Multi-Dimensional Concerns in Parallel Program Design

Parallel programming is difficult, thus design patterns.
Despite rapid advances in parallel hardware performance, the full potential of processing power is not being exploited in the community for one clear reason: difficulty of designing parallel software. Identifying tasks, designing parallel algorithm, and managing the balanced load among the processing units has been a daunting task for novice programmers, and even the experienced programmers are often trapped with design decisions that underachieve in potential peak performance. Over the last decade there have been several approaches in identifying common patterns that are repeatedly used in parallel software design process. Documentation of these "design patterns" helps the programmers by providing definition, solution and guidelines for common parallelization problems. Such patterns can take the form of templates in software tools, written paragraphs with detailed description, or even charts, diagrams and examples.

The current parallel patterns prove to be ineffective.
In practice, however, the parallel design patterns identified thus far [over the last two decades!] contribute little—if none at all—to development of non-trivial parallel software. In short, the parallel design patterns are too trivial and fail to give detailed guidelines for specific parameters that can affect the performance of wide range of complex problem requirements. By experience in the community, it is coming to a consensus that there are only limited number of patterns; namely *pipeline, master & slave, divide & conquer, geometric, replicable, repository*, and not many more. So far the community efforts were focused on discovering more design patterns, but new patterns vary a little and fall into range that is not far from one of the patterns mentioned above. The source of ineffectiveness of these patterns, in fact, does not come from its lack of variety, but it comes from inflexibility of how existing patterns are presented.

Tyranny of Dominant Decomposition
To facilitate the idea to be presented, we look at an analogy from the recent progress in the software engineering community. The object oriented programming has been a popular technique in encapsulating independent entities in a program. These entities are identified by the hierarchical organization structure that the programmer rigidly decides. By using object and its hierarchy, object oriented approach was successful in giving the programmer an abstraction that are easy to utilize. However, in reality one dominant hierarchy of objects are often problematic in capturing different concerns. A concern can be a part of software that is relevant to a particular concept, goal, or purpose. A fixed hierarchy might be successful in isolating *some* of the concern, but can fail to satisfy others, causing one concern to be crosscutted throughout the objects in the fixed hierarchy. Many call this phenomenon the *tyranny of dominant decomposition*, because one dominant way of decomposing the program imposes a structure on the software that makes it difficult or impossible to encapsulate other kinds of concerns. Since the whole purpose of introducing object was to identify different concerns in the first place, separation of concerns [even the crosscutting ones] have brought attention to the community.

Separation of Concerns
Aspect-oriented programming introduces the notion of separation of concerns in multi-dimensional space. In this approach, there is more than one hierarchy for the same software, each dimension representing a space of different concerns. The tyranny of dominant decomposition is

now removed by having multiple dimensions representing their own set of concerns, which can be isolated using an appropriate hierarchy. There are many different approaches in achieving the separation of concerns, but the main idea is to partition each dimension of concerns into slices and identify the relationship among them within and across the dimensions. These slices in the spaces then can be chosen in a "mix-and-match" fashion that will contain the concerns of interest. Thus, the driving force behind the separation of concerns is the partitioning of concerns and defining relationship [interaction] between each tuple in the hyperspace. Note there can also be "orthogonal" concerns that will be independent from each other, which in that case we need not define any interaction.

Current design patterns impose inflexible decision-making process.
We analogously argue that the existing design patterns reflect the "tyranny of dominant decomposition". Namely, the design process has been geared towards "applying existing algorithms" mentioned above, and it fails to expose diverse set of concerns [that governs communication, decomposition, etc…] and their interactions. By rigidly structuring the design patterns only along the algorithm space, many other concerns are predetermined as a byproduct without consciously making decision over them. The problem lies not in the incompleteness of the algorithm space but in lack of flexibility to embrace other concerns in noninvasive manner. Thus, to identify and separate other concerns other than algorithmic issues in designing parallel software we must have multiple hierarchical perspectives that allow a programmer to look at different space of concerns. A programmer with multidimensional decision space can now choose several concerns that fit the main design goals and work to meet the composition of these concerns as their interactions [thus possibly performance parameters] are closely adjusted. This differs from having one dominant decision making process because one concern is separated, not a produced, by other.

Dimension, Concern, Unit
A software designer may wish to include many dimensions of concerns, such as programmability and scalability, and it should be the long-term goal of this study. However, in general the essential design decisions of parallel software design can be limited to the following three areas of interest:

1) Decomposition: Identify the tasks for efficiency.
2) Communication: Minimize the time caused by overhead.
3) Load-Balancing: Make sure each processor spend equal amount of time working.

