On the Codification of Coordination: An Ontological Tool for Pattern Mining
Celina Gibbs, Katherine Gunion, Yvonne Coady
University of Victoria
{celinag, kegunion, ycoady}@cs.uvic.ca

ABSTRACT
One of the challenges developers face when dealing with parallelism is that purely static views of code tend not to reveal internal dynamics and causal relationships that can be problematic. This paper considers parallelism from multiple perspectives—from kinesthetic exercises involving elementary school children, to lines of code spanning six different parallelization mechanisms. We attempt to reconcile these views and develop an ontology designed to support pattern mining in parallel code bases.

We show that, both kinesthetically and in the code bases, coordination emerges as a subtle entity that is difficult to identify in a coherent and conceptually concise manner. We believe that low level implementation patterns and micro patterns will be crucial for comprehension of coordination, and propose tool support for effective mining. Further, we suggest a means of accomplishing a semi-automated ontological mapping for parallelization mechanisms, and offer this up to help designers uncover patterns and pattern compositions.

1. INTRODUCTION
With the growing number of parallel support mechanisms available and the considerable variation between them, patterns hold great promise to provide language-agnostic knowledge representation of crucial relationships. In particular, patterns are key in conveying relationships involving coordination—one of the most subtle features of parallel implementations.

As opposed to thinking about patterns early in the software lifecycle, we are interested in the design of patterns for program comprehension. In order to provide a platform for pattern designers, this work starts with a general ontology for conceptualizing key entities in parallelism. This ontology could provide a means of sharing knowledge, reasoning strategies, and even asserting specific propositions about any implementation that involves concurrency. Overlaying this ontology directly onto code could complement the modular decomposition of the system, support pattern mining, and aid reasoning about temporal or dynamic relationships.

In an attempt to best capture an effective vocabulary for key entities of parallelism, we started in a classroom of grade 7 students and observed their natural solutions to real-life parallelism, such as having two people wash dishes (Section 2). Results from this experiment form the top level of our proposed ontology: computation and communication. We then take these entities and apply them to six different code bases, each using a different parallel support mechanism (Section 3). We find significant overlap in this coarse grained entity classification, and discover that deepening the ontology not only clarifies the overlap, but supports the emergence of another top level entity: coordination (Section 4). Initial validation of this proposed ontology is performed from several perspectives: in the context of kinesthetic exercises and Petri nets (Section 5), and as a full mapping onto the original code bases (Section 6). Finally, a proposal for semi-automated tool support and the potential application of this tool for pattern designers is presented (Section 7).

1.1 Background
Current trajectories suggest that future hardware platforms will house thousands of cores. Nvidia's Tesla 960 cores for less than $10,000, demonstrates the reality of our situation [10]. Millions of cores are even available, such as IBM's project KittyHawk [12]. Parallelism introduces critical issues such as resource utilization and contention which had largely been factored out of mainstream development practices for high-level applications executing in a sequential environment. Though parallelism itself is not a new challenge, the current state of flux for applications and the degree to which they need to be transformed is relatively new and somewhat alarming [9].

The daunting task of efficient programming for highly parallel systems is currently receiving much attention from several perspectives within computer science [11]. This offers an opportunity for researchers to rethink programming models, system software, and hardware architectures from the ground up. Though Amdahl's law [13], simply put, can be used to find the maximum expected speedup for an overall system when only part of the system can be parallelized, the question as to which part of the system should expose the details of concurrency—the application, the infrastructure, or both—remains an open issue. Current parallel support mechanisms range from fine-grained, architecture specific control to high-level abstractions that conceal some of the complexity and minimize developer control.

In this work we consider mechanisms for low level C-based systems. Though this is admittedly one small domain in the parallel universe, the variety of mechanisms for parallelization is considerable. For example, while PThreads [15] provides an interface that is flexible (and dangerous), OpenMP [7] trades this in for simplicity, with compiler pragmas lightly sprinkling a code base. These pragmas generally identify computation that can be parallelized without any explicit need for a developer to think about communication/coordination issues on the partitioned data set.

Approaches based on MapReduce [1] are increasingly available in opensource frameworks. Designed for distributing data across nodes in a cluster, and performing simple computations in a fault tolerant fashion, the tradeoff once again is in flexibility for
simplicity. MapReduce provides well defined semantics for parallelized computation, and conceals communication through shared files.

A growing number of approaches to parallelization include additional processing units that are typically not as general purpose as the host they support. In this category, we consider the Cell processor [4] and devices harnessing large numbers of GPUs [10]. Parallelization mechanisms include the Cell programming support [4], CUDA warps [6], and the new language specification for OpenCL [5]. Though OpenCL is relatively new, its additional linguistic support to manage the asymmetry and shared data models in these environments will most likely be a substantial contribution to the state of the art.

