SCOOP

Language extensions and compiler optimizations for task-based programming models

Foivos S. Zakkak

University of Crete
School of Sciences and Engineering
Computer Science Department
and
Institute of Computer Science
Foundation for Research and Technology Hellas

24/02/2012

Table of Contents

Concurrent programming

Shared Memory

- **implicit** communication
  - requires synchronization to avoid concurrency errors
- requires sophisticated scheduling and data allocation to reduce **memory traffic** (especially on NUMA)
- non deterministic

Message Passing

- **explicit** communication
  - requires communication buffer management
- requires sophisticated scheduling and data allocation to reduce the **number of messages**
- non deterministic

Task-based Programming Models

Task-based Programming Model

- High level
- **Implicit** communication
  (through shared memory or through the runtime)
- **Synchronization**
  - explicit in early models (OpenMP, Cilk)
  - **implicit** in recent models

We consider a task:

- a **piece of code** that can execute in parallel with other tasks
- **the data** that it will access

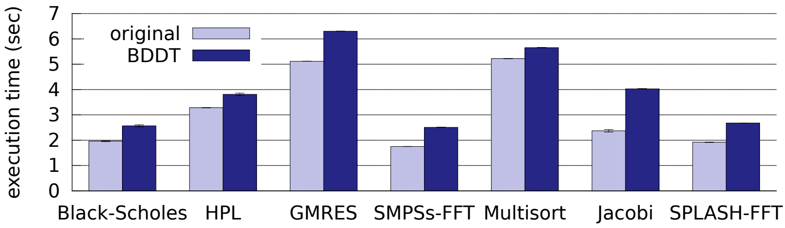
BDDT

Block-level **D**ynamic Dependence Analysis for **D**eterministic **T**ask-Based Parallelism

- Requires **memory footprints**
- **Dynamically** detects and resolves task dependencies based on the memory footprints
  - **implicit synchronization**
- **Flexible way to express parallelism**
- **Deterministic**

A memory footprint is a description of the memory locations the task will access (read/write/both)

Runtime overheads (running on a single core)

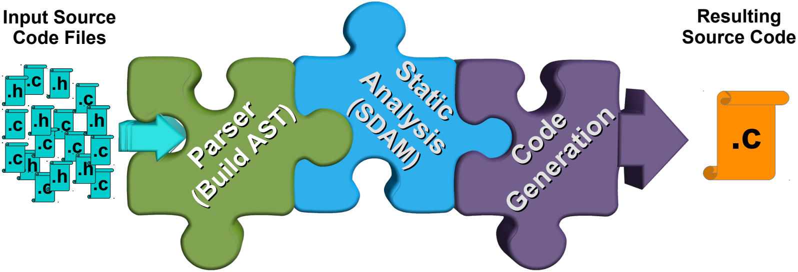


- BDDT incurs an overhead of 7%-41%
- Best et al. also report overhead from under 5% to over 40% in SvS (another task-based runtime)

Ran on Intel Xeon E5520 2.27 GHz 4-core and 12GB main memory.

Table of Contents

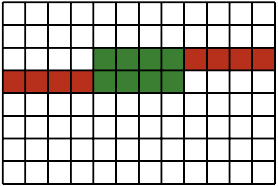
SCOOP Stack



Input Source Code Files
.h
.c
Parser (Build AST)
Static Analysis (SDAM)
Code Generation
Resulting Source Code
.c

SCOOP Syntax

- SMPSs-like `#pragma` directives to define tasks and their footprints.
- We mark task creation at the calling context.
  This way:
  1. we are able to differentiate when a function is called sequentially or asynchronously as a parallel task
  2. we are able to fix the task footprint for each invocation, marking its arguments as safe or not
- Tiled array accesses through stride arguments.



Argument Independence Inference

SCOOP queries SDAM (**S**tatic **D**ependence **A**nalysis **M**odule) for independent arguments.

SDAM infers argument independence in three steps.

1. computes aliasing information for all memory locations in the program
2. computes which tasks can run in parallel
3. checks whether a memory location (through any alias) is never accessed in parallel by more than one task.

Code Generation

- Transforms the input program to use BDDT for creating tasks
- Disables BDDT's runtime dependence checks for inferred or declared independent arguments
- Optimizes the interaction with BDDT's generic library API by producing custom code
  1. No **va_args**
  2. **Inline** code
  3. No **if** statements
  4. Scalars are **passed by value**

Code Example

```c
void t1(int *arg1, int arg2) {
  //function that will be called in parallel
}
void t2(int *arg1) {
  //function that will be called in parallel
}
int *foo(int *x, int sz) {
  x = (int *)malloc(size);
   ...
  return x;
}
int main(void) {
   ...
  res1 = foo(arg1, size);
  #pragma scoop start(number_of_spes)
  for (...) {
    #pragma scoop task input(a) inout(res1[size])
    t1(res1, a);
  }
  #pragma scoop wait all
  for (...) {
    #pragma scoop task input(arg2[size])
    t2(arg1);
  }
  t1(res1, a);
  #pragma scoop finish
   ...
}
```

```c
...
int main(void) {
  ...
  res1 = foo(arg1, size);
  bddt_init(number_of_spes);
  for (...) {
    ... //create task descriptor and pass to runtime
    task_descriptor->arguments[0].addr = res1;
    task_descriptor->arguments[0].flag = INOUT;
    task_descriptor->arguments[0].size = size;
    task_descriptor->arguments[0].addr = &a;
    task_descriptor->arguments[1].flag = INPUT|SAFE;
    task_descriptor->arguments[1].size = sizeof(a);
    ...
  }
  bddt_wait_all();
  for (...) {
    ... //create task descriptor and pass to runtime
    task_descriptor->arguments[0].addr = arg1;
    task_descriptor->arguments[0].flag = INPUT|SAFE;
    task_descriptor->arguments[0].size = size;
    ...
  }
  t1(res1, a);
  bddt_shutdown();
  ...
}
```

