Morphing Parallelization Strategy to Support On-the-Fly Video Analysis

Munehiro Fukuda
Comp. & Soft. Systems
University of Washington
Bothell, WA
mfukuda@u.washington.edu

Shuichi Kurabayashi Jeremy Hall
Media & Governance
Keio University
Fujisawa, Japan
{kurabaya, jhall}@sfc.keio.ac.jp

Yasushi Kiyoki
Env. & Info. Studies
Keio University
Fujisawa, Japan
kiyoki@sfc.keio.ac.jp

Abstract

To automate and accelerate Internet video search, we propose a parallelized on-the-fly video analysis that is distinguished from the conventional approaches in the following two features. One is to morph its parallelization by first screening out hundreds of videos at each processor in a low quality of analysis, then repeating this screening process as increasing the analyzing quality, and finally examining each of the final candidates among multiple processors. The other is to download and handle any portion of a video clip independently at a different processor, which is differentiated from the conventional stream-based analysis that scans a video from beginning to end.

We have evaluated the promising performance of our proposed system by parallelizing the Auotonoesis color-schema-based video analyzer on top of the AgentTeamwork parallel-computing middleware. This paper presents implementation techniques and preliminary performance to support our parallelization strategy.

Keywords: parallel video analysis, parallel video downloading, switched parallelism, mixed parallelism

1. Introduction

Lengthy manual efforts are still required for users to find videos on the Internet matching their expectations. Despite that video-search engines help facilitate the narrowing of a search by providing results prioritization, subtitles listing, and videos rating, users are not yet relieved from their heuristic trial-and-error video search. This is because legacy video databases depend on keyword search that may list hundreds of videos; impossible to pick through in a reasonable amount of time.

A query by video clip [3] can reduce this burden by sampling each video in a series of screen shots so as to check how close they appear to a user's intention. More specifically, once a list of candidate videos are identified with a keyword search, their screen shots are compared in color and texture to those of another video clip (or even a set of independent pictures) which visually represents a user's query. However, this type of search still needs a large amount of analyzing time, thus unacceptable for the use of on-the-fly video retrieval.

Among hundreds of candidates, the majority is a set of unlikely videos. There is no reason to precisely analyze each of them from beginning to end. Hence, a key to realizing quick video search is how fast the analysis can drop off unlikely videos through a rough analysis toward a detailed analysis of prospective candidates, while leaving the most-likely videos in the final set.

Parallel computing is a typical solution to accelerate such video analysis. The easiest is the bag-of-task parallelization that allows computing nodes to share a list of target videos and to pick up one by one from the list for its content analysis until the list is exhausted. This **multi-videos-over-multi-processors (MVMP)** parallelization can efficiently deal with hundreds of videos but is lower-bound to time for analyzing the longest video. Therefore, when it comes to a detailed analysis of the final candidate videos, we need to also consider the **single-video-over-multi-processors (SVMP)** parallelization that partitions a single video into smaller stripes, each analyzed at a different computing node.

Based on these observations, we propose a parallelized on-the-fly video analysis that changes its parallelization from the MVMP to SVMP strategy as repeating its analyzing iteration from the roughest to the most precise quality. Two software technologies are used for actual video analysis and parallelization: Autonoesis [5] and AgentTeamwork [2]. The former decodes a flash video, samples it in a series of screen shots, and creates an archive of color histograms, each corresponding to a different screen shot. The latter deploys a hierarchy of mobile agents to launch a user job over a collection of remote computing nodes. Using them, we have prototyped each component of our proposed parallel video analyzer including a parallel video

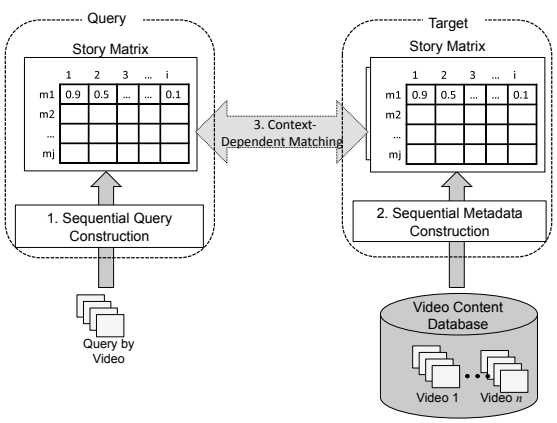


Figure 1. Autonoesis' system overview

downloader, and MVMP as well as SVMP-based analyzers.

