Inference and Declaration of Independence in Task-Parallel Programs

Foivos S. Zakkak[1], Dimitrios Chasapis[1], Polyvios Pratikakis[1],
Angelos Bilas[1], and Dimitrios S. Nikolopoulos[2]

[1] Foundation for Research and Technology - Hellas
[2] Queens University of Belfast

Abstract. The inherent difficulty of thread-based shared-memory programming has recently motivated research in high-level, task-parallel programming models. Recent advances of Task-Parallel models add implicit synchronization, where the system automatically detects and satisfies data dependencies among spawned tasks. However, dynamic dependence analysis incurs significant runtime overheads, because the runtime must track task resources and use this information to schedule tasks while avoiding conflicts and races.
We present SCOOP, a compiler that effectively integrates static and dynamic analysis in code generation. SCOOP combines context-sensitive points-to, control-flow, escape, and effect analyses to remove redundant dependence checks at runtime. Our static analysis can work in combination with existing dynamic analyses and task-parallel runtimes that use annotations to specify tasks and their memory footprints. We use our static dependence analysis to detect non-conflicting tasks and an existing dynamic analysis to handle the remaining dependencies. We evaluate the resulting hybrid dependence analysis on a set of task-parallel programs.

Keywords: Task-Parallelism, Static Analysis, Dependence Analysis, Deterministic Execution

1 Introduction

The inherent difficulty and complexity of thread-programming has recently lead to the development of several task-based programming models [1–4]. Task-based parallelism offers a higher level abstraction to the programmer, making it easier to express parallel computation. Although early task-based parallel languages required manual synchronization, recent task-based systems implicitly synchronize tasks, using a task's memory footprint at compile or at run time to detect and avoid concurrent accesses or even produce deterministic execution [5–9]. In order for such a dependence analysis to benefit program performance, it must (i) be accurate, so that it does not discover false dependencies; and (ii) have low overhead, so that it does not nullify the benefit of discovering extra parallelism.

Static systems detect possibly conflicting tasks in the program code and insert synchronization prohibiting concurrent access to shared memory among all runtime instances of possibly conflicting tasks. As this can be too restrictive, some existing static systems speculatively allow conflicting task instances to run in parallel and use dynamic techniques to detect and correct conflicts [6].

Dynamic dependence analysis offers the benefit of potentially discovering more parallelism than is possible to describe statically in the program, as it checks all runtime task instances for conflicts and only synchronizes task instances that actually (not potentially) access the same resources. However, dynamic dependence analysis incurs a high overhead compared to hand-crafted synchronization. It requires a complex runtime system to manage and track memory allocation, check for conflicts on every task instance, and schedule parallel tasks. Often, the runtime cost of checking for conflicts in pessimistic, or rolling back a task in optimistic runtimes becomes itself a bottleneck, limiting the achievable speedup as the core count grows.

This paper aims to alleviate the overhead of dynamic dependence analysis without sacrificing the benefit of implicit synchronization. We develop SCOOP, a compiler that brings together static and dynamic analyses into a hybrid dependence analysis in task-parallel programs. SCOOP uses a static dependence analysis to detect and remove runtime dependence checks when unnecessary. It then inserts calls to the task-parallel runtime dynamic analysis to resolve the remaining dependencies only when necessary. Our work makes the following contributions:

- We present a static analysis that detects independent task arguments and reduces the runtime overhead of dynamic analysis. We implement our analysis in SCOOP, a source-to-source compiler for task-parallel C, using OpenMP-Task extensions to define tasks and their memory footprints.
- We combine our static dependence analysis with an existing dynamic analysis, resulting in an efficient hybrid dependence analysis for parallel tasks. SCOOP uses the static dependence analysis to infer redundant and unnecessary dependence checks and inserts custom code to use the dynamic analysis only for the remaining task dependencies at runtime.
- We evaluate the effect of our analysis using an existing runtime system. On a representative set of benchmarks, SCOOP discovers almost all independent task arguments. In applications with independent task arguments, SCOOP achieved speedups up to 68%.

2 Motivation

Consider the C program in Figure 1. This program has three global integer variables, `a`, `b` and `c` (line 1) and a global pointer `alias` (line 2) that points to `b`. Function `set()` copies the value of its second argument to the first (line 4) and function `addto()` adds the value of its second argument to the value of its first (line 5). The two functions are then invoked in two parallel tasks, to add `c` to `b` (lines 8–9) and to set the value of `a` to the value pointed to by `alias` (lines 11–12). The first task reads and writes its first argument, `b`, and reads from its second argument, `c`. Similarly, the second task writes to its first argument, `a`, and reads from its second argument `alias`. The program then waits at a synchronization point for the first two tasks to finish (line 14) and then spawns a third task that reads from `c` and writes to `a` (lines 16–17).

To execute this program preserving the sequential semantics, the second task `set` needs to wait until the value of `b` is produced by the first task, i.e., there is a *dependence* on memory location `b`. Note, however, that since the third task cannot be spawned until the first two return, memory location `c` is only accessed by the first task and `a`

```
1  int a = 1, b = 2, c = 3;
2  int *alias = &b;
3
4  void set(int *x, int *y) { *x = *y; }
5  void addto(int *x, int *y) { *x += *y; }
6
7  int main() {
8    #pragma task inout(&b) in(&c)
9    addto(&b, &c);
10
11   #pragma task out(&a) in(alias)
12   set(&a, alias);
13
14   #pragma wait all
15
16   #pragma task out(&a) in(&c)
17   set(&a, &c);
18 }
```

Fig. 1: Tasks with independent arguments

is only accessed by the second. So, any dependence analysis time spent checking for conflicts on `a` or `c` before it starts the first two tasks is unnecessary overhead that delays the creation of the parallel tasks, possibly restricting available parallelism and thus the scalability of the program. So, the `#pragma task` directive spawning these tasks states that `c` and `a` are *safe* or *independent* arguments, that the analysis does not need to track. For the same reason, both the arguments of the third task are safe, meaning it can start to run without checking for dependencies.

