In Proceedings of the Workshop on Parallel and Distributed Multimedia, Columbus, Ohio, August 2006.

Model-Based OpenMP Implementation of a 3D Facial Pose Tracking System

Sankalita Saha[1,3], Chung-Ching Shen[1,3], Chia-Jui Hsu[1,3], Gaurav Aggarwal[1,3], Ashok Veeraraghavan[1,3], Alan Sussman[2,3] and Shuvra S. Bhattacharyya[1,3]

(1) ECE Dept. / (2) CS Dept. / (3) Institute for Advanced Computer Studies, University of Maryland, College Park MD, USA

***Abstract*— Most image processing applications are characterized by computation-intensive operations, and high memory and performance requirements. Parallelized implementation on shared-memory systems offer an attractive solution to this class of applications. However, we cannot thoroughly exploit the advantages of such architectures without proper modeling and analysis of the application. In this paper we describe our implementation of a 3D facial pose tracking system using the OpenMP platform. Our implementation is based on a design methodology that uses coarse-grain dataflow graphs to model and schedule the application. We present our modeling approach, details of the implementation that we derived based on this modeling approach, and associated performance results. The parallelized implementation achieves significant speedup, and meets or exceeds the target frame rate under various configurations.**

I. Introduction

A large class of image processing applications is characterized by computation- and memory- intensive operations. Most of these applications have inherent parallelism in them, both data as well as instruction-level parallelism. Such applications yield significant performance gains when this parallelism is properly exploited. Proper exploitation of this parallelism is not always easy, however, since there are significant memory operations involved — much of the data parallelism involves operations on image frames that are of considerable size — that can overshadow the performance gain obtained by parallelization.

For instance, in many serial implementations there is only one reading of the image frame together with sequential operations that process the frame. However, in a parallel implementation, each processor usually requires its own local copy of the image data, which adds significantly to the memory overhead. Clearly, there is a trade-off involved here. As the number of processors is increased, more parallelism in the operations can be exploited, but at the same time, more memory is required to store the local sets of image data for the processors. Thus, for optimized implementation of such applications, we need to understand their high-level structure in relation to the target architecture, and we need to balance the trade-offs suitably across the available processors.

In this paper we present our work on the implementation of such an application — a particle-filter based 3D facial pose tracking system. A parallelized multiprocessor implementation of this system requires each processor to store a local copy of the image. These local copies are needed for reading purposes, but not necessarily for writing purposes. Since tracking belongs to the class of applications that need to meet real-time constraints, performance is an important criterion. At the same time, memory optimization is important due to the large numbers of pixels that must be processed. For such applications, shared-memory parallel systems provide a promising solution space. For our implementation, we use the openMP model, which is gradually becoming a popular standard for shared memory systems.

To more effectively exploit the parallelism of signal processing applications, such as applications in the image processing domain, it is beneficial to model the high-level application structure using coarse-grain dataflow graphs (e.g., see [4]).

In this paper, we employ one of the most popular forms of coarse-grain dataflow, called *synchronous dataflow*, for signal processing applications. In the context of embedded multiprocessor implementation, synchronous dataflow has primarily been applied to one-dimensional signal processing applications, most notably, in the domain of digital communications. In this, paper we apply synchronous dataflow as a formal modeling tools for the multi-dimensional signal processing domain of image processing. Specifically, we demonstrate how synchronous dataflow based design can be used to expose the inherent parallelism present in a 3D tracking system, and how this parallelism can be exploited during system implementation to achieve an effective trade-off between performance and resource requirements.

The main contribution of this paper is therefore in our integration of coarse-grain dataflow graph modeling, dataflow-based memory/performance trade-off analysis, and OpenMP parallel implementation for a challenging image processing application. In our experiments, the modeling and scheduling of the dataflow graph has been performed using the dataflow interchange format (DIF), which is an evolving language and associated design tool framework for dataflow-based modeling, analysis, porting, synthesis, and optimization [11].

The rest of the paper is organized as follows. In Section II we present an overview of related work. In Sections III and IV we present the details of the targeted 3D tracking application, and a discussion of dataflow modeling for signal processing applications. Section V discusses our study in dataflow modeling, scheduling, and synthesis of the tracking application using DIF. Section VI and VII present OpenMP implementation details and results, followed by the conclusions, which are presented in section VIII.