We call these areas *dimensions*. Each dimension has a goal to accomplish, and contains several specific means of arriving the goal. These means of achieving the dimensional goal is a main **interest [concern]** for the programmers, hence the term *concern*. The primitive blocks that makes up each concern is called *units*. Units are the possible design choices or fixed characteristic that a problem inherently possesses. A concern is overlapped if they share common units [decision choices, characteristics], and orthogonal [independent] otherwise. Units are not bound to a concern or a dimension, but are building blocks for the whole design process. Thus, any concerns in the design process will be considered in the basis of same set of units [decision choices, characteristics]. This enables the designer to observe the interaction of the concerns, while at the same time approach the design process from different dimensions of interest.

Table 1. Description of Multidimensional Design Process
(Unit << Concern << Dimension)

Component	Description
Dimension	An area of interest. Includes a goal and a collection of common concerns.
Concern	A concern is a mean of reaching the goal for the dimension. Main concern for the design makers.
Unit	The characteristic and decision choices that the designer works upon.

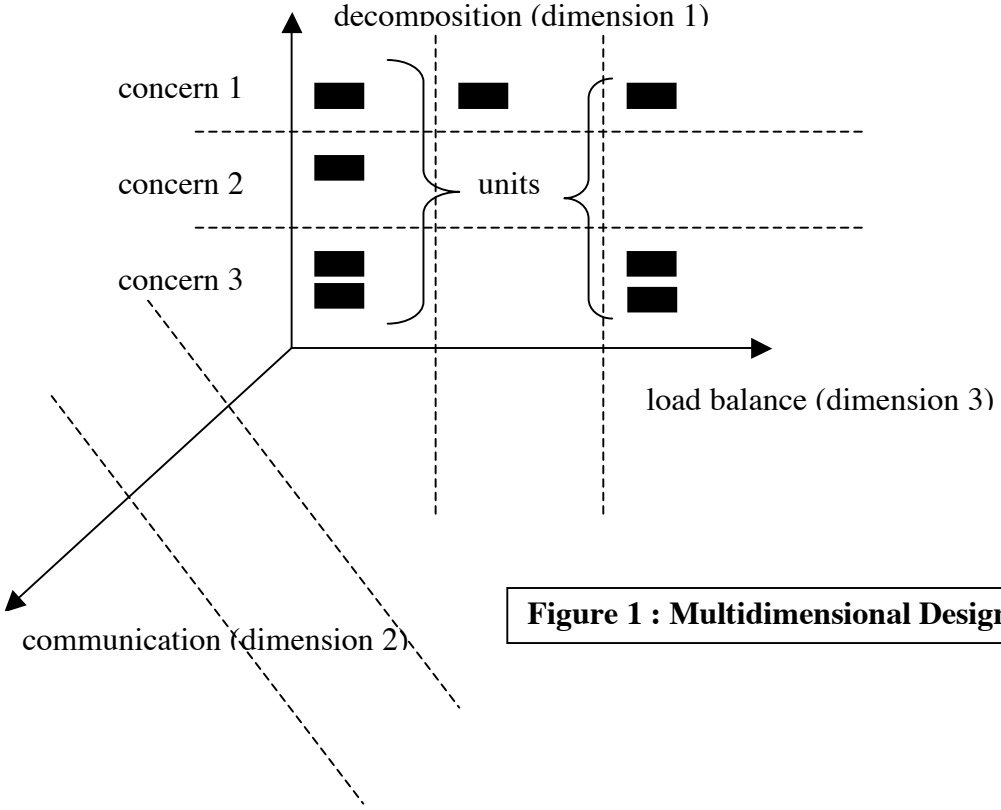

Figure 1 : Multidimensional Design Process

Each design space dimension includes concerns and decision choices that can be asked without intervention of other dimension. In other words, these autonomous units of concerns answered and provided within a single dimension are self-sufficient to achieve the goal of the dimension that it resides in. Each dimension should be treated as an independent sub-goal of the whole design process as much as possible. However, the designer should pay attention to the *interaction* of the overlapped concerns across the dimensions. There will be a point in the process where a decision in a dimension affects a decision in another dimension. If *tradeoffs* occur, a conscious adjustment has to be made according to the merit of each decision. [This flexibility is in fact the strength of this multidimensional approach].

I. Decomposition :: *Identify task for efficiency*

1) Exploit Full Concurrency :: FullConcurrency

Relevant Units:

- Granularity
- Number of PE
- Data dependency shape
- Code Segment
- Search Space
- Speculative Value (Computable Intermediate Result)
- How recursive
-

2) Minimize Dependency :: MinDependency

Relevant Units:

- Granularity
- Data dependency shape
- Code Segment
- Search Space
- Speculative Value (Computable Intermediate Result)
- How recursive

II. Communication :: *Minimize overhead*

1) Minimize Frequency of Communication (spatial locality) :: MinCommFrequency

Relevant Units:

- Periodicity of the phases (strict, relaxed, chaotic)
- Availability of data in advance? or not available due to dependency.
- Static / Dynamic Communication (knowledge of sender and receiver)

