Pipelined Execution of LTE Algorithms

Ankur Krishna Gautam
Indian Institute of Technology,
BHU

Roya Alizadeh
Ecole polytechnique De
Montreal
Montreal, Quebec

ABSTRACT
To study the latest cellular mobile communication standard and protocols, there should be a publicly available simulation environment to support researchers of the world to do testing, reformulation of the existing technologies and optimization on a common well known platform [1],[2],[3]. A possible solution is offered by open source simulation environment from TUWIEN that supports link and system level simulations of the Universal Mobile Telecommunication System (UMTS) Long-Term Evolution (LTE) and which is specifically designed to accentuate reproducibility [1]. In this paper, we pipelined execution of the LTE algorithms on blade servers to achieve minimum latency caused during communication [4]. And then the pipeline model is compared in different context including a single (multi-core) processor, multiple processors on a single board, multiple processors on different boards on a single server and multiple processors on two different servers and etc. Moreover, we obtained the ratio of the communication and the computation time in the execution of LTE algorithms in the TUWIEN simulator by profiling which provides the best preferable configuration for the LTE algorithms. Finally, LTE processes are distributed to desirable processors by using OpenMPI to decrease latency.

General Term
openMPI,profiling,blade servers

Keywords
Pipeline model, TUWIEN LTE simulator

1. INTRODUCTION
TUWIEN System level simulator will be the standard platform for our work. The simulator can be used as a reference to validation of algorithms, generate LTE signals, analyze the LTE algorithms. To keep the code maintainable and well functional, the simulator is coded in Matlab though there is no compromise with the computational power and execution time as the simulator uses MEX files for highly computational and repetitive tasks which are not easy to vectorized along with that it is based on the Parallel Computing Toolbox of Matlab which further helps to provide parallelization to some more extent. But still due to the overhead of the high level language like Matlab, it is very exciting to think in terms of parallel models and configuration for the distribution of LTE algorithms.

So, as a preliminary part of the big umbrella, our work will be analysis and profiling of the Matlab code to obtain the representative amount of time and data transfer and then visualize the various LTE steps or algorithms only with the concerned processing time and the data transfer. And once we have the involved data and time, we can do create models based on parallelization using Inter Processing Communication (IPC) and can test our model on various configurations including single (multi-core) processor, multiple processors on same socket, processors on different sockets, cluster of server and etc. and the This will further make us able to compare the processing time of various models on various configurations. Furthermore, we can know the actual latencies, overhead and compromises involved in distributing the LTE algorithms. Moreover, we can compare the communication and the computation time in the overall execution time on different configurations, and we can also find a relation between communication time and amount of data transferred.

Though there has been lot of work on implementing components of LTE on GPUs [5] and implementation of LTE parts on multi core platforms [7],[8],[9].Our work is unique in the sense as it directly focuses on latencies cause while distributing the LTE algorithms on various configurations. We have access to two blade servers having 160 cores each (8 sockets each having 10 cores and hyper threading enabled). We also propose the best and the worst case sceneries with respect to the Non Uniform Memory Access (NUMA) architecture and the variation in the execution time. At the end of the paper we provide detailed results of the profiling in terms of the computation and communication time.

The structure of the paper is as follows: The section 2 is aimed for the purpose of building a comprehensible platform with the current technologies. In subsection 2.1 we define the features of LTE in general and put TUWIEN LTE simulator in light along with description of its components. Subsection 2.2 describes the Message Passing Interface, the required library and the way of distributing processes to desirable processors. In section 3 we describe our contribution. Subsection 3.1 consists of our approach to do profiling of the Matlab based TUWIEN simulator and in subsection 3.2 we define the pipeline scheme in detail. Subsection 3.3 deals significantly with NUMA and rank file in OpenMPI to distribute processes on processors. Section 4 shows conclusions of our work along with the future directions. Section 5, 6 stands for the acknowledgement and references respectively.

2. BACKGROUND

2.1 Introduction to LTE
LTE is a standard for wireless communication of high speed data for mobile phones and data terminals. It is based on the GSM/EDGE and UMTS/HSPA network technologies, increasing the capacity and speed using a different ratio interface together with core network improvements based on the feedback from user side.
Peak download/upload data transfer rates, low latency over handover and connection set up time, improved support for mobility, OFDMA, SC-FDMA, simplified architecture, packet switched radio interface, MIMO data processing, user's feedback utilization, scheduling at base station to improve

channel quality, appropriate level of security are some of the main features of LTE. The low latency, improved coverage, flexible spectrum deployment, scalable bandwidth, co-existence with 3GPP Radio Access Technologies, low cost, improved services are some of the motivation factor behind the development of the LTE.
To provide a standard testing platform to the researchers TUWIEN developed LTE simulator available with non-commercial academic use license and written in Matlab and requires the following packages: Matlab 7.8.0 (2009a), statistics toolbox, image toolbox, mapping toolbox. The Matlab simulator is available for both System level and link level supporting both uplink and downlink. Higher order of optimization has been done using MEX library to perform operations in the C/C++ environment,vectorization, parallelization using Matlab Parallel Computing toolbox, calculating and saving the result for some repetitive tasks such as ENodeB-dependent large scale pathloss maps, site dependent shadow fading maps and time dependent small scale fading maps which highly reduces the computational load.In the development and standardization of LTE, as well as in the implementation process of equipment manufacturers, simulations are necessary to test and optimize algorithms and procedures. This has to be carried out on the physical layer (link level) and in the network (system level) context.

2.1 Link level simulator:

It focuses on channel estimation, tracking and predicting algorithms as well as synchronous algorithms [10, 11], MIMO gains, adaptive modulation and coding. Receiver structure neglects inter-cell interference and impact of scheduling as this increases simulation complexity and execution time to the larger extent. This simulator also deals with modeling of channel encoding and decoding.

2.2 System level simulator:

It focuses more on network related issues such as resource allocation and scheduling, multi user handling, mobility management, admission control, interference management and network planning optimization. It also calculate different types of fading maps including shadow fading, path loss and micro scale fading maps.

We will mainly be dealing with the system level simulator for our testing and profiling purpose. We will divide the whole source code into 31 steps and then will do the profiling of each step to obtain the equivalent amount of time and amount of data transferred.

2.3 Introduction to MPI

MPI primarily addresses the message-passing parallel programming model. Data is moved from the address space of one process to other process through cooperative operations on each process. This interface attempts to be practical, portable, efficient and flexible. MPICH2 and OpenMPI are two biggest implementation of MPI.Figure 1 illustrates the structure of an OpenMPI program

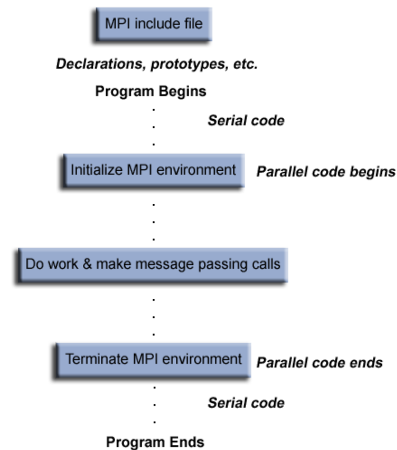

Fig 1 Structure of an openMPI program [11]

There are 2 types of communication routines in MPI:

I. Point to point communications routines :

MPI point-to-point operation typically involves message passing between two, and only two, different MPI tasks. One send is performing a send operation and the other task is performing a matching receive operation.

openMPI supports different mechanism for the send and receive functions including synchronous send, blocking send/receive, on-blocking send/receive, buffered send, combined send /receive, ready send etc. A blocking send routine will only return its execution after it is safe to modify the application buffer for reuse or in other words data has arrived and is ready for use by the program. A blocking send may be synchronous or asynchronous depending upon the fact that if it is waiting for the handshaking with the receive task or using a system buffer to hold the data for eventual delivery respectively.

II.Collective communication :

It involves more than two processes at the same time. Any collective communication routine must involve all process within the scope of a communicator. Figure 2 illustrates collective communication in OpenMPI.

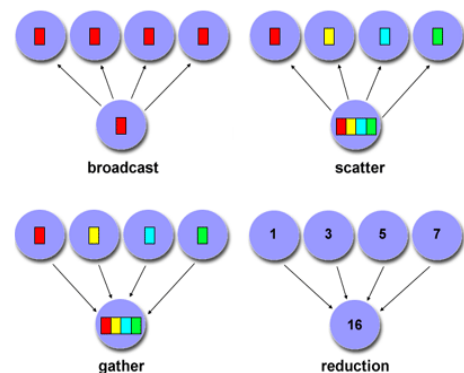

Fig 2 Collective communication in openMPI[11]

3. OUR CONTRIBUTION

We will define our contribution in 3 terms. First we will consider TUWIEN Matlab System level simulator as the standard for testing LTE algorithms then start profiling the

large Matlab source code. Since the source code is larger enough, it is not easy to analyze as a whole and to divide the source code in various algorithms or the functional calls that we call a STEP seems logical. We divide the System Level source code in 31 steps in which some steps occurred more than once depending upon if they are called in a loop construct. Once we have steps, we can start thinking in terms of different steps connected via a sequential pipeline. Profiling comes into the picture when we need to have the execution time of the every step and data transferred for the steps to be executed which is equivalent to the summation of input and output size, if the step is considered as a function call.

The system Level simulator is divided into following steps:

- Load Params
- Load Default Params
- Load BLER curves
- Generated enodeB network
- Get neighbor nodes in a loop construct
- Generate shadow fading
- Determine cell capacity
- Generate users
- Calculate small scale fast fading
- SINR averaging algorithm
- Trace, attach users to sectors , Downlink models
- Main simulation loop

3.1 Profiling of the simulator

Execution time of the steps :

There are three approaches for determining the execution time of a step

1. We can count the number of instructions or operators in the step and can divide by IPS (Instruction per Second) to get the time elapsed during the execution. The best way to count the instruction is to use some extra variables and counters .But since the Matlab code has a huge overhead even for each instruction, the time taken in using variables and counters throughout the code of the step will be significant.

2. We can use inbuilt profiling functions of Matlab by executing **profiling on** and then **profile viewer** after the execution. Profiling will record all the information related to processing time for each function call, it encounters. A step needs not to be a function call every time .It might be a file loading call or some computational instruction set other than that the lesser flexibility option makes the method less appropriate.

3. Using clock() to get the difference in time between the code starts and code end.

In Matlab , we are provided with the very user friendly functions tic[to start clock] and toc[to take elapsed time at a given interval]

```
%test program to measure execution time
step=tic;
t_start=toc(step);
s;
t_finish=toc(step);
```

$$test_time = t_finish - t_start \quad 1.1$$

Where s is a piece of source code .
One problem is that the overhead in computing test_time is not taken into account. This overhead involves at least the time of two calls to the clock function (which in our case are tic & toc) and the assignments to t_start & t_finish. The overhead can be found by running the following control program.

```
%control program
step=tic;
t_start=toc(step);
% this line is empty.It is a null program
t_finish=toc(step);
```

$$control_time = t_finish - t_start \quad 1.2$$

Then actual running time of s is then the difference
$$s_time = test_time - control_time \quad 1.3$$

Some other source of inaccuracy is due to interferences from the underlying operating system. Disabling interrupts during a timing run will assure that no operating systems tasks preempt the test or control tasks. However, it is always not possible to disable all system interrupts.
For example, the clock itself is often driven by periodic **tick** interrupts.
Finally, there are significant errors due to the tick granularity and accuracy of the clock.
Let Ct be the time returned from the clock function,p be the tick period or tick interval, and rt be the perfect real time (without overhead).
If the clock is called at real-time rt, then the time Ct returned will satisfy the relation

$$Ct = rt - \delta p, 0 \leq \delta < 1 \quad 1.4$$

Consider now the difference ΔCtobtained from two calls of clock:

$$\Delta Ct = \Delta rt \mp \epsilon , \epsilon < p \quad 1.5$$

Thus the difference between two clocks times will have an error bounded by p.The s_time from equation 1.3 will then have a maximum error of $2p$, since it is the difference between two differences.
To overcome these problems, to execute the piece of code many times seems a good option.

```
%test program 2
step=tic;
t_start=toc(step);
for i=1:n
s;% piece of code to be analyzed
end
t_finish=toc(step);
test_time=t_finish-t_start;
```

```
And hence,
```

$$test_{time} - control_{time} = n * s_{time} + error \quad 1.6$$

Or,

$$S_{time} = \frac{test_{time} - control_{time}}{n} + \frac{error}{n} \quad 1.7$$

