Perspectives in parallel programming

MARCO DANELUTTO
PHD COURSE
MAY 2015
Parallel programming methodology

1. Figure out parallelism (exploitable concurrency)
2. Implement the parallel activity graph
   - With the available mechanisms
   - Taking care of the target hw features
3. Debug
4. Performance tuning
   (possibly loop back to 1. !!!)
Methodology (alternative): wish list

Learn from past
Reuse
Write once run everywhere
Tool support
Program *what* rather than *how*
  ◦ Functional vs non functional ...
Two alternative scenarios

Algorithmic skeletons
- From HPC community
- Early ’90
- Ready to go systems

Parallel design pattern
- From SW ENG community
- Early ‘2000
- Recipes
Parallel design patterns

Patterns
  ◦ Description/recipes for parallelism exploitation

Two distinct problems solved
  ◦ Find potentially concurrent activities
    ◦ alternative decompositions
    ◦ with possibly radically differences
  ◦ Parallelism exploitation
    ◦ program activities (threads, processes)
    ◦ program interactions (communications, synchronizations)
    ◦ → overhead
Parallel patterns refs (initial)

Researchers active since beginning of the century


- Berna L. Massingill, Timothy G. Mattson, Beverly A. Sanders, Patterns for Finding Concurrency for Parallel Application Programs, (pre-book)

- Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill, Patterns for parallel programming, Addison Wesley, Pearson Education, 2005
Architecting parallel software with design patterns, not just parallel programming languages. Our situation is similar to that found in other engineering disciplines where a new challenge emerges that requires a top-to-bottom rethinking of the entire engineering process; for example, in civil architecture, Filippo Brunelleschi’s solution in 1418 for how to construct the dome for the Cathedral of Florence required innovations in tools and building techniques, as well as rethinking the whole process of developing an architecture. All computer science faces a similar challenge; parallel programming is overdue for a fundamental rethinking of the process of designing software.

Programmers have been trying to craft parallel code for decades and learned a great deal about what works and what doesn’t work. Automatic parallelism doesn’t work. Compilers are great at low-level scheduling decisions but can’t discover new algorithms to exploit concurrency. Programmers in high-performance computing have shown that explicit technologies (such as MPI and OpenMP) can be made to work but too often require heroic effort untenable for most commercial software vendors.

To engineer high-quality parallel software, we plan to rearchitect the software through a “design pattern language.” As explored in his 1977 book, civil architect Christopher Alexander wrote that “design patterns” describe time-tested solutions to recurring problems within a well-defined context.
Parallel design patterns
The design pattern spaces

Four *design spaces*

- different concerns
- different “kind of programmers” involved
  - upper layers → application programmers
  - lower layers → system programmers

- Finding concurrency
- Algorithm structure
- Supporting structure
- Implementation mechanisms
Finding concurrency

Decomposition
○ Decomposition of problems into pieces that can be computed concurrently

Dependency analysis
○ support task grouping and dependency analysis

Design evaluation
○ aimed at supporting evaluation of alternatives

Used in an iterative process:
○ design → evaluate → redesign → ...
Decomposition

Task decomposition
How can a problem be decomposed into tasks that can execute concurrently?

Data decomposition
How can a problem’s data be decomposed into units that can be operated on relatively independently?

Forces
- Flexibility
- Efficiency
- Simplicity
Dependency analysis

Group tasks
How can the tasks make up a problem be grouped to simplify the job of managing dependencies?

Order tasks
Given a way of decomposing a problem into tasks and a way of collecting these tasks into logically related groups, how must these groups of tasks be ordered to satisfy constraints among tasks?

Data sharing
Given a data and task decomposition for a problem, how is data shared among the tasks?
Design evaluation pattern

Is the decomposition and dependency analysis so far good to move on to the next design space, or should the design be revisited?

- Suitability for the target platform (PE available, sharing support, coordination of PE activities, overheads)
- Design quality (flexibility, efficiency, simplicity)
- Preparation for the next phase of the design (regularity of the solution, synchronous/asynchronous interactions, task grouping)
Algorithmic structure

Organize by task
◦ when execution by tasks is the best organizing principle

Organize by data decomposition
◦ when main source of parallelisms is data

Organize by flow analysis
◦ flow of data imposing ordering on (groups of) tasks
Organize by task

Task Parallelism
When the problem is best decomposed into a collection of tasks that can execute concurrently, how can this concurrency be exploited efficiently? → dependency analysis, scheduling, ...

Divide & conquer
Suppose the problem is formulated using the sequential divide & conquer strategy. How can the potential concurrency be exploited? → dependency analysis, communication costs, ...
Organize by data decomposition

Geometric decomposition
How can an algorithm be organized around a data structure that has been decomposed into concurrently updatable “chunks”? 

Recursive data
Suppose the problem involves an operation on a recursive data structure (such as a list, tree or graph) that appears to require sequential processing. How can operations on these data structures be performed in parallel?
Organize by flow of data

Pipeline
Suppose that the overall computation involves performing a calculation on many sets of data, where the calculation can be viewed in terms of data flowing through a sequence of stages. How can potential concurrency be exploited?

