Encouraging Parallel Thinking through Explicit Coordination Modeling
Sirong Lin and Deborah Tatar
Center for Human-Computer Interaction
Department of Computer Science
Virginia Tech
Blacksburg, Virginia 24061
silin@vt.edu, tatar@cs.vt.edu

ABSTRACT
Parallel thinking is a mindset that allows people to create support for activities that happen concurrently in a program. It cuts across extant computer science boundaries, including parallel processing, network programming and multi-user systems, indeed, any system that involves the distribution and reintegration of work. Recent efforts to integrate parallelism across the CS curriculum begin to address the support of parallel thinking. We approach the pedagogy of parallel thinking by teaching students to model coordination explicitly using a specialized coordination language. We report a study of an experimental class taking this approach, finding that advanced CS students lack a good understanding of coordination but that the explicit modeling of coordination can address this lack.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education

General Terms
Design, Experimentation, Human Factors

Keywords
Parallel thinking, pedagogy, TupleSpace, coordination modeling, parallelism, parallel computing, distributed computing

1. INTRODUCTION
As the computing world moves towards multi-core technologies, parallelism, though not a new concept, is, appropriately, attracting pedagogical attention. Multi-core technologies challenge almost every aspect of our assumptions about how computers operate and how they can operate more efficiently. Furthermore, connectivity is now a bread-and-butter business in computer science and complex connectivity implies parallel machines and parallel user actions. And ubiquitous and pervasive computing brings parallelism into the everyday experiences of non-computer scientists. Designing for virtually everyone on the planet is an essential and heavy responsibility.

Some argue that parallelism should be more than an advanced elective topic in CS. Indeed, Blelloch [3] has gone so far as to argue that for teaching parallelism from the start and embedding it throughout the curriculum with the idea of getting students to think about parallelism as the most natural form of computation and sequential computation as a special case. We agree that we need to encourage a parallel thinking mindset much earlier in the curriculum and broadly through multiple CS courses.

We define parallel thinking as a mindset that includes all the elements required to allow computer scientists to think about and implement support for activities that are allowed to happen concurrently in a program.

1.1 Support for a Pedagogy of Parallelism
In the past, most pedagogical attempts to improve teaching of parallel programming created new curriculum to make parallel programming techniques and algorithms easier to understand and learn, or developed new tools [2, 23, 26] to reduce difficulties in programming. For example, Robbins [26] reported using remote logging for teaching concurrency to help students visualize, understand, and debug concurrent programs.

Most recently, the pedagogical trend has been to integrate parallelism into the entire CS curriculum [13, 14, 16], from the entry-level courses. Ernst and Stevenson [14] attempted to increase students’ exposure to concurrency early and often in their curriculum through multi-threading programming. Wittman [13] also reported integration of concurrency into the freshmen courses through threaded programming. Their challenge was how to bootstrap concurrent programming with students having early, weak programming knowledge and skills. These were good steps of training students in a parallel thinking mindset. However, to program with concurrency concepts such as threads by the end of the first semester required acceleration in teaching Java in the first month [13]. Forcing parallel constructs could actually complicate the transition to parallel thinking in the sense that students may not learn either sequentiality or parallel thinking well. Furthermore, Wrinn [13] pointed out that curriculum design in parallelism education cannot keep up with the rapid development of hardware. Educators are trying to anticipate the state of parallel systems. These problems raise the questions “what essentials in parallelism should be taught?” and “how to integrate these essentials into the CS curriculum, early and broadly?”. Last year, a SIGCSE’10’s panel discussion called for ideas and solutions to “re-frame parallelism using simpler concepts, preferably ones that students can think about and understand in a natural way” [1].

We suggest solutions to these issues by sharing our experience teaching and studying an experimental class encouraging parallel thinking. In particular, we pick out a fundamental aspect of parallelism that we believe underlies much of the pedagogical challenge, modeling coordination, and we concentrate on that. We
propose that if coordination is understood well, other aspects of parallelism (syntax, optimization, efficiency, control, etc.) will be seen as solutions to the problems posed by the basic situation. Thus, we are proposing a pedagogical sequence, in the spirit of Bransford’s notion of contrasting cases [4], that starts by raising questions that it does not fully answer, leaving a conceptual gap for later instruction to fill in.