This paper presents the implementation techniques and performance evaluations to support our proposed parallelization. The rest of paper is organized as follows: Section 2 gives an overview of Autonoesis and AgentTeamwork; Section 3 explains our implementation strategy; Section 4 shows its preliminary performance; Section 5 discusses related work from video analysis and parallelization viewpoints, and Section 6 presents our conclusions.

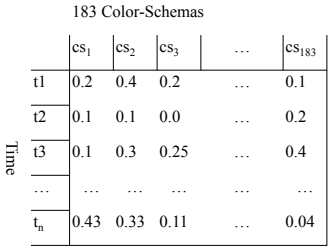
2. Software Infrastructure

This section gives an overview of Autonoesis and AgentTeamwork which together form the software infrastructure of our on-the-fly video analyzer.

2.1. Autonoesis

Autonoesis is a video search engine that performs the following three tasks as shown in Figure 1: (1) generating color-schema-based meta-data, (named a **story matrix**) for a user's query by video, (2) constructing a story matrix for each of target videos retrieved from databases, and (3) comparing the query with the targets in terms of their story matrices in order to rank their proximity to the query.

Figure 2 illiterates a story matrix whose columns and rows respectively correspond to a graphic set of 183 different color schemas and a timeline-based set of video frames. To construct a matrix, a video clip is decoded into frames along the timeline, each then converted from its RGB to HSV presentation, from which we create the corresponding color histogram, based on 150 representative colors of the Munsell system. Thereafter, the appearance ratio of each color schema in the histogram is calculated, and only the *x* biggest ratios (where *x* is a user-given number) are recorded



	cs_1	cs_2	cs_3	…	cs_{183}
t1	0.2	0.4	0.2	…	0.1
t2	0.1	0.1	0.0	…	0.2
t3	0.1	0.3	0.25	…	0.4
…	…	…	…	…	…
t_n	0.43	0.33	0.11	…	0.04

Figure 2. A story matrix

in the matrix cells specified with their corresponding color schema and timeline-based frame.

Assuming that a query by video and each target video have both the same number of timeline-based frames, (defined as n), Autonoesis computes EV, the proximity between the query and the target in terms of their story matrix, using the following formula:

$$EV = \sum_{i=0}^{n-1} (\sum_{j=0}^{182} Q_{ij} \cdot M_{ij}) \quad (1)$$

where

- Q_{ij}: the appearance ratio of the color scheme cs_j found in the query's frame at time i
- M_{ij}: the appearance ratio of the color scheme cs_j found in the target's frame at time i
- $\sum_{j=0}^{182} Q_{ij} \cdot M_{ij}$: the sum of the cs product between Q's and M's corresponding matrix items at time i

By sorting EVs of all targets, Autonoesis finds the best target fitted to a user's intention.

2.2. AgentTeamwork

AgentTeamwork is a grid-computing middleware system that coordinates parallel and fault-tolerant job execution with mobile agents [2]. A new computing node can join the system by running AgentTeamwork's mobile-agent execution daemon to exchange agents with others. The system distinguishes several types of agents such as commander, resource, sentinel, and bookkeeper agents, each specialized in job submission, resource selection, job deployment and monitoring, and job-execution bookkeeping respectively.

