An improvement of OpenMP pipeline parallelism with the BatchQueue algorithm

Thomas Preud'homme, Julien Sopena, Gaël Thomas and Bertil Folliot
LIP6
Paris, France
Email: firstname.last-name@lip6.fr

***Abstract*—In the context of multicore programming, pipeline parallelism is a solution to easily transform a sequential program into a parallel one without requiring a whole rewriting of the code. The OpenMP stream-computing extension presented by Pop and Cohen proposes an extension of OpenMP to handle pipeline parallelism. However, their communication algorithm relies on multiple producer multiple consumer queues, while pipelined application mostly deals with linear chains of communication, i.e., with only a single producer and a single producer.**

To improve the communication performance of the OpenMP stream-extension, we propose to use, when it is possible, a more specialized single producer single consumer communication algorithm called BatchQueue. Our evaluation shows that BatchQueue is then able to improve the throughput by up to 30% for real applications and by up to 200% for an example application which is fully parallelizable communication intensive micro benchmark. Our study shows therefore that using specialized and efficient communication algorithms can have a significant impact on the overall performance of pipelined applications.

I. Introduction

Since several years, the frequency of processors stagnates because of physical constraints. Chip makers now take advantage of the growing number of transistors per square inch to increase the number of cores on a die [1]. However, many applications are still single-threaded or are parallel but only scale to a limited number of cores. In such a case, pipeline parallelism appears like an attractive solution to exploit multicore since slight modifications to existing code suffice.

Pipeline parallelism consists of splitting the sequential processing of data in a stream into several stages forming a pipeline, each stage being assigned to a different core. Hence, data need to flow from one core to another to be entirely processed. Several consecutive data can then be partially processed by different stages at the same time, resulting in parallelism. Pipeline parallelism offers several advantages over other techniques. First, it preserves the processing of data sequential which means that the algorithm doing the processing does not need to be rewritten and that data dependencies remain respected. Second, it hides memory latency for long streams of data since the total time will mostly be a function of the throughput rather than the latency. Third, pipeline parallelism limits its use of memory bandwidth by favoring on-die communication, which enables it to coexist well with other applications.

The main limit of scalability of pipeline parallelism is in the inter-core communication time. Indeed, given (i) a sequential application whose data processing time is T_{seq}, and (ii) a communication time T_{comm}, the stage processing time after parallelization on n cores can at best be $\frac{T_{seq}}{n} + T_{comm}$. Thus, a bigger number of cores means that the sequential task is split in more stages, with each stage executing for a smaller amount of time. For a given algorithm, the communication time is fixed; therefore the communication overhead increases when more cores are used. It is thus essential to have a fast communication algorithm to keep the overhead low.

In the domain of parallelism on shared memory systems, OpenMP [2] (Open Multi-Processing) stands out as a reference. Its wide adoption is the result of three advantages: OpenMP (i) is a cross-platform API, (ii) is integrated in compilers and (iii) is easy to use. While the current release of OpenMP only proposes data and task parallelism, an extension proposing pipeline parallelism exists [3], which offers good performance.

This paper presents how a significant improvement of this extension is obtained by replacing its native communication algorithm by a more specialized one. The improvement stems from the fact that the native communication algorithm supports communication between multiple producers and multiple consumers while communication between a single producer and a single consumer is the most common scenario in pipeline parallelism. We propose in this paper to replace, whenever it is possible, the native communication algorithm by a faster single producer single consumer queue called BatchQueue which we proposed in a previous article [4]. The work described in this article contains the following contributions: (i) an Open Source integration [5] of BatchQueue inside OpenMP stream-computing extension and (ii) an extensive evaluation with real applications showing an improvement of performance up to 30%.

The remaining of this article is laid out in the following way. Section II presents in more details the technique of pipeline parallelism and how to use it. Section III then details both the communication algorithm used in OpenMP stream-computing extension and the BatchQueue algorithm. Section IV describes of all the tests we performed and the performance improvements we obtained. Section V compares BatchQueue to related work and section VI concludes

this article.

II. Background

Compared to the more common parallelism techniques that are data and task parallelism, pipeline parallelism provides an interesting alternative. This technique can parallelize algorithms with strong dependencies between data, while the others cannot. However, pipeline parallelism comes with its own limitations with regard to scalability. Hence, before introducing our improvements in the next section, this section starts by presenting what pipeline parallelism consists of and what are the limitations it suffers from. The section also presents an extension to the OpenMP API to seamlessly parallelize an application through pipeline parallelism.