Event based coordination
Suppose the application can be decomposed into groups of semi-independent tasks interacting in an irregular fashion. The interaction is determined by the flow of data between them which implies ordering constraints between the tasks. How can these tasks and their interaction be implemented so they can execute concurrently?
Design spaces

Design 1-2

Implement 3-4

Finding concurrency
Algorithm structure
Supporting structure
Implementation mechanisms
Supporting structures

Two main blocks:
- Program structures → approaches for structuring source code
- Data structures → data dependency management

Forces:
- Clarity of abstraction
- Scalability
- Efficiency
- Maintainability
- Environmental affinity
- Sequential equivalence
Supporting structures

Program Structures
- SPMD
- Master/worker
- Loop parallelism
- Fork/Join

Data Structures
- Shared data
- Shared queue
- Distributed array
Program structures (1)

SPMD

The interactions between the various UEs cause most of the problems when writing correct and efficient parallel programs. How can programmers structure their parallel programs to make these interactions more manageable and easier to integrate with the core computations?

Master/Worker

How should a program be organized when the design is dominated by the need to dynamically balance the work on a set of tasks among the UEs?
Program structures (2)

Loop parallelism

Given a serial program whose runtime is dominated by a set of computationally intensive loops, how can it be translated into a parallel program?

Fork/join

In some programs the number of concurrent tasks varies as the program executes, and the way these tasks are related prevents the use of simple control structures such as parallel loops. How can a parallel program be constructed around such complicated sets of dynamic tasks?
Data structures

Shared data

How does one explicitly manage shared data inside a set of concurrent tasks?

Shared queue

How can concurrently-executing UEs safely share a queue data structure?

Distributed array

Arrays often need to be partitioned between multiple UEs. How can we do this so the resulting program is both readable and efficient?
Implementation mechanisms

Directly related to the target architecture:

◦ to provide mechanisms suitable to create a set of concurrent activities (UE Units of Execution)
  ◦ → threads, processes (creation, destruction)

◦ to support interactions among the UEs
  ◦ → locks, mutexes, semaphores, memory fences, barriers, monitors, ...

◦ to support data exchange among the UEs
  ◦ → communication channels, queues, shared memory, collective operations (broadcasts, multicast, barrier, reduce) ...
Patterns & industry

Intel® Threading Building Blocks
Design Patterns

Document Number 323512-003US
World Wide Web: http://www.intel.com

Contents
1 Introduction
2 Agglomeration
3 Elementwise
4 Odd-Even Communication
5 Wavefront
6 Reduction
7 Divide and Conquer
8 GUI Thread
9 Non-Preemptive Priorities
10 Local Serializer
11 Fenced Data Transfer
12 Lazy Initialization
13 Reference Counting
14 Compare and Swap Loop
General References

https://software.intel.com/sites/default/files/m/4/8/1/e/e/33963-Design_Patterns.pdf
Intel TBB patterns

A design pattern description is much more than a rote coding recipe. The description of each pattern has the following format:

- **Problem** – describes the problem to be solved.
- **Context** – describes contexts in which the problem arises.
- **Forces** – considerations that drive use of the pattern.
- **Solution** – describes how to implement the pattern.
- **Example** – presents an example implementation.
# Intel TBB patterns: D&C

## 7 Divide and Conquer

### Problem

Parallelize a divide and conquer algorithm.

### Context

Divide and conquer is widely used in serial algorithms. Common examples are quicksort and mergesort.

### Forces

- Problem can be transformed into subproblems that can be solved independently.
- Splitting problem or merging solutions is relatively cheap compared to cost of solving the subproblems.
TBB D&C

Solution

There are several ways to implement divide and conquer in Intel® Threading Building Blocks (Intel® TBB). The best choice depends upon circumstances.

- If division always yields the same number of subproblems, use recursion and `tbb::parallel_invoke`.
- If the number of subproblems varies, use recursion and `tbb::task_group`.
- If ultimate efficiency and scalability is important, use `tbb::task` and continuation passing style.

```cpp
void ParallelQuicksort( T* begin, T* end ) {
    if( end-begin>=500 ) {
        using namespace std;
        T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
        swap( *begin, mid[-1] );
        tbb::parallel_invoke( [=]{ParallelQuicksort( begin, mid-1 );},
                              [=]{ParallelQuicksort( mid, end );} );
    } else {
        SerialQuicksort( begin, end );
    }
}
```
13 Reference Counting

Problem

Destroy an object when it will no longer be used.

Context

Often it is desirable to destroy an object when it is known that it will not be used in the future. Reference counting is a common serial solution that extends to parallel programming if done carefully.

Forces

- If there are cycles of references, basic reference counting is insufficient unless the cycle is explicitly broken.
- Atomic counting is relatively expensive in hardware.

```cpp
template<typename T>
class counted {
  tbb::atomic<int> my_count;
  T value;

public:
  // Construct object with a single reference to it.
  counted() {my_count=1;}

  // Add reference
  void add_ref() {++my_count;}

  // Remove reference. Return true if it was the last reference.
  bool remove_ref() {return --my_count==0;}

  // Get reference to underlying object
  T& get() {
    assert(my_count>0);
    return my_value;
  }
```
Parallel Patterns Library (PPL)

The Parallel Patterns Library (PPL) provides an imperative programming model that promotes scalability and ease-of-use for developing concurrent applications. The PPL builds on the scheduling and resource management components of the Concurrency Runtime. It raises the level of abstraction between your application code and the underlying threading mechanism by providing generic, type-safe algorithms and containers that act on data in parallel. The PPL also lets you develop applications that scale by providing alternatives to shared state.