In alignment with the previous research, we also find that relatively advanced CS students do not understand coordination well. However, we find that by (1) teaching a coordination language that explicitly separates the coordination model from the computational model and (2) grounding students’ imagination in familiar real world concepts and experiences, we can allow students to encounter their conceptual barriers in both design and implementation.

2. STUDY OF PROGRAMMERS’ COGNITION

2.1 Coordination, Parallelism and Computer Science

Our focus on coordination is not new. Denning [11] identified coordination as one of the five “windows” of computing mechanics into the realm of computing. He defines it as “multiple entities cooperating toward a single result”. In this context, communication, a relevant concept to coordination, was defined as “sending messages from one point to another”. At their simplest, communication and coordination involve only two actors; however, we are interested in situations that involve more complex networks.

Parallelism is the natural way to solve complex problems in human problem-solving history: when one problem becomes too complex to solve by one person (e.g., specialist), we divide the problem into small subtasks, and each subtask can be conquered separately. In computing, parallelism becomes a natural topic when computer scientists pursue a way to run programs faster, no matter the goal is to keep multiple cores in a processor productive in calculating, or perform a complicated search over a distributed database. Despite many differences between parallel computing and distributed computing, one common aspect is the need of managing the physically independent processes, either processors or computers (over networks), to collaboratively achieve a common goal or accomplish a large task. From this perspective, both parallel programs and distributed programs are coordinated programs [10]. Coordinating these individual entities through allocating subtasks (the individual entity will finish the task asynchronously) and facilitating the communication to synchronize individual subtask is the central problem in the coordinated activities.

In early 1990s, Carriero and Gelernter, the authors of TupleSpace, anticipated “parallelism will become, in the not too distant future, an essential part of every programmer’s repertoire.” [10] Twenty years later, their claim has become true and been accepted by more and more people in the field. However, their claim of “coordination — a general phenomenon of which parallelism is one example — will become a basic and widespread phenomenon in computer science” has not been yet given enough attention.

To identify the process of learning that prevents programmers from more spontaneously and successfully approaching the design and development of parallel-distributed coordinated systems to support complex human interaction and coordination, we use a TupleSpace-based class to identify the cognitive challenges that programmers encounter in designing and programming parallel-distributed coordinated activities. The pedagogical goal was to examine the ways to help student programmers better learn collaborative system design and development. In this paper, we focus on discussing the results related to pragmatic meaning of training CS students with “parallel thinking”.

2.2 TupleSpace — a Coordination Language

Gelernter and Carriero brought attention to coordination in the computing field by promoting a specialized language, TupleSpace. Originally known as Linda [9], TupleSpace can handle complex coordination issues with simple atomic operations: out, in, and read. It allows coordination of different processes or machines by writing/reading structured data, in the form of a Tuple, to/from a common, globally-shared, associatively addressed memory space [20]. Template (a Tuple with one or more field unspecified) matching, i.e., contents matching, is used to search and return Tuples in/from the shared memory space. This interaction mechanism enables very loosely coupled communication in destination, space, and time [17].

The most significant benefit of using TupleSpace is to separate the coordinative activities from computational activities and create separate models — a coordination model and a computational model [15]. The role of a coordination model is to glue separate computational activities into an ensemble. The advantage of separation is its orthogonality, which further enables the generality and portability of the language. Therefore, TupleSpace can be added to any programming language, such as Java and C/C++. By using a specialized language, programmers can easily focus on thinking about coordination through the explicit exchange of resources.

TupleSpace has been used to successfully solve a number of coordination problems in the past thirty years in parallel computing (e.g. [8, 29]), distributed computing (e.g. [7, 20, 25]), and Human Computer Interaction (HCI) (e.g. [19]). The concept was implemented as network middleware, e.g. TSpaces by IBM, JavaSpaces by Sun, in the past as well. Most of these efforts treated TupleSpace as an underlying infrastructure to support the distributed structure. We explore TupleSpace as an infrastructure to support coordination.