A. Principle of pipeline parallelism

Like data parallelism, pipeline parallelism acts on loops. However, data parallelism can only deal with loops whose iterations process independent data while pipeline parallelism can deal with loops whose successive iterations process data dependent on each others. Pipeline parallelism consists of splitting the sequential processing of data in each iteration of the loop into several "stages" forming a pipeline. The stages are then assigned to different cores. Each stage consists of a partial processing of data and the entire processing is done by making the data flows from one core to another in order to go through all the stages. In other words, each stage takes as input the output of the previous stage. Several consecutive data can be partially processed by different stages at the same time, that is how parallelism appears while still preserving the order in which data are processed. The resulting dataflow is depicted in fig 1.

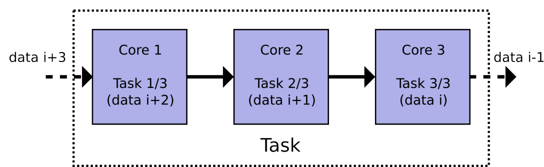

Figure 1: Flow of data in pipeline parallelism

For parallelizing an algorithm which processes data with dependencies between them, pipeline parallelism appears as an obvious choice to improve its efficiency. Yet, when it comes to scalability with the number of cores, this technique suffers from a limitation: the throughput speedup depends on the communication overhead.

Several data are partially processed together in different stages. When this partial processing is over, the data migrates to the next stage. For the data on the last core, it means its processing is finished and it exits the pipeline. Given a communication time between two cores T_{comm} and an application whose total time to process one data is T_{seq}, a data exits the pipeline every $\frac{T_{seq}}{n} + T_{comm}$ seconds. Increasing the number of cores only reduce the stage time, that is $\frac{T_{seq}}{n}$. The throughput is thus limited to one data every T_{comm} seconds. Another consequence is that the throughput is improved significantly when increasing the number of cores as long as $T_{comm} << \frac{T_{seq}}{n}$. It is thus essential to have a communication algorithm as fast as possible. Figure 2 shows how the speedup vary for several value of the ratio $\frac{T_{comm}}{T_{seq}}$.

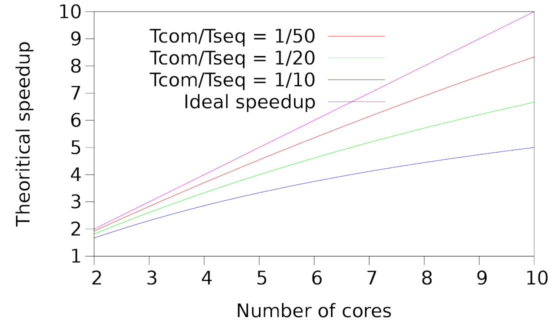

Figure 2: Influence of communication throughput on pipeline parallelism speedup

B. OpenMP stream-computing extension

Despite the limitations seen above, pipeline parallelism can improve the performance of a number of algorithms, especially those dealing with a stream of data. Audio and video processing is such an example where each data is processed through a set of filters chained together. However, parallelizing an algorithm can require a significant amount of work: threads must be created, some data are to be made shareable and an important amount of synchronization must be added to the logic of the algorithm. Doing these modifications is error-prone and it adds some complexity to the algorithm which in turn may lead to more bugs when the algorithm is modified.

For all these reasons, there are tools to help converting an application to use parallelism with as little effort as necessary. These tools serve two purposes: providing an abstraction layer to handle hardware heterogeneity, and hiding the low level machinery of thread management and synchronization. The goal is achieved by providing a runtime library with an high level API. The API can be a set of headers but can also take the form of new keywords or annotations to use in the code. For data and task parallelism, there are numerous tools available: OpenMP[2], Threading Building Blocks[6], Cilk Plus[7], Intel Array Building Block[8] to cite some of them. For pipeline parallelism, an extension of OpenMP has been proposed by Pop and Cohen [3] which allows a programmer to explicit dependencies between data.

OpenMP is a de facto standard in parallelism on shared memory systems. The reason is that OpenMP provides some

noticeable advantages over competing solutions: it is multi-platform, integrated in the compilers and relatively easy to use. As a matter of consequence, improving algorithm inside OpenMP has a much bigger impact than improving competing solutions. The same goes for the stream-computing extension of OpenMP as well, and hence the interest expressed in this paper to improve pipeline parallelism as is proposed by this extension.