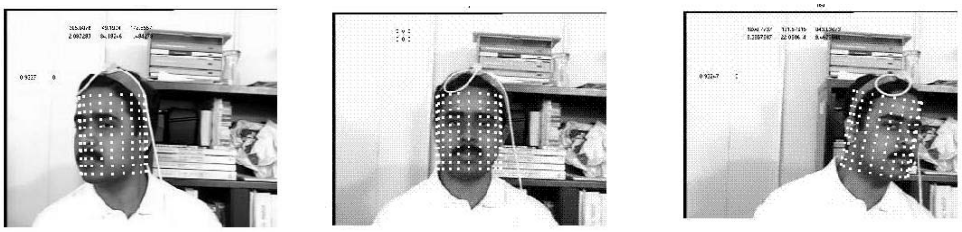


Fig. 1. Image frames with tracking cylindrical mesh.

II. Related Work

Exploring parallelized implementations for performance improvement of computationally-intensive operations has been an active research field for many years. OpenMP [7] is gradually emerging as a standard for parallel programming on shared-memory architectures. It is the first successful effort to standardize shared memory programming directives and is gaining more popularity compared to the message passing model, because of its ease of use. However, for distributed systems, MPI remains a more attractive choice. A combination of MPI and OpenMP at different grains of parallelism is a useful option for many applications. A comparative study of the application of these two paradigms, either jointly or singly, to a multitude of applications is presented in [5].

A variety of image processing applications have been implemented on parallel architectures. In [17], an edge-detection system has been implemented in a PVM network. In [13], parallel implementations of numerical and image processing applications on a multiprocessor systems have been presented. In [16] the implementation of the H.264 encoder using Intel's hyper-threading architecture has been presented, while in [8] a 3D-FWT video encoder has been implemented using both OpenMP as well as Pthreads.

While these efforts have resulted in impressive performance enhancements, they generally lack a systematic approach towards exposing and exploiting the various levels of parallelism inherent in image processing applications. There has also been a significant amount of work done in automatic parallelization of serial code [9], [3] but these efforts mainly rely on program analysis and message passing.

In this paper, we present the parallelized implementation of a particle-filter based 3D facial pose tracking system for video using OpenMP. Our methods for analysis and parallelization throughout the design and implementation process are based on dataflow modeling and scheduling techniques that we apply at the application level rather than program level.

III. 3D Facial Pose Tracking

The aim in a 3D facial pose tracking system is to recover the 3D configuration of a face in each frame of a video. The tracking algorithm explored in the work of [1] combines the structural advantages of geometric modeling with the statistical gains of particle-filter based inference and differs from other related work on face tracking that uses 2D appearance based models.

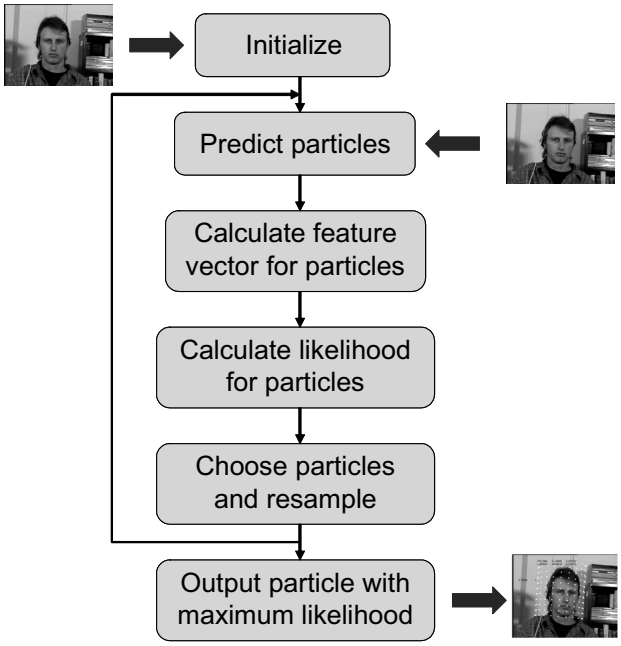


Fig. 2. 3D Facial Pose tracking algorithm.

In this tracking algorithm, there are three main aspects that capture the whole system. The first is the model to represent the facial structure; the second is the feature vector used; and the third is the tracking framework used. Each of these aspects is elaborated on in the following sections.

A. Model to represent the facial structure

A model attempts to approximate the shape of the object to be tracked in the video. One would like the model to capture the shape as precisely as possible, but at the same time, the model should not be very complicated, since that makes tracking more computation-intensive. For 3D tracking we need a 3D model that approximates the shape of the object. A cylinder with an elliptical cross-section is a suitable 3D model that can be used to represent the 3D structure of faces.

B. Feature vector

For practical implementation, characteristics from the image have to be used to capture the model in the image. A rectangular grid superimposed around the curved surface of the elliptical cylinder is used as the feature vector. The mean intensity for each of the visible grids forms the feature vector. Figure 1 illustrates the model along with the feature vector.

C. Tracking Framework

For the tracking framework, i.e., estimating the configuration or pose of the moving face in each frame of a given video, particle filter based inference is used. The motion of the face is characterized by 3 translation and 3 orientation (yaw, pitch and roll) parameters. For each new image frame read in from the camera, multiple predictions for these parameters are made. Each prediction is denoted as a particle. The number of particles for a system is fixed and usually decided by the user during initialization. The feature vector is extracted for each particle and then the particle that yields the best likelihood value is said to give the position of the face in the frame. The likelihood value is an estimate of the probability of the cylinder position and depends on the distance of the cylinder in the current frame from its position in the previous frame.

Figure 2 shows the algorithmic flow for the system.

IV. Dataflow Modeling

In the area of digital signal processing (DSP), dataflow is widely recognized as a natural model for specifying applications. In dataflow, a program is represented as a directed graph, called a *dataflow graph*, in which vertices, called *actors*, represent computations and edges represent FIFO channels, (also called *buffers*). These channels queue data values, in the form of tokens, which are passed from the output of one actor to the input of another. When an actor is executed (*fired*), it consumes a certain number of tokens from its inputs, and produces a certain number of tokens at its outputs.

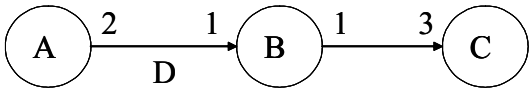
The synchronous dataflow (SDF) model [12], proposed by Lee and Messerschmitt, is a restricted version of dataflow. In SDF the number of tokens produced (or consumed) by an actor firing, on each output (or input) is a fixed number that is known at compile time. This provides the major advantage that SDF has over other dataflow models: whether a valid static schedule for a given SDF graph G exists can be determined at compile time, and furthermore, when such a schedule exists, it constructed at compile time. Here, by a valid schedule we mean a schedule that can be repeated indefinitely with bounded memory requirements, and without bringing the application into a deadlocked state. The ability to iterate the graph indefinitely is critical in many signal processing applications, where the length of the input data stream (e.g., the number of voice samples or image frames) to be processed is often not known or bounded in advance.

The minimum number of times an actor must be fired in a valid schedule is represented by a vector q_G, called the *repetitions vector*, that is indexed by the actors in G. These minimum numbers of firings can be derived by finding the minimum positive integer solution to the *balance equations* for G, which specify that q_G must satisfy

$$q_G(src(e)) \times p(e) = q_G(snk(e)) \times c(e), \text{ for edge } e, \quad (1)$$

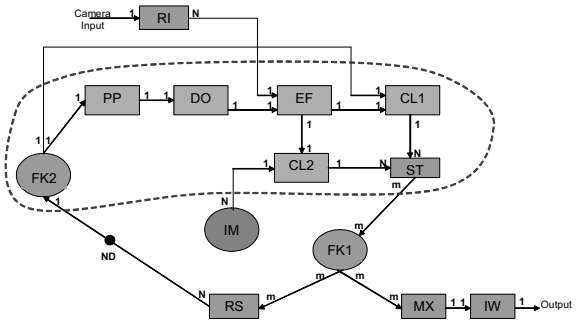
where $p(e)$ and $c(e)$ denote the constant production and consumption rates on e.

A schedule L for G is a *minimal* periodic schedule if it invokes each actor A in G exactly $q_G(A)$ times. Figure 3 shows a simple SDF graph with a minimum periodic schedule.



Schedule S_1: (3A)(6B)(2C)

Fig. 3. A simple SDF graph and its schedule.

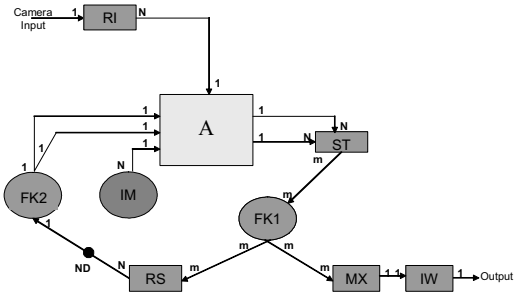


RI:Read Image	PP:Predict Particles	DO:Derive Observation
EF:Extract Features	CL:Compute Likelihood	ST:Sort
FK:Fork	IM:Initial Model	RS:Resample
MX:Max	IW:Image Write	

Fig. 4. Synchronous dataflow graph for 3D facial pose tracking system.

Figure 4 shows an SDF representation for the 3D facial pose tracking system. The "*ND*" on the edge from actor *RS* to *FK2* represents N units of *delay* that are present on this edge. Each unit of delay can be implemented by placing an initial token on the associated buffer. In this case, the N delay tokens on the edge are required to avoid deadlock in the graph.

Conceptually, the *fork* actors (*FK1* and *FK2*) replicate each token on their input data streams onto their output streams. In practice, fork actors do not require any copying of data, and can be implemented through manipulation of read and write pointers into a common buffer.



RI:Read Image	A: Hierarchical Actor	ST:Sort
FK:Fork	IM:Initial Model	RS:Resample
MX:Max	IW:Image Write	

Fig. 5. Hierarchical SDF graph for 3D facial pose tracking system.

The portion of the graph marked with dotted lines shows a group of operations that occur N times for a given image frame, where N is the total number of particles of the system. These N executions are independent of each other since there are no data dependencies among them. This is represented clearly in the SDF graph: if we consider the marked portion of the graph as one single (hierarchical) actor then we obtain the hierarchical SDF graph in Figure 5. Applying the balance equations (1), we obtain the repetition count of A to be N while the rest of the actors have a repetition counts of 1. Due to N delays on the edge between RS and FK2, it is possible to execute the N firings of A independently (provided that the initial tokens corresponding to the delays are first used to execute N invocations of FK2). Therefore, the N firings of A may be parallelized over multiple processors. All of these actor firings would read the same image frame as shown in the graph but would not write to it. After the N iterations, there is a final write to the image that marks the tracked face on the image. This provides the main scope for parallelization across actor firings.

V. Design And Implementation Using DIF

The SDF-based specification and scheduling for our targeted application has been done using the DIF framework. In the next three sections, we present a brief overview of DIF followed by a discussion of our modeling and scheduling efforts using DIF. The final code was derived by hand based on a schedule that was generated automatically by DIF.

A. DIF Overview

The dataflow interchange format (DIF) project [10,11], an ongoing research project at the University of Maryland, is a framework for developing dataflow-based application models, analysis algorithms, and design tools for signal processing applications.

DIF involves two main parts: the *DIF language* [11] and the *DIF package*. The DIF language is a language for specifying and working with mixed-grain dataflow models for DSP systems. It provides a unique set of semantic features for specifying graph topologies, hierarchies, dataflow-related properties, and actor-specific information. The DIF package is an associated Java-based software package that provides object-oriented intermediate representations, algorithm implementations, and infrastructure for scheduling, optimization, porting, and software synthesis.

Figure 6 illustrates the methodology of using DIF to interface various dataflow models, system designs, software libraries, dataflow-based design tools, and their supported embedded processing platforms. The shaded areas in Figure 6 show the facilities in the DIF package that we have used for implementing the 3D facial pose tracking system.

B. DIF Specification

DIF, in contrast to other dataflow-based design environments, such as ADS from Agilent [14], the Autocoding Toolset from MCCI [15], and the signal processing oriented subsystem of LabVIEW by National Instruments, is developed as a text-based, rather than graphics-based, programming and specification format for DSP-oriented dataflow graphs. The text-based format is useful for developing and managing large-scale designs, while generators can be used to construct graphics-based representations from textual DIF specifications.

After sketching the 3D facial pose tracking system in the SDF graph form as shown in Figure 4, we specify the complete SDF modeling semantics using the DIF language. Figure 7 presents the corresponding DIF specification of Figure 4, where $N = 100$ and $m = 5$. For a full description of the DIF language syntax, we refer the reader to [11].

C. Scheduling in the DIF Package

The DIF package provides implementations of various SDF scheduling algorithms. These algorithm construct valid schedules under various combinations of implementation objectives and constraints. By compiling the DIF language specification (Figure 7) of the 3D facial pose tracking system through the DIF front-end tool (as shown in Figure 6), we can obtain a Java-based intermediate representation of the SDF graph. To this intermediate represent, we apply the LIAF [4] and APGAN [4] scheduling algorithms on the SDF graph, which are geared towards minimizing memory requirements. From these schedules, we obtain the following "looped" schedule of the 3D facial pose tracking system: $IM\ RI\ (100\ FK2\ PP\ DO\ EF\ CL1\ CL2)\ ST\ FK1\ RS\ MX\ IW$. Here, each parenthesized term (*schedule loop*), which is of the general form $(nS_1S_2\ldots S_m)$, represents n successive execution of the subschedule $S_1S_2\ldots S_m$, where each S_i represents either an actor or a (nested) schedule loop.

