Chapter 5

Parallelization of Tracking Framework

5.1 Introduction

Advancement of multi core processor provides researchers a platform to enhance the speed of existing algorithm by exploiting parallelism. In the existing algorithm to speed up the processing, thread level parallelism is applied which enhance the running time of the algorithm by executing the independent instruction in different core.

All the vision tracking algorithm requires a lots of computation and memory intensive operation. Most of these applications can be exploited to data level parallelism to enhance the speed of processing. These computational intensive works can be executed parallely for real time application to enhance computational efficiency of the system. In continuation to the last chapter, here we describe the multi threaded implementation of our algorithm in order to exploit parallelism. Serial implementation of vision tracking methods normally operated on the SISD architecture for their task. In all serial implementation of vision tracking method, image reading operation is carried out serially along with all its sequential operations for processing the frame. In contrast, parallel processing provided a set of operations act on the portion
of image data that belongs to its local memory. Multi thread implementation follow the SIMD parallelism where group of independent instruction executed parallely by different core. Ideally, with the increasing number of processor more parallelism for independent operation can be enhanced, but at the same time larger amount of memory is required to keep the local set of images for processing.

These above factors motivated us towards implementing our object tracking framework in parallel platform. Using OpenMP, we have implemented the group of independent instruction of our tracking framework to execute concurrently by the group of parallel threads. This will reduce the execution time of our covariance tracker.

**Problem Definition:** To design a parallelization framework for a covariance tracker in order to reduce the computation time of the single object tracking.

**Objective:** The proposed model has been developed to achieve the following key objectives with respect to the literature survey 2.4.

- Reduces the computational complexity by distributing of workload equally among all the cores.

- Improving the tracking accuracy by searching the target in the given search area in a faster way by exploiting the parallelism and also removing the false detection of the target in the frame.

**Contribution:**

- A multi-threaded implementation is carried out by OpenMP libraries to reduce the computational time.

- A faster mode of tracking algorithm for detecting and tracking the single object using the parallel thread on multi core architecture.

- False detection and drifting of the object from the trajectory is reduced by faster mode of tracking for the moving video.
5.2 Proposed Parallel Framework for Tracking

Our parallelization framework for tracking consists of two phases. In the first phase of the framework, distribution of the particles in the sample space is carried out with the help of the distance criterion function. These estimations are executed concurrently by the multiple number of threads. In the second stage, update of the referential model and resampling the particles for the successive iteration is carried out. Figure 5.1, shows the work flow of our parallelization model.

![Parallel Framework for Tracking](image)

Figure 5.1: Parallel Framework for Tracking
5.2.1 Prediction Of Particles

This step is used for estimating the location of the target present in the image. We have used particle filtering method for the estimation of the target in the given frame. In this method, location of the target is estimated by distributing the set of samples or particles in the search space of the image and then likelihood score is estimated for each particle. On the basis of maximum likelihood score, location of the target is identified.

As cited earlier in 2.3, Particle filtering algorithm is a two step process i.e prediction and the updation step. prediction stage uses the probabilistic system transition model to predict the posterior at time $t$. Given all the set of observed state up to time $t$, i.e represented as the $Y_{1..t} = y_1, y_2,...y_t$. State of the object $x_t$ is estimated as

$$p(x_t|y_{t-1}) = \frac{1}{N} \sum_{i=1}^{N} p(x_t|x_t^i) \tag{5.1}$$

In the update stage, The posterior estimation $p(x_t|y_{1..t})$ at time $t$ can be updated

$$p(x_t|z_{1:t}) = \sum_{i=1}^{N} (w_t^i \delta(x_t - x_t^i)) \tag{5.2}$$

Here $w_t^i$ is the weight of each sample and it is expressed as

$$w_t^i = \frac{p(y_t|x_t^i)p(x_t^i|x_{t-1}^i)}{q(x_t|x_{1:t-1}, y_{1:t})}$$

where $q(.)$ is the importance density distribution with likelihood function $p(y_t|x_t^i)$ and the set of weight criteria for the sample must satisfies

$$\sum_{i=1}^{N} (w_t^i) = 1 \tag{5.3}$$

The number of particles for a system is fixed and usually decided by the user during initialization. So for each particle sample posterior estimation can be found out using the above estimation equation.
5.2.2 Construction of Low Dimensional Feature Space

A low dimensional feature covariance matrix is constructed for each particle that is present in the search space of the object. As shown in the Figure 5.2, for each frame captured from the camera, a search space is constructed around the object and then particles are distributed in the search space.

