Capsules: Expressing Composable Computations in a Parallel Programming Model

Hasnain A. Mandviwala[1], Umakishore Ramachandran[1], and Kathleen Knobe[2]

[1] College of Computing, Georgia Institute of Technology (mandvi|rama@cc.gatech.edu)
[2] Intel Corporation Inc. (kath.knobe@intel.com)

Abstract. A well-known problem in designing high-level parallel programming models and languages is the "granularity problem", where the execution of parallel task instances that are too fine-grain incur large overheads in the parallel run-time and decrease the speed-up achieved by parallel execution. On the other hand, tasks that are too coarse-grain create load-imbalance and do not adequately utilize the parallel machine. In this work we attempt to address this issue with a concept of expressing "composable computations" in a parallel programming model called "Capsules". Such composability allows adjustment of execution granularity at run-time.

In Capsules, we provide a unifying framework that allows composition and adjustment of granularity for both data and computation over iteration space and computation space. We show that this concept not only allows the user to express the decision on granularity of execution, but also the decision on the granularity of garbage collection, and other features that may be supported by the programming model.

We argue that this adaptability of execution granularity leads to efficient parallel execution by matching the available application concurrency to the available hardware concurrency, thereby reducing parallelization overhead. By matching, we refer to creating coarse-grain Computation Capsules, that encompass multiple instances of fine-grain computation instances. In effect, creating coarse-grain computations reduces overhead by simply reducing the number of parallel computations. This leads to: (1) Reduced synchronization cost such as for blocked searches in shared data-structures; (2) Reduced distribution and scheduling cost for parallel computation instances; and (3) Reduced book-keeping cost maintain data-structures such as for unfulfilled data requests.

Capsules builds on our prior work, TStreams, a data-flow oriented parallel programming framework. Our results on an SMP machine using the Cascade Face Detector, and the Stereo Vision Depth applications show that adjusting execution granularity through profiling helps determine optimal coarse-grain serial execution granularity, reduces parallelization overhead and yields maximum application performance.

1 Introduction

Parallel programming is difficult [18]. Even more daunting is the task of writing a parallel program that executes efficiently on varying amounts of available concurrency *without* source code modification. Different platforms provide a different level of hardware parallelism, for example, the Cell B.E. processor has 1 Power Processing Element

(PPE) and 8 Synergistic Processing Elements (SPEs), whereas the Intel Core2 Duo processor has upto four general purpose cores. In all these examples, the available hardware parallelism varies depending on the platform. Even on a given platform, depending on the workload mix, the parallelism available for a given application may change over time. Clearly, an application programmer would like to exploit all available hardware parallelism without having to re-compile code (on the same platform). The traditional solution was to extract all potential application parallelism and map it evenly among the available processors. However, if the granularity of parallel tasks is too fine, and the available hardware concurrency does not match the application concurrency, the application would incur excessive run-time overhead in executing the fine-grain computations on the limited available hardware. Ideally, one would like to shield the application programmer from the vagaries of resource availability while maximizing performance. Therefore, there is a need to dynamically adapt the application granularity (without change in source code) to match the available hardware parallelism and thus reduce the *parallelization overhead*.

Current parallel programming models lack the semantic ability to express a granularity adaptation mechanism for parallel tasks, where the granularity of execution could be changed for greater execution efficiency. Previous high-level parallel programming models such as Jade [9, 16, 17], Cilk [2], OpenMP [3, 6, 8] and even surveys [1] on parallel programming trends have acknowledged the problem of high run-time overhead when executing fine-grain computations. However, the granularity problem is not addressed at the programming model level and the programmer is left to encode parallel tasks to have sufficient granularity to avoid high parallelization overheads.

In this paper, we introduce the Capsules parallel programming model, which exposes the notion of composable computations and in-turn allows the dynamic adjustment of execution granularity for concurrent tasks. The application is written with the granularity that makes sense from the point of view of the application. Capsules provides software abstractions that allow dynamic composition of fine-grained computations into coarser grain modules when there are insufficient hardware resources. Such a dynamic composition results in a two-fold advantage: (1) The run-time needs to manage fewer parallel tasks thus reducing the book-keeping, scheduling, and distribution overheads. (2) The synchronization costs for shared data access is reduced by amortizing these overhead costs within the more useful work done for the composed coarse-grain computations.

We show that the Capsules model is a unifying framework that allows the application programmer to not only make decisions on adjusting the granularity of execution, but also allows him/her to adjust the granularity of other features. Features such as garbage collection (GC) of items can be made to occur at different granularities depending on how aggressive the programmer would like it to be. Similarly, features such as check-pointing and debugging can also occur at different granularities.

To evaluate the Capsules programming model and its run-time, we have parallelized two applications, namely: (1) The Cascade Face Detector (FD) [19] and (2) the Stereo Vision Depth (SV) [20] algorithm. Our results show that increasing the execution granularity helps reduce run-time overhead, and simultaneously yield an increase in performance.

