**Architectural Modeling for Parametric Characterization of Parallel Algorithms**

**Tak Wong**

**University of Massachusetts, Lowell**

**7/4/2010**

**Part III Parametric Estimation of Parallel Processing Speedup**

*Abstract: Gene Amdahl’s 1967 paper on large scale computing* **[1]** *asserted that the effort for parallelizing computation was bounded by the serial overhead that could not be improved by the same magnitude as the parallel part that took advantage of performance gain with extra processors. This has been widely interpreted as the limit of parallel computing, regardless of the number of processors being used. Gustafson refuted the claim with several experimental cases that showed that large-scale speedup independent of the serial part could be achieved, if the size of the problem was allowed to grow* **[2]***. In reality, both arguments are very “rough-grained” estimates with specific, and perhaps unrealistic assumptions on the characteristics of the problems that had led to the respective conclusions. It has been pointed out that both these laws of estimate might be misapplied in most cases. This paper shows both Amdahl’s and Gustafson’s estimates are fundamentally equivalent, and the conflicting observations were the result of different parametric conditions in the processing of the algorithms. Moreover, Amdahl’s Law can be extended by additional predictive parameters and then can be used as a practical tool for complexity analysis and characterization of generic parallel algorithms, thus leading to a more pragmatic approach in the construction of the architectural platform.*

Introduction

Gene Amdahl’s approach to estimating the performance of parallel processors, as derived from his short classic paper **[1]**, is an approximation of the speedup when a single task is parallelized to run on multiple processors. The representation of the speedup is based on the proportional percentages of the serial part and parallel part of the algorithm for the task. The so-called “Amdahl’s Law” has been generally interpreted and summarized as the following equation:

Speedup = 3.1

whereby s + p is normalized to 1, therefore, p = 1 – s.

Thus, Amdahl’s speedup estimate is a ratio between a single processor system and a system with n processors that can be applied on the parallelized part of the algorithm. Amdahl noted that, when n 🡪 ∞, the speedup approached . That is to say the upper limit of speedup is bounded by the cost of the serial part regardless of the number of processors that can be deployed to run in parallel - this was basically the conclusion in Amdahl’s argument. Traditionally, there has always been some confusion about the true meaning of this statement. For one, in most cases where Amdahl’s Law is used, the serial part is assumed to be a constant cost associated with program setup, management, and other “housekeeping”[[1]](#footnote-1) duties, which should not increase as the number of processors scales up. Moreover, Amdahl’s Law assumed a fixed-size problem but allowed the number of processors to vary. This latter issue led to the development of Gustafson’s paper **[2]**, which argued that the speedup might not necessarily be bounded by the same serial percentage s, if the problem size m is allowed to grow. Gustafson claimed that while the problem size grew infinitely, the serial part, being constant, will become an insignificant percentage, and the speedup would approach the idealistic n, the number of processors used in parallel. Both of these issues are somewhat related and contribute to the inaccurate estimation of parallel processing speedup. First, it is unrealistic to assume that the sequential processing part of an algorithm would never grow as the number of tasks running in parallel increases. It should not be difficult to see, in general, the increase in parallel tasks will entail additional overheads in serial synchronization and communication among the parallelized tasks. Secondly, as the size of the problem and number of processors scale up, the algorithm itself might require adjustment and modification to take advantage of the additional resources. Potentially, a solution with different complexities might be applied each time the problem size changes. Neither Amdahl’s nor Gustafson’s estimation addressed the potential growth of overheads or the change in the algorithm to take advantage of the system architecture.

Amdahl vs Gustafson

Whereas Amdahl warned that optimizing the serial part was just as essential as the parallel part, Gustafson pointed out the serial part consisting of setting up the tasks and I/O bottlenecks would not grow with the problem size.[[2]](#footnote-2) In Gustafson’s experiments, there were applications that achieved close to n times speedup when the system scaled to as many processors. Gustafson’s Law, however, is a little counter-intuitive in the expression of the speedup:

Let m be the size of a task, and let the time to complete the task in a system with n processors be

Then the time required for a single-processor system to complete the same task is

The speedup on the n-processor system, accordingly, is

=

when s(m) + p(m) is normalized to 1.

Gustafson’s claim was that as m 🡪 ∞, the constant percentage serial part s(m) 🡪 0. Then the speedup 🡪 n. There is hardly anything new here in Gustafson’s representation. Indeed, in Amdahl’s Law, if s is allowed to become very small:

= = n , if s 🡪 0 3.2

By defining and relating the scaled and non-scaled serial parts of Amdahl and Gustafson’s respective equations, Yan Shi has pointed out that these two estimates are actually equivalent[[3]](#footnote-3). The following is a simpler and more practical way to demonstrate the equivalence: According to Gustafson’s logic, if an n-processor system completes the parallel part in p time, when the same parallel part will require np time in a single-processor system. Then the n-processor machine will spend time to complete the parallel part that takes p time in a single-processor:

=

Then

= =

Add the serial part s(m) to get the total time for completing the entire job in an n-processor machine:

To get the speed up ratio, compared that to the time Gustafson believed would require to complete with 1 processor:

=

Then

Speedup = = 3.3

which is exactly Amdahl’s Law, with the problem size as a parameter. The essential difference between the two formulas is that Amdahl normalized the time required for a single processor to 1, whereas Gustafson normalized the time for n processors to 1.

So, these were actually the same estimation but with two different interpretations. The conflicting claims arose from the view and treatment of the serial part s(m). Amdahl was concerned that s(m) will be the upper limit of speedup, if the serial part cannot achieve the same magnitude of optimization as the parallel part in a solution with n processors. On the other hand, Gustafson’s experiments were scientific applications that made use of the enhanced features of the parallel architecture, which kept the serial part a constant in those specific cases. Therefore, there are characteristic conditions in the nature of the problem, the algorithm, and the processor architecture that will determine the bounds of the speedup. To put it simply, Amdahl’s limit is in effect, if the algorithm is processed such that

🡪 = , as n 🡪 ∞

Gustafson’s formula, on the other hand, indicates that the speedup 🡪 n if this condition is met:

🡪 0 , as m 🡪 ∞

Remember both Amdahl and Gustafson held that the serial part s(m) was a constant. Then it is clear that, in Amdahl’s equation, if the problem size m is allowed to grow, the ratio will render s(m) a very small percentage of the total and

🡪 n , as m 🡪 ∞.

