Data Parallel Design Patterns
Tim Mattson

Data Parallelism has moved to the forefront of parallel computing. This is driven in part by hardware trends. Vector instructions have been integrated into the instruction sets in most commercially important CPUs. GPUs memory hierarchies and floating point capabilities make them suitable for applications well beyond graphics. At a more fundamental level, however, the increasing emphasis on data parallelism emerges from the need to find more fine grained tasks in a problem. As core counts climb, we can no longer achieve the scalability we need through coarse grained tasks.

In this note, we will describe the design patterns associated with data parallel algorithms. These patterns will be aligned with the three lower layers in OPL

- **Concurrent Algorithm Strategy patterns**: Define a high level approach to exploiting the concurrency in an algorithm.
- **Implementation Strategy patterns**: Techniques to express concurrency in software.
- **Parallel execution patterns**: Describe how concurrent structures in software execute on actual hardware

The relevant patterns we will consider are:

- **Data Parallelism**: An algorithm can be designed in terms of operations applied concurrently to the elements of a set of data structures. The concurrency is in the data. This pattern can be generalized by defining an index space. The data structures within a problem are aligned to this index space and concurrency is introduced by applying a stream of operations for each point in the index space.

- **Strict data parallelism**: Many data parallel algorithms can be implemented in software as one single stream of instructions applied to each point in the data parallel index space. This approach is easy to appreciate when the data elements at each point in the index space do not depend on any other data points. It can also be used when the sharing can be represented by regular shift operations or through more collective operations (e.g. parallel pre-fix or shift and mask operations).

- **SIMD**: An execution strategy in which instructions execute in lockstep by a set of processing elements but on their own streams of data. SIMD programs use specialized data structure, data alignment operations, and collective operations to extend this pattern to a wider range of data parallel problems.

These three patterns are closely linked. In many cases, the three “travel together” and merge together into a large scale data parallelism framework (as shown in [Catanzaro09]). Another common approach is to start with the top level data-parallelism pattern, and then implement the algorithm in software using the SPMD-pattern (commonly used with OpenCL) or loop-level parallelism (commonly used with OpenMP). The fact that the single common idea of data parallelism maps onto such different implementation mechanisms shows the value of keeping these three patterns distinct.
The data parallel pattern

**Problem:**

Some problems are naturally expressed as a sequence of instructions separately applied to each member of a collection of data; i.e. the concurrency is in the data. What strategy can we use to exploit this concurrency in a parallel program?

**Context:**

Consider a typical graphics rendering pipeline. A scene is tessellated into polygons which are then decomposed into triangles. Each triangle is analyzed relative to its environment (lighting model, geometry, texture map, etc.) to produce pixels within an image.

The operations applied are roughly identical for each triangle and can proceed independently. Concurrency can be expressed simply by specifying “do this to each triangle in the scene”.

This is the classic data parallel strategy for exploiting the concurrency in a problem. A problem is understood in terms of its core data structures and concurrency is expressed in terms of concurrent updates to different members of these data structures.

**Forces:**

**Single-Instruction vs. Multi-Instruction:** Applying the same stream of instructions on each processing element limits the range of problems where this pattern can be applied. Letting the instructions vary between processing elements breaks the basic idea of this pattern but if done carefully can greatly expand the scope of this pattern.

**Communication/work tradeoffs:** Communication can be slow compared to computation. Parallel programmers may choose to redundantly compute values just to avoid the cost of communication. In the case of the data parallel pattern, however, redundant work may be needed to avoid branches support the use of a single instruction stream.

**Solution:**

In a data parallel algorithm, a single sequence of instructions is applied to each element of a set of data items. The basic solution unfolds as follows.

1. Define an index space.
2. Align the key data structures in your problem around this index space.
3. Express the instruction that comprise the solution to your problem in terms of the index space.
4. Apply the instructions ‘for each” point in the index space.

The following trivial example is sufficient to explain each of these steps. Consider the vector addition problem. Given two vectors:

\[ A, B \in R^N \]

Construct a third vector, C, as

\[ C \in R^N \]
\[ C_i = A_i + B_i \]
\[ i \in Z^N, 0 \leq i < N \]

Applying the steps for our data parallel strategy to this admittedly trivial problem we have

1. Define an index space.
   • In this case the array index i defines the index space
2. Align the key data structures in your problem around this index space.
   • the indices of the three vectors A, B, and C are naturally aligned with the index, i.
3. Express the instructions that comprise the solution to your problem in terms of the index space.
   • there is only one instruction in this case which we can write as
     \[ C_i = A_i + B_i \]
4. Apply the instructions ‘for each” point in the index space.
   • Express as:
     \[ \text{foreach}(i \in (0, N))C_i = A_i + B_i \]

This example is trivial since the index space immediately maps on to an index in the original problem. A surprisingly large number of problems encountered in practice are this simple. In other words, if the problem has as a natural part of its definition an iterator that traverses simple arrays, the mapping onto a data parallel algorithm strategy is relatively simple.

