Generating Fine-Grain Multithreaded Applications Using a Multigrain Approach

JAIME ARTEAGA, University of Delaware
STÉPHANE ZUCKERMAN, Michigan Technological University
GUANG R. GAO, University of Delaware

The recent evolution in hardware landscape, aimed at producing high-performance computing systems capable of reaching extreme-scale performance, has reignited the interest in fine-grain multithreading, particularly at the intranode level. Indeed, popular parallel programming environments, such as OpenMP, which features a simple interface for the parallelization of programs, are now incorporating fine-grain constructs. However, since coarse-grain directives are still heavily used, the OpenMP runtime is forced to support both coarse- and fine-grain models of execution, potentially reducing the advantages obtained when executing an application in a fully fine-grain environment. To evaluate the type of applications that benefit from executing in a unified fine-grain program execution model, this article presents a *multigrain parallel programming environment* for the generation of fine-grain multithreaded applications from programs featuring OpenMP's API, allowing OpenMP programs to be run on top of a fine-grain event-driven program execution model. Experimental results with five scientific benchmarks show that fine-grain applications, generated by and run on our environment with two runtimes implementing a fine-grain event-driven program execution model, are competitive and can outperform their OpenMP counterparts, especially for data-intensive workloads with irregular and dynamic parallelism, reaching speedups as high as 2.6× for Graph500 and 51× for NAS Data Cube.

CCS Concepts: • **Computing methodologies** → Parallel programming languages; **Concurrent programming languages**; • **Software and its engineering** → **Compilers**;

Additional Key Words and Phrases: Codelets, DARTS, extreme-scale computing, multigrain parallel compiler, runtime, SWARM

ACM Reference format:
Jaime Arteaga, Stéphane Zuckerman, and Guang R. Gao. 2017. Generating Fine-Grain Multithreaded Applications Using a Multigrain Approach. *ACM Trans. Archit. Code Optim.* 14, 4, Article 47 (December 2017), 26 pages.
https://doi.org/10.1145/3155288

Extension of conference paper (Arteaga et al. 2017). New content includes a more detailed presentation of the proposed compiler techniques and an expanded experimental section with an additional fine-grain runtime.
This work was partially supported by the National Science Foundation, under award XPS-1439097.
Authors' addresses: J. Arteaga and G. R. Gao, University of Delaware, Department of Electrical and Computer Engineering, Room 140 Evans Hall, 139 The Green, Newark, DE 19716, USA; emails: jaime@udel.edu, ggao.capsl@gmail.com; S. Zuckerman, Michigan Technological University, Department of Computer Science, Rekhi Hall 1400 Townsend Drive, Houghton, MI, 49931, USA; email: szuckerm@mtu.edu.

https://doi.org/10.1145/3155288

1 INTRODUCTION

Current and upcoming changes in hardware, and in particular the advent of "true" general-purpose manycore chips (e.g., Intel's latest Xeon and IBM's POWER8 processors offering close to a hundred processing elements) designed to help supercomputers reach extreme-scale performance, must be accompanied by a paradigm shift in the programming models (PM) and program execution models (PXM) used to write and execute parallel applications (DoE 2010).[1] This has renewed the interest in fine-grain PMs and PXMs because of their potential in better handling the high core count of shared-memory nodes of future extreme-scale systems and in executing efficiently modern workloads with dynamic and unstructured parallelism, like those containing unbounded loops and recursive functions. One such PXM is the *Codelet Model* (Zuckerman et al. 2011), a hybrid dataflow/von Neumann fine-grain event-driven PXM that provides programmers with a comprehensive application programming interface (API) to develop parallel programs with several levels of granularity and parallelism. *Delaware Adaptive Runtime System (DARTS)* (Suettlerlein et al. 2013) and *SWift Adaptive Runtime Machine (SWARM)* (Lauderdale and Khan 2012) are some of the runtimes that have been developed based on the Codelet Model. They differ on how relaxed their implementations of the Codelet Model are, their APIs, and their hardware support, among other features.

Fine-grain multithreading has also been added to legacy PMs, like OpenMP's (OpenMP4.0 2011). OpenMP is a parallel programming environment, comprising three aspects: a set of compiler directives, a runtime library, and a set of environment variables accessible at execution time by the user. OpenMP is particularly attractive because of its straightforward and simple interface that can be added incrementally to sequential code. At first, OpenMP focused on offering coarse-grain constructs for workloads with structured parallelism (e.g., numerical applications with `for` loops). But recent versions of the standard now also include fine-grain constructs, named `tasks`, which are created and scheduled dynamically at runtime and are more suitable for modern workloads with unstructured parallelism.[2]

While the enrichment of OpenMP's programming model is most welcome, it results in a significant increase in complexity with respect to its PXM. Indeed, OpenMP must support both coarse-grain and fine-grain models of execution and allow them to coexist during execution *at the same time*. We believe that this may be counterproductive when executing certain applications, such as those with irregular and dynamic workloads, which may be more likely to achieve maximum performance when running on top of a unified fine-grain environment. Moreover, we posit "regular" coarse-grain constructs in the PM can be safely converted to fine-grain ones in the PXM with little or no loss in performance.

It is then important to provide developers with tools that allow them to determine the execution model that is most appropriate for each application while keeping, at a high level, a common and well-known interface to write parallel programs. This article presents a *multigrain parallel programming environment* designed around the Codelet Model for the fine-grain execution of OpenMP programs. This environment takes an OpenMP program and generates an equivalent fine-grain event-driven application, either in DARTS or SWARM, to fully exploit the high core count available in the underlying hardware. By using this environment, programmers will be able to keep using OpenMP's simple and familiar API and, at the same time, benefit from the

[1] In this article, we define *Programming Model* (PM) as the way to express *what* parallel constructs and behaviors can be expressed by the programmer, whereas a *program execution model* (PXM) expresses *how* such parallel behaviors are created and destroyed, how they are orchestrated, and how they react when faced with memory accesses (shared or distributed).
[2] The keyword `depends` is also an important addition to the standard as it provides a data-driven way of expressing parallelism.

advantages offered by a model specifically designed for fine-grain execution. Moreover, this environment may help programmers transition into the use of fine-grain PXMs to write parallel applications for many and multicore architectures.

The major component in our environment is *omp2cd*, a *multigrain parallel compiler* based on *clang*, which takes an input OpenMP program written in C and transforms it into a fine-grain application, automatically determining tasks' granularities at several levels of parallelism. This is done depending on the type of parallelism present in the application, as well as on the OpenMP directives and compiler hints in the form of OpenMP clause extensions used by the programmer.

This article presents the following contributions:

- *omp2cd*, a multigrain parallel compiler for the generation of fine-grain applications from OpenMP programs to be run on top of the Codelet Model, using DARTS or SWARM.
- An analysis of the parameters and runtime options available in DARTS to fine-tune its highly configurable *abstract machine model* (AMM) according to the application's features and underlying hardware.
- Experimental evaluation of the multigrain parallel programming environment using a set of scientific benchmarks, showing that the fine-grain applications generated by our environment are highly competitive and can outperform their OpenMP counterparts.

This article is structured as follows: Section 2 introduces the Codelet Model and two fine-grain runtimes implementing it: DARTS, and SWARM. Section 3 presents the multigrain parallel programming environment and its compiler, as well as some of the new optimizations introduced to DARTS during the development of this work. Section 4 exposes the experimental evaluation of the environment. Section 5 discusses some of the related work found in the literature. The article closes discussing the major conclusions derived from this work and future work.

2 BACKGROUND

2.1 The Codelet Model

The Codelet Model is a fine-grain event-driven PXM designed to fully exploit architectures featuring a high degree of parallelism by decomposing an application into a great number of lightweight asynchronous tasks called *codelets*. Codelets can be easily scheduled among cores, depending on resource availability, proximity to data to compute, and other constraints (Zuckerman et al. 2011).

Each codelet has a collection of data inputs (respectively, outputs) coming from (respectively, going to) other codelets. Additionally, a codelet may require a set of specific resources for its execution. When both data and resource dependencies are satisfied, a codelet is said to be *ready* and is scheduled for non-preemptive execution, potentially reducing the overhead produced by context switching. Also, since codelets only act on their data inputs and outputs, which are local, long-latency memory operations are avoided and power consumption decreased. Interactions between codelets are described in the form of a *Codelet Graph (CDG)*, a directed graph, where nodes represent codelets and arcs the dependencies between them. CDGs are usually defined statically,[3] leaving to the runtime the instantiation and execution of TPs and codelets upon satisfaction of dependencies.

Codelets may be grouped into *Threaded Procedures* (TPs). A TP is an asynchronous function that acts as a container for a CDG and the data accessed by its codelets. This provides a second level of parallelism in the Codelet Model that can be exploited for increasing locality of data. The number

[3]Being statically determined does not mean the codelet is generated at compile time. However, once a given CDG will be *fully* specified when it is instantiated (i.e., according to the model), a codelet cannot be dynamically added to an already existing CDG.

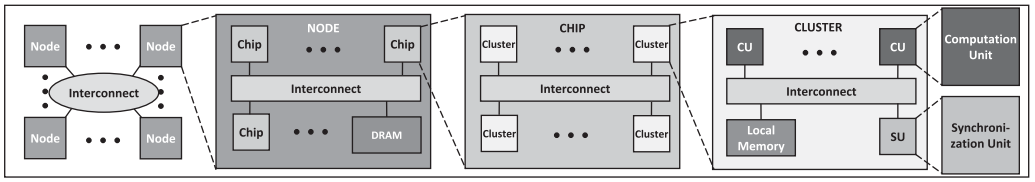


Fig. 1. Codelet model's Abstract Machine Model (AMM).

of TPs within an application, as well as the number of codelets per TP and statements per codelet, may vary, providing users with the ability to decompose an application with several degrees of granularity at different levels of parallelism.

The Codelet Model relies upon an *Abstract Machine Model* (AMM), shown in Figure 1, which is mapped at runtime to the target many-core system. The system is hierarchically decomposed into compute nodes, each featuring at least one chip. Each chip is composed of *clusters* of cores. Cores can be either *Synchronization Units* (SUs) or *Compute Units* (CUs), with each cluster containing one SU and zero or more CUs. A SU is in charge of allocating TPs and scheduling codelets to the CUs within the same cluster. Additionally, an SU may execute a codelet if all of its associated CUs are busy. CUs execute codelets and send TP allocation requests to their local SUs. Nevertheless, and depending on the runtime implementation, such requests may be ultimately fulfilled by an SU in a different cluster. Once a TP has been officially allocated by a given cluster, the codelets it contains cannot migrate outside of it. This guarantees a certain degree of locality for code execution.

2.2 Delaware Adaptive Runtime System - DARTS

Delaware Adaptive Runtime System (DARTS) is currently the most faithful runtime implementation of the Codelet Model and its AMM (Suettlerlein et al. 2013). It is written in C++ and distributed as free and open-source software (CAPSL 2013). In DARTS, TPs and codelets are defined as class objects at compile time and are instantiated at runtime. A TP is instantiated when a codelet performs a *TP invocation*. This operation creates a TP handle and sends a request to the runtime to allocate memory for the TP instance and the instances of the codelets it contains. Once this is done, codelet instances are executed upon satisfaction of their dependencies.

