A high performance face detection system using OpenMP[*]

P. E. Hadjidoukas, V. V. Dimakopoulos
Department of Computer Science, University of Ioannina, 45110 Ioannina, Greece

M. Delakis, C. Garcia
Orange Labs, 4, rue du Clos Courtel, 35512 Rennes, France

Abstract

We present the development of a novel high-performance face detection system using a neural network-based classification algorithm and an efficient parallelization with OpenMP. We discuss the design of the system in detail along with experimental assessment. Our parallelization strategy starts with one level of threads and moves to the exploitation of nested parallel regions in order to further improve, by up to 19%, the image processing capability. The presented system is able to process images in real time (38 images/sec) by sustaining almost linear speedups on a system with a quad-core processor and a particular OpenMP runtime library.

Keywords: face detection, image processing, nested parallelism, OpenMP, multi-core computing.

1 INTRODUCTION

Over the last decade automatic face detection has attracted the attention of researchers as a necessary preparation step before face recognition, facial feature extraction or other advanced human/computer interaction systems. The value of a face detection system is determined first by the detection accuracy and the false alarm rate it scores, and second, by the computational cost it incurs to process an image i.e., the cost to detect and localize all the faces that may exist in the image. Speed and efficiency became quite important from the moment of the first commercial applications that need face detection.

Following the need for a very fast face detection system, Viola and Jones [1] proposed an approach based on a serial cascade of Haar-based classifiers of increasing complexity: the simple ones are charged to reject quickly image areas quite distant from a face, while the more complex classifiers are used to make the final decision in the remaining and more difficult to handle image areas. While the key feature of their system is the ability to process video in real-time, good detection accuracy rates are also achieved.

Garcia and Delakis [2] presented a face detection approach based on a special highly-structured neural network topology, called Convolutional Neural Networks. This approach features a very good degree of robustness to face pose, illumination, facial expression and also scored the best results so far reported on the standard benchmark of face detection [3]. In terms of speed, however, the system cannot hit top performance as it can only process a few frames per second, depending on the processor. One potential source of optimization is the exploitation of concurrency that is present at various levels. In this paper,

[*]This is an author-generated preprint of the article "A high-performance face detection system using OpenMP", published in Concurrency and Computation: Practice and Experience, Vol. 21, No. 15, Oct. 2009. The original article can be found here.

we present the parallel version of this face detector using an efficient parallelization strategy on shared-memory multiprocessors. Our motivation for this work is to take advantage of parallel processing in order to boost the performance of the best so far, in terms of accuracy, face detector. With the advent of multi-core processors, our system succeeds in providing high-performance/high-accuracy face detection in a very cost-effective way.

The parallel programming model that we used to parallelize the face detection system is OpenMP [4], which is the standard parallel programming model for the shared memory architecture. Most vendors participate in the evolution of OpenMP and provide custom implementations. OpenMP allows for incremental and scalable parallelization of existing code, providing thus a fast way of parallelizing the face detection system with minimal changes to its data structures and algorithm. The parallelization of the face detection system demonstrates significant and cost-effective improvements in the processing time of a single image. For a typical single-face image, the sequential version fails to achieve a processing rate higher than 11 images per second on a quad-core processor system. In contrast, the parallel version manages to provide real-time response (i.e. $\geq$25 images/sec) when 3 or more OpenMP threads are used.

Our parallelization strategy also includes nested parallelization of the face detection system. Nested parallelism is a major feature of OpenMP that can improve application performance in many cases. Our custom OpenMP implementation, based on the OMPi OpenMP C compiler with lightweight threading, overcomes the significant runtime overheads of other OpenMP compilers when multiple parallel regions are simultaneously active. As a result, we manage to exploit effectively the fine-grain parallelism of the face detection system and to further improve the overall performance by up to 19%. To the best of our knowledge, it is the first time that nested OpenMP parallelization has been applied to such an application. Finally, OpenMP significantly increases the throughput of the face detection system, through the concurrent processing of multiple images. The dynamic assignment of parallel loop iterations that OpenMP supports provides balanced processing of images by multiple processors.

The paper is organized as follows. In Section 2, we present an overview of the sequential version of the face detection system, discussing the topology of the convolutional neural network, the search strategy for finding faces and the construction of the special filtering pipeline for speeding up computation. The parallelization of the face detection system using OpenMP is discussed in Section 3 and the implementation details are presented in Section 4. Sections 5 and 6 discuss the exploitation of nested parallelism and the efficient batch processing of multiple images, respectively. Finally, Section 7 concludes our work.

2 THE SEQUENTIAL FACE DETECTION SYSTEM

In this section we provide the necessary background for the sequential version of the face detector. We present the topology of the neural network and the way it is used in order to scan an image for faces. The interested reader is referred to [2] for a more detailed presentation and experimental analysis of the face detection system. At the end of this section, we provide a pseudocode summary of the whole face detection operation, which will serve as a discussion basis for the following sections of this paper.

2.1 The Neural Network Topology

The neural network we used for face detection is based on the special Convolutional Neural Network architecture model, originally introduced by LeCun *et al* [5] in the field of optical character recognition. Convolutional Neural Networks are standard Multilayer Perceptrons, which make extensive use of convolutions with adjustable masks and subsampling operations in order to extract appropriate and robust