2. KINESTHETIC PARALLELISM

In the early 80's, Multilogo was the first implementation of Logo that supported multiple turtles running together. Early indicators demonstrated that children adapted quite well to an environment that included parallelism [22].

Inspired by this work and the Computer Science Unplugged Team [21], we developed a series of kinesthetic learning activities related to parallelism. Ultimately, we are interested in the ways in which natural perspectives align with realities in today's parallelized computational settings.

Breaking a task into subtasks that execute concurrently is easily mapped into a setting where participants themselves perform independent subtasks. In each of the activities we developed, we drew from examples of real world parallelism—going to a movie, washing dishes or crossing a single lane bridge.

Here we consider the after dinner task of washing the dishes. It involves two participants, and a sense of urgency to get the job done. The elements of work that need to be divided in this case are the subtasks: (1) washing each dish in soap and water, (2) drying each dish and (3) putting each dish away in the cupboard. The students must consider the ways in which the different pieces of work relate to one another—they are faced with issues of heterogeneous subtasks, communication and workflow, as highlighted in Table 1.

Table 1: Dishwashing activity and related concepts.

<table>
<thead>
<tr>
<th>Description</th>
<th>Find the most efficient way to do the dishes with two people, given a breakdown of the job into 3 subtasks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concepts</td>
<td>Workflow</td>
</tr>
<tr>
<td></td>
<td>Subtasks</td>
</tr>
<tr>
<td></td>
<td>Communication</td>
</tr>
</tbody>
</table>

We presented this scenario to a group of 36 grade 7 students, and asked questions to determine their methodologies for dividing and organizing the tasks. After the students helped the facilitators act out the dishwashing scenario, they were asked four questions. Three questions forced them to think about workflow, and one to commit to a final plan. The three questions that had the students consider workflow were designed to force the students to contemplate the runtime of each task, as well as allow for us to examine the results for a co-relation between questions 1-3 (Figures 1, 2, and 3) and question 4 (Figure 4).

1. Which of the following tasks is the fastest for a single dish?
   a) Washing the dish
   b) Drying the dish
   c) Putting the dish away
   d) Each task takes the same amount of time

   ![Figure 1: Student responses to question 1.](image1)

2. Which of the following tasks is the slowest for a single dish?
   a) Washing the dish
   b) Drying the dish
   c) Putting the dish away
   d) Each task takes the same amount of time

   ![Figure 2: Student responses to question 2.](image2)

3. Which is the faster way to put away a set of dishes?
   a) Putting away one dish at a time
   b) Putting away multiple stacked dishes at once
   c) Neither, they would take the same amount of time

   ![Figure 3: Student responses to question 3.](image3)

For the majority of the students, the responses demonstrate an appreciation that the tasks vary in terms of their runtime, and further appreciated a concept related to data parallelism, involving multiple dishes.

4. Based on your answers to the first three questions, how would you split up the work between you and your friend?
a) Both of you work on all three steps together, completing one step at a time.
b) One of you washes the dishes and the other person dries the dishes and places them on the counter. When all the dishes are clean, you both put the dishes away together.
c) One of you washes the dishes and the other person dries the dishes and puts them away.
d) One of you washes the dishes and the other person dries the dishes and puts them away. When the washer is done washing all the dishes, they put the clean dishes away.

Figure 4: Student responses to question 4.

Though there could be many reasons for these answers, ranging from social to technical, the majority of the students chose (b). Looking at the general flow of their responses (Table 2), the students felt that washing a dish was slowest, and drying a dish was fastest; the group response (b) seemed predictable. This answer identifies the division of tasks and also removes the most contention from the situation.

Table 2: Synopsis of student responses

<table>
<thead>
<tr>
<th>Question</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Answer</td>
<td>#</td>
<td>%</td>
<td>#</td>
<td>%</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>8.6</td>
<td>20</td>
<td>58.8</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>31.4</td>
<td>6</td>
<td>17.6</td>
</tr>
<tr>
<td>C</td>
<td>15</td>
<td>42.9</td>
<td>2</td>
<td>5.9</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>17.1</td>
<td>6</td>
<td>17.6</td>
</tr>
</tbody>
</table>

Observations during the activity revealed many other unanticipated suggestions, showing a definite appreciation of underlying synchronization challenges and creative solutions. These included introducing a dry tray (a buffer), and using predictions of speed based on previous dishwashing experience.