Common issues to consider when working with data parallel algorithm strategies include:

1. Aligning arrays may require transformations of problem indices. For example, in a matrix multiplication problem \((C = A * B)\), we can express the update of each element of the product matrix \(C\) as an inner product of a row of \(A\) and a column of \(B\):

\[ A, B, C \in R^{N,N} \]
\[ C_{i,j} = (A_i)^T \cdot B_j \]
Hence, a single set of indices does not span the elements of the two vectors in the inner product. The solution, as suggested by the mathematical definition of the problem, is to input the vector \( A \) in transposed order so a single index indeed spans both vectors.

2. The operations defined at each point are gathered into a function. This function is frequently referred to as a kernel. Application of a kernel implies streaming the full set of relevant data through the kernel for each instance of a kernel invocation in a program. Since data movement can dominate performance in a problem, it makes sense to fuse kernels. For example, consider the problem:

\[
\text{foreach}(i \in (0,N)) \text{Func1}(A,B,i);
\text{foreach}(i \in (0,N)) \text{Func2}(C,i);
\text{foreach}(i \in (0,N)) \text{Func3}(A,D,i);
\]

If possible, you should fuse the three kernels into a single kernel and hence do all the computations for a single pass through the data

\[
\text{foreach}(i \in (0,N)) \text{Func123}(A,B,C,D,i);
\]

3. Data movement costs dominate performance. The algorithm strategy may require tiling of the arrays to maximize useful work per data access. We explore this issue in more detail in the following section. This topic is also addressed in the geometric decomposition pattern.

\textbf{Example:}

The N-body problem (see the N-body computational pattern) is essential for modeling the evolution of systems of particles in both gravitational and coulomb potentials. The N-body method scales as the number of particles squared if a naïve direct sum algorithm is used. For large numbers of particles more sophisticated methods are used that scale logarithmically in the number of bodies. For modest numbers of particles, however, the direct sum method may still be preferred.
In this section we will consider this direct sum method (based on the presentation in [Mattson09]). The basic algorithm is shown in figure 1.

The force is computed for each particle based on the location of every other particle; hence the N squared scaling of the computation. Parallelism can be achieved simply by running the outer loop in parallel (see figure 2).

```plaintext
forall bodies i in parallel {  // outer loop over i
    force = 0;
    pos = position[i];

    foreach body j {            // inner loop over j
        force += computeForce(pos, position[j]);
    }
}
Figure 2: Naïve parallelization of direct sum N-body method, where we focus on the force computation alone.
```

The problem with the code in figure 2 is that for each point in the system, the positions of all other points will be read. This excessive data movement will all but assure that the problem is bandwidth bound on most systems. A better approach is to give the program the opportunity to exploit local memory (should it be available) so all the particles in the outer loop share the positions input for the inner loop. This is shown in figure 3 where we introduce an intermediate loop over tiles of particles.

```plaintext
forall bodies i in parallel {  // outer loop over particles
    force = 0;
    pos = position[i];

    foreach tile q {            // loop inside a tile
        foreach body j in tile q { // loop inside a tile
            local[p] = position[q*tile_size + p];  // collect positions inside a tile
        }
        barrier

        force += computeForce(pos, local[j]);
    }
    barrier
}
Figure 3: Tiled algorithm so to do more computing for each particle position read from memory.
```
When working with tiled algorithms, the size of a tile is chosen such that the local data will fit in a stage of the local memory that is close to the processing elements. Note that by organizing the algorithm into tiles, we overlap with the solution in the geometric decomposition pattern. This is not surprising since geometric decomposition is a type of data parallelism.

**Known uses:**

This pattern is extensively used in graphics applications becoming so ubiquitous that a class of processors (GPUs) were created just to support data parallelism.

More recently, support for the data parallel pattern resulted in data parallel programming languages such as CUDA and much later, OpenCL. Older language support for data parallelism can be found in HPF.

General purpose GPU (GPGPU) applications often use this pattern and support a diverse range of application from protein folding and galaxy formation, to options pricing models.

**Related patterns:**

This pattern is closely related to the geometric decomposition pattern; especially for structured mesh problems where regular array structures are common.

The strictly data parallel and SIMD patterns are common lower level patterns used with the data parallel pattern.

If \( (\text{data} + \text{single\_stream\_of\_instructions}) = \text{task} \), then the task parallelism pattern is an alternative way to express data parallelism.

Fine grained applications of the SPMD pattern work well for data parallelism (see CUDA and OpenCL).
Strict Data Parallelism Pattern

This pattern has not yet been written.

SIMD

This pattern has already been written and can be found both online in the OPL online patterns and in the paper [Massingill07].

References