2.3 The Class Design and Implementation

The experimental class, Designing Distributed, Networked Activities for Learning, was taught in the Department of Computer Science at Virginia Tech in Fall 2005 and subsequently in Fall 2006. The researchers engaged in teaching and studying the 4000-level semester-long experimental class. We tracked fourteen intermediate to advanced level CS students’ learning and using TSpaces, an implementation of TupleSpace, to design and develop multi-user, distributed, fine-grained collaborative systems. All students were in the fourth year of an undergraduate major or the first year of the graduate program in Computer Science at a technically oriented research-one university. Our students successfully designed and implemented a number of game-like coordinated systems, though not effortlessly.

The class in the first year consisted of 31 sessions (75 minutes per session), held twice/week. Four of the class sessions were dedicated to the students working on the team projects. Students
spent the first half of the semester learning background support and systems, TupleSpace concepts and programming, practicing design, and programming collaborative activities through various exercises and assignments. The teaching materials included Java, Eclipse and Swing/SWT tutorials, TSpaces programming materials from IBM website, published papers in HCI, CSCW, or CSCL areas (e.g. scenario based design (SBD), coordination, learning theory, education design, design rational, etc.), class lecture slides, sample programs and videos of users. The project vision was delineated as being one of Mock or Playground-like games, rather than more familiar computer games [6]. In the second half of the semester, class activities were devoted to support of design and implementation of the team projects, which involved all phases of design and implementation of a classroom pedagogical activity or a simple, multi-player, coordinated game. All teams (the students worked in pairs) produced working, playable games that varied in the kind of connectivity employed: Apples-to-Apples, Hangman, Pictionary®, the Algorithm Enactor, Collaborative Crossword, Krypto, and MathBingo [27].

Six senior undergraduates and eight first-year graduates in the CS program registered in our class and participated in the study. They received and signed informed consent letters. A separation was maintained between their grade in the class and their participation in the research, such that they could drop out of the research at any time without losing class credit, while none did.

Based on their programming expertise and education background, the students could be considered intermediate if not experienced programmers. However, as is often the case in Computer Science, they varied in knowledge and skills. Furthermore, they were all novices as coordination designers and programmers. Although many had created web pages, they did not have any experience thinking about coordination among users. One had previously designed or implemented a distributed multi-user application.

### 2.4 Research Approach
Following design-based research [5, 18] approach, data were collected primarily during the first class offering in 2005 with some supplementation from the second year. To maximize methodological triangulation and ensure the noting of potentially important phenomenon [18], multiple data collection methods were used. Research data include: (a) observation notes and videos of every class session, (b) semi-structured face-to-face interviews with open-ended questions conducted at the beginning and the end of the class, (c) documentation of in-class exercise and assignment programming source code, documentation, and reports, (d) documentation of email messages and class website board discussions, and (e) questionnaires and surveys.

Data analysis was guided by Grounded Theory, which uses a systematic set of procedures to develop an inductively derived theory about a phenomenon [12]. By combining these research data, summary tables of classroom activities and between-class progress were made; important classroom events were identified in the videotapes; the interviews were transcribed for analysis; the progress of project development was outlined from in-class discussion, assignments, and code registration. The data were coded and a series of codes were generated based on our research questions supported by the use of qualitative data analysis software, NVivo. Different categories of observed behavior emerged through multiple rounds of coding.

### 3. FINDINGS
The major findings from the study are that: (1) the overwhelming majority of students started with an undevloped imagination for coordination that affected their initial project design; (2) some students initially lacked the technical competence to model and implement coordinated activities; (3) TupleSpace appeared to help all students focus on and develop their understanding of coordination issues; and (4) familiar social coordinated activities catalyzed students’ imagination about coordination.

#### 3.1 Undeveloped Imagination for Coordination

With limited knowledge of distributed multi-user applications, the students encountered a variety of sociological and technological challenges in the class [21]. Among these challenges, almost all students showed an undevloped imagination of what was entailed in coordination. This lack emerged primarily in design phase but also required instructional attention in implementation phase.

