigate how the two levels of parallelism, l_0 and l_1, influence the parallel efficiency. It should be noted that even though all tests have 16 processors available, they may not all be utilized, depending on the number of threads in the particular test case. The nested version creates more than 16 threads in some tests. In order to prevent the threads to use more than 16 processor cores in these cases, a 16 core processor affinity set was specified using the `SUNW_MP_PROCBIND` environment variable for all runs with the nested version. This method may potentially lead to uneven load on the cores, but dynamic workload scheduling counters this effect and no negative effects are observed in the results. Even though the main focus of the tests is parallel efficiency, scalability trends can also be extracted from the results of the nested version.

Figures 6 and 7 illustrate the speedup obtained in feature sets 2 and 3 for the nested version by increasing the number of threads at l_0 with different data set partitioning at l_1. As mentioned in Sect. 4, one l_1-thread is created for each subset. The measurements of feature set 1, 4 and 5 are not significantly different from what can be observed in feature set 2 and 3, thus they are not shown.

Parallelization at l_0 does not impose any parallelization overhead except for thread creation overhead. However, parallelism is limited to nine l_0 threads in feature set 2. Linear relative speedup should be expected, when more threads can be created to utilize more processors. This can be observed in Fig. 6 for one to eight threads with no data set partitioning for feature set 2, which means $w = 9$. As discussed in Sect. 3.2, going from 8 to 16 threads would double the theoretical speedup since load imbalance is improved. However, a speedup of only 1.5 is obtained, because $t_r > t_l$ meaning that data has to be fetched from a remote Uniboard leading to higher memory latency.

This effect has been confirmed by measuring the execution time of each l_0-thread, when running three and nine threads in parallel without any nested l_1-threads. The Sun Fire E6900 UltraSPARC IV Uniboards have four processors each with two cores, which means that if more than eight threads are used, some of them will be running on different processor boards. Figures 8 and 9 show histograms of thread execution

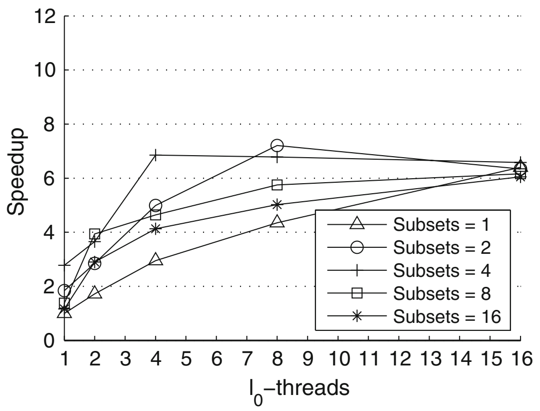

Fig. 6 Speedups for the nested version of feature set 2 with 16 processors

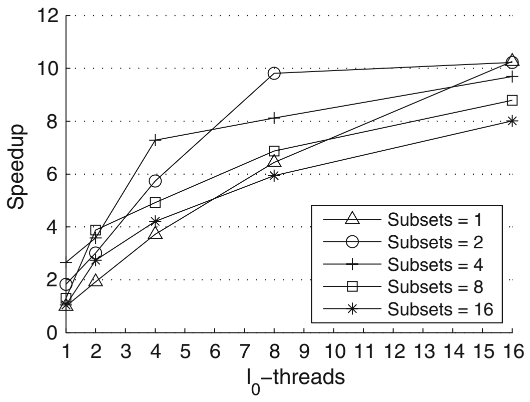

Fig. 7 Speedups for the nested version of feature set 3 with 16 processors

time using three and nine threads. It can be seen that using three threads, the histogram has a narrow range, while the histogram of nine threads is spread out. The lower part represents threads running on the board that holds the main memory containing the images, while the upper part is slow threads running on a different board. The ratio between a fast and a slow thread match the speedup obtained going from eight to 16 l_0-threads in Fig. 6.

As discussed in Sect. 3.1 parallelization at l_1 has sequential overhead. This can be observed in Figs. 6 and 7 when comparing the speedups of tests with one l_0-thread and increasing the number of l_1 threads. Even though more processors are utilized, the sequential merge eventually outweighs the parallelization speedup. Having more threads than processors also adds thread switching overhead as several threads share a single processor core. It can be observed on both Figs. 6 and 7 that matching $s_0 \times s_1 = |P|$ leads to best results in general.

