Parallelisation of Block Matching Motion Estimation Algorithms

M.K. Steliaros, G.R. Martin, R.A. Packwood

RR320, January 1997

Abstract

Block matching motion estimation accounts for a high computational overhead in most low bit-rate video coders intended for video-conferencing and multimedia applications. Conventional fixed size block matching (FSBM) and more sophisticated variable size block matching (VSBM) algorithms are profiled to determine how parallelism can be applied to increase efficiency. Three parallel implementations are considered: a homogeneous workstation network using PVM; a Parsytec Super-Cluster, 128-node T800 transputer-based MIMD machine; and a shared memory multiprocessor, using a 'threads' programming model. It is shown that significant improvements in computational performance can be made, even on commonly available computing platforms. Using an MPEG-4 image test sequence, detailed evaluation of a POSIX-conformant threads implementation resulted in near-theoretical maximum speedups being obtained.

Contents

List of Figures

1 Introduction

Most video-conferencing and multimedia video coding systems employ block matching motion compensation of the type originally described by Jain and Jain [4]. The technique is relatively simple to implement, but remains the most computationally demanding phase of the predictive coding scheme. This is especially true when exhaustive searching is employed, the only method guaranteed to provide the best match for each block.

Recently, research effort has been applied to the development of more sophisticated block matching algorithms, to better represent 'true' motion in the image sequence, and hence provide more accurate predictive coding schemes. While successful in this aim, the new techniques invariably have increased the inherent computational complexity. In most software-based video codecs, block matching motion estimation is implemented as a collection of sequential processes. Many of the underlying operations are repetitive in nature, and while this property has been exploited by some application-specific hardware designs, little consideration has been given to parallel implementations on commonly available computing platforms.

2 Block-matching Motion Estimation

The majority of predictive coding implementations use block-matching as the first step in the prediction process. A good match requires that the block in question undergoes uniform translation and avoids overlap between portions of the image that do not. The underlying motivation for the whole process is that the improvement in prediction is significantly higher than the cost of transmitting the motion vector.

2.1 Fixed Size Block-Matching

The fixed size block matching (FSBM) technique, has been used in a number of codecs (e.g. ITU-T H.263 [3]) and is currently proposed for the MPEG-4 video standard [2]. The algorithm relies on partitioning the current frame into $M \times N$ blocks (the block size usually[1] being 16×16 pixels) and then calculating the best motion vector for each block:

```
set best_error to MAX_ERROR
for y in [-rad, rad) in half (or full) pixel steps
  for x in [-rad, rad) in half (or full) pixel steps
    set error to SAD(current, previous, offset by (x, y))
    if error < best_error
      set best_vector to (x, y)
      set best_error to error
```

[1] MPEG-4 proposes also an advanced prediction mode with a 8×8 block size.

The comparison criterion is based on the Sum Absolute Difference (SAD) between the two blocks being compared:

$$SAD = \sum_{j=1}^{n} \sum_{i=1}^{n} |current - previous|,$$

where n is the block size (16 or 8 depending on the implementation) and *current* and *previous* are the pixel intensities in the current and previous frames respectively.

(a) (b)

Figure 1: *FSBM (a) & VSBM (b) motion vector structure for frame 5 of "Stefan".*

Figure 1(a) shows an example of how the FSBM algorithm operates in practice using the MPEG-4 video test sequence "Stefan" (Class C). In this example (frame 5) it is clear that although large portions of the background have the same motion vector[2], the nature of FSBM does not allow for a way of describing this in the encoded video stream.

2.2 Variable Size Block-Matching

FSBM, although simple to implement, is limited by the trade-off between block size (which dictates the number of motion vectors) and prediction quality. Furthermore, it cannot make the step of choosing one motion vector for a group of blocks that undergo the same motion.

Proposals have been made to improve FSBM by using variable size block-matching

[2]Motion vectors are denoted by line segments starting from the middle of each block. They indicate displacement in half-pixel steps relative to the last frame.