2 Reducing Run-time overhead

In most parallel programming models, overhead is caused during *synchronization points*, which represents access to shared data either through explicit *put/get* [11, 14, 15, 13, 7] calls or implicitly through data access mechanisms such as *closures* and *continuations* [2] or *access declarations* [16]. There may also be book-keeping costs incurred to track the data requirement of computations/tasks running concurrently. Scheduling and distribution of tasks also contribute to this overhead. Therefore, the total overhead cost in such parallel systems is directly proportional to the number of concurrent tasks that execute and the number of synchronization points reached by those concurrent tasks during the entire application execution.

Therefore, in the absence of sufficient hardware concurrency, it is important to reduce this cost of parallelization. This can be achieved partially by reducing the total number of parallel tasks the run-time system needs to manage during the execution of a parallel program. Reducing the total number of tasks means increasing the amount of computation each parallel task needs to achieve. We refer to this as increasing the granularity of parallel tasks. Decreasing the number of parallel tasks can also decrease the number of synchronization points required to access shared data, which is also composed to a coarser granularity. Synchronization points are reduced by moving the shared data accesses to the boundary of coarser-grain composed computations.

Our approach towards reducing the number of concurrent tasks is to create coarser-grain tasks from finer-grain tasks dynamically during the parallel execution. The finer-grain computations inside the coarse-grain computation then execute serially. We introduce the notion of composable computations to the programming model level that enables instances of fine-grain computations to be merged together to form coarse-grain computations and at the same time reduce overall parallelization overhead.

3 Composing Computations Dynamically

In our work, we build upon the TStreams [7] parallel programming model to incorporate the notion of Composable Computations to enable adjustable granularity. We call our new parallel programming model *Capsules*. A user of Capsules can express maximum potential application parallelism by defining an application task-graph using finest-grain computational pieces and finest-grain data abstractions. Then, fine-grain computations can be dynamically composed together by the user to form more efficient coarse-grain computations. The mechanisms for composability are divided into two sub-mechanism that work orthogonal to each other. They are: (1) *Composition by Iteration Space*, and (2) *Composition by Computation Space*. Each mechanism is dynamic, and allows run-time determination of granularity that can affect application performance.

3.1 Software Abstractions: Step, Item, and Tag Capsules

In this section we describe software abstractions that allow expressing composable computation within a parallel programming model. These abstractions are (1) *StepCapsules*, (2) *ItemCapsules* and (3) *TagCapsules*. These abstractions are similar to the primary objects in TStreams [7], and differ only in the extra information they encapsulate to allow composability. Each Capsule object either contains only *one* object instance, or a *collection* of object instances representing a coarser-granularity. The granularity of these Capsules is user-defined, and can be dynamically adjusted at run-time.

To make a distinction between static and dynamic information about capsule objects, each object abstraction is separated into *Spaces* [7] and *Instances*. The notion of Capsule Spaces is analogous to the notion of *Classes* in Object Oriented Programming (OOP), which refers to the static specification of the object. Capsule Instances, therefore, are dynamic incarnations that conform to the specification of a Capsule Space (or object class). These distinctions provide Capsules with clean object oriented semantics, making it an easy development model for parallel programming.

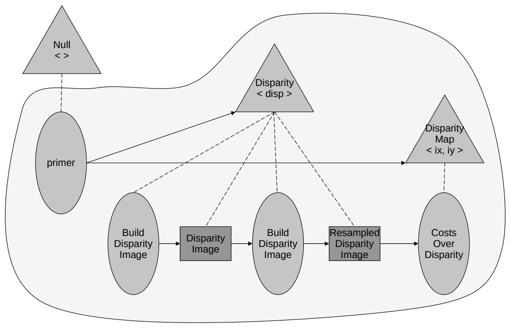

Fig. 1. Stereo Vision Depth application graph in Capsules

Figure 1 illustrates an application constructed using Capsules. The task-graph denotes Capsule Spaces and relationship edges that remain static during program execution. The triangular shapes represent TagCapsule Spaces that denote iteration spaces for computation and data. The relationship between the iteration spaces and the computation and/or data is denoted by the dotted line. The oval shapes here represent computations or StepCapsule Spaces. Finally, the rectangular shapes represent data or ItemCapsule Spaces. These store data objects communicated between computations during program execution.

Listed below are the three basic objects found in TStreams that we use and extend from in Capsules:

Tag Instances are unique identifiers for a given Step or an Item instance (similar to Tuples in Linda [5]). Tags are multi-dimensional, where each dimension represents an iteration dimension specifying a range of possible values. These dimensions can be of any arbitrary type but only integer Tag dimensions are supported in the current implementation. A Collection of Tag Instances are called TagCapsule Instances.

Step Instances are function calls to the finest-grain user-defined indivisible computations. Each step instance is uniquely identified by a parametrizing Tag instance. Step instances produce Item instances or Tag instances via the producer relation. They also produce ItemCapsule instances and TagCapsule instances. Step instances also consume

Item instances and ItemCapsule instances via the consumer relationship. A collection of finer-grain Step Instances is called a StepCapsule Instance.

