Honeybee: A Programming Framework for Mobile Crowd Computing

Niroshinie Fernando, Seng W. Loke, and Wenny Rahayu

Department of Computer Science and Computer Engineering, La Trobe University, Australia
tnfernando@students.latrobe.edu.au, {s.loke,w.rahayu}@latrobe.edu.au

Abstract. Although smartphones are increasingly becoming more and more powerful, enabling pervasiveness is severely hindered by the resource limitations of mobile devices. The combination of social interactions and mobile devices in the form of 'crowd computing' has the potential to surpass these limitations. In this paper, we introduce Honeybee; a crowd computing framework for mobile devices. Honeybee enables mobile devices to share work, utilize local resources and human collaboration in the mobile context. It employs 'work stealing' to effectively load balance tasks across nodes that are a priori unknown. We describe the design of Honeybee, and report initial experimental data from applications implemented using Honeybee.

Keywords: mobile crowd computing, mobile cloud computing, remote execution, offloading, crowd sourcing.

1 Introduction

Collaboration among mobile devices paves the way for greater computing opportunities in two ways; Firstly, it solves the inherent resource limitations of mobile devices [19]. Secondly, a mobile device is usually accompanied by a human user, who can use his/her 'human expertise' to crowd source problems that need a human element [12]. Increasing usage and capabilities of smartphones, combined with the potential of crowd computing [17] can provide a collaborative opportunistic resource pool. This paper aims to provide Honeybee, a programming framework that facilitates the development of mobile crowd computing applications exploiting such resources. We define 'mobile crowd computing' as a local 'mobile resource cloud' comprising a collection of mobile devices, and their users.

We build on previous work where we first investigated static job farming among a heterogeneous cluster of mobile devices in [8], and a more load balanced approach in [9] using 'work stealing' [5]. To our knowledge, no other work has used work stealing in the mobile computing domain, although it has been employed for job scheduling with load balancing in distributed environments such as Cilk ([4], [14]), and Parallel XML processing [15].

There exists a number of proposed systems on mobile clouds [10] and crowd computing [11,21,18]. The work on mobile clouds mainly focus on offloading

© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2013

machine computation to or with the support of a remote server, while crowd computing systems focus on either collecting data or coordinating human intelligence tasks, also using a remote server. To our knowledge, no single system supports both kinds of 'work sharing' mentioned above, i.e, machine computation tasks, and human computation tasks, using local neighbourhood resources.

Typical grid/distributed computing solutions are not applicable in mobile environments due to the following characteristics of mobile resources: less processing power, finite energy, high volatility of the resource pool resulting in inconsistent node availability, unknown devices a priori calling for opportunistic behaviour, and heterogeneity. Therefore, mobile crowd computing requires a dynamic load balancing method that is decentralized, proactive and self-adaptive instead of conventional master-slave static work farming.

The main contributions of this paper are, incorporating work stealing on a mobile resource pool to achieve load balancing without a prior knowledge of participating nodes, an API to support job sharing and crowd-sourcing among mobile devices, and evaluation of Honeybee using three different applications.

Outline. Key related work are discussed in §2, and the concepts and design principles of Honeybee in §3. An overview of implementation details are given in §4, and experiments conducted on three different applications are described in §5, with conclusions and future work outlined in §6.

2 Related Work

Opportunistic computing on mobile devices has been explored recently in a number of contexts. Job sharing via cyber foraging [20] in particular plays a main role. At the other end of the spectrum, Crowd-sourcing [12] utilizes the collective power of human expertise to solve problems. In Honeybee, we focus on a programming framework that provides an API to enable applications that cater to both ends in a mobile context.

In a large number of cases concerning mobile task offloading, a central server is essential to either co-ordinate the tasks among the mobile devices ([16]), or to offload the processing on to [6]. However, our focus is on local and decentralized job sharing owing to the issues in connecting to a remote server such as latency, bandwidth issues [20], network unavailability, battery drain when connecting via 3G, and data costs.