(VSBM). Chan, Yu and Constantinides [1] describe a scheme in which relatively large blocks are repeatedly divided until a locally minimum error or the maximum number of blocks are obtained. Using such a top-down method may generate block structures that match real motion in an image. However, it seems that an approach which directly seeks areas of uniform motion might be more appropriate.

We have described a bottom-up quad-tree based VSBM algorithm [5] [6] with a computational efficiency comparable to FSBM and yet providing better prediction quality.

In a similar manner to FSBM, the algorithm partitions the current frame into $M \times N$ blocks using, however, a much smaller block size (4×4 pixels). Then for each block it creates a bit-set representing all motion vectors whose Mean Absolute Error)[3] (MAE is below a predefined threshold using the following algorithm:

```
clear intersection set
set vector_bit to 0
for y in [-rad, rad) in inc pixel steps
  for x in [-rad, rad) in inc pixel steps
    set error to SAD(current, previous, offset by (x, y))
    if error / (4 x 4) < threshold
      add vector_bit to intersection set
    increment vector_bit
```

where `inc` can be 0.5 or 1 for half or full pixel search respectively. For a search radius of 2 pixels the motion vector bit-set might look like:

```
     -2 -1  0 +1 +2

 -2   0  0  0  1  0
 -1   0  0  0  1  0
  0   0  0  1  0  0   =   0 00100001 00010000 00001000
 +1   0  0  0  0  0
 +2   0  1  0  0  0
```

indicating that the motion vectors $\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} -1 \\ -2 \end{bmatrix}$ are below the threshold. This bit-set can be packed in 25 bits (it is equivalent to `0x00211008` using a C `long`). Using such a representation, it is straightforward to implement all set operations very efficiently using standard bit operations.

Having calculated the bit-sets for all 4×4 blocks they are then merged four at time in a quad-tree fashion as long as their bit-set intersection is not empty using the following recursive algorithm[4]:

---

[3] $MAE = \frac{SAD}{4 \times 4}$

[4] The root size is $2^n \times 2^n$ such that: $(n > 0) \wedge (2^{n-1} < S) \wedge (2^n \geq S)$, where S is $\mathbf{max}(M \times 4, N \times 4)$.

```
if node not nil                    # Termination condition
  merge node.top-left
  merge node.top-right
  merge node.bottom-right
  merge node.bottom-left

  if node.top-left is leaf and       # All children must be leaves
     node.top-right is leaf and
     node.bottom-right is leaf and
     node.bottom-left is leaf
    set intersection to INTERSECT(node.top-left.intersection,
                                  node.top-right.intersection,
                                  node.bottom-right.intersection,
                                  node.bottom-left.intersection)
    if intersection not empty
      set node.intersection to intersection
      discard node children          # Turn node into a leaf
```

Having performed all possible merging, the best motion vector for each of the variable size blocks is calculated. If the intersection set is empty the same algorithm as FSBM is used. Otherwise we need only check motion vectors that are members of the intersection set thus reducing the search time.

In practice, all SADs are calculated in advance and stored in memory, to avoid unnecessary re-computation. Furthermore, the SAD of a merged block is equal to the sum of the SADs of its children.

Figure 1(b) shows an example of how the VSBM algorithm (using the same number of blocks as FSBM) identifies areas of uniform motion and merges them into one block, thus leaving a large portion of the block budget for areas with more complex motion.

3 Profiling

Profiling both FSBM and VSBM using the MPEG-4 video test sequence "Container Ship"[5] (Class A) with a 16 pixel search radius in $\frac{1}{2}$ pixel increments, produced the following results:

FSBM - 94.5% of time spent on block-matching 16×16 blocks.
Average time per frame is $41.5s$ on a 50MHz Sparc.

VSBM - 57.6% of time spent on block-matching 4×4 blocks.
21.4% on deciding which vectors should be included in the intersection set.
9.5% on block-matching final merged variable size blocks
Average time per frame is $94.0s$ on a 50MHz Sparc.