Item Instances are fine-grain data produced by other computation Step instances. Each item instance is uniquely identified by a Tag instance. A collection of Item Instances is called an ItemCapsule Instance.

Now we list objects specifically added to Capsules to allow for composability:

TagCapsules Instances are *tree* structures that store multiple Tag instances in a compressed form. This is the abstraction that enables composition over iteration space. The depth i of the tree represents the dimension i of a Tag instance. Each tree node consist of a Tag dimension value. Enumeration of Tags is achieved by the cross-product of a Tag dimension value at depth i with the child Tag dimension values at depth $i+1$. Since trees have a hierarchical structure with fewer root nodes than child nodes, this structure also specifies the hierarchical compression of the Tags dimension values at different dimensions. Tag dimension values that are higher in the tree are compressed more (have fewer nodes representing them) than Tag dimension values lower in the tree.

StepCapsule Instances are coarse-grain computations that are composed from other coarse-grain Step, Item and Tag Capsules enabling composition over computation space. StepCapsules play a dual role in the composable computation paradigm. They not only represent coarse-grain computations, but also represent the GC boundary for an automatic constrained GC mechanism (described in detail in sec. 4.3). StepCapsule instances are also hierarchical tree data-structures, where each non-leaf node represents a coarse-grain computation and a leaf-nodes represents fine-grain Step instances.

ItemCapsule Instance is also a collection of Items forming a coarse-grain data Capsule. It is also a tree structure similar to the TagCapsule instance tree. Each node at depth i of the ItemCapsule instance tree represents the Tag dimension value of the parametrizing TagCapsule instance tree at the same depth i. At the leaf-nodes of the tree, the actual items are stored. The items stored in a leaf-node are parametrized by the Tags represented by the parent hierarchy of the leaf-node.

Finally, we enumerate the Capsule primitives that specify the static relationships in the application task-graph. These Spaces, encapsulate the common denominator properties of Capsule object instances that belong to the same space.

TagCapsule Spaces, contain the static dimension information, namely, the number of dimensions in the iteration space and the name of each dimension. TagCapsule Spaces also store information about the objects they *parametrize*. Parametrization is a relationship between TagCapsule Spaces and other ItemCapsule Spaces or StepCapsule Spaces that specify which objects the TagCapsule instances uniquely identify.

StepCapsule Spaces contain static information about its parent StepCapsule Space, its parametrizing TagCapsule Space and child that are contained within it. They also contain producer/consumer relationship information between itself and other ItemCapsule and TagCapsule Spaces.

ItemCapsule Spaces also contain static information about its parametrizing TagCapsule Space and its parent StepCapsule Space.

3.2 Reducing Synchronization Points

In Capsules, synchronization points or data-access points to shared data structures can be reduced by creating coarser-grain data objects and coarser-grain computations. The synchronization points accessing coarse-grain data are moved to the *border* of the coarse-grain computations. Each coarse-grain computation requires a *serialization schedule* that defines the execution order of its constituent fine-grain computation. For StepCapsules created by composing over iteration space, the serialization schedule is determined by inspecting the StepCapsule instance's parametrizing TagCapsule instance. For StepCapsules created by composing over computation space, the serialization schedule requires analysis of data-dependencies between the component computations.

Moving synchronization or data-access points to the border of the serialization schedule refers to the transformation required to the data-access pattern and the granularity of input ItemCapsules, such that the total number of synchronization points in the application execution are reduced. When a StepCapsule instance is composed over iteration space, moving synchronization points to the boundary of the coarse-grain StepCapsule depends on the relationship of the dimensions between the producer/consumer StepCapsule Space and its ItemCapsule Space. However, for a StepCapsule instance composed over computation space, moving synchronization points requires analysis of the producer/consumer edge information between the composed coarse-grain StepCapsule Space and its ItemCapsule Spaces.

4 Composing by Computation Space

Composition over Computation Space is based on the notion of combining distinct computations or distinct pieces of code to create coarse-grain computations. Furthermore, these composed computations allow further composability by combining with other computation pieces like an erector set. This is a concept derived from functional and procedural languages where a coarse-grain function can be composed from fine-grain functions.

4.1 Serialization Order when Composing over Computation Space

To execute coarse-grain StepCapsules created by composition over computation space, a serial execution schedule is required to define the execution order of the fine-grain computations contained within it. We call this the serialization order for the coarse-grain StepCapsule. In Capsules, programmers are only required to provide data-dependencies between computation with the help of producer and consumer edges. Therefore, in order to construct a non-blocking serial execution schedule, a-priori resolution of data-dependencies via edge analysis is required. We have not yet implemented the automatic generation of a serialization schedule and leave it for future work.

4.2 Moving Synchronization Points to Coarse-Grain Computation Boundary

When composing over computation space, data access information for the composed coarse-grain StepCapsule Space needs to be distilled from the application graph. Explicit *get()* calls to retrieve data from external ItemCapsule Spaces are only required at the beginning of the coarse-grain StepCapsule. Likewise, *put()* calls to produce data and instantiate further StepCapsule instances are only needed at the end of the composed coarse-grain computation. This transformation has also not been implemented and is part of our future work.