Crowd Computing is introduced in [17] which shows the potential of using mobile devices in a social context to perform large scaled distributed computations. They use message forwarding in opportunistic networks as basis and use a static task farming approach. We, however, show that work stealing can give better results compared to a static farming approach. In CrowdSearch, [21], image search on mobile devices is performed with human validation via the Amazon mechanical turk (AMT)[1]. CrowdSearch requires a backend server as well, since the processing is done on both local and remote resources. [11] is a query processing system that uses human expertise to answer queries that database systems and search engines find difficult. In Medusa [18], a crowd sensing framework for collecting sensor data from mobile phones, users are able to specify high level abstractions for sensing tasks, using AMT. Our work is different from these in terms of using only local mobile resources opportunistically. Furthermore, our focus is on a framework that can be used to implement a variety of tasks, not limited to query processing, sensing, or human validation. However, previous work showed us that user participation is at a considerable level, and 'micro tasking' is viable. WiFi or 3G has been the popular choice among many of these work, except in cases such as the MMPI framework [7], which is a mobile version of the standard MPI over Bluetooth, and in [3] where swarm intelligence techniques has been adopted for message propagation. We have only used Bluetooth in this implementation due to its widespread availability and low energy consumption. However, other protocols shall be implemented in future work.

3 Honeybee: Concept and Design

The objective of the Honeybee framework is twofold:

1. In case of human aided computation such as qualitative classification tasks, to enable collaboration of multiple human users using mobile devices. For example, conducting surveys, asking for opinions, image identification and comparison, audio transcribing, collecting data etc.
2. In case of machine computation, improve the efficiency of the execution by either giving a speedup gain, and/or conserving resources (eg: battery). For example, image processing, natural language processing, e-learning, and multimedia search.

Honeybee's work stealing algorithm for mobile devices was described in detail in our previous work [9], which will be briefly explained here using an example scenario that combines human and machine intelligence.

3.1 How Honeybee Works: Lost Child Scenario

Consider a carnival setting attended by hundreds of people. Among these attendees, a toddler goes missing and the security officials are notified. Charged with quickly locating the missing child among the throng of visitors, the authorities decide to use crowd computing. They broadcast a request, notifying the attendees of the situation, and send out a photograph of the child through opportunistic forwarding. Here, the initial 'task' is the request containing details about the child, his photograph, and whom to contact if seen. Instead of selecting a few worker nodes, the crowd computing system specifies that this is a message needing to be propagated to all encountering devices. People who receive the message go through their photo gallery and check if a child can be spotted in their photographs taken on the carnival. However, it is common for many of people to take many photographs at such an event. To go through all of them requires time and patience. Let us say John is such an attendee. He has taken over hundred photographs in the carnival and he chooses instead to first filter out photographs containing faces. However, facial detection is an expensive task. Therefore, he employs Honeybee to share the face detection work among other people, possibly employing an app similar to our prototype in Section 5.1. Here, John's mobile device is the 'delegator' and Honeybee will first identify available 'worker nodes' in the vicinity and then proceed to distribute the 'job pool' containing the images to be processed among them. Since there is no way to determine device capabilities a priori, Honeybee will initially distribute the job pool equally. As time progresses, the nodes who complete running the face detection on their share of jobs will attempt to 'steal' jobs from other nodes, ensuring that 'faster' nodes will receive more jobs, contributing to a higher speedup. John's device, which is the delegator, continues to do a part of the work as well, while listening for incoming 'result transmissions'. Once all the jobs results are collected, the delegator sends out a termination signal that notifies the workers that task is done. Once the face detection is done, John checks the filtered photographs manually to check for an appearance of the child.

There are several issues that arise considering such a scenario as above: What kind of tasks can be 'shared' using a mobile resource pool? Where would the job pool be stored? How does the delegator manage executing, stealing and distributing jobs? How does device mobility affect program execution? Why not distribute jobs one by that would ensure load balancing? What will motivates users to participate in such a scheme? These shall be addressed in the following section.