![Particle sample distribution in the sample space](image)

Using Tuzel et al. [8], a low dimensional feature covariance matrix is constructed for each of the candidate target region. Each pixel of the target region is represented with the feature descriptor

\[ f_k = [x, y, I(x, y), I_x, I_y, \sqrt{I_x^2 + I_y^2}, \arctan \frac{I_x}{I_y}] \]  

where \( x \) represent the coordinate position in the x direction \( y \) represent the coordinate position in the y direction \( I(x, y) \) is the intensity value of the pixel \( I_x \) is the derivative of the pixel in the x direction \( I_y \) is the derivative of the pixel in the y direction

Mathematically, it can be written as the image region with size \( m \times n \) is converted into the feature matrix of size \( m \times n \times d \) where \( d \) is the number of feature descriptor for each pixel. This feature matrix \( F_{m \times n \times d} \) is again converted into the covariance matrix of size \( d \times d \). We have used the Forstner [7] distance for similarity matching between the target region and the candidate assign the weight to each of the particles.
\[ \rho(C_i, C_j) = \sum_{k=1}^{d} (\sqrt{\log^2 \lambda_k}(C_i, C_j)) \] (5.5)

where \( \lambda_k(C_i, C_j) \) are the generalized eigenvalues of \( C_i, C_j \) which is derived from the equation

\[ \lambda_k C_i X_k - C_j X_k = 0 \] (5.6)

Here \( X_k \) represent the generalized eigen values. The distance measure \( \rho \) satisfies the metric axioms positivity, symmetry, triangle inequality, for positive definite symmetric matrices. In the proposed method, best candidate object is selected on the basis of minimum value.

### 5.2.3 Multi Thread Initialization

Multiple parallel threads \( t \) are employed to estimate the likelihood value of each sample. Master thread initiating the process of constructing \( N \) number of threads. Estimating the weight of each particle is independent task. In order to process each particle for estimating the weight, \( t \) number of threads are initialised. This task have to be processed simultaneously by the parallel thread in order to reduce the computation time. For the optimal use of resources, number of threads \( t \) that is initialized must be greater than or equal to the number of core of the machine. Estimation of likelihood score of each particle depends on the distance of the of the particle from the target. An exponential function of the distance is adopted as the local likelihood in the particle filter.

\[ p(y_t|x_t) = \exp(\lambda \rho^2(C^*, C(x_t^*))) \] (5.7)

Algorithm 5.1, illustrate our proposed parallel tracking algorithm. Here, master thread initialize the \( t \) number of the thread, each thread then undergoes computation of the particle weights. Now, master thread collects all those weights of the particles from the slave thread and then select the
5.2.1 Prediction Of Particles

Particle with maximum score. Thus the particle with highest score will be selected for the location of the target.

Algorithm 5.1 Parallel algorithm for Tracking

Input \([X_n, C_r, t \leftarrow no - of - thread]\)

Output : \(C_t\)

Initialise for each \(X_j \in [1, 2, 3, ... n]\) identify the position of particles

for each \(i \geq 1\) to \(n/t\) do

Compute \(C_p\) where \(p = t_{id} + i - 1 \times t\)

Compute \(\rho_p(C_p, C_j) = \sum_{k=1}^{d}(\sqrt{\log^2 \lambda_k(C_p, C_r)})\)

Calculate the likelihood of each particle as

\(w^i_i = p(y_i|x_i) = \exp(\lambda \rho^2(C^*, C(x^*_n)))\)

Normalize the probability distribution of particle

\(w^i_i = p(y^n_i) = p(y^i_i)/\sum_{i=1}^{n} p^i_y\)

Select the location of target as a particle with minimum distance.