III. Efficient inter-core communication

As expressed in the previous section, performance of inter-core communication plays vital role in scalability of pipeline parallelism. For this reason, authors of OpenMP stream-computing extension took care of the efficiency of the communication algorithm used in the extension. Nevertheless, some aspects can be improved, notably taking more consideration of the memory coherency protocol. This section presents in the first part the algorithm used in OpenMP stream-computing extension and highlight some of its shortcomings. In the second part, the section presents the BatchQueue algorithm used to improve performance of OpenMP stream-computing extension.

A. Native OpenMP stream-computing extension algorithm

In multicore and multiprocessor systems, MOESI[1] protocol [9] ensures coherency between all the caches. However, two issues arise due to this coherency. First, although MOESI protocol aims at minimizing communication needed to achieve coherency, it cannot be avoided. Indeed, communication is needed in two cases: (i) a modified cache line needs to be invalidated in other caches, (ii) an invalid cache line needs to be updated from another cache or memory. The second issue occurs when two unrelated data items, which lie on the same cache line, are updated. This issue is known as false sharing.

The original OpenMP stream-computing extension algorithm, called *native* algorithm hereafter, tries to address both these issues. The native algorithm is a MPMC queue – Multiple Producers Multiple Consumers queue – which makes the communication issue more difficult to address. Indeed, competition for shared variables also happens between producers themselves and between consumers themselves.

The main principle employed in the algorithm to tackle the communication issue is to cache the values of shared data whenever possible. For efficiency, producers and consumers are not directly connected to the queue. Instead, producers and consumers are connected to structures representing the set of all producers, and respectively consumers. This structure layout allows for different levels of caching, in the same way as processors have several levels of cache. In practice, this means data for which a lot of contention happens are cached once in the structure representing the set of producers (resp. consumers) and once in the producers (resp. consumers) themselves.

For eleminating false sharing, the solution consists of assigning some variables an entire cache line to avoid conflicts. However the communication buffer itself can suffer from false sharing. Participants – producers and/or consumers – can work concurrently on different areas of the buffer fitting in the same cache line. Though, this scenario does not occur in the native algorithm because the amount of data sent together through the queue is a multiple of 32 times the size of the base element. Since elements are usually greater than one byte, producers and consumers work on different cache line all the time.

From the above it is clear that the native algorithm is designed to efficiently handle inter-core communication. It tackles both issues related to memory coherency on multi-core systems: excessive communication and false sharing. However, this algorithm handle complex scenarios where multiple producers communicate with multiple consumers. As such, some amount of synchronization is required between the producers and between the consumers on the variables they share. When communication happens between one producer and one consumer, these synchronizations are useless and hurt the throughput. Since this scenario is the most frequent one in pipeline parallelism, an opportunity is missed to optimize communication in this case.

B. BatchQueue

Contrary to the native algorithm, our algorithm BatchQueue is a SPSC queue – Single Producer Single Consumer queue. This property makes it solve more efficiently the two memory coherency issues discussed above, that is minimizing communication between cores and avoiding false sharing. BatchQueue's algorithm addresses these two issues in three different ways:

- by reducing the number of shared variable;
- by sending an amount of data which is a multiple of the cache line size;
- by segregating producer and consumer in different cache lines.

The first two propositions address the communication issue. Reducing the number of shared variables reduces the need for coherency and thus communication while sending several data at the same time implies some factorization of the synchronization. The last proposition, on the other hand, addresses the false sharing issue by preventing consumer's cache from being invalidated every time the producer produces a data.

The algorithm we propose is presented in functions produce and consume. The principle is to have a communication buffer divided in two parts, called semi-buffers, whose size is a multiple of the cache line size. This allows for the production and consumption to be done from different buffers, thus avoiding false sharing. The semi-buffers are

[1]MOESI stands for Modified Owned Exclusive Shared Invalid which correspond to the 5 states this protocol has.

read from and written to at one go. When the semi-buffers are totally processed, a synchronization happens to exchange them. The consumer starts reading the data from the semi-buffer that the producer just filled and the sender can write in the buffer that the consumer already read entirely.

$tab[ind_{prod}] \longleftarrow data$;
$ind_{prod} \longleftarrow$ next entry ;
if ind_{prod} = *Start of cache line* **then**
  Wait *status* = **false** ;
  status $\longleftarrow$ **true** ;
end

Function produce

Wait *status* = **true** ;
for $i \longleftarrow ind_{cons}$ **to** *end of cache line* **do**
  $copy_buf[i] \longleftarrow tab[i]$;
end
$ind_{cons} \longleftarrow$ index of the other buffer ;
status $\longleftarrow$ **false** ;

Function consume

The synchronization to exchange the two semi-buffers rely on a single shared boolean variable called *status*. Producer can only flip the variable when its value is false while consumer can only flip it when its value is true. This invariant ensure producer and consumer can never update the variable concurrently.

