# Code Parallelization for Multi-Core Software Defined Radio Platforms with OpenMP

Michael Schwall, Stefan Nagel and Friedrich K. Jondral
Karlsruhe Institute of Technology, Germany {michael.schwall, stefan.nagel, friedrich.jondral}@kit.edu

*Abstract*— **Since the number of processing cores in a General Purpose Processor (GPP) increases steadily, parallelization of algorithms is a well known topic in computer science. Algorithms have to be adapted to this new system architecture to fully exploit the available processing power. This development equally affects the Software Defined Radio (SDR) technology because the GPP has become an important alternative hardware solution for SDR platforms.**

**To make use of the entire processing power of a multi-core GPP and hence to avoid system inefficiency, this paper provides an approach to parallelize C/C++ code using OpenMP. This application programming interface provides a rapid way to parallelize code using compiler directives inserted at appropriate positions in the code. The processing load is shared between all cores. We use Matlab Simulink as a framework for a model-based design and evaluate the processing gain of embedded handwritten C code blocks with OpenMP support.**

**We will show that with OpenMP the core utilization is increased. Compared to a single-core GPP, we will present the increase of the processing speed depending on the number of cores. We will also highlight the limitations of code parallelization.**

**In our results, we will show that a straightforward implementation of algorithms without multi-core consideration will cause an underutilized system.**

## I. Introduction

In recent years, there has been a great progress regarding SDR waveform development and the number of available platforms on the market. The SDR platforms differ in the architecture but also in the integrated digital processing units. However, it can be observed that the use of GPPs in cooperation with Digital Signal Processors or Field Programmable Gate Arrays (FPGAs) becomes more important since they enable fast signal processing and the use of various compilers. Due to the fact that GPPs evolved from single-core to multi-core chips to gain computing power by parallelization, this has to been taken into account for the SDR waveform development and hence the programming of algorithms.

An example for such an SDR platform is the Universal Software Radio Peripheral (USRP) from Ettus Research [1]. The USRP comprises the radio frontend, the digital to analog conversion and the resampling, which is handled by an FPGA. The actual digital signal processing is shifted to a GPP running on a host, which is connected to the USRP via the Universal Serial Bus.

We will present an approach of adapting an existing model-based waveform development environment to consider multi-core GPPs. The environment will be based on Matlab Simulink (hereinafter referred to as *Simulink*) and we apply OpenMP for code parallelization due to the common shared memory multiprocessing architecture of GPPs. Furthermore, we will show the speedup of parallelized code for standard and more complex signal processing operations.

Section II introduces the model-based design flow with Simulink and the integration of OpenMP. In section III the parallelization with OpenMP support and the results are demonstrated with case studies. We will point out the occurred difficulties and the solutions. Section IV outlines the importance of multi-core consideration in programming and summarizes the results.

## II. The Toolbox

### *Model-based Waveform Development with Simulink*

As a framework for model-based software design we apply Simulink from The MathWorks [2]. Not only the modeling and simulation of dynamic systems, but also the possibility to generate code for various digital signal processing hardware meet the requirements of current SDR waveform development. Simulink enables an intuitive way to model complex systems: Signal processing elements, for example a digital filter, are mapped to functional blocks. An entire system is created by interlinking and parameterizing these blocks.

The basic waveform development approach, depicted on the left hand side of figure 1, is derived from the Model Driven Architecture and adapted to the physical layer of wireless communication systems as described below [3][4].

The Computation Independent Model (CIM) describes the requirements independent from the implementation and is actually the specification of the radio standard. The transformation to the Platform Independent Model (PIM) is done by implementing the waveform's functionality in Simulink. By extending the PIM with platform specific aspects, the Platform Specific Model (PSM) is created. On the one hand, there are infrastructural platform aspects like the data buses on the system or the configuration of the RF frontends and ADCs and on the other hand processor specific aspects for example the adaption of algorithms for fixed-point representations. The last transformation from the PSM to the executable code is done in two steps: First, C-code is automatically generated using the Real Time Workshop and afterwards compiled with the Microsoft Visual Studio C++ compiler or the GNU compiler collection (GCC).

