Parallel Programming Course
Methodology

Paul Guermonprez
www.Intel-Software-Academic-Program.com
paul.guermonprez@intel.com
Intel Software
2012-03-14
The wrong methodology

Imagine a algorithm thinking in serial
Code this serial prototype
Parallelize without collecting data
Notice a low scalability and parallel bugs
Try to optimize the parallel code

It doesn't work

Let's see why ...
Amdhal's Law
Amdahl's Law

Imagine you have a software with a large part (80% of the execution time) you can parallelize.

If your code is perfectly parallelized:

1 core: reference time $t$

2 cores: execution time: $20% + 80%/2 = 0.6t$
speedup: $1.66x$ for 2 cores

4 cores: execution time: $20% + 80%/4 = 0.4t$
speedup: $2.5x$ for 4 cores

$\infty$ cores: execution time: $20% + 80%/$$\infty = 0.2t$
speedup: $5x$ for $\infty$ cores
Amdahl's Law

Because you try to parallelize 80% of your code you think it will scale.

In fact, the maximum speedup is 5x on an infinite number of cores.

Before you start coding, evaluate the maximum scalability of your algorithm with Amdhal's Law and ask yourself:

Is it enough?
Do you need to redesign your algorithm?
Serial vs Parallel Performance
Serial Performance

Serial performance bottlenecks usually come from the sub optimal or chaotic utilization of hardware resources:

● memory random accesses,
● branch coding,
● sub optimal compilation, …

(see the course on serial performance)
Parallel Performance

Parallel performance bottlenecks usually come from the parallel design of your algorithm or parallel coding:

- excessive synchronization,
- load imbalance ...

(see the performance part of this course)
Serial vs Parallel

But you want to solve your serial problems before your parallel problems:

- serial performance problems usually get a lot worse when run in parallel,
- the design of your parallel software depends on performance data collected in serial,
- solving serial and parallel problems at the same time is too complex.

Conclusion: solve your serial performance problems before you start parallelizing
Parallel Programming Methodology
1. Design a parallel algorithm

Design a parallel algorithm:

What is the maximum theoretical scalability of my algorithm?

Is my algorithm still interesting in a few years, when many-core machines will be standard?

Nowadays, some serial algorithms are not meant to be implemented because they'll never run fast enough.
2. Code serial and optimize serial performance

Code and optimize serial performance:
Thoroughly debug your code, optimize for serial performance, use performance libraries.

Collect performance data (oprofile, Intel VTune):
• see how they match your algorithm predictions,
• prepare parallelization of your code,
• evaluate if parallelization is still worth it.

Collect detailed performance data.
3. Introduce parallelism

Introduce parallelism:

Pick the right technology for your problem. (*OpenMP, explicit threading, TBB, OpenCL, performance libraries ...*)

Pick the right place in your code to introduce it.

Predict maximum scalability based on serial performance data.
4. Debug parallel bugs

Debug parallel bugs:

A perfectly working serial code can give wrong results when run in parallel if the parallelism was not introduced correctly.

A serial debug tool won't help.

See tools like Intel Parallel Inspector and/or code inspection.
5. Optimize parallel performance

Optimize parallel performance:
When your serial performance problems are solved, you'll have a clear view of your parallel performance problems.
System tools, logging and Intel Parallel Amplifier can help you collect the right information.
Decomposition
Decomposition

To design a parallel algorithm or parallelize an existing code, you first have to understand if you have **dependencies**.

If data can be processed or tasks executed **independently**, you can process them in **parallel**. You have two types of decomposition:

- **data decomposition**
- **task (functional) decomposition**
Example

Suppose you want to apply two filters on a batch of images and save them in jpg and png formats.

Can you process different images in parallel?
Can you process different pixels in parallel?
Can you open/save files, apply filters in parallel?
Data decomposition
Parallel treatment of files

In our case, data decomposition means you'll try to execute in parallel the treatment of different images or different pixels.

Images are independent from each other, it's perfect and easy to implement. But if you have a low number of images to treat and a large number of cores available, you won't use them all.

A lot of images and a dual-core : easy to use all cores efficiently.

Few images and a many-core (50+) : data decomposition by file only won't be enough to use all cores.
Data decomposition
Parallel treatment of pixels

For pixels, it's **easy** to do for a basic luminosity filter as **each pixel is processed independently** from the others. But the blur filter requires information from neighboring pixels.

It can be done in parallel with some approximations or a lot of communication between threads (synchronization), impacting scalability.

Luminosity adjustment can be done independently on each pixel. **Easy to parallelize efficiently.**

Blur filter requires information from other pixels. Requires heavy **synchronization** OR flexibility regarding the **result.**
Another way to solve our problem is to execute different operations on the same data in parallel.

Saving as .png and saving as .jpg are totally independent (and cpu intensive) operations. It can easily be parallelized.

But luminosity and blur can't be done in parallel on the same image independently.
Data and task decompositions

For our problem, each decomposition has pros and cons depending on the number of files to process, precision required for the blur filter, number of cores, developer skills and time ...

A good real life solution would be to implement different levels of nested parallelism mixing data and task decomposition.
Parallel Performance
Parallel without performance is not enough

Adding some parallelism to your software is often not enough to take advantage of many-core processors with efficiency and flexibility.

Typical parallel performance issues:

- Parallel overhead
- Synchronization
- Load imbalance
- Granularity
Parallel overhead

All forms of parallelism bring a small **overhead**: loading a library, launching threads, scheduling ... 

Solutions:

- **Monitor** software and OS resources (memory usage, context switches, number of threads ...)

- Remember that some parallel framework are light, designed for single computers and small task while others are very heavy, designed for large clusters.
Synchronization

Some algorithms (like the blur filter) require **communications, synchronizations** between parallel executions, often blocking execution.

Solutions :

- Is another algorithm possible?
- Do you accept a slightly different result?
- Adapt your code to work with local variables?
- **Optimize** synchronization (→ OpenMP course)
Load imbalance

Uneven distribution of chunks of data over the worker threads is a typical performance problem.

Solutions:

• Insert parallelism deeper in the call stack (pixels instead of files in our example)

• Propose a new usage model for your software, easier to parallelize (in our example, process files in batch more easily)

• Adapt the settings of your parallel framework (→ OpenMP algorithms and chunk size)
Granularity

You have granularity problems if the chunks of data distributed to your threads are too big or too small. Too big they may cause a load imbalance. Too small a parallel overhead.

Solutions:

- Partition your data with flexibility. Hardware, data and usage models change rapidly.
- Adapt the distribution algorithm and chunk size in your parallel framework.
Conclusion
Think parallel, think manycore

To take advantage of multi-core processors today and many-core tomorrow, you need to think parallel:

- Design and select parallel algorithms.
- Be open minded:
  Can you accept a slightly different output? Propose another usage model?
- Do not code for 2 or 4 cores. Think many-core and remember Amdahl's law.
License Creative Commons - By 2.5

You are free:
  to Share — to copy, distribute and transmit the work
  to Remix — to adapt the work
  to make commercial use of the work

Under the following conditions:
  Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

With the understanding that:
  Waiver — Any of the above conditions can be waived if you get permission from the copyright holder.
  Public Domain — Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.
  Other Rights — In no way are any of the following rights affected by the license: