Illinois, and the Pervasive Parallel Laboratory at Stanford propose a two-layer framework, consisting of the productivity layer where domain experts, assumed to have limited experience with parallel programming, can focus on application development, and the efficiency layer where computer scientists with strong background in parallel programming can focus on improving the efficiency of the application [5].

Similar to the aforementioned framework, programming models such as algorithmic skeletons have also been proposed, aiming to benefit from multi-core architectures while decoupling the hassle of thread management from common programming. Skandium and Calcium [4] provide high-level parallel programming libraries based on the thread pool and ExecutorService frameworks in JAVA. Users only need to provide a threshold for threads and a set of initial parameters.

Daniel C. Doolan, and Laurence T. Yang in year 2006. They considered the problem of matrix multiplication to show and demonstrates that mobile devices are capable of parallel computation using Mobile Message Passing Interface (MMPI). MMPI allows parallel programming of mobile devices over a Bluetooth network [2].

Panya Chanawangsa, and Chang Wen Chen in year 2012. They demonstrate how proper utilization of a dual-core mobile processer can achieve tremendous speedup in mobile application [1].

4. Development Platform

The system was developed on Android 4.1.2 (Jelly Bean). Released in September 2012. The phone's most outstanding feature is its processor – a superscalar quad-core 1.4 GHz Arm Cortex-A9 with 2 GB RAM [8]. Optimized for high performance and low power consumption, the Galaxy SIII is indeed an ideal platform for this system.

5. Matrix Multiplication

Matrix multiplication is a time consuming operation because for an $n \times n$ matrix the best possible time complexity is $O(n^2)$ it cannot be less than this as all n^2 cells must be visited. Assume two matrices are to be multiplied, if A is an $n \times m$ matrix and B is an $m \times p$ matrix, the result AB of their multiplication is C $n \times p$ matrix defined only if the number of columns m in A is equal to the number of rows m in B. When multiplying matrices, the elements of the rows in the first matrix are multiplied with corresponding columns in the second matrix. One may compute each entry in the third matrix one at a time [2][3]. For two matrices

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}$$

(where *necessarily* the number of columns in **A** equals the number of rows in **B** equals m) the matrix product **AB** is defined by

$$\mathbf{AB} = \begin{pmatrix} (AB)_{11} & (AB)_{12} & \cdots & (AB)_{1p} \\ (AB)_{21} & (AB)_{22} & \cdots & (AB)_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (AB)_{n1} & (AB)_{n2} & \cdots & (AB)_{np} \end{pmatrix}$$

where **AB** has entries defined by

$$(AB)_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}. \qquad (1)$$

The implementation of matrix multiplication in application code is generally a question of three loops as shown in Figure (1).

```
private void mult(int a[][],int b[][],
int c[][])
{for (int i = 0; i < a.length; i++)
{for (int j = 0; j < b[0].length; j++)
{for (int k = 0; k < a[0].length; k++)
{c[i][j] += a[i][k] * b[k][j];}}}}
```

Fig. 1 Method to calculate product of two matrices.

6. Parallel Processing

Parallel processing involves multiple processes which are active simultaneously and solving a given problem, generally on multiple processors. We are underplaying the role of physical multiprocessing here, because the study of parallel processing does not require multiple processors. Most of OS today provide multiprocessing / tasking, which can be exploited to study the issues and programming aspects of pp. But, we must have multiple processes (or rather, independent execution units) which are simultaneously active. These units take the role of different processing units (or processors). The critical aspect here is "solving a given problem". The processes must all be concerned with the solution of one single problem.

In other words, there must be interaction among the units. For example, one unite compiling file1 and another compiling file2, will not be considered as parallel processing, because these two are absolutely independent tasks according to the definition of C language. But, if one

unit is compiling a statement from file1 and the other is compiling the next statement from the same file, they need to share variable declaration and scoping information, properly between them. Hence they are solving a single problem and not two separate problems. We can, therefore, consider that as parallel processing [9].

7. Multithreading and Processor Utilization

Since a mobile phone is considered a general-purpose device, application-level parallelism is the best we can achieve. Without explicitly using multiple threads, speedup from a multi-core architecture will not be obvious. In this paper, we propose a general guideline for breaking down a global task into multiple subtasks and later demonstrate how to apply this idea on a matrix multiplication task.
The first step towards parallelizing a task is to determine the optimal number of threads to use. Limiting thread contention is crucial for application speedup. Spawning too many threads than necessary not only disrupts other applications, but may also result in a longer execution time of the application due to the overhead associated with context-switching. A processor core can handle only one thread at a time. For efficiency purposes, a simple rule is to spawn as many threads as the number of cores available, thereby delegating one thread to each core and eliminating the need for time-slicing. A simulation was conducted by spawning different numbers of threads to execute certain tasks. A dramatic improvement in execution time can be seen when we increase the number of worker threads from 1 to4. However, since there are only four available cores, increasing the number of threads does not enhance but aggravates the performance, resulting in a slightly longer execution time.