### *OpenMP - Introduction and Usage*

Open Multi-Processing (OpenMP) is an application programming interface (API) that is jointly developed by software

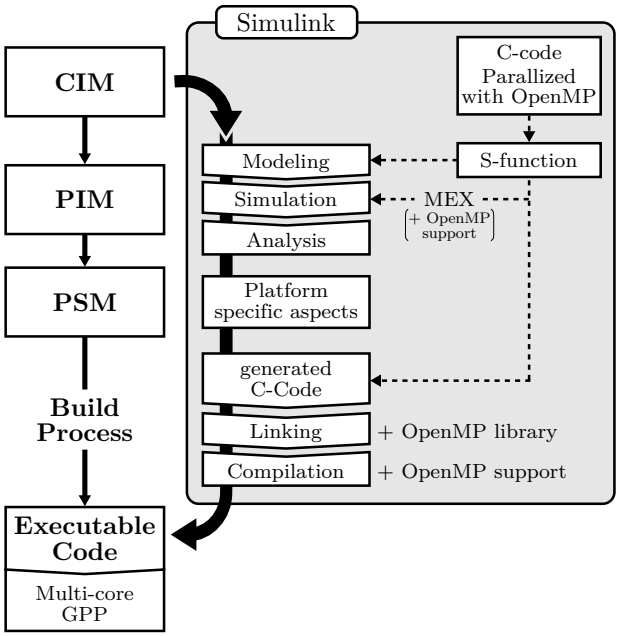


Fig. 1. Model-based waveform development with Simulink and OpenMP support for multi-core GPPs

and hardware vendors since 1997. It is an open standard for shared memory multiprocessing programming that focuses on the parallelization of C, C++ and Fortran code [5][6][7]. The API is implemented in various commercial and open source compilers. The major advantage compared to other parallelization approaches is its elementary possibility to parallelize an existing application as we will demonstrate.

The basic parallelization strategy is based on the *fork/join* execution model as depicted in figure 2. The *Initial Thread forks* in different threads (*team of threads*) that run in parallel and share the calculation load, managed by the *Master Thread*. The number of threads $n$ is independent of the number of processing cores in the GPP. After the parallel section, all threads *join* and the Initial Thread continues. OpenMP provides compiler directives to initiate the fork procedure and to synchronize the parallel threads. Furthermore, it comes with a list of inbuilt functions and environment variables.

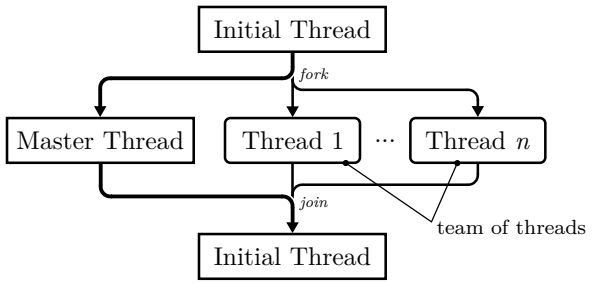


Fig. 2. OpenMP's fork/join execution model

In the following, we will focus on C-code and explain the components as well as the usage of OpenMP on the typical *Hello World* example in listing 1. The main item of the parallelization is the directive in code line 6: `#pragma omp parallel`. A pragma indicates the compiler to execute an inbuilt operation. In this case, `omp parallel` instructs the compiler to activate the OpenMP API and hence to parallelize the following code section. The programmer should not worry about the initialization, the starting or the termination of the threads since this is accomplished by the API. Each available thread will execute the print command and post its internal thread number. The appropriate function `omp_get_thread_num` becomes available by including the OpenMP header file `omp.h` in line 2.

Listing 1. Hello World C-code example

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main() {
5
6    #pragma omp parallel
7    {
8      printf("Hello World from thread %d\n",
           omp_get_thread_num());
9    }
10
11 }
```

The output of the Hello World executable with four utilized threads is shown in listing 2. To activate the OpenMP support, it is necessary to set a compiler specific flag (e.g. GCC: `-fopenmp`). This example demonstrates two important things: First, the parallelization is achieved by adding one code line and second, the order in which the threads finish their tasks is not specified. The last one depends on the current workload of each core and the involved scheduling of the operating system.

Listing 2. Hello World output

