The Case for Crowd Computing

Derek G. Murray, Eiko Yoneki, Jon Crowcroft, and Steven Hand
University of Cambridge Computer Laboratory
Cambridge, United Kingdom
{Firstname.Lastname}@cl.cam.ac.uk

ABSTRACT

We introduce and motivate *crowd computing*, which combines mobile devices and social interactions to achieve large-scale distributed computation. An opportunistic network of mobile devices offers substantial aggregate bandwidth and processing power. In this paper, we analyse encounter traces to place an upper bound on the amount of computation that is possible in such networks. We also investigate a practical task-farming algorithm that approaches this upper bound, and show that exploiting social structure can dramatically increase its performance.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures—*mobile processors*

General Terms

Algorithms, Measurement

Keywords

Distributed computation, human mobility, task farming

1. INTRODUCTION

Today's smartphone is a powerful computer. It is equipped with a range of sensors, a gigahertz-range CPU and high-bandwidth wireless networking capabilities [5]. Inspired by the increasing prevalence of smartphones, and research into opportunistic networking [4, 19], we have evaluated the potential of using these devices to carry out large-scale distributed computations. In this paper, we introduce *crowd computing*, in which opportunistic networks can be used to spread computation and collect results.

A crowd computation spreads opportunistically through a network, using ad-hoc wireless connections that form as devices come into proximity. The devices can exchange input data and intermediate results. In parallel work, we are developing programming languages that enable developers to implement a crowd computation [17, 25]; this paper focuses on the aggregate *utility* of such a computation, in terms of how much work each device can carry out.

MobiHeld 2010, August 30, 2010, New Delhi, India.

Why is crowd computing attractive? Previous work has shown that people will voluntarily contribute their desktop computer resources for running scientific workloads [6]. We could imagine a similar application for mobile devices that provides free content or functionality in exchange for volunteered cycles. Furthermore, unusual devices such as graphics cards [20] and games consoles [3] have been used to perform high-throughput computing. A modern smartphone has several special-purpose cores (such as DSPs and A/V codecs) [5], which could similarly be applied to large-scale problems. Moreover, opportunistic networks in which the nodes are mobile offer potentially huge bandwidth [10], turning a collection of smartphones into a mobile supercomputer.

Alternatively, we can use crowd computing as a means of distributing *human interaction tasks* to mobile devices. For example, Amazon Mechanical Turk has created a marketplace for carrying out work that is difficult for computers to process, but relatively simple for humans [1]. For example, many qualitative classification tasks are much easier for humans than computers, such as "What is the best Sushi restaurant in San Francisco?" By combining this model with crowd computing, it would be possible to exploit geographic locality in the respondents.

We begin by seeking an upper bound for the computational capacity of an opportunistic network (Section 2). We contrive an idealised distributed computation that can spread epidemically with negligible data exchange, and simulate its execution on a variety of human encounter traces. By positing that each person in the trace possesses a smartphone, we can measure the total work done by simulation.

Of course, few realistic computations fit our ideal model. We therefore consider the common *task farming* approach, and evaluate its performance on the same traces (Section 3). We find that, on average, it achieves 40% of the performance of our ideal computation. Switching to a concrete model introduces more variables, so we consider the effect of master choice, task size and node capacity on the overall utility of the system.

We build on previous work that has shown how social network analysis can greatly improve the efficiency of message forwarding in opportunistic networks [7, 11]. We investigate how a similar technique can be used to improve the performance of task farming (Section 4). In particular, we observe that dividing an opportunistic network into communities, and running a separate task farm within each community, improves the throughput of task farming by an average of 50%.

In this paper, we aim to show that an opportunistic network of mobile devices is an interesting platform for distributed computation. Our results demonstrate that such networks can provide a high degree of parallelism. We are currently developing the first crowd computing applications that exploit this approach.