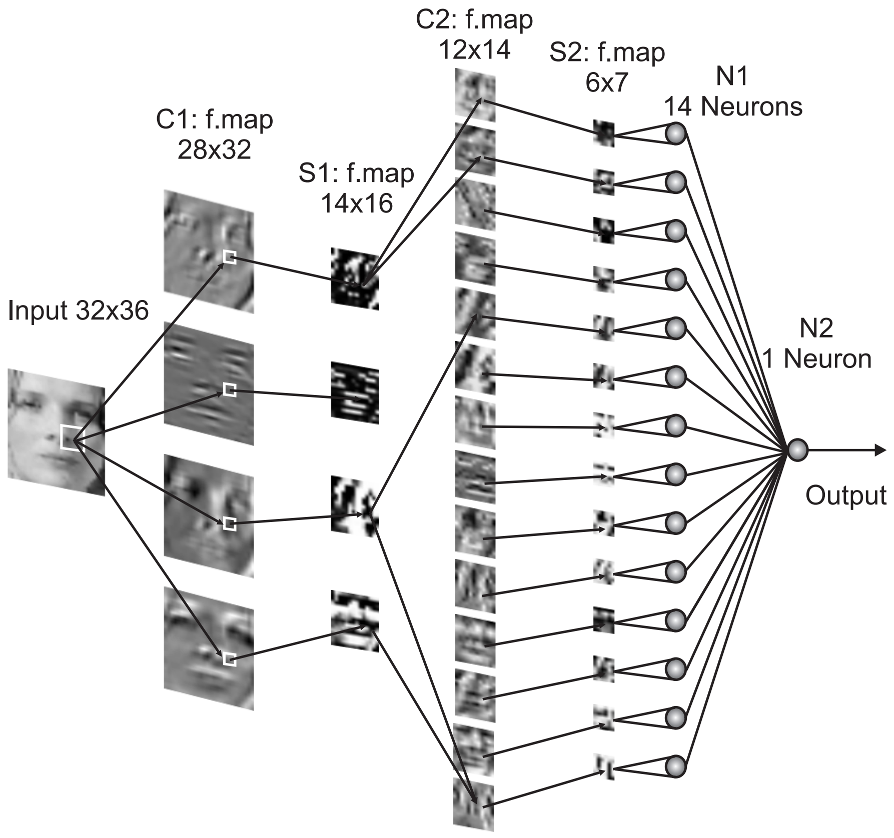


Figure 1: The convolutional neural network architecture.

features at the first layers. The last layers of network contain standard sigmoid neuron layers, which perform the actual classification and produce the network output. This class of neural network is particularly interesting for face detection as the preprocessing and feature set extraction are built-in in the network and not hand-crafted.

The topology of the network is illustrated in Fig. 1. The input of the network consists of a gray image of 32×36 pixels, with the intensity values linearly scaled between -1.0 and 1.0. Layers C_1 through S_2 perform successive convolutions and subsampling operations, while the last two layers produce the output of the network. The desired response of the network is set to +1.0 when the input contains a well-centralized frontal or semi-frontal face, and is set to -1.0 in the other cases.

Layer C_1 performs four distinct convolutions on the input image in order to extract four feature sets at various locations of the input. The convolutional filters are linear and consist of four 5×5 masks. After a convolution, an adjustable bias is added to the final result at each location. The net result of the operation of the first layer is four 28×32 planes, called *feature maps*, which contain an initial set of features. In fig. 1, we provide an approximate visualization of the actual extracted features from each feature map.

The convolutional layer C_1 is followed by the S_1 layer that performs non-linear subsampling over 2×2 non-overlapping areas at the corresponding feature map of C_1. More precisely, the four corresponding inputs are averaged, multiplied by a trainable coefficient, added to a trainable bias, and finally passed through a sigmoid. This layer thereby contains four feature maps of approximately half-dimension of the network input. While a feature set exhibiting explicit invariance to translation and rotation is not targeted, the features extracted by the convolutional network have a limited built-in invariance to translation, rotation, and deformation of the input. The range of tolerance is ultimately defined by the training set. In our experiments, all the face examples were cloned with a 20 degrees rotation and a smoothed version of the original face.

Layer C_2 contains convolutional features maps, as in C_1, partially connected to the feature maps of the S_1 layer. Mixing the outputs of feature maps helps in combining different features, thus in extracting more complex information. The network of Fig. 1 has 14 feature maps in the C_2 layer. Each of the 4 subsampling feature maps of S_1 is convolved by two different trainable 3×3 masks, providing 8 feature maps in C_2. The remaining 6 feature maps of C_2 are obtained by fusing the results of 2 convolutions on each possible pair of feature maps of S_1. The operation of the S_2 and C_2 layers is similar to that of S_1 and C_1, respectively. The signal now is reduced to 14 feature maps of quite small dimensionality of 6×7, compared to the 32×36 of the input.

Layers N_1 and N_2 contain standard sigmoid neurons. In layer N_1, each neuron is fully connected to only one feature map of the layer S_2. The unique output neuron is connected to all the neurons of layer N_1. Overall, the network contains 951 weights. We used the standard backpropagation algorithm and a training set of several thousands face and non-face patterns to adjust them.

2.2 Image Scanning

In Section 2.1, we described a neural network trained to respond with positive answers when its input contains a face and with negative ones otherwise. To accomplish the task of face detection, however, we also need a search strategy to detect an unknown number of faces that may exist in an any given image, at every possible location or scale (i.e., the size of the face). Our search strategy is described first. Next, we describe how the image scanning operation can be transformed to a fast filtering pipeline by applying the network filters to the whole image at once.

2.2.1 Search Strategy

The operation of image scanning for faces using the trained neural network is divided in four phases. First of all, in order to detect faces at different scales, the input image is repeatedly sub-sampled by a factor of 1.2 resulting in a pyramid of images. In the second step and for every image of this pyramid, we perform a rough scanning for face candidates by applying the network with a step of 4 pixels in both directions. After the filtering of the images, the results from each image scale are projected back to the original image. The situation is depicted in the top left image of Fig. 2, where we see the position and scale of each positive response of the network.