Section 3 describes the static analysis we use to discover independent task arguments like `a` and `c` above. Inferring that a task argument does not need to be checked for dependencies requires verifying that no other task can access that argument. In short, the static analysis infers this independence in three steps. First, we compute aliasing information for all memory locations in the program. Second, we compute which tasks can run in parallel; we do not need to check for conflicting arguments between for example the second and third task in the example of Figure 1, even though `a` is accessed by both, because the barrier prohibits them from running at the same time. Third, we check whether a memory location (through any alias) is never accessed in parallel by more than one task. We can then safely omit checking this location at runtime. We can extend this idea by differentiating between read and write accesses and allowing for concurrent reads without checking for dependencies, as long as no writes can happen in parallel.

3 Static Independence Analysis

This section presents the core algorithm of the independence analysis. To simplify the presentation, we use a small language $\lambda_{\|}$, and do not differentiate between reads and writes. Section 4 describes how we extended our analysis to the C full programming language.

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

Fig. 2: $\lambda_{\|}$: A simple task-based parallel language

3.1 The Language $\lambda_{\|}$

Figure 2 presents $\lambda_{\|}$, a simple task-parallel programming language. $\lambda_{\|}$ is a simply-typed lambda calculus extended with dynamic memory allocation and updatable references, task creation and barrier synchronization. Values include integer constants n, the unit value `()` and functions $\lambda\, x\,.\, e$. Program expressions include variables x, function application $e_1\ e_2$, sequencing, memory operations and task operations. Specifically, expression `ref` e allocates some memory, initializes it with the result of evaluating e, and returns a pointer to that memory; expression $e_1 := e_2$ evaluates e_1 to a pointer and updates the pointed memory using the value of e_2; and expression $!e$ evaluates e to a pointer and returns the value in that memory location. Expression $\texttt{task}(e_1, \ldots, e_n)\,\{e\}$ evaluates each e_i to a pointer and then evaluates the task body e, possibly in parallel. The task body e must always return `()` and can only access the given pointers; if e is evaluated in a parallel task, the task expression immediately returns `()`. Finally, expression `barrier` waits until all tasks issued until this point have been executed.

3.2 Type System

We use a type system to generate a set of constraints C and infer independence of task arguments. Figures 3(a) and 3(b) shows the type language of $\lambda_{\|}$, which includes integer and unit types, function types $(\tau, \phi) \to (\tau, \phi)$ and reference (or pointer) types $\mathit{ref}^{\rho}(\tau)$. We annotate function and reference types with inference labels ϕ and ρ, and use them to compute the *control flow graph* among ϕ labels and the *points-to graph* among ρ labels, respectively. Specifically, typing the program creates a constraint graph C, which has three kinds of vertices. Location labels ρ annotating reference types abstract over memory locations, control flow labels ϕ abstract over a control flow point in the program execution, and task labels π abstract over parallel tasks in the program. Typing a program e in $\lambda_{\|}$ creates a constraint graph C. Constraint $\tau_1 \leq \tau_2$ requires τ_1 to be a subtype of τ_2. Constraint $\rho_1 \leq \rho_2$ (ρ_1 "flows to" ρ_2) means abstract memory location ρ_2 references all locations that ρ_1 references. Constraint $\phi_1 \leq \phi_2$ means the execution of control flow point ϕ_2 follows immediately after the execution of ϕ_1. Constraint $\rho \leq \pi$ (ρ "is an argument of" π) means an abstract memory location ρ is in the memory footprint of task π. Constraint $\pi_1 \| \pi_2$ (π_1 "can happen in parallel with" π_2) means there may be an execution where tasks represented by π_1 and π_2 are executed in parallel. Constraint $\phi : \mathsf{Barrier}$ means there is barrier synchronization at control flow

$$\text{[T-INT]}\ \frac{}{C;\phi;\Gamma \vdash n : \mathit{int};\phi} \qquad \text{[T-UNIT]}\ \frac{}{C;\phi;\Gamma \vdash () : \mathit{unit};\phi}$$

$$\text{[T-FUN]}\ \frac{\phi_1\text{–}\mathit{fresh} \qquad C;\phi_1;\Gamma, x:\tau_1 \vdash e : \tau_2;\phi_2}{C;\phi;\Gamma \vdash \lambda\, x\,.\, e : (\tau_1,\phi_1) \to (\tau_2,\phi_2);\phi}$$

$$\text{[T-VAR]}\ \frac{\Gamma(x) = \tau}{C;\phi;\Gamma \vdash x : \tau;\phi} \qquad \text{[T-SEQ]}\ \frac{C;\phi;\Gamma \vdash e_1 : \mathit{unit};\phi_1 \qquad C;\phi_1;\Gamma \vdash e_2 : \tau;\phi_2}{C;\phi;\Gamma \vdash e_1; e_2 : \tau;\phi_2}$$

$$\text{[T-APP]}\ \frac{C;\phi;\Gamma \vdash e_1 : (\tau_1,\phi_1) \to (\tau_2,\phi_2);\phi' \qquad C;\phi';\Gamma \vdash e_2 : \tau_1;\phi'' \qquad C \vdash \phi'' \le \phi_1}{C;\phi;\Gamma \vdash e_1\ e_2 : \tau_2;\phi_2}$$