Table of Contents

- Introduction
- SCOOP
- Evaluation
- Conclusions and Discussion

	Benchmark	LOC	Tasks	Total Args	Scalar Args
x86 SMP	Black-Scholes	1540	1	8	1
	SMPSs-FFT	2147	8	36	25
	SPLASH-FFT	2920	4	12	0
	GMRES	2652	18	72	20
	HPL	2396	11	63	35
	Jacobi	1076	1	6	0
	Multisort	1118	3	8	4
Cell BE	Cholesky	2195	4	8	0
	LU	2819	3	10	3
	SAXPY	1675	1	3	1
	SGEMV	2159	1	4	1

Measurement Methodology

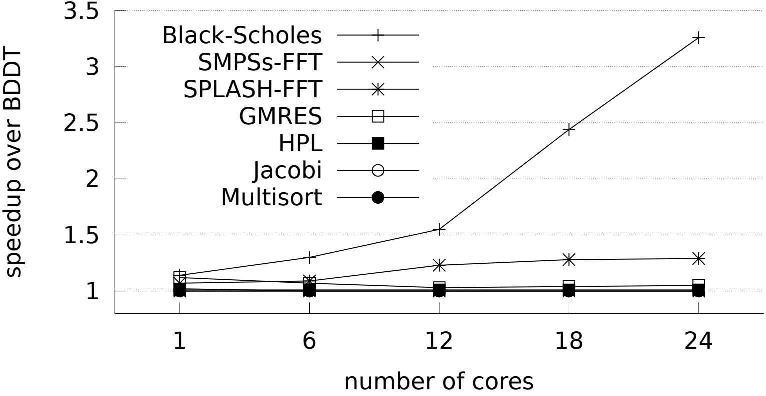
- Initialization and I/O are excluded
- We ran:
  1. a version written using BDDT API
  2. a version written using the SCOOP annotations

Performance Improvement

	Benchmarks	Speedup Over BDDT	Inferred Args	Non scalar Args
x86 SMP	Black-Scholes	3.26	**7**	**7**
	SMPSs-FFT	1	0	11
	SPLASH-FFT	1.29	7	12
	GMRES	1.05	9	52
	HPL	1.01	1	28
	Jacobi	1	0	6
	Multisort	1	0	4
Cell BE	Cholesky	1	0	8
	LU	1.01	3	7
	SAXPY	1.02	**2**	**2**
	SGEMV	1.18	2	3

- Average speedup 1.26
- Ran on a 24-core computer node of a Cray XE6
  2x AMD 2.1 GHz 12-core and 32GB main memory

Scalability on x86



Exposing Independencies

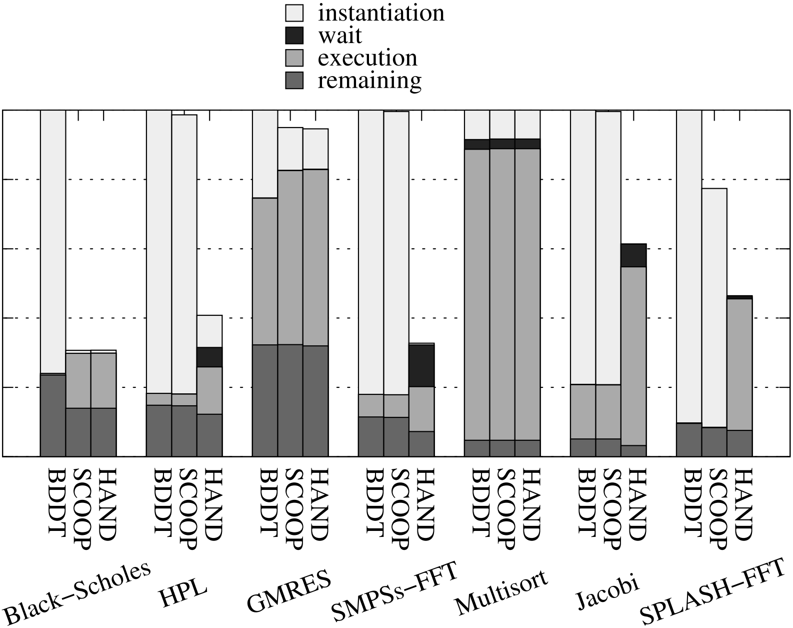


Table of Contents

Conclusions

SCOOP, with its evaluation, confirm that static analysis along with compile-time transformations can drastically improve the performance of task-based programming models.

SCOOP managed to:

1. reduce BDDT's dependence analysis overhead
2. improve the benchmarks' scalability

Discussion

Our experience taught us that:

- There is space for compile time optimizations in task-based programming models
- SCOOP's design allows easy porting to completely different architectures
- SCOOP's C extensions make programming a lot easier than using BDDT's API
- SCOOP could be used also as a tool increasing the programmer's productivity. With some extra effort it can:
  - report possible wrong memory footprints
  - infer the task memory footprints

Future Work

Regions

1. Express complex task footprints
   (ie. lists, hashtables)
2. Dynamically allocate or deallocate memory within tasks
   (ie. add/remove node)
3. Reduce memory management overhead
   (less mallocs due to memory pool)
4. Reduce dependence analysis overhead
   (a single dependency check for the whole region)
5. Increase memory locality
   (by implementation)

Port to other platforms

- SCC/BDDT
- Formic/Myrmics

Acknowledgements

Advisor:

- A. Bilas

Co-advisors:

- D. Nikolopoulos
- P. Pratikakis

Committee:

- E. Markatos

Colleagues:

- D. Chasapis
- G. Tzenakis