Block-Level Dynamic Dependence Analysis for Task-Based Parallelism

Dimitrios S. Nikolopoulos

School of EEECS, Queen's University of Belfast

July 18, 2012

Table of Contents

Thread-Based Programming

- Hard for programmer to reason about thread interleavings and concurrent memory accesses
- Error-prone:
  - Races, deadlocks, livelocks, all hard to reproduce
- Task-based parallelism offers a higher level of abstraction
- Early models (e.g. OpenMP, Cilk) based on explicit synchronization
- New models based on automatic dependence analysis are emerging
  - Runtime dependence analysis based on programmer's annotation of memory footprint (OmpSs, SvS)
  - Static dependence analysis based on compiler's inference of memory footprint (DPJ)
  - Hybrid static-dynamic schemes – this talk

Opportunity for Dependence Analysis: FFT

```
void FFT_1D (long N, long N_SQRT, long FFT_BS, long TR_BS,
             double _Complex A[N_SQRT][N_SQRT])
{
  const size_t rowsz = N_SQRT * 2 * sizeof(double);
  const size_t longsz = sizeof(long);

  // Loop 1: Transpose
  for (long I = 0; I < N_SQRT; I += TR_BS) {
    void * tile_ii  = &A[I][I];

    #pragma task \
      in(N[longsz], N_SQRT[longsz], TR_BS[longsz]) \
      inout(tile_ii[rowsz][TR_BS|TR_BS*2*sizeof(double)]);
    trsp_blk (N, N_SQRT, TR_BS, tile_ii);

    for (long J = I + TR_BS; J < N_SQRT; J += TR_BS) {
      void * tile_ij  = &A[I][J];
      void * tile_ji  = &A[J][I];

      #pragma task \
        in(N[longsz], N_SQRT[longsz], TR_BS[longsz]) \
        inout(tile_ij[rowsz][TR_BS|TR_BS*2*sizeof(double)], \
              tile_ji [rowsz][TR_BS|TR_BS*2*sizeof(double)]);
      trsp_swap(N, N_SQRT, TR_BS, tile1, tile2);
    }
  }

  // Loop 2: First FFT round
  for (long J = 0; J < N_SQRT; J += FFT_BS) {
     tile  = &A[J][0];

    #pragma task \
      in(N_SQRT[longsz], FFT_BS[longsz]) \
      inout(tile[FFT_BS*rowsz]);
    FFT1D(N_SQRT, FFT_BS, &A[J][0]);
  }

   ...
}
```

Table of Contents

Whole Object Analysis

- Analysis of whole task arguments (objects)
  - Simple: use object base address for detecting dependencies
  - Assumes objects are contiguous in memory
    - Can not analyze dependencies between multi-dimensional array blocks
    - Can not analyze dependencies static or dynamic, arbitrary collections of objects
    - Can not detect partial overlaps
  - May require data layout transformations (memory copies)
  - Can not be used to write parallel code operating on dynamic data structures

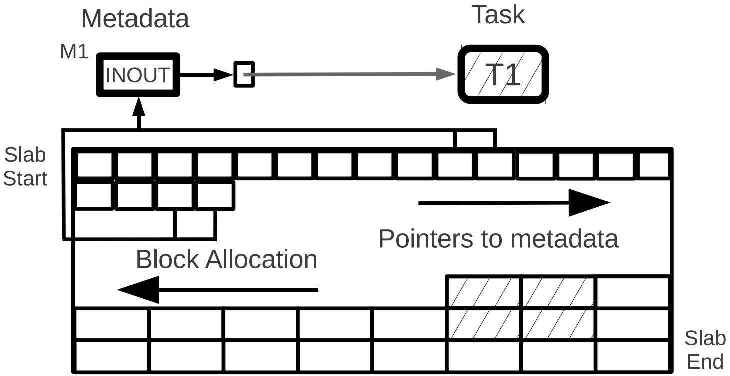
Block-Level Analysis

- Block-level analysis
  - Intuition: partition virtual memory into fixed-size blocks
  - Analyze dependencies between blocks instead of program objects
  - Treat objects as arbitrary collections of memory blocks
- Can analyze dependencies between multi-dimensional array blocks, arbitrary collections of objects, static or dynamic
- Can detect partial overlaps
- Programmability: easier to write parallel code operating on static or dynamic structures, array-based or linked
- Overhead: Objects composed of N blocks need N analysis checks

