CHALMERS

A Cilk Implementation of LTE Base-station up-link on the TILEPro64 Processor

Master of Science Thesis in the Programme of Integrated Electronic System Design

HAO LI

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, November 2011

A Cilk Implementation of LTE Base-station uplink on the TILEPro64 Processor

HAO LI

Examiner: Sally A. McKee

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-255 1668

Supervisor: Magnus Själander

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1075

Cover:
A Cilk Implementation of LTE Base-station uplink on the TILEPro64 Processor.

Department of Computer Science and Engineering
Göteborg, Sweden 2011

MASTER'S THESIS 2011:11

A Cilk Implementation of LTE Base-station uplink on the TILEPro64 Processor

Master's Thesis in Integrated Electronic System Design

HAO LI

Department of Computer Science and Engineering
Division of Computer Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2011

A Cilk Implementation of LTE Base-station uplink on the TILEPro64 Processor
Master's Thesis in the Master's Program in Integrated Electronic System Design
HAO LI
Department of Computer Science and Engineering
Division of Computer Engineering
Chalmers University of Technology

Abstract

Cilk is a multithreaded programming language developed by MIT. It has a significant feature that its runtime system can schedule the tasks among different processors automatically, leaving the programmer away from those burdens and focusing on exploring parallelism and locality. In this thesis, a Cilk program which implements an LTE Base-station uplink has been written and ported to the TILEPro64 processor. The performance of the Cilk program on X86 platform and Tilera platform is evaluated and compared with a Pthreads version of the same serial version.

Keywords: Cilk, LTE, uplink, TILEPro64

Contents

Acknowledgements

I would like to express my gratitude to the examiner of the thesis Prof. Sally A. McKee from Chalmers University of Technology, and my supervisor Magnus Själander from Chalmers University of Technology.

1. Introduction

The motivation of this thesis is to evaluate the parallelism power Cilk can provide on different platforms. An existing serial version of the LTE base-station uplink is given and the first phase of the thesis is to rewrite this program in Cilk and evaluate its performance together with a Pthreads version written by my supervisor. The second phase is to port the Cilk version of the program onto a Multicore platform, TILEPro64 processor, and evaluate the performance on this platform and the scalability of Cilk.

In chapter 1, multicore processor technology and the importance of parallel computing will be elaborated; different parallel frameworks will also be introduced. In chapter 2, an introduction to the Cilk programming language will be given. Several keywords of Cilk and its runtime system will be illustrated in details. In chapter 3 some concepts of the LTE uplink will be introduced and a rough description of how the LTE uplink is written serially will also be given. In chapter 4 specific details of my Cilk program of the LTE uplink will be given and I will also explain some key factors to boost its parallelism. In chapter 5 The TILEPro64 processor's architecture and mechanism is elaborated. In chapter 6 the steps and methods of porting the Cilk runtime and tool chain support for the Tilera platform will be explained. The evaluation of the performance on different platforms and comparison between different parallel programming languages will be the main content of Chapter 7.

1.1. Parallel computing and multi-core processor technology

Accelerating computations has always been a main motivation in the computer science realm. Building faster hardware was a valid measure of increasing computation speed: sequential instructions can be executed faster if the clock speed of the CPU is faster. Programmers benefited from this method for decades, until the prospects of Moore's law looked dimmer in recent years. Another method to increase computation speed is exploiting parallel execution among sequential instructions, which is called Instruction Level Parallelism (ILP). This method does not increase the speed of the execution of a single instruction, rather it increases the number of instructions executed in a unit of time. Instruction pipelining, very long instruction word (VLIW) and out-of-order execution are typical examples of Instruction Level Parallelism. But this method is still based on a single-core processor and the instructions are still written in a sequential way.

Computer manufacturers had a major shift in strategy from concentrating on a single-core processor: to integrate more than one CPU core onto the processor die, which is called multi-core technology. Adding another CPU core onto the processor die which already has one would boost performance 100% in theory, adding more cores will increase the performance more, multiple instructions can execute simultaneously on different CPU cores independently. It is the multi-core technology which fully realizes parallel computation. Intel Core Duo and AMD Phenom II X6 are the typical representatives of dual-core processors. As the multi-core technology develops, the scale of cores on a single chip has reached to 100 cores. The multi-core processor used in this thesis is the TILEPro64 processor, one decent representative of the multi-core processors market. There are also some more ambitious establishments to realize parallel computing than only integrating cores on one die. Supercomputers are one typical example of them. A supercomputer is a very powerful mainframe computer; the parallelism can be reached in the scale of thousands of processors for one supercomputer. Clusters and servers are also good examples of parallel computation hardware.

1.2. Different parallel programming frameworks

The software developers have shifted their strategy from using sequential programming languages to developing advanced parallel technologies, including languages, compilers, runtime systems, etc. Three typical parallel programming frameworks are introduced in this section: Intel Threading Building Blocks, Habanero Multicore Software Research Project and Wool.

1.2.1. Intel Threading Building Blocks

Intel Threading Building Blocks (Intel TBB) is a runtime library for parallel programming in C++[6]. Its runtime system can help the programmer to exploit parallelism without dealing with managing specific threads. In another word, Intel TBB does not work in the low level of raw threads[7]. Intel TBB uses a working theory called ”task stealing” to create, modify and terminate threads. As there can be multiple processors available at the beginning of the execution of the program, the runtime system of Intel TBB first distributes the work load evenly among the available processors. Then during execution, if one processor has finished the work assigned to it, the processor becomes available again. The runtime system then assigns work from those processors that are still busy to this processor, which makes it look like the idle processor steal tasks from others, this is why it is called ”task stealing”. By this working theory, Intel TBB can orchestrate the existing threads. Though Intel TBB has the advantage of sparing the programmer from working with raw threads, it can be used with other threading packages simultaneously, i.e. POSIX Threads. Intel TBB consists of libraries and header files,

etc[7]. The library files include several basic and advanced algorithms and data struc-

Library (*.dll, lib*.so, or lib*.dylib)	Description	When to Use
tbb_debug tbbmalloc_debug tbbmalloc_proxy_debug	These versions have extensive internal checking for correct use of the library.	Use with code that is compiled with the macro TBB_USE_DEBUG set to 1.
tbb tbbmalloc tbbmalloc_proxy	These versions deliver top performance. They eliminate most checking for correct use of the library.	Use with code compiled with TBB_USE_DEBUG undefined or set to zero.

Figure 1.1.: Dynamic Shared Libraries Included in Intel Threading Building Blocks[7]

tures. The dynamic shared library includes two versions; one library has the general support while the other library features scalable memory allocation[7].

1.2.2. Habanero Multicore Software Research Project

Rice University launched the Habanero project in fall 2007. Quite a few new parallel programming technologies have been developed by this project since then. Habanero Programming languages consist of Habanero Java, Habanero C and Habanero Scala, etc. New compiler research, virtual machine research and parallel runtime system research have also been done within this project. The Habanero team does not only validate these new parallel programming technologies by standard benchmark applications, but also by using other target applications, including medical imaging applications, seismic data processing and computer graphics and visualization. The Habanero team has also been engaged in a number of multicore platforms, homogeneous as well as heterogeneous.

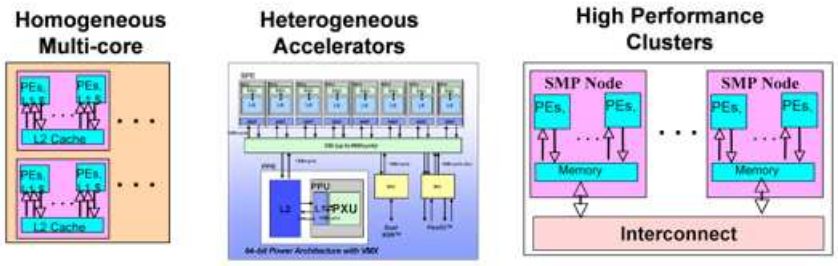

Figure 1.2.: Habanero Multicore Platforms[8]

The Habanero C (HC) language is a new language developed for portable parallelism

based on ANSI C. Habanero C is a task-based parallel programming language with two basic primitives: *async* and *finish* . The *async* primitive creates a new task from the parent task while the parent task resumes its execution. The *finish* <*stml* >primitive guarantees all the tasks created in <*stml* >must be finished before other tasks proceed[9]. Like Intel TBB, Habanero-C implements a work-stealing scheduler[10].

1.2.3. Wool

Wool is a C library for fine grained independent task parallelism[11]. It is developed by the Swedish Institute of Computer Science (SICS). Wool aims to obtain low overhead while achieving task-based parallelism. It is not a certain programming language nor a runtime system but a library that contains macros and inline functions[11]. Wool implements work-stealing scheme to achieve parallelism, which has been introduced in the two previous subsections, which is an idle processors can steal work from other busy processors. As a task-based C library, the main applications for Wool are fully independent computation tasks, so synchronization and dependencies among different threads are averted in Wool.

The Wool Application Programmer Interface (API) provides constructs for task definition, creation and synchronization[11]. Task definition is set by *TASK_n* , in which *n* is the number of arguments of this command. Task creation is done by *SPAWN* , which starts a parallel computation together with its ancestor process. A *SYNC* construct is used as a barrier for previous spawned processes to complete. To build the library, the Wool library itself is built as an object file and linked with the program[11].

2. Cilk language

2.1. Introduction to Cilk

Cilk is a multithreaded programming language. It is developed by the MIT CSAIL Supertech Research Group. The first version of Cilk was introduced in 1994, now it has been updated to Cilk-5.4.6. Cilk exists as a commercial version from Intel named Intel Cilk Plus.

Cilk is designed to explore data parallelism efficiently and intensively. It is a task based language and is provided with a very smart runtime system to schedule and distribute the tasks among given number of threads to realize parallel computation. The Cilk runtime system provides the work-stealing techniques that keeps the programmer away from the burden of scheduling specific tasks for certain processors, leaving the programmer concentrating on discovering possible tasks that can be executed in parallel.

Cilk derives from ANSI C and is a faithful extension to C. In syntax, the difference between Cilk and C is that Cilk contains a few additional keywords that C does not have. This feature guarantees coding simplicity of Cilk. The Cilk keywords include *cilk*, *spawn* and *sync* . These keywords enable Cilk to create scalable parallel applications. If we remove the Cilk keywords from a typical Cilk program, it would be a corresponding serial version of an ANSI C program that can run correctly on a single processor.

Cilk splits a certain program into multiple threads that can operate simultaneously and collectively. The basis for Cilk multithreaded execution is procedure abstraction [1]. A typical cilk program contains a parent procedure at the beginning. This parent procedure can spawn its own procedures, which are called children. The parent procedure will start execution at the beginning. During its execution, procedures will be spawned and start execution. One important feature of Cilk is illustrated here: when the spawned procedures start execution, the parent does not stop but continue its own execution, which means the parent and its children can execute in parallel.

2.2. Cilk keywords

In this section three Cilk keywords will be introduced and analyzed carefully. These keywords are *cilk* , *spawn* and *sync* .

The first keyword introduced is *cilk* itself. It is positioned before the normal C function body. The cilk keyword indicates this function is a Cilk function and not a normal ANSI C function.