This variable is not modified during the processing of the semi-buffers. It is changed only when synchronization is needed, that is during the semi-buffers exchange. In other words, real sharing of the variable only happens while exchanging the semi-buffers. Since throughput is related to the amount of synchronization and this synchronization only happens when the semi-buffers are exchanged, it is possible to increase the throughput by increasing the size of the semi-buffers. However, bigger semi-buffers means a bigger delay before sending data and hence increased latency. Choosing the right size for the semi-buffers is thus a tradeoff between throughput and latency.

Like most recent communication algorithms[2], BatchQueue makes it possible for production and consumption to happen at the same time without involving data sharing. The synchronization between producer and consumer is only performed once per semi-buffer. This favors the slowest participant: only the synchronization variable is considered when one of the sides finishes processing its semi-buffer, the other side is not slowed down.

[2]See section V

IV. Evaluation

The detailed presentation of BatchQueue in section III-B, the empirical evaluation of BatchQueue presented in [4], and the theoretical comparison with alternative solutions in section V suggests that BatchQueue's design makes it more efficient for pipeline parallelism. This section presents a comparison of the performance between OpenMP stream-computing extension and the improved version using BatchQueue. Performance is measured on a micro benchmark and also three practical applications: GNU FMradio, trellis computation and pipeline template.

Software platform: For each program, two experimentations are compared to the sequential code: they are named “Native” and “BatchQueue” in the graphs. “Native” represents the execution of the code parallelized with the addition of annotations from the OpenMP stream-computing extension. The communication algorithm is the default one provided with the extension. “BatchQueue” denotes the use of BatchQueue as a replacement of the native one. In this case the annotations are the same as in “Native”.

Hardware platform: The machine used for all evaluations features a dual quad-core Xeon X5472 with a frequency of 3 GHz and 10 GiB of RAM. The cores have a 32KiB L1 cache and 6 MiB L2 cache shared between each pair of cores. The operating system is a Debian GNU/Linux 6.0 “Squeeze” installed in 64-bit mode. Both the code of OpenMP stream-computing extension with BatchQueue [5] and the benchmarks [10] can be found online.

A. Micro benchmark evaluation

The design of BatchQueue as described in section III and the previous evaluations [4, 11] performed against state of the art single producer single consumer queues suggest that it should be more efficient than the native algorithm. However, given the complexity of cache behavior, an explicit comparison might give different results, especially when the communication algorithms are used to connect more than two nodes. Indeed, in a long chain of nodes, when one node is blocked to wait for some data to be available it slows down all the downstream nodes by domino effect.

In order to measure the maximum throughput which both the algorithms can achieve, a micro benchmark was conducted. Two configurations are used for the benchmark: one where only the two cores are connected and one where the two cores are the two end points of a chain of 4 cores. Performing the evaluation with these two configurations allow to measure separately the throughput the algorithms can achieve and how well they behave when used to connect more than 2 cores. The benchmark consists in sending a fixed amount of data between two cores: around 32GB for the 2 cores configuration and around 3.2GB for the 4 cores chain configuration. The amount of data sent is carefully chosen to be a multiple of the size of the communication buffer which fits in 64 cache lines of 64 bytes. Bandwidth is

then computed from the time spent and the amount of data sent. We measured the time it takes for both BatchQueue, and the native algorithm.

For each configurations, two variants of the test are proposed: one test with intensive communication called "*comm*" and one test of intensive computation called "*matrix*". In the test *comm*, no action is performed between two data sent which allows to compute the maximum throughput algorithms can achieve. In the test *matrix*, a matrix multiplication is done between two data sent in the purpose of measuring performance of algorithms when the first level cache is under heavy use. The results of these two variants are presented in figure 3 for the configuration with 2 cores, and in figure 4 for the configuration with 4 cores.