[5] Tests used the central 256×256 portion of the CIF sequence (Y component only). The CIF size sequence (352×288) was obtained from the full size YUV sequence using the MPEG-4 down-sampling software (ISO/IEC JTC1/SC29/WG11/M1552).

Although FSBM and VSBM block-matching should be directly comparable, the VSBM block-matching component ($54.1s$) is approximately 38% more time consuming than FSBM ($39.2s$) due to the greater number of iterations (64×64 block-matching operations compared to 16×16 for FSBM with a 256×256 image size).

With the addition of the intersection algorithm to VSBM, plus the fact that it is executed 8 times (on average) in order to calculate the correct threshold, VSBM is on average 2.3 times slower than FSBM.

4 Parallelisation

Three parallel "platforms" were considered:

- *Homogeneous workstation network* - using PVM.
- *Parsytec Super-Cluster* - MIMD transputer (T800) based parallel computer.
- *Shared memory multi-processor computer* - using a thread programming model.

The trivial parallelisation method of distributing frames among processors is very well suited to PVM at the expense of high latencies. Finer grain parallelisation (as described in section 4.1), although better suited to real codec implementations, carries overheads[6] that make PVM implementations impractical.

While the FSBM algorithm is easily parallelisable and implementable on the *Super-Cluster* either with PVM or through the native C compiler constructs, an efficient implementation of it's VSBM counterpart is very memory intensive as it requires keeping all block-matching results in memory until the correct threshold is calculated; an option clearly infeasible since each processing node's memory is limited to 4MB (minimum average requirement being 32MB for a 16 pixel search radius in half-pixel steps). A less memory intensive implementation is possible (re-calculating block-matching information as needed) but is on average eight times slower than FSBM. Since the *Super-Cluster* computing node processing power is limited when compared to, for example, a 50MHz Sparc CPU[7], such and implementation would render the 128 processor *Super-Cluster* equivalent to, or slower than a 4 CPU Sparc 20 workstation.

With shared memory multi-processor machines becoming common, it was decided to concentrate on the shared memory implementation using the POSIX conformant threads programming model (`pthreads`) provided by the *Solaris* 5.5.x operating system.

[6] Setup and communication costs are too high for fine grain parallelism.

[7] A T800 (stack-based), will on average need 3 times as many instructions to perform the same task as a 3 address Sparc CPU. Taking into account the clock speed difference, a 30MHz T800 can be as much as 5 times slower than a 50MHz Sparc.

4.1 Block Matching (FSBM & VSBM)

Profiling indicates that the block-matching step amounts to 94.5% of the FSBM algorithm complexity and 57.6% of the VSBM algorithm.

Parallelisation is straightforward with four obvious solutions:

1. Fine grain -

```
for each block
  for each search position
    new thread
      block-match
```

2. Medium grain -

```
for each block
  new thread
    for each search position
      block-match
```

3. Coarse grain -

```
for 1 to nblocks / MAX_THREADS
  for 1 to MAX_THREADS
    new thread
      for each search position
        block-match
```

4. Very coarse grain -

```
for each processor
  new thread
    for nblocks / nprocs
      for each search position
        block-match
```

In the first three cases we rely on the operating system scheduler to distribute threads amongst processors, while in the last case the distribution becomes part of the algorithm. For a 256×256 image with 4×4 blocks (typical VSBM example), the numbers of active threads required at any one time are:

1. $4096 \times 64 \times 64 = 16777216$ threads!
2. 4096 threads.
3. `MAX_THREADS` (optimal depends on the number of processors).
4. `nprocs`.

Clearly (1), although ideally suited to hardware implementations, is infeasible on a shared memory computer (16 million threads is more than the system imposed limit of 4096[8]).

Case (2) is a option but an implementation and test run on a 4 CPU machine indicates that scheduling 4096 threads is expensive in terms of scheduler overheads.