8. Matrix slicing

The next step is to determine if data parallelism is possible and appropriate. For this work, matrix multiplication, data comes in the form of matrices, or on the lower level 2-dimensional arrays. In many cases, they can be split up into smaller independent chunks and processed concurrently, reducing the execution time while producing the same output as when processed sequentially. For instance, to multiply two matrices A and B , we can split matrix A up into smaller sub-matrices (matrix slicing) and multiply each sup-matrix from matrix A with matrix B concurrently. However, if the size of data to be processed is not significantly large, employing data parallelism will not yield much speedup as a result of thread overhead. In the case of this application, since the matrices are large (1500×1500), data parallelism is well worth a try. Generally speaking, given k processor cores, we should divide the input data of size n into n/k smaller chunks and distribute them across k cores with each core running a single thread. Figure (2).

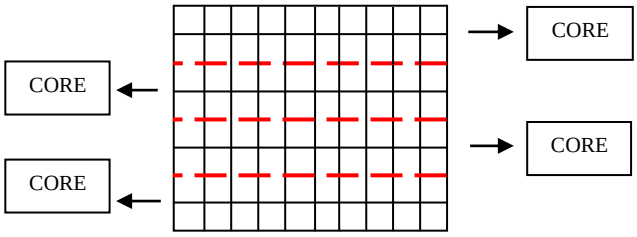

Fig. 2 Matrix slicing

9. Speedup

Speedup is the expected performance benefit from running an application on a multi-core versus a single-core machine. When speedup is measured, single-core machine performance is the baseline. For example, assume that the duration of an application on a single-core machine is six hours. The duration is reduced to three hours when the application runs on a quad machine. The speedup is 2-(6/3)- in other words, the application is twice as fast [7]. Speedup can be found by using the formula below

$$\text{Total Speedup} = T_s / T_p \quad (2)$$

T_s : is the runtime without parallelism.

T_p : is the runtime with parallelism.

10. Putting It All Together: The Proposed System In Action

For the parallel computation we consider that the matrices are square. The parallel computation requires several steps that are not required in the sequential version of the application (Figure 3). One of the first main differences is that we need to determine the number of threads suitable for the phone hardware. Since Samsung Galaxy SIII has four cores, four threads are ideal for execution the task in hand. So that the matrix must be split in to four same size blocks called sub-matrix, then send each sub-matrix to one of the four cores we have. Once this operation has been completed each core can compute its own section of the matrix. The final stage is to gather all the results back in to one matrix. The result of the computation can then be available for the user.
To gain full control of task management, we also make use of the FutureTask class , allowing us to track the progress of the submitted tasks and block until all of them have been completed. The four tasks are eventually submitted to

an ExecutorService, which takes care of thread pool creation and assigns a submitted task to an available thread. More importantly, by using the ExecutorService, memory consistency is guaranteed, thus eliminating the trouble of thread synchronization. Figure (4).

