CrowdLang — First Steps Towards Programmable Human Computers for General Computation

Patrick Minder
Dynamic and Distributed Information System Group
University of Zurich
minder@ifi.uzh.ch

Abraham Bernstein
Dynamic and Distributed Information System Group
University of Zurich
bernstein@ifi.uzh.ch

Abstract

Crowdsourcing markets such as Amazon's Mechanical Turk provide an enormous potential for accomplishing work by combining human and machine computation. Today crowdsourcing is mostly used for massive parallel information processing for a variety of tasks such as image labeling. However, as we move to more sophisticated problem-solving there is little knowledge about managing dependencies between steps and a lack of tools for doing so. As the contribution of this paper, we present a concept of an executable, model-based programming language and a general purpose framework for accomplishing more sophisticated problems. Our approach is inspired by coordination theory and an analysis of emergent collective intelligence. We illustrate the applicability of our proposed language by combining machine and human computation based on existing interaction patterns for several general computation problems.

Introduction

Crowdsourcing markets such as Amazon's Mechanical Turk (MTurk) provide an enormous potential to perform a variety of different human intelligent tasks for both research and business in a fraction of the time and money used by traditional methods. Crowdsourcing provides massive parallel computation for tasks that are difficult for computers to solve by using human agents—we call them "human computers"—as information processors. This use of collective intelligence (also called distributed human computation) was shown to be successful for a variety of tasks such as image labeling, tagging and item categorization. These typical types of requests posted on crowdsourcing markets can be characterized by relatively low complexity in solution decomposition, low interdependence between single assignments and relatively little cognitive effort to complete. Therefore, the problems solved in these task markets are mostly easily parallelizable among crowd workers and do not use the full potential of collective intelligence.

However, as we move to more sophisticated problems such as writing articles or using the crowd as sensors in research projects the community has little experience in programming and treating crowd workers as operators for problem solving. Programming human computers is more complex concerning the required coordination mechanisms. Therefore, we believe that we need a more general programming language and general purpose framework, which include task decomposition, the identification of required coordination mechanisms, the integration of several dimensions of design attributes as well as the possibility to recombine identified emergent patterns flexibly (Bernstein, Klein, and Malone 1999) rather than only breaking a solution into its parts. Although we know that this idea of programming humans has negative connotations—just consider movies such as Modern Times or The Black Whole—we think that the combination of humans and machines has some important impacts which can help us solve bigger problems. Therefore, we believe that this gap between the interest in solving more sophisticated problems by combining human and machine computation and the tools for doing so lead to an open research question: *How can we combine emergent patterns of collective intelligence to invent accurate and efficient sourced applications for the crowd of tomorrow?*

Recent research partially concern this gap by providing programming frameworks and models (Little et al. 2009) (Little et al. 2010b) (Kittur, Smus, and Kraut 2011), concepts for planning and controlling dependencies (Zhang et al. 2011), and theoretical deductive analysis of emergent collective intelligence (Malone, Laubacher, and Dellarocas 2009). Taking the perspective that programming for the crowd can be more than just parallelizing tasks into multiple assignments and aggregating the results, in this paper, we present the concept of a general-purpose *programming language* and a *framework* which allows easy recombinations of existing successful interaction patterns and a sourcing of complex processes and general computation in the crowd.
Therefore, our contribution is threefolded: (1) We present an architecture of a general-purpose framework as a tool for computing sophisticated problems in the crowd (2) Based on basic interaction patterns (Malone, Laubacher, and Dellarocas 2009)(Malone and Crowston 1994) and design attributes (Quinn and Bederson 2009), we define the concept of a programming language which enhances

the designing and controlling of workflows for complex human computation tasks (3) We show the practical use of this concept by illustrating solutions for more sophisticated problems.

To this end, we first ground our idea by giving an overview of recent research and introduce the basic interaction patterns and design attributes of emergent collective intelligence. Then, we describe our general-purpose framework and introduce the concept of the programming language CrowdLang. We will conclude by the illustration of a complex computation in terms of CrowdLang, a brief discussion and our future work.