In the third step, all these positive answers are then grouped according to their proximity in image positions and scale, resulting in a list of candidate faces. In the top right image of Fig. 2, six such groups are formed. In the fourth and last step of this procedure, every candidate face is processed by the network in a local and finer pyramid of scales and positions around it. This step gives us a more reliable estimate about the positive activation around it and the exact location and scale. We see around each target an increased activation of the network with that of the true face being much stronger. In the light of this observation and using a predefined threshold, we can easily discard the false targets. The exact location and scale of the face are given from the location and scale of the maximal response of the network during the fine scanning. The outcome of this process is depicted in the bottom right image of Fig. 2, where the (true) face of the image was correctly detected while the false targets were discarded. The exact location and scale of the face are correctly detected, as well.

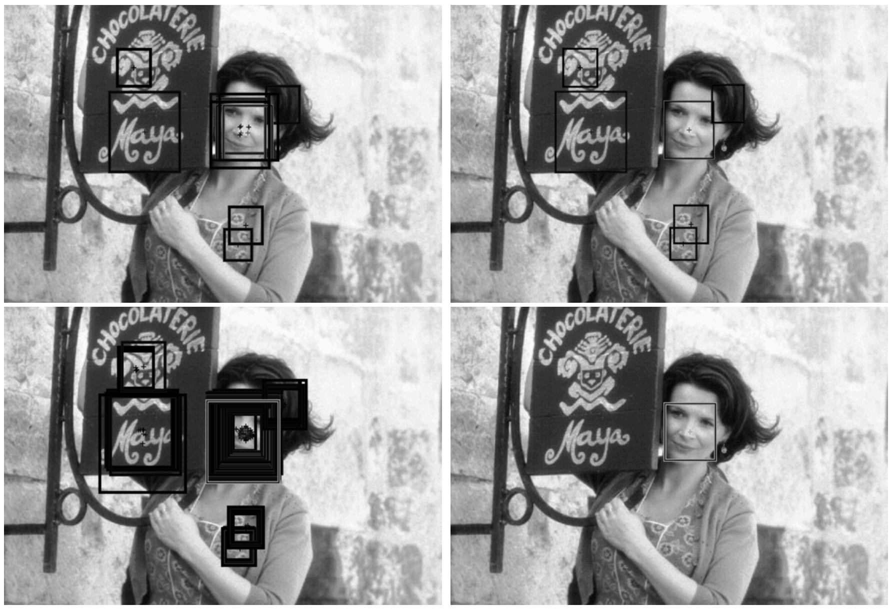


Figure 2: An illustration of the image scanning process.

2.2.2 Fast Filtering Pipeline

The proposed topology, being a Convolutional Neural Network, has another decisive advantage. Each layer of the network essentially performs parallel convolutions of small-size kernels. For each convolution, a very large part of the computation is identical between two neighboring input windows and therefore is redundant if the network is applied distinctively in neighboring locations. This redundancy is naturally eliminated by performing the convolutions of each layer on the entire input image at once. The overall computation consists of a pipeline of convolutions and non-linear transformations over the entire image. The filtering pathway is identical to that of Fig. 1, with the only difference that the input can now be of variable size. The final outcome of this pipeline is an image four times smaller than the input one, carrying the results of the network scanning as if this had been applied distinctively on the input image with a step of 4 pixels. We used this pipeline in the second step of the image scanning procedure. A similar pipeline, providing with the results of the network with step 1 pixel, is used in the fourth step, that of the fine scanning around the targets.

2.3 Face Detection Accuracy

Regarding the detection accuracy of the system, is was experimentally tested on the widely referenced CMU face detection benchmark [3]. The system detected 90.3% of the 507 faces with 8 false alarms, which is, to our knowledge, the best reported performance on this benchmark. The system of Viola *et al* [1] achieved 88.4% detection rate with 31 false alarms. In addition, we tested further our system on two more test sets with different image conditions and statistical properties achieving similar results. The network also exhibits a great degree of robustness to input image noise and deformations like blurring, Gaussian noise, and varying face pose. A detailed experimental analysis of our system with respect to face detection quality can be found in [2].

```
 1    make image pyramid                  /* step 1 */
 2    for each image scale S              /* step 2 */
 3    {
 4        for i=1 to 4
 5        {
 6            convolve with kernel C1i
 7            subsample with weights S1i
 8        }
 9        for i=1 to 14
10        {
11            convolve with kernel C2i
12            subsample with weights S2i
13        }
14        for i=1 to 14
15        {
16            apply neuron N1i
17        }
18        apply neuron N2
19    }
20    grouping to target areas            /* step 3 */
21    for each target                     /* step 4 */
22        fine scanning
```

Figure 3: Summary of the image scanning procedure.

2.4 Summary

To help the discussion on the parallelization process, we summarize in Fig. 3 the image scanning procedure in the form of pseudocode. In line 1, we construct the pyramid of different scales (first step). The application of the pipeline for target detection follows in lines 2 to 19. The third step is performed in line 20. Finally, the local searches around targets (fourth step) are performed in lines 21 and 22.

3 PARALLELIZATION FOR REAL-TIME PERFORMANCE

Our face detection system utilizes a fast pipeline method, in which the whole image scanning for faces is reduced into a series of basic image filtering operations like convolutions and subsamplings. This procedure allows for elimination of redundant computation which occurs when scanning the image by applying the neural network at every possible position distinctively. Due to its simplicity, this filtering pipeline is highly parallelizable.

The parallelization of the face detection system is based on OpenMP [4], the standard programming model for a wide range of parallel platforms including small-scale SMPs, multi-core servers and scalable ccNUMA multiprocessors. OpenMP defines a portable programming interface based on directives, i.e. annotations that are inserted explicitly by the user. The directives are interpreted by the OpenMP compiler into appropriate runtime calls and multiple threads participate in the parallel execution of the application. These threads share the program's global data but each one has also its own private memory. Since OpenMP uses the fork/join execution mode, an application starts with a single (main) thread and creates a team of additional threads once it encounters a parallel region. The threads of a team synchronize after the end of a parallel region and return to the runtime system or the operating system, while the main thread continues the program execution.