As shown in Figure 3, a user submits a new job with a commander agent that receives from a resource agent a collection of remote machines fitted to the job execution. The commander agent thereafter spawns a pair of sentinel and bookkeeper agents, each hierarchically deploying as many children as the number of the remote machines. Each sentinel launches a user process at a different machine with a

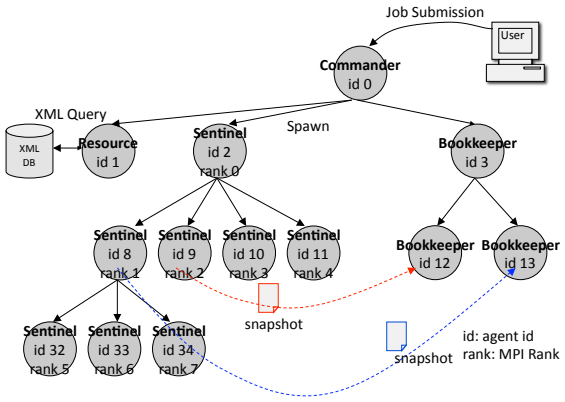


Figure 3. AgentTeamwork's system overview

unique MPI rank, takes a new execution snapshot periodically, sends it to the corresponding bookkeeper, monitors its parent and child agents, and resumes them upon a crash. A bookkeeper maintains and retrieves the corresponding sentinel's snapshot upon request.

A user program is wrapped with and check-pointed by a user program wrapper, one of the threads running within a sentinel agent. The wrapper internally facilitates error-recoverable TCP and file libraries, each named **GridTcp** and **GridFile** respectively, which serialize and de-serialize an execution snapshot with in-transit messages and buffered file data. A user program can take advantage of these fault-tolerant features by inheriting the **AteamProg** class that has re-implemented Java socket, file, and even MPI classes with **GridTcp** and **GridFile**.

3. Parallelization

It is our ultimate goal to dynamically decide the best values of parameters that optimize our parallelization morphing and video-clip partitioning approaches. In this section, we first define these parameters through our mathematical analysis, and thereafter explain the programming details of our parallel video analyzer.

3.1. Mathematical Analysis

Our parallelized video analyzer repeats an iterative analyzing stage of creating a story matrix for a target video i, comparing it with a query matrix, and increasing the number of frames sampled per second for an analysis at the next stage. Given the following four parameters, the time to analyze a target video i at the iterative stage j, (namely $Tstage_{ij}$) is calculated with Formula 2.

- $Tvideo_i$: the length of a video i in seconds
- S_j: the number of frames sampled per second at a given analyzing stage j. (For example, $S_j = 1/8$ samples one frame every eight seconds.)
- P_i: pixels per frame, (i.e., ppf) of a given video i
- $Tpix$: the analyzing time for each pixel

$$Tstage_{ij} = Tvideo_i \cdot S_j \cdot Tpix \cdot P_i \tag{2}$$

Let us assume that top R_j ratio of the videos at each stage j (where $R_0 = 1.0$) is passed for their analysis at the next stage $j+1$. The total time to find the final candidate clip among all Nv videos received from Internet (named $Ttotal$) can be estimated with Formula 3

- Nv: the total number of videos to analyze
- Ns: the number of analyzing stages
- R_j: the ratio of keeping the best videos at stage j
- Nt: targets to be analyzed at stage j, defined as $Nv \prod_{k=0}^{j-1} R_k$
- Np: the number of processors used for parallel video analysis

$$\begin{aligned} Ttotal &= \sum_{j=1}^{Ns} \sum_{i=1}^{Nt} Tstage_{ij} \\ &= \sum_{j=1}^{Ns} \sum_{i=1}^{Nt} Tvideo_i \cdot S_j \cdot Tpix \cdot P_i \end{aligned} \tag{3}$$

Formula 3 assumes that all Nv video clips have been already loaded in memory. Without even considering any file-downloading and processor-communication overheads, the ideal parallelization with Np processors will be performed in $Ttotal/Np$. For parallelization morphing, $Np > Nt$ is the condition to switch from the MVMP to SVMP parallelization strategy at stage j. Even at stages before j (where $Np < Nt$), we may apply not only MVMP to the top Np-divisible number of videos, which is $(Nt - Nt \bmod Np)$ videos, but also SVMP to the rest, namely $(Nt \bmod Np)$ videos, in order to keep all processors busy.