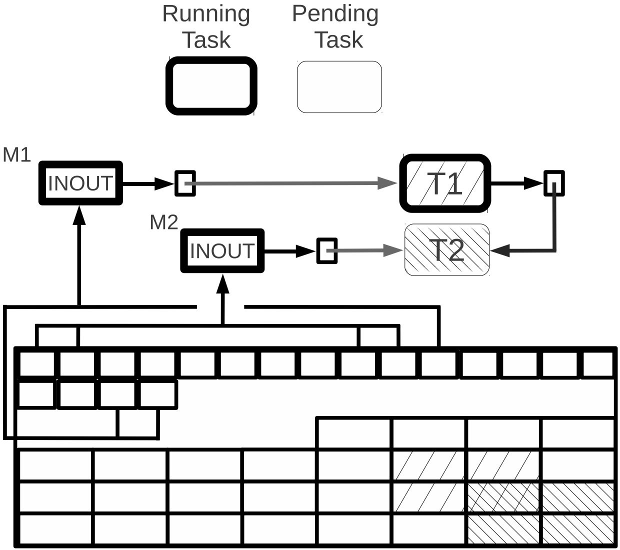
Elements of Block-Level Analysis

- Task descriptor containing the closure of task
  - Code, memory footprint and other dependent tasks
- Block descriptors containing queue of tasks waiting to access the block
  - Include access mode (in, out, inout)
  - Use versioning (renaming) on writes
  - Versions reveal parallelism
- Efficient implementation
  - Custom memory allocator with task metadata embedded with memory allocator metadata
  - O(1) lookups for all metadata
  - Small memory footprint for low cache pollution

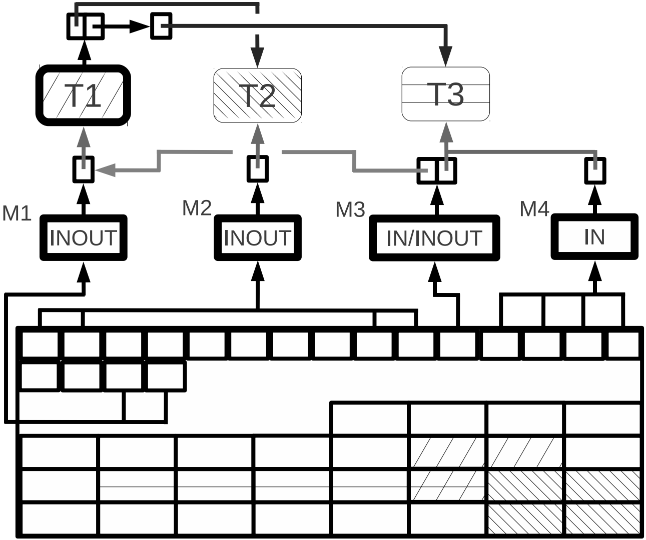
Block-Level Analysis by Example



Block-Level Analysis by Example



Block-Level Analysis by Example



Block-Level Analysis by Example

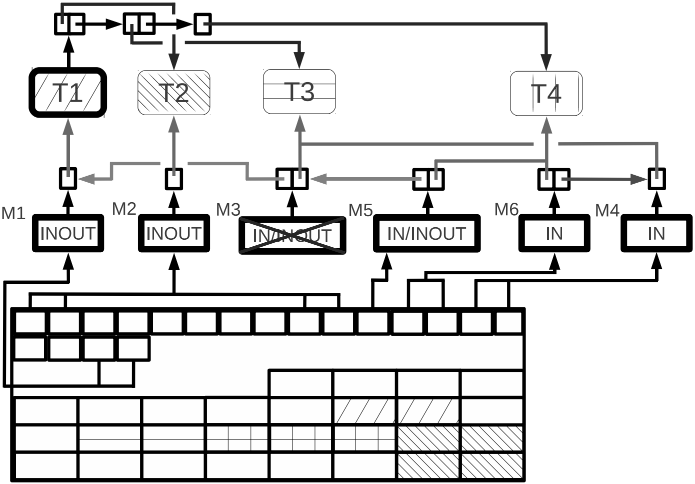


Table of Contents

Opportunity for Static Analysis

```
int a = 1, b = 2, c = 3;
int *alias = &b;

void set(int *x, int *y) { *x = *y; }
void addto(int *x, int *y) { *x += *y; }

int main() {
  #pragma task inout(&b) safe(&c);
  addto(&b, &c);

  #pragma task safe(&a) in(alias);
  set(&a, alias );

  #pragma wait all

  #pragma task safe(&a) safe(&c);
  set(&a, &c);
}
```