OpenMP offers the advantage of simplicity, since the shared-memory API can be used to construct a parallel program as a natural extension of its sequential counterpart, thus enabling incremental code development. It supports both coarse and fine grain parallelism and allows for runtime scheduling of tasks. The OpenMP directives provide a powerful set for parallelism creation and an extended parallel programming model, covering a wide range of application needs, from loop-level to functional parallelism. Furthermore, OpenMP hides the architectural details and relieves the programmer from the duty of data distribution among processors.

Disadvantages of OpenMP include the problems generated by the data placement policies and the runtime overheads during the execution of small cost loops. Moreover, coarse grain parallelism requires similar management as in the case of the Message Passing Interface (MPI) [6], the standard programming model for distributed memory multiprocessors, making application development particularly cumbersome.

The parallelization of the face detection system using OpenMP aims at two distinct targets:

- Fast response time for a single image: the OpenMP parallelization provides a portable, efficient and cost-effective solution for real-time face detection, especially on small-scale SMP systems.
- Faster processing of large volumes of images: OpenMP can also increase the throughput of our face detection system when it is applied to large image databases. The dynamic assignment of parallel tasks that OpenMP supports provides balanced processing of images by multiple processors.

4 PARALLELIZATION STRATEGY I: SINGLE-LEVEL PARALLELISM

The first step of the algorithm (line 1 of Fig. 3) performs the construction of the pyramid of scales and has negligible execution time. Therefore, we chose not to parallelize this step, which is executed by the master thread only.

We first applied the OpenMP parallelization to the loop of the second step (line 2 of Fig. 3) using the OpenMP `parallel for` directive with the dynamic scheduling policy. Two were the most important issues we faced:

- The OpenMP standard specifies that the number of iterations of a parallel loop must be known before its execution. However, in the sequential version, this is not the case because the number of iterations of this loop is equal to the number of different scales, which is computed at runtime. We worked around this problem by adding an extra precomputation step which is executed by the master thread before issuing the parallel loop.
- Load imbalance: the number of the iterations in this step depends on the size of the image and usually lies between 8 and 12. However, all iterations do not incur the same processing load. Iterations with larger index correspond to lower execution time. We thus decided on using the dynamic scheduling policy with the default chunk size, which is equal to 1. According to this policy, loop iterations are assigned one-by-one to threads, as the threads request them. Each thread executes a single iteration and then requests another one, until all iterations have been exhausted.

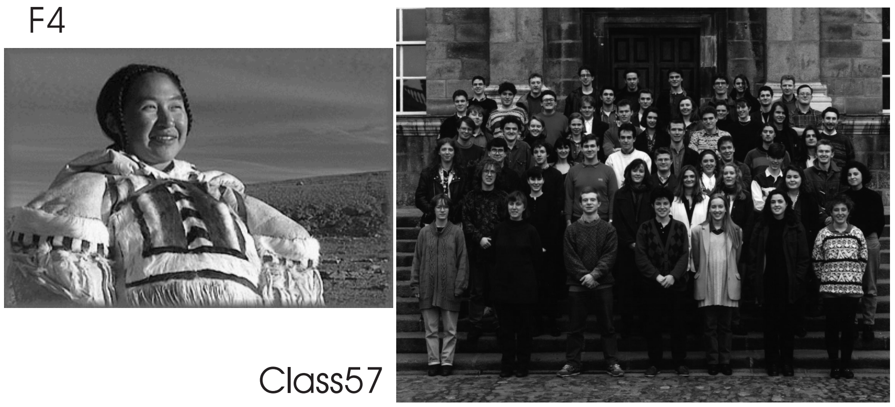


Figure 4: The two images we used for performance evaluation. The f4 image (355×237 pixels wide) is part of the Cinema test set of [2], while class57 (1280×1024 pixels wide) is part of the CMU test set [3].

Table 1: Runtime features and percentage of execution time.

Image	Scales - Step 2	Targets - Step 4
f4	11 (85.5%)	2 (14.5%)
class57	13 (80.0%)	65 (20.0%)

Like the first step, the cost of the third step (line 20 of Fig. 3) is negligible and thus it is executed sequentially too. Finally, the loop of local searches in line 21 of Fig. 3 is parallelized using OpenMP, employing the dynamic scheduling policy with the default chunk size as well. The number of iterations of this loop equals the number of targets discovered, which cannot be known beforehand because it depends on the image complexity and the number of existing faces. Furthermore, the varying processing time for each target justifies the selection of the particular loop scheduling policy.

4.1 Experimental Setup

The OpenMP parallelization does not alter the functionality of our face detection system. This means that the OpenMP version produces identical output for a given image as the sequential one, regardless of the number of threads used. In what follows, we are interested exclusively in the speed of our system. We evaluate the performance of the parallel face detection system on a small-scale SMP system.

For the performance evaluation of our OpenMP implementation, we have used the two images illustrated in Fig. 4. The first image (f4) represents a typical single-face image while the second one (class57) corresponds to an extremely computational demanding case because of its large size and the high number of existing faces (57). Table 1 contains the number of scales and targets of both images and the approximate execution time breakdown of their processing.

We conducted all our experiments on a server equipped with an Intel Xeon Quad-core X5355 processor (2.66GHz, 4MB L2 cache) and 2GB of main memory. The operating system was Debian Linux 2.6. We provide performance results for two free commercial and two freeware OpenMP C compilers. The commercial compilers are the Intel C++ 10.0 compiler (ICC) and Sun Studio 12 (SUNCC) for Linux. The

Table 2: OpenMP compilers and their characteristics.