A *spawn* is positioned before the invocated Cilk function. This invocated Cilk function is spawned by its parent procedure. It is the Cilk analog of the function call in ANSI C. The difference is mentioned as before that the parent procedure will not stop after the spawned child start execution, and here is how parallelism is generated in Cilk.

The keyword *sync* has similar capability as a memory barrier . When a *sync* is executed in the cilk program, all the children spawned by the procedure calling *sync* in the program must finish their execution before the program proceeds. The procedure after *sync* can only use the return values of its children when their execution finish. The *sync* keyword is not a global memory barrier. It only prevents a certain procedure to resume until all its spawned children are finished, other simultaneously executing procedures can continue their execution regardless of the *sync* keyword.

2.3. Cilk DAG

A directed acyclic graph(DAG) [2] illustrates how the procedures and their children execute in a Cilk program. A directed acyclic graph is a directed graph but with no loop back to a certain start.

A Cilk program divides a single task into several independent procedures that can execute in parallel. The Fibonacci function acts as an example to illustrate how it evolves in a directed acyclic graph:

```
Fib(n)
   if n < 2
      then return n;
      else x = spawn Fib(n-1);
           y = spawn Fib(n-2);
           sync;
           return (x + y);
```

The Fibonacci function is a function to calculate the Fibonacci numbers. The argument n in Fib(n) denotes the number of Fibonacci number to calculate. We can analyze the computation process by the DAG in figure 2.1.

Cilk is a multithreaded programming language. A thread refers to a certain amount of serial ANSI C code, which is equivalent to a sequence of instructions without Cilk keywords in a Cilk program. Cilk works on shared-memory multiprocessors, e.g. multiple threads in one Cilk program have access to a shared memory. In the DAG, each

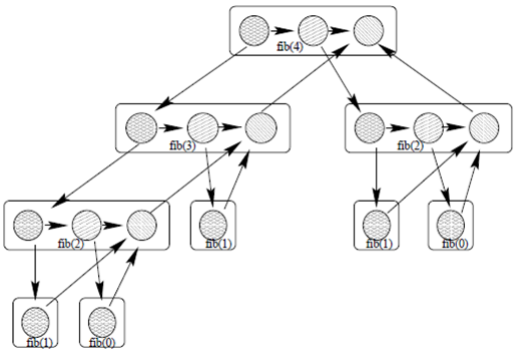

Figure 2.1.: a directed acyclic graph[1]

circle is a thread, the threads in a frame constitute a procedure, the sequence of execution in each procedure is from left to right horizontally as the arrow denotes. A downward arrow denotes spawning a new procedure from the parent procedure at the start of that arrow. An upward arrow denotes a child procedure has finished its execution and will send its return value to its parent. As in this DAG, fib(4) procedure spawns two children, fib(2) and fib(3). After spawning, the fib(4) procedure resumes its own execution without stalling, as the second circle in the procedure denotes. After executing the first two threads in the fib(4) procedure, it must wait for its two children to finish their execution and send their return values back in order to process its third thread. As for the threads of fib(4), fib(3) and fib(2), they would go through a similar way in execution as their parents, spawning their children, resume execution and wait for the return values from their children. The spawning will stop when the last procedure that can be run in parallel is spawned. This DAG illustrates that it is possible to explore parallelism among procedures, without knowing the actual number of processors of the chip we run our code on.

2.4. How Cilk program compiles and runs

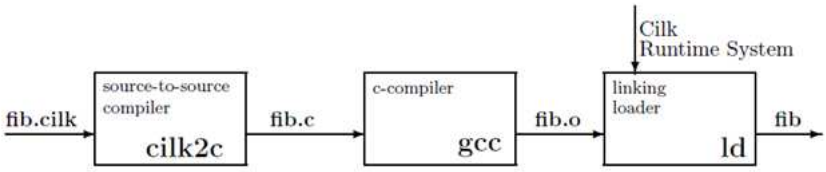

Figure 2.2.: Cilk compiling process[3]

The current version of Cilk, Cilk 5.4.6, provides us with the *cilkc* command, which is

a Cilk analogue to the *gcc* compiler. Many arguments for the *gcc* compiler can be used in the same way for the *cilkc* command. The *cilkc* command will translate a *.cilk* file into an executable file.

The details of the processes within the mechanism of *cilkc* command can be seen in figure 2.2. A *.cilk* file is first translated into an ordinary *.c* file by *cilk2c* , then the *gcc* compiler (assuming the code is run on an x86 machine) translates it into an object file. At last, the linker links the object file with the Cilk runtime system and produces an executable file.

In addition, some preprocessing work is completed first. A *.cilkI* file is produced at the beginning when the *.cilk* file and its header file are combined. The *.cilkI* file is then preprocessed by the C preprocessor and turned into a *.cilki* file. This file is processed by the *cilk2c* tool, by which the Cilk constructs are translated into ANSI C constructs [3].

2.5. Cilk runtime system

Cilk has a smart runtime system to schedule the computation tasks of Cilk programs among the processors automatically, thus taking this burden from the programmers so they can focus on exploring possible parallelism. The current Cilk runtime system supports scheduling a Cilk program on one or multiple processors [3]. The tool to schedule a Cilk program on one single processor is called the nanoscheduler. The tool to schedule a Cilk program on multiple processors is called the microscheduler.

The nanoscheduler simply removes the Cilk keywords in the Cilk program and maps the program into a serial execution of an ordinary C program. Some extra code is added to the program[3] for further scheduling of this program on multiple processors by the microscheduler. Some concepts of data structures are introduced here. A frame is used to keep the state of one procedure. A queue of frames is used to keep the status of certain procedures, it has a head and a tail regarding the procedures. A deque of frames has the same purpose as a queue, but a deque has no head or tail, the procedures can come and go in both ends, it is the analog of the stack in C. The nanoscheduler creates a frame and push it into the deque when a procedure is invocated. Here the `_INIT_FRAME` macro is added in the code. The state of the procedure is saved in this frame during execution. Some overhead occurs here as the current state of the procedure is constantly saved when spawning children happens. When the execution of the procedure is done, the frame is popped off from the deque using `_BEFORE_RETURN_FAST` macro.

The microscheduler implements a work-stealing scheme to maximize the efficiency of all available processors. Some terms are introduced here. A thief is one processor that has processed its own frames and trying to steal unprocessed frames from other processors, which are called victim processors. The distribution of the initialized frames among processors is randomized. A thief will constantly look for unprocessed frames

from others, so this is also an automatic process. Some overhead of this scheduling is inevitable in reality, but the overhead is very small compared to the efficiency the microscheduler brings.

2.6. Performance boost

Two parameters are introduced here. The work denotes the total time for processing all procedures in a Cilk program in serial, i.e. all procedures is processed in a single processor and no parallelism is achieved. The Critical-path length denotes the total time for processing all the procedures that cannot be parallelized [3]. Assuming P processors are to process the Cilk program. T_1 denotes the work time, T_P denotes the critical-path length and T_∞ denotes the processing time of infinite number of processors. According to Amdahl's law, the maximum expected speedup of a parallelized implementation is limited by the fraction of serial work in this implementation, so the lower bound is

$$T_p \geq \frac{1}{P} T_1 \tag{2.1}$$

Cilk's work-stealing scheme guarantees the following performance model, which means in theory, Cilk can achieve optimal parallelism.

$$T_p \approx \frac{1}{P} T_1 + T_\infty \tag{2.2}$$

2.7. For loops in Cilk

The serial version of the LTE base-station uplink benchmark contains several levels of for loops, therefore writing for loops in Cilk is crucial in this thesis. To optimize serial for loops in Cilk, the keyword spawn must be added into the original serial program. The thought about writing a for loop in Cilk at early stage is to spawn the loop body at each iteration in the for loop:

```
void sample_for_loop(int n)
     for (int i = 0; i < n; i++){
          spawn iteration_context;
     }
sync;
```

This way of parallelism is not very efficient when encountering a long loop. The spawning of children is linear. A spawned child would only do one iteration's context work before *sync* comes, where scheduling overhead happens. The DAG of this serial

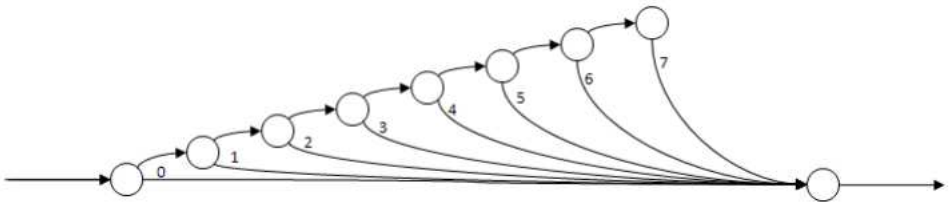

Figure 2.3.: serial way to construct a Cilk for loop[16]

version of spawns is shown in figure 2.3. It is necessary to balance the work load of the iterations.

Cilk is capable to exploit data-level parallelism so it is a data-parallel programming language. The original data set to be processed will be divided into subsets of data. The multiple processors will work on different parts of the original data simultaneously, so as to realize parallelism. In the case of data-level parallelism, the performance can be increased if more processors are added. This feature makes Cilk quite scalable to large number of processors.

The size of each subset of data is called the grain size. Divide-and-conquer algorithm is implemented in parallelizing for loops in Cilk. The iterations in the for loop will recursively break into two branches, each contains half the size of the previous iteration and is divided thus way until size is equal or smaller than the grain size. The DAG of this implementation is shown in figure 2.4.

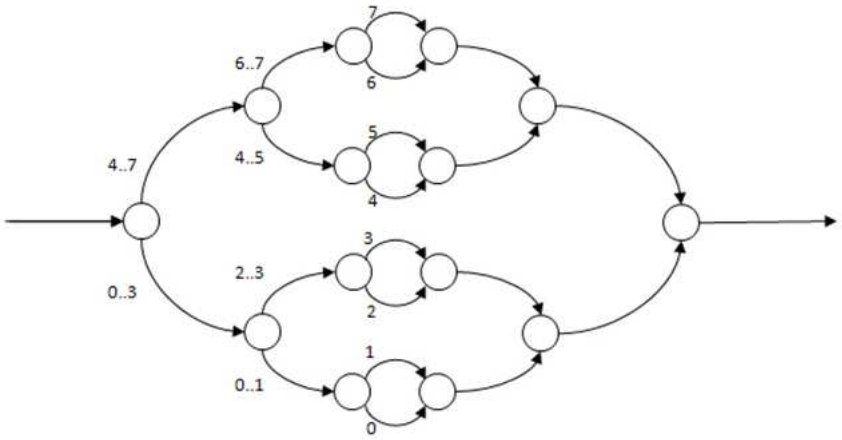

Figure 2.4.: divide-and-conquer way to construct a Cilk for loop[16]

Intel's Cilk++ Programmer guide provides a pseudo code for realizing a divide and conquer algorithm in for loops in Cilk, which is shown below after minor modification[16]:

```
void run_loop(first, last){
```

```
if (last - first < grainsize){
    for (int i = first; i < last; ++i) LOOP_BODY;
    }
else {
      int mid = (last-first)/2;
      spawn run_loop(first, mid);
      spawn run_loop(mid, last);
      }
   }
```

Each time a child is spawned, the unfinished work is split in half and each half is the input to a spawned child. Compared with the simple linear way of spawning iterations in the for loop, a child will not only do one iteration-context work but will do the half of the unfinished work when it is spawned before the child reaches sync and scheduling overhead takes place. The scheduling overhead for sync and spawning is in the order of O(log2N), compared with the early stage with a scheduling overhead in the order of O(N). This is why the divide-and-conquer way is more efficient when N is large. In this thesis, the parallelisms of for loops in Cilk are all implemented in this way.