According to the schedule, we first execute IM (initial model) and RI (read image), then we execute PP (predict particles), DO (derive observation), EF (extract features), CL1 (compute likelihood), and CL2 as a loop that is iterated 100 times, and finally we execute ST (sort), MX (max), and IW

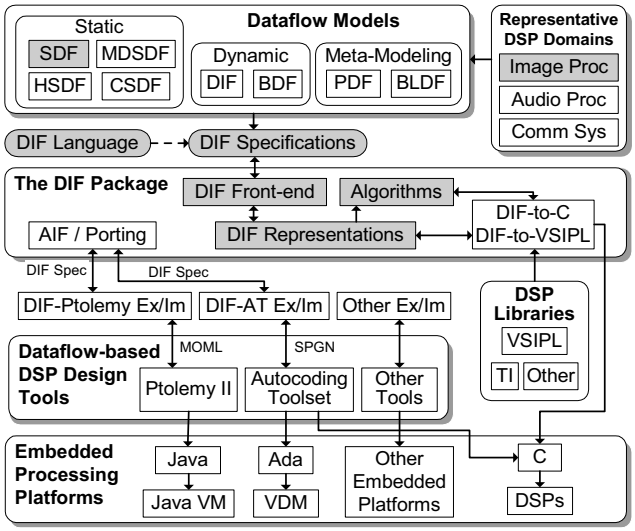


Fig. 6. DIF-based design methodology.

```
sdf graph FaceTracking {
  topology {
    nodes = RI, EF, CL1, CL2, IM, ST, FK1, RS, FK2,
PP, DO, MX, IW;
    edges = e1(RI,EF), e2(EF,CL1), e3(EF,CL2),
e4(IM,CL2), e5(CL1,ST), e6(CL2,ST), e7(ST,FK1),
e8(FK1,RS), e9(RS,FK2), e10(FK2,PP), e11(PP,DO),
e12(DO,EF), e13(FK2,CL1), e14(FK1,MX), e15(MX,IW);
  }
  production {
    e1=100; e2=1; e3=1; e4=100; e5=1; e6=1; e7=5;
e8=5; e9=100; e10=1; e11=1; e12=1; e13=1; e14=5;
e15=1;
  }
  consumption {
    e1=1; e2=1; e3=1; e4=1; e5=100; e6=100; e7=5;
e8=5; e9=1; e10=1; e11=1; e12=1; e13=1; e14=5; e15=1;
  }
  delay {
    e1=0; e2=0; e3=0; e4=0; e5=0; e6=0; e7=0; e8=0;
e9=100; e10=0; e11=0; e12=0; e13=0; e14=0; e15=0;
  }
}
```

Fig. 7. DIF Specification of the 3D Facial Pose Tracking System.

(image write). Note that the fork actors (FK1 and FK2) are used to represent data branches in the SDF graph and are not implemented in our actual code. By referring back to the SDF graph model of the 3D facial pose tracking, we can easily explore the parallel configurations of this schedule — in particular, the sub-schedule (100 FK2 PP DO EF CL1 CL2), together with examination of the associated SDF subgraph, reveals that the iterations of the loop (100 FK2 PP DO EF CL1 CL2) can be executed in parallel (based on the number of available processors).

VI. Implementation In OpenMP

Our original implementation of the system was developed in MATLAB. Prototyping the initial design in MATLAB is useful to work out the basic functional correctness and calibrate algorithm parameters. From the initial MATLAB specification, we derived the parallel implementation in two steps. In the first step, we derived a C-based serial version, and in the second step, we parallelized our serial C version using OpenMP together with the SDF-based analysis described in Sections IV and V.

The resulting program structure follows the common structure of many parallel programs — that is, a serial portion involving initialization followed by parallel code, followed again by serial code that gathers data from parallel regions. In the case of our design, this whole program structure repeats itself for each image frame. The overall program structure is illustrated in Figure 8. Here, the serial code was obtained by first generating the modules for the actors in the SDF graph as shown in Figure 4 and then ordering them according to the schedule generated using the DIF package. The code was parameterized in terms of the number of particles. As described in section V, a high-level looped schedule was generated through DIF for the program, and the repetition count of the core loop in this schedule was equal to the number of particles. Thus varying the code for different numbers of particles essentially involved changing only a single loop iteration count.