3.2 Design Considerations

We extensively focus on the concept of *busyness*, to keep participating devices busy as much as possible and minimize idling. Honeybee is catered for tasks that can be broken down into several independent jobs that can be executed in parallel, so that the complete task $J = \sum_{i=1}^{n} j_i$ where there are a total of n jobs. The delegator selects at random, a pool of f workers such that $f \leq n$. The sub task distribution is twofold: initial jobs and stolen jobs. Initial tasks are distributed by the delegator at time t_0 such that each participating device receives n/f jobs. The delegator also starts executing its share of jobs at time t_0, in parallel to the job distribution. The workers read the job parameters, and start the execution as

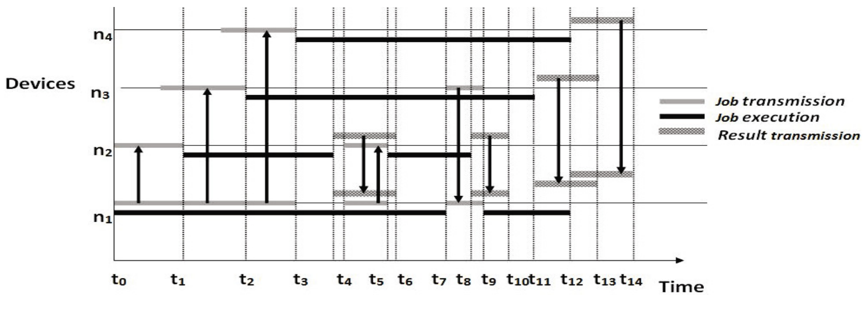


Fig. 1. Four nodes working in collaboration using work stealing

soon as they receive them. So, worker n_2 shall start execution at time t_1, n_3 at t_2, and so on. Depending on device capabilities, resources, start time of computation, and job size/complexity, some nodes are likely to finish their jobs before others. If this occurs, the finished nodes shall transmit their finished result back to the delegator, and attempt to 'steal' some jobs from another node.

This scenario is illustrated in Figure 1 where four nodes are collaborating together. In this example, the four nodes are of different capabilities. That is, while n_0 does 1 job in time t, n_1 might do 3 jobs. As can be seen in the figure, the delegator n_1 distributes the jobs in parallel with its own share of job execution, until time t_3, when it finishes transmitting. From thereon, n_1 carries out its computations until time t_7. However, before it reaches t_7, a couple of communication incidents have occurred: at time t_4, node n_2 finishes its jobs, and starts transmitting the results back to n_1. This transmission occurs until t_6. Meanwhile, node n_2 also steals some jobs from n_1, since it has finished its queue. This 'stealing' process occurs until t_5, where n_1 is seen transmitting some of its jobs to n_2. After finishing its jobs at t_7, n_1 also steals some jobs from the job queue of n_3. This cycle of job execution, result transmission and stealing continues until all jobs are finished, and collected, which occurs at time t_{14}. Note that the nodes are kept busy throughout.

Job Expiry. In a practical scenario, it is unrealistic to assume all nodes will be static, or at least be relatively static during the execution. What happens if a worker node moves out of proximity before it can transfer the results? To overcome this problem, a 'deadline' needs to be set by the delegator n_1. So if the delegator does not receive any result from a particular worker within this time, it will add those jobs back to its queue and either proceed to execute them itself, or have them 'stolen' eventually to be done by another worker.

Device Mobility. Three constraints must be taken into consideration; (a)the job distribution time must not exceed the time the devices are in contact with the delegator, (b)to enable work stealing, at least several devices must be in contact with each other for the duration of execution, and (c)worker devices must transmit the result/s before an 'expiration time'. It should be noted that for different classes of applications, different settings apply. For example, if the objective of using Honeybee is to gain a speedup, performance time is of great importance. In that case, a low entropy setting where a group of people are stationary relative to each other is most suitable. Some examples for this kind of topology are, a group of passengers in a train, a hiking group, and a group of people at a restaurant. A certain percentage of devices being out of range at times is acceptable however, and these can be handled via fault tolerance methods. If the objective is to conserve resources, or data collection, where the expiration time is greater, higher entropy topologies can be considered, and in some cases are even more suitable. In this setting, a larger number of nodes will be available, and will be moving around, within a specific area. Examples are, a shopping mall, a sporting event, and an airport. Worker nodes can pass their results either through direct encounters, or opportunistic forwarding.