3. LTE Uplink Receiver benchmark

In this chapter, the LTE standard will be discussed, two crucial techniques in the LTE uplink design will be introduced and the target of parallelism in the LTE Uplink Receiver benchmark will be discussed.

3.1. Introduction to the LTE standard

3GPP Long Term Evolution (LTE) is the latest major standard for mobile communication. This standard is developed in the 3rd Generation Partnership Project (3GPP). It is generally used in 4G wireless communication and is the global standard for 4G.

LTE is based on multiple access technology predecessors. LTE predecessors include GSM/GPRS/EDGE, which belongs to the Second Generation access technology and UMTS and HSPA, which are from the category of the third generation (3G) access technology. The second generation (2G) cellular telecom networks provides digital signals, compared with the analog radio signals in the first generation (1G) network. The multiple access technologies used in 2G include TDMA and FDMA. 3G access technology prevails 2G in higher data rate for downlink and uplink and support multimedia transmission. 3G access technology embraces CDMA technology. In addition, a more reliable security encryption scheme is applied in 3G.

LTE offers a variety of benefits to the world: high peak rates, high spectrum efficiency, low latency, high capacity and lower maintenance cost and wide range of applications. As a result, LTE has been applied and positioned worldwide since 2011, especially in North America and Scandinavia. One main feature of LTE is that packet-switched communication method used in all transmissions in LTE. This feature prevails UMTS which is based on the GSM standard. The main communication technologies used in LTE include OFDMA, MIMO and SC-FDMA. The peak downlink speed of LTE can reach more than 300Mbit/s, while the peak uplink speed of LTE is more than 80Mbit/s. The downlink speed is the data transmission rate from the base station to the user equipment, e.g., a cell phone or a computer. The uplink speed is the data transmission rate from the user equipment to the base station. In addition, LTE supports a variable range of bandwidth, from 1.4 to 20 MHz.

3.2. LTE Uplink Frame Structure

Multiple users carry data to transmit simultaneously in the LTE uplink. The number of users of an LTE uplink can vary from time to time. The data users carry are in two dimensions: time and frequency. For a given amount of time, users are allocated a certain number of subcarriers [4]. These allocated blocks of data are denoted as physical resource blocks. A base-station has a certain frequency band that is allocated by the mobile network operator. This frequency band is divided into several kHz wide sub-carriers, while each user occupies a certain number of them. A resource block is made up of a certain number of sub-channels and it is also called a unit. In time dimension, the time is divided into subframes that each lasts for 1 millisecond. Each subframe is divided into two slots. The two slots have 0.5 millisecond latency between them. Each slot contains seven symbols, in an order that 3 data symbols comes first, then one reference symbol which is used for channel estimation, and 3 data symbols come at last.

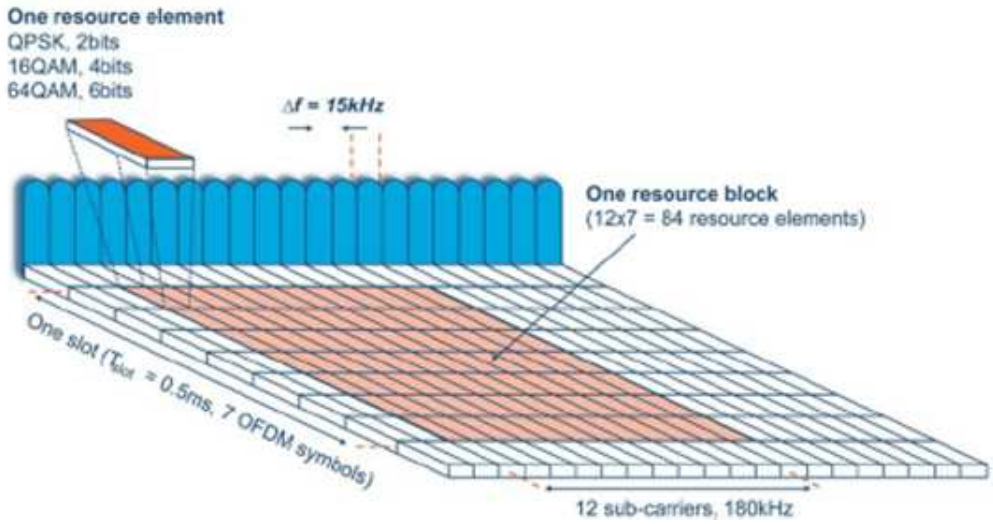

Figure 3.1.: LTE uplink resource block[12]

3.2.1. Signal processing for one user

The front-end of the LTE-base-station receiver will first perform some signal processing tasks that cannot be parallelized. This part is not considered in this thesis project.

The incoming data will go through a radio receiver, filters, cyclic prefix removal and fast Fourier transform (FFT). After that, different resource blocks of a subframe of the input data can be grouped together and processed. The signal processing kernels are shown in figure 3.3.

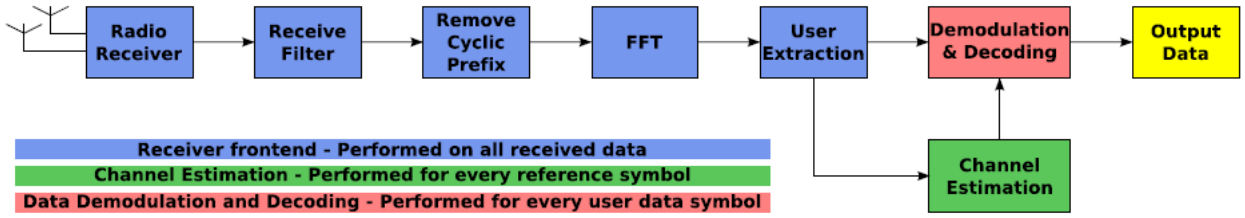

Figure 3.2.: signal processing tasks for one user [5]

For each slot in a subframe, the channel estimation for every reference symbol must be performed first before the data symbols can be decoded. As a result, the first three data symbols of a slot must be buffered before the channel estimation of the corresponding reference symbol is finished. The channel estimation process is divided into five stages. First, the reference symbol must go through a matched filter. Second, an iFFT operation will be performed on the data, to transform the data from frequency domain to time domain. Then channel estimation is performed to calculate noise level and power estimation. After that, the data will be transformed back into frequency domain by an FFT operation. At the final stage of channel estimation, the combiner weights are calculated for decoding the data symbols.

After the channel estimation, the data decoding of corresponding data symbols in the same slot can be performed. From the combiner weights calculated during the channel estimation stage, the data from multiple antennas can be combined. After that, the processed data is transformed back to the time domain by an iFFT. Then a series of signal processing steps will be performed serially: interleaving, soft symbol demapping, turbo decoding and CRC.

3.3. Crucial techniques in LTE uplink design

The LTE uplink transmits data from the user equipments (UE) to the base-station (which is termed eNodeB in LTE literatures). Two crucial techniques in LTE Uplink design, OFDM and MIMO will be introduced in this section.

3.3.1. Orthogonal frequency-division multiplexing

The LTE uplink uses Single Carrier Frequency Division Multiple Access (SC-FDMA) as its transmission scheme, while LTE downlink uses Orthogonal Frequency-Division Multiplexing Access (OFDMA). OFDMA has many similarities with SC-FDMA thus will be introduced here. OFDM is used in many applications. These applications include digital television, wireless local area networks (WLANs), asymmetric digital

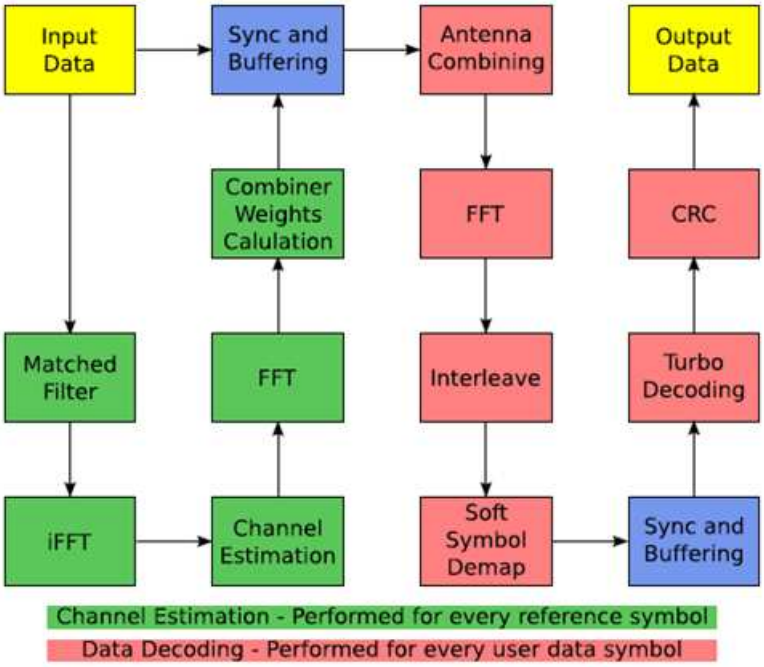

Figure 3.3.: signal processing tasks to be written in Cilk [5]

subscriber loop (ADSL), etc. This technology divides a certain frequency bandwidth available into a large number of sub-carriers. Multiple symbols can each occupy one sub-carrier and thus these signals can transmit simultaneously, this results in a low data stream in each sub-carrier while high data transmission rates of the system can be guaranteed. OFDM denotes orthogonality among these subcarriers. This orthogonal feature and a guard interval at the beginning of every symbol prevents inter symbol interference (ISI). Since ISI can be prevented at the modulation phase, the complexity of the equalization procedures at the receiver can be much lower, thus the whole workload can be reduced.

OFDM is widely used in broadband systems in that it can reduce the multipath problems in broadband systems. Multipath is the phenomenon that there are multiple transmission paths between transmitter and receiver, due to environmental and geological situations, so the received signal contains multiple transmitted signals with different delay and attenuation. OFDM can provide longer symbol period and cyclic prefix to solve this problem. A cyclic prefix is positioned as a guard interval prior to each OFDM symbol as shown in figure 3.4, it can eliminate the interference from the previous interval[13].