Compiler	Version	Availability	Threading model
Intel C++	10.1	commercial, free	kernel-level
Sun Studio	12	commercial, free	kernel-level
GNU GCC	4.2	freeware	kernel-level
OMPi	0.9.0	freeware	hybrid (kernel & user level)

freeware ones are GNU GCC 4.2.0 and OMPi 0.9.0.

The Intel compiler [7] incorporates many well-known and advanced optimization techniques that fully leverage Intel processor features for higher performance. The Intel compiler uses a common intermediate representation (named IL0) so that the OpenMP parallelization and a majority of optimization techniques are applicable through a single high-level intermediate code generation and transformation, irrespective of the source language.

The Sun Studio application development suite [8] is a comprehensive, integrated set of compilers and tools for the development and deployment of applications on Sun and Linux platforms. The Sun Studio tools support writing, debugging, and analyzing the performance of OpenMP applications.

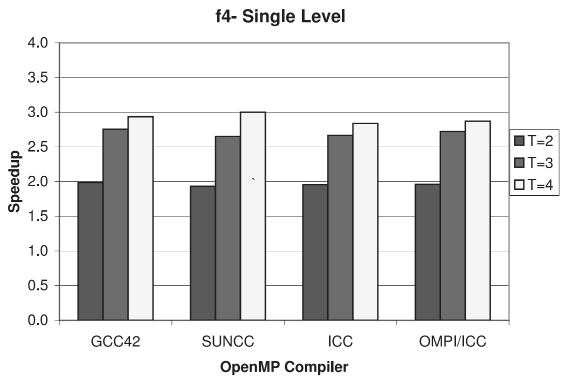
The GOMP project developed an OpenMP implementation for the C, C++, and Fortran 95 compilers in the GNU Compiler Collection. libGOMP [9], the runtime library of the system is designed as a wrapper around the POSIX threads library, with some target-specific optimizations for systems that provide lighter weight implementation of certain primitives.

OMPi [10] is an open source OpenMP C compiler that conforms to the 2.5 version of the specification. OMPi is a source-to-source compiler which takes C source code with OpenMP pragmas and produces transformed multithreaded C code, ready to be compiled by the native compiler of the system. OMPi currently supports a number of thread libraries through a generic thread interface. They include a POSIX threads based library, which is highly tuned for single-level parallelism. Another library can be used in Sun / Solaris machines, where the user has the option of producing code with Solaris threads calls. OMPi has been recently enhanced with lightweight runtime support of nested parallelism based on user-level multithreading [11]. A large number of threads can be spawned for every `parallel` region and multiple levels of parallelism are supported efficiently, without introducing additional overheads to the OpenMP library. Because OMPi is a source-to-source translator, we have used both GCC and ICC as native back-end compilers, with similar performance; here we provide results only for ICC.

4.2 Results

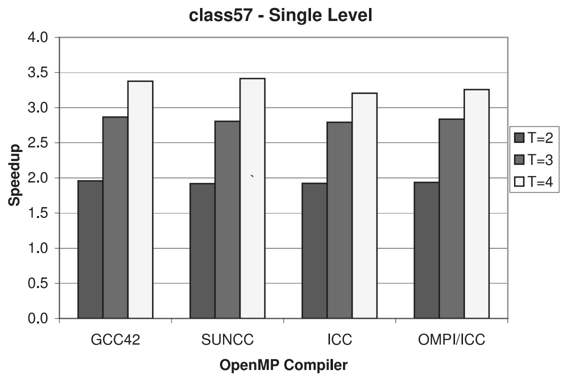
Figures 5 and 6 depict the speedups and the execution times of the OpenMP version of the face detection system for the two images, using the four OpenMP compilers. The timing is performed within our application code using the omp_wtime() function. We measure the time for applying all the steps of the face detection algorithm without including the time for loading the image.

The speedup is calculated as the ratio of the execution time of the sequential version of the OpenMP program to that of its parallel version using two or more threads. The sequential version of an OpenMP program is produced by simply instructing the OpenMP compiler to ignore the parallelization directives. In all experiments, the difference between the execution time of the sequential version and that of the parallel version running on a single OpenMP thread was negligible. Therefore, we present experimental results for 2, 3 and 4 OpenMP threads (denoted as T).



	GCC42	SUNCC	ICC	OMPi/ICC
SEQ	135	114	88	89
T=2	70	59	45	45
T=3	51	43	33	33
T=4	46	38	31	31

Figure 5: Overall Speedups and Execution Times (ms) for the f4 image.



	GCC42	SUNCC	ICC	OMPi/ICC
SEQ	2698	2284	1792	1795
T=2	1378	1190	932	927
T=3	941	814	642	633
T=4	799	669	559	551

Figure 6: Overall Speedups and Execution Times (ms) for the class57 image.

We observe that all configurations have similar performance and manage to improve the face detection responsiveness. The execution times for OMPi and ICC are very close because of their common native compiler and despite their different thread creation approaches and runtime libraries. In both OMPi and GCC, thread creation is based on the *outlining* technique [12]. According to this technique, the code inside a `parallel` region is moved to a separate function, which is executed by the spawned threads. In contrast, the Intel compiler uses an efficient threaded code generation technique, named Multi-Entry Threading [13]. In this case, the compiler generates a microtask to encapsulate the code lexically contained within a `parallel` region and the microtask is nested into the original function containing that `parallel` region. We also observe that GCC results in higher execution times. According to the above, this is attributed to the less efficient executable code that GCC generates in comparison to the commercial compilers, rather than to performance differences in the OpenMP runtime libraries.

Our parallel face detection system exploits the underlying multiprocessor hardware regardless of the OpenMP compiler used. It attains its fastest responsiveness when running with 4 OpenMP threads (T=4). The highest speedups are 3.01x and 3.41x for the f4 and class57 images, respectively.