```
1  Hello World from thread 1
2  Hello World from thread 3
3  Hello World from thread 4
4  Hello World from thread 2
```

One of the advantages of OpenMP is the feature to parallelize loops. Since loops are omnipresent in Digital Signal Processing (DSP), this is the major application area and therefore the field of interest. Listing 3 shows the implementation of a Multiply-Accumulate (MAC) operation. The elements of the vectors `b` and `c` are multiplied separately and accumulated to the corresponding element of `a`.

Listing 3. MAC C-code example

```
1  #pragma omp parallel for
2  for (i=0; i<N; i++) {
3
4    a[i] += b[i] * c[i];
5
6  }
```

The parallelization works as follows: The number of loop cycles is distributed to the *team of threads*, depending on the OpenMP scheduling. Each thread executes the loop content for

an assigned range of the loop variable `i`. For example, thread 3 passes the loop four times using the values `i={8,9,10,11}`.

Due to the fact that no data dependencies inside the loop exist (e.g. a recursive structure), no further modifications regarding the parallelization of listing 3 are necessary. Otherwise, OpenMP provides features to synchronize threads that are demonstrated in section III.

*OpenMP and Simulink*

Parallelized C-Code can already be included in the modeling of the PIM as depicted in 1. Simulink provides *S-functions* for embedding C-code into a model. These specifically constructed functions can be handwritten or automatically generated using the *legacy code tool*. The S-functions can be treated like ordinary C-code and can therefore be parallelized with OpenMP. To simulate an integrated S-function in the PIM, it has to be compiled to a Matlab Executable (MEX) file. For generating an executable code, the S-function is directly embedded in the overall model C-code and subsequently compiled with OpenMP support. The last one has to be activated in the Simulink build process.

In Simulink, signals can be processed sample- or frame-based. That means that in every simulation step the operations are performed on one sample or on multiple samples as depicted in figure 3. Frame-based processing is the method of choice due to the fact that it is a common format in real-time systems and the fixed process overhead is distributed across many samples. At the same time, frame-based processing offers potential for performing parallelization: The operation on one frame can be realized by a loop and hence parallelized with OpenMP. The occurring difficulties like data dependencies are discussed in section III.

## III. Case Study and Results

In this section we provide case studies that show the processing gain achieved by parallelization. All measurements were performed on an AMD Phenom II X4 955 processor with four cores. We used Matlab R2010a and GCC in its version 4.4.3 on a Ubuntu 10.04 64-bit system. Since the performance is not dependent on platform specific APIs, the evaluations are without SDR hardware considerations. The PIM was transformed directly into an executable and benchmarked on the processor. The Simulink model, which was used as a template for all measurements, is depicted in figure 4. The S-function block represents the parallelized C-code that was adapted to the following case studies:

- Standard DSP operations
- Finite Impulse Response filter
- Fast Fourier Transformation

The processing time for the S-function is measured using time stamp blocks and averaged over a meaningful number of simulation steps. For each case study, we will show the impact of different frame lengths $N$ and the number of utilized threads. The frame length is the number of samples in one frame, which will be varied exponentially from $N$=8 to 8192. The number of threads $n$ will be 2, 3 and 4 since we use a quad-core GPP for the measurements. Each thread occupies one core on the GPP due to the OpenMP scheduling. Using more threads than available cores reduces the performance of the parallelized code.

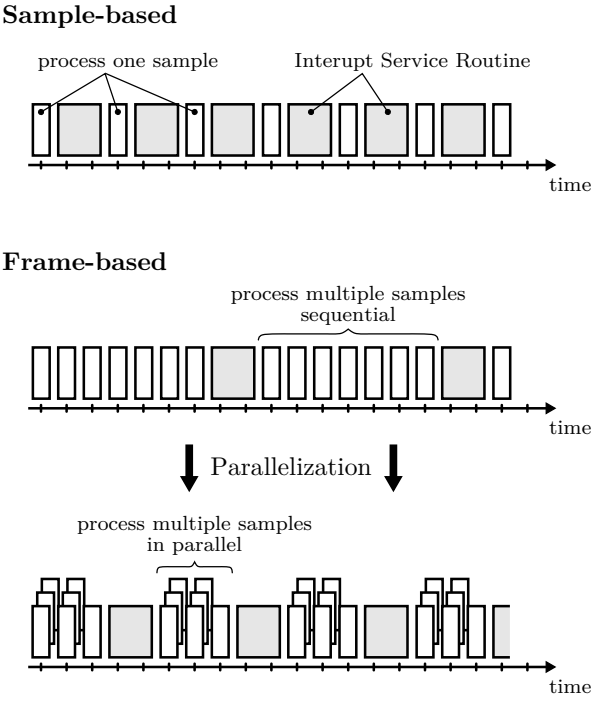


Fig. 3. Sample-based and frame-based signal processing in Simulink

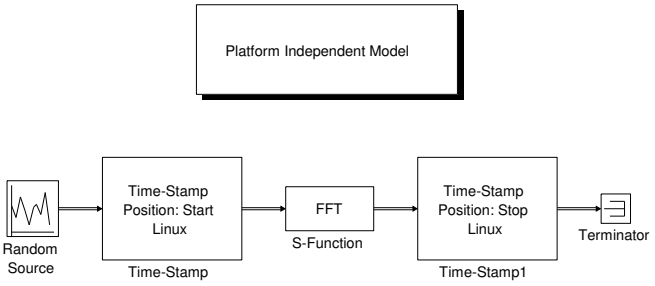


Fig. 4. PIM template in Simulink

*Figures of Merit*

To measure the performance gain of parallelized applications, we introduce the dimensionless factors *speedup* and *efficiency*. If a sequential running program consumes the processing time $t_1$ and a parallel running the time $t_n$, the speedup $s_n$ can be described as

$$s_n = \frac{t_1}{t_n} \, . \tag{1}$$

The index $n$ specifies the number of threads used in parallel. To point out the parallel scalability of the calculation problem, the efficiency is introduced as

$$e_n = \frac{s_n}{n} \, . \tag{2}$$

To give an example: An application on a dual-core processor that utilizes all cores has a speedup of $s_2$=180 % and an efficiency of $e_2$=90 %.

*Case Study 1 - Standard DSP Operations*

The first case study demonstrates the addition and multiplication of samples (equation 3, 4), the magnitude squared and phase of complex samples (equation 5, 6) and the dot product of two frames (equation 7). Whereas $a$, $b$ and $c$ are real numbers, $z$ is a complex one.

$$c(i) = a(i) + b(i), \quad i = 0, \ldots, N-1 \tag{3}$$
$$c(i) = a(i) \cdot b(i) \tag{4}$$
$$c(i) = |z(i)|^2 \tag{5}$$
$$c(i) = \angle z(i) \tag{6}$$
$$c = \sum_{i=0}^{N-1} a(i) \cdot b(i) \tag{7}$$

Since the first four operations are structurally similar to listing 3 and have no data dependencies within the frame, only the implementation of the dot product is shown in listing 4.

Listing 4. Parallelized dot product implementation