A two-level scheduling mechanism, comprising TP schedulers and microschedulers, is used for the scheduling and execution of codelets. A TP scheduler is a core which has been assigned the role of SU and has a TP queue (TPQ) with TP instantiation requests and a ready-codelets queue (RCQ). A microscheduler is a core acting as a CU with a local codelet queue (LCQ), which is fed by its local TP scheduler with codelets from its RCQ.

DARTS allows users to determine the "shape" of the AMM by setting the number of clusters in the machine and the number of CUs associated with an SU. This can be done based on the topology of the target architecture, such as the number of sockets or NUMA nodes. Hence, the two-level parallelism exposed by the Codelet Model is preserved and configurable in DARTS.

Users can also fine-tune DARTS AMM by setting the scheduler affinity policy as `COMPACT` (SUs and CUs are pinned down to physically contiguous cores), `SPREAD` (schedulers are assigned in a round-robin fashion to the different sockets of the platform), or user-defined using an environment variable that follows the same notation as GCC's `GOMP_CPU_AFFINITY`.

In addition, different scheduling policies are available: namely, `standard` and `work-stealing`. With a standard policy, a TP scheduler only initializes TPs assigned locally by one of its microschedulers and distributes codelets in round-robin fashion to them. With work-stealing, on the other hand, a TP scheduler may also steal TP handles from another TP scheduler's TPQ. A

microscheduler with standard policy only executes codelets pushed into its local queue by its TP scheduler. However, if its policy is work-stealing, it may steal codelets from the TP scheduler's RCQ. Other new optimizations available to fine-tune the AMM that have been developed in conjunction with the design of the environment are presented in Section 3.5.

2.3 SWift Adaptive Runtime Machine - *SWARM*

The *SWift Adaptive Runtime Machine (SWARM)* is a commercial implementation of the Codelet Model written in C and developed by E.T. International, Inc. (Lauderdale and Khan 2012). As with DARTS, the basic unit of computation is also a codelet, which runs to completion without blocking. However, unlike DARTS, SWARM allows codelets to be preempted in certain circumstances, such as avoiding having a codelet running infinitely. TPs are not defined in SWARM, and fine-grain applications are described only in terms of codelets (this contrasts with DARTS, where the use of TPs is enforced by its PM). Instead, SWARM uses the concept of *codelet complexes*, defined as a set of codelets that work cooperatively to complete a specific task and that share a private context frame. SWARM also defines *locales* to describe the hardware segments on which the runtime environment is divided. With locales, programmers can specify the processing element where a codelet should be executed or the memory area where an object should reside.

Locales are organized hierarchically and each one is associated with a scheduler, which is in charge of managing and scheduling work for the hardware associated with the locale. When a codelet is ready to run, it is placed in the scheduler's queue and then executed when a thread in the scheduler's locale is available. If, on the other hand, the scheduler is idle and there are no ready codelets to run, it may steal work from other schedulers in the locale's hierarchy. If no work is found, the thread associated with the scheduler may be placed to sleep until there is work to do.

3 MULTIGRAIN PARALLEL PROGRAMMING ENVIRONMENT

This section presents two of the main components of our multigrain parallel programming environment for the fine-grain execution of OpenMP programs. The first one is *omp2cd*, a multigrain parallel compiler which generates a fine-grain codelet application targeting a runtime implementation of the Codelet Model, either DARTS or SWARM, depending on the command line options used during compilation. The second component is an optimized version of the DARTS runtime, which we have tuned-up during the development of the environment to improve the performance of DARTS applications with respect to OpenMP. For SWARM, the implementation provided and extensively optimized by E.T. International, Inc. is used.

3.1 Compiler Overview

For the generation of fine-grain applications from OpenMP programs, the multigrain parallel compiler *omp2cd* was designed. The compiler is based on clang 3.9, accepts input source files in C language, and supports OpenMP v2.5 (partially[4]), as well as the `task` and `taskwait` constructs defined in v3.0 and the `depend` clause from v4.0 (OpenMP4.0 2011).

A diagram of the compiler's internal structure is presented in Figure 2. The compiler takes a set of OpenMP C source files and uses clang's front-end to generate an Abstract Syntax Tree (AST) for each translation unit. Each AST is then visited and annotated with information needed by other compiler's stages, such as the OpenMP directives and clauses used by the programmer, control flow patterns in the code (e.g., selection, iteration, fork-join), and synchronization points, among others. These annotations allow the compiler to later determine how to implement

[4]Missing support from OpenMP v2.5 includes the `workshare` and `ordered` directives and runtime routines to handle nestable locks. Work is currently being done to add them to the compiler.

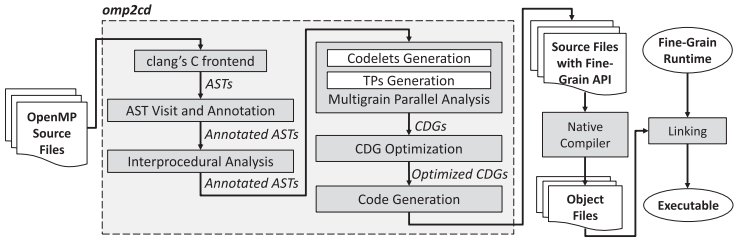


Fig. 2. *omp2cd*'s internal structure and interaction with other components of the environment.

each section of code in the input program (either as a collection of TPs or codelets), to generate the correct set of statements to invoke TPs and signal codelets, and to apply certain available optimizations.

The annotated ASTs then go through an *interprocedural analysis*, which identifies the functions containing `omp` executable directives and/or being called from `omp` regions, to correctly implement these functions and function calls using the target fine-grain programming model's API. In the case of DARTS, for example, a function may be converted into a TP, and every call to such function is translated into a TP invocation.

The next stage takes the annotated ASTs and translates them into a collection of CDGs by grouping AST nodes into codelets. This is done through a *multigrain parallel analysis* (Section 3.2) based on the annotations made previously in the ASTs. This stage also creates upper levels of parallelism, like grouping codelets into TPs, provided the target runtime selected by the user supports it, like DARTS.

A *multigrain optimization* stage (see Section 3.3) transforms the CDGs at different levels of parallelism to improve the application's performance in a fine-grain execution environment. Last, the optimized CDGs are translated into source files containing the target API, either DARTS or SWARM. In the case of DARTS CDGs, the user can then configure the abstract machine at runtime to better suit locality and/or affinity requirements and further improve performance.

All mechanisms and optimizations available on DARTS are currently supported by *omp2cd* and integrated in the output applications. When using SWARM as a back-end, we only use regular codelets—*codelet complexes*, *codelet chaining*, and the use of *locales*, will be included in future versions of the compiler.

3.2 Creation of CDGs Using a Multigrain Parallel Analysis

To generate CDGs from the annotated ASTs, *omp2cd* uses a multigrain parallel analysis that follows the specifications of the Codelet Model. This analysis defines the number and granularities of the parallel actors to be used by the output application to execute the input OpenMP program in a fine-grain manner. This is done at two levels of parallelism: The top level corresponds to TPs, with their granularity defined as the number of codelets a TP contains; the bottom level comprises codelets, with each codelet's granularity being the number of statements to execute when fired.

Generation of TPs and codelets from each annotated AST is made according to the following set of rules. These have been defined based on an extensive study of the Codelet Model and of handwritten applications written in DARTS and SWARM, like those used by Suettlerlein et al. (2013) and those contained in the form of examples in the release versions of the runtimes. After applying these rules on the ASTs of a valid OpenMP program, a set of CDGs is created, which are later optimized.

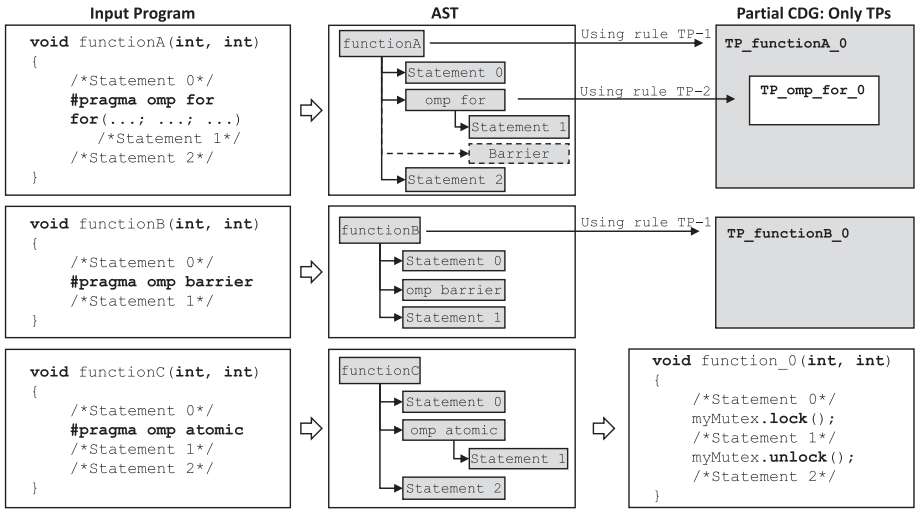


Fig. 3. Generation of TPs based on the rules defined by *omp2cd*. The `for` region's implicit barrier in `functionA` is shown in a dotted rectangle since it is not actually present in the code. The atomic statement in `functionC` is implemented using locks or GCC atomic built-ins, depending on the data type being processed.

3.2.1 Rules to Define TPs. To determine the nodes for which a TP must be used, the compiler traverses each annotated AST in depth-first order and creates a TP definition for each node containing:

TP- 1: The declaration of a function, other than `main()`, that contains an `omp` executable directive.[5]
TP- 2: An `omp` region.[6]

Both rules ignore `atomic` and `critical` regions since they are translated into atomic statements or protected sections of code using locks. If a function contains at least a `barrier` or `taskwait` directive, such function is converted into a TP to implement the synchronization construct. However, these directives are not translated into TPs since they are stand-alone directives with no associated code. Instead, they are implemented as codelets, as shown later.

The generation of TPs from a program's AST using these rules is shown with three examples in Figure 3, where TPs are represented with rectangles. In the first example, `functionA` is a function containing an `omp for` region. Following rule TP-1, a TP is defined for this function; the same occurs for the `omp for` region according to rule TP-2. The TP defined for the `omp for` region is a child of the function's TP since that is the same relationship their nodes have in the AST. A TP is also defined for `functionB` as defined by rule TP-1. No TP is defined for its `omp barrier` directive since the compiler uses only codelets for its implementation. Finally, `functionC` comprises an `omp atomic` region. This is implemented using locks or GCC atomic built-ins depending on the data type of the variable being updated. Since no codelets are used for this region, there is no need to define a TP to group them. Consequently, the function is not translated into a TP but left in sequential form.