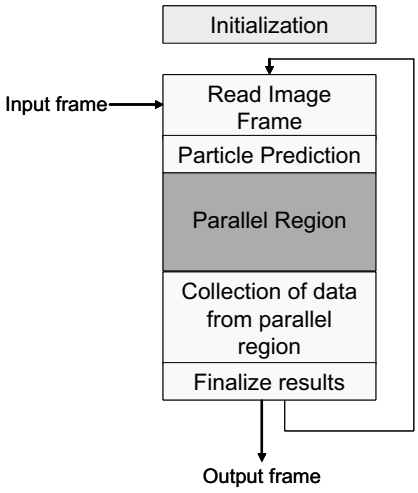


Fig. 8. OpenMP code structure.

The most important function in the implementation in terms of tracking correctness was the prediction of particles. The role of this function is to estimate the rotation and translation vectors for the face for the next frame relative to the current frame. The prediction is done based on a Gaussian distribution, which requires a random number generator that generates normally distributed random numbers. Without this prediction function, the tracking accuracy of the algorithm deteriorates gradually over frames. The random number generator provided by ANSI C generates uniformly distributed random numbers. We performed a conversion from uniform to Gaussian distribution using inverse mapping. The overall system turned out to be extremely sensitive to the random number generator function, and the following mapping was used in our final implementation:

$$z = (\sqrt{-(2 \cdot \log(x))}) \cdot \cos(2 \cdot \pi \cdot y), \qquad (2)$$

where x and y are random numbers with uniform distribution using thread-safe random number generators.

The tracking correctness was sensitive to the number of particles used as well. This is expected, as more particles yields a wider range of values from which prediction can be made, which in turn ensures higher prediction accuracy. For our implementation, we used particle populations ranging from 100 to 1000.

Another important function in the implementation — which is an important step for all practical implementations of particle filters — is *resampling*. Without resampling, a particle filter is highly likely to degenerate, and reduce the accuracy of the overall system. Resampling is the act of redrawing particles from the same density such that the weights of the particles are approximately equal. Several methods for resampling exist. In the algorithm used in this work, *systematic resampling* was used. In systematic resampling, the new samples are exact replicas of some of the old samples, but they occur with multiplic-

Fig. 9. Sample tracking results (still) for the first benchmark video. The superimposed cylinder moves and tracks the face in each frame.

ities that are proportional to their previous weights. In our case, the previous weight of a particle is its likelihood value. The new weights are all reset to $1/N$.

For scheduling the threads, dynamic scheduling was used. Since the load to a very large extent was uniformly distributed, separate load-balancing techniques were not used. The image was shared amongst all the threads. Apart from the image, all the major data structures were shared, which provided a useful degree of memory optimization, and also demonstrated the suitability of the shared memory model. For example, the initialization data and the prediction vector generated at the beginning of the parallel region for all the particles were shared. Each particle modifies the likelihood vector and this vector was also shared. Since no two threads modify the same element of this vector at any given time, it was ensured that there would be no writing violations.

It was observed that the execution times did not improve by merely increasing the number of threads. In fact, there was generally a point beyond which we found that increasing the number of threads resulted in the execution time to increase. This may be explained using Amdahl's Law [2]. For parallel computing, Amdahl's law states that if F is the fraction of a calculation that is sequential (i.e. cannot benefit from parallelization), and $(1-F)$ is the fraction that can be parallelized, then the maximum speedup that can be achieved by using N processors is

$$\frac{1}{\left(F+\frac{(1-F)}{N}\right)}. \tag{3}$$

When $N \to \infty$, the maximum speedup is given by $1/F$. Thus, theoretically the maximum speedup that can be achieved in a parallelized code is inversely proportional to the execution time of the sequential part in the code. But in practice, the speedup is affected by other factors such as scheduling, communication and synchronization. In our case, scheduling the threads and synchronizing them at the end of their executions were important factors that decided the maximum speedup. These overheads are dependent on the number of threads and can be expressed as a function of N as

$$S = f(N). \tag{4}$$

Thus, the maximum speedup can now be expressed as

$$\frac{1}{\left(F+\frac{(1-F)}{N}+f(N)\right)}. \tag{5}$$

Since, we used a run-time scheduler to schedule the threads, defining the function $f(N)$ precisely is not possible. However, approximations can be derived by appropriate analysis, and this is an interesting direction to explore for future work.