Case (3), tested with a full 300 frame sequence on a 4 CPU machine (`MAX_THREADS` = 16) achieves 85% CPU utilisation for FSBM and 70% for VSBM.

Case (4) will guarantee the highest CPU utilisation so long as the thread life-time is significantly larger than (3). This implies that the thread will need to block-match multiple blocks in it's life-time making the programming model slightly more complex due to the need for some form of communication (either through message passing or shared memory).

It was decided to implement case (3).

4.2 VSBM overheads

In addition to block-matching, a significant portion of the VSBM algorithm is spent on generating the vector bit-sets and performing the block merge.

21.3% of the VSBM algorithm is spent visiting each of the initial (4×4) blocks before any merging, deciding which motion vectors pass the threshold and can be used in the merging process. In the 256×256 image example this translates to 4096 such operations which can be dealt with in parallel.

A further 9.5% of the algorithm is spent visiting each block after the intersection based merge (the merge itself, only takes 0.75% of the total algorithm time), deciding on the best motion vector for the merged block. The number of blocks will be either the same as FSBM (256 in our example) or less if we are matching the error [7] (the average depending on the sequence motion).

Both these parts, accounting for a further 30.8% of the VSBM algorithm, are operations performed on each leaf of the VSBM quad-tree structure.

The quad-tree leaf traversal in itself only constitutes 0.26% of the traversal problem (0.08% of the total algorithm time). Hence, executing the quad-tree leaf operations in parallel will parallelise a further 30.5% of the VSBM algorithm.

[8] This is a *Solaris* 5.5.1 hard limit; other OSs may vary.

5 Results

As we are only parallelising a certain portion of the problem, Amdahl's law applies, i.e. the speedup is limited by the sequential portion of the problem:

$$S = \frac{1}{F + \frac{1-F}{N}}, \tag{1}$$

where S is the speedup, F is the percentage of the sequential portion of the problem and N is the number of processors. Assuming no overheads the theoretical limit to the problem can be calculated:

$$S_{FSBM} = \frac{1}{0.0547 + \frac{0.9453}{N}} \quad (\lim_{N\to\infty} S_{FSBM} = 18.3). \tag{2}$$

$$S_{VSBM} = \frac{1}{0.1171 + \frac{0.8829}{N}} \quad (\lim_{N\to\infty} S_{VSBM} = 8.54). \tag{3}$$

These are only absolute maximum bounds. In practice, overheads are going to exist so the maximum speedup will be less than the maximum predicted. This overhead comes from "packaging" the partitioned problem, executing additional system code that will create and synchronise the required processes and finally scheduling these processes.

The chosen parallelisation method relies on the thread scheduler for load balancing. This method assumes the existence of enough threads active at any one time to achieve a high CPU utilisation, but not so many that most of the time is spent in the scheduler.

For the example used in profiling (41.5s per FSBM frame and 94.0s per VSBM frame), using a 4 CPU Sparc 20 ($N = 4$ in equations (2) and (3)) we can expect speedups less than:

- 3.43 for FSBM ($\equiv$ 12.1s per frame),
- 2.96 for VSBM ($\equiv$ 31.8s per frame).

As expected, the algorithm time (user time in figure 2) remains fairly constant for both FSBM and VSBM. System time however, which mostly takes the form of