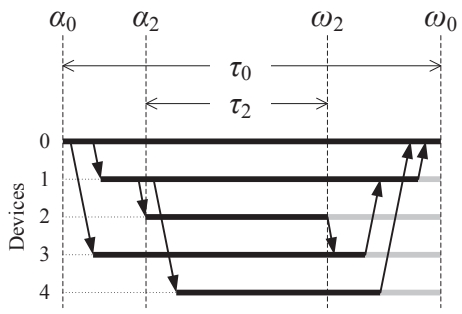

Figure 1: The progress of a computation in a network of five nodes. Each arrow corresponds to an encounter between nodes. Bold black lines correspond to useful computation, and bold gray lines correspond to wasted computation.

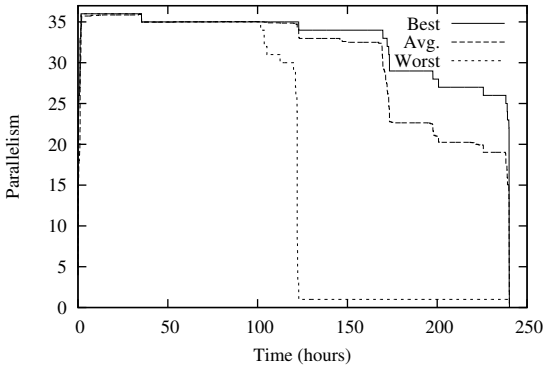

Figure 2: Achievable parallelism for an ideal distributed computation running on the Cambridge trace.

Algorithm 1 Algorithm for computing α_i

```
A ← {0}
α_0 ← start time
for (i, j, t) ∈ trace do
  if i ∈ A ∧ j ∉ A then
    α_j ← t, A ← A ∪ {j}
  else if j ∈ A ∧ i ∉ A then
    α_i ← t, A ← A ∪ {i}
  end if
end for
```

2. BEST CASE SCENARIO

We first determine an upper bound for the amount of computation achievable in an opportunistic network. The goal of a crowd computation is to have long periods of *useful parallelism*. This means that a device must not only receive a message that causes it to join the computation, but it must send a message containing its result that eventually reaches the initiator (Figure 1). In this section, we first define our model of an ideal distributed computation (§2.1), and then evaluate it on real-world encounter traces (§2.2).

2.1 Definitions

We assume a set of n identical mobile devices that participate in the computation. Device zero is the *initiator*, and starts computing (becomes active) at time α_0.

For optimal delivery, coordination and result messages spread by flooding. All devices listen for radio transmissions at all times; all active devices continually broadcast a probe message to discover nearby devices[1].

When an active device meets another device, they exchange a sequence of messages. First, the active device sends a message describing the computation. On receiving this message, an inactive device becomes active: thus the computation floods throughout the network. Each active device stores a partial result that includes the result of its computation, and any partial results received from other devices. When two devices meet, they exchange their current partial result: this ensures that the results also flood through the network, which maximises the probability that they reach the initiator.

[1]In practice, power considerations and wireless MAC protocols will limit the ability of devices to broadcast continually.

Finally, at time ω_0, the initiator ends the computation[2], and computes the final result from the messages that it has received.

Now consider device i. It starts computing at α_i. At ω_i, it sends the last message that (in one or more hops) reaches the initiator before ω_0. An encounter is a tuple (i, j, t), indicating that nodes i and j meet at time t. Given a chronologically-ordered sequence of encounters, we compute α_i using Algorithm 1. We compute ω_i by reversing the order of the encounter trace, and rerunning Algorithm 1 (substituting ω for α).

Device i is *useful* for duration τ_i, defined as follows:

$$\tau_i = \begin{cases} \omega_i - \alpha_i & \text{if } \alpha_i < \omega_i \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

In the ideal case, each cycle spent on the computation has constant utility. Therefore the utility of a device, $u_i = \tau_i$, and the overall utility, $U = \sum_i u_i$. In order to achieve this, each device must be given enough work to occupy it fully between α_i and ω_i. This requires either an omniscient scheduler or a computation that can be repeated *ad infinitum*. Monte Carlo simulation is an example of the latter case. We evaluate a simple scheduler design in Section 3.