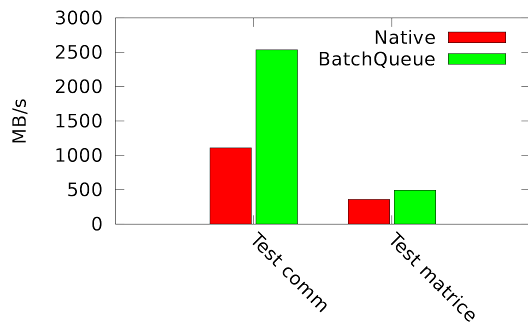

Figure 3: Throughput achieved by BatchQueue and native OpenMP stream communication algorithm under micro benchmark between 2 cores

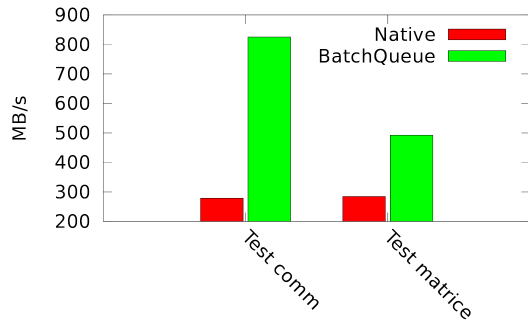

Figure 4: Throughput achieved by BatchQueue and native OpenMP stream communication algorithm under micro benchmark in a chain of 4 cores

The results show a strong advantage in using BatchQueue. In the 2 cores configuration, BatchQueue improves the throughput by a factor 2.3 when communication is intensive and by a factor 1.4 when computation is intensive compared to the native algorithm. In the 4 cores chain configuration, BatchQueue performs even better by improving by a factor 3 when communication is intensive and by a factor 1.7 when computation is intensive.

The smaller improvement in the variant where computation is intensive can be explained by the use of the total time to compute the throughput. Since the matrix multiplication account for more than half of the total time spent by the run, a given improvement in the communication time translate into a smaller improvement in throughput. The throughput speedup is thus closer to the factor 3 than it seems. It ensues that the throughput BatchQueue is capable of achieving is quite insensitive to the contention for the first level cache. Overall, it appears from this micro benchmark that the good results of BatchQueue in terms of throughput are confirmed both when used between 2 cores and when used in a chain of 4 cores, which confirms the interest in using BatchQueue as a communication algorithm to realize pipeline parallelism.

B. Synchronization bound benchmark with GNU FMradio

The results of the micro benchmark confirm that, in terms of throughput, Batchqueue performs better than the native algorithm. Therefore, better performance can be expected by parallelizing a program using Batchqueue, instead of the native algorithm. To confirm this, our first real-application evaluation was done on GNU FMradio. FMradio is one of the three applications transformed for evaluation in [3]. The two other applications are: a Fast Fourier Transformation (FFT) program, and a 802.11a production code from Nokia. These applications were not evaluated because of their unsuitability for our comparison: FFT parallelization required hand tuning of the application code and, 802.11a production code is not presented in the paper.

Two annotated versions of FMradio were done by Pop and Cohen in their evaluation of OpenMP stream-computing extension: one version with pipeline parallelism only and another version accumulating pipeline and data parallelism. We use the pipeline parallelism only version for the evaluation because we want to evaluate the impact of BatchQueue on pipeline parallelism and the other version cannot be fully automatically parallelized. The number of cores involved is fixed to 12 since annotations in the code of FMradio does not allow the number of cores to vary. The results are shown in fig. 5 and presents the speedup compared to the sequential version.

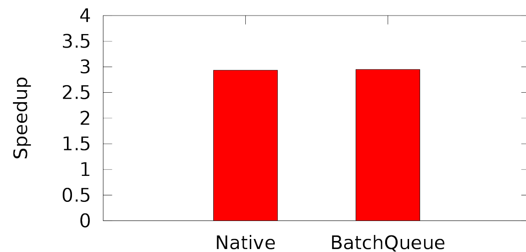

Figure 5: Speedup achieved by BatchQueue and native OpenMP stream-computing communication algorithm for GNU FMradio

Despite the good performance of BatchQueue in the micro benchmark, the performance is unchanged in the case of FMradio. An analysis of the structure of the pipeline was performed at runtime by looking at the actual graph of structures in memory. The observed structure is depicted in fig. 6 and shows that the data flow is not linear, it contains several branches.

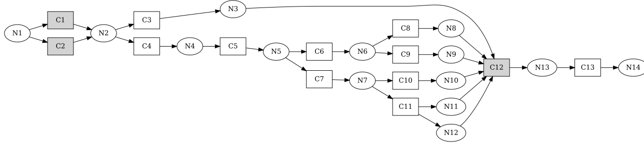

Figure 6: Structure of the pipeline with FMradio

Because of the implied synchronization, every node producing in more branches than it is consuming from is likely to be a bottleneck. This is the case of node C12 for instance, which is the node colored in gray at the right of the pipeline structure displayed above. An extended analysis including the time spent in busy loop in each channel, on both producer and consumer side, shows that the problem is indeed structural but mainly lies in channel C1 and C2, also colored in gray but lying on the left of the structure. The time spent in busy loop is greatly unbalanced between channels C1 and C2: one channel is waiting for the other. Therefore, FMradio presents two limitations which prevent it from scaling with pipeline parallelism: excessive synchronization on one node and unbalanced branches between two same nodes. Overall, FMradio does not scale with pipeline parallelism because the resulting pipeline contains branches and thus does not conform to the linear structure, which pipeline parallelism is good at dealing with.

