Data-dependencies and Thread Interaction in Parallel Loops

Per Larsen, Sven Karlsson and Jan Madsen
{pl,ska,jm}@imm.dtu.dk

DTU Informatics
Department of Informatics and Mathematical Modeling

Outline

- Motivation
- Inter-task dependencies in parallelized stencil operations
- Proposed solution
- Empirical evaluation
- Wrap-up

Synthesizing Task Graphs from Source Code

functional prototype
C/C++ with
OpenMP directives

???

some form of
task graph

scheduled and
mapped code

Synthesizing Task Graphs from Source Code

functional prototype
C/C++ with
OpenMP directives+
new directive

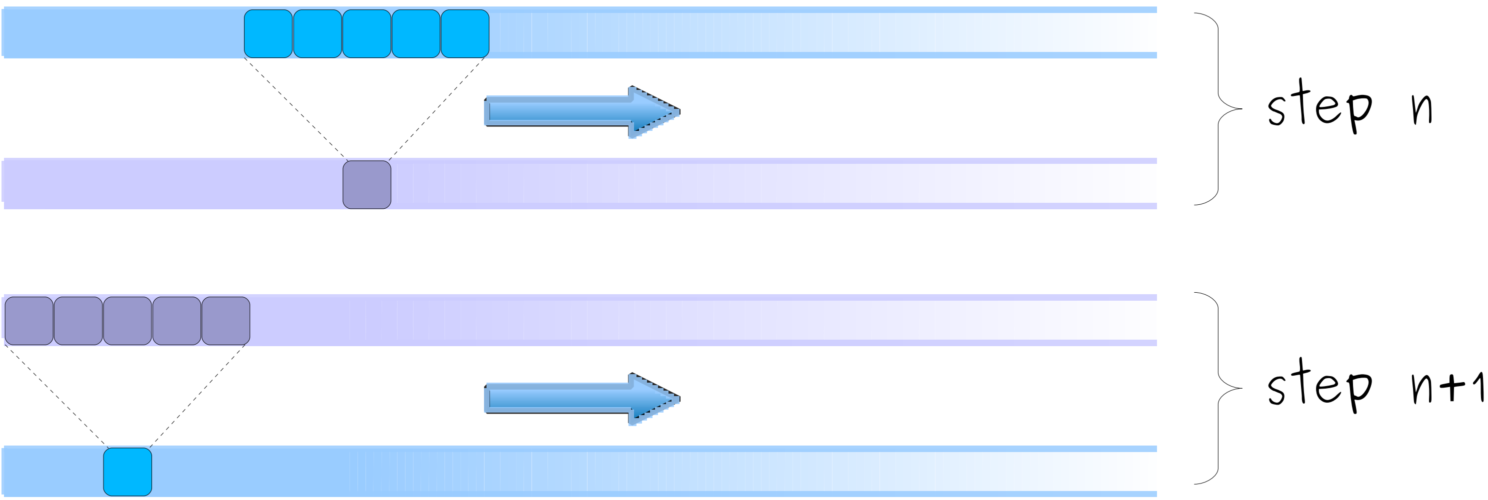
some form of
task graph

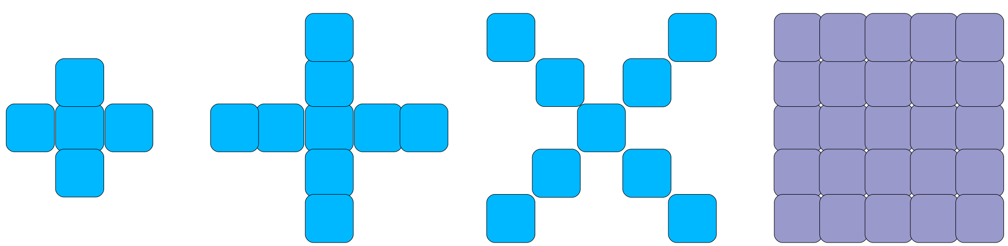
scheduled and
mapped code+
runtime checks

Why Add Directives?
Approaches Task Graph Synthesis

- **Program analysis:** correct results for **every** possible execution
  – Use simplifying assumptions to make problem tractable
  – produces over-approximations of inter-task dependencies
- **Simulation:** captures detailed performance behavior
  – Only accurate for a single program execution
  – How to combine results from multiple program executions?
- **Manual synthesis**: leverages human high-level understanding
  – prone to errors
  – does not scale with increasing
    - code churn
    - lines of source code