4.3 StepCapsule Space: A software abstraction enabling Composition over Computation Space

A StepCapsule Space is constructed by combining together Step/Item/Tag Capsule Spaces. We refer to these finer-grain Spaces composed to form a coarse-grain StepCapsule Space as *Inner Spaces*. A composed StepCapsule Space also contains information about the producer/consumer relationships between the inner spaces.

A Hierarchy of Composable Computations As fine-grain StepCapsule Spaces are composed into coarse-grain StepCapsule Spaces, a hierarchical StepCapsule tree is formed. Each *finest-grain* Step, Item and Tag Capsule Space in the application task-graph occurs *exactly once* as a *leaf node* of this tree. Each intermediate node of the tree represents the coarse-grain *composed* StepCapsule Spaces. For a given application, a StepCapsule Space hierarchy tree can be constructed in multiple ways using an API. Figures 1 and 2 illustrate two applications in their composed hierarchical form.

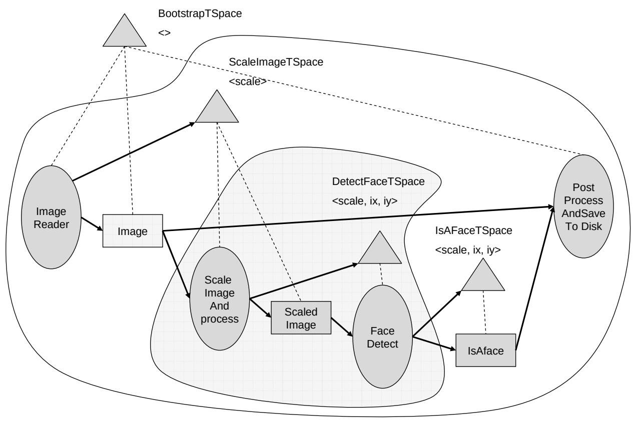

Fig. 2. Cascade Face Detector application, with a hierarchy of StepCapsule Spaces composed over computation space. The *ScaleImageAndProcess* and *FaceDetect* StepCapsule Spaces, along with the *ScaleImage* ItemCapsule Space and the *DetectFaceTSpace* TagCapsule Space are composed into one *coarse-grain* StepCapsule Space. This StepCapsule Space and the remainder objects form the outermost StepCapsule Space.

Selecting a Computation Space Hierarchy Constructing the right computation space hierarchy is dependent on how the application needs to be partitioned along its data and computation boundaries to extract parallelism. Selecting the best hierarchy is dependent on complex variables such as available resources (memory and hardware concurrency), hardware platform characteristics such as shared memory or distributed memory, and exploitable application parallelism. Furthermore, in future hardware platforms, which will likely have hierarchical memory structures, the ability to hierarchically express computations for locality would be significant for performance. For our current system, we leave the determination of composable computation hierarchy to the application developer.

It is important to distinguish that the computation space hierarchy is statically defined by the user using the composition API. This hierarchy *cannot* change once the application begins execution. However, the decision to use the composed StepCapsule Space for coarse-grain serial execution is made dynamically at run time for each StepCapsule instance. Different StepCapsule instances can therefore be made to execute either serially in their coarse-grain form or simply allow the inner fine-grain StepCapsules instances to execute in parallel.

StepCapsule as a Boundary for GC and Scope The coarse-grain StepCapsule instance also acts as a data-structure to maintain a GC boundary for all the ItemCapsules contained within it. Once all inner StepCapsule instances are done executing, the coarse-grain parent StepCapsule instance is marked *executed*, allowing all inner ItemCapsule instances to be GC'ed.

From a scoping perspective, inner ItemCapsule instances are visible only to inner StepCapsule instances contained within the same coarse-grain StepCapsule instance. However, inner StepCapsule spaces have visibility to Item/Tag Capsule spaces defined anywhere in their parent StepCapsule Space hierarchy.

4.4 Rules for Constructing a StepCapsule

The rules for composition over computation space are summarized as restrictions that guarantee the composed coarse-grain StepCapsule Space to be (1) atomic in execution, (2) to be uniquely tagged, (3) to terminate and (4) to not contain any non-reachable computations. These rules rules keep the execution model simple and avoid dead-locks during parallel execution.

5 Composition over Iteration Space

In this section we describe the notion of composing over iteration space within the context of parallel programming models.

Iteration space simply refers to all possible values that Tag instances can have. For example, a TagCapsule Space $< int\ x, int\ y >$ can span the entire space of two dimensional positive integer values. Therefore, the notion of composition over iteration space is defined by a collection of Tag instances put together over one or more dimensions of a TagCapsule Space. Since Tag instances actually parametrize computations (Steps) and data (Items), composition over iteration space is said to be a generalized form of the concept of composing together multiple instances of the same computation or composing together multiple instances of the same data-type.

