Parallel design patterns

Marco Danelutto
Dept. of Computer Science, University of Pisa, Italy

May 2011, Pisa
# Contents

1. Introduction
2. Parallel design patterns
3. Finding concurrency design space
4. Algorithm Structure design space
5. Supporting structures design space
6. Implementation mechanisms design space
7. Conclusions
Parallel computing

The problem

- Solve a problem using $n_w$ processing resources
- Obtaining a (close to) $n_w$ speedup with respect to time spent sequentially
Parallel computing

The problem

- Solve a problem using \( n_w \) processing resources
- Obtaining a (close to) \( n_w \) speedup with respect to time spent sequentially

\[
s(n) = \frac{T_{seq}}{T_{par}(n)} \quad \text{(speedup)}
\]

\[
\epsilon(n) = \frac{T_{id}}{T_{par}(n)} = \frac{T_{seq}}{n \times T_{par}(n)} \quad \text{(efficiency)}
\]

\[
s(n) = \frac{1}{f + \frac{1-f}{n}} \quad \text{(Amdhal law)}
\]
The problems

Find potentially concurrent activities
- alternative decompositions
- with possibly radically differences
The problems

Find potentially concurrent activities
- alternative decompositions
- with possibly radically differences

Parallelism exploitation
- program activities (threads, processes)
- program interactions (communications, synchronizations)

→ overhead
Structured parallel programming

Parallel design patterns
M. Danelutto

Introduction

Parallel design patterns
Finding concurrency design space
Algorithm Structure design space
Supporting structures design space
Implementation mechanisms design space
Conclusions
Structured parallel programming

**Algorithmic skeletons**

- Cole 1988 → common, parametric, reusable parallelism exploitation pattern
- languages & libraries since ’90 (P3L, Skil, eSkel, ASSIST, Muesli, SkeTo, Mallba, Muskel, Skipper, BS, ...)
- high level parallel abstractions (parallel programming community)
  - hiding most of the technicalities related to parallelism exploitation
  - directly exposed to application programmers
Structured parallel programming

Algorithmic skeletons

- Cole 1988 → common, parametric, reusable parallelism exploitation pattern
- languages & libraries since ’90 (P3L, Skil, eSkel, ASSIST, Muesli, SkeTo, Mallba, Muskel, Skipper, BS, ...)
- high level parallel abstractions (parallel programming community)
  - hiding most of the technicalities related to parallelism exploitation
  - directly exposed to application programmers

Parallel design patterns

- design patterns à la Gamma book
  - name, problem, solution, use cases, etc.
- concurrency, algorithms, implementation, mechanisms
Concept evolution

Parallelism

- parallelism exploitation patterns shared among applications
- separation of concerns:
  - system programmers $\rightarrow$ efficient implementation of parallel patterns
  - application programmers $\rightarrow$ application specific details
Parallelism

- parallelism exploitation patterns shared among applications
- separation of concerns:
  - system programmers $\rightarrow$ efficient implementation of parallel patterns
  - application programmers $\rightarrow$ application specific details

New architectures

- *Heterogeneous* in Hw & Sw
- *Multicore* NUMA, cache coherent architectures
Parallelism

- parallelism exploitation patterns shared among applications
- separation of concerns:
  - system programmers $\rightarrow$ efficient implementation of parallel patterns
  - application programmers $\rightarrow$ application specific details

New architectures

- *Heterogeneous* in Hw & Sw
- *Multicore* NUMA, cache coherent architectures

Further non functional concerns

- security, fault tolerance, power management, …
Parallel design patterns

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
Parallel design patterns

M. Danelutto

Introduction

Parallel design patterns

Finding concurrency design space

Algorithm Structure design space

Supporting structures design space

Implementation mechanisms design space

Conclusions
The pattern framework

Four pattern classes:

1. Finding concurrency
2. Algorithm structure
3. Supporting structure
4. Implementation mechanisms

These are “design spaces”

- different concerns
- different “kind of programmers” involved
  - upper layers $\rightarrow$ application programmers
  - lower layers $\rightarrow$ system programmers
Finding concurrency design space

Three main blocks

1. **Decomposition**
   - Decomposition of problems into pieces that can be computed concurrently

2. **Dependency analysis**
   - Support task grouping and dependency analysis

3. **Design evaluation**
   - Aimed at supporting evaluation of alternatives

Used in an iterative process:
- design → evaluate → redesign → …
Finding concurrency design space

Figure 3.1: Overview of the Finding Concurrency design space and its place in the pattern language
Decomposition patterns

**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 patterns

### 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 constrains 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?