$$\text{[T-REF]}\ \frac{C;\phi;\Gamma \vdash e : \tau;\phi' \qquad \rho\text{–}\mathit{fresh}}{C;\phi;\Gamma \vdash \texttt{ref}\ e : \mathit{ref}^{\rho}(\tau);\phi'} \qquad \text{[T-DEREF]}\ \frac{C;\phi;\Gamma \vdash e : \mathit{ref}^{\rho}(\tau);\phi'}{C;\phi;\Gamma \vdash\, !\, e : \tau;\phi'}$$

$$\text{[T-ASGN]}\ \frac{C;\phi;\Gamma \vdash e_1 : \mathit{ref}^{\rho}(\tau);\phi' \qquad C;\phi';\Gamma \vdash e_2 : \tau;\phi''}{C;\phi;\Gamma \vdash e_1 := e_2 : \tau;\phi''}$$

$$\text{[T-TASK]}\ \frac{\forall i \in [1..n]\,.\, C;\phi_i;\Gamma \vdash e_i : \mathit{ref}^{\rho_i}(\tau_i);\phi_{i+1} \qquad \phi', \pi\text{–}\mathit{fresh} \qquad C \vdash \phi_{n+1} \le \phi' \qquad C \vdash \phi' : \pi \qquad \forall i \in [1..n]\,.\, C \vdash \rho_i \le \pi \qquad C;\phi';\Gamma \vdash e' : \mathit{unit};\phi''}{C;\phi_1;\Gamma \vdash \texttt{task}(e_1,\ldots,e_n)\,\{e'\} : \mathit{unit};\phi'}$$

$$\text{[T-BARRIER]}\ \frac{\phi'\text{–}\mathit{fresh} \qquad C \vdash \phi' : \mathsf{Barrier} \qquad C \vdash \phi \le \phi'}{C;\phi;\Gamma \vdash \texttt{barrier} : \mathit{unit};\phi'} \qquad \text{[T-SUB]}\ \frac{C;\phi;\Gamma \vdash e : \tau;\phi' \qquad C \vdash \tau \le \tau'}{C;\phi;\Gamma \vdash e : \tau';\phi'}$$

(a) Type Inference Rules

$$\begin{aligned}
C \cup \{\mathit{int} \le \mathit{int}\} &\ \Rightarrow\ C \\
C \cup \{\mathit{unit} \le \mathit{unit}\} &\ \Rightarrow\ C \\
C \cup \{(\tau_1,\phi_1) \to (\tau_2,\phi_2) \le (\tau_1',\phi_1') \to (\tau_2',\phi_2')\} &\ \Rightarrow \\
&\quad C \cup \{\tau_1' \le \tau_1, \tau_2 \le \tau_2', \phi_1' \le \phi_1, \phi_2 \le \phi_2'\} \\
C \cup \{\mathit{ref}^{\rho_1}(\tau_1) \le \mathit{ref}^{\rho_2}(\tau_2)\} &\ \Rightarrow\ C \cup \{\rho_1 \le \rho_2, \tau_1 \le \tau_2, \tau_2 \le \tau_1\} \\
C \cup \{\rho \le \rho', \rho' \le \rho''\} &\ \cup\!\Rightarrow \{\rho \le \rho''\} \\
C \cup \{\rho \le \rho', \rho' \le \pi\} &\ \cup\!\Rightarrow \{\rho \le \pi\} \\
C \cup \{\phi_1 \le \phi_2, \phi_1 : \pi\} &\ \cup\!\Rightarrow \{\phi_2 : \pi\} \text{ when } \{\phi_2 : \mathsf{Barrier}\} \notin C \\
C \cup \{\phi : \pi_1, \phi : \pi_2\} &\ \cup\!\Rightarrow \{\pi_1 \| \pi_2\}
\end{aligned}$$

(b) Constraint Solving Rules

Fig. 3: Constraint generation and solving

point ϕ of all executions. Finally, constraint $\phi : \pi$ means there can be an execution where task π is executed in parallel while control flow reaches point ϕ.

Figure 3(a) shows the type system for $\lambda_{\|}$. Typing judgments have the form $C;\phi;\Gamma \vdash e : \tau;\phi'$, meaning program expression e has type τ under assumptions Γ and constraint set C. Rules [T-INT] and [T-UNIT] and [T-VAR] are standard, with the addition of control flow point ϕ as both starting and ending point. Rule [T-FUN] types function definitions. The function type $(\tau_1,\phi_1) \to (\tau_2,\phi_2)$ includes the starting and ending con-

trol flow point of the function body. As with typing the other values, function definitions do not change control flow point ϕ. Rule [T-SEQ] types sequence, where e_1 must have type $unit$, and the sequence expression has the type of e_2. The ending control flow point ϕ_1 of e_1 is the starting point of e_2. The first two premises in rule [T-APP] type the function expression e_1 and argument e_2 capturing the control flow order, the third premise "inlines" the control flow of the function by setting the function starting point immediately after the evaluation of the argument. The function application ends at ϕ_2. Rule [T-REF] creates a fresh label ρ that represents all memory locations produced by the expression, and annotates the resulting type. Rules [T-DEREF] and [T-ASGN] are straightforward, and type reference read and write expressions respectively.