Bundle Size. Bundle size refers to the number of jobs assigned to a participating device at a given time. Rather than maintaining a central job queue at the delegator, we have designed Honeybee to distribute the jobs among workers such that bundle size is ≥ 1. In most cases, bundle size is n/f. Unlike in a typical distributed system setting, a resource cloud of mobile devices need to be vary of limited battery, mobility and communication. If bundle size is 1, load balancing would be automatic, but worker devices would have to continuously poll the delegator for new jobs, and the delegator would have to maintain many connections simultaneously, causing heavy communication costs. Furthermore, devices are liable to move away from the delegator during the course of execution. By setting a bundle size that is ≥ 1, we limit the probability of workers starving of jobs if they are not in proximity to the delegator.

User Participation. As with all crowd sourcing apps, the success of Honeybee depends on device participation, and participation depends on the incentives. Experiences with other micro task frameworks such as AMT have given positive indications on monetary incentives, and Wikipedia is a good example of human collaboration for non-financial gain. Incentives could also be in the form of social contract such as in a group of friends, or common goals such as in [13]. Since users may hesitate to form connections with arbitrary devices due to privacy and security concerns, a crowd computing framework must ensure secure communications, possibly by granting anonymity as suggested in [18]. Although not yet implemented in Honeybee, this is an essential part of our future work.

3.3 Upper Bound for Speedup

We now give a theoretical upper bound for speedup using Honeybee, versus monolithic execution. It is useful to formulate an upper bound to evaluate and understand the best possible result for our practical implementation. We define a speedup as the time taken to complete a task using Honeybee divided by the time taken to complete the task on the delegating device alone. We make the following assumptions for the upper bound: a)the complete task is composed of l equal jobs, and there are a total of f devices in the opportunistic network, b)communication costs are not considered, and c)there are no restrictions on number of connections per delegating device.

The delegating device shall be denoted as n_1, and others as n_i. The time to complete m jobs (where $m < l$) on device n_i is given as t_i. Therefore, the time to complete l jobs on n_1, and therefore the time for monolithic execution, is given as $\frac{t_1}{m}l$. Let us say there exists a non negative constant k_i for each n_i device such that $\frac{t_i}{t_1} = k_i$. Thus, total number of jobs done in t_1 time in the network is equal to $m[1 + \frac{1}{k_2} + \frac{1}{k_3} + ...\frac{1}{k_i}... + +\frac{1}{k_f}]$. Therefore, the Speedup $S = \frac{t_1 l}{m} \times \frac{m[1+\frac{1}{k_2}+\frac{1}{k_3}+...\frac{1}{k_i}...++\frac{1}{k_f}]}{t_1 l}$, and so, $S = 1 + \frac{1}{k_2} + \frac{1}{k_3} + ... + \frac{1}{k_f}$. Due to non-negligible communication in actual scenarios, the actual speedup would be less than S.

4 System Implementation

The Honeybee framework is implemented on Android, using Bluetooth as the connection protocol. The framework contains interfaces and methods for developing mobile crowd applications. We have implemented three applications using Honeybee, which will be explained in the Applications section. Figure 2 shows the main components of the system from the delegator's perspective.