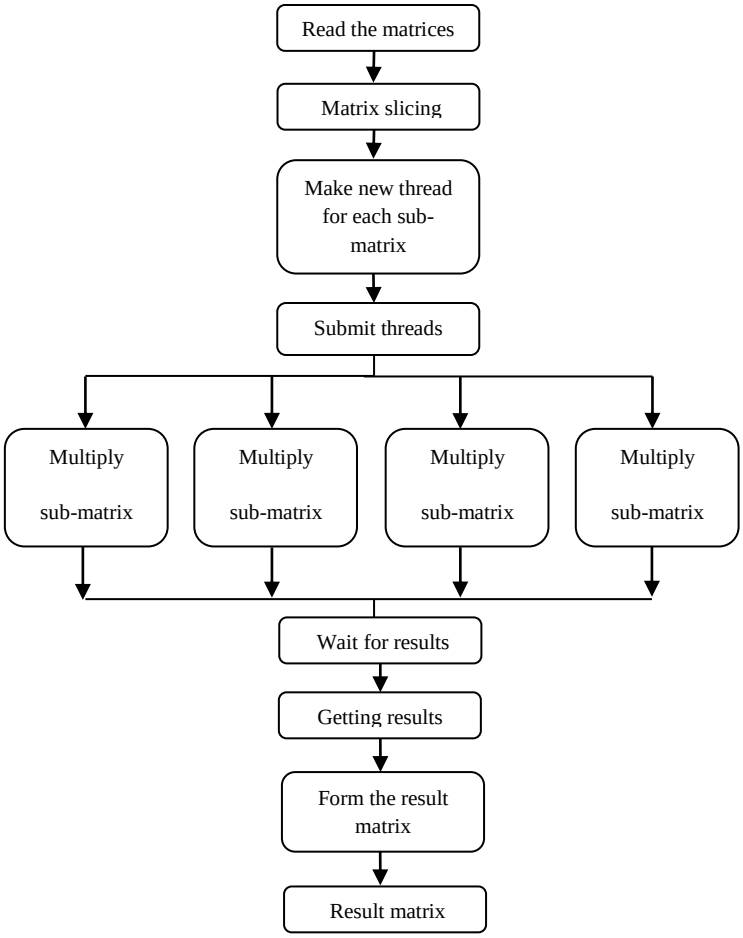

Fig. 3 Overall functionality of the system.

11. Experimental Results

The system tested on different size matrices, The results are shown in (Table (1)) below:

Table 1: Experimental results

Matrix size	Time in sequential	Time in parallel
500×500	7435ms	2466ms
1000×1000	80497ms	22645ms
1500×1500	319766ms	101394ms

```
Final ExecutorService es;
int nsize=size/4;
int[][] a1=new int[nsize][size];
int[][] a2=new int[nsize][size];
int[][] a3=new int[nsize][size];
int[][] a4=new int[nsize][size];
for(int i =0;i<nsize;i++)
for(int j=0;j<size;j++)
{a1[i][j]=a[i][j];
a2[i][j]=a[i+nsize][j];
a3[i][j]=a[i+nsize*2][j];
a4[i][j]=a[i+nsize*3][j];}
Future f1; Future f2; Future f3; Future f4;
f1=es.submit(new Callable<Object>(){
@Override
public Object call() throws Exception
{mult1(a1);
return true; }});
f2=es.submit(new Callable<Object>(){
@Override
public Object call() throws Exception
{mult2(a2);
return true; }});
f3=es.submit(new Callable<Object>(){
@Override
public Object call() throws Exception
{mult3(a3);
return true; }});
f4=es.submit(new Callable<Object>(){
@Override
public Object call() throws Exception
{mult4(a4);
return true; }});
try {
f1.get();f2.get();f3.get();f4.get();
Toast.makeText(getBaseContext(),
 "Getting results",Toast.LENGTH_SHORT).show();
} catch (InterruptedException e) {
e.printStackTrace();}}
```

Fig. 4 Code snippet illustrating matrix slicing, task submission and use of ExecutorService.

Figure (5) below shows runtime comparison between sequential and parallel.

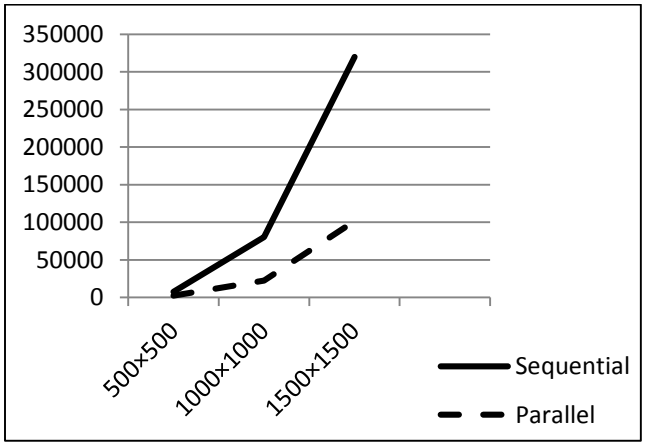

Fig 5. Runtime comparison between sequential and parallel.

Figure (6) below shows expected performance comparison.

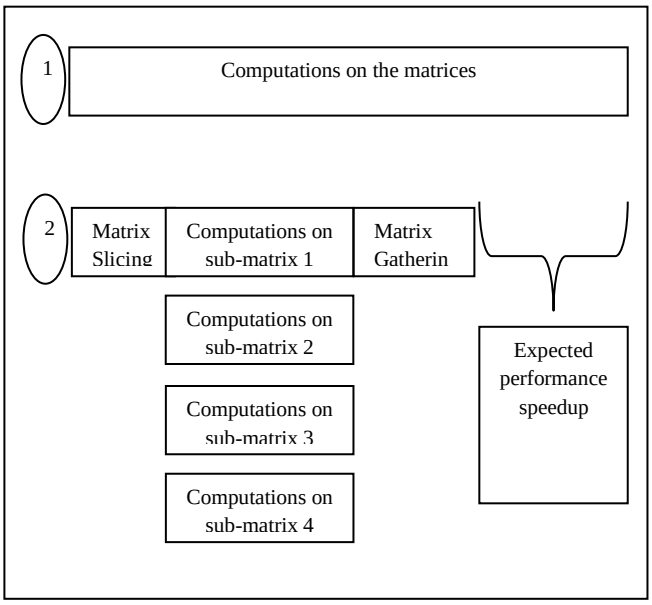

Fig. 6 Expected performance comparison : (1) Matrix multiplication without using the proposed system vs. (2) Matrix multiplication using the proposed system.