Related Work

Distributed human computation has recently received a lot of attention. At this point, we refer to (Quinn and Bederson 2009) which give a comprehensive overview about the topic in general and a classification of different genres such as "Games-With-A-Purpose" or "Crowdsourcing". Furthermore, recently different concepts and concrete implementations for programming and managing crowds arose. (Zhang et al. 2011) address the topic by proposing the use of the crowd as a general problem solver where only a problem statement as input is given and then the crowd has to guide the control flow of an algorithm. (Little et al. 2009) enabling the use of iterative tasks in MTurk. Therefore, they provide TurKit, a procedural programming toolkit. Furthermore, to avoid wasting money and time, they introduced a crash-and-rerun programming model (Little et al. 2010b) which allows a re-execution of applications after crashes by storing the results of human computation. Finally, (Kittur, Smus, and Kraut 2011) propose the use of CrowdForge, a framework for distributed human computation inspired by MapReduce. Their framework tries to coordinate human computation by generalizing the problem in three interdependent processing steps. Namely the partitioning, where the problem is decomposed in discrete subtasks; the mapping, where different tasks are accomplished by the crowd; and finally the reduce, where the results of multiple workers are merged.

Towards Programmable Human Computers

For building a general purpose framework and a programming language, an extended understanding of the fundamental building blocks of collective intelligence is required. Concerning the understanding of these dependencies and building blocks, (Malone, Laubacher, and Dellarocas 2009) examined different examples of Web enabled collective intelligence and presented them in a conceptual classification framework of building blocks using two pairs of related questions. Firstly, they considered staffing *(Who is performing the task?)* and different kind of incentives *(Why are they doing it?)*. This pair of questions allows a better understanding of different kind of incentives and the assignment of a task to a specific crowd worker in terms of design attributes of a programming language. An additional listing of design attributes is provided by (Quinn and Bederson 2009) which describe collective intelligence applications in terms of six different dimensions: *Motivation, Quality, Aggregation, Human Skill, Participation Time and Cognitive Load*. These two basic approaches build a good framework for the definition of the design attributes of the proposed programming language.

Furthermore, (Malone, Laubacher, and Dellarocas 2009) addressed the goal *(What is being accomplished?)* and the decomposed structure of crowdsourced general computation *(How is it being done?)*. With this second pair of questions, they analyze concrete problem-solving in terms of a task decomposition and a workflow, which recombines the basic fundamental blocks of collective intelligence systems. The main building blocks of these workflows—Malone et al. call them *genes* of collective intelligence—are variations of the generation of something new (*independent collection* or *dependent collaboration*) or decisions by the crowd (*individual independent* or dependent *group decision*). To identify the coordination requirements of a programming language we can also interpret these building blocks in terms of coordination theory, presented in (Malone and Crowston 1994), which is used in related fields such as distributed computation in general, workflow enactment or business process management.

A General-Purpose Framework

Today, one of the reasons behind the limited use of planned collective intelligence is the lack of tools for doing so and the little understanding of dependencies among different subtasks in problem solving.

Key Ideas

We provide a concept of a general-purpose framework and executable model-based programming language that can be used to compute complex problems by combining humans in the crowd and machines. We assume, based on (Kittur, Smus, and Kraut 2011), that the computation of a complex problem can be characterized by a sequence of tasks which starts with: (1) Define the problem statement; (2) Plan the solution method by vertically decomposing the problem in smaller subproblems; (3) Horizontally plan an ordered sequence of contributions by humans or machines for each subproblem; (4) execute the plan (we can also talk about executing an algorithm or a workflow description); and finally, (5) Aggregate the solutions for the subproblem to a solution for the initial problem statement. However, human actors have only bounded rationality and consequently such plans are often imperfect or the workflow for solving a problem statement is not well-structured (Bernstein 2000). Therefore, the framework has to allow run time changes for an original plan and support also unstructured plans.

Architecture

This leads us to a general architecture of the framework, presented in Figure 1. On the one hand we have a *Requestor*