[5] An `omp` *executable directive* is "An OpenMP directive that...may be placed in an executable context (OpenMP4.0 2011)."
[6] An `omp` *region* is "All code encountered during a specific instance of the execution of a given construct...," while an `omp` *construct* is "An OpenMP executable directive...and the associated statement, loop or structured block, if any..." (OpenMP4.0 2011).

If the target fine-grain runtime does not support TPs (e.g., SWARM), TPs are not created but the associated nodes are further annotated for subsequent compiler stages.

3.2.2 Rules to Define Codelets. The compiler defines codelets by traversing the annotated ASTs and grouping statement nodes using the concept of *Codelet Basic Blocks (CBB)*. A CBB is a set of statements that can be executed *by the same codelet* and whose *leader* (i.e., the first statement in the CBB) is defined by the presence of nodes annotated as TPs or that contain synchronization directives. Once a leader has been found, the statements that follow it are attached to the newly created CBB until the leader of a new CBB is defined. Each CBB is assigned a consecutive ID and a new codelet, which executes the CBB's statements when fired.

CBB's leaders are defined according to the following rules, which, as with the rules to create TPs, also ignore `atomic` and `critical` regions. An AST statement is a CBB leader if it is:

CBB- 1: An `omp` executable directive.
CBB- 2: A call to a function containing an `omp` executable directive.
CBB- 3: The first statement in an `omp` region.
CBB- 4: The first statement in a function, other than `main()`, containing an `omp` executable directive.
CBB- 5: The first statement of a branch, provided any of the branch's parent nodes are part of a CBB.
CBB- 6: The implicit barrier of an `omp` region.
CBB- 7: The statement following a CBB whose leader was created using rules CBB-1, CBB-2, or CBB-6.

Depending on the rule used to determine its leader, a new codelet is defined for the CBB. The codelet may be one of the following codelet categories:

- *Invocation Codelet:* Created for CBBs containing a TP invocation. This codelet is also used to monitor the threads that have reached the point in the program where the TP is invoked and are allowed to continue: *Rules CBB-1, CBB-2*. Since SWARM does not define TPs, SWARM applications do not contain invocation codelets. The use of these codelets during the instantiation of a TP is explained in Section 3.2.4.
- *Computation Codelet:* Used for CBBs whose statements are neither associated with TP invocations nor synchronization constructs: *Rules CBB-3, CBB-4, CBB-5, and CBB-7*.
- *Synchronization Codelet:* Defined for CBBs representing a synchronization or task scheduling point in the program, such as `barrier` and `taskwait` directives: *Rule CBB-6*.

Invocation and computation codelets may be instantiated up to N times for concurrent execution, with N being the number of OpenMP threads defined for the current region. Each codelet instance is assigned an ID from 0 to $N - 1$. Synchronization codelets may be instantiated only once or implemented as a tree to reduce signaling overhead.

Figure 4 shows how CBBs and codelets are defined for the AST of a given OpenMP program (the definition of dependencies between codelets is explained in Section 3.2.3). The figure presents a function containing an `omp for` a region and two sets of statements placed before and after such region. As with `functionA` in Figure 3, `functionD` in Figure 4 is translated into a TP using rule TP-1. Then, following rule CBB-4, a new CBB is defined for `Statement 0`, which becomes the CBB's leader. `Statement 1` is attached to this CBB since no rule applies to it. Because the rule used was CBB-4, the codelet defined for this CBB is a computation one.

The `omp for` region generates a new TP, according rule TP-2. In order to invoke it, this directive is translated into a new CBB according to rule CBB-1, which is implemented using an invocation

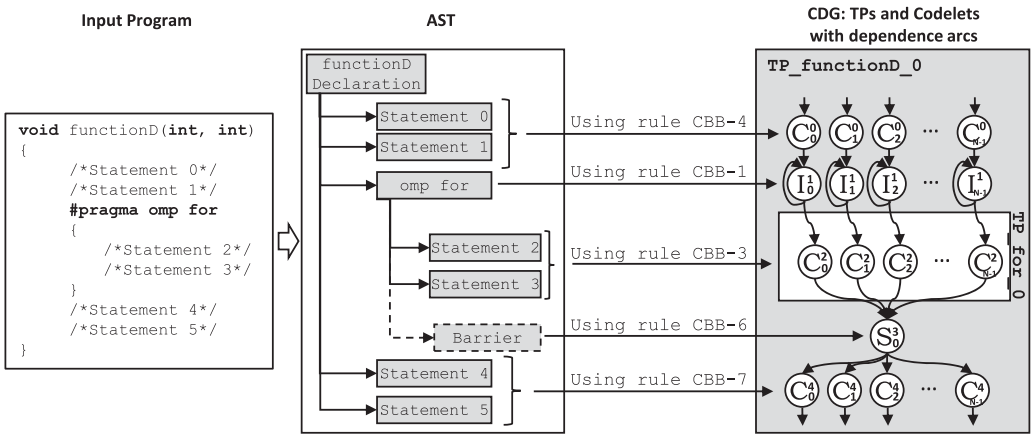


Fig. 4. Generation of codelets from the AST of an OpenMP program following the rules to define a CBB's leaders. TPs are represented with rectangles, while circles are used for codelets. Superscripts represent CBB's IDs; subscripts represent instance's IDs from 0 to $N-1$, with N being the number of OpenMP threads in the region. The `for` region's implicit barrier is shown in a dotted rectangle since it is not actually present in the code, but it is used by the compiler to define a synchronization codelet.

codelet. The statements in the `for` region are grouped into a new CBB, for which `Statement 2` is defined as leader, following rule CBB-3. For the region's implicit barrier, a CBB with a synchronization codelet is created, as indicated by rule CBB-6. Finally, the two statements that follow the region are grouped into a new CBB, implemented as a computation codelet whose leader corresponds to `Statement 4` according to rule CBB-7.

3.2.3 Definition of Dependencies between Codelets. To correctly define the dependencies between codelets, *omp2cd* keeps track of the AST's locations where CBB's leaders have been set and the codelet's category (i.e., invocation, computation, or synchronization codelet) assigned to each CBB. With this information, the compiler annotates CBBs with the set of codelets they must signal, as well as with the number of data dependencies that must be satisfied before their execution.

When the compiler defines a CBB's leader (and, consequently, a codelet), AST statements are attached to the current CBB until finding a new leader due to a TP invocation or synchronization point. When this occurs, the compiler creates a dependence arc between the current codelet (i.e., the predecessor codelet) and the new one (i.e., the dependent codelet) to maintain the program's flow and its correctness. In the case any of these codelets is meant to be instantiated more than once at execution time (which depends on the codelet's category), the compiler adjusts the number of dependencies each codelet must be initialized with.

For instance, if an arc is defined between a computation codelet and an invocation one (or vice versa), the number of dependencies for both is set to one. This dependency is satisfied by the instance of the predecessor codelet with the same ID as the instance for the dependent codelet, since both computation and invocation codelets are duplicated equally up to the number of OpenMP threads defined for the current region. This can be seen in Figure 4, where there are dependency arcs between computation codelet C_0^0 and invocation codelet I_0^1 (which have the same instance $ID = 0$), between codelets C_1^0 and I_1^1 (instance $ID = 1$), and so on. At runtime, an invocation codelet may also reset its dependence counter to one and satisfy it itself with a back edge to allow the fine-grain application to perform the instantiation of TPs without blocking the codelet (see Section 3.2.4).

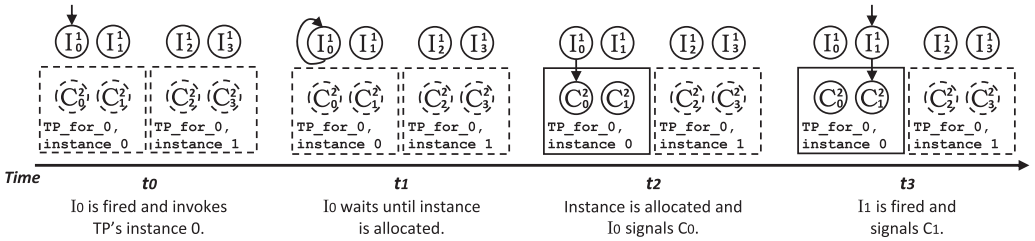


Fig. 5. Instantiation at runtime of the `for` region in Figure 4. Case is shown when the number of threads defined for the region is $N = 4$ and two instances are defined for the region's TP. Dotted lines are used to indicate that the corresponding object has not yet been allocated in memory.

A different situation occurs with a synchronization codelet, which may be instantiated only once at runtime. If such a codelet is a predecessor codelet, it must signal all instances of its dependent codelet. This is shown in Figure 4, with S_0^3 signaling all instances of computation codelet C^4. Conversely, if a synchronization codelet is a dependent codelet, its number of dependencies is set to the number of threads in the region. These dependences are satisfied by all the instances of its predecessor codelet. In Figure 4, for instance, the number of dependencies for S_0^3 is set to N. These are satisfied by all instances of C^2. Arcs between two synchronization codelets are usually optimized by merging their CBBs.

A special case occurs when the input program makes use of the depend clause. The clause defines at compile time and, in an explicit way, data dependencies that must be enforced at runtime during the execution of OpenMP tasks. Depending on the type of dependency defined by the clause (either true, anti, or output dependence), *omp2cd* creates an arc at compile time between the last codelet defined for the task region producing the dependence and the first codelet on each of the task regions expecting it.

3.2.4 Instantiation of TPs and Back Edges in Invocation Codelets. As explained in Section 2.2, a TP invocation is a request sent to the runtime for the instantiation of a TP. This request is made by a microscheduler when it finds a TP invocation operation in the current codelet it is executing. The request is sent to its local TP scheduler, which allocates memory for the new TP instance and for the instances of the codelets it contains. The number of instances allocated at runtime for each TP depends on the type of OpenMP region that it represents. For instance, the TP defined for a `single` region may be instantiated only once, whereas several instances of a `for` region's TP may be instantiated to balance the workload across the target machine (Section 3.3).

Because currently DARTS relies on the OS's kernel for any memory allocation, the instantiation of TPs may be considered a long-latency operation and can cause a codelet to be blocked while waiting for the instantiation to complete. To avoid this, TP invocation operations are performed exclusively by invocation codelets. Additionally, once the allocation is made, invocation codelets are in charge of signaling the codelet's instances of the new TP instance so they can be scheduled for execution.

This process is shown in Figure 5. This figure contains a timeline of events occurring at runtime when the `for` region from Figure 4 is instantiated twice, the C^2 computation codelet defined for the loop's body is instantiated up to the number of available OpenMP threads in the region (i.e., $N = 4$), and the codelet's instances are evenly distributed among the TP instances.

In t_0, invocation codelet with ID = 0, I_0^1, is fired. Then, this instance checks if its assigned TP instance for the `for` region, instance number 0, has been invoked, and, assuming it has not, it invokes it. I_0^1 now waits until the TP instance is allocated to use the pointer to it to signal one of its

codelets. However, the Codelet Model prohibits having codelets spinning idly until the completion of an event—in this case, the allocation of a new TP instance. So the invocation codelet resets itself to be sent back to the local TP scheduler and be scheduled again for execution. By doing this, the invocation codelet releases the processing unit that it was using to allow the execution of another ready codelet. The process of having the I_0^1 invocation codelet reseting itself until its TP instance is created is shown in t_1 with a back edge.