2.2 Real-world traces

We now evaluate the upper bound for several real-world scenarios, by applying the above algorithm to several human encounter traces. In this and the following sections, we use traces from various sources. These traces can be obtained from CRAWDAD [2]:

MIT In the MIT Reality Mining project, 97 smart phones were deployed to students and staff at MIT over a period of 9 months [9].

Cambridge In the Haggle project, 36 iMotes were deployed to 1st year and 2nd year undergraduate students at the University of Cambridge for 10 days [4].

Infocom 78 iMotes were deployed at the Infocom 2006 conference for 3 days.

In our first experiment, we consider the lifespan of a single computation. We simulate the execution of an ideal computation on the whole Cambridge trace, choosing each device in turn as the initiator. We record two metrics: the number of useful devices at time

[2]The initiator may broadcast ω_0 with the initial announcement in order to reduce the amount of wasted computation: however, this requires synchronised clocks.

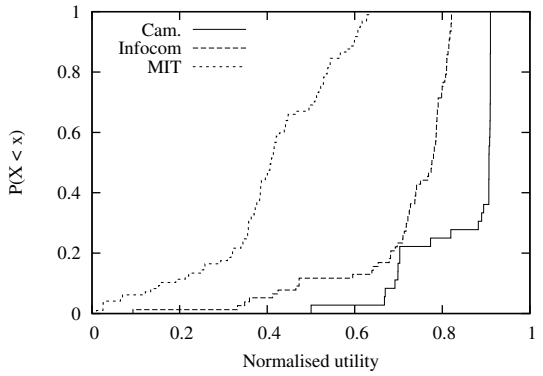

Figure 3: CDFs of normalised utility for the Cambridge, MIT and Infocom traces.

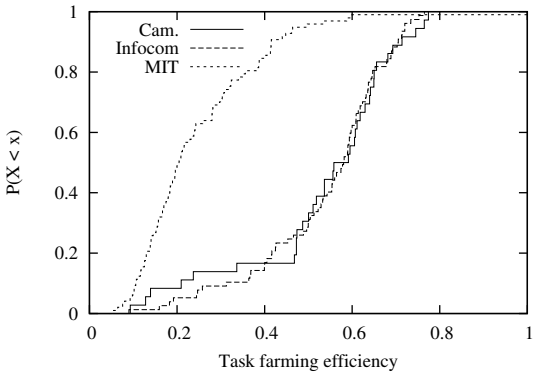

Figure 5: CDF of task farming efficiency for the Cambridge, MIT and Infocom traces.

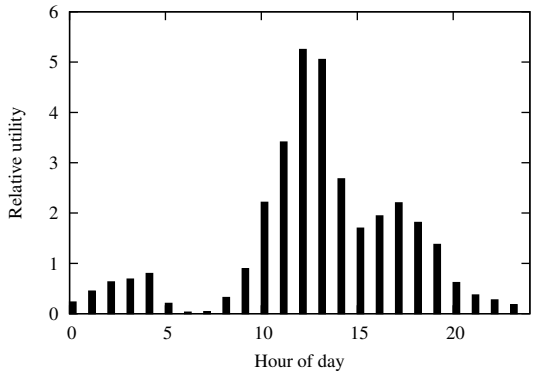

Figure 4: Relative utility for an hour-long job in the Cambridge trace, depending on the hour of the day.

t, $P(t)$, and the total utility of the computation, $U = \int_{\alpha_0}^{\omega_0} P(t)\,dt$. Figure 2 shows how $P(t)$ changes throughout the simulation, for the best, worst and average case. Note that the best case has at least 26 devices doing useful work until shortly before the end of the computation, whereas in the worst case, the initiator does not see any other devices after the halfway point of the computation. The average case achieves 93% of the best case total utility.