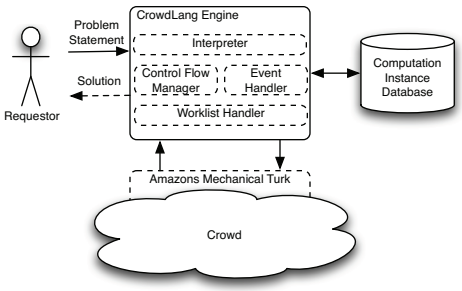

Figure 1: Architecture of the General-Purpose Framework

who wants to solve a problem by providing a problem statement. There are mainly two approaches to achieve the solution for the problem statement: First, by defining a workflow description through programming ex-ante or second, by defining only the problem statement and then let the crowd or machines rout the algorithm themselves. As we have mentioned before, it is our goal to provide a system which provides both highly unspecified and dynamic processes as well as specified routine processes. This first concept only concerns a framework and programming language for the execution of ex-ante defined workflows. However, this will help us in future to evaluate these two methods against each other by giving the same language to the crowd which allows them to manage the solution process on their own. The concept of such a programming language, called CrowdLang, will be elaborated in the next section. The *CrowdLang Engine* is responsible for managing a planed solution process and controlling the execution. The engine consists of four different modules. First, the *Interpreter* translates the problem statement into an executable control flow. Second, the *Control Flow Manager* manages the problem solving process by handling incoming events, creating new tasks, and routing the problem solving process. Third, the *Event Handler* manages different type of events such as finished assignments by crowd worker or exceptions by storing the state of the problem solving process in the *Computation Instance Database*. Similar to the *Crash-and-Rerun* programming model by (Little et al. 2010b) we have to ensure that the results of human computation can be reproduced for free after exceptions or crashes. Finally, the *Worklist Handler* has to manage the asynchronous interactions with the crowd, namely: publishing new tasks to the crowd (e.g. to MTurk), providing resources for the task (web interface) and required data, and recognizing finished tasks.

CrowdLang - A Programming Language

The programming language CrowdLang is inspired by common patterns of emergent collective intelligence and commonly used orchestration concepts such as those used in service orchestration or workflow enactment. As described above it is a part of the concept that both procedure-like and ad-hoc type parts can exist together by recombining variations of the fundamental building blocks such that it is not a prerequisite that the execution of the general computation is pre-planned. Furthermore, as also described in CrowdForge (Kittur, Smus, and Kraut 2011), the language support operators for the vertical task decomposition, the horizontal control flow of the algorithm, as well as for the aggregation of results. In the remainder of this section we introduce CrowdLang by presenting different operators and defining their semantics and coordination dependencies.

Fundamental Operators of CrowdLang

To enable an effective and efficient programming language, CrowdLang is based on a small set of operators which then can be aggregated to basic coordination patterns and the building block of collective intelligence.

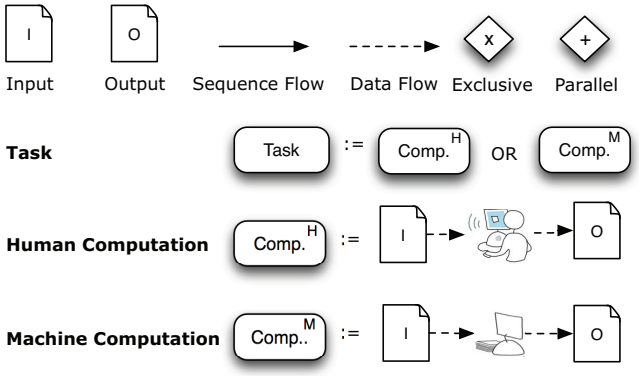

Figure 2: CrowdLang Operators

Transforming Problem Statements Into Solutions A *Task* is a single unit of work which transforms a given problem statement into a solution. More precisely a task can either be *Human Computation* or *Machine Computation*, whereby the transformation is done by humans or machines respectively. A *Problem Statement* and a *Solution* are data items which represent the problem by defining it and providing required data and a proposed solution respectively. *Data Flow* represents the consumption or production of a problem statement or solution. *Sequence Flow* defines the execution order of single tasks and manages therefore classical producer/consumer relationships in the form of a prerequisite constraint, where the produced output of a previous task is consumed as an input in the next task. Given these operators it is possible to represent a sequence of tasks such as initially translating a given sentence with a machine translator from German to English and then let the result of the machine translation be post-edited by a crowd worker. These operators are illustrated in Figure 2.

Routing, Distributing and Task Decomposition For enabling more sophisticated problem-solving, CrowdLang provides a set of routing operators for distributing computation and aggregating results, as illustrated in Figure 4. The *Divide-and-Conquer* and *Aggregate* operator are used to decompose a problem statement and to aggregate the results

of the computation in a vertical direction. The *Divide-and-Conquer* operator can be used for decomposing a problem statement into multiple parallelized subproblems. Hereby, P' and P'' are subproblems of the initial problem statement P such that $P' \subset P \wedge P'' \subset P$ and $P' \cap P'' \equiv \emptyset$. This decomposed parallelization of a problem statement allows to recursively break down a problem into several subproblems and solving these in parallel. On the other side the *Aggregate* operator can be used to collect the results of subtasks to the solution of initial problem statement, whereas $S' \equiv Solution(P')$.