Once the instance has been allocated in t_2, I_0^1 can use the pointer to it to signal the first codelet in the TP with its same instance ID. In Figure 5, this corresponds to the C_0^2 computation codelet, which becomes a ready codelet and is scheduled for execution. Finally, in t_3, I_1^1 is fired and, since the pointer to its TP instance is already available, it signals C_1^2 right away, which has its same instance ID. A similar line of events occurs for the TP's instance number 1 when I_2^1 and I_3^1 are fired.

3.3 Multigrain Optimization

To improve the fine-grain execution of the input program, *omp2cd* may apply a series of optimizations to each CDG depending on the parallelism available in the application. These optimizations may be complemented at runtime by tuning up the fine-grain runtime's AMM parameters. Two of the main optimizations applied are:

3.3.1 Region Inlining. If the number of statements to be executed by an `omp` region is less than a *instruction granularity threshold* set by the user in the command line, the compiler merges the CBB (or TP) for such region with the CBB (or TP) implemented for the enclosing one, and all its statements (or codelets) are moved one level "up." By doing this, the runtime avoids the overhead of instantiating and scheduling an instance of the original TP. This optimization is mainly applied to `single` and `master` regions, as well as to small `for` regions, like those initializing an array.

3.3.2 OpenMP Clause Extension – `codelet`. An `omp` region can be also inlined by using the `codelet` clause. With it, developers can instruct the compiler to implement the intended region using *only* codelets, with no TPs. This clause overrides the instruction parameter threshold, and, consequently, the region is effectively inlined, regardless the number of statement it contains. Currently, this clause is supported for `for` and single regions and is ignored when generating SWARM applications since this runtime does not support TPs.

3.3.3 Loops (DARTS Only). A dedicated stage to optimize `for` regions is included since they are very dominant in scientific numerical applications. One of two types of optimizations may be applied. The first optimization (applied by default) translates the `for` region into a TP containing the CDG generated for the loop's body. Then, at runtime, the TP is instantiated M times. Users can set the value for M using an environment variable or let the runtime use the number of clusters set in the AMM. These M instances are distributed across the different clusters in the AMM, and each instance is assigned up to $ceiling(N/M)$ codelets with consecutive IDs, where N is the number of OpenMP threads defined for the region. Loop's iteration range is distributed hierarchically among TP instances and then between codelets, following the loop's scheduling policy.

The second optimization, which is available through a compiler's command line option, uses a set of special parallel constructs provided by DARTS, named *TPLoops* (Suettlerlein et al. 2013). With this optimization, the compiler generates three CDGs for the loop's body. Then, at runtime, the fine-grain application selects one of them to execute the loop's body depending on the current range and a set of thresholds set by the user. The first CDG contains a single computation codelet, which is instantiated only once at runtime and is in charge of executing all the loop's iterations. The second CDG, called *Codelet Parallel For (CPF) Loop*, defines a TP with a single codelet inside;

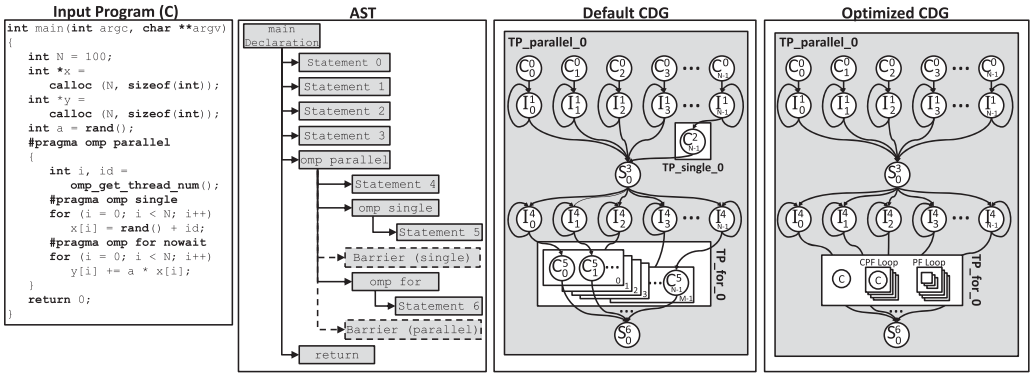


Fig. 6. Default and optimized multigrain transformation of an OpenMP program. Superscripts represent CBB's IDs, whereas subscripts represent the instance's ID from 0 to $N-1$, with N being the number of OpenMP threads in the region.

during execution, the TP is instantiated P times and the loop's range is distributed uniformly among the instances. The third CDG is called *Parallel For (PF) Loop* and defines a TP having a CPF Loop. In other words, a PF Loop is a CPF Loop contained inside another TP. At runtime, Q instances of the PF Loop are created, each one having P instances of their CPFs. The total number of a loop's iterations is distributed uniformly among the Q instances of the PF Loop, and then each one of these instances divides its subrange among the local P instances of its CPF Loop. Values for P and Q and the TPLoops thresholds are set by the user at runtime depending on the configuration of the AMM, the maximum number of OpenMP threads, and any heuristics or experimental data the user relies on.

Both optimizations aim at improving locality of the data accessed and processed by the `omp for` region, but they differ on the type of loops for which they have been designed. The default optimization is intended for loops that have the same range throughout the lifetime of an application, set either statically or dynamically. An example of such type of application is matrix multiply, where the number of iterations to be executed is fixed as the number of rows or columns in the matrix or the tile, in case a tiling algorithm is used. TPLoops, on the other hand, are more suitable for loops with dynamic ranges, where the number of iterations varies. This is the case, for example, with Graph500, which makes use of a Breadth-First Search algorithm that uses a parallel loop to examine a set of nodes in the graph, known as the graph's frontier (Graph500 2010). At runtime, and depending on the graph's size, the frontier can be as small as a dozen nodes (for which a single codelet is more suitable), as big as several hundreds or thousands (for which a PF Loop is better), or something in the middle (in which case, CPF Loops are preferred).

3.4 Example

Figure 6 presents the default and optimized versions of the CDG generated by the compiler for a DAXPY OpenMP program. The program contains a `main()` function with one `parallel` region, which has a `single` region for the vector initialization and a `for` region for the computation. A TP is generated for the `parallel` region (rule TP-2), but not for the function since it is `main()`.

The first statement in the region, `statement 4`, is translated into a computation codelet to be instantiated N times (rule CBB-3). In the default CDG, a TP is defined for the `single` region (rule TP-2) with one single computation codelet, C^2, implementing `statement 5` (rule CBB-3). N instances of the invocation codelet I^1 are also defined for this region to determine at runtime the

```
#include <darts.h>
using namespace darts;
Runtime *myDARTSRuntime;
int main(int argc, char **argv)
{
  ThreadAffinity *affin = new ThreadAffinity(NUM_CS,
    NUM_TPS, AFFINITY, TPS_SCHD, MCS_SCHD);
  myDARTSRuntime = new Runtime(affin);
  int N = 100;
  int *x = calloc (N, sizeof(int));
  int *y = calloc (N, sizeof(int));
  int a = rand();
  myDARTSRuntime->run(launch<TP_parallel_0>(&N,
    &x, &y, &a));
  return 0;
}
class TP_parallel_0: public ThreadedProcedure{
  class Compute_0: public Codelet {
    public: void fire(void){
      myTP->id_darts_0[this->getID()] =
        this->getID();
      myTP->invoke_1[this->getID()].decDep();
    }
  };
  class Invoke_1: public Codelet {
    public: void fire(void){
      if (!__sync_val_compare_and_swap(&(
        myTP->TP_single_launched_0), 0, 1)){
        for (int i = 0; i < N; i++)
          myTP->x_darts_0[i] = rand() +
            myTP->id_darts_0[this->getID()];
      }
      myTP->synch_3->decDep();
    }
  };
  class Synch_3: public Codelet {
    public: void fire(void){
      for (size_t c = 0; c < myTP->numThreads; c++)
        myTP->invoke_4[c].decDep();
    }
  };
```

```
  class Invoke_4: public Codelet {
    public: void fire(void){
      size_t idx = this->getID() / NUM_MCS;
      if (idx < NUM_TPS) {
        if (!__sync_val_compare_and_swap(
          &(myTP->TP_for_launched_0[idx]), 0, 1)) {
          int initIt, lastIt;
          determineSubRange(0,
            myTP->N_darts_parallel, &initIt, &lastIt);
          invoke<TP_for_0>(initIt, lastIt,
            myTP->N_darts_parallel,
            myTP->a_darts_parallel,
            myTP->x_darts_parallel,
            myTP->y_darts_parallel,
            &(myTP->TP_for_0_ptr[idx]));
        } else {
          if (myTP->TP_for_0_ptr[idx])
            myTP->TP_for_0_ptr[idx]->
              dispatchCodelet(this->getID());
          else
            this->resetCodelet();
        }
      }
    }
  };
  class Synch_6: public Codelet {
    public: void fire(void) {
      myDARTSRuntime->finalCodelet->decDep();
    }
  };
  TP_parallel_0(int *in_N, int *in_a, int **in_x,
    int **in_y){
    this->synch_6 = Synch_6(this->numThreads, this,0);
    this->synch_3 = Synch_3(this->numThreads, this,0);
    for (int c = 0; c < this->numThreads; c++) {
      this->invoke_4[c] = new Invoke_4(1, this, 0);
      this->invoke_1[c] = new Invoke_1(1, this, 0);
      this->compute_0[c] = new Compute_0(1, this, 0);
    }
  }
};
```

Fig. 7. DARTS application in C++ for the optimized CDG in Figure 6 (stripped-down code shown for brevity).

thread executing its TP (for simplicity, this thread is $N-1$ in Figure 6). Provided the instruction granularity threshold set by the user in the command line is less than the number of statements to execute by the `single` region, or that the `codelet` clause is used, this region may be inlined to avoid the overhead of allocating and scheduling a TP. This is shown in the optimized CDG, where the C^2 computation codelet has been merged with the previous invocation codelets. In this case, the invocation operation in the I^1 codelets is replaced with the code from C^2. The implementation of the `single` region ends with a synchronization codelet, S^3, for its implicit barrier.

The `for` region is also translated into a TP using rule TP-2. As mentioned previously, two implementations are available for this type of region. In the default implementation, the TP defined for the region is instantiated M times, with each instance handling up to N/M codelets. Iteration chunks are assigned depending on the scheduling policy set for the loop (`static` by default). The optimized implementation uses TPLoops with three different CDGs, which are selected at runtime depending on the loop's current range (Section 3.3.3). This optimization is only available in DARTS, so SWARM would only implement the default CDG for this region. N invocation codelets are also instantiated to distribute the available N OpenMP threads among the different TP instances.

The `for` region does not have an implicit barrier, so the codelets in its TP may continue individually and signal the synchronization codelet defined for the `parallel` region's implicit barrier, S^6. This codelet may be implemented as a tree to reduce signaling overhead. However, for simplicity, this is represented as one codelet instance in Figure 6.

