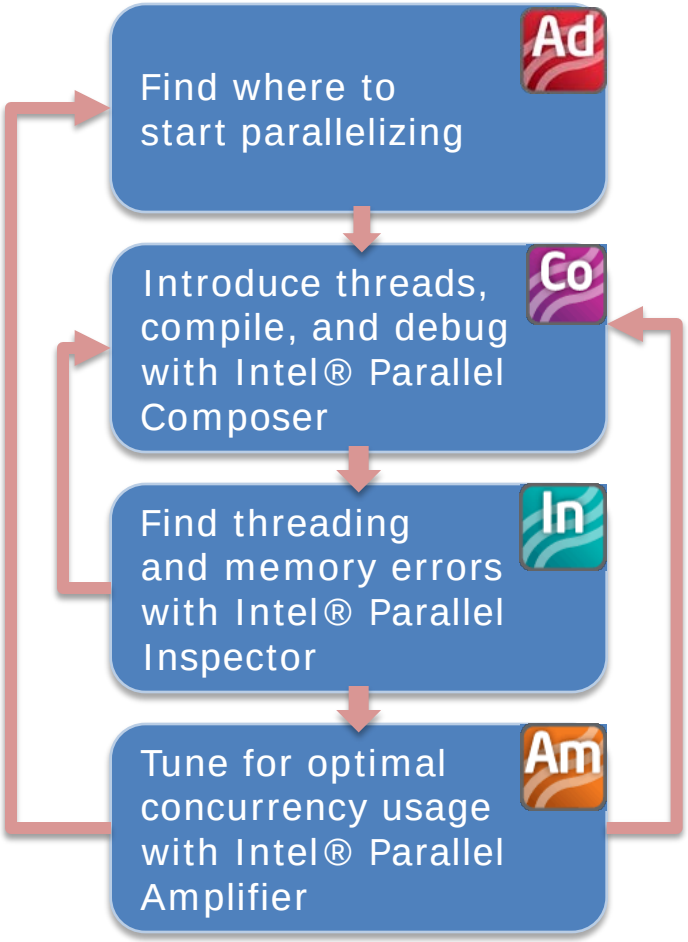
The Key to Scaling Applications for Multicore

Mark Davis / Paul Petersen
Developer Products Division
Intel Corporation
May 05, 2009

Outline

- Introduction
- Intel® Parallel Advisor Lite Workflow
- Choose Tasks
  - Parallel Scaling – Amdahl's Law
  - Hot Spot Profiling
  - Task Patterns
- Model Performance
- Model Correctness
- Find, Understand, and Fix Data Sharing Problems
- Add Parallelism
- Debug, Verify, and Tune
- Summary

Four Steps to a Parallel Application. Add-ins to Microsoft* Visual Studio.



DESIGN – Intel® Parallel Advisor Lite
Gain insight on where parallelism will most benefit existing source code – usually begins with a "hotspot"

CODE, DEBUG – Intel® Parallel Composer
Develop effective applications with a C/C++ compiler and comprehensive threaded libraries and API's, and a parallel debugger

VERIFY – Intel® Parallel Inspector
Help ensure application reliability with proactive parallel memory and threading error checking

TUNE – Intel® Parallel Amplifier
Enhance applications with an intuitive performance analyzer and tuner

Intel Parallel Studio is now available.
Get free eval software:
intel.com/software/prodcuts/eval

www.intel.com/go/parallel

Introduction

- The drive toward parallel programs:
  - Now chips have more cores, not more speed
  - You must add parallelism to your program to achieve increased performance

- *Scaling to Multicore* using **Intel® Parallel Advisor Lite**: a Methodology and set of tools that help you find and add scalable parallelism.