Parallel Stencil Operations

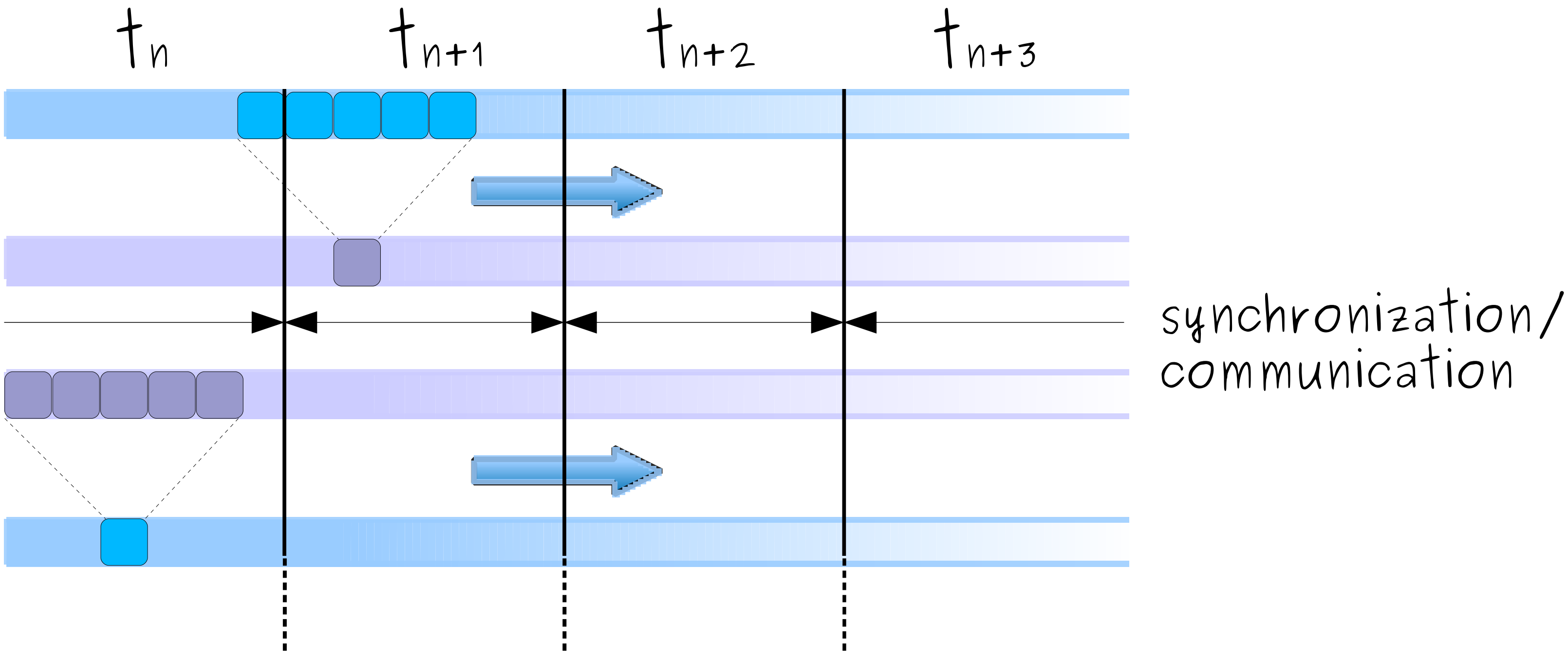




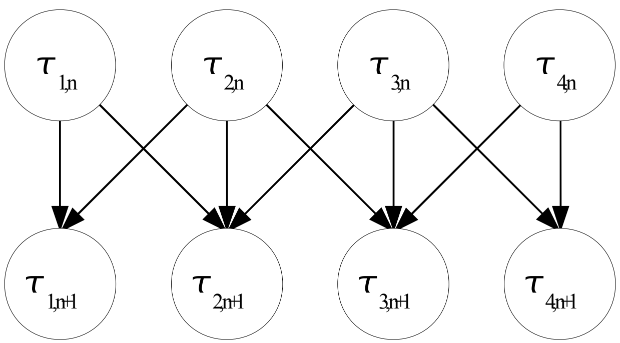
stencils can vary in size, shape and dimensions

they all result in loops with predictable array access patterns!