The code produced in DARTS for the optimized CDG in Figure 6 is presented in Figure 7. The `main` function is left in its sequential form with two fundamental changes. First, is the addition at its beginning of any statement necessary to initialize the runtime. This is shown in Figure 7 with calls to `ThreadAffinity` to set up the DARTS AMM (parameters include the number of microschedulers and TP schedulers, their affinity scheme, and the schedulers' policies) and to `Runtime` to create the

runtime. The second change is a call to the runtime whenever a `parallel` region is found. Calls to the runtime require the name of the class defined for the `parallel` region (`TP_parallel_0`), along with all variables defined outside the region but used inside of it.

The definition of the `TP_parallel_0` class appears after the `main` function and contains the classes of the TP's codelets. Each codelet class has a `fire` method, which is called upon satisfaction of dependencies of the corresponding instance. It also uses the *myTP* variable to access the TP's data frame and `getMyID` to get its instance ID. Each invocation and computation codelet is instantiated in the `TP_parallel_0`'s constructor up to the number of threads set for the `parallel` region, whose value is accessed with `numThreads`, with each instance having one dependency; synchronization codelets are instantiated only once with `this->numThreads` dependences. `TP_parallel_0`'s constructor also signals all instances of the `Compute_0` class to start their execution since they represent the first codelet in the region, C^0. After each of these instances execute `statement 4`, they signal the instance of the next codelet class, `Invoke_1`, having the same instance ID.

The `Invoke_1` class implements the invocation codelet defined for the `single` region. An atomic operation is used to select the instance executing the statements in the region, which are contained in the same invocation codelet since the region has been inlined. All `Invoke_1`'s instances signal the only instance defined for the `Synch_3` class that represents the region's barrier's codelet. When this instance is fired, it signals all instances of the `Invoke_4` class, which has been defined for the I^4 invocation codelet. This codelet is in charge of invoking *NUM_TPS* instances of the class defined for the `for` region's TP, `TP_for_0`. If an `Invoke_4`'s instance determines its TP instance has already been allocated, it calls the TP's `dispatchCodelet` function to signal the first codelet in the `for` region, as explained in Section 3.2.4.

The `TP_for_0` class, not shown for brevity, contains the classes of the codelets executing the loop's statements. When the instances of these codelets finish, they signal the `Synch_6` class' instance that implements the `parallel` region's implicit barrier. Since this is the last codelet in the region, its instance signals the runtime's `finalCodelet` to shut down the runtime and execute the next statement after the `parallel` region in the `main` function.

3.5 Optimized DARTS Runtime

The DARTS runtime used in the environment is an optimized version we modified to improve its performance with respect to OpenMP. This implementation includes a set of new features available to users through the API and environment variables. We describe them here.

3.5.1 TP Placement. By default, newly invoked TPs are assigned to the same cluster where they had been originated. This might not be the best approach for embarrassingly parallel types of code regions, such as *forall* loops, where data are evenly distributed across the machine. To support the assignment of a new TP to a specific cluster, we have added the *TP Placement* function, which takes the target cluster as an additional parameter.

3.5.2 Scheduler Downtime and Delayed Context Deletion. When executing a DARTS fine-grain application, runtime overheads may occur as a result of having schedulers busy-waiting for long periods of time. This can heavily tax the memory subsystem of compute nodes as they rely on costly atomic operations. Another source of overhead may be the constant dynamic allocation and deallocation of underlying runtime context objects. When the last codelet of a TP is done firing, the TP is ready to be deallocated. As DARTS does not have a custom allocator and relies on the OS memory manager for allocation, this may result in multiple cores attempting to access it simultaneously, generating several system calls which can only be serialized to access the Linux kernel.

To reduce the runtime overhead from these two sources, we added a *Scheduler Downtime and Delayed Context Deletion (SD+DCD)* option. SD+DCD reduces the time schedulers are idle by using an exponential backoff strategy with a saturation cap, which can be fine-tuned by the user, allowing schedulers to enter into sleep mode when there are no TPs to create nor codelets to execute. Additionally, this option delays the deletion of objects until there is no work to perform. When used, schedulers act as follows: When a TP is marked for deletion, it is added to a deletion queue. Later, when a scheduler has no work to perform, it checks if the deletion queue has reached a threshold set in the runtime and proceeds to delete all completed TPs in sequence; otherwise, it enters into sleep mode.

4 EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of the fine-grain applications generated by our environment. Experiments were performed for the two fine-grain runtimes supported by *omp2cd*: DARTS and SWARM. The applications used feature different types of parallelism and memory access patterns, which makes them ideal to determine the types of workloads for which our environment is more suitable and to compare the performance of both fine-grain runtimes.

4.1 Experimental Testbeds

Experiments were performed on two Intel-based shared-memory nodes. All cores feature 32 KiB private L1 instruction and data caches, and 256KiB private unified L2 caches. The first platform is a two-socket node, with 10 cores per socket, featuring an Intel Xeon E5-2660V2 (Ivy Bridge) clocked at 2.60GHz with Hyper-Threading (HT) disabled. A 25MiB unified L3 cache is shared by all the cores in the same socket (50MiB in total). The total memory is 64GiB of DRAM divided into two NUMA domains. The second compute node is a four-socket node, with six cores per socket, with an Intel Xeon E5-4610V1 processor (Sandy Bridge) clocked at 2.2GHz with HT activated. All cores in the same socket share a 15MiB unified L3 cache (60MiB in total). All 48 threads have access to 128GiB DDR3 DRAM, distributed across four NUMA domains. The Ivy Bridge testbed runs CentOS v6.6, while the Sandy-Bridge runs Ubuntu v14.04. All programs were compiled with GCC v6.3.0 with optimizations set to `-O3`. OpenMP applications were linked against Intel's OpenMP runtime released with clang 3.9 (LLVM-OpenMP 2017).

4.2 Experimental Design

Experiments were performed on five benchmarks written in C and parallelized and optimized by their authors using OpenMP 2.5 directives. Unless expressed otherwise, no major changes or additional optimizations were introduced in the benchmarks' source code in order to perform a fair comparison and to use them in the same form that is available to other people.

Each OpenMP benchmark was compiled with *omp2cd* to generate two equivalent fine-grain applications: One in DARTS and other in SWARM. These were then compared against the reference OpenMP application. Since *omp2cd* generates a CDG describing the input OpenMP code in terms of the Codelet Model and uses the same CDG to generate the DARTS and SWARM output applications, experimental results were also used to compare not only the performance of the fine-grain applications against OpenMP, but also the performance of one fine-grain runtime against the other.

At runtime, threads were pinned down evenly across each testbed using `GOMP_CPU_AFFINITY` on OpenMP, `DARTS_AFFINITY` on DARTS, and `SWARM_PIN_THREADS` on SWARM. On all experiments, iteration space in parallel loops was partitioned statically (as specified in the original OpenMP codes) and spread evenly across all NUMA nodes in a round-robin fashion using `numactl -i all`.

For DARTS applications, around 16 combinations of configurations/optimizations of the AMM were evaluated at runtime, with the best ones being presented in Table 1. The number of schedulers

Table 1. Fine-Tuning of the AMM Implemented by DARTS for Each Benchmark

	Ivy Bridge				Sandy Bridge			
Bench.	Clusters per Socket	TP Plac.	SD+DCD [μs]	Affin.	Clusters per Socket	TP Plac.	SD+DCD [μs]	Affin.
Graph500	1	No	No	compact	1	No	No	compact
NAS-DC	10	No	512; 4,096	sparse	12	No	512; 4,096	sparse
SCF	1	Yes	512; 4,096	compact	1	Yes	16; 4,096	compact
CoMD	2	Yes	No	compact	2	Yes	No	compact
CoSP2	1	No	16; 16,384	compact	1	No	1,024; 8,192	compact

Affinity schemes used are Compact (0-19 in Ivy B.; 0-47 in Sandy B.) and Sparse (0, 10, 1, 11, … in Ivy B.; 0, 6, 12, 18, 1, 7, … in Sandy B). For benchmarks using SD+DCD, saturation caps for the sleep times of TP and microschedulers are shown.

per cluster was calculated as the ratio between `OMP_NUM_THREADS` and the number of clusters used by the application, with one of them being a TP scheduler and the rest microschedulers. For benchmarks using the SD+DCD option, the table shows the maximum time in microseconds that TP and microschedulers spend in sleep mode waiting for new work to perform. All DARTS versions were run using work-stealing policies for the schedulers. For SWARM applications, we set the number of schedulers equal to `OMP_NUM_THREADS`. Other SWARM runtime options were left as default. However, SWARM lacks proper documentation to determine these values .

Results shown in the following sections were taken after performing between 20 and 30 runs for each benchmark version. Unless otherwise specified, bar heights in the following figures represent the results' medians calculated from those runs, and error bars represent the results' maximum and minimum values. Figures also present each DARTS and SWARM bar with the percentage difference obtained with respect to OpenMP.

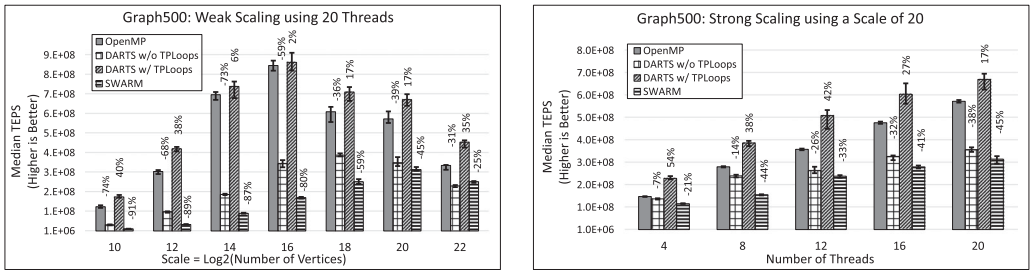
4.3 Graph500

Graph500 performs a parallel Breadth-First Search (BFS) in a large undirected graph. This benchmark is typically used to measure the performance of a system when dealing with irregular data-intensive problems. We used the reference OpenMP code from Graph500 (2010) with a minor modification in the search kernel: We moved the `parallel` region inside of the `while` loop as we noticed the instructions preceding the loop in the original code can be executed faster in sequential form without loss of correctness. All versions were executed using `OMP_SCHEDULE=static`.[7]

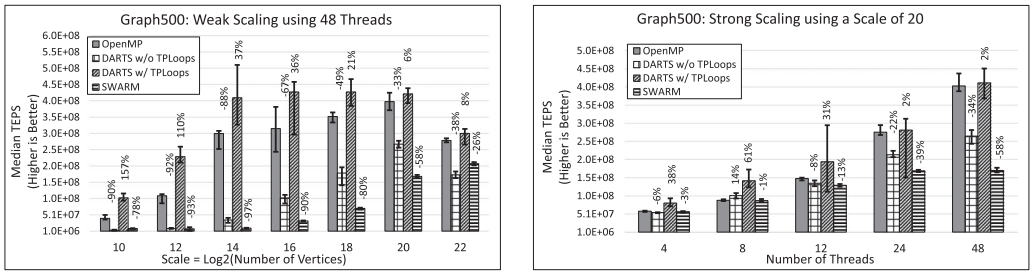
Experimental results for Graph500 are presented in Figure 8, which presents DARTS with and without TPLoops (see Section 3.3) in order to evaluate the impact the TPLoops optimization has in the application. TPLoops' thresholds were experimentally fine-tuned based on the graph's size, the maximum graph's frontier (i.e., the number of vertices to visit), and results obtained from run trials on each platform. The scale parameter was varied between 10 and 22 (higher scales were not evaluated due to testbed memory constraints); other parameters were left with their default values. Performance is measured in Traversed Edges per Second (TEPS).