- Parallel Advisor Lite on whatif.intel.com in Q2; requires Intel® Parallel Studio.

whatif.intel.com

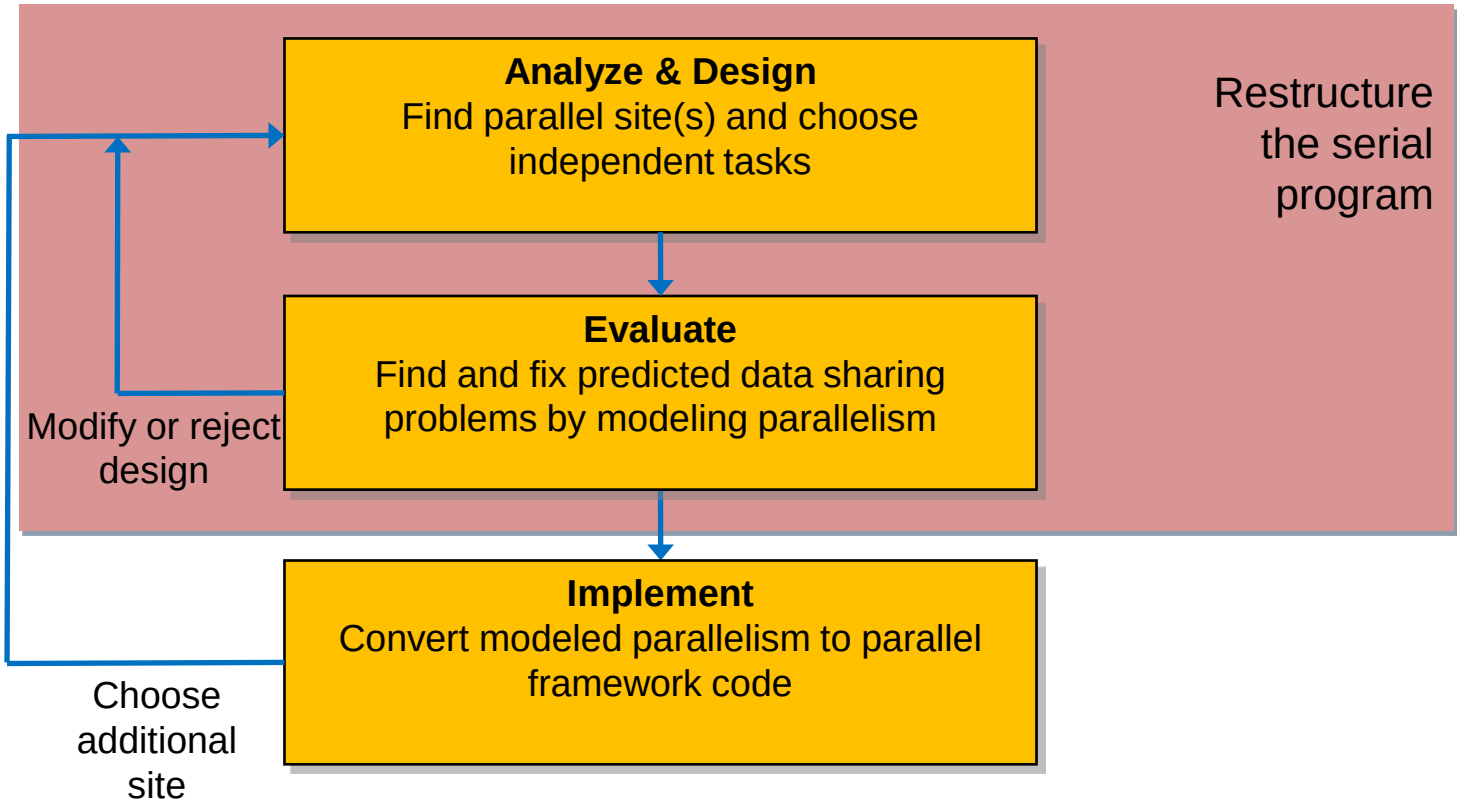
The *Sequential* Approach

- Work on the **sequential** (or "**serial**") program to gain a deeper understanding of its potential **parallel** behavior. You *model* its parallel characteristics.

- Perform semantics-preserving refactoring to clean up the serial code in preparation for parallelization.
  - Experimenting with and modifying a serial program is easier and more reliable than using a parallel version!

- Final step: you express the parallelism using a high-level parallel framework like Intel® Threading Building Blocks (TBB) or OpenMP*, or low-level threading APIs.

High Level Workflow



Workflow in more detail

1. **Choose Tasks** - using your knowledge of the program and *profiling* information.
2. **Model Performance** - to confirm that the task selection will generate benefits.
3. **Model Correctness** - will there be data sharing problems when your program runs in parallel?
4. **Fix Problems** - and retest, repeating until no more problems are identified.
5. **Add Parallelism**
6. **Debug, Verify, and Tune**

Choose Tasks

- Find a parallel site and partition its code into tasks that will run in parallel with each other.