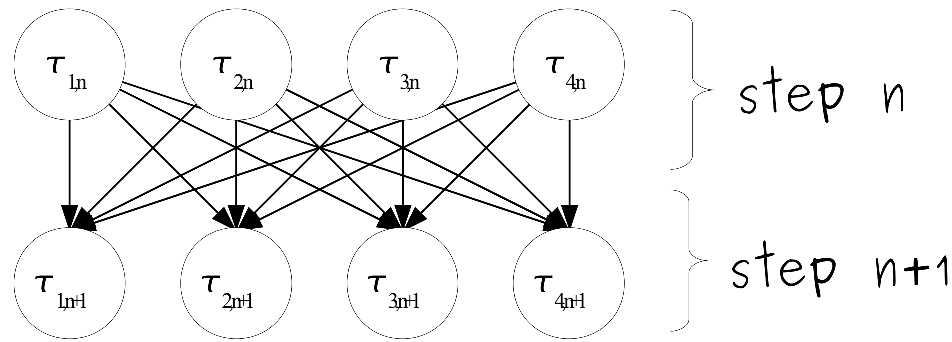
Parallel Stencil Operations



Comparing Actual Dependencies with Over-approximated Dependencies



1a) actual dependencies



1b) conservative approximation

major contributors to over-approximation:

- number of threads executing loop is unknown (by design)
- collapsing multi-dimensional arrays
- using pointer arithmetic
- pointer aliasing
- loop ranges unknown
- scheduling of loop iterations to threads unknown
- stencil size unknown

refactor!

use **restrict** kw

use directive!

The `taskshare` Directive

- Need to know how many tasks access each slice of an array to compute actual inter-task dependencies (cf. Fig. 1a)
  - number of tasks ts accessing slices with length at least s_min with a stencil of size k is bounded by:

$$ts \geq \lceil \frac{k}{s_{min}} \rceil \rightarrow s_{min} \geq \lceil \frac{k}{ts} \rceil \qquad (1)$$

- Idea: user annotates parallel loop with

```
#pragma taskshare(arr,ts)
```

meaning that at most ts tasks will access each slice when stencil size is k

  - minimal slice size s_min depends on number of threads executing loop, number of iterations, scheduling of iterations to threads, etc.
  - stencil k size may also be unknown at compile time
    1) Calculate s_min at *runtime*
    2) raise exception if Eq. (1) does *not* hold