Again, the two formulas seem to be reconcilable if Amdahl’s Law allows for the growth of the problem size, also. However, the crucial issue here is the assumption that the serial processing will remain constant regardless how big the problem is and how many processors are used.

In general, multiple active processors would incur synchronization and communication overheads which must be handled sequentially. Moreover, this type of sequential processing will potentially grow as the number of processors scales up. The following example demonstrates the effect of serial overheads with a simple and common algorithm:

Example 1 - M\_Loops:

for (i = 1; i <= m; i++)

{

array[i] = i;

sum += array[i];

}

Assuming there are m processors available, this can be broken down into m parallel tasks, as in Example 1.1:

Example 1.1 – M\_Tasks:

{

array[1] = 1;

lock(sum);

sum +=array[1];

unlock(sum);

}

{

array[2] = 2;

lock(sum);

sum += array[2];

unlock(sum);

}

etc.

The parallelization of the algorithm comes at a cost: namely, the additional lock/unlock instructions for exclusive access, and perhaps internally the extra time to get to the shared memory. In the worst case, when the tasks always collide when accessing the sum variable, then the synchronization cost is in the order of O(m). Then as problem size m 🡪 ∞, the cost of synchronization/communication also approaches ∞. Even if the average simultaneous access of the common variable is assumed to be (1 + m)/2, it shows a continuing upward growth of the synchronization overheads when multiple processors are accessing shared data.

The above performance can be improved by breaking down the loop to n parallel tasks, where n is a “reasonable” fixed value < ∞, assuming m > n:

Example 1.2 – N\_Tasks:

for (i = 1; i <= m/n; i++)

{

array[i] = i;

lock(sum);

sum += array[i];

unlock(sum);

}

for (i = m/n; i <= 2m/n; i++)

{

array[i] = i;

lock(sum);

sum += array[i];

unlock(sum);

}

etc

In the N\_Tasks example, the overhead on synchronization is bounded by O(n) – at most n tasks accessing the same data simultaneously. The serial part that must set up the lock mechanism can be considered as a constant, as Gustafson’s Law proposed. Even though the worst-case cost of synchronization is reduced by having fewer processors, the frequency of access is increased in the order of m/n. Thus the overall serial processing is still getting bigger, not disappearing as in Gustafson’s experiments when the size of the problem grows. In the above example, because of the frequency of loops, even a very small number of parallel tasks may contribute to the latency in accessing shared resources, causing synchronization “saturation” as the algorithm potentially has to block on wait many times because of simultaneous access of shared data. However, in the worst-case when all access of shared data collide, the max wait time is O(n), where n < m, even though the total synchronization incidents are the same as the M\_Loop algorithm. Then no task has to wait longer than O(n) time, an improvement over M\_Loop. It should be clear that, having many more processors may not always be desirable, if collision and synchronization in accessing shared data become an issue. Still, the code in Example 1.2 can be modified to add up all the i values locally, and then write to the shared data sum only at the end of the loop:

temp\_sum = 0;

for (i = 1; i <= m/n; i++)

{

array[i] = i;

temp\_sum += array[i];

if (i == m/n)

{

lock(sum);

sum += temp\_sum;

unlock(sum);

}

}

This will fix the synchronization cost independent of the growth of m, but, clearly, this is no longer an algorithm with the same complexity as Example 1.1.

The above conclusion essentially raises the following counter-arguments to Gustafson’s assertion:

1. The infinite growths in problem size and processor number do not necessarily guarantee optimal speedup.
2. The massive scaling of the problem and computing architecture might necessitate a change in the strategy of the algorithm, which potentially may lead to a different proportion of the serial and parallel parts.
3. The potential synchronization and communication overheads caused by multiple processors is additional serial processing. Thus the serial part cannot always be assumed to be constant.

Algorithm Analysis Using Speedup Estimations

Despite the weaknesses and simplicity in these estimation formulas, Amdahl’s and Gustafson’s Laws can be used for the purpose of system-wide characterization of the structure and strategy of parallel processing based on a desired speedup. For example: On a single-processor multi-tasking system, the total cost of operation on this serial platform is c = s + p, with s and p a percentage of c and s, p ≥ 0. In order to achieve a desired speedup by a factor of r > 1 without significant modification of the original algorithm, the number of processors used can be represented by the following equation:

then

x = 3.4

Alternatively, if the number of processors n is fixed, the serial part s needs to be reduced:

3.5

where > ,

Then

x = 3.6

The actual amount of reduction for the serial part is then

s - = c - -

=

=

= 3.7

The above indicates the serial part requires a reduction of in order to achieve a speedup of factor r in the total cost. Validating the result with Amdahl’s Law:

Speedup =

To improve the performance further by a factor of r, by utilizing n processors, subtract the expected reduction, , from the serial part:

=

=

But c = s +

Then

Speedup =

So, indeed, if features in the architecture exist that can reduce the cost of the serial part by ,then the overall speedup is a factor of r by reduction of the serial part alone, with a fixed number of processors. Thus the usage of Amdahl and Gustafson estimates above can help determine the type of system architecture and algorithm required for a desired speedup.

However, these formulas can only be applied with strong assumptions and constraints. Amdahl’s Law requires the parallel part to be evenly distributed among the n processors. Gustafson’s formula assumes that all n processors will be occupied by and running the same algorithm. These assumptions are acceptable for estimating a system-wide speedup, where the algorithms for the tasks are invariant, and the cost of the serial part remains constant, despite the number of processors in the system configuration might vary. In general, it is neither adequate nor accurate to use the same formula to characterize an algorithm or architecture that might behave very differently when more processors are added.