- What criteria should be used for this selection?

Limits to Parallel Scalability

Suppose you want to bake a cake. The steps in the recipe are

1. Gather ingredients.
2. Mix ingredients and pour into 2 cake pans.
3. Bake at 350 degrees for 30 minutes.

You might be able to use 2 bakers in parallel to speed up steps 1 and 2, but the cake must still remain in the oven for 30 minutes – the baking step is not amenable to parallelism.

Parallel Scaling – Amdahl's Law

- Amdahl's Law: parallel speed-up achievable is limited by the time spent in the portion of the program that is ***not*** parallel, e.g., the time in the oven.

- For example, if the parallel part took 75% of the time when it was serial, and is now run on an infinite number of cores so its elapsed time is zero, the remaining serial code will still take 25% of the original time, and the program will only speed up by a factor of 4 (100% / 25%).

Profile to find the Hot Spots

- For best ***scaling*** find regions that consume the most time – *hot spots*.

- Don't guess – *Profile* to find hot spots. You need:
  - How much time?
  - Where?
  - Who calls them?

Are the Hot Spots the Only Candidates for Parallel Sites?

- 1 cook is making dinner on the stove in 4 pots. Profiling: *stir_a_pot()* function is the hot spot.

- Hire 3 more cooks and parallelize the stirring loop in the body of *stir_a_pot*. But when a pot needs to be stirred, all 4 cooks try to stir the **same pot**, at the same time, probably getting in each other's way. Result: no speed-up!

What you really want is for each cook to get their own pot, and the cook stirs *that* pot when stirring is needed. Thus parallelization is applied at a **higher level** than the hot spot.

Are the Hot Spots the Only Candidates? No!

- To optimize a **serial** program, the objective is to reduce the number of instructions => focus solely on hot spots.

- But, when **parallelizing** a program, the objective instead is to *distribute* those instructions over as many parallel tasks as possible – reducing the **elapsed** time.

  A hot spot may be a good parallel site, but locations higher up the call tree often are better candidates and provide better scalability!

Task Patterns: Data Parallelism …

- **Data Parallelism** pattern: for cases where an operation is repeatedly applied to items in a collection of data.

  Process each data item or subset of the data items in a separate task.

```
// parallel site begin
for (int i = 0; i != n; i++) {
    // task begin
    process(a[i]);
    // task end
}
// parallel site end
```

Data Parallelism tends to ***scale*** well, as long as you can increase the data set size to match the number of cores.

… and Task Parallelism

- **Task Parallelism** pattern: for cases where work is divided into several distinct activities.

Process each activity in a different task body.

```
initialize(data);
while (!done) {
   old_data = data;
   // parallel site begin
      // task begin
      display_on_screen(old_data);
      // task end

      // task begin
      update(data);
      // task end
   // parallel site end
}
```

Task Parallelism does not scale as well because the number of tasks tends to be limited.

Indicate Parallel Site and its Tasks

- Use the profile information and knowledge of the 2 kinds of *task patterns* to choose the tasks at the parallel site, and indicate their locations in your code.
- You now have a candidate parallel structure.
- Before looking for potential problems when the program is made parallel, do some rough *performance modeling* to increase confidence that your work will yield benefits and scalability.

Model Performance

In *Performance Modeling*, you address three questions:

1. **Amdahl's Law**: is the serial part going to overwhelm what can be gained by parallelizing here?
   Data: % of time in parallel site
2. **Decomposition**: is the granularity too fine, so that tasking overhead will negate the advantage of parallelism? (This is usually not an issue for parallel loops in TBB and OpenMP*, which automatically group multiple iterations into a single task.)
   Data: average time in each task > 10 microseconds ?
3. **Load balance**: is the granularity too coarse, so there is not enough work to keep all the cores equally busy?
   Data: average # of tasks per site too small?; high variability of task times?

Using the profile results, gather the execution times of the parallel site and its tasks' regions; also gather their execution counts. Use this data to answer the questions.