Rule [T-TASK] types task creation expressions. The first premise types the task argument expressions $e_1, \ldots, e_n$ with reference types in that control flow order. The next three premises create a constraint that task π runs in parallel with control flow point ϕ' of the task-create expression. The fifth premise marks all locations ρ_i of the arguments as the footprint of task π, and the last premise requires the task body to have type $unit$. The task's control flow ends at any control flow point ϕ''. Rule [T-BARRIER] types barrier expressions, marking control flow point ϕ' as a barrier synchronization operation. Finally, rule [T-SUB] is standard subsumption.

3.3 Constraint Resolution

Applying the type system shown in Figure 3(a) generates a set of constraints C. To infer task arguments that are safe to skip during the runtime dependence analysis, we first compute the may-happen-in-parallel relation $\pi_1 \| \pi_2$ among tasks by solving the constraints C. Figure 3(b) shows the constraint resolution algorithm as a set of rewriting rules that are applied exhaustively until C cannot change any further. Here, $\cup \Rightarrow$ rewrites the constraints on the left to be the union of the constraints on both the left and right side.

The first four rules reduce subtyping constraints into edges between abstract labels: we drop integer and unit subtyping; we replace function subtyping with contravariant edges between the starting control flow points and arguments, and covariant edges between the returning control flow points and results; and we replace reference subtyping with equality on the referenced type (note both directions of subtyping) and a flow constraint on the abstract location labels. The fifth rule solves the points-to graph by adding all transitivity edges between abstract memory locations, and the seventh rule marks any locations aliasing task arguments also as task arguments. The seventh rule amounts to a forwards data-flow analysis on the control flow graph. Namely, for every control flow edge $\phi_1 \leq \phi_2$ we propagate any task π that executes in parallel with ϕ_1 to also execute in parallel with ϕ_2, unless ϕ_2 is a barrier. Finally, the last rule marks any two tasks π_1 and π_2 that both run in parallel with any control flow point ϕ, as also in parallel with one another. We use C^* to represent the result of exhaustively applying the constraint resolution rules on a set C.

3.4 Task Argument Independence

Having solved the constraints C of a program, we can now infer independent task arguments, namely arguments that cannot be accessed concurrently by any two parallel tasks. Formally, we define the dependent set $D_C(\rho)$ of location ρ under constraints C to be the set of tasks that can access ρ in parallel:

$$D_C(\rho) \doteq \{\pi \mid C^* \vdash \rho \leq \pi\}$$

We can now compute independent task arguments, i.e., memory locations that can be at most accessed by one task:

$$C \vdash \mathit{Safe}(\rho) \iff |D_C(\rho)| \leq 1$$

4 Implementation

We have extended the algorithm presented in Section 3 to the full C programming language in a compiler for task-parallel programs with implicit synchronization. To handle the full C language, we make several assumptions concerning data- and control-flow. Our pointer analysis assumes that all allocation in the program is done through the `libc` memory allocator functions and that no pointers are constructed from integers. We treat unsafe casts and pointer arithmetic conservatively and conflate the related memory locations. We perform field-inference for structs and type-inference for `void*` pointers to increase the precision of the pointer analysis. We currently assume there is no `setjmp/longjmp` control-flow.

The compiler is structured in three phases. The first extends the C front-end with support for OpenMP-like `#pragma` directives to define tasks and task footprints. We have chosen to mark task creation at the calling context, instead of marking a function definition and have every invocation of the function create a parallel task for better precision; this way we are able to call the same function both sequentially or as a parallel task without rewriting it or creating wrapper functions. The syntax for declaring task footprints supports strided memory access patterns, so that we can describe multidimensional array tiles as task arguments. When not explicitly given, we assume that the size of a task argument is the size of its type.

The second phase uses a type-system to generate points-to and control flow constraints and solves them to infer argument independence, as described in Section 3. In Section 3, however, we have made several simplifying assumptions to improve the presentation of the algorithm, that must be addressed when applying the analysis on the full C language.

Although in the formal presentation we do not differentiate between read and write effects in the task footprint, we actually treat input and output task arguments differently. In particular, we match the behavior of the runtime system, which allows multiple reader tasks of a memory location to run in parallel. Thus, we also mark task arguments that are only read in parallel as independent.

To increase the analysis precision, we use a context-sensitive, field sensitive points-to analysis, and a context-sensitive control flow analysis. In both cases, context sensitivity is encoded as CFL-reachability, with either points-to or control flow edges that enter or exit a calling context marked as special *open* or *close parenthesis* edges [10].

Finally, in several benchmarks tasks within loops access disjoint parts of the same array. However, the points-to analysis treats all array elements as one abstract location, producing false aliasing and causing such safe arguments to be missed. To rectify this, in part, we have implemented a simple loop-dependence analysis that discovers when different loop iterations access non-overlapping array elements. This (orthogonal) prob-

lem has been extensively studied in the past [11, 12], resulting in many techniques that can be applied to improve the precision of this optimization.

The final phase transforms the input program to use the runtime system to create tasks and perform dependence checks for task arguments not inferred or declared independent. As an optimization, the compiler produces custom code to interact with the runtime structures instead of using generic runtime API calls. In particular, for each `#pragma task` call, the compiler generates custom code that creates a task descriptor (closure) with the original function as task body, registers the task arguments with the runtime dependence analysis, and replaces the specified function call with the generated code.