ISI is seen by receiving the symbols at wrong time at the receiver stage, since the channel delay is larger than a symbol period. If the symbol period is much larger than

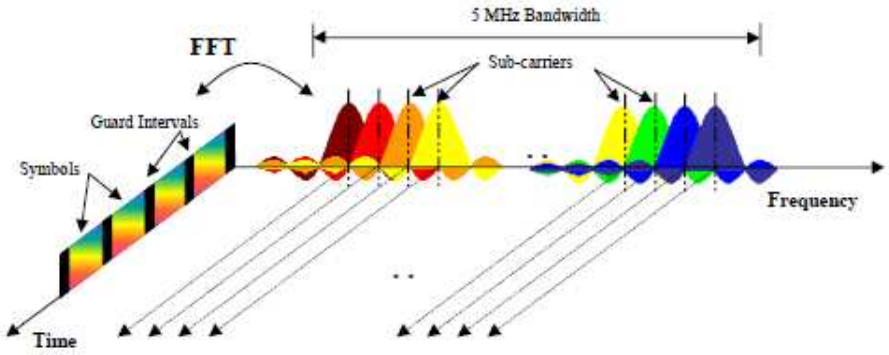

Figure 3.4.: Frequency-time Representation of an OFDM Signal[13]

the channel delay, the symbols can be identified correctly at the receiver stage. To obtain a much larger symbol period while keeping a high transmission rate of data seems a dilemma, but OFDM solves this perfectly. OFDM splits the high-speed data stream into several parallel low-speed data streams and deploys them at separate subcarriers so they can transmit simultaneously, each subcarrier is independently modulated by phase shift keying or quadrature amplitude modulation. An OFDM symbol is a sum of its subcarriers[14]. This method can be viewed in figure 3.5.

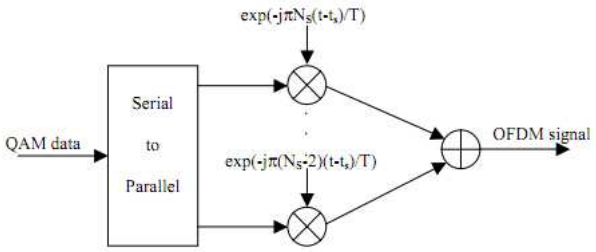

Figure 3.5.: OFDM modulator[**?**]

The design complexity of the OFDM receiver is relatively low, this advantage is gained at the cost of a more complex transmitter, which leads to a much higher Peak-to-Average Power Ratio(PAPR) [4]. The requirement of low PAPR is fulfilled at the uplink design by using an alternative to OFDMA: SC (single-carrier)-FDMA. It is similar to OFDM in many aspects. The difference between them is that SC-FDMA adds a pre-processing block: a DFT processing block before the normal OFDM processing. The feature of the single-carrier waveform guarantees a lower PAPR.

3.3.2. MIMO

Multiple-Input Multiple-Output (MIMO) technology uses multiple transmitters and multiple receivers at the same time to improve communication data-rate. Multiple antennas at the transmitter side can send signals independently and the multiple antennas at the receiver side can receive and recover the original signals. Derived from the conventional time and frequency dimension in Single-Input, Single-Output (SISO), MIMO technology adds a spatial dimension to increase channel capacity and throughput without incurring additional occupancy of bandwidth or power.

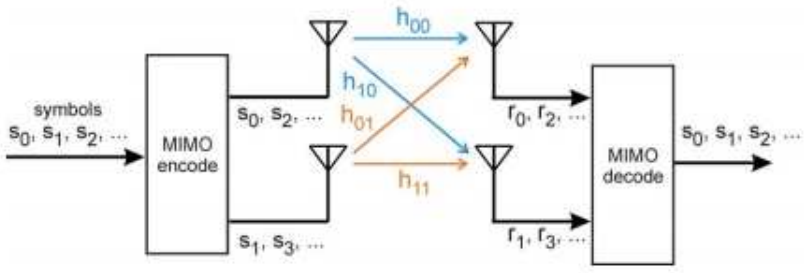

Figure 3.6.: Simplified 2x2 MIMO block diagram[15]

In MIMO systems, the signals at the transmitter side will be precoded first and divide into subset of signals to transmit simultaneously, by this one single data flow will be lower, the transmission distance can be increased, and the receiving range of the receiver antennas will be increased also. MIMO includes several specific technologies: spatial multiplexing, spatial diversity, beamforming and precoding. Spatial multiplexing splits a high-speed data stream into several low-speed data streams and send them by different antennas in the same bandwidth. The receiver antennas can distinguish and recover those low-speed data streams thanks to the difference in spatial dimension. The spatial multiplexing gain provided by MIMO can increase the channel capacity. The spatial diversity technology sends the same data stream on multiple antennas. This measure can enhance the reliability of the channel and reduce the error rate.

MIMO takes advantage of multiple paths to transmit data, therefore multi-path fade could occur, as shown in figure 3.7. The transmitter design would also be complex because of the spatial multiplexing schemes. So OFDM technology is often accompanied with MIMO technology, to solve the multipath problem and reduce the complexity of the transmitter.

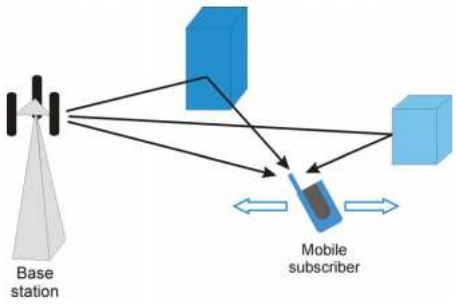

Figure 3.7.: Typical multipath fading scenario[15]

3.4. Target parallelism in the workload of LTE uplink

The target parallelism to seek in the signal processing tasks mentioned in chapter 2 can be realized in two parts and is shown in figure 3.8. The first part is channel estimation. The second part is in the first two steps of data decoding. The incoming data comes as

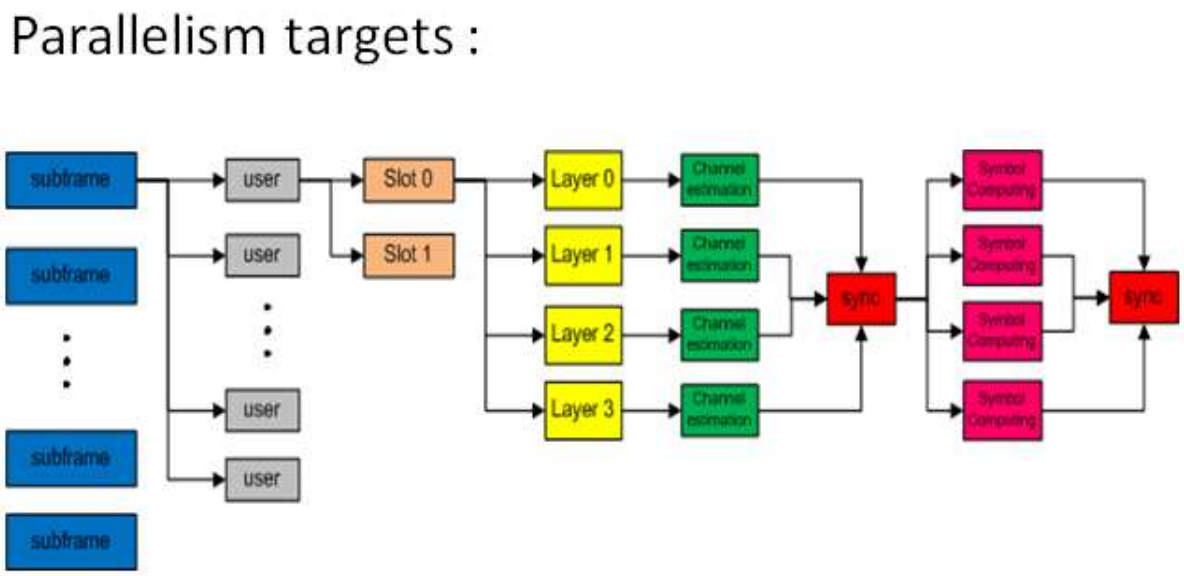

Figure 3.8.: Target parallelism

a stream of subframes every DELTA microseconds. Spatial Diversity is implemented here as multiple users will send the same subframe simultaneously. A constraint is set for the maximum number of subframes that can be performed at the same time. If the numbers of subframes unfinished exceeds that limit, the whole system will halt. We

must seek parallelism between different subframes, since normally the workload of one subframe for multiple users cannot be completed before the next subframe arrives. For each subframe, parallelism can be exploited between different users who are processing the subframe simultaneously. Multiple layers exist in each slot of each user of each subframe, this indicates possible parallelism also exists among slots and layers.

A serial version of ANSI C pseudocode of how to perform the LTE uplink signal processing can be generalized as follows.

```
int main(int argc, char* argv[])
{
    init_data();
    crcInit();
    for ( subframe = 0; subframe < 100; subframe++ ) //
    {
      nmbUsers = user_control(user_data);
      // In this unparallelized program, we process over each user
      for ( user = 0; user < nmbUsers; user++ )
      {// OK two slots have always the same set of user_data
          for ( slot = 0; slot < 1; slot++)
             {// Process each layer seperatly
                 for ( layer = 0; layer < user_data[user].nLayer; layer++ )
                     {//some code for channel estimation
                 for ( layer = 0; layer < user_data[user].nLayer; layer++ )
                     {//some code for symbol computing
                     } // layer loop
                     //some code including data symbol computing
             } // slot loop
             //some code for data decoding
       } // user loop
     } // subframe for loop
     return 0;
}
```

4. Cilk implementation of LTE uplink benchmark

4.1. Creation of input data

MIMO technology prompts the number of transmitter antennas to four. It leads to huge increase in data throughput at no cost of more bandwidth occupied, which results in higher spectral efficiency. A function called `init_data` is created to initialize the input data. The number of maximum data frames is assumed as 10, so first 10 sets of subframe data are created as follows.

```
/* Input data */
static input_data data[NMB_DATA_FRAMES];
void init_data(void) {
  int ofdm, frame;
  int i,slot;

  for (frame=0; frame<NMB_DATA_FRAMES; frame++) {
    for (slot=0; slot<NMB_SLOT; slot++) {
      for (ofdm=0; ofdm<OFDM_IN_SLOT; ofdm++) {
for (i=0; i<MAX_SC; i++) {
  data[frame].in_data[slot][ofdm][0][i].re = rand();
  data[frame].in_data[slot][ofdm][0][i].im = rand();
  data[frame].in_data[slot][ofdm][1][i].re = rand();
  data[frame].in_data[slot][ofdm][1][i].im = rand();
  data[frame].in_data[slot][ofdm][2][i].re = rand();
  data[frame].in_data[slot][ofdm][2][i].im = rand();
  data[frame].in_data[slot][ofdm][3][i].re = rand();
  data[frame].in_data[slot][ofdm][3][i].im = rand();
}
      }
      for (i=0; i<MAX_SC; i++) {
data[frame].fftw[slot][i].re = rand();
data[frame].fftw[slot][i].im = rand();
```

```
      }
    }
    // One reference symbol per layer
    for (i=0; i<MAX_LAYERS; i++)
      RsGen(MAX_RB, data[frame].in_rs[slot][i]);
  }
}
```

After that, the contents for each subframe are initialized. Each subframe contains three subsets of data: `in_data`, `fftw` and `in_rs`. Each subframe has two slots, while each slot has seven OFDM symbols. As there are four antennas at the transmitter, each OFDM symbol has four subsets of data. Each set contains a complex number for each subcarrier.

In this code, the initialization of input data is applied for every user. The number of users for each subframe can be varied but limited to the maximum number set in input data initialization. The `uplink_parameters` function generates users' parameters, including users' number of layers and modulation schemes. The distribution of users' parameters can be either fixed or random at request.