If the answers are unsatisfactory, use this information to drive a better selection of tasks.

Data Sharing Problems

- Serial program: the exact order of its operations are known.
- Parallel tasks: any operation in either task can execute before, after, or simultaneously with any operation in the other task.
- Errors can occur when parallel tasks access/modify the same memory location. These are called by several names: *data sharing problems, data conflicts,* or *data races*.

Model Correctness – Find Data Sharing Problems

There is a *Correctness Modeling* tool that works on a sequential program after the locations of parallel sites and tasks have been indicated.

- It runs the serial program, keeping track of site and task boundaries, synchronization operations, and data accesses.
- It then *models* how the parallel tasks would behave with respect to each other and to data shared between them.
- The tool calculates data sharing problems that might arise during parallel execution.

Understand and Fix Data Sharing Problems

Recall the case of the 4 cooks and their 4 pots…

- New information: they only have ***1 spoon***, because that was all that was needed for 1 cook (the original serial program). The spoon represents a shared "data" problem.

- There are three ways to characterize sharing, and to fix the problem.

Incidental Sharing

1. *Incidental sharing*: They do not need exactly the same spoon – they can each have their own spoon.

Solution:

In programming terms, the shared object can be *privatized* by using a copy for each task. You need to decide what initial value it receives, and what value (if any) to put into the shared object when the task completes.

Independent Updates

2. *Independent updates*: It does not matter what order they use the spoon, but for some reason it is not possible for each to have their own spoon – so they have to take turns. (For example, if it was an electric mixer instead of a spoon, it might be too expensive for each cook to have their own.)

Solution:

Synchronize access to the shared object. Use a *lock* or *mutex* so that only one task can reference the object at a time.

True Dependence between Tasks

3. *True dependence*: The shared data accesses in different tasks must occur in the original sequential order, usually because values are passed from one task to another. This eliminates (most of) the parallelism, because the tasks cannot execute at the same time – they must stay in order.

Solution:

Chose the tasks or parallel site differently, or restructure, e.g., combine the tasks that use the shared data.

True Dependence with Cake

- The 2 cake layers are cooked, they need frosting on them – can the job be parallelized with the 2 bakers?
  - The tops can be frosted in parallel
  - Unfortunately there's a true ordering dependency: they have to be stacked *before* the sides are frosted.

- Is there a better way?

Problem/Program Restructuring

- Original requirement: Birthday Dessert.
- Serial program was: bake and frost a cake.
- Satisfy the requirement with a variation: make cup cakes!
- ***Restructure*** for better parallelism:
  - Granularity has been improved: more small pieces (*data parallel*).
  - Cup cakes can be frosted in parallel, and scale to more than 2 bakers.
  - Amdahl is happier: cup cakes cook faster - reducing the serial portion of the process and yielding higher speed-up.

Test, and Rerun Correctness Modeling

After you have fixed the problems:

1. Incidental sharing -> privatize.
2. Independent updates -> synchronize: indicate synchronization locations.
3. True dependence -> restructure program.

- Run regular testing to check that the (serial) program still runs correctly.

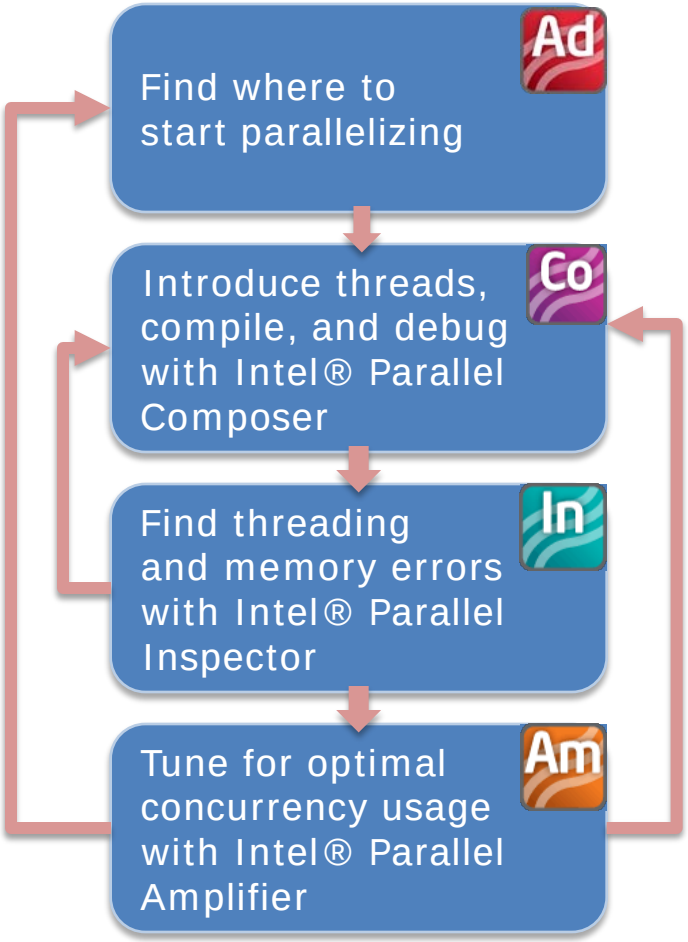
- Run Correctness Modeling again. Check that the problems are gone. It is also possible that new problems are discovered.