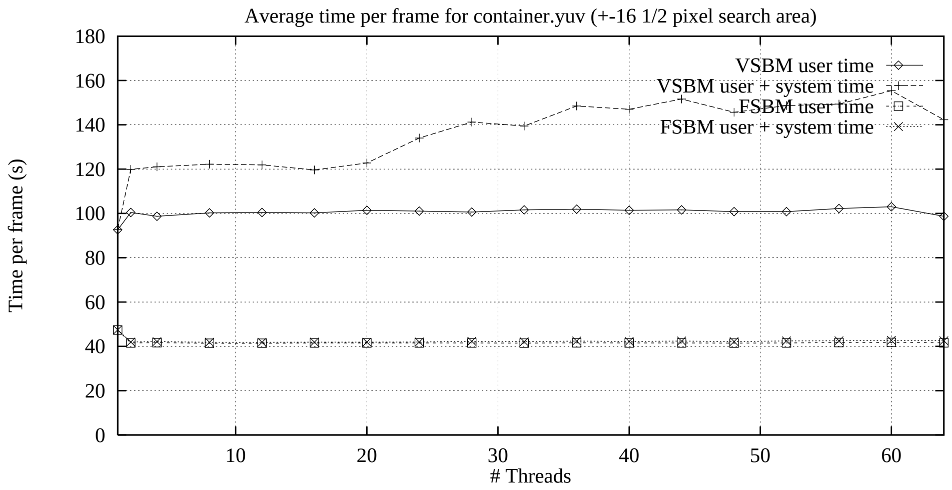

Figure 2: *FSBM & VSBM times for varying number of threads on 4 CPUs.*

scheduling, increases with the number of threads, the effect being more prominent with VSBM[9].

Tests with varying numbers of threads on a 4 processor Sparc 20 indicate that 16 threads (figure 3) achieve the highest speedup[10].

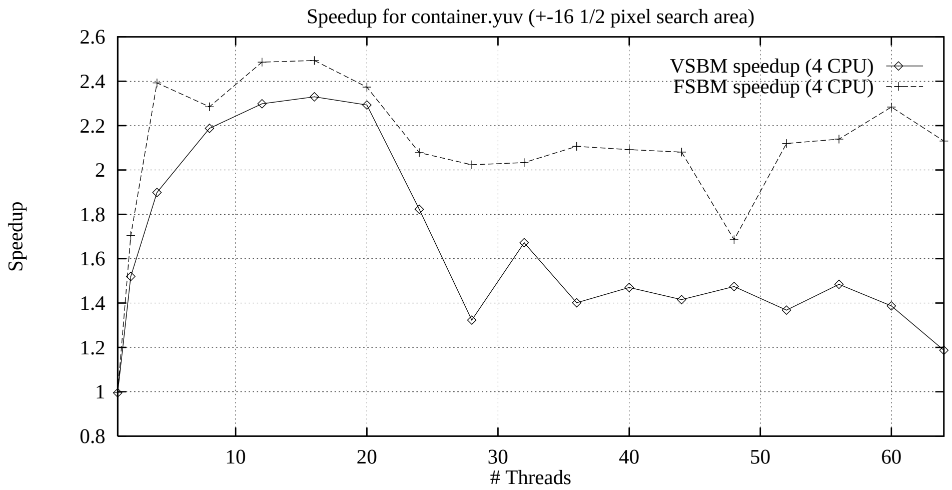

Figure 3: *FSBM & VSBM speedups for varying number of threads on 4 CPUs.*

Timing both FSBM and VSBM for the above configuration with the first 50 frames

[9]The life-time of a VSBM thread is 16 times less than that of FSBM (4×4 size blocks as opposed to 16×16).

[10]It also appears that 16 threads achieve the best CPU utilisation; possibly an artifact of the way the thread scheduler operates.

of "Container Ship" (central 256×256 of Y component of CIF sequence), produces the following speedups:

- 2.49 for FSBM ($\equiv 16.6s$ per frame),
- 2.33 for VSBM ($\equiv 43.0s$ per frame).

The deviations from the maximum (27% for FSBM and 21% for VSBM) are an indication of system overhead and a lower than expected CPU utilisation. It appears that the *Solaris* thread scheduler will not necessarily schedule a new thread on a different processor; even if there is one available[11]. In our tests, for most cases, a high CPU utilisation was not achieved until after the first 10 frames.

6 Improvements

Having identified what appears to be a limitation of the *Solaris* thread scheduler when faced with multiple short-lived threads, it seems that using method (4) is the only way to guarantee high CPU utilisation. Using as many threads as there are processors (each thread bound to a light-weight process) and keeping each thread active for the duration of the algorithm will ensure the best performance at the expense of a more complex programming model. The small number of threads will require manual load balancing in order to achieve the best possible CPU utilisation.