Regarding the image processing time, we observe that the OpenMP parallelization is the key factor that provides real-time performance (i.e. $\geq$25 images/sec) to the face detection system. For the f4 image, the sequential version does not achieve a processing rate higher than 11 images/sec, even when the Intel compiler is used. For SUNCC this goal is achieved (26 images/sec) when 4 OpenMP threads are used. The highest performance is obtained with the Intel and OMPi compilers, which corresponds to 30 and 32 images/sec for 3 and 4 OpenMP threads respectively.

5 PARALLELIZATION STRATEGY II: NESTED PARALLELISM

In the previous section, we observed that the performance of the face detection system does not scale linearly as the processor count increases and this largely depends on the nature of the processed image. This is attributed to the inherent load imbalance, which is higher for the small (f4) image. For both f4 and class57, the second step of the algorithm includes a small number of iterations (scales), which is 11 and 13 respectively. Thus, the computational load cannot be distributed evenly to the 4 worker threads. Moreover, the fourth step does not generate enough parallelism for the f4 image because its 2 iterations (targets) are less that the number of workers. On the other hand, the class57 image produces 65 targets and the computational load is balanced more effectively, yielding better speedups.

The exploitation of nested parallelism can provide an effective solution to the above and thus ameliorate the performance of the face detection system because additional fine-grain parallelism is extracted from the application. Nested parallelism, a major feature of OpenMP, has the potential of benefiting a broad class of parallel applications to achieve optimal load balance and speedup, by allowing multiple levels of parallelism to be active simultaneously. This is particularly relevant these days in emerging SMP environments with multi-core processors. In this section, we discuss the exploitation of nested parallelism in our face detection system.

5.1 Exploiting Multiple Levels of Parallelism

The OpenMP specification leaves support for nested parallelism as optional. Implementations are considered compliant even if they do not support nested parallelism; they are allowed to execute the nested `parallel` region a thread encounters by a team of just 1 thread, i.e. nested `parallel` regions may be serialized. The number of OpenMP threads that execute a parallel region is determined through the omp_set_num_threads() function or the OMP_NUM_THREADS environment variable or set explicitly by the `num_threads` clause. If this number equals T and a second level of parallelism is spawned, then T^2 total OpenMP threads will be active.

Nowadays, several research and commercial OpenMP compilers support more than one level of parallelism. The majority of OpenMP implementations instantiate their OpenMP threads with kernel-level threads, utilizing either the POSIX-threads API or the native threads provided by the operating system. The support of nested parallelism is implemented by maintaining a pool of kernel threads that can be used as slave threads in parallel regions. The utilization of kernel threads introduces significant overheads in the runtime library, especially if multiple levels of parallelism are exploited. When the number of threads that compete for hardware recourses significantly exceeds the number of available processors, the system is oversubscribed and the parallelization overheads can outweigh any performance benefits.

The kernel-level threading model has been adopted by all OpenMP compilers used in our experiments but the OMPi compiler. In order to efficiently support unlimited nested parallelism, OMPi utilizes a runtime library based on user-level threads [11], named `psthreads`. The `psthreads` library implements a two-level thread model, where user-level threads are executed on top of kernel-level threads that act as *virtual processors*. Each virtual processor runs a dispatch loop, selecting the next-to-run user-level thread from a set of ready queues, where threads are submitted for execution. The value of the OMP_NUM_THREADS environment variable determines the number of virtual processors in the `psthreads` library. If this variable has not been set or its value exceeds the system's processor count, the number of virtual processors is set equal to the number of physical processors. Thus, OMPi maps the OpenMP threads to lightweight `psthreads` and the number of kernel-level threads never exceeds the number of physical processors.

Lightweight runtime support is crucial for our face detection system, considering the low processing time required for a single image. We focus on the exploitation of nested parallelism in the second step of the algorithm, which corresponds approximately to 80-85% of the total execution time (Table 1). As the number of iterations in this step is small, load imbalance is likely to occur even for small processor counts.

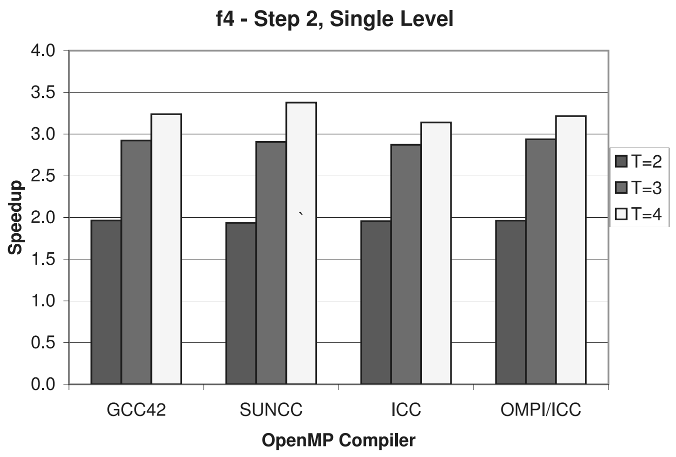
This step, however, includes additional loop level parallelism that can be expressed in a straightforward way using OpenMP directives. These inner loops, which belong to the second level of parallelism, are found in lines 4, 9 and 14 of Fig.3 and executed for each image scale. Similarly to the outer loop (line 2), they are parallelized using the OpenMP `parallel for` directive with the dynamic scheduling policy. The inner loops belong to the second level of parallelism, which is executed by new teams of OpenMP threads given that the OMP_NESTED environment variable is set, at runtime, to TRUE.

5.2 Performance Results

Figures 7 and 8 show the performance speedups obtained for the second step of the face detection algorithm, using the four OpenMP compilers and exploiting either a single level or two levels of parallelism.