2.1 Autocatakinetic Systems

When humans work together they have many characteristics of an autocatakinetic system:

Autocatakinetic systems are those (a) that continuously self organize, (b) whose global identities are maintained through continuous dynamic activity, and (c) that maintain themselves “by pulling potentials or resources into their own self-production”. [23]

Though we do not expect parallelized software to be autocatakinetic, we observed that the nature of interaction in the dishwashing scenario is greatly facilitated by the multi-sensory communication used by the students to complete the task successfully. Here we begin by considering their approaches strictly in terms of two key coarse grained ontological entities in this system, computation and communication.

Perhaps the most traditional lens through which we can view the exercise is in terms of task parallelism as opposed to data parallelism. Students were comfortable with the concept of subtasks with different physical acts of washing (soaping, scrubbing and rinsing), drying (picking up drying with a towel, putting down), and putting each dish away (carrying to the cupboard, placing on the shelf). Though, kinesthetically, these subtasks can be encapsulated and thought of in isolation, the students also appreciated the system perspective in which the computation implicitly involved communication between entities (the two dishwashers). Here we witnessed their autocatakinetic abilities, aided by subtle multi-sensory inputs, outputs, and feedback loops. That is, communication of the status of the towel, the sink and the dishes, along with the intended actions of each of the participants are relayed using both visual and verbal communication.

Most importantly, we found that the distinction between these two categories—computation and communication—is not exactly discrete and arguably there is overlap; the state of a computation can constitute subtle communication. This intersection introduces some interesting and subtle interactions between the subtasks. The washer communicates that data is ready in the buffer (drying rack) by placing a dish in the rack, and the dryer visually sees that there is a dish in the rack. The actions are interleaved with computation. For example, placing a dish in the rack is the final step in washing subtask. Similar interactions exist with regards to the state of the towel and the sink. The students worked comfortably, and arguably almost subconsciously, with the fact that different designs of the entire system were dictated by these subtle relationships between computation and communication.

3. CASE STUDIES

This section focuses on the two case studies centered on two parallelizable problems: a simple reduction and Fast Fourier Transform [2]. Each case study analyzes three implementations, in total spanning six different parallel support mechanisms.

Though the case studies consider different problems, we focus on the common elements that launch the parallelization. That is, the break down between the host which handles the distribution of tasks and data across the workers, which correspond to the units that perform the computation. Though this study considers only a small portion of FFT, we chose it to demonstrate the need for additional conceptual support at this phase of the computation, and the common elements that exist between all six implementations.

3.1 A Simple Reduction

Arguably a “poster child” for parallelism, here we consider the implementation of a simple reduction involving the summation of a large set of values. The implementations surveyed here employ three different parallelization support mechanisms: a framework for MapReduce [20], OpenCL [5] and CUDA [6]. The MapReduce framework allows developers to write specialized map and reduce functions, whereas CUDA and OpenCL localize code that will execute on the workers in a separate kernel module.
Though the implementation of the reduction algorithm varies considerably between MapReduce (Figures 5 and 6), CUDA (Figures 7 and 8) and OpenCL (Figures 9 and 10), we begin by considering each in general terms of computation and communication, which form the highest layer of our simple ontology.

### 3.1.1 Computation

The user defined map and reduce functions, shown in Figure 5, constitute the application specific code to be parallelized by the underlying framework. These functions house the core computation of this algorithm, revealing the two loops that constitute the core calculation (Figure 5: lines 5-7, 17-20). The corresponding CUDA and OpenCL kernel modules similarly contain the computation, albeit couched in a slightly more optimized and architecture specific form than MapReduce.

### 3.1.2 Communication

MapReduce uses an emit function to communicate intermediate results from the worker to the host (Figure 5: lines 10, 25) while the host uses shared memory (Figure 6: line 8) for communication of data to the workers (Figure 5: line 15). The final result is retrieved by the host through a shared memory space (Figure 6: line 30).

CUDA and OpenCL use memory copying and synchronization mechanisms for communication at the kernel level (Figure 7: lines 6, 12, 15 and Figure 9: lines 11, 15 respectively). At the level of the host, CUDA makes use of library synchronization mechanisms (Figure 8: lines 36, 51) and synchronization supported memory copying (Figure 8: lines 23-26). Similarly, OpenCL uses library functions with built in buffers for distributing data to workers (Figure 10: line 12) and reading back the result from workers (Figure 10: line 25).

### 3.1.3 Dual Responsibilities: Computation and Communication

For many of these lines of code, it is actually surprisingly difficult to align them with just one entity, as they serve dual purpose in terms of both computation and communication. Something as simple of an assignment to shared memory has this kind of property. Similarly, the considerable effort invested to handle resource setup in terms of processing units and memory in each implementation serves this dual purpose: it computes provisioning, in a way that also serves as communication.