C. A 40% improvement with trellis computation

As seen in the case of GNU FMradio, scalability cannot always be achieved due to intrinsic limitation of the program to parallelize. In particular, non linear pipeline tends to perform badly, either because of the synchronization involved, or because of unbalanced sibling branches. Therefore, the following experiments, starting with the trellis computation, only consider programs whose dependencies are linear.

The trellis computation originates from a work led at Alcatel Lucent[12, 13]. It consists of a rewrite in C of a portion of C++ code whose purpose is to decode some packets coming from an unreliable network in the most sensible way. The data comes in analog form and may have been altered by the network. The analog physical signal is converted into a binary packet by looking, for each bit, at the probability of it being a 0 or a 1. The probability of a given bit is computed from the value of the signal for this bit and from the probability of all previous bits. The computation of the probabilities is done by filling a trellis progressively. The trellis computation hence deals with a stream of packets as an analog signal and each packet is processed in a sequential algorithm.

As opposed to GNU FMradio, the trellis computation algorithm has the typical structure of what pipeline parallelism deals with: a stream of data with sequential processing of each data. The flow of data in this sequential processing is perfectly linear and so is the pipeline generated when annotating the program with OpenMP stream-computing extension. It should be noted however that each packet is processed independently from the others, so the same technique as in the RPS Linux patch [14] — Receive Packet Steering — could be used for processing the packets. Packets would then be sent in a round robin fashion on different processors and be processed entirely there, thus also avoiding the communication overhead.

The program was parallelized so that each core deals with one part of the packet and transmits the result to the next core to process the next set of bits. The results are presented in fig. 7.

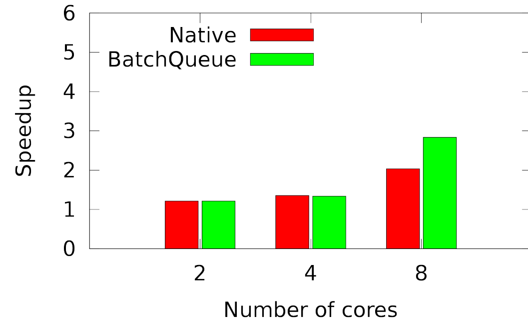

Figure 7: Speedup achieved by BatchQueue and native OpenMP stream-computing communication algorithm for the trellis computation

Contrary to FMradio, the trellis computation presents an improvement with BatchQueue over the native algorithm for a big enough number of cores. On 8 cores, BatchQueue improves throughput by 40% over the native algorithm. Despite this important improvement, the speedup is still far from ideal: the speedup for 8 cores is around 2.85. This suggests that communication is not the limiting factor.

The reason is that some initialization is done for each packet before it is sent and this initialization is not parallelized. Part of the initialization is generation of random analog physical packets in order to avoid reading packets from traces on disk, which would be even slower. Doing the random generation of all packets before the time starts to be accounted would then exhibits a more linear speedup with the number of cores and thus an even better improvement but was not done due to lack of time.

D. 2x speedup with template

The case of trellis computation is promising as it shows speedup improvement is possible by using BatchQueue.

Moreover, the way speedup evolves with the number of cores indicates the reason for this partial scaling does not lie in the communication algorithm, but is again related to the program parallelized. However, unlike previously, the reason is not the structure of the pipeline but is a piece of code which cannot be parallelized. The last experiment, shown below, involves a template code with all the characteristics making it completely suitable for pipeline parallelism. The use of a template code allows to obtain the maximum speedup improvement that can be obtained from BatchQueue when used inside OpenMP stream-computing extension. It also helps programmers who consider pipeline parallelism to estimate the speedup they can expect with their own program, according to how much it conforms to the template.