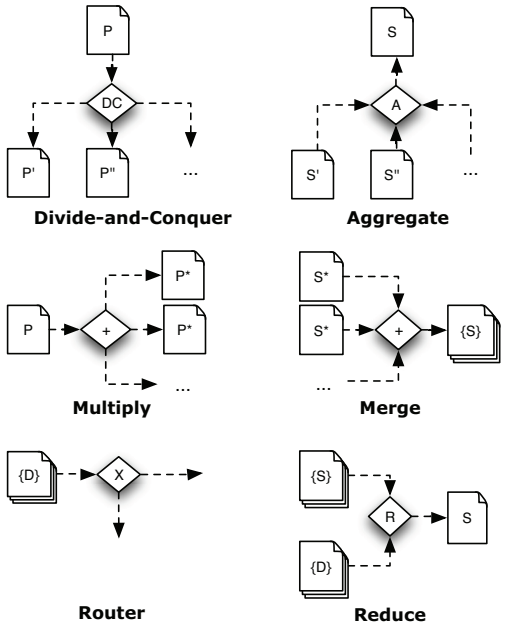

Figure 3: Routing, Aggregation, and Task Decomposition

Furthermore, CrowdLang provides routing operators for managing the horizontal control flow. The *Multiply* operator allows to compute a problem statement several times in parallel, by multiplying the problem such that each representation $P*$ is a copy of P. This operator transfers data in a sense of producer/consumer coordination dependency. In contrast, the *Merge* operator can be used to merge several different solutions of the same problem statement into a set of possible solutions $\{S\}$ and $S^* = solve(P^*)$. Additionally, the *Reduce* operator can be used to filter the best solution S from a set of possible solutions $\{S\}$ and a set of decisions $\{D\}$ by applying a reduce function. Finally, the *Routing* operator can be used to determine the correct execution path by evaluating the results of a voting mechanism. This feature is mainly used to model conditionals.

Building Blocks of Collective Intelligence

Finally, we are interested in the effective building blocks of human respectively machine computation for problem solving. Given a problem statement in form of a task definition and required data items humans or machines do some computation. Based on (Malone, Laubacher, and Dellarocas 2009) these contributions can be separated into two different basic genes called *Create* and *Decide*.

Create Genes (Malone, Laubacher, and Dellarocas 2009) define two variations of the create gene: Collaboration and Collection.

A *Collection* occurs when different members of a crowd contribute independently of each other. In contrast to Malone et al., which illustrated the Collection gene on posting videos on YouTube or contributing to a contest such as on Threadless, for our purpose we define a Collection as a multiplied independent transformation of a given problem statement into a proposed solution. As a fundamental building block, we define two variations of the Collection gene. A *Job* is a set of n-times multiplied problem statements which are computed in parallel by different independent agents and can be easily aggregated to a set of proposed solutions $\{S\}$. Jobs define a typical case of human computation used nowadays on MTurk. Furthermore, a *Contest* is a Job, whereby the proposed set

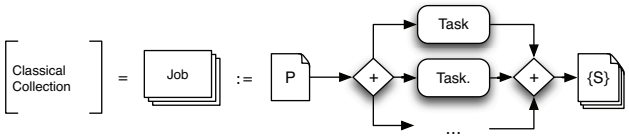

Figure 4: Jobs: Classical Collection

of solutions is reduced to the best solution by selecting a solution S based on a set of decisions $\{D\}$ contributed by the crowd or machines and the set $\{S\}$, presented in Figure 5.

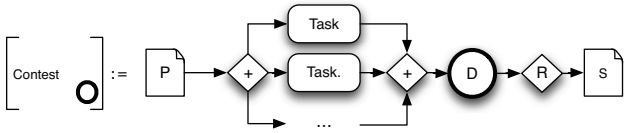

Figure 5: Contest

A *Collaboration* occurs when there exist dependencies between the contributions of a set of crowd workers such as when they work on different parts of a problem statement or improve a solution iteratively. We assume two variations of collaboration in the real world, as presented in Figure 6. First, we concern *Iterative Collaboration* as an *iterative process of interdependent solution improvement* whereas the submitted contributions are strongly interdependent on previous ones. The impact of this approach, where workers build iteratively on each other's work, is also presented in (Little et al. 2010a) and can be illustrated by writing Wikipedia articles. Based on a problem statement a

crowd worker builds an initial version of the solution. Then, either the crowd or a machine has to decide whether the current solution needs further improvements by voting or using a statistical decision function respectively. This basic sequence will be iteratively repeated until the crowd or machine decides that a current solution fulfills the qualitative requirements of the problem statement. Another variation

