Recursive Splitting

Problem

Consider the problem where an algorithm can be expressed as the composition of a series of tasks that are generated recursively or generated during the traversal of a recursive data structure. How can such problems be implemented on parallel hardware efficiently?

Context

All data structures can be considered recursive and algorithms that traverse them expressed recursively. For example, an array can be expressed recursively as

\[
\text{Array} := \langle \text{element, Array} \rangle | < > \quad \text{i.e. an array has an element followed by an array (recursive definition) or is empty (base case). Other data structures like trees, graphs etc also have recursive definitions. Algorithms that traverse these structures can be expressed as recursive algorithms by thinking of the data structures as recursive structures.}
\]

We also have algorithms that are best expressed recursively (Many flavors of sorting, for example, though non-recursive sorting algorithms are just as popular). In such cases, usually a problem is recursively split into smaller problems until the problem is small enough to solve directly. A more commonly known concept in algorithms is “Divide & Conquer”. A large problem is split into several smaller pieces and each one of them solved separately. Then all the sub-solutions are merged to form the solution to the original large problem. In most cases, the sub problems themselves can be split into smaller problems and so on recursively. Such a structure is a natural fit for the recursive splitting pattern. However, note that not all divide & conquer problems are expressed (or implemented) as recursive algorithms or use recursive data structures and hence other algorithmic patterns like data parallelism, task parallelism or geometric decomposition may be appropriate, depending on the problem at hand.

The solutions to the recursive splitting problems can be viewed as solving recursively-generated tasks - e.g. quick sort or binary tree search. This involves either recursively partitioned data structures or regular geometric data structures with tasks defined recursively. Computational patterns like Backtrack Branch & Bound, some instances of Dense and sparse linear algebra, n-body methods etc can generate such recursive splitting.
Also, many dynamic programming problems can be implemented using recursive data structures.

This pattern is particularly useful when the knowledge of the hardware resources available (e.g., Number of hardware thread contexts available) is not always known. In those events, an algorithmic strategy that produces tasks but does not know about handling them is desirable. The handling of the generated tasks can then be handled by implementation strategy patterns like task queue, fork-join etc. This pattern can be considered a specific instantiation of the more general task parallelism pattern, which is concerned with parallelizing any algorithm that generates tasks, not necessarily recursively.

The major concerns with mapping such recursive structures to parallel platforms are determining recursion-depth-vs-computation-per-node, load balancing and locality considerations.

**Forces**

**Universal Forces:**

1. Many data structures can be decomposed into geometric chunks or recursively split. The choice of splitting lies in the algorithm being implemented and ease of programming. Recursive splitting may be the most natural way of expressing an algorithm (with less performance) where geometric decomposition may lead to better performance at the cost of increased programmer effort.

**Implementation Forces:**

1. Generating tasks from recursive data structures can lead to tasks with widely varying computation requirements. Trying to keep all the tasks uniform pushes us towards using regular patterns like geometric decomposition, away from recursive structures.
2. Defining a small base case for the recursion generates a large amount of concurrent tasks and keeps the hardware busy, however, the efficiency per thread can be low. Defining larger base cases can improve the computational efficiency per thread by keeping the overheads low, but might not generate enough concurrency.
3. Generating more tasks can have more overheads, but is good for dynamic load balancing. Generating few large tasks is efficient, but only when the tasks are known to have very similar computational requirements and will not lead to load imbalance.

4. Recursive tasks produced from a single task may share read-only or read-write data. Parallelizing these tasks leads to contention, pushing us towards serial execution as much as possible. Patterns like Agent & Repository or Mutual Exclusion can help in resolving such conflicts.

**Solution**

The following steps are needed for an algorithmic strategy involving recursive splitting:

1. Express problem recursively with more than one task generated per call
2. Use a balanced data structure, if possible
3. Use a fork-join or task-queue implementation
4. Use optimizations to improve locality

Take the example of quick sort. The following example is written in NESL, but the user can ignore the syntactic details and focus just on the algorithm. The algorithm works as follows – Given an array to be sorted, we take an element in the array (could be in any position) and place it in its appropriate position in the sorted array i.e. reorder the array so that we have two unsorted arrays, both having elements lesser than or greater than the element and the element itself placed in the correct position. Then sort the two sub-arrays using quicksort recursively.

```nesl
def quicksort(a) =
  if (#a < 2) then a
  else
    let pivot = a[#a/2];
    lesser = {e in a| e < pivot};
    equal = {e in a| e == pivot};
    greater = {e in a| e > pivot};
    result = {quicksort(v): v in [lesser,greater]};
    in result[0] ++ equal ++ result[1];

quicksort([8, 14, -8, -9, 5, -9, -3, 0, 17, 19]);```

Express problem recursively with more than one task generated per call

The key concept here is to generate **more than one** task per task recursively. Generating exactly one task will not create more concurrency and will not map to hardware parallelism. If the tasks are generated by traversing a recursive data structure, the recursive definition used to express the data structure needs to be inspected. For example, an array can be recursively defined in many ways – two such definitions are shown below:

```plaintext
Array := <element, Array> | <>
Array := <Array, element, Array> | <>
```

Using the latter definition might be better than using the former definition if we want to express the concurrency in the algorithm, as acting on one array generates two sub-arrays recursively.

In the quicksort example above, the statement

```plaintext
result = {quicksort(v): v in [lesser,greater]};
```

performs the recursive task generation. For each array to be sorted, it generates two other sub-arrays to be sorted. This is in contrast to sorting techniques like selection sort[http://en.wikipedia.org/wiki/Selection_sort] which written recursively, would use the former recursive definition of an array. Selection sort sorts an array by choosing the smallest element in the array, swapping it with the first element and then recursively do it for the sub-array with the first element removed. Since it only produces one task (on a smaller array) recursively, it is does not create more concurrency and violates our starting premise.

The major implementation-level question that needs to be considered here is the number of tasks that are generated from each task. Generating a large number of tasks from each task can overwhelm the resources of the machine, especially if the lower-level implementation of task queues (say) is not very efficient. On the other hand, generating few tasks may be optimal, if each task takes a substantial amount of time to execute. In the quicksort example, each task takes time proportional to the sub-array it receives, before generating sub-tasks. It might be worthwhile to consider parallelizing the work inside each task itself. We could parallelize the generation of the “lesser”, “equal” and “greater” arrays in quicksort using **data parallelism** pattern,

```plaintext
lesser = {e in a| e < pivot};
equal = {e in a| e == pivot};
greater = {e in a| e > pivot};
```
This implies that we would be composing \textbf{data parallelism} inside \textbf{recursive splitting}.

\textbf{Use a balanced data structure}

This section discusses the importance of having a good base case for the recursion and load balancing. In our context, “balanced data structures” are recursive data structures where the recursive sub-units are of almost equal sizes or the tasks operating on them have almost equal computational requirements. Using “balanced” data structures is very important for load balancing purposes. This balancing can be enforced on many data structures. E.g. Binary trees can be made complete or almost complete. Techniques exist for making binary search trees to be self-balancing. Refer to AVL trees or Red-black trees for more details. In many cases, it is possible to choose particular versions of data structures that would maximize computational efficiency (e.g. tree search algorithms do not care about the exact structure of the tree, and it would be our advantage to preprocess the trees to make them more balanced).

In the quick sort example, balancing is not guaranteed – i.e. the “lesser” and “greater” arrays need not be of the same length. If the array is not almost-sorted to begin with, then on average, these differences do not matter. Parallel quicksort using recursive splitting is not a good idea if the array is almost-sorted. For some bad selections of the pivot, quick sort on an almost sorted would degenerate to a selection sort. Hence, even though recursive splitting generates concurrent tasks, half of these tasks operate on small ($O(1)$ sized) arrays, while the other half operate on almost the entire array.

Another concern is to specify the base case of the right size. A small base case will lead to the generation of too many tasks (many of which only serve to spawn other tasks). A large base case may not generate enough active tasks to keep the hardware busy. In the quicksort example, the base case is too small – an array of size 1 is sorted by default. This is not the ideal scenario – a better option is to use small efficient sorts like insertion sort or a serial version of quicksort within a task if the array size is less than a threshold (depending on the problem size and machine capabilities).

\textbf{Use a fork-join or task-queue implementation}

A developer writing a recursive splitting algorithm may decide to write his/her own task management system or may choose to use other existing implementations. In this regard, the
most relevant implementation strategy patterns to look at are **fork/join** and **task-queue**. The ideas behind these patterns are essential to any developer who wants an efficient recursive splitting implementation.

We provide a one-line description of the two patterns here and ask the reader to read their pattern language descriptions for more details.

**Fork/join:** Threads are logically created (forked), used to carry out a computation, and then terminated after possibly combining results form the computations (joined).

**Task-queue:** Threads generate work units/tasks, which are then placed in a concurrent taskqueue. Any thread that is finished with its work queries the task queue and gets another task to execute (if unfinished tasks exist) or retires itself (if no more tasks remain).

### Use optimizations to improve locality

Recursive algorithms naturally preserve locality on a single processor, as subtasks share data with large tasks that spawned them. However, with multiple parallel processors, the situation is more complex. Some optimizations to preserve locality include:

1. Try to schedule parent tasks and child tasks on the same processor
2. Use recursive algorithms that do not require sharing data after the task split is done (e.g., quicksort instead of mergesort). Note that the tasks can share the data structure itself as long as they do not share the same portions of the structure (e.g., in quicksort, the subtasks operate on different portions of the same array).
3. If all tasks need access to a common shared data structure, see if a particular task schedule could improve the data access characteristics (e.g., the task queue implementation could support cases where a thread might ask for a new task that could reuse some of the data already present in its cache/local stores).

### Invariants

### Preconditions

1. A recursive version of the algorithm that generates more than one task per call is given.
2. A recursive specification should create smaller sub-problems that can be composed to solve the original problem.
Post-conditions

Invariants
1. The number of tasks generated should be finite.
2. The number of active tasks should decrease eventually and go to one as the problem is solved.

Example

Binary tree search – using cilk

A binary tree [5] is a tree data structure in which each node has at most two children. Shown below is an example of a binary tree that is neither sorted nor balanced.