Forces:

- 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)
Algorithm structure

Three main blocks

1. Organize by task
   → when execution by tasks is the best organizing principle

2. Organize by data decomposition
   → when main source of parallelisms is data

3. Organize by flow analysis
   → flow of data imposing ordering on (groups of) tasks
Algorithm structure

Figure 4.1: Overview of the Algorithm Structure design space and its place in the pattern language
**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, ...
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?
Supporting structures

Two main blocks:

1. Program structures
   → approaches for structuring source code
2. Data structures
   → data dependency management

Forces:
- Clarity of abstraction
- Scalability
- Efficiency
- Maintainability
- Environmental affinity
- Sequential equivalence
Figure 5.1: Overview of the Supporting Structures design space and its place in the pattern language
SPMD (Single Program Multiple Data)

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?
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:

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

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

3. to support data exchange among the UEs
   → communication channels, queues, shared memory, collective operations (broadcasts, multicast, barrier, reduce) ...
Figure 6.1: Overview of the Implementation Mechanisms design space and its place in the pattern language.
Design patterns vs. algorithmic skeletons

Sw engineering vs. HPC community

**SwEng** Focus on efficiency of the programming process

**HPC** Focus on performance (and programmer productivity)
Design patterns vs. algorithmic skeletons

Sw engineering vs. HPC community

**SwEng** Focus on efficiency of the programming process

**HPC** Focus on performance
(and programmer productivity)

Then:

**Design patterns** “recipes” to be implemented in order to get a working program

**Algorithmic skeletons** predefined program constructs
(language constructs, classes, library entries) implementing parallel patterns
Typical skeleton based program development

1. Figure out which skeleton (composition) models your problem
2. Instantiate skeletons
   - functional parameters (e.g. code, data types) & non functional ones (e.g parallelism degree)
3. Fine tune program performance
   - parameter sweeping, bottleneck analysis or skeleton restructuring
     → no parallel debugging, correctness guaranteed
     → no possibility to use parallel patterns not supported by the skeleton set
Muesli (H. Kuchen, Munster Univ. D)

- C++ class library
- stream parallel skeletons
  - Pipeline, Farm, Branch&Bound, Divide&Conquer (Atomic, Filter, Final, Initial)
- data parallel skeletons
  - DistributedXXX ($XXX \in \{\text{Array, Matrix, SparseMatrix}\}$)
    + fold, map, scan, zip
- target architecture: C++ (MPI + OpenMP)
- all communication, synchronization, shared access problems solved in the skeleton implementation
- program declaration separated from execution
Parallel design patterns

Principles

- Architecting parallel software with design patterns, not just parallel programming languages
- Split productivity and efficiency layers, not just a single general-purpose layer
- Generating code with search-based autotuners, not compilers
- Synthesis with sketching
- Verification and testing, not one or the other
- ...
TBB (Thread Building Block library by Intel, 2005)

- C++ library
- currently version 3.0 (since late 2010)
- base building blocks for parallel programming with thread
  - parallel loop, reduce, pipeline
  - tasks
  - parallel containers
  - mutexes
- since 3.0 → TBB Design patterns
  - Agglomeration, Elementwise, Odd-even communication, Wavefront, Reduction, Divide & Conquer, GUI thread, Non-preemptive priorities, Local serializer, Fenced data transfer, Lazy initialization, Reference counting, Compare-and-swap loop.
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.

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 \texttt{tbb::parallel\_invoke}.
- If the number of subproblems varies, use recursion and \texttt{tbb::task\_group}.
- If ultimate efficiency and scalability is important, use \texttt{tbb::task} and continuation passing style.
Sample TBB pattern: D&C

Example

The number of subsorts is fixed at two, so `tbb::parallel_invoke` provides a simple way to parallelize it. The parallel code is shown below:

```cpp
void ParallelQuicksort( T* begin, T* end ) {
    if( end-begin>1 ) {
        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 );} );
    }
}
```
Parallel design patterns
M. Danelutto

Introduction

Parallel design patterns

Finding concurrency design space

Algorithm Structure design space

Supporting structures design space

Implementation mechanisms design space

Conclusions

Sample APPL

Processing a stream of images

Stream of images, available at different times

The problem

Each to be filtered with 2 different filters

truecolor (normalize colors)

sharpening

Finding concurrency

Algorithm structure

Supporting structures

Implementation mechanisms
Sample APPL

Parallel design patterns
M. Danelutto

Introduction
Parallel design patterns
Finding concurrency design space
Algorithm structure design space
Supporting structures design space
Implementation mechanisms design space
Conclusions

The problem
Finding concurrency
Algorithm structure

Supporting structures

Processing a stream of images

Implementation mechanisms

Program structures
SPMD to implement task parallelism
Master/worker each worker is processing a single image
Shared queue to support stream of images
Distributed array to support SPMD single image computation

Data structures

Multicore
Synchronization
Communications
UE management
COW/NOW
Synchronization
Communication
UE management
Processes
distributed lock structures
socket messages
Sockets

Threads
Lock
Mutexes
Through shared memory

Conclusions

Become an expert in parallel computing
by studying and applying parallel design patterns !!!
Conclusions

Become an expert in parallel computing
by studying and applying parallel design patterns !!!

You will need it
also to program iPhone applications!