5.1 Serialization Order when Composing over Iteration Space

Similar to coarse-grain computations created by composition over computation space, coarse-grain computations created by composition over iteration space also require a serialization schedule for execution. The serial execution schedule of a StepCapsule is based on the structure of its parametrizing TagCapsule instance tree. The schedule simply traverses the sparse TagCapsule tree *in-order*, and appends the Tag dimension value found at depth i as the value for Tag dimension i. At every leaf node at depth N, the fine-grain StepCapsule instance is executed with the specified Tag Instance.

Clearly, the serial execution order is dependent on the order of Tags in the TagCapsule instance tree. As Tag dimension values at any level of a TagCapsule tree are created by a user-defined fine-grain StepCapsule function, the serialization order is therefore created by these producers by virtue of their serial creation order.

5.2 Moving Synchronization Points to Coarse-Grain Computation Boundary

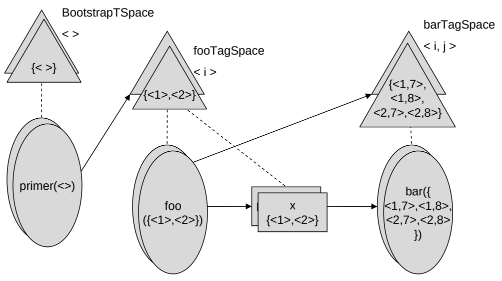

Fig. 3. An example of Composition by Iteration Space

Reducing the number of synchronization points means reducing the number of accesses to ItemCapsule Spaces. This requires the ItemCapsule instances to be coarse-grain so as to satisfy the data-requirements of the coarse-grain StepCapsule instance.

Retrieving any ItemCapsule instance in the Capsules programming model requires the run-time to define its parametrizing TagCapsule instance. This TagCapsule instance is derived from the parametrizing TagCapsule instance of the executing StepCapsule instance. Specifically, the matching dimensions between the parametrizing TagCapsule spaces of both the ItemCapsule space and the StepCapsule space determine what sub-tree in the StepCapsule instance's TagCapsule instance will be used to retrieve the ItemCapsule instance. The rules for valid producer and consumer relationships between StepCapsules and ItemCapsules are summarized in the next section. The ability to express coarse-grain ItemCapsules and performing data-access on them reduces the total number of synchronization points during program execution.

The following code example is used to explain the transformation that reduces synchronization points by moving them to the coarse-grain computation boundary:

```
foo() parametrized by TagCapsule Space <i, j>: foo<i, j>
foo<i, j> requires a[i] and b[j] as input
foo<i, j> produces c[i][j] as output

    Ai = getItem(A, <i>);
    Bj = getItem(B, <j>);
    Cij = foo(Ai, Bj);
```

```
  putItem(Cij, <i, j>);
```

Here, if *foo()* is executed in its finest granularity, there are $(I * J)$ instances of *foo()*, each performing *one* synchronized *get()* on each Item Spaces *A* and *B* and *one* synchronized *put()* on Item Space *C* to satisfy its input/output data requirements. Therefore, overall $2(I * J)$ *get()* calls and $(I * J)$ *put()* calls are performed. However, if *foo()* is composed along a dimension of its parametrizing iteration space i, there should then only be J coarse-grain instances of *foo()*. Each coarse-grain instance of foo$< j >$ performs only one synchronized *get()* on *A[0:I]* and one on *B[j]*. Overall, there are only $(2 * J)$ synchronized *get()* calls. Similarly, each coarse-grain instance of foo$< j >$ performs one *put()* of *C[0:I,j]*. Below is a code representation of the transformation:

```
Aall = getItem(A, <0:I>);
Bj   = getItem(B, <j>);
for(int i = 0; i <= I; i++) {
  Cj[i] = foo(Aall[i], Bj);
}
putItem(Cj[0:I], <0:I, j>);
```

This transformation reduces the number of synchronization points required for foo$< i, j >$ by creating coarse-grain serial executions of foo$< j >$.

5.3 TagCapsule Space: A software abstraction enabling composition over Iteration Space

Similar to Tag Spaces in TStreams, TagCapsule Spaces in Capsules *parametrize* Item/-Tag Capsules Spaces. Although more general, TagCapsules enable a behavior similar to that of tiling [4].

Since a TagCapsule instance represents a collection of Tag instances, when a TagCapsule instance parametrizes a StepCapsule instance, each inner Tag instance inherently parametrizes a Step instance to form a collection of parametrized Step instances. However, from the stand point of the Capsules parallel programming model, all step instances parametrized by the TagCapsule are denoted as *one* coarse-grain StepCapsule instance that executes *atomically* and *serially* over the finer-grain Step instances.

TagCapsule instances denote the same granularity for ItemCapsule instances as they do for the StepCapsule instances they parametrize. This implies that for a given TagCapsule instance only *one* coarse-grain ItemCapsule instance would exist, where the inner fine-grain item instances would have a one to one mapping with the inner fine-grain Tag instances in the TagCapsule. For example, in fig. 3, the *fooAndItemTagSpace* parametrizes both the *foo()* StepCapsule space and the ItemCapsule Space. Therefore, the granularity of the TagCapsule instances in *fooAndItemTagSpace* denotes the granularity of ItemCapsule instances in ItemCapsule Space.