2) Minimize Total Volume of Communication (temporal locality) :: MinCommVolume

Relevant Units:

- Dimension of mapped partition (surface-to-volume effect).
- Use of intermediate result.
- Static / Dynamic Communication

3) Minimize Contention & Hot Spots :: MinContention

Relevant Units:

- Communication Dependency
  (if serial or geometric, we can pipeline, avoid central).
- Availability of data in advance.
- Static / Dynamic communication
- Periodicity of phases (for pipelining)
- Mapping centralized / decentralized

4) Overlap Computation with Communication :: OverlapComputation

Relevant Units:

- Granularity (multiple task resident in one PE, then overlap chance)
- Paradigm supporting nonblocking calls
- Static / Dynamic communication
- Prefetch available (hardware, software)
- Predictable computation availability

5) Replicate Data or Computation :: ReplicateDataAndComp

Relevant Units:

- Memory Requirement
- Number of processes
- Self-computable result
- Comparison estimate of transfer vs. computation.

6) Use Optimized Collective Operations :: CollectiveOp

Relevant Units:

- Existence of group-involving communication.
- Periodicity

7) Overlap Communications :: OverlapCommunication

Relevant Units:

- Communication dependency (serial, geometric, hypercubic)

III. Load Balancing :: *Make each processor have equal amount of work.*

1) Achieve Good Mapping in the First Place :: StaticMap

Relevant Units:

- Geometric shape of data dependency
- Granularity
- Size of data associated with task
- Static / Dynamic generation of tasks
- Dependency graph
- Method of partitioning
  (domain decomposition: block, cyclic, random, hierarchical, graph partitioning)

2) Minimize the Frequency of Stealing :: MinStealing

Relevant Units:

- Static mapping or dynamic mapping
- Granularity
- Size of data associated with task
- periodicity of phases
- Dependency graph

3) Don't Leave Any PE Idle :: NoIdleProcess

Relevant Units:

- Granularity
- Dependency graph
- Cost size of data transfer
- Data availability known

*IMPORTANT: Define which are orthogonal, and which are overlapped.

Overlapped Concerns

We observe the units of individual concerns within and across the dimensions to find out if any overlapping occurs. Any two concerns that share one or more common unit are considered to be overlapping, and must be examined for conflict/trade-off situation. The overlapping concerns do not have to be a pair, but rather a set of concerns that exhibit a common unit.

To begin with representative examples, we have the following units of interest:

1) *Granularity* { FullConcurrency, MinDependency, MinStealing, NoIdlePE }
   - There is a tradeoff situation in decomposition dimension and load balancing dimension. For coarse-grained tasks, obviously dependencies are reduced. However this increases the chance of load imbalance due to uneven sized tasks distributed to the processing units. If the granularity is fine-grained, then full concurrency can be exploited. But because of the small task sizes, the frequency of dynamically stealing task the workload has to be increased.

2) *Communication Topology* { MinContention, MinDependency, FullConcurrency }
   - The identification of the tasks during the decomposition process determines a communication topology. Although the resulting topology might best suite the concerns of minimizing dependency and exploiting the full concurrency, hotspot might not be avoided as the result. For example, Master&Worker communication topology might suffer from contention as master node continues to assign tasks to other workers.

3) *Code Segment* { FullConcurrency, MinDependency, MinFrequency }
   - A modular code segments that can be grouped as a functional block can be identified as a control decomposition unit. A block of functional code segment can be separated from the program between two intermittent communication activities. Control decomposition obtains parallelism by pipelining linear functional units together, and it is only successful when enough tasks are contained in the pipeline concurrently. However, deeper pipeline with large number of tasks generates frequent communications in between the functional units, thus conflicting with the MinFrequency concern.

4) *Dimension* { MinDependency, MinFrequency, MinTotalVolume }
   - Due to the surface-to-volume effect, decomposing the data into higher dimension reduces the total volume of the interprocessor communication. But higher dimension also introduces more surfaces which translates to more number of targets. This means that a task now needs to communicate with more neighbors surrounding its dimension, and equivalently more dependencies are observed. Thus there are certain tradeoff situation between minimizing the total volume of communication and minimizing the dependencies (hence also the frequency) among the processors.

5) *Speculative Decomposition* { ReplicateData, FullConcurrency }
   - If the designer chooses to use speculative decomposition, a task begins communicating with other tasks before the decision has been made. That is, the data that determines the

decision process is “speculated” before it is computed and arrives in the task. Upon correct speculation, the tasks takes advantage of doing some critical work that otherwise would have been untouched during idle time. If the designer chooses this decomposition technique then the option of replicating the data should be ignored. Replication of data and speculation contradict each other because speculation does not need ready data to go onto the next set of operations, nor the replication of data allows speculation to occur because the data is already there.