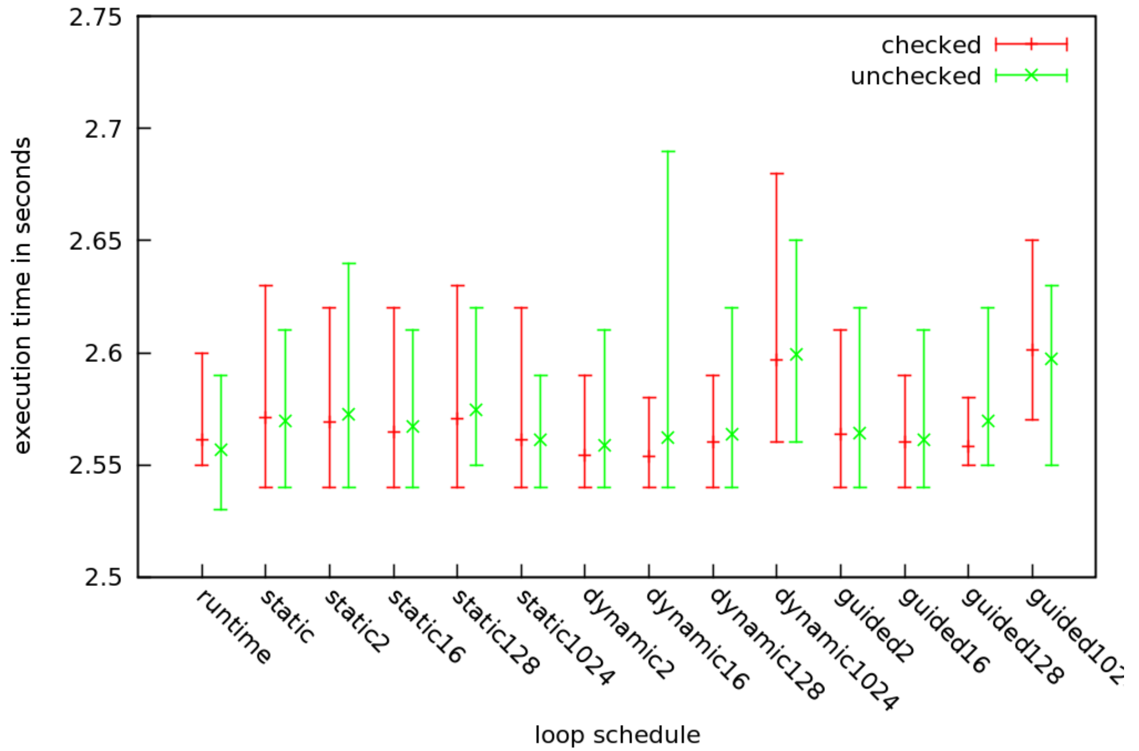
Instrumentation

- Used OpenMP directives for parallelization
- Wrote compiler* plug-in which inserts calls to check correctness of **`taskshare`** directives at runtime
  - OpenMP directives are translated into OpenMP runtime calls by compiler
  - calls to openmp runtime are wrapped by a set of functions which calculate checks slice sizes and raise an error if **`taskshare`** directive is violated
- Two variations of runtime checking:
  - Minimal slice size can be calculated at entry to the loop:
    - overhead per loop is constant
  - Must check each time a new slice is mapped to a thread by openmp runtime:
    - overhead per loop is proportional to number of slices in loop

*Used llvm-gcc compiler – a combination of gcc 4.2.1 front-end and llvm 2.5 back-end

Runtime Overhead – Edge Detection

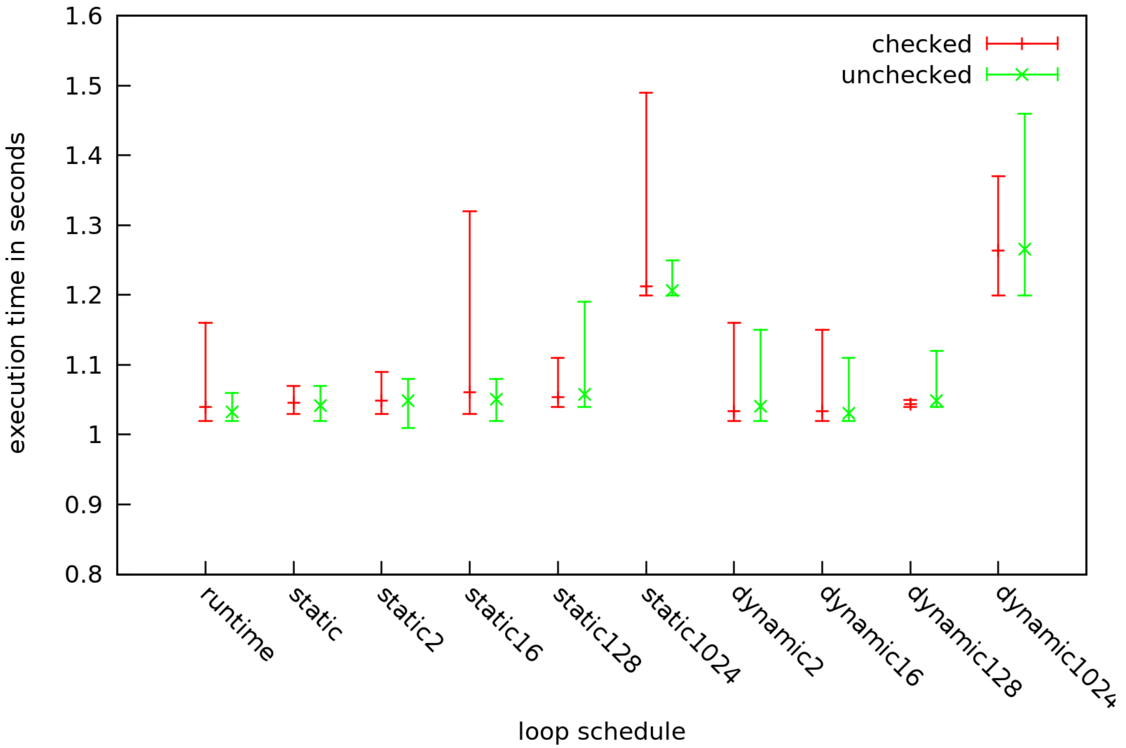
- UTDSP benchmark performing edge detection
- Used variant with arrays and no software pipelining for parallelization
- Used 4096x4096 element data-set for input
- Annotated 1 loop