And if the loop count is sufficiently large, the error term can be ignored. [12]

Data transfer between steps :

To list the used variables in Matlab workspace, with sizes and types, there is a function provided with name Whos(variable_name).
So if we consider a step to be more generally like a function, data transfer will be equivalent to the size of inputs and outputs of the function.
e.g. consider the step of loading parameters for the simulators.

```
function LTE_config
=LTE_load_params(varargin,param)

%saperately consider the case of fixed
number of inputs
%inputs and variable number of inputs

s1=whos('varargin');
s2=whos('param');
inputsize=s1.bytes-nargin*112+s2.bytes;
% set the return value i.e. LTE_config
s3=whos('LTE_config');
outputsize=s3.bytes;

end
```

And the overall code is modified in such a way that a text file having the details of execution time and data transferred is generated for each simulation.
And once we have the details, we can start thinking in terms of virtual computation and communication. For each step,

$$Tcomputation = processing\ time\ of\ step$$

$$Tcommunication = communication\ time\ of\ step$$

3.1 Pipeline model

Once we have the text file as a result of simulation, having data transferred & execution detail of each step, we can visualize the source code with the computational sleep for the representative amount of time and virtual transfer of the represented amount of data. We can use some models using Inter Process Communication to distribute the steps between different processors. We can also use core & socket affinity for any particular step using some flags during the $mpirun$ command. We can think of these models while writing c program: The text file generated from the simulation is accessed by all the processes to get the execution time and corresponding data transfer for each step (they are 31 in number).

Root processes initiate the pipeline and other processes propagate by doing the corresponding computational sleep and receiving, sending the data.

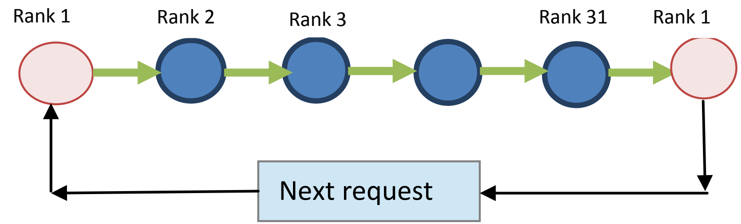

Fig 3 Pipeline model

If the execution times of different processes are $t0, t1, t2$ in order of the rank, the overall latency can be found by $t31 - t0$.And we can repeat the flow of execution to get the high accuracy of the latency per execution.

This scheme is implemented by maintaining two integers left & right which governs the direction of data transfer such that each process have to receive data from the process with rank equal to the left and send data to the process with rank equal to the right .And root process starts its task by first making a sleep and sending data to the next process (i.e. process with rank 1) and other processes propagates similarly.

Socket/core affinity:

Since all the cores present in a socket are connected by a memory pipeline, it is more convenient to transfer data between the processes executing on the cores of a same socket .Hence there could be a high variation depending upon the availability of processes to cores & sockets.

We are provided several functionalities in openMPI to choose the core and socket affinity for a particular step depending upon the characteristics. We can use a rankfile to define the affinity for each process.

When there are 4 processes

$$rank0 = host0\ slot = 2$$
$$rank1 = host1\ slot = 4 - 7{,}0$$
$$rank2 = host0\ slot = 1{:}0$$
$$rank3 = host1\ slot = 1{:}2 - 3$$

Where host1, host2 are the addresses of the blade servers.
If we are running the program on same server, we can define localhost at the place of name of the host. Slot 1:0 means the core 0 of the socket 1.

Running on cluster:

A computer cluster consists of a set of loosely connected or tightly connected computers that work together so that in many respects they can be viewed as a single system.SSH is used as the basis of communication between the computers. Later we used to NUMA aspects to distinguish between the distribution of the processes to the processors.