Discrepancies in Applying The Estimation Formulas

It is also noteworthy that, in both Amdahl’s and Gustafson’s studies, the fundamental assumption was, the same parallel algorithm was being measured for all processor configurations, and the speedup measure was only based on the variation of number of processors. This approach has led to the much confusion on applying the respective formulas, since when an algorithm is parallelized, in most cases, there would be variations in the processing steps as well. Furthermore, if the two laws provide the same estimates, as shown in 3.3, how could Gustafson come up with such a different prospect of parallel computing than that of Amdahl? The answer actually lies in the properties of the hidden overheads that are not represented in either formula. Gustafson’s Law assumed there were no hidden overheads in communication and synchronization as the problem size m and processor number n grew. This was also the case with Gustafson’s own experiments, which he used as counter-arguments to Amdahl’s assertion that the general speedup was limited by the serial part.

Let be the synchronization overheads for n processes trying to access shared resources. Asymptotically, Gustafson’s estimate can be modified to include this overhead cost. If the time to complete a job with an n-processor machine is

= 1

where implicitly includes the overheads .

Then the time required to complete the same job with a single-processor machine without synchronization overheads is

3.8

Similarly, Amdhal’s estimation formula should be expressed as

Speedup = 3.9

It is clear that is an over-estimation based on the worst-case scenario. Moreover, the above assumes all overheads are the same for all the different tasks running on different processors. Both the Amdahl and Gustafson estimates assume the behaviors of all processors are also homogeneous and the access of shared data uniform, practically limiting the usage of the estimation formulas to only symmetric multiprocessor architecture and uniform memory access. Nonetheless, by including the potential overheads incurred by synchronizing multiple tasks, the speedup ratio in principle is more accurate than the original estimate.

There are still fundamental questions on how the serial and parallel percentages can be derived without some reasonable level of understanding of the algorithm and the architecture it is running on. As Shi lamented, the ones applying Amdahl and Gustafson’s Laws seem to be at a loss in deriving the serial percentage to use for the equations. Indeed, the serial percentage cannot be known a priori, without fully understanding the dependencies on the algorithm and the underlying architecture. In general, when more processors become available, the original algorithm in most cases will need to be revised. Because of that, an estimation based on the unchanged algorithm with more processors does not provide the true potential speedup. Since a parallel algorithm is usually designed closely with the features of the architecture, an algorithm cannot be fixed when the number of processors is a variant, as the Amdahl and Gustafson formulas were applied traditionally.

Parametric Extensions

To encompass the variation in processor properties and algorithm variations, Gustafson’s formula is further refined as

3.10

where

is the constant serial part despite a varying size m. The parameter m

is included for compatibility with Gustafson’s original formula

is the sum of a vector representing the individual time of each processor required to complete the total job of size m on a single-processor machine.[[4]](#footnote-4)

is the sum of a vector representing the individual overheads of each parallelized task = ie. = total overheads incurred by the different parallel part running on n independent processors for a problem of size m.

If a problem is sub-divided into x tasks, not necessarily running the same algorithm, and is the work done by task i, the total cost of the parallel part is

when the tasks are processed in parallel with n processors. Note that it is not necessary that x = n, but x ≥ n should hold. Each pi(n) represents the work done by task i when n processors are available in the system. Thus, the problem size m has been spread out into x different tasks, each of which might process instructions at a different speed. If n = 1,

then

and that is the cost of running the entire algorithm serially.

In practice, the amount of work done can be related to each individual processor rather than each task, so P(m, n) can always be taken as

3.11

Each pi now represents the actual work done by the processor i, which may encompass the cost of actually running multiple tasks, when the number of tasks exceeds the number of processors available.

In general, it cannot be assumed that all parallel tasks will be starting at the same time, either. Initially, some tasks might be idle, depending on events or other tasks to complete before starting to run. Therefore, the time for the completion of all the tasks do not reflect the actual work done – ie when a processor is idle, if is considered no work being done. For simplicity, the idling overheads will be absorbed into the synchronization overheads – ie some tasks have to wait for other tasks to get into some state. Then, the actual time for completing all the parallel tasks when they are run parallel should be the time required for running the longest task, idle time included. That is

Algorithm Completion Time = MAX()

In addition, each task will incur synchronization overheads that might not have a uniform cost. Then, the worst-case overheads cannot be expressed as a simple product nv, as shown in 3.10, but as a sum based on the different potential cost for each individual processor:

where is the overheads cost for processor i. However, this is still not a practical value

for the estimation, as it is unlikely that all the tasks would be accessing the same

shared resources simultaneously all the time. It would be more pragmatic and reasonable to use a statistical model for the average occurring overheads based on the probability of simultaneous access by different tasks, since each process generally is running in parallel, and thus independently until the shared access occurs. Indeed, the overheads resulting from access collision is a random variable. Therefore, the probability of each occurrence of access shared data for each task, and the probability of all their combined collisions have to be taken into account and calculated. As such, the random variable that is the expected value of the combinations of different tasks colliding in shared data access will serve this purpose.

Let

where P is the set of tasks running on n different processors, and |Zr| = r, where

0 < r ≤ n. Define

Let v(zi) be the cost of synchronization overhead for process zi. Then the expected cost for , is

3.12

Where Pr(zi) = probability of the overheads occurring for zi  while the process is running.

The possible different combinations of r tasks synchronizing at the same time is

The sets of Zr with r members are enumerated as follows:

Zr1, Zr2, …, Zr(n!/r!(n – r)!) 3.13

The sum of expected overhead cost for all combinations of r tasks:

*∑*

r=1

The expected cost for all the possible combinations from 1 to r tasks accessing at the same time is then

n

*∑∑* *≤*  3.14

r=1 j= 1

When the probability of all the overhead events are equal, ie with n tasks, then

Pr(zi) =

and

Pr(Zr) =

If all costs of synchronization overheads are also equal to v, then

E(Zr1) = E(Zr2) = E(Zr3) = … =

and the average expected overheads for n processors is then

n n

∑∑ = ∑ 3.15

r=1 j= 1 r = 1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Gustafson Convergence and Amdahl Limit

Apply the Gustafson formula with the addition of overheads vector sum:

n n

1 = s(m) + ∑pi - ∑

i = 1 r = 1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

If the overhead costs and probability of these occurring are known a priori, it can determined whether the parallel speedup advantage will continue to scale as n and m become large by checking if

🡪 0, as m 🡪 ∞ 3.16

Expanding with equations 3.18 and 3.13:

n

∑∑

r=1 j= 1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 🡪 0, as m 🡪∞ 3.17

This is the convergence that Gustafson perceived in his original study that resulted in an “absolute speedup” from using n processors without additional overhead. If the speed of all the processors and the overheads are equal, then it just becomes

🡪 0 , as m 🡪 ∞

Where the absolute speedup is n, as shown by Gustafson’s experiments.