Our intention is to present the improvements in the scalability of the second step, which represents more than 80% of the total execution time, due to our lightweight OpenMP runtime library.The overall performance speedup of the face detection procedure depends on the speedups of both the second and the fourth step. For the f4 image, the overall speedup (Fig. 5) is lower than the speedup of the second step (Fig. 7) because the fourth step does not scale well (according to Table 1, only two possible targets are found and examined in parallel). For the class57 image, both steps (2 and 4) have many parallel iterations and the performance scalability remains high. This is why Figures 6 and 8 are almost identical for this particular image.

We observe that the scalability of the second step is higher when nested parallelism is exploited using the OMPi compiler and 4 OpenMP threads. Nested parallelism is not always effective, depending on the



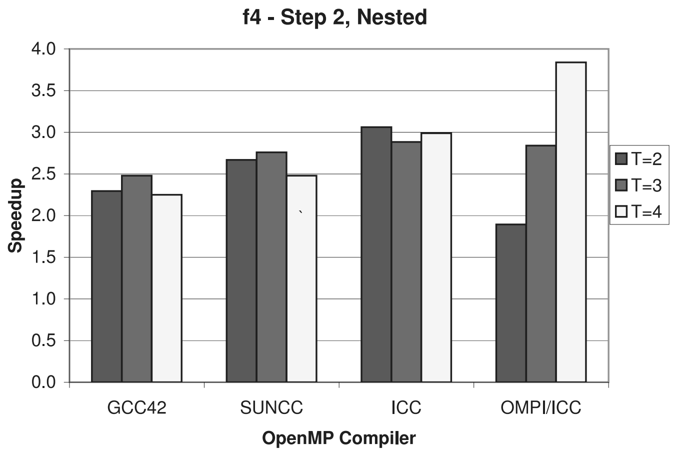
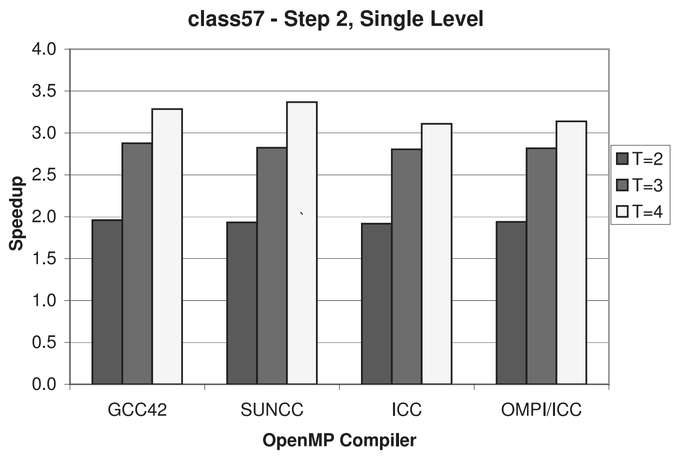


Figure 7: Speedups of the second step for the f4 image, exploiting a single level and two levels of parallelism. T is the value of the OMP_NUM_THREADS environment variable.



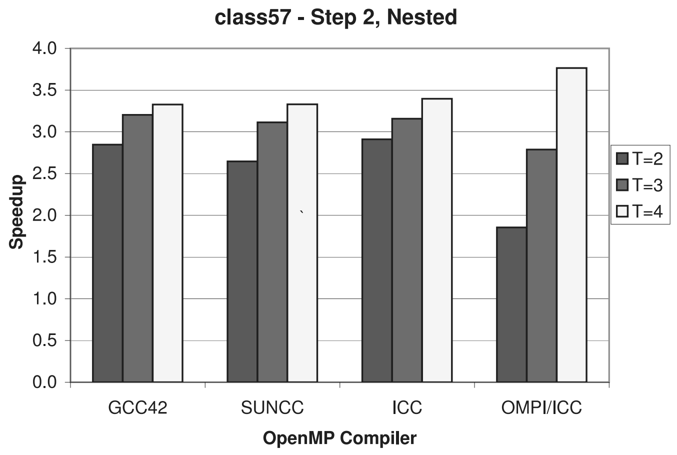


Figure 8: Speedups of the second step for the class57 image, exploiting a single level and two levels of parallelism. T is the value of the OMP_NUM_THREADS environment variable.

application and the runtime system used. For example, for the f4 image, in all implementations (except OMPi), the best single-level performance (T=4) is clearly superior to their best two-level performance (which occurs for T=2 or T=3). In this case, the OMPi compiler attains the maximum speedups, which are 3.84x for the f4 image and 3.77x for the class57 image. The speedups obtained with OMPi for the second step are significantly improved, by up to 20%, compared to the corresponding 3.22x and 3.14x speedups for the single-level parallelism case. The shared bus and memory bandwidth of the quad-core system prevents linear application scalability. It also results in slightly lower speedup for the large image (class57), for which more memory transactions occur and cache memory utilization is not as effective as for the small image (f4).

When two OpenMP threads are used (T=2) for each parallel region, the three OpenMP compilers exhibit speedups that are better than that of OMPi and higher than the corresponding number of threads. This is reasonable because these configurations utilize 4 (2×2) total kernel threads, which run on all system processors. On the other hand, OMPi utilizes only two kernel threads. Using 3 OpenMP threads (T=3), we observe that all the compilers exhibit similar performance, with slightly higher speedup for the Intel compiler. OMPi, however, uses one less processor, which means that the overheads of the other OpenMP compilers are significantly higher. As the number of OpenMP threads increases, more fine grain parallelism is exploited. Thus, the contention of the OpenMP kernel-level threads, which become more than the physical processors, is also increased.

Since the face detection algorithm spends most of its time in the second step, the successful exploitation of nested parallelism in this step also improves the overall application performance. If OMPi is used, the processing time required for the f4 image is further reduced and thus the processing rate increases approximately from 32 to 38 images/sec ($\simeq$19%). The overall improvement is slightly lower ($\simeq$16%) for class57, because the second step for this image corresponds to a smaller portion (80%) of the overall processing time compared to the portion for f4 (85%).