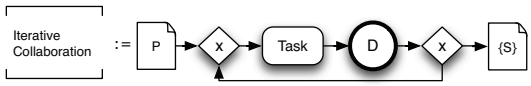

Figure 6: Iterative Collaboration

is *Parallelized Interdependent Subproblem Solving*, where crowd worker contribute solutions for subproblems which then have to be aggregated to the solution of the initial problem statement. Each branch of this pattern can use a sequence of jobs, decisions and reduction functions to build a high quality solution for a subproblem. The main advantage of this pattern, which can be illustrated for example in open source programming, is that it is possible to first split up a problem in a set of subproblems (e.g. the implementation of different classes) which are interdependent and then can be solved in parallel. Finally, these solutions have to be aggregated (e.g. integration into components).

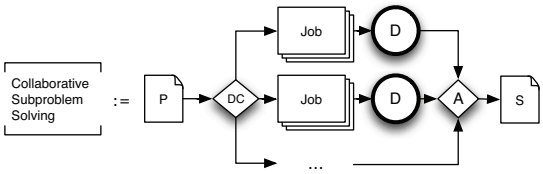

Figure 7: Collaboration: Parallelized Interdependent Subproblem Solving

Decide Genes Furthermore (Malone, Laubacher, and Dellarocas 2009) define variations of so called Decide genes which generate decisions for either a group as a whole or a single agent. They mainly distinguish between group and individual decisions, which is interpreted either as a decision which has to hold for a group as a whole (e.g. a winning design on Threadless) or as a decision that does not need to be identical for all (e.g. YouTube user decides for himself which videos to watch). For our purpose, as we assume that all decisions have to hold for the whole group, we interpret these genes more from a methodical point of view by asking the question: *How to evaluate the best contribution or predict the correct solution?*. These two variations are presented in Figure 8.

Therefore, we define a *Group Decision* as a mechanism which determines the best solution by using multiple crowd worker in an independent manner. An independent variation of a group decision could be the evaluation of different

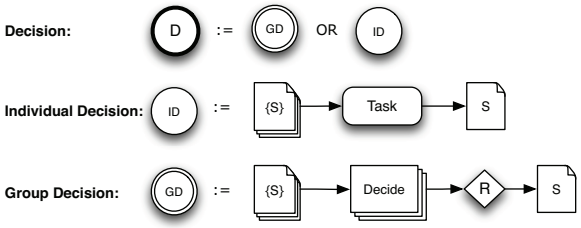

Figure 8: The Decide Gene: Individual and Group Decisions

solutions by voting, forced agreement or guessing in a parallelized manner and then to automatically select the best solution based on the results of these contributions.
Finally, we can define *Individual Decisions* as a decision by a single agent, as the truth for the whole group. Mainly, an individual decision can be defined as selection of a solution by a human or a machine. It is important to mention, that especially using a decision by a machine there can be use a huge set of statistical models such as predicting the accuracy of a solution in an iterative collaboration by predicting a dependency on the number of iterations.

Evaluation

We verified our formalism for the programming language CrowdLang by representing an existing crowdsourcing solution in terms of CrowdLang. As an example we choose the Find-Fix-Verify pattern by (Bernstein et al. 2010) for letting short text by the crowd. We choose this pattern because it is a good example for a multistage algorithm including both task decomposition and routing elements. We present the final representation of the pattern in CrowdLang in Figure 9. In the following we describe the pattern based on our representation.
(1) The workflow starts by splitting up the input into paragraphs which can be modeled using the Divide-and-Conquer operator. (2) In the Find stage crowd workers are asked to identify candidate areas for shortening in a paragraph, which can be represented with a classical collection, whereas the task is assigned to n-different crowd workers. This number can be modeled as a design attribute. If at least 20% of the crowd worker agree on a candidate which then can be further processed. We modeled this using a combination of (3) the Router operator for terminating the subprocess when no region remains and (4) the Divide-and-Conquer operator for parallelize the solving of each candidate. (5) Each candidate area then will be rewritten by five crowd workers in parallel such that the highlighted paragraph is semantically equivalent and syntactical shorter. Then in the Verify stage these additional versions of the paragraphs are evaluated by five other crowd worker by both deleting the significantly baddest version and the best version. Then resulting is a minimal set of patch candidates. We implemented the Fix and Verify stage using a Contest pattern. This pattern combines both the rewriting as well as the selection of the best can-

didate by forced agreement. (6) Finally, the resulting patch will be used by a computer program for replacing the selected area. (7) the results of the Fix-Verify stage and the (8) Find stage are aggregated to the resulting shorter text.