It follows that when

🡪 d, 0 < d ≤ 1 as m 🡪 ∞

Then

V(m,n) 🡪 dP(m,n) as m 🡪 ∞

s(m) + P(m,n) – V(m,n) = P(m,n) – dP(m,n), as m 🡪 ∞

The potential speedup is then

(1 – d)P(m,n) as m 🡪 ∞

Then d is said to be the degree of Gustafson Convergence that affords a (1 – d) fraction of the absolute speedup with n processors.

Also, note that when

≥ 1

* V(m, n) ≥ , as n 🡪 ∞ 3.18

In Amdahl’s original study, he was concerned that, despite the addition of parallel processors, the speedup is bounded by a constant serial cost, if the architecture could not support reduction of the serial percentage. In the extended equation 3.18, the overall serial overheads are a varying percentage. Another way of interpreting 3,18 is the amount of overheads required will be equal to Gustafson’s ideal speedup, if the above condition is shown to be true. Then the speedup can be said to have reached the “Amdahl Limit” of serial overheads – that the increase in computing power is limited by the overall serial processing cost that grows with the number of processors. This is expanded to

n n!/r!(n – r)!

s(m) + ∑∑ E(Zrj)

r=1 j= 1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ≥ 1, as n 🡪 ∞

n n n!/r!(n – r)!

(s(m) + ∑pi) - ∑∑ E(Zrj)

i = 1 r=1 j= 1

n n!/r!(n – r)! n

∑∑ E(Zrj) ≥ ∑pi , m, n 🡪 ∞ 3.19

r =1 j = 1 i = 1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

As a addendum, if the ratio of the overheads versus the actual gain with multiple process is

≥ 1

* V(m, n) ≥ , as n 🡪 ∞ 3.20

That is to say, ultimately, as much or more overhead cost is required to achieve as much gain, the advantage of multi-processing might be neutralized.

Even when the Gustafson Convergence cannot be achieved, ideally, the serial cost should be maintained or improved as the problem grows. The percentage of the original s(m) can be maintained or improved, if the ratio of serial overheads versus overall time can be maintained at

≤ s(m) , m, n 🡪 ∞

V(m, n) ≤ s(m)(s(m) + P(m, n) – 1)

Since

s(m)(P(m, n) – 1) ≤ s(m)(s(m) + P(m, n) – 1) , for 0 < s(m) < 1

then

V(m, n) ≤ s(m)(P(m, n) – 1) 3.21

If the processor synchronization overheads can be maintained as shown in equation 3.21, the overall serial percentage will not increase as the size of the problem and the processors scale up. The speedup is limited by a steady percentage, but not a constant time.

Re-Writing Amdahl’s Law

As stated earlier, there should not be any assumption that the multiple parallel tasks are distributed evenly amongst the processors. Thus let

P(m. n) = + + …, +

Where pi is the actual work done in terms of time spent by processor i. Then

MAX()

determines the time requires for all the parallel parts to complete. Then, the extended Amdahl’s Estimate is

1

Speedup = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 3.22

n n!/r!(n – r)!

s(m) + MAX4(p1, p2, … pn)- ∑∑ E(Zrj)

r=1 j= 1

\_\_\_\_\_\_\_\_\_\_\_\_

**Part IV The Diagonal MBRP Algorithm**

Introduction

The Wave-Front MBRP (Macro Block Region Partition) algorithm developed by Sun, Wang, and Chen[[5]](#footnote-5) serves to demonstrate a practical and effective approach in solving a generic, large-scale problem with parallel computing. It is not considered a conventional scientific computing problem that has specific properties of substructure partition and self-similarity that lends itself naturally to a divide-and-conquer methodology. In fact, this algorithm which breaks down a large MPEG data block into equal-size smaller partitions is not easily parallelized because of the process ordering dependencies between the small blocks. As Meenderinck et al pointed out, some blocks have to wait for information from previously processed blocks to become available before getting processed [5]. However, H.264 encoding is a straightforward, but computation-intensive process that can benefit from having multiple tasks working on the smaller partitions at the same time. Given these properties, H.264 encoding is an ideal prototype for modeling a solution in multi-computing because of these key properties:

* the problem is large and requires heavy computation
* the natural solution is direct and simple, but time-consuming
* the problem does not lend itself to partitioning by substructures
* the sub-problems may not be processed independently and require

synchronization and communication

The last condition above entails the parallel algorithm to deal with significant serial overheads – ie. the synchronization and communication that must be processed sequentially, which normally is the main challenge in parallel computing.

As such, a number of algorithms in parallelizing H.264 encoding have been developed (See references in [4]). The Macro Block Region Partition presents a “wave-front” technology where the subdivisions of the problem are processed in a strict schedule assigned to different processors, each responsible for working on a sub-problem. The scheduling is necessary because of the inter-dependence of the sub-problems. The authors have pointed out the different complexities in the encoding makes it very difficult to distribute the functional workload uniformly to different processors, thus making standard task pipe-lining infeasible. However, the wave-front methodology shows a sort of assembly-line process because each sub-problem may have temporal and spatial dependencies on other sub-problems.

The MRBP algorithm can be summarized as follows:

1. Subdivide an H.264 frame evenly and uniformly into smaller, proportional macro blocks
2. Organize the macro blocks in row/column format
3. To encode each macro block, the blocks to its left, top, and top-right must already have been processed, unless such blocks do not exist (eg. a block in the leftmost column of the frame has no block to the left)

Technically, the current macro block being processed is the motion vector (MV), and the three blocks it depends on are called predicted motion vectors (PMV). In order to encode the macro block correctly and efficiently, the differences between the PMV and the MV must be computed. The diagram below displays the wave-front technique in progress.