The students showed great interest in examples of coordinated activities, and started to acquaint themselves with the concept of coordination. However, their preliminary project designs revealed that they used a fundamentally linear approach to address distributed activities. Almost all initial designs, for example:

1. failed to provide a coordination mechanism.
   A successful game implementation requires not only the support for game moves, e.g. laying down a card, but also the support for the interactions among multiple players. Game players need necessary information from each other to take different actions or have a satisfactory playing strategy at different stages during a game, e.g. either in a storyboard form or via the design of the appropriate Tuples protocol. However, the initial design ignored the necessities of supporting such a decision making process, or the exploration of alternatives. For example, the Krypto™ game involved a group of players collaboratively arranging five numeric cards and four arithmetic operator cards to calculate a target number. The designers initially proposed the support of strict turn taking, in which one person would lay down a card, then the next player, and so forth. This implementation did not pay enough attention to the essential coordination among players and neglected the fact: laying down each card severely constrained the possible solutions. The first players were under-constrained, while the last players were typically left with an unsolvable problem.

2. failed to take advantage of parallelism.
   Like Krypto™, the collaborative Pictionary™ and Collaborative Crossword games initially followed a pattern in which multiple players acted in strict sequence, although the primary benefit of making these games distributed was that multiple players could act at the same time.

Another way that initial designs failed to take advantage of parallelism was that students did not envision the idea that other players (like other processors!) might be waiting for something to do. For example, the first proposal for Collaborative Crossword puzzle game involved each player waiting while all the other players took turns choosing a clue and then entering a word. The only way that one player would know what another player had accomplished was when the entire word appeared on his/her screen. Failure to envision the “wait state” of the players...
was one example of a system element that needed more consideration.

(3) manifested difficulty envisioning all the resources on which the system could draw.

Another important system element that needed to be addressed was the choice of coordination paths. The collaborative games were intended to be played face-to-face, by human beings. This meant that the most ordinary of human capacities, such as talking, not only could be used, but constituted part of the fun. Yet all students initially proposed to implement separate IM-like channels as coordination paths among players. As a result, in games like Crossword Puzzle, rather than telling the other person that they had made a mistake, one person would write a message to the person next to her/him. While there was nothing a priori wrong with implementing such systems, neither was it a rational decision. It was an unexamined assumption that added a large implementation component without improving the coordinated nature of the game by anyone’s criteria.

These design impulses suggested that students brought strong presumptions and values into the design process, and that these presumptions and values did not initially include the envisionment of passing resources back and forth among multiple parties. They did not have deep understanding of coordination models.

In the implementation phase, some students also had difficulty envisioning the distributed structure. Even when students understood the mechanics of EventRegisters and Callbacks (a request for the client program to be notified on the occurrence of certain events), they had difficulty envisioning the sequences and flow created by the combinations of normal TupleSpace operations, posting Callbacks, and receiving them: “the user logs in, Callback would happen, I know that part. But after Callback happens, what happens next, you know, what happens to the information from Callbacks, like locally, what happens in TupleSpace?” (P01, post-class interview, December 04, 2005). Their superficial knowledge of Callbacks needed to be stretched.

### 3.2 Undeveloped Modeling of Coordinated Activities

The advantage of using a coordination language, like TupleSpace, to model coordinated activities is to let the coordination model handle coordinative issues, and the computational model to handle computational issues that do not require coordination and communication. As we found from the experimental class, a number of students had difficulty differentiating these two models. For example, some students:

1. used a shared resource, implemented as a Tuple, in both the coordination model and computational model, at different times.

Creating Tuples and using them to create the underlying communication protocol is the process of building the coordination model. Tuples form the coordination model and indicate coordination needs. In Krypto™, two types of Tuples, AnswerNumTuple and AnswerOpTuple were created to represent the number cards and operator cards. These Tuples were supposed to appear only in the coordination model when the end-user made his/her solution public. However, they were also used in computational model when an end-user moved a card around on his/her screen without showing the change to others. This design led to other users losing their own private work (attempts to move cards in their own screen) without a notification when one user broadcasted his/her change.