Formula 3 can be simplified by considering the following two restrictions: (1) we focus on only targets whose length, (i.e., $Tvideo_i$) is identical to that of a query by video clip, and (2) all videos have the same P_i, (e.g., $640 \times 480 ppf$).

$$Ttotal = Tvideo \cdot Tpix \cdot P \sum_{j=1}^{Ns} \sum_{i=1}^{Nv \prod_{k=0}^{j-1} R_k} S_j \tag{4}$$

Therefore, initially given Nv videos with the same $Tvideo$ length, we should take into account only those three

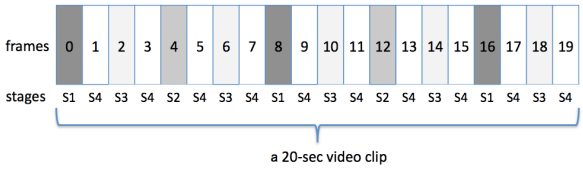


Figure 4. Analyzing a 20-second video clip through four stages

parameters of Ns, S_j, and R_j for balancing the quality of analysis and the total analysis time. The following gives three examples for choosing Ns, S_j, and R_j to analyze 100 videos, each with a 20-second length.

- $Ns = 4$, where $S_1 = 1/8$, $S_2 = 1/8$, $S_3 = 1/4$ and $S_4 = 1/2$ samples per second. As shown in Figure 4, stage 1 should take samples at 0^{th}, 8^{th}, and 16^{th} seconds; stage 2 at 4^{th} and 12^{th} seconds; stage 3 at $2^{nd}, 6^{th}, 10^{th}, 14^{th}$, and 18^{th}; and finally stage 4 at $1^{st}, 3^{rd}, 5^{th}...19^{th}$ seconds. R_i should be $1 - S_{i-1}$, (i.e., $R_1 = .88$, $R_2 = .88$, and $R_3 = .75$) so as to avoid dropping off promising candidates. Applying these numbers to Formula 4, the total analyzing time is $(100 \times 3 + 88 \times 2 + 75 \times 5 + 56 \times 10)Tpix \cdot P = 1411Tpix \cdot P$.
- $Ns = 2$, where both S_1 and S_2 are $1/2$ samples per second. Then, stages 1 and 2 should take samples at even- and odd-number seconds respectively. R_i should be .5 for example. The total analyzing time will be $(100 \times 10 + 50 \times 10)Tpix \cdot P = 1500Tpix \cdot P$.
- $Ns = 1$, where $S_1 = 1$. Obviously, each video needs $20Tpix \cdot P$ analyzing time. Therefore, the total will be $(100 \times 20)Tpix \cdot P = 2000Tpix \cdot P$

Needless to say, Np is the fourth parameter that reduces the total execution time, while being restricted to runtime resource availability.

3.2. Implementation of Parallel Video Analyzer

Regardless of sequential or parallel execution, essential primitives for our video analysis include video downloading and analysis, each prototyped as follows;

1. **void download(string vURL, int start, int end);** downloads a range between **start** and **end** seconds of a video clip from **vURL** and stores it under the /tmp directory.
2. **InputStream autonoesis(File vFile, double offset, double samples);** reads a video file from /tmp/**vFile**, starts an analysis at **offset** seconds from its top, takes **samples** per second, and returns an **InputStream** archive.