We now investigate the properties of different traces. Since each trace has a different duration and number of devices, we must normalise U in order to compare traces. Figure 3 shows CDFs of utility for the Cambridge, MIT and Infocom traces, normalised by the length of the trace and number of devices. Each trace exhibits different performance. The MIT trace has the worst performance, which we suspect is due to the relatively infrequent encounters between devices in a diverse group of participants [9]. By contrast, the participants in the Cambridge and Infocom traces were more homogeneous (all computer science undergraduates or conference attendees), and hence more likely to occupy the same space.

The amount of useful computation depends greatly on the choice of the initiator, which we investigate further in Section 4. The choice of start time (α_0) and finish time (ω_0) also have a predictable effect: running a computation at night or at the weekend, when encounters are rarer, leads to less parallelism and less utility. We ran one million simulated computations each lasting one hour, with the start time and initiator chosen uniformly at random, using the Cambridge trace. Figure 4 shows the average utility of these computations, binned into hours of the day. There is a period of approximately 10 hours each day when the utility is dramatically greater, including two hours of very high utility, which correspond to the peak lecture hours when most participants would be colocated.

3. TASK FARMING

In order to build a practical system for mobile distributed computation, we require a realistic scheme for achieving parallelism. In this section, we consider *task farming* as one possible scheme. We simulate the effect of task farming on several encounter traces, and evaluate its performance with respect to the upper bounds established in the previous section.

Task farming is the basis of many distributed computing systems, including Condor [22], BOINC [6], MapReduce [8] and Dryad [13]. In all of these systems, a single master process manages a queue of tasks, and distributes these amongst an ensemble of worker processes. When a worker completes a task, it requests another from the master. The algorithm naturally handles worker failure and load balancing [21]. Task farming is therefore an obvious candidate for distributing work in our crowd computing system.

We modify our model of distributed computation as follows. The overall job can be decomposed into a large number of atomic, independent tasks, which have a constant duration, d. The initiator acts as the *master*, which maintains a (potentially infinite) queue of tasks to be executed. All other devices are *workers*, which maintain a local queue of length c, and can process a task every d seconds. When the master meets a worker, it fills the worker's queue with up to c new tasks and collects the results of completed tasks. A *successful* task is one that has been processed by the worker and the result of which has been communicated to the master. (**N.B.** We assume that a useful result can be obtained from *any* subset of task results: however tasks may be lost, in which case task replication [6] or encoding [16] techniques may be appropriate.)

We simulated the execution of a task farming computation for the Cambridge, MIT and Infocom traces, choosing each node in turn as master. In these experiments, we set $d = 100$ seconds and $c = \infty$, which is the optimal configuration as no node will ever be idle once activated. We will discuss the effect of varying c and d later in this section. The utility of the task farming computation is simply the number of successful tasks multiplied by d. We can therefore compute the ratio of the task farming utility to the best-case utility for each configuration, which gives us a measure of *efficiency*. Figure 5 shows the CDFs of efficiency for the Cambridge, MIT and Infocom traces. Note that, as in Figure 3, the Cambridge and Infocom traces

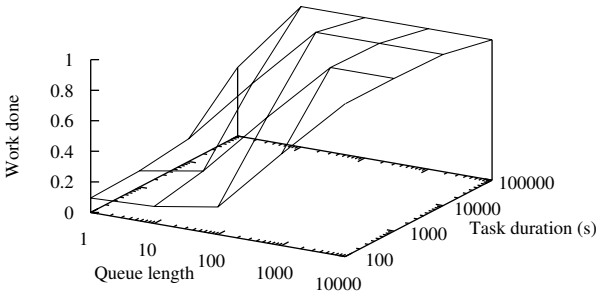

Figure 6: Effect of varying worker queue length (c) and task duration (d) on total successful work done. (Work done is normalised to make the optimal case equal to 1.0.)