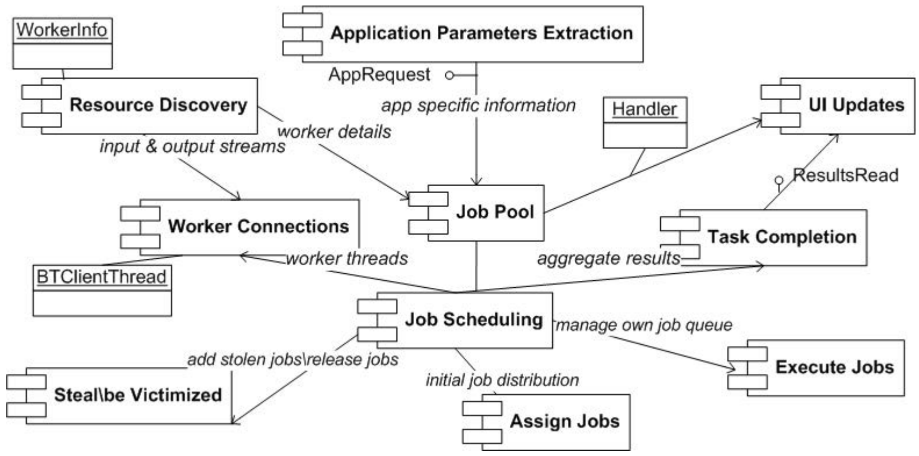
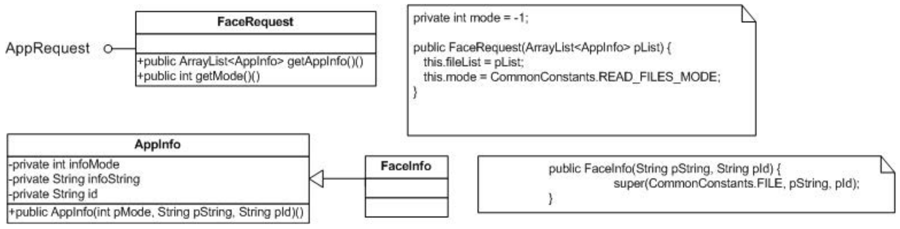


Fig. 2. Main components of Honeybee

Application Parameters. As the starting point of execution, the application passes app specific task parameters to the framework using interface *AppRequest*. In Figure 3, we show a snippet from the Face detection app (5.1), where the complete task is stored as a *FaceRequest* object, that has a list of subtasks (*FaceInfo* objects). When Honeybee processes a FaceRequest, it knows the job parameters are multiple files (from mode in FaceRequest), and each file is represented as a FaceInfo object.


AppRequest
FaceRequest
+public ArrayList<AppInfo> getAppInfo()()
+public int getMode()()
AppInfo
-private int infoMode
-private String infoString
-private String id
+public AppInfo(int pMode, String pString, String pId)()
FaceInfo

```
private int mode = -1;

public FaceRequest(ArrayList<AppInfo> pList) {
  this.fileList = pList;
  this.mode = CommonConstants.READ_FILES_MODE;
}
```

```
public FaceInfo(String pString, String pId) {
        super(CommonConstants.FILE, pString, pId);
}
```

Fig. 3. Abstraction of jobs in the API

Resource Discovery and Worker Connections. Connecting to workers is achieved by calling the *ResourceDiscoveryActivity* class, which looks for available devices, connects, and creates threads for handling each connection. Each successful connection is registered with the system as a *WorkerInfo* instance.

Job Pool. The job pool is initiated by calling the *initJobPool(AppRequest pReq)* method of factory class *JobPool*, using the information passed in AppRequest. These jobs are then assigned to workers in individual threads. The 'mode' specified in AppRequest is needed when the delegator transmits the jobs to workers. The transmitting thread first writes the 'mode' of the parameters (eg: string, file, integer, etc), and then proceeds to transmit the actual parameters themselves. Constants defined in class *CommonConstants* are used to specify the modes.

UI Updates. UI Updates to the application are handled through a *Handler* object that notifies the application classes whenever a change occurs to the device job list (i.e. completed/stolen/added), and the callback interface *ResultsRead*, which receives notification when all the results are collected. It is up to the application to provide the implementation of the processing of jobs and/or results. For example, in the Face detection app, Honeybee notifies the application activity whenever a new job is received, and this triggers the *doWork()* method containing the program specific logic.

Job Scheduling. Job scheduling is closely associated with the worker thread pool that was created in the resource discovery phase. Firstly, the job scheduler must start on executing own share of jobs, while assigning each connected worker their initial job list. As explained in Figure 1, stealing, or victim threads may also be created and run. It should be noted that all the aforementioned threads need to be carefully synchronized since (a)they all need access to the job pool, and (b)same bluetooth connection is used for all communications between a worker-delegator pair. An example of such a communication conflict is when a worker device starts transmitting results, and steal from the delegator simultaneously.