Given these two primitives, our implementation focuses on the following two important strategies:

1. How to control download( ) and autonoesis( ) over multiple processors, namely in MVMP and SVMP
2. How to repeat MVMP and SVMP over multiple analyzing stages as relaying intermediate results from one stage after another

3.2.1 Parallelization over Multiple Processors

To maintain a downloaded video for analysis, we use a data structure named **VClip** that is defined in Figure 5. The array **targets[]** includes all video clips downloaded from Internet with a keyword search. The variable **query** is a query by a video clip that may be also downloaded from Internet.

```
1 private class VClip{
2  string vURL;                    // YouTube URL
3  File   file;                    // file in /tmp
4  double score;                   // degree of matching query
5 }
6 private VClip query, targets[];// video clips
7 private double score[];          // degree of matching query
```

Figure 5. Video data

Assuming that query and targets[] have been already filled, we focus on a certain stage that starts conducting an MVMP or SVMP analysis on the range [**L**..**R**] of targets as well as query at **offset** seconds from their beginning by sampling **S** frames per second. Therefore, both MVMP and SVMP are designed as a function with the same argument list, each named **mvmp**( ) and **svmp**( ). In the following discussions, we also assume that two data members such as **rank** and **size** have been initialized by MPI.COMM_WORLD.Rank( ) and MPI.COMM_WORLD.Size( ) upon an analyzer invocation.

```
1 private void mvmp(VClip query, VClip[] targets, int L,
2                   int R, double offset, double S){
3  InputStream qIn=autonoesis(query.file, offset, S);
4  int start=L+(R-L+1)/size*rank;
5  int end = (rank==size-1) ? R: start+(R-L+1)/size-1;
6  for(int i=start; i<=end; i++){
7   if(!targets[i].file.canRead())
8    download(targets[i].vURL, 0, 99999);// till EOF
9   InputStream tIn=autonoesis(targets[i].file,offset,S);
10  score[i]=targets[i].score=matching(qIn, tIn);
11 }}
```

Figure 6. Multi-videos over multiple processors (MVMP)

In MVMP as shown in Figure 6, starting with an analysis of a given query video (line 3), each processor (or rank) equally divides the range [L..R] of the target videos into

```
1 private void svmp(VClip qry, VClip[] targets, int L,
2                   int R, double offset, double S){
3  InputStream qIn=autonoesis(qry.file,offset+rank,S/size);
4  for(int i=L; i<=R; i++){
5   if(!targets[i].file.canRead())
6    download(targets[i].vURL, 0, 99999);// till EOF
7   InputStream tIn=autonoesis(targets.file[i],
8                              offset+ran,S/size);
9   score[i]=target[i].score=matching(qIn, tIn);
10 }}
```

Figure 7. Single video over multiple processors (SVMP)

subranges, one of which is assigned to the processor, based on its rank (lines 4-5). The processor picks up each target from its subrange (line 6), downloads it from Internet if it has not yet been done so (lines 7-8), analyzes its contents (line 9), and compares the result with the query (line 10).

In SVMP, each processor changes its parallelizing strategy into sharing all the videos with the other processors and analyzing a different portion of each video. The algorithm samples the whole range of each video but by every $S/size$ frames per second from the offset, so that all videos are eventually analyzed by S frames per second. For this purpose, svmp( ) analyzes the query file at the $S/size$ sampling rate (line 3) in Figure 7 and then picks up each of all the videos (line 4), for which svmp( ) performs a file downloading (lines 5-6), an $S/size$-sampled analysis(lines 7-8), and a comparison with the query (line 9). We can also consider an alternative SVMP algorithm where each processor downloads and samples only a $1/size$ range of each video but by every S frames per second.

3.2.2 Parallelization over Multiple Stages

Figure 8 shows how to control MVMP and SVMP over multiple analyzing stages, in particular when focusing on the four-stage example illustrated in Figure 4.