In MapReduce, data partition size, number of elements and memory allocation are dealt with in the worker functions (Figure 5: lines 2, 3, 14) and at the host (Figure 6: lines 1-5). The MapReduce host deals with the majority of the argument setup for the task execution and data distribution. For example, the selection of a splitter and partition function (Figure 6: lines 14, 17) for the communication of intermediate results and distribution of data. Again, much of the parameter tuning occurs in the host (Figure 6: lines 19-24), including cache size and number of figures.
threads (Figure 6: lines 20-23), lie in this intersection between computation and communication.

```c
int main(int argc, char** argv)
{
    int typeChoice;
    int desiredMatch = 0;
    int numThreads = 1;
    int sharedDevice = 0;
    int stride = 0;
    int groupSize = 8128;
    if (GROUP_SIZE <= 812)
    {
        return 0;
    }
}
```

In OpenCL, library support abstracts communication details in the form of program, context and command queue setup (Figure 10: lines 5-9, 17, 18). Similarly, buffer creation to house data for computation and for distribution of data for host/worker communication (Figure 10: lines 11, 14, 15) is library supported.

Both the CUDA (Figure 7: line 3) and OpenCL (Figure 9: line 4) kernels have single lines of code that perform both computation in the form of addition and communication through storing the result in global memory. Additionally, CUDA requires a combination of explicit device setup and synchronization mechanism (Figure 8: lines 7-13) required for communication.

Though synchronization is not an explicit form of communication, we argue that a developer must still consider how the interleaved synchronization calls are managing the unwanted communication (through shared memory) between the host and the workers and even between the workers themselves. Even the explicit identification of the functions requiring no synchronization (Figure 8: lines 23-26) requires a developer to think about the nature of communication involved. Invocation of computation also overlaps with the communication for storage of results on host, while computation is farmed out to the workers (Figure 8: lines 28-29). Synchronization mechanisms are also interleaved within the CUDA host code to ensure correct workflow through the kernel (Figure 8: lines 36, 51).

```c
int main(int argc, char** argv)
{
    int sharedDevice = 0;
    int groupSize = 8128;
    if (GROUP_SIZE <= 812)
    {
        return 0;
    }
}
```

3.2 Fast Fourier Transform

Our second case study considers the FFT algorithm that was first published by Cooley and Tukey in 1965 [16], though we limit or view of the code to the initial launch of the first phase of parallelism. The three implementations under analysis are drawn from the Fastest Fourier Transform in the West (FFTW) [2], a library of highly tuned and performance centric FFT implementations. This study investigates implementations using POSIX Threads (PThreads) [15], OpenMP [7], and Cell [4] libraries for support. We begin again with an analysis of the PThreads (Figure 11), OpenMP (Figure 12) and CUDA (Figure 13) implementations first in simple terms of computation and communication.
Given that this case study considers only a portion of these three code bases, even though FFT involves a more complex algorithm, the amount of code designated purely to computation is small, even in comparison to the simple reduction. Specifically, in this analysis we examine only the breakdown of the computation and its dispatch to the functions which perform the actual sub-transform computations.

In the main loop of the PThreads implementation shown in Figure 11, a chunk of data is selected and passed on for parallelized computation (Figure 11: line 13). The OpenMP implementation has a similar loop preceded by an OpenMP directive: ```#pragma omp parallel private(d)``` (Figure 12: line 2). The OpenMP directive, if enacted, essentially flattens this loop and the computation on each chunk of data is handled concurrently by an individual thread. So, though the directive is a catalyst for possible communication, from this developer perspective, the whole loop (Figure 12: lines 3-10) can also be classified as merely computation.

The Cell specific implementation has a similar modular breakdown to that of OpenCL and CUDA in that it provides host code (Figure 14) that will run on the Power Processing Unit (PPU), overseeing operations (Figure 13) running on each Synergistic Processing Unit (SPU). This FFT SPU code loops through a simple finite state machine, moving through the stages of the computation kicked off by the PPU (Figure 14: line 7).

The fine-grained control of PThreads supports local computation as we saw in Section 3.2.1 (Figure 11: line 13) or the dispatch of that computation to a worker. This subtle coupling of computation and communication can allow for thread reuse, otherwise a new worker is created for the current computation (Figure 11: lines 19-25).

Though the OpenMP implementation lacks the fine-grained explicit control of PThreads, the OpenMP compiler can directly determine the number of threads created and can assign computation based on the number of iterations of the loop.
Computation and communication do overlap (implicitly) at the pragma from developer’s perspective. The determination of the amount of data to pass to a worker for processing is dependent on the number of workers available.

While the code shown in Figure 13 is dedicated to run on the SPUs, it is not pure computation. The code segment demonstrates how each SPU must get/put the current context (Figure 13: lines 3 and 14) through communication via shared memory with the PPU. The code then further communicates through a switch statement the correct computation to be performed based on the context (Figure 13: lines 5-12).