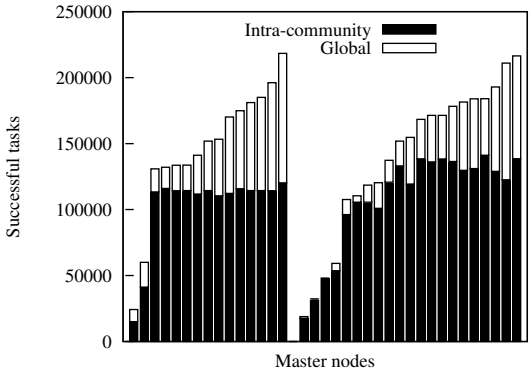

Figure 7: Comparison of intra-community and global task farming for the Cambridge trace.

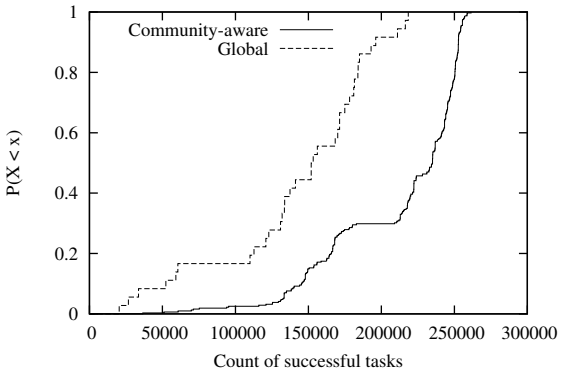

Figure 8: CDFs of successful task count for single-master and per-community master task farming.

outperform the MIT trace. The average efficiency across all configurations is 40.2%. We investigate methods of improving this performance in Section 4.

Realistically, our devices will have finite capacity (c), and the duration of tasks (d) may be longer than 100 seconds. There are two main challenges when setting these parameters. A short queue may lead to a device becoming idle if it exhausts all of its tasks before meeting the master again. We can partly address this by increasing d, but note that a task duration that is much longer than the master-worker inter-contact time means that opportunities to retrieve task results will be missed.

Figure 6 shows how varying c and d affects the overall amount of successful work in the Cambridge trace. The optimal utility is achieved with $c = 10^4$ and $d = 100$, which is equivalent to $c = \infty$, since it would take longer than the trace duration to exhaust such a queue. Setting $c = 1$ never yields more than 45% of the optimal utility. However, the configuration $c = 10, d = 10^5$ gives 92% of the optimal utility, while offering much greater flexibility and consuming fewer resources. We note that this is a large and complicated parameter space, and further investigation is required to set the parameters optimally.

4. SOCIAL-AWARE TASK FARMING

We can improve the efficiency of distributed computation by exploiting the social network formed by human interaction. Previous work has looked at the influence of graph structure [7] and community detection [11] on the efficiency of opportunistic networks used for communication. In this section, we investigate the use of community structure to improve the overall utility of a computation.

In the model of Section 3, the master communicates directly with the workers, so it must encounter them. Therefore, we naturally prefer to choose a master that meets a large number of other devices. If we had a single master, we might choose the device that meets the greatest number of devices in the most recent time period. However, human interaction exhibits community structure: the set of devices can be partitioned into groups that are highly connected, while having relatively few connections between groups [18]. Therefore, our naïve approach would achieve many successful task results from nodes in the same community as the master, but few from other communities.

If we knew the community structure, we could exploit it by assigning one master node to each community. We would also modify the task farming algorithm slightly so that workers only accept tasks from a master if it is in the same community. We expect that this would improve the overall utility of the system, because the community structure makes it more likely that a master will meet its worker again to collect the results[3].

The Cambridge data set has two communities: each is a different class of undergraduate students [24]. We divided the devices into their respective communities, and simulated task farming (i) using all nodes as workers, and (ii) using only nodes in the same community as the master. In both cases, $c = \infty$ and $d = 100$ seconds. Figure 7 shows the total utility in both cases, for each possible master. We see that, on average, 78% of the successful tasks are computed by nodes in the same community.