The effects observed in the results of feature set 2 can also be seen for feature set 3. However, the amount of parallelism available at l_0 is potentially 36 data sets. This leads to better parallel efficiency as less parallelism needs to be extracted at the l_1 level, where the sequential parts are limiting. The efficiency observed in feature set

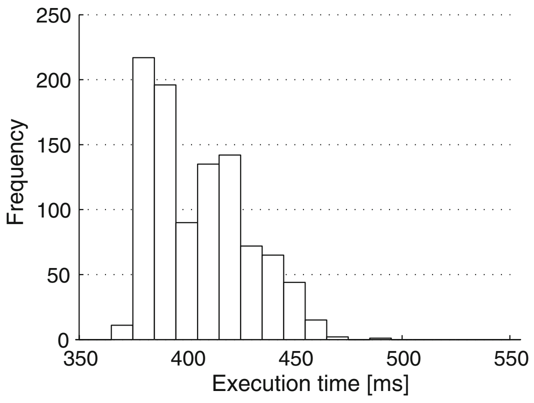

Fig. 8 Thread execution time histogram when running 3 threads

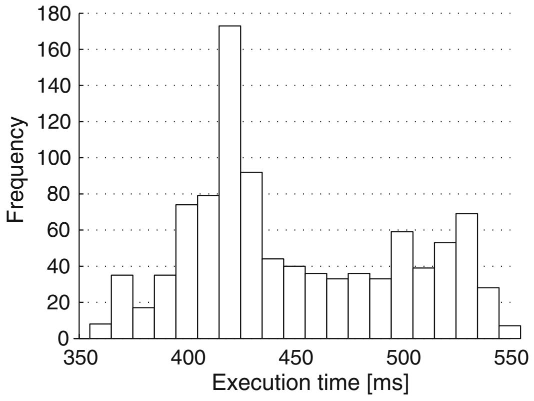

Fig. 9 Thread execution time histogram when running 9 threads

2 is considered more realistic for real uses of this application, as only a subset of the features is typically needed [17].

The relation between work partitioning and the number of threads is removed in the non-nested versions. The number of threads is completely independent of the subset partitioning. Splitting the data sets creates more workloads that may lead to better workload balancing among the threads. In Figs. 10 and 11, it can be observed that the improved non-nested version performs up to 24% better than the nested version. Comparing speedup of the two implementations, when the number of subsets increases, shows the overhead of having more threads than processors. With few threads, the two implementations have very similar performance, while the improved nested version performs significantly better with many subsets. The graphs representing the nested version in Figs. 10 and 11 show the best performing thread configuration with the corresponding number of subsets. However, it can be seen that when increasing the number of threads, parallelization overhead counters any speedup gained by increased parallelism. The effect of the parallelization overhead at l_1 can also be observed clearly for the improved non-nested implementation, as it is the cause of the decreasing speedup when having 8 and 16 subsets.

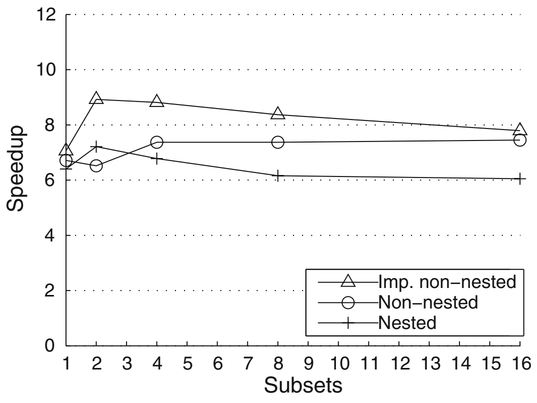

Fig. 10 Speedups for all implementations of feature set 2 with 16 processors

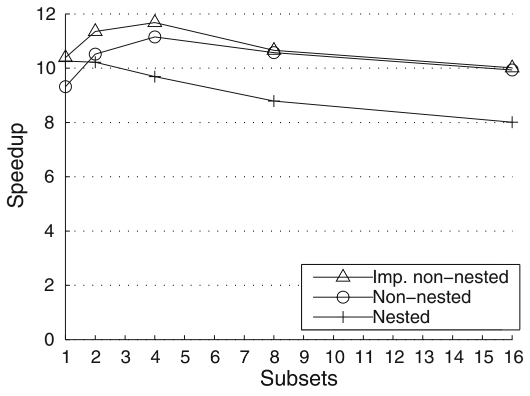

Fig. 11 Speedups for all implementations of feature set 3 with 16 processors

The performance of the initial non-nested implementation is difficult to predict due to its issues with regard to cache utilization. At best it can reach the performance of the improved version, but the real performance depends on the thread scheduling done at run-time and the OpenMP library implementation. When having only one or two subsets per data set, the impact of thread scheduling is large, as merging data on a processor without any of the subsets in its cache, will perform significantly worse than if it had a subset present in its cache. This effect will diminish as the number of subsets increases, since the subsets become smaller. The difference of accessing all data in other caches or main memory and having one small subset in the local cache becomes very small. This effect is shown in Figs. 10 and 11, where the speedup of the initial non-nested implementation approaches the improved one for larger numbers of subsets. The observed speedup of the initial non-nested implementation stresses the importance of considering cache utilization in parallel programming.

6 Conclusions

This paper has investigated an image processing application that can be targeted for a future multi-processor system-on-chip embedded system. Such a system is inherently parallel, and the major challenges in parallelizing the application have been identified.

These challenges include limited directly exploitable parallelism, a significant parallelization overhead caused by small workloads and difficult load balancing which is aggravated by non-uniform memory latencies.