```
1  c=0;
2
3  #pragma omp parallel for reduction (+:c)
4  for (i=0; i<N; i++) {
5
6    c += a[i] * b[i];
7
8  }
```

Since all threads add the result of the multiplication to the same variable `c`, the `reduction` clause is used for synchronization purpose to avoid race conditions. It causes that each thread gets a local copy of the variable. The values of the local copies will be summarized (reduced) into a global shared variable once the threads *join*.

*Results:* Due to the fact that the performed operations are based on standard GPP operations, the overhead generated by the OpenMP thread scheduling cannot be compensated for the measured frame lengths. Figure 5 shows exemplarily the speedup and the efficiency for the magnitude squared calculation of complex input samples.

In all three cases, the speedup is lower than 100 % and hence the parallelized version of the code is slower than the sequential one. The measurements for the other examples show similar results.

*Case Study 2 - Finite Impulse Response Filter*

This section introduces a more complex signal processing example: The filtering of digital data with a Finite Impulse Response (FIR) filter. Based on the $N_h$ filter coefficients $h(k)$, the filter operation can be described as

$$y(i) = \sum_{k=0}^{N_h-1} h(k)x(i-k) \; . \tag{8}$$

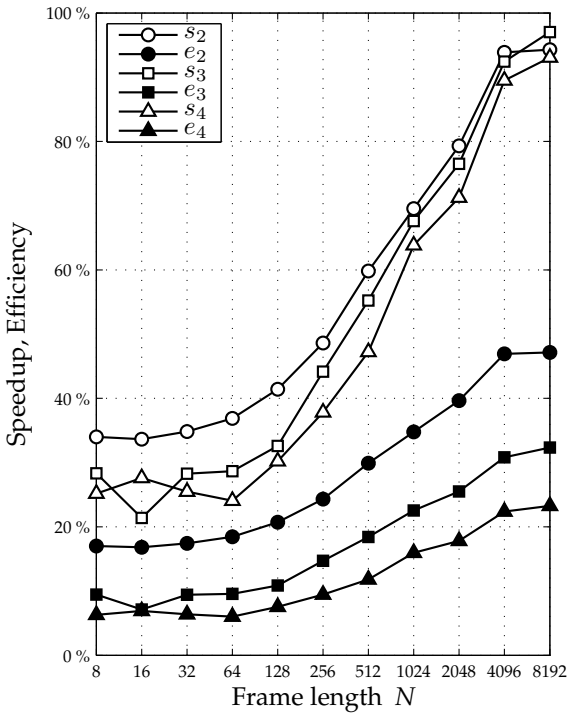


Fig. 5. Speedup and efficiency of the magnitude squared calculation

The input signal $x(i)$ is convolved with the coefficients $h(k)$ to calculate the filtered output $y(i)$. A straightforward implementation of equation 8 is shown in listing 5.

Listing 5. Straightforward FIR filter implementation