The actual context assignment, placement of data and the initiation of SPU execution occurs at the PPU and is difficult to trace through a manual inspection. A portion of this coordination is shown in Figure 14. The approach taken in this FFT implementation is similar to that of OpenMP and PThreads using a loop that iterates through each SPU, setting its operation context and assigning it a portion of memory holding the data it will operate on. An example of the setup coupling communication and computation is shown in context acquisition (Figure 14: lines 4, 5). In particular, the communication of data is tied to the computation in the Cell architecture in that a given SPU can only perform computation on a contiguous data block (Figure 14: line 11).

### 3.3 A Simple Mapping

Table 3 outlines some of the linguistic support and naming conventions used within the three implementations surveyed in Section 3.1 in terms of computation and communication. Though these are simple notions at first blush, their relationship is complex, as some lines of code serve a dual purpose accomplishing both.

<table>
<thead>
<tr>
<th>Parallel Mechanism</th>
<th>Computation</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA</td>
<td><em>global</em></td>
<td>cuda_malloc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cudaMemcpy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cudaThreadSynchronize</td>
</tr>
<tr>
<td></td>
<td></td>
<td>syncthreads</td>
</tr>
<tr>
<td>OpenCL</td>
<td><em>kernel</em></td>
<td>barrier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>malloc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>commandqueue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clReleaseMemObject</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clEnqueueWriteBuffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clEnqueueReadBuffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clCreateBuffer</td>
</tr>
<tr>
<td>MapReduce</td>
<td>task_data</td>
<td>use_1_q_per_task</td>
</tr>
<tr>
<td></td>
<td>data_size</td>
<td>l1_cache_size</td>
</tr>
<tr>
<td></td>
<td>key_match_factor</td>
<td>data_size</td>
</tr>
<tr>
<td></td>
<td>splitter, partition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>num_procs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unit_size</td>
<td></td>
</tr>
<tr>
<td></td>
<td>num_map_threads</td>
<td></td>
</tr>
<tr>
<td></td>
<td>num_merge_threads</td>
<td></td>
</tr>
</tbody>
</table>

CUDA provides library support mechanisms for memory allocation and thread synchronization to support communication, but in terms of computation, identifiers are limited to the `global` tag to mark the kernel module. This kernel module concept also exists in OpenCL and is identified with a `kernel` tag.

In addition to this tag, OpenCL provides extensive linguistic support for both computation and communication issues—making many steps actually more explicit. For example, device and task management functions are provided, as well as those associated with memory allocation, buffer creation and buffer read/writes through the OpenCL library.

MapReduce provides library support for partitioning of data, and semantic support in the form of naming conventions for scheduling arguments.

![Figure 15: Computation and communication breakdown in three implementations of a simple reduction.](image)

![Figure 16: Computation and communication intersection in three implementations of a simple reduction.](image)
exceeds 100% of the code base. Figure 16 isolates this intersection between computation and communication, showing that the intersection of these two entities is in fact substantially larger than either in isolation for all three examples.

Table 4 shows some of the keywords used within the PThreads, OpenMP and Cell implementations that helped in the code categorization. Linguistic support is minimal in both PThreads and OpenMP, but for very different reasons. PThreads yields a high level of control in terms of how computation is split up and performed, and generic support for communication in the form of synchronization constructs. OpenMP handles all of these resource decisions and synchronization in the background, concealing the need for linguistic support. The Cell implementation leverages naming conventions in key identifiers that assist in the identification of computation code that would run on the SPUs. Function naming conventions and built in library mechanisms for memory allocation, reading data and coordinating the computation, helped to more identify code associated with communication.

<table>
<thead>
<tr>
<th>Parallel Mechanism</th>
<th>Computation</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>PThreads</td>
<td>implicit</td>
<td><em>sem</em> mutex</td>
</tr>
<tr>
<td></td>
<td></td>
<td>kill</td>
</tr>
<tr>
<td>OpenMP</td>
<td>implicit</td>
<td>implicit</td>
</tr>
<tr>
<td>Cell</td>
<td>cell_nspe</td>
<td>spu_read_in_mbox,</td>
</tr>
<tr>
<td></td>
<td>spudma1d</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>*.spuc files</td>
<td>cell_spe_awake_all</td>
</tr>
<tr>
<td></td>
<td>spu_main</td>
<td>cell_spe_wait_all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cell_aligned_malloc</td>
</tr>
</tbody>
</table>

Figures 17 and 18 show that, in the FFTW code base, this dominant intersection is again a feature in this implementation. We now plunge deeper into its possible ramifications.