Note:

process started

process completed

|  |  |  |  |
| --- | --- | --- | --- |
| (1,1) | (1,2) | (1,3) | (1,4) |
| (2.1) | (2,2) | (2,3) | (2,4 |
| (3,1) | (3,2) | (3,3) | (3,4) |
| (4,1) | (4,2) | (4,3) | (4,4) |

Blocks (1,1), (1,2), (1,3) completed

Block (1,4) started

Blocks (2,1), completed

Block (2,2) started

Block (3,1) must wait for (2,2) to complete

before it can start

Block (4,1) must wait for (3,2) to complete

before it can start

Figure 1

The Wave-Front Process

Since the size of the sub-divisions can be arbitrary – the minimal macro block requires 4 x 4 pixels for H.256, the solution for this problem can will assume there is always an unused processor available to run the parallel algorithm for a macro block regardless how many macro blocks are in the frame. However, because of the above-mentioned inter-dependence, the processors cannot be all started at the same time. In the beginning, some processors must wait for some others to finish their tasks. Moreover, towards completion of the big frame, some processors will be freed up and become idle. To utilize all the available processors efficiently, the process should start as soon as possible as a macro block is ready to be processed (ie. its PMV’s are already processed), and the processors should be freed up quickly so they can be used for the next big frame. In other words, if a large frame is sub-divided into r \* c macro blocks, even if r \* c processors are available, this algorithm will not be able to utilize all r \* c processors at the same time. In fact, fewer than r \* c processors will ever be active simultaneously throughout the process.

Applying the straight MBRP algorithm on a (5 x 5) frame, the following demonstrates the waiting pattern:

Time line: t0 t1 t2 t3 t4 t5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Stage 1

The first block starts processing at t0. 1 processor is active between t0 and t1.

[ Note: tx denotes the completion time of xth macro block. It is assumed that the time required for processing each block is uniform, since the processor is running the same algorithm on data of the same size. ]

t0 t1 t2 t3 t4 t5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Stage 2

In Stage 2 above, the first block on the top row has been processed at t1, and the second block has started. Still 1 processor is active. When it completes at t2,

the first block in row 2 can be started in Stage 3, because its top and top-right block have finished. Now there are 2 processors active at the same time.

t0 t1 t2 t3 t4 t5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Stage 3

In Stage 4, still 2 processors are active simultaneously:

t0 t1 t2 t3 t4 t5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Stage 4

Stage 5: As the last macro block on the first row is being completed between t4 and t5, the first 2 blocks on row 2 have been completed. Now a third processor can be started to process the first block on row 3:

t0 t1 t2 t3 t4 t5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Stage 5

Stage 6: At t5 when the first row has been completed, it is back to 2 processors being utilized at the same time.

t0 t1 t2 t3 t4 t5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Stage 6

Stage 7: At t6 when the second row on the last block, a third processor is started for the first block on the fourth row:

t0 t1 t2 t3 t4 t5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Stage 7

Stage 8: At t7 the second row is completed. Only 2 processors can be active at the same time.

t0 t1 t2 t3 t4 t5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Stage 8

Stage 9: At t8, a third processor starts processing the first block of the fifth row:

t0 t1 t2 t3 t4 t5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Stage 9

Continuing the procedure, following pattern of processor usage emerges:

Time Processors Active

t0 1

t1 1

t2 2

t3 2

t4 3

t5 2

t6 3

t7 2

t8 3

t9 2

t10 2

t11 1

t12 1

Table I

Analysis of the Algorithm

In general, when the MPEG4 frame is sub-divided into r x c (row, column) macro blocks, the time required to complete all the macros blocks is 2r + c - 2, given there are as many processors needed available. Here is the proof :

Given an MPEG4 frame sub-divided into r x c macro blocks, the time required to complete the top row is c. Each row below starts only after the top and top-right blocks above are completed. Therefore, each row below is always 2 block-time behind the row above. The last row to finish, ie. the rth row, will be 2(r – 1) blocks behind the top row, because the last row cannot start its first block until the r – 1 rows above it all have completed at least 2 blocks. If r <= c, by the time the top row is done, the additional time needed to complete the last row is

2(r – 1). Therefore, the total time required will be c + 2(r – 1) = 2r + c - 2.

If r > c, the rth row is started after 2(r – 1) block-time, and it will take c additional time to complete the nth row. Thus the total time required is also

2r + c – 2 4.1

Here is the generalization on processor usage with a (r x c)-block frame:

Considering a separate processor is activated for each macro block and deactivated when the macro block processing completes, there are in total r x c processors being activated in the processing of the entire frame. The average processor usage is then

1 if r or c = 1

{ 4.2

(r \* c)/(2r + c – 2) otherwise

Essentially, the algorithm assigns a single process for a macro block in each row that is ready for processing, and on each row, there can only be 1 macro block being processed at a time. Therefore, the maximum number of processors being active simultaneously is related to the maximum number of rows being processed. For the trivial cases where r or c = 1, the maximum processor usage is 1. For all other cases r, c > 1, since each macro block can only be processed when the row above it has completed processing the top and top-right blocks, when the first macro block in a row has started processing, the maximum number of rows above it that have macro blocks still under process cannot exceed (c – 1)/2. So the peak process usage is

1, if r or c = 1

{ **|** ((c - 1)/2) **|** + 1, if r ≥ **|** ((c – 1)/2) **|** + 1 4.3

r, if r <  **|** ((c – 1)/2) **|** + 1

Speedup Analysis

Equation 4.1 above shows the cost of running the wave-front algorithm for an r x c subdivided frame with at least the peak number of processors available (equation 4.3).

Compare that with the cost if the subdivided macro blocks are run in a single processor, ie. r \* c, the cost ratio is

r \* c

2r + c – 2 4.4

For a valid cost improvement, it is necessary that r > 1, and c > 2.