end for

Update the referential frame using the Equation 4.28

\[\mu_{o,jk} = \left(\frac{\sigma_{i,jk}^2 + \sigma_{j,k}^2}{2}\right) + \delta_1 \delta_2 + \frac{1}{\sqrt{2\pi}} (\delta_1 \sigma_{1j} + \delta_2 \sigma_{1k}) + \frac{1}{\sqrt{(2\pi)}} (\delta_1 \sigma_{2j} + \delta_2 \sigma_{2k})\]

5.2.4 Resampling

In Particle filtering technique, weight degeneracy problem is often found. As particles with higher weights are promoted and lower weights are discarded for the next iteration. So, resampling or reassign step is crucial one to generate an unweighted set according to their importance weight. This portion of particle filter can not be carried out in parallel because of data dependencies of the particles. In the proposed work the re sampling part of the particle filter is evaluated in serial execution order. Once the likelihood estimation of all the particles are completed, then they are stored on an array to keep this value preserved. Using the systematic sampling approach, each of the normalized value of the particle are processed in an array to find the cumulative
Chapter 5. Parallelization of Tracking Framework

sum of the weights and then compare it against the set of random number to decide which particle has the maximum chance of matching. So for each iteration, above mentioned process is followed to avoid degeneracy problem.

5.3 OpenMP Implementation

OpenMP is an Application Program Interface (API). It is developed for executing the multi thread application written in C/C++ and FORTRAN. OpenMP is a master slave programming model.

A master thread is responsible for constructing $t$ number of parallel thread and then the scheduler assign these threads to different core for processing. OpenMP consists of set of compiler directives, runtime calls and environment variables. As shown in the Figure 5.3, program starts with the master thread and then the worker threads are created by the scheduler on receiving the ‘"# pragma omp parallel " ’ of the program. Scheduler also responsible for assigning these parallel thread to the different core for the processing.

parallel regions are executed concurrently by a team of thread initiated by the master thread. OpenMP programming model supports the shared memory model. As shown in the Figure 5.4, in multi core architecture, each core is interconnect to the shared memory and provided with small local cache memory. Overhead of the data accessing is reduced by keeping the data
in the private cache. For object tracking, each thread reads the candidate image region from the shared memory and then keeps a local copy in the private memory during the computation of the feature covariance matrix and calculating the likelihood parameter of particles. Variables present in the local memory only active for the computation part. For construction and execution of thread in OpenMP, \texttt{#include <omp.h>} should be included to access OpenMP functions and capabilities. The \texttt{#pragma omp} work-sharing construct enables programmers to share and execute the independent task. This construct applies to \texttt{for loops} where subsequent iterations are independent of each other that is, changing the order in which iterations are called does not change the result.

\texttt{#pragma omp parallel}
{
    ......  
    ......  
    id = omp\_get\_thread\_num()  
    Nthread = omp\_get\_thread()  
    istart = id * N/Nthread;  
    iend = (id + 1) * N/Nthread;  
    for\(i = \text{istart}; i < \text{iend}; i++\)  
    {
        Ci = mapping(Ii);  
        p(y_i|x_i) = p(y_i) / \sum_{i=1}^{n};  
    }
Using the above construct, we have estimated the weights of $N$ number of particles with $t$ number of parallel thread for the detected region $R_t$. Each of the likelihood value of the particles is then updated in the global shared memory. On the basis of the maximum likelihood value, location of the target in the current frame is selected. Similarly another parallel construct is used to update the referential model for detected region up to $R_{t-1}$.

The second work-sharing construct of the OpenMP is `#pragma omp sections` which allows the programmer to parallelly assign the different tasks across processing element, where each processing element runs a unique piece of code. In our work, we have used two section. First OMP section handles the estimation of weights for the frame $R_t$ whereas the second section is employed to update the reference object up to detected region $R_{t-1}$.

```plaintext
#pragma omp sections
{
#pragma omp section
Feature_cov(gtid);
#pragma omp section
Update_ref_model(gtid);
......
}
```

As shown in the above pseudo code, each of the `pragma omp sections` are executed parallely. For exploiting the parallelism properly, `pragma omp sections` are implemented with pipelined concept. When one group of thread execute the first OMP section for evaluating weight of the $n^{th}$ iteration then at the same time another group of thread execute the model update for $(n - 1)^{th}$ iteration simultaneously.
5.4 Performance Evaluation

Our proposed model is implemented using OpenCV -2.4.2 libraries on Ubuntu platform and this simulator is configured with OpenMp library to support the thread level parallelism. All the experiment are conducted in a intel core i-7 machine. Performance of the proposed method is tested on two pedestrian video sequences executed for 200 frames. These video sequence has been collected from the tracker benchmark V1.0 repository. The covariance matrix descriptor of a color image region is represented by 23 X 23 symmetric matrix.