2. used a Tuple to represent aspects of the program that were not coordination resources or that contained mixed information about different resources. For example, in Pictionary™ activity, GameCreatingTuple had mixed information about the game and users.

Furthermore, the difficulty in imagining the underlying structure also revealed that a few students lacked the imagination of what actions should be counted as “local” events, things done by a single user’s machine, thus in the computational model; and what should be counted as “remote” events, things done by other users and reported to the central TupleSpace server, thus in the coordination model.

### 3.3 TupleSpace Helped Students Understand and Concentrate on Coordination Issues

Students were encouraged to use TupleSpaces constructs in the TSSpaces language to conceptualize coordination and its operations to form the internal communication protocols for each design and programming assignment. Students’ feedback about their learning experience with TupleSpace and the analysis of their Tuple design showed that TupleSpace helped all students focus on coordination issues.

First, students reported that the TupleSpace’s concept and basic operations were easy to understand and use. Almost every team reported that TupleSpace ultimately simplified their coordination design. Evidence of growth in understanding coordination was that many teams reduced the number of Tuples in the final design as they realized that coordination was simpler than they originally thought. Second, TupleSpace facilitated students’ conceptualization of coordination activities at both end-user level and system level. For instance, Apples-to-Apples® is a game that involves red and green “apple” cards that are resources that may be passed from one player to the group or vice-versa. The Apples-to-Apples team designed seven Tuples that delineated the coordination they wanted to implement through the computer; PlayerAnswerTuple conveyed information passed between end-users (e.g. player’s selection of a red apple card); RedAppleTuple and GreenAppleTuple were used to represent available resources in the game (i.e. red/green apple cards). Two students particularly pointed out in their interviews that they were “abstractions of data that is for interaction” with the help of TupleSpace. Pedagogical benefits of TupleSpace included that it:

1. represented physical space analogues naturally.

As a shared virtual space, TupleSpace can expressively represent: (a) the different resources, i.e. objects in the physical space, which can be defined by different types of Tuples; (b) the amounts of each resource, which can be reflected by the number of instance for each type of Tuple (represented by a Tuple class).

2. optimized the grain size in coordination to a right level.

TupleSpace helped students to find appropriate level of data abstraction — defined a proper grain size for coordination. For instance, in the Collaborative Crossword Puzzle activity, different players solved a puzzle by working on different clues. One person could take a clue and fill in the answer in the board while others worked on other clues. Thus, conflict could exist between two clues if they shared a cell in the grid. The team changed the grain size of a Tuple from “clue” to “cell” in the
design stage, thereby simplifying the implementation of both the underlying data structure and the corresponding UI.

(3) hid parallel and networking details from students.
TupleSpace (and some extensions in TSpaces) hid many implemental details, such as sockets and threads. Similarly, many general parallelism terms such as race condition and locking did not appear in the class discussion. These simplifications helped students concentrate on facilitating coordination, which is also useful for other courses.

3.4 Familiar Social Coordinated Activities Can Catalyze Students’ Imagination of Coordination
Familiar social coordination activities were brought into the class as a reference frame to catalyze students’ imagination. The activities:

(1) inspired students’ ideas of designing interesting and effective features to support coordination. Some students played familiar collaborative board games while others observed.

(2) helped students focus more clearly on a setting of coordination between multiple parties. We brought in videos of children playing playground games and also read anthropological accounts of game playing which focus on negotiation, rule and role creation as a crucial component. The students were encouraged to think of coordination through playground game play [22].