VII. Results

In this section, we present results that we obtained on two benchmark videos. The data for both the videos were taken from the BU data set [6]. The system was implemented on a Sunfire 6800 containing 24 SUN UltraSparcIII machines running at 750 MHz using 72GB of RAM. Two video sequences from the data set, each 200 frames long and with an image frame of size 36KB, were used to verify the correctness of the implementation. The tracking of the facial pose was accurate for both benchmark videos and sample tracking results from video for the first benchmark are shown in Figure 9. The number of particles for this figure is 1000. Also, for this case, Figure 10 shows comparison of the ground truth values and the estimated orientation values as predicted by the tracker. It may be observed that the values match closely, demonstrating the

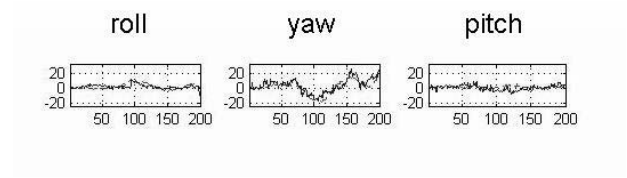


Fig. 10. Comparison of the orientation parameters. The dashed curve depicts the ground truth. The solid curve depicts the estimated values.

correctness of the tracking operation.

The performance results shown in Figures 11 and 12 are obtained for numbers of particles varying from 100 to 1000 and numbers of threads varying from 1 to 16. When only one thread is used, essentially the serial implementation is realized and hence the serial performance is given by the execution times using one thread. As may be observed from the results, the implementation using threads in OpenMP shows very large performance improvements compared to the serial version (one thread). The best performance is obtained for 8 threads and not the maximum number of threads, which was 16. Beyond 8 threads, the amount of time spent in scheduling and coordinating the threads starts overshadowing the gains obtained from parallelization, and hence the execution time starts increasing.

The MATLAB code, when run for 100 particles using MATLAB version 7.0, requires an execution time of 650.421 seconds for the first benchmark and an execution time of 885.954 seconds for the second benchmark, which indicates a level of performance that is off from the target frame rate of 30 frames per second (fps) by a large margin. For the first benchmark, the best frame rate achieved is 33 fps using 8 threads for 100 particles which over-achieves the target rate, while the worst is 3 fps using 2 threads for 1000 particles. For the second benchmark, the best frame rate achieved is 34 fps using 8 threads for 100 particles, which once again over-achieves the target frame rate, while the worst is 3 fps using 2 threads for 1000 particles.

Although our parallelized implementation does not always meet the target frame rate, in general for all variations in numbers of particles, the implementation yields significant performance improvement compared to the serial-C and MATLAB versions. Also, for many relevant tracking applications, frame rates up to 10 fps can be tolerated, and in most of the cases that we experimented with, this requirement was met.

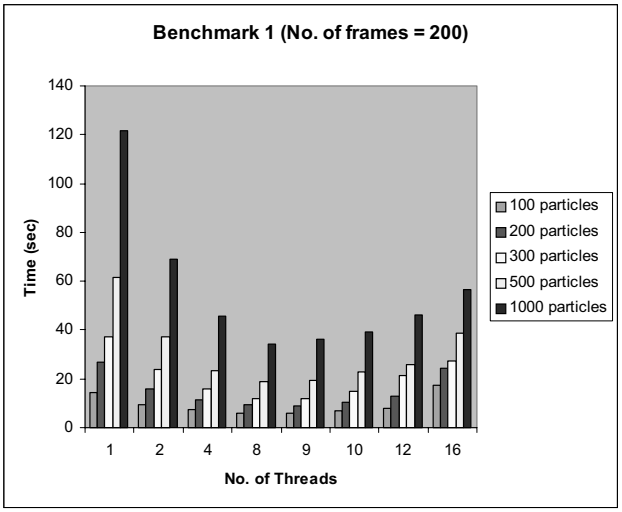


Fig. 11. Performance results for first benchmark

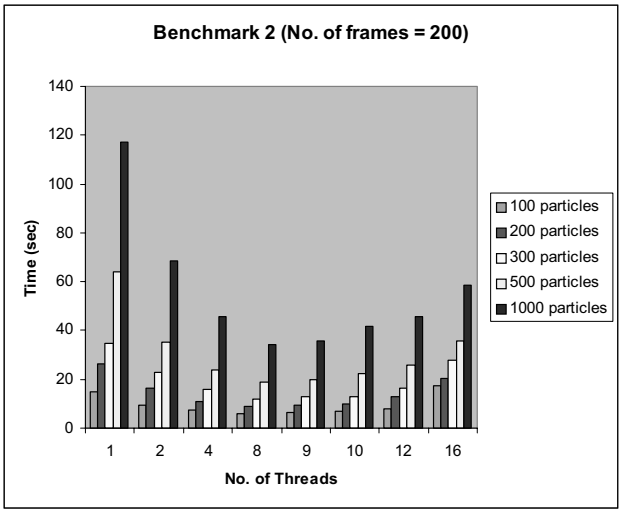


Fig. 12. Performance results for second benchmark

VIII. Conclusions

Shared memory architectures offer promising solutions for computation- and memory-intensive image processing applications, and OpenMP provides a convenient platform to exploit these architectures. However, to benefit from these advantages, one needs to carefully exploit the parallelism that is inherent in the application.