However, the above is derived from a conventional cost analysis that is directly related to the asymptotic growth of the problem but does not provide insight into the serial overheads and the effect of the multiprocessors. A more precise speedup ratio can be obtained through the extended Amdahl’s estimate. Take the example of the (5 x 5) frame, and apply the extended Amdahl equation from 3.11. Since at each different stage of the process, a different number of processors might be active, the speedup for any stage that has x processors running in parallel can expressed as

Speedup = 1

s(m) + (1 – s(m))/x + xv

The serial overhead in the algorithm is the notification of completion at the end of processing each macro block, so that some other macro block depending on the information can start processing. This is a deterministic and ordered overhead, so it is taken as the sum from x processors, rather than a statistical expected value as showed in 3.18. This is also the only serial overhead in the process. The above is simplified to

Speedup = 1

1/x + xv

Table I shows there are 4 stages that use 1 processor only (1, 2, 12, 13), 6 stages that had 2 processors running in parallel (3 , 4, 6, 8, 10, 11), and 3 stages with 3 processors running in parallel (5 , 7, 9).

The expected speedup of any single stage of the entire process with the x processors running in parallel is

1

13(1/x + xv)

Therefore, the speedup for the (5 x 5) frame example is

4 stages with 1 processor 6 stages with 2 processors 3 stages with 3 processors

4 + 6 + 3

13(1 + v) 13( ½ + 2v) 13(1/3 + 3v)

Clearly, whether there is a valid gain in speedup depends on the value of the overhead v – that the above sum in speedup exceeds 1. Hypothetically, if the overhead v is 0, then

4 + 12 + 9 = 25 13 13

which is exactly the same cost ratio for a (5 x 5) frame, if the equation 4.4 is applied.

With the overhead v > 0, the speedup will be neutralized when

4( ½ + 2v) (1/3 + 3v) + 6(1 + v) (1/3 + 3v) + 3(1 + v) ( ½ + 2v) = 1 4.5

13(1 + v)( ½ + 2v)(1/3 + 3v)

🡪 288v2 + 217v + 26 = 1

13(36v3 + 49v2 + 14v + 1)

🡪 468v3 + 637v2 + 182v + 13 = 288v2 + 217v + 26

🡪 468v3 + 349v2 - 35v -13 = 0

Solving the 3rd degree equation:

🡪 v ≈ 0.21 4.6

The above result indicates that the serial overhead must be kept approximately below

1/5 of the macro block processing cost in order to have a positive gain of performance.

The Wave-Front MBRP Algorithm requires each row to have a 2-block lead time on the next following row. If this lead time can be reduced, the number of blocks being processed in parallel and the peak usage of processors will be improved. In the following section, a modified version of the MBRP Algorithm that tasks the blocks diagonally instead of processing horizontally by the row will be discussed, and it will show a significant improvement not only in the utilization of the parallel processors, but also the containment of synchronization overheads. The proof that this algorithm actually works is found in Appendix A.

The Diagonal-Front MBRP Algorithm

The Diagonal process is depicted graphically in figure 2 below:

t0 t1/2 t1 t3/2 t2 t5/2 t3 t7/2 t4 t9/2

direction

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Basically, the diagonal macro blocks of a large frame are picked and processed from left to right along diagonal lines until all the macro blocks are consumed. Observe the following properties of the algorithm:

1. Every macro block starts from the top-left corner and ends at the bottom-right corner.
2. A previously inactive processor is activated when a macro block is processed.
3. One or more macro blocks, as long as they lie on the diagonal line as shown above, can be started at the same time.
4. When the upper half of a macro block M is completed, the adjacent blocks to the right and to the bottom, if they exist, can be started, and when the adjacent blocks are half-way completed, the entire block M is completed.

.

The breakdown of the time sequence is according to the diagonals running across each macro block. When only half a macro block is processed, as shown at t1/2, it is assumed that half the time has expired compared to a full block being processed.

Note that, from t0 to t1/2, there is 1 processor active. Starting at t1/2, the blocks to the right and below of the current block can start processing, since there is enough information from the adjacent Predictive Motion Vectors for computing. In general, there can be as many processors active simultaneously as the macro blocks that are traversed by the diagonal line at any point in time. The pattern is as follows:

Time Processors Active

t0 1

t1/2 3

t1 5

t3/2 7

t2 9

t5/2 9

t3 7

t7/2 5

t4 3

t9/2 1

t5 0

Table II

Compared to the Wave-Front MBRP algorithm, the Diagonal algorithm takes half the time to run through (assuming there are as many processors as needed) and utilizes more processors at any one time and frees them more quickly. Here is the proof for the special case of an (r x r) matrix:

Given an (r x r) sub-division of an MPEG4 frame, assuming the time to process

1 macro block diagonally is equivalent to processing it laterally, as in the original

MBRP algorithm, the time used by the processor assigned to process all the

diagonal blocks is r, because there are only r diagonal blocks in an (r x r) matrix. If there are as many processor available as needed, while the diagonal blocks are being processed, other processors can be utilized to work on other blocks that are ready. By the time the diagonal blocks are done, all the other blocks are also completed. So, the total time required is r.

Observe the macro blocks on the first row, and note that the diagonal that cuts the nth block into 2 equal halves cuts through exactly r blocks diagonally. This can be proven fairly easily using induction. At the point where the diagonal cuts through r blocks, at most r processors are active. Now, between the diagonal the cuts through r – 1 blocks and the diagonal that cuts through r blocks, there can be as many as r + r – 1 processors active simultaneously, because there are as many blocks that can be processed. Also note that the 2r – 1 processors are active for the second half of the previous r – 1 blocks, and the first half of the current r blocks. The leftmost block on the top row is processed by 1 processor for the first half, and when the second half is processing, 2 other processors are started to work on the block to the right and the block below. As mentioned above, this will be consistent all through with the second half of all the blocks. Putting it all together, with an (r x r) matrix, the total usage of processors for processing the top leftmost half of MPEG4 frame is described by

½ \* 1 + ½ \* 3 + ½ \* 5 + … + ½ \* (2r – 1)

Since the bottom rightmost half is symmetric, the total utilization of the processors is

1 + 3 + 5 + … + 2r – 1 = r2  (from summation formula)

Assuming it takes 1 uniform unit of time to process 1 macro block, the average utilization of the processors is

r2/r = r

From equation 4.2, for c > 1, the average utilization of the processors is ≤ r/2. So, the modified, diagonal processing on the average utilizes more than 2 times the processors. Moreover, the total time for completion for an (r x r) frame is

r vs 3r – 2 4.7

This improves the performance by almost an order of 3 for a square matrix.