5.4 Rules for Composition over Iteration Space

In this section, we discuss rules that define composition over iteration space. In general, these rules provide restrictions on certain application graphs that make composability either expensive or impossible. We eliminate this class of application graphs to maintain a balance between a simple and efficient parallel run-time and a parallel programming model that is general enough to sufficiently address the composability requirements for the class of high performance vision applications targeted in this work.

To elaborate further, these rules describe restrictions on the *dimensions* of object spaces with producer/consumer relationships between them. In order words, these rules define what producer/consumer relationships/edges are allowed between StepCapsule Spaces and other Tag/Item Capsule Spaces.

For simplification, let us call the dimensions of the parametrizing TagCapsule Space of an Item/Step Capsule Space as the dimensions of that Item/Step Capsule Space.

The rules of composition limit StepCapsule Spaces to emit into Item/Tag Capsule Spaces that have the same number or more dimensions as itself (i.e. *dimensional expansion*), with matching dimensions listed in the same order. This disallows any case of *dimensional reduction*, where the produced objects have fewer dimensions that the producer. Such cases, which could lead to Tag value collisions, are invalid in the Dynamic Single Assignment (DSA) [12] property of the underlying programming mode [7].

Consumer StepCapsule Spaces, on the other hand, can have fewer or greater dimensions that their input ItemCapsule Spaces. If a consumer StepCapsule has fewer dimensions, the extra dimensions on the ItemCapsule Space can be defined with the help of *dimension definition functions* specified for the input edge. These functions can also inspect other ItemCapsule Spaces (also known as dependent edges) to define the missing dimensions. However, these dependent ItemCapsule Spaces should have dimensions fully specified by the consumer StepCapsule's dimensions. Although an implementation specific restriction that can be removed with additional support in the API, cycles are also currently disallowed in the application task-graph.

5.5 ItemCapsule Spaces: Composed over Iteration Space

When creating coarse-grain computations by composing over iteration space (sec. 5), it is crucial to also have the ability to change the granularity of data objects. We call these composable data objects, ItemCapsules. The granularity of an ItemCapsule instance depends on the granularity of the TagCapsule instance that parametrizes it. As described earlier in sec. 3.1, ItemCapsules are tree data-structures that mimic the structure of their parametrizing TagCapsule instance. This is essential to allow efficient querying of the ItemCapsule tree for relevant Items that are required to fulfill the data request of executing StepCapsule instances.

6 SMP Run-time Implementation

The current C++ Capsules SMP run-time supports only composition over iteration space.

The run-time implementation is based on a simple execution model of work queues. Each processor or processing core is considered as a separate Processing Element (PE) with each PE assigned a unique work queue and a work thread. The work threads continuously pop StepCapsule instances from the head of the work queue for execution, whereas new StepCapsule instances are inserted at the tail of the queue.

Each StepCapsule instance is either a fine-grain StepCapsule instance or is a composed StepCapsule instance of multiple Step/Item/Tag Capsule instances. In the case it is an indivisible StepCapsule instance, the execution invokes the serialization schedule of the StepCapsule and executes all the fine-grain Step instances represented by the parametrizing TagCapsule instance. Composed StepCapsule instances can be operated on in one of two ways. They can also be executed serially, requiring a serialization schedule that would define the execution order of the inner StepCapsule instances, or they can be used as a GC container. The inner StepCapsule instances in this case are executed in parallel in work queues, whereas the composed StepCapsule GC container keeps track of executed inner StepCapsules with the help of a counter. Once all inner StepCapsule instances are done executing in parallel, the GC container and all inner Item/Tag Capsule instances are GC'ed.

Every application execution is encapsulated in a default StepCapsule GC instance that contains all application StepCapsule instances. Application termination is achieved once this outermost StepCapsule GC container is GC'ed.

7 Performance Evaluation and Results

In this section we describe our evaluation methodology, test cases and results.

7.1 Evaluation Methodology

The metrics used to evaluate the application performance and the underlying Capsule mechanisms are as follows:

The **Normalized Execution Time** is the ratio of the parallel execution time with respect to the serial execution time of the original unmodified application.

The **Percentage Overhead** represents the fraction of the CPU time spent in the Capsules run-time with respect to the total work done on all PEs. The run-time overhead consists of all non-application related operations performed by the system. The overhead percentage represents the efficiency with which the Capsules programming model is able to execute the application in parallel.

The OProfile [10] statistical sampling tool was used to measure the run-time overhead. Samples gathered in application functions were separated from samples captured in the run-time to compute the application percentage overhead. Samples were captured at every 90K clock cycle intervals with a call-graph depth size of 10.