Our improved algorithm would choose one master from each community. We simulate this by computing the number of successful tasks for each pair of nodes in different communities. Figure 8 shows that we should expect two randomly-chosen per-community masters to outperform a randomly-chosen global master. On average, per-community masters complete 49% more tasks than a single master, and 62% of the community-aware configurations outperform the best global master.

The main limitation of this scheme is that there are now two masters that collect partial results, and we have not specified a way for them to communicate—indeed, they might never meet. We therefore require a protocol that enables the collection of a single, global

[3]This would also reduce the number of *unsuccessful* tasks: i.e. those that are processed without the master being notified.

result. One approach is to rely on a *deus ex machina*: we could give the master nodes access to some infrastructure—such as a satellite telephone—that enables them to communicate; or, if it is available, we could allow the masters to communicate over the cellular network. A more intellectually-satisfying approach would be to use opportunistic forwarding to exchange synchronising messages between the masters [19]. Both these solutions are costly (either in real money or extra bandwidth), and the cost increases with the number of masters, so this gives a natural trade-off between the performance of the system and its cost.

In this section, we have considered only simple task farming policies, and several enhancements are possible. For example, we could allow worker nodes to act as masters for other devices that they meet, and thereby build a spanning tree through the entire network. We could run an adaptive algorithm that selects the optimal nodes as masters and migrates the state as necessary. Indeed, if the computation decomposes spatially [21], or into a dependency graph (as in Dryad [13]), we could attempt to embed the problem domain into the encounter graph itself [25].

5. RELATED WORK

The idea of using mobile devices for parallel computation is relatively new. However, we draw on several related areas of research, which we summarise in this section.

As noted earlier, several systems achieve distribution and parallelism through task farming. Condor harnesses the idle cycles from a network of desktop workstations, and uses these to run batch-submitted tasks [22]. The BOINC project allows volunteers from around the world to process tasks on their desktop computers, for projects such as SETI@home, Folding@home and Climateprediction.net [6]. Task farming is also used in the data center. Google's MapReduce [8] and Microsoft's Dryad [13] both use task farming to schedule parallel processing on large (multi-terabyte) data sets. Each of these programming models could be implemented on top of a task scheduler for crowd computing.

We recently became aware of Hyrax, which includes a port of MapReduce to the Android operating system [15]. Hyrax assumes a relatively static cluster and treats device mobility as a problem of fault tolerance; by contrast, we show that it is often advantageous to assume that nodes will meet again in the future.

The use of network analysis in Section 4 is inspired by previous work in mobile routing. PRoPHET routing uses the history of past encounters in order to make probabilistic decisions about message forwarding [14]. SimBet routes messages via nodes that are "similar" to the destination, based on their connectivity [7]. The BUBBLE Rap algorithm uses community structure to improve message forwarding efficiency in a delay-tolerant (i.e. disconnected) network [11]. BUBBLE Rap also includes a simple distributed algorithm for community detection, which could be applied to selecting masters in our social-aware task farming system.

Wireless sensor networks also use mobile devices to perform distributed computation. *Directed diffusion* combines routing, caching and aggregation for data in a sensor network [12]. Welsh and Mainland describe a programming model for in-network processing of sensor data in order to reduce the bandwidth consumption [23]. We anticipate potential synergies between a sensor network that collects data, and a crowd computing system that analyses it.

6. CONCLUSIONS

In this paper, we have shown preliminary results that indicate the potential for crowd computing. Human interaction can be used to spread computation through an opportunistic network, and collect results. Furthermore, a simple task farming model can achieve reasonable performance in such a network, and dramatically better performance when community detection is used.

Due to the lack of space, this paper leaves several questions open. Power consumption is an important consideration: we must ensure that the crowd computation does not drain the mobile devices' batteries. A crowd computation should be energy-efficient, so the amount of wasted work (the results of which never reach the initiator) must be small. We must consider an efficient replication or encoding scheme that compensates for the loss of some results without reducing performance unduly. We intend to investigate these in future work.