In the case of FSBM, block-matching is the dominant part of the algorithm. The remaining 5.5% of the measured time is spent on I/O and other setting-up tasks that cannot easily be improved on.

VSBM on the other hand has two distinct parts. Although the initial block-matching, similar to FSBM, cannot be improved, one would think that the threshold calculation/intersection based merge leaves room for improvement.

Ideally the threshold should be predicted in one step. Previously we described such a predictor that produces, on average, the same number of blocks as FSBM [8]. Using matched error [7] should provide a good, prediction/control strategy with enough flexibility to keep the average number of blocks better than FSBM, while at the same time removing the iterative element from the threshold calculation process. A one step threshold prediction method would reduce the sequential VSBM average time per frame from $94.0s$ to $69.2s$[12] on a single 50MHz Sparc and the parallel version theoretical maximum from $31.8s$ to $20.0s$[13] on four of the above (i.e. a 4 CPU Sparc 20) making VSBM only 20% slower than FSBM.

[11] It is possible to change the contention scope of a thread to "global" at the expense of it being scheduled by the OS and not the threads scheduler.

[12] $94 \times (1 - 0.576 - 0.308) + \frac{94 \times 0.308}{8}$, assuming an average of 8 iterations per frame for threshold calculation.

[13] $\frac{1}{S + \frac{1-S}{4}}$, where $S = \frac{94 \times 0.308}{8 \times 69.2}$

7 Conclusion

We have shown through algorithm profiling that both our bottom-up variable sized block matching motion compensation algorithm and the popular but less sophisticated fixed size block matching approach can be greatly improved in performance through parallelisation.

The parallelised versions of the algorithms were evaluated using the MPEG-4 test sequence "Container Ship". Using a light-weight threads programming model, we have achieved a 249% speedup for FSBM and a 233% speedup for VSBM over their sequential versions on a 4 CPU Sparc 20, results which are only 27% and 21% less than the theoretical maxima for FSBM and VSBM respectively.

Acknowledgments

The authors are pleased to acknowledge the support for this work from the UK Engineering and Physical Sciences Research Council, Grant No. GR/J79997.

References

[1] M.H. Chan, Y.B. Yu, and A.G. Constantinides. Variable size block matching motion compensation with applications to video coding. *IEE Proccedings*, 137(4):205–212, August 1990.

[2] MPEG Video Group. Mpeg-4 video verification model version 6.0. In *ISO/IEC/JTC1/SC29/WG11/MPEG96/N1582*, Seville, February 1997.

[3] ITU-T Recommendation H.263. Video coding for low bit rate communication. Geneva, March 1996.

[4] J.R. Jain and A.K. Jain. Displacement measurement and its applications in intraframe image coding. *IEEE Trans. Commun.*, COM-29:1799–1808, 1981.

[5] G.R. Martin, R.A. Packwood, and I. Rhee. Variable size block matching motion estimation. In *International Organisation for Standardisation ISO/IEC JTC1/SC29/WG11 Document MPEG95/0334*, pages 324–333, Dallas meeting, November 1995.

[6] G.R. Martin, R.A. Packwood, and I. Rhee. Variable size block matching motion estimation with minimal error. In *SPIE Proccedings of Digital Video Compression: Algorithms and Technologies 1996*, volume 2668, pages 324–333, San Jose, USA, February 1996.

[7] G.R. Martin, R.A. Packwood, and M.K. Steliaros. Reduced entropy motion compensation using variable sized blocks. In *SPIE Proceedings of Visual Communications and Image Processing*, volume 3024, pages 293–302, San Jose, USA, February 1997.

[8] M.K. Steliaros, R.A. Packwood, and G.R. Martin. Adaptive block matching motion compensation for low bit-rate video coding. In *Proceedings of 1st Advanced Digital Video Compression Engineering Proceedings*, pages 81–88, Cambridge, UK, July 1996.