The optimized DARTS version with TPLoops performs considerably better than OpenMP and the other fine-grain versions (i.e., DARTS without TPLoops and SWARM), especially in the lower end of the scale. This can be attributed to the irregular parallelism present in the application. Graph500's search kernel contains a parallel loop whose range varies at runtime depending on the graph's frontier. To better balance this work among threads, the size of iteration chunks has

[7] As explained by Suettlerlein et al. (2013), various chunk sizes using a dynamic policy yielded worse results than the default static one.



(a) Ivy Bridge.



(b) Sandy Bridge.

Fig. 8. Weak and strong scalability of Graph500 on (*a*) Ivy Bridge and (*b*) Sandy Bridge. DARTS results are presented with and without using TPLoops optimization.

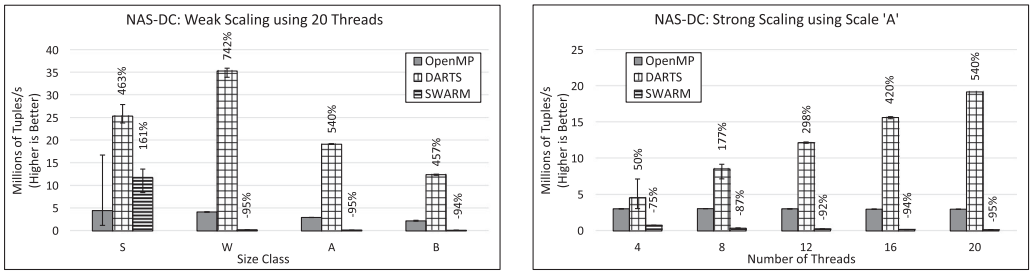
to change at execution time, which DARTS does by selecting the most appropriate CDG for the current range out of the three generated at compile-time for the TPLoops. Such capabilities are not present on SWARM, and a similar behavior cannot be obtained with OpenMP unless the search kernel is significantly rewritten.

As with the results presented by Suettlerlein et al. (2013), weak scalability results in Figure 8 show that DARTS with TPLoops outperforms OpenMP across all scales tested. Suetterlein's work uses a handwritten DARTS BFS application with TPLoops, with fixed thresholds for all scales. Conversely, our fine-grain application has been generated by *omp2cd*, reducing the time needed to write the program and only requiring users to tune certain parameters according to the target architecture and the application.

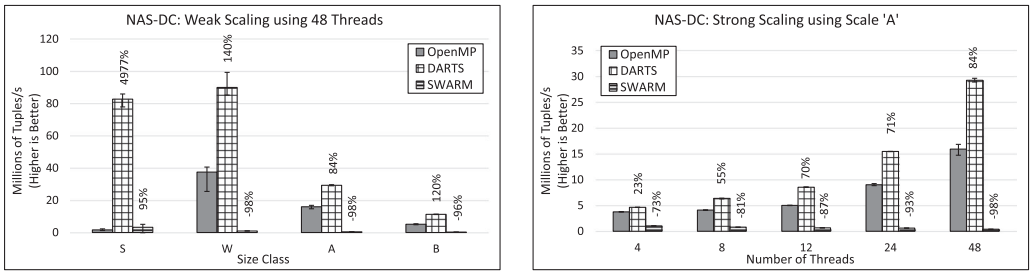
4.4 NAS-DC

NASA Advanced Supercomputing Division (NAS) Parallel Benchmarks is a suite of applications designed to evaluate the performance of supercomputers (NAS 2016). One of this is Data Cube (DC), which implements a user's view of a decision-support database, with each cube's cell having a set of attributes of interest for the user (Harinarayan et al. 1996). The benchmark is defined for four different problem sizes or *classes* (S, W, A, and B), containing a different number of attributes per cell in the data cube. S is the smallest class ($1E3$ tuples of up to five attributes), while B is the largest ($1E7$ tuples of up to 20 attributes).

Results for NAS-DC are shown in Figure 9. DARTS greatly improves with respect to OpenMP and SWARM for most cases. This may be attributed to three factors. First, unlike other benchmarks in the experimental set, NAS-DC does not contain parallel loops or barriers, thus reducing coarse-grain synchronization and allowing threads in the fine-grain runtime to progress individually. Second, fine-grain PXMs, like the Codelet Model, are more suitable for dynamic data structures,



(a) Ivy Bridge.



(b) Sandy Bridge.

Fig. 9. Weak and strong scalability of NAS-DC on (*a*) Ivy Bridge and on (*b*) Sandy Bridge.

such as the lattice of views used in NAS-DC, which is constantly changing at runtime based on each view's implementation cost. Finally, as seen later in Section 4.8, DARTS speedups are obtained thanks to its ability to tune-up its AMM for maximum performance, whereas SWARM offers users fewer options to do so. For instance, Figure 13(a) shows that when DARTS uses only two TP schedulers for NAS-DC with a size class A, its speedup is only 0.2, contrary to the one obtained when all 20 cores available are used as TP schedulers, which is 6.4.

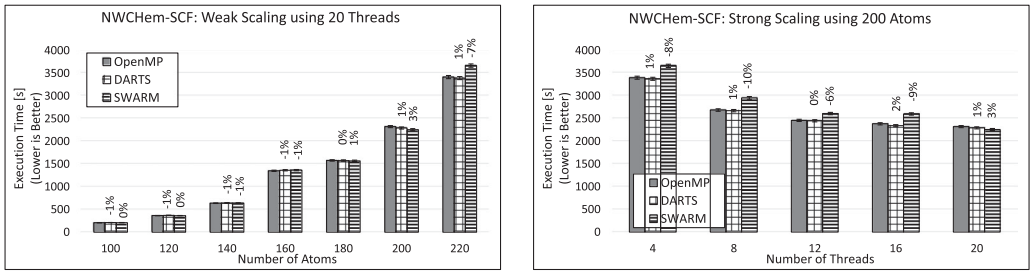
4.5 PNNL's NWChem-SCF

NWChem-SCF is an implementation of the Self-Consistent Field (SCF) method from Pacific Northwest National Laboratory (PNNL) and part of the NWChem package software (PNNL 2014), a high-performance computational software for the computation of large chemistry problems. SCF is an iterative fixed-point algorithm solving a nonlinear system with a Schrödinger's equation in the form of a self-consistent eigenvalue problem.

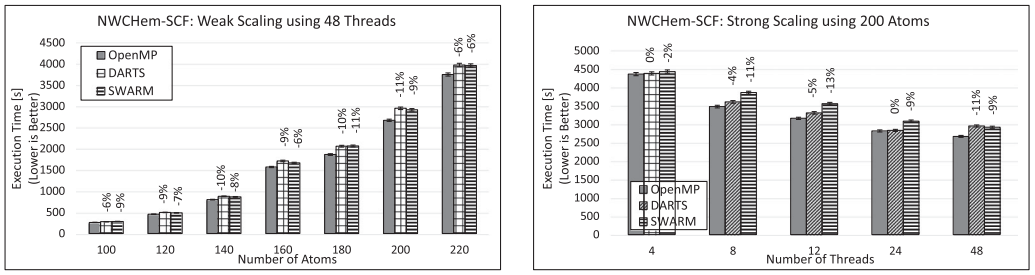
Figure 10 presents the results for weak and strong scalability. On both platforms, DARTS and SWARM performances remain, on average, within ±10% of OpenMP's, with both fine-grain runtimes performing slightly better on Ivy Bridge. While general implementations of SCF tend to be loosely structured, the one used for our experiments (obtained from ETI 2013) is very regular, comprising several rectangular nested loops. OpenMP has been extensively tuned for such types of workloads, which explains why speedups obtained with DARTS for this application are smaller than those for Graph500 and NAS-DC.

4.6 ExMatEx's CoMD

CoMD (Molecular Dynamics simulation) is one of the proxy applications of the Department of Energy's Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx) (DoE 2012a). These are used for the co-design of algorithms, software, and hardware for exascale

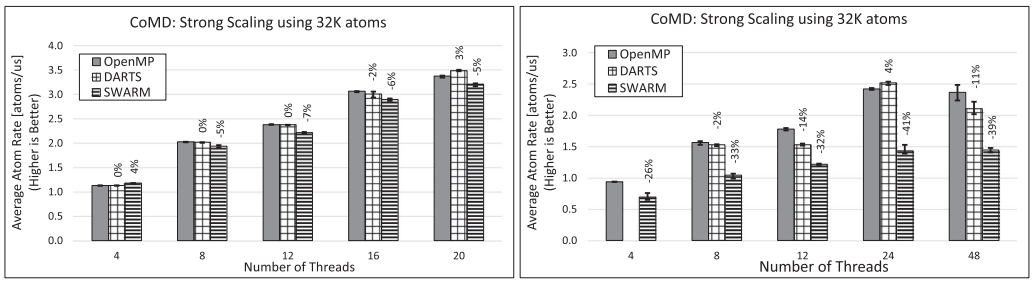


(a) Ivy Bridge.



(b) Sandy Bridge.

Fig. 10. Weak and strong scalability of NWChem-SCF on (*a*) Ivy Bridge and on (*b*) Sandy Bridge.



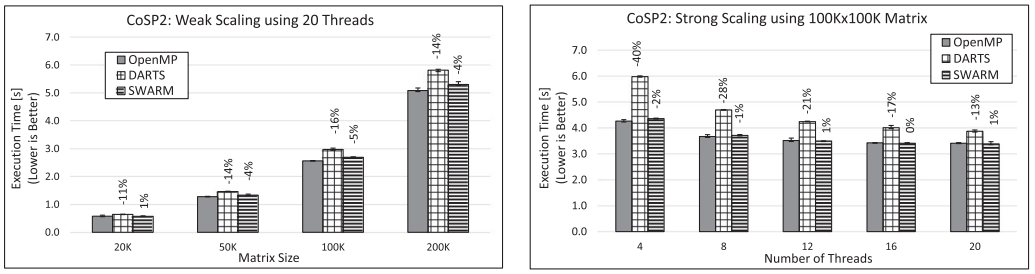
(a) Ivy Bridge.

(b) Sandy Bridge.