In this paper, we have presented our design and implementation of a 3D facial pose tracking algorithm on a shared memory architecture. In our development, we have shown how dataflow modeling techniques can be used to expose and exploit parallelism in a simple yet powerful manner. Through the high level structure of a well-designed dataflow representation, one can efficiently explore trade-offs among algorithm accuracy parameters, resource requirements, and performance to tailor the characteristics of an implementation for the given set of design objectives and constraints.

The tracking algorithm used in this work focuses on a single-camera system. In our future work, we will explore multiple-camera systems. One approach that we are investigating in this line of work is integration with the MPI paradigm to enable communication across the different processing platforms for individual cameras.

IX. Acknowledgements

This research was supported in part by grant number 0325119 from the U.S. National Science Foundation.

REFERENCES

[1] G. Aggarwal, A. Veeraraghavan, and R. Chellappa. 3D Facial Pose Tracking in Uncalibrated Videos. In *International Conference on Pattern Recognition and Machine Intelligence* (PReMI), 2005.

[2] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *Proc. of AFIPS Conference*, (30), pp. 483-485, 1967.

[3] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211-243, 1993.

[4] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers. 1996.

[5] S. Bova, C. Breshears, H. Gabb, R. Eigenmann, G. Gaertner, B. Kuhn, B. Magro, S. Salvini and H. Scott. Parallel Programming with Message Passing and Directives. *Computing in Science & Engineering*, 3(5), 2001, pp. 22-37.

[6] M. La Cascia, S. Sclaroff and V. Athitsos. Fast, Reliable Head Tracking under Varying Illumination: An Approach Based on Robust Registration of Texture-Mapped 3D Models. In *IEEE Transactions on Pattern Analysis and Machine Intelligence* (PAMI), 22(4), April, 2000.

[7] L. Dagum and R. Menon. OpenMP: An Industry-StandardAPI for Shared-Memory Programming.In *IEEE Computational Science and Engineering,* 5(1), 1998, pp. 46-55.

[8] R. Fernandez, J. M. Garcia, G. Bernabe and M. E. Acacio. Optimizing a 3D-FWT Video Encoder for SMPs and HyperThreading Architectures. In *13th Euromico Conference on Parallel, Distributed and Network-Based Processing*, (PDP), 2005.

[9] J. Gu, Z. Li. Efficient Interprocedural array Data-flow Analysis for Automatic Program Parallelization. *IEEE Transactions on Software Engineering,* Volume 26, Issue 3, March 2000, Page(s):244 - 261.

[10] C. Hsu, M. Ko, and S. S. Bhattacharyya. Software synthesis from the dataflow interchange format. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pages 37-49, Dallas, Texas, September 2005.

[11] C. Hsu and S. S. Bhattacharyya. *Dataflow interchange format version 0.2*. Technical Report UMIACS-TR-2004-66, Institute for Advanced Computer Studies, University of Maryland, College Park, November 2004.

[12] E.A. Lee and D.G. Messerschmitt. Static Scheduling of Synchronous Data-flow Programs for Digital Signal Processing,. In *IEEE Transactions on Computers*,Vol. C-36, No.2, February, 1987.

[13] B. M. Maxiarz and V. K. Jain. Rapid Prototyping of Parallel Processing Systems on TESH network. In *Proceedings of Ninth International Workshop on rapid System Prototyping*, 1998.

[14] J. L. Pino and K. Kalbasi. Cosimulating synchronous DSP applications with analog RF circuits. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, Nov. 1998.

[15] C. B. Robbins. Autocoding toolset software tools for automatic generation of parallel application software. Management Communications and Control, Inc., Technical Report, 2002

[16] G. Steven, X. Tian, and Y. Chen. Efficient Multithreading Implementation of M.264 Encoder on Intel Hyper-Threading Architectures. In *Proceedings of the Joint Conference on Information, Communication and Signal Processing and the Fourth Pacific Rim Conference on Multimedia,* 2003.

[17] X. Zhang, H. Deng. Distributed Image Edge Detection Methods and Performance. In *Sixth IEEE Symposium on Parallel and Distributed Processing,* 1994.