```
A Binary tree can be recursively defined as
BinTree := <BinTree,Element,BinTree> | <>
i.e. a binary tree is empty or is composed of an element at the node and two binary trees as its left and right children.
```

If we want to search for a particular element in the binary tree, a recursive splitting algorithm using cilk would look like this:

```
cilk int node::search_binary_tree(int query)
{
    if(node == NULL)
    {
        return false;
    }
    if(node.value == query)
    {
```
There are several problems with this simple implementation – Very little computation per task, load balancing etc. We can solve these problems in the following ways:

1. Use a balanced tree structure – this helps in load balancing
2. For each node, keep a another variable that stores the count of the number of nodes in the sub-tree rooted at that node – this helps in load balancing in unbalanced trees
3. Search a small sub-tree (not just a single node) before spawning off tasks – this increases the computational load per task

Note that our algorithm still has to traverse the entire tree in the worst case. If search efficiency is an important property for the tree, we should consider sorted variants of binary trees like binary search trees.

**Search in unsorted array – comparison between geometric decomposition & recursive splitting**

The following is an example of searching in an unsorted array. The concurrency exposed is through geometric decomposition of the array into “num_omp_threads” chunks. The code is written in C with OpenMP pragmas.

```c
search(int *a, int N, int q)
{
    #pragma omp parallel for
```
for(i=0;i<num_omp_threads; i++)
{
    for(int j=i*N/num_omp_threads;j<(i+1)*N/num_omp_threads;j++)
    {
        if(a[j]==q)
        {
            return true;
        }
    }
}
}

The same search algorithm written using recursive splitting in a C-like language. There are
two artificial constructs here to illustrate the point:

1. add_task : creates a new task and adds it to a task queue
2. wait : puts the executing thread in inactive state till the task for which it is waiting on is complete

search(int* a,int N,int q)
{
    if(N==0)
    {
        return false;
    }
    if(a[N/2]==q)
    {
        return true;
    }
    else
    {
        bool x,y;
        add_task: x=search(a,N/2,q);
        add_task: y=search(a+N/2+1, N/2, q);
        wait(x);
        wait(y);
    }
}
Known uses

Divide and conquer
Cilk/Cilk++
Intel Thread Building Blocks – Task parallelism constructs

Related patterns

Upper level

Dense Linear Algebra – Many dense linear algebra problems can be written recursively. Frameworks like FLAME[4] help generate efficient code for linear algebra routines given a recursive definition.

N-body methods – Hierarchical space partitioning using recursive data structures like quad-trees and oct-trees are used in N-body methods

Backtrack-Branch-and-Bound – The generation of sub-problems to be solved from larger problems can be done recursively.

Dynamic programming – A recursive definition of the problem is usually the starting point for dynamic programming.

Lower level

Fork-Join – Implementation strategy pattern mainly useful for implementing a recursive splitting pattern

Task queue - Implementation strategy pattern mainly useful for implementing a recursive splitting pattern

Same level
**Geometric Decomposition** – Algorithmic strategy pattern which can be used as an alternative pattern if the division of work into tasks is based on a regular decomposition of the data structure(s) involved.

**Task parallelism** – Algorithmic strategy pattern where tasks are generated, but not necessarily recursively. Can be considered a more general pattern to recursive splitting.

**References**


**Authors**

Narayanan Sundaram 04/06/2009
Shepherd : Kaushik Ravindran