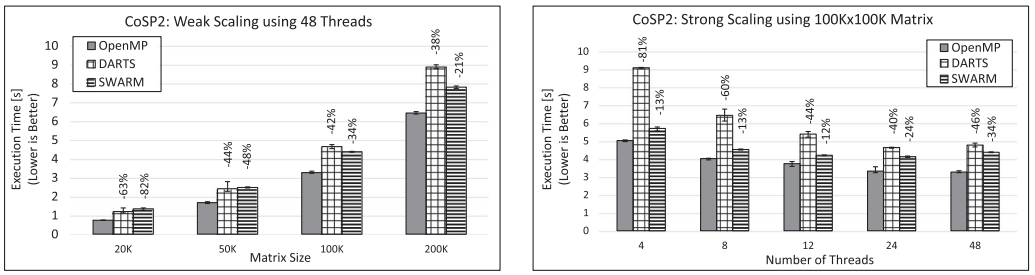
Fig. 11. Strong scalability of CoMD using 32000 atoms on (*a*) Ivy Bridge and on (*b*) Sandy Bridge. DARTS results on Sandy Bridge with four threads were not computed since the AMM was configured with eight TP schedulers.

simulation frameworks of extreme mechanical and radiation environments. CoMD evaluates the force imposed on each atom by the rest of the particles in a system within a cutoff radius using Lennard-Jones and embedded-atom method potentials.

Strong scalability results for this application are shown in Figure 11. As with NWChem-SCF, CoMD's parallelism is highly structured with static memory access patterns, which explains why DARTS performance in Figure 11 is also on par with OpenMP, but not better. In particular, OpenMP provides guarantees for statically scheduled loops which help improve data locality. This is not the case with SWARM, whose performance is greatly reduced on Sandy Bridge (Figure 11(b)). Unlike DARTS, SWARM does not define TPs and, therefore, cannot benefit from the increase locality of data obtained by having codelets access their data on each loop's iteration from their parent TP's data frame. Moreover, codelets in SWARM are not bound to a specific socket and may be executed



(a) Ivy Bridge.



(b) Sandy Bridge.

Fig. 12. Weak and strong scalability of CoSP2 on (*a*) Ivy Bridge and on (*b*) Sandy Bridge with a 500×500 submatrix.

by any core in the machine, ultimately reducing the locality of the application and its performance, which is more noticeable in Figure 11.

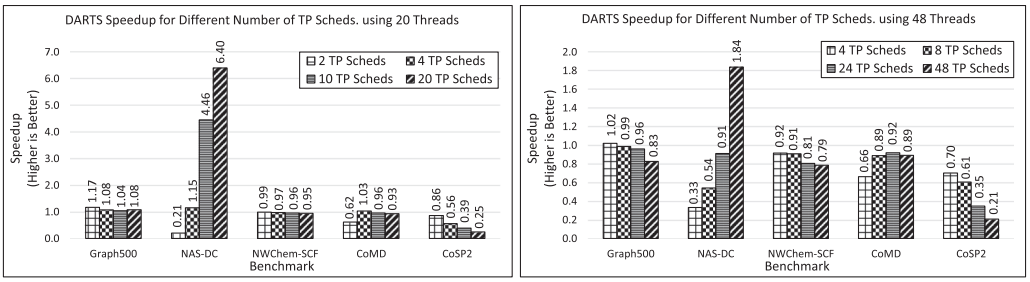
4.7 ExMatEx's CoSP2

CoSP2 (Second Order Spectral Projection for Electronic Structure Calculations) benchmark is another ExMatEx's proxy application (DoE 2012b). CoSP2 implements a sparse linear algebra parallel algorithm to determine density matrices in electronic structure theory using a recursive second-order Fermi-operator expansion method (SP2) defined as a recursive series of matrix multiplications.

Figure 12 presents results for weak and strong scalability. On both architectures, OpenMP has a lower execution time than the fine grain runtimes, with SWARM performing better than DARTS. On Sandy Bridge, however, the performance of both fine-grain runtimes is reduced, particularly for DARTS. The difference between both fine-grain applications indicates that sparse matrices (and particularly the ELL format used by CoSP2) are more suitable for fine-grain execution when using only one level of scheduling (as in SWARM that uses only codelets) rather than two levels (i.e., TPs and codelets in DARTS). This contrasts with applications with dense matrices, like NWChem-SCF and CoMD, where the data frame on each TP might improve the application's performance by increasing the locality of the data used by the TP's codelets.

4.8 Impact of Different Configuration Options in DARTS AMM in Application's Performance

As seen previously, applications with irregular parallelism and/or dynamic data structures, like Graph500 and NAS-DC, may perform better when executed in a fine-grain environment, such as the Codelet Model implemented by DARTS. However, improvements are not only due to the



(a) Ivy Bridge. (b) Sandy Bridge.

Fig. 13. DARTS speedup with respect to OpenMP for different numbers of TP Schedulers in the AMM.

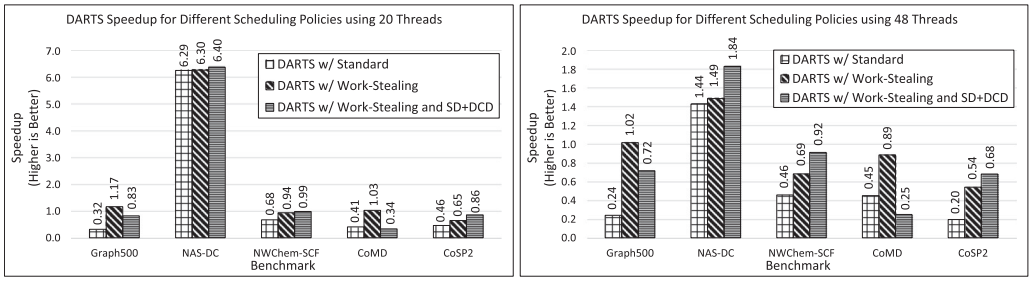
fine-grain nature of the runtime, but also to the ability users have to configure the AMM implemented by DARTS and to tune it up for each application. This section presents an analysis of two AMM configuration parameters available in DARTS and evaluates their impact in an application's performance. Experiments were performed on Graph500 for a scale of 20, NAS-DC for size class A, NWChem-SCF for 160 atoms, CoMD for 32k atoms, and CoSP2 for a 100k × 100k matrix and 500 × 500 submatrix. All results used 20 threads on Ivy Bridge and 48 threads on Sandy Bridge. Other parameters have the same values as those mentioned previously for each benchmark.

4.8.1 Number of TP Schedulers. The number of TP schedulers determines the number of clusters in the AMM, as well as the number of microschedulers per cluster used to implement DARTS's two-level scheduling mechanism. This has a direct impact in the locality of data accessed by the codelets and in the scheduling overhead produced by having a cluster's TP scheduler distribute ready codelets to each one of its microschedulers. To evaluate this, Figure 13 presents the speedup obtained with respect to OpenMP for each of the previous benchmarks for different AMM shapes.

From examining the results in Figure 13 and the particularities of each benchmark, we can determine that the impact that the number of TP schedulers has in an application's performance depends, at least partially, on the presence or absence of parallel loops. In fact, NAS-DC, which does not contain parallel loops, is the only application whose performance clearly improves for increasing numbers of TP schedulers. For all other applications that are designed around parallel loops whose ranges either change dynamically (like Graph500) or are fixed, performance improves when the number of TP schedulers is close to the number of sockets or NUMA nodes in the target architecture: Two on Ivy Bridge and four on Sandy Bridge. CoMD is a special case, requiring 4 TP schedulers on Ivy Bridge but 24 on Sandy Bridge.

This is related to the way that TPs are assigned to clusters and how codelets access their data. When a TP is invoked, it is assigned to a cluster's TP scheduler, which instantiates it and distributes its ready codelets to each one of the microschedulers of the same cluster for execution. If these microschedulers are mapped to the same socket or NUMA node as their cluster's TP scheduler (as it is the case for those applications with *compact* affinity in Table 1), the locality of data is increased since codelets access their data from their TP instance's data frame. This is especially important in applications based on parallel loops, where increased locality may reduce the number of memory accesses performed on each iteration, ultimately improving the application's performance.

However, if more than one cluster is mapped to the same socket or NUMA node, contention over the memory bank may increase, producing higher data latencies on each iteration. To avoid this, results in Figure 13 indicate that the "sweet spot" for most of the applications containing parallel loops is having only one cluster per socket or NUMA domain in the target architecture (i.e., two



(a) Ivy Bridge. (b) Sandy Bridge.

Fig. 14. DARTS speedup with respect to OpenMP for different scheduling policies.

2 on Ivy Bridge and four on Sandy Bridge). On the other hand, if an application does not contain such control-flow patterns (i.e., NAS-DC), no benefit is obtained from having clusters of TP and microschedulers in the same hardware region, and it is actually better to have a single level of scheduling (i.e., only TP schedulers in the machine with no microschedulers).

4.8.2 Scheduling Policy and Options. In addition to the number of TP and microschedulers in the abstract machine, users can choose from two main types of scheduling policies (see Section 2.2): *Standard*, where microschedulers only execute codelets pushed into their local queues by their parent TP scheduler, and *work-stealing*, where they steal codelets from their local TP scheduler's RCQ. Additionally, schedulers may have the option to go into sleep mode and to reduce the overhead of constantly deleting context objects by enqueueing them and deleting them afterward if the SD+DCD option (see Section 3.5.2) is activated. The impact that these scheduling policies and options have on DARTS application performance is presented in Figure 14.

All benchmarks perform better when work-stealing is used instead of standard policies, and, with the exception of Graph500 and CoMD, SD+DCD further improves the application's performance. When standard policy is used, schedulers only execute codelets whose TPs have been invoked in the same cluster. For microschedulers, this policy also means that they only execute codelets assigned by their local TP scheduler. If a microscheduler's local queue is empty, it spins idly until new work is pushed into it, even if its local TP scheduler has ready codelets on its own RCQ.

With work-stealing, however, a microscheduler may steal those codelets as soon as it has none on its own queue, thus reducing the time it spends idly. However, contention in the TP scheduler's RCQ may increase, as well as coherence traffic overall, because not only the TP scheduler is popping codelets out of it, but potentially also all of its microschedulers. Nevertheless, results in Figure 14 show that, for the benchmarks presented, this overhead is small in comparison with the benefits of having schedulers constantly busy executing work. Moreover, when SD+DCD is enabled, performance obtained with work-stealing is further enhanced by having schedulers sleep or delete previously queued context objects when there are no codelets to execute nor TPs to create. For Graph500 and CoMD, results show that SD+DCD actually decreases the application's speedup obtained with work-stealing alone. This indicates that, for these applications, schedulers are constantly executing codelets and benefit more from deallocating context objects immediately rather than later.

4.9 Summary of the Environment's Opportunities and Tradeoffs

Experimental results show that the multigrain parallel programming environment is more suitable for certain types of applications and that performance of the DARTS runtime can be highly

dependent on the set of configuration parameters used for its AMM. Here, we summarize the scenarios where an increase in performance with respect to OpenMP can be expected and the tradeoffs a user of the environment may face:

(1) Applications with irregular parallelism and/or dynamic data structures may perform better when written and executed under a fine-grain runtime, such as DARTS.
(2) OpenMP tends to get slightly better performance than the Codelet Model with regular workloads.
(3) Performance of a DARTS application featuring parallel loops can be increased when the number of clusters in the AMM matches the number of sockets in the target architecture.
(4) DARTS implements a highly configurable AMM, offering more flexibility to users to tune-up applications at execution time than SWARM.