### 4. DEFINING THE ONTOLOGY

The preliminary case studies described in Section 3 support the need for a more fine-grained classification, clearing up the contents of the overlap or intersection as much as possible. We want to retain the high-level conceptual model of computation and communication, as they were useful as a start. However, to become more precise, we believe a finer-granularity representation is best achieved through a hierarchical structure provided by an ontology. Further, we hope to derive an ontology that could serve as a useful platform for pattern designers—revealing relationships that could be shared between code bases. Here we propose a more fine-grained breakdown of computation and communication based on observations outlined in Section 3.

#### 4.1 Identifying Entities

In this first stage of deriving the ontology, we use a combination of related work and concrete evidence from the code bases to identify key entities involved.

##### 4.1.1 Computation

In terms of pure computation in a parallel application, from our observations we have identified two finer-grained key entities: task and sequential. The task encompasses the actual computation performed in parallel, or its direct invocation, versus the sequential portion, some might consider the limiting factor in Amdahl’s law.

This task entity aligns with the parallel programming model originally identified in embedded applications [19] and high-performance computing [18], in which one of the critical steps in parallelism is ‘the division of the application into parallel tasks’.

The sequential portion of the code in the implementation mechanisms considered here show a good deal of variation in the
degree to which they are implicit/explicit regarding resource provisioning in the setup for the actual parallelism.

4.1.2 Communication
In both case studies, the examples were purely based on data parallelism, with no inter-worker communication involved. Pure communication in these examples was primarily through some form of shared memory or shared state. From these observations we have identified two finer-grained key entities: data distribution and synchronization. Data can be distributed through shared memory or placement of the data within a worker’s local memory space, whereas synchronization mechanisms tend to communicate through shared state. This finer-grained breakdown of communication aligns with the following characteristics of parallel support mechanisms from [11]: ‘distribution of data to memory elements’ and ‘inter-task synchronization’.

4.1.3 Breaking Down the Intersection
While we can somewhat logically parcel out the entities of pure computation and communication, these four entities do not encompass everything that is going on in the code. Our case study identified an intersection that primarily contains code necessary for setup for computation and communication. From these observations of the code we identify two entities corresponding to this setup code as task coordination and data coordination. While the task and data distribution are very distinct entities, task coordination and data coordination are more tightly coupled. They both involve provisioning of resources:

- **task coordination** handles resource provisioning primarily for computation, but it must also be communicated to tasks, and
- **data coordination** handling resource provisioning primarily for communication, associated with computation.

Figure 19 illustrates this overlap and begins to develop our ontology for representing these necessary characteristics of parallel applications.

![Figure 19. Relationships between computation and communication entities.](image)

Table 5 provides a summary of how each of these six entities could map to a language independent implementation. **Sequential** would be algorithm dependant, but in our case studies tends to be associated with the setup phase. **Task** maps to the computation, whereas **task coordination** corresponds to resource and context management. **Data coordination** maps to memory allocation, partition sizes and buffer creation, whereas **data distribution** would be the actual data copying or assignment. Finally, **synchronization** would map to the application of any provided or derived synchronization primitives.

<table>
<thead>
<tr>
<th>Table 5: Mapping fine-grained entities to implementation.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entity</strong></td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Sequential</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Task</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Task Coordination</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Data Coordination</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Synchronization</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Data Distribution</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

4.2 Revealing Relationships
These results begin to identify the magnitude of coordination necessary in something as simple as the setup stage in a parallel application. Yet, looking at each task on its own—coordination often does not appear to be explicit. Similarly, resource provisioning is only implicitly associated with coordination.

In fact, given the finer-granularity of entities provided in Table 5, we can now re-categorize task coordination, data coordination, and synchronization explicitly to be associated with a new higher-level entity: coordination. Here we argue that, though task coordination can be cleanly reassigned to this higher-level, both synchronization and data distribution still contribute to communication as well, resulting in the hierarchy in Figure 20.

This extended ontology of varying granularity supports not only the comprehension of fine-grained entities in isolation, but also in a holistic view of entity relationships. The key characteristic of coordination entities is that they identify relationships between other entities. These tiers in the ontology provide perspectives that can work in synergy to support the linking of low-level implementation details to high-level abstractions and patterns.

![Figure 20: Extended ontology.](image)

5. Coordination in Autocatakinetic Systems
Our previous analysis of the dishwashing activity and the grade 7’s solutions to this problem was through the simplified task-centric perspective of computation and communication (Section 2). The proposed ontology derived from the code bases makes coordination a first-class, top-level entity. Here we consider its validity in the context of autocatakinetic scenarios.
Though explicitly, there was very little coordination that the students consciously exhibited, the overall system contained combinations of computation/communication based elements that resulted implicitly in coordination, often through multi-sensory means, and resulting in sophisticated feedback loops. So, much like in the derivation of the ontology from the code bases, we can see that coordination emerges as the agreed upon protocol in dishwashing.