We ran the resulting programs using the BDDT runtime [8, 13] to perform dynamic dependence analysis and check for any dependencies that are not ruled out statically. The BDDT runtime system maintains a representation of every task instance and its footprint at run time, and uses these to check for overlap among task arguments and compare their access properties to detect task dependencies. To do that, BDDT splits task arguments into virtual memory blocks of configurable size and analyzes dependencies between blocks. Similarly to whole-object dependence analysis used in tools such as SMPSs, SvS, and OoOJava, block-based analysis detects true read-after-write (RAW) dependencies, or write-after-write (WAW) and write-after-read (WAR) anti-dependencies between blocks, by comparing block starting addresses and checking their access attributes. We selected BDDT for our experiments due to its good performance and because it is easy to disable specific dynamic checks on specific task instances using its API. However, the SCOOP static independence analysis can be used to remove unnecessary dynamic checks from other task-parallel runtimes with implicit synchronization.

We used a region-based allocator to support dynamic memory allocation in tasks, and allow for tasks that operate on complex data structures. This way, we extend BDDT to handle task footprints that include dynamic regions specially: the task footprint language allows several task arguments to belong to a dynamic region; the task footprint then includes the region instead of the individual arguments, and SCOOP registers only the region descriptor with the dependence analysis.

5 Evaluation

We evaluated the effect of the static independence analysis in SCOOP on a set of representative benchmarks, including several computational kernels and small-sized parallel applications. We ran the experiments on a Cray XE6 compute node with 32GB memory and two AMD Interlagos 16-core 2.3GHz dual-processors, a total of 32 cores. We compiled all benchmarks with GNU GCC 4.4.5 using the -O3 optimization flag. As is standard in evaluation of task-parallel systems in the literature, we measured the performance of the parallel section of the code, excluding any initialization and I/O at the start and end of each benchmark. We used barriers to separate the initialization phase from the measured computation. Finally, to minimize variation among different runs, we report the average measurements over twenty runs for each benchmark.

We use the following benchmarks in our evaluation, listed in Table 1. *Black-Scholes* is a parallel implementation of a mathematical model for price variations in financial

Benchmark	LOC	Tasks	Total Args	Scalar Args	Analysis (s)	Graph Nodes	Safe Args
Black-Scholes	3564	1	8	1	3.17	1790	6
Ferret	30145	1	2	0	699.05	85128	2
Cholesky	1734	4	16	8	1.06	7571	0
GMRES	2661	18	72	20	2.21	7957	9
HPL	2442	11	59	35	1.47	9330	0
Jacobi	1084	1	6	0	0.74	3980	0
FFT	2935	4	12	4	1.72	9750	3
Multisort	1215	2	8	4	1.02	4016	0
Intruder	6452	1	5	1	19.89	16855	3

Table 1: Benchmark description and analysis performance

markets with derivative investment instruments, taken from the PARSEC [14] benchmark suite. *Ferret* is a content-based similarity search engine toolkit for feature rich data types (video, audio, images, 3D shapes, etc), from the PARSEC benchmark suite. *Cholesky* is a factorization kernel used to solve normal equations in linear least squares problems. *GMRES* is an implementation of the iterative Generalized Minimal Residual method for solving systems of linear equations. *HPL* solves a random dense linear system in double precision arithmetic. *Jacobi* is a parallel implementation of the Jacobian method for solving systems of linear equations. *FFT* is a kernel implementing a 2-dimensional Fourier algorithm, taken from the SPLASH-2 [15] benchmark suite. It is implemented in alternating transpose and computation phases. On each transpose the data gets reordered, creating irregular dependencies between the two phases. *Multisort* is a parallel implementation of Mergesort. Multisort is an alternative implementation of the Cilksort test from Cilk [1]. It has two phases: during the first phase, it divides the data into chunks and sorts each chunk. During the second phase, it merges those chunks. *Intruder* is a Signature-based network intrusion detection systems (NIDS), from the STAMP benchmark suite [16]. It processes network packets in parallel in three phases: capture, reassembly, and detection. The reassembly phase uses a dictionary that contains linked lists of packets that belong to the same session. The lists are allocated using dynamic regions. Intruder issues a new task for each packet it receives. When a task reassembles the last packet of a session it also executes the detection algorithm.

The second column (*LOC*) of Table 1 shows the size of each benchmark in lines of code[3]. The third column (*Tasks*) shows the number of task invocations in the code. The fourth column (*Total Args*) shows the total number of arguments of all task invocations. We report the total number of arguments as each such argument incurs the additional overhead of a runtime dependency check. The fifth column (*Scalar Args*) shows how many of those arguments are scalars passed by value, since it is trivial for either the programmer or the analysis to find them, thus we do not count them as *independent* arguments discovered by the static analysis.

The last three columns of Table 1 show the performance and precision of the static analysis. Namely, the sixth column (*Analysis*) shows the total running time of the static dependence analysis in seconds. The seventh column (*Graph Nodes*) shows the num-

[3] We count lines of code not including comments after merging all program sources in one file.