Choose Parallel Framework, and Add Parallel Constructs

After successful testing and rerunning Correctness Modeling,

- Choose a parallel framework like Intel® Threading Building Blocks or OpenMP* or a low-level threading API, and
- Replace the site/task/synch indications with parallel constructs.

Debug, Verify, Tune



DESIGN – Intel® Parallel Advisor Lite
Gain insight on where parallelism will most benefit existing source code – usually begins with a "hotspot"

CODE, DEBUG – Intel® Parallel Composer
Develop effective applications with a C/C++ compiler and comprehensive threaded libraries and API's, and a parallel debugger

VERIFY – Intel® Parallel Inspector
Help ensure application reliability with proactive parallel memory and threading error checking

TUNE – Intel® Parallel Amplifier
Enhance applications with an intuitive performance analyzer and tuner

Intel Parallel Studio is now available.
Get free eval software:
intel.com/software/prodcuts/eval

www.intel.com/go/parallel

Intel Software Products

Summary

- Intel® Parallel Advisor Lite has a methodology and set of tools to help find and add scalable parallelism.
- Stay with the sequential program for analysis and making code changes.
- Amdahl's Law guides selection of parallel sites and tasks – Performance Modeling confirms the choice.
- Correctness Modeling finds data sharing problems – fix before going parallel.
- Apply Intel® Parallel Studio when your program becomes parallel.

Upcoming webinars…

Live Date	Event	Title	Speaker
3/10/2009	Webinar	Go-Parallelism! Ease the Onramp for C/C++ Windows* Development	James Reinders
3/17/2009	Tech. Session	Solve Parallelism with Intel® Parallel Studio	Joe Wolf
3/24/2009	Webinar	Simplify Parallelism with Intel® Parallel Composer	Joe Wolf
3/31/2009	Tech. Session	Parallel Implementation Methods with Intel® Parallel Composer	Ganesh Rao
4/7/2009	Webinar	Debugging Parallel Code for Fast, Reliable Applications	Jay Desouza
4/14/2009	Tech. Session	Find Errors in Windows C++ Parallel Applications	Gerold Mueller or Robert Mueller
4/21/2009	Webinar	Easy Ways to Solve Parallel Performance Challenges	Gary Carleton
4/28/2009	Tech. Session	The Good, the Bad, and the Ugly: Improve Parallel Application Quality and Performance	Eric Moore
5/5/2009	Webinar	The Key to Scaling Applications for Multicore	Paul Petersen/Mark Davis
5/12/2009	Tech. Session	Identify and Address Threading Opportunities	Caroline Davidson
5/19/2009	Webinar	Image Processing: Stop Developing Code from Scratch	Walt Shands, Intel Corporation
5/26/009	Tech. Session	Simplifying Parallelism Implementation with Intel® Threading Building Blocks	Mike D'Mello
6/2/2009	Tech. Session	Static Analysis and Intel® C++ Compilers	Dmitry Putunin

Intel Parallel Studio is now available.
Get free eval software:
intel.com/software/prodcuts/eval

www.intel.com/go/parallel

These future products are in beta testing currently. Features and capabilities are subject to change without notice.

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101