There is one category of software whose structure is a perfect fit for pipeline parallelism: audio and video processing. As explained in [15], audio and video processing is organized as a graph of filters: data flows in the form of frames from one filter to another. Filters can be stateful, which means their output can depend on the current frame and previous frames processed. This dependency chain means that even the packet steering technique cannot be used to parallelize this code, only pipeline parallelism can. However, parallelizing such a code requires a significant amount of work. Hence, we propose instead in this paper a template code following the same constraints: processing sequentially a stream of data with a dependency chain. Like in the case of the trellis computation, the resulting pipeline is completely linear. Figure 8 shows the results of its parallelization in terms of speedup over the sequential code.

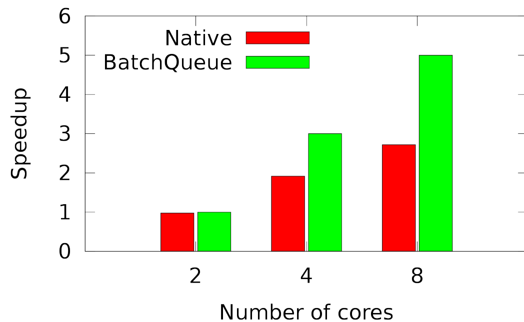

Figure 8: Speedup achieved by BatchQueue and native OpenMP stream-computing communication algorithm for the template program

As expected, the template code presents good scalability. Increasing the number of cores leads to increased throughput, in a relatively linear way. We can see that BatchQueue outperforms the native communication algorithm of OpenMP by a factor 2.

V. Related work

The previous sections present the details of BatchQueue algorithm and how it manages to improve pipeline parallelism. In particular, emphasis is put on the way it reduces the number of synchronizations required between a producer and a consumer and avoids false sharing. This section describe previous work with similar objectives and relate them to BatchQueue. Due to lack of space, this section only compares algorithms at a theoritical level but a benchmark comparison can be found in [11].

FastForward: Among all the alternative algorithms, FastForward [16] is the most simple. The solution retained to avoid sharing producer and consumer's indices is to use one of the value data cannot take to indicate that a buffer entry is empty. By doing this, the producer and consumer only need to access their own buffer index and to read the value of the next buffer entry to know if they can produce or consume.

Although elegant, this approach still exhibits false sharing if the producer and the consumer work on close buffer entries, since they might fit on the same cache line. To prevent from false sharing to happen, the authors of this work explains that a delay must be maintained between the production and the consumption. To be more precise, the producer and the consumer shall always work on data from different cache line.

For this purpose, they propose an algorithm to enforce this delay, based on a busy loop checking that the distance between producer and consumer is not too small. Hence, FastForward only work satisfyingly if the producer and consumer work roughly at the same speed. Indeed, the busy loop compares the indices of the producer and the consumer to compute the distance between them. This comparison implies data sharing and thus should be avoided as much as possible.

BatchQueue solve this problem by enforcing the producer and the consumer to never work on the same cache line. A single shared variable is required to realize it and is read and written only when a semi-buffer is full, that is when the semi-buffer are exchanged.

DBLS and MCRingBuffer: DBLS [17] and MCRingBuffer [18] propose to delay producing data until N cache lines are filled, N being greater than 1. This delayed production of data allows to completely avoid false sharing. Sharing of producer and consumer buffer indices is also avoided by using, for each of these indices, one shared variable and two local variables. The producer and the consumer each have two local variables: one "local copy" of their own index – the index updated after each produce or consume operation – and one "mirror copy" of the shared index of the other participant, storing its position. Local copies are updated for every data produced (resp. consumed) while mirror copies and shared variables are updated for every N cache line filled (resp. emptied).

MCRingBuffer reduces the frequency a bit more compared to DBLS by updating the shared variables only when no progression can be done from the mirror copies. Furthermore, the authors of MCRingBuffer emphasize that producer's variable should be in separate cache line than consumer's variable, to avoid false sharing.

BatchQueue distinguishes itself from DBLS and MCRingBuffer by the number of variables used, and especially the number of shared variables. DBLS and MCRingBuffer use 8 variables, including 2 shared variables indicating what is the next entry to be used for consumption and production. BatchQueue only needs 3 variables, including a single shared variable. Furthermore, BatchQueue takes into account the effects of prefetching in the layout of the variables, to avoid any extra false sharing.

Clustered software queue: The clustered software queue [19] – or CSQ – delays the production of data to minimize data sharing **and** use a bit flip to notify when the communication buffer is full or empty. Despite an algorithm similar to BatchQueue, two differences remain between them. Firstly, with the parameters recommended in the article, CSQ uses a bigger number of buffers[3], each having one synchronization bit to tell whether the corresponding buffer is full or empty. The idea is to offer more flexibility in the synchronization between the producer and the consumer. Moreover, the various elements of the data structure are layout out sequentially in memory.