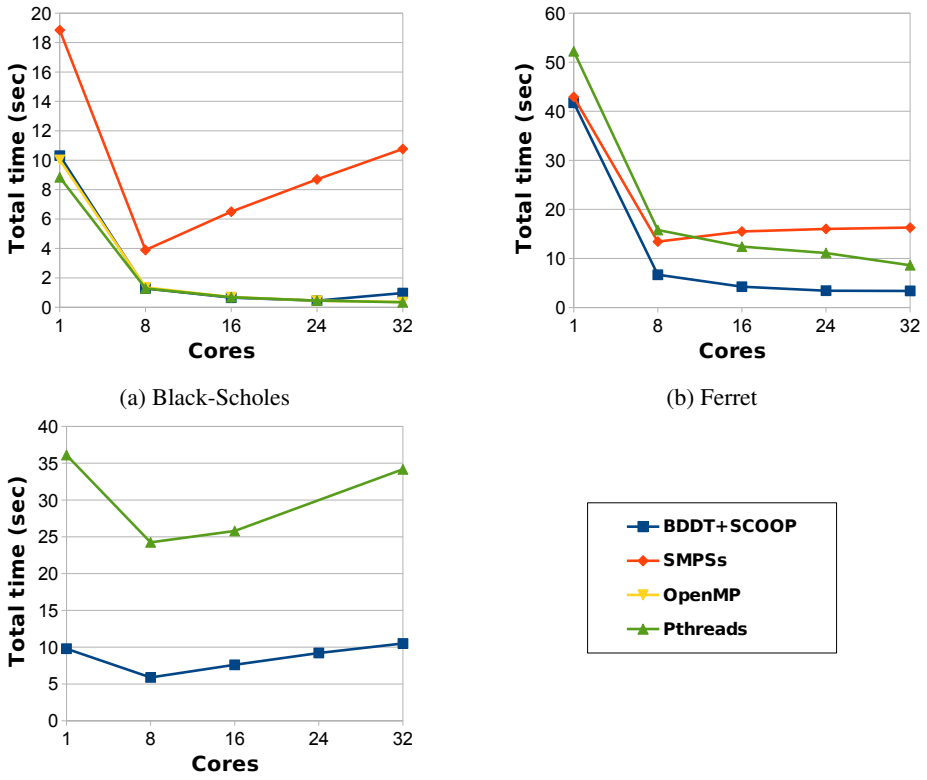
Benchmark	Task Instances	Dynamic (ms)	Standard Deviation	Static & Dynamic (ms)	Standard Deviation	Speedup
Black-Scholes	234375	1618	5.83 %	963	4.51 %	1.68
Ferret	1000	3344	1.10 %	3344	1.33 %	1.00
Cholesky	45760	983	0.23 %	981	0.32 %	1.00
GMRES	5170	16640	4.42 %	13947	2.65 %	1.19
HPL	28480	1628	0.44 %	1574	0.37 %	1.03
Jacobi	204800	11499	0.39 %	11588	0.53 %	0.99
FFT	28864	2028	0.10 %	1849	0.29 %	1.10
Multisort	11264	3683	15.77 %	3446	15.15 %	1.07
Intruder	$>$ 4M	12572	1.23 %	10332	1.76 %	1.22

Table 2: Impact of the analysis on performance

ber of nodes in the constraint graph. The last column (*Safe Args*) shows the number of independent task arguments inferred by the analysis. Note that Ferret, the largest benchmark, creates the largest constraint graph, causing an analysis time of over 11 minutes. This is because the context sensitive analysis has cubic complexity in the size of the constraint graph.

Table 2 shows the effect of the optimization on the total running time of all benchmarks, on 32 cores. Specifically, the second column (*Task Instances*) shows the total number of task spawns by each benchmark during execution. The third column (*Dynamic*) shows the total running time in milliseconds for each benchmark, without using the static analysis. Here, we use BDDT to perform runtime dependence analysis on *all* the non-scalar task arguments of all tasks. The fourth column (*Standard Deviation*) shows the standard deviation of the *Dynamic* total running time for twenty runs. The fifth column (*Static & Dynamic*) shows the total running time in milliseconds for each benchmark compiled with SCOOP, where the runtime dependence analysis is disabled for any arguments found safe by the static analysis. The sixth column (*Standard Deviation*) shows the standard deviation of the *Static & Dynamic* total running time. Finally, the last column shows the speedup factor gained by removing redundant checks for arguments found independent by the static analysis, compared to always checking arguments dynamically. Note that even though the static analysis does not infer independent task arguments in Multisort and HPL, we observe a speedup of 1.07 and 1.03 respectively when compiling with SCOOP compared to the dynamic-only execution. This happens because SCOOP generates code to interface directly with the BDDT runtime internals, whereas the BDDT API may perform various checks, e.g., on scalar arguments. We consider the 0.99 speedup (slowdown) in Jacobi to be well within the noise due to cache effects, other processes executing, etc., as seen by the deviation observed among twenty runs.

The dependence analysis is able to infer safe task arguments only in Black-Scholes, Ferret, GMRES, FFT and Intruder. In these benchmarks, inferring independent arguments has a large impact on the overhead and scalability of the dependence analysis, producing substantial speedup over the original BDDT versions for four benchmarks. The reduction of dependence analysis overhead is not noticeable in Ferret because the



(a) Black-Scholes

(b) Ferret

(c) Intruder

Fig. 4: Comparison with alternative runtimes

tasks are very coarse grain. On the rest of the benchmarks (Cholesky, HPL, Jacobi and Multisort) the dependence analysis fails to find any safe arguments. We examined all benchmarks manually and found that there are no safe arguments.

For reference, we compare the three largest benchmarks with related parallel runtimes. We have ported each benchmark to all runtimes so that they are as comparable as possible and express the same parallelism. Figure 4 shows the results. Specifically, Figure 4a compares four parallel implementations of Black-Scholes: the original Pthreads and OpenMP implementations from the PARSEC benchmark suite, as well as two ports of the OpenMP version into SMPSs and BDDT, using SCOOP. Note that SCOOP finds and removes the redundant dependency checks. As Black-Scholes is a data-parallel application, this removes most of the runtime overhead and matches in performance the fine-tuned Pthreads and OpenMP implementations. In comparison, the SMPSs runtime scales up to 8 cores, mainly due to the overhead caused by redundant checks on independent task arguments.