4.2. Channel Estimation

The serial program realizing the channel estimation in the layer loop is shown below:

```
// Process each layer separately
for (layer=0; layer< user -> nmbLayer; layer++){
mf(&user->data->in_data[slot][3][0][startSc], &user->data->in_rs[slot][startSc],
nmbSc, layer_data[layer][0], &pow[0]);
mf(&user->data->in_data[slot][3][1][startSc], &user->data->in_rs[slot][startSc],
nmbSc, layer_data[layer][1], &pow[1]);
mf(&user->data->in_data[slot][3][2][startSc], &user->data->in_rs[slot][startSc],
nmbSc, layer_data[layer][2], &pow[2]);
mf(&user->data->in_data[slot][3][3][startSc], &user->data->in_rs[slot][startSc],
nmbSc, layer_data[layer][3], &pow[3]);
ifft((short int*) layer_data[layer][0], nmbSc, (short int*)user->data->fftw[slot]);
ifft((short int*) layer_data[layer][1], nmbSc, (short int*)user->data->fftw[slot]);
ifft((short int*) layer_data[layer][2], nmbSc, (short int*)user->data->fftw[slot]);
ifft((short int*) layer_data[layer][3], nmbSc, (short int*)user->data->fftw[slot]);
chest(layer_data[layer][0], pow[0], nmbSc, layer_data[layer][0], &res_power[0]);
chest(layer_data[layer][1], pow[1], nmbSc, layer_data[layer][1], &res_power[1]);
chest(layer_data[layer][2], pow[2], nmbSc, layer_data[layer][2], &res_power[2]);
```

```
chest(layer_data[layer][3], pow[3], nmbSc, layer_data[layer][3], &res_power[3]);
// Put power values in the R matrix,
R[layer][0] = cmake(res_power[0],0);
R[layer][1] = cmake(res_power[1],0);
R[layer][2] = cmake(res_power[2],0);
R[layer][3] = cmake(res_power[3],0);
fft((short int*) layer_data[layer][0], nmbSc, (short int*)user->data->fftw[slot]);
fft((short int*) layer_data[layer][1], nmbSc, (short int*)user->data->fftw[slot]);
fft((short int*) layer_data[layer][2], nmbSc, (short int*)user->data->fftw[slot]);
fft((short int*) layer_data[layer][3], nmbSc, (short int*)user->data->fftw[slot]);
} // layer loop
```

The signal processing tasks are performed serially for four receive antennas. The *ifft* function for the four antennas can only begin before the match filtering is done for all the four antennas. Equivalent scenario applies for the *chest* function, *R* function and *fft* function in channel estimation.

One way to parallelize the channel estimation is to spawn each function for the four receive antennas and place a *sync* after each function, e.g. the four *mf* are spawned and a *sync* is positioned after them. Parallelism are gained from spawning four identical functions for the four antennas. The next function can only start when all the four antennas finish their previous function, e.g. the *ifft* function for the four antennas have to wait for the completion of the *mf* function for the four antennas.

Another way is to parallelize all the functions in the channel estimations for each antenna. The advantage of parallelizing function sets among antennas is reduced waiting time for completion of previous functions. In this way, the next function can start when the previous function of the same antenna is finished. The code of this part is shown below:

```
cilk void chan_est(user_queue_item *user, int layer_i, int slot_id, int inner_id) {
  mf(&user->data->in_data[slot_id][3][inner_id][user->startSc],
     &user->data->in_rs[slot_id][user->startSc][layer_i],
     user->nmbSc, user->slot_items[slot_id].layer_data[layer_i][inner_id],
     &user->slot_items[slot_id].layer_items[layer_i].pow[inner_id]);
  ifft(user->slot_items[slot_id].layer_data[layer_i][inner_id], user->nmbSc,
       user->data->fftw[slot_id]);
  chest((user->slot_items[slot_id].layer_data)[layer_i][inner_id],
         user->slot_items[slot_id].layer_items[layer_i].pow[inner_id],
        user->nmbSc, (user->slot_items[slot_id].layer_data)[layer_i][inner_id],
        &user->slot_items[slot_id].layer_items[layer_i].res_power[inner_id]);
  user->slot_items[slot_id].R[layer_i][inner_id] =
    cmake(user->slot_items[slot_id].layer_items[layer_i].res_power[inner_id],0);
```

```
  fft(user->slot_items[slot_id].layer_data[layer_i][inner_id],
       user->nmbSc, user->data->fftw[slot_id]);
}
```

The same method is implemented in symbol computing. The functions of each antenna to perform the symbol computing are grouped together. In slot loops for each subframe, two slots are spawned separately.

4.3. Modeling incoming input data streams

The model of incoming data is that a new subframe carrying a variable number of users comes to the base-station every DELTA microseconds. To model this, a timer must be set to interrupt a process every DELTA microseconds. This feature is realized by using *ualarm(DELTA, DELTA)* function. This function can generate the SIGALRM signal every time delta microseconds elapse. The *signal(SIGALRM, uplink_alarm_handle)* function will trigger the *uplink_alarm_handle* function to start processing a new user's data.

5. TILEPro64 Processor

The second phase of the thesis is to port the X86 Cilk version of base-station uplink to the TILEPro64 processor. An introduction of this processor is provided in this chapter.

5.1. Introduction

The TILEPro64 processor was brought to the market in 2008 by Tilera Corporation. Tilera Corporation is a world-leading company in scalable multicore embedded processor design. Its products include a series of full-featured multicore processors, a complete set of multicore programming development tools and a series of high performance boards.

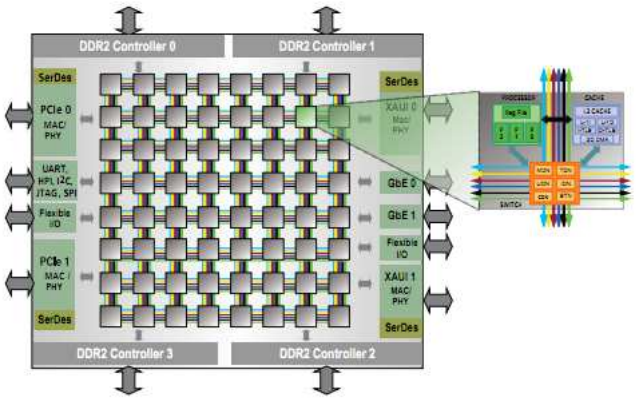

Figure 5.1.: structure of TILEPro64 processor[17]

The TILEPro64 processor provides high level of performance together with high power efficiency. It offers flexibility by its capability to be programmable in ANSI C. It is comprised of 64 homogeneous processor cores (tiles) to provide significantly high compute performance. The 64 homogenous processor cores (tiles) on the TILEPro64 processor are arranged as a two-dimensional, eight by eight grids.

These tiles are connected with each other and to the external off-chip memory. The communication between these tiles is realized by the iMesh network, which is based on Tileras iMesh technology. The processor also incorporates a Dynamic Distributed

Cache (DDC) system. Compared with conventional memory hierarchy where a large fixed L3 cache is shared by multiple cores, the L2 cache in DDC system is spread all over the tiles of the processor. This feature greatly improves cache utilization and therefore improves performance of the processor. The TILEPro64 processor also provides four DDR2 controllers with optional ECC, two four-lane PCIe interfaces and flexible I/O interfaces. These 64 tiles in one TILEPro64 processor have two kinds of clock frequencies: one called -9 device has a clock frequency of 866MHz; the other kind called -7 device has a clock frequency of 700 MHz.

5.2. Three key engines of the Tile Processor Architecture

Each tile on TILEPro64 processor is a full-featured processor core and an entire operating system can be run on a single tile. The TILEPro64 processor provides a technology called multicore hardwall, this technology guarantees that each tile can perform its own execution without interference from other tiles. Figure 5.2 shows how it works. Multicore hardwall mechanism can block traffic and interrupt a certain unwanted process to protect desired applications or operating systems.

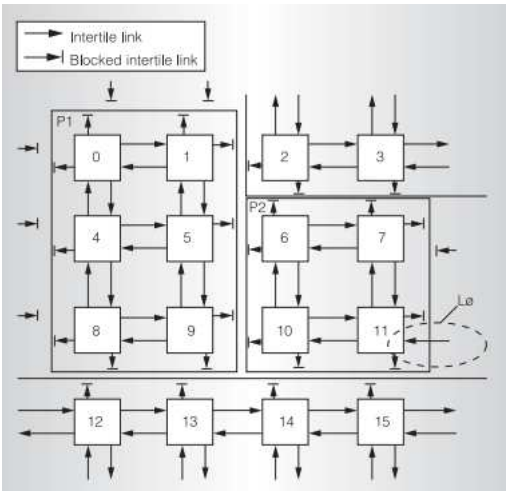

Figure 5.2.: protection structure of multicore hardwall[18]

Three engines in a tile guarantee its performance: a Processor Engine, a Cache Engine and a Switch Engine.

The Processor Engine takes care of normal computations and memory management. All the features a conventional single core processor possesses are included in a tile. A tile contains a 32-bit processor-engine, which implements a three way VLIW instruction set that provides three instruction execution pipelines. A tile can either be used to

run one certain operating system, or cooperate with other cores to run a multi-core operating system, thanks to the Processor Engine.

The Cache Engine contains two level of caches and Translation Lookaside Buffers, which can translate virtual addresses to physical addresses. The two levels of caches can reduce the waiting time of the processor for the memory latency. In this way it can prevent the processor to stall and thus guarantee the processors performance. It is in charge of the communication of instructions and data among the tiles and the on-chip memory. The Cache Engine also supports direct memory access (DMA), leaving the processor to perform other operations when data transfer is underway. Last but not least, cache coherence is also maintained by the Cache Engine.

The Switch Engine comprises six networks including one called the Static Network and five dynamic networks. These six networks are all full-duplex and transmit packet data to other tiles and on-chip memory. These six networks is the basic structure of the Tilera iMesh network.

6. Porting Cilk to the TILEPro64 processor

Porting Cilk applications to the Tilera multicore platform can be accomplished with the help of the Tilera's Multicore Development Environment(MDE). The MDE is a set of software tools, including ANSI C and C++ compilers, a runtime environment of Symmetric multiprocessing (SMP) Linux, an Integrated Development Environment (IDE) and standard debug and profiling tools. It releases the programmers from working on isolated cores so that the programmers can manage multiple cores on a higher level.

As the MDE supports ANSI C and SMP Linux, the porting task can be generally described in four steps: the first step is to compile the Cilk runtime system for the Tilera architecture, the second step is to compile the target benchmark on the Tilera's multicore platform, the third step is to link the benchmark with the Cilk runtime system and the final step is to run the application on that platform.