(3) helped students identify problems in the designs, e.g. why strict turn-taking does not work under certain circumstance. We asked them to act out (engage in the game and play) their own designs team by team. As in the familiar HCI technique of designing with paper user interfaces [28], students could use paper, cards, and whatever else they needed, as if the end-users were interacting with the computer user interface and each other. By passing pieces of paper back and forth to represent the exchange of resources of different types, students could understand the processes more easily. They witnessed the details and the whole of their activities, the necessity of certain technical features, whether the end-user coordination was interesting or fun, and the interplay of information shown on the screen and that spoken out loud. They also could see (and hear!) people who were “abandoned” or had nothing to do for long periods of time. These experiences highlighted the effects of the coordination and computational models and implementation issues such as the grain-size and nature of sharing. It also helped them see whether their implementation schedule was realistic.

4. DISCUSSION
Our findings confirm that the students had difficulty imagining both coordinative activities and separating a model of coordinative activities from other aspects of computing. However, by calling coordination out explicitly, via the creation of a TupleSpace model and in conjunction with familiar reminders of ordinary coordinative experiences, they were able to improve. By cultivating imagination for coordination, we also promoted parallel thinking. We created opportunities in which design decisions about coordination in a parallel context can be made the subject of critical reflection.

TcpSpace gave students the tools to create models. The initial models were incomplete and unsatisfactory, but within a semester every team was able to implement a working game with both coordinative and computational models.

4.1 Using a Coordination Language in Coordinated System Design
As expected, TupleSpace played an important role in our class. First, it helped students differentiate coordination issues from computational ones, so that a coordination model could be created. Second, through the design of expressive resource types, amounts and limits, it effectively helped students envision the virtual space (system) that they built. Third, it facilitated the consideration of coordination grain size. Last, students were not overwhelmed by a complex language or by the need to handle cumbersome programming tasks, e.g. creating sockets connection, overhead of managing

TupleSpace laid the groundwork. However, the students still had difficulty knowing what to do with coordinative capacities. Few students were initially able to identify potential coordination features in a collaborative activity. Familiar social activities such as games helped them envision the systems more clearly. Other more technical approaches are possible. For example, we could teach related knowledge and techniques such as identifying dependencies; however, we believe that drawing on more fundamental human experiences might draw a wider range of students into a deep understanding of concurrency.

4.2 Coordination is Fundamental in Parallelism Education
Students faced different challenges in conceptualizing coordination: presumptions about coordination, difficulty that they had envisioning that players might have gaps in their activities, and difficulty they experienced envisioning the sequences and flow created by the combinations of normal TupleSpace operations and distributed events. The capacity to envision the coordinative situation, what supports it, what detracts from it, and how it could work better constitutes the mindset of parallel thinking that is essential for further development. We suggest that coordination should be an early focus in teaching parallelism. Focus on coordination should concentrate on basic concepts and mechanisms related to concurrency, rather than relatively mutable technical details.

4.3 Collaborative Games — a Familiar Way of Teaching Parallelism
Games have been used to teach different topics in computer science. We used collaborative playground and board games first because they naturally involve rich coordinative features, e.g. communication need, negotiation process, resource allocation and limitation. Second, they are familiar from childhood. Many parallelism concepts and techniques, such as synchronization/asynchronization communication, one-many/many-many interaction, and resource/task dependence are easily understood in such a context. Similarly Neeman, Lee, etc. [24] successfully taught non-computer scientists with analogies and storytelling (e.g. Jigsaw Puzzle) to illustrate key concepts in parallel computing to avoid technical details. Compared with these, coordinated games have richer coordination features and can give students an experience akin to “direct manipulation”.
5. CONCLUSION
Our study of student programmers’ conceptualization of the design and development of multi-user, parallel-distributed, coordinated systems promoted a parallel thinking mindset through the focus on coordination. Students encountered a raft of technical and imaginative problems in such a way that they were able to make progress on them and come to understand the area more deeply. The implications for CS education are that (1) coordination should be taught as a fundamental topic in parallelism education; (2) explicit modeling coordination is crucial to coordinated system design and development; and (3) familiar social coordination activities can be used to catalyze students’ imagination of coordination in a natural way.

6. ACKNOWLEDGMENT
This work was supported by NSF ITR/IERI Grant #REC 0427783. Thanks to all the members of the Tuples project, and CS4984 from Fall semester, 2005.

7. REFERENCES