All computation starts from **compute( )**. Initialized to -1, variable **stage** increments its value to 0, 1, 2, and 3, each used to not only indicate an analyzing stage (line 13) but also keep track of an independent snapshot (line 27). The compute( ) method first carries out a keyword search with YouTube, stores URLs for candidate videos in **target[]**, and downloads a query video. (lines 10-12). Thereafter, compute( ) repeats an analyzing stage from 0 to 3 (lines 13-22). For each stage, if the number of targets for inspection is larger than or equals to that of all computing nodes, namely **size** in the code (line 14), we apply MVMP to the first size-divisible number of clips (lines 15-17), while analyzing the rest with SVMP (lines 20-22). If the number of targets is less than **size**, we focus on SVMP only. Figure 9 describes how to allocate 100 video clips over 70 processors and to analyze them with MVMP and SVMP throughout

```
1  private int stage = -1;          // the current analysis
2
3  private void compute(){
4   int      top = targets.length;// # of top videos to analyze
5   int      Ns = 4;              // # of stages
6   double[] off = {0, 4, 2, 1};  // sampling offset
7   double[] Sj={0.125, 0.125, 0.25, 0.5};// sampling interval
8   double[] Rj={0.875, 0.875, 0.75};// ratio to keep best videos
9
10  initTargets(FlvDownLoader.
11           getVideoUrlsByKeywords(args, MAX), targets);
12  download(query.vURL, 0, 99999);// download query till EOF
13  for(++stage; stage<Ns; stage++){
14   if(top / size > 0){ // mvmp for first #CPUs-divisible video clips
15    int L = 0, R = top/size*size-1;
16    mvmp(query, targets, L, R, off[stage], Sj[stage]);
17    MPI.COMM_WORLD.Alltoallv(score,0,R,...,MPI.Double);
18   }
19   if(top % size > 0){ // svmp for the remaining video clips
20    int L = top-top%size, R = top-1;
21    svmp(query, targets, L, R, off[stage], Sj[stage]);
22    MPI.COMM_WORLD.Allreduce(score, L, R,..., MPI.Sum);
23  }
24  for(int i=0; i<top; i++) target[i].score=score[i];
25  Arrays.sort(targets, 0, top - 1);
26  top *= Rj[stage];
27  ateam.takeSnapshot(stage); // check-pointing for future recovery
28 }}
```

Figure 8. Total design

four stages.

Of importance is how to pass intermediate results, (i.e., **score[]**). In MVMP, each processor maintains a full score[] of its own subrange and thus needs to exchange it with all the other processors, for which purpose we use MPI's **AlltoallV( )** group-messaging function (line 17). On other hand, in SVMP, each processor maintains a partial score[] of all over the range and therefore must sum up score[] with all the others, using **Allreduce( )** group-arithmetic function (line 22). After receiving the full score[] values, compute( ) sorts targets[] based on their score and chooses the top **Rj** set of targets for the next stage (lines 24-26).

4. Preliminary Performance

We prototyped and evaluated components to download videos in parallel and to perform MVMP- as well as SVMP-based analysis. For video analysis, we only sampled target videos, thus excluding a query analysis and comparison.

4.1. Parallel Video-Downloading

Fitted to both MVMP and SVMP analyses, our video downloader allows each processor to obtain in parallel not only different video clips but also a different range of the same video clip, each called **parallel multi-video downloading** and **parallel single-video downloading** in the following evaluation. For the measurement, we used a 1Gbps-network cluster of 32 Xeon machines, (24 nodes driven at 3.2GHz and 8 nodes at 2.8GHz, each with 512MB memory).

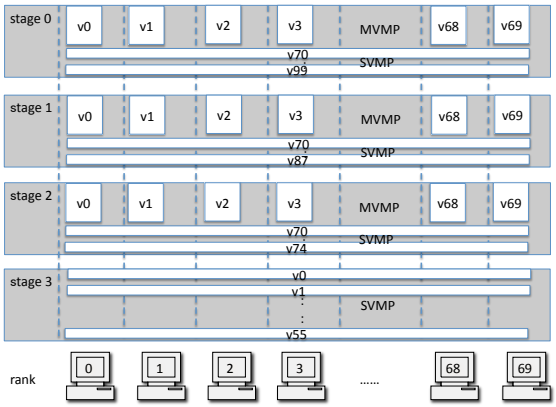


Figure 9. Analyzing 100 video clips with 70 computing nodes

Figure 10 demonstrates a scalable performance of parallel multi-video downloading that delivers the top 32 videos with 7084 seconds and 265MB in total, (i.e., each with 221 seconds and 8.2MB in average) from YouTube (in response to the keyword "solar system") over up to 32 processors.