```
1  for (i=0; i<N; i++) {
2
3    for (k = N_h-1; k>0; k--) {
4      buffer[k] = buffer[k-1];
5    }
6
7    buffer[0] = x[i];
8    y[i] = 0;
9    for (k= 0; k<N_h; k++) {
10     y[i] += buffer[k]*h[k];
11   }
12 }
```

It can be seen that this C-code shows data dependencies since the values `y[i]` can not be calculated independently of each other. For example, the buffer content is successively shifted in each loop cycle and is therefore dependent on the previous cycles. Even a FIR implementation based on a circular buffer would lead to the same difficulties.

To parallelize the FIR C-code the data dependencies have to be resolved. This means that the calculation of the output value `y[i]` at the time step `i` has to be independent of all other steps. The structure of the alternative FIR implementation is depicted in figure 6. The buffer content is not manipulated during the frame processing. But due to the fact that the calculation window can arbitrarily be shifted, the structure can be parallelized. Listing 6 shows the corresponding parallelized

C-code. The variables `temp` and `k` have to be set as private since each thread requires its own copy. Otherwise, the threads would randomly manipulate the data with one another. After the filtering, the buffer has to be refilled with the last $N_h-1$ samples of the input frame in order to guarantee a continuous convolution.

Listing 6. Parallelized FIR filter implementation

```
#pragma omp parallel for private(temp, k)
for (i=0; i<N; i++) {
  temp = 0;
  for (k=0; k<N_h-1-i; k++) {