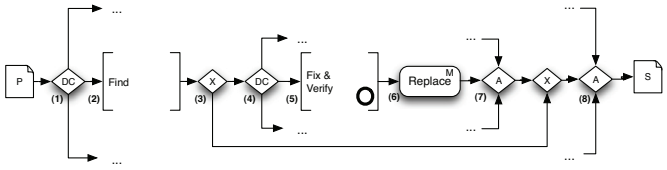

Figure 9: Visualization of Find-Fix-Verify by (Bernstein et al. 2010)

Contributions and Conclusions

We presented ongoing research and concepts for moving forward to harnessing crowds for general computation. The primary contribution of this position paper is twofold:

First, we introduced a novel concept for a general-purpose framework for crowdsourcing general computation. In future this framework can be used for easily deploying general computation into the crowd. By using this framework we can easily evaluate different design parameters such as the influence of reward functions, Q&A mechanisms, and different planing approaches on quality, price and throughput time. We expect that the use of this infrastructure will be helpful for both research and business because it allows to easily evaluate different research topics such as predicting quality, evaluating the influence of different process configurations, as well as enhance the use of the crowd for sophisticated computation in business.

Second, we introduced a first-version of CrowdLang, an executable model-based and crowd-aware programming language for defining workflows in the crowd. We showed that complex mixtures between crowds and machines can be constructed rather than just having them emerge by using CrowdLang such that we easily can decompose problem statements into executable process maps respectively algorithms. Even though the focus of this paper was not to empirically test this first concept, we provided some evidence to its plausibility by modeling an existing complex pattern for general computation in the crowd. Furthermore, by using CrowdLang as a standardized programming language for crowdsourcing it will be easily possible to recombine existing process maps onto new problem statements and evaluate different configurations of these maps on quality, cost and throughput time.

Future Work

In the next step, we will enhance the semantics of the language by including different types of design attributes such as reward functions or definitions for different types of human intelligent tasks. Furthermore, to show the practical use of this concept, we will implement the programming language CrowdLang and the CrowdLang engine and compare constructed processes for general computation against contributions of experts and fully automated systems and by evaluating the resulting quality, costs and throughput time as well as the usability of this programming language for programmers.

Acknowledgemnts

We would like to thank Tom Malone for his support to the ideas underlying this paper. Special thank also to Nicolas Hoby, Minh Khoa Nguyen, Cosmin Basca and Lorenz Fischer for valuable feedbacks.

References

Bernstein, M. S.; G; R; Hartmann, B.; Ackerman, M. S.; Karger, D. R.; Crowell, D.; and Panovich, K. 2010. Soylent: a word processor with a crowd inside. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*.

Bernstein, A.; Klein, M.; and Malone, T. W. 1999. The process recombinator: a tool for generating new business process ideas. In *Proceedings of the 20th international conference on Information Systems*, ICIS '99, 178–192.

Bernstein, A. 2000. How can cooperative work tools support dynamic group processes? bridging the specificity frontier. In *In Proceedings of the Computer Supported Cooperative Work*.

Kittur, A.; Smus, B.; and Kraut, R. 2011. Crowdforge: Crowdsourcing complex work. In *ACM CHI Conference on Human Factors in Computing Systems 2011*.

Little, G.; Chilton, L.; Goldman, M.; and Miller, R. C. 2009. Turkit: Tools for iterative tasks on mechanical turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation ACM 2009*.

Little, G.; Chilton, L.; Goldman, M.; and Miller, R. C. 2010a. Exploring iterative and parallel human computation processes. In *KDD 2010 Workshop on Human Computation*.

Little, G.; Chilton, L.; Goldman, M.; and Miller, R. C. 2010b. Turkit: human computation algorithms on mechanical turk. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*.

Malone, T. W., and Crowston, K. 1994. The interdisciplinary study of coordination. *ACM Computing Survey* (26).

Malone, T. W.; Laubacher, R.; and Dellarocas, C. 2009. Harnessing crowds: Mapping the genome of collective intelligence. Technical report, MIT.

Quinn, A., and Bederson, B. 2009. A taxonomy of distributed human computation. Technical report, University of Maryland, College Park.

Zhang, H.; Horvitz, E.; Miller, R. C.; and Parkes, D. C. 2011. Crowdsourcing general computation. In *ACM CHI 2011 Workshop on Crowdsourcing and Human Computation*.