Proof for the general case for an (r x c) frame:

Given an r x c sub-division of an MPEG4 frame, assuming the time to process

1 macro block diagonally is equivalent to processing it laterally, as in the original

MBRP algorithm, the longest diagonal is defined by the smaller dimension:

min(r, c). Note that the diagonal blocks cannot be processed in parallel because

of the dependence of adjacent blocks. The time used by the processor assigned to

process the longest diagonal is min(r, c). If the sub-divided blocks are processed diagonally from left to right, when a block is half way completed, the adjacent blocks (to the right and below) can be started in parallel. So, the adjacent blocks will always be ½ time behind the diagonal block. If r ≠ c, then there will be c – r extra adjacent blocks at the end, either to the right or below the last diagonal block. By time min(r, c), the last diagonal block is completed, and ½ of the next adjacent is also completed. This adjacent block will take an additional ½ time to complete, and ½ of the next adjacent block is also completed. So, if there remain |r – c| adjacent blocks, it will take an additional ½ (| r – c|) time to complete each of the adjacent blocks to the diagonal that cannot be processed in parallel. Then, the total time required is min(r, c) + ½ (| r – c |). If r = c, then the total time is r.

As is with the straight MBRP algorithm, the diagonal-MBRP algorithm also applies a single processor for processing each macro block. Therefore, the total processor usage is still r \* c.

Assuming it also takes 1 uniform unit of time to process 1 macro block diagonally, the average utilization of the processors is

(r \* c) / (min(r, c) + ½ (| r – c |))

If r = c, then we have r2/r = r, an average utilization of r processors per

time t.

if r ≠ c, then the average utilization of the processors actually grows,

since (r \* c) / min(r, c) = max(r, c), for all integers r,c > 0.

Therefore, (r \* c) / (min(r, c) + ½ (| r – c |)) > max(r, c).

According to the properties of the diagonal algorithm, the maximum number of

macro blocks that can be started simultaneously is related to the maximum number of blocks in a diagonal, which is min(r, c). As min(r, c) macro blocks are processed half way, another min(r, c) macro blocks can be started. Therefore, the peak processor utilization is ≤ 2 \* min(r, c).

For r, c > 1, The total time for completing the entire frame with the original MBRP algorithm is 2r + c - 2 ≥ 2 \* min(r, c). From the following ratio

min(r, c) + ½ (| r – c |) / (2r + c – 2)

for r = c, the diagonal algorithm improves the performance by an order of at least 2. But when min(r, c) + ½ (| r – c |) ≥ (2r + c – 2), the diagonal algorithm loses the advantage. When

min(r, c) + ½ (| r – c |) = (2r + c – 2) > 2 \* min(r, c), for r, c > 1

min(r, c) + ½ (| r – c |) > 2 \* min(r, c)

then

| r – c | > 2 \* min(r, c) 4.8

So, the above equation must be maintained for the diagonal algorithm to be more efficient than the straight MBRP.

Comparing the processor utilization:

max(r, c) / [(r \* c)/(2r + c - 1)] => max(r, c) / [max(r, c)/2]

So, the modified, diagonal processing on the average utilizes more than 2 times the processors.

Most significantly, there is an alignment of the processor utilization with the dimension of the matrix in the diagonal process. However, that comes at a price – that at the peak of processor utilization, 2 \* min(r, c) processors may be required to run at the same time. With the original MBRP, at most lower((c – 1)/2) + 1 processors are required for an (r x r) matrix. The diagonal algorithm requires about 4 times the number of processors for peak usage.

The average processor utilizations of the two algorithms are

(r \* c) / (2r + c – 2) Wave-Front MBRP

(r \* c) / min(r, c) + ½ (| r – c |) Diagonal MBRP

The diagonal algorithm is more efficient in processor utilization as long as

min(r, c) + ½ (| r – c |) < 2 \* min(r, c) ≤ 2r + c - 2

🡺 | r – c | < 2 \* min(r, c)

Serial Overheads Constraint with the Diagonal Algorithm

Clearly, when a frame is sub-divided into a square matrix with (r x r) macro blocks, where r > 1, the Diagonal MBRP algorithm achieves a nearly 3 times speedup advantage over the straight MBRP algorithm. The speedup is even more significant than the single processor solution – a reduction factor by the square root of the cost (n vs n2). But how about the serial overheads cost? Again, the only overheads for processing in paralell are those for the macro blocks to notify other tasks that they can start processing macro blocks that are ready. However, unlike the straight MBRP algorithm, the diagonal algorithm is required to send notification twice – once at the half-way point after completion of the upper diagonal half of the block, and then another at the completion of the bottom half of the block. Taking the information from Table II on the diagonal processing of the (5 x 5) frame example, and applying the extended Amdahl equation again, the expect total cost including communication overheads is shown below. Note that the overhead value is doubled because of 2 notifications for each macro block.

2 stages each with 1, 3, 5, 7, 9 processors

2 + 2 + 2 + 2 + 2

10(1 + 2v) 10(1/3 + 6v) 10(1/5 + 10v) 10(1/7 + 14v) 10(1/9 + 18v)

When v = 0, the overall speedup = 5, which is the exact ratio of r2/r when compared to the single processor algorithm.