    temp += buffer[k]*h[k+i+1];
  }
  if (i < N_h-1) {
    for (k=0; k < i+1; k++) {

      temp += x[i-k]*h[k];
    }
  }
  else {
    for (k=0; k < N_h; k++) {

      temp += x[i-k]*h[k];
    }
    /* Refill buffer */
    if (N-1-i < N_h-1) {

      buffer[N-1-i] = x[i];
    }
  }

  y[i] = temp;
}
```

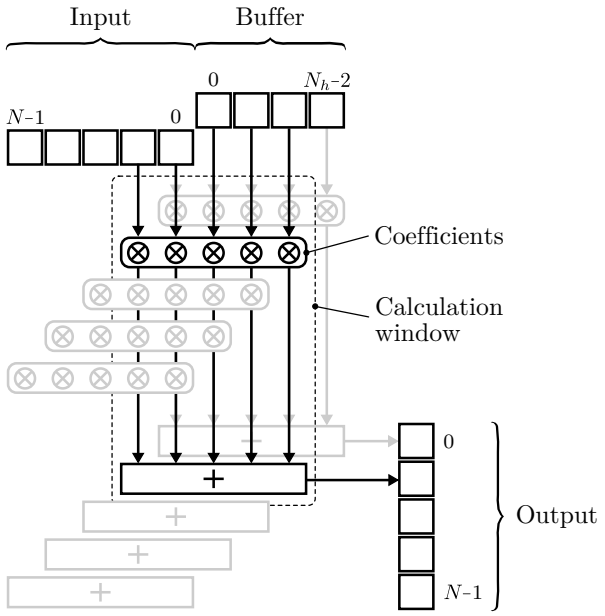


Fig. 6. Parallelized FIR filter implementation structure

*Results:* Figure 7 shows the results for the parallelized FIR filter implementation and $N_h$=100 coefficients. With an input frame length of 16 samples, the parallelized version is already faster than the sequential one. The results depend highly on the frame size. The nearly constant processing overhead caused by the OpenMP thread scheduling becomes smaller compared to the calculation effort, which increases with the frame length. The same is true for the number of coefficients since it affects equally the calculation effort. The utilization of the processor cores is most efficient by the use of only two or three threads and represents almost $100\,\%$ for frame lengths greater than 1024.

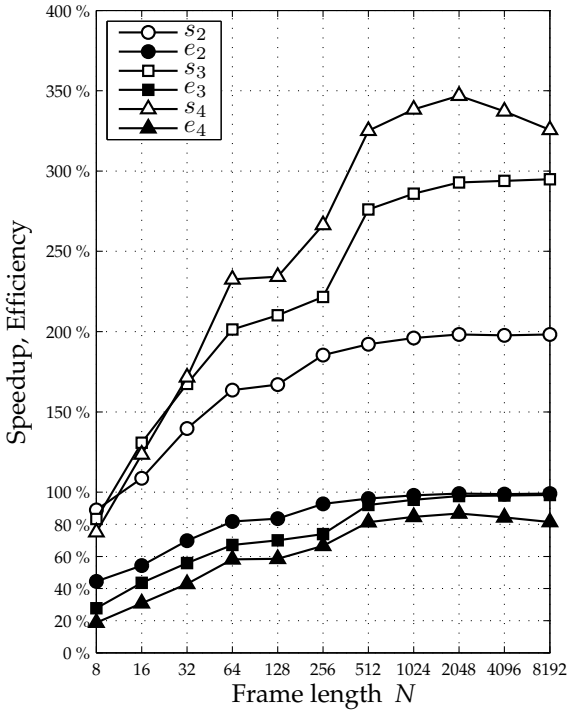


Fig. 7. Speedup and efficiency of the parallelized FIR filter implementation with 100 coefficients

*Case Study 3 - Fast Fourier Transformation*

The Fast Fourier Transformation (FFT) is one of the most important operations for digital communication and spectral analysis. The basic structure of a length $N$=8 Radix-2 FFT with time domain reduction is depicted in figure 8. After reordering (*bit reversal*), the time domain samples pass $N_s=\log_2(N)$ stages successively. In each stage, $N_b=\frac{N}{2}$ butterfly operations are performed. Due to the fact that the butterflies can be calculated in parallel, this is the starting point for the code parallelization with OpenMP.

To avoid data dependencies in the implementation, the indexes of each butterfly input must be a function of the current stage $l=0,\ldots,N_s-1$ and the current butterfly $m=0,\ldots,N_b-1$. The same applies to the exponent $n$ of the twiddle factor $W^n=e^{-j2\pi\frac{n}{N}}$. Listing 7 shows the structure of the parallelized FFT-Code.

Listing 7. Parallelized FFT implementation

```
/* Bit reversal (Time domain reduction) */
```

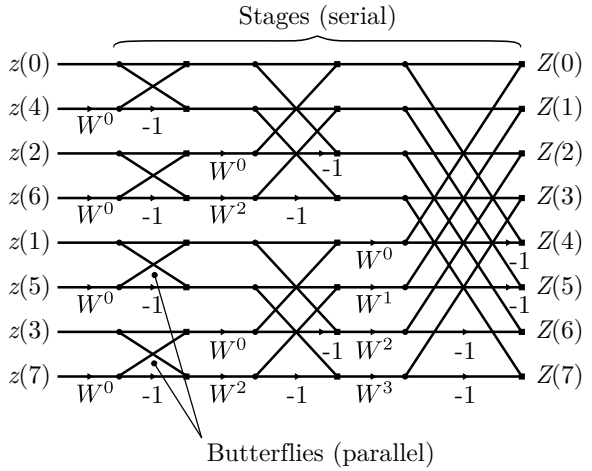


Fig. 8. Radix-2 FFT structure with $N$=8 complex input samples