Each of these two differences has an impact on throughput performance of CSQ. Firstly, although allowing more variation in the throughput, the bigger number of buffers in CSQ increases the number of synchronizations. To produce an amount of data equal to the aggregate size of all the buffers, as many synchronizations are needed as the number of buffers. With only two buffers, BatchQueue only needs two synchronizations. Finally, accesses to the various buffers and synchronization bits being sequential, the prefetching will fetch automatically the next elements. However, the next elements are those being modified, leading to unwanted data sharing.

It stands out from this comparison with related work that BatchQueue compares well with alternative solutions. Besides, we have also shown in [11] that BatchQueue outperforms all these solutions and thus fulfill its objective of performance. The reason is that BatchQueue manages to remove false sharing due to producer and consumer indices and to reduce data sharing to a single bit shared variable. BatchQueue also removes false sharing as a side effect of processor prefetching by moving away the various data structures used by the algorithm from each other.

[3]The paper also studies the influence of the number of buffers on performance but never with less than 32 buffers.

VI. Conclusion

In this paper, we propose to use the BatchQueue algorithm in replacement to the communication algorithm used in the OpenMP stream-computing extension [3]. Compared to this algorithm, BatchQueue is able to improve the overall performance of pipeline parallelism applications up to a factor 2. BatchQueue achieves this result by reducing data sharing by reducing the number of synchronizations and by avoiding false sharing, included due to prefetching. The good results obtained in our evaluation show that by working solely on the efficiency of communication algorithms in parallel applications on multicore hardware, a significant improvement can be achieved.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach*, fourth edition ed. Morgan Kaufmann, 2007.

[2] O. A. R. Board, "OpenMP API specification for parallel programming," **http://openmp.org**.

[3] A. Pop and A. Cohen, "A stream-computing extension to OpenMP," in *International Conference on High Performance and Embedded Architectures and Compilers*. ACM, 2011, pp. 5–14.

[4] T. Preud'homme, J. Sopena, G. Thomas, and B. Folliot, "Batchqueue: Fast and memory-thrifty core to core communication," in *SBAC-PAD*, 2010, pp. 215–222.

[5] "Git repository of OpenMP stream extension with BatchQueue," **git://git.celest.fr/rt_gccstream.git**.

[6] Intel, "Threading Building Blocks," **http://threadingbuildingblocks.org/**.

[7] ——, "Cilk Plus," **http://software.intel.com/en-us/articles/intel-cilk-plus/**.

[8] ——, "Array Building Blocks," **http://software.intel.com/en-us/articles/intel-array-building-blocks/**.

[9] AMD, "AMD64 technology," **http://support.amd.com/us/Embedded_TechDocs/24593.pdf**.

[10] "Git repository of BatchQueue's benchmarks," **git://git.celest.fr/rt_benchs.git/pipepar**.

[11] T. Preud'homme, J. Sopena, G. Thomas, and B. Folliot, "Batchqueue: file producteur/consommateur optimisée pour les multi-cœurs," in *CFSE'08*, 2011.

[12] C. Marin, Y. Leprovost, M. Kieffer, and P. Duhamel, "Robust maclite and soft header recovery for packetized multimedia transmission," *Communications, IEEE Transactions on*, vol. 58, no. 3, pp. 775–784, 2010.

[13] R. Hu, X. Huang, M. Kieffer, O. Derrien, and P. Duhamel, "Robust critical data recovery for mpeg-4 aac encoded bitstreams," in *ICASSP*, 2010, pp. 397–400.

[14] "Receive Packet steering," **http://lwn.net/Articles/362339/**.

[15] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ASPLOS-XII*, 2006, pp. 151–162.

[16] J. Giacomoni, T. Mosely, and M. Vachharajani, "Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue," in *PPoPP'08*, 2008, pp. 43–52.

[17] C. Wang, H.-s. Kim, Y. Wu, and V. Ying, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *CGO'07*, 2007, pp. 244–258.

[18] P. Lee, T. Bu, and G. Chandranmenon, "A Lock-Free, Cache-Efficient Multi-Core Synchronization Mechanism for Line-Rate Network Traffic Monitoring," in *IPDPS'10*, 2010.

[19] Y. Zhang, K. Ootsu, T. Yokota, and T. Baba, "Clustered Communication for Efficient Pipelined Multithreading on Commodity MCPs," *IAENG International Journal of Computer Science*, vol. 36, no. 4, pp. 275–283, 2009.