To run the Cilk application on X86 machine, a *gcc* compiler is used to compile the Cilk code and link it with the Cilk runtime system. When it comes to porting the application on the Tilera platform, the Cilk runtime system must first be compiled for the Tilera architecture. To do so, a Tilera cross compiler must be used instead of *gcc*, which is called *tile-cc*. To change compilers and compile the Cilk runtime system for the Tilera architecture, the following instruction must be performed before compiling:

```
CC=/opt/tilepro/bin/tile-cc ./configure
```

`/opt/tilepro/bin/ is the location of tile-cc.`

In addition, the system dependency file in Cilk runtime system, *cilk-sysdep.h* must be modified to add the Tilera architecture dependencies and a specific atomic swap routine for the Tilera architecture as follows:

```
/*------------------------
TILE
------------------------*/
#ifdef __tile__
#define CILK_CACHE_LINE 64
#define CILK_MB() __asm__ __volatile__ ("mf" : : : "memory")
#define CILK_RMB() CILK_MB()
#define CILK_CACHE_LINE_PAD char __dummy[CILK_CACHE_LINE]
#define CILK_WMB() CILK_MB()
#include <atomic.h>
```

```
/* atomic swap operation */
static inline int Cilk_xchg(volatile int *m, int val)
{
return atomic_exchange_and_add(m, val);
}
#endif /* __tile__ */
#ifndef CILK_CACHE_LINE
# error "Unsupported CPU"
#endif
```

The reason the constant CILK_CACHE_LINE is set to 64 is that the Level 1 data cache line size of the TILEPro64 processor is 64 bytes.

The whole benchmark compiling process can be viewed in figure 6.1.

The first step of the compiling process is to pass the target Cilk program through

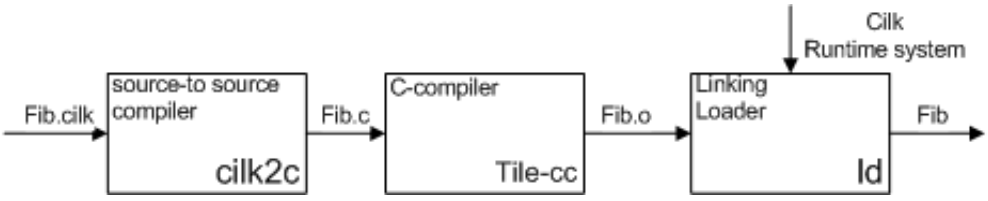

Figure 6.1.: Tilera compiling process[3]

the *cilk2c* checking preprocessor, this preprocessor can translate all Cilk constructs into ANSI C constructs[3]. After that, the *tile-cc* compiler compiles the ANSI C code obtained from the previous step into optimized machine code that can run on the Tilera multicore platform. One advantage of the *tile-cc* compiler is that the arguments in the *gcc* compiler can be applied in it. In the last step of the compiling process, the Cilk runtime system is linked with the machine code and an executable file is produced.

The *tile-cc* compiler can ease the small cache problem on the Tilera platform. As a multicore platform, the cache volume on each tile of the TILEPro64 processor is relatively small compared with traditional single-core processors. To keep reasonable locality in temporal and spatial dimension, efforts should be made to divide large chunk of data into smaller ones for each tile and avoid large communication overhead among different tiles. The Imesh interconnect network contributes to reducing the communication latency and power compared to traditional architectures using a bus for transferring data between components.

The system software stack of the Tilera architecture consists of three layers. The first layer is the hypervisor, the second is the supervisor layer and the third layer is the application layer. The hypervisor is a virtual machine manager that manages multiple operating systems to run on a host computer. In our case, one version of Symmetric

Multiprocessing (SMP) Linux, which is called Tile Linux and implemented by Tilera, is the guest operating system that runs on the hypervisor. The application layer is the closest layer to the end-users, end-users interact with this layer to run applications in user space.

7. Performance analysis

7.1. Performance comparison between Cilk and Pthreads

One of the main criteria of the performance of the LTE uplink benchmark is the interval between incoming subframes. Incoming subframes are provided to the processor every DELTA microseconds. The tolerance for a shorter interval indicates a faster speed of the program. The maximum number of subframes the processor can maintain at the same time is 10. If more than 10 unfinished subframes are on the processor, an error will come out and interrupt the whole processing.

Two versions of code of the LTE uplink benchmark will be tested and compared. One is the Cilk version and the other is the Pthreads version. The two versions of code will be tested on two platforms. The first platform is a Linux 2.6.35-23 Server, Lucid Ubuntu SMP. This server consists of two quad cores (e5620) with hyperthreading gives 16 virtual cores each containing a Intel Xeon 2.40 GHz CPU. The second platform is the TILEPro64 processor. This processor contains 64 independent tiles. These tiles have two kinds of clock frequencies: one called -9 device has a clock frequency of 866MHz, the other called -7 device has a clock frequency of 700 MHz.

In the first step the lowest DELTA the two versions of LTE uplink benchmark can tolerate are tested on the TILEPro64 processor. By typing the command:

```
make run_cilk_pci
```

the Cilk `uplink` executable is produced and run. By modifying the makefile, the Cilk benchmark is tested on 4, 8, 12, 16, 20, 32, 48 and 64 parallel tiles on the server. As for the Pthreads version, by typing the command:

```
make run_pci
```

the Pthread `uplink` executable is produced and run. No argument is used to set the number of possible parallel threads but the program seeks parallelism automatically. The results of the lowest DELTA for the two versions of code on the TILPro64 processor platform are shown in figure 7.1.

The Cilk program has the lowest DELTA value 38 ms when it is running on 8 or 12 tiles. The DELTA on 16 and 20 tiles is 40 ms, which is also close to the lowest value. When the program is running on 4 tiles, the speed decreases significantly as the DELTA reaches the highest value of 130 ms. The Pthreada version has a DELTA value of 5 ms, which indicates that the Pthreads version runs much faster than the Cilk version on the

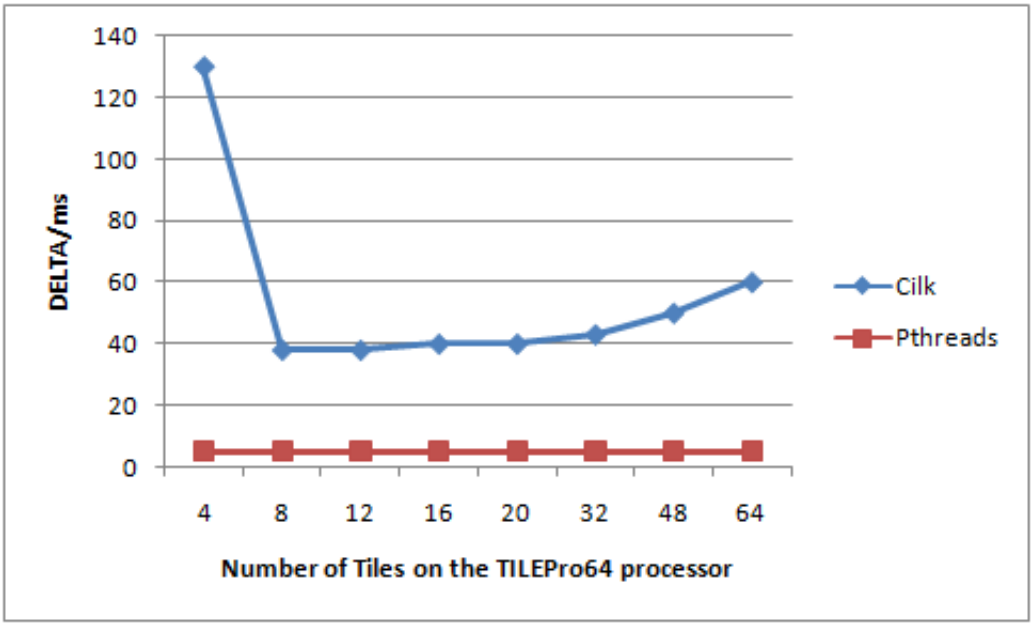

Figure 7.1.: smallest DELTA on the TILEPro64 processor

TILEPro64 processor platform.

From this test some analyses can be made. First, the dependencies among possible parallel threads in the Cilk program have some negative impacts on the scheduling tasks on the TILEPro64 processor. Second, the overhead of scheduling tasks among tiles increases significantly when the number of tiles is large, in this test the number is around 16 to 20.

In the second step the lowest DELTA the two versions of LTE uplink benchmark can tolerate are tested on the Lucid Ubuntu SMP. By typing the command:

```
make run_cilk NPROC=16
```

the Cilk `uplink` executable is produced and run. The argument NPROC=16 denotes the executable runs on 16 processors, by modifying this arguments, the Cilk benchmark is tested on 4, 8, 12, 16 processors. The argument has also been modified to test hyperthreading to 20, 32, 48 and 64 parallel threads on the server. As for the Pthreads version, by typing the command:

```
make run_x86
```

the Pthreads `uplink` executable is produced and run. No argument is used to set the number of possible parallel threads but the program seeks parallelism automatically. The results of the lowest DELTA for the two versions of code on the Lucid Ubuntu SMP platform are shown in figure 7.2.

The Cilk program has the lowest DELTA value 1.6 ms when it is running on 16 processors. Hyperthreading to more than 16 cores cannot boost the performance as the

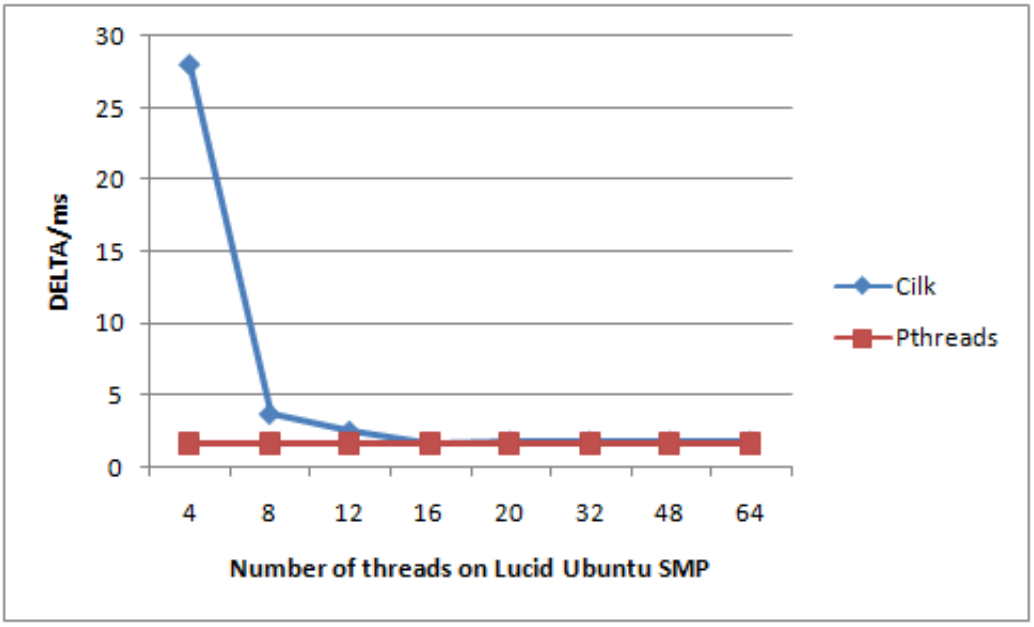

Figure 7.2.: smallest DELTA on Lucid Ubuntu SMP

DELTA value remain almost the same as 1.6 ms when the program is running on 20, 32, 48 and 64 possible parallel threads. The speed of the program decreases linearly when the number of available processors is 12 or 8, and when the program is running on 4 processors, the speed decreases significantly as the DELTA reaches the highest value of 28 ms. The Pthreads version has a DELTA value of 1.6 ms. The Cilk version and the Pthreads version run at the same speed on the Lucid Ubuntu SMP server. Cilk copes with the dependencies problem among possible parallel threads well on this Linux server.