The `task` class and related types that are defined in `ppltasks.h` are portable across platforms. The parallel algorithms and containers are not portable.

The PPL provides the following features:

- **Task Parallelism**: a mechanism to execute several work items (tasks) in parallel
- **Parallel algorithms**: generic algorithms that act on collections of data in parallel
- **Parallel containers and objects**: generic container types that provide safe concurrent access to their elements
Sample patterns

Parallel for

```cpp
// Use the for_each algorithm to compute the results serially.
elapsed = time_call([&]
    {
        for_each (begin(a), end(a), [&](int n) {
            results1.push_back(make_tuple(n, fibonacci(n)));
        });
    });
    wcout << L"serial time: " << elapsed << L" ms" << endl;

// Use the parallel_for_each algorithm to perform the same task.
elapsed = time_call([&]
    {
        parallel_for_each (begin(a), end(a), [&](int n) {
            results2.push_back(make_tuple(n, fibonacci(n)));
        });
        // Because parallel_for_each acts concurrently, the results do not
        // have a pre-determined order. Sort the concurrent_vector object
        // so that the results match the serial version.
        sort(begin(results2), end(results2));
    });
    wcout << L"parallel time: " << elapsed << L" ms" << endl << endl;
```
// Call parallel_for in the context of a cancellation token to search for the element.
cancellation_token_source cts;
run_with_cancellation_token([&count, what, &a, &position, &cts]()
{
    parallel_for(std::size_t(0), count, [&what, &a, &position, &cts](int n) {
        if (a[n] == what)
        {
            // Set the return value and cancel the remaining tasks.
            position = n;
            cts.cancel();
        }
    });
    }, cts.get_token());

return position;
}
Parallel for (vectorization)

Even more important/efficient than coarse grain parallelization, in most cases

```c
void loop_test(int u) {
    #pragma loop(hint_parallel(8))
    for (int i=0; i<u; ++i)
        A[i] = B[i] * C[i];
}
```
Sample patterns

Parallel invoke (fork/join)

```cpp
wcout << L"Running parallel version...";
elapsed = time_call([&] {
    parallel_invoke(
        [&] { common_words = find_common_words(words, 5, 9); },
        [&] { longest_sequence = find_longest_sequence(words); },
        [&] { palindromes = find_palindromes(words, 5); }
    );
});
```

Mapreduce

```cpp
// The Map operation
parallel_transform(begin(v), end(v), begin(map), MapFunc());

// The Reduce operation
unordered_map<wstring, size_t> result = parallel_reduce(
    begin(map), end(map), unordered_map<wstring, size_t>(), ReduceFunc());
```
Sample application workflow

Processing a stream of images

- The problem
  - Stream of images, available at different times
  - Each to be filtered with 2 different filters
  - truecolor (normalize colors)
  - sharpening

- Finding concurrency
- Algorithm structure
- Supporting structures
- Implementation mechanisms
Finding concurrency

Processing a stream of images

The problem

Finding concurrency

Algorithm structure

Supporting structures

Implementation mechanisms

Task decomposition: each input image is an independent task
Data decomposition: each image decomposed in sub images: each sub image processed independently
Task ordering: results should respect the input task ordering

Suitable to target different architectures
Configurable parallelism degrees
Task parallelism
Data parallelism (per task)

Design quality: structured
Preparation for the next phase: regular decomposition
Algorithm structure

Processing a stream of images

- Algorithm structure
  - Organize by task (process each image)
  - Organize by data decomposition
  - Organize by flow of data
  - Pipeline
  - Geometric decomposition
    - processing parts of image in parallel
    - filter (true color)
    - filter (sharpening)
  - task parallelism
    - processing each image as an independent task

- The problem
- Finding concurrency

- Supporting structures

- Implementation mechanisms
Supporting structure

Processing a stream of images

- The problem
- Finding concurrency
- Algorithm structure
- Supporting structures

- Program structures
- Data structures
- Implementation mechanisms

- Master/worker: to implement task parallelism
- Each worker is processing a single image

- SPMD: to implement each worker
- Alternatively: loop parallelism

- Shared queue: to support stream of images
- Distributed array: to support SPMD single image computation
Implementation mechanisms

- The problem
- Finding concurrency
- Algorithm structure
- Supporting structures

Processing a stream of images

- Implementation mechanisms
  - Multicore
  - Synchronization
  - Communications
  - UE management
  - Locks
  - Mutexes
  - Through shared memory
  - Processes
  - UE management
  - Distributed lock structures
  - Socket messages
  - Communication
  - Sockets
Workflow

Design (concurrency+algorithm)

Implement

Code

Once per application