Three different parallelization approaches have been applied, each of which required increasing implementation effort. Restrictions on thread management when using nested parallelism in OpenMP makes it difficult to optimize the number of threads and thread utilization. Flattening the nested loops removes these restrictions, but also removes implicit information on completion of the individual tasks. Our results show that this causes poor cache performance and has led us to implement a cache optimized version of the application using explicit locks. Using these application specific improvements, a 14–24% gain in parallel efficiency was observed.

We have shown that despite these challenges, a relative speedup in excess of 9 on a 16 CPU system can be achieved.

Acknowledgements We gratefully acknowledge the support from the Danish Center for Scientific Computing at the Technical University of Denmark. In particular we acknowledge the technical support from Bernd Dammann. We also would like to thank Jens Sparsø for many thoughtful comments. This work has been supported by the HiPEAC^2 European Network of Excellence.

References

1. Vangal, S.R., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Singh, A., Jacob, T., Jain, S., Erraguntla, V., Roberts, C., Hoskote, Y., Borkar, N., Borkar, S.: An 80-Tile Sub-100-W TeraFLOPS processor in 65-nm CMOS. IEEE J. Solid-State Circ. **43**(1), 29–41 (2008)
2. Shah, M., Barren, J., Brooks, J., Golla, R., Grohoski, G., Gura, N., Hertherington, R., Jordan, P., Luttrell, M., Olson, C., Sana, B., Sheahan, D., Spracklen, L., Wynn, A.: UltraSPARC T2: a highly-threaded, power-efficient, SPARC SOC. In: Proceedings of IEEE Asian Solid-State Circuits Conference, pp. 22–25 (2007)
3. Asanovic, K., Bodik, R., Catanzaro B, C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report. EECS Department, University of California, Berkeley (2006)
4. OpenMP Architecture Review Board. OpenMP Application Program Interface 2.5 [Online]. Available: http://www.openmp.org. Accessed 8 Oct 2008 (2005)
5. OpenMP Architecture Review Board. OpenMP Application Program Interface 3.0 [Online]. Available: http://www.openmp.org. Accessed 8 Oct 2008 (2008)
6. Magarshack, P., Paulin, P.: System-on-chip beyond the nanometer wall. In: Proceedings of Design Automation Conference, pp. 419–424 (2003)
7. Benini, L., De Micheli, G.: Networks on chips: a new soc paradigm. Computer **35**(1), 70–78 (2002)
8. Bertozzi, D., Jalabert, A., Murali, S., Tamhankar, R., Stergiou, S., Benini, L., De Micheli, G.: NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. IEEE Trans. Parallel Distrib. Syst. **16**(2), 113–129 (2005)
9. Bell, S., Edwards, B., Amann, J., Conlin, R., Joyce, K., Leung, V., MacKay, J., Reif, M., Bao, L., Brown, J., Mattina, M., Miao, C.-C, Ramey, C., Wentzlaff, D., Anderson, W., Berger, E., Fairbanks, N., Khan, D., Montenegro, F., Stickney, J., Zook, J.: TILE64 processor: a 64-Core SoC with mesh interconnect. In: IEEE International Solid-State Circuits Conference—Digest of Technical Papers, pp. 88–598 (2008)

10. Phillips, R., Watson, L., Wynne, R.: Hybrid image classification and parameter selection using a shared memory parallel algorithm. Comput. Geosci. **33**(7), 875–897 (2007)
11. Meerwald, P., Norcen, R., Uhl, A.: Parallel JPEG2000 image coding on multiprocessors. In: Proceedings of the 16th International Parallel and Distributed Processing Symposium, pp. 9–14 (2002)
12. Terboven, C., Deselaers, T., Bischof, C., Ney, H.: Shared-memory parallelization for content-based image retrieval. In: Proceedings of European Conference on Computer Vision Workshop on Computation Intensive Methods for Computer Vision, Graz, Austria (May 2006)
13. an Mey, D., Sarholz, S., Terboven, C.: Nested parallelization with OpenMP. Int. J. Parallel Program. **35**(5), 459–476 (2007)
14. Blikberg, R., Srevik, T.: Load balancing and OpenMP implementation of nested parallelism. Parallel Comput. **31**(10–12), 984–998 (2005)
15. Duran A., Silvera R., Corbalan J., Labarta J.: Runtime adjustment of parallel nested loops. In: Proceedings of the Workshop on OpenMP Applications and Tools, pp. 137–147 (May 2004)
16. Duran, A., Gonzalez, M., Corbalan, J.: Automatic thread distribution for nested parallelism in OpenMP. In: Proceedings of the International Conference on Supercomputing, pp. 121–130 (2005)
17. Clemmensen, L.H., Hansen, M.E., Frisvad, J.C., Ersboll, B.K.: A method for comparison of growth media in objective identification of penicillium based on multispectral imaging. J. Microbiol. Methods **69**(2), 249–255 (2007)