An open question we are exploring is the potential to design low level patterns of coordination based on a representation of Petri nets, in terms of states, transitions, firing and tokens [25]. Though there is no abstraction that directly corresponds to different tasks within a Petri net, nodes are states that are organized with a simple representation of multiple control flows (tokens that progress) and transitions. For a transition to take place, all of the token requirements for the vertices providing input into the transition must be met, capturing coordination.

Figures 21 and 22 represent the two most popular solutions (b and c) the students chose for the dishwashing scenario as Petri nets. Though these nets cleanly demonstrate the coordination between very fine grained parts of a computation, there is no corresponding visual support as to which participant is performing which step of the task. As the abstraction of tasks is implicit, so is the need for communication—ironically, all that is left is a high level perspective of coordination at a granularity that is finer than that of task.

![Figure 21: Petri net of solution B](image1)

![Figure 22: Petri net of solution C](image2)
Petri nets have been expanded in a few directions, for example, coloured Petri nets highlight data flow [26] and Object Oriented Petri Nets (OOPNets) are designed to demonstrate the communication and coordination of objects within a system [27]. But to the best of our knowledge, Petri nets have not yet been extensively leveraged by the design pattern community, though they appear to hold promise in the context of low level patterns for coordination.

We believe the dishwashing scenario serves to validate the emergence of coordination in the ontology. Though it seems to be a simple problem, many complexities arise associated with coordination—buffering, mutual exclusion, and timing—are all nested within the space.

6. FULL ONTOLOGY MAPPING

In this section we refer back to the reduction and FFT case studies to consider the applicability of this finer-grained mapping to an actual code base. Table 6 highlights the ways in which keywords associated with implementation map to the ontology overviewed in Tables 3 and 4 in terms of computation and communication. This table allows us to consider this same data from the perspective of the fine-grained entities and how they are represented within each parallel implementation.

The sequential portion of the code is implicit across the board. Similarly, task and data distribution are highly algorithm and implementation dependant and therefore the support provided by most parallel mechanisms is implicit. Though a thorough comparison of mechanisms it out of the scope of this paper, it is interesting to note that newer mechanisms in OpenCL are moving towards a more explicit identification of core computation and support for reading/writing to memory. The coordination of the task and data is also better supported through library functions. For example, OpenCL’s support for device setup and queue management far exceeds that provided by CUDA. MapReduce also provides task and data related scheduling parameters for tuning, but the support to make full use of these parameters is not intuitive in our experience.

Of the parallel mechanisms that require synchronization to be dealt with by the developer, they each provide some form of linguistic support in the form of locking and timing mechanisms.

Leveraging the mapping discussed above, coupled with user interactive assistance during analysis, we suggest that it may be possible to semi-automate mapping of the fine-grained ontology proposed to the six implementations discussed in Section 3. Subsections 6.1 and 6.2 outline the results of these fine grain mappings to the simple reduction and FFT implementations respectively.

6.1 Simple Reduction

Figure 23 shows the results of mapping the full ontology to the MapReduce, CUDA and OpenCL implementations of a reduction algorithm. The simplicity of this algorithm is highlighted by the small percentage of code necessary for the task implementation. The CUDA results show a slightly higher percentage of task code than MapReduce and OpenCL, but this is due to the fact that the CUDA reduction provides six alternative kernels with optimization variations.

If we consider these results from the perspective of the top tier entities of computation, coordination and communication, we see that for every code base, coordination is the largest portion of the code. This lines-of-code count is a preliminary indicator that coordination may be the most labour intensive component of code to write. It becomes equally apparent that frameworks like MapReduce and OpenMP work to minimize the amount of work associated with this coordination. For example, in these results MapReduce is the only mechanism in which task coordination is smaller than data coordination and the synchronization is completely implicit.

<table>
<thead>
<tr>
<th>Parallel Mechanism</th>
<th>Sequential</th>
<th>Task Coordination</th>
<th>Synchronization</th>
<th>Data Coordination</th>
<th>Data Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>PThreads</td>
<td>implicit</td>
<td>implicit</td>
<td>sem* mutex kill</td>
<td>malloc</td>
<td>implicit</td>
</tr>
<tr>
<td>OpenMP</td>
<td>implicit</td>
<td>implicit</td>
<td>implicit</td>
<td>implicit</td>
<td>implicit</td>
</tr>
<tr>
<td>Cell</td>
<td>implicit</td>
<td>*.spuc files</td>
<td>spu_read_in_mbox</td>
<td>cell_aligned_malloc</td>
<td>implicit</td>
</tr>
<tr>
<td>CUDA</td>
<td>implicit</td>
<td><strong>global</strong></td>
<td>cudaThreadSynchronize</td>
<td>cuda_malloc</td>
<td>cudaMemcpy</td>
</tr>
<tr>
<td>OpenCL</td>
<td>implicit</td>
<td><strong>kernel</strong></td>
<td>barrier</td>
<td>malloc</td>
<td>clEnqueueWriteBuffer, clEnqueueReadBuffer,</td>
</tr>
<tr>
<td>MapReduce</td>
<td>implicit</td>
<td>splitter</td>
<td>task_data</td>
<td>clReleaseMemObject, clEnqueueWriteBuffer, clEnqueueReadBuffer,</td>
<td></td>
</tr>
</tbody>
</table>