From these two tests and comparisons, some conclusions can be made. Cilk takes the advantage of its simplicity in coding and its smart runtime system. But it is not always efficient and sometimes not stable. In addition, Cilk has some difficulties in handling interruptions when calling a Cilk function in an ANSI C program, which is the main reason why the most dependencies are created in this Cilk program. The Pthreads version is more complex and harder to write, and the scheduling of tasks among processors is done by the programmer. But it provides more flexibility to assign the tasks by the programmer.

7.2. Performance boost in Cilk

In the following tests, the LTE uplink benchmark is modified to measure the processing time for 10000 incoming subframes. Intervals and delay in the Cilk code is removed to test the computation ability of Cilk on the two multicore platforms: the Lucid Ubuntu SMP and the TILEPro64 processor. In the first step the processing time of 10000 incoming subframes by the Cilk program is tested on the TILEPro64 processor. By typing the command:

```
make run_cilk_pci
```

the Cilk `uplink` executable is produced and run. By modifying the makefile, the Cilk benchmark is tested on 1, 2, 4, 8, 12, 16, 20, 32, 48 and 64 parallel tiles on the server. The results of the processing time by the Cilk program on the TILPro64 processor platform are shown in figure 7.3.

The smallest processing time is 285 seconds when the benchmark is running on 16

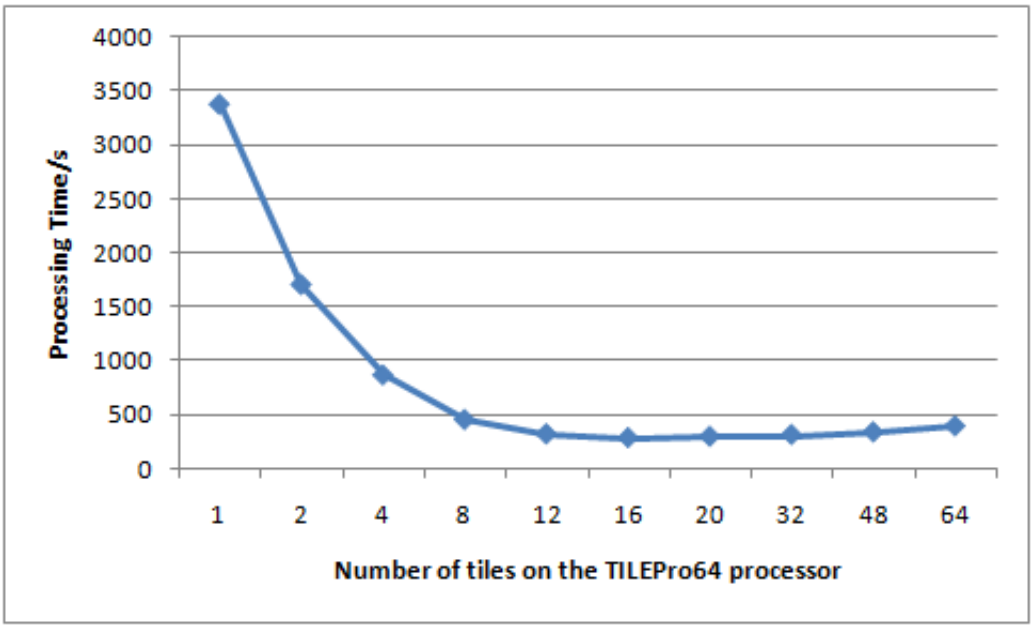

Figure 7.3.: time of processing 10000 subframes on the TILEPro64 processor

tiles. The processing time on 8, 4, 2, 1 tiles is inversely proportional to the number of tiles, which demonstrates the scalability of Cilk program. The scalability also has a certain limit and as in the previous tests, as the number of tiles grows, the overhead of scheduling tasks among tiles increases, which explains why the processing time on 20, 32, 48 and 64 tiles is larger than 285 seconds.

In the second step the processing time of 10000 incoming subframes by the Cilk program is tested on the Lucid Ubuntu SMP. By typing the command:

```
make run_cilk NPROC=16
```

the Cilk `uplink` executable is produced and run. The argument NPROC=16 denotes the executable runs on 16 processors, by modifying this arguments, the Cilk benchmark is tested on 4, 8, 12, 16 processors. The argument has also been modified to test hyperthreading to 20, 32, 48 and 64 parallel threads on the server. The results of the processing time by the Cilk program on the Lucid Ubuntu SMP platform are shown in figure 7.4.

The smallest processing time is 19.35 seconds when the benchmark is running on 16

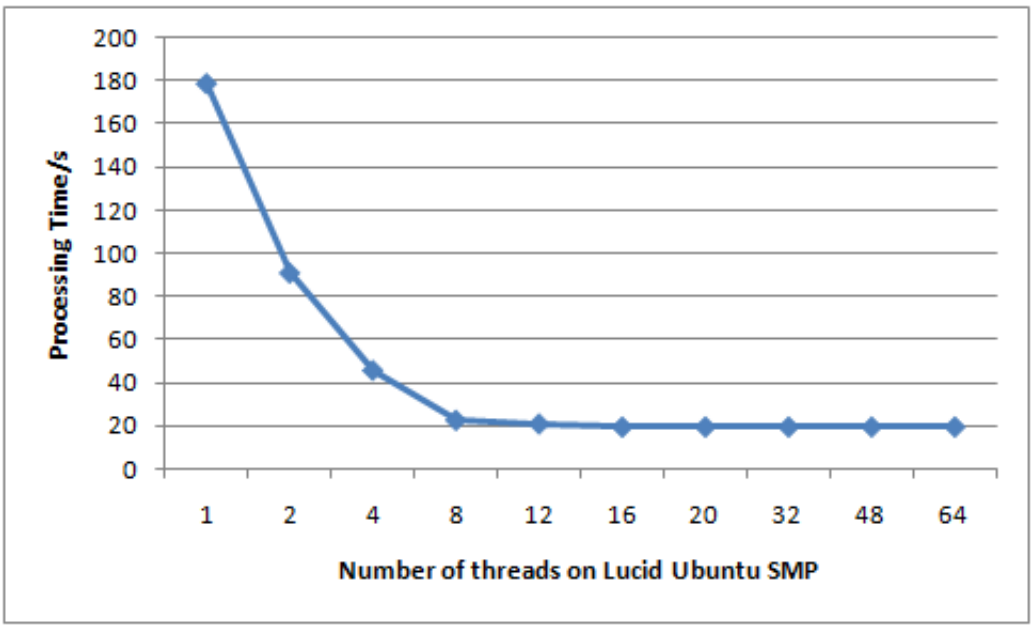

Figure 7.4.: time of processing 10000 subframes on Lucid Ubuntu SMP

processors. Hyperthreading to more than 16 cores cannot boost the performance as the processing time remain almost the same as 19.35 ms when the program is running on 20, 32, 48 and 64 possible parallel threads. The processing time on 8, 4, 2, 1 tiles is inversely proportional to the number of tiles, which demonstrates the scalability of Cilk program, the speedup is almost proportional to the number of processors. This series of tests demonstrate Cilk's ability to exploit asynchronous parallelism optimally.

7.3. Performance analysis in the Cilk program

In this section, Oprofile is used to profile the LTE uplink benchmark on the TILEPro64 processor. The proportion of time occupied of the benchmark by each application is shown below.

Profiling result of the Cilk version on the TILEPro64 processor

samples	%	app name	symbol name
1213	15.7799	libtile-cc.so.1	(no symbols)
859	11.1747	vmlinux	(no symbols)
844	10.9796	no-vmlinux	(no symbols)
787	10.2381	uplink	do_what_it_says
380	4.9434	uplink	cmul
371	4.8263	uplink	Cilk_scheduler
339	4.4100	uplink	fft
327	4.2539	uplink	matrix_a_a_hermite_plus_b_4xX_complex
243	3.1612	uplink	cadd
217	2.8229	uplink	ant_comb
182	2.3676	libpthread-0.9.29.so	(no symbols)
175	2.2766	uplink	extend_global_pool
157	2.0424	uplink	Cilk_exception_handler
152	1.9774	libuClibc-0.9.29.so	(no symbols)
138	1.7952	uplink	Cilk_parse_command_line
127	1.6521	uplink	soft_demap
123	1.6001	uplink	matrix_mult_4xX_complex
102	1.3269	uplink	cscale
91	1.1838	uplink	deque_xtract_top
68	0.8846	uplink	mmse_by_cholsolve_4xX_complex

Profiling result of the Pthreads version on the TILEPro64 processor

samples	%	app name	symbol name
2214	28.8019	uplink	.plt
1213	15.7799	libtile-cc.so.1	(no symbols)
859	11.1747	vmlinux	(no symbols)
844	10.9796	no-vmlinux	(no symbols)
405	5.2686	uplink	cmul
296	3.8507	uplink	matrix_a_a_hermite_plus_b_4xX_complex
285	3.7076	uplink	fft
281	3.6555	uplink	cadd
198	2.5758	uplink	ant_comb
182	2.3676	libpthread-0.9.29.so	(no symbols)
152	1.9774	libuClibc-0.9.29.so	(no symbols)
118	1.5351	uplink	matrix_mult_4xX_complex
91	1.1838	uplink	deactivate_cpu
86	1.1188	uplink	cscale
83	1.0797	uplink	soft_demap

```
80        1.0407  uplink                  cholsolve_4xX_complex
65        0.8456  uplink                  mmse_by_cholsolve_4xX_complex
```

The scheduling overhead of the Cilk version is about 22% of the total time. The scheduling overhead of the Pthread version is 29% takes about of a third of the total time. But the scheduling in the Pthreads version on the TILEPro64 processor is more efficient than the counterpart in the Cilk version, as the Pthreads version runs faster on the TILEPro64 processor.

8. Summary

The aim of this thesis project has been accomplished. The scalability of the Cilk language has been demonstrated and tested on two different multicore platforms, especially when seeking asynchronous parallelism. The ability of the work-stealing scheduler of the Cilk runtime system to dynamically schedule the workload on multicore platforms with very low overhead have also been investigated and approved. The comparison between the Cilk language and the POSIX Threads have been done, advantages and disadvantages of the Cilk language have been examined thanks to the comparison. Cilk is light-weighted and easy to program, with a strong ability to realize asynchronous parallelism. Cilk is aimed at parallel programming but not multithreading. Parallel programming is about explore the power of the processors that are available as much as possible to do one single task and this is what Cilk mainly do. The parallelism power of Cilk can be discovered as the number of cores increases, because Cilk can automatically take advantage of the power of the added cores without changing the code. The key advantage of Cilk is that Cilk eliminates the manual management of threads; the only thing the programmer needs to focus on is to discover the potential parallelism while the serial semantics of the program is preserved.

References

[1] Charles E. Leiserson, Harald Prokop: A Minicourse on Multithreaded Programming, MIT Laboratory for Computer Science, July 17, 1998.

[2] Robert D. Blumofe , Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, Yuli Zhou: Cilk: An Efficient Multithreaded Runtime System, In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207-216, 1995.