Testbench: Intel Core i7 2.66GHz CPU, 32-bit Ubuntu Linux 9.04, llvm-gcc 2.5 -O2

Runtime Overhead - Demosaicing

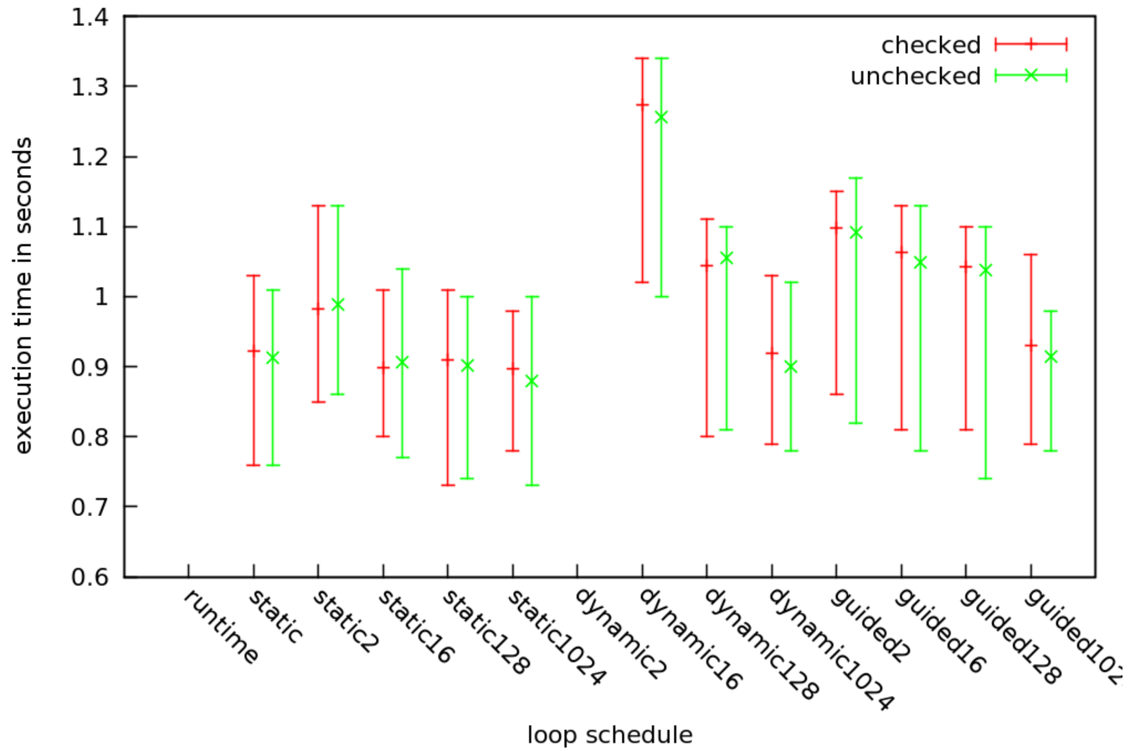
- Digital camera application
- Interpolates input from CFA sensor in camera into bitmap.
- Annotated 3 loops
- No measurements for “guided” schedule due to compiler bug.



Testbench: Intel Core i7 2.66GHz CPU
32-bit Ubuntu Linux 9.04, llvm-gcc 2.5 -O2

Runtime Overhead – Heat diffusion

- Heat diffusion simulation using finite difference method
- Common in high performance computing
- Annotated inner loop



Testbench: Intel Core i7 2.66GHz CPU, 32-bit Ubuntu Linux 9.04, llvm-gcc 2.5 -O2

Summary

- Aim to generate task graphs from simple loops expressed as C code parallelized with OpenMP.
- Multiple factors complicate this
  - use of arrays in ways which is opaque to compiler analysis
    - refactor array accesses
  - pointer aliasing
    - use **`restrict`** keyword and/or more precise analysis
  - factors determining inter-task dependencies only known at runtime
    - use **`taskshare`** directive to quantify inter-task dependencies
- Experiments suggests that impact of using **`taskshare`** directive on
  - coding effort
  - overhead of runtime checking

… is neglible

- Extend work to more general kinds of parallel loops

Questions?

Thanks for your attention!

Reach me at pl@imm.dtu.dk