The main thread starts reading an image and after that \( n \) numbers of parallel thread are created using pragma omp. These parallel thread construct the feature vector and then find the posterior distribution of the particle. Location of the target is decided on the basis of maximum likelihood score of the particle. Master thread will collect all the likelihood score from the slave threads and then find the particle with the highest score. Location of the target is then traced on the basis of particle that provide the highest score. Figure 5.5 shows the visual tracking result of the two vision pedestrian video sequence running for 200 frames.

Our proposed parallel method is implemented with different number of thread and particles for the the above mentioned dataset and executed for 200 frames. Table 5.1 and Table 5.2 shows the Execution time of the algorithm implemented on the two pedestrian dataset with different number of particles varying from 40 to 200 and number of threads varying from 1 to 16 number of thread.

The Performance measurement parameter speed of computation are evaluated for these two pedestrian dataset and the details analysis are discussed below.

**Speedup:** It is the ratio between the execution time of a serial process to the parallel process. It is practically impossible to obtain the \( n \) times speed up rate by using the \( n \) number of thread. It is observed from the Amdahl’s law that \( N \) times speedup rate cannot be achieved using \( N \) core as all the
portion of a program cannot execute parallelly. According to Amdahl’s law, Over all Speedup that is gained by the parallelism can be estimated as

\[
\text{SpeedUp} = \frac{1}{(1 - f) + \frac{f}{N}}
\]

where \( f \) is the portion that can be enhanced execution time for using the parallelism and \((1 - f)\) is the fraction of time that cannot be parallelized.

<table>
<thead>
<tr>
<th>Particles/Thread</th>
<th>40Particles</th>
<th>80Particles</th>
<th>160Particles</th>
<th>200Particles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>70</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>25</td>
<td>28</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>10</td>
<td>13</td>
<td>23</td>
</tr>
</tbody>
</table>

Table 5.1: Processing time in seconds of Dataset I with different number of thread Execution

When only one thread is used, essentially the serial implementation is realized and therefore, the serial performance is computed by the executing
5.4 Performance Evaluation

<table>
<thead>
<tr>
<th>Particles/Thread</th>
<th>40Particles</th>
<th>80Particles</th>
<th>160Particles</th>
<th>200Particles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18</td>
<td>38</td>
<td>44</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>25</td>
<td>34</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>14</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
<td>14</td>
<td>16</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 5.2: Processing time in seconds of Dataset II with different number of thread Execution

the program under single thread. As shown in Figure 5.6 and Figure 5.7, Speedup is evaluated for the two benchmark dataset. It has been observed that the performance does not improves greatly after 8 number of threads. 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.

Figure 5.6: Speedup measurement for dataset I
Our proposed parallel algorithm also implemented with the MATLAB 7.0 and applied on both the dataset with 80 numbers of particles. It has been observed that the MATLAB simulator requires an execution time of 330 sec for the first dataset and an execution 382 sec for the second dataset. For the run time application, normally frames are captured at 30 fps. So large execution time create false detection of target by the tracker. Compare to MATLAB code, Execution of our algorithm in OpenMP with 8 number of threads achieve the target rate for processing 30 frames in less than one seconds.

### 5.5 Conclusion

Parallel implementation of appearance based particle filtering algorithm is very effective in tracking the non-rigid object. Use of OpenMP master slave model to parallelize the serial version of our algorithm makes the computation time considerably less in searching the target. OpenMP work sharing
construct enables the thread in executing the parallel region of the program simultaneously. For single object tracking, the current method is very efficient and more accurate in detecting the target. One of the limitation of the current method is that for multi object tracking, the performance of the approach is not efficient as more dependency of instruction is found and also more memory is consumed in keeping the several target object for the model update. This approach can further be extended from CPU based multi thread to GPU based multi thread as more hardware support is available in GPU for parallelization.