5 Experimental Evaluation

Our testbed contains a total of five Android devices of varying capabilities, including Nexus S[1], Ideos[2], and Galaxy SII[3]. We have implemented the following three applications using Honeybee framework to evaluate its performance and feasibility:(a)Distributed face detection, (b)Distributed mandelbrot set generation, and (c)Collaborative photography. We have selected these applications for their different job characteristics, that are listed in Table 1.

[1] `http://www.google.com/nexus/#/galaxy/specs`

[2] `http://www.huaweidevice.com/worldwide/productFeatures.do?pinfoId=2831&directoryId=6001&treeId=3745&tab=0`

[3] `http://www.samsung.com/global/microsite/galaxys2/html/specification.html`

Table 1. Job characteristics of applications used for evaluation

Application	Job type	Data size of I/O
Face detection	Machine centric: CPU and memory intensive	Big inputs/small outputs
Mandelbrot set generation	Machine centric: CPU intensive	Small inputs/big outputs
Collaborative photography	Human centric processing	Small inputs/big outputs

5.1 Distributed Face Detection

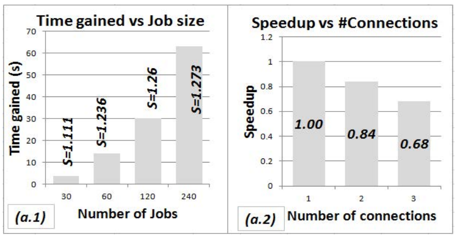
In this application, we run Android's native face detection on a collection of photographs. Face detection is heavy in terms of CPU cycles, and memory. The main objective of using Honeybee to distribute the face detection computations is to increase the performance in terms of speedup. Furthermore, because of its heavy memory allocation requirements, running face detection on a collection of images is difficult, and as we found, causes OutOfMemoryExceptions if we executed on low powered devices such as Ideos. But via Honeybee, such an Ideos device can achieve such resource intensive computations by offloading jobs to other more powerful devices.

Here, the job pool contains thirty images that are stored on the delegator. On a Nexus S, to run face detection takes around 74 seconds. We mainly focused on two sets of experiments; share with similarly capable devices, and share with more capable devices. In the first category, we used two other Nexus S devices and a Galaxy S, since they can be considered equals in terms of CPU and memory capabilities. In the second category, we used an Ideos as the delegating device, to offload work with the aforementioned more powerful devices.

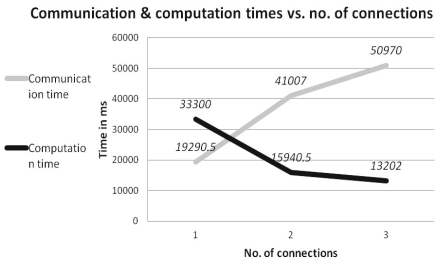
Evaluation Objectives:

- Examine how the performance varies with job size.
- Examine the performance results for jobs with parameters that are large in terms of inputs. The outputs of this application is very small in terms of size. We have already discussed findings of an application (Mandelbrot) with opposite parameter characteristics (small inputs, large outputs) in our earlier work [9].Compared to Mandelbrot generation, Face detection's input parameters are of considerable size (at least $\geq$ 8MB).
- Implement an application with different job and steal parameters than for our previous work in [9]. We have extended Honeybee to implement applications with different types of jobs and job parameters.

Results and Discussion. The performance results are summarized in Figure 4 (a). In Figure 4 (a.1), where we show the 'time gain' versus the job size, using two devices, it is evident that the performance increases with job size. For example, for 30 images, the distributed implementation is only 4 seconds faster, but for 240 images, the shared version finished 63 seconds earlier. When comparing the



(a) Performance results for Face detection



(b) Time breakdown for Face detection

Fig. 4. Results of Face detection app

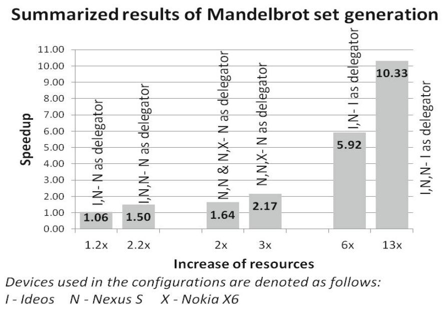
theoretical upper bound for this case (which is 2 as explained in Section 3.3), and the actual best result of 1.3 at 240 images, the effects of communication cost (of Bluetooth in particular) is evident. Figure (a.2) shows the variation of speedup results for a fixed job size (30 images) as number of connections are increased. In theory, more devices in the resource pool should have yielded better speedups. However the addition of devices seems to have degraded the performance.