- Dependence analysis unnecessary on a and c in first two tasks because of barrier
- Dependence analysis unnecessary on a and c in third task because of barrier
- Tedious for programmer to manage, best handled by compiler

Language $\lambda_{\|}$

$$
\begin{array}{lrcl}
\textbf{Values} & v & ::= & n \mid () \mid \lambda\, x\, .\, e \\
\textbf{Expressions} & e & ::= & v \mid x \mid e; e \mid e\ e \mid \texttt{ref}\ e \mid !\, e \mid e := e \\
 & & \mid & \texttt{task}(e_1, \ldots, e_n)\,\{e\} \mid \texttt{barrier} \\
\textbf{Locations} & \rho & \in & \mathcal{L} \\
\textbf{CFG Points} & \phi & \in & \mathcal{F} \\
\textbf{Tasks} & \pi & \in & \mathcal{T} \\
\textbf{Types} & \tau & ::= & \mathit{int} \mid \mathit{unit} \mid (\tau, \phi) \to (\tau, \phi) \mid \mathit{ref}^{\,\rho}(\tau) \\
\textbf{Constraints} & C & ::= & \emptyset \mid C \cup C \mid \tau \leq \tau \mid \rho \leq \rho \mid \phi \leq \phi \\
 & & \mid & \rho \leq \pi \mid \pi \| \pi \mid \phi : \mathsf{Barrier} \mid \phi : \pi \\
\textbf{Environments} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau
\end{array}
$$

- Simply-typed lambda calculus
- Extensions: dynamic memory allocation, task creation, barrier synchronization

Language $\lambda_{\parallel}$

$$
\begin{array}{llcl}
\textbf{Values} & v & ::= & n \mid () \mid \lambda\, x\,.\, e \\
\textbf{Expressions} & e & ::= & v \mid x \mid e; e \mid e\ e \mid \texttt{ref}\ e \mid !\,e \mid e := e \\
 & & \mid & \texttt{task}(e_1, \ldots, e_n)\, \{e\} \mid \texttt{barrier} \\
\textbf{Locations} & \rho & \in & \mathcal{L} \\
\textbf{CFG Points} & \phi & \in & \mathcal{F} \\
\textbf{Tasks} & \pi & \in & \mathcal{T} \\
\textbf{Types} & \tau & ::= & int \mid unit \mid (\tau, \phi) \to (\tau, \phi) \mid ref^{\rho}(\tau) \\
\textbf{Constraints} & C & ::= & \emptyset \mid C \cup C \mid \tau \leq \tau \mid \rho \leq \rho \mid \phi \leq \phi \\
 & & \mid & \rho \leq \pi \mid \pi \| \pi \mid \phi : \text{Barrier} \mid \phi : \pi \\
\textbf{Environments} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau
\end{array}
$$

- Type system generates set of constraints solved by rewriting rules:
  - Inference labels ϕ and ρ generate control-flow and points-to graphs
  - Constraints for data ($\rho_1 \leq \rho_2$), control ($\phi_1 \leq \phi_2$), task memory footprints ($\rho \leq \pi$), can happen in parallel ($\pi_1 \parallel \pi_2$) and barrier (ϕ)