[3] Supercomputing Technologies Group at MIT Laboratory for Computer Science: Cilk 5.4.6 Reference Manual, 2011.

[4] Jin Zyren:Overview of the 3GPP Long Term Evolution Physical Layer, In *White paper 3GPPEVOLUTIONWP, Freescale Semiconductor*, July 2007.

[5] Magnus Själander: LTE Uplink Receiver PHY Benchmark, Department of Computer Science and Engineering, Chalmers University of Technology, November 2, 2011.

[6] Intel(R) Threading Building Blocks Getting Started Guide, Intel Corporation, October, 2011.

[7] Intel(R) Threading Building Blocks Getting Tutorial, Intel Corporation, October, 2011.

[8] Habanero Multicore Platforms, "http://habanero.rice.edu/Multicore_Platforms.htm"

[9] Habanero-C, "https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C"

[10] Yi Guo, Jisheng Zhao, Vincent Cave, Vivek Sarkar: SLAW: a Scalable Locality-aware Adaptive Work-stealing Scheduler.Department of Computer Science, Rice University, 2010.

[11] Karl-Filip Faxén: Wool-A work stealing library, ACM SIGARCH Computer Architecture News, Volume 36 Issue 5, December 2008

[12] LTE-an introduction, "www.ericsson.com/res/docs/2011/lte_an_introduction.pdf", Ericsson.

[13] Application Note 1MA111, UMTS Long Term Evolution (LTE) Technology Introduction, ROHDE&SCHWARZ, March, 2007.

[14] Anibal Luis Intini: Orthogonal Frequency Division Multiplexing for Wireless Networks, Department of Electrical and Computer Engineering at University of California Santa Barbara, December 2000.

[15] Agilent: Agilent MIMO Channel Modeling and Emulation Test Challenges, pg. 10, January 22, 2010, accessed September 16, 2011.

[16] Intel : Intel Cilk++ SDK Programmer's Guide, 2009.

[17] Tilera Corporation: TILEPro64 Processor Product Brief, 2011.

[18] Wentzlaff, D.;Griffin, P.; Hoffmann, H.; Liewei Bao; Edwards, B.; Ramey, C.; Mattina, M.; Chyi-Chang Miao; Brown, J.F.; Agarwal, A.; On-chip interconnection architecture of the TILE Processor, Micro, IEEE, Volume 27, Issue 5, 2007

A. Cilk Scripts

```
/*****************************************************************************
 *                       LTE UPLINK RECEIVER PHY BENCHMARK                   *
 *                                                                           *
 * This file is distributed under the license terms given by LICENSE.TXT     *
 *****************************************************************************
 * Author: Hao Li                                                            *
 *****************************************************************************/

#include "uplink.cilkh"
#include "uplink_verify.h"
#include <signal.h>
#include <cilk.h>
/* define some global variables */
static int subframe_id;
pthread_mutex_t mutexsum;
parameter_model pmodel;
CilkContext* context;
char* argv[] = { "f", "--nproc", "4", 0 };
int argc = 3;

cilk void chan_est(user_queue_item *user, int layer_i, int slot_id, int inner_id) {
  mf(&user->data->in_data[slot_id][3][inner_id][user->startSc],
     &user->data->in_rs[slot_id][user->startSc][layer_i],
     user->nmbSc, user->slot_items[slot_id].layer_data[layer_i][inner_id],
     &user->slot_items[slot_id].layer_items[layer_i].pow[inner_id]);
  ifft(user->slot_items[slot_id].layer_data[layer_i][inner_id], user->nmbSc,
       user->data->fftw[slot_id]);
  chest((user->slot_items[slot_id].layer_data)[layer_i][inner_id],
         user->slot_items[slot_id].layer_items[layer_i].pow[inner_id],
        user->nmbSc, (user->slot_items[slot_id].layer_data)[layer_i][inner_id],
        &user->slot_items[slot_id].layer_items[layer_i].res_power[inner_id]);
  user->slot_items[slot_id].R[layer_i][inner_id] =
    cmake(user->slot_items[slot_id].layer_items[layer_i].res_power[inner_id],0);
```

```
  fft(user->slot_items[slot_id].layer_data[layer_i][inner_id],
       user->nmbSc, user->data->fftw[slot_id]);
}

cilk void symbol_comp(complex *in[4], user_queue_item *user, int slot_id,
                      int layer_id, int index_out) {
  ant_comb(in, user->slot_items[slot_id].combWeight[layer_id],
           user->nmbSc, &user->symbols[index_out]);
  /* Now transform data back to time plane */
  ifft(&user->symbols[index_out], user->nmbSc,
       user->data->fftw[slot_id]);
}

cilk void layer_inner_loop(int first, int last, user_queue_item *user,
                           int slot_id) {
  int layer_i, layer_mid;
  if (last - first < LAYER_GRAINSIZE) {
    for (layer_i = first; layer_i <= last; ++layer_i) {
      spawn chan_est(user, layer_i, slot_id, 0);
      spawn chan_est(user, layer_i, slot_id, 1);
      spawn chan_est(user, layer_i, slot_id, 2);
      spawn chan_est(user, layer_i, slot_id, 3);
    }
  } else {
    layer_mid = (first + last)/2;
    spawn layer_inner_loop(first, layer_mid, user, slot_id);
    spawn layer_inner_loop(layer_mid+1 ,last, user, slot_id);
  }
  return;
}

cilk void layer_loop(userS *user, int first, int last,
                     user_queue_item *user_item, int slot_id) {
  int layer_id, rx, sc, ofdm;

  spawn layer_inner_loop(first, last, user_item, slot_id);
  sync;

  uplink_layer_verify(user->subframe, user_item->slot_items[slot_id].layer_data,
                     user_item->slot_items[slot_id].R,user_item->nmbSc,
```

```
                    user_item->nmbLayer, slot_id);
  comb_w_calc(user_item->slot_items[slot_id].layer_data, user_item->nmbSc,
              user_item->nmbLayer, user_item->slot_items[slot_id].R,
              user_item->slot_items[slot_id].comb_w);

  for (layer_id = 0; layer_id < user_item->nmbLayer; layer_id++)
    for ( rx = 0; rx < RX_ANT; rx++ )
      for ( sc = 0; sc < user_item->nmbSc; sc++ )
user_item->slot_items[slot_id].combWeight[layer_id][sc][rx] =
        user_item->slot_items[slot_id].comb_w[sc][rx][layer_id];

  uplink_weight_verify(user->subframe, user_item->slot_items[slot_id].combWeight,
                       user_item->nmbSc, user_item->nmbLayer, slot_id);

  for (layer_id = 0; layer_id < user_item->nmbLayer; layer_id++) {
    int ofdm_count = 0, index_out;
    complex* in[4];
    for (ofdm = 0; ofdm < OFDM_IN_SLOT; ofdm++) {
      /* Collect the data for each layer in one vector */
      if (ofdm != 3) {
in[0] = &user_item->data->in_data[slot_id][ofdm][0][user_item->startSc];
in[1] = &user_item->data->in_data[slot_id][ofdm][1][user_item->startSc];
in[2] = &user_item->data->in_data[slot_id][ofdm][2][user_item->startSc];
in[3] = &user_item->data->in_data[slot_id][ofdm][3][user_item->startSc];
/* Put all demodulated symbols in one long vector */
index_out = user_item->nmbSc*ofdm_count + slot_id*(OFDM_IN_SLOT-1)*
                    user_item->nmbSc
       + layer_id*2*(OFDM_IN_SLOT-1)*user_item->nmbSc;
spawn symbol_comp(in, user_item, slot_id, layer_id, index_out);
ofdm_count++;
      }
    }
  }
}

cilk void slot_loop(userS *user) {
  user_queue_item user_item;

  user_item.nmbLayer = user->nmbLayer;
  user_item.startRB  = user->startRB;
```

```
  user_item.nmbRB    = user->nmbRB;
  user_item.mod      = user->mod;
  user_item.startSc  = user->startRB*SC_PER_RB;
  user_item.nmbSc    = user->nmbRB*SC_PER_RB;
  user_item.data     = user->data;

  spawn layer_loop(user, 0, user->nmbLayer, &user_item, 0);
  usleep(DELTA/2);
  spawn layer_loop(user, 0, user->nmbLayer, &user_item, 1);
  sync;

  user_item.nmbSymbols = 2*user->nmbLayer*user_item.nmbSc*(OFDM_IN_SLOT-1);
  uplink_symbol_verify(user->subframe, user_item.symbols,
                       user_item.nmbSymbols);
  interleave(user_item.symbols, user_item.deint_symbols,
             user_item.nmbSymbols);
  uplink_interleave_verify(user->subframe, user_item.deint_symbols,
                           user_item.nmbSymbols);

  soft_demap(user_item.deint_symbols,
          user_item.slot_items[1].layer_items[user_item.nmbLayer-1].pow[0],
          user_item.mod, user_item.nmbSymbols, user_item.softbits);

  user_item.nmbSoftbits = user_item.nmbSymbols * user_item.mod;
  uplink_verify(user->subframe, user_item.softbits, user_item.nmbSoftbits);
}

cilk void user_loop(int first, int last, userS **users) {
   int user_id, user_mid;
   if(last - first < USER_GRAINSIZE) {
     for(user_id = first; user_id <last; user_id++) {
        /* start spawning processing each user */
       spawn slot_loop(users[user_id]);
     }
   }
   else{
    user_mid = (first + last) / 2;
    spawn user_loop(first, user_mid, users);
    spawn user_loop(user_mid, last, users);
   }
```

```
}

cilk int dosubframe() {

  user_parameters *parameters;
  int nmbUsers, i;
  userS **user_array;

  parameters = uplink_parameters(&pmodel);
  user_array = malloc(MAX_USERS*sizeof(userS *));
  nmbUsers = 0;
  while (parameters->first) {
    user_array[nmbUsers++] = parameters->first;
    parameters->first = parameters->first->next;
  }

  i = 0;
  spawn user_loop(0, nmbUsers, user_array);
  sync;

  for (i=0; i<nmbUsers; i++) {
    free(user_array[i]);
  }
  free(user_array);
  free(parameters);

  pthread_mutex_lock(&mutexsum);
  subframe_id--;  /* subtract subframe_id by 1 when one subframe is done */
  pthread_mutex_unlock(&mutexsum);
  return subframe_id;
}
//the following code are used to call a Cilk procedure
//  from a C function indirectly

int EXPORT(dosubframe) (CilkContext *const context);

//signal handling function
int handle(int sig){
  int y;
  y = EXPORT(dosubframe)(context);
```

```
  pthread_mutex_lock (&mutexsum);
  subframe_id++;
  pthread_mutex_unlock (&mutexsum);
  return 0;
}

cilk void main(int argc, char *argv[]) {
  subframe_id = 0;

  init_data();
  init_verify();
  pthread_mutex_init(&mutexsum, NULL);
  init_parameter_model(&pmodel);
  crcInit();
  context = Cilk_init(&argc, argv);
  signal(SIGALRM, handle); //set a function to handle the SIGALRM signal
  ualarm(DELTA, DELTA); //start a new alarm in DELTA microseconds

  while(subframe_id < 10)
  {
   ;
  }
  Cilk_terminate(context);
  return;
}
```