Figure 4b compares the original Pthreads implementation of Ferret with the SMPSs and BDDT runtime. The Pthread version uses one thread to run each computation phase, causing load imbalance. In comparison, BDDT and SMPSs perform dynamic

task scheduling that hides load imbalance and distributes computation to processors more evenly. Although SCOOP detects and removes redundant runtime checks, Ferret tasks are computationally heavy and coarse-grain, hiding the effect of the optimization. The difference in performance between SMPSs and BDDT is mainly due to constant-factor overheads in task scheduling.

Figure 4c compares the original Pthreads implementation of Intruder from the STAMP benchmark, with a port for BDDT. The STAMP implementation uses software transactional memory to synchronize threads, which causes high contention effects above 8 cores, limiting performance. In comparison, BDDT incurs lower overheads and uses pessimistic synchronization that also removes the cost of rollbacks.

6 Related Work

Task Parallelism: There are several programming models and runtime systems that support task parallelism. Most, like OpenMP [2], Thread Building Blocks [17], Cilk [1], and Sequoia [3], use tasks to express recursive or data parallelism, but require manual synchronization in the presence of task dependencies. That usually forces programmers to use locks, barriers, or other synchronization techniques that are not point-to-point, and result in loss of parallelism even among task instances that do not actually access the same memory.

Some programming models and languages aim to automatically infer synchronization among parallel sections of code. Transactional Memory [18] preserves the atomicity of parallel tasks, or transactions, by detecting and retrying any conflicting code. Static lock allocation [19] provides the same serializability guarantees by automatically inferring locks for atomic sections of code. These attempts, however, allow non-deterministic parallel executions, as they only enforce race freedom or serializability, not ordering constraints among parallel tasks.

Jade [20] is a parallel language that extends C with parallel coarse-grain tasks. Similarly, StarSs, SMPSs and OpenMP-Ss [9, 21] are task-based programming models for scientific computations in C that use annotations on task arguments to dynamically detect argument dependencies between tasks. All of these runtimes could benefit from independencies discovered by SCOOP to reduce the overhead of runtime checks.

Static analysis has been used in combination with dynamic analysis in parallel programs in the past. SvS [7] uses static analysis to determine possible argument dependencies among tasks and drive a runtime-analysis that computes task dependencies with overlapping approximate footprints. SvS assumes all tasks to be commutative and does not preserve the original program order as SCOOP and BDDT. Prabhu et al. [22] define sets of commutative tasks in parallel programs. The compiler uses this information to allow more possible orderings in a program and extract parallelism. As with all compiler-only parallelization techniques, this approach is limited by over-approximation in static pointer and control flow analyses that might cause many tasks to be run sequentially, because only two instances have clearly disjoint memory footprints. To avoid this, CommSets uses optimistic transactional memory, which is not suitable for programs with high contention or effects that cannot be rolled back.

Deterministic parallelism: Recent research has developed methods for the deterministic execution of parallel programs. Kendo [23] enforces a deterministic execution for

race-free programs by fixing the lock-acquisition order, using performance counters. Grace [24] produces deterministic executions of multithreaded programs by using process memory isolation and a strict sequential-order commit protocol to control thread interactions through shared memory. DMP [25] uses a combination of hardware ownership tracking and transactional memory to detect thread interactions through memory. Both systems produce deterministic executions, even though they may not be equivalent to the sequential program. Instead, they enforce the appearance of the same arbitrary interleaving across all executions.

Out-of-Order Java [5] and Deterministic Parallel Java [6], task-parallel extensions of Java. They use a combination of data-flow, type-based, region and effect analyses to statically detect or check the task footprints and dependencies in Java programs. OoOJava then enforces mutual exclusion of tasks that may conflict at run time; DPJ restricts execution to the deterministic sequential program order using transactional memory to roll back tasks in case of conflict. As task footprints are inferred (OoOJava) or checked (DPJ) statically in terms of objects or regions, these techniques require a type-safe language and cannot be directly applied on C programs with pointer arithmetic and tiled array accesses.

Chimera [26] proposes a hybrid system that detects and transforms races, so that the runtime system can then enforce deterministic execution.

Static and Dynamic Dependence Analysis: Static dependence analysis is often employed in compilers and tools that optimize existing parallel programs or for automatic parallelization. Early parallelizing compilers used loop dependence analysis to detect data parallelism in loops operating on arrays [12, 27], and even dynamic dependence analysis to automatically synchronize loops [28] These systems, however do not handle inter-loop dependencies and do not work well in the presence of pointers. Recently, Holewinski et al. [29] use dynamic analysis of the dynamic dependence graph of sequential execution to detect SIMD parallelism.

Several pointer analyses have been used to detect dependencies and interactions in parallel programs. Naik and Aiken [30] extend pointer analysis with *must-not-alias* analysis to detect memory accesses that cannot lead to data races. Pratikakis et al. [31, 32] use a context-sensitive pointer and effect analysis to detect memory locations accessed by many threads in the Locksmith race detector. SCOOP uses the pointer analysis in Locksmith to detect aliasing between task footprints, and extends Locksmith's flow-sensitive dataflow analysis to detect tasks that can run in parallel.

7 Conclusions

This paper presents SCOOP, a compiler for a task-parallel extension of C with implicit synchronization. SCOOP targets task parallel runtimes such as BDDT and OpenMP-Task, that use dynamic dependence analysis to automatically synchronize and schedule parallel tasks. SCOOP uses static analysis to infer safe task arguments and reduce the runtime overhead for detecting dependencies. We have tested SCOOP using the BDDT runtime system on a set of parallel benchmarks, where it finds and removes unnecessary runtime checks on task arguments. Overall, we believe that task dependence analysis is an important direction in parallel programming abstractions, and that using static analysis to reduce its overheads is a major step in its practical application.

Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7), under the ENCORE Project in Computing Systems (www.encore-project.eu), grant agreement N^o 248647, and the HiPEAC Network of Excellence (www.hipeac.net), grant agreement N^o 287759.

References

1. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: Proceedings of the ACM symposium on Principles and Practice of Parallel Programming. (1995)
2. Dagum, L., Menon, R.: OpenMP: An industry-standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5** (January 1998)
3. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. SC '06, New York, NY, USA, ACM (2006)
4. Pop, A., Cohen, A.: OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. ACM Transactions on Architecture and Code Optimization **9**(4) (January 2013) 53:1–53:25
5. Jenista, J.C., Eom, Y.H., Demsky, B.: OoOJava: Software out-of-order execution. In: Proceedings of the ACM symposium on Principles and Practice of Parallel Programming. (2011)
6. Bocchino, Jr., R.L., Heumann, S., Honarmand, N., Adve, S.V., Adve, V.S., Welc, A., Shpeisman, T.: Safe nondeterminism in a deterministic-by-default parallel language. In: Proceedings of the ACM symposium on Principles Of Programming Languages. (2011)
7. Best, M.J., Mottishaw, S., Mustard, C., Roth, M., Fedorova, A., Brownsword, A.: Synchronization via scheduling: Techniques for efficiently managing shared state. In: Proceedings of the ACM conference on Programming Language Design and Implementation. (2011)
8. Tzenakis, G., Papatriantafyllou, A., Vandierendonck, H., Pratikakis, P., Nikolopoulos, D.S.: BDDT: Block-level dynamic dependence analysis for task-based parallelism. In: International Conference on Advanced Parallel Processing Technology. (2013)
9. Pérez, J.M., Badia, R.M., Labarta, J.: Handling task dependencies under strided and aliased references. In: Proceedings of the International Conference on Supercomputing. (2010)
10. Rehof, J., Fähndrich, M.: Type-base flow analysis: from polymorphic subtyping to cfl-reachability. In: Proceedings of the ACM symposium on Principles Of Programming Languages. (2001)
11. Wolf, M.E.: Improving locality and parallelism in nested loops. PhD thesis, Stanford University, Stanford, CA, USA (1992)
12. Maydan, D.E., Hennessy, J.L., Lam, M.S.: Efficient and exact data dependence analysis. In: Proceedings of the ACM conference on Programming Language Design and Implementation. (1991)
13. Tzenakis, G., Papatriantafyllou, A., Zakkak, F., Vandierendonck, H., Pratikakis, P., Nikolopoulos, D.S.: BDDT: Block-level dynamic dependence analysis for deterministic task-based parallelism. Tech Report 426, FORTH (February 2012)
14. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: Characterization and architectural implications. In: International Conference on Parallel Architectures and Compilation Techniques. (2008)
15. Woo, S., Ohara, M., Torrie, E., Singh, J., Gupta, A.: The splash-2 programs: Characterization and methodological considerations. In: Proceedings of the International Symposium on Computer Architecture. (1995)

16. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IEEE International Symposium on Workload Characterization. (2008)
17. Reinders, J.: Intel threading building blocks. First edn. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2007)
18. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the International Symposium on Computer Architecture. (1993)
19. Cherem, S., Chilimbi, T., Gulwani, S.: Inferring locks for atomic sections. In: Proceedings of the ACM conference on Programming Language Design and Implementation. (2008)
20. Rinard, M.C., Lam, M.S.: The design, implementation, and evaluation of Jade. ACM Transactions on Programming Languages and Systems **20**(3) (1998) 483–545
21. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Hierarchical task-based programming with StarSs. International Journal of High Performance Computing Applications **23**(3) (2009) 284–299
22. Prabhu, P., Ghosh, S., Zhang, Y., Johnson, N.P., August, D.I.: Commutative set: A language extension for implicit parallel programming. In: Proceedings of the ACM conference on Programming Language Design and Implementation. (2011)
23. Olszewski, M., Ansel, J., Amarasinghe, S.: Kendo: efficient deterministic multithreading in software. In: International Conference on Architectural Support for Programming Languages and Operating Systems. (2009)
24. Berger, E.D., Yang, T., Liu, T., Novark, G.: Grace: safe multithreaded programming for c/c++. In: Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages, and Applications. (2009)
25. Devietti, J., Nelson, J., Bergan, T., Ceze, L., Grossman, D.: Rcdc: a relaxed consistency deterministic computer. In: International Conference on Architectural Support for Programming Languages and Operating Systems. (2011)
26. Lee, D., Chen, P.M., Flinn, J., Narayanasamy, S.: Chimera: hybrid program analysis for determinism. In: Proceedings of the ACM conference on Programming Language Design and Implementation. (2012)
27. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM **8** (1992) 4–13
28. Rauchwerger, L., Padua, D.: The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: Proceedings of the ACM conference on Programming Language Design and Implementation. (1995)
29. Holewinski, J., Ramamurthi, R., Ravishankar, M., Fauzia, N., Pouchet, L.N., Rountev, A., Sadayappan, P.: Dynamic trace-based analysis of vectorization potential of applications. In: Proceedings of the ACM conference on Programming Language Design and Implementation. (2012)
30. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: Proceedings of the ACM symposium on Principles Of Programming Languages. (2007)
31. Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: Practical static race detection for C. ACM Transactions on Programming Languages and Systems **33** (January 2011) 3:1–3:55
32. Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: context-sensitive correlation analysis for race detection. In: Proceedings of the ACM conference on Programming Language Design and Implementation. (2006)