The hardware platform used was an 8-Way SMP machine with 2 x 1.6GHz Intel Quad Core, Core 2 Duo Clovertown processors, with 2 GB RAM. A 32bit Fedora Core 6 Linux OS with kernel 2.6.20-1.2962 was used.

7.2 Applications

We parallelized two applications using Capsules: (1) The Cascade Face Detector (FD) [19] (fig. 2) and (2) a Stereo Vision depth (SV) algorithm [20] (fig. 1).

The face detector applies a cascade of pre-computed simple facial features on a window of fixed size in the image to detect whether a face exists in that window or not. This detection process is performed over the entire input image by shifting the window over the *X* and *Y* axes of the image to detect faces on different locations in the image. In order to detect faces of different sizes, the image is scaled down and the detection process is repeated over again. This scaling and re-detection is repeated until the image scales down to the size of the feature detection window. A post processing phase, takes all detected faces and prunes duplicates that are close to each other.

The Stereo correlation algorithm is used to detect how far objects are placed in a given scene. It outputs a depth map from two stereo input images. The first stage in the algorithm consists of building multiple disparity-maps for the two input images. The second stage re-samples the disparity-maps to find the highest disparity values. The final stage composes the re-sampled disparity-maps to create the final depth-map.

7.3 Results

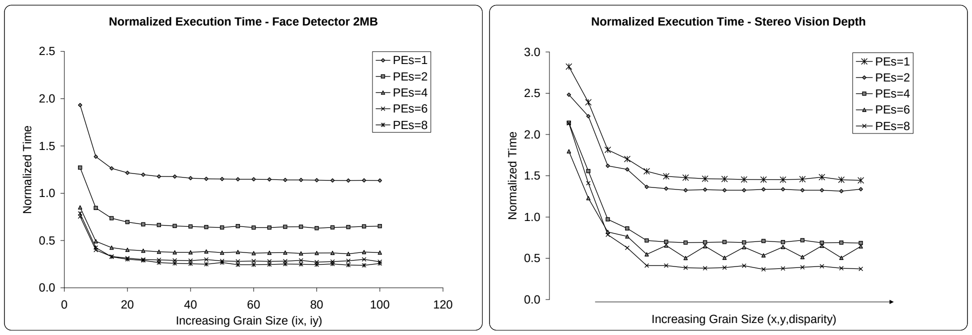

Fig. 4. Performance, Normalized Execution Time:*(left to right)*: Cascade Face Detector, Stereo Vision Depth; x-axis: Increasing Granularity; y-axis: Normalized time.

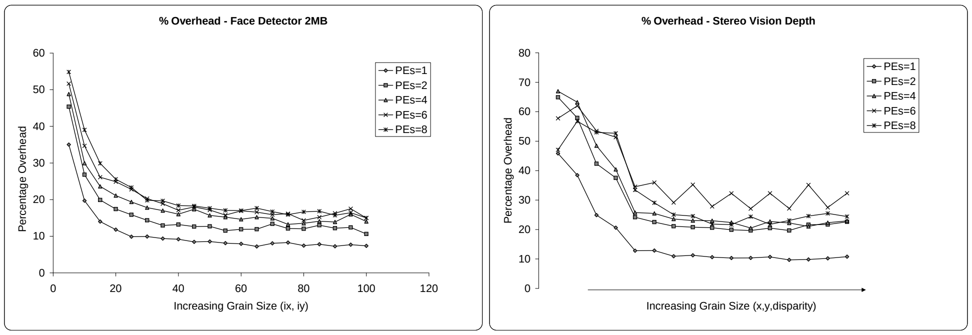

Fig. 5. Performance, Percentage Overhead: *(left to right)*: Cascade Face Detector, Stereo Vision Depth; x-axis: Increasing Granularity; y-axis: Percentage Overhead

Figure 4 illustrates the *normalized execution time* of the two applications on different number of available PEs. The x-axis for the FD graph represents the granularity selected for the detection windows in both the x and y axis location in the image $< ix, iy >$. The total number of detection windows grouped together is therefore the square of the granularity selected. The x-axis for the SV graph represents a three-dimensional granularity parameter $< ix, iy, disparity >$. The first-two parameters are again the x and y pixel locations grouped together for the first phase of processing. The third dimension disparity represents the disparity-maps grouped together for processing of the second phase of the algorithm.

Figure 5 illustrates the *percentage overhead* of all run-time mechanisms with respect to the total work done by all the PEs for a given execution.

Figure 4 clearly shows that increasing the granularity on different dimensions increases the performance of the application's parallel execution regardless of the number of PEs. Both applications speed-up by several factors with the help of granularity adjustment. Furthermore, percentage overhead in fig. 5 shows that increasing the granularity actually increases the efficiency of the parallel execution with less overhead incurred by the run-time. This is due to several factors. There is a total reduction in StepCapsule instances created during the application execution, which in-turn reduces the total overhead due to synchronization, distribution and scheduling, and book-keeping cost. The percentage overhead confirms the hypothesis that composability in Capsules, even though more memory intensive due to its dynamic data-structures than other systems such as Cilk, can be used to dynamically create efficient coarse-grain computations to reduce the total overhead of parallelization. The fewer coarser-grain computations reduce the overhead to useful work ratio of resources and thereby improve the application's efficiency in using the underlying hardware concurrency for speed-up.