6 HIGH PERFORMANCE BATCH PROCESSING

Apart from the improved response time for the single-image case, OpenMP can also provide high throughput batch processing of large image databases. In this case, a `parallel for` directive is applied to the loop that processes a workload of images, as depicted in Fig. 9. Each OpenMP thread executes dynamically iterations of this loop, applying each time the sequential face detection algorithm to a single image. Alternatively, the program can process the images in sequential order and exploit multiple processors by applying the parallel version of the algorithm. The two approaches correspond to the exploitation of outer and inner parallelism, respectively. Although nested parallelism allows the combination of these approaches, the low processing time for a single image eliminates load imbalance and thus the need for such an approach.

Table 3 depicts the execution times and the corresponding speedups for the batch processing of a workload of 161 images on the quad-core system. For this experiment, we provide measurements only for the OMPi compiler. We observe that if the images are processed in sequential order and `inner` (per image) parallelism is exploited, the speedups of the workload are 1.92x and 3.20x for 2 and 4 OpenMP threads respectively. On the other hand, when OpenMP is applied only at the `outer` level of parallelism, the corresponding speedups are 1.97x and 3.86x. If images are processed in parallel, the obtained speedup is considerably higher. This is attributed to the execution of the face detection algorithm for an image on a single processor, which results in:

- Minimal runtime overheads because the processing of a single image does not spawn additional parallelism.

```
for each entry in image database
{
    load image
    apply face detection algorithm
        make image pyramid
        process each image scale S
        apply grouping to targets
        fine scan each target
    report found faces
}
```

Figure 9: Batch processing of image database.

Table 3: Batch processing time (sec) and speedup for a workload of multiple images.

	Time	Speedup
SEQ	31.72	1.00
T=2 (inner)	16.51	1.92
T=2 (outer)	16.12	1.97
T=2 (nested)	16.27	1.95
T=4 (inner)	9.92	3.20
T=4 (outer)	8.21	3.86
T=4 (nested)	8.27	3.84

Table 4: Batch processing times (sec) and overheads when nested parallelism is enabled.

	Outer	Nested	Overhead
GCC42	13.05	16.22	24.24%
SUNCC	11.30	13.74	21.57%
ICC	8.10	11.40	40.76%
OMPi/ICC	8.21	8.27	0.73%

- Efficient load balancing since multiple images are distributed to the processors in a better way.
- Good data locality as the memory required for processing an image is accessed by a single processor.

When two levels of parallelism are exploited, the speedups remain the same because OMPi introduces minimal overheads and favors the execution of the inner level parallelism on the same processor. In contrast, the performance of the other OpenMP compilers drops drastically when 4 OpenMP threads are used for each parallel region. The experimental results, depicted in Table 4, show that the execution time for processing the image database can be up to 40% higher compared to the case that exploits only the outer task-level parallelism.

7 SUMMARY

In this paper, we have presented a parallel face detection system for shared memory multiprocessors using the OpenMP programming model. OpenMP provides a fast and portable way of parallelizing the sequential code and allows for efficient exploitation of the computational resources.

The experimental results demonstrate significant improvement in the processing time of a single image. Our efficient runtime support, based on the OMPi OpenMP compiler with lightweight multithreading, provides efficient exploitation of the nested loop-level parallelism and better load balancing in the face detection algorithm. The experiments on a quad-core system show that the speedup of the second step of the algorithm is improved by up to 20%, when multiple levels of parallelism are exploited. OpenMP also increases the throughput of the face detection system through the concurrent processing of multiple images.

Our future work includes further study and exploitation of nested parallelism in the face detection system, performance optimizations in the OpenMP runtime library and processing of large image databases on computation clusters.

References

[1] Viola P, Jones M. Robust real-time face detection. *International Journal of Computer Vision* 2004; **57**(2):137–154.

[2] Garcia C, Delakis M. Convolutional face finder: A neural architecture for fast and robust face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 2004; **26**(11):1408–1423.

[3] Rowley H, Baluja S, Kanade T. Neural network-based face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1998; **20**(1):23–38.

[4] OpenMP Architecture Review Board. *OpenMP specifications*. http://openmp.org/wp/openmp-specifications/ [June 2008].

[5] LeCun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 1998; **86**(11):2278–2324.

[6] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing* 1994; **6**(3-4):159–416.

[7] Tian X, Hoeflinger JP, Haab G, Chen YK, Girkar M, Shah S. A compiler for exploiting nested parallelism in OpenMP programs. *Parallel Computing* 2005; **31**:960–983.

[8] Sun Microsystems. *Sun Studio 12: OpenMP API User's Guide* 2007; P.N. 819-5270.

[9] Novillo D. OpenMP and automatic parallelization in GCC. In *Proc. of the 2006 GCC Summit,* Ottawa, Canada, June 2006.

[10] Dimakopoulos VV, Leontiadis E, Tzoumas G. A Portable C Compiler for OpenMP V.2.0. In *Proc. of the 5th European Workshop on OpenMP (EWOMP '03),* Aachen, Germany, October 2003.

[11] Hadjidoukas PE, Dimakopoulos VV. Nested parallelism in the OMPi OpenMP C compiler. In *Proc. of the European Conference on Parallel Computing (EUROPAR '07),* Rennes, France, August 2007.

[12] Chow JH, Lyon LE, Sarkar V. Automatic parallelization for symmetric shared-memory multiprocessors. In *Proc. of the 1996 conference of the Centre for Advanced Studies on Collaborative Research (CASCON '96),* Torontom, Canada, November 1996.

[13] Tian X, Bik A, Girkar M, Gray P, Saito H, Su E. Intel openmp c++/fortran compiler for hyper-threading technology: Implementation and performance. *Intel Technology Journal* Q1 2002; **6**.