Let us now examine the time breakdown that pertains to these results as shown in 4 (b). It is evident that the data rate drops considerably with the addition of new connections. The brunt of this communication time is spent on distributing the job parameters, i.e the data, to the worker devices. Although in all three cases, the amount of data transferred remains the same, managing additional connections has slowed down the delegator's throughput. Although the delegator's computation time does drop with each addition, this does not improve the overall performance. This is due to the fact that although the delegator steals jobs from workers, it still has to wait a long period of time until the jobs are distributed. This suggests using faster inter-device networking (e.g. WiFi-Direct which promises speeds up to 250 Mbps and longer range [2]) in our future work.

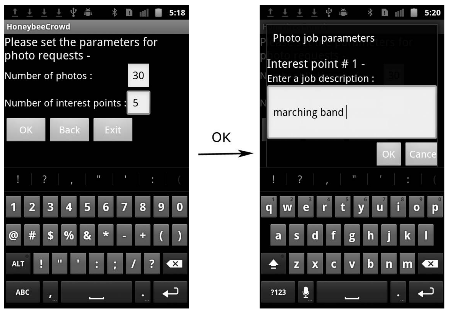
As another solution, dynamic initial job assigning can be explored. Instead of assigning equal jobs to all devices, the delegator is allowed to keep consuming jobs from the head of job queue. Meanwhile, jobs are transferred from the tail of the queue to workers, ensuring that the delegator's computation thread will not starve/wait till distribution is complete. In a sense, this is incorporating work stealing to the job consuming threads, since the delegator's worker thread and communication threads are consuming jobs from the same queue.

5.2 Distributed Mandelbrot Set Generation and Collaborative Photography

We have discussed the results of Mandelbrot set generation over a heterogeneous set of devices including Android and Nokia smartphones, and collaborative photography using crowd sourcing in our previous work [9]. The results of Mandelbrot experiments are summarized in Figure 5(a). By benchmarking each device in the Mandelbrot algorithm, we were able to determine that Nexus S and Nokia X6 performed the same, while both were around 6 times faster than Ideos. We have depicted this in the graph in the horizontal axis as resources in the distributed version relative to the monolith version. For example, when Nexus S shares work with an Ideos, new resources are $(1 + 0.2)x$ compared to initial $1x$ amount of resources (thus a negligible increase). We also experimented with adding a PC to the resource pool, and were able to gain speedups upto 23. These results show that even with communication overheads, Honeybee is always able to give a speedup.



(a) Speedups for Mandelbrot set generation


HoneybeeCrowd

Please set the parameters for photo requests -

Number of photos : 30

Number of interest points : 5

Photo job parameters

Interest point # 1 -

Enter a job description : marching band

(b) Screenshots from Collaborative photography

Fig. 5. Results of Mandelbrot set generation and Collaborative photography

Figure 5(b) illustrates main screens from the Collaborative photography app, which illustrates using work stealing concept in human centric computation. In this application, delegator specifies 'photo jobs' describing the requirements for photographs via 'interest points'. The faster photographers (talented/better cameras/good vantage points etc) will be able to steal additional jobs from slower workers, thus achieving load balancing.

6 Conclusions and Future Work

We present three main conclusions formed by our experiments: Firstly, results from initial prototype apps, implemented on Honeybee is evidence that a generalized framework for work/resource/expertise sharing on mobile crowd computing is viable. We have achieved this through abstracting jobs, and enabling parameterization for different types of jobs mentioned in Table 1. Secondly, for all three applications, load balancing has been achieved with work stealing, limiting device idle time. Thirdly, adding computational resources can prove to be ineffective in cases of large communication overheads. Therefore a small group of powerful devices achieves a better performance speedup than a large group of relatively weaker devices.