OpenMPI supports execution of program on cluster using a hostfile.

```
# hostfile
# The Hostfile for Open MPI
# The master node, 'slots=160' is used
because it has 160 cores.
Srvgrm04 slots=160
# The following are the slave nodes
Srvgrm06 slots=160
```

Once we set the password less ssh between the servers, we are ready to taste out MPI program on cluster. And we also have these functionalities in openMPI implementation to distribute processes among the processors of each server independently.

-npernode, --npernode <#pernode>
On each node, launch this many processes.

-pernode, --pernode
On each node, launch one process -- equivalent to -*npernode* 1.

To map processes to nodes:

-loadbalance, --loadbalance
Uniform distribution of ranks across all nodes.

-nolocal, --nolocal
Do not run any copies of the launched application on the same node as orterun is running. This option will override listing the localhost with **--host** or any other host-specifying mechanism.

-nooversubscribe, --nooversubscribe
Do not oversubscribe any nodes; error (without starting any processes) if the requested number of processes would cause oversubscription. This option implicitly sets "max_slots" equal to the "slots" value for each node.

-bynode, --bynode
Launch processes one per node, cycling by node in a round-robin fashion. This spreads processes evenly among nodes and assigns ranks in a round-robin, "by node" manner.

3.3 Introduction to NUMA

Non-Uniform Memory Access (**NUMA**) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors).

Architecture of NUMA could be like this. The processor connected to the bus or crossbar by connections of varying thickness/number. This shows that different CPUs have different access priorities to memory based on their relative location.

System's architecture can be examined using the lstopo utility, part of the hwloc library, which comes with Open MPI. Unfortunately lstopo cannot tell you how fast each memory channel is and how fast/latent the links between the NUMA nodes are. Since it is dependent on the relative location, thinking in terms of the processor/socket affinity becomes crucial . And it drives us to know the worst case and best scenario of our pipeline scheme. **Server has 8 sockets each having 10 cores .**When all the processors of all sockets of the blade server are used or all the processes are going to be executed on the processors of a single socket are worst case and best case scenarios respectively.

Best case:

$--bind--core$**:** This will result in the distribution of processes on cores in a sequence.

So during the whole scheme very less data transfer will be occurred between the processes executing on two different sockets.

```
mpiexec --report-bindings --bind-to-core --bycore
[hostname:39646] MCW rank 0 bound to socket 0[core 0]: [B . . . . .][. . . . . .]
[hostname:39646] MCW rank 1 bound to socket 0[core 1]: [. B . . . .][. . . . . .]
[hostname:39646] MCW rank 2 bound to socket 0[core 2]: [. . B . . .][. . . . . .]
[hostname:39646] MCW rank 3 bound to socket 0[core 3]: [. . . B . .][. . . . . .]
```

Worst case

$-bind--core--bysocket$: This will result in distribution of process on sockets such that each data transfer will be occurred between the processes executing on two different sockets.

```
mpiexec --report-bindings –bind-to-core --bysocket
[hostname:39646] MCW rank 0 bound to socket 0[core 0]: [B . . . . .][. . . . . .]
[hostname:39646] MCW rank 1 bound to socket 0[core 1]: [. . . . . .][B . . . . .]
```

For seeing the binding of processes and processors, there is a function being provided in openMPI implementation i.e. $--report-binidings$

Other possible cases include

$--bind--socket--bysocket$:

```
mpiexec --report-bindings --bind-to-socket --bysocket
[hostname:13888] MCW rank 0 bound to socket 0[core 0-5]: [B B B B B ][. . . . . ]
[hostname:13888] MCW rank 1 bound to socket 1[core 0-5]: [. . . . . ][B B B B B ]
[hostname:13888] MCW rank 2 bound to socket 0[core 0-5]: [B B B B B ][. . . . . ]
[hostname:13888] MCW rank 3 bound to socket 1[core 0-5]: [. . . . . ][B B B B B ]
```

4. CONCLUSION/RESULTS

For our pipeline model we make the code sleep for the representative amount of time as a computational task and send virtual data equal to the representative amount of data transferred in the step as a communication task. We run our pipeline scheme on several configurations and note the overall processing time and to measure the latency. Detailed Result of the profiling of the simulator can be seen in the files generated as the result of the execution of the modified simulator. We introduce two new terms: $Tcomm$ is the time elapsed in data transfer while $Tcomp$ denoted the sleep time.