5 RELATED WORK

The Optimally Scheduled Advanced Multiprocessor (OSCAR) multigrain parallelizing compiler (Kimura et al. 2010) decomposes an application with several grains of parallelism. It generates a parallelized application from a sequential program that may be compiled by an OpenMP compiler or translated into a Fortran or C program with runtime library calls. In this sense, OSCAR output programs are run using OpenMP's coarse-grain PXM, unlike *omp2cd*, which generates applications to run over a fine-grain PXM.

Weng's doctoral dissertation (Weng 2003) proposes the translation of OpenMP programs to a dataflow PXM using the Cougar compiler and SMARTS runtime system, increasing the data locality by finding, at compile time, the best mapping of nodes in the task graph to processors. *omp2cd*, on the other hand, improves data locality by automatically determining the size and number of TPs to be used in the application in order to exploit the implicit locality provided by the two-level parallelism scheme found in DARTS.

KaCC is an ongoing project for the design of a compiler to parse OpenMP programs into applications running on a dataflow-like runtime with different task grain sizes; it is named libKOMP (Broquedis et al. 2012) and is based on XKaapi (Gautier et al. 2013). The main difference between KaCC and *omp2cd* is that the latter performs a two-level granularity analysis in the input OpenMP application (TPs and codelets), while the former decomposes the input application into a single type of parallel construct needed in libKOMP, (i.e., tasks).

OpenStream allows programmers to express dynamic dependent tasks by annotating an OpenMP application (Pop and Cohen 2013). In this way, an application's dataflow task graph can be specified using first-class dataflow streams. *omp2cd* is different by not requiring users to modify the original OpenMP application in order to obtain a dataflow-like application that uses the Codelet model.

OmpSs provides OpenMP extensions to support the StarSs programming model in a single interface (BSC 2017). StarSs is a programming model that computes, at runtime, the dependencies between tasks to execute a sequential task-based program in parallel. The task graph in OmpSs is thus by nature dynamic. In *omp2cd*, using DARTS as a back-end, codelet graphs contained within a TP are *static* (i.e., all the required codelets, *along with their dependence relationships*, are computed once and for all). This allows for a lower overhead task graph instantiation.

Cilk/Cilk++ (Leiserson 2010) comprises a set of language extensions for C/C++ for the development of multithreaded applications using a divide-and-conquer model and providing a simple interface through the use of a small set of keywords. Like DARTS and SWARM, Cilk/Cilk++ scheduler uses a work-stealing algorithm; however, it does not provide locality-aware mechanisms like

those provided by the Codelet Model and DARTS through the use of TPs and their assignment to specific clusters in the machine.

Rice University's Habanero (Kumar et al. 2014) project comprises a set of parallel programming languages, an execution model, compilers, and a runtime for the development of parallel applications for extreme-scale systems in a wide variety of architectures. It supports dynamic creation and termination of lightweight tasks by transforming at compile-time tasking constructs into runtime calls. These tasks may block when executing, which is an important difference with respect to the Codelet Model, which specifies that codelets are non-preemptive and must run until completion without blocking.

7 CONCLUSION

We have presented a multigrain parallel programming environment based on the Codelet Model for the translation and fine-grain execution of OpenMP programs. It includes a multigrain parallel compiler, *omp2cd*, which automatically determines the granularity of fine-grain tasks at two levels of parallelism and generates a set of source files to be executed on top of a runtime implementing the Codelet Model: DARTS or SWARM. The environment also includes an optimized version of the DARTS runtime that has been modified to improve its performance with respect to OpenMP.

Experimental evaluation showed that DARTS improves the performance of workloads with irregular parallelism and dynamic structures with respect to OpenMP and SWARM. For applications having a more structured parallelism designed around nested parallel loops and static memory access patterns, and for which OpenMP has been broadly optimized, the output fine-grain applications can match OpenMP's performance by either DARTS or SWARM or both. Their performance on these type of workloads may vary depending on factors like the DARTS AMM's configuration and the type of arrays used by the algorithm, with SWARM performing better when the benchmark employ sparse matrices.

Results also showed that fine-tuning DARTS AMM can have a great impact on an application's performance by determining the number of codelets executed by each cluster, which has a direct impact on the locality of data used by the codelets. Applications containing parallel loops are more likely to improve their performance when the number of clusters in the AMM is close to the number of sockets in the underlying architecture. Conversely, applications with no parallel loops benefit more from having only one level of scheduling in DARTS (i.e., having all threads acting as TP schedulers). Further improvements may be achieved by selecting the appropriate scheduling policy and options, like allowing schedulers to enter into sleep mode when there is no work to do and performing a fine-tuning of the corresponding sleep time.

Work is currently being done in improving the performance of the environment for additional scientific applications, such as EPCC OpenMP micro-benchmark suite (of Edinburgh 2016), NAS Integer Sort, and Barcelona Supercomputing Center's OpenMP Tasks Suite (Duran et al. 2009). Additionally, future work will focus on the integration in *omp2cd* of additional mechanisms offered by SWARM to complement the experimental evaluation of the environment and the performance comparison between the two fine-grain runtimes. We will also focus on making the *omp2cd* compiler fully compliant with OpenMP v2.5, as well as in completely supporting the tasking constructs and clauses from OpenMP v3.0 and v.4.0. New clauses extending OpenMP will also be added to guide the compilation process with specific power and resilience constraints.

REFERENCES

Jaime Arteaga, Stephane Zuckerman, and Guang R. Gao. 2017. Multigrain parallelism: Bridging coarse-grain parallel programs and fine-grain event-driven multithreading. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 799–808. DOI: http://dx.doi.org/10.1109/IPDPS.2017.63

François Broquedis, Thierry Gautier, and Vincent Danjean. 2012. libKOMP, an efficient openMP runtime system for both fork-join and data flow paradigms. In *OpenMP in a Heterogeneous World: Proceedings of the 8th International Workshop on OpenMP (IWOMP'12)*. Springer, Berlin, 102–115. https://link.springer.com/chapter/10.1007%2F978-3-642-30961-8_8.

BSC. 2017. Barcelona Supercomputing Center: The OmpSs Programming Model. Retrieved March 31, 2017 from https://pm.bsc.es/ompss.

CAPSL. 2013. Computer Architecture and Parallel Systems Laboratory at University of Delaware: The Codelet Execution Model. Retrieved March 31, 2017 from http://www.capsl.udel.edu/codelets.shtml.

DoE. 2010. US Department of Energy, Office of Science, Office of Advanced Scientific Computing Research: The Opportunities and Challenges of Exascale Computing. Retrieved March 31, 2017 from http://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf.

DoE. 2012a. DoE Exascale Co-Design Center for Materials in Extreme Environments, ExMatEx - Proxy Applications - CoMD. Retrieved October 18, 2017 from http://www.exmatex.org/comd.html.

DoE. 2012b. DoE Exascale Co-Design Center for Materials in Extreme Environments, ExMatEx Proxy Applications - CoSP2. Retrieved October 18, 2017 from http://www.exmatex.org/cosp2.html.

Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona openMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openMP. In *Proceedings of the 2009 International Conference on Parallel Processing (ICPP'09)*. IEEE Computer Society, Washington, DC, Article 16, 8 pages. DOI: http://dx.doi.org/10.1109/ICPP.2009.64

ETI. 2013. ET International Inc., DynAX - Extreme Scale Software Stack. Retrieved October 18, 2017 from https://wiki.modelado.org/images/1/15/Scf.tar.gz.

Thierry Gautier, Joao V. F. Lima, Nicolas Maillard, and Bruno Raffin. 2013. XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13)*. IEEE Computer Society, Washington, DC, 1299–1308. DOI: http://dx.doi.org/10.1109/IPDPS.2013.66

Graph500. 2010. Graph500 Reference Implementation Version 2.1.4. Retrieved October 18, 2017 from https://github.com/graph500/graph500/releases/tag/graph500-2.1.4.

Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1996. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*. ACM, New York, 205–216. DOI: http://dx.doi.org/10.1145/233269.233333

Keiji Kimura, Masayoshi Mase, Hiroki Mikami, Takamichi Miyamoto, Jun Shirako, and Hironori Kasahara. 2010. OSCAR API for real-time low-power multicores and its performance on multicores and SMP servers. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC'09)*. Springer, Berlin, 188–202. DOI: http://dx.doi.org/10.1007/978-3-642-13374-9_13

Vivek Kumar, Yili Zheng, Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. 2014. HabaneroUPC++: A compiler-free PGAS library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS'14)*. ACM, New York, Article 5, 10 pages. DOI: http://dx.doi.org/10.1145/2676870.2676879

Christopher Lauderdale and Rishi Khan. 2012. Towards a codelet-based runtime for exascale computing: Position paper. In *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT'12)*. ACM, New York, 21–26. DOI: http://dx.doi.org/10.1145/2185475.2185478

Charles E. Leiserson. 2010. The cilk++ concurrency platform. *The Journal of Supercomputing* 51, 3 (2010), 244–257. DOI: http://dx.doi.org/10.1007/s11227-010-0405-3

LLVM-OpenMP. 2017. OpenMP: Support for the OpenMP Language. Retrieved March 1, 2017 from http://openmp.llvm.org/index.html.

NAS. 2016. NASA Advanced Supercomputing Division Parallel Benchmarks. Retrieved September 20, 2017 from http://www.nas.nasa.gov/publications/npb.html.

The University of Edinburgh. 2016. EPCC OpenMP Micro-benchmark Suite. Retrieved March 31, 2017 from https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite.

OpenMP4.0. 2011. OpenMP Application Program Interface Version 4.0. Retrieved March 31, 2017 from http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

PNNL. 2014. Pacific Northwest National Laboratory: Hartree-Fock Theory for Molecules - NWCHEM. Retrieved March 31, 2017 from http://www.nwchem-sw.org/index.php/Release65:Hartree-Fock_Theory_for% _Molecules.

Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and data-flow compilation of openMP streaming programs. *ACM Transactions on Architecture and Code Optimization* 9, 4, Article 53 (Jan. 2013), 25 pages. DOI: http://dx.doi.org/10.1145/2400682.2400712

Joshua Suettlerlein, Stéphane Zuckerman, and Guang R. Gao. 2013. An implementation of the Codelet model. In *Proceedings of the 19th International Conference on Parallel Processing (Euro-Par'13)*. Springer, Berlin, 633–644.

Tien-Hsiung Weng. 2003. *Translation of Openmp to Dataflow Execution Model for Data Locality and Efficient Parallel Execution*. Ph.D. Dissertation. University of Houston, TX.

Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. 2011. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT'11)*. ACM, New York, 64–69. DOI: http://dx.doi.org/10.1145/2000417.2000424

Received March 2017; revised September 2017; accepted October 2017