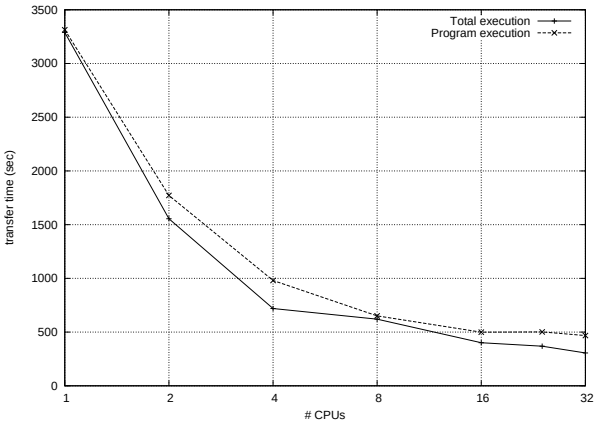


Figure 10. Performance of parallel multi-video downloading

Figure 11 shows a performance of parallel single-video downloading that distributes a $\frac{1}{\#processors}$ segment of the same video over 1 through to 32 processors. Chosen from those 32 clips used in the above experiment, the video has a 137-second and 9.5MB length. The results show that a video clip with a couple of minutes is too short to download to more than eight processors in parallel, because it incurs too much system overhead.

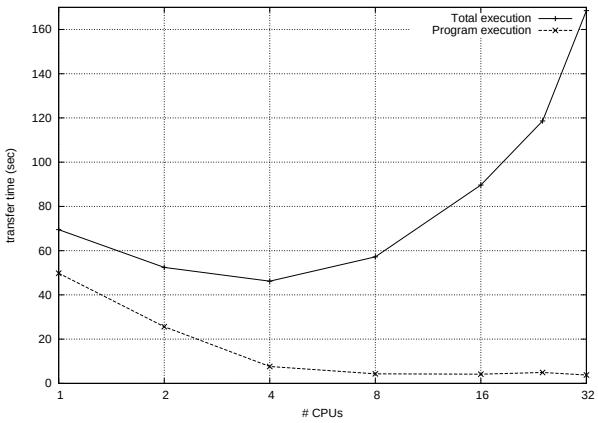


Figure 11. Performance of parallel single-video downloading

4.2. Parallel Video Analysis

Due to Autonoesis' current porting restrictions, we used a 1Gbps-network cluster of 32 iMac machines, (each with 2.16GHz Intel Core-2 Duo and 2GB memory) for both MVMP and SVMP analyses.

MVMP distributed 128 copies of a video clip (with 166 seconds and 6.66MB) to remote computing nodes through AgentTeamwork's file system. Figure 12 confirms MVMP's scalability when using up to 32 computing nodes.

SVMP directly downloaded a different range of a clip (with 140 seconds and 7.1MB) at each remote node. Figure 13 shows the performance of SVMP analysis, parallelizable with only up to 4 or 8 processors. This in turn means that we must use SVMP carefully for parallelizing a collection of small videos or a large video clip without terminating AgentTeamwork for each run, which can alleviate most system overheads.

5. Related Work

This section emphasizes the originality of our parallel video analyzer from the viewpoints of video analysis and parallelization.

Since a video clip is considered as a collection of independent frames, each as even a collection of pixels, most