As future work incorporating stealing in initial job assignment should be explored to minimize the negative impacts of communication overheads. Handling incentives (social/monetary/reciprocal) and providing a secure platform is also essential for user participation. Although we have only used Bluetooth in this initial implementation, we hope to enable communications in WiFi direct as well. Furthermore, a degree of job redundancy needs to be supported to ensure robustness.

References

1. Amazon mechanical turk, `https://www.mturk.com/`
2. Wi-fi direct, `http://www.wi-fi.org/discover-and-learn/wi-fi-direct`
3. Afridi, A.H.: Mobile social computing: Swarm intelligence based collaboration. Lecture Notes in Engineering and Computer Science, vol. 2198 (2012)
4. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. SIGPLAN Not. 30, 207–216 (1995)
5. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 46(5), 720–748 (1999)
6. Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: elastic execution between mobile device and cloud. In: Proceedings of the Sixth Conference on Computer Systems, EuroSys 2011, pp. 301–314. ACM, New York (2011)
7. Doolan, D.C., Tabirca, S., Yang, L.T.: Mmpi a message passing interface for the mobile environment. In: Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia, MoMM 2008, pp. 317–321. ACM, New York (2008)
8. Fernando, N., Loke, S.W., Rahayu, W.: Dynamic mobile cloud computing: Ad hoc and opportunistic job sharing. In: 2011 Fourth IEEE International Conference on Utility and Cloud Computing (UCC), pp. 281–286 (December 2011)
9. Fernando, N., Loke, S.W., Rahayu, W.: Mobile crowd computing with work stealing. In: Proceedings of the 15th International Workshop on Mobile Cloud Computing Technologies and Applications (in NBiS) (September 2012)
10. Fernando, N., Loke, S.W., Rahayu, W.: Mobile cloud computing: A survey. Future Generation Computer Systems 29(1), 84–106 (2013)
11. Franklin, M.J., Kossmann, D., Kraska, T., Ramesh, S., Xin, R.: Crowddb: answering queries with crowdsourcing. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, SIGMOD 2011, pp. 61–72. ACM, New York (2011)
12. Howe, J.: The rise of crowdsourcing (2006), `http://www.wired.com/wired/archive/14.06/crowds.html`
13. Huerta-Canepa, G., Lee, D.: A virtual cloud computing provider for mobile devices. In: Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond, MCS 2010, pp. 6:1–6:5. ACM, New York (2010)
14. Jovanovic, N., Bender, M.A.: Task scheduling in distributed systems by work stealing and mugging - a simulation study. In: Proceedings of the 24th International Conference on Information Technology Interfaces, ITI 2002, vol. 1, pp. 259–264 (2002)
15. Lu, W., Gannon, D.: Parallel xml processing by work stealing. In: Proceedings of the 2007 Workshop on Service-Oriented Computing Performance: Aspects, Issues, and Approaches, SOCP 2007, pp. 31–38. ACM, New York (2007)

16. Marinelli, E.E.: Hyrax: Cloud Computing on Mobile Devices using MapReduce. Carnegie Mellon University, Masters thesis (2009)
17. Murray, D.G., Yoneki, E., Crowcroft, J., Hand, S.: The case for crowd computing. In: Proceedings of the Second ACM SIGCOMM Workshop on Networking, Systems, and Applications on Mobile Handhelds, MobiHeld 2010, pp. 39–44. ACM, New York (2010)
18. Ra, M.-R., Liu, B., Porta, T.F.L., Govindan, R.: Medusa: a programming framework for crowd-sensing applications. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, MobiSys 2012, pp. 337–350. ACM, New York (2012)
19. Satyanarayanan, M.: Fundamental challenges in mobile computing. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing PODC 1996, pp. 1–7. ACM, New York (1996)
20. Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N.: The case for vm-based cloudlets in mobile computing. IEEE Pervasive Computing 8(4), 14–23 (2009)
21. Yan, T., Kumar, V., Ganesan, D.: CrowdSearch: exploiting crowds for accurate real-time image search on mobile phones. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, MobiSys 2010, pp. 77–90. ACM, New York (2010)