```
2  #pragma omp parallel for
3  for (i=0; i<N; i++) {
4
5    /* Calculate the bit reversed index */
6    [...]
7  }
8
9  /* FFT Calculation */
10 /* N_s sequential operations */
11 for (l=0; l<N_s; l++) {
12
13   /* N_b parallel operations */
14   #pragma omp parallel for
15   for (m=0; m<N_b; m++) {
16
17     /* Calculate the indexes of the
           butterfly inputs according to m,l*/
18     [...]
19
20     /* Calculate the twiddle factors
           according to m,l*/
21     [...]
22
23     /* butterfly operation */
24     [...]
25   }
26 }
```

*Results:* Similar to the results of the FIR filter, the parallelized version of the FFT shows good results with respect to the speedup and the efficiency (figure 9). The speedup starts at a frame length of 32 complex samples and increases continuously. With a length of 1024, the efficiency represents more than $80\,\%$ for every case. The most efficient utilization is again obtained by the use of only two threads.

## IV. Conclusion

We presented an approach to include OpenMP into Simulink in order to enhance a model-based waveform development with multi-core GPP support. Parallelized code can be simulated and executed within an entire communication system. Since OpenMP undertakes the task of managing the parallel threads, the parallelization of code is simplified to the point of just one code line.

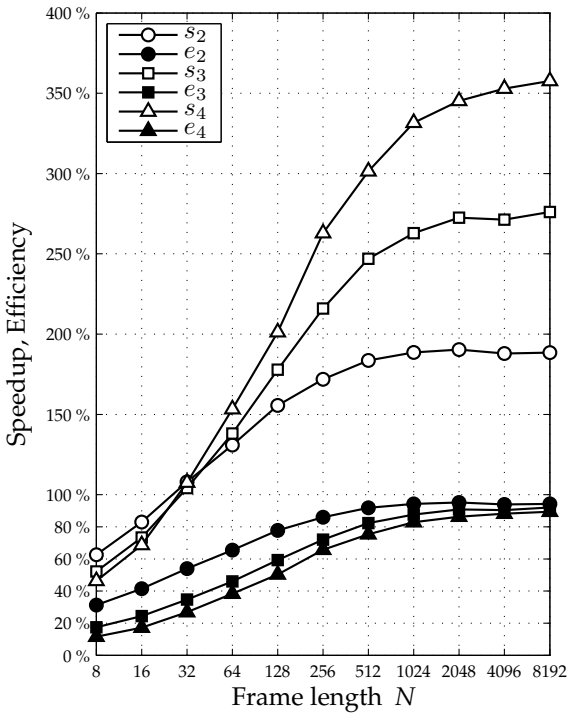


Fig. 9. Speedup and efficiency of the FFT calculation

The results show that an OpenMP-based code parallelization can speedup algorithms and increase the efficiency of multi-core GPPs. Nevertheless, the computational complexity of DSP problems has to dominate the processing overhead, caused by the thread scheduling, before the parallelization becomes profitable. In other words, the more basic operations $(+, -, \cdot, /)$ are parallelized at once, the more efficient the parallelization gets. Furthermore, existing algorithms have to be parallelized at first to resolve data dependencies. But once they are parallelized, the algorithms can be executed on various multi-core GPPs with a scalable speedup.

## References


[1] (2011, Mar.) Ettus Research LLC. [Online]. Available: www.ettus.com
[2] (2011, Mar.) The MathWorks. [Online]. Available: www.mathworks.com
[3] S. Nagel, M. Schwall, and F. K. Jondral, "Porting of waveform: Principles and implementation," *FREQUENZ*, vol. 64, Heft 11-12, pp. 218–223, nov/dec 2010.
[4] S. Nagel, M. Schwall and F. K. Jondral, "Portable Waveform Design," in *Proceedings of 20th Virginia Tech Symposium on Wireless Communications*. Blackburg VA, June 2010.
[5] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP - Portable Shared Memory Parallel Programming*, 1st ed. The MIT Press, Cambridge, Massachusetts, London, England, 2008.
[6] S. Hoffmann and R. Lienhart, *OpenMP - Eine Einführung in die parallele Programmierung mit C/C++*, 1st ed. Springer-Verlag, Berlin, Heidelberg, 2008.
[7] (2011, Mar.) OpenMP.org. [Online]. Available: www.openmp.org