Table 6. Mapping mechanism to entities.
Synchronization is the smallest piece of coordination code in all three implementations. This demonstrates that the number of lines of code is not the best measure of comprehensive burden, as the difficulty with synchronization is well known. We believe these results highlight however, that this complexity lies in its relationship to data and task coordination.

6.2 FFT

Figure 24 provides the full ontology mapping results for the three FFT implementations using OpenMP, PThreads and Cell. Similarly to the simple reduction code examples, data distribution and task entities are small in terms of code percentages. This result can be attributed to the limited scope of the code actually being analyzed. Also similar to the results from the reduction case study, coordination accounts for the largest portion of the code.

OpenMP completely conceals all synchronization and data distribution. This is in contrast to the result we see with the Cell and PThreads implementation, which introduce more than double the amount of code than any other mechanism across both case studies. In addition, the Cell implementation requires more explicit data coordination than that of PThreads in order to ensure the data for each worker resides in a contiguous blocks of memory.

Figure 24: Full mapping onto 3 FFT implementations.

7. DISCUSSION AND CONCLUSIONS

This study is based on a line of code analysis of mechanism specific implementations of two common algorithms, but this metric is by no means an indicator of the true cognitive burden these lines introduce. Coordination arguably poses disproportional difficulty in terms of program comprehension, at least in part due to its lack of modularity, but also due to the fact that it subtly emerges from a complex relationship between computation and communication.

Currently we face the precarious situation where parallelism is challenging because developers lack a means of exploring possible internal dynamics and causal relationships that tend to be problematic in these code bases. Our study shows that many important relationships are actually concealed or implicit. Abstractions and mechanisms that hide these relationships as opposed to accentuating them maybe in fact do more harm than good in future code bases. Mechanisms and abstractions for parallelism are rapidly evolving, and though attempting to conceal details is important for comprehension—concealing the wrong details could further obscure the ability to reveal critical patterns that would enhance and support knowledge sharing.
We believe that the low-level application of an ontology mapping to a code base can be used to help strengthen conceptual and methodological approaches developers must apply to any parallelized source. Through refinement, our proposed ontology was able to represent relationships that can be viewed in a more coherent and conceptually concise manner.

We further propose a general framework for the automated representation of the static and dynamic properties of parallel applications [3]. As an extension of this work, we envision semi-automated tool support to aid developers in the navigation of ‘Computation, Communication and Coordination for Parallel Objectives’ (C3PO). This navigation of parallel applications from an ontological perspective will help to clarify subtle relationships in a static representation, and would be amenable to views that include patterns and relationships between patterns. As a proof of concept, Figure 25 illustrates a simple code coloring based on the six fine-grained entities of our ontology. Though this was manually derived, tool support could allow levels of zoom between top and second tier ontology entities.

With the growing number of parallel support mechanisms, benefits also lie in the ability to compare across multiple implementations as we did in this study. As described above, a design pattern related to an implementation specific micropattern could be leveraged to create a similar implementation using another parallel mechanism to target a different architecture. Further, the relationship between multiple design patterns that comprise an implementation could be viewed at the granularity of entities.

Reasoning at the level of a single dwarf allows one to link dynamic communication and computation patterns to the static code view. This view can both assist in application tuning as well as the identification of dwarf compositions. For example, this case study revealed the potential composition of a Monte Carlo dwarf within the FFT spectral method. Understanding dwarf compositions can be further enhanced by comparing the fine-grained static representation of ontological entities between dwarfs.

Our goal is to put this ontology to work in the hands of pattern designers. Here we have explored its application across six different parallelization support mechanisms in the context of two representative applications. Our results show the challenges of categorizing lines of code and the utility of these entities and their relationships in program comprehension independent of language specific representations. We believe that the study of additional code bases leveraging alternative mechanisms guided by this ontology will reveal critical relationships common within these code bases, and support the better representation and exposure of patterns to aid code comprehension.

8. REFERENCES


http://www.nvidia.com/object/personal
supercomputing.html, 2008.


?name=Content&pa=showpage&pid=445&page=3.