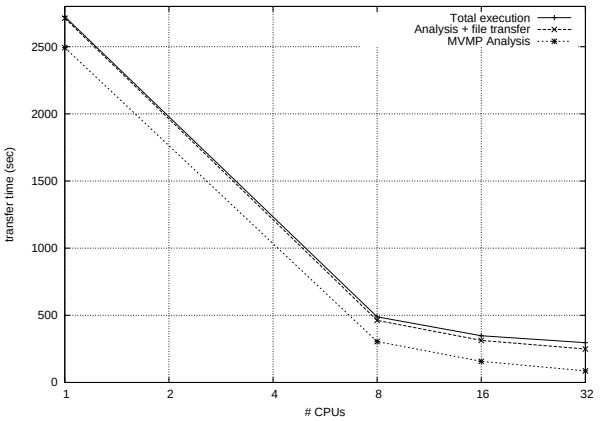


Figure 12. Performance of MVMP analysis

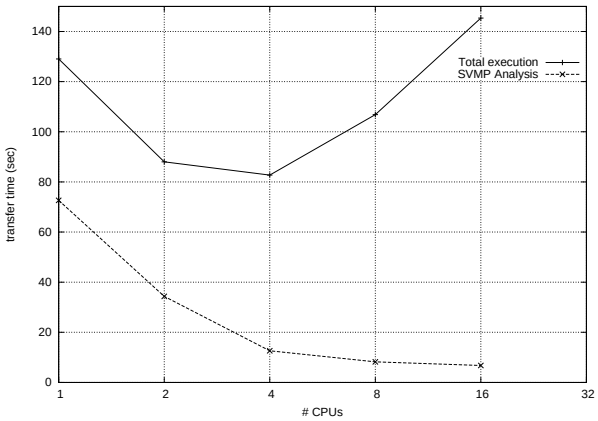


Figure 13. Performance of SVMP analysis

video retrieval, encoding, and analysis have been accelerated with data parallelism. For instance, Parallel-Horus is a multimedia programming library [4] that first scatters video frames over cluster nodes, thereafter performs content analysis on each frame at a different node, and finally collects the results into a single file. Even based on data parallelism, our parallelization is distinguished by the following two points: (1) we download any portions of a given video as independent data items, whereas most systems read each frame one by one (as a stream) before partitioning frames for parallel analysis, and (2) we dynamically change data granularity to be analyzed from a whole video clip to frames, (i.e. from MVMP to SVMP), while other systems fix their granularity to frames or pixel blocks.

Morphing parallelism was discussed in [1] as switched/mixed parallelism that has been used in many applications including matrix factorization and discrete-event circuit simulation. These applications generally start with data parallelism, keep dividing or minimizing a data block into smaller sub-blocks, each eventually worthless to spread over multiprocessors, and thus switch to task parallelism to assign each sub-block to a single processor. Our morphing strategy allocates an entire video to a single processor and subsequently over multiprocessors, which actually transitions from task (as well as mixed) to data parallelism in the reverse direction of switched parallelism.

6. Conclusions

We presented a parallel video analysis that morphs its parallelism from MVMP to SVMP by repeatedly reducing the number of candidate videos while increasing the quality of analysis. Our performance evaluation demonstrated MVMP's scalability, while asserting the careful use of SVMP for a collection of small video clips or a large clip. Our next plan is to finalize the implementation, and afterward to generalize our system as a post-processor of multimedia database search.

References

[1] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the benefits of mixed data and task parallelism. In *Proc. of the 7th Annual ACM Symposium on Parallel Algorithms and Architecutre - SPAA'95*, pages 74–83, Santa Barbara, CA, July 1995. ACM Press.

[2] Munehiro Fukuda, Koichi Kashiwagi, and Shinya Kobayashi. AgentTeamwork: Coordinating grid-computing jobs with mobile agents. *Int'l Journal of Applied Intelligence*, Vol.25(No.2):181–198, 2006.

[3] Anil K. Jain, Aditya Vailaya, and Xiong Wei. Query by video clip. *Journal of Multimedia Systems*, Vol.7(No.5):369–384, September 1999.

[4] Frank J. Seinstra, Jan-Mark Geusebroek, Dennis Koelma, Cees G.M. Snoek, Marcel Worring, and Arnold W.M. Smeulders. High-performance distributed video content analysis with parallel-horus. *IEEE MultiMedia*, Vol.14(No.4):64–75, 2007.

[5] Taichi Uyeno, Shuichi Kurabayashi, and Yasushi Kiyoki. A color-schema-based video search engine with story query construction mechanisms. IPSJ SIG Notes DBS-146, pages 349–354, IPSJ, Tokyo, 2008.