Finally, we have presented only one realistic model for crowd computing: static task farming. Opportunistic networks are highly dynamic, and so we expect that an adaptive system will perform even better. For example, we rely on direct master-worker encounters in order to relay results, but it would be possible to do better by using opportunistic forwarding. In conjunction with the D^3N project, we are investigating programming models that directly exploit social structure [25]. Some programming frameworks, such as MapReduce [8] and Dryad [13], allow users to specify dependencies between tasks. This creates an opportunity for new scheduling algorithms that take execution order into account when assigning tasks to workers. We have recently ported our Skywriting runtime [17] to the Android operating system, and we are investigating how to make it exploit the unique characteristics of mobile devices.

Acknowledgements

We thank Malte Schwarzkopf for his constructive comments on a draft of this paper. This research is part-funded by the EU grants for the Haggle project (IST-4-027918) and the SOCIALNETS project (217141); and the EPSRC DDEPI Project, EP/H003959.

7. REFERENCES

[1] Amazon Mechanical Turk. `http://www.mturk.com/`.

[2] CRAWDAD. `http://crawdad.cs.dartmouth.edu/`.

[3] Folding@home PS3 FAQ. `http://folding.stanford.edu/English/FAQ-PS3`.

[4] Haggle. `http://www.haggleproject.org/`.

[5] Nexus One Phone - Feature overview & Technical Specifications. `http://www.google.com/phone/static/en_US-nexusone_tech_specs.html`.

[6] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the 5th international workshop on Grid Computing*, 2004.

[7] E. M. Daly and M. Haahr. Social Network Analysis for Routing in Disconnected Delay-Tolerant MANETs. In *Proceedings of MobiHoc*, 2007.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of OSDI*, 2004.

[9] N. Eagle and A. Pentland. Reality mining: sensing complex social systems. *Personal and Ubiquitous Computing*, V10(4):255–268, May 2006.

[10] M. Grossglauser and D. N. C. Tse. Mobility increases the capacity of ad hoc wireless networks. *IEEE/ACM Trans. Netw.*, 10(4):477–486, 2002.

[11] P. Hui, J. Crowcroft, and E. Yoneki. BUBBLE Rap: Social-based Forwarding in Delay Tolerant Networks. In *Proceedings of MobiHoc*, 2008.

[12] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1), 2003.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys*, 2007.

[14] A. Lindgren, A. Doria, and O. Schelén. Probabilistic Routing in Intermittently Connected Networks. *LNCS*, 3126:239–254, 2004.

[15] E. E. Marinelli. Hyrax: Cloud Computing on Mobile Devices using MapReduce. Master's thesis, Carnegie Mellon University, 2009.

[16] D. G. Murray and S. Hand. Spread-spectrum computation. In *Proceedings of HotDep*, 2008.

[17] D. G. Murray and S. Hand. Scripting the cloud with Skywriting. In *Proceedings of HotCloud*, 2010.

[18] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69(2):026113, Feb 2004.

[19] L. Pelusi, A. Passarella, and M. Conti. Opportunistic networking: Data forwarding in disconnected mobile ad hoc networks. *IEEE Communications Magazine*, 44(11):134–141, 2006.

[20] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of PPoPP*, 2008.

[21] L. Silva and R. Buyya. Parallel programming models and paradigms. *High Performance Cluster Computing: Architectures and Systems*, 2, 1999.

[22] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency: Practice and Experience*, 17(2–4), 2005.

[23] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proceedings of NSDI*, 2004.

[24] E. Yoneki. Visualizing communities and centralities from encounter traces. In *Proceedings of CHANTS*, 2008.

[25] E. Yoneki, I. Baltopoulos, and J. Crowcroft. D^3N: Programming distributed computation in pocket switched networks. In *Proceedings of MobiHeld*, 2009.