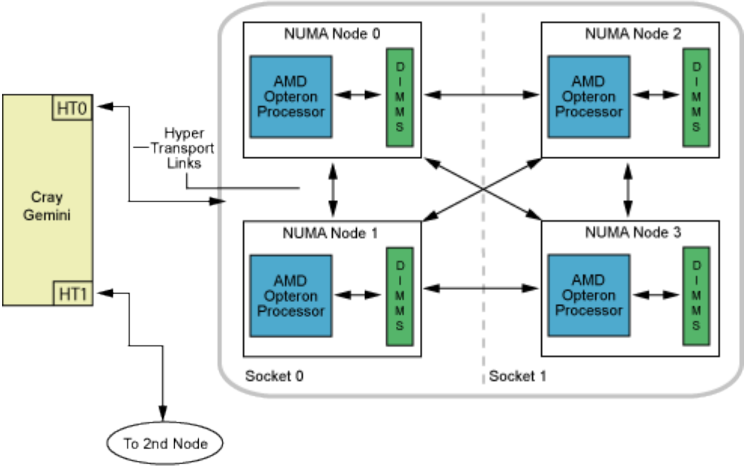
Table of Contents

Implementation Details

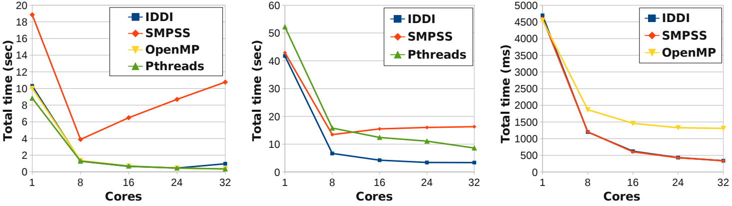
- Source-to-source compiler using CIL
- OpenMP and OmpSs-like syntax supported
  - Strided arguments, multi-dimensional array tiles, and dynamic regions
- Locksmith engine for points-to analysis
- Scalable runtime support:
  - Concurrent queues, NUMA-aware data allocation and task scheduling
  - Provable low bounds and determinism

Experimental Platform

- Cray XE6 compute node on Hector, 32 GB DRAM, NUMA

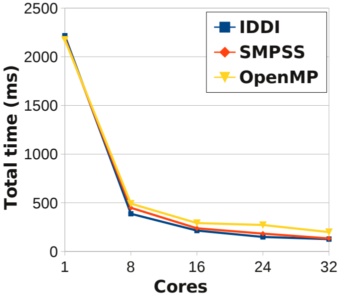


Experimental Results

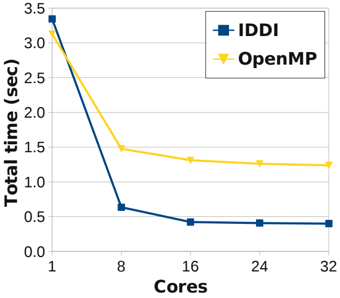


(a) Black-Scholes (b) Ferret (c) Cholesky

Experimental Results

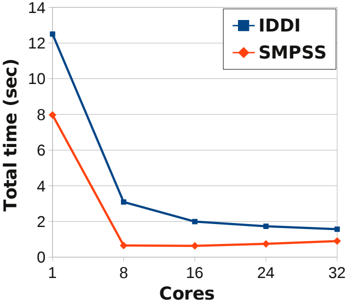


(g) SMPSS-FFT

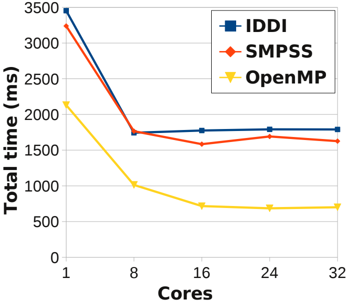


(h) SPLASH-FFT

Experimental Results



(e) HPL



(f) Jacobi

Table of Contents

Lessons and Challenges

- Lesson: Task-based parallelism achieves good balance between productivity and performance
- Lesson: Automatic dependence analysis uncovers more parallelism and enables further optimization
- Lesson: A new opportunity for parallelizing compilers
- Challenge: Overhead still present in some codes with low operational intensity (e.g. stencils)
- Challenge: Points-to analysis still limited (even in simple cases)
- Challenge: no magic recipe, expose analysis options as user knobs or autotuners

Acknowledgment

- Collaborators: Angelos Bilas, Angelos Papatriantafyllou, Polyvios Pratikakis, George Tzenakis, Hans Vandierendonck
- Funding: EU FP7 (ENCORE, TEXT, I-CORES)
- Infrastructure: Hector (EPCC), BSC