From the time results in (Table (1)), we can see that the system achieved a great speedup as shown in (Table (2)) below:

Table 2**:** Speedup results

Matrix size	Speed up
500×500	3.01
1000×1000	3.55
1500×1500	3.15

Figure (7) below shows speedup results.

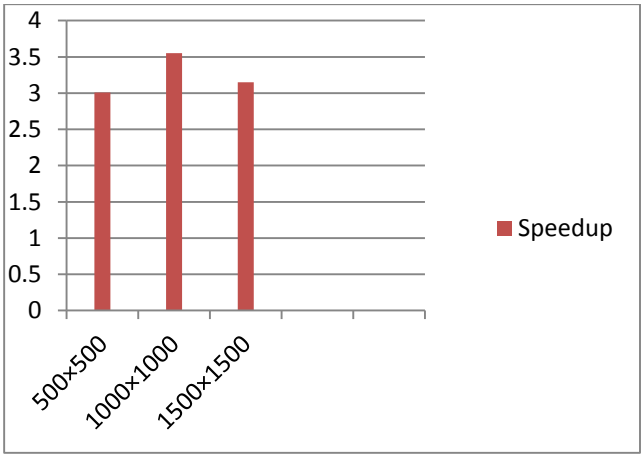

Fig 7. Speedup results

12. Conclusions

In this paper, we have demonstrated how to achieve speedup in a big matrices multiplication system written entirely in JAVA by using the proposed parallel programming approach, based on the idea of task and data parallelism. Running on a quad-core Samsung Galaxy SIII. The proposed system shows significant reduction in the overall processing time and great speedup.

13. References

[1] Panya Chanawangsa, Chang Wen Chen, "A New Smartphone Lane Detection System: Realizing TruePotential of Multi-core Mobile Devices", MoVid'12, 2012, pp.19-24.
[2] Laurence T. Yang, Daniel C. Doolan, "Mobile Parallel computing", Proceedings of The Fifth International Symposium on Parallel and Distributed Computing, IEEE International,2006.
[3] Gene H. Golub, Charles F. Van Loan, Matrix Computations, The johns Hopkins University Press,2013.
[4] Tsogkas Panagiotis, "Evaluating Skandium's Divide-and-Conquer Skeleton", Master Thesis, School of Information, University of Edinburgh, 2010.
[5] Catanzaro, B., et al., "Ubiquitous Parallel Computing form Berkeley, Illinois, and Stanford", IEEE Computer Society, 2010, pp. 41-55.
[6] Herve Guihot, Pro Android Apps Performance Optimization, Apress, 2012.
[7] Donis Marshall, Parallel Programming with Microsoft Visual Studio, Microsoft Corporation by: O'Reilly Media, 2011.
[8] Samsung I9305 Galaxy S III Full Specifications, http://www.gsmarena.com/samsung_i9305_galaxy_s_iii-5001.php
[9] M. Sasikumar, Dinesh Shikhare, P. Ravi Prakash, Introduction To Parallel Processing, Prentice-Hall of India Private Limited, 2006.
[10] Daniel C Doolan, Sabin Tabirca, Laurence T Yang," MMPI a Message Passing Interface for the Mobile Environment", Proceedings of MoMM2008, Linz, Austria, 2008.

Dhuha Albazaz is the head of Computer Sciences Department, College of Computers and Mathematics, University of Mosul. She received her PhD degree in computer sciences in 2004 in the speciality of computer architecture and operating system. She supervised many Master degree students in operating system, computer architecture, dataflow machines, mobile computing, real time, and distributed databases. She supervised three PhD students in FPGA field, distributed real time systems, and Linux clustering. She also leads and teaches modules at both BSc, MSc, and PhD levels in computer science. Also, she teaches many subjects for PhD and master students.

Mohammed M. Al-Hafidh is a master student in Computer Sciences Department, College of Computers and Mathematics, University of Mosul. He interest with networks, Databases, and operating system subjects