Figure 4 shows that application speed-up becomes constant beyond a certain point and does not change with further increasing granularity. A similar trend is seen in fig. 5 where the percentage overhead does not go down any further beyond increasing the granularity after a certain point. This is due to the fact that the dynamic data-structures required to enable coarse-grain computations begin to out-weigh the marginal gain achieved by reducing the cost of parallelization.

8 Conclusion

We introduce *Capsules*, a parallel programming model that brings together two distinct forms of composability namely, *composability over computation space* and *composability over iteration space*. Composability at the programming model level enable a user to dynamically adjust the granularity of parallel tasks and reduce system overhead. We show in our experiments that overhead due to synchronization and run-time book-keeping costs can be minimized by adjusting the granularity of an application's concurrent tasks and moving the synchronization points to the boundary of those coarse-grain computations. Overall, composability at the programming model level enables the application developer to write a parallel application once, and tune its granularity parameters later to extract the optimal amount of potential application parallelism required to efficiently utilize the hardware concurrency.

9 Acknowledgments

The work has been funded in part by an NSF ITR grant CCR-01-21638, NSF NMI grant CCR-03-30639, NSF CPA grant CCR-05-41079, and the Georgia Tech Broadband Institute. The equipment used in the experimental studies is funded in part by an NSF Research Infrastructure award EIA-99-72872, and Intel Corp. We thank the members of the Embedded Pervasive Lab at Georgia Tech (http://wiki.cc.gatech.edu/epl/) for their helpful feedback on our work.

References

1. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
2. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPOPP '95: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, New York, NY, USA, 1995. ACM Press.
3. O. A. R. Board. OpenMP: Simple, Portable, Scalable SMP Programming, 2006.
4. L. Carter, J. Ferrante, S. F. Hummel, B. Alpern, and K.-S. Gatlin. Hierarchical Tiling: A Methodology for High Performance. Technical Report CS-96-508, University of California at San Diego, San Diego, CA, 1996.
5. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
6. Intel. C++ Compiler 9.1 for Linux.
7. K. Knobe and K. Offner. TStreams: How to Write a Parallel Program. Technical Report HPL-2004-193, Hewlet Packard Labs - Cambridge Research Laboratory, Cambridge, MA, 2004.
8. K. Kusano, S. Satoh, and M. Sato. Performance Evaluation of the Omni OpenMP Compiler. In *Proc. for Third International Symposium on High Performance Computing, (ISHPC), Springer-Verlag LNCS 892*, page 403, Tokyo, Japan, October 16–18 2000.
9. M. S. Lam and M. C. Rinard. Coarse-grain parallel programming in Jade. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 94–105, New York, NY, USA, 1991. ACM Press.
10. J. Levon. OProfile, a system-wide profiler for Linux systems.
11. R. S. Nikhil, U. Ramachandran, J. M. Rehg, R. H. Halstead, Jr., C. F. Joerg, and L. Kontothanassis. Stampede: A programming system for emerging scalable interactive multimedia applications. In *Proc. Eleventh Intl. Wkshp. on Languages and Compilers for Parallel Computing (LCPC 98)*, Chapel Hill, NC, August 7-9 1998.
12. C. Offner and K. Knobe. Weak Dynamic Single Assignment Form. Technical Report HPL-2003-169R1, Hewlet Packard Labs - Cambridge Research Laboratory, Cambridge, MA, 2003.
13. U. Ramachandran, R. Nikhil, J. M. Rehg, Y. Angelov, S. Adhikari, K. Mackenzie, N. Harel, and K. Knobe. Stampede: A Cluster Programming Middleware for Interactive Stream-oriented Applications. *IEEE Transactions on Parallel and Distributed Systems*, 2003.
14. U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications. In *Proc. Principles and Practice of Parallel Programming (PPoPP'99)*, Atlanta, GA, May 1999.
15. J. M. Rehg, K. Knobe, U. Ramachandran, R. S. Nikhil, and A. Chauhan. Integrated Task and Data Parallel Support for Dynamic Applications. *Scientific Programming*, 7(3-4):289–302, 1999. Invited paper, selected from 1998 Workshop on Languages, Compilers, and Run-Time Systems.
16. M. C. Rinard, D. J. Scales, and M. S. Lam. Heterogeneous Parallel Programming in Jade. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 245–256, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
17. M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *Computer*, 26(6):28–38, 1993.
18. H. Sutter and J. Larus. Software and the Concurrency Revolution. *Queue*, 3(7):54–62, 2005.
19. P. Viola and M. Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. *cvpr*, 01:511, 2001.
20. R. Yang and M. Pollefeys. A Versatile Stereo Implementation on Commodity Graphics Hardware. *Journal of Real-Time Imaging*, 11:7–18, February 2005.