The speedup is neutralized when v > 0 and the above sum equals 1. Equation 4.4 showed that the overhead cost v should be kept under 1/5 for a positive gain in speedup. Applying v = 1/5 to the above fractions:

* 0.97

The above indicates that even when the overheads cost is doubled for each macro block processing, the threshold constraint for the overheads is almost the same as that for the straight MBRP algorithm. Furthermore, the algorithm can be adjusted to group notifications for all the macro blocks that are processed in parallel, since their completion coincide exactly with the diagonal-front line. In that case, there will be uniformly 1 notification overhead at each stage. The above formula is re-written as

2 + 2 + 2 + 2 + 2

10(1 + v) 10(1/3 + v) 10(1/5 + v) 10(1/7 + v) 10(1/9 + v)

Putting in the threshold cost v = 1/5:

* 1.89

which indicates a positive speedup in the Diagonal processing.

Appendix A - The Diagonal MBRP Algorithm

Meenderinck et al [5] explained the “Wave-Front” process and the inter-dependences between related blocks. The same logic can be applied on the principle of the Diagonal MBRP Algorithm.

Proof A1:

To show that the algorithm actually works because of the inter-dependent properties, examine the relationship between a block in process and its PMV (Predicted Motion Vectors):

A macro block x is ready for processing, if and only if the PMV blocks for x have already been processed, as shown in the table below, where the PMV blocks are marked

a, b, and c.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  | b | c |  |  |  |  |  |
|  |  | a | x |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

Figure A1

With the diagonal algorithm, however, the left upper corner of x is ready for processing, if and only if both blocks a and b have processed past their respective upper halves:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  | b | c |  |  |  |  |  |
|  |  | a | x |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

Figure A2

This can be shown graphically by magnifying the inter-dependent blocks:

|  |  |  |
| --- | --- | --- |
|  | b | c |
| a  **y** | x |  |

Figure A3

Clearly, the part of macro block x that is marked y is ready for processing, because y’s “micro”-PMV blocks have already been done. That implies macro block x can start processing when there is just enough information from the adjacent blocks a, b, and, c available, while the 3 PMV blocks have not been completed, yet. This is the most significant difference between the Wave-Front Algorithm and the Diagonal MBRP Algorithm, and it is the reason that the latter can utilize more processors and run in shorter time. As the diagonal process continues, all of x will be processed under the same condition.

There is an obvious tradeoff in communication: In the Wave-Front MBRP algorithm, each PMV block completes processing and communicates all the required information all at once to the MV block. In the Diagonal-MBRP, there is a continuous stream of small amount of information being sent to the MV block. The total amount of information from the PMV blocks can be assumed to be the same for the Wave-Front and Diagonal cases. Then if the communication cost is linear in respect to the amount of data, then the D-MBRP algorithm has an advantage of load-balancing and scheduling the processors more efficiently. However, if transferring data in large blocks is more efficient than the accumulative small amounts of data, the constant communication can be an extra overhead.

Claim: The Diagonal MBRP Algorithm is optimal for processing the (r x r) square frame in terms of cost in time and utilization of processors.

Proof A2:

Refer to the diagram below:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Figure A4

Note that all the direct diagonal blocks cannot be processed simultaneously, because each diagonal block except the first relies on some minimal information from at least the block above and the block to the left. Except for the very first block, which has no dependences, in order for any amount of information to be available from the PMV blocks for a diagonal block, the previous diagonal block has to be completely processed first. In other words, the n diagonal blocks have to be processed sequentially. Therefore, in order to process completely n diagonal blocks, at least n time units are required. In order to complete an MPEG4 frame that is sub-divided into (r x r) macro blocks, the n diagonal blocks must be processed, and that will take at least n time units already. Therefore, the optimal time for processing the (r x r) matrix is r.

In Part IV of the paper, the proof shows Diagonal MBRP Algorithm to complete an

(r x r) frame in time r.

As shown before, the average utilization of processors over time for the diagonal-MBRP method is r. Now, suppose there is another method that has utilization of r + 1 or better and spends the same time r for completing the entire MPEG4 frame, then, the total processor usage for the more efficient algorithm is at least r(r + 1) = r2 + r. Assuming there are unlimited number of processors available, and a different processor is used for each different macro block, then for an (r x r) matrix, there are at most r2 processors that are necessary for processing as many macro blocks. It is not possible to utilize more than r2 processors in total to complete the entire (r x r) frame.

**References**

1. Amdahl, G.M. *Validity of the single-processor approach to achieving large scale*

*computing capabilities*.In *AFIPS Conference Proceedings* vol. 30 (Atlantic City, N.J.,

Apr. 18–20). AFIPS Press, Reston, Va.,1967, pp. 483–485

2. Gustafson, John L. *Communications of the ACM May 1088 Volume 31 Number 5,*

*1988. p p532-533*

3. Shi, Yuan *Re-evaluating Amdahl’s Law and Gustafson’s Law*

<http://www.cis.temple.edu/~shi/docs/amdahl/amdahl.html>

4. Sun, S., Wang, D., and Chen, S. A Highly Efficient Parallel Algorithm for H.264

Encoder Based on Macro-Block Region Partition Computer School, National

University of Defense Technology, Changsha, China

5. Cor Meenderinck, Arnaldo Azevedo, Mauricio Alvarez, Ben Juurlink, and

Alex Ramirez *Parallel Scalability of H.264*

1. Amdahl, G.M. Validity of the single-processor approach to achieving large scale computing capabilities.In *AFIPS Conference Proceedings* vol. 30 (Atlantic City, N.J., Apr. 18–20). AFIPS Press, Reston, Va.,1967, pp. 483–485 [↑](#footnote-ref-1)
2. John L. Gustafson, *Communications of the ACM May 1088 Volume 31 Number 5, 1988. p533* [↑](#footnote-ref-2)
3. Shi, Yuan *Re-evaluating Amdahl’s Law and Gustafson’s Law*

   <http://www.cis.temple.edu/~shi/docs/amdahl/amdahl.html>, pp 3-5 [↑](#footnote-ref-3)
4. This is relative to the normalized time of s(m) + p(m) = 1 for the time the job is completed with n processors working together. [↑](#footnote-ref-4)
5. A Highly Efficient Parallel Algorithm for H.264 Encoder Based on Macro-Block Region Partition

   Shuwei Sun, Dong Wang, and Shuming Chen Computer School, National University of Defense

   Technology, Changsha, China [↑](#footnote-ref-5)