Table 1 Processing time on various configurations

Configuration	Execution time
A single (multi-core) processor	72.0325
Multiple processors on a single board	72.0684
Processors on different boards in a single server	72.2783
Processors on two different server	74.1627

We can compare the best and worst case of processor/socket affinity:

Table 2 Best/worst case scenarios

Case	$Tcomm$(sec)	$Tcomp$(sec)
Best	48.23	24.0116
Worst	48.86	24.0052

Based on the above results, below conclusion can be made.

1. There is a linear relation between the communication time (time taken in transferring data) and amount of data. The relation can be expressed like this,

$tcomm = tl + tc * data$

Wheretl and tc are constants for the relation.

And the values are: $tl = 0.0001 tc = 2.44$

2. Error induced and the overhead by using clock() in Matlab profiling can be minimized to the minimum level by considering the time taken in calling the clock() and by running too many instances of the same code.

3. There is a latency of 0.19 seconds in distributing LTE algorithms on multiple sockets.

5. ACKNOWLEDGEMENT

I am also very thankful to MITACS organization for funding and the priceless guidance from my mentor Mr. Normand Bélanger must be appreciated heartily. I especially like to acknowledge the motivational environment and the limit less support of all the members of GRM Lab. Special Thanks to Prof.O.P.Singh for timeless encouragement.

The source code of the modified simulator /makefiles/results can be found at http://sourceforge.net/projects/pipelinelte/

6. REFERENCES

[1] Christian Mehlführer, Josep Colom Colom Ikuno, Michal Šimko, Stefan Schwarz, Martin Wrulich and Markus Rupp2011. The Vienna LTE simulators – Enabling reproducibility in wireless communications research

[2] Markus V. S. Lima1 , Camila M. G. Gussen1 , Breno N. Esp´ındola1 , Tadeu N. Ferreira2 , 3,1 Wallace A. Martins , Paulo S. R. Diniz1.OPEN-SOURCE PHYSICAL-LAYER SIMULATOR FOR LTE SYSTEMS in 2012

[3] Bum-Gon Choi, Jun Suk Kim, and Min Young Chung.Development of a System-Level Simulator for Evaluating Performance of Device-to-Device Communication Underlaying LTE-Advanced Networks in 2012

[4] Inkeun Cho*, Tomasz Patykt, David Guevorkian+, Jarmo Takala§, and Shuvra Bhauacharyya.Pipelined FFT for Wireless Communications Supporting 128-2048 / 1536 - Point Transforms

[5] Dipl.-Ing Michal Šimko,Dipl.-Ing. Mag. Dr.techn. Sebastian Caban, March 2011. Implementation of LTE mini receiver on GPUs. .

[6] Agilent Technologies,2009.3GPP Long Term Evolution,System overview,product development and test challenges.

[7] Agilent Technologies,2009.Introducing LTE-Advanced

[8] Julien Heulot , Jani Boutellier , Maxime Pelcat , Jean-Franc ois Nezan , Slaheddine Aridhi.Applying the Adaptive Hybrid Flow-Shop Scheduling Method to Schedule a 3GPP LTE Physical Layer Algorithm onto Many-Core Digital Signal Processors in 2013

[9] Maofei He, Jiajie Zhang, Wenhua Fan, Zhiyi Yu*, Xiaoyang Zeng.A Channel Estimator for LTE Downlink Mapped on a Multi-Core Processor Platform

[10] Anas Showk ,Attila Bilgic .A Novel Scheduling Methodology Based on SDL Process Migration for the LTE Higher Layer Protocol on Multi-Core Mobile Terminals

[11] Messaging passing Interface Tutorial, https://computing.llnl.gov/tutorials/mpi/